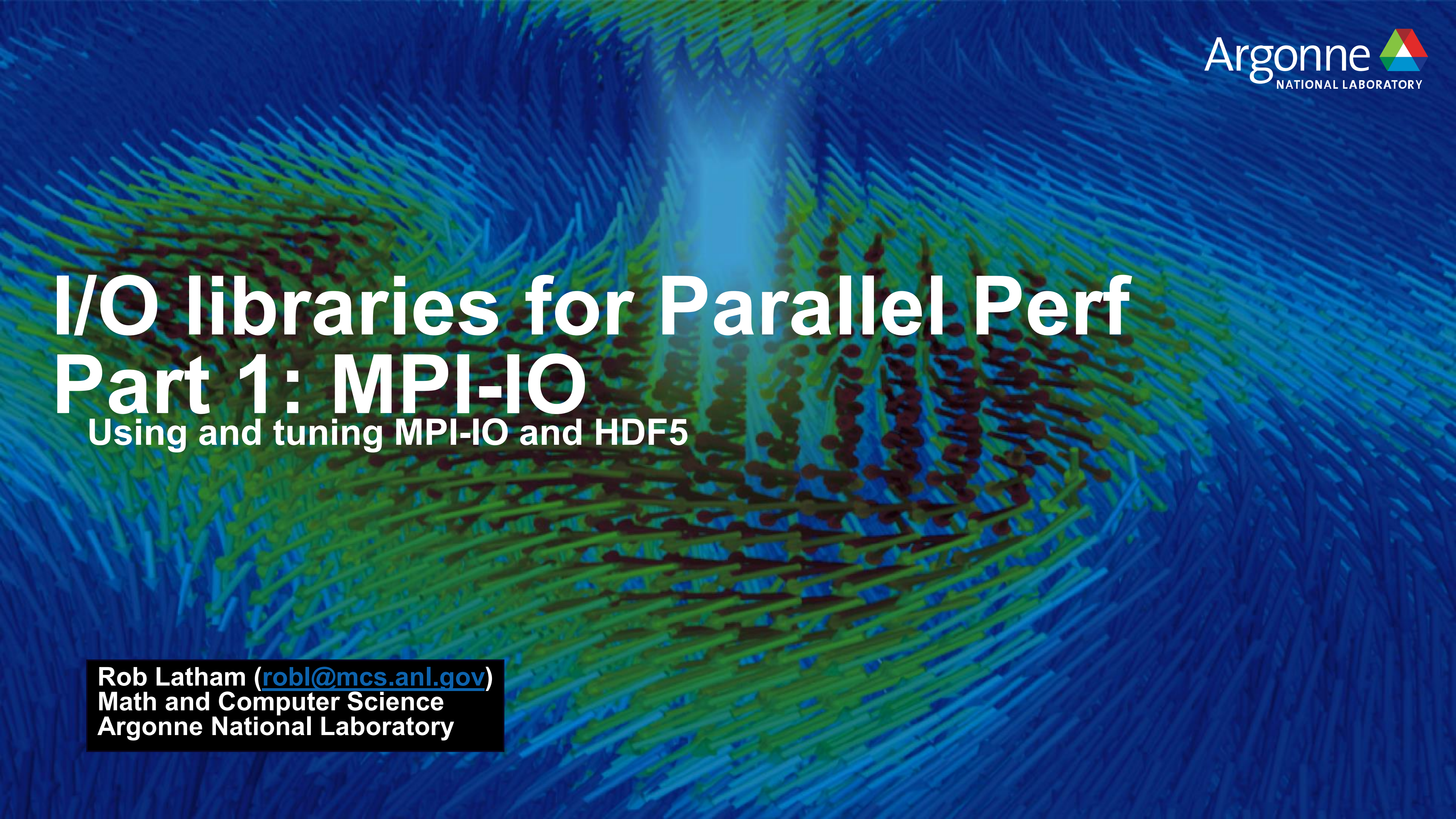
October 29-31, 2024

ALCF Hands-on HPC Workshop



I/O libraries for Parallel Perf Part 1: MPI-IO

Using and tuning MPI-IO and HDF5

Rob Latham (robl@mcs.anl.gov)
Math and Computer Science
Argonne National Laboratory

MPI-IO

- I/O interface specification for use in MPI apps

- Data model is same as POSIX: stream of bytes in a file
- Like classic POSIX in some ways…
  - Open() → MPI_File_open()
  - Pwrite() → MPI_File_write()
  - Close() → MPI_File_close()
- Features many improvements over POSIX:
  - Collective I/O
  - Noncontiguous I/O with MPI datatypes and file views
  - Nonblocking I/O
  - Fortran bindings (and additional languages)
- Implementations available on most (all?) platforms

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

"Hello World" MPI-IO style: contiguous

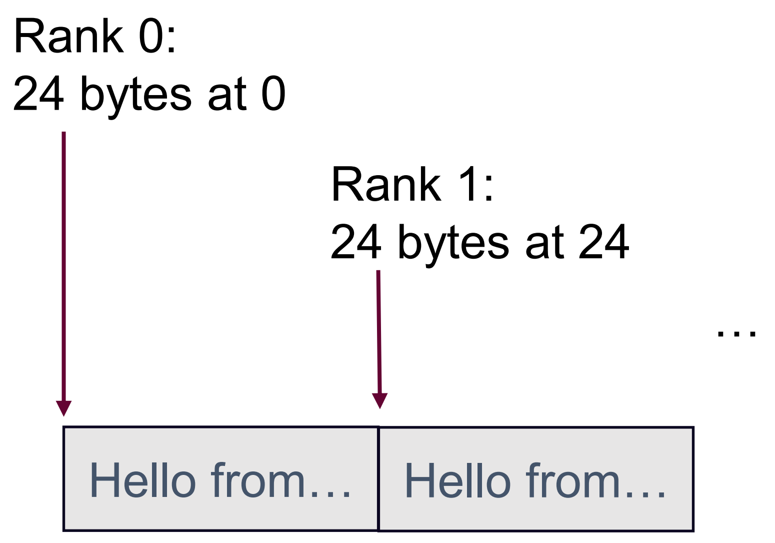
```
/* an "Info object":  these store key-value strings for tuning the
 * underlying MPI-IO implementation */
MPI_Info_create(&info);

snprintf(buf, BUFSIZE, "Hello from rank %d of %d\n", rank, nprocs);
len = strlen(buf);
/* We're working with strings here but this approach works well
 * whenever amounts of data vary from process to process. */
MPI_Exscan(&len, &offset, 1, MPI_OFFSET, MPI_SUM, MPI_COMM_WORLD);

MPI_CHECK(MPI_File_open(MPI_COMM_WORLD, argv[1],
            MPI_MODE_CREATE|MPI_MODE_WRONLY, info, &fh));

/* _all means collective.  Even if we had no data to write, we would
 * still have to make this call.  In exchange for this coordination,
 * the underlyng library might be able to greatly optimize the I/O */
MPI_CHECK(MPI_File_write_at_all(fh, offset, buf, len, MPI_CHAR,
            &status));

MPI_CHECK(MPI_File_close(&fh));
```



code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

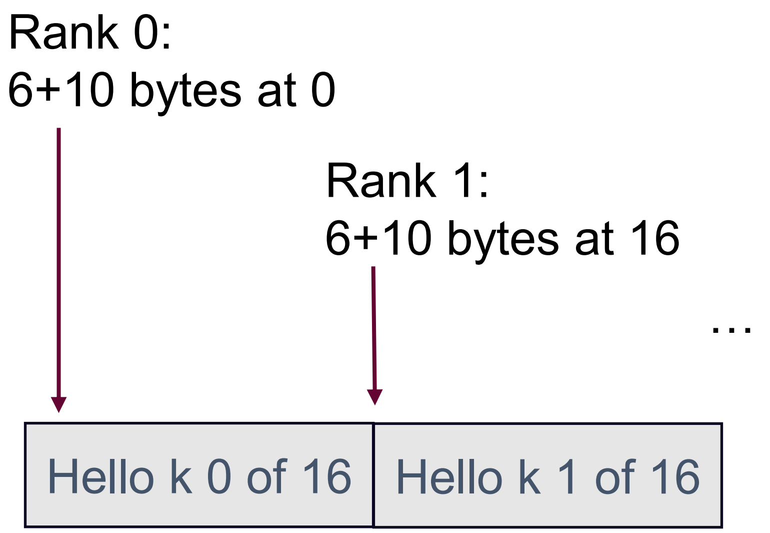
“Hello World” MPI-IO style: non-contiguous in memory

```
MPI_Datatype memtype;
MPI_Count memtype_size;

…
/* sample string:
 * Hello from rank 8 of 16
 * ------        ----------
 *
 * the '-' indicates which elements an indexed type with
 *  lengths 6 and 10  at displacemnts 0 and
 * "10 from end of string" would select: */
int lengths[2] = {6, 10};
int displacements[2] = {0, len-10};
MPI_Type_indexed(2, lengths, displacements, MPI_CHAR, &memtype);
MPI_Type_commit(&memtype);
MPI_Type_size_x(memtype, &memtype_size);
…
MPI_CHECK(MPI_File_write_at_all(fh, offset, buf, 1, memtype,
              &status));
```

Hello from rank 1 of 16

‘lengths” and “displacements”: each rank sends first six and last ten characters to file



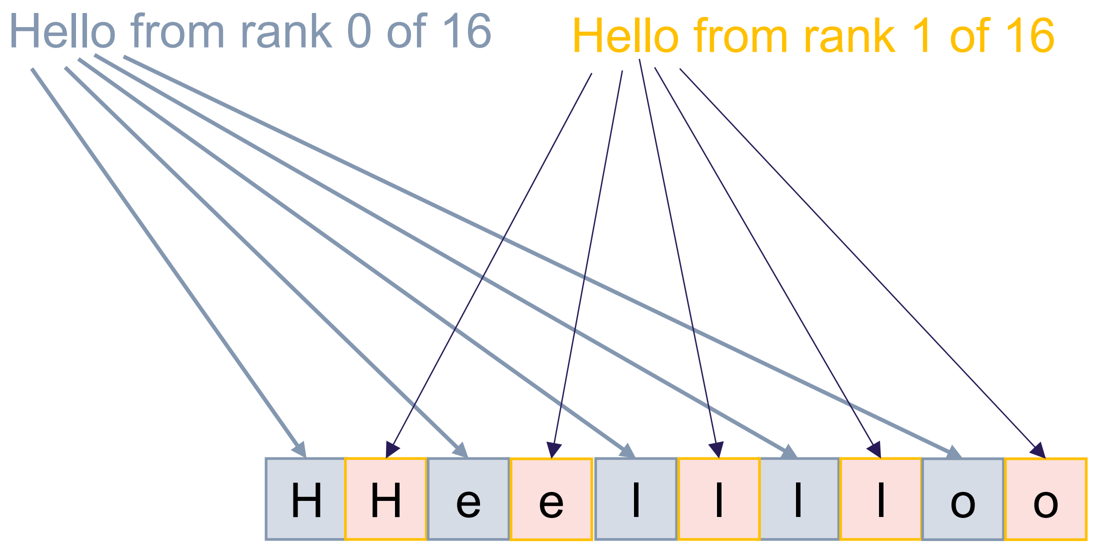
code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

“Hello World” MPI-IO style: non-contiguous in file

```
/* noncontiguous in file requres a "file view*/
MPI_Datatype viewtype;
int *displacements;
displacements = malloc(len*sizeof(*displacements));

/* each process will write to its own "view" of the file:
 * Rank 0:
 * H e l l o   f r o m  ...
 * Rank 1:
 *  H e l l o  f r o m ...
 */
for (int i=0; i< len; i++)
    displacements[i] = rank+(i*nprocs);
MPI_Type_create_indexed_block(len, 1, displacements, MPI_CHAR, &viewtype);
MPI_Type_commit(&viewtype);
free(displacements);

MPI_CHECK(MPI_File_open(MPI_COMM_WORLD, argv[1],
            MPI_MODE_CREATE|MPI_MODE_WRONLY, info, &fh));
MPI_CHECK(MPI_File_set_view(fh, 0, MPI_CHAR, viewtype, "native", info));
MPI_CHECK(MPI_File_write_at_all(fh, offset, buf, len, MPI_CHAR,
            &status));
```



While this access describes lots of small regions, the library sees it as one single access and can optimize.

