Data Models and Libraries: Application-oriented I/O

Presented to
ATPESC 2017 Participants

Rob Latham Phil Carns
Math and Computer Science Division

Argonne National Laboratory

Q Center, St. Charles, IL (USA)
Date 08/04/2017

Reminder: HPC I/O Software Stack

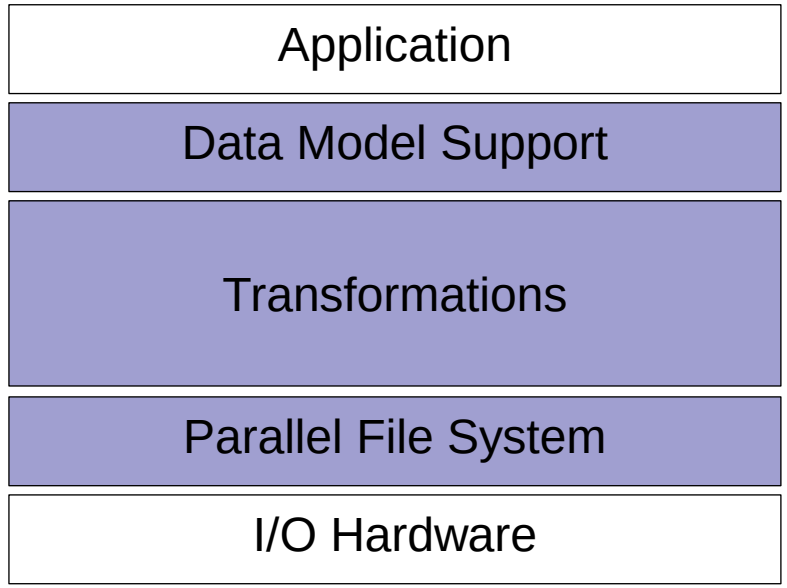
The software used to provide data model support and to transform I/O to better perform on today's I/O systems is often referred to as the *I/O stack.*

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.

PVFS, PanFS, GPFS, Lustre



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

MPI-IO, GLEAN, PLFS

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, IOFSL, 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

How It Works: The Parallel netCDF Interface and File Format

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

www.mcs.anl.gov/parallel-netcdf

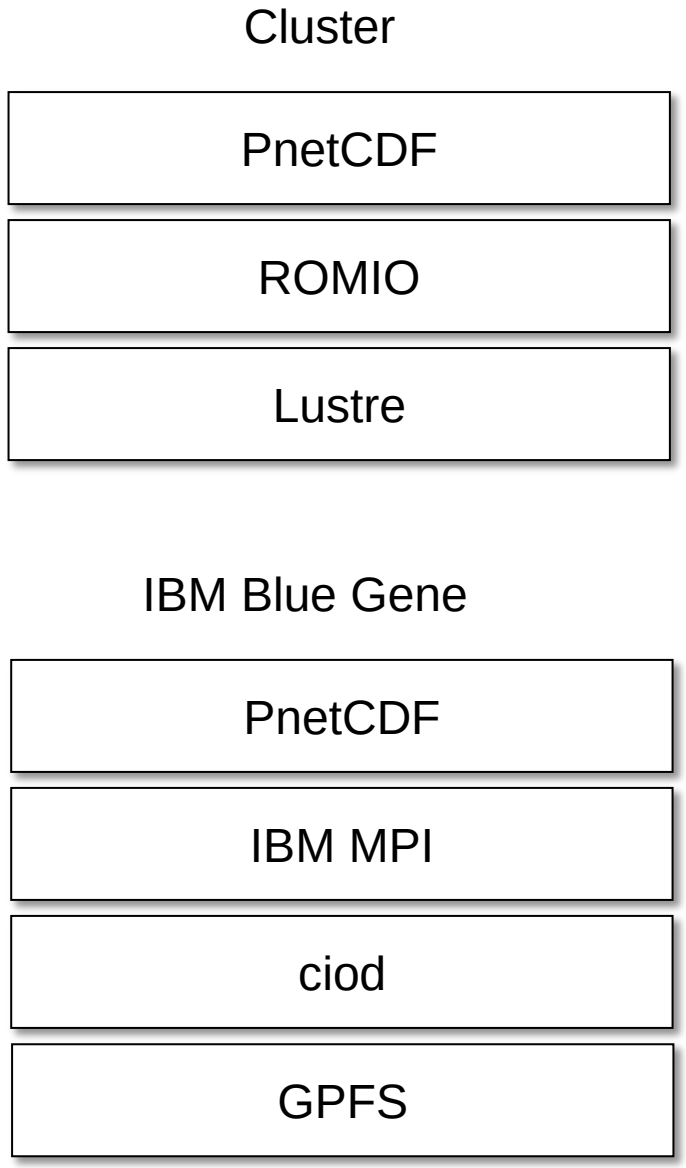
Parallel NetCDF (PnetCDF)

- 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

Parallel netCDF (PnetCDF)

