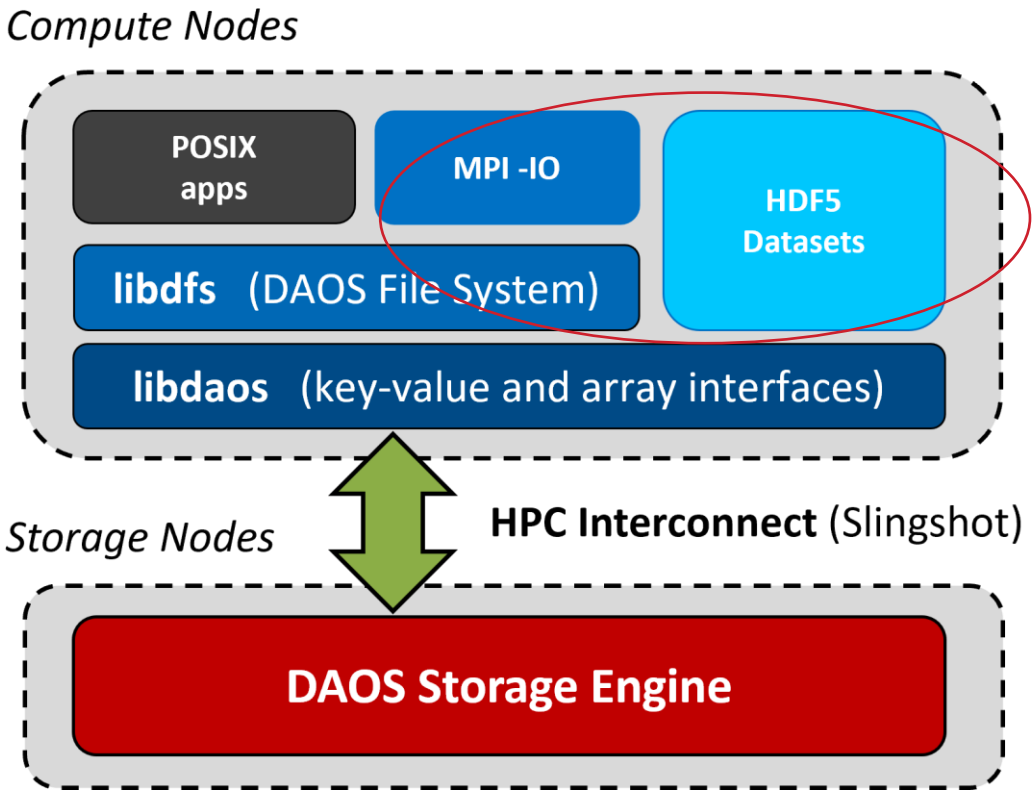
MPI-IO

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…

DAOS OVERVIEW

- Object store
- Concurrency focused
- Many interfaces
  - We'll focus on MPI-IO and a bit of HDF5 today
  - POSIX interface for legacy codes
- Hands-on scripts demonstrate daos access
- https://docs.alcf.anl.gov/aurora/data-management/daos/daos-overview for more details



Hands on exercises:
https://github.com/radix-io/hands-on

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

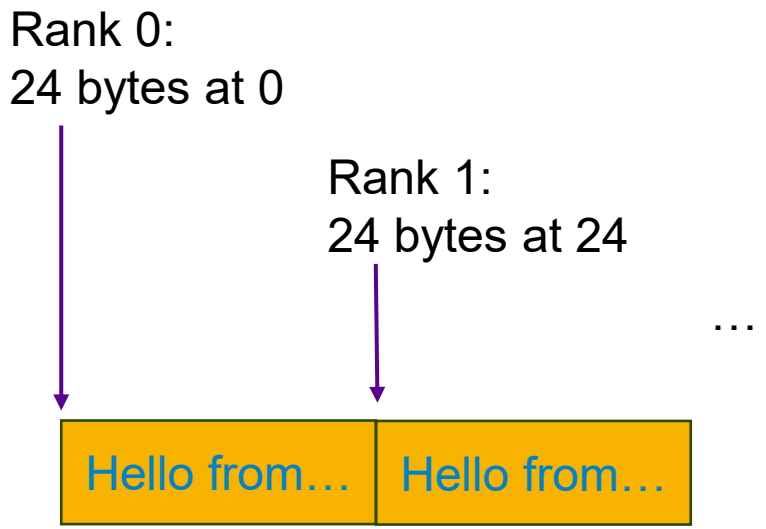
```
/* an "Info object":  these store key-value strings for tuning the
 * underlying MPI-IO implementation */
MPI_Info_create(&info);

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

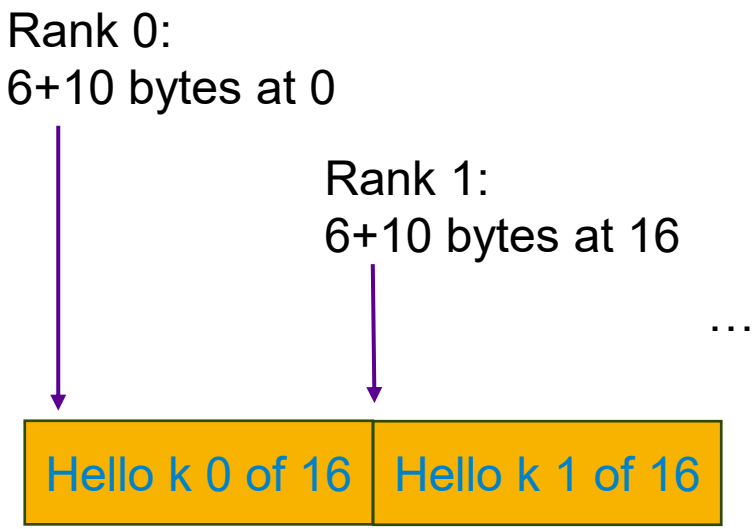
```
MPI_Datatype memtype;
MPI_Count memtype_size;

…