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

RUNNING

- Submit to the “HandsOnHPCScale” queue and use the “alcf_training” account (polaris)
  - qsub -q HandsOnHPCScale -A alcf_training …
- Which file system to use?
  - Tried to make scripts do right thing by default
  - Please don’t use the NFS-mounted home directory
  - Given scripts should already point you to the right parallel directory
    - Polaris: `/grand/alcf_training/HandsOnHPC24/$USER`
- Make a directory for your data
  - `Polaris: mkdir -p /grand/alcf_training/HandsOnHPC24/$USER/`
- Set sensible striping (more on that later)
  - `lfs setstripe –stripe-count -1 /grand/alcf_training/HandsOnHPC24/$USER/`

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

Running on Polaris

```
#!/bin/bash -l
#PBS -A ATPESC2024
#PBS -l walltime=00:10:00
#PBS -l select=1
#PBS -l place=scatter
#PBS -l filesystems=home:eagle
#PBS -q debug
#PBS -N hello-io
#PBS -V

OUTPUT=/eagle/ATPESC2024/usr/${USER}/hello
mkdir -p ${OUTPUT}

NNODES=$(wc -l < $PBS_NODEFILE)
NRANKS_PER_NODE=32
NTOTRANKS=$(( NNODES * NRANKS_PER_NODE ))

cd $PBS_O_WORKDIR

mpiexec -n $NTOTRANKS --ppn $NRANKS_PER_NODE
              ./hello-mpiio ${OUTPUT}/hello.out

mpiexec -n $NTOTRANKS --ppn $NRANKS_PER_NODE
              ./hello-mpiio-noncontig ${OUTPUT}/hello-noncontig.out

mpiexec -n $NTOTRANKS --ppn $NRANKS_PER_NODE
              ./hello-mpiio-view ${OUTPUT}/hello-view.out
```

Job submission script

```
% cat /eagle/ATPESC2024/usr/${USER}/hello.out
Hello from rank 0 of 32
Hello from rank 1 of 32
…
Hello from rank 30 of 32
Hello from rank 31 of 32

$ cat  /eagle/ATPESC2024/usr/${USER}/hello/hello-noncontig.out
Hello k 0 of 32
Hello k 1 of 32
Hello k 2 of 32
…
Hello  30 of 32
Hello  31 of 32

$ cat  /eagle/ATPESC2024/usr/${USER}/hello/hello-view.out
HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
llllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllll
oooooooooooooooooooooooooooooooo
fffffff…
```

Output of our hello programs

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

“Hello World” MPI-IO style

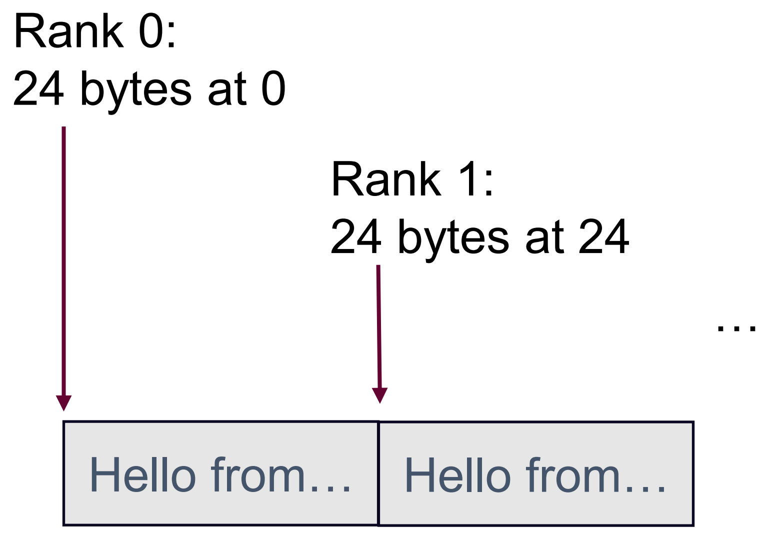
```
/* an "Info object":  these store key-value strings for tuning the
 * underlying MPI-IO implementation */
MPI_Info_create(&info);

snprintf(buf, BUFSIZE, "Hello from rank %d of %d\n", rank, nprocs);
len = strlen(buf);
/* We're working with strings here but this approach works well
 * whenever amounts of data vary from process to process. */
MPI_Exscan(&len, &offset, 1, MPI_OFFSET, MPI_SUM, MPI_COMM_WORLD);

MPI_CHECK(MPI_File_open(MPI_COMM_WORLD, argv[1],
            MPI_MODE_CREATE|MPI_MODE_WRONLY, info, &fh));

/* _all means collective.  Even if we had no data to write, we would
 * still have to make this call.  In exchange for this coordination,
 * the underlyng library might be able to greatly optimize the I/O */
MPI_CHECK(MPI_File_write_at_all(fh, offset, buf, len, MPI_CHAR,
            &status));

MPI_CHECK(MPI_File_close(&fh));
```



code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

Running on Polaris

```
#!/bin/bash -l
#PBS -A fallwkshp23
#PBS -l walltime=00:10:00
#PBS -l select=1
#PBS -l place=scatter
#PBS -l filesystems=home:eagle
#PBS -q debug
#PBS -N hello-io
#PBS -V

mkdir -p /eagle/fallwkshp23/${USER}

NNODES=$(wc -l < $PBS_NODEFILE)
NRANKS_PER_NODE=32
NTOTRANKS=$(( NNODES * NRANKS_PER_NODE ))

cd $PBS_O_WORKDIR
mpiexec -n $NTOTRANKS --ppn $NRANKS_PER_NODE \
            ./hello-mpiio /eagle/fallwkshp23/${USER}/hello.out
```

Job submission script

```
% cat /eagle/fallwkshp23/${USER}/hello.out
Hello from rank 0 of 32
Hello from rank 1 of 32
Hello from rank 2 of 32
Hello from rank 3 of 32
Hello from rank 4 of 32
…
Hello from rank 29 of 32
Hello from rank 30 of 32
Hello from rank 31 of 32
```

Output of “hello-mpiio”

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

Key takeaways

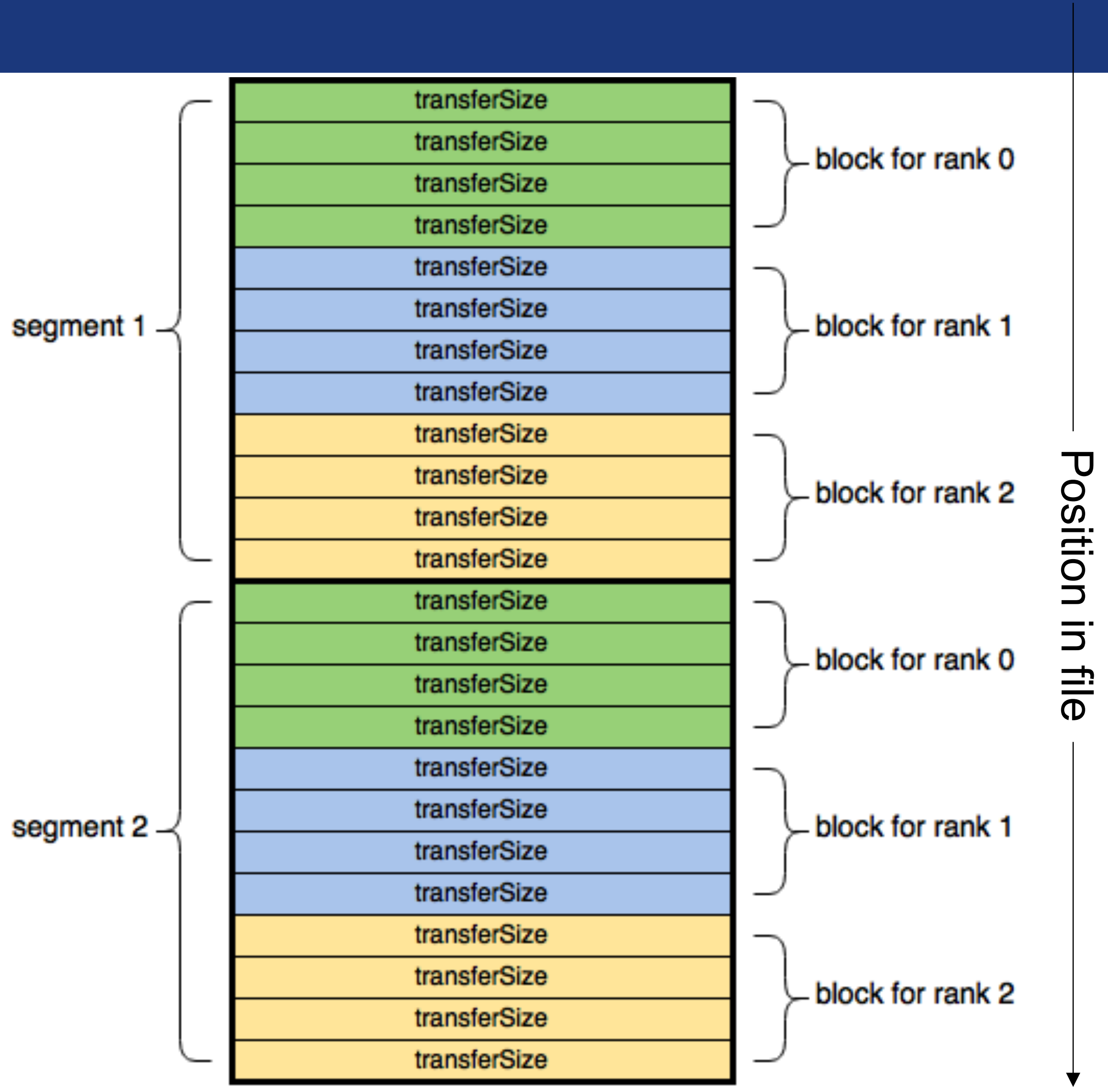
- Simple example but still captures important concepts
  - Info objects: tuning parameters:
    - enable/disable optimizations
    - Adjust buffer sizes
    - Select alternate strategies
  - Data placement in file specified by user
    - “shared file pointer” possible but not optimized
  - Collective vs independent I/O
  - Error checking!!!

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

The IOR benchmark

- MPI application benchmark
  - reads and writes data in configurable ways
  - I/O pattern can be interleaved or random
- Input:
  - transfer size, block size, segment count
  - interleaved or random
- Output: Bandwidth and IOPS
- Configurable backends
  - POSIX, STDIO, MPI-IO
  - HDF5, PnetCDF, S3, rados

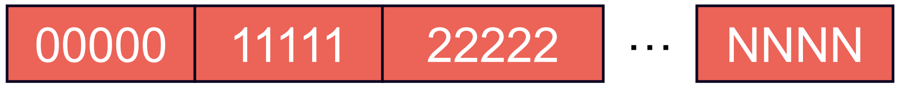
https://github.com/hpc/ior



Hands-on: IOR and stripe size

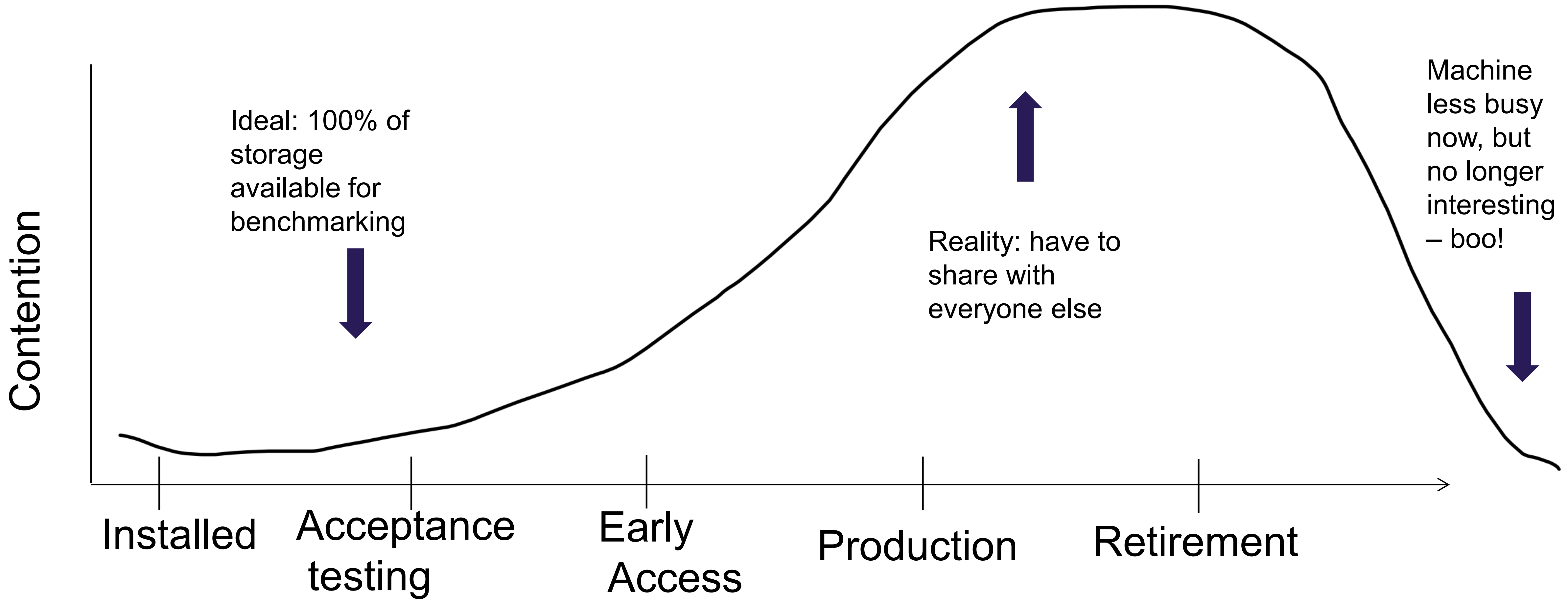
- For a fixed number of nodes, MPI processes, block size, and transfer size…
- Vary the stripe count
  - IOR environment variables
  - Cray MPI-IO environment variables
  - `lfs setstripe`

```
$stripe=1
rm  -f ${OUTPUT}/ior-stripe-$stripe.out
export IOR_HINT__MPI__striping_factor=$stripe
  # -a MPIIO: using MPI-IO so we can pass the "striping_factor" hint
  # -e      : fsync after each write phase: push out dirty data to storage
  # -C      : reorder ranks: read from a different rank than the one that wrote
  # -s      : segments: each client will write to eight regions
  # -i      : repeat experiment five times: lots of variability in I/O
  # -t      : transfer size: how big each request will be
  # -b      : block size:  how big each region will be in the file (needs to
                    be a multiple of transfer size).
mpiexec -n ${NTOTRANKS} --ppn ${NRANKS_PER_NODE} \
       ior --mpiio.showHints -a MPIIO \
          -e -C -s 8 -i 5 \
          -t 1MiB -b 64MiB -o ${OUTPUT}/ior-stripe-$stripe.out
```