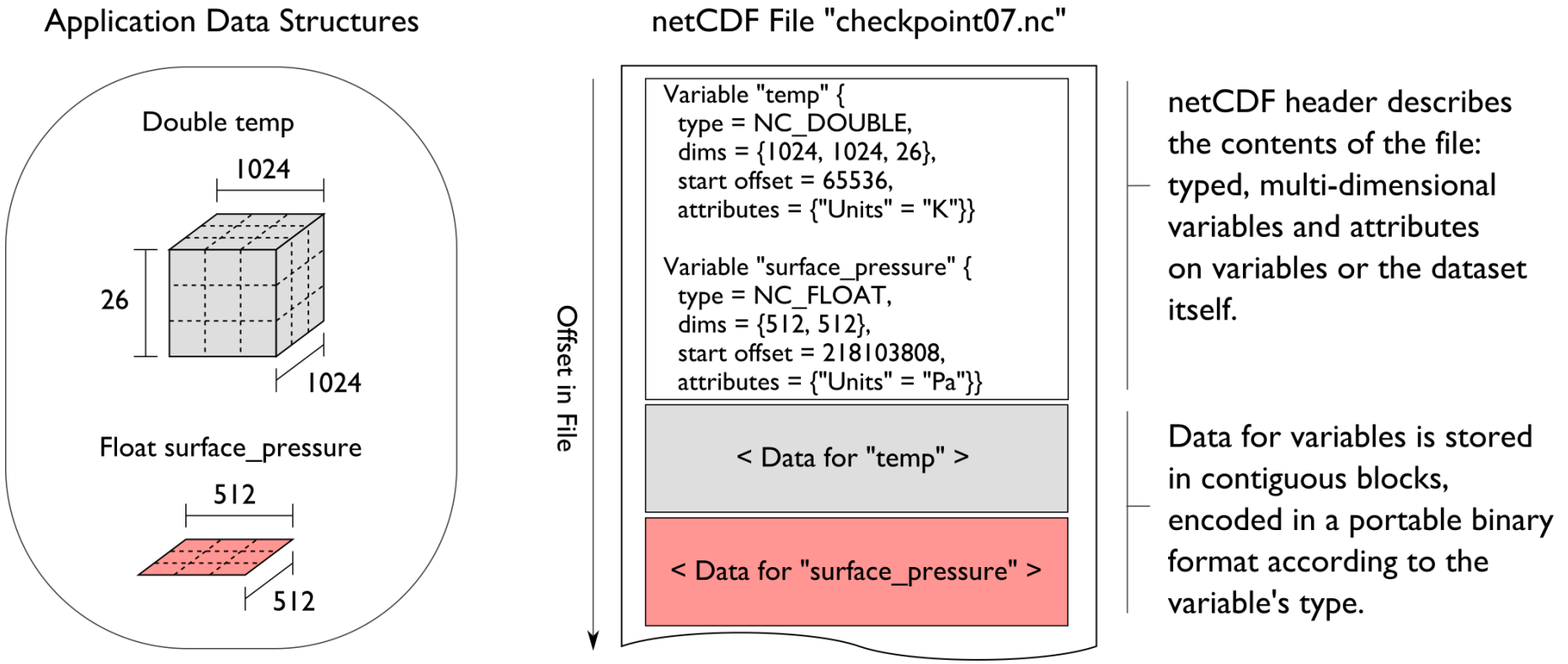
- (Serial) netCDF
  - API for accessing multi-dimensional data sets
  - Portable file format
  - Popular in both fusion and climate communities

- 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.

