Parallel HDF5

The HDF Group

Outline

- Overview of Parallel I/O
- Overview of Parallel HDF5 design
- PHDF5 Programming Model
- Performance Analysis and use cases

Overview of Parallel IO

Def. Parallel I/O

At the application level

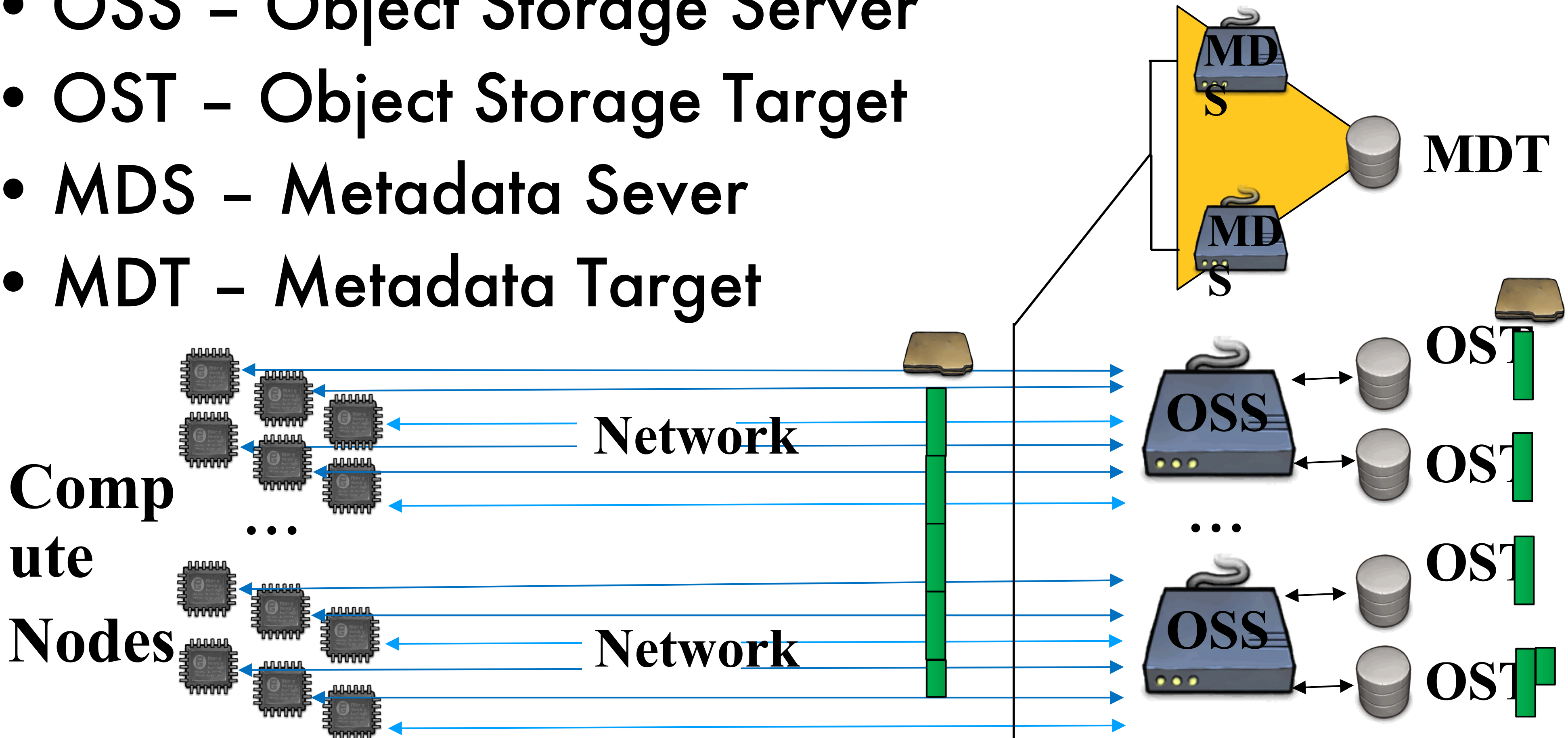
- Concurrent writing to and reading from a single file from multiple processes

At the system level

- Parallel file system which supports concurrent process access

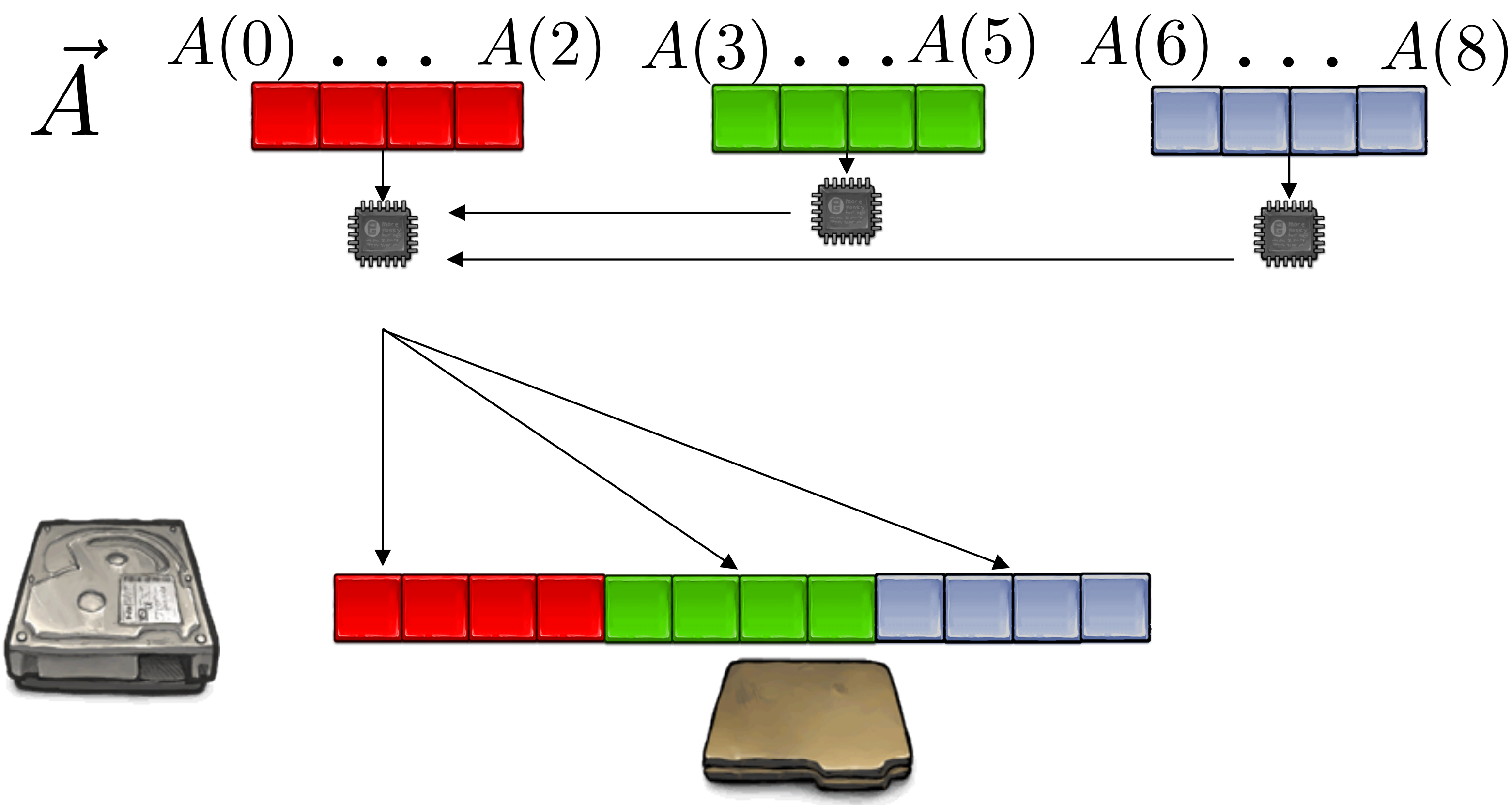
Parallel File System Nomenclature

- OSS – Object Storage Server
- OST – Object Storage Target
- MDS – Metadata Sever
- MDT – Metadata Target



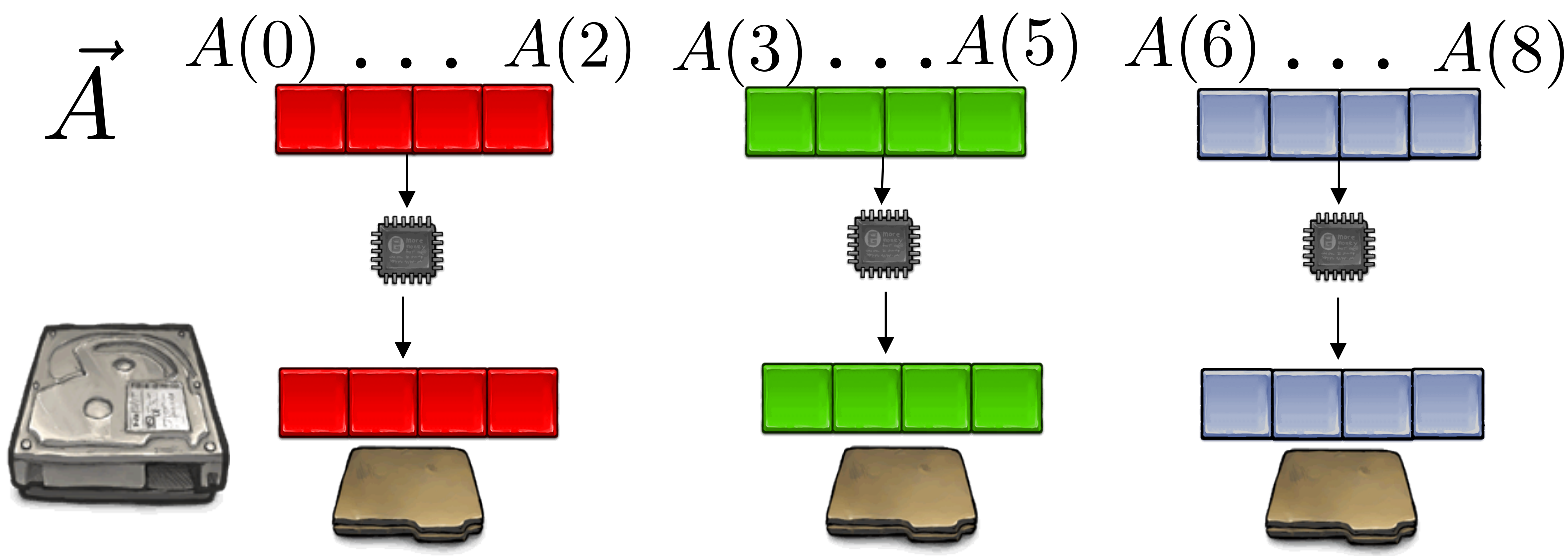
- The file is striped over multiple "disks" (e.g., Lustre OSTs) depending on the stripe size and stripe count with which the file was created
  - https://www.nics.tennessee.edu/computing-resources/file-systems/io-lustre-tips

Non-parallel I/O[1]

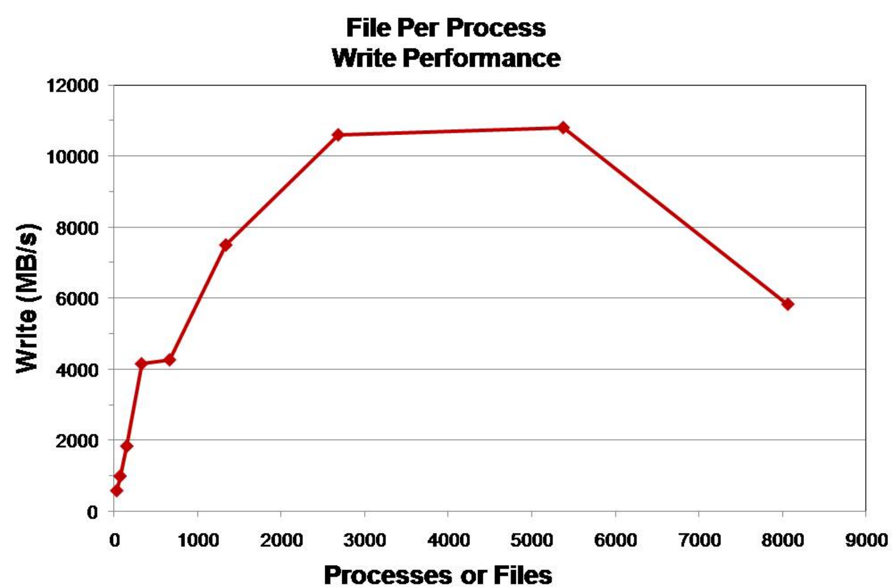


- *Friends don't let friends to this*
- Very bad performance
- Not scalable

Independent Parallel I/O[1]

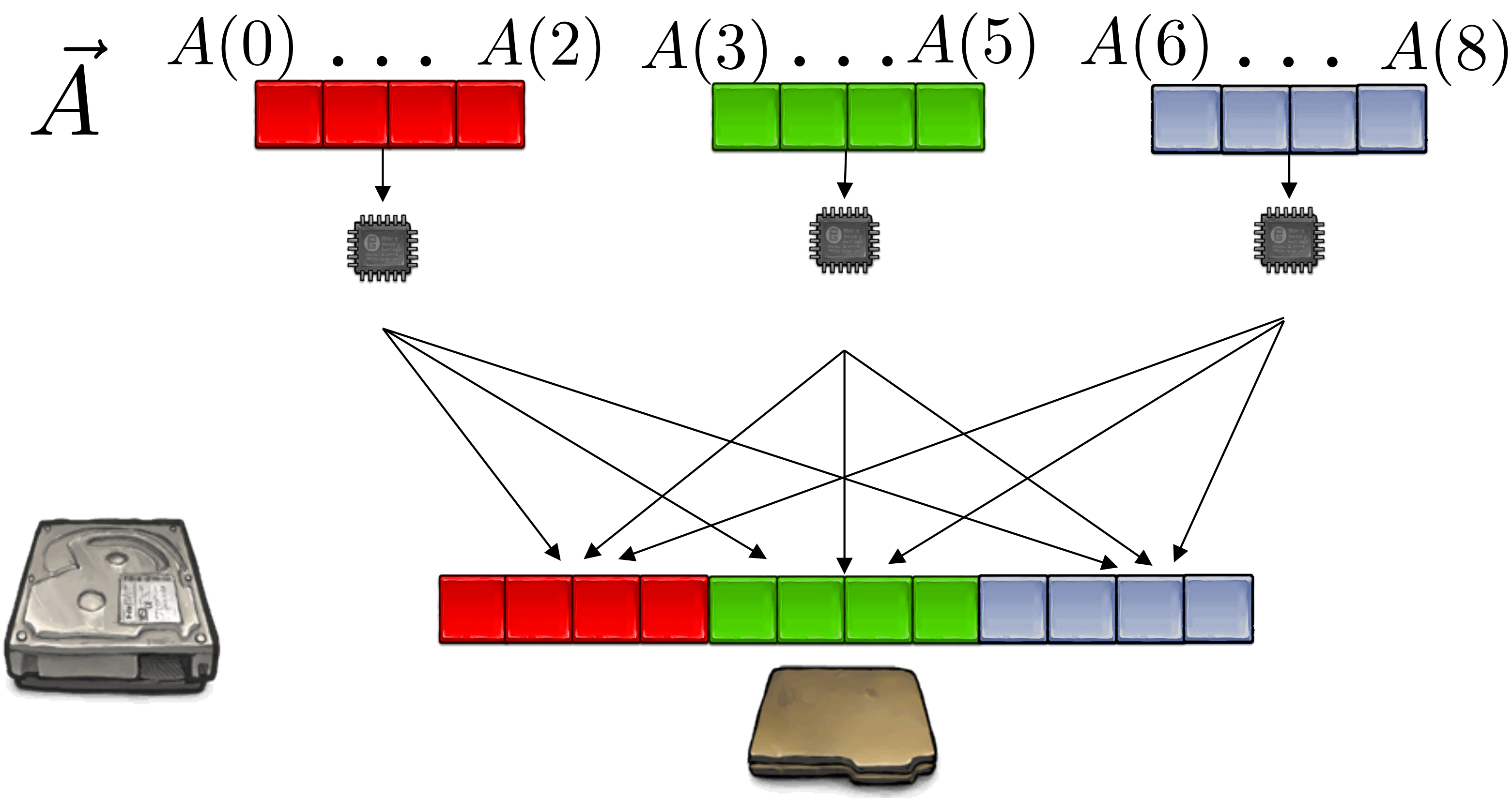


- Results in a large number of files
  - May run into file system limitation on the number of open files allowed
- Not usable from a different number of processes
- Usually, have to post-process the files
- Can achieve very good I/O performance



https://www.nics.tennessee.edu/computing-resources/file-systems/io-lustre-tips

Cooperative Parallel I/O[1]



