ARGONNE ATPESC2025 EXTREME-SCALE COMPUTING

Introduction to MPI-IO

Rob Latham

Research Software Developer, Argonne National Laboratory

Hands on materials

- Code for available on our github site: https://github.com/radix-io/hands-on
  - This session:
    - “hello-io” basics
    - Simple array I/O
    - IOR recipes
  - Other sessions today:
    - Darshan
    - HDF5
  - “Bonus Content:”
    - Game of Life I/O
    - Sparse Matrix I/O
- Work through examples when you can. We’re going to do this “cooking show” style…

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
  - I'll be talking a lot about the ROMIO implementation

“Hello World” MPI-IO style: contiguous

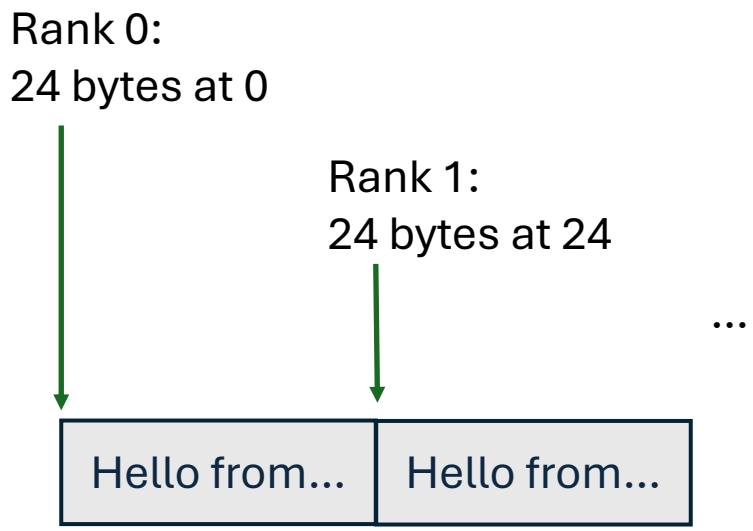
```
/* an "Info object":  these store key-value strings for tuning the
 * underlying MPI-IO implementation */
MPI_Info_create(&info);

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));
```



“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));
```

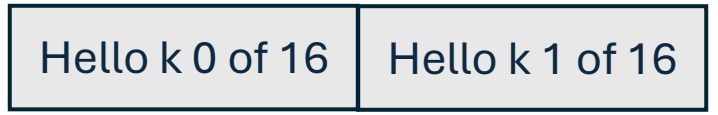
Hello from rank 1 of 16

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

Rank 0:
6+10 bytes at 0

Rank 1:
6+10 bytes at 16

…

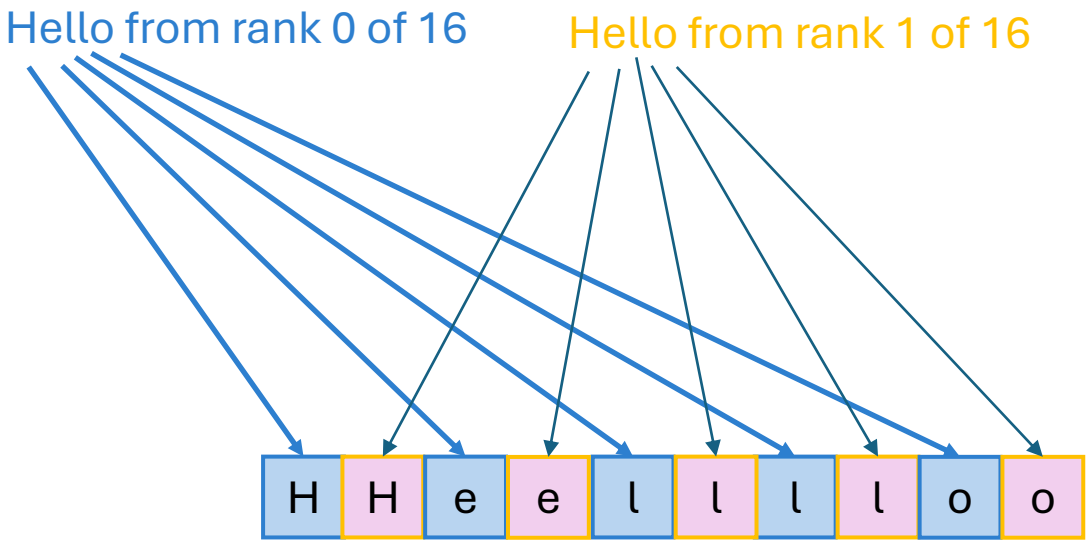


“Hello World” MPI-IO style: non-contiguous in file

```
/* 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.

RUNNING

- Submit to the ‘ATPESC2025’ queue (aurora)
- I’ve provided a ‘hello-aurora.sh’ shell script
  - `qsub hello-aurora.sh`
- We’ll use the DAOS file system
  - ALCF has made a “ATPESC2025_0” pool on the “daos_perf” service
  - Job script will create your own container inside that pool
    - `daos container create --type POSIX $DAOS_POOL $DAOS_CONT`
  - DAOS is always running, but we have to “launch” the file system view of it
    - `launch-dfuse_perf.sh ${DAOS_POOL}:${DAOS_CONT}`
    - Now shell tools can operate on `/tmp/${DAOS_POOL}/${DAOS_CONT}`
- There’s a special “cpu binding” to place processes such that they use all 8 Aurora network cards.

Output on Aurora

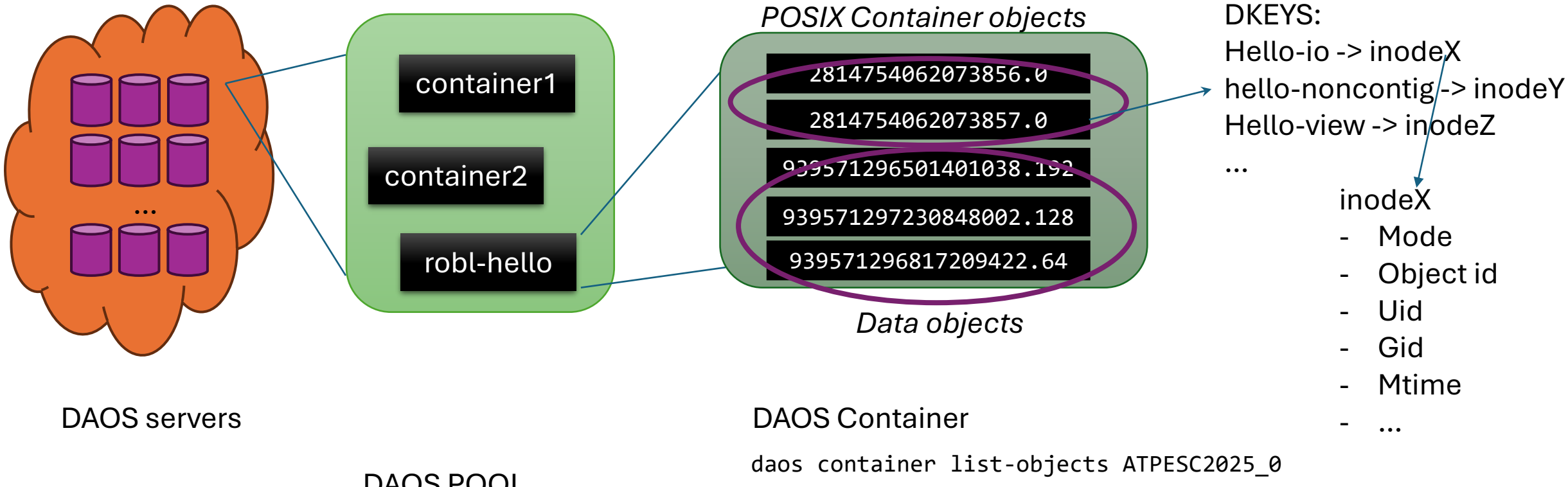
```
==== contiguous in memory and file
cat /tmp/ATPESC2025_0/robl-hello/hello.out
Hello from rank 0 of 16
Hello from rank 1 of 16
…
Hello from rank 15 of 16


==== noncontiguous in memory
cat /tmp/ATPESC2025_0/robl-hello/hello-
noncontig.out
Hello k 0 of 16
Hello k 1 of 16
…
Hello  15 of 16
```

```
==== noncontiguous in file
cat /tmp/ATPESC2025_0/robl-hello/hello-
view.out
HHHHHHHHHHHHHHHeeeeeeeeeeeeeeeellllllll
lllllllllllllllllllllllloooooooooooooo
oo
fffffffffffffffrrrrrrrrrrrrrrrooooooo
ooooooooommmmmmmmmmmmmmmm
rrrrrrrrrrrrrraaaaaaaaaaaaaannnnnnn
nnnnnnnnnkkkkkkkkkkkkkkk
11111012345678901234 5
oooooooooooooooffffffffffffffff
11111111111111166666666666666666
```

