ARGONNE
ATPESC2023
EXTREME-SCALE COMPUTING

Introduction to MPI-IO

Rob Latham
Research Software Developer, Argonne National Laboratory

ECP EXASCALE COMPUTING PROJECT
Argonne NATIONAL LABORATORY

Plan of attack

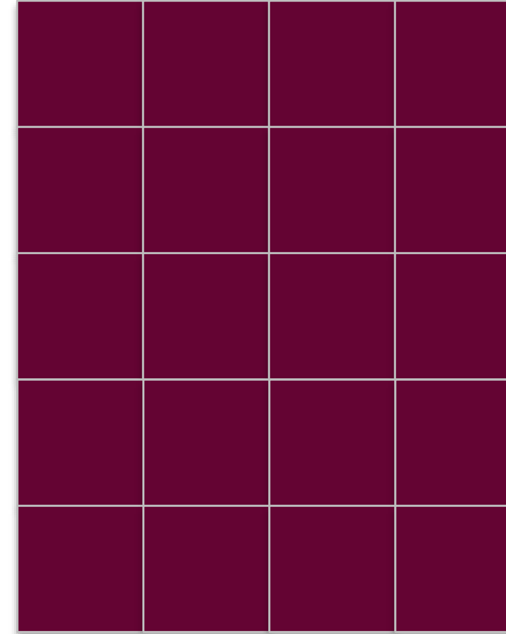
- Bottom-up tour of I/O interfaces
  - POSIX routines called by MPI-IO implementations
  - Parallel-NetCDF routines build on top of MPI-IO
- Simple toy programs
  - Refining example several times throughout session
  - You can apply these lessons to your own code
- Heads up: going to do things the "hard way", then show "easier way"
- Demonstrating some tools for understanding what's going on

Hands on materials

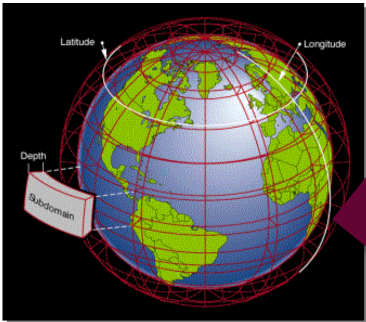
- Code for this …
  - Simple array I/O
- … and other sections available on our gitlab site:
  - Game of Life I/O
  - Sparse Matrix I/O
  - Darshan
  - HDF5
  - IOR recipes
  - https://github.com/radix-io/hands-on
- Work through examples when you can. We're going to do this "cooking show" style…

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.

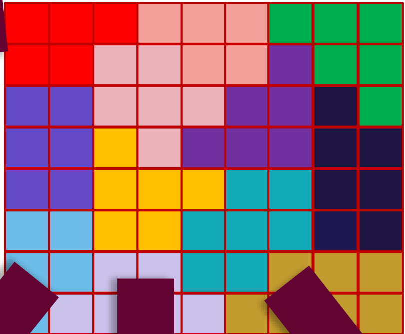


Decomposition

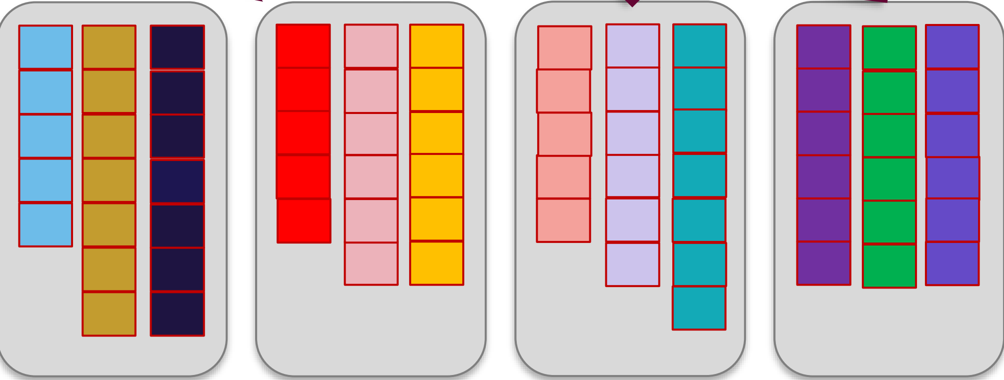


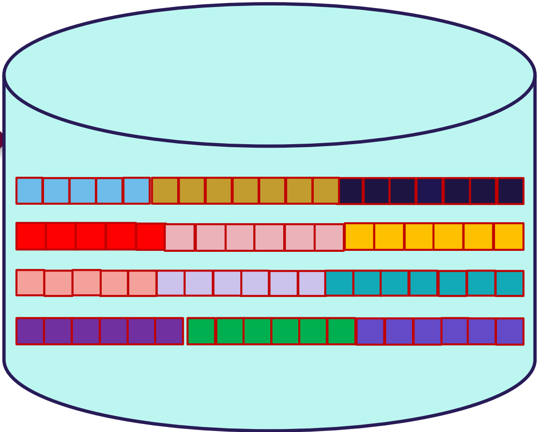
Graphic from J. Tannahill, LLNL

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



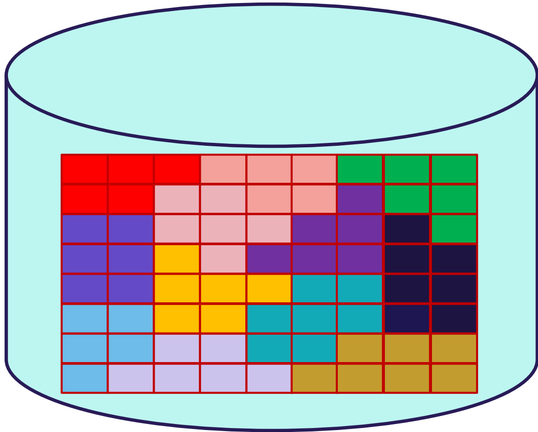
These regions are then distributed to cores (columns) on nodes (grey boxes) for computation.