- Coordination between processes to a single file
- Must use MPI IO
- Can achieve excellent performance

MPI-IO vs. HDF5

- MPI-IO is an Input/Output API[1]
  - Replacement functions for POSIX I/O
  - Handles multiple I/O schemes
    - Including schemes only available via MPI
  - It treats the data file as a "linear byte stream"
    - Each MPI process needs to provide its own file view and data representations to interpret those bytes.
    - Programmer handles
      - Collective coordination of operations
      - Creating user-defined datatypes for both in memory and file layout
      - Process layout of data in a file

[1] W. Gropp, parallel@illinois

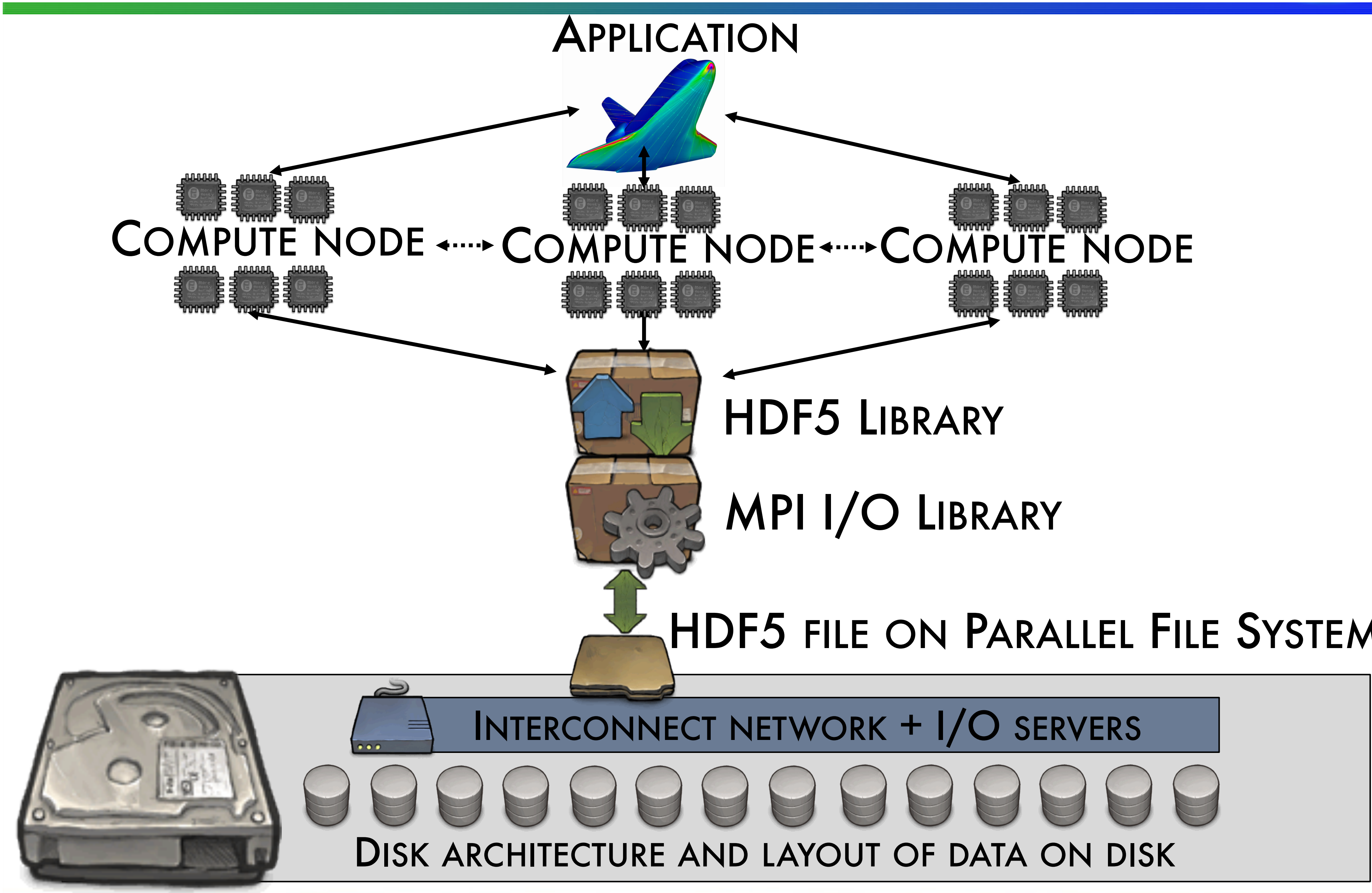
MPI-IO vs. PHDF5

- PHDF5 is data management software
  - It stores data and metadata according to the HDF5 data format specification

http://support.hdfgroup.org/HDF5/doc/H5.format.html

- PHDF5 files are HDF5 files
  - Portable to different platforms
- PHDF5 handles the MPI I/O details at a higher level
  - It requires MPI I/O for parallel I/O‡
- The PHDF5 API consists of:
  - The standard HDF5 API
  - A few extra knobs and calls
  - A parallel "etiquette"

PHDF5 implementation layers



OVERVIEW OF PARALLEL HDF5 DESIGN

How to Compile PHDF5 Applications

- Cmake
- Autotools (configure, make, etc..)
- Wrappers contain compiler/linker flags
  - h5pcc – HDF5 C compiler command
    - Similar to mpicc
  - h5pfc – HDF5 Fortran compiler command
    - Similar to mpif90

- To compile:

```
h5pcc h5prog.c
h5pfc h5prog.f90
```

h5pcc/h5pfc -show option

- -show displays the compiler commands and options without executing them, i.e., dry run

```
h5pcc -show Sample_mpio.c
 mpicc -I/home/packages/phdf5/include \
 -D_LARGEFILE_SOURCE -D_LARGEFILE64_SOURCE \
 -D_FILE_OFFSET_BITS=64 -D_POSIX_SOURCE \
 -D_BSD_SOURCE -std=c99 -c Sample_mpio.c

mpicc -std=c99 Sample_mpio.o \
 -L/home/packages/phdf5/lib \
 home/packages/phdf5/lib/libhdf5_hl.a \
 /home/packages/phdf5/lib/libhdf5.a -lz -lm -Wl,-rpath \
 -Wl,/home/packages/phdf5/lib
```

Collective vs. Independent Calls

- MPI definition of collective calls
  - All processes of the communicator must participate in the right order. E.g.,

	Process1	Process2
✔	call A(); call B();	call A(); call B(); ...CORRECT
⊖	call A(); call B();	call B(); call A(); ...WRONG

- Neither mode is preferable *a priori*

Collective I/O: Attempts to combine multiple smaller independent I/O ops into fewer larger ops

Programming Restrictions

- Most PHDF5 APIs are collective
- PHDF5 opens a parallel file with a communicator
  - Returns a file-handle
  - Future access to the file via the file-handle
- All processes must participate in collective PHDF5 APIs
- Different files can be opened via different communicators

Collective HDF5 calls

- All HDF5 APIs that modify structural metadata are collective!
  - File operations
    - `H5Fcreate, H5Fopen, H5Fclose, etc`
  - Object creation
    - `H5Dcreate, H5Dclose, etc`
  - Object structure modification (e.g., dataset extent modification)
    - `H5Dset_extent, etc`
- http://www.hdfgroup.org/HDF5/doc/RM/CollectiveCalls.html

Other HDF5 calls

- Array data transfer can be collective or independent
  - Dataset operations: `H5Dwrite, H5Dread`
- Collectiveness is indicated by function parameters, not by function names as in MPI API

What does PHDF5 support ?

- After a file is opened by the processes of a communicator
  - All parts of file are accessible by all processes
  - All objects in the file are accessible by all processes
  - Multiple processes may write to the same data array
  - Each process may write to individual data array

PHDF5 API languages

- C and Fortran language interfaces
- Most platforms with MPI-IO supported. e.g.,
  - IBM AIX
  - Linux clusters
  - Cray XT

Programming model

- HDF5 uses access template object (property list) to control the file access mechanism

- General model to access an HDF5 file in parallel:
  - Set-up MPI I/O access template (file access property list)

H5Fcreate (H5Fopen)	create (open) File
H5Screate_simple/H5Screate	create dataSpace
H5Dcreate (H5Dopen)	create (open) Dataset
H5Dread, H5Dwrite	**access Dataset**
H5Dclose	close Dataset
H5Sclose	close dataSpace
H5Fclose	close File

Writing patterns

EXAMPLE

Parallel HDF5 tutorial examples

- For simple examples how to write different data patterns see

http://www.hdfgroup.org/HDF5/Tutor/parallel.html

General Programming model

- Each process defines memory and file hyperslabs using `H5Sselect_hyperslab`
- Each process executes a write/read call using hyperslabs defined, which can be either collective or independent
- The hyperslab parameters define the portion of the dataset to write to
  - Contiguous hyperslab
  - Regularly spaced data (column or row)
  - Pattern
  - Blocks

Setup MPI-IO access template

Each process of the MPI communicator creates an access template and sets it up with MPI parallel access information

C:

```
herr_t H5Pset_fapl_mpio(hid_t plist_id,
       MPI_Comm comm,  MPI_Info info);
```

Fortran:

```
h5pset_fapl_mpio_f(plist_id, comm, info)
integer(hid_t) :: plist_id
integer        :: comm, info
```

`plist_id` is a file access property list identifier

C Example Parallel File Create

```c
comm = MPI_COMM_WORLD;
info = MPI_INFO_NULL;
/*
 * Initialize MPI
 */
MPI_Init(&argc, &argv);
/*
 * Set up file access property list for MPI-IO access
 */
plist_id = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_fapl_mpio(plist_id, comm, info);

file_id = H5Fcreate(H5FILE_NAME, H5F_ACC_TRUNC,
   H5P_DEFAULT, plist_id);
/*
 * Close the file.
 */
H5Fclose(file_id);
MPI_Finalize();
```

Fortran Example Parallel File Create

```
   comm = MPI_COMM_WORLD
   info = MPI_INFO_NULL
   CALL MPI_INIT(mpierror)
   !
   !Initialize FORTRAN predefined datatypes
   CALL h5open_f(error)
   !
   !Setup file access property list for MPI-IO access.
-> CALL h5pcreate_f(H5P_FILE_ACCESS_F, plist_id, error)
-> CALL h5pset_fapl_mpio_f(plist_id, comm, info, error)
   !
   !Create the file collectively.
-> CALL h5fcreate_f(filename, H5F_ACC_TRUNC_F, file_id,
           error, access_prp = plist_id)
   !
   !Close the file.
   CALL h5fclose_f(file_id, error)
   !
   !Close FORTRAN interface
   CALL h5close_f(error)
   CALL MPI_FINALIZE(mpierror)
```

Creating and Opening Dataset

- All processes of the communicator open/close a dataset by a collective call
  - ✓ C: H5Dcreate or H5Dopen; H5Dclose
  - ✓ Fortran: h5dcreate_f or h5dopen_f; h5dclose_f
- All processes of the communicator must extend an unlimited dimension dataset before writing to it
  - ✓ C: H5Dextend
  - ✓ Fortran: h5dextend_f

C Example: Create Dataset

```
file_id = H5Fcreate(…);
/*
 * Create the dataspace for the dataset.
 */
dimsf[0] = NX;
dimsf[1] = NY;
filespace = H5Screate_simple(RANK, dimsf, NULL);

/*
 * Create the dataset with default properties collective.
 */
dset_id = H5Dcreate(file_id, “dataset1”, H5T_NATIVE_INT,
                    filespace, H5P_DEFAULT);

H5Dclose(dset_id);
/*
 * Close the file.
 */
H5Fclose(file_id);
```

Fortran Example: Create Dataset

```
CALL h5fcreate_f(filename, H5F_ACC_TRUNC_F, file_id,
     error, access_prp = plist_id)
CALL h5screate_simple_f(rank, dimsf, filespace, error)
!
! Create the dataset with default properties.
!
->CALL h5dcreate_f(file_id, “dataset1”, H5T_NATIVE_INTEGER,
                  filespace, dset_id, error)
!
! Close the dataset.
CALL h5dclose_f(dset_id, error)
!
! Close the file.
CALL h5fclose_f(file_id, error)
```

Accessing a Dataset

- All processes that have opened dataset may do collective I/O
- Each process may do independent and arbitrary number of data I/O access calls
  - C: H5Dwrite and H5Dread
  - Fortran: h5dwrite_f and h5dread_f

Programming model for dataset access

- Create and set dataset transfer property
  - C: H5Pset_dxpl_mpio
    - H5FD_MPIO_COLLECTIVE
    - H5FD_MPIO_INDEPENDENT (default)
  - Fortran: h5pset_dxpl_mpio_f
    - H5FD_MPIO_COLLECTIVE_F
    - H5FD_MPIO_INDEPENDENT_F (default)
- Access dataset with the defined transfer property

C Example: Collective write

```c
/*
 * Create property list for collective dataset write.
 */
plist_id = H5Pcreate(H5P_DATASET_XFER);
H5Pset_dxpl_mpio(plist_id, H5FD_MPIO_COLLECTIVE);

status = H5Dwrite(dset_id, H5T_NATIVE_INT,
                  memspace, filespace, plist_id, data);
```

Fortran Example: Collective write

```
! Create property list for collective dataset write
!
CALL h5pcreate_f(H5P_DATASET_XFER_F, plist_id, error)
CALL h5pset_dxpl_mpio_f(plist_id, &
                        H5FD_MPIO_COLLECTIVE_F, error)

!
! Write the dataset collectively.
!
CALL h5dwrite_f(dset_id, H5T_NATIVE_INTEGER, data, &
                error, &
                file_space_id = filespace, &
                mem_space_id = memspace, &
                xfer_prp = plist_id)
```

