PARALLEL I/O BEST PRACTICES

ROB LATHAM
robl@mcs.anl.gov

PHIL CARNS

KEVIN HARMS

23 August
Petascale Computing Institute

Argonne National Laboratory is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC.

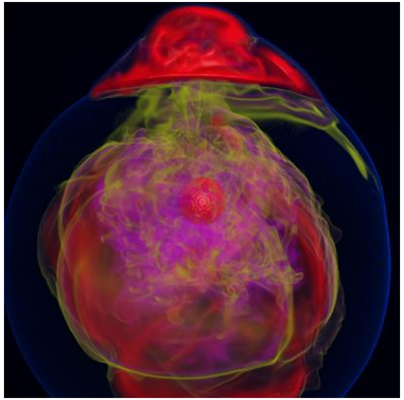
COMPUTATIONAL SCIENCE

- Computer simulation as a tool promotes greater understanding of the real world
  - Complements experimentation and theory

- Problems are increasingly computationally expensive
  - Large parallel machines are needed to perform calculations
  - Leveraging parallelism in all phases is critical

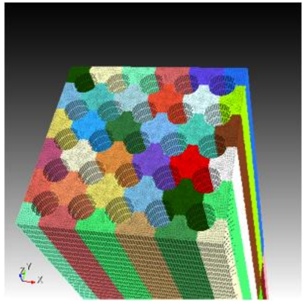
- Data access is a huge challenge and includes
  - Using parallelism to obtain performance
  - Finding usable, efficient, and portable interfaces
  - Understanding and tuning I/O

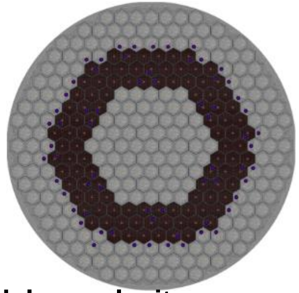
IBM Blue Gene/Q system at Argonne National Laboratory.


Visualization of entropy in Terascale Supernova Initiative application. Image from Kwan-Liu Ma's visualization team at UC Davis.

APPLICATION DATASET COMPLEXITY VS. I/O

- I/O systems have very simple data models
  - Tree-based hierarchy of containers
  - Some containers have streams of bytes (files)
  - Others hold collections of other containers (directories or folders)
- Applications have data models appropriate to domain
  - Multidimensional typed arrays, images composed of scan lines, records of variable length
  - Headers, attributes on data
- Someone has to map from one to the other!





Model complexity:
Spectral element mesh (top) for thermal hydraulics computation coupled with finite element mesh (bottom) for neutronics calculation

Scale complexity:
Spatial range from the reactor core, in meters, to fuel pellets, in millimeters

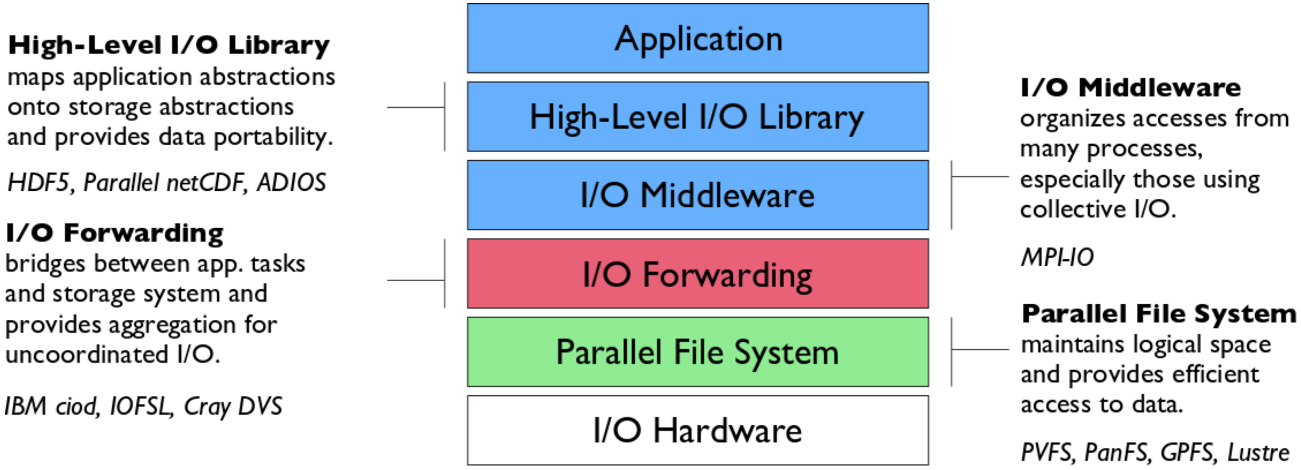
Images from T. Tautges (Argonne) (upper left), M. Smith (Argonne) (lower left), and K. Smith (MIT) (right).

CHALLENGES IN APPLICATION I/O

- Leveraging aggregate communication and I/O bandwidth of clients
  - … While not overwhelming a resource-limited I/O system with uncoordinated accesses!
- Limiting number of files that must be managed
  - Also a performance issue
- Avoiding unnecessary post-processing
- Interacting with storage through convenient abstractions
- Storing in portable formats
  - Often, application teams spend so much time on these two storage issues that they never get any further

Parallel I/O software is available that, when used appropriately, can address all of these problems.

I/O FOR COMPUTATIONAL SCIENCE



Additional I/O software provides improved performance and usability over accessing the parallel file system directly. Reduces or (ideally) eliminates need for optimization in application codes.