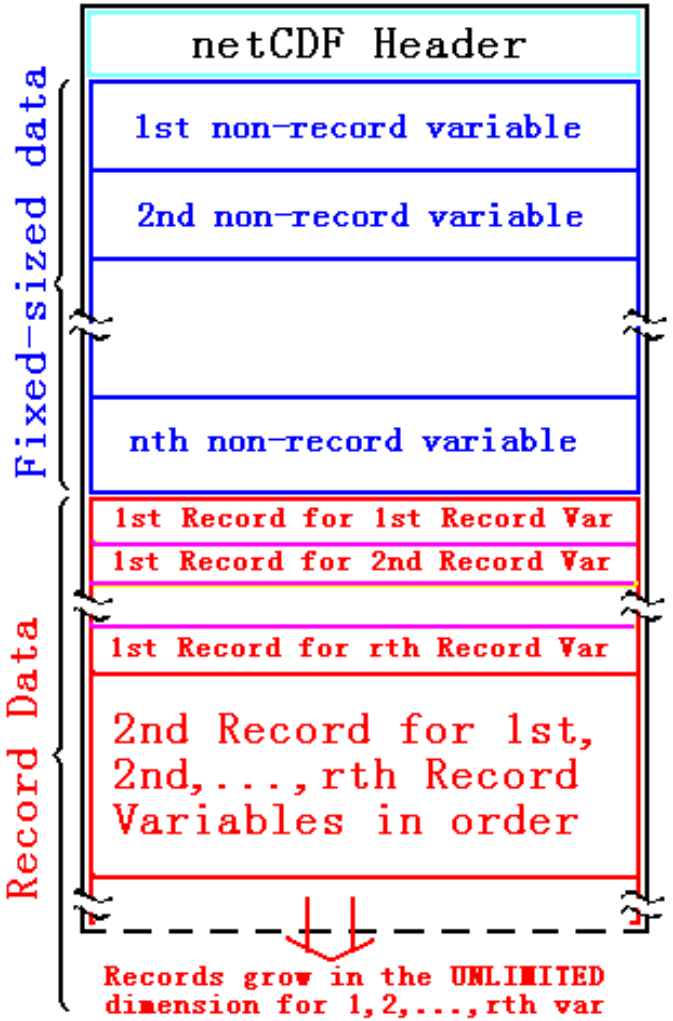


Argonne NATIONAL LABORATORY

ECP EXASCALE COMPUTING PROJECT

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



Argonne NATIONAL LABORATORY

ECP EXASCALE COMPUTING PROJECT

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)

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!

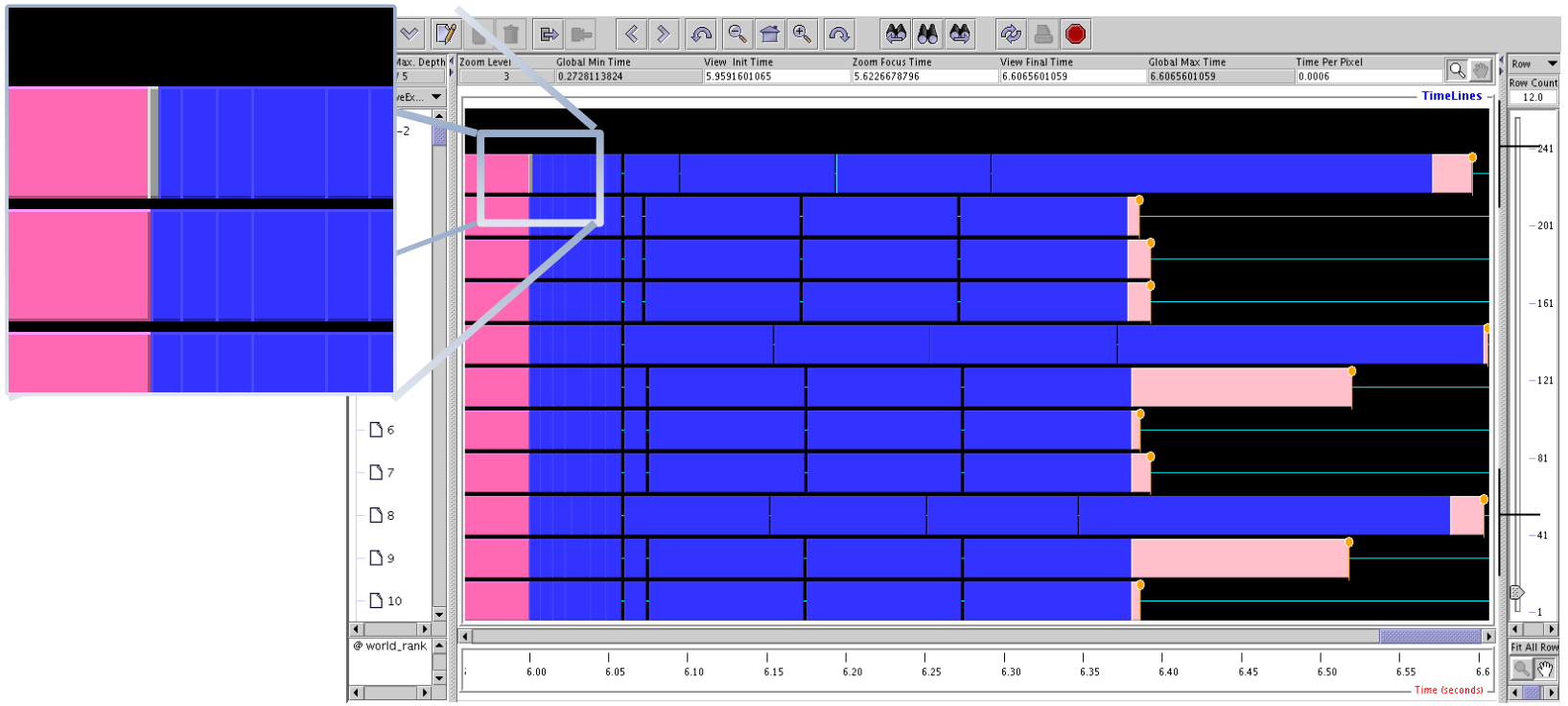
Inside PnetCDF Data Mode

- 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

Inside Parallel netCDF: TIME-line view

1: Rank 0 write header (independent I/O)

