ARGONNE ATPESC 2023 EXTREME-SCALE COMPUTING

Higher-level I/O libraries

Parallel-NetCDF and friends

Rob Latham
Research Software Developer, Argonne National Laboratory

Reminder: HPC I/O Software Stack

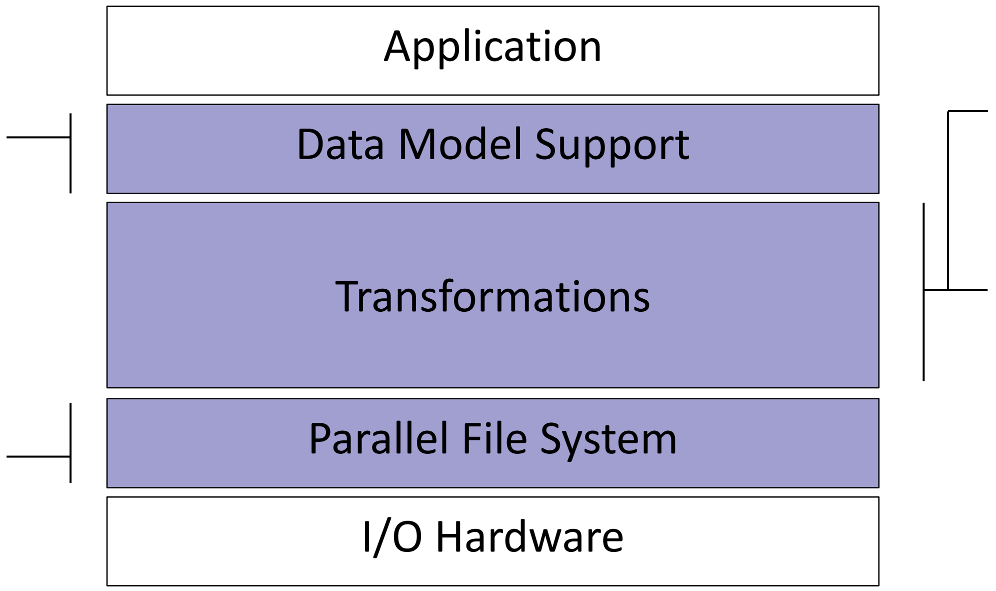
The software used to provide data model support and to transform I/O to better perform on today's I/O systems is often referred to as the *I/O stack.*

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

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

In Practice: The Parallel netCDF Interface and File Format

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

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

Parallel NetCDF (PnetCDF)

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

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

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

Unrelated to netCDF-4 work

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

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

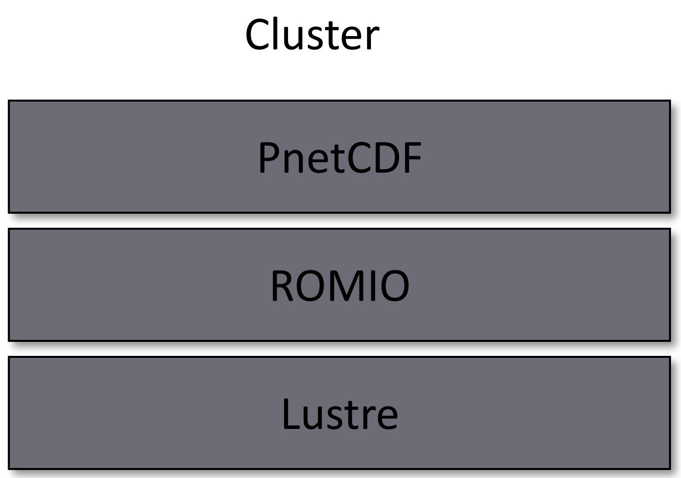
Parallel netCDF (PnetCDF)