Output of our hello programs

Under the hood: DAOS (essentially)



DAOS servers

DAOS POOL

```
daos pool list-containers ATPESC2025_0
```

DAOS Container

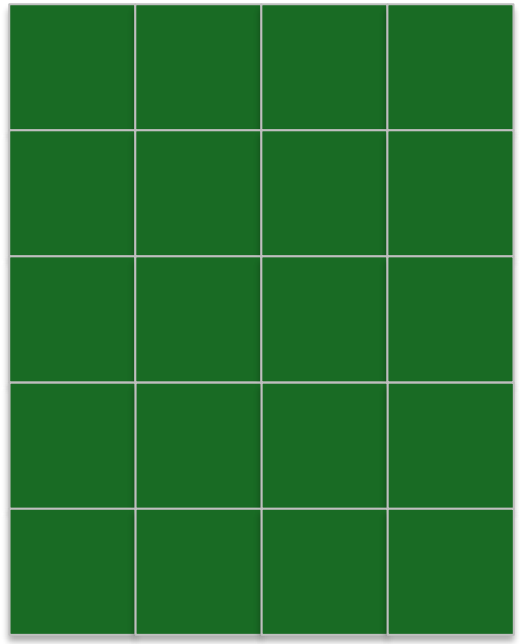
```
daos container list-objects ATPESC2025_0
```

Key takeaways

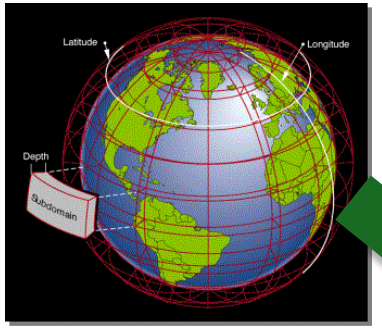
- Simple example but still captures important concepts
  - Info objects: tuning parameters:
    - enable/disable optimizations
    - Adjust buffer sizes
    - Select alternate strategies
  - Data placement in file specified by user
    - “shared file pointer” possible but not optimized
  - Collective vs independent I/O
  - Error checking!!!
- A lot of complexity of DAOS abstracted away under “DAOS file system” and ROMIO’s DAOS driver
  - DAOS optimizations like “resolve on one, broadcast to all”
  - Portable to any supported file system: could write to lustre simply by changing the path

Operating on Arrays

- Arrays show up in many scientific applications
  - Matrix operations
  - Particle maps
  - Regions of space
  - Time series
  - Images
- Probably your real application more complicated but an array or two (or more) is in there somewhere, I'd wager.



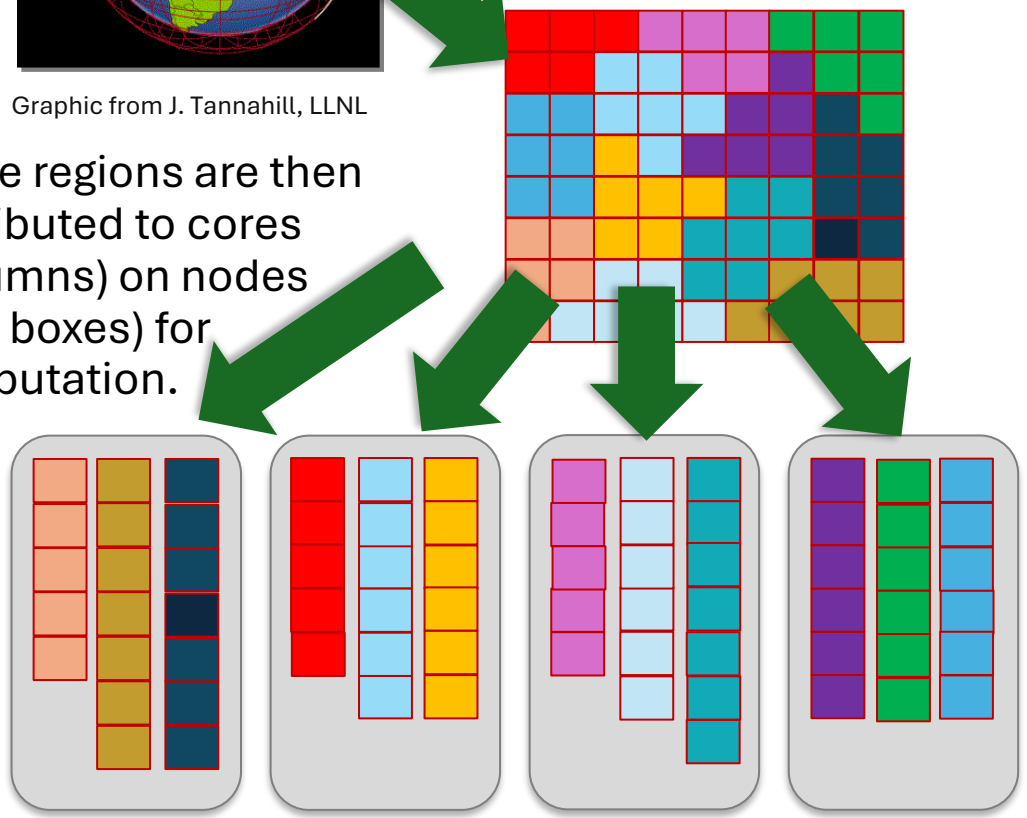
Decomposition



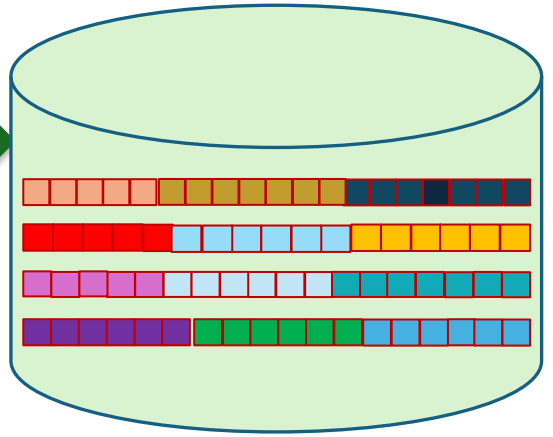
Graphic from J. Tannahill, LLNL

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

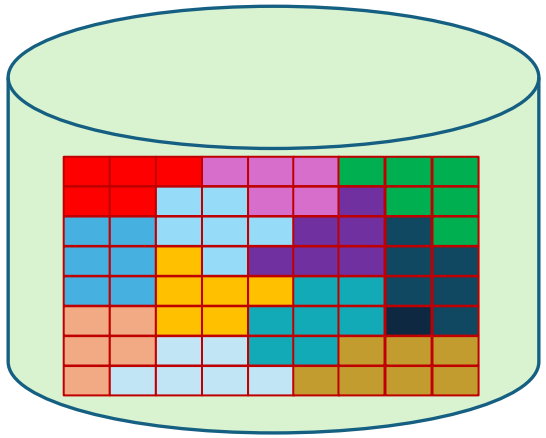
These regions are then distributed to cores (columns) on nodes (grey boxes) for computation.



or



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.