3: Collectively write 4 variables



I/O Aggr

2: Collectively write app grid, AMR data

4: Close file

File open

Indep. write

Collective write

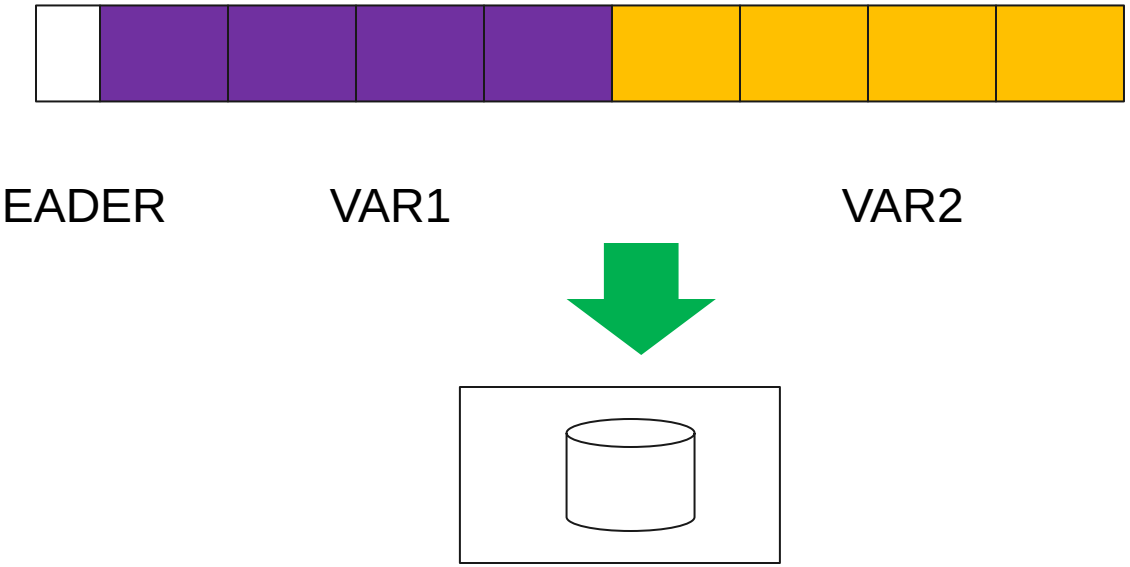
File close

Parallel-NetCDF write-combining optimization

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

```
ncmpi_iput_vara(ncfile, varid2,

ncmpi_wait_all(ncfile, 2,
```



- 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

PnetCDF Life Checkpoint/Restart Code Walkthrough

- Stores matrix as a two-dimensional array of integers
  - Same canonical ordering as in MPI-IO version
- Iteration number stored as an attribute
- Note: A naïve reader **will** know how to read this

integer iter

Iteration

integer “matrix” [rows][cols]

Global Matrix

See mlife-io-pnetcdf.c pp. 1-5 for code example.

```
 1: /* SLIDE: PnetCDF Life Checkpoint Code Walkthrough */
 2: /* -*- Mode: C; c-basic-offset:4 ; -*- */
 3: /*
 4:  *  (C) 2004 by University of Chicago.
 5:  *      See COPYRIGHT in top-level directory.
 6:  */
 7: #include <stdio.h>
 8: #include <stdlib.h>
 9: #include <mpi.h>
10: #include <pnetcdf.h>
11: #include "mlife-io.h"
12:
13: /* Parallel netCDF implementation of checkpoint and restart for
14:  * MPI Life
15:  *
16:  * Data stored in a 2D variable called "matrix" in matrix order,
17:  * with dimensions "row" and "col".
18:  *
19:  * Each checkpoint is stored in its own file.
20:  */
21: static MPI_Comm mlifeio_comm = MPI_COMM_NULL;
22:
23: int MLIFEIO_Init(MPI_Comm comm)
24: {
25:     int err;
26:     err = MPI_Comm_dup(comm, &mlifeio_comm);
27:     return err;
28: }
29:
```