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.

or



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?
- 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

POSIX I/O

- POSIX is the IEEE Portable Operating System Interface for Computing Environments
- “POSIX defines a standard way for an application program to obtain basic services from the operating system”
  - Mechanism almost all serial applications use to perform I/O
- POSIX was created when a single computer owned its own file system

Deficiencies in serial interfaces

POSIX:

```
fd = open("some_file", O_WRONLY|O_CREAT,
 S_IRUSR|S_IWUSR);
ret = write(fd, w_data, nbytes);
ret = lseek(fd, 0, SEEK_SET);
ret = read(fd, r_data, nbytes);
ret = close(fd);
```

FORTRAN:

```
OPEN(10, FILE='some_file',  &
    STATUS="replace", &
   ACCESS="direct", RECL=16);
WRITE(10, REC=2) 15324
CLOSE(10);
```

- Typical (serial) I/O calls seen in applications
- No notion of other processors
- Primitive (if any) data description methods
- Tuning limited to open flags
- No mechanism for data portability
  - Fortran not even portable between compilers

HANDS-ON: simple data descriptions (no I/O yet)

- Consider an application that operates on a 2-d array of integers.
  1. Write code declaring a 2-d array of integers
     - Probably want to allocate on heap, not stack
     - Later steps will be easier if you make it a single allocation
  2. Define a data structure describing the experiment
     - E.g. C struct with row, column, iteration
- Use whatever language you like…
  - … but we can be most helpful if you use C (c.f. RobL's python "solutions")
- Source "`polaris-setup-env.sh`" to load necessary modules
- Could run this first example on laptop if you want: shouldn't require any libraries

HANDS-ON: simple I/O

- We haven't talked about MPI-IO or I/O libraries, but we can still checkpoint.
  - Serial I/O, not parallel
  - Memory: 2-d array of data plus small header describing structure (see 'util.c')
  - File: same as memory

- Implement "write_data" in `posix.c`
  - Will create file and fill in data
  - Prototype:
    - `int write_data(char *filename)`
  - Use system calls (`open(), write(), close()`) , not "stdio" calls (`fopen(), fwrite(), fclose()`): will map more closely to MPI-IO later
  - How will you know it worked?
  - We are going to repeatedly revise `write_data()` (and later `read_data()`) with each exercise

RUNNING

- Submit to the ‘ATPESC2023’ queue (polaris)
- I’ve provided a ‘submit-polaris.sh’ shell script
  - `qsub -v "APPLICATION=posix,FILENAME=whatever" submit-polaris.sh`
    - If you don’t give FILENAME, then ‘testfile’ used.
- 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/ATPESC2023/usr/$USER`
- Make a directory for your data
  - `Polaris: mkdir –p /grand/ATPESC2023/usr/$USER/`
- Set sensible striping:
  - `lfs setstripe –stripe-count -1 /grand/ATPESC2023/usr/$USER/`

Solution fragments:

```
int write_data(char *filename)
{
    science data = {
        .row = YDIM,
        .col = XDIM,
        .iter = 1
    };

    int *array;
    int fd;
    int ret=0;

    array = buffer_create(0, XDIM, YDIM);

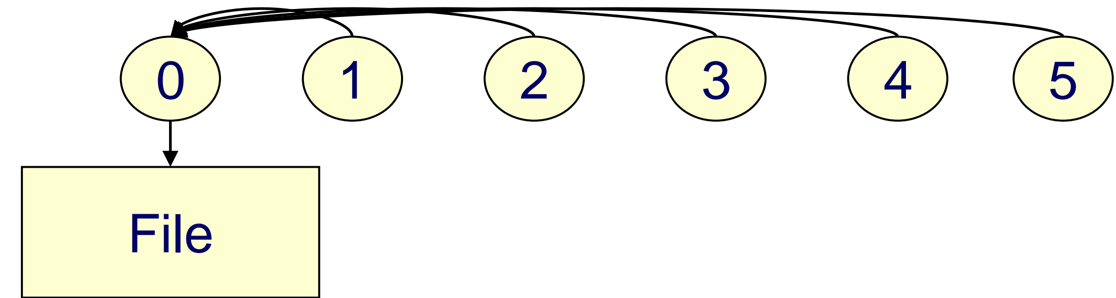
    fd = open(filename, O_CREAT|O_WRONLY,
         S_IRUSR|S_IWUSR);
    ret = write(fd, &data, sizeof(data));
    ret = write(fd, array, XDIM*YDIM*sizeof(int));
    ret = close(fd);

    return ret;
}
```

Reading a **binary file**: "cat" won't work. Could write a c program to read. Several utilities available. I like 'od': (historically it only did an "octal dump"). The (t)ype argument can select (d)ecimal

```
% od -td testfile
0000000           1           5           1           0
0000020           1           2           3           4
0000040
```

HANDS-ON: send-to-master



- Parallel program, but serial I/O
  1. `Write_data()` should take an MPI Communicator
  2. Call MPI_Init() and MPI_Finalize()
  3. Use MPI_Gather to collect all data onto rank 0:
- Only rank 0 does I/O; writes header and all array information
- What's good about send-to-master? What's bad?

