High-Performance I/O for Scientific Applications

Rob Ross and Rob Latham

{rross, robl}@mcs.anl.gov

Mathematics & Computer Science Division

Argonne National Laboratory

May 2005

Computational Science

- Use of computer simulation as a tool for greater understanding of the real world
- Complements experimentation and theory
- As our simulations become ever more complicated:
  - Leveraging parallelism becomes more important
    - Thus large parallel machines
  - Managing code complexity bigger issue as well
    - Thus use of libraries (e.g. MPI, BLAS)
- Because data often plays a role, same issues apply there

Parallel I/O Tools

- Collections of system software and libraries have grown up to address I/O issues
  - Parallel file systems
  - MPI-IO
  - High level libraries
- Relationships between these are not always clear
- Choosing between tools can be difficult

Goals of this Tutorial

- Familiarity with available I/O tools
- Organization of tools into I/O stacks
- Understanding of what happens behind the scenes
- Guidelines for performance

- Basic MPI programming knowledge is assumed

Outline

- Introduction and I/O stacks
  - Application I/O vs. parallel I/O
  - Bridging the gap with I/O stacks
  - I/O stacks for computational science
- I/O interfaces and formats, with examples
  - POSIX file system interface
  - MPI-IO interface
  - Parallel netCDF (PnetCDF)
  - Hierarchical Data Format (HDF5)
- I/O best practices
  - Choosing an I/O interface
  - Guidelines for I/O performance
  - Tuning I/O stacks with hints
  - Enlisting the experts
- Conclusions and supplemental material

Printed References

- John May, *Parallel I/O for High Performance Computing*, Morgan Kaufmann, October 9, 2000.
  - Good coverage of basic concepts, some MPI-IO, HDF5, and serial netCDF
- William Gropp, Ewing Lusk, and Rajeev Thakur, *Using MPI-2: Advanced Features of the Message Passing Interface*, MIT Press, November 26, 1999.
  - In-depth coverage of MPI-IO API, including a very detailed description of the MPI-IO consistency semantics

On-Line References (1)

- netCDF

  `http://www.unidata.ucar.edu/packages/netcdf/`

- PnetCDF

  `http://www.mcs.anl.gov/parallel-netcdf/`

- ROMIO MPI-IO

  `http://www.mcs.anl.gov/romio/`

- HDF5 and HDF5 Tutorial

  `http://hdf.ncsa.uiuc.edu/HDF5/`

  `http://hdf.ncsa.uiuc.edu/HDF5/doc/Tutor/index.html`

On-Line References (2)

- PVFS and PVFS2

  `http://www.parl.clemson.edu/pvfs/`

  `http://www.pvfs.org/pvfs2/`

- Lustre

  `http://www.lustre.org/`

- GPFS

  `http://www.almaden.ibm.com/storagesystems/file_systems/GPFS/`

- FLASH and FLASH I/O Benchmark

  `http://flash.uchicago.edu/`

  `http://flash.uchicago.edu/~jbgallag/io_bench/`

Introduction and I/O Stacks

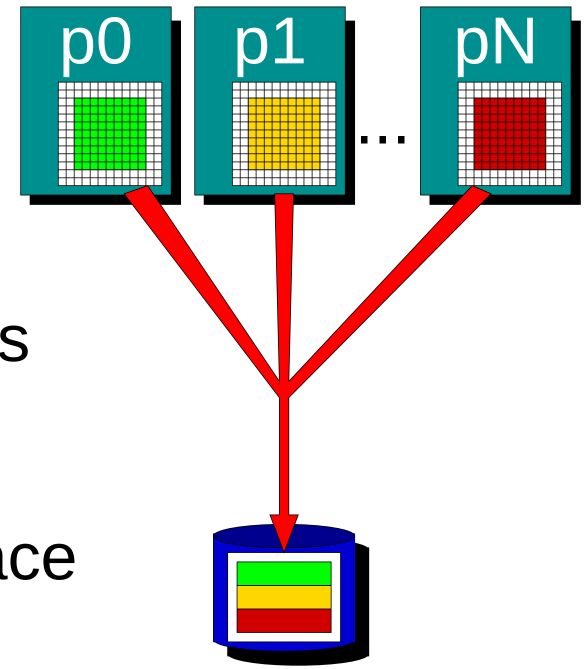
Application View of Data

- Applications have data models appropriate to domain
  - Multidimensional typed arrays, images composed of scan lines, variable length records
  - Headers, attributes on data
- Parallel file system API is an awful match
  - Bytes
  - Blocks or contiguous regions of files
  - Independent access
- Need more software!

Supporting Application I/O

(1) Provide mapping of app. domain data abstractions

- API that uses language meaningful to app. programmers



(2) Coordinate access by many processes

- Collective I/O, consistency semantics

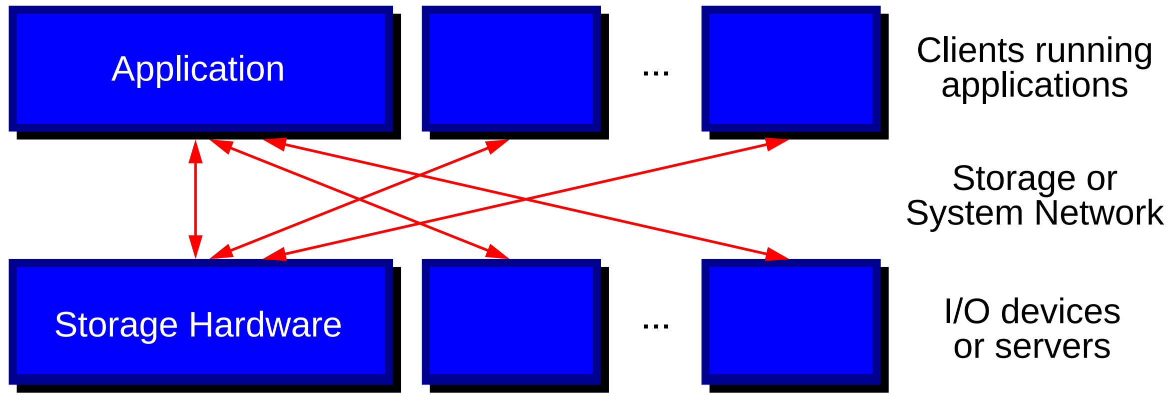
(3) Organize I/O devices into a single space

- Convenient utilities and file model

And also

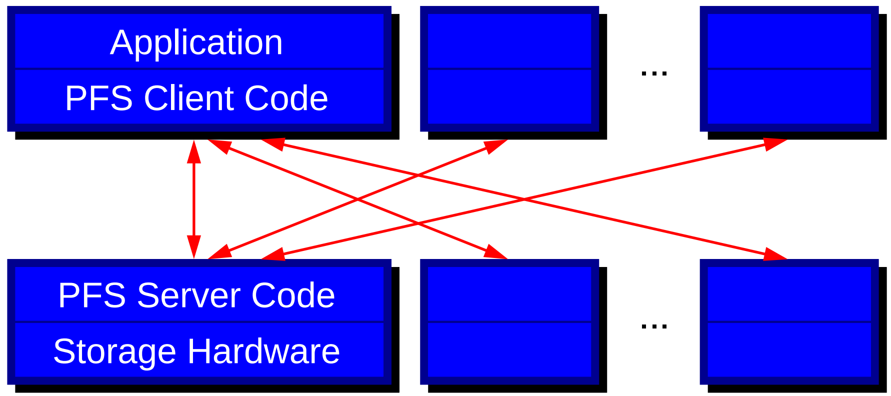
- Insulate applications from I/O system changes
- Maintain performance!!!

What about Parallel I/O?



- Focus of parallel I/O is on using parallelism to increase bandwidth
- Use multiple data sources/sinks in concert
  - Both multiple storage devices and multiple/wide paths to them
- But applications don’t want to deal with block devices and network protocols,
- So we add software layers.

Parallel File Systems (PFSs)

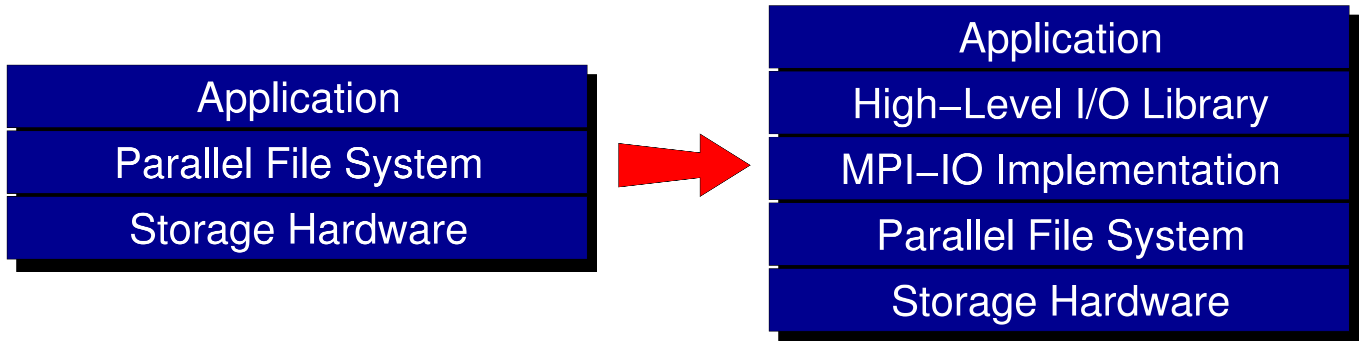


- Organize I/O devices into a single logical space
  - Striping files across devices for performance
- Export a well-defined API, usually POSIX
  - Access data in contiguous regions of bytes
  - Very general
- This is only $1/3$ of what we said we needed!