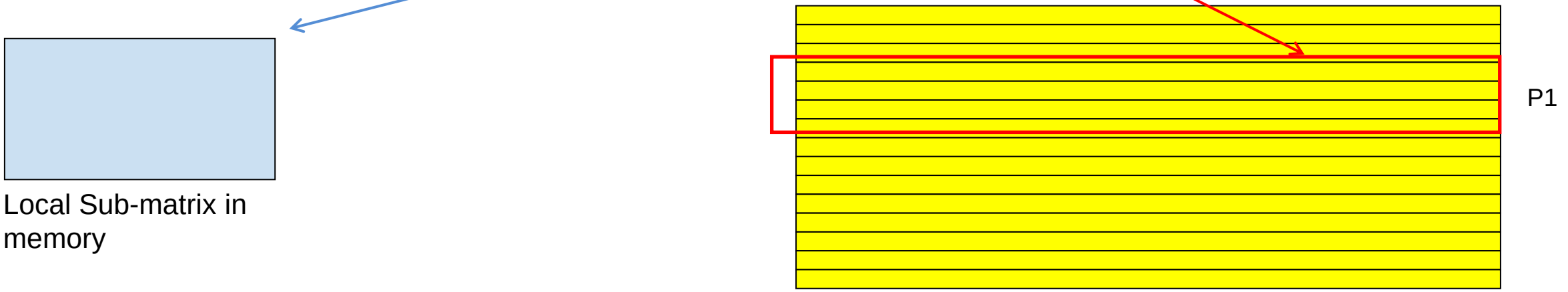
```
30: /* SLIDE: PnetCDF Life Checkpoint Code Walkthrough */
31: int MLIFEIO_Finalize(void)
32: {
33:     int err;
34:
35:     err = MPI_Comm_free(&mlifeio_comm);
36:
37:     return err;
38: }
39:
40: int MLIFEIO_Can_restart(void)
41: {
42:     return 1;
43: }
44:
```

```
45: /* SLIDE: PnetCDF Life Checkpoint Code Walkthrough */
46: int MLIFEIO_Checkpoint(char *prefix, int **matrix, int rows,
47:                         int cols, int iter, MPI_Info info)
48: {
49:     int err;
50:     int cmode = 0;
51:     int rank, nprocs;
52:     int myrows, myoffset;
53:
54:     int ncid, varid, coldim, rowdim, dims[2];
55:     MPI_Offset start[2];
56:     MPI_Offset count[2];
57:     int i, j, *buf;
58:     char filename[64];
59:
60:     MPI_Comm_size(mlifeio_comm, &nprocs);
61:     MPI_Comm_rank(mlifeio_comm, &rank);
62:
63:     myrows    = MLIFE_myrows(rows, rank, nprocs);
64:     myoffset = MLIFE_myrowoffset(rows, rank, nprocs);
65:
66:     snprintf(filename, 63, "%s-%d.nc", prefix, iter);
67:
68:     err = ncmpi_create(mlifeio_comm, filename, cmode, info, &ncid);
69:     if (err != 0) {
70:         fprintf(stderr, "Error opening %s.\n", filename);
71:         return MPI_ERR_IO;
72:     }
73:
```

Describing Subarray Access in PnetCDF

- PnetCDF provides calls for reading/writing subarrays in a single (collective) call:

```
ncmpi_put_vara_all(ncid,
                   varid,
                   start[], count[],
                   buf, count,
                   datatype)
```



- Define mode vs data mode
- Can describe anything in memory, but constrained to multidimensional arrays in storage

File: mlife-io-pnetcdf.c

```
 74: /* SLIDE: PnetCDF Life Checkpoint Code Walkthrough */
 75:     ncmpi_def_dim(ncid, "col", cols, &coldim);
 76:     ncmpi_def_dim(ncid, "row", rows, &rowdim);
 77:     dims[0] = coldim;
 78:     dims[1] = rowdim;
 79:     ncmpi_def_var(ncid, "matrix", NC_INT, 2, dims, &varid);
 80:
 81:     /* store iteration as global attribute */
 82:     ncmpi_put_att_int(ncid, NC_GLOBAL, "iter", NC_INT, 1, &iter);
 83:
 84:     ncmpi_enddef(ncid);
 85:
 86:     start[0] = 0; /* col start */
 87:     start[1] = myoffset; /* row start */
 88:     count[0] = cols;
 89:     count[1] = myrows;
 90:
 91:     MLIFEIO_Type_create_rowblk(matrix, myrows, cols, &type);
 92:     MPI_Type_commit(&type);
 93:
 94:     ncmpi_put_vara_all(ncid, varid, start, count, MPI_BOTTOM, 1,
 95:                        type);
 96:
 97:     MPI_Type_free(&type);
 98:
 99:     ncmpi_close(ncid);
100:     return MPI_SUCCESS;
101: }
102:
```

```
103: /* SLIDE: PnetCDF Life Checkpoint Code Walkthrough */
104: int MLIFEIO_Restart(char *prefix, int **matrix, int rows,
105:                     int cols, int iter, MPI_Info info)
106: {
107:     int err = MPI_SUCCESS;
108:     int rank, nprocs;
109:     int myrows, myoffset;
110:     int flag;
111:
112:     int cmode = 0;
113:     int ncid, varid, dims[2];
114:     MPI_Offset start[2];
115:     MPI_Offset count[2];
116:     MPI_Offset coldimsz, rowdimsz;
117:     int i, j, *buf;
118:     char filename[64];
119:
120:     MPI_Comm_size(mlifeio_comm, &nprocs);
121:     MPI_Comm_rank(mlifeio_comm, &rank);
122:
123:     myrows   = MLIFE_myrows(rows, rank, nprocs);
124:     myoffset = MLIFE_myrowoffset(rows, rank, nprocs);
125:
126:     snprintf(filename, 63, "%s-%d.nc", prefix, iter);
127:     err = ncmpi_open(mlifeio_comm, filename, cmode, info, &ncid);
128:     if (err != 0) {
129:         fprintf(stderr, "Error opening %s.\n", filename);
130:         return MPI_ERR_IO;
131:     }
```