Scientific I/O constraints

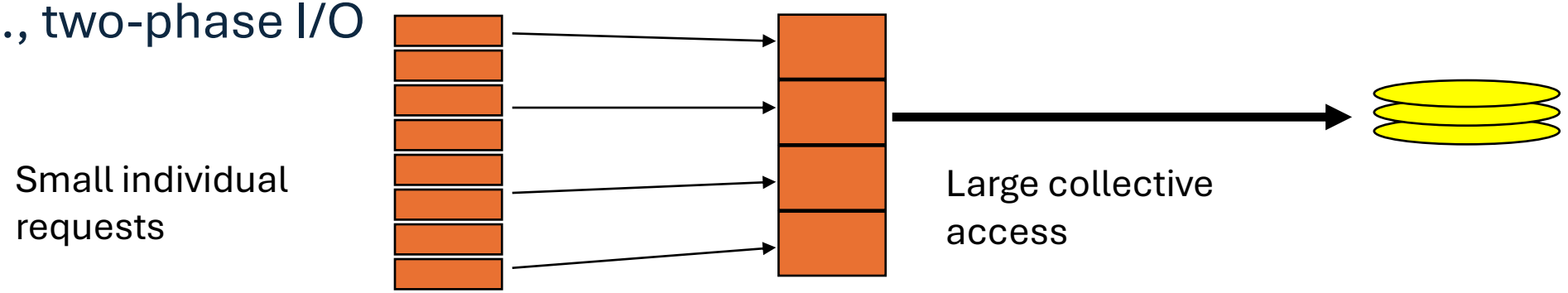
- Defensive I/O:
  - Guard against node failures or program errors with checkpointing
  - Application saves its own state
  - With a bit of extra effort, can be a portable, canonical representation
  - Ideally Independent of number of processes
- Restarting:
  - Canonical representation aids restarting with a different number of processes
- Data analysis
  - Who will consume this data?
- AI and Machine Learning
  - “why is my [random small read] workload so slow?”

Defining a Checkpoint

- Need enough to restart
  - Header information
    - Size of problem (e.g. matrix dimensions)
    - Description of environment (e.g. input parameters)
  - Program state
    - Should represent the global (canonical) view of the data
- Ideally stored in a convenient container
  - Single “thing” (file, object, keyval store…)
- If all processes checkpoint at once, naturally a parallel, collective operation

Collective I/O

- A critical optimization in parallel I/O
- All processes (in the communicator) must call the collective I/O function
- Allows communication of “big picture” to file system
  - Framework for I/O transformations/optimizations at the MPI-IO layer
  - e.g., two-phase I/O

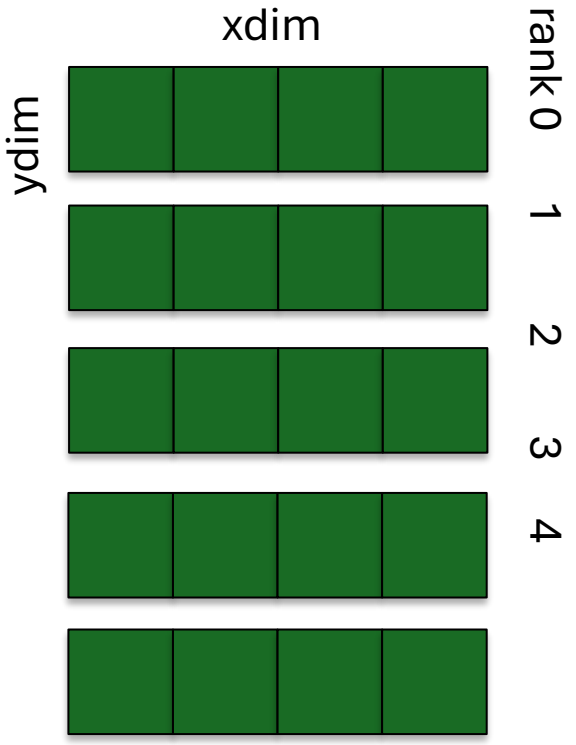


Collective MPI I/O Functions

- Not going to go through the MPI-IO API in excruciating detail
  - Happy to discuss in slack, chat, email
- **MPI_File_write_at_all**, etc.
  - **_all** indicates that all processes in the group specified by the communicator passed to MPI_File_open will call this function
  - **_at** indicates that the position in the file is specified as part of the call; this provides thread-safety and clearer code than using a separate "seek" call
- Each process specifies only its own access information
  - the argument list is the same as for the non-collective functions
  - OK to participate with zero data
    - All processes must call a collective
    - Process providing zero data might participate behind the scenes anyway

HANDS-ON: writing with MPI-IO

- Write our toy checkpoint to a file in parallel (`array/array-mpiio-write.c`)
- Use `MPI_File_open` instead of open
- Only one process needs to write `header`
  - Independent `MPI_File_write`
  - Could combine, but header I/O small and checkpoint (typically) vastly larger
- Every process sets a "file view"
  - Need to skip over header – file view has an "offset" field just for this case
  - The "file view" here is not complicated: we are operating on integers, not bytes:
    - `MPI_File_set_view(fh, sizeof(header), MPI_INT, MPI_INT, "native", info));`
- Each process writes one slice/row of array
  - MPI_File_write_at_all
  - Offset: "rank*XDIM*YDIM" – no 'sizeof': specified ints in file view
  - "(bufer, count, datatype)" tuple: (`values, XDIM*YDIM, MPI_INT`)



Solution fragments

Header I/O from rank 0:

```
if (rank == 0) {
    MPI_CHECK(MPI_File_write(fh,
        &header, sizeof(header), MPI_BYTE,
        MPI_STATUS_IGNORE) );
}
```

Collective I/O from all ranks

```
MPI_File_write_at_all(fh, rank*XDIM*YDIM,
        values, XDIM*YDIM, MPI_INT,
        MPI_STATUS_IGNORE));
```

Hands-on continued: Darshan

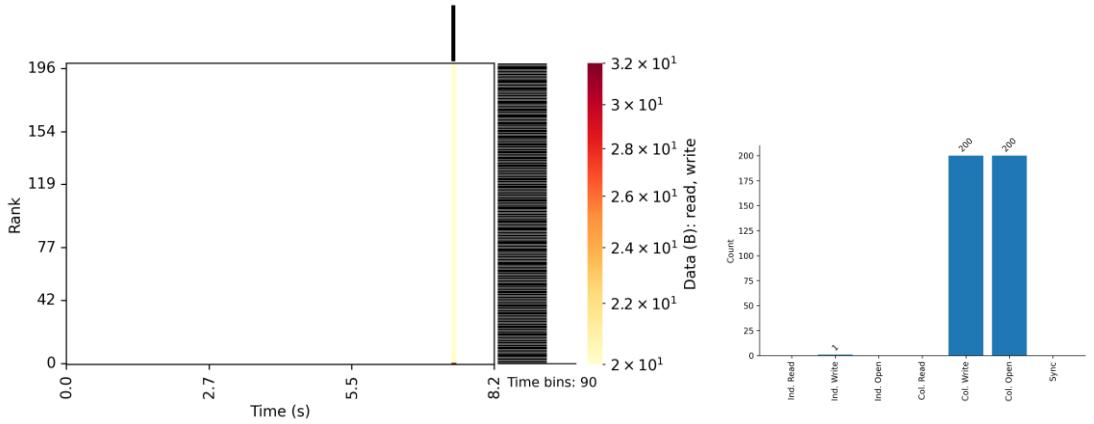
- Let's use Darshan
  - Find Darshan log file, but don't generate report right away
- What do you think the report will say?
- OK, now generate the report. Were you surprised?
  - Counts of POSIX calls (POSIX_WRITES) vs MPI-IO calls (MPIIO_COLL_WRITES)
  - Sizes of POSIX calls vs sizes of MPI-IO calls
- MPI-IO "info" hints to guide optimizations