I/O Stacks

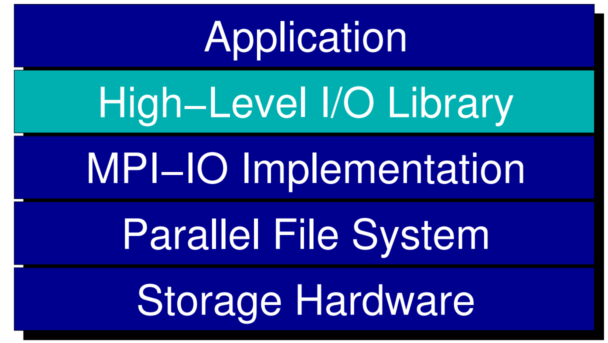
- Idea: Add some additional software components to address remaining issues
  - Coordination of access
  - Mapping from application model to I/O model
- These components will be increasingly specialized as we add layers
- Bridge this gap between existing I/O systems and application needs

I/O for Computational Science



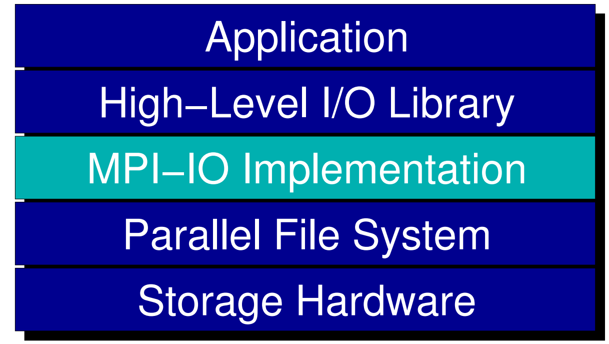
- Break up support into multiple layers:
  - High level I/O library maps app. abstractions to a structured, portable file format (e.g. HDF5, Parallel netCDF)
  - Middleware layer deals with organizing access by many processes (e.g. MPI-IO, UPC-IO)
  - Parallel file system maintains logical space, provides efficient access to data (e.g. PVFS, GPFS, Lustre)

High Level Libraries



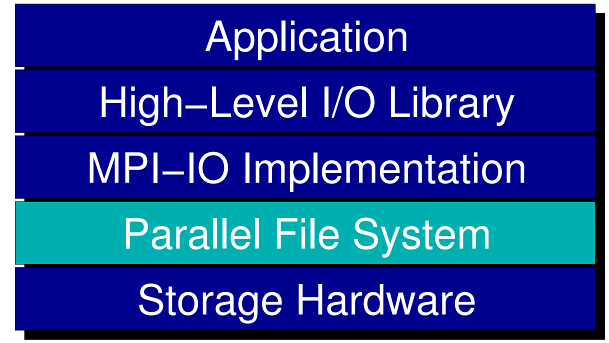
- Provide an appropriate abstraction for domain
  - Multidimensional datasets
  - Typed variables
  - Attributes
- Self-describing, structured file format
- Map to middleware interface
  - Encourage collective I/O
- Provide optimizations that middleware cannot

I/O Middleware



- Facilitate concurrent access by groups of processes
  - Collective I/O
  - Atomicity rules
- Expose a generic interface
  - Good building block for high-level libraries
- Match the underlying prog. model (e.g. MPI)
- Efficiently map middleware operations into PFS ones
  - Leverage any rich PFS access constructs

Parallel File System



- Manage storage hardware
  - Present single view
- Focus on concurrent, independent access
  - Knowledge of collective I/O usually very limited
- Publish an interface that middleware can use effectively
  - Rich I/O language
  - Relaxed but sufficient semantics

Next: I/O APIs and Formats

- Introduce the four interfaces:
  - POSIX I/O interface
  - MPI-IO interface
  - Parallel netCDF (PnetCDF) interface
  - HDF5 interface
- Example for each
  - Serial POSIX “cp” code
  - MPI-IO vizualization code
  - FLASH/PnetCDF
  - FLASH/HDF5
- Look in-depth at what happens in the I/O system
- Introduce components from the bottom up

POSIX I/O Interface

POSIX I/O

- Standard I/O interface across many platforms
- Mechanism almost all serial applications use to perform I/O
- No way of describing collective access
- Warning: semantics differ between file systems!
  - NFS is the worst of these, supporting API but not semantics
  - Determining FS type is nontrivial

Simple POSIX Examples

- POSIX I/O version of “Hello World”
- First program writes a file with text in it
- Second program reads back the file and prints the contents
- Show basic API use, error checking

Simple POSIX I/O: Writing

```
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char **argv)
{
  int fd, ret;
  char buf[13] = "Hello World\n";

  fd = open("myfile", O_WRONLY | O_CREAT, 0755);
  if (fd < 0) return 1;

  ret = write(fd, buf, 13);
  if (ret < 13) return 1;

  close(fd);
  return 0;
}
```

Simple POSIX I/O: Reading

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
  int fd, ret;
  char buf[13];

  fd = open("myfile", O_RDONLY);
  if (fd < 0) return 1;

  ret = read(fd, buf, 13);
  if (ret < 13) return 1;

  printf("%s", buf);

  close(fd);
  return 0;
}
```

Compiling and Running

```
;gcc -Wall posix-hello-write.c -o posix-hello-write
;gcc -Wall posix-hello-read.c -o posix-hello-read

;./posix-hello-write
;./posix-hello-read
Hello World

;ls myfile
-rwxr-xr-x    1 rross    rross         13 Mar 28 20:18 myfile

;cat myfile
Hello World
```

Example: cp

- Copy data from one file to another
- Easy to code, very little setup
- Easy to detect exit condition
  - `read` returns negative value

cp Code (1)

```
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char **argv)
{
  int infd, outfd, readsz, writesz;
  char buf[65536];

  if (argc < 3) return 1;

  infd = open(argv[1], O_RDONLY);
  if (infd < 0) return 1;

  outfd = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0777);
  if (outfd < 0) return 1;

  /* continues on next slide */
```

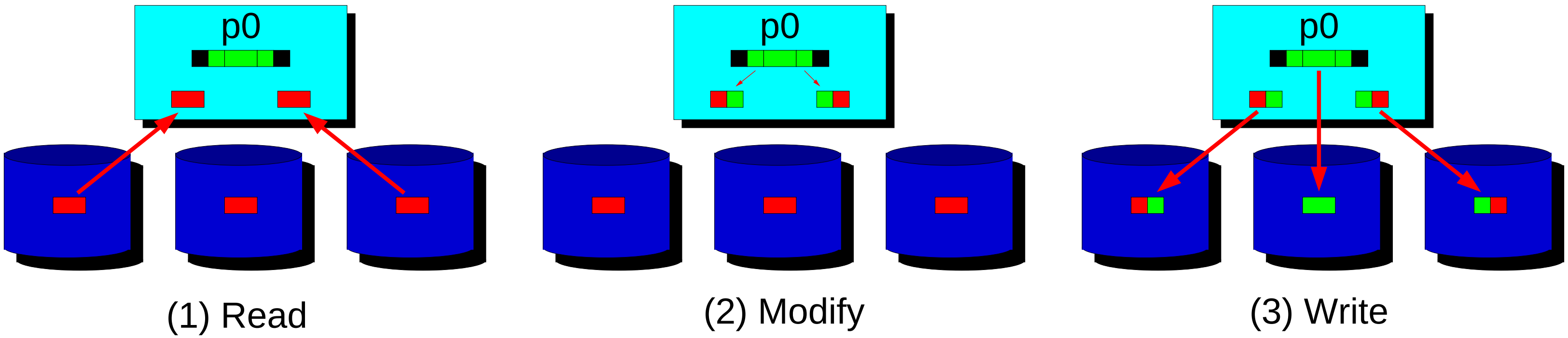
cp Code (2)