UNDERSTANDING I/O BEHAVIOR AND PERFORMANCE

Thanks to the following for much of this material:

Philip Carns, Kevin Harms, Charles Bacon, Sam Lang, Bill Allcock
Math and Computer Science Division and
Argonne Leadership Computing Facility
Argonne National Laboratory

Katie Antypas, Jialin Liu, Quincey Koziol
National Energy Research Scientific Computing Center
Lawrence Berkeley National Laboratory

For more information, see:

- P. Carns, et al., "Understanding and improving computational science storage access through continuous characterization," *ACM TOS* 2011.
- P. Carns, et al., "Production I/O characterization on the Cray XE6," CUG 2013. May 2013.

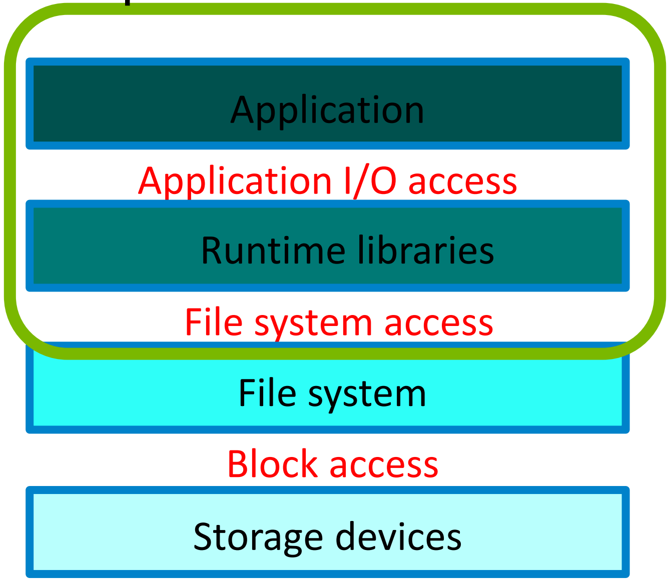
CHARACTERIZING APPLICATION I/O

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?

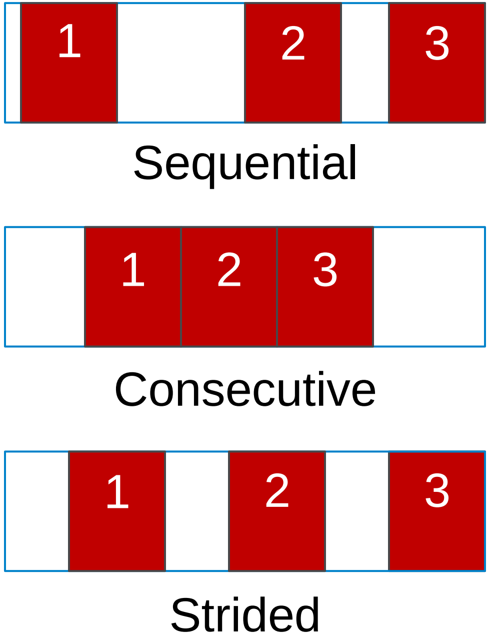
Simplified HPC I/O stack



DARSHAN: A SCALABLE HPC I/O CHARACTERIZATION TOOL

Darshan is a scalable HPC I/O characterization tool that captures ***application*** I/O behavior with minimum overhead.

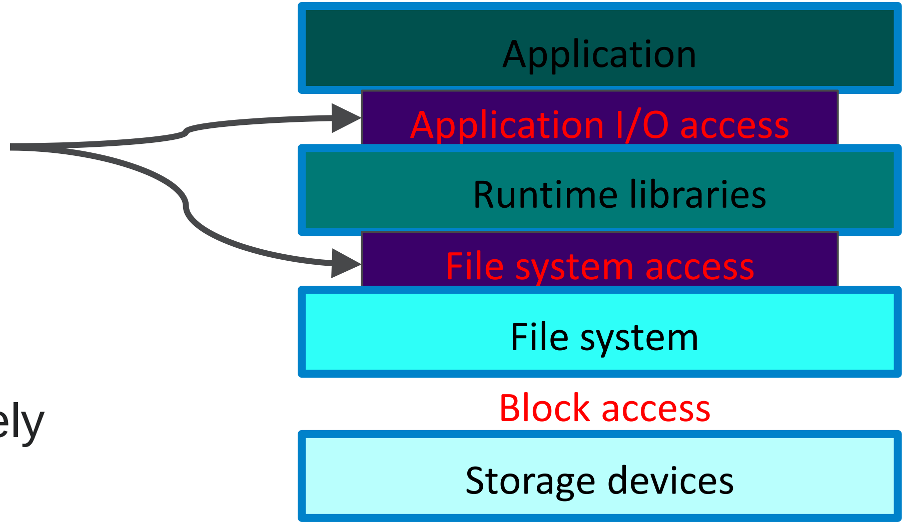
- No code changes, easy to use
  - Negligible performance impact: just "leave it on"
  - Enabled by default at ALCF, NERSC, NCSA, and KAUST
  - Installed and available for case-by-case use at many other sites
- Near-zero overhead
  - Intercepts function calls, but only logs to memory
  - Defers log generation until MPI_Finalize()
- Produces a ***summary*** of I/O activity for each job
  - Counters for file access operations (open/read/write/etc.)
  - Time stamps and cumulative timers for key operations
  - Histograms of access, stride, data type, and extent sizes



UCHICAGO ARGONNE LLC
U.S. DEPARTMENT OF ENERGY
Argonne National Laboratory is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC.
Argonne NATIONAL LABORATORY

DARSHAN DESIGN PRINCIPLES

- Darshan runtime library inserted at link time or at run time
- Transparent wrappers for I/O functions collect per-file statistics
  - Statistics are stored in bounded memory at each rank
  - At MPI_Finalize(), counters are reduced, compressed, and collectively written to a single log
- No communication or storage operations until shutdown
- Command-line tools used to post-process Darshan logs