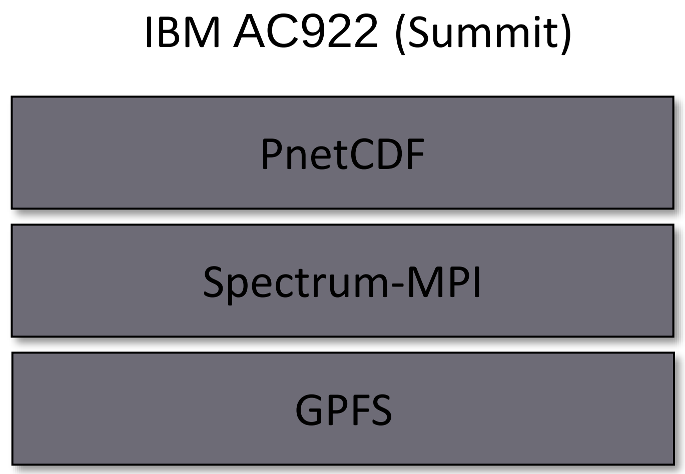
(Serial) netCDF

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

Parallel netCDF

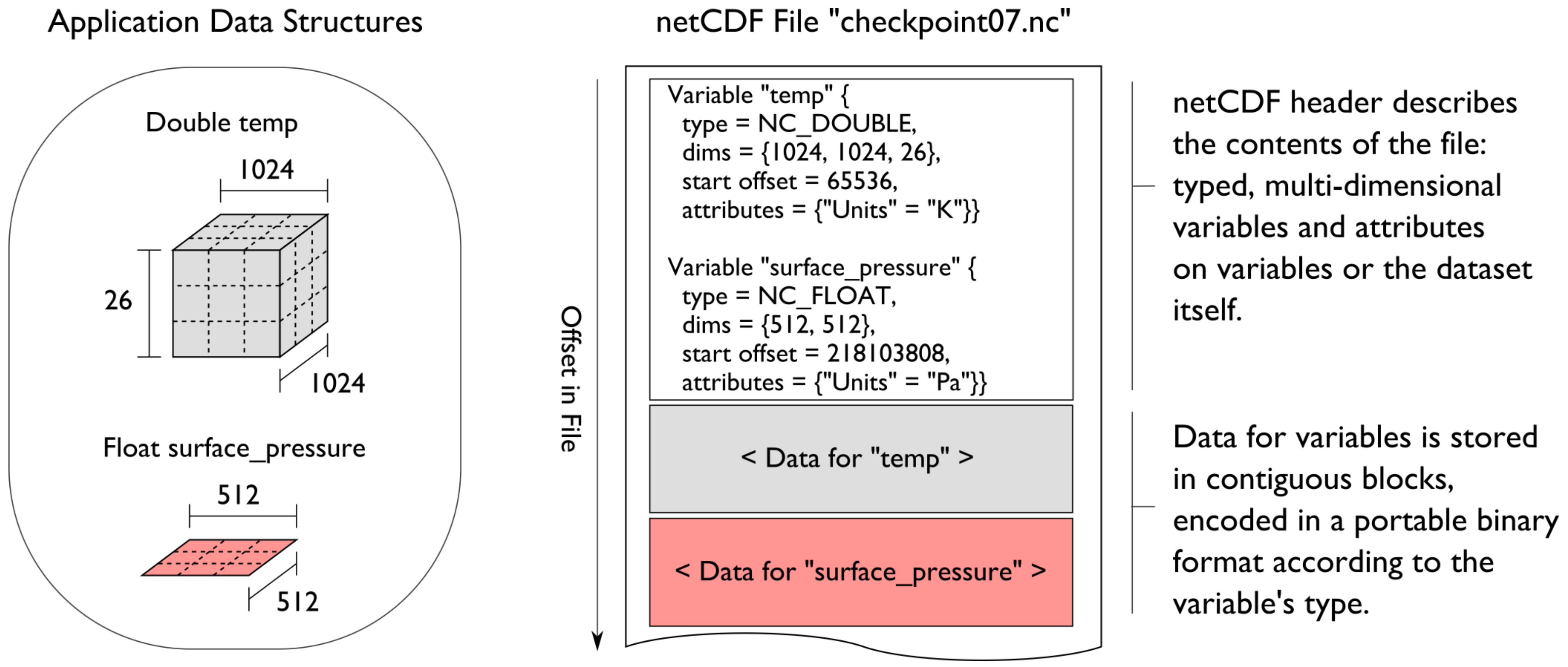
- Very similar API to netCDF
- Tuned for better performance in today's computing environments
- Retains the file format so netCDF and PnetCDF applications can share files
- PnetCDF builds on top of any MPI-IO implementation