```
  /* priming read */
  readsz = read(infd, buf, 65536);
  while (readsz > 0) {
    writesz = write(outfd, buf, readsz);
    if (writesz != readsz) return 1;

    readsz = read(infd, buf, 65536);
  }

  close(infd);
  close(outfd);

  return 0;
}
```

Under the PFS Covers

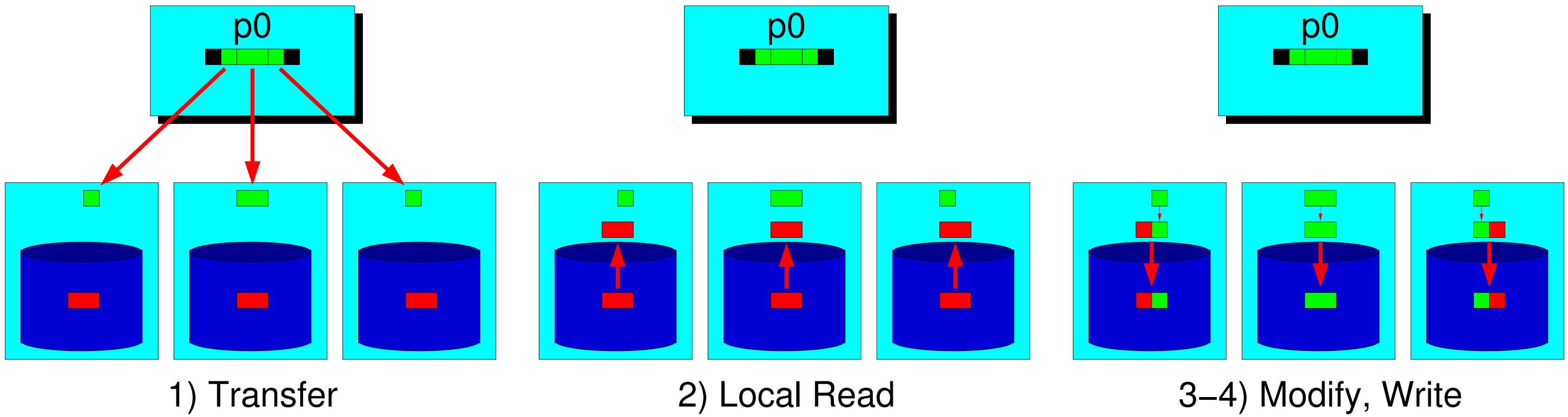
- Parallel file system software has to get data from user buffers into disk blocks (and vice versa)
- Two basic ways that PFSs manage this
  - Block-oriented access
  - Region-oriented access
- The mechanism used by the PFS does have a significant impact on the performance for some workloads
  - Region-oriented is more flexible

PFS Write: Block Accesses



- Block-oriented file systems (e.g. ones using SANs) must perform operations in terms of whole blocks
- Can require read-modify-write
  - Imagine lots of processes needing to modify the same block...

PFS Write: Region Accesses



- Some file systems can access at byte granularity
- Move less data over the network
- Manage modification of blocks locally
- In some cases they can handle noncontiguous accesses as well (e.g. PVFS, PVFS2)

POSIX Wrap-Up

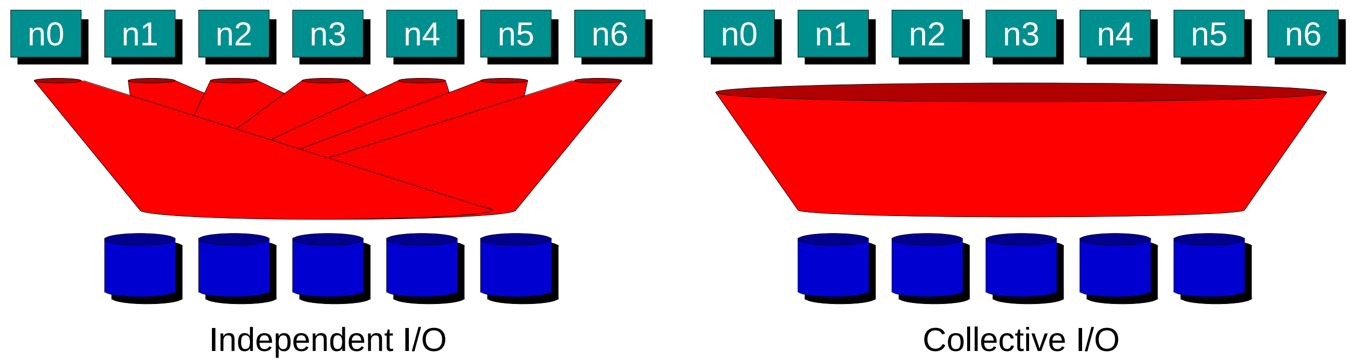
- POSIX interface is a useful, ubiquitous interface for building basic I/O tools
- No constructs useful for parallel I/O
- Should not be used in parallel applications if performance is desired

MPI-IO Interface

MPI-IO

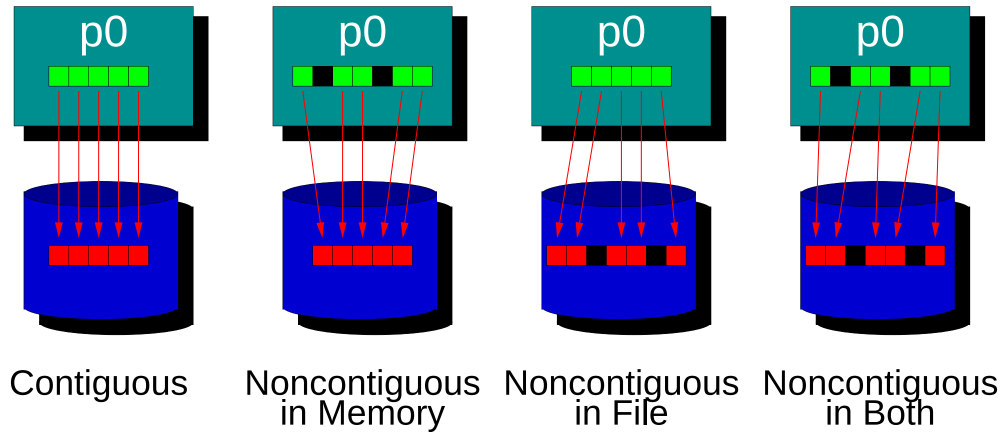
- I/O interface *specification* for use in MPI apps
- Data Model:
  - Stream of bytes in a file
  - Portable data format (external32)
    - Not self-describing
- Features:
  - Collective I/O
  - Noncontiguous I/O with MPI datatypes and file views
  - Nonblocking I/O
  - C and Fortran bindings (and more)
- Available on most platforms

Collective I/O



- Many applications have phases of computation and I/O
- During I/O phases, all processes read/write data
  - We can say they are *collectively* accessing storage
- Collective I/O is coordinated access to storage by a group of processes
  - Collective I/O functions must be called by all processes participating in I/O
  - Allows I/O layers to know more about access as a whole
- *Independent* I/O is not organized in this way
  - No apparent order or structure to accesses

Noncontiguous I/O



- *Contiguous* I/O moves data from a single block in memory into a single region of storage
- *Noncontiguous* I/O has three forms:
  - Noncontiguous in memory, noncontiguous in file, or noncontiguous in both
- Structured data leads naturally to noncontiguous I/O

Nonblocking, Asynchronous I/O

- *Blocking*, or *Synchronous*, I/O operations return when buffer may be reused
  - Data in system buffers or on disk
- Some applications like to overlap I/O and computation
  - Hiding writes, prefetching, pipelining
- A *nonblocking* interface allows for submitting I/O operations and testing for completion later
- If the system also supports *asynchronous* I/O, progress on operations can occur in the background
  - Depends on implementation
- Otherwise progress is made at start, test, wait calls

Simple MPI-IO Examples

- MPI-IO version of “Hello World”
- First program writes a file with text in it
- Second program reads back the file and prints the contents
- Show basic API use, error checking

Simple MPI-IO: Writing (1)

```
#include <mpi.h>
#include <mpio.h> /* may be necessary on some systems */

int main(int argc, char **argv)
{
  int ret, count;
  char buf[13] = "Hello World\n";
  MPI_File fh;
  MPI_Status status; /* size of data written */

  MPI_Init(&argc, &argv);

  ret = MPI_File_open(MPI_COMM_WORLD, "myfile",
                      MPI_MODE_WRONLY | MPI_MODE_CREATE,
                      MPI_INFO_NULL, &fh);
  if (ret != MPI_SUCCESS) return 1;

  /* continues on next slide */
```

Simple MPI-IO: Writing (2)

```
  ret = MPI_File_write(fh, buf, 13, MPI_CHAR, &status);
  if (ret != MPI_SUCCESS) return 1;

  MPI_Get_count(&status, MPI_CHAR, &count);
  if (count != 13) return 1;

  MPI_File_close(&fh);
  MPI_Finalize();
  return 0;
}
```

Simple MPI-IO: Reading (1)