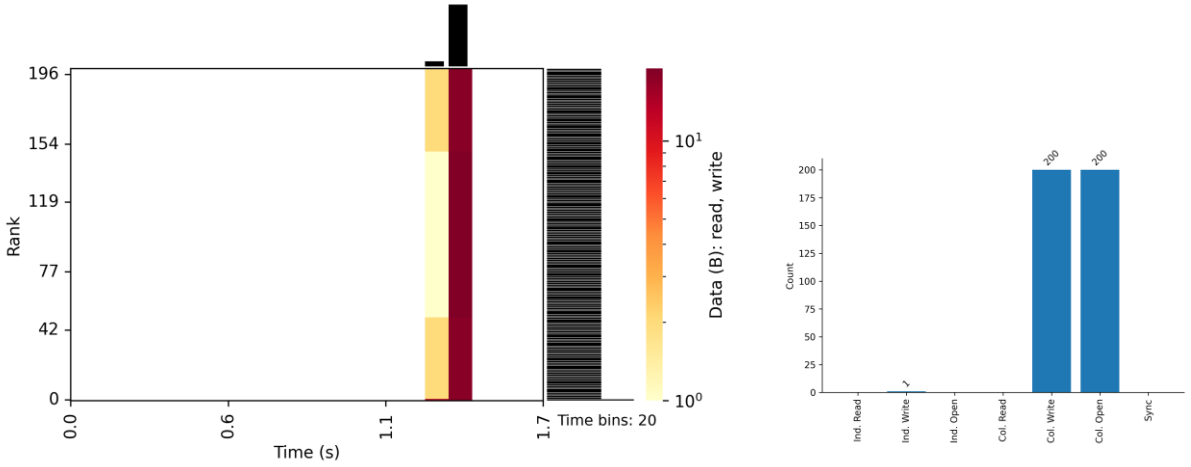
Hands-on continued: Darshan

Default (independent)

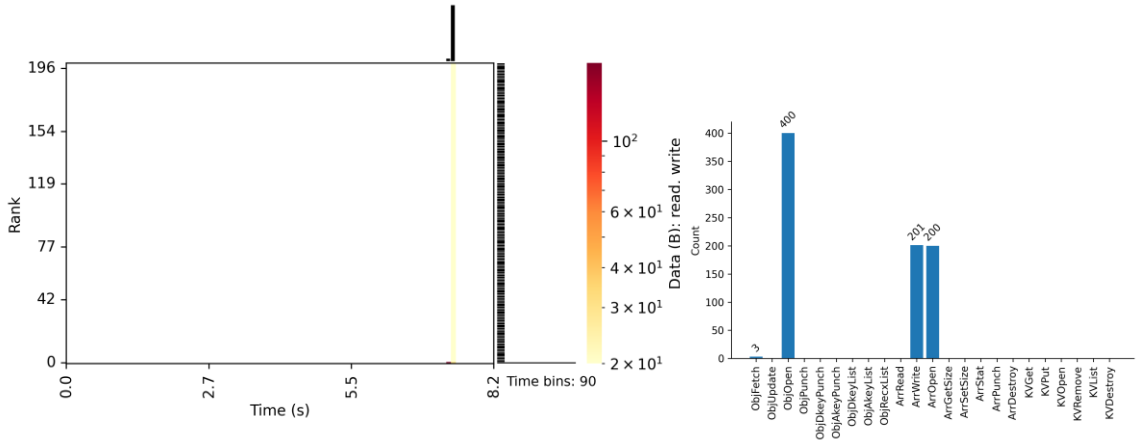
Hinted (collective)

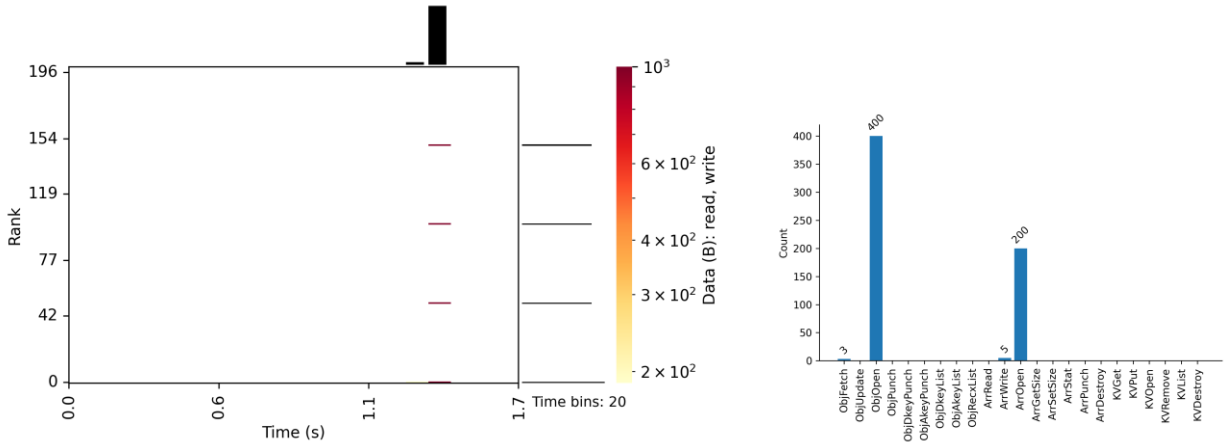
MPI-IO





DAOS



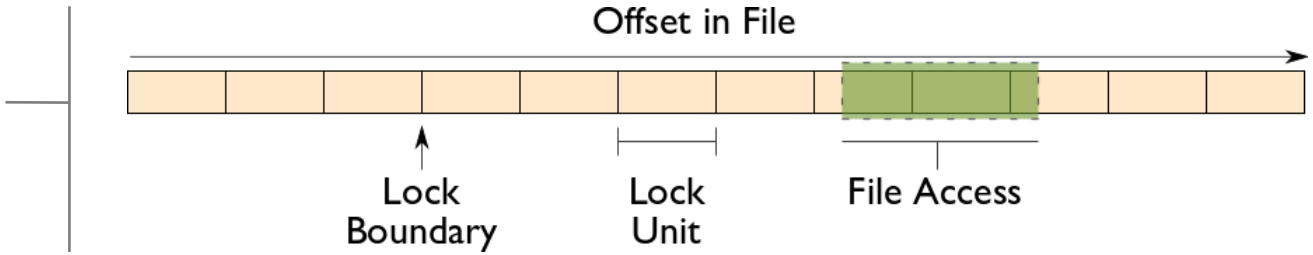


Operation counts

Managing Concurrent Access

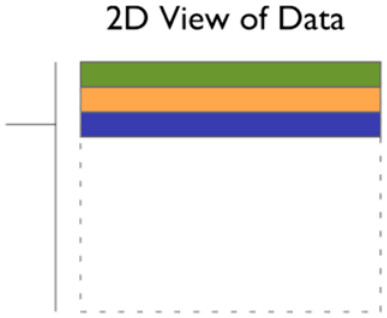
- **Files are treated like global shared memory regions. Locks are used to manage concurrent access:**
- Files are broken up into lock units
  - Unit boundaries are dictated by the storage system, regardless of access pattern
- Clients obtain locks on units that they will access before I/O occurs
- Enables 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 others desire access

If an access touches any data in a lock unit, the lock for that region must be obtained before access occurs.



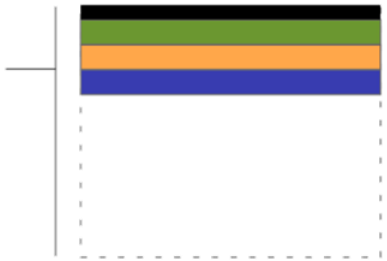
Implications of Locking in Concurrent Access

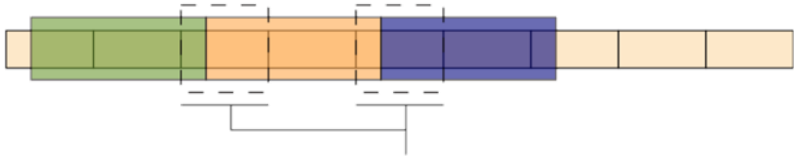
The left diagram shows a row-block distribution of data for three processes. On the right we see how these accesses map onto locking units in the file.