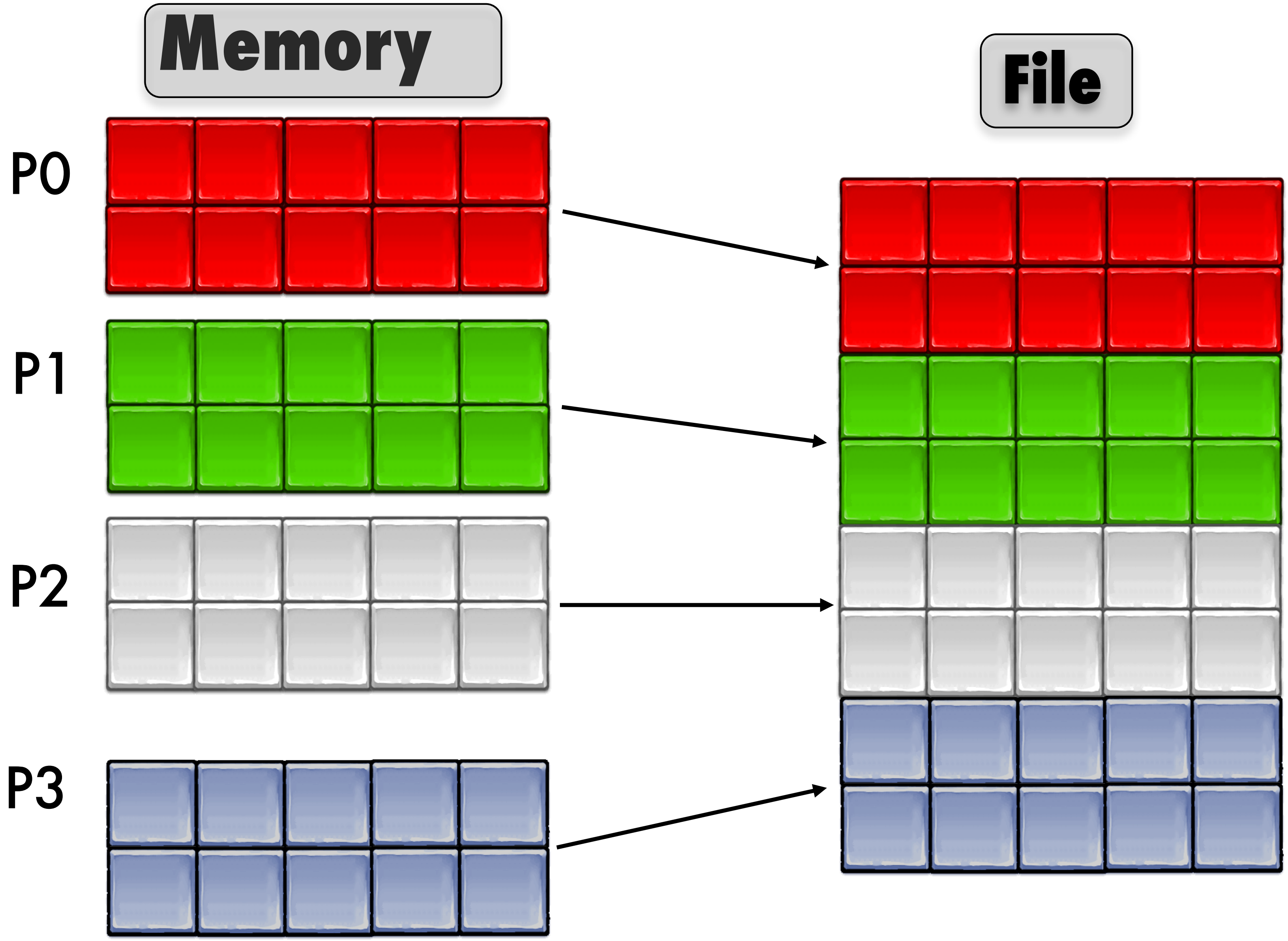
Writing and Reading Hyperslabs

- Distributed memory model: data is split among processes
- PHDF5 uses HDF5 hyperslab model
- Each process defines memory and file hyperslabs

```
H5Sselect_hyperslab(
   filespace,H5S_SELECT_SET,
   offset, stride, count, block
)
```

- Each process executes partial write/read call
  - Collective calls
  - Independent calls

Example 1: Writing dataset by rows

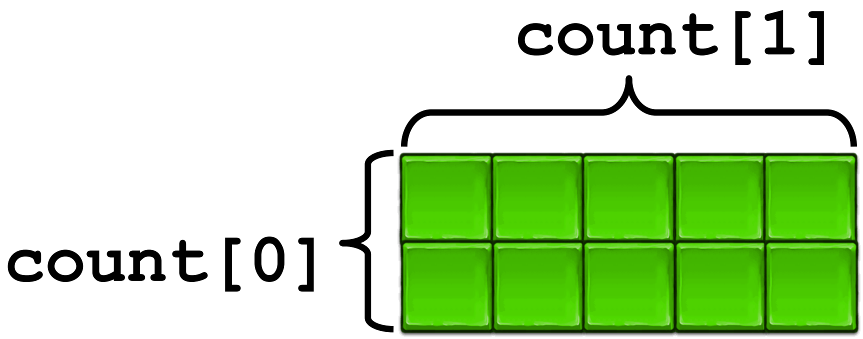


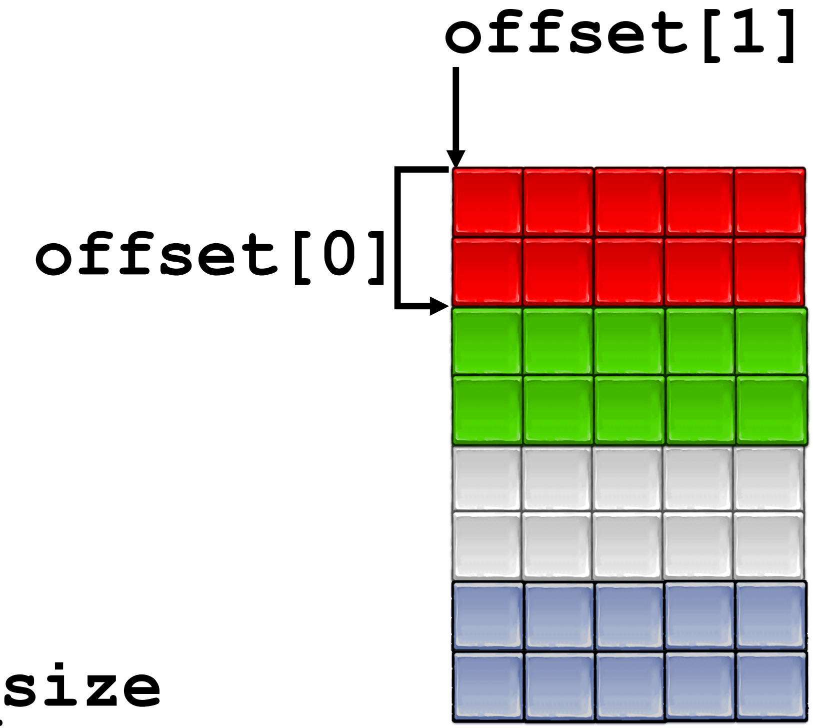
Example 1: *Writing dataset by rows*

Memory

File

Process P1






```
count[0] = dimsf[0]/mpi_size
count[1] = dimsf[1];
offset[0] = mpi_rank * count[0];  /* = 2 */
offset[1] = 0;
```

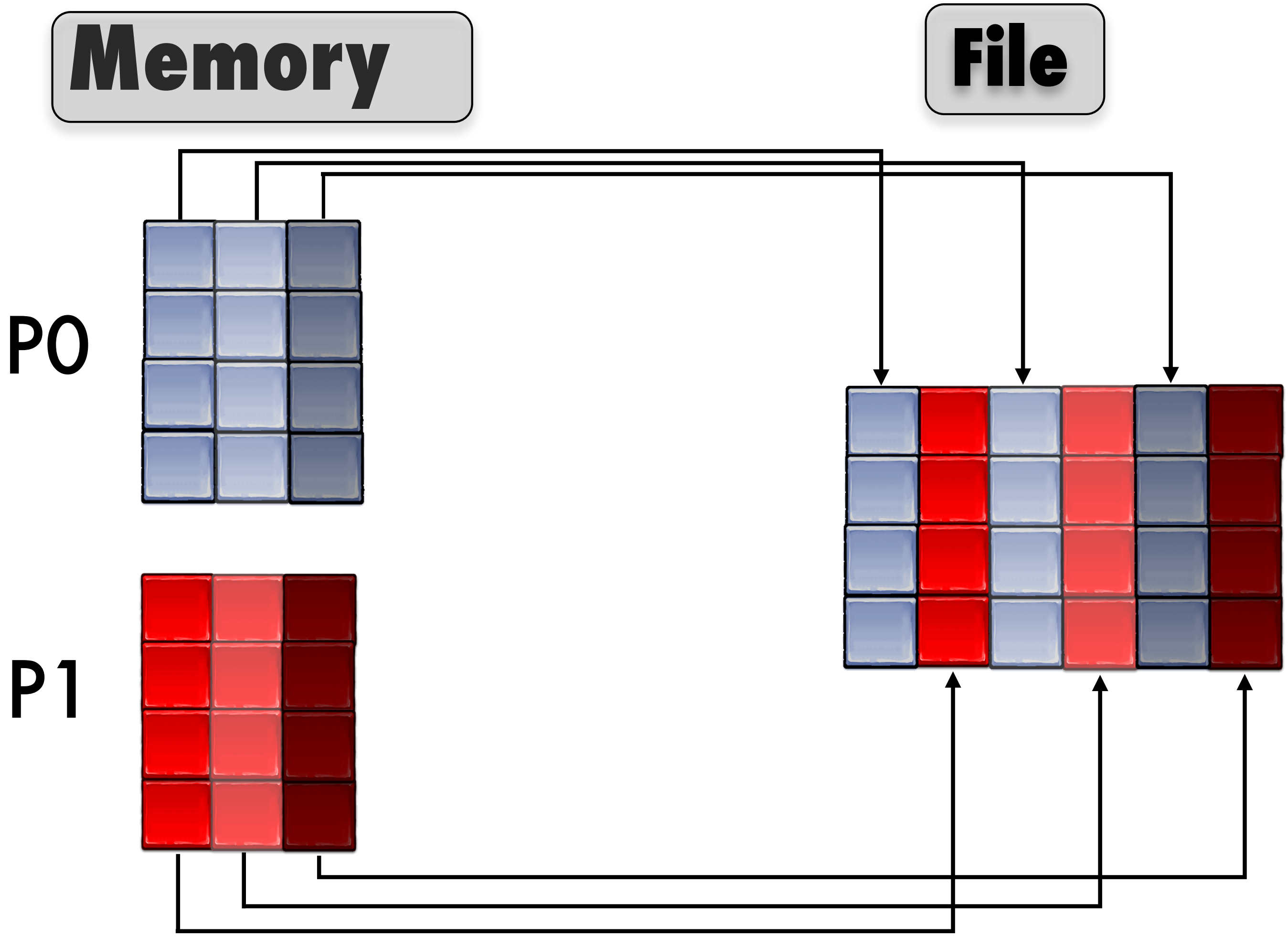
Example 1: *Writing dataset by rows*

```
/*
 * Each process defines dataset in memory and
 * writes it to the hyperslab
 * in the file.
 */
 count[0] = dimsf[0]/mpi_size;
 count[1] = dimsf[1];
 offset[0] = mpi_rank * count[0];
 offset[1] = 0;
 memspace = H5Screate_simple(RANK,count,NULL);

/*
 * Select hyperslab in the file.
 */
 filespace = H5Dget_space(dset_id);
 H5Sselect_hyperslab(filespace,
     H5S_SELECT_SET,offset,NULL,count,NULL);
```

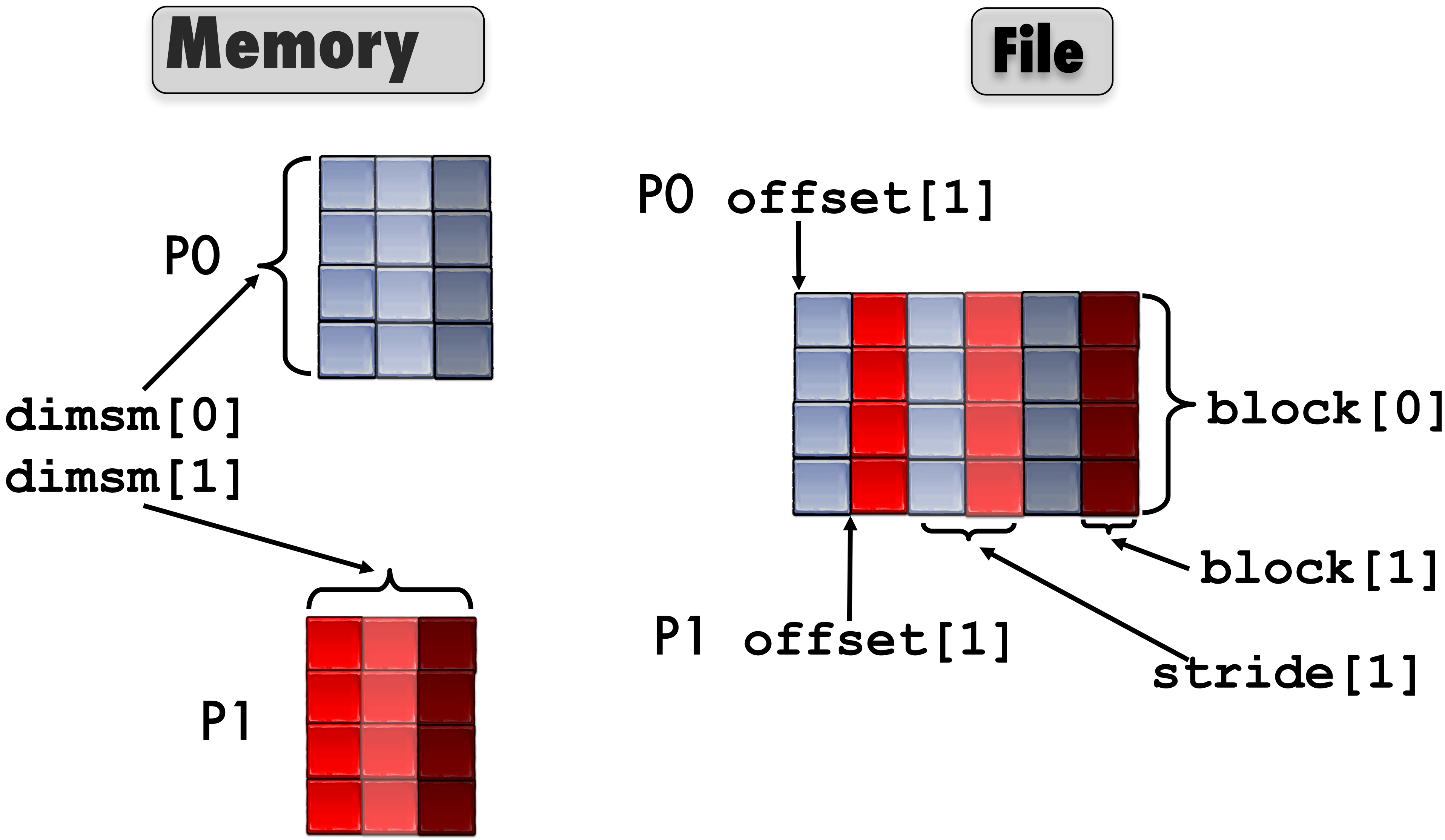
Writing by rows: Output of *h5dump*

```
HDF5 "SDS_row.h5" {
GROUP "/" {
   DATASET "IntArray" {
      DATATYPE  H5T_STD_I32BE
      DATASPACE  SIMPLE { ( 8, 5 ) / ( 8, 5 ) }
      DATA {
         10, 10, 10, 10, 10,
         10, 10, 10, 10, 10,
         11, 11, 11, 11, 11,
         11, 11, 11, 11, 11,
         12, 12, 12, 12, 12,
         12, 12, 12, 12, 12,
         13, 13, 13, 13, 13,
         13, 13, 13, 13, 13
      }
   }
}
}
```