```
#include <mpi.h>
#include <mpio.h>
#include <stdio.h>

int main(int argc, char **argv)
{
  int ret, count;
  char buf[13];
  MPI_File fh;
  MPI_Status status;

  MPI_Init(&argc, &argv);

  ret = MPI_File_open(MPI_COMM_WORLD, "myfile",
                      MPI_MODE_RDONLY,
                      MPI_INFO_NULL, &fh);
  if (ret != MPI_SUCCESS) return 1;

  /* continues on next slide */
```

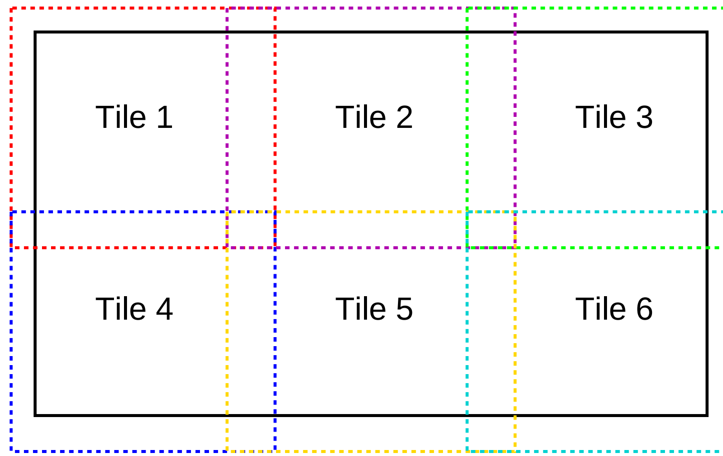
Simple MPI-IO: Reading (2)

```
    ret = MPI_File_read(fh, buf, 13, MPI_CHAR, &status);
    if (ret != MPI_SUCCESS) return 1;

    MPI_Get_count(&status, MPI_CHAR, &count);
    if (count != 13) return 1;

    printf("%s", buf);

    MPI_File_close(&fh);
    MPI_Finalize();
    return 0;
}
```

Compiling and Running

```
;mpicc mpiio-hello-write.c -o mpiio-hello-write
;mpicc mpiio-hello-read.c -o mpiio-hello-read

;mpirun -np 1 mpiio-hello-write
;mpirun -np 1 mpiio-hello-read
Hello World

;ls myfile
-rwxr-xr-x    1 rross    rross         13 Mar 28 19:18 myfile

;cat myfile
Hello World
```

Example: Visualization Staging

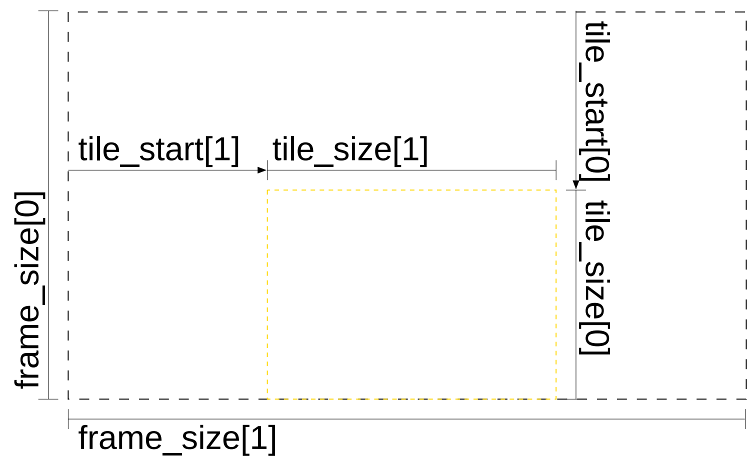


- Often large frames must be preprocessed before display on a tiled display
- First step in process is extracting “tiles” that will go to each projector
  - Perform scaling, etc.
- Parallel I/O can be used to speed up reading of tiles

Opening the File, Defining Types

```
MPI_File filehandle;
MPI_Datatype rgb;

success = MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
success = MPI_File_open(MPI_COMM_WORLD, filename, MPI_MODE_RDONLY,
                        MPI_INFO_NULL, &filehandle);

success = MPI_Type_contiguous(3, MPI_BYTE, &rgb);
success = MPI_Type_commit(&rgb);

/* in C order, last array value changes most quickly (X) */
frame_size[1] = 3*1024;  frame_size[0] = 2*768;
tile_size[1] = 1024; tile_size[0] = 768;
tile_start[1] = 1024 * (myrank % 3);
tile_start[0] = (myrank < 3) ? 0 : 768;

success = MPI_Type_create_subarray(2, frame_size, tile_size, tile_start,
                                   MPI_ORDER_C, rgb, &filetype);
success = MPI_Type_commit(&filetype);
```

MPI Subarray Datatype



- `MPI_Type_create_subarray` can describe arbitrary contiguous regions of an array
- In this case we use it to pull out a tile
- Tiles can overlap if we need them to
- Generally the MPI implementation uses vectors and indexed types under the covers

Reading Data

```
MPI_Status status;

/* set file view, skipping header */
success = MPI_File_set_view(filehandle, file_header_size, rgb,
                            filetype, "native", MPI_INFO_NULL);

/* collectively read data */
success = MPI_File_read_all(filehandle, buffer,
                            tile_size[0] * tile_size[1],
                            rgb, &status);