/* 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

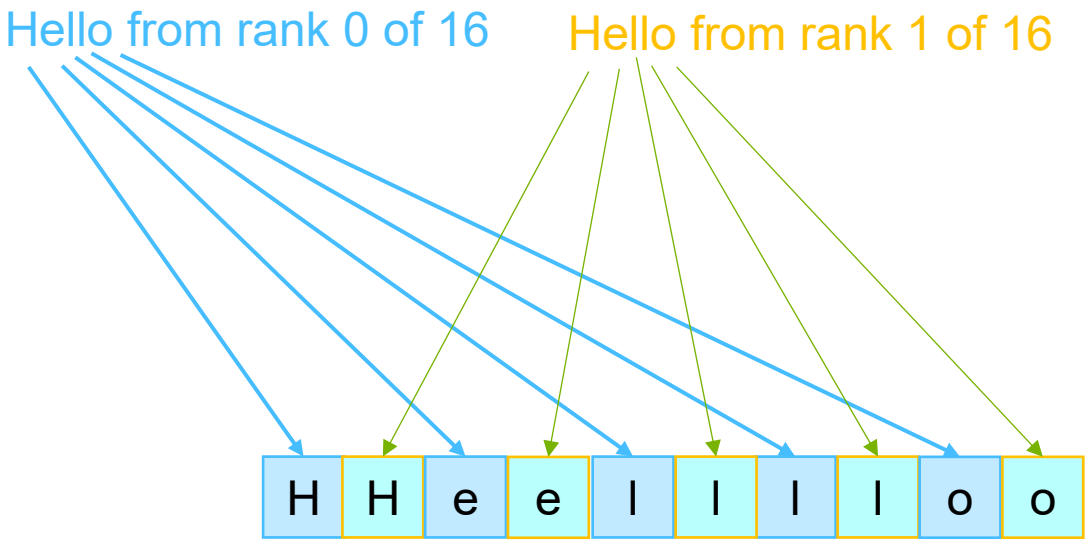


"HELLO WORLD" MPI-IO STYLE: NON-CONTIGUOUS IN FILE

```
/* 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 ‘ATPESC2026’ 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 “ATPESC2026” pool on the “daos_user” 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.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

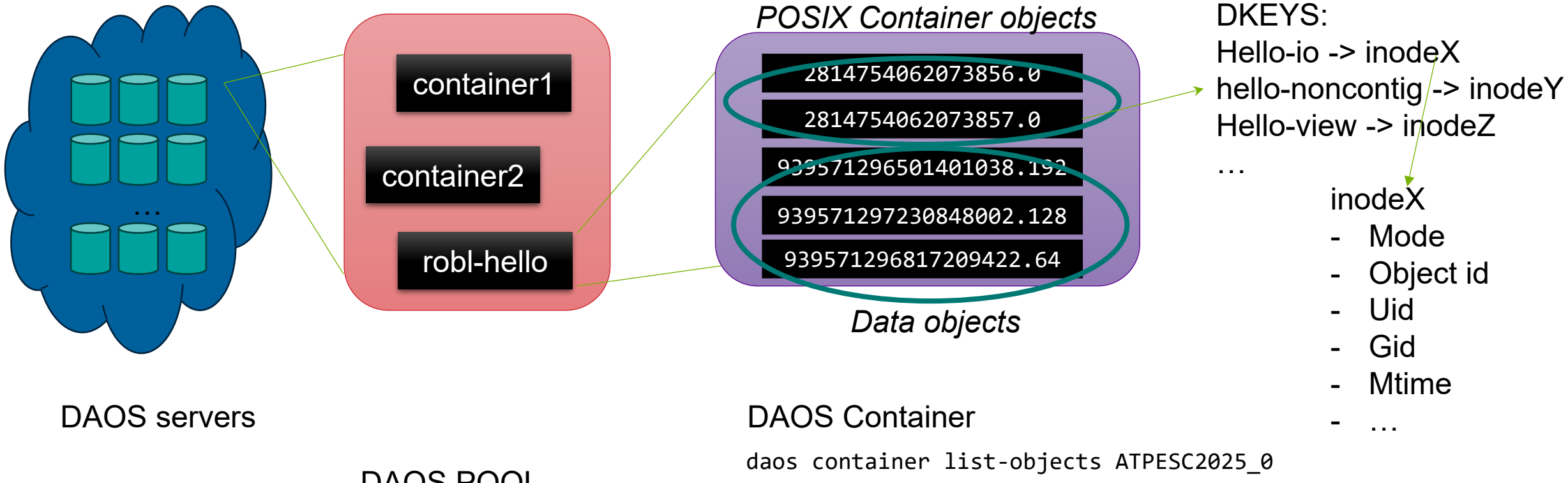
```
==== contiguous in memory and file
cat /tmp/ATPESC2026/robl-hello/hello.out
Hello from rank 0 of 16
Hello from rank 1 of 16
…
Hello from rank 15 of 16


==== noncontiguous in memory
cat /tmp/ATPESC2026/robl-hello/hello-
noncontig.out
Hello k 0 of 16
Hello k 1 of 16
…
Hello  15 of 16
```

```
==== noncontiguous in file
cat /tmp/ATPESC2026/robl-hello/hello-
view.out
HHHHHHHHHHHHHHHeeeeeeeeeeeeeeeellllll