code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

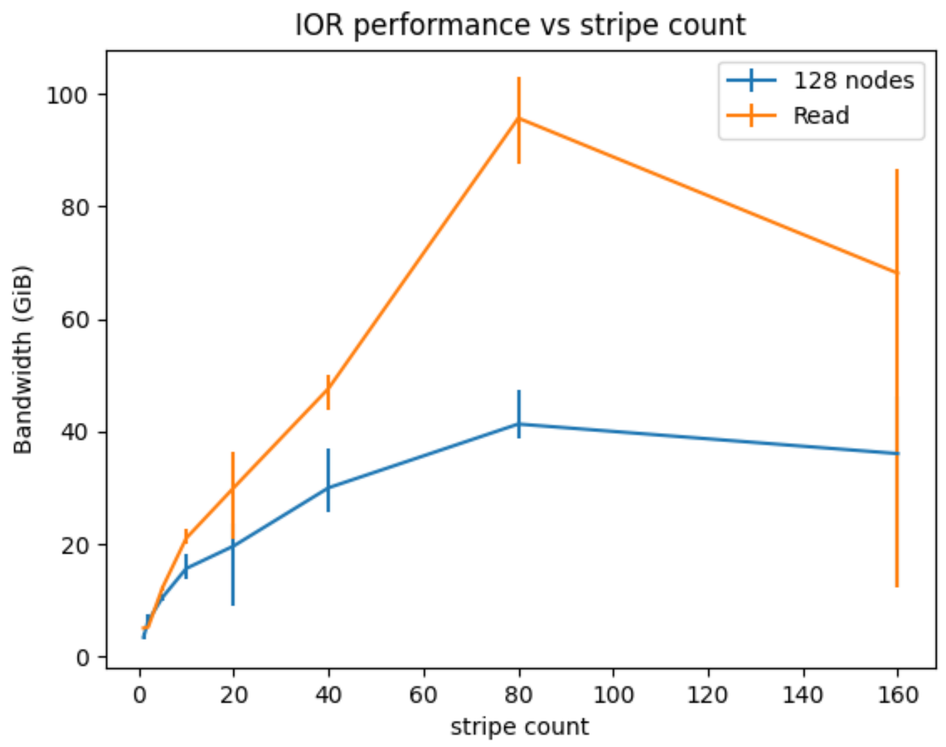
Contention in benchmarkig



code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

Hands on: IOR and stripe count

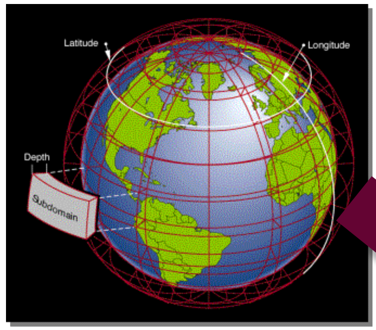
- Default stripe size is 1
  - Why? Most files small: optimizing for common case
- "All the servers" doesn't seem to hurt performance here
  - lfs setstripe -1 /path/to/file
- Could go further with "overstriping"
  - Didn't work on Polaris: investigating
- "Where's my bandwidth?"
  - 128 nodes (network links) here
  - Shared file (so I can experiment with stripe count) means lustre locking overhead/coordination
- Graph at right from February 2023 – any changes today?



visualization_io/mpiio-hdf5/io-sleuthing/examples/striping

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

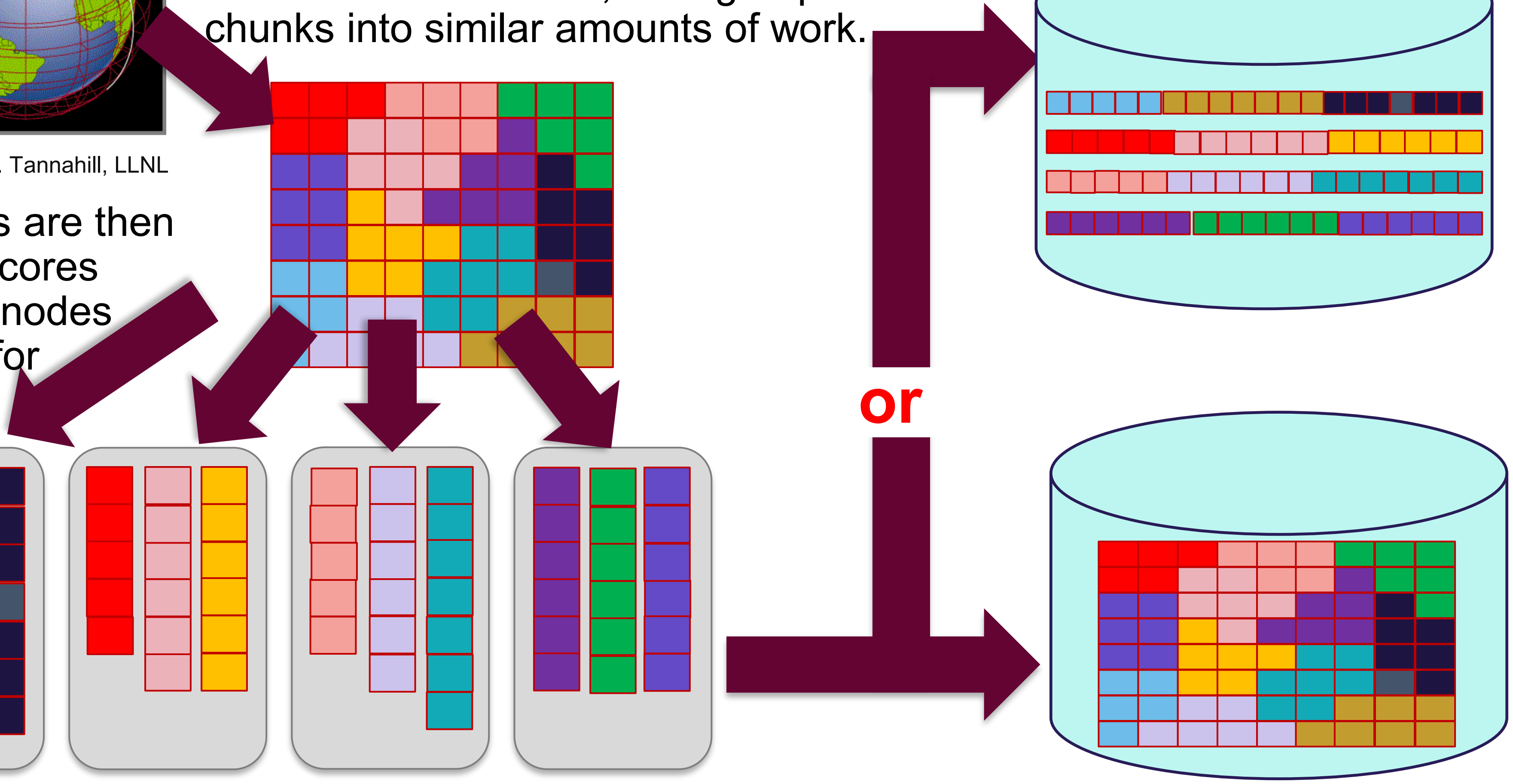
Decomposition



Graphic from J. Tannahill, LLNL

Typical simulations divide up the region being simulated into chunks, then group those chunks into similar amounts of work.

These regions are then distributed to cores (columns) on nodes (grey boxes) for computation.



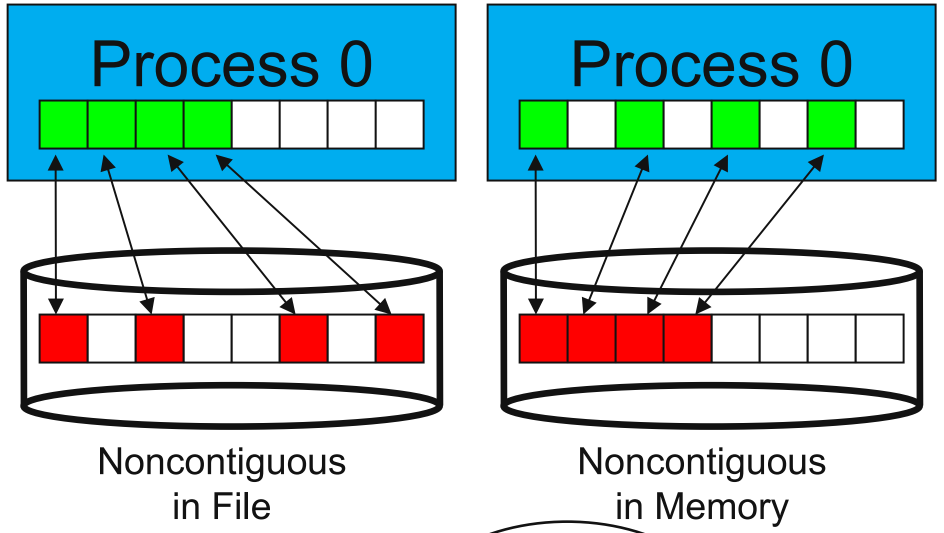
When speed of writing is the priority, ***blobs*** of data are written from each node into individual files that must then be post-processed for analysis.

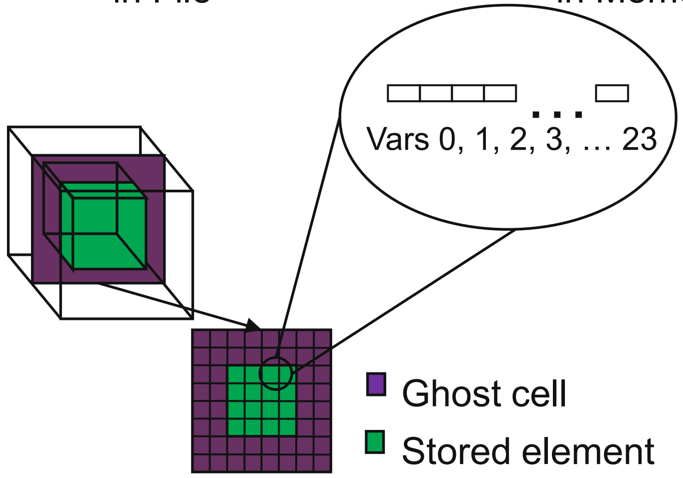
To prepare data for analysis, a code can write in a ***canonica***l view by processing the data while it is in memory, resulting in a better organized dataset.

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

Contiguous and Noncontiguous I/O

- Contiguous I/O moves data from a single memory block into a single file region
- Noncontiguous I/O has three forms:
  - Noncontiguous in memory
  - Noncontiguous in file
  - Noncontiguous in both
- Structured data leads naturally to noncontiguous I/O (e.g., block decomposition)
- Describing noncontiguous accesses with a single operation passes more knowledge to I/O system





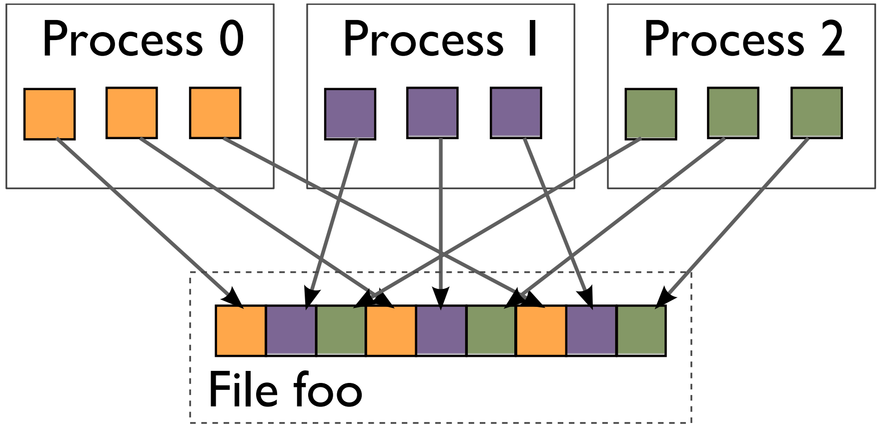
Extracting variables from a block and skipping ghost cells will result in noncontiguous I/O

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

I/O Transformations

Software between the application and the PFS performs transformations, primarily to improve performance

- Goals of transformations:
  - Reduce number of I/O operations to PFS (avoid latency, improve bandwidth)
  - Avoid lock contention (eliminate serialization)
  - Hide huge number of clients from PFS servers
- "Transparent" transformations don't change the final file layout
  - File system is still aware of the actual data organization
  - File can be later manipulated using serial POSIX I/O



When we think about I/O transformations, we consider the mapping of data between application processes and locations in file