Hdr			
0	1	2	3
10	11	12	13
20	21	22	23
30	31	32	33
40	41	42	43

Solution fragments: MPI_Gather: collect all data on rank 0

```
MPI_Comm_rank(comm, &rank);
MPI_Comm_size(comm, &nprocs);
/* every process creates its own buffer */
array = buffer_create(rank, XDIM, YDIM);

/* and then sends it to rank 0  */
int *buffer =
  malloc(XDIM*YDIM*nprocs*sizeof(int));

MPI_CHECK(MPI_Gather(
    /* sender (buffer,count,type) tuple */
    array, XDIM*YDIM, MPI_INT,
    /* receiver tuple */
    buffer, XDIM*YDIM, MPI_INT,
/* who gathers and across which context */
           0, comm));
```

Solution fragments: writing from rank 0

```
if (rank == 0) {
/* looks like serial with more data */
…
/* writing (logically) global array, not
just our local piece of it */
    data.row = YDIM*nprocs;
    data.col = XDIM;
    data.iter = 1;

    ret = write(fd, &data, sizeof(data));
    ret = write(fd, buffer,
         XDIM*YDIM*nprocs*sizeof(int));

    ret = close(fd);
    return ret;
}
```

Other questions:

- Lots of machines (your laptop; Polaris) represent integers as 32 bit little endian. What if you went back in time and ran this code on BlueGene
  - Summit and ascent are powerpc64le
- We wrote row-wise. What if you wanted to write a column of data?
- What impact would a header have on data layout? Are there other options?

HANDS-ON: using Darshan

1. Find the darshan log for the last exercise
2. View the raw counters with “darshan-parser”
3. Generate a report
   - You might have to transfer PDF locally to view
4. Find the darshan log for the exercise #2
   - Hint: you can’t! – why not?
   - …or can you?

- Hint: https://docs.alcf.anl.gov/theta/performance-tools/darshan/ (for older Theta machine but still applies to Polaris)

Parallel I/O and MPI

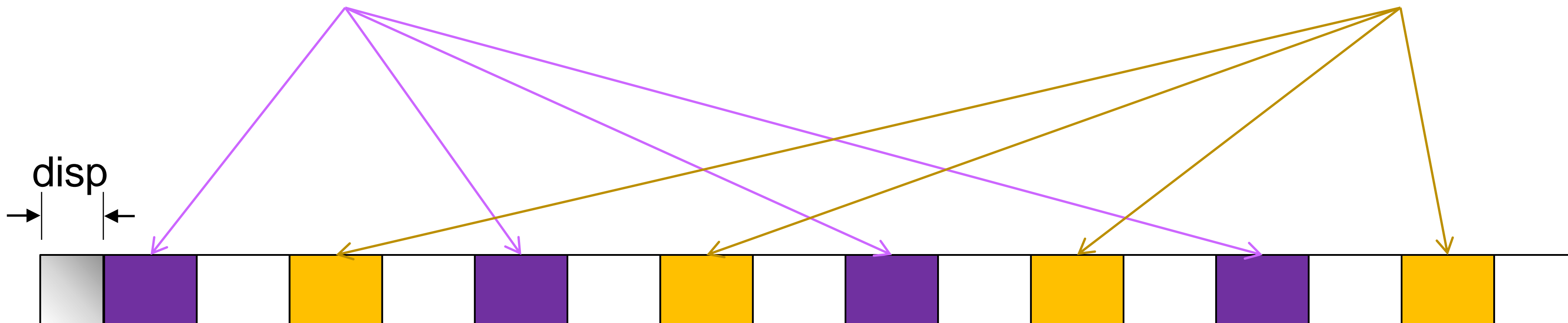
- The stdio checkpoint routine works but is not parallel
  - One process is responsible for all I/O
    - No concurrency in I/O; single link to storage
    - Memory pressure
  - Wouldn't want to use this approach for real
- How can we get the full benefit of a parallel file system?
  - We first look at how parallel I/O works in MPI
  - We then implement a fully parallel checkpoint routine
- MPI is a good setting for parallel I/O
  - Writing is like sending and reading is like receiving
  - Any parallel I/O system will need:
    - collective operations
    - user-defined datatypes to describe both memory and file layout
    - communicators to separate application-level message passing from I/O-related message passing
    - non-blocking operations
  - i.e., lots of MPI-like machinery

Simple MPI-IO

- Collective open: all processes in communicator
- File-side data layout with *file views*
- Memory-side data layout with *MPI datatype* passed to write