lllllllllllllllllllllllooooooooooooo
oo
fffffffffffffffrrrrrrrrrrrrrrrooooooo
ooooooooommmmmmmmmmmmmmmm
rrrrrrrrrrrrrraaaaaaaaaaaaaaannnnnnn
nnnnnnnnnkkkkkkkkkkkkkkk
11111101234567890 12345
ooooooooooooooofffffffffffffff
11111111111111166666666666666666
```

Output of our hello programs

UNDER THE HOOD: DAOS (ESSENTIALLY)



DAOS servers

DAOS POOL

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

DAOS Container

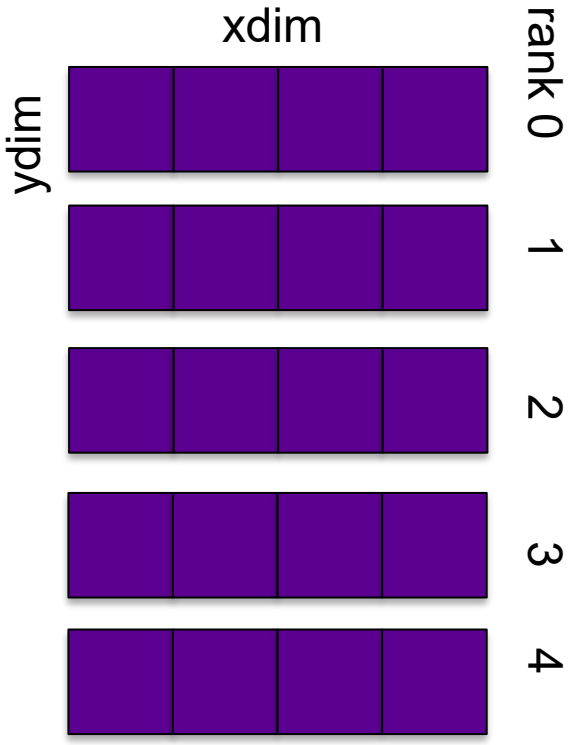
```
daos container list-objects ATPESC2025_0
```

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

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

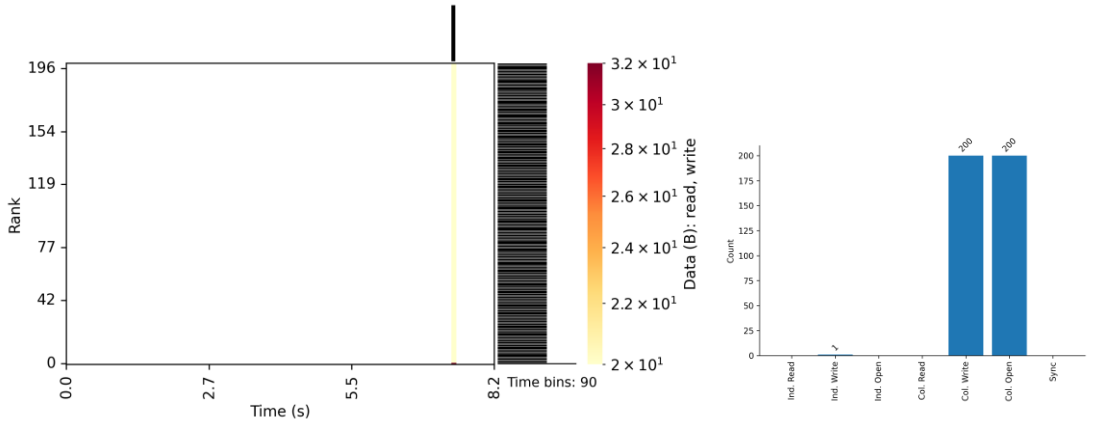
- Let's use Darshan
  - Find Darshan log file, but don't generate report right away
- What do you think the report will say?
- OK, now generate the report. Were you surprised?