netCDF Data Model

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



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

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

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

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

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

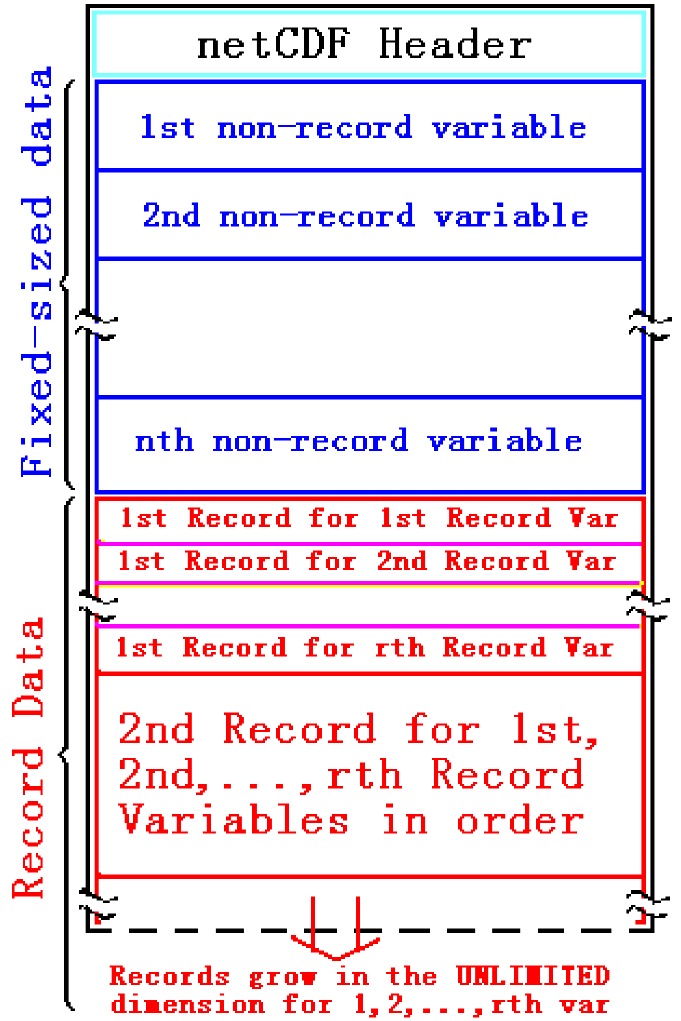
Record Variables in netCDF

Record variables are defined to have a single "unlimited" dimension

- Convenient when a dimension size is unknown at time of variable creation

Record variables are stored after all the other variables in an interleaved format

- Using more than one in a file is likely to result in poor performance due to number of noncontiguous accesses



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

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])) );
```

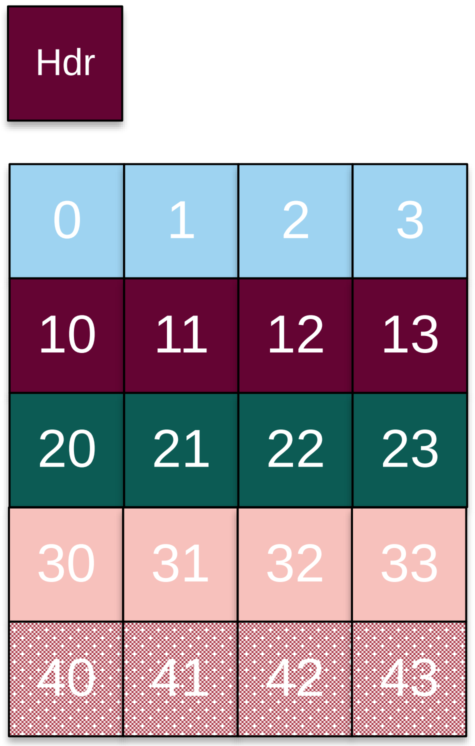
Defining variable: give name, “rank” and dimensions (id); get ID
Attributes: can be placed globally, on variables, dimensions