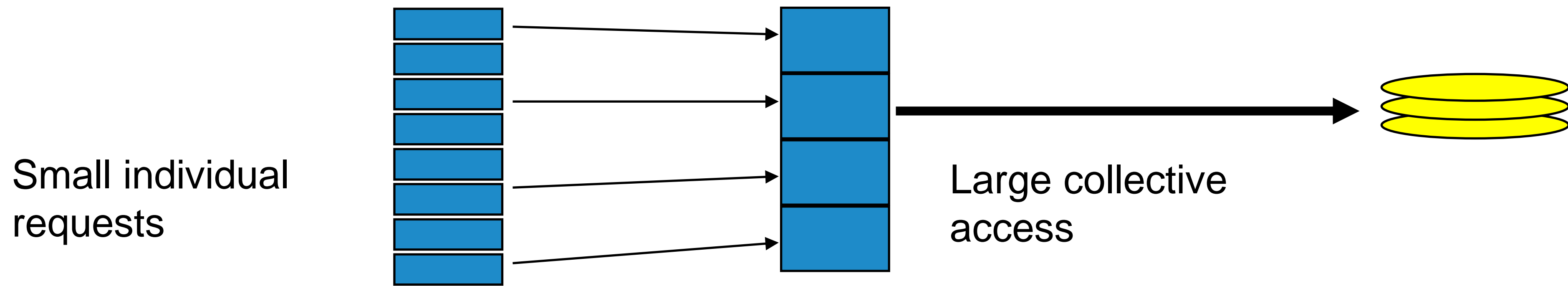
```
MPI_File_open(COMM, name, mode,
        info, fh);
MPI_File_set_view(fh, disp, etype,
        filetype, datarep, info);
MPI_File_write_all(fh, buf, count,
        datatype, status);
```

```
MPI_File_open(COMM, name, mode,
        info, fh);
MPI_File_set_view(fh, disp, etype,
        filetype, datarep, info);
MPI_File_write_all(fh, buf, count,
        datatype, status);
```



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

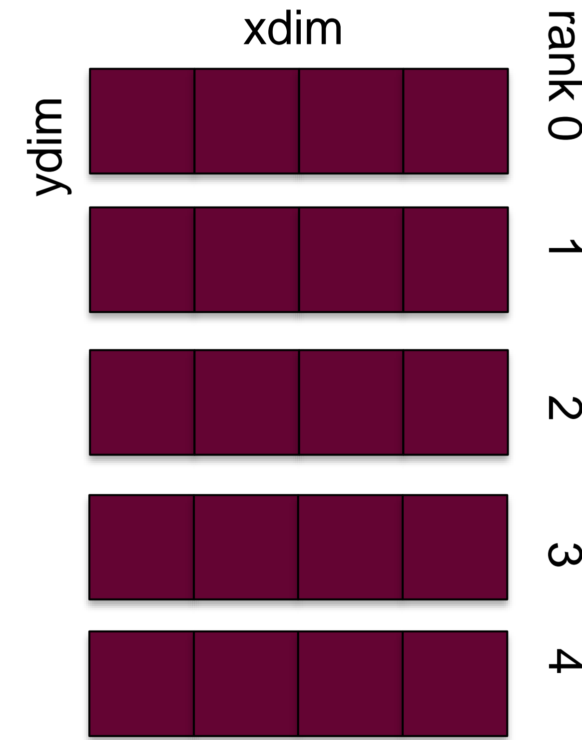


Collective MPI I/O Functions

- 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

- Let's take "I/O from master" example and make it parallel
- 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 for Hands-On 5