success = MPI_File_close(&filehandle);
```

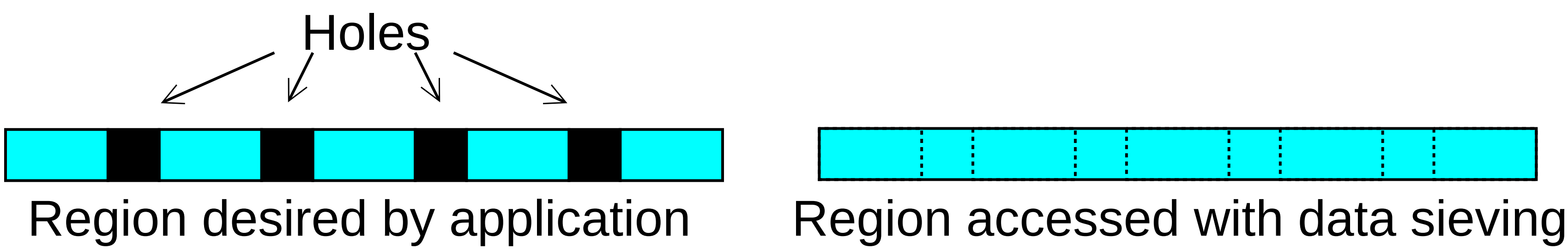
Noncontiguous File I/O

- `MPI_File_set_view` is the MPI-IO mechanism for describing noncontiguous regions in a file
  - In this case we used it to skip a header and read a subarray
- Using file views, rather than reading individual pieces, gives the implementation more information to work with

Under the Covers of MPI-IO

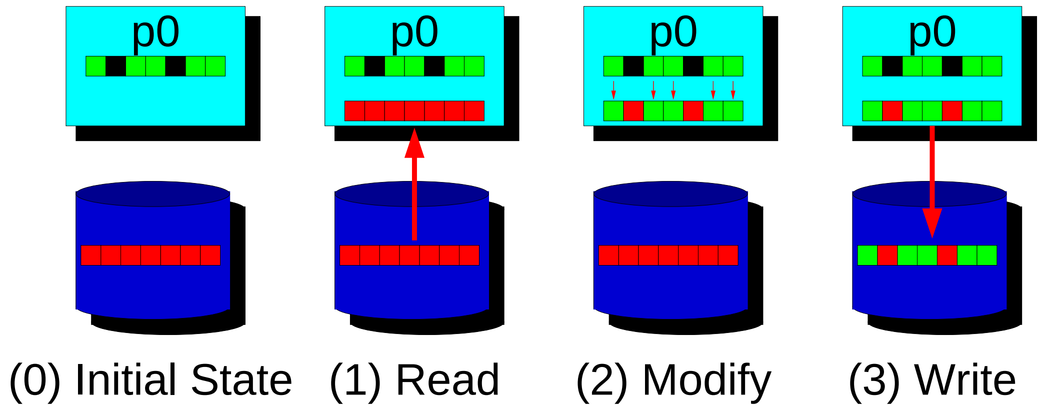
- MPI-IO implementation given a lot of information in this case:
  - Collection of processes reading data
  - Structured description of the regions
- Implementation has some options for how to obtain this data
  - Noncontiguous data access optimizations
  - Collective I/O optimizations

Data Sieving



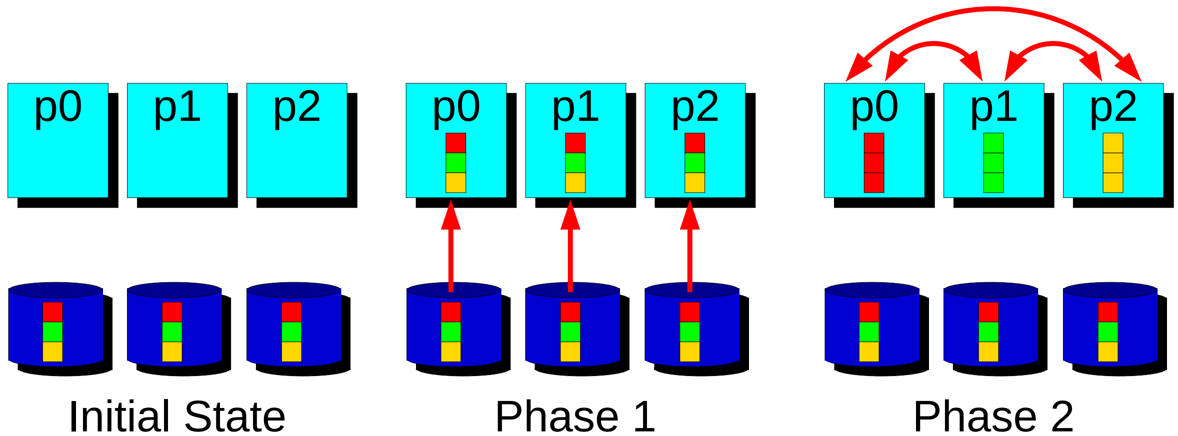
- Data sieving is used to combine lots of small accesses into a single larger one
  - Remote file systems (parallel or not) tend to have high latencies
  - Reducing # of operations important
- Generally very effective, but not as good as having a PFS that supports noncontiguous access

Data Sieving Writes



- Using data sieving for writes is more complicated
  - Must read the entire region first
  - Then make our changes
  - Then write the block back
- Requires locking in the file system
  - Can result in false sharing (interleaved access)
  - PFS supporting noncontiguous writes is preferred

Two-Phase Collective I/O

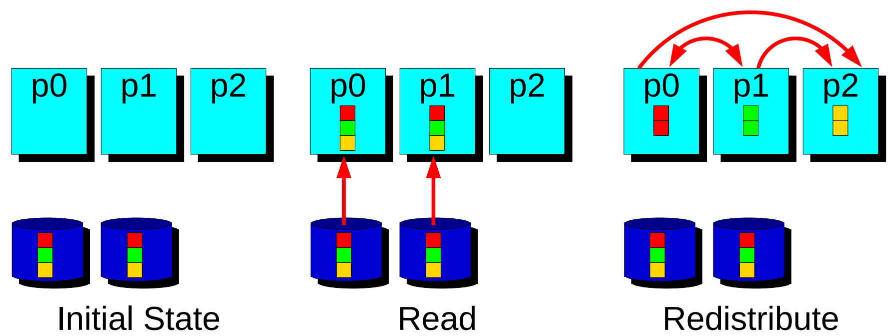


- Problems with independent, noncontiguous access
  - Lots of small accesses
  - Independent data sieving reads lots of extra data
- Idea: Reorganize access to match layout on disks
  - Single processes use data sieving to get data for many
  - Often reduces total I/O through sharing of common blocks
- Second "phase" moves data to final destinations

Two-Phase Writes

- Similarly to data sieving we need to perform a read/modify/write for two-phase writes
- Overhead is substantially lower than independent access to the same regions because there is little or no false sharing
- Note that two-phase is usually applied to file regions, not to actual blocks

Aggregation

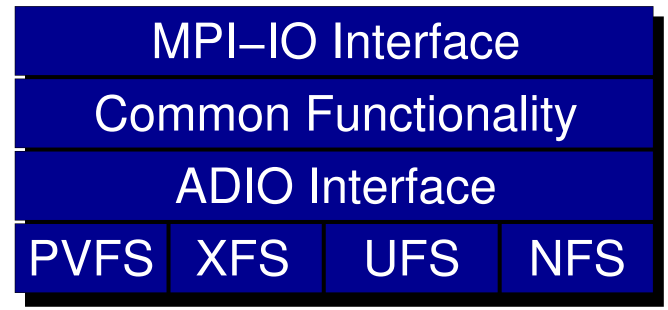


- Aggregation refers to the more general application of this concept of moving data through intermediate nodes
  - Different #s of nodes performing I/O
  - Could also be applied to independent I/O
- Can also be used for remote I/O, where aggregator processes are on an entirely different system

MPI-IO Implementations

- There are a collection of different MPI-IO implementations
- Each one has its own set of special features
- Three better-known ones are:
  - ROMIO from Argonne National Laboratory
  - MPI-IO/GPFS from IBM
  - MPI/SX and MPI/PC-32 from NEC
- Quick overview of these

ROMIO MPI-IO Implementation



- ANL implementation
- Leverages MPI-1 communication
- Layered implementation supports many storage types
  - Local file systems (e.g. XFS)
  - Parallel file systems (e.g. PVFS2)
  - NFS, Remote I/O (RFS)
- UFS implementation works for most other file systems
  - e.g. GPFS and Lustre
- Included with many MPI implementations
- Includes data sieving and two-phase optimizations

IBM MPI-IO Implementation

- For GPFS on the AIX platform
- Includes two special optimizations
  - *Data shipping* – mechanism for coordinating access to a file to alleviate lock contention (type of aggregation)
  - *Controlled prefetching* – using MPI file views and access patterns to predict regions to be accessed in future
- Not available for GPFS on Linux
  - Use ROMIO instead

NEC MPI-IO Implementation

- For NEC SX platform (MPI/SX) and Myrinet-coupled PC clusters (MPI/PC-32)
- Includes *listless I/O* optimization
  - Fast handling of noncontiguous I/O accesses in MPI layer – great for situations where the file system is lock based and/or has only contiguous I/O primitives

MPI-IO Wrap-Up

- MPI-IO provides a rich interface allowing us to describe
  - Noncontiguous accesses in memory, file, or both
  - Collective I/O
- This allows implementations to perform many transformations in order to get better I/O performance
- Also forms solid basis for high-level I/O libraries
  - But they must take advantage of these features!
- The interface honestly isn't very intuitive

Higher Level I/O Interfaces

- Provide structure to files
  - Well-defined, portable formats
  - Self-describing
  - Organization of data
  - Interfaces for discovering contents
- Present APIs more appropriate for comp. sci.
  - Typed data
  - Noncontiguous regions in memory and file
  - Multidimensional arrays
- Both implemented on top of MPI-IO

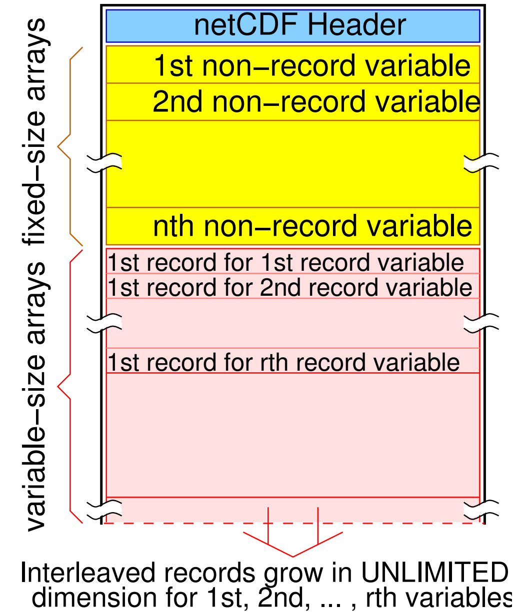
PnetCDF Interface and File Format

Parallel netCDF (PnetCDF)

- Based on original “Network Common Data Format” (netCDF) work from Unidata
- Data Model:
  - Collection of variables in single file
  - Typed, multidimensional array variables
  - Attributes on file and variables
- Features:
  - C and Fortran interfaces
  - Portable data format (same as netCDF)
  - Noncontiguous I/O in memory using MPI datatypes
  - Noncontiguous I/O in file using sub-arrays
  - Collective I/O

netCDF/PnetCDF Files

- PnetCDF files consist of three regions
  - Header
  - Non-record variables (all dimensions specified)
  - Record variables (ones with an unlimited dimension)



- Record variables are interleaved, so using more than one in a file is likely to result in poor performance due to noncontiguous accesses
- Data is written in a big-endian format

Storing Data in PnetCDF

- Create a *dataset* (file)
  - Puts dataset in *define* mode
  - Allows us to describe the contents
    - Define *dimensions* for variables
    - Define *variables* using dimensions
    - Store *attributes* if desired (for variable or dataset)
- Switch from define mode to *data* mode to write variables
  - Store variable data
- Close the dataset

Simple PnetCDF Examples

- Simplest possible PnetCDF version of “Hello World”
- First program creates a dataset with a single attribute
- Second program reads back the attribute and prints it
- Shows very basic API use and error checking

Simple PnetCDF: Writing (1)