When accesses are to large contiguous regions, and aligned with lock boundaries, locking overhead is minimal.

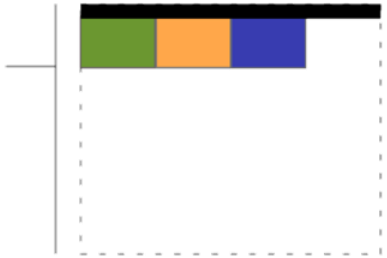
In this example a header (black) has been prepended to the data. If the header is not aligned with lock boundaries, false sharing will occur.

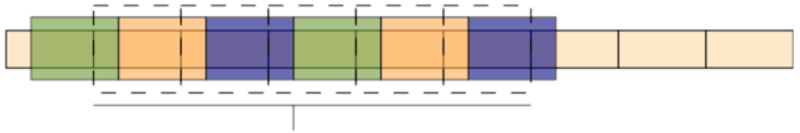




These two regions exhibit *false sharing*: no bytes are accessed by both processes, but because each block is accessed by more than one process, there is contention for locks.

In this example, processes exhibit a block-block access pattern (e.g. accessing a subarray). This results in many interleaved accesses in the file.

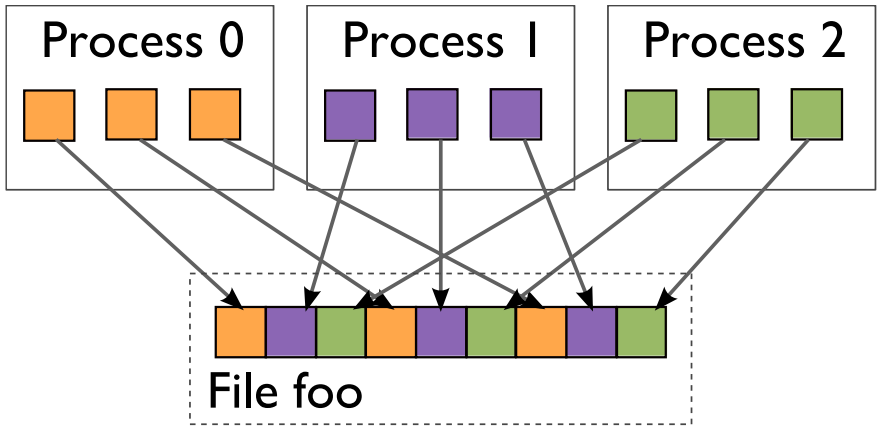




When a block distribution is used, sub-rows cause a higher degree of false sharing, especially if data is not aligned with lock boundaries.

I/O Transformations

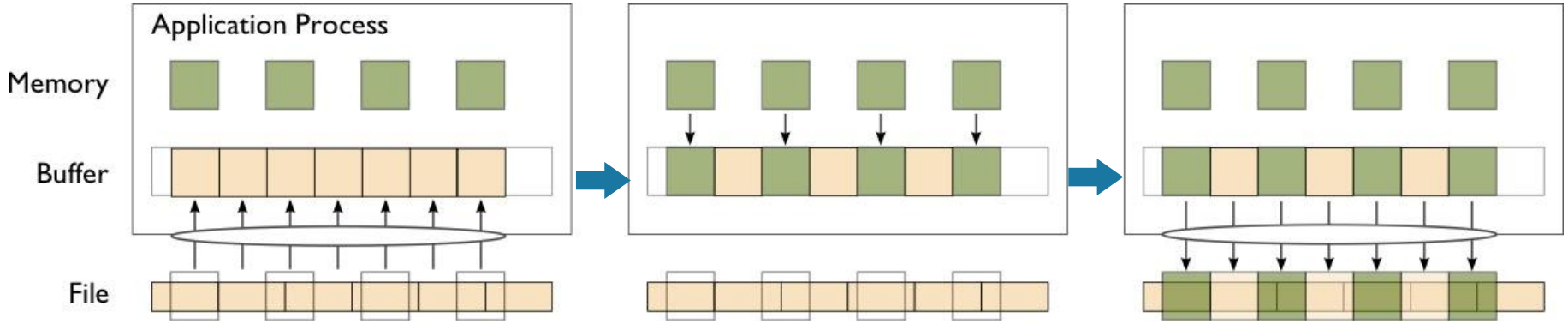
- **Software between the application and the file system performs transformations, primarily to improve performance.**
  - Goals of transformations:
    - Reduce number of operations to PFS (avoiding latency)
    - Avoid lock contention (increasing level of concurrency)
    - Hide number of clients (more on this later)
  - With "transparent" transformations, data ends up in the same locations in the file as it would have been normally
    - i.e., the file system is still aware of the actual data organization
  - I/O libraries do these for you already



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

Reducing Number of Operations

- **Because most operations go over multiple networks, 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)
- Somewhat counter-intuitive: do extra I/O to avoid contention



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

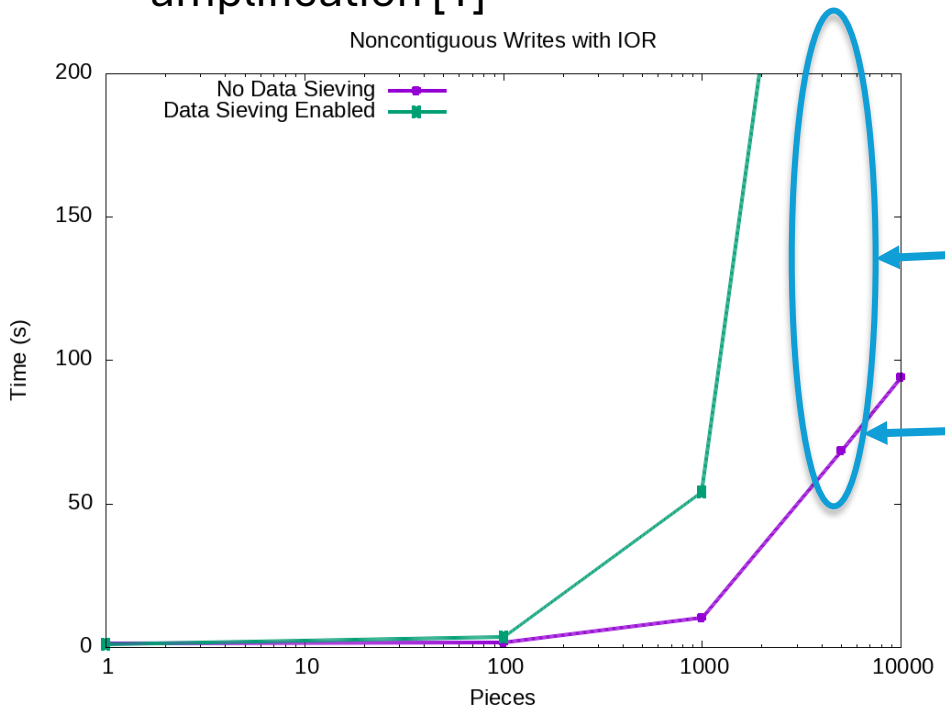
Step 2: Elements to be written to file are replaced in intermediate buffer.

