Overview of Parallel HDF5 and Performance Tuning in HDF5 Library

HDF and HDF-EOS Workshop VI

Elena Pourmal, Albert Cheng

Outline

- Overview of Parallel HDF5 design
- Setting up parallel environment
- Programming model for
  - Creating and accessing a File
  - Creating and accessing a Dataset
  - Writing and reading Hyperslabs
- Performance tuning in HDF5
- Parallel tutorial available at
  - http://hdf.ncsa.uiuc.edu/HDF5/doc/Tutor

PHDF5 Initial Target

- Support for MPI programming
- Not for shared memory programming
  - Threads
  - OpenMP
- Has some experiments with
  - Thread-safe support for Pthreads
  - OpenMP if called "correctly"

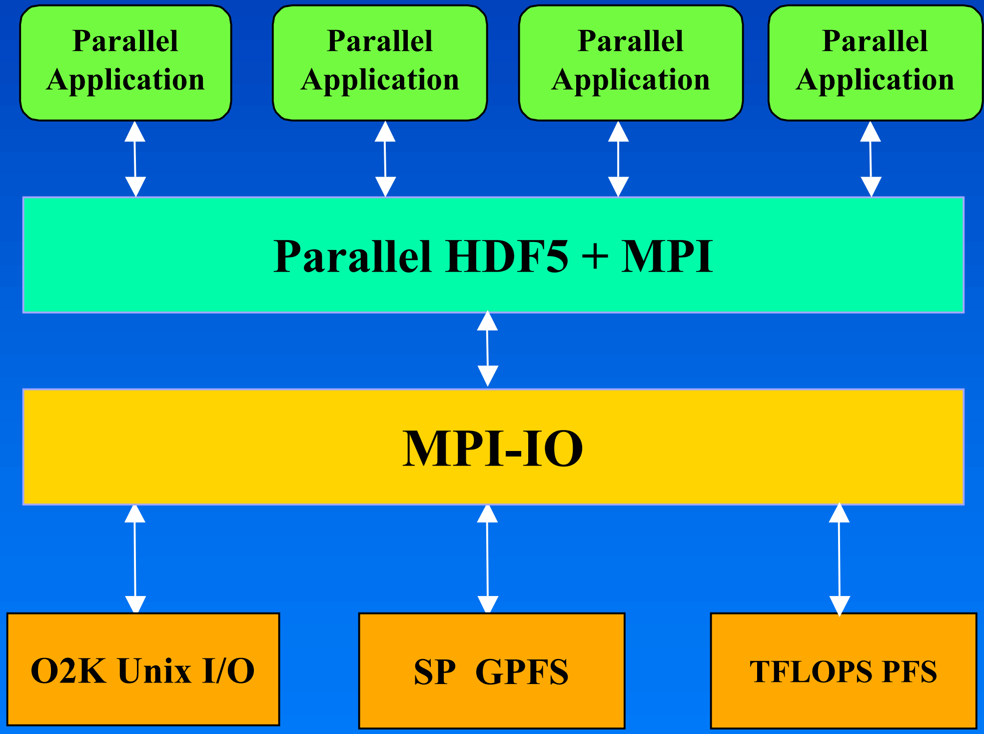
PHDF5 Requirements

- PHDF5 files compatible with serial HDF5 files
  - Shareable between different serial or parallel platforms
- Single file image to all processes
  - One file per process design is undesirable
    - Expensive post processing
    - Not useable by different number of processes
- Standard parallel I/O interface
  - Must be portable to different platforms

Implementation Requirements

- No use of Threads
  - Not commonly supported (1998)
- No reserved process
  - May interfere with parallel algorithms
- No spawn process
  - Not commonly supported even now

PHDF5 Implementation Layers



Parallel Application

Parallel Application

Parallel Application

Parallel Application

Parallel HDF5 + MPI

MPI-IO

O2K Unix I/O

SP GPFS

TFLOPS PFS

User Applications

HDF library

Parallel I/O layer

Parallel File systems

Parallel Environment Requirements

- MPI with MPI-IO
  - Argonne ROMIO
  - Vendor’s MPI-IO
- Parallel file system
  - IBM GPFS
  - PVFS

How to Compile PHDF5

- h5cc _ HDF5 compiler command
  - _Similar to mpicc
- To compile:

```
% h5cc h5prog.c
```

- Show the compiler commands without executing them (i.e., dryrun):

```
% h5cc _show h5prog.c
```

Collective vs. Independent Calls

- MPI definition of collective call
  - All processes of the communicator must participate in the right order
- Independent means not collective
- Collective is not necessarily synchronous

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

Examples of PHDF5 API

- Examples of PHDF5 collective API
  - File operations: H5Fcreate, H5Fopen, H5Fclose
  - Objects creation: H5Dcreate, H5Dopen, H5Dclose
  - Objects structure: H5Dextend (increase dimension sizes)
- Array data transfer can be collective or independent
  - Dataset operations: H5Dwrite, H5Dread

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 write to the same data array
  - Each process writes to individual data array

PHDF5 API Languages

- C and F90 language interfaces
- Platforms supported:
  - IBM SP2 and SP3
  - Intel TFLOPS
  - SGI Origin 2000
  - HP-UX 11.00 System V
  - Alpha Compaq Clusters
  - Linux clusters
  - SUN clusters
  - Cray T3E

Creating and Accessing a File Programming model

- HDF5 uses access template object to control the file access mechanism
- General model to access HDF5 file in parallel:
  - _Setup MPI-IO access template
  - _Open File
  - _Close File

Setup 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);
```

F90:

```
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();
```

F90 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 MPI communicator open/close a dataset by a collective call
  - C: H5Dcreate or H5Dopen; H5Dclose
  - F90: h5dcreate_f or h5dopen_f; h5dclose_f
- All processes of the MPI communicator extend dataset with unlimited dimensions before writing to it
  - C: H5Dextend
  - F90: h5dextend_f

C Example
Parallel Dataset Create

```c
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);
```

F90 Example
Parallel Dataset Create

```
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
  - F90: h5dwrite_f and h5dread_f

Accessing a Dataset Programming model

- Create and set dataset transfer property
  - C: H5Pset_dxpl_mpio
    - H5FD_MPIO_COLLECTIVE
    - H5FD_MPIO_INDEPENDENT (default)
  - F90: h5pset_dxpl_mpio_f
    - H5FD_MPIO_COLLECTIVE_F
    - H5FD_MPIO_INDEPENDENT_F (default)
- Access dataset with the defined transfer property

C Example: Collective write

```
/*
 * 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);
```

F90 Example: Collective write

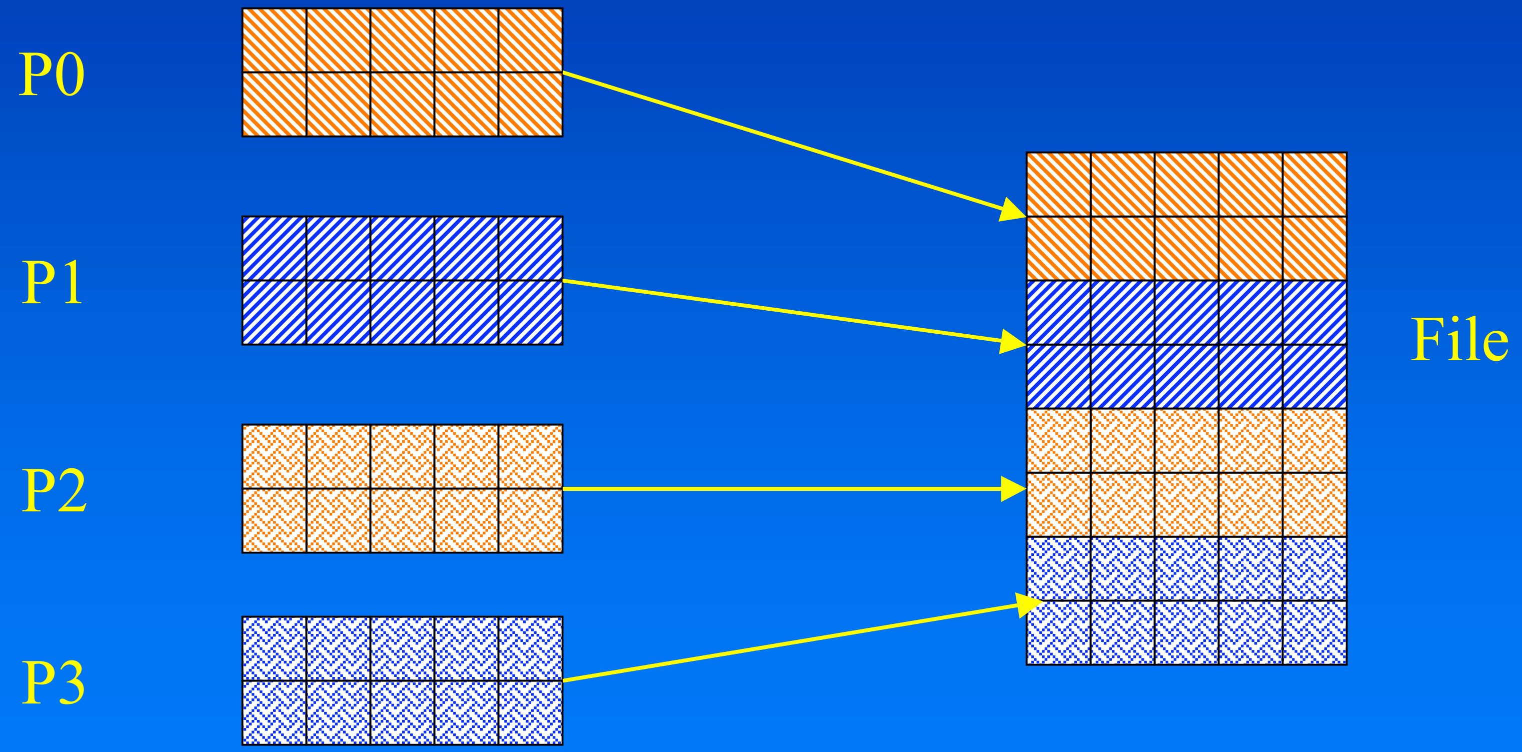
```
! Create property list for collective dataset write
!
CALL h5pcreate_f(H5P_DATASET_XFER_F, plist_id, error)
CALL h5pset_dxpl_mpio_f(plist_id, &
                        H5FD_MPIO_COLLECTIVE_F, error)

!
! 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 Programming model

- Distributed memory model: data is split among processes
- PHDF5 uses hyperslab model
- Each process defines memory and file hyperslabs
- Each process executes partial write/read call
  - Collective calls
  - Independent calls

Hyperslab Example 1

Writing dataset by rows

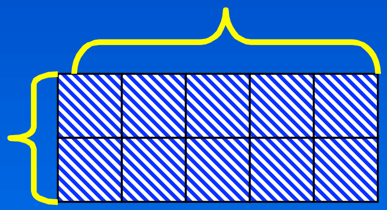


Writing by rows
Output of h5dump utility

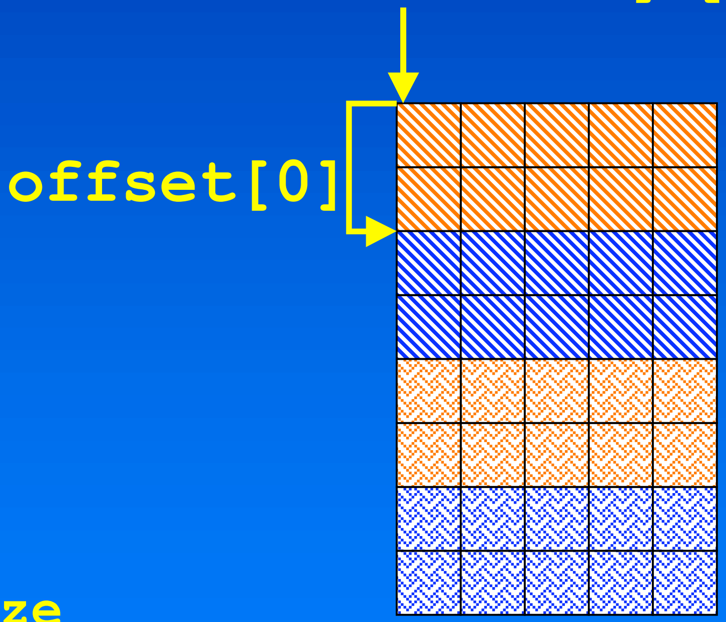
```
HDF5 "SDS_row.h5" {
GROUP "/" {
   DATASET "IntArray" {
      DATATYPE  H5T_STD_I32BE
      DATASPACE  SIMPLE { ( 8, 5 ) / ( 8, 5 ) }
      DATA {
         10, 10, 10, 10, 10,
         10, 10, 10, 10, 10,
         11, 11, 11, 11, 11,
         11, 11, 11, 11, 11,
         12, 12, 12, 12, 12,
    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

Example 1
Writing dataset by rows

P1 (memory space)



File

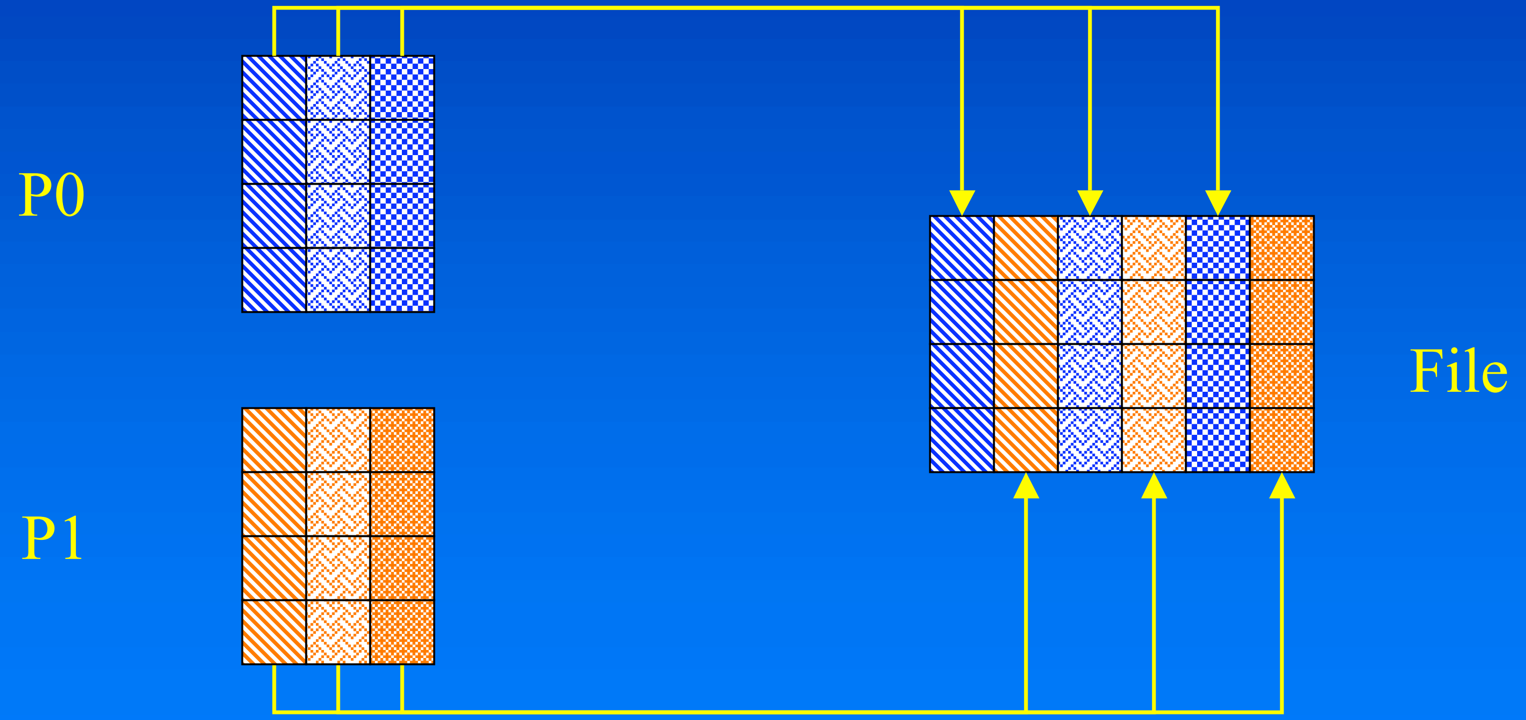


```
count[0] = dimsf[0]/mpi_size
count[1] = dimsf[1];
offset[0] = mpi_rank * count[0];  /* = 2 */
offset[1] = 0;
```

C Example 1

```
/*
 * 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);
```

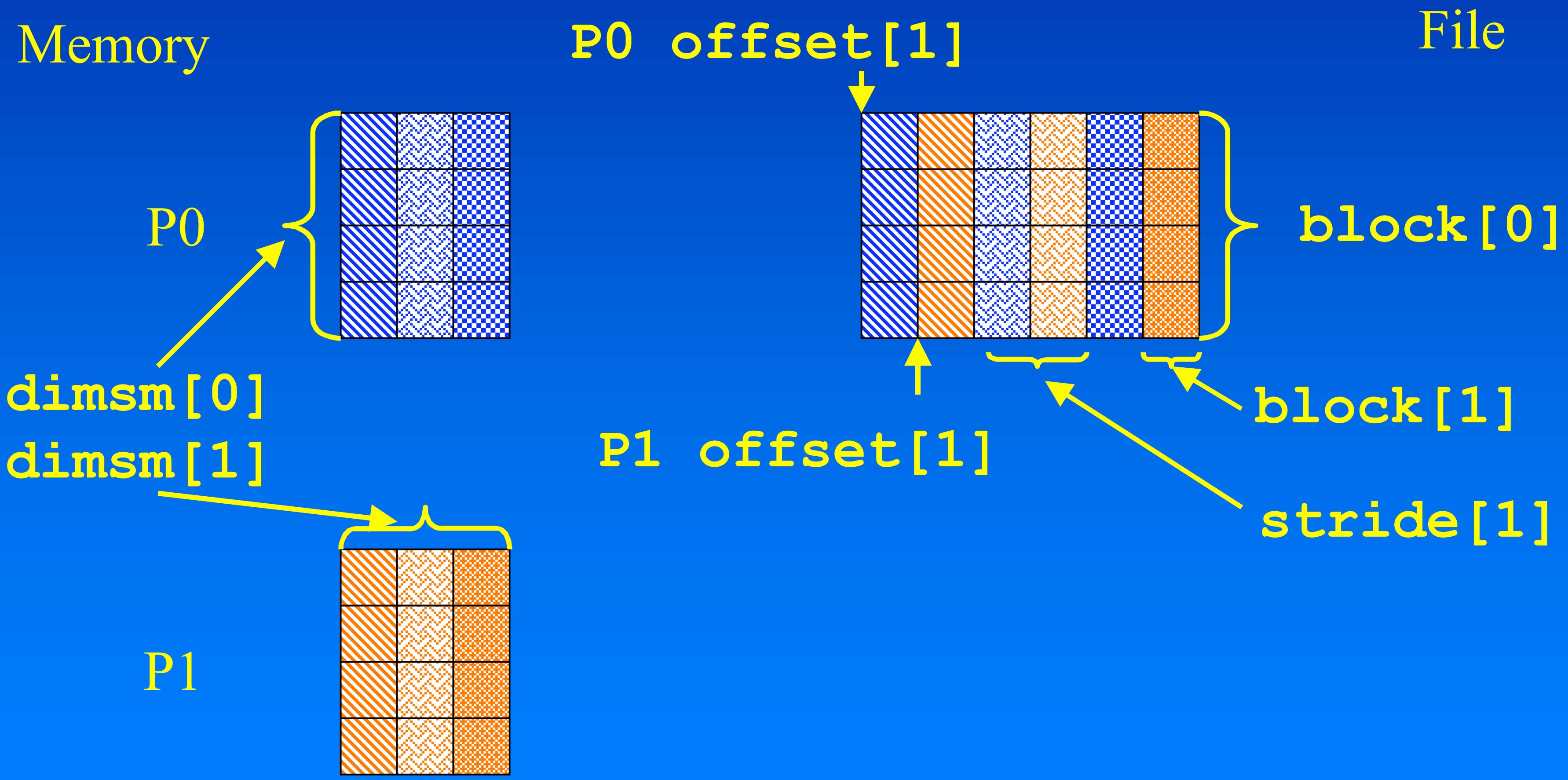
Hyperslab Example 2
Writing dataset by columns



Writing by columns
Output of h5dump utility

```
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
      }
   }
 }
}
```

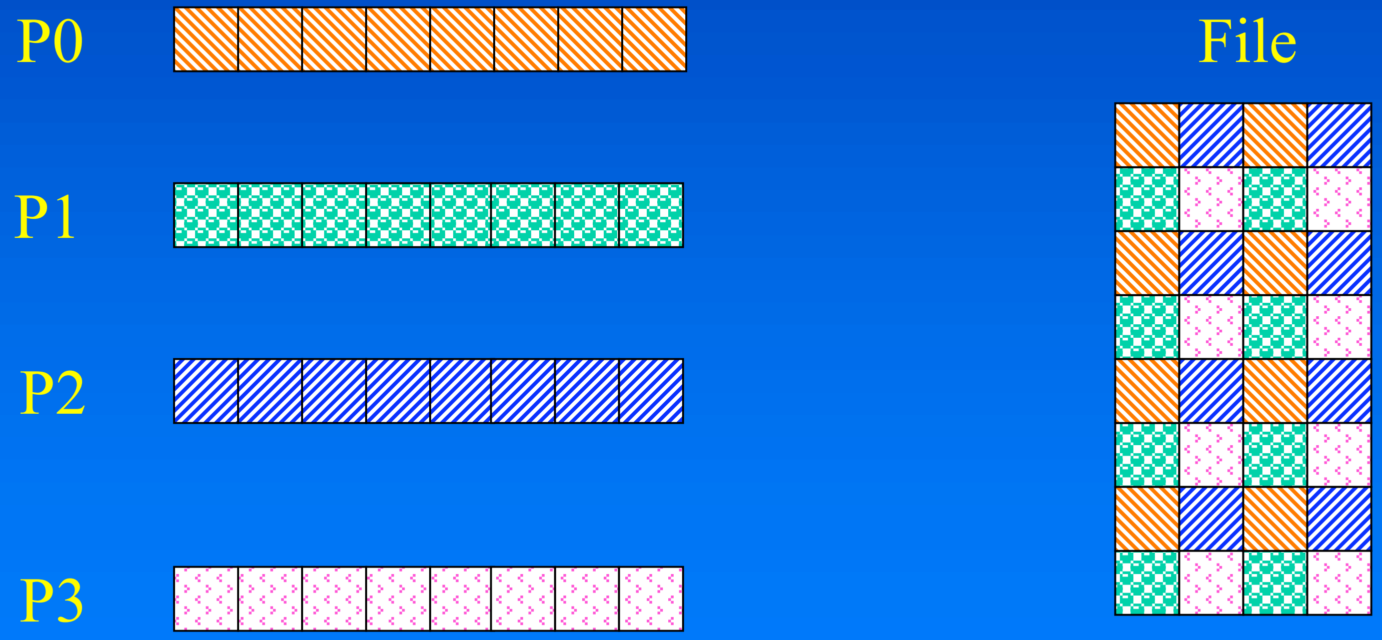
Example 2
Writing Dataset by Column



C Example 2

```
/*
 * Each process defines 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 hyperslab.
 */
filespace = H5Dget_space(dset_id);
H5Sselect_hyperslab(filespace,
   H5S_SELECT_SET, offset, stride,
   count, block);
```

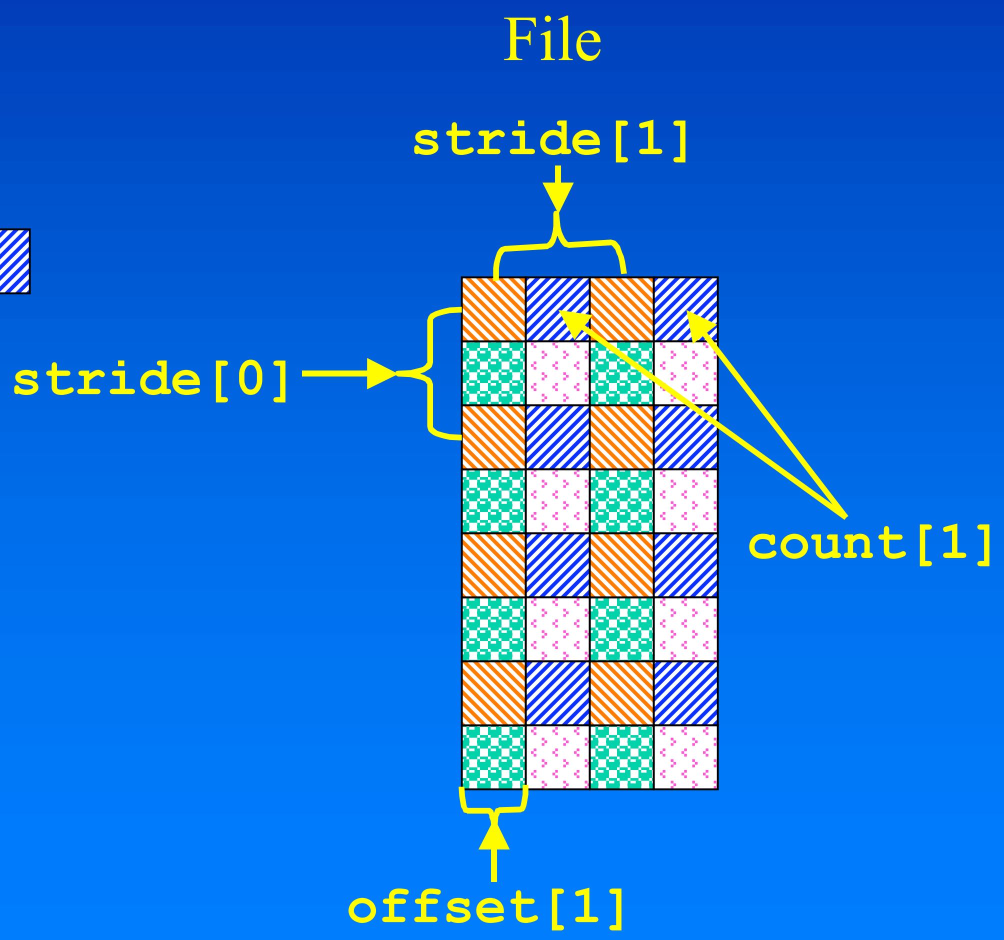
Hyperslab Example 3
Writing dataset by pattern



Writing by Pattern
Output of h5dump utility

```
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 3
Writing dataset by pattern

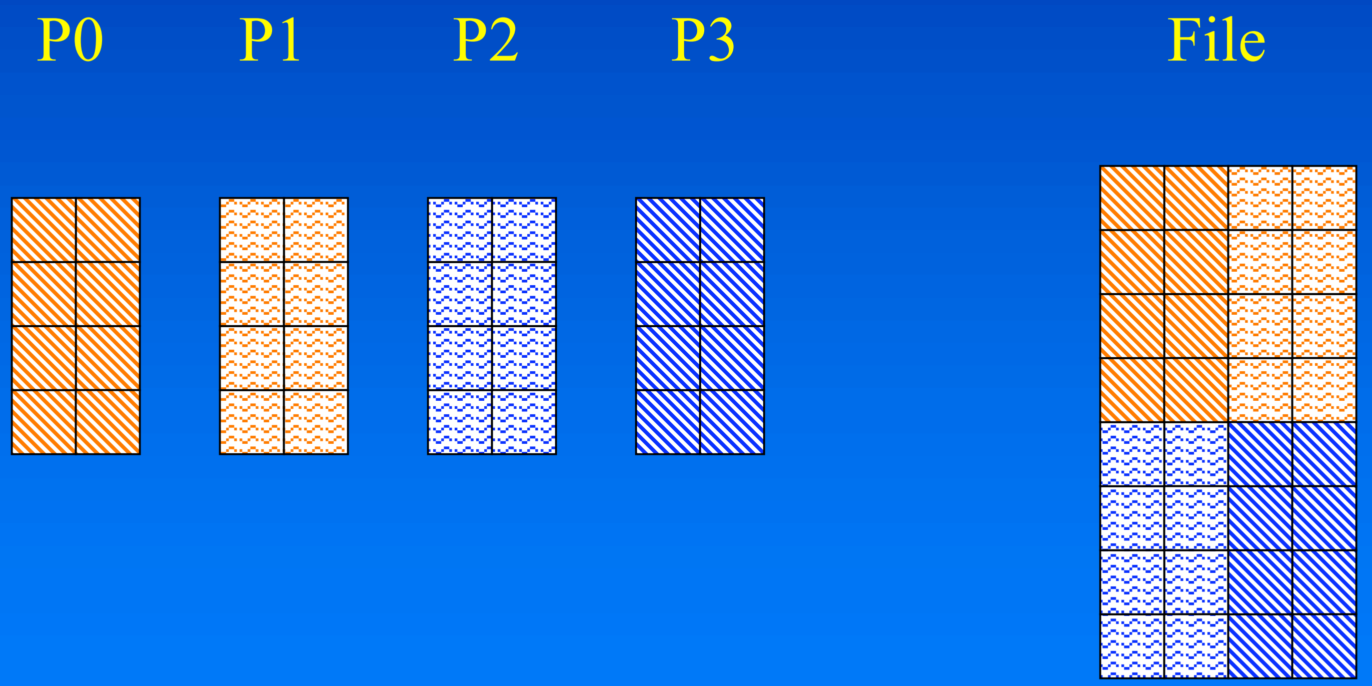


```
offset[0] = 0;
offset[1] = 1;
count[0] = 4;
count[1] = 2;
stride[0] = 2;
stride[1] = 2;
```

C 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;
}
```

Hyperslab Example 4
Writing dataset by chunks

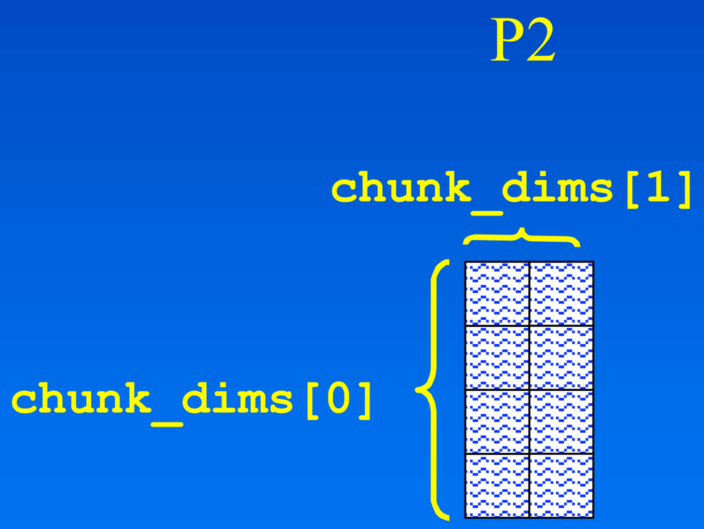


Writing by Chunks
Output of h5dump utility