```
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) );
```



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

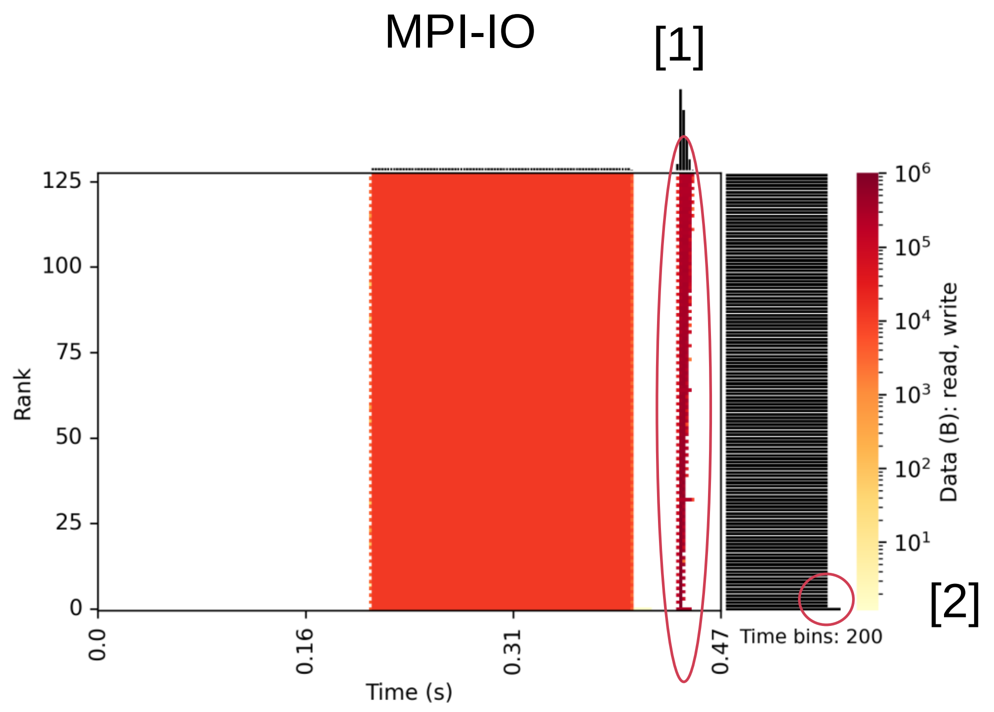
- Process 0 writes header with `MPI_File_write_at`
- `MPI_Bcast` result to others
- Everyone has header data in memory, understands placement of all variables
  - No need for any additional header I/O during data mode!

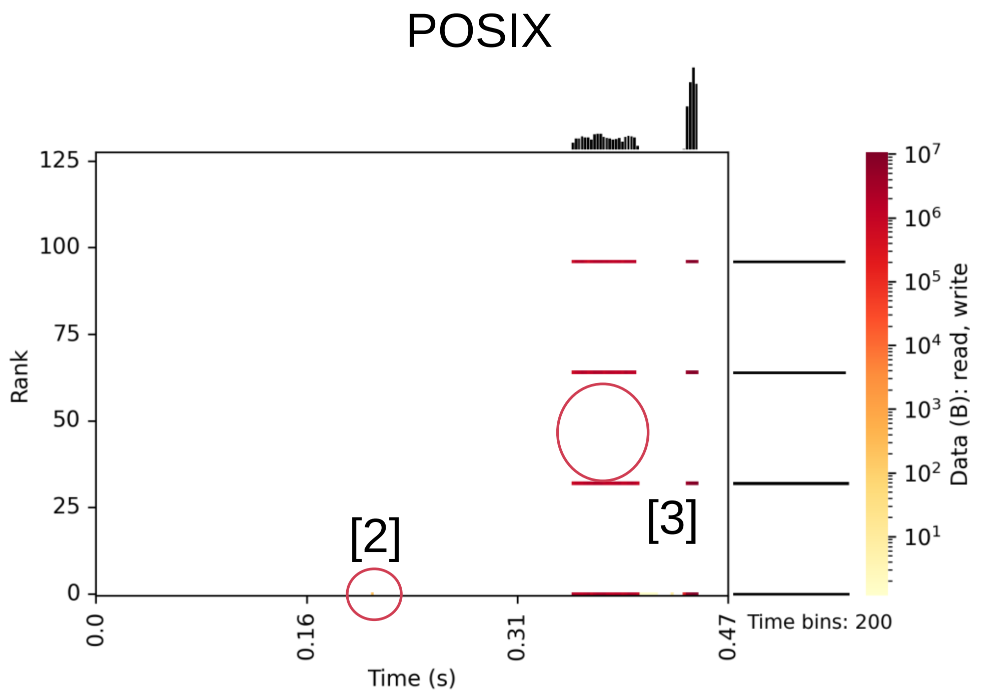
Inside PnetCDF Data Mode

- 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
  - `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

Inside PnetCDF: Darshan heatmap analysis

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





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

Hands-on continued

Take a look at the Darshan report for your job.

Account for the number of MPI-IO and POSIX write operations

HACC: understanding cosmos via simulation

"Cosmology = Physics + Simulation " (Salman Habib)