  - Counts of POSIX calls (POSIX_WRITES) vs MPI-IO calls (MPIIO_COLL_WRITES)
  - Sizes of POSIX calls vs sizes of MPI-IO calls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

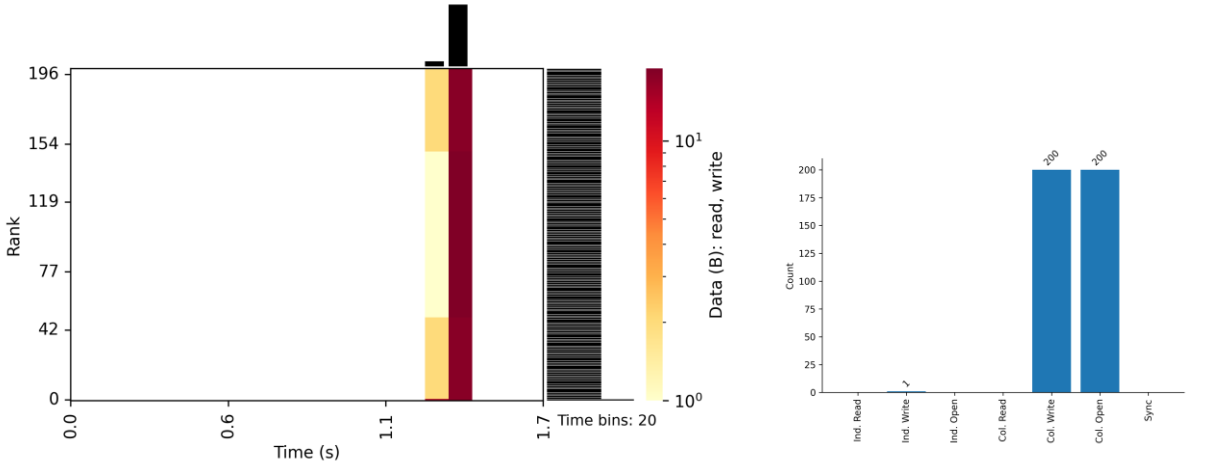
HANDS-ON CONTINUED: DARSHAN

Default (independent)

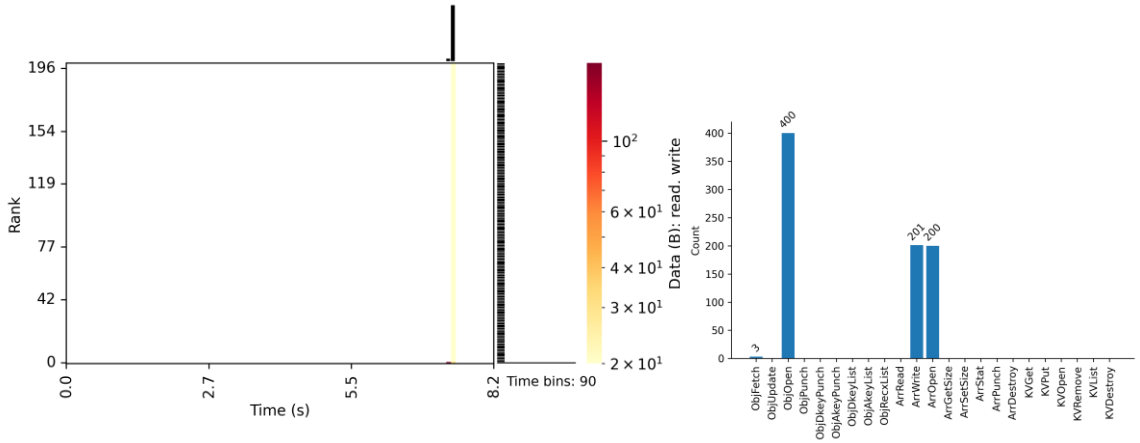
Hinted (collective)

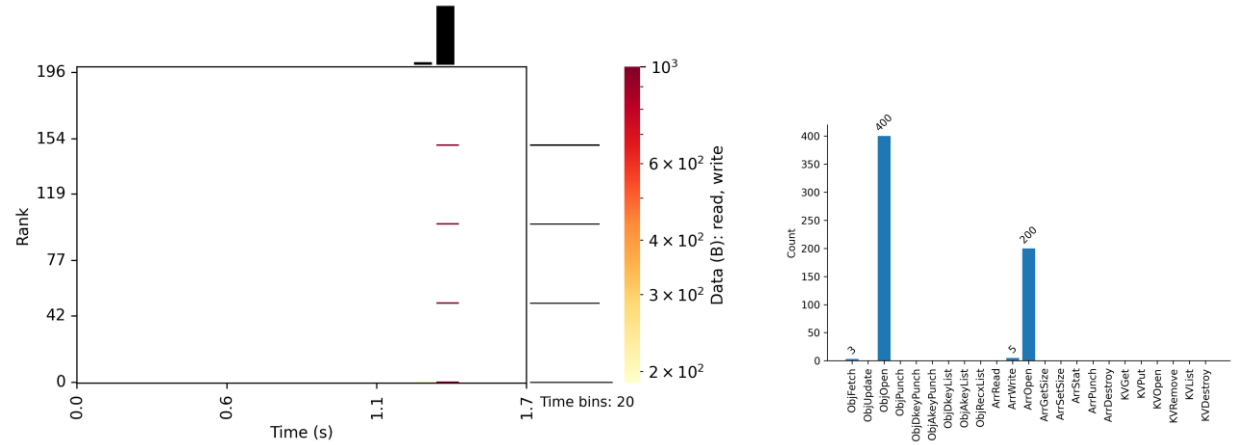
MPI-IO





DAOS

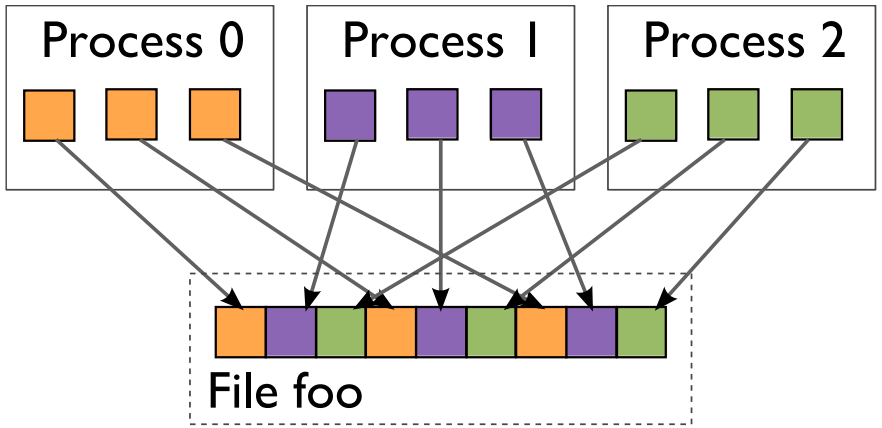




Operation counts

I/O TRANSFORMATIONS

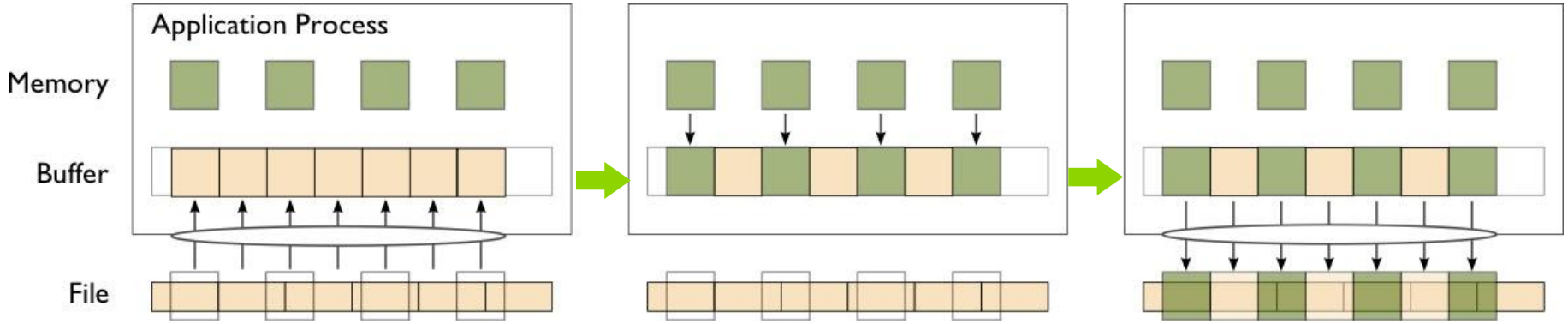