ANALYZING I/O PERFORMANCE PROBLEMS WITH DARSHAN

Lightweight nature of Darshan means it can be always on and help both **users** and HPC **operators**

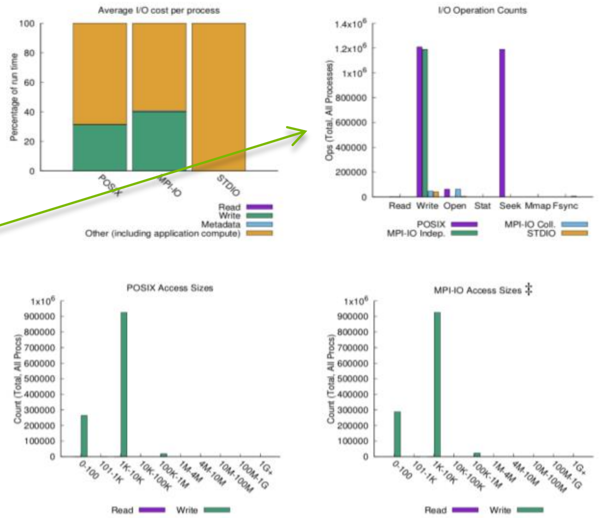
- Users
  - Many I/O problems can be observed from these logs
  - Study applications on demand to debug specific jobs
- Operators
  - Mine logs to catch problems proactively
  - Analyze user behavior, misbehavior, and knowledge gaps

JOB-LEVEL PERFORMANCE ANALYSIS

- Darshan provides insight into the I/O behavior and performance of a job
- **darshan-job-summary.pl** creates a PDF file summarizing various aspects of I/O performance
  - Percent of runtime spent in I/O
  - Operation counts
  - Access size histogram
  - Access type histogram
  - File usage

jobid: 9544193	uid: 59902	nprocs: 4720	runtime: 1512 seconds

I/O performance *estimate* (at the MPI-IO layer): transferred 411195 MiB at 24.77 MiB/s
I/O performance *estimate* (at the STDIO layer): transferred 0.4 MiB at 0.77 MiB/s



Most Common Access Sizes
(POSIX or MPI-IO)

	access size	count
POSIX	4096	660800
	68	47200
	64	47200
	24	23500
MPI-IO ‡	4096	660801
	64	47200
	68	47200
	24	23500

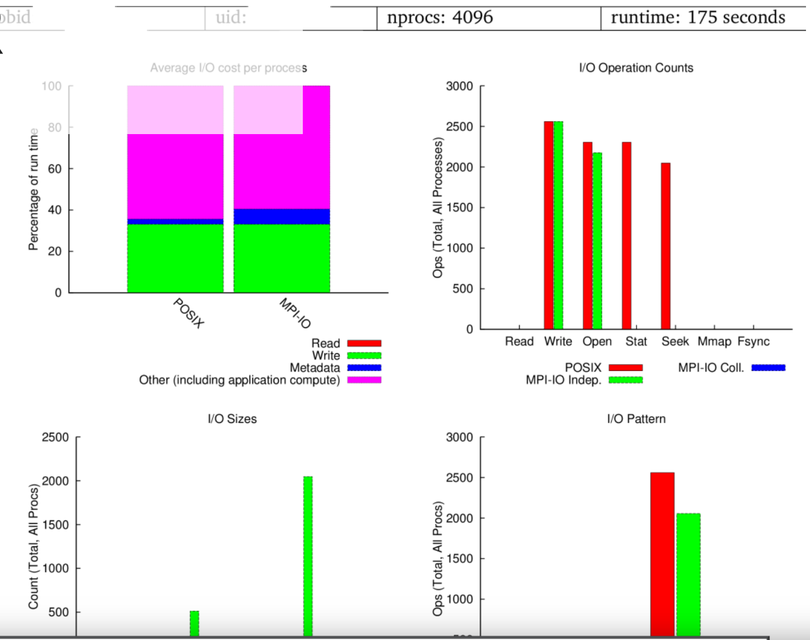
‡ NOTE: MPI-IO accesses are given in terms of aggregate datatype size.

File Count Summary
(estimated by POSIX I/O access offsets)

type	number of files	avg. size	max size
total opened	13	1.4G	3.7G
read-only files	1	2.1K	2.1K
write-only files	11	1.6G	3.7G
read/write files	0	0	0
created files	11	1.6G	3.7G

EXAMPLE: CHECKING USER EXPECTATIONS

- App opens 129 files (one “control” file, 128 data files)
- User *expected* one ~40 KiB header per data file
- Darshan showed 512 headers being written
- Code bug: header was written 4× per file



Most Common Access Sizes

access size	count
67108864	2048
41120	512
8	4
4	3

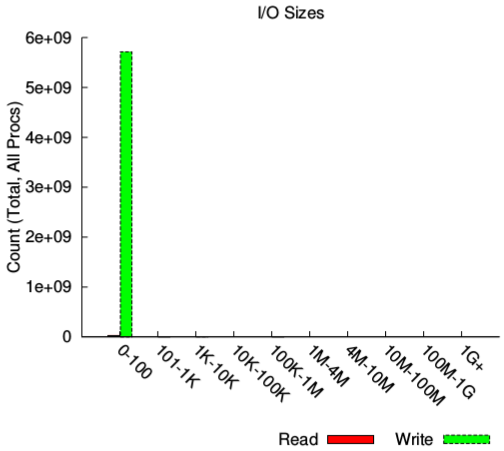
File Count Summary

type	number of files	avg. size	max size
total opened	129	1017M	1.1G
read-only files	0	0	0
write-only files	129	1017M	1.1G
read/write files	0	0	0
created files	129	1017M	1.1G