Discovering Variable Dimensions

- Because netCDF is self-describing, applications can inquire about data in netCDF files:

```
err = ncmpi_inq_dimlen(ncid,
                       dims[0],
                       &coldimsz);
```

- Allows us to discover the dimensions of our matrix at restart time

See mlife-io-pnetcdf.c pp. 6-7 for code example.

```
132: /* SLIDE: Discovering Variable Dimensions */
133:     err = ncmpi_inq_varid(ncid, "matrix", &varid);
134:     if (err != 0) {
135:         return MPI_ERR_IO;
136:     }
137:
138:     /* verify that dimensions in file are same as input row/col */
139:     err = ncmpi_inq_vardimid(ncid, varid, dims);
140:     if (err != 0) {
141:         return MPI_ERR_IO;
142:     }
143:
144:     err = ncmpi_inq_dimlen(ncid, dims[0], &coldimsz);
145:     if (coldimsz != cols) {
146:         fprintf(stderr, "cols does not match\n");
147:         return MPI_ERR_IO;
148:     }
149:
150:     err = ncmpi_inq_dimlen(ncid, dims[1], &rowdimsz);
151:     if (rowdimsz != rows) {
152:         fprintf(stderr, "rows does not match\n");
153:         return MPI_ERR_IO;
154:     }
155:
```

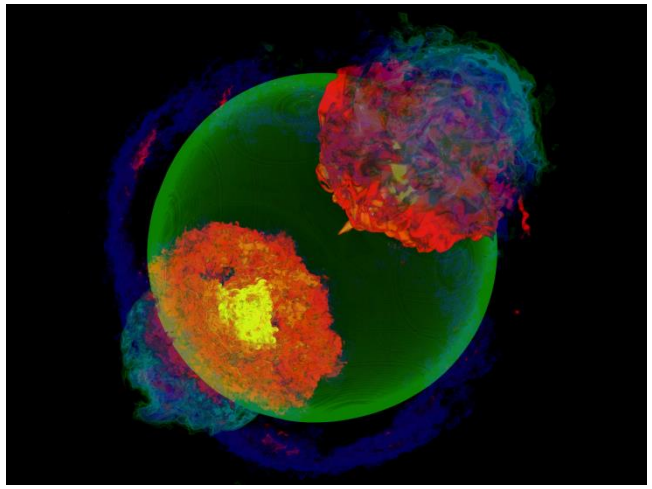
```
156:  /* SLIDE: Discovering Variable Dimensions */
157:      buf = (int *) malloc(myrows * cols * sizeof(int));
158:      flag = (buf == NULL);
159:      /* See if any process failed to allocate memory */
160:      MPI_Allreduce(MPI_IN_PLACE, &flag, 1, MPI_INT, MPI_LOR,
161:                    mlifeio_comm);
162:      if (flag) {
163:          return MPI_ERR_IO;
164:      }
165:
166:      start[0] = 0; /* col start */
167:      start[1] = myoffset; /* row start */
168:      count[0] = cols;
169:      count[1] = myrows;
170:      ncmpi_get_vara_int_all(ncid, varid, start, count, buf);
171:
172:      for (i=0; i < myrows; i++) {
173:          for (j=0; j < cols; j++) {
174:              matrix[i+1][j] = buf[(i*cols) + j];
175:          }
176:      }
177:
178:      free(buf);
179:
180:      return MPI_SUCCESS;
181: }
```

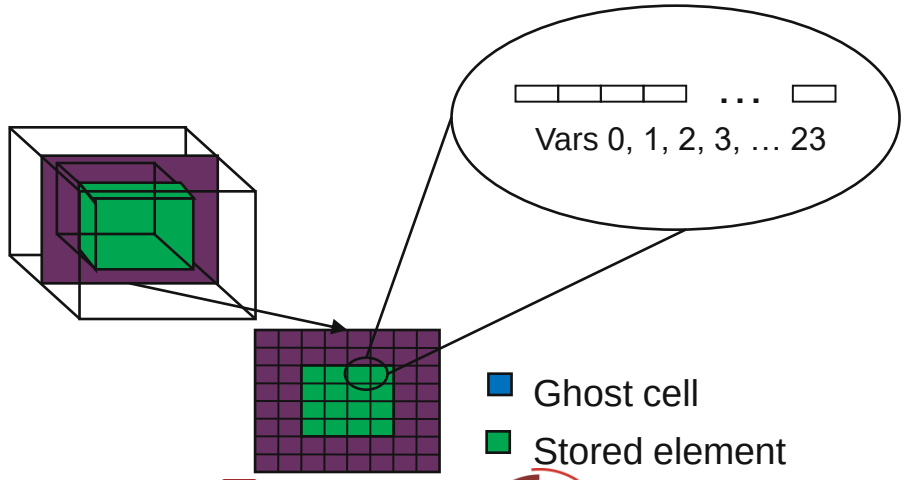
Takeaway from PnetCDF Game of Life Example