```
#include <mpi.h>
#include <pnetcdf.h>

int main(int argc, char **argv)
{
  int ncfile, ret, count;
  char buf[13] = "Hello World\n";

  MPI_Init(&argc, &argv);

  ret = ncmpi_create(MPI_COMM_WORLD, "myfile.nc", NC_CLOBBER,
                     MPI_INFO_NULL, &ncfile);
  if (ret != NC_NOERR) return 1;

  /* continues on next slide */
```

Simple PnetCDF: Writing (2)

```
  ret = ncmpi_put_att_text(ncfile, NC_GLOBAL, "string", 13, buf);
  if (ret != NC_NOERR) return 1;

  ncmpi_enddef(ncfile);

  ncmpi_close(ncfile);
  MPI_Finalize();
  return 0;
}
```

Retrieving Data in PnetCDF

- Open a dataset in read-only mode (`NC_NOWRITE`)
- Obtain identifiers for dimensions
- Obtain identifiers for variables
- Read variable data
- Close the dataset

Simple PnetCDF: Reading (1)

```
#include <mpi.h>
#include <pnetcdf.h>

int main(int argc, char **argv)
{
  int ncfile, ret, count;
  char buf[13];

  MPI_Init(&argc, &argv);

  ret = ncmpi_open(MPI_COMM_WORLD, "myfile.nc", NC_NOWRITE,
                   MPI_INFO_NULL, &ncfile);
  if (ret != NC_NOERR) return 1;

  /* continues on next slide */
```

Simple PnetCDF: Reading (2)

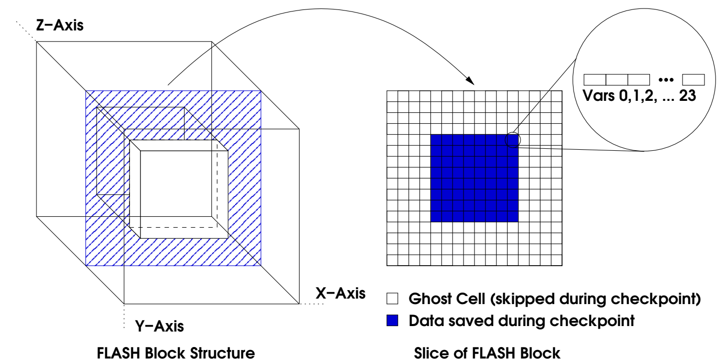
```
    ret = ncmpi_inq_attlen(ncfile, NC_GLOBAL, "string", &count);
    if (ret != NC_NOERR || count != 13) return 1;

    ret = ncmpi_get_att_text(ncfile, NC_GLOBAL, "string", buf);
    if (ret != NC_NOERR) return 1;

    printf("%s", buf);

    ncmpi_close(ncfile);
    MPI_Finalize();
    return 0;
}
```

Compiling and Running

```
;mpicc pnetcdf-hello-write.c -I /usr/local/pnetcdf/include/
  -L /usr/local/pnetcdf/lib -lpnetcdf -o pnetcdf-hello-write
;mpicc pnetcdf-hello-read.c -I /usr/local/pnetcdf/include/
  -L /usr/local/pnetcdf/lib -lpnetcdf -o pnetcdf-hello-read

;mpirun -np 1 pnetcdf-hello-write
;mpirun -np 1 pnetcdf-hello-read
Hello World

;ls -l myfile.nc
-rw-r--r--    1 rross    rross        68 Mar 26 10:00 myfile.nc

;strings myfile.nc
string
Hello World
```

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

Example: FLASH with PnetCDF

- Impose an ordering on the AMR blocks
- One file for a checkpoint
- Store each variable in its own array (minus ghost cells)
- Attributes describing run time, total blocks, etc.

Defining Dimensions

```
int status, ncid, dim_tot_blks, dim_nxb, dim_nyb, dim_nzb;
MPI_Info hints;

/* create dataset (file) */
status = ncmpi_create(MPI_COMM_WORLD, filename, NC_CLOBBER,
                      hints, &file_id);

/* define dimensions */
status = ncmpi_def_dim(ncid, "dim_tot_blks", tot_blks,
                       &dim_tot_blks);
status = ncmpi_def_dim(ncid, "dim_nxb", nzones_block[0], &dim_nxb);
status = ncmpi_def_dim(ncid, "dim_nyb", nzones_block[1], &dim_nyb);
status = ncmpi_def_dim(ncid, "dim_nzb", nzones_block[2], &dim_nzb);
```

Variables and Attributes

```
int dims = 4, dimids[4];
int varids[NVARS];

/* define variables (X changes most quickly) */
dimids[0] = dim_tot_blks;
dimids[1] = dim_nzb;  dimids[2] = dim_nyb;  dimids[3] = dim_nxb;

for (i=0; i < NVARS; i++) {
   status = ncmpi_def_var(ncid, unk_label[i], NC_DOUBLE, dims,
                          dimids, &varids[i]);
}

/* store attributes of checkpoint */
status = ncmpi_put_att_text(ncid, NC_GLOBAL, "file_creation_time",
                            string_size, file_creation_time);
status = ncmpi_put_att_int(ncid, NC_GLOBAL, "total_blocks",  NC_INT,
                            1, tot_blks);

status = ncmpi_enddef(file_id); /* enter data mode */
```

Writing Variables

```
double *unknowns; /* unknowns[blk][nzb][nyb][nxb] */
size_t start_4d[4], count_4d[4];

start_4d[0] = global_offset; /* different for each process */
start_4d[1] = start_4d[2] = start_4d[3] = 0;

count_4d[0] = local_blocks;
count_4d[1] = nzb;  count_4d[2] = nyb;  count_4d[3] = nxb;

for (i=0; i < NVARS; i++) {
   /* ... copy data into unknowns buffer ... */

   /* collectively write out all values of a single variable */
   ncmpi_put_vara_double_all(ncid, varids[i], start_4d, count_4d,
                             unknowns);
}

status = ncmpi_close(file_id);
```

Inside PnetCDF Define Mode

- In define mode (collective)
  - Use `MPI_File_open` to create file at create time
  - Set hints as appropriate
  - 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

Inside PnetCDF Data Mode

- Inside `ncmpi_put_vara_double_all`
  - Each process performs data conversion into internal buffer
  - Uses `MPI_File_set_view` to define file region
    - Contiguous in FLASH case
  - `MPI_File_write_all` collectively writes data
- At `ncmpi_close`
  - `MPI_File_close` ensures data is written to storage

MPI-IO and PFS

- As in previous examples:
  - MPI-IO performs optimizations
    - Two-phase probably applied
    - Data sieving if necessary
  - Converts to PFS operations
  - PFS client code communicates with servers, stores data

PnetCDF Wrap-Up

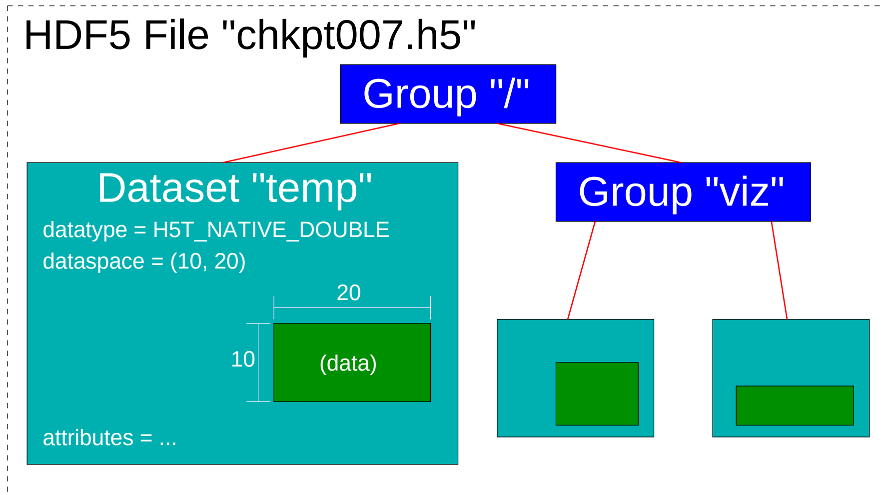
- PnetCDF gives us
  - Simple, self-describing container for data
  - Collective I/O
  - Data structures closely mapping to the variables described
- If PnetCDF meets application needs, it is likely to give good performance
  - Type conversion to portable format does add overhead

HDF5 Interface and File Format

HDF5

- Hierarchical Data Format, from NCSA
- Data Model:
  - Hierarchical data organization in single file
  - Typed, multidimensional array storage
  - Attributes on dataset, data
- Features:
  - C, C++, and Fortran interfaces
  - Portable data format
  - Optional compression
  - Data reordering (chunking)
  - Noncontiguous I/O (memory and file) with hyperslabs

HDF5 Files



- HDF5 files consist of groups, datasets, and attributes
  - A *group* is like a directory, holding other groups and datasets
  - A *dataset* holds an array of typed data
    - A *datatype* describes the type
    - A *dataspace* gives the dimensions of the array
  - *Attributes* are small datasets associated with the file, a group, or another dataset
    - Have a datatype and dataspace just as a dataset does
    - Can only be accessed as a unit

HDF5 Data Chunking

- Apps often read subsets of arrays (subarrays)
- Performance of subarray access depends in part on how data is laid out in the file
  - e.g. column vs. row major
- Apps also sometimes store sparse data sets
- *Chunking* describes a reordering of array data
  - Subarray placement in file determined lazily
  - Can reduce worst-case performance for subarray access
  - Can lead to efficient storage of sparse data
- Coordination cost in this dynamic ordering

Simple HDF5 Examples

- HDF5 version of “Hello World”
- First program creates a character array, writes text into it
- Second program reads back the array and prints the contents
- Shows basic API use

Storing Data in HDF5

- Create the HDF5 file
- Create a new group if desired
- Define a dataspace (variable dimensions)
- Define the datatype (variable type)
- Create the dataset (dataspace plus datatype)
- Store attributes if desired
- Store dataset data
- Close everything (file, group, dataspace, dataset, attributes)

Simple HDF5: Writing