Example 2: *Writing dataset by columns*



Example 2: *Writing dataset by column*



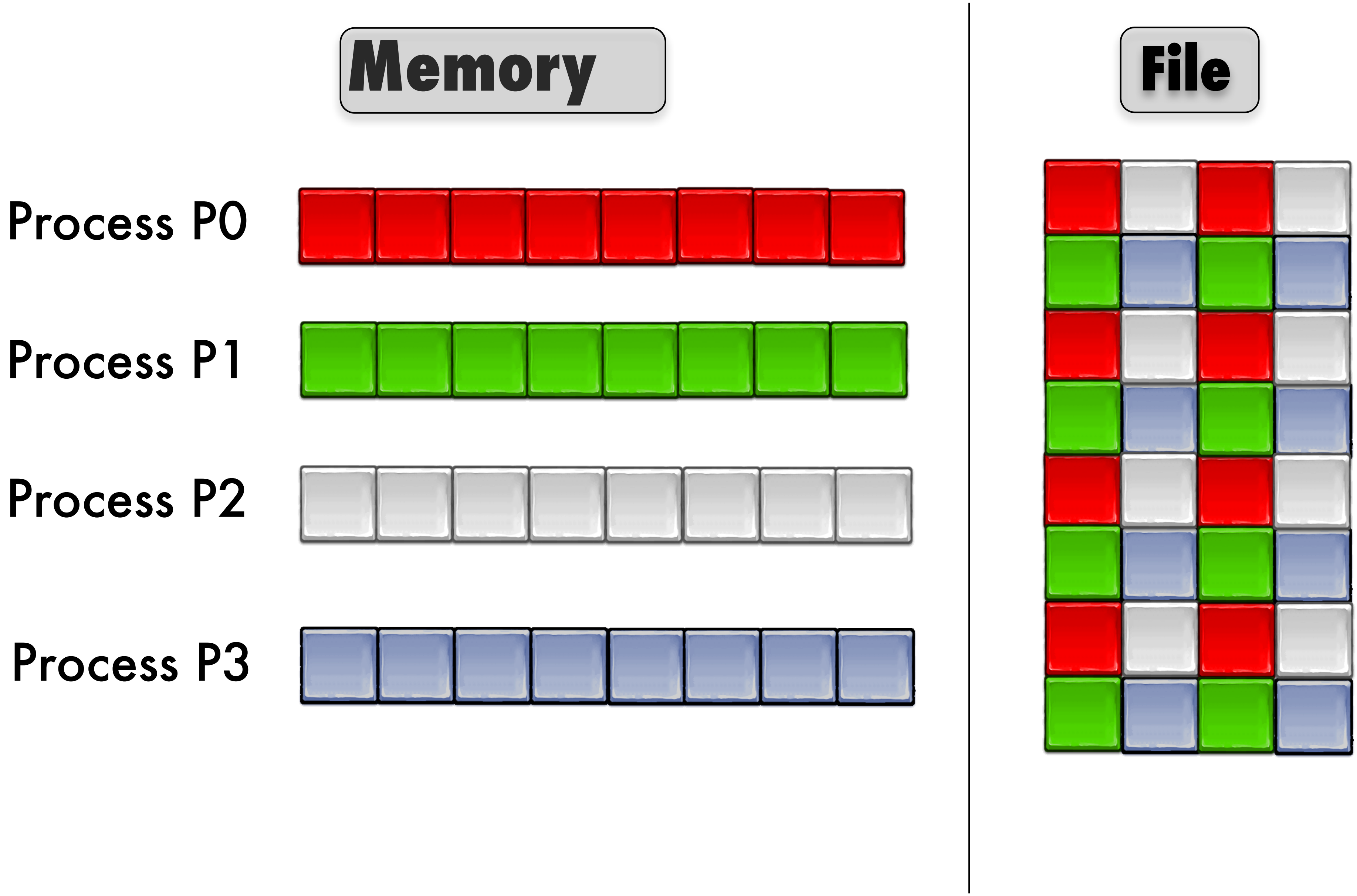
Example 2: Writing dataset by column

```
/*
 * Each process defines a hyperslab in
    * the file
 */
   count[0]   = 1;
   count[1]   = dimsm[1];
   offset[0]  = 0;
   offset[1]  = mpi_rank;
   stride[0]  = 1;
   stride[1]  = 2;
   block[0]   = dimsf[0];
   block[1]   = 1;

/*
 * Each process selects a hyperslab.
    */
   filespace = H5Dget_space(dset_id);

   H5Sselect_hyperslab(filespace,
         H5S_SELECT_SET, offset, stride,
         count, block);
```

Writing by columns: Output of h5dump

```
HDF5 "SDS_col.h5" {
GROUP "/" {
   DATASET "IntArray" {
      DATATYPE  H5T_STD_I32BE
      DATASPACE  SIMPLE { ( 8, 6 ) / ( 8, 6 ) }
      DATA {
         1, 2, 10, 20, 100, 200,
         1, 2, 10, 20, 100, 200,
         1, 2, 10, 20, 100, 200,
         1, 2, 10, 20, 100, 200,
         1, 2, 10, 20, 100, 200,
         1, 2, 10, 20, 100, 200,
         1, 2, 10, 20, 100, 200,
         1, 2, 10, 20, 100, 200
      }
   }
 }
}
```

Example 3: *Writing dataset by pattern*

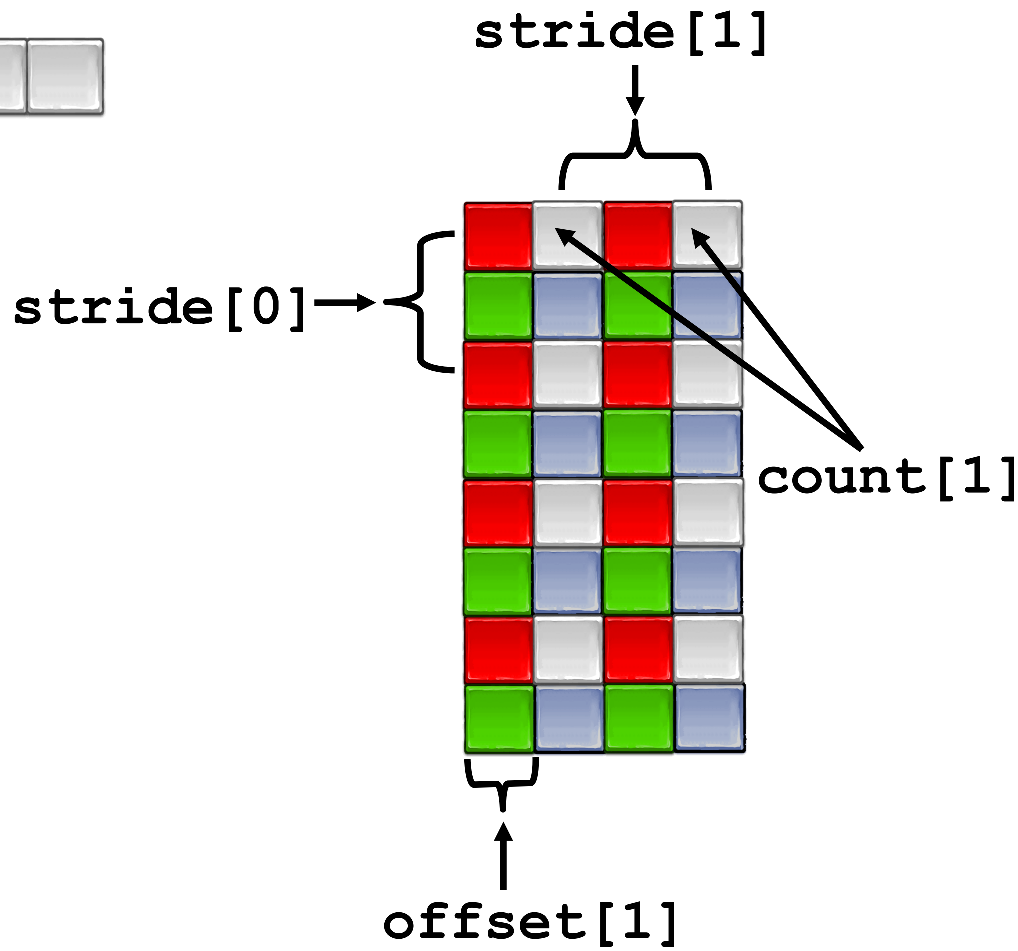


Example 3: Writing dataset by pattern

Memory

File

Process P2 




```
offset[0] = 0;
offset[1] = 1;
count[0]  = 4;
count[1]  = 2;
stride[0] = 2;
stride[1] = 2;
```

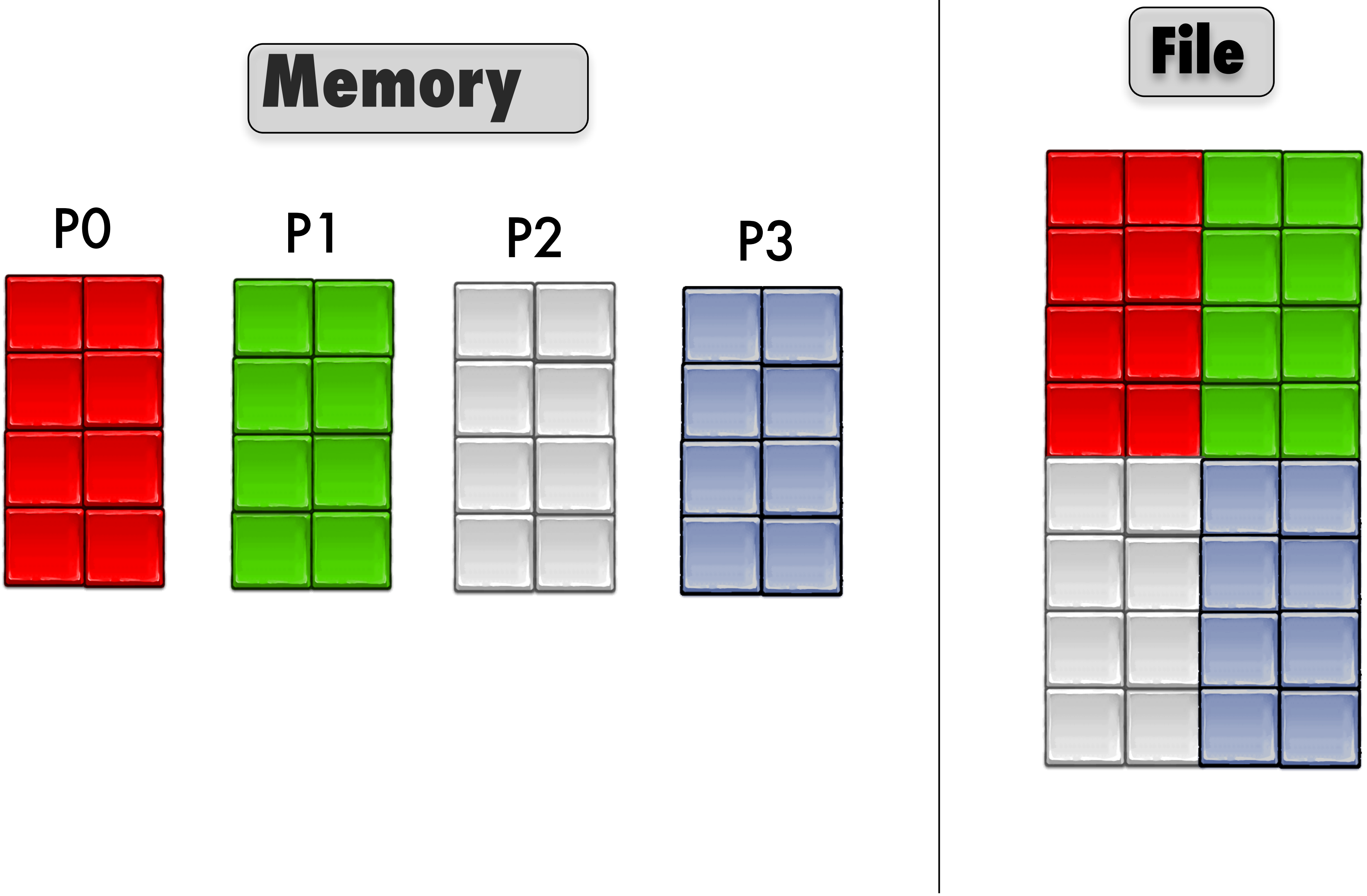
Example 3: Writing by pattern

```
/* Each process defines dataset in memory and
 * writes it to the hyperslab in the file.
 */
count[0] = 4;
count[1] = 2;
stride[0] = 2;
stride[1] = 2;
if(mpi_rank == 0) {
    offset[0] = 0;
    offset[1] = 0;
}
if(mpi_rank == 1) {
    offset[0] = 1;
    offset[1] = 0;
}
if(mpi_rank == 2) {
    offset[0] = 0;
    offset[1] = 1;
}
if(mpi_rank == 3) {
    offset[0] = 1;
    offset[1] = 1;
}
```

Writing by Pattern: *Output of h5dump*

```
HDF5 "SDS_pat.h5" {
GROUP "/" {
   DATASET "IntArray" {
      DATATYPE  H5T_STD_I32BE
      DATASPACE  SIMPLE { ( 8, 4 ) / ( 8, 4 ) }
      DATA {
         1, 3, 1, 3,
         2, 4, 2, 4,
         1, 3, 1, 3,
         2, 4, 2, 4,
         1, 3, 1, 3,
         2, 4, 2, 4,
         1, 3, 1, 3,
         2, 4, 2, 4
      }
   }
}
}
```

Example 4: Writing dataset by chunks

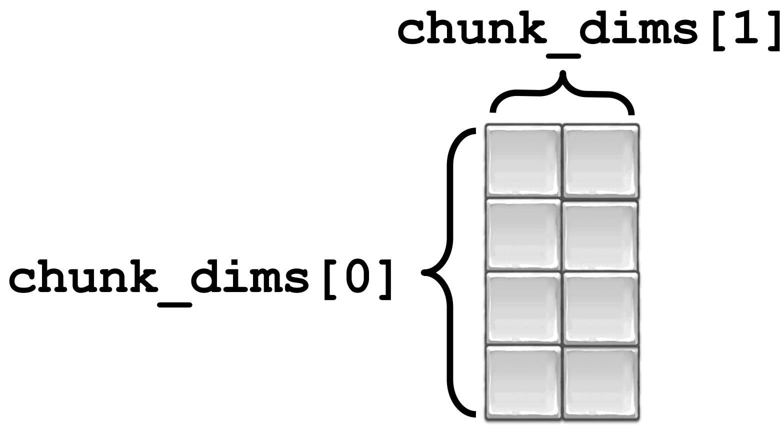


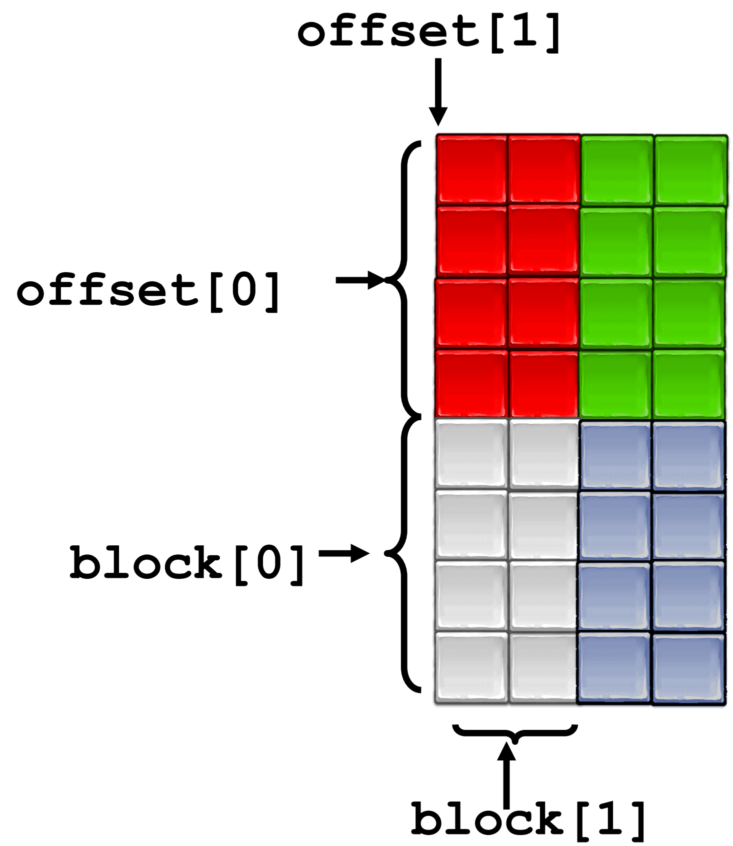
Example 4: Writing dataset by chunks

Memory

File

Process P2






```
block[0]  = chunk_dims[0];
block[1]  = chunk_dims[1];
offset[0] = chunk_dims[0];
offset[1] = 0;
```