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

Data Sieving in Practice (Polaris, Lustre)

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]



5 000 pieces
2 000 bytes each

[1]

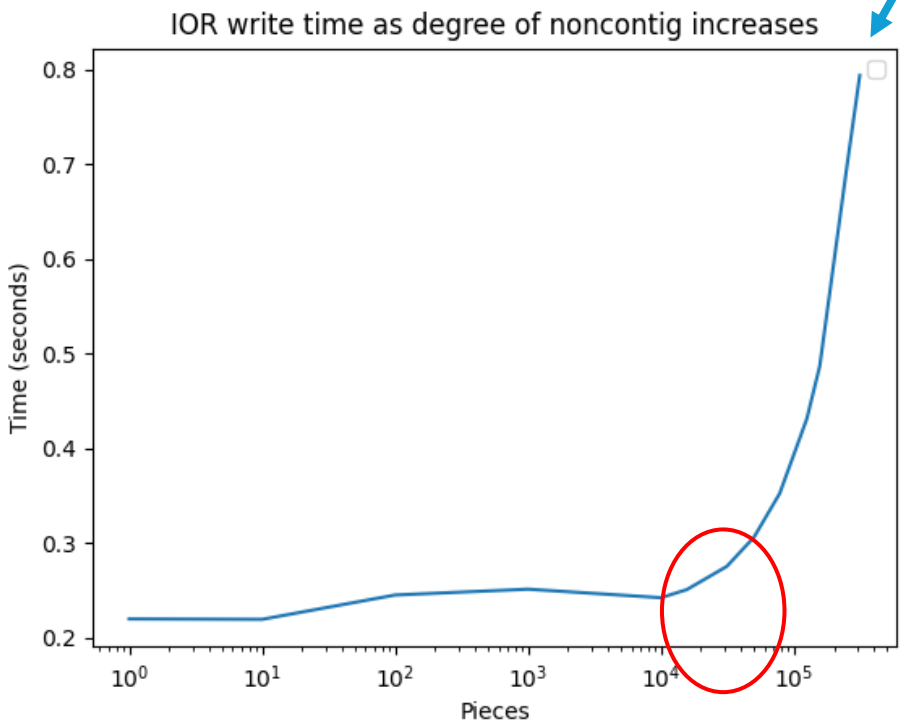
	Naiive	Data Sieving
MPI-IO writes	960	960
MPI-IO Reads	0	0
Posix Writes	4 800 000	4 800 000
Posix Reads	0	4 800 784
MPI-IO bytes written	8.9 GiB	8.9 GiB
MPI-IO bytes read	0	0
Posix bytes read	0	2334 GiB
Posix bytes written	8.9 GiB	2343 GiB
Runtime (sec)	68.8	404.2

Selected Darshan statistics for 5000 segments

Data Sieving Alternative: scatter-gather (list-io)

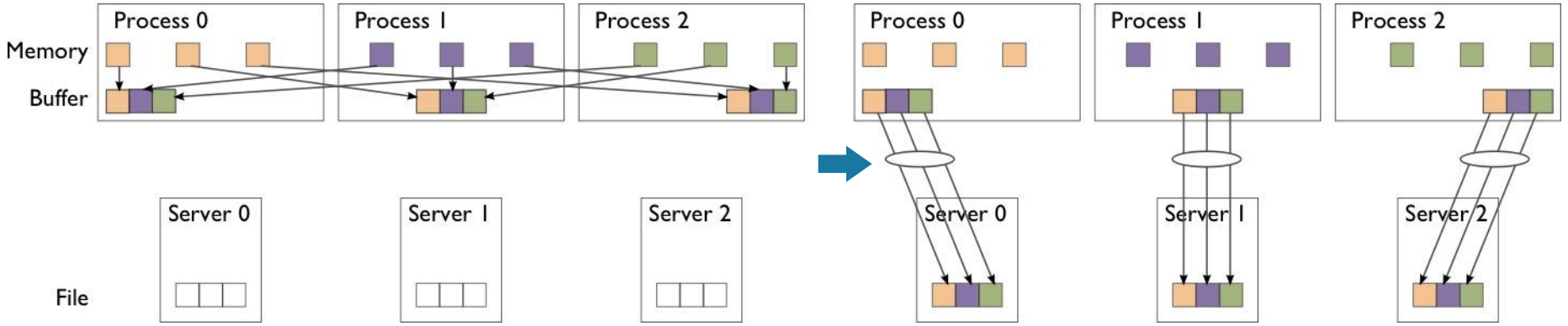
- Same IOR experiment, this time on Aurora's DAOS
- DAOS provides an alternative approach: describe the entire I/O request with a scatter-gather list (d_sg_list_t):
  - `int dfs_write(dfs_t *dfs, dfs_obj_t *obj, d_sg_list_t *sgl, daos_off_t off, daos_event_t *ev);`
- ROMIO driver does this for you
- Curve starts to bend at 50 000 elements:
  - note y axis – still under one second
  - We think due to server side processing of these very long lists
  - Some new optimizations in the pipeline as well

312 500 pieces
32 bytes each



Avoiding Lock Contention

- **We can reorder data among processes to avoid lock contention.** *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)



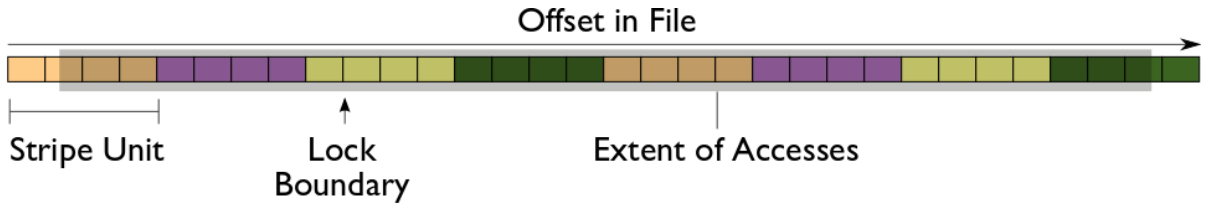
Phase 1: Data are exchanged between processes based on organization of data in file.

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

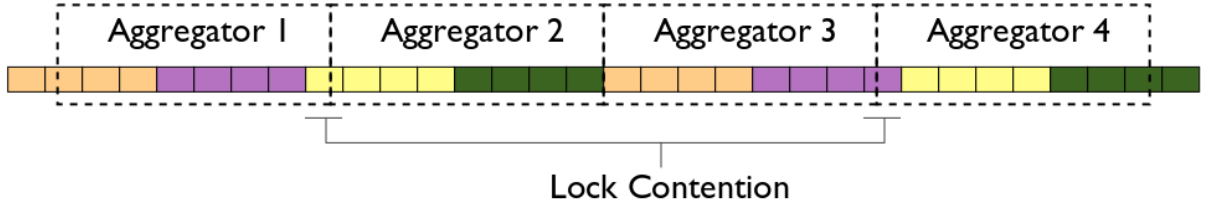
Two-Phase I/O Algorithms

(or, You don't want to do this yourself…)

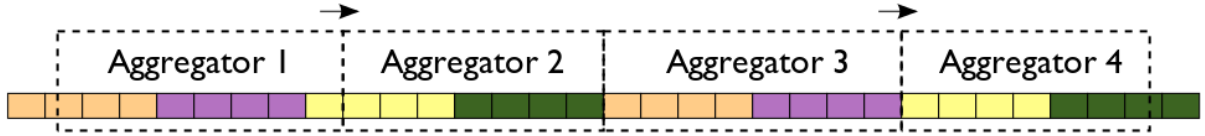
Imagine a collective I/O access using four aggregators to a file striped over four file servers (indicated by colors):



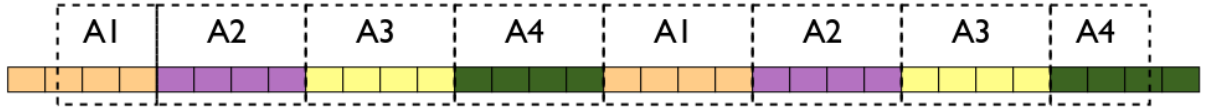
One approach is to evenly divide the region accessed across aggregators.



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).