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));
```

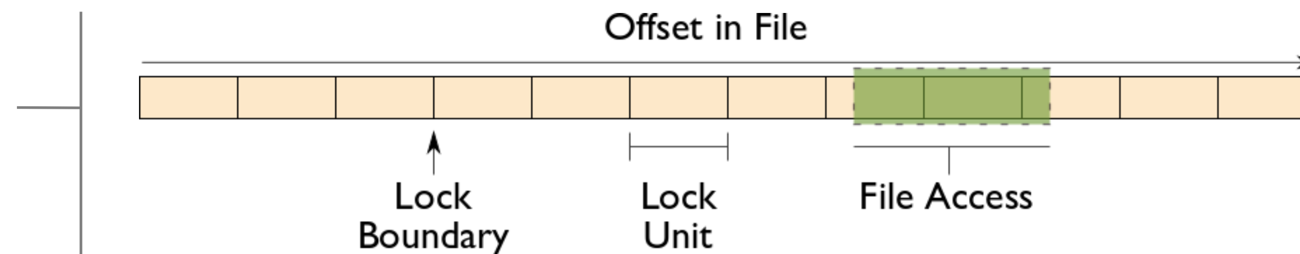
Hands-on continued: Darshan

- 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 vs MPI-IO calls
  - Sizes of POSIX calls vs sizes of MPI-IO calls

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.

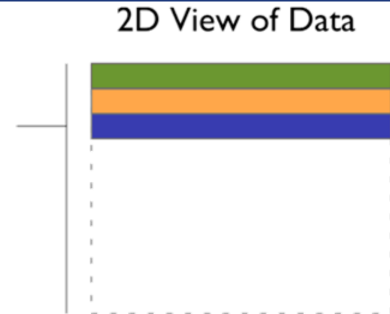


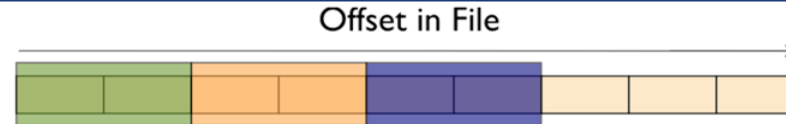
Implications of Locking in Concurrent Access

2D View of Data

Offset in File

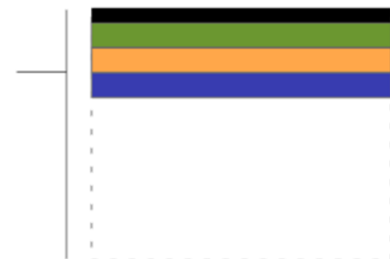
The left diagram shows a row-block distribution of data for three processes. On the right we see how these accesses map onto locking units in the file.

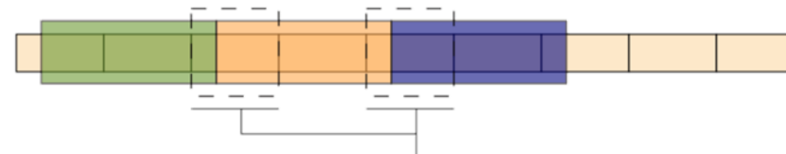




When accesses are to large contiguous regions, and aligned with lock boundaries, locking overhead is minimal.

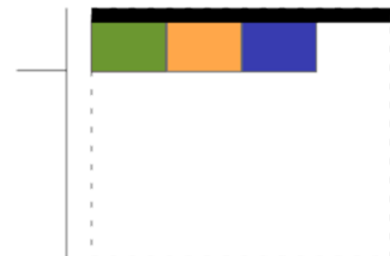
In this example a header (black) has been prepended to the data. If the header is not aligned with lock boundaries, false sharing will occur.

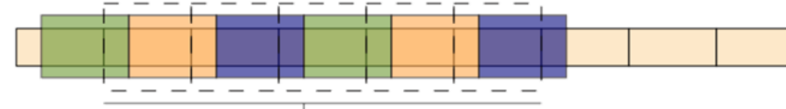




These two regions exhibit *false sharing*: no bytes are accessed by both processes, but because each block is accessed by more than one process, there is contention for locks.

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