Example 4: Writing by chunks

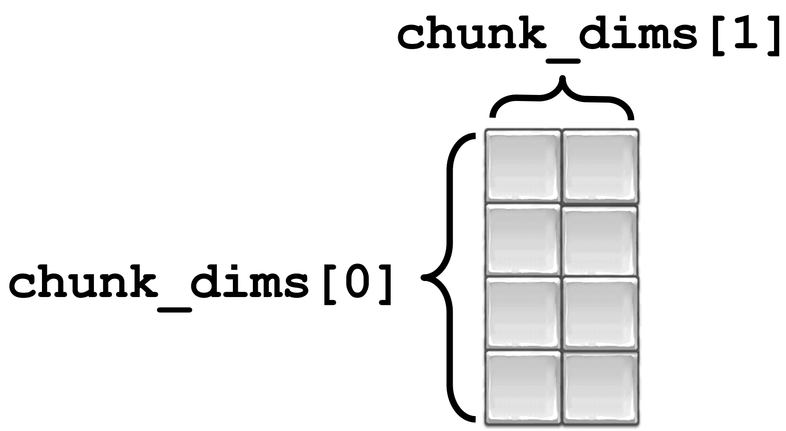
```
count[0] = 1;
      count[1] = 1 ;
      stride[0] = 1;
      stride[1] = 1;
      block[0] = chunk_dims[0];
      block[1] = chunk_dims[1];
      if(mpi_rank == 0) {
         offset[0] = 0;
         offset[1] = 0;
      }
      if(mpi_rank == 1) {
         offset[0] = 0;
         offset[1] = chunk_dims[1];
      }
      if(mpi_rank == 2) {
         offset[0] = chunk_dims[0];
         offset[1] = 0;
      }
      if(mpi_rank == 3) {
         offset[0] = chunk_dims[0];
         offset[1] = chunk_dims[1];
      }
```

Writing by Chunks: Output of h5dump

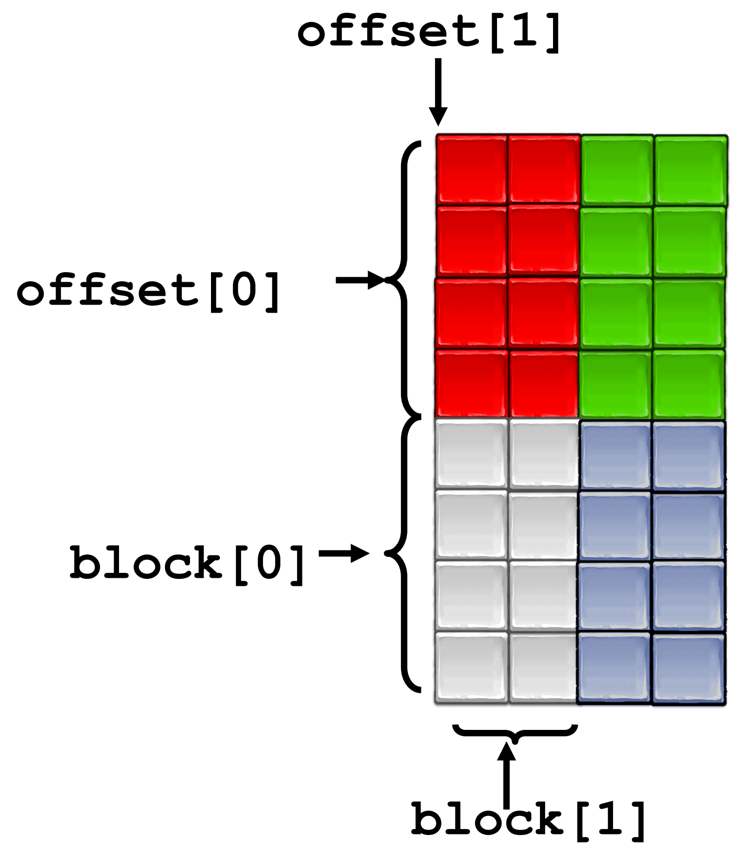
```
HDF5 "SDS_chnk.h5" {
GROUP "/" {
   DATASET "IntArray" {
      DATATYPE  H5T_STD_I32BE
      DATASPACE  SIMPLE { ( 8, 4 ) / ( 8, 4 ) }
      DATA {
         1, 1, 2, 2,
         1, 1, 2, 2,
         1, 1, 2, 2,
         1, 1, 2, 2,
         3, 3, 4, 4,
         3, 3, 4, 4,
         3, 3, 4, 4,
         3, 3, 4, 4
      }
   }
}
}
```

Demo

Memory



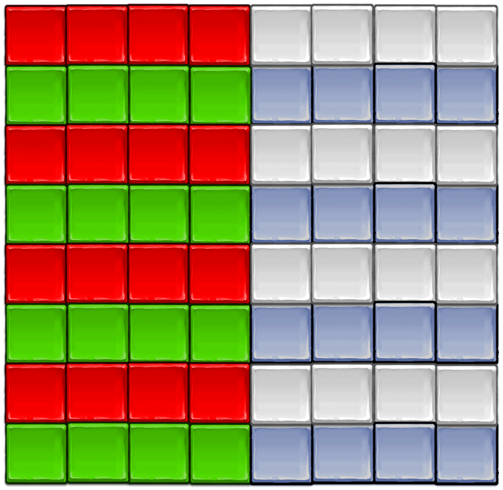
File

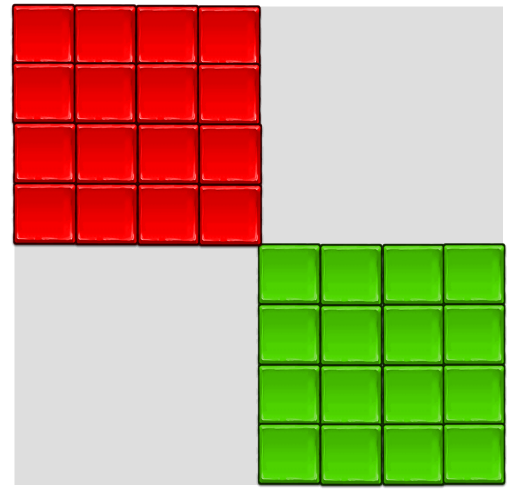


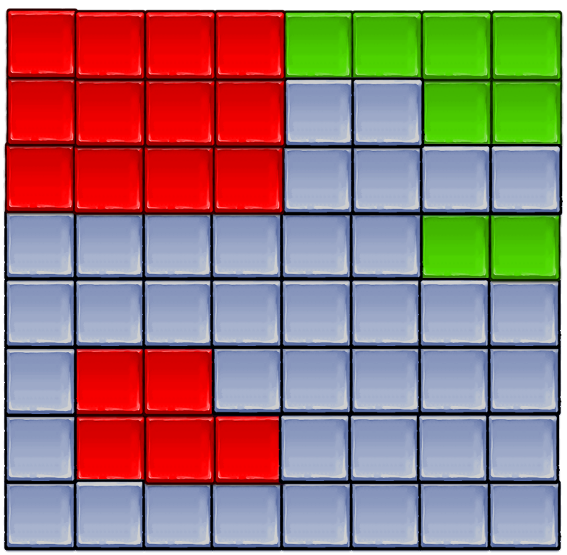

```
block[0]  = chunk_dims[0];
block[1]  = chunk_dims[1];
offset[0] = chunk_dims[0];
offset[1] = 0;
```

Complex data patterns

HDF5 doesn’t have restrictions on data patterns and balance



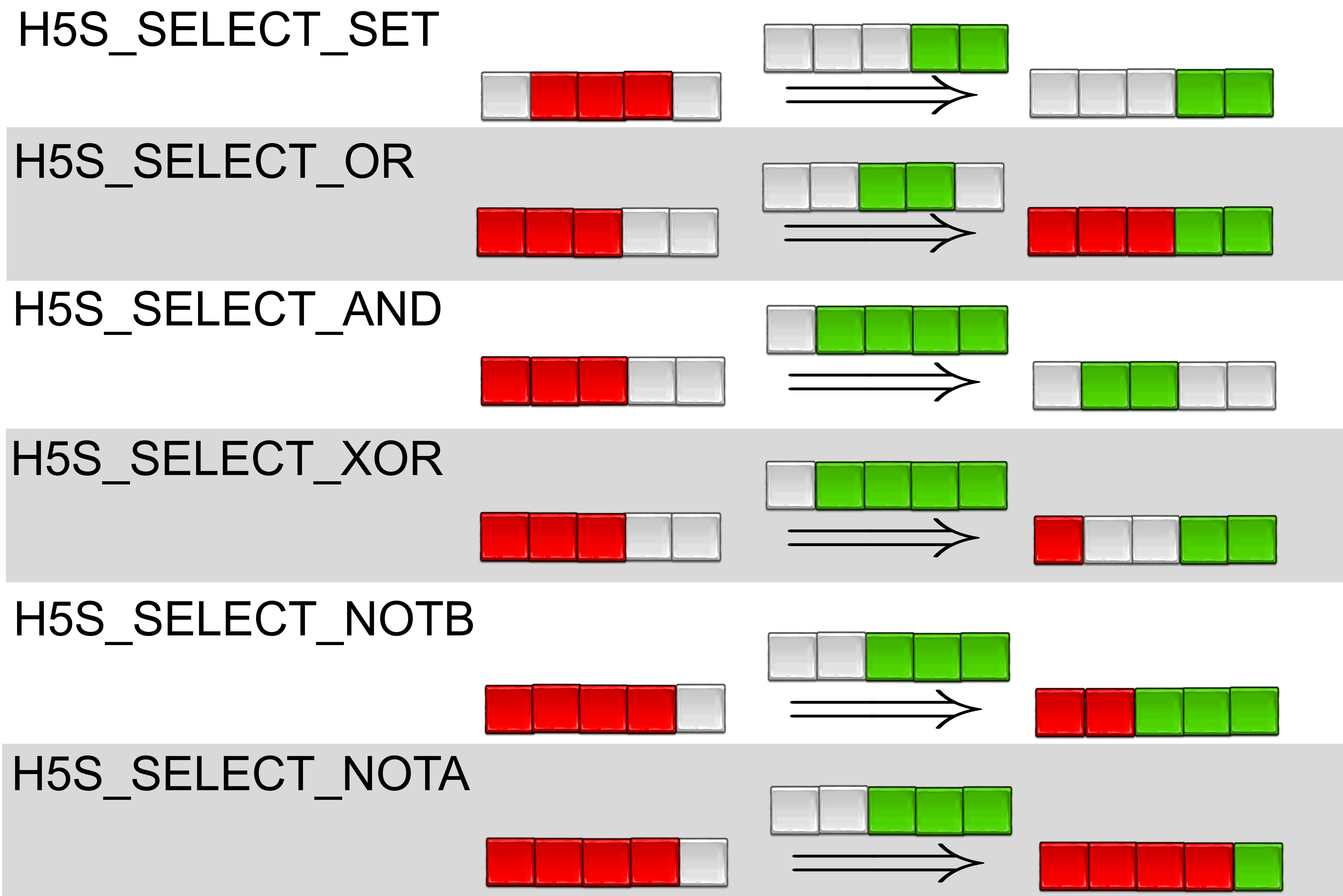




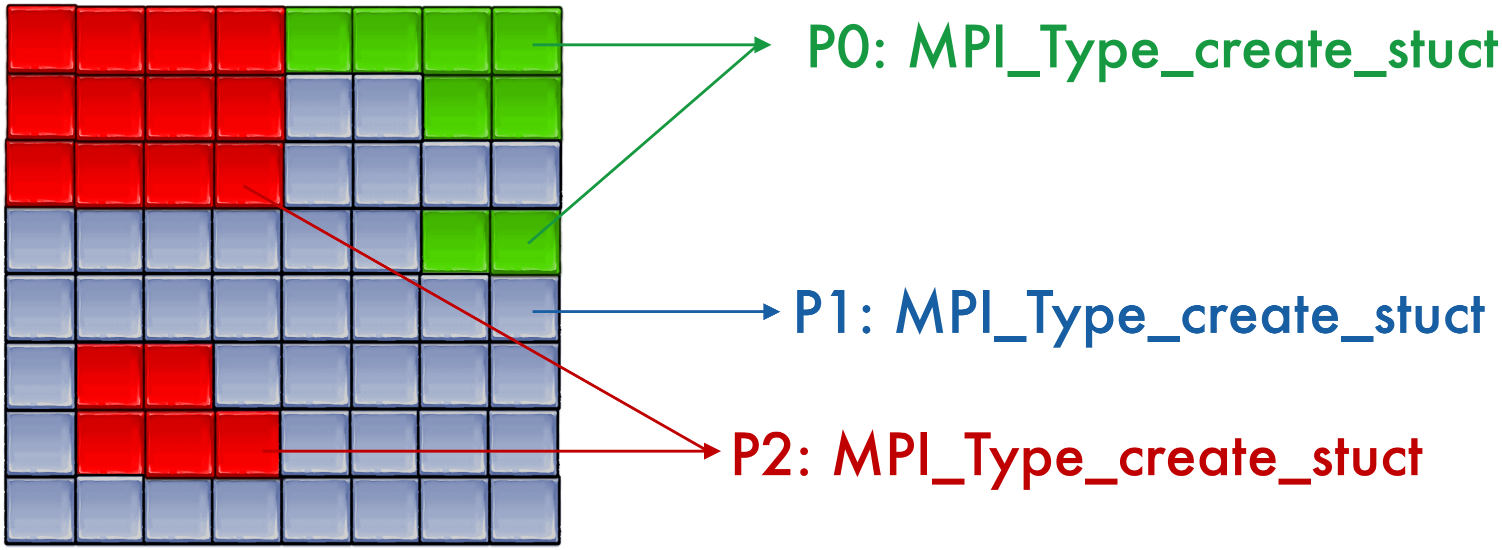
- Irregular hyperslabs created by union operators

```
H5Sselect_hyperslab(space_id, op, start, stride, count, block )
```

Complex data patterns – Selection



Examples of irregular selection



Internally…

1. The HDF5 library creates an MPI datatype for each lower dimension in the selection
2. It then combines those types into one giant structured MPI datatype

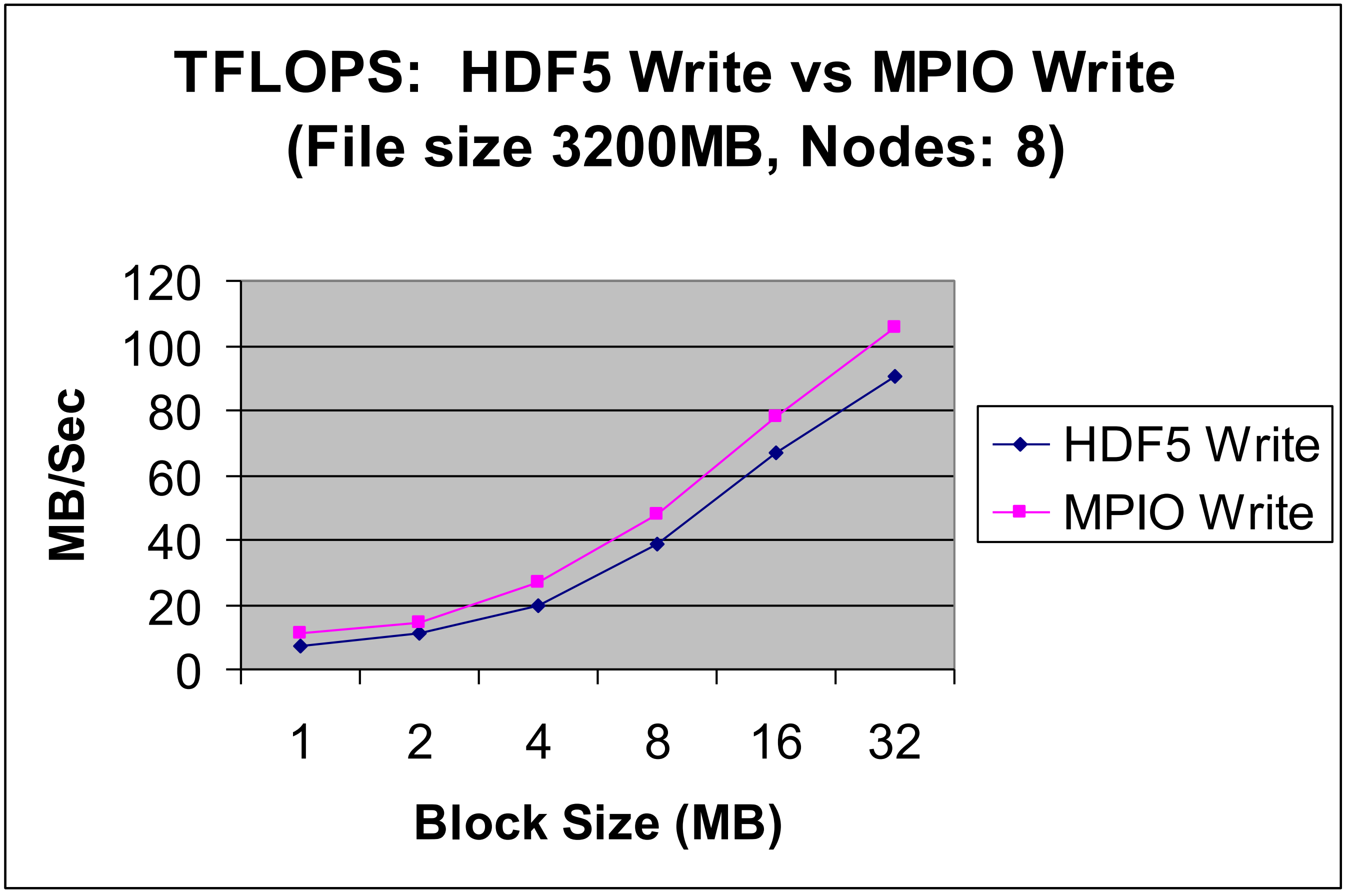
PERFORMANCE ANALYSIS

My PHDF5 Application I/O is slow

Common Solutions to poor performance

- Use larger I/O data sizes
- Independent vs. Collective I/O
- Specific I/O system hints