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

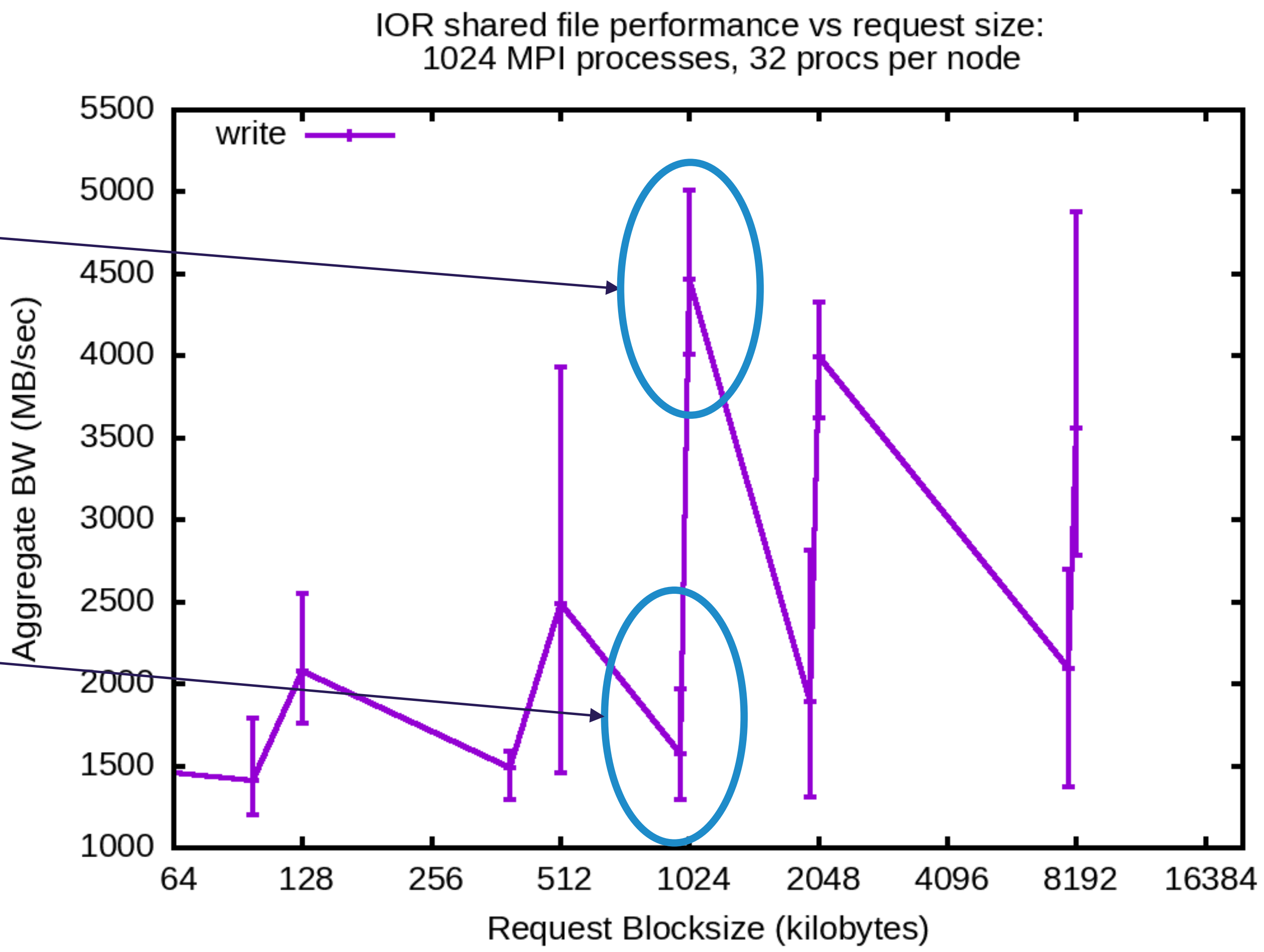
Request Size and I/O Rate

Request matches Lustre "stripe size": good performance with low variability

Small deviations from "power of two" (e.g. 1024k vs 10^6) can tank performance

In general, larger requests better.

19

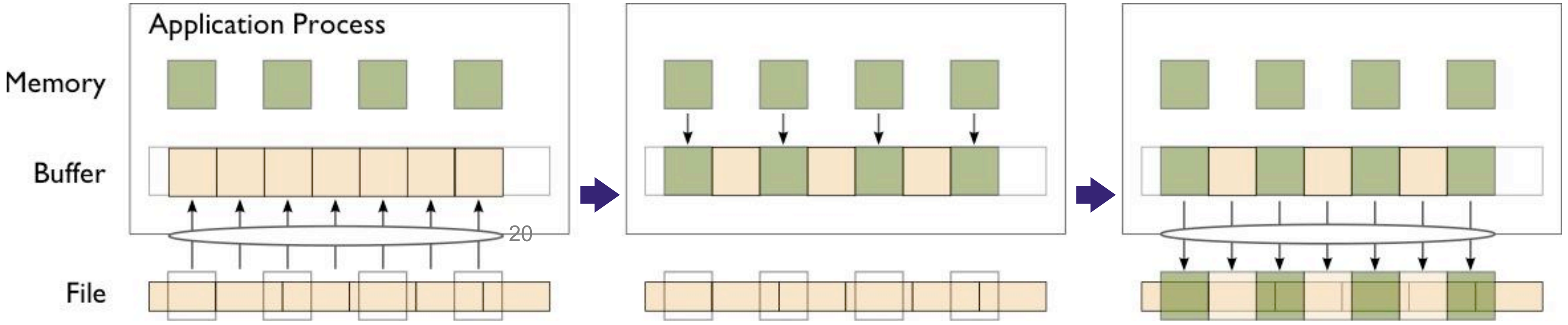


Tests run on 1K processes of HPE/Cray Theta at Argonne

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

Reducing Number, Increasing Size of Operations

- **Because most operations go over the network, I/O to a PFS incurs more latency than with a local FS**
- *Data sieving* is a technique to address I/O latency by combining operations:
  - When reading, application process reads a large region holding all needed data and pulls out what is needed
  - When writing, three steps required (below)



Step 1: Data in region to be modified are read into intermediate buffer (1 read).

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

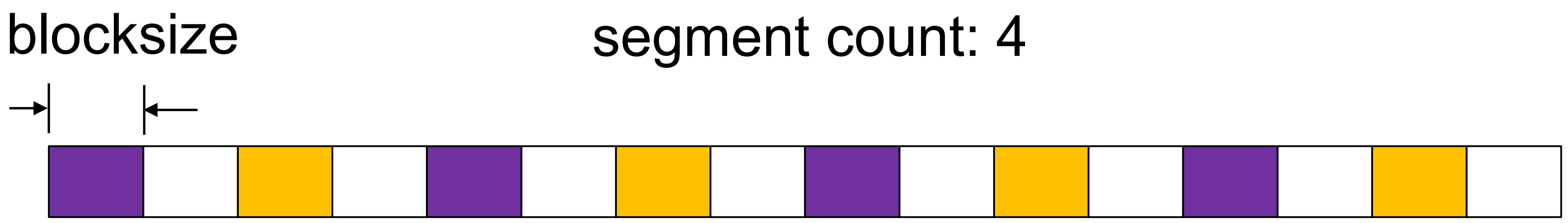
Step 3: Entire region is written back to storage with a single write operation.

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

Noncontig with IOR

- IOR can describe access with an MPI datatype
  - `--mpiio.useStridedDatatype –b … -s …`

- (buggy in recent versions: use 4.0rc1 or newer)



code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

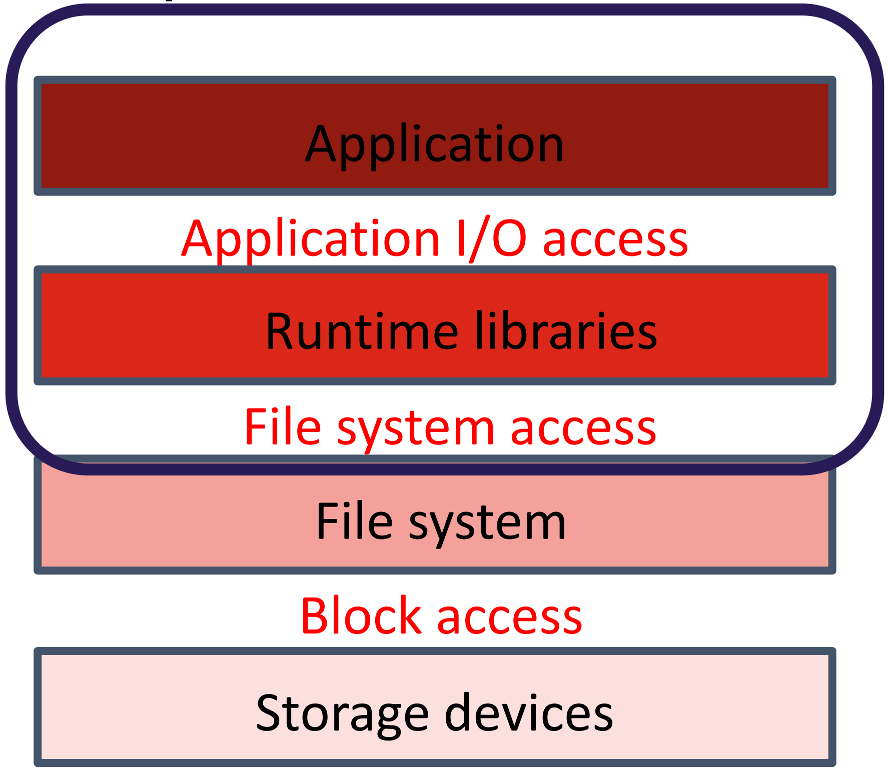
Darshan: Characterizing Application I/O

How is an application using the I/O system?
How successful is it at attaining high performance?

Strategy: observe I/O behavior at the application and library level

- What did the application intend to do?
- How much time did it take to do it?
- What can be done to tune and improve?

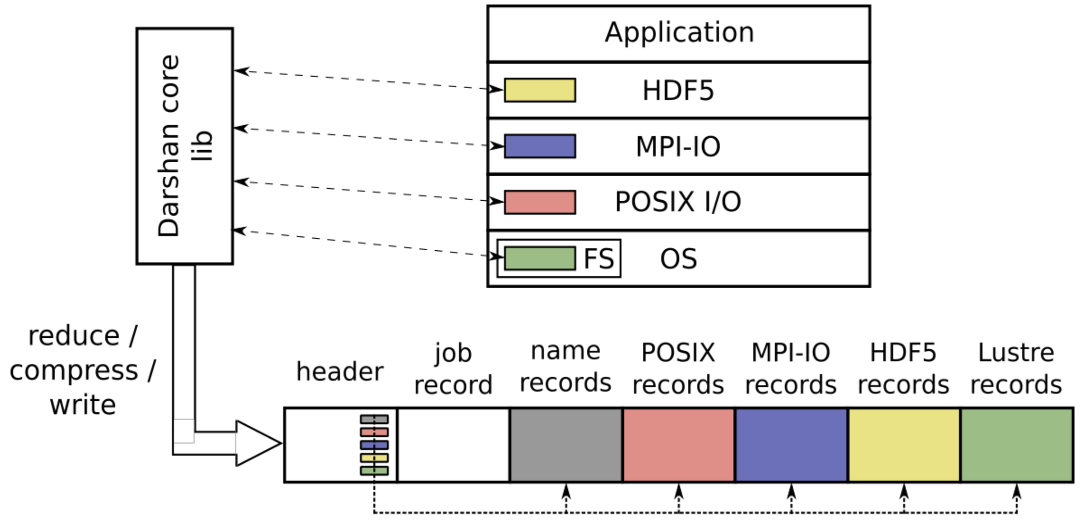
Simplified HPC I/O stack



code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

How does Darshan work?

- Darshan records file access statistics independently on each process
- At app shutdown, collect, aggregate, compress, and write log data
- After job completes, analyze Darshan log data
  - `darshan-parser` - provides complete text-format dump of all counters in a log file
  - *PyDarshan* - Python analysis module for Darshan logs, including a summary tool for creating HTML reports



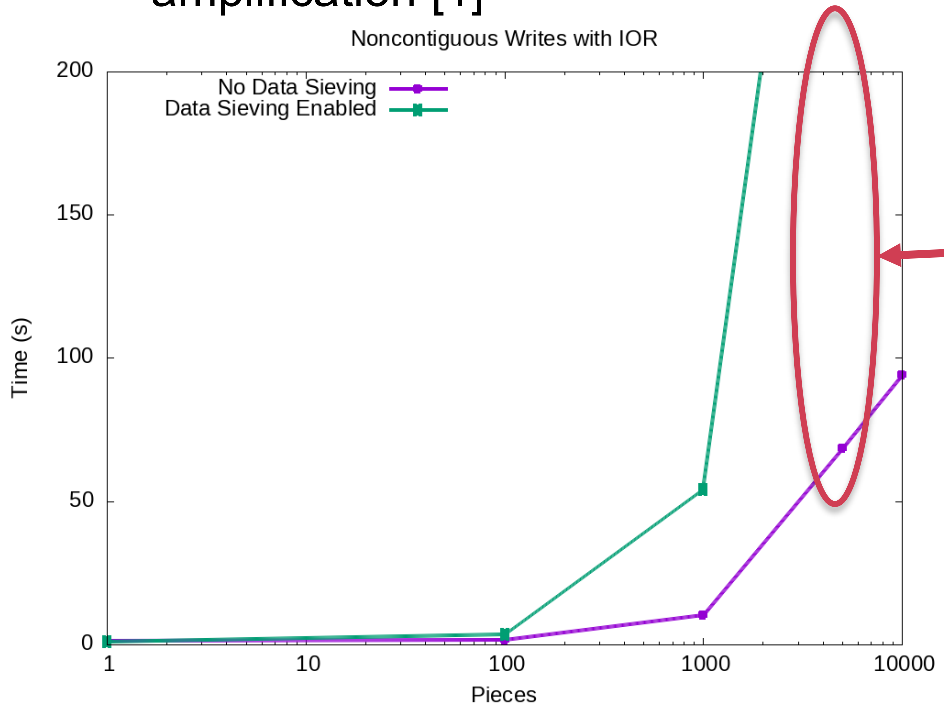
- Originally designed for MPI applications, but in recent Darshan versions (3.2+) any dynamically-linked executable can be instrumented
  - ➢ In MPI mode, a log is generated for each *app*
  - ➢ In non-MPI mode, a log is generated for each *process*

➢ *More information: https://docs.alcf.anl.gov/theta/performance-tools/darshan/ or Shane's (concurrent) session*

Data Sieving in Practice

Not always a win, particularly for writing:

- IOR benchmark, fixed file size, increasing segments
- Enabling data sieving instead made writes slower: why?
  - Locking to prevent false sharing (not needed for reads)
  - Multiple processes per node writing simultaneously
  - Internal ROMIO buffer too small, resulting in write amplification [1]



[1]

	Naiive	Data Sieving
MPI-IO writes	960	960
MPI-IO Reads	0	0
Posix Writes	4 800 000	4 800 000
Posix Reads	0	4 800 784
MPI-IO bytes written	8.9 GiB	8.9 GiB
MPI-IO bytes read	0	0
Posix bytes read	0	2334 GiB
Posix bytes written	8.9 GiB	2343 GiB
Runtime (sec)	68.8	404.2