- **Software between the application and the file system performs transformations, primarily to improve performance.**
  - Goals of transformations:
    - Reduce number of operations to PFS (avoiding latency)
    - Avoid lock contention (increasing level of concurrency)
    - Hide number of clients (more 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).

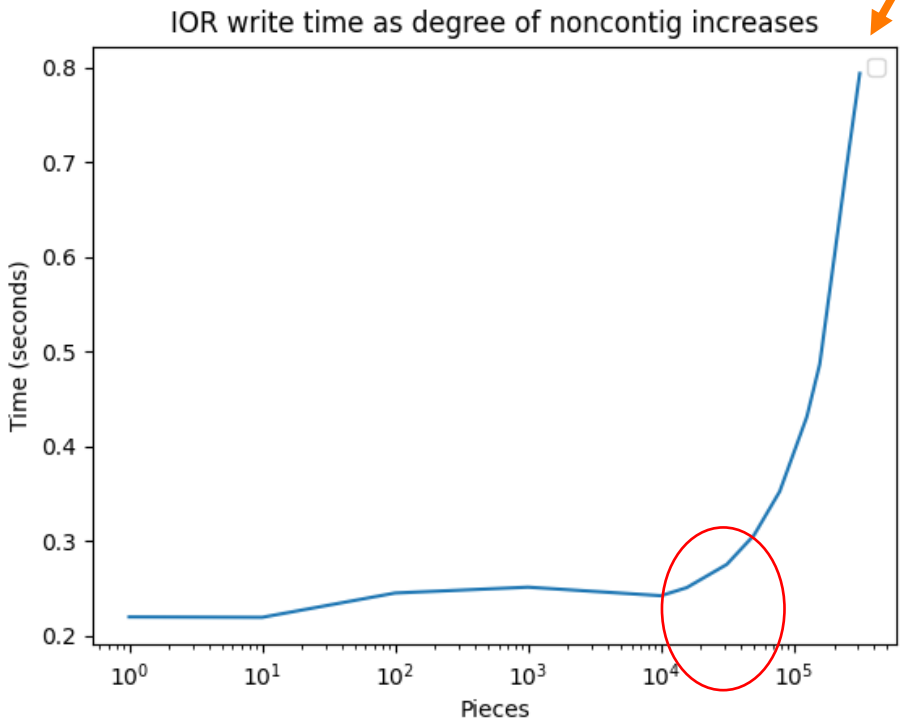
Step 2: Elements to be written to file are replaced in intermediate buffer.