Write Speed vs. Block Size



Tip: Minimize I/O calls by performing large data I/O

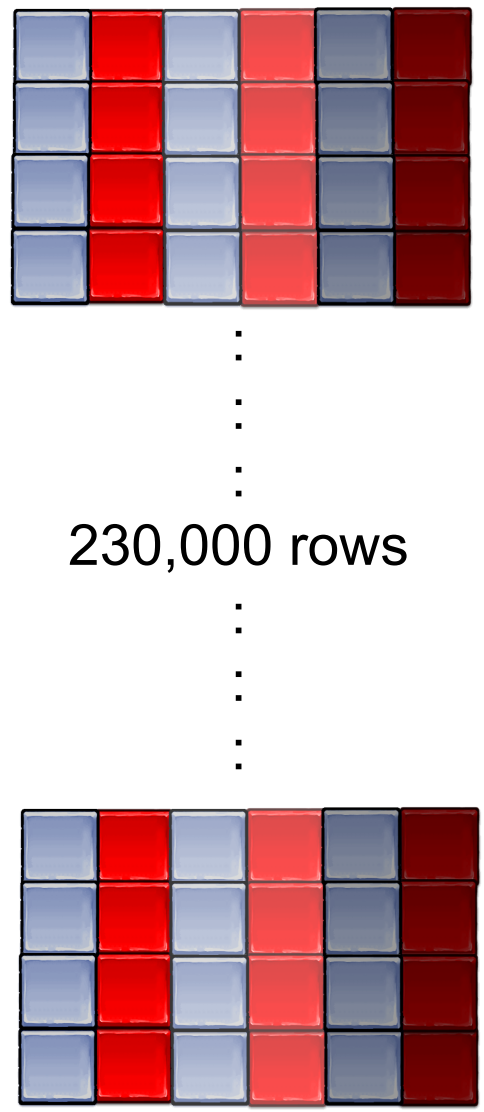
My PHDF5 Application I/O is slow

- Use larger I/O data sizes
- **Independent vs. Collective I/O**
- Specific I/O system hints

Independent vs. Collective Access

User Report

- Independent data transfer mode is much slower than the Collective data transfer mode
- Data array is tall and thin: 230,000 rows by 6 columns



Debug Slow Parallel I/O Speed(1)

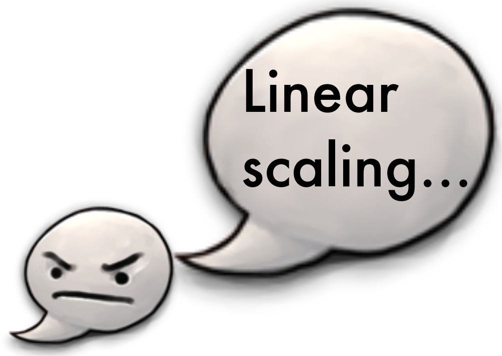
- Writing to one dataset
  - Using 4 processes == 4 columns
  - Datatype is 8-byte float (doubles)
  - 4 processes x1000 rows x 8 bytes = 32,000 bytes

mpirun -np 4 ./a.out 1000

  - Execution time: 1.783798 s

mpirun -np 4 ./a.out 2000

  - Execution time: 3.838858 s



- 2 seconds for 1000 more rows = 32,000 bytes.

Speed of 16KB/sec!!! *Way too slow.*

Debug slow parallel I/O speed(2)

- Build a version of PHDF5 with

```
./configure --enable-build-mode=debug --enable-parallel …
```

  - This allows the tracing of MPIO I/O calls in the HDF5 library such as MPI_File_read_xx and MPI_File_write_xx

- To trace

```
setenv H5FD_mpio_Debug "rw"
```

The report will look something like this...

Debug slow parallel I/O speed(3)

```
setenv H5FD_mpio_Debug 'rw'
mpirun -np 4 ./a.out 1000        # Indep.; contiguous.
in H5FD_mpio_write  mpi_off=0  size_i=96
in H5FD_mpio_write  mpi_off=0  size_i=96
in H5FD_mpio_write  mpi_off=0  size_i=96
in H5FD_mpio_write  mpi_off=0  size_i=96
in H5FD_mpio_write  mpi_off=2056  size_i=8
in H5FD_mpio_write  mpi_off=2048  size_i=8
in H5FD_mpio_write  mpi_off=2072  size_i=8
in H5FD_mpio_write  mpi_off=2064  size_i=8
in H5FD_mpio_write  mpi_off=2088  size_i=8
in H5FD_mpio_write  mpi_off=2080  size_i=8
...
```

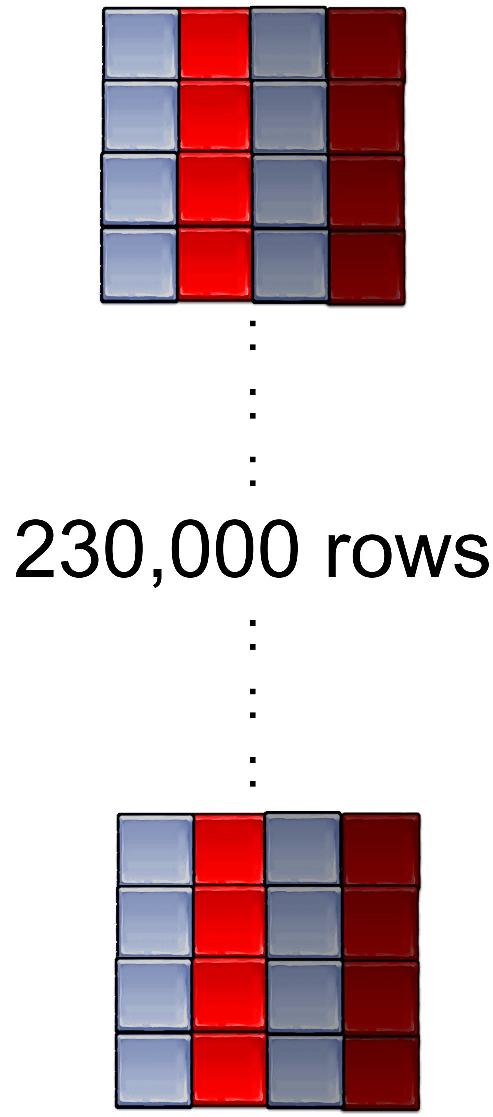
- Total of 4000 of these little 8 bytes writes == 32,000 bytes.

Independent calls are many and small

Diagnosis

- Each process writes one element of one row, skips to next row, writes one element, and so on…

Each process issues 230,000 writes of 8 bytes each.

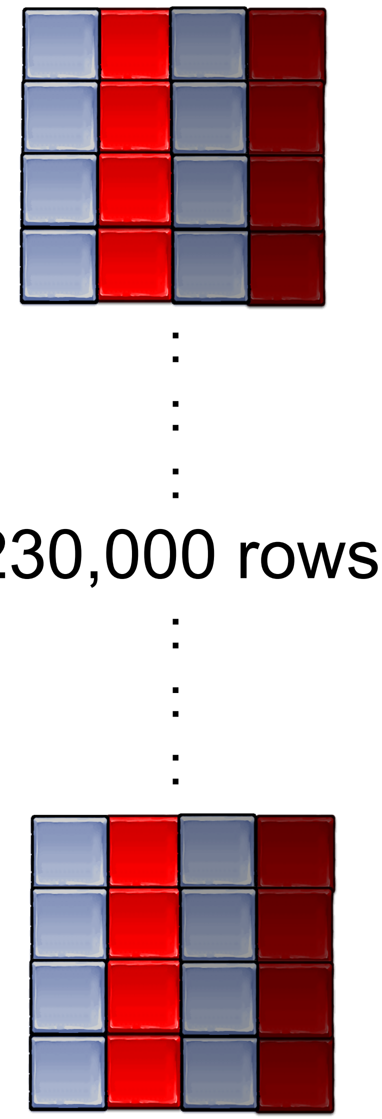


Use Collective Mode or Chunked Storage

Remedy

- Collective I/O mode will combine many small independent calls into few but bigger calls

- Chunks of columns speeds up too



Debug slow parallel I/O speed (4)

```
setenv H5FD_mpio_Debug 'rw'
mpirun -np 4 ./a.out 1000
```

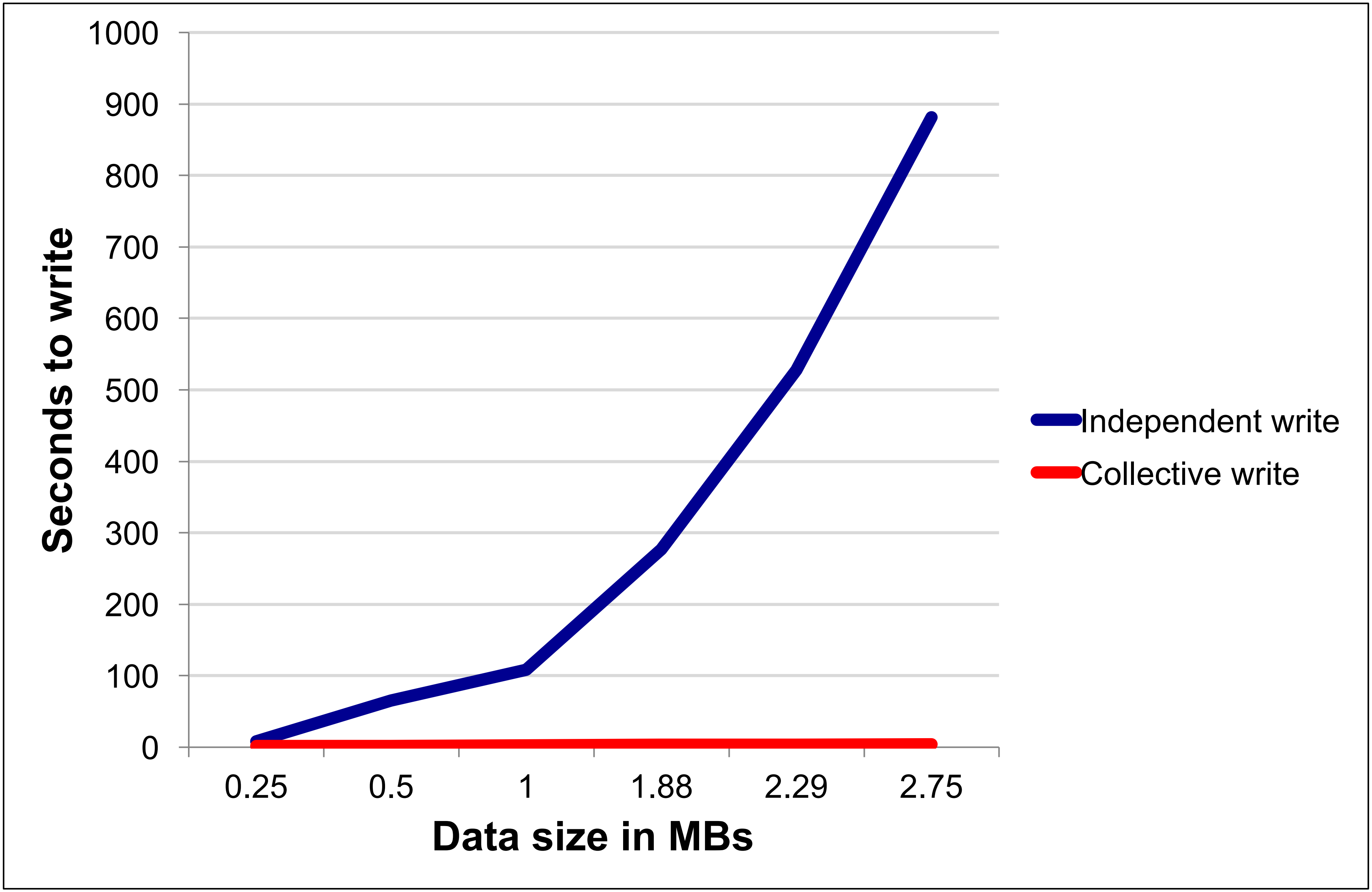
Indep., Chunked by column.

in H5FD_mpio_write	mpi_off=0	size_i=96
in H5FD_mpio_write	mpi_off=0	size_i=96
in H5FD_mpio_write	mpi_off=0	size_i=96
in H5FD_mpio_write	mpi_off=0	size_i=96
in H5FD_mpio_write	mpi_off=3688	size_i=8000
in H5FD_mpio_write	mpi_off=11688	size_i=8000
in H5FD_mpio_write	mpi_off=27688	size_i=8000
in H5FD_mpio_write	mpi_off=19688	size_i=8000
in H5FD_mpio_write	mpi_off=96	size_i=40
in H5FD_mpio_write	mpi_off=136	size_i=544
in H5FD_mpio_write	mpi_off=680	size_i=120
in H5FD_mpio_write	mpi_off=800	size_i=272

...

Execution time: 0.011599 s.