Selected Darshan statistics for 5000 segments

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

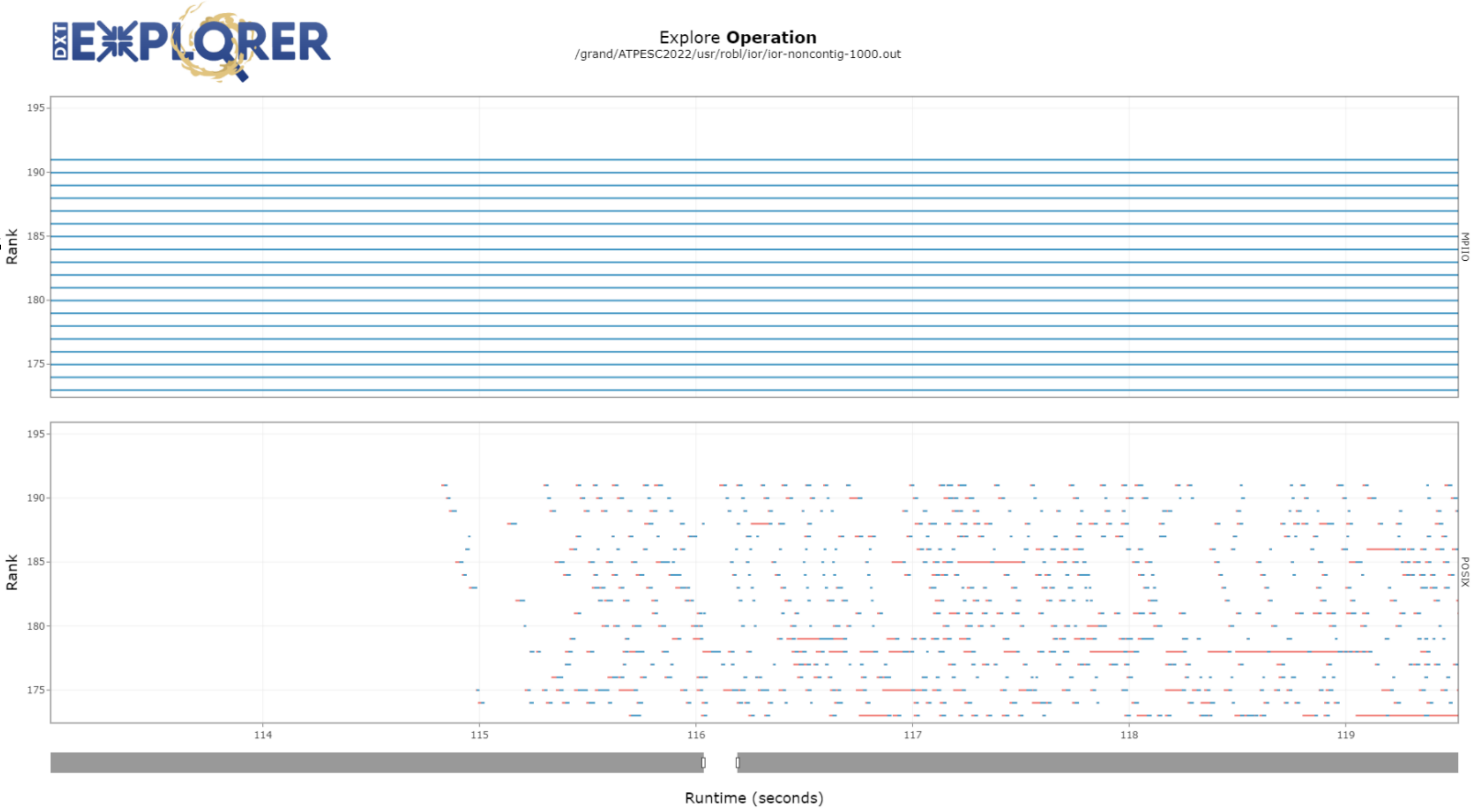
Data Sieving: time line

Top: MPI I/O call describing noncontiguous regions

One MPI I/O call (top) turns into many POSIX operations (below)

Independent: no coordination possible. Each process does its own data sieving.

Gaps between operations show lock acquisition.

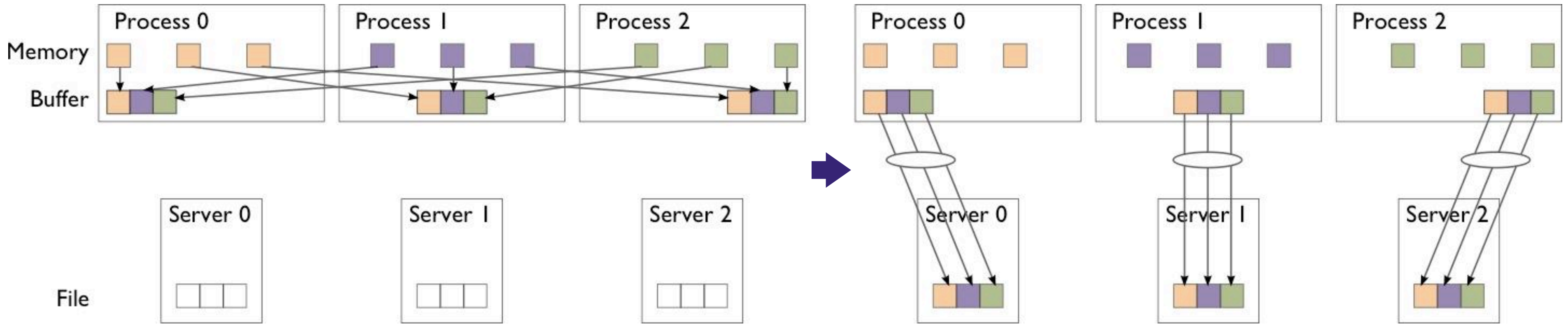


https://github.com/hpc-io/dxt-explorer Interactive log analysis tool by Jean Luca Bez

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

Avoiding Lock Contention

- **To avoid lock contention when writing to a shared file, we can reorganize data between processes**
- *Two-phase I/O* splits I/O into a data reorganization phase and an interaction with the storage system (two-phase write depicted):
  - Data exchanged between processes to match file layout
  - 0th phase determines exchange schedule (not shown)



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

Phase 2: Data are written to file (storage servers) with large writes, no contention.

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

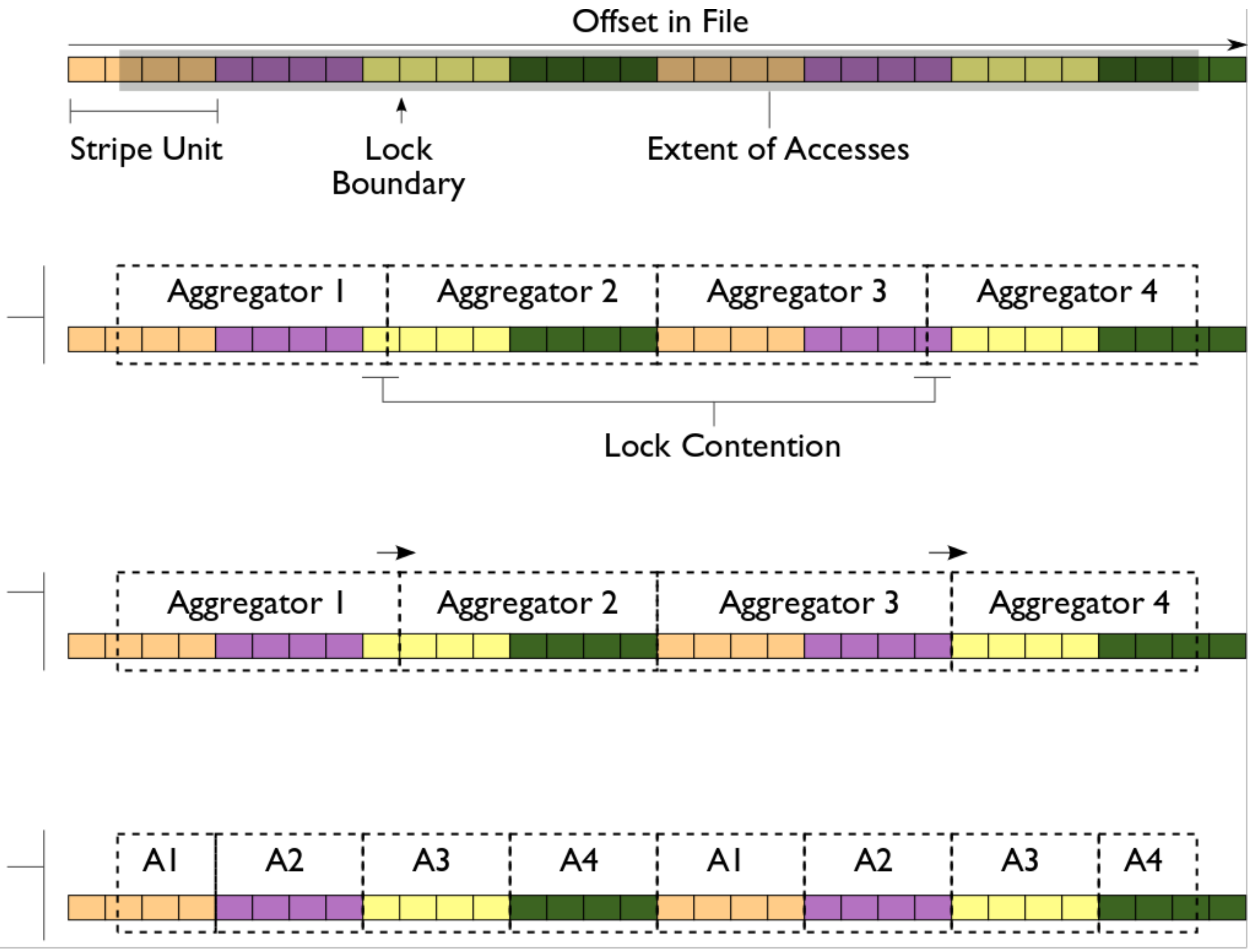
Two-Phase I/O Algorithms

Imagine a collective I/O access using four aggregators to a file striped over four file servers (indicated by colors):

One approach is to evenly divide the region accessed across aggregators.

Aligning regions with lock boundaries eliminates lock contention.

Mapping aggregators to servers reduces the number of concurrent operations on a single server and can be helpful when locks are handed out on a per-server basis (e.g., Lustre).

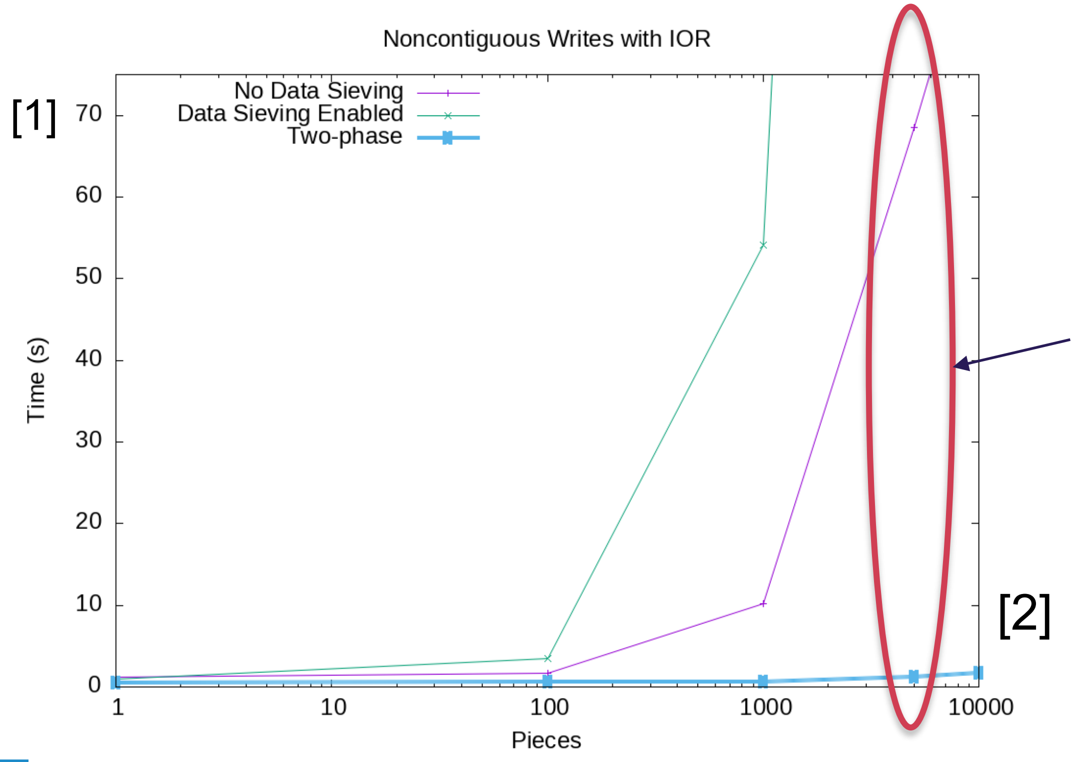


For more information, see W.K. Liao and A. Choudhary, “Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols,” SC2008, November 2008.

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

Two-phase I/O in Practice

- Consistent performance independent of access pattern
    - Note re-scaled y axis [1]
- No write amplification, no read-modify-write
- Some network communication but networks are fast
- Requires "temporal locality" -- not great if writes "skewed", imbalanced, or some process enter collective late.