- PnetCDF abstracts away the file system model, giving us something closer to (many) domain models
  - Arrays
  - Types
  - Attributes
- Captures metadata for us (e.g., rows, columns, types) and allows us to programmatically explore datasets
- Uses MPI-IO underneath, takes advantage of data sieving and two-phase I/O when possible

Example: FLASH Astrophysics

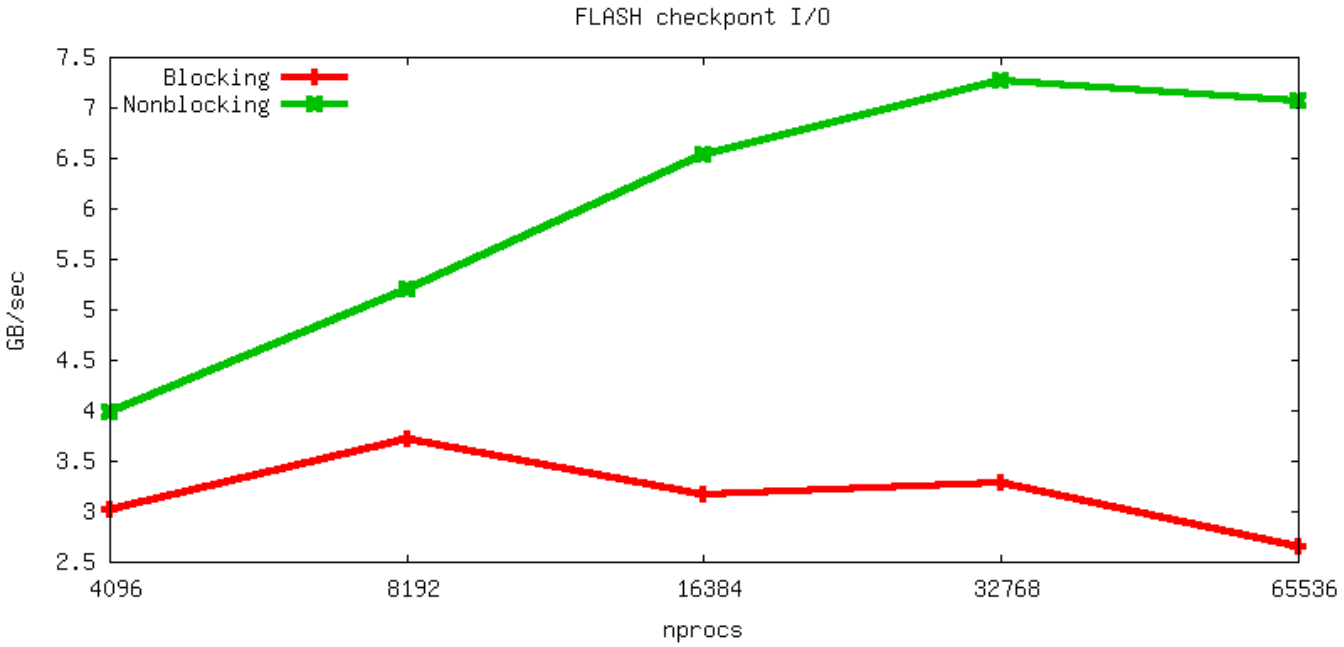
- FLASH is an astrophysics code for studying events such as supernovae
  - Adaptive-mesh hydrodynamics
  - Scales to 1000s of processors
  - MPI for communication
- Frequently checkpoints:
  - Large blocks of typed variables from all processes
  - Portable format
  - Canonical ordering (different than 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.