EXAMPLE: SMALL WRITES TO SHARED FILES

- Scenario: Small writes can contribute to poor performance
  - Particularly when writing to shared files
  - Candidates for collective I/O or batching/buffering of write operations

- Example:
  - Issued 5.7 billion writes to shared files, each less than 100 bytes in size
  - Averaged just over 1 MiB/s per process during shared write phase



Most Common Access Sizes

access size	count
1	3418409696
15	2275400442
24	42289948
12	14725053

SYSTEM-LEVEL PERFORMANCE ANALYSIS

- "Always-on" nature of Darshan enables system-wide I/O analysis
- Daily Top 10 I/O Users list at NERSC to identify users…
  - Running jobs in their home directory
  - Who might benefit from the burst buffer
- Can develop heuristics to detect anomalous I/O behavior
  - Highlight jobs spending a lot of time in metadata
  - Automated triggering/alerting

```
#         Users  Read(GiB) Write(GiB)   # Jobs
==============================================
1.         john     9727.3    10192.7    16432
2.         mary     3672.1     3662.1      701
3.         jane     6777.8      155.6        2

#  File Systems  Read(GiB) Write(GiB)   # Jobs
==============================================
1.     cscratch    18978.4    16940.1     4026
2.        homes    10122.0    10692.7    16565
3.    bb-shared      233.9        8.0        1

#                Applications  Read(GiB) Write(GiB)   # Jobs
============================================================
1.                   vasp_std    10078.2    10528.6    16844
2.                       pw.x     3672.1     3662.1      701
3.                   lmp_cori     6699.4        0.0        1

#                 User/App/FS  Read(GiB) Write(GiB)   # Jobs
============================================================
1.        john/vasp_std/homes     9727.3    10192.7    16432
2.         mary/pw.x/cscratch     3672.1     3662.1      701
3.     jane/lmp_cori/cscratch     6699.4        0.0        1
```

Carns et al., "Production I/O Characterization on the Cray XE6," in *Proceedings of the Cray User Group* (CUG'13), 2013.

AVAILABLE DARSHAN ANALYSIS TOOLS

- Documentation: http://www.mcs.anl.gov/research/projects/darshan/docs/darshan-util.html
- Officially supported tools
  - **darshan-job-summary.pl**: Creates PDF with graphs for initial analysis
  - **darshan-summary-per-file.sh**: Similar to above, but produces a separate PDF summary for every file opened by application
  - **darshan-parser**: Dumps all information into text format
    - For example, darshan-parser user_app_numbers.darshan | grep write
    - Useful for building your own analysis
- Third-party tools
  - **darshan-ruby**: Ruby bindings for darshan-util C library https://xgitlab.cels.anl.gov/darshan/darshan-ruby
  - **HArshaD**: Easily find and compare Darshan logs https://kaust-ksl.github.io/HArshaD/
  - **pytokio**: Detect slow Lustre OSTs, create Darshan scoreboards, etc. https://pytokio.readthedocs.io/

SITE-SPECIFIC DARSHAN INFO

- ALCF
  - https://www.alcf.anl.gov/user-guides/darshan
- NERSC
  - https://www.nersc.gov/users/software/performance-and-debugging-tools/darshan/
- NCSA
  - https://bluewaters.ncsa.illinois.edu/darshan

I/O UNDERSTANDING TAKEAWAY

- Scalable tools like Darshan can yield useful insight
  – Identify characteristics that make applications successful (and those that cause problems)
  – Identify problems to address through I/O research

- Petascale performance tools require special considerations
  – Target the problem domain carefully to minimize amount of data
  – Avoid shared resources
  – Use collectives where possible

- For more information, see: http://www.mcs.anl.gov/research/projects/darshan

MPI-IO

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);
```

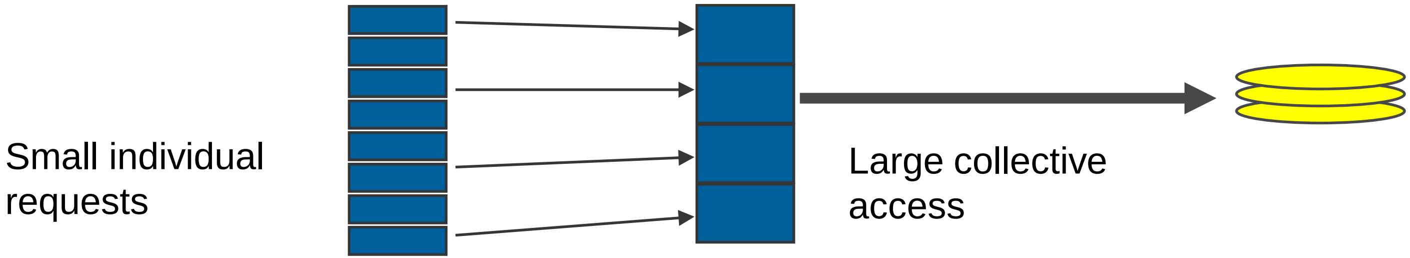
- Typical (serial) I/O calls seen in applications
- No notion of other processors
- Primitive (if any) data description methods
- Tuning limited to open flags
- No mechanism for data portability
  - Fortran not even portable between compilers

PARALLEL I/O AND MPI

- 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

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
- Not ideal if processes enter I/O phase at different times

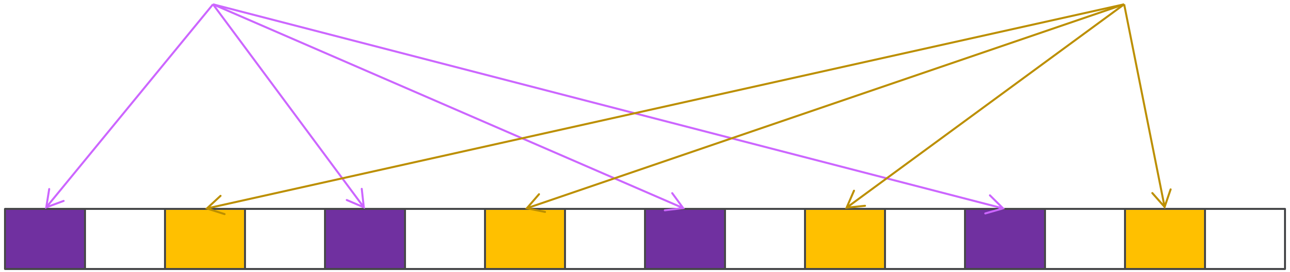


SIMPLE MPI-IO

- 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);
```