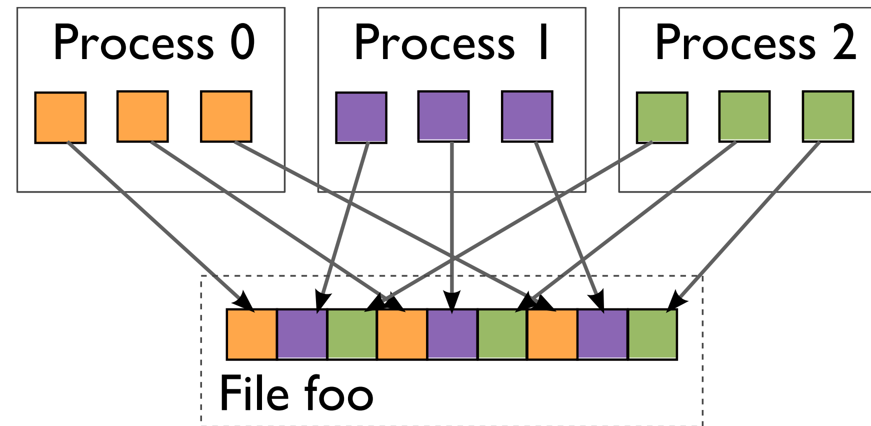


When a block distribution is used, sub-rows cause a higher degree of false sharing, especially if data is not aligned with lock boundaries.

I/O Transformations

- **Software between the application and the file system performs transformations, primarily to improve performance.**

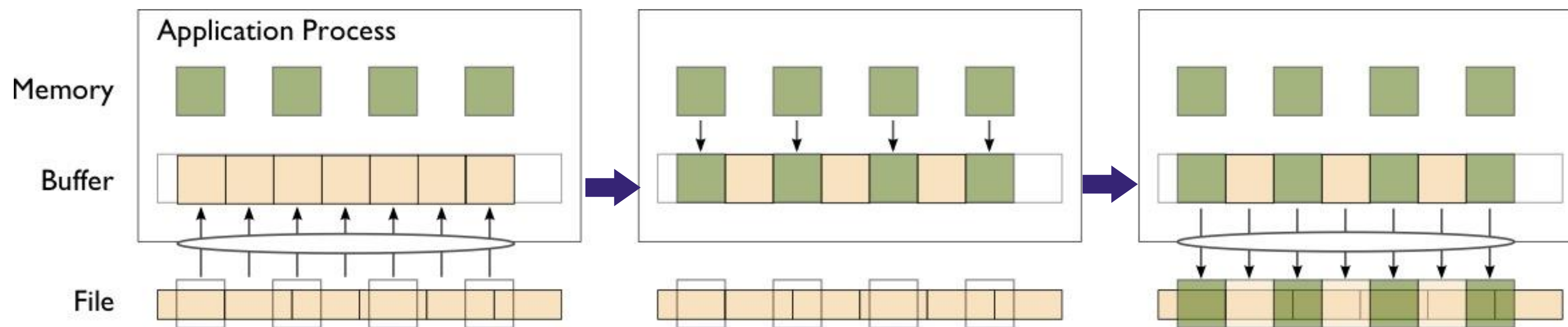
- Goals of transformations:
  - Reduce number of operations to PFS (avoiding latency)
  - Avoid lock contention (increasing level of concurrency)
  - Hide number of clients (more on this later)
- With "transparent" transformations, data ends up in the same locations in the file as it would have been 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).

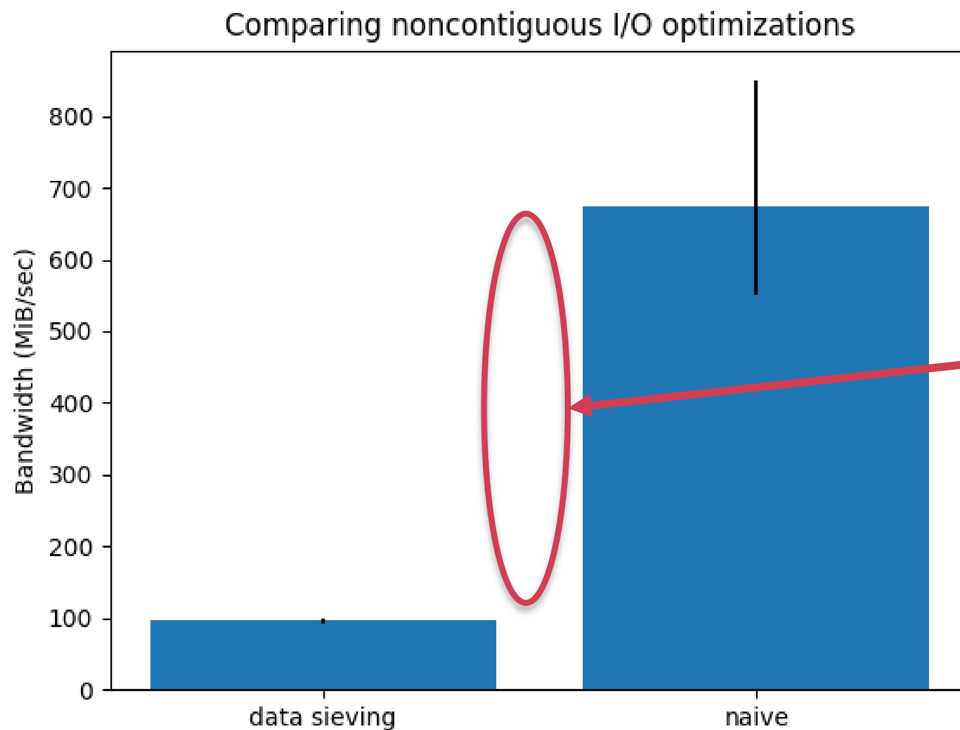
Step 2: Elements to be written to file are replaced in intermediate buffer.

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

Data Sieving in Practice

Not always a win, particularly for writing:

- 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]



	Naiive	Data Sieving
MPI-IO writes	192	192
MPI-IO Reads	0	0
Posix Writes	192000	192000
Posix Reads	0	192015
MPI-IO bytes written	1 920 000 000	1 920 000 000
MPI-IO bytes read	0	0
Posix bytes read	0	100 039 006 128
[1] Posix bytes written	1 920 000 000	100 564 552 704

Selected Darshan statistics