	Naiive	Data Sieving	Two-phase
MPI-IO writes	960	960	960
MPI-IO Reads	0	0	0
Posix Writes	4 800 000	4 800 000	9156
Posix Reads	0	4 800 784	0
MPI-IO bytes written	8.9 GiB	8.9 GiB	8.9 GiB
MPI-IO bytes read	0	0	0
Posix bytes read	0	2334 GiB	0
Posix bytes written	8.9 GiB	2343 GiB	8.9 GiB
Runtime (sec)	68.8	404.2	1.56

Selected Darshan statistics, 5000 segments

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

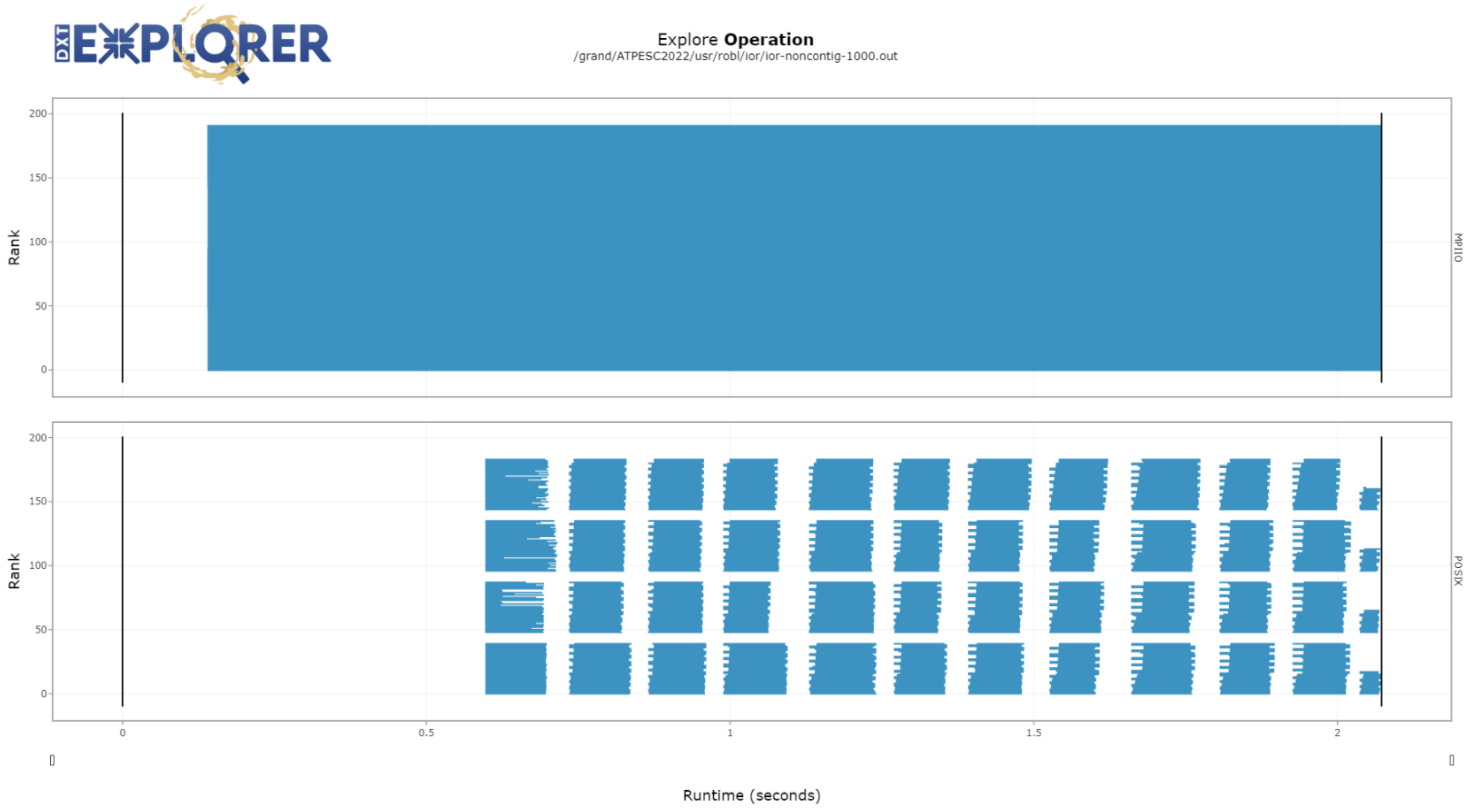
Two-phase I/O: time line

Top: collective MPI I/O call describing noncontiguous regions

One collective MPI I/O call per process: library transforms request.

Lustre-specific optimization: select processes and request sizes based on file stripe size, stripe count.

Gaps between operations show data exchange over network



code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

Tuning MPI-IO: info objects

- You will likely never need these, but can help in specific situations:
- Both keys and values are strings
- Applicable to all ROMIO-based MPI-IO libraries

Hint	Default Value	effect
cb_buffer_size	16777216	An internal buffer for “two phase i/o”. Bigger value takes away application memory, but results in fewer rounds of I/O
romio_cb_read romio_cb_write	Enable (on cray) automatic (ROMIO)	Turn on/off collective i/o: code will fall through to independent case
romio_no_indep_rw cb_config_list	True “*:*” (on Cray) or “*.1” elsewhere	“deferred open” – only i/o aggregators open the file. Open time not usually dominant factor unless total I/O moved per file fairly small

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

Tuning MPI-IO: cray-specific hints

- Hints that only work on Cray systems
- Perfectly fine to pass these (or anything) to any MPI library: libraries will ignore hints they don't recognize.
- More cray tuning at https://cpe.ext.hpe.com/docs/mpt/mpich/intro_mpi.html#mpi-io-environment-variables

Info key	Default value	effect
cray_cb_write_lock_mode	0	Set to "2" to try out "lock ahead": should allow greater concurrency
cray_cb_nodes_multiplier	1	Depending on stripe size and number of nodes, "2" or more might improve performance

Data Model Libraries

- Scientific applications work with structured data and desire more self-describing file formats
- PnetCDF and HDF5 are two popular “higher level” I/O libraries
  - Abstract away details of file layout
  - Provide standard, portable file formats
  - Include metadata describing contents
- For parallel machines, these use MPI and probably MPI-IO
  - MPI-IO implementations are sometimes poor on specific platforms, in which case libraries might directly call POSIX calls instead

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

The Parallel netCDF Interface and File Format

- Thanks to Wei-Keng Liao, Alok Choudhary, and Kaiyuan Hou (NWU) for their help in the development of PnetCDF.

- https://parallel-netcdf.github.io/

Parallel NetCDF (PnetCDF)

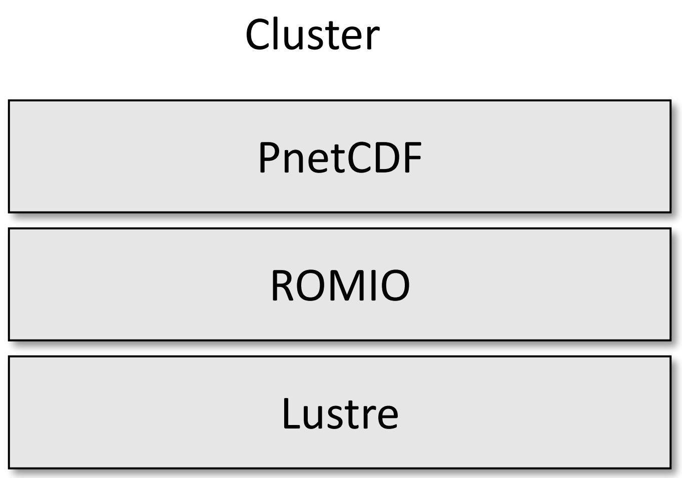
- Based on original "Network Common Data Format" (netCDF) work from Unidata
  - Derived from their source code
- Data Model:
  - Collection of variables in single file
  - Typed, multidimensional array variables
  - Attributes on file and variables
- Features:
  - C, Fortran, and F90 interfaces (no python)
  - Portable data format (identical to netCDF)
  - Noncontiguous I/O in memory using MPI datatypes
  - Noncontiguous I/O in file using sub-arrays
  - Collective I/O
  - Non-blocking I/O
- Unrelated to netCDF-4 work
- Parallel-NetCDF tutorial:
  - https://parallel-netcdf.github.io/wiki/QuickTutorial.html
- Interface guide:
  - http://cucis.ece.northwestern.edu/projects/PnetCDF/doc/pnetcdf-c/index.html
  - 'man pnetcdf' on polaris (after loading module)

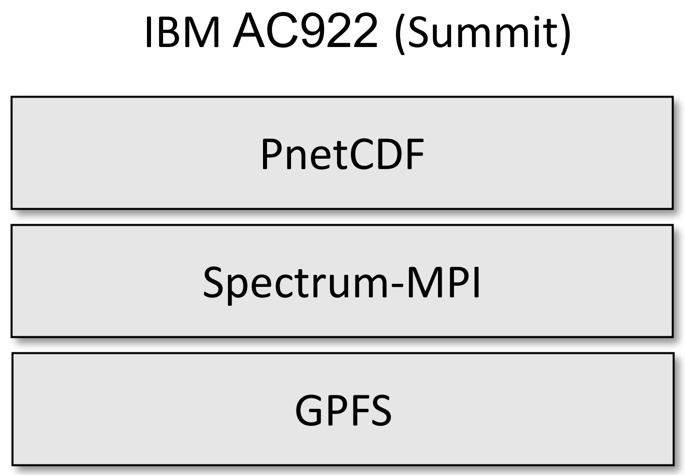
code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

Parallel netCDF (PnetCDF)

- (Serial) netCDF
  - API for accessing multi-dimensional data sets
  - Portable file format
  - Popular in both fusion and climate communities

- Parallel netCDF
  - Very similar API to netCDF
  - Tuned for better performance in today's computing environments
  - Retains the file format so netCDF and PnetCDF applications can share files
  - PnetCDF builds on top of any MPI-IO implementation

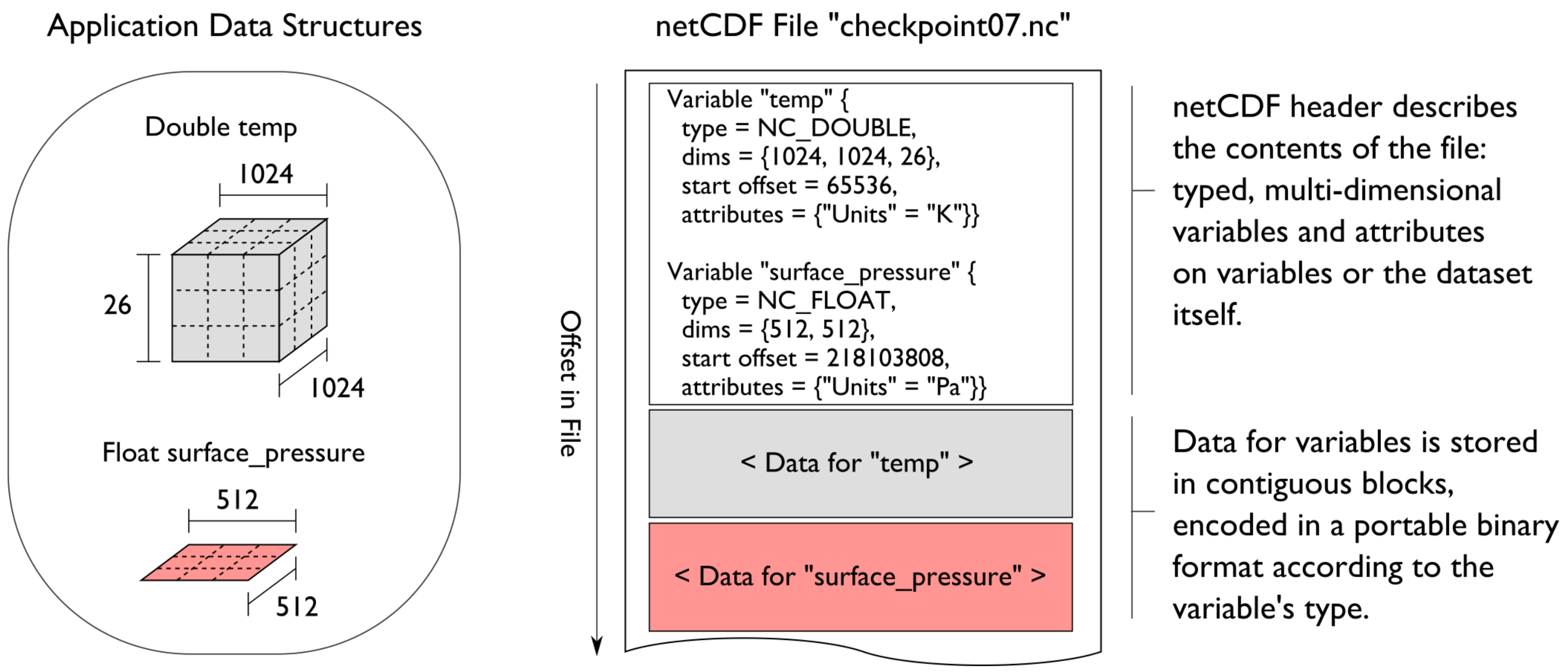




code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

netCDF Data Model

- **The netCDF model provides a means for storing multiple, multi-dimensional arrays in a single file.**


Application Data Structures
Double temp
1024
26
1024
Float surface_pressure
512
512
netCDF File "checkpoint07.nc"
Offset in File
< Data for "temp" >
< Data for "surface_pressure" >

```
Variable "temp" {
  type = NC_DOUBLE,
  dims = {1024, 1024, 26},
  start offset = 65536,
  attributes = {"Units" = "K"}}

Variable "surface_pressure" {
  type = NC_FLOAT,
  dims = {512, 512},
  start offset = 218103808,
  attributes = {"Units" = "Pa"}}
```

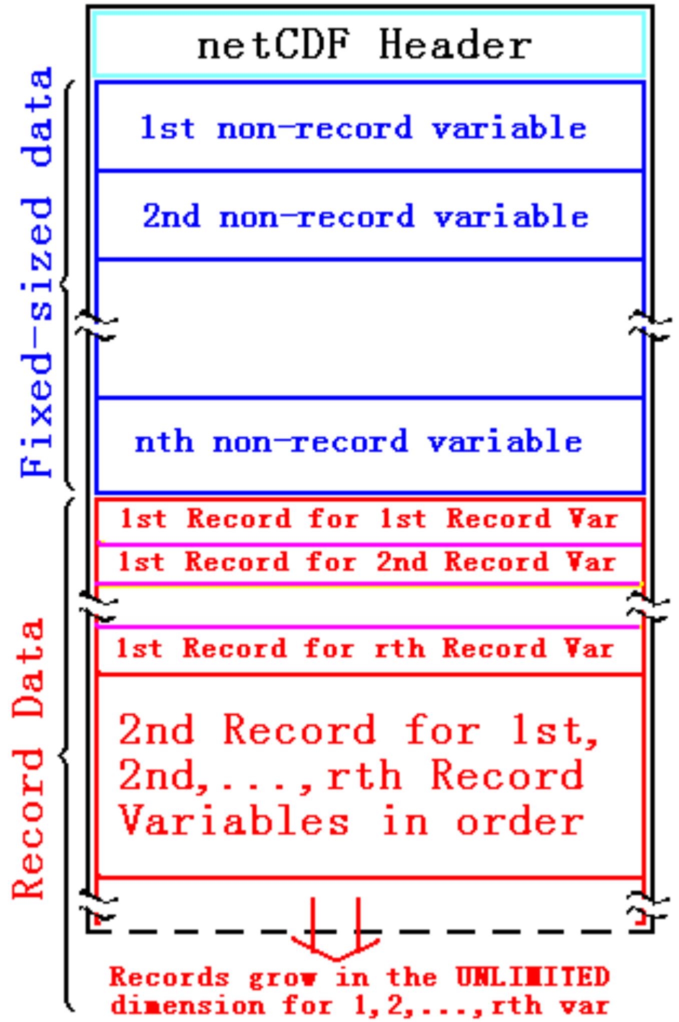
netCDF header describes the contents of the file: typed, multi-dimensional variables and attributes on variables or the dataset itself.