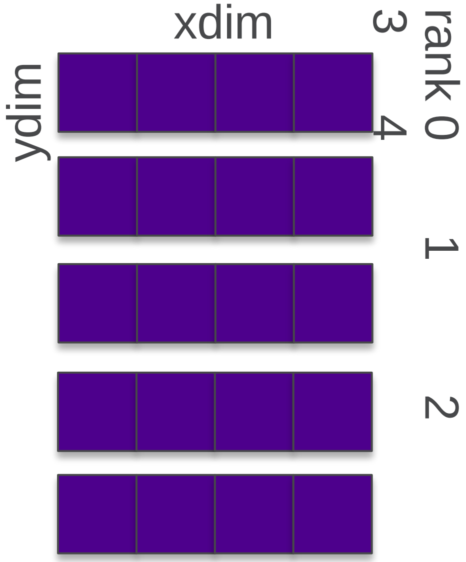
COLLECTIVE MPI I/O FUNCTIONS

- Unable to go through MPI-IO API in detail
- **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

https://xgitlab.cels.anl.gov/robl/hands-on

- Every process writes a row to a 2d array
- Use `MPI_File_open` instead of open
- Only one process needs to write header
  - Independent `MPI_File_write`
- 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 but 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”
  - “(bufer, count, datatype)” tuple: (`values, XDIM*YDIM, MPI_INT`)



UCHICAGO ARGONNE LLC
U.S. DEPARTMENT OF ENERGY
Argonne National Laboratory is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC.
Argonne NATIONAL LABORATORY