```
#include <hdf5.h>

int main(int argc, char **argv)
{
  hid_t file, string_datatype, string_dataspace, string_dataset;
  hsize_t dim = 13;
  herr_t status;
  char buf[13] = "Hello World\n";

  file = H5Fcreate("myfile.h5", H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT)

  string_dataspace = H5Screate_simple(1, &dim, NULL);
  string_datatype  = H5Tcopy(H5T_NATIVE_CHAR);
  string_dataset   = H5Dcreate(file, "string", string_datatype,
                               string_dataspace, H5P_DEFAULT);

  status = H5Dwrite(string_dataset, H5T_NATIVE_CHAR, H5S_ALL, H5S_ALL,
                    H5P_DEFAULT, buf);

  H5Sclose(string_dataspace);
```

Retrieving Data in HDF5

- Open the HDF5 file
- Open the group (if one was used)
- Open the dataset
- Get the dataspace
- Get the dimensions of the dataspace
- Read dataset data
- Close everything (file, group, dataset, dataspace)

Simple HDF5: Reading

```
#include <hdf5.h>
#include <stdio.h>

int main(int argc, char **argv)
{
  hid_t file, string_dataset;
  herr_t status;
  char buf[13];

  file = H5Fopen("myfile.h5", H5F_ACC_RDONLY, H5P_DEFAULT);
  string_dataset = H5Dopen(file, "string");

  status = H5Dread(string_dataset, H5T_NATIVE_CHAR, H5S_ALL, H5S_ALL,
                   H5P_DEFAULT, buf);

  printf("%s", buf);

  H5Dclose(string_dataset);
  H5Fclose(file);
```

Compiling and Running

```
;mpicc hdf5-hello-write.c -I /usr/local/hdf5/include
  -L /usr/local/hdf5/lib/ -lhdf5 -o hdf5-hello-write
;mpicc hdf5-hello-read.c -I /usr/local/hdf5/include
  -L /usr/local/hdf5/lib/ -lhdf5 -o hdf5-hello-read

;mpirun -np 1 hdf5-hello-write
;mpirun -np 1 hdf5-hello-read
Hello World

;ls -l myfile.h5
-rw-r--r--    1 rross    rross      2061 Mar 27 23:06 myfile.h5

;strings myfile.h5
HEAP
string
TREE
P]f@
SNOD
Hello World
```

Example: FLASH with HDF5

- Same approach as with PnetCDF
- Impose an ordering on the AMR blocks
- One file for a checkpoint
- Store each variable in its own array (minus ghost cells)
- Portable format (stored natively)
- Attributes describing run time, total blocks, etc.

Setting up the File

```
int string_size = 40;
hid_t dataspace, dataset, file_id, string_type;
herr_t status;

file_id = H5Fcreate(filename, H5F_ACC_TRUNC,
                    H5P_DEFAULT, acc_template);

/* store string creation time attribute */
string_type = H5Tcopy(H5T_C_S1);
H5Tset_size(string_type, string_size);

dataspace = H5Screate_simple(4, &dimens_1d, NULL);
dataset   = H5Dcreate(file_id, "file creation time",
                      string_type, dataspace, H5P_DEFAULT);

if (myrank == 0) status = H5Dwrite(dataset, string_type, H5S_ALL,
                                   H5S_ALL, H5P_DEFAULT, create_time);

H5Tclose(string_type); H5Sclose(dataspace); H5Dclose(dataset);
```

Writing Variables (1)

```
hsize_t dimens_4d[4], start_4d[4], count_4d[4], stride_4d[4];

/* setup dataspace dimensions description */
dimens[0] = dim_tot_blks;
dimens[1] = nzb;
dimens[2] = nyb;
dimens[3] = nxb;

/* setup hyperslab description for dataset in file */
start_4d[0] = global_offset;
start_4d[1] = start_4d[2] = start_4d[3] = 0;

stride_4d[0] = stride_4d[1] = stride_4d[2] = stride_4d[3] = 1;

count_4d[0] = local_blocks;
count_4d[1] = nzb;  count_4d[2] = nyb;  count_4d[3] = nxb;

/* continues on next slide */
```

Writing Variables (2)

```
for (i=0; i < NVARS; i++) {
   hid_t dataspace, dataset_plist, dataset;

   dataspace = H5Screate_simple(rank, dimens, NULL);
   dataset_plist = H5Pcreate(H5P_DATASET_CREATE);
   dataset = H5Dcreate(file_id, record_label_new, H5T_NATIVE_DOUBLE,
                       dataspace, dataset_plist);

   status = H5Sselect_hyperslab(dataspace, H5S_SELECT_SET, start_4d,
                                stride_4d, count_4d, NULL);

   memspace = H5Screate_simple(1, nxb*nyb*nzb*dim_tot_blks, NULL);

   /* for() continued on next slide */
```

Writing Variables (3)

```
    /* for() continued from last slide */

    dxfer_template = H5Pcreate(H5P_DATASET_XFER);

    /* specify collective I/O */
    ierr = H5Pset_dxpl_mpio(dxfer_template, H5FD_MPIO_COLLECTIVE);
    ierr = H5Pset_preserve(dxfer_template, 0u);

    /* ... copy data into unknowns buffer ... */

    status = H5Dwrite(dataset, H5T_NATIVE_DOUBLE, memspace,
                      dataspace, dxfer_template, unknowns);

    H5Sclose(dxfer_template);  H5Sclose(memspace);
    H5Sclose(dataspace);  H5Dclose(dataset);
}