Sky surveys collecting massive amounts of data

- (~100 PB)

Understanding of these massive datasets rests on modeling distribution of cosmic entities

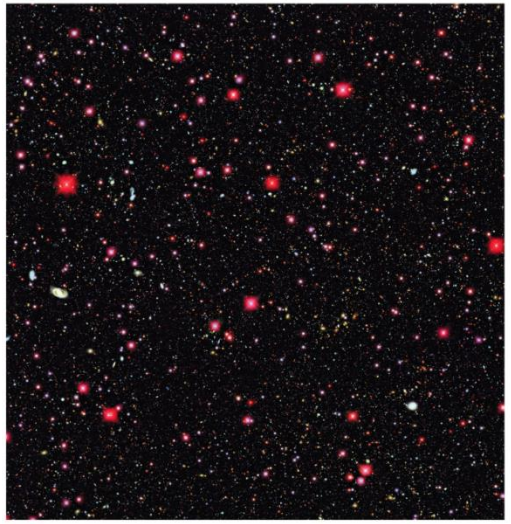
Seed simulations with initial conditions

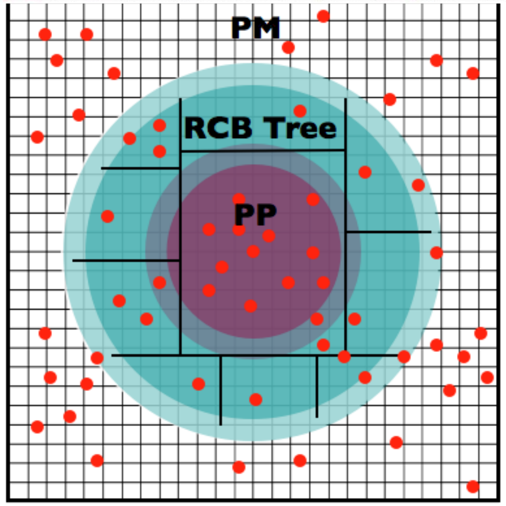
Run for 13 billion (simulated) years

Comparison with observed data validates physics model.

I/O challenges:

- Checkpointing
- analysis





Parallel NetCDF Particle Output

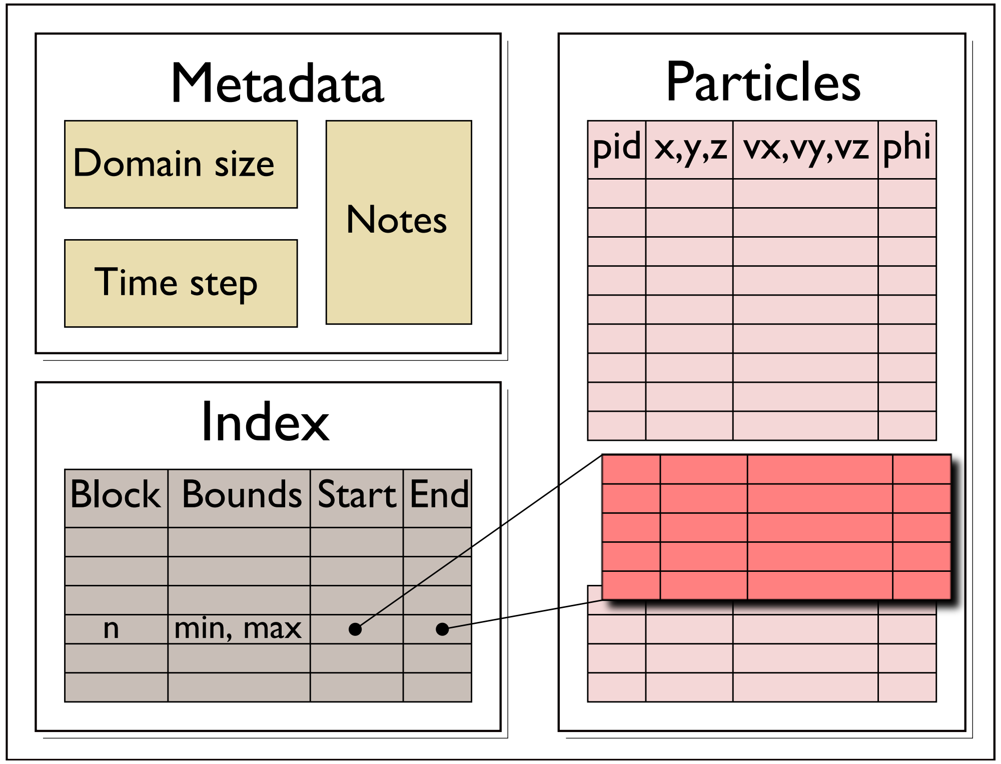
Metadata, index, and particle data

Self-describing portable format

Can be read with different number of processes than written

Can be queried for particles within spatial bounds

Collaboration with Northwestern and Argonne: research demonstration



File schema for analysis output enables spatial queries of particle data in a high-level self-describing format.

HACC particles with pnetcdf: metadata (1/2)