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

DATA SIEVING ALTERNATIVE: SCATTER-GATHER (LIST-IO)

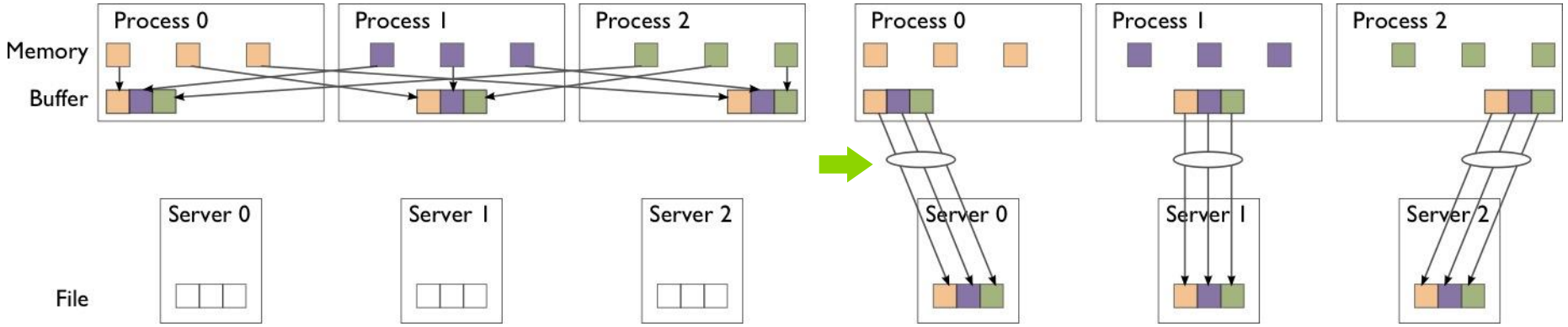
- Same IOR experiment, this time on Aurora's DAOS
- DAOS provides an alternative approach: describe the entire I/O request with a scatter-gather list (d_sg_list_t):
  - `int dfs_write(dfs_t *dfs, dfs_obj_t *obj, d_sg_list_t *sgl, daos_off_t off, daos_event_t *ev);`
- ROMIO driver does this for you
- 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
- 0^{th} phase determines exchange schedule (not shown)



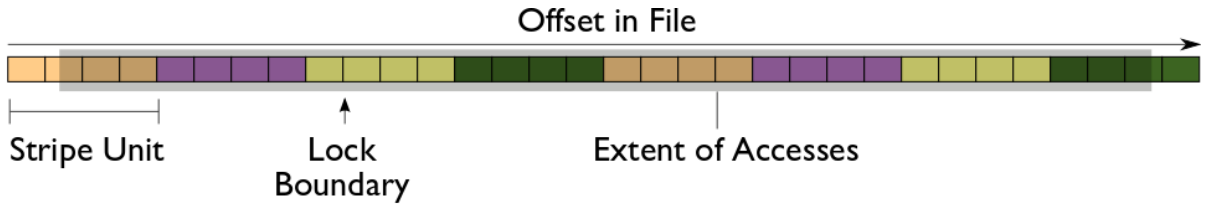
Phase 1: Data are exchanged between processes based on organization of data in file.

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

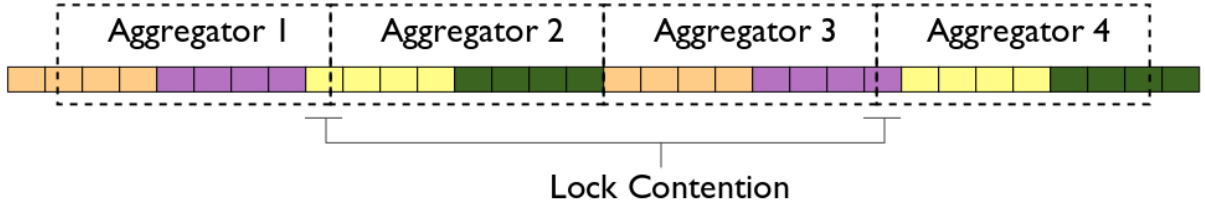
TWO-PHASE I/O ALGORITHMS

(OR, YOU DON'T WANT TO DO THIS YOURSELF…)

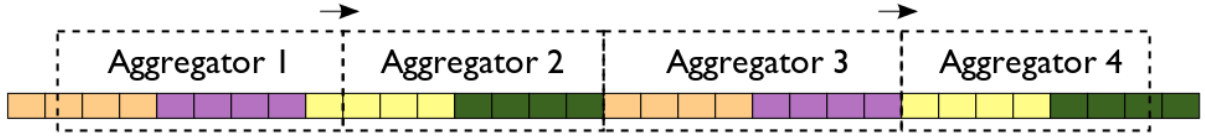
Imagine a collective I/O access using four aggregators to a file striped over four file servers (indicated by colors):



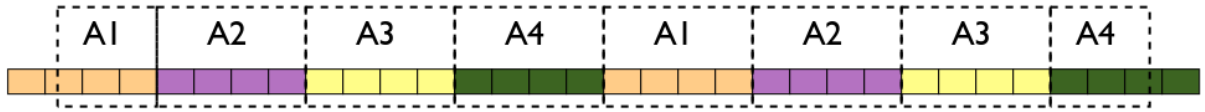
One approach is to evenly divide the region accessed across aggregators.