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.

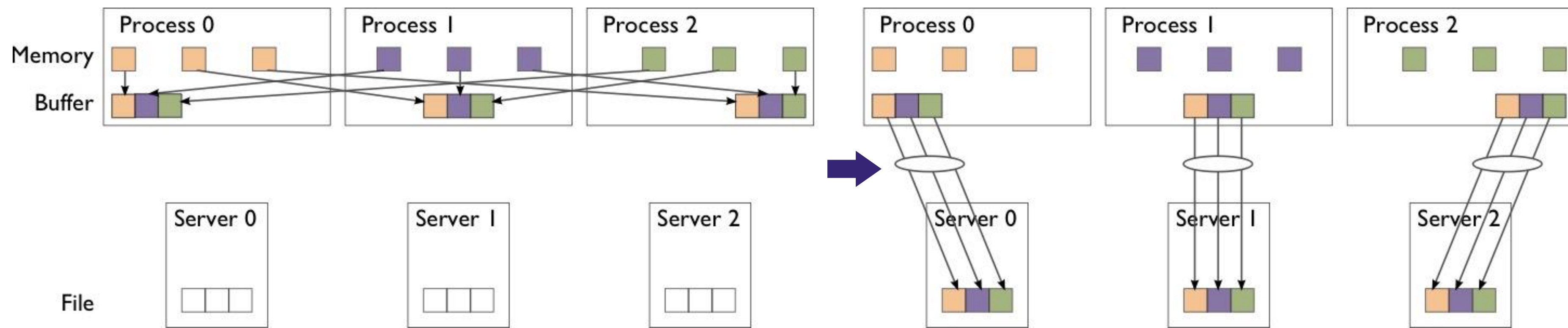
Explore Operation
/grand/ATPESC2022/usr/robl/ior/ior-noncontig-1000.out

Runtime (seconds)

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

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

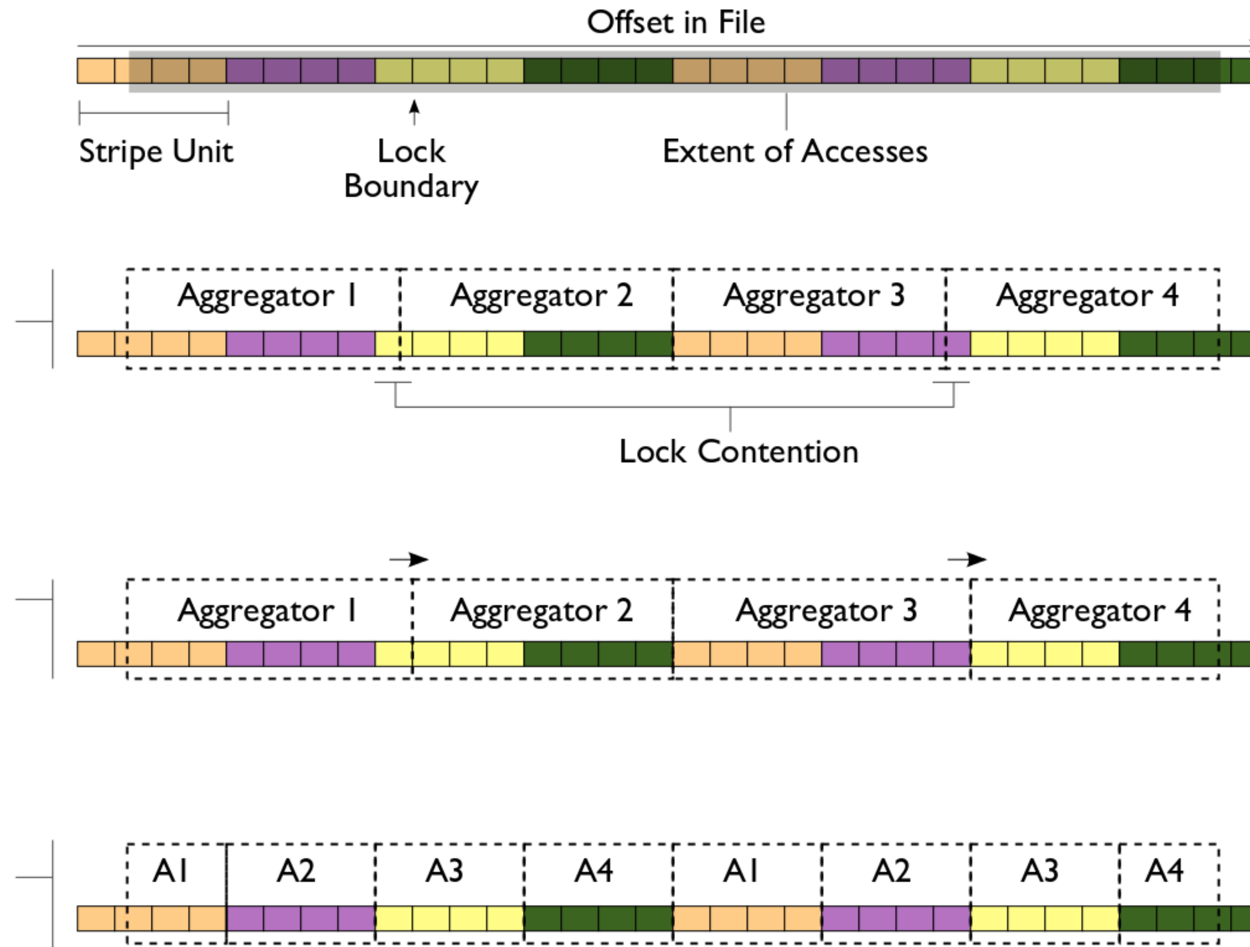
(or, You don't want to do this yourself…)

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).



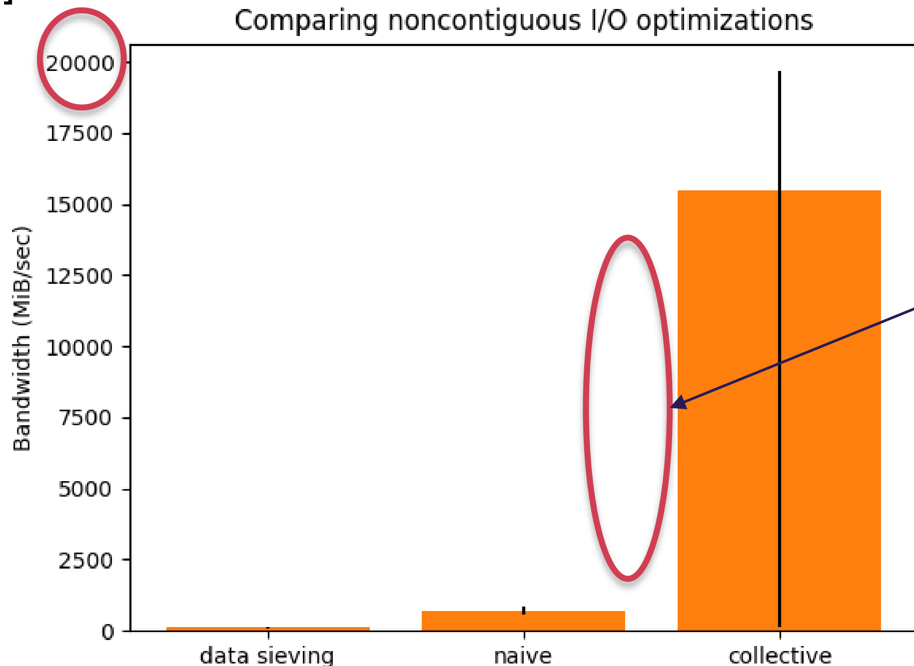
For more information, see W.K. Liao and A. Choudhary, "Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols," SC2008, November, 2008.

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

Two-phase I/O in Practice

- 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.
- (Yes, those are some "impressive" error bars…)

[1]



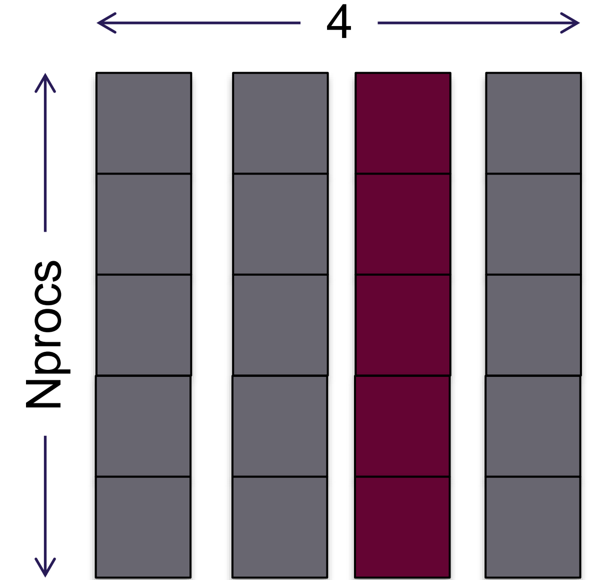
[2]

	Naiive	Data Sieving	Two-phase
MPI-IO writes	192	192	**192**
MPI-IO Reads	0	0	**0**
Posix Writes	192000	192000	**1832**
Posix Reads	0	192015	**0**
MPI-IO bytes written	1 920 000 000	1 920 000 000	**1 920 000 000**
MPI-IO bytes read	0	0	**0**
Posix bytes read	0	100 039 006 128	**0**
Posix bytes written	1 920 000 000	100 564 552 704	**1 920 000 000**

Selected Darshan statistics

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