```
/* class constructor creates dataset */
IO::IO(int mode, char *filename, MPI_Comm comm) {
      ncmpi_create(comm, filename, NC_64BIT_DATA,
                        MPI_INFO_NULL, &ncfile);
}


/* describe simulation metadata, not pnetcdf metadata */
void IO::WriteMetadata(char *notes, float *block_size,
      float *global_min, int *num_blocks,
      int first_time_step, int last_time_step,
      int this_time_step, int num_secondary_keys,
      char **secondary_keys) {
  ncmpi_put_att_text(ncfile, NC_GLOBAL, "notes",
      strlen(notes), notes);
  ncmpi_put_att_float(ncfile, NC_GLOBAL, "global_min_z",
       NC_FLOAT, 1,&global_min[2]);
}
```

HACC particles with pnetcdf: metadata (2/2)

```
void IO::DefineDims() {
  ncmpi_def_dim(ncfile, "KeyIndex", key_index,        &dim_keyindex);
  char str_attribute[100 =
    "num_blocks_x * num_blocks_y * num_blocks_z *     num_kys";

  /* variable with no dimensions: “scalar” */
  ncmpi_def_var(ncfile, "KeyIndex", NC_INT, 0,
       NULL, &var_keyindex);
  ncmpi_put_att_text(ncfile, var_keyindex, "Key_Index",
                   strlen(str_attribute), str_attribute);
  /* pnetcdf knows shape and type, but application must
      annotate with units */
  strcpy(unit, “km/s”);
  ncmpi_def_var(ncfile, “Velocity”, NC_FLOAT,
       ndims, dimpids, &var_velid);
  ncmpi_put_att_text(ncfile, var_velid, “unit_of_velocity”, strlen(unit),
unit);
}
```

HACC particles with pnetcdf: data

```
void IO::WriteData(int num_particles, float *xx, float *yy, float *zz,
                   float *vx, float *vy, float *vz,
                   float *phi, int64_t *pid, float *mins,
                   float *maxs) {
 // calculate total number of particles and individual array offsets
  nParticles = num_particles; // typecast to MPI_Offset
  myOffset   = 0; // particle offset of this process
  MPI_Exscan(&nParticles, &myOffset, 1, MPI_OFFSET, MPI_SUM, comm);
  MPI_Allreduce(MPI_IN_PLACE, &nParticles, 1, MPI_OFFSET,
        MPI_SUM, comm);

  start[0] = myOffset;  start[1] = 0;
  count[0] = num_particles;  count[1] = 3;  /* ZYX dimensions */

  // write "Velocity" in parallel, partitioned
  // along dimension nParticles
  // "Velocity" is of size nParticles x nDimensions
  //  data_vel array set up based on method parameters
  ncmpi_put_vara_float_all(ncfile, var_velid, start, count,
                                 &data_vel[0][0]);

}
```

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();
    }
```

HANDS-ON: reading with pnetcdf

Similar to MPI-IO reader: just read one row

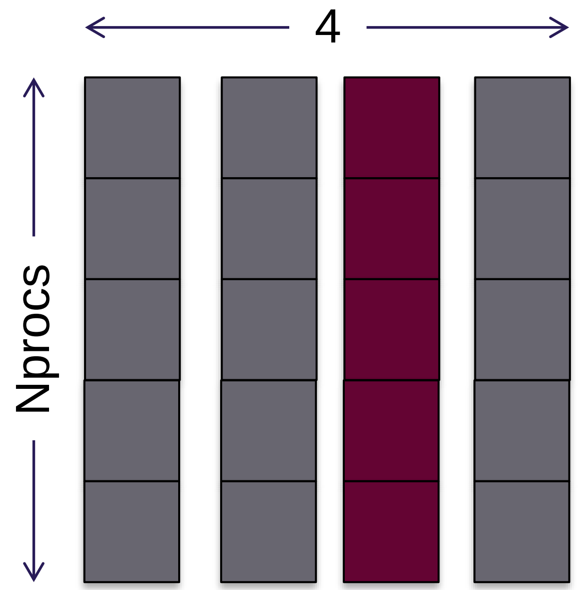
Operate on netcdf arrays, not MPI datatypes

Shortcut: can rely on “convention”

- One could know nothing about file as in previous slide
- In our case we know there’s a variable called “array” (id of 0) and an attribute called “iteration”

Routines you’ll need:

- `ncmpi_inq_dim` to turn dimension id to dimension length
- `ncmpi_get_att_int` to read “iteration” attribute
- `ncmpi_get_vara_int_all` to read column of array