Data for variables is stored in contiguous blocks, encoded in a portable binary format according to the variable's type.

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

Record Variables in netCDF

- Record variables are defined to have a single "unlimited" dimension
  - Convenient when a dimension size is unknown at time of variable creation
- Record variables are stored after all the other variables in an interleaved format
  - Using more than one in a file is likely to result in poor performance due to number of noncontiguous accesses


netCDF Header
Fixed-sized data
1st non-record variable
2nd non-record variable
nth non-record variable
Record Data
1st Record for 1st Record Var
1st Record for 2nd Record Var
1st Record for rth Record Var
2nd Record for 1st, 2nd,...,rth Record Variables in order
Records grow in the UNLIMITED dimension for 1,2,...,rth var

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

Pre-declaring I/O

- netCDF / Parallel-NetCDF: bimodal write interface
  - Define mode: “here are my dimensions, variables, and attributes”
  - Data mode: “now I’m writing out those values”
- Decoupling of description and execution shows up several places
  - MPI non-blocking communication
  - Parallel-NetCDF “write combining” (talk more in a few slides)
  - MPI datatypes to a collective routines (if you squint really hard)

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

“Hello world” Parallel-NetCDF style

```
NC_CHECK(ncmpi_create(MPI_COMM_WORLD, argv[1],
           NC_CLOBBER|NC_64BIT_OFFSET, MPI_INFO_NULL, &ncfile));

/* just one big string in this silly example */
NC_CHECK(ncmpi_def_dim(ncfile, "d1", varlen, &dimid));
NC_CHECK(ncmpi_def_var(ncfile, "v1", NC_CHAR, 1, &dimid, &varid));

NC_CHECK(ncmpi_enddef(ncfile));

NC_CHECK(ncmpi_put_vara_text_all(ncfile, varid, &offset, &len, buf));

NC_CHECK(ncmpi_close(ncfile));
```

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

Running on Polaris

```
#!/bin/bash -l
#PBS -A ATPESC2024
#PBS -l walltime=00:10:00
#PBS -l select=1
#PBS -l place=scatter
#PBS -l filesystems=home:eagle
#PBS -q debug
#PBS -N hello-io
#PBS -V

OUTPUT=/eagle/radix-io/${USER}/hello
mkdir -p ${OUTPUT}

NNODES=$(wc -l < $PBS_NODEFILE)
NRANKS_PER_NODE=32
NTOTRANKS=$(( NNODES * NRANKS_PER_NODE ))

cd $PBS_O_WORKDIR

mpiexec -n $NTOTRANKS -ppn $NRANKS_PER_NODE \
        ./hello-pnetcdf ${OUTPUT}/hello-pnetcdf.nc
```

Job submission script

```
% ncmpidump /eagle/radix-io/${USER}/hello/hello-pnetcdf.nc
netcdf hello-pnetcdf {
// file format: CDF-2 (large file)
dimensions:
        d1 = 790 ;
variables:
        char v1(d1) ;
data:

 v1 = "Hello from rank 0 of 32\n",
    "Hello from rank 1 of 32\n",
    "Hello from rank 2 of 32\n",
    […]
    "Hello from rank 27 of 32\n",
    "Hello from rank 28 of 32\n",
    "Hello from rank 29 of 32\n",
    "Hello from rank 30 of 32\n",
    "Hello from rank 31 of 32\n",
    "" ;
}
```

Output of “hello-pnetcdf”

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

HANDS-ON: writing with Parallel-NetCDF

- 2-D array in file, each rank writes 'YDIM' (1) rows
- Many details managed by pnetcdf library
  - MPI-IO File views
  - offsets
- Be mindful of define/data mode: call `ncmpi_enddef()`
- Library will take care of header i/o for you

1. Define two dimensions
   - `ncmpi_def_dim()`
2. Define one variable
   - `ncmpi_def_var()`
3. Collectively put variable
   - `ncmpi_put_vara_int_all()`
   - 'start' and 'count' arrays: each process selects different regions
4. Check your work with '`ncdump <filename>`'
   - Hey look at that: serial tool reading parallel-written data: interoperability at work

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

Solution fragments for Hands-on

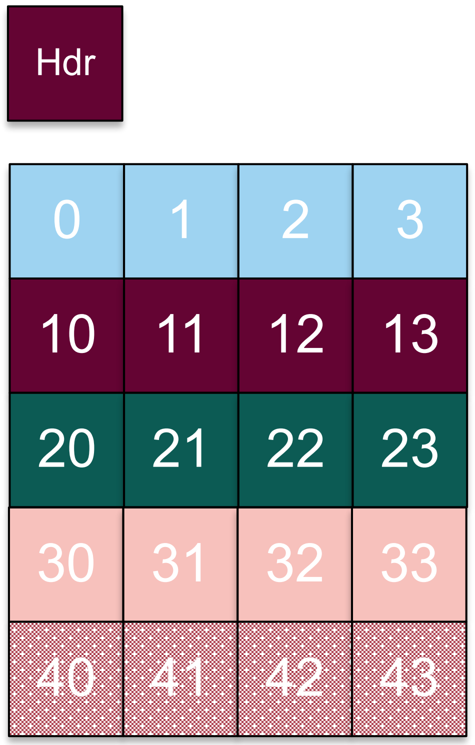
Defining dimension: give name, size; get ID

```
/* row-major ordering */
NC_CHECK(ncmpi_def_dim(ncfile, "rows", YDIM*nprocs, &(dims[0])) );
NC_CHECK(ncmpi_def_dim(ncfile, "elements", XDIM, &(dims[1])) );
```

Defining variable: give name, “rank” and dimensions (id); get ID
Attributes: can be placed globally, on variables, dimensions

```
NC_CHECK(ncmpi_def_var(ncfile, "array", NC_INT, NDIMS, dims,
                 &varid_array));

iterations=1;
NC_CHECK(ncmpi_put_att_int(ncfile, varid_array,
                 "iteration", NC_INT, 1, &iterations));
```



I/O: ‘start’ and ‘count’ give location, shape of subarray. ‘All’ means collective

```
start[0] = rank*YDIM; start[1] = 0;
count[0] = YDIM; count[1] = XDIM;
NC_CHECK(ncmpi_put_vara_int_all(ncfile, varid_array, start, count, values) );
```

Full example in visualization_io/mpiio-hdf5/hands-on/array

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

Inside PnetCDF Define Mode

- In define mode (collective)
  - Use `MPI_File_open` to create file at create time
  - Set hints as appropriate (more later)
  - Locally cache header information in memory
    - All changes are made to local copies at each process
- At ncmpi_enddef
  - Process 0 writes header with `MPI_File_write_at`
  - `MPI_Bcast` result to others
  - Everyone has header data in memory, understands placement of all variables
    - No need for any additional header I/O during data mode!

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

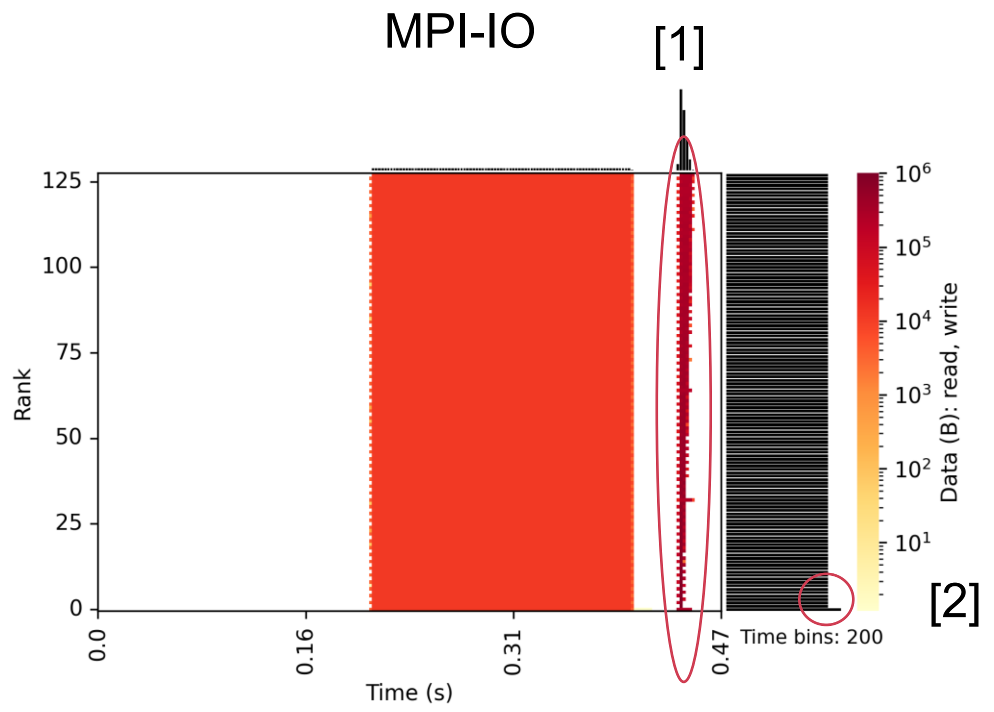
Inside PnetCDF Data Mode

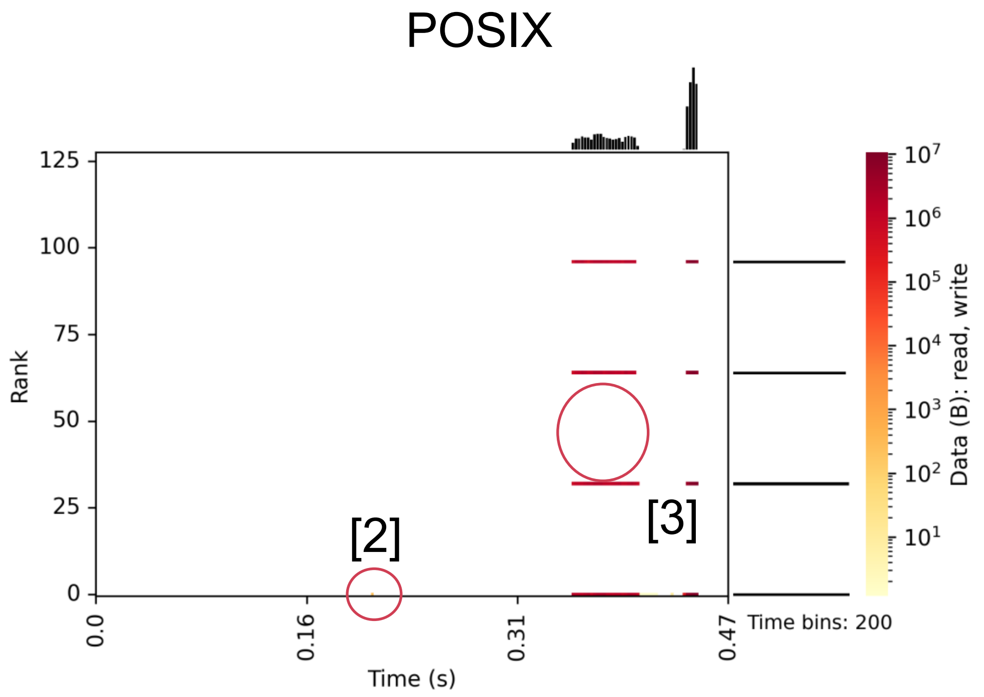
- Inside `ncmpi_put_vara_all` (once per variable)
  - Each process performs data conversion into internal buffer
  - Uses `MPI_File_set_view` to define file region
  - `MPI_File_write_all` collectively writes data
- At ncmpi_close
  - `MPI_File_close` ensures data is written to storage
- MPI-IO performs optimizations
  - Two-phase possibly applied when writing variables
- MPI-IO makes PFS calls
  - PFS client code communicates with servers and stores data

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

Inside PnetCDF: Darshan heatmap analysis

IOR writing Parallel-NetCDF (see visualization_io/mpiio-hdf5/hands-on/ior/polaris/ior-pnetcdf.sh)