H5Fclose(file_id);
```

Inside HDF5

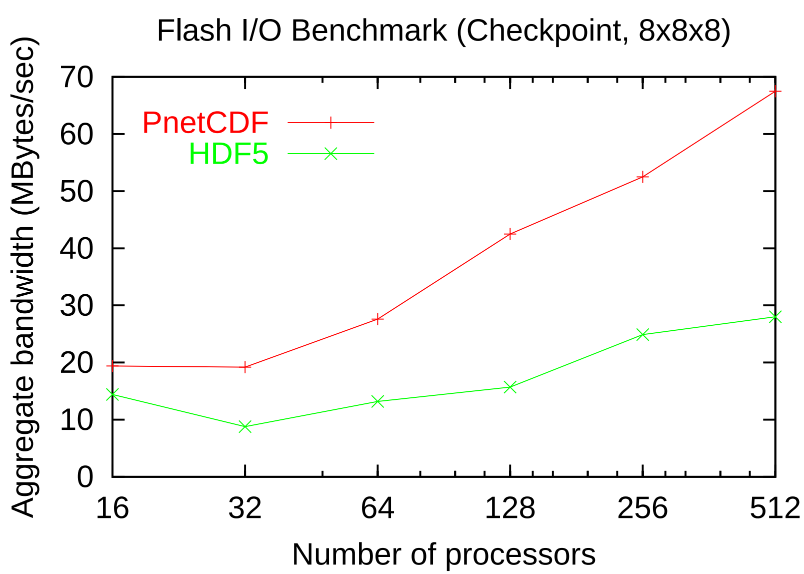
- Not so much happens before writes
- `MPI_File_open` used to open file
- Because there is no "define" mode, file layout is determined at write time
- In `H5Dwrite`:
  - Processes communicate to determine file layout
  - Process 0 performs metadata updates
  - Call `MPI_File_set_view`
  - Call `MPI_File_write_all` to collectively write
    - Only if this was turned on (more later)
- Memory hyperslab could have been used to define noncontiguous region in memory
- Data is kept in native format and converted at read time (defers overhead)

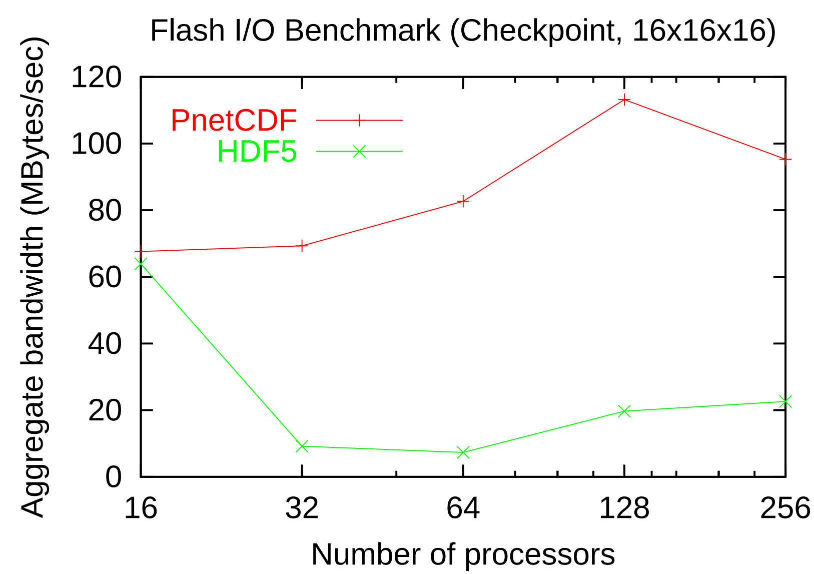
MPI-IO and PFS

- Mapping between HDF5 and MPI-IO operations is less clear than with PnetCDF
- Metadata updates at every write are a bit of a bottleneck
  - MPI-IO from process 0 introduces some skew

FLASH/HDF5 Final Notes

- FLASH doesn't use a lot of the HDF5 functionality
  - HDF5 is somewhat overkill for this application





(Numbers from ASCI White Frost, compliments of Brad Gallagher)

I/O Best Practices

How do I choose an API?

- Your programming model will limit choices.
  - Domain might too (e.g. Climate, existing netCDF data)
- Find something that matches your data model.
- Avoid APIs with lots of features you won't use.
  - Potential for overhead costing performance is high.
- Maybe the right API isn't available?
  - Get I/O people interested, consider designing a new library

Summary of API Capabilities

	POSIX	MPI-IO	PnetCDF	HDF5
Noncontig. Memory	yes	yes	yes	yes
Noncontig. File		yes	yes	yes
Coll. I/O		yes	yes	yes
Portable Format		yes	yes	yes
Self-Describing			yes	yes
Attributes			yes	yes
Chunking				yes
Hierarchical File				yes

Tuning Application I/O (1)

- Have realistic goals:
  - What is peak I/O rate?
  - What other testing has been done?
- Describe as much as possible to the I/O system:
  - Open with appropriate mode.
  - Use collective calls when available.
  - Describe data movement with fewest possible operations.
- Match file organization to process partitioning if possible
  - Order dimensions so relatively large blocks are contiguous with respect to data decomposition

Tuning Application I/O (2)

- Know what you can control:
  - What I/O components are in use?
  - What hints are accepted?
- Consider system architecture as a whole:
  - Is storage network faster than comm. network?
  - Do some nodes have better storage access than others?
- These guide our selection of hints

Controlling I/O Stack Behavior

- Most systems accept *hints* through one mechanism or another
  - Parameters to file “open” calls
  - Proprietary POSIX `ioctl` calls
  - MPI_Info
  - HDF5 transfer templates
- Allow the programmer to:
  - Explain more about the I/O pattern
  - Specify particular optimizations
  - Impose resource limitations
- Generally pass information that is used only during a particular set of accesses (between open and close, for example)

MPI-IO Hints

- MPI-IO hints may be passed via:
  - `MPI_File_open`
  - `MPI_File_set_info`
  - `MPI_File_set_view`
- Hints are optional – implementations are guaranteed to ignore ones they do not understand
  - Different implementations, even different underlying file systems, support different hints
- `MPI_File_get_info` used to get list of hints
- Next few slides cover only some hints

MPI-IO Hints: Data Sieving

- `ind_rd_buffer_size` – Controls the size (in bytes) of the intermediate buffer used by ROMIO when performing data sieving reads
- `ind_wr_buffer_size` – Controls the size (in bytes) of the intermediate buffer used by ROMIO when performing data sieving writes
- `romio_ds_read` – Determines when ROMIO will choose to perform data sieving for reads (enable, disable, auto)
- `romio_ds_write` – Determines when ROMIO will choose to perform data sieving for writes

MPI-IO Hints: Collective I/O

- cb_buffer_size – Controls the size (in bytes) of the intermediate buffer used in two-phase collective I/O
- cb_nodes – Controls the maximum number of aggregators to be used
- romio_cb_read – Controls when collective buffering is applied to collective read operations
- romio_cb_write – Controls when collective buffering is applied to collective write operations
- cb_config_list – Provides explicit control over aggregators (see ROMIO User's Guide)

MPI-IO Hints: FS-Specific

- `striping_factor` – Controls the number of I/O devices to stripe across
- `striping_unit` – Controls the striping unit (in bytes)
- `start_iodevice` – Determines what I/O device data will first be written to
- `direct_read` – Controls direct I/O for reads
- `direct_write` – Controls direct I/O for writes

Using MPI_Info

- Example: setting data sieving buffer to be a whole "frame"

```
char info_value[16];
MPI_Info info;
MPI_File fh;

MPI_Info_create(&info);
snprintf(info_value, 15, "%d", 3*1024 * 2*768 * 3);
MPI_Info_set(info, "ind_rd_buffer_size", info_value);

MPI_File_open(comm, filename, MPI_MODE_RDONLY, info, &fh);

MPI_Info_free(&info);
```

Hints and PnetCDF

- Uses MPI_Info, so almost identical

```
char info_value[16];
MPI_Info info;
MPI_File fh;

MPI_Info_create(&info);
snprintf(info_value, 15, "%d", 3*1024 * 2*768 * 3);
MPI_Info_set(info, "ind_rd_buffer_size", info_value);

ncmpi_open(comm, filename, NC_NOWRITE, info, &ncfile);

MPI_Info_free(&info);
```

Hints and HDF5

- HDF5 uses a combination of *property lists* and MPI_Info structures for passing hints
  - Property list holds HDF5-specific hints
  - `H5Pset_set_fapl_mpio` used to pass MPI_Info in as well
- HDF5 is very configurable; lots of options
- We've been talking about details like this long enough :)

Helping I/O Experts Help You

- Scenarios
  - Explaining logically what you are doing
  - Separate the conceptual structures from their representation on storage
  - Common vs. infrequent patterns
  - Possible consistency management simplifications
- Application I/O kernels
  - Simple codes exhibiting similar I/O behavior
  - Easier for I/O group to work with
  - Useful for acceptance testing!
  - Needs to be pretty close to the real thing...

Wrapping Up

- We've covered a lot of ground in a short time
  - Very low-level, serial interfaces
  - High-level, hierarchical file formats
- There is no magic in high performance I/O
  - Under the covers it looks a lot like shared memory or message passing
  - Knowing how things work will lead you to better performance
- Things will continue to get more complicated, but hopefully easier too!
  - Remote access to data
  - More layers to I/O stack
  - Domain-specific application interfaces

Additional Notes

Building ROMIO in MPICH1

- It's important to tell ROMIO what file systems to support

```
tar xzf mpich-1.2.5.2.tar.gz
cd mpich-1.2.5.2
RSHCOMMAND=ssh
export RSHCOMMAND
./configure --with-romio="--file_system=ufs+testfs" \
    --without-mpe --prefix=/usr/local/mpich-1.2.5.2
make
make install
```

Building PnetCDF

- PnetCDF will discover the mpicc if you tell it where MPI is installed.
- See READMEs for various systems if there are problems.

```
tar xjf parallel-netcdf-0.9.3.tar.bz2
cd parallel-netcdf-0.9.3/
./configure --with-mpi=/usr/local/mpich-1.2.5.2/ \
    --prefix=/usr/local/parallel-netcdf-0.9.3
make
make install
```

Building HDF5

- HDF5 wants you to define CC to be your MPI compiler

```
tar xzf hdf5-1.6.2.tar.gz
cd hdf5-1.6.2/
PATH=‘echo $PATH‘:/usr/local/mpich-1.2.5.2/bin/
CC=mpicc
export CC
./configure --with-parallel \
    --prefix=/usr/local/hdf5-1.6.2
make
make install
```

Acknowledgements

This work is supported in part by U.S. Department of Energy Grant DE-FC02-01ER25506, by National Science Foundation Grants EIA-9986052, CCR-0204429, and CCR-0311542, and by the U.S. Department of Energy under Contract W-31-109-ENG-38.