Solution fragments: reading with pnetcdf

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

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

The "Iteration" attribute

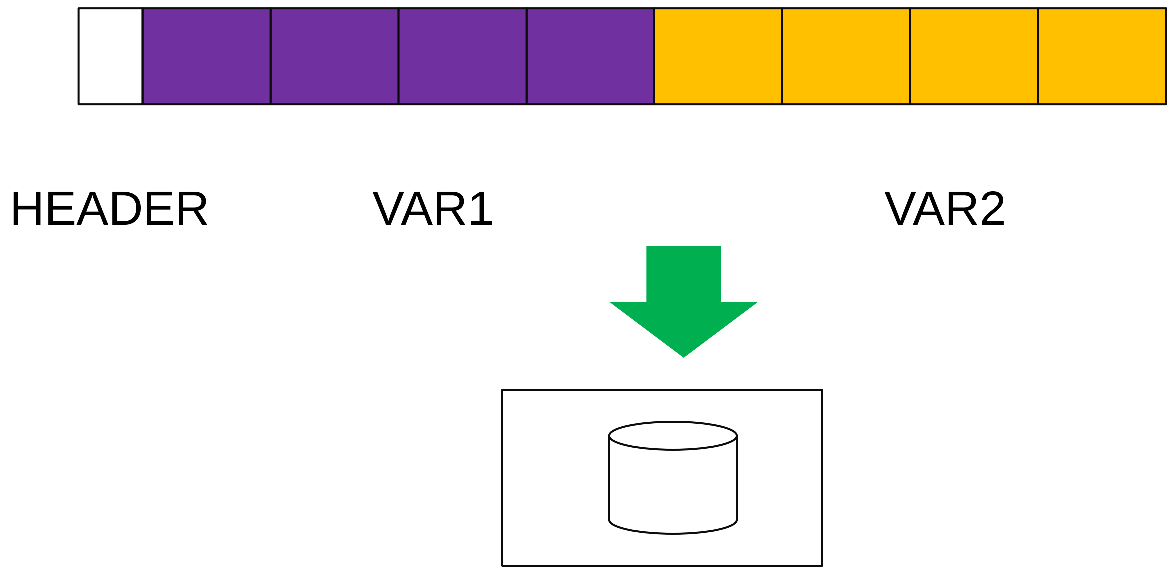
```
NC_CHECK(ncmpi_get_att_int(ncfile, 0, "iteration", &iterations));
```

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

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

Parallel-NetCDF write-combining optimization

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



netCDF variables laid out contiguously

Applications typically store data in separate variables

- temperature(lat, long, elevation)
- Velocity_x(x, y, z, timestep)

Operations posted independently, completed collectively

- Defer, coalesce synchronization
- Increase average request size

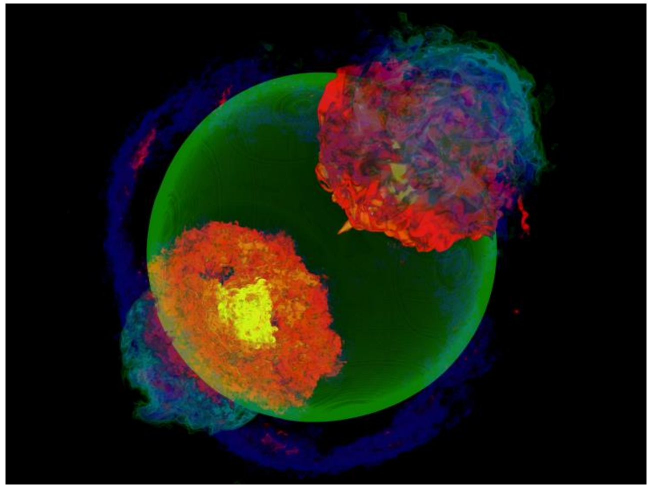
Example: FLASH Astrophysics

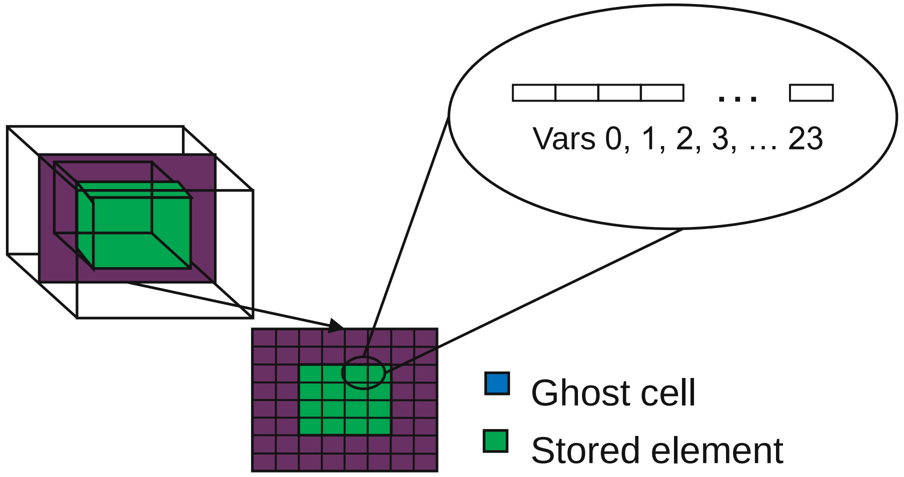
FLASH is an astrophysics code for studying events such as supernovae