Collective vs. independent write



Collective I/O in HDF5

- Set-up using a Data Transfer Property List (DXPL)
- All processes must participate in the I/O call (H5Dread/write) with a selection (which could be a NULL selection)
- Some cases where collective I/O is not used even when the use asks for it:
  - Data conversion
  - Compressed Storage
  - Chunking Storage:
    - When the chunk is not selected by a certain number of processes

Collective I/O in HDF5

- Can query Data Transfer Property List (DXPL) after I/O for collective I/O status:
  - *H5Pget_mpio_actual_io_mode*
    - Retrieves the type of I/O that HDF5 actually performed on the last parallel I/O call
  - *H5Pget_mpio_no_collective_cause*
    - Retrieves local and global causes that broke collective I/O on the last parallel I/O call
  - *H5Pget_mpio_actual_chunk_opt_mode*
    - Retrieves the type of chunk optimization that HDF5 actually performed on the last parallel I/O call. This is not necessarily the type of optimization requested

Enabling Collective Parallel I/O

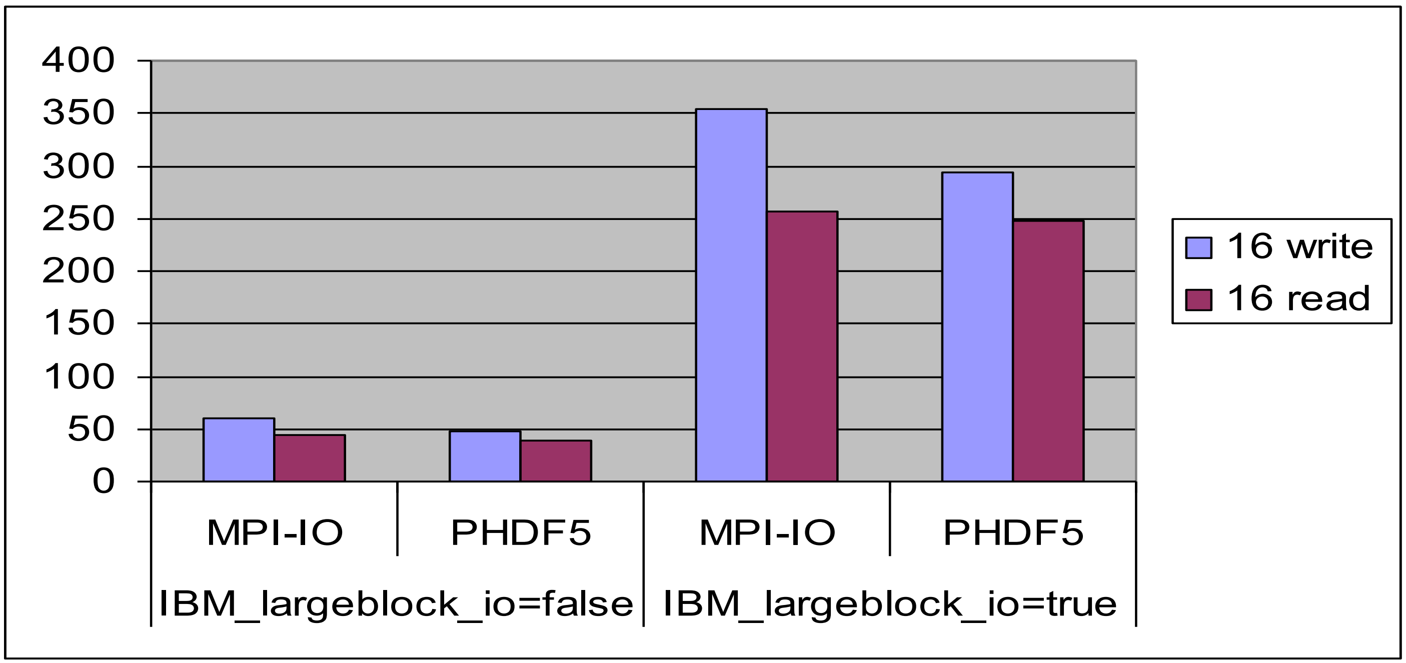
```
/* Set up file access property list w/parallel I/O access */
fa_plist_id = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_fapl_mpio(fa_plist_id, comm, info);

/* Create a new file collectively */
file_id = H5Fcreate(filename, H5F_ACC_TRUNC,
      H5P_DEFAULT, fa_plist_id);

/* <omitted data decomposition for brevity> */

/* Set up data transfer property list w/collective MPI-IO */
dx_plist_id = H5Pcreate(H5P_DATASET_XFER);
H5Pset_dxpl_mpio(dx_plist_id, H5FD_MPIO_COLLECTIVE);

/* Write data elements to the dataset */
status = H5Dwrite(dset_id, H5T_NATIVE_INT,
      memspace, filespace, dx_plist_id, data);
```

My PHDF5 Application I/O is slow

- Use larger I/O data sizes
- Independent vs. Collective I/O
- **Specific I/O system hints**

Effects of I/O Hints: IBM_largeblock_io

- GPFS at LLNL ASCI Blue machine
  - 4 nodes, 16 tasks
  - Total data size 1024MB
  - I/O buffer size 1MB



Parallel I/O Profiling Tools

- Two kinds of tools
  - I/O benchmarks for measuring a system's I/O capabilities
  - I/O profilers for characterizing applications' I/O behavior

- Couple of examples
  - H5perf (in the HDF5 source code distro)
  - Darshan (from Argonne National Laboratory)

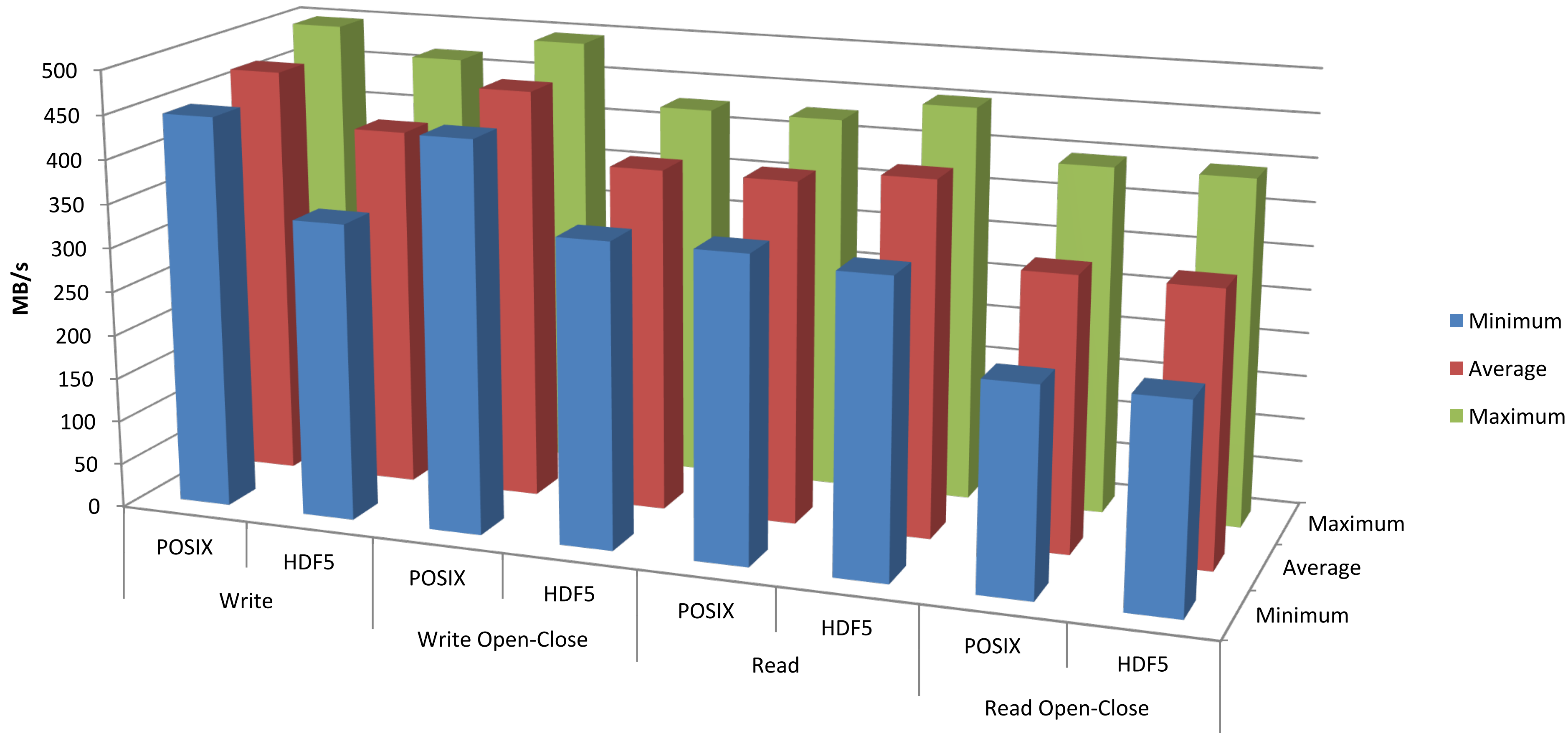
- Profilers have to compromise between
  - A lot of detail → large trace files and overhead
  - Aggregation → loss of detail, but low overhead

h5perf

- Measures filesystem performance for different I/O patterns and APIs
- Three File I/O types in one convent package
  1. POSIX I/O (open/write/read/close...)
  2. MPI I/O (MPI_File_{open,write,read,close})
  3. PHDF5
- An indication of I/O speed ranges and HDF5 overhead

A Serial Run

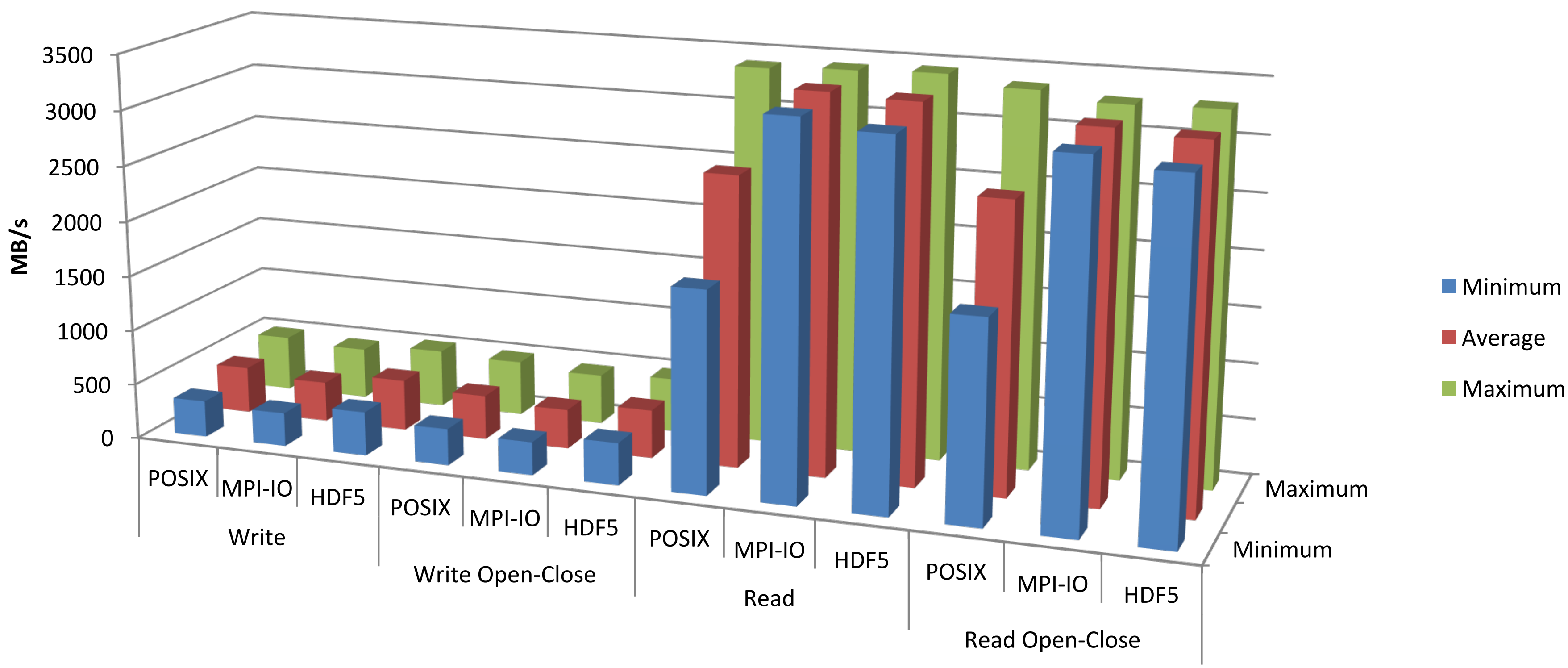
h5perf_serial, 3 iterations, 1 GB dataset, 1 MB transfer buffer, HDF5 dataset contiguous storage, HDF5 SVN trunk, NCSA BW



h5perf

A Parallel Run

h5perf, 3 MPI processes, 3 iterations, 3 GB dataset (total), 1 GB per process, 1 GB transfer buffer, HDF5 dataset contiguous storage, HDF5 SVN trunk, NCSA BW



h5perf: example output 1/3

```
mpirun -np 4 h5perf        # Ran in a Linux system
Number of processors = 4
    Transfer Buffer Size: 131072 bytes, File size: 1.00 MBs
      # of files: 1, # of datasets: 1, dataset size: 1.00 MBs
        IO API = POSIX
            Write (1 iteration(s)):
                Maximum Throughput:  18.75 MB/s
                Average Throughput:  18.75 MB/s
                Minimum Throughput:  18.75 MB/s
            Write Open-Close (1 iteration(s)):
                Maximum Throughput:  10.79 MB/s
                Average Throughput:  10.79 MB/s
                Minimum Throughput:  10.79 MB/s
            Read (1 iteration(s)):
                Maximum Throughput: 2241.74 MB/s
                Average Throughput: 2241.74 MB/s
                Minimum Throughput: 2241.74 MB/s
            Read Open-Close (1 iteration(s)):
                Maximum Throughput: 756.41 MB/s
                Average Throughput: 756.41 MB/s
                Minimum Throughput: 756.41 MB/s
```

h5perf: example output 2/3

```
mpirun -np 4 h5perf
…
        IO API = MPIO
            Write (1 iteration(s)):
                Maximum Throughput: 611.95 MB/s
                Average Throughput: 611.95 MB/s
                Minimum Throughput: 611.95 MB/s
            Write Open-Close (1 iteration(s)):
                Maximum Throughput:  16.89 MB/s
                Average Throughput:  16.89 MB/s
                Minimum Throughput:  16.89 MB/s
            Read (1 iteration(s)):
                Maximum Throughput: 421.75 MB/s
                Average Throughput: 421.75 MB/s
                Minimum Throughput: 421.75 MB/s
            Read Open-Close (1 iteration(s)):
                Maximum Throughput: 109.22 MB/s
                Average Throughput: 109.22 MB/s
                Minimum Throughput: 109.22 MB/s
```

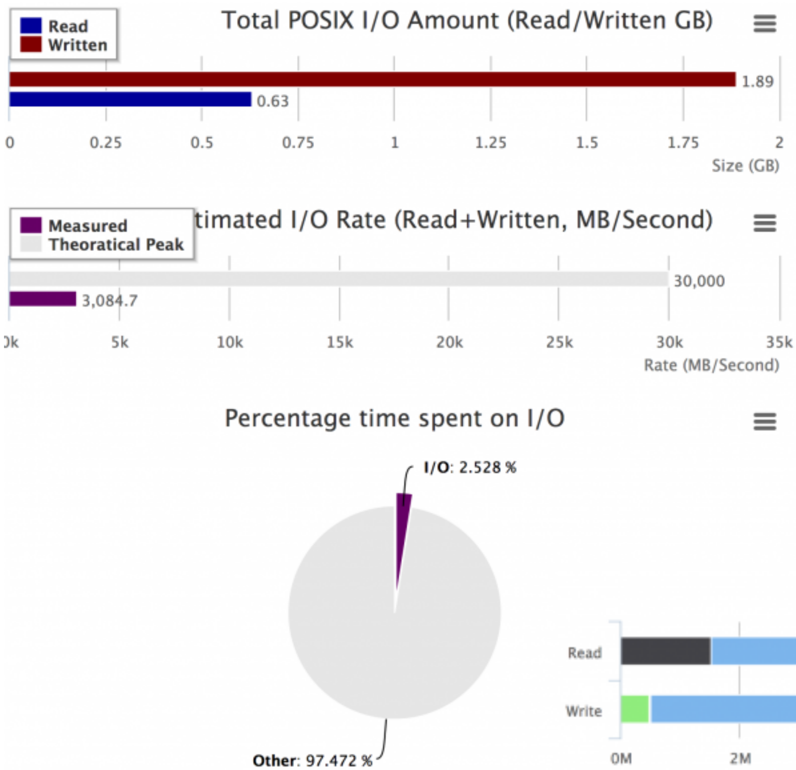
h5perf: example output 3/3

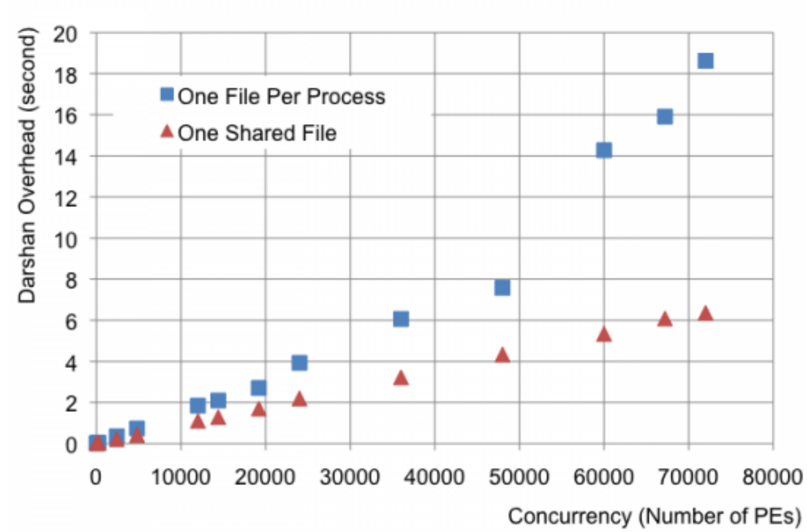
```
mpirun -np 4 h5perf
...
        IO API = PHDF5 (w/MPI-I/O driver)
            Write (1 iteration(s)):
                Maximum Throughput: 304.40 MB/s
                Average Throughput: 304.40 MB/s
                Minimum Throughput: 304.40 MB/s
            Write Open-Close (1 iteration(s)):
                Maximum Throughput:  15.14 MB/s
                Average Throughput:  15.14 MB/s
                Minimum Throughput:  15.14 MB/s
            Read (1 iteration(s)):
                Maximum Throughput: 1718.27 MB/s
                Average Throughput: 1718.27 MB/s
                Minimum Throughput: 1718.27 MB/s
            Read Open-Close (1 iteration(s)):
                Maximum Throughput:  78.06 MB/s
                Average Throughput:  78.06 MB/s
                Minimum Throughput:  78.06 MB/s
    Transfer Buffer Size: 262144 bytes, File size: 1.00 MBs
      # of files: 1, # of datasets: 1, dataset size: 1.00 MBs
```