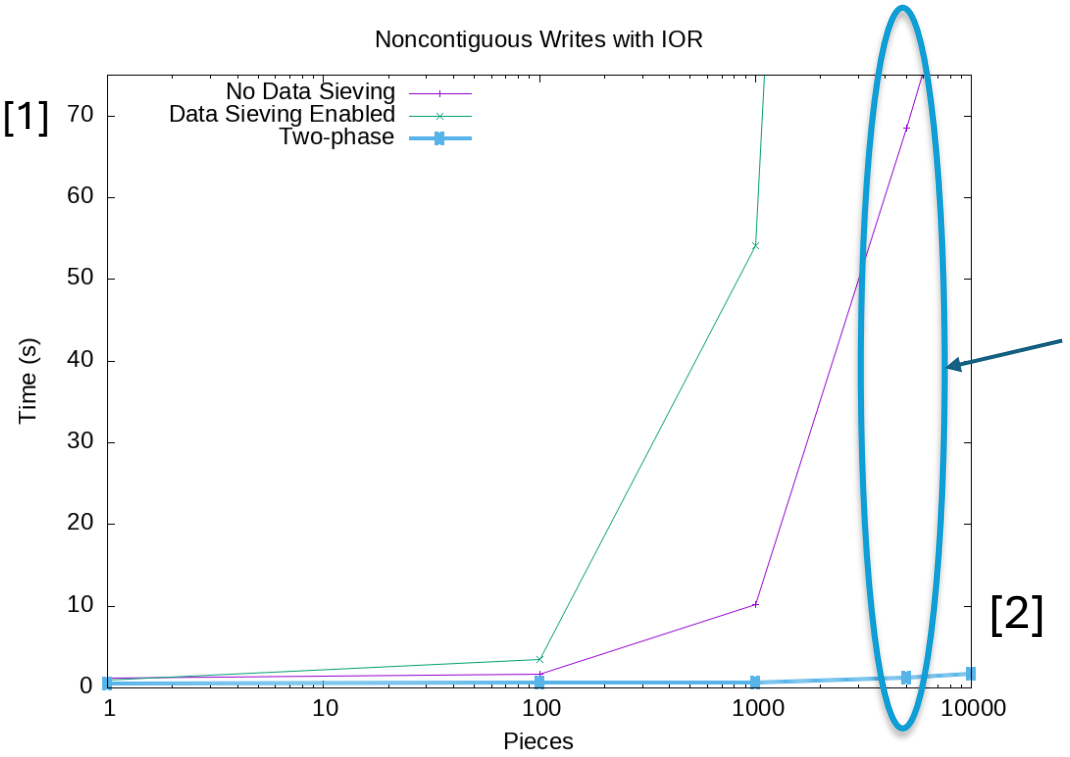
For more information, see W.K. Liao and A. Choudhary, "Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols," SC2008, November, 2008.

Today's systems also choose aggregators that are "best" for storage

Two-phase I/O in Practice (Polaris, Lustre)

- 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.

[1]

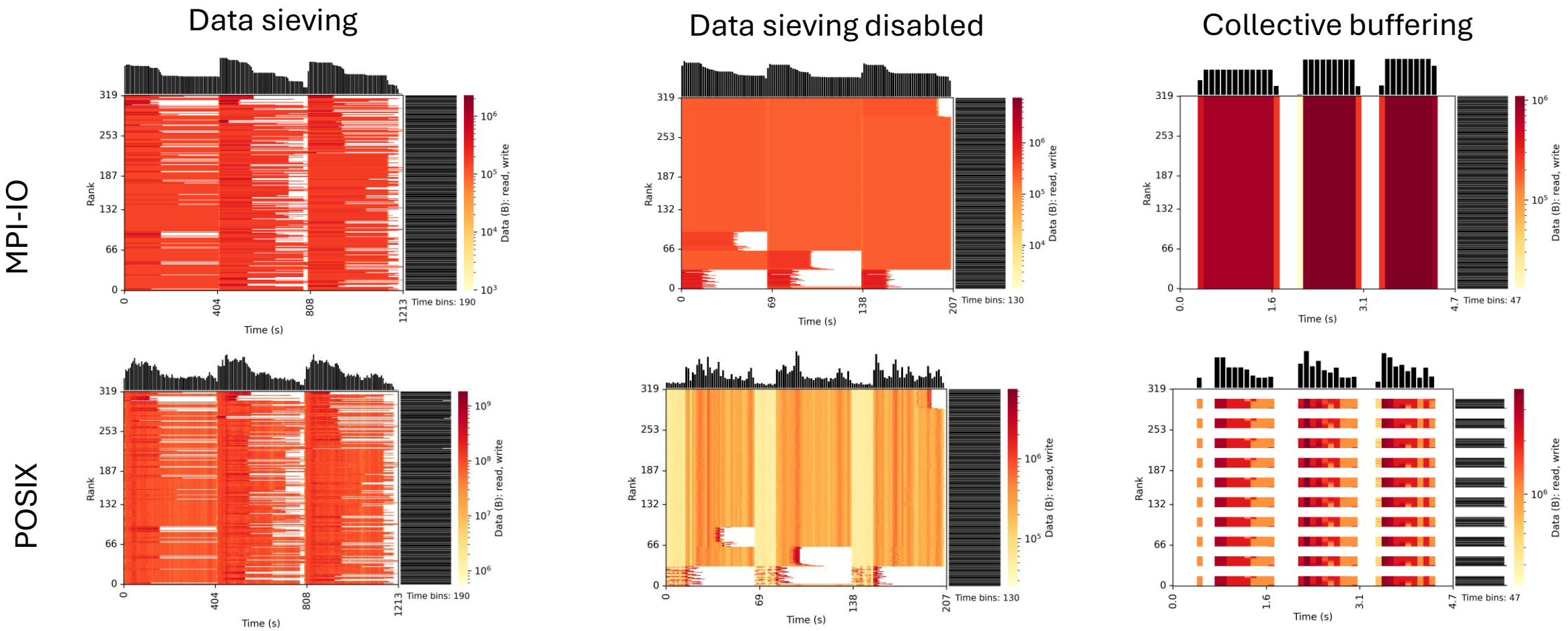


[2]

	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

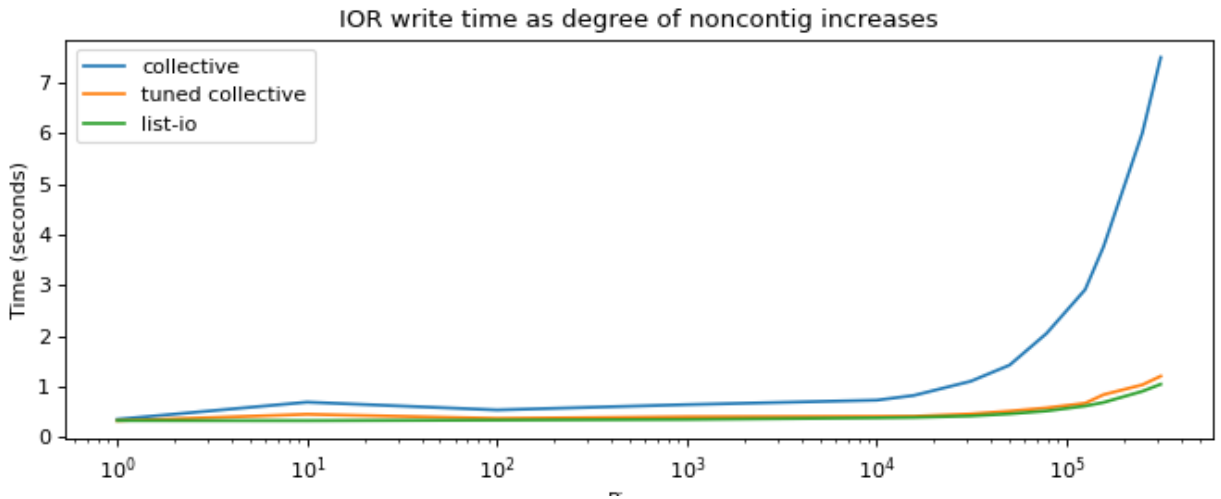
More investigation: Darshan heatmaps (Polaris, Lustre)