- Adaptive-mesh hydrodynamics
- Scales to 1000s of processors
- MPI for communication

Frequently checkpoints:

- Large blocks of typed variables from all processes
- Portable format
- Canonical ordering (different than in memory)
- Skipping ghost cells





FLASH Astrophysics and the write-combining optimization

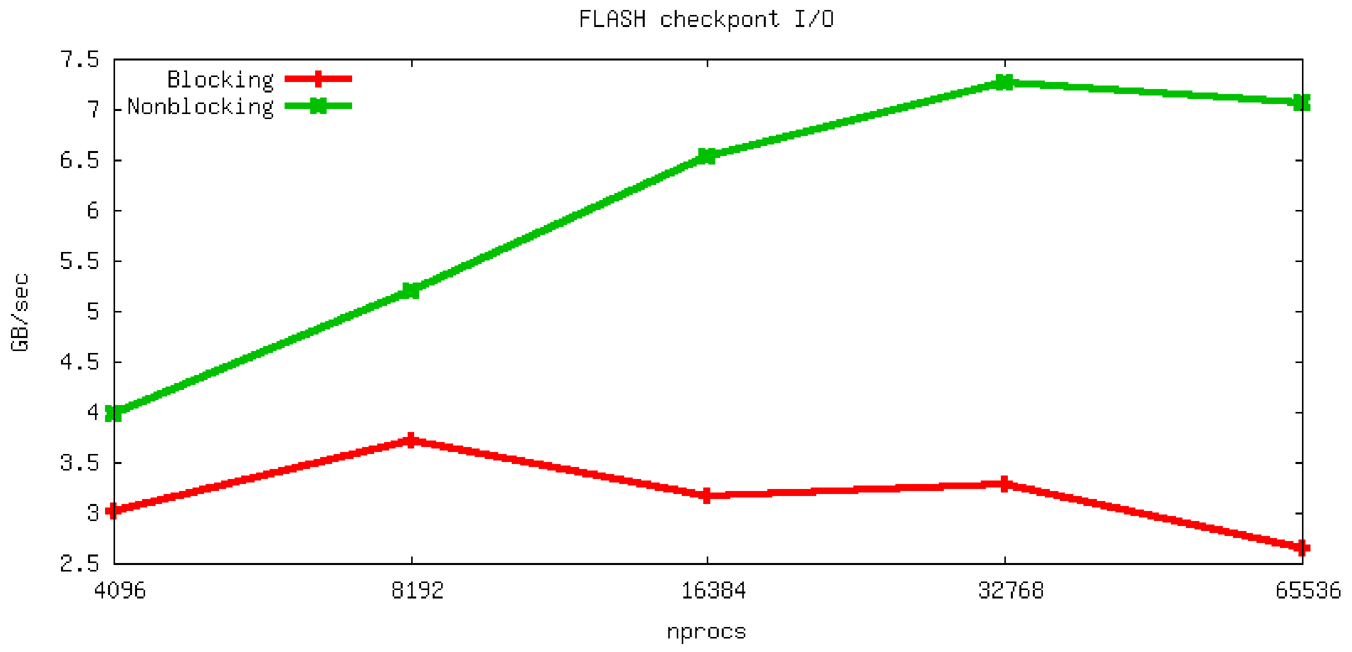
FLASH writes one variable at a time

Could combine all 4D variables (temperature, pressure, etc) into one 5D variable

- Altered file format (conventions) requires updating entire analysis toolchain

Write-combining provides improved performance with same file conventions

- Larger requests, less synchronization.



HANDS-ON: pnetcdf write-combining

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

Solution fragments for write-combining

Defining a second variable

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

The non-blocking interface: looks a lot like MPI

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

Waiting for I/O to complete

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

Hands-on continued

Look at the darshan output. Compare to darshan output for single-variable writing or reading

- Results on polaris surprised me: vendor might know something I don’t
  - Maybe some kind of small-io optimization?

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.