```c
/* 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

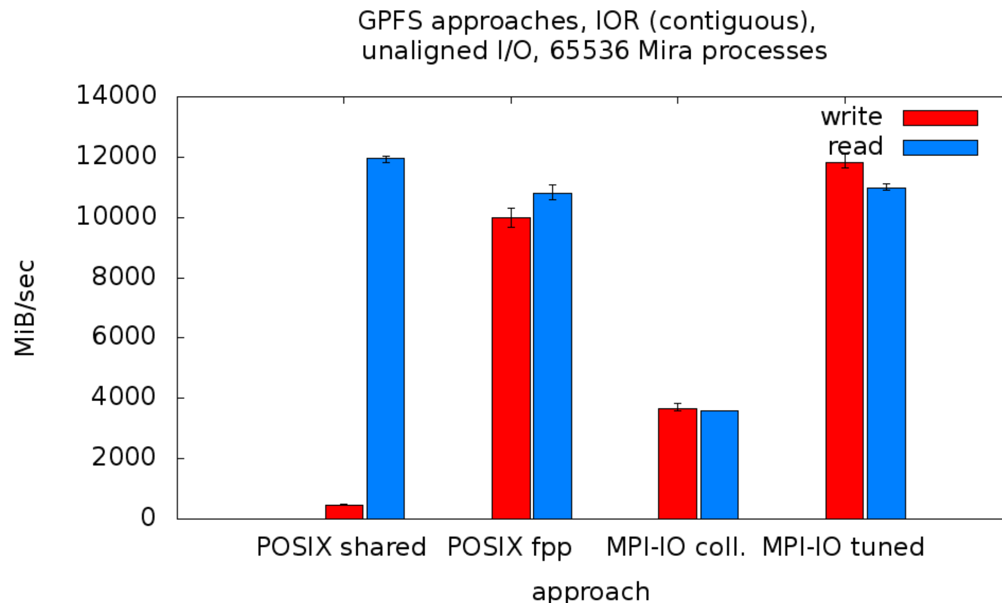
```c
MPI_CHECK(MPI_File_set_view(fh, sizeof(header),
  MPI_INT, subarray, "native", info));
MPI_Type_free(&subarray);
MPI_CHECK(MPI_File_read_all(fh,
  read_buf, nprocs, MPI_INT, MPI_STATUS_IGNORE);
```

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.

GPFS Access three ways

- POSIX shared vs MPI-IO collective
  - Locking overhead for unaligned writes hits POSIX hard
- Default MPI-IO parameters not ideal
  - Reported to IBM; simple tuning brings MPI-IO back to parity
  - “Vendor Defaults” might give you bad first impression
- File per process (fpp) extremely seductive, but entirely untenable on current generation.



Performance portability in I/O:

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

Darshan Counter	Polaris (Lustre)	Ascent (GPFS)
MPIIO_ACCESS1_ACCESS	1 000 000	1 000 000
POSIX_WRITES	46	3
POSIX_BYTES_WRITTEN	48000000	48000000
POSIX_SIZE_WRITE_100K_1M	46	0
POSIX_SIZE_WRITE_10M_100M	0	3
POSIX_FILE_ALIGNMENT	4096	-1(*)
POSIX_SLOWEST_RANK_BYTES	2097152	96000000

MPI-IO Takeaway

- 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