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

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 an Aurora
- Darshan confirms identical MPI-IO workload
- Different transformations for different file systems
  - OST-oriented vs file block

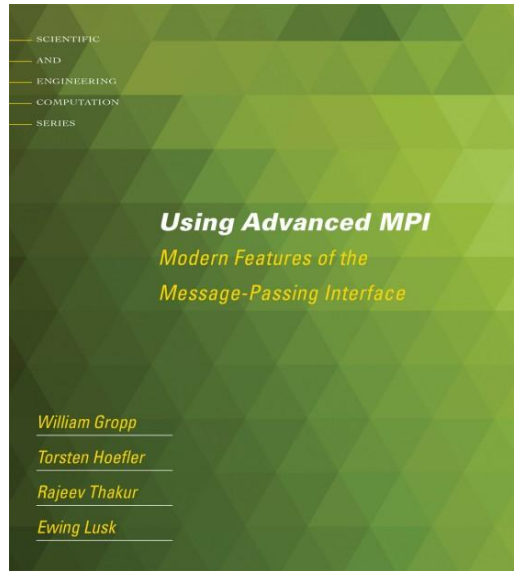
Darshan Counter	Polaris (Lustre)	Ascent (GPFS)	Aurora (DAOS)
MPIIO_ACCESS1_ACCESS	1 000 000	1 000 000	1 000 000
POSIX_WRITES	46	3	
DFS_WRITES			3
POSIX_BYTES_WRITTEN	48000000	48000000	
DFS_BYTES_WRITTEN			48000000
POSIX_SIZE_WRITE_100K_1M	46	0	
POSIX_SIZE_WRITE_10M_100M	0	3	
DFS_SIZE_WRITE_10M_100M			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



PARALLEL-NETCDF

REMINDER: HPC I/O SOFTWARE STACK

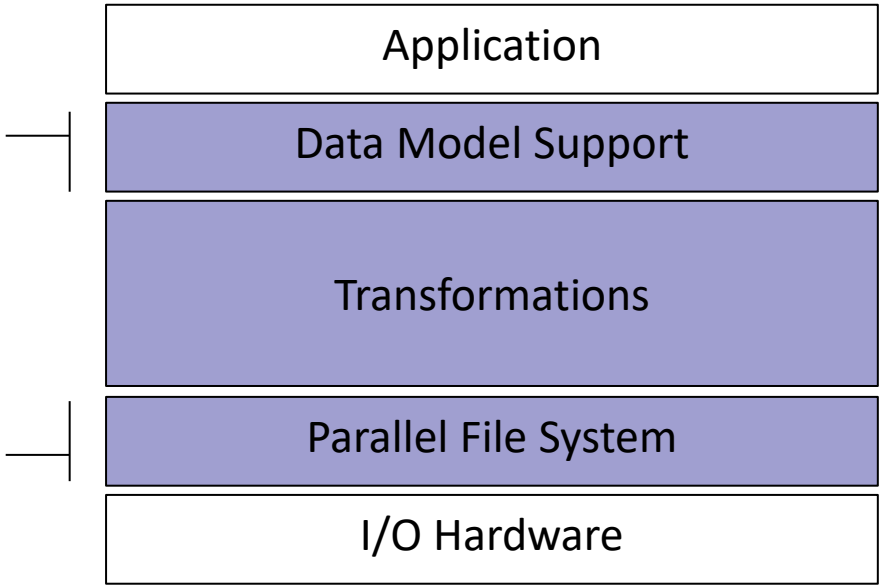
The software used to provide data model support and to transform I/O to better perform on today's I/O systems is often referred to as the *I/O stack*.

Data Model Libraries map application abstractions onto storage abstractions and provide data portability.

HDF5, Parallel netCDF, ADIOS

Parallel file system maintains logical file model and provides efficient access to data.

DAOS, PanFS, GPFS, Lustre



I/O Middleware organizes accesses from many processes, especially those using collective I/O.

MPI-IO

I/O Forwarding transforms I/O from many clients into fewer, larger request; reduces lock contention; and bridges between the HPC system and external storage.