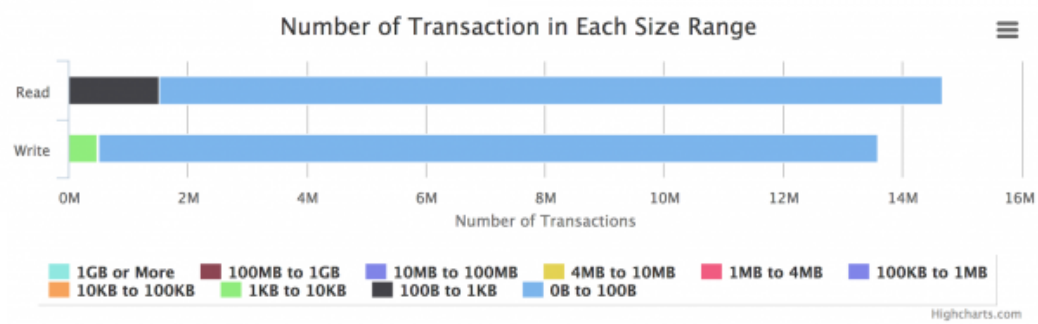
Darshan (ANL)

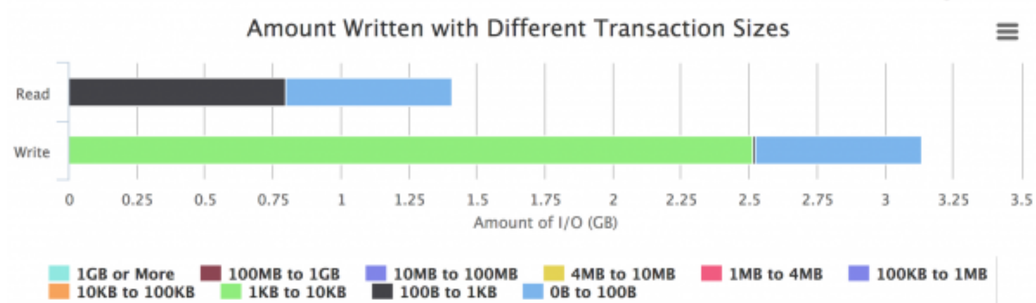
- Design goals:
  - Transparent integration with user environment
  - Negligible impact on application performance
- Provides aggregate figures for:
  - Operation counts (POSIX, MPI-IO, HDF5, PnetCDF, ...)
  - Datatypes and hint usage
  - Access patterns: alignments, sequentially, access size
  - Cumulative I/O time, intervals of I/O activity
- Does not provide I/O behavior over time
- Excellent starting point, maybe not your final stop

Darshan Sample Output









Source: NERSC

Lastly, Don't reinvent the wheel...

- Make use of libraries which utilize HDF, but represent the scientific data using a set of conventions, i.e., standard way for
  - Representing meshes
  - Variable definitions
  - Multiple datasets
  - Component definitions
- Some high-level parallel formats by field using HDF
  - Computation Fluid Dynamics: CGNS
  - Meshless/Particle Methods: H5Part
  - Finite element method: MOAB
  - Earth science: NetCDF

- Hides the complexity of PHDF5, but still must know concepts of parallel I/O and the underlying file systems

Examples

CGNS

Reference:

Parallel and Large-scale Simulation Enhancements to CGNS, By Scot Breitenfeld, The HDF Group, 2015.

CFD Standard

- CGNS = Computational Fluid Dynamics (CFD) General Notation System
- An effort to standardize CFD input and output data including:
  - Grid (both structured and unstructured), flow solution
  - Connectivity, boundary conditions, auxiliary information.
- Two parts:
  - A standard format for recording the data
  - Software that reads, writes, and modifies data in that format.
- An American Institute of Aeronautics and Astronautics Recommended Practice

AIAA
Shaping the Future of Aerospace

CGNS Storage Evolution

- CGNS data was originally stored in ADF ('Advanced Data Format')
  - ADF lacks parallel I/O or data compression capabilities
  - Doesn't have HDF5's support base and tools
- HDF5 superseded ADF as the official storage mechanism for CGNS
- CGNS introduced parallel I/O APIs w/ parallel HDF5 in 2013
  - However, poor performance of the new parallel APIs in most circumstances
  - In 2014, NASA provided funding for The HDF Group with the goal to improve the under-performing parallel capabilities of the CGNS library

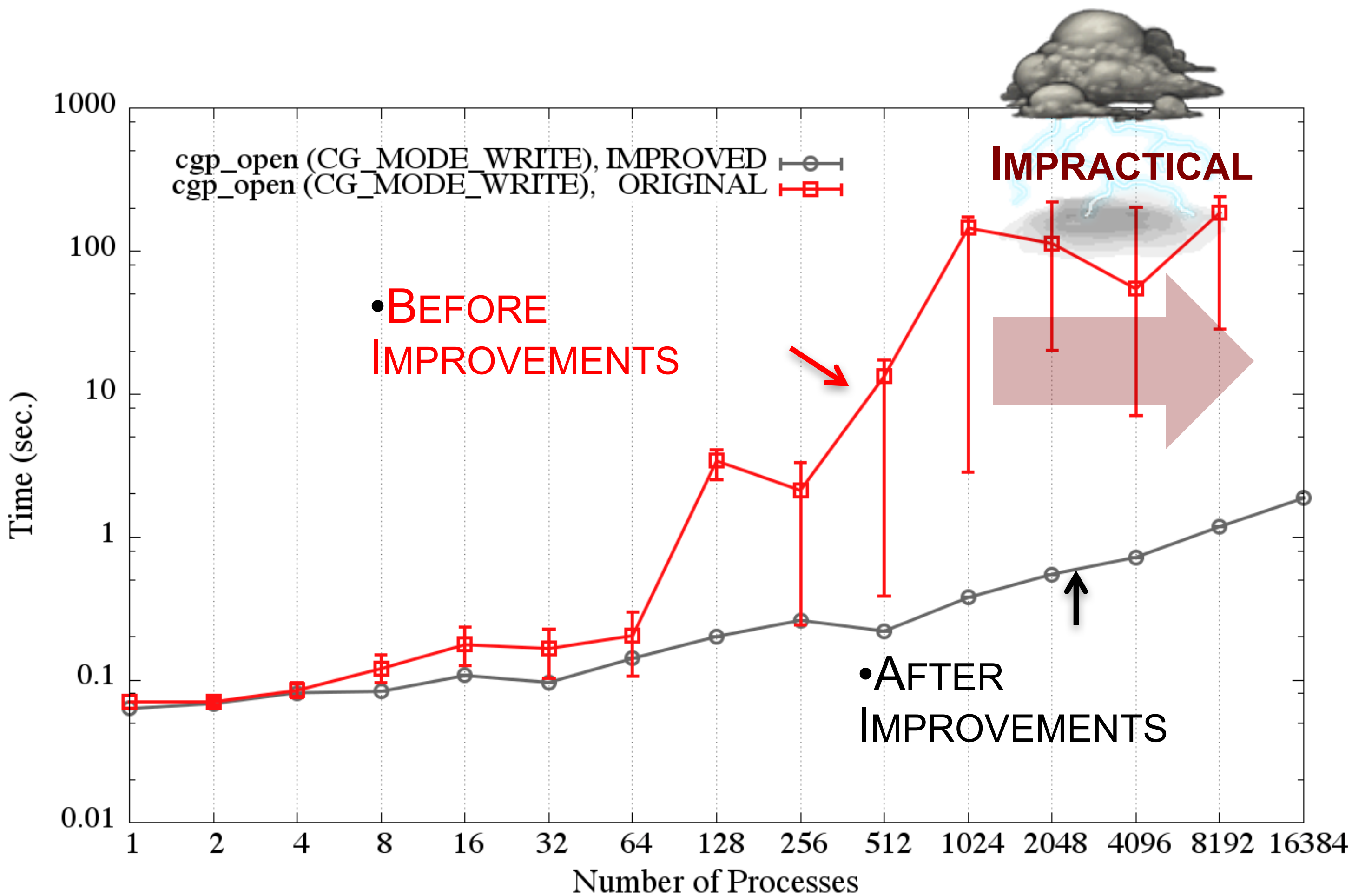
CGNS Performance Problems

- Opening an existing file
  - CGNS reads the entire HDF5 file structure, loading a lot of (HDF5) metadata
  - Reads occur independently on ALL ranks competing for the same metadata

    ➔Termed "Read Storm"

- Closing a CGNS file
  - Triggers HDF5 flush of a large amount of small metadata entries
  - Implemented as iterative, independent writes, an unsuitable workload for parallel file systems

Opening CGNS File ...



Metadata Read Storm Problem (I)

- All metadata "write" operations are required to be collective:

```
if(0 == rank)
    H5Dcreate("dataset1");
else if(1 == rank)
    H5Dcreate("dataset2");
```

```
/* All ranks have to call */
H5Dcreate("dataset1");
H5Dcreate("dataset2");
```

- Metadata read operations are not required to be collective:

```
•if(0 == rank)
•    H5Dopen("dataset1");
•else if(1 == rank)
•    H5Dopen("dataset2");
```

```
•/* All ranks have to call */
•H5Dopen("dataset1");
•H5Dopen("dataset2");
```

Metadata Read Storm Problem (II)

- Metadata read operations are treated by the library as independent read operations.
- Consider a very large MPI job size where all processes want to open a dataset that already exists in the file.

All processes…

- Call `H5Dopen("/G1/G2/D1");`
- Read the same metadata to get to the dataset and the metadata of the dataset itself
- IF metadata not in cache, THEN read it from disk.
- Might issue read requests to the file system for the same small metadata.

Read Storm

Avoiding a Read Storm

- Application sets hint that metadata access is done collectively
  - A property on an access property list
  - If set on the file access property list, then all metadata read operations will be required to be collective
- Can be set on individual object property list
- If set, MPI rank 0 will issue the read for a metadata entry to the file system and broadcast to all other ranks

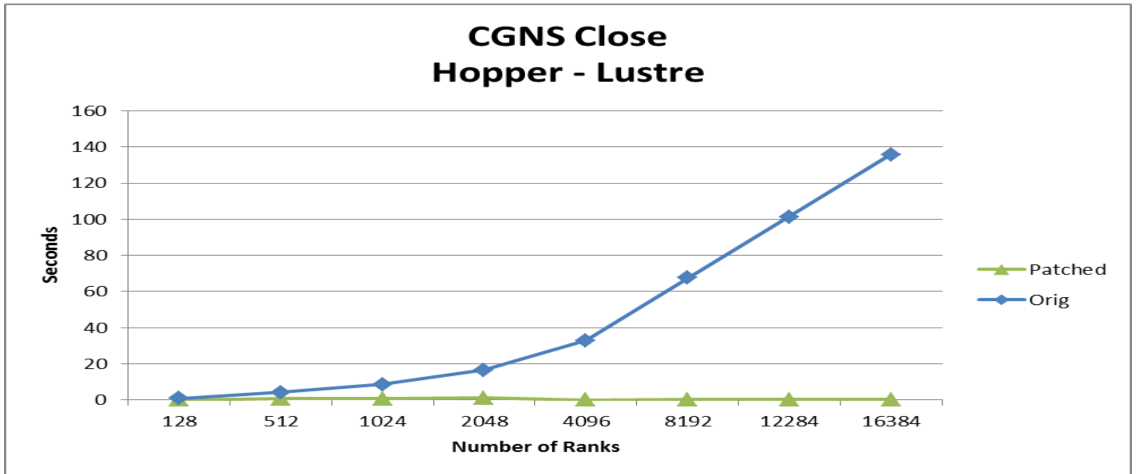
Write Metadata Collectively!

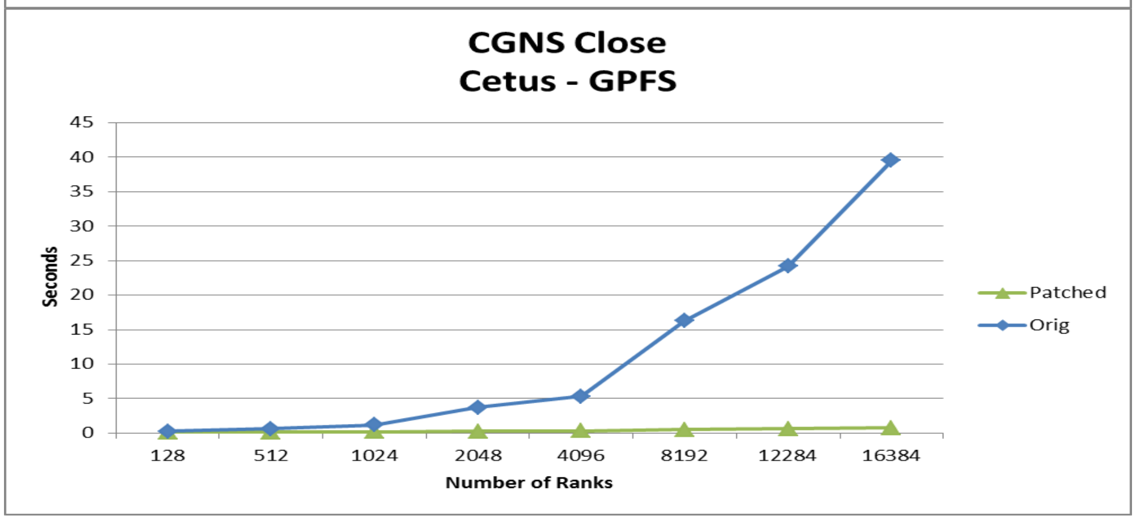
- **Symptoms:** Many users reported that `H5Fclose()` is very slow and doesn't scale well on parallel file systems.

- **Diagnosis:** HDF5 metadata cache issues very small accesses (one write per entry). We know that parallel file systems don't do well with small I/O accesses.

- **Solution:** Gather up all the entries of an epoch, create an MPI derived datatype, and issue a single collective MPI write.

Closing a CGNS File …





Useful parallel HDF5 links

- Parallel HDF information site
  http://www.hdfgroup.org/HDF5/PHDF5/
- Parallel HDF5 tutorial available at
  http://www.hdfgroup.org/HDF5/Tutor/
- HDF Help email address
  help@hdfgroup.org

Questions?

Acknowledgements
Support: Lawrence Berkeley Nation Lab – SCIDAC
Graphics: Buuf, CC BY-NC-SA 2.5