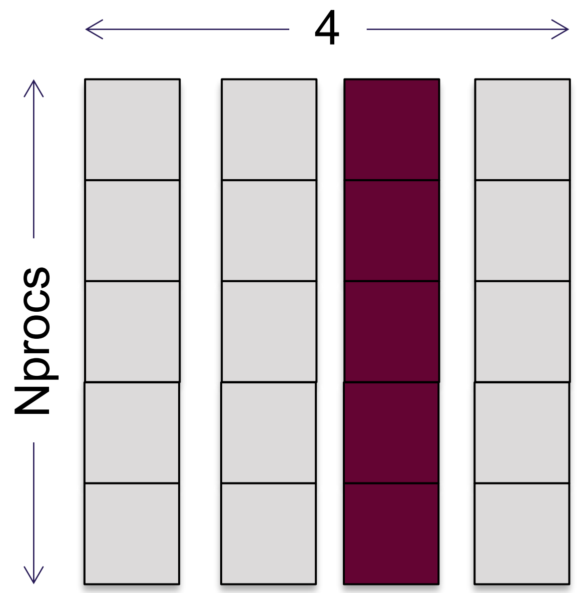


[1]: all processes call MPI write and read – re-reading going to be fast (cached)
[2]: one process wrote header -- small: just one pixel in POSIX
[3]: what you don't see – only "aggregators" actually do I/O

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

HANDS-ON: reading with pnetcdf

- Similar to MPI-IO reader: just read one row
- Operate on netcdf arrays, not MPI datatypes
- Shortcut: can rely on “convention”
  - One could know nothing about file as in previous slide
  - In our case we know there’s a variable called “array” (id of 0) and an attribute called “iteration”
- Routines you’ll need:
  - `ncmpi_inq_dim` to turn dimension id to dimension length
  - `ncmpi_get_att_int` to read “iteration” attribute
  - `ncmpi_get_vara_int_all` to read column of array



code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

Solution fragments: reading with pnetcdf

*Making **inq**uiry about variable, dimensions*

```
NC_CHECK(ncmpi_inq_var(ncfile, 0, varname, &vartype, &nr_dims,
     dim_ids,&nr_attrs));
NC_CHECK(ncmpi_inq_dim(ncfile, dim_ids[0], NULL, &(dim_lens[0])) );
NC_CHECK(ncmpi_inq_dim(ncfile, dim_ids[1], NULL, &(dim_lens[1])) );
```

The “Iteration” attribute

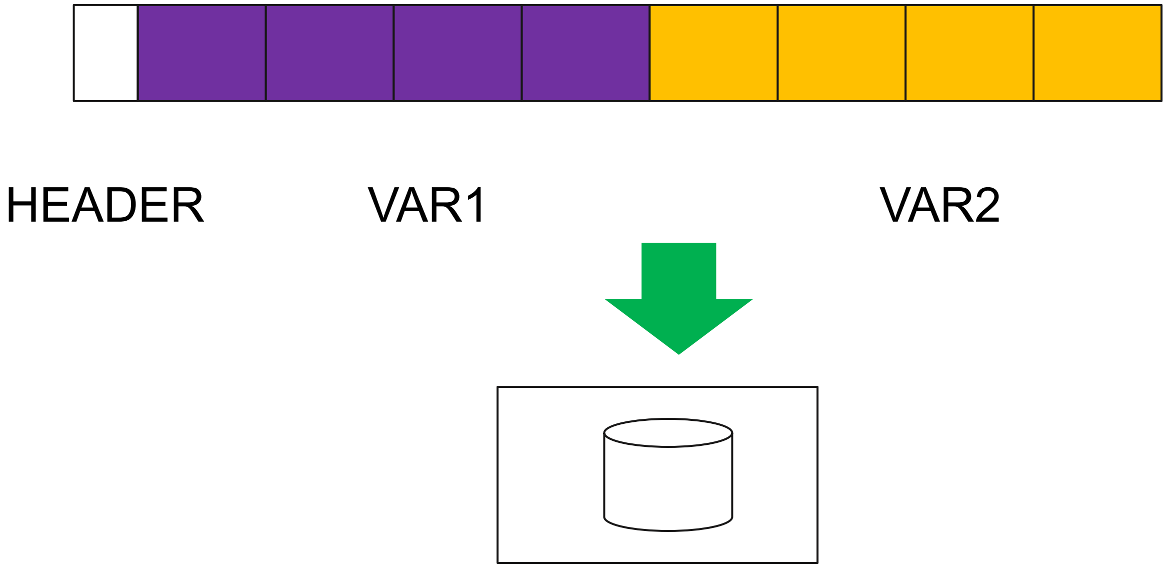
```
NC_CHECK(ncmpi_get_att_int(ncfile, 0, "iteration", &iterations));
```

No file views or datatypes: just a starting coordinate and size – everyone reads same slice in this case

```
count[0] = dim_lens[0]; count[1] = 1;
starts[0] = 0;      starts[1] = XDIM/2;
NC_CHECK(ncmpi_get_vara_int_all(ncfile, 0, starts, count, read_buf));
```

Parallel-NetCDF write-combining optimization

```
ncmpi_iput_vara(ncfile, varid1, &start, &count, &data,
        count, MPI_INT, &requests[0]);
ncmpi_iput_vara(ncfile, varid2, &start, &count, &data,
        count, MPI_INT, &requests[1]);
ncmpi_wait_all(ncfile, 2, requests, statuses);
```

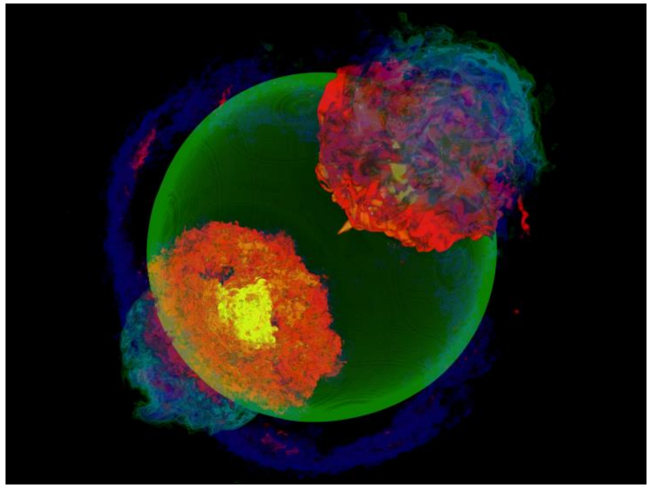


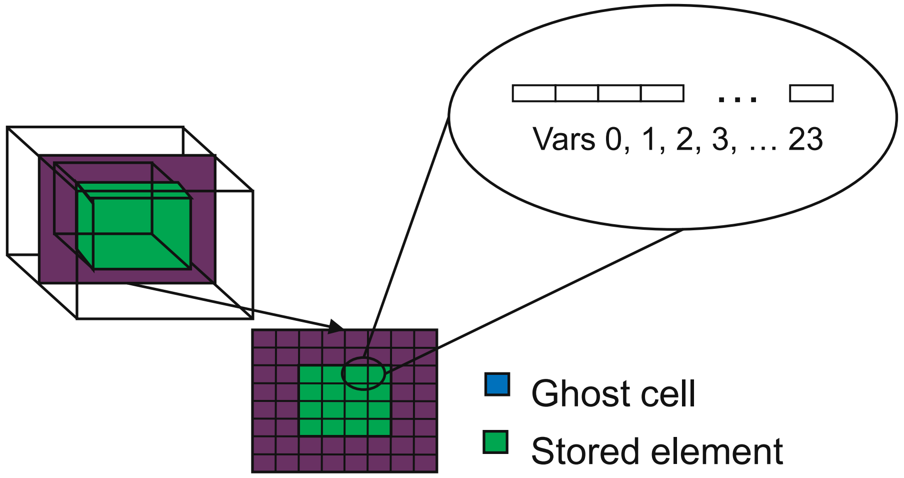
- netCDF variables laid out contiguously
- Applications typically store data in separate variables
  - temperature(lat, long, elevation)
  - Velocity_x(x, y, z, timestep)
- Operations posted independently, completed collectively
  - Defer, coalesce synchronization
  - Increase average request size

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

Example: FLASH Astrophysics

- FLASH is an astrophysics code for studying events such as supernovae
  - Adaptive-mesh hydrodynamics
  - Scales to 1000s of processors
  - MPI for communication
- Frequently checkpoints:
  - Large blocks of typed variables from all processes
  - Portable format
  - Canonical ordering (different than in memory)
  - Skipping ghost cells

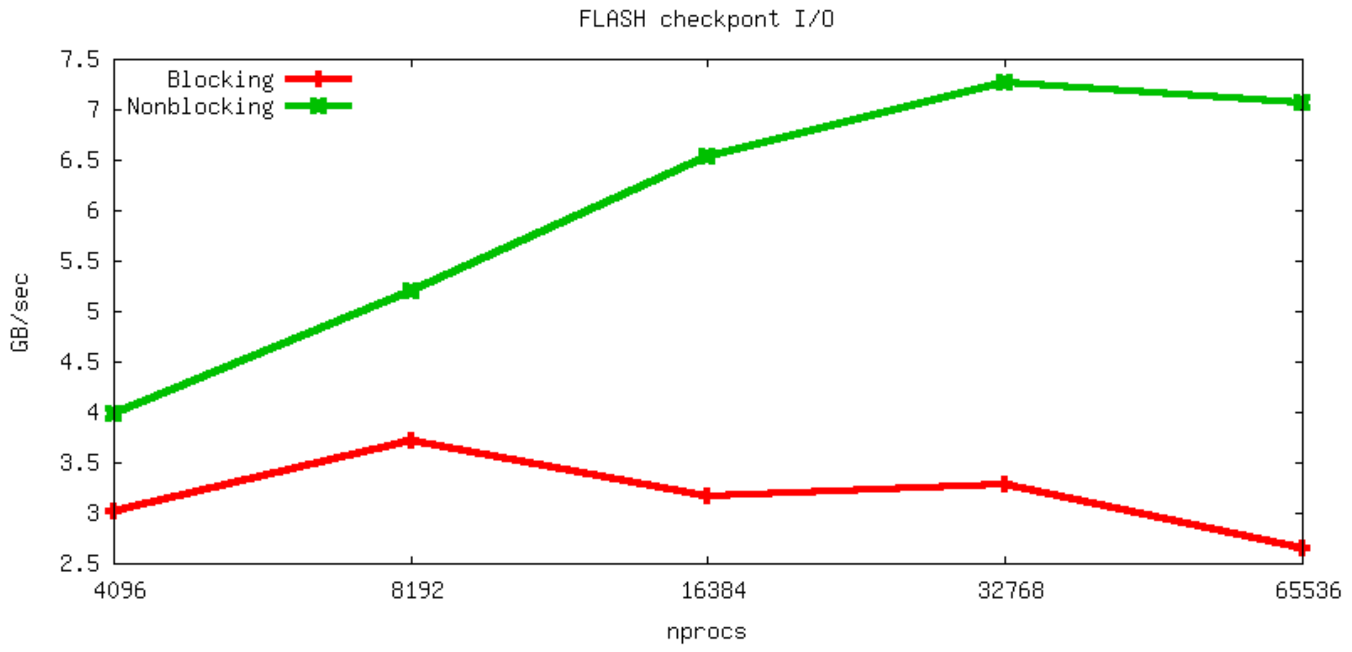




code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

FLASH Astrophysics and the write-combining optimization

- FLASH writes one variable at a time
- Could combine all 4D variables (temperature, pressure, etc) into one 5D variable
  - Altered file format (conventions) requires updating entire analysis toolchain
- Write-combining provides improved performance with same file conventions
  - Larger requests, less synchronization.



code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

HANDS-ON: pnetcdf write-combining

1. Define a second variable, changing only the name
2. Write this second variable to the netcdf file
3. Convert to the non-blocking interface (`ncmpi_iput_vara_int`)
   - not collective – “collectiveness” happens in `ncmpi_wait_all`
   - takes an additional ‘request’ argument
4. Wait (collectively) for completion

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

Solution fragments for write-combining

Defining a second variable

```
NC_CHECK(ncmpi_def_var(ncfile, "array", NC_INT, NDIMS, dims,
                    &varid_array));
NC_CHECK(ncmpi_def_var(ncfile, "other array", NC_INT, NDIMS, dims,
                     &varid_other));
```

The non-blocking interface: looks a lot like MPI

```
NC_CHECK(ncmpi_iput_vara_int(ncfile, varid_array, start, count,
                    values, &(reqs[0]) ) );
NC_CHECK(ncmpi_iput_vara_int(ncfile, varid_other, start, count,
                    values, &(reqs[1]) ) );
```

Waiting for I/O to complete

```
/* all the I/O actually happens here */
NC_CHECK(ncmpi_wait_all(ncfile, 2, reqs, status));
```

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

Hands-on continued

- Look at the darshan output. Compare to darshan output for single-variable writing or reading
  - Results on polaris surprised me: vendor might know something I don't
    - Maybe some kind of small-io optimization?

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop

PnetCDF Wrap-Up

- PnetCDF gives us
  - Simple, portable, self-describing container for data
  - Collective I/O
  - Data structures closely mapping to the variables described
- If PnetCDF meets application needs, it is likely to give good performance
  - Type conversion to portable format does add overhead
- Some limits on (old, common CDF-2) file format:
  - Fixed-size variable: < 4 GiB
  - Per-record size of record variable: < 4 GiB
  - 2^{32} -1 records
  - Contributed extended file format to relax these limits (CDF-5, released in pnetcdf-1.1.0, November 2009, integrated in Unidata NetCDF-4.4)

code etc: https://github.com/argonne-lcf/ALCF_Hands_on_HPC_Workshop