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));
```

https://xgitlab.cels.anl.gov/robl/hands-on

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?

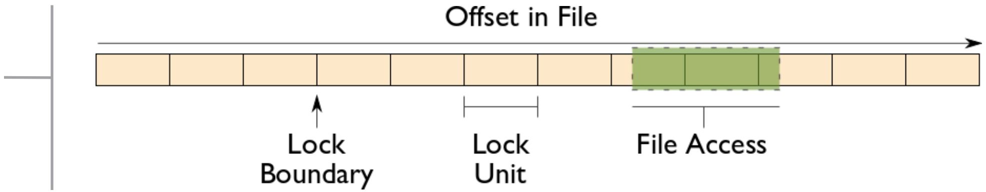
https://xgitlab.cels.anl.gov/robl/hands-on

MANAGING CONCURRENT ACCESS

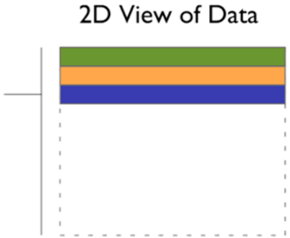
Files are treated like global shared memory regions. Locks are used to manage concurrent access:

- 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

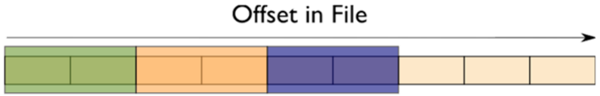
If an access touches any data in a lock unit, the lock for that region must be obtained before access occurs.



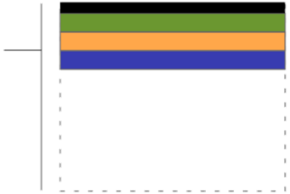
IMPLICATIONS OF LOCKING IN CONCURRENT ACCESS



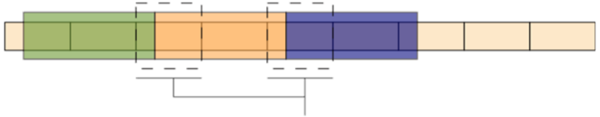
The left diagram shows a row-block distribution of data for three processes. On the right we see how these accesses map onto locking units in the file.



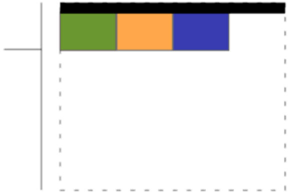
When accesses are to large contiguous regions, and aligned with lock boundaries, locking overhead is minimal.



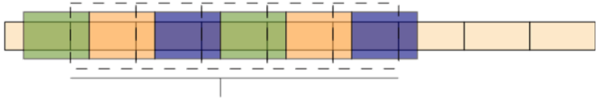
In this example a header (black) has been prepended to the data. If the header is not aligned with lock boundaries, false sharing will occur.



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.

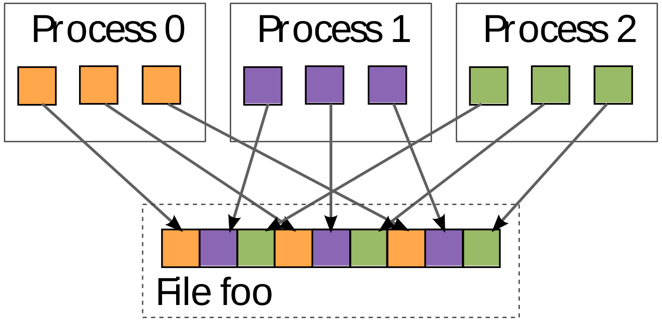


When a block distribution is used, sub-rows cause a higher degree of false sharing, especially if data is not aligned with lock boundaries.

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
- Libraries can do most of this for you!

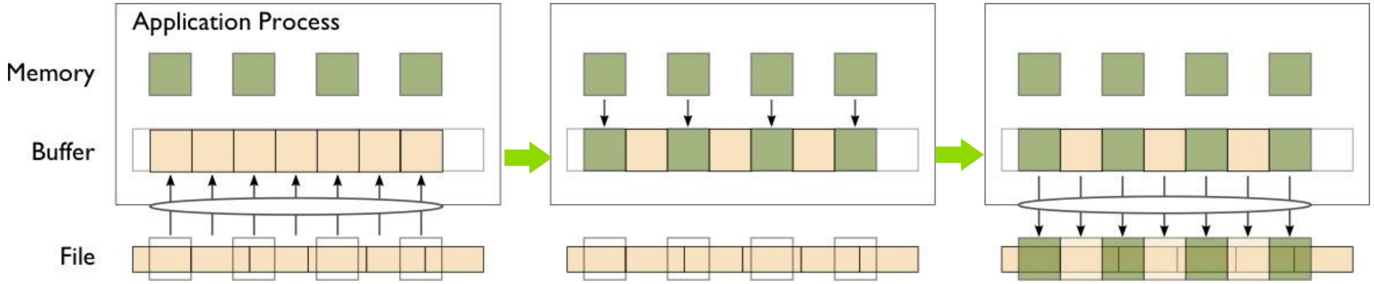


When we think about I/O transformations, we consider the mapping of data between application processes and locations in file.

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).

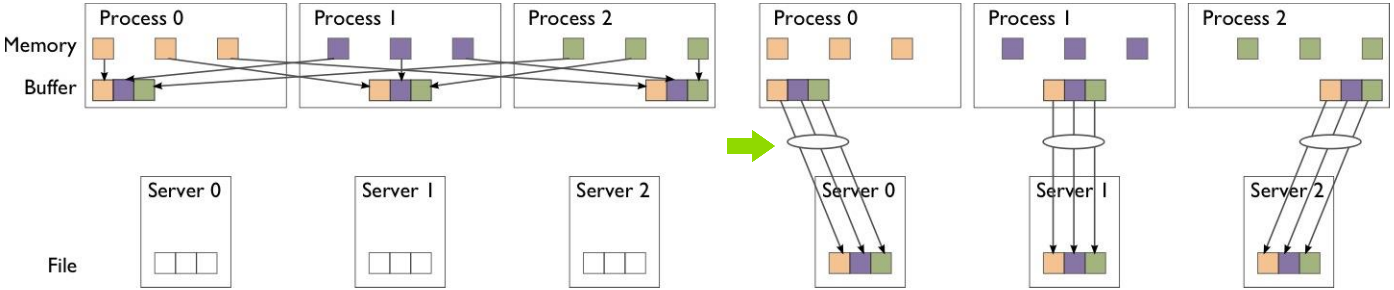
Step 2: Elements to be written to file are replaced in intermediate buffer.

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

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…)

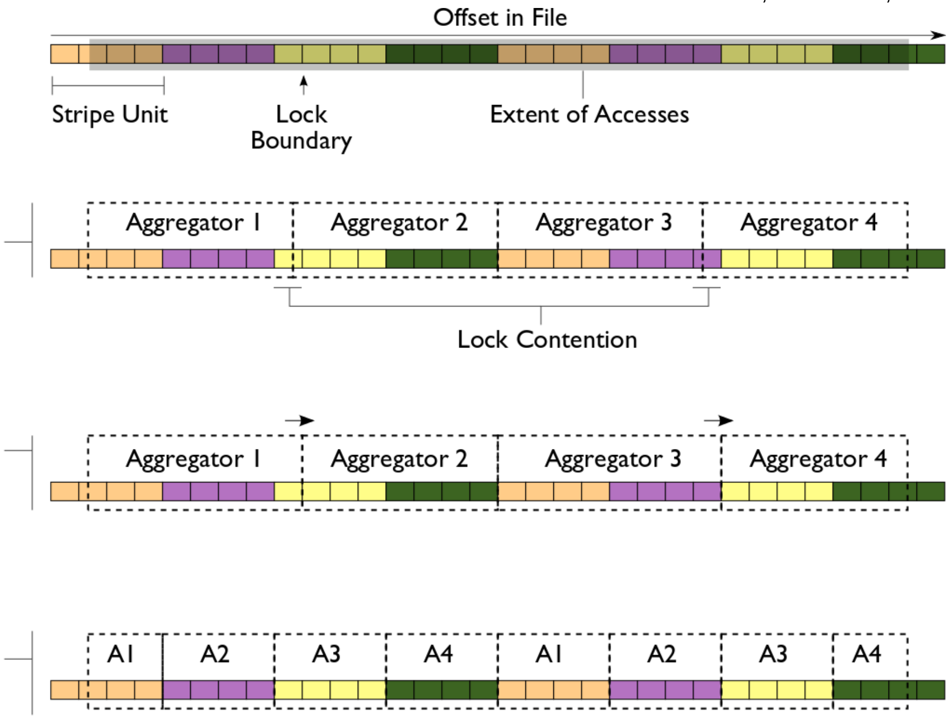
For more information, see W.K. Liao and A. Choudhary, "Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols," SC2008, November, 2008.

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.

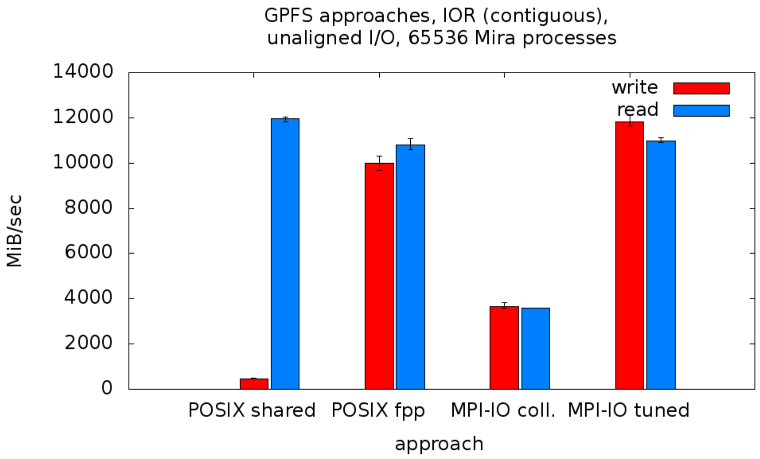
Mapping aggregators to servers reduces the number of concurrent operations on a single server and can be helpful when locks are handed out on a per-server basis (e.g., Lustre).



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

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.

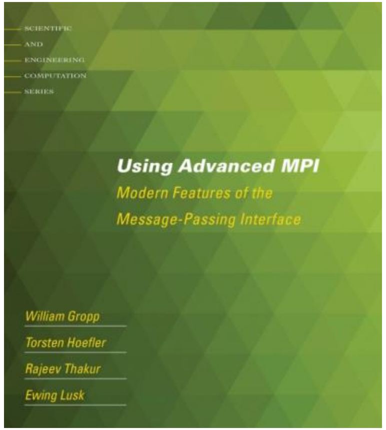


MPI-IO TAKEAWAY

- 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

MPI-IO REFERENCES

- 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



HIGHER-LEVEL I/O LIBRARIES (PNETCDF AND HDF5)

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

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.

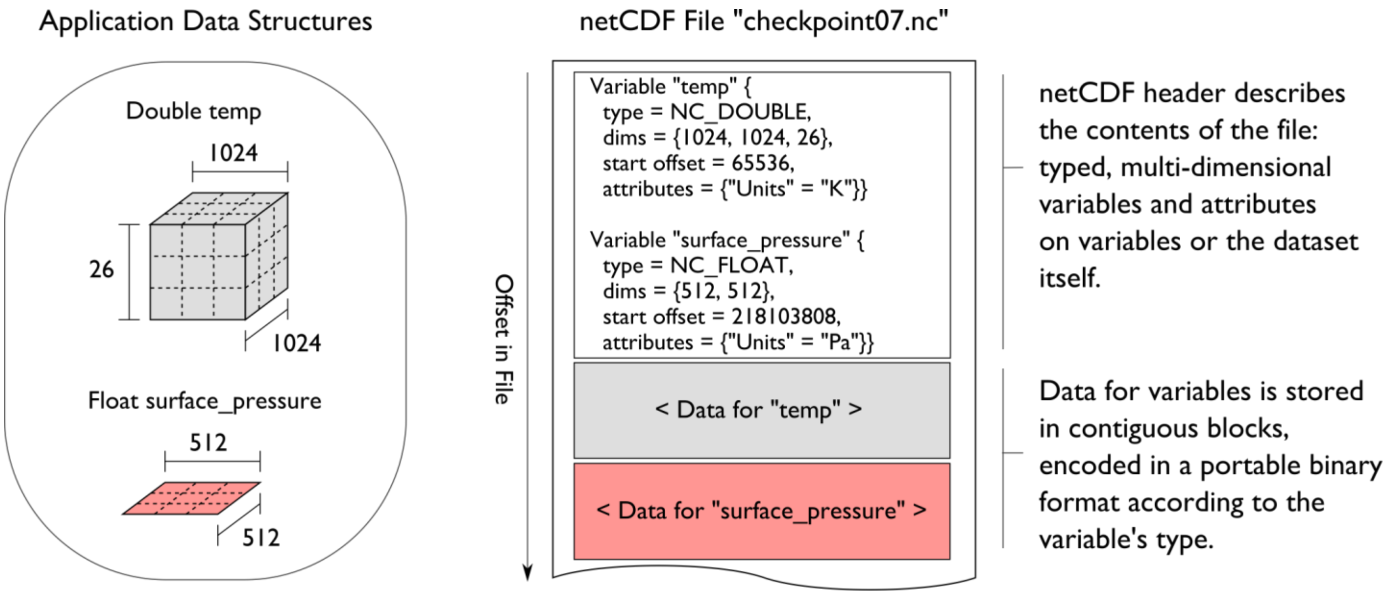
www.mcs.anl.gov/parallel-netcdf

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
  - 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:
  - http://trac.mcs.anl.gov/projects/parallel-netcdf/wiki/QuickTutorial
- Interface guide:
  - http://cucis.ece.northwestern.edu/projects/PnetCDF/doc/pnetcdf-c/index.html

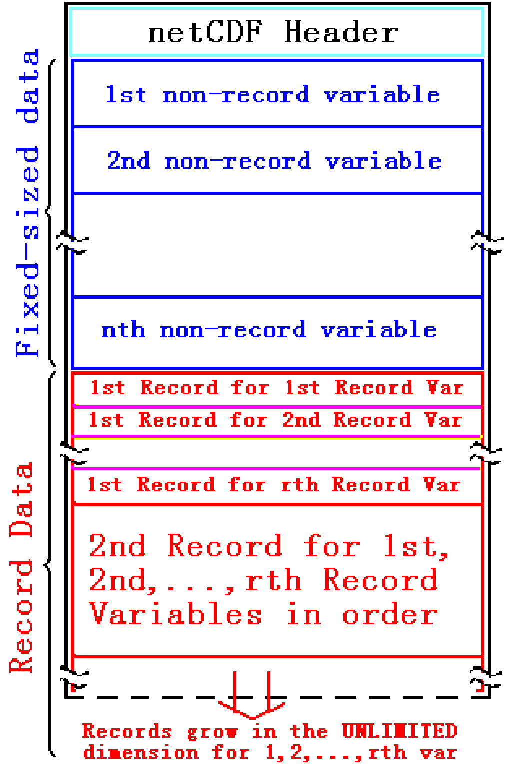
NETCDF DATA MODEL

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



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



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)

HANDS-ON: WRITING WITH PARALLEL-NETCDF

- Many details managed by pnetcdf library
- 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()`

https://xgitlab.cels.anl.gov/robl/hands-on

SOLUTION FRAGMENTS FOR HANDS-ON #7

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) );
```