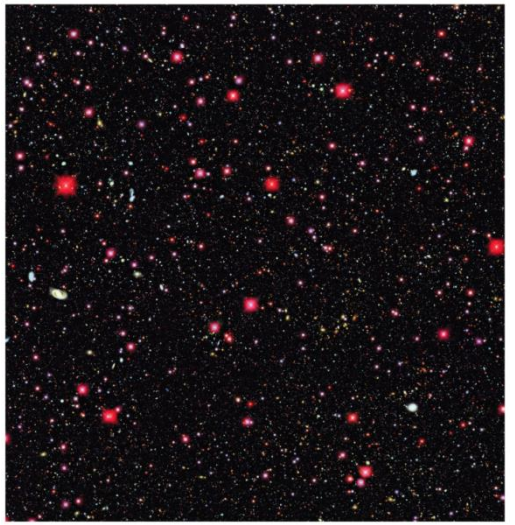


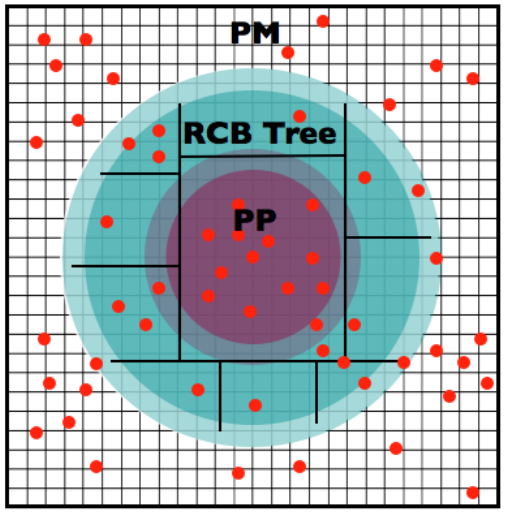
Argonne NATIONAL LABORATORY

ECP EXASCALE COMPUTING PROJECT

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



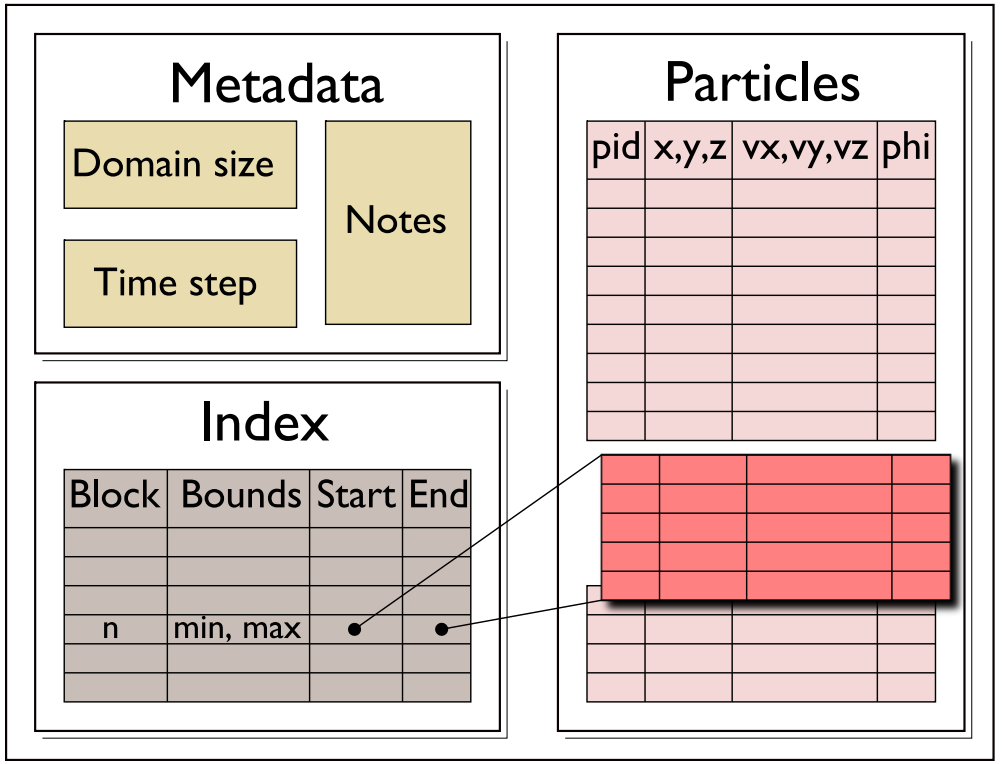


Argonne NATIONAL LABORATORY

ECP EXASCALE COMPUTING PROJECT

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]) );
    }
    for(i=0; i<nvars; i++) {
3       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();
}
```

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)

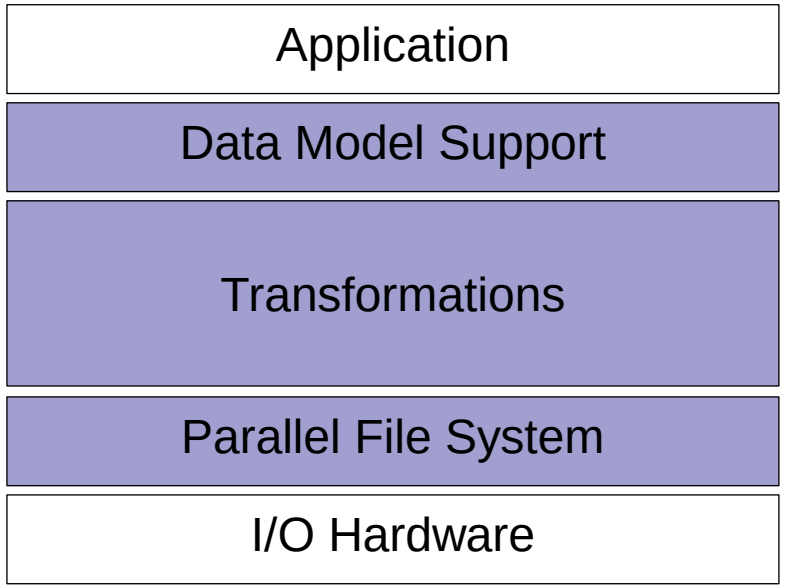
Additional I/O Interfaces

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.

PVFS, PanFS, GPFS, Lustre



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

MPI-IO, GLEAN, PLFS

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, IOFSL, Cray DVS

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.