Effect of ROMIO optimizations on IOR benchmark: 5000 non-contiguous segments, three iterations. Note the x axis

DAOS: Collective I/O vs scatter-gather I/O

- Same IOR experiment but on Aurora this time
  - 2 nodes, 96 processes per node
- List-IO lets us avoid two sources of overhead
  - "rounds" of I/O – no buffering at intermediate aggregator
  - No network exchange of data
- tuned: – asking for more aggregators per node lets us use all 8 network cards
- Since List-IO does not aggregate, could be a problem at larger scale (evaluation "on my list")
  - Obviously, combining both approaches would be great (that's "on my list" now too…)

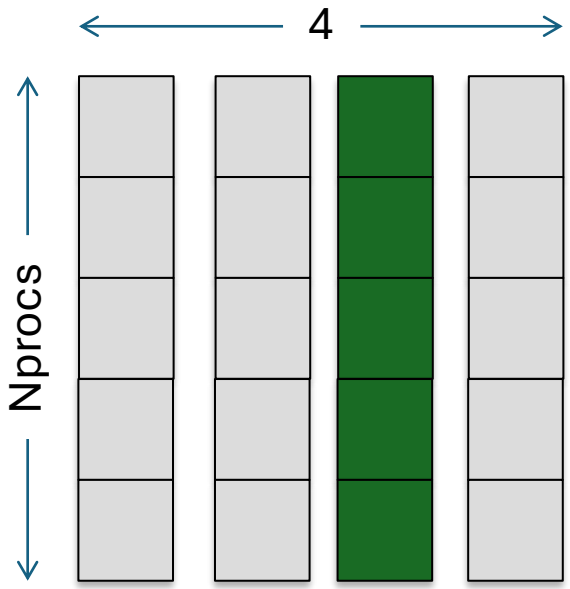


	Two-phase	Tuned Two-phase	List-IO
MPI-IO writes	1152	1152	1152
MPI-IO Reads	0	0	0
DAOS Writes	696	768	1152
DAOS Reads	0	0	0
MPI-IO bytes written	10.7 GiB	10.7 GiB	10.7 GiB
MPI-IO bytes read	0	0	0
DAOS bytes read	0	0	0
DAOS bytes written	10.7 GiB	10.7 GiB	10.7 GiB
Max MPI-IO write time	1.335 sec	0.35 sec	0.22 sec
Max DAOS write time	3.10 msec	3.485 msec	0.22 sec

Selected Darshan statistics, 5000 segments

HANDS-ON: reading with MPI-IO

- Slightly different: all processes read one column
  - For simplicity, same row
- File view will be more complicated, use MPI “Subarray” datatype
- In C, array access is described in “row-major”
  - `array_size[0] = 5; array_size[1] = 4;`
- File view uses derived ‘subarray’, not built-in MPI_INT
- Location in file given with subarray type; no offset in `MPI_File_read_all`
  - Still provide a “buffer, count, datatype” tuple for memory layout



Solution fragments

Type creation

```
/* In C-order the arrays are row-major:
 *
 * |-----|
 * |-----|
 * |-----|
 *
 * The 'sizes' of the above array would be 3,5
 * The last column would be a "subsize" of 3,1
 * And a "start" of 0,5 */

 sizes[0] = nprocs; sizes[1] = XDIM;
 sub[0] = nprocs;   sub[1] = 1;
 starts[0] = 0;     starts[1] = XDIM/2;

 MPI_Type_create_subarray(NDIMS,
    sizes, sub, starts,
    MPI_ORDER_C, MPI_INT, &subarray);
 MPI_Type_commit(&subarray);
```

File view and read

```
MPI_CHECK(MPI_File_set_view(fh, sizeof(header),
  MPI_INT, subarray, "native", info));
MPI_Type_free(&subarray);
MPI_CHECK(MPI_File_read_all(fh,
  read_buf, nprocs, MPI_INT, MPI_STATUS_IGNORE);
```

Hands on continued: Darshan

- How does this workload differ from the write?
- Change the ‘read_all’ to an independent ‘read’
  - What do you think the Darshan output will say? Find out.

Performance portability in I/O:

- Let's look more closely at file-system specific optimizations
- Simple ior benchmark on Polaris vs Ascent (baby Summit) vs Aurora
  - 1 000 000 bytes per process, 48 processes
  - Collective I/O forced on Ascent and Aurora
- Darshan confirms identical MPI-IO workload
- Different transformations for different file systems
  - OST-oriented vs file block

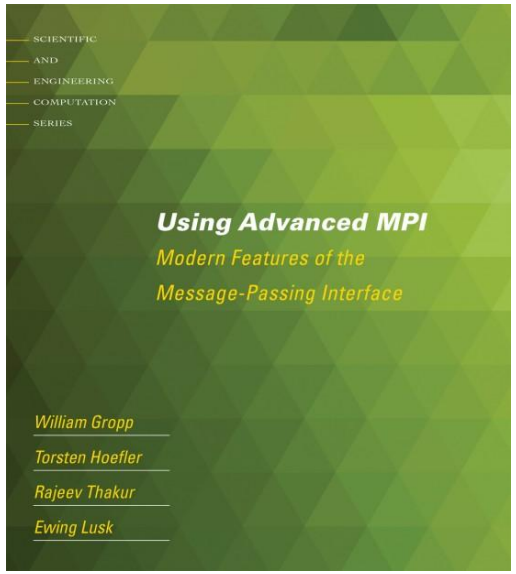
Darshan Counter	Polaris (Lustre)	Ascent (GPFS)	Aurora (DAOS)
MPIIO_ACCESS1_ACCESS	1 000 000	1 000 000	1 000 000
POSIX_WRITES	46	3	
DFS_WRITES			3
POSIX_BYTES_WRITTEN	48000000	48000000	
DFS_BYTES_WRITTEN			48000000
POSIX_SIZE_WRITE_100K_1M	46	0	
POSIX_SIZE_WRITE_10M_100M	0	3	
DFS_SIZE_WRITE_10M_100M			3
POSIX_FILE_ALIGNMENT	4096	-1(*)	
POSIX_SLOWEST_RANK_BYTES	2097152	96000000	
DFS_SLOWEST_RANK_BYTES			49000000

MPI-IO Takeaway

- “Performance Portability”
  - Describe your I/O pattern to MPI-IO and the library will sort out FS-specific approaches/interfaces
- Sometimes it makes sense to build a custom library that uses MPI-IO (or maybe even MPI + POSIX) to write a custom format
  - e.g., a data format for your domain already exists, need parallel API
- We’ve only touched on the API here
  - There is support for data that is noncontiguous in file and memory
  - There are independent calls that allow processes to operate without coordination
- In general we suggest using data model libraries
  - They do more for you
  - Performance can be competitive

Additional Resources

- *I/O Sleuthing*: Another approach towards thinking about tuning IO codes, including MPI-IO
- https://github.com/radix-io/io-sleuthing
- On Cray systems, “man intro_mpi” for 3,000 lines of tuning parameters, debug configuration
- *Using Advanced MPI*, Gropp, Hoeffler, Thakur, Lusk
  - Chapter on MPI I/O routines covers entire API as well as consistency semantics
- Mpi4py: Python bindings to MPI
  - https://mpi4py.readthedocs.io/en/stable/index.html



ARGONNE
ATPESC2025
EXTREME-SCALE COMPUTING

ARGONNE TRAINING PROGRAM ON EXTREME-SCALE COMPUTING

Produced by Argonne National Laboratory, a U.S. Department of Energy Laboratory managed by UChicagoArgonne, LLC under contract DE-AC02-06CH11357.

Special thanks to the National Energy Research Scientific Computing Center (NERSC) and Oak Ridge Leadership Computing Facility (OLCF) for the use of their resources during the training event.