https://xgitlab.cels.anl.gov/robl/hands-on

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!

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
    - Contiguous region for each process in FLASH case
  - `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

HANDS-ON 7 CONTINUED

- Take a look at the Darshan report for your job.

PARALLEL-NETCDF INQUIRY ROUTINES

- Talked a lot about writing, but what about reading?
- Parallel-NetCDF QuickTutorial contains examples of several approaches to reading and writing
- General approach
  1. Obtain simple counts of entities (similar to MPI datatype “envelope”)
  2. Inquire about length of dimensions
  3. Inquire about type, associated dimensions of variable
- Real application might assume convention, skip some steps
- A full parallel reader would, after determining shape of variables, assign regions of variable to each rank (“decompose”).
  - Next slide focuses only on inquiry routines. (See website for I/O code)

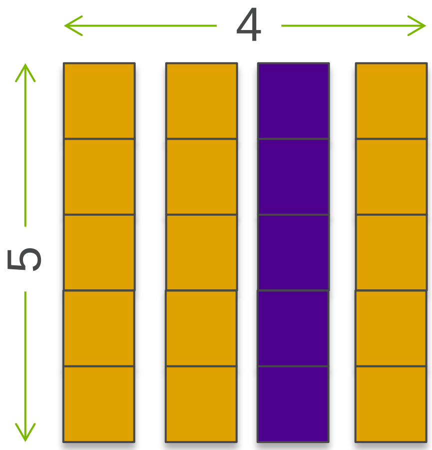
PARALLEL NETCDF INQUIRY ROUTINES

```
int main(int argc, char **argv) {
    /* extracted from
     *http://trac.mcs.anl.gov/projects/parallel-netcdf/wiki/QuickTutorial
     * "Reading Data via standard API" */
    MPI_Init(&argc, &argv);
    ncmpi_open(MPI_COMM_WORLD, argv[1], NC_NOWRITE,
            MPI_INFO_NULL, &ncfile);

    /* reader knows nothing about dataset, but we can interrogate with
     * query routines: ncmpi_inq tells us how many of each kind of
     * "thing" (dimension, variable, attribute) we will find in file */

1   ncmpi_inq(ncfile, &ndims, &nvars, &ngatts, &has_unlimited);
    /* no communication needed after ncmpi_open: all processors have a
     * cached view of the metadata once ncmpi_open returns */

    dim_sizes = calloc(ndims, sizeof(MPI_Offset));
    /* netcdf dimension identifiers are allocated sequentially starting
     * at zero; same for variable identifiers */
    for(i=0; i<ndims; i++)  {
2       ncmpi_inq_dimlen(ncfile, i, &(dim_sizes[i]) );
    }
3   for(i=0; i<nvars; i++) {
        ncmpi_inq_var(ncfile, i, varname, &type, &var_ndims, dimids,
                &var_natts);
        printf("variable %d has name %s with %d dimensions"
                " and %d attributes\n",
                i, varname, var_ndims, var_natts);
    }
    ncmpi_close(ncfile);
    MPI_Finalize();
}
```

Argonne National Laboratory is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC.

HANDS-ON 8: 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



https://xgitlab.cels.anl.gov/robl/hands-on

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, datatypes: just a starting coordinate and size

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

https://xgitlab.cels.anl.gov/robl/hands-on

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
  - New extended file format to relax these limits (CDF-5, released in pnetcdf-1.1.0, November 2009, integrated in Unidata NetCDF-4.4)

DATA MODEL I/O LIBRARIES

- Parallel-NetCDF: http://www.mcs.anl.gov/pnetcdf
- HDF5: http://www.hdfgroup.org/HDF5/
- NetCDF-4: http://www.unidata.ucar.edu/software/netcdf/netcdf-4/
  - netCDF API with HDF5 back-end
- ADIOS: http://adiosapi.org
  - Configurable (xml) I/O approaches
- SILO: https://wci.llnl.gov/codes/silo/
  - A mesh and field library on top of HDF5 (and others)
- H5part: http://vis.lbl.gov/Research/AcceleratorSAPP/
  - simplified HDF5 API for particle simulations
- GIO: https://svn.pnl.gov/gcrm
  - Targeting geodesic grids as part of GCRM
- PIO:
  - climate-oriented I/O library; supports raw binary, parallel-netcdf, or serial-netcdf (from master)
- … Many more: consider existing libs before deciding to make your own.

THANKS!

REMEMBER -- I/O PROBLEMS: YOU ARE NOT ALONE

UCHICAGO ARGONNE LLC
U.S. DEPARTMENT OF ENERGY
Argonne National Laboratory is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC.
Argonne NATIONAL LABORATORY