IBM ciod, Cray DVS

PARALLEL NETCDF (PNETCDF)

Based on original “Network Common Data Format” (netCDF) work from Unidata
- Derived from their source code

Data Model:
- Collection of variables in single file
- Typed, multidimensional array variables
- Attributes on file and variables

Features:
- C, Fortran, and F90 interfaces (no python)
- Portable data format (identical to netCDF)
- Noncontiguous I/O in memory using MPI datatypes
- Noncontiguous I/O in file using sub-arrays
- Collective I/O
- Non-blocking I/O

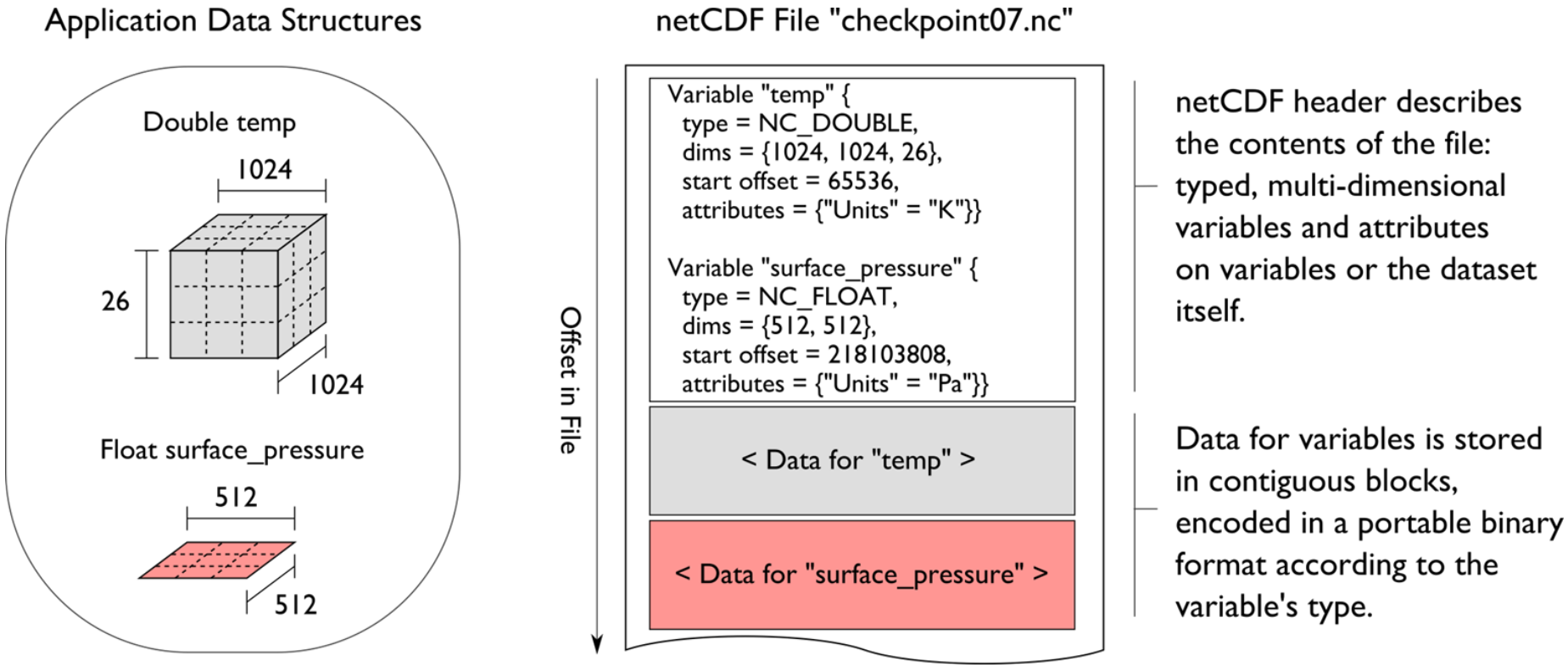
Unrelated to netCDF-4 work

Parallel-NetCDF tutorial:
- https://parallel-netcdf.github.io/wiki/QuickTutorial.html

Interface guide:
- http://cucis.ece.northwestern.edu/projects/PnetCDF/doc/pnetcdf-c/index.html
- ‘man pnetcdf’ on polaris (after loading module)

NETCDF DATA MODEL

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



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

Like MPI-IO example: 2-D array in file, each rank writes 'YDIM' (1) rows

Many details managed by pnetcdf library

- File views
- offsets

Be mindful of define/data mode: call `ncmpi_enddef()`

Library will take care of header i/o for you

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

SOLUTION FRAGMENTS FOR HANDS-ON

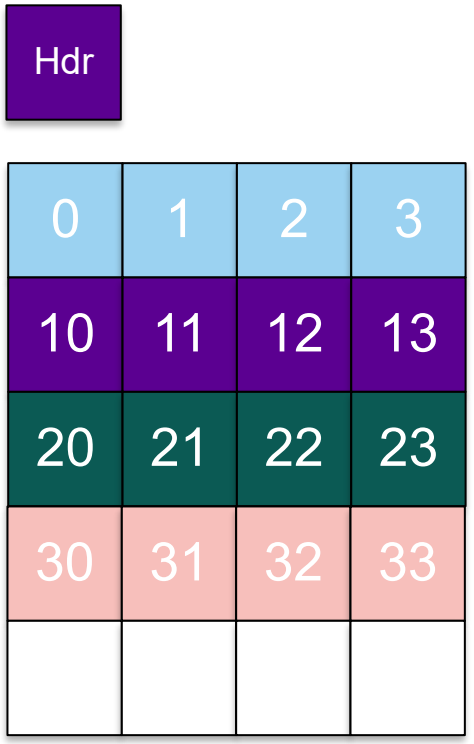
Defining dimension: give name, size; get ID

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

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

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

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



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

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

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

```c
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 */

    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++)  {
        ncmpi_inq_dimlen(ncfile, i, &(dim_sizes[i]) );
    }
    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();
}
```

1

2

3

PNETCDF WRAP-UP

PnetCDF gives us

- Simple, portable, self-describing container for data
- Collective I/O
- Data structures closely mapping to the variables described

If PnetCDF meets application needs, it is likely to give good performance

- Type conversion to portable format does add overhead

Some limits on (old, common CDF-2) file format:

- Fixed-size variable: < 4 GiB
- Per-record size of record variable: < 4 GiB
- 2^{32} -1 records
- Contributed extended file format to relax these limits (CDF-5, released in pnetcdf-1.1.0, November 2009, integrated in Unidata NetCDF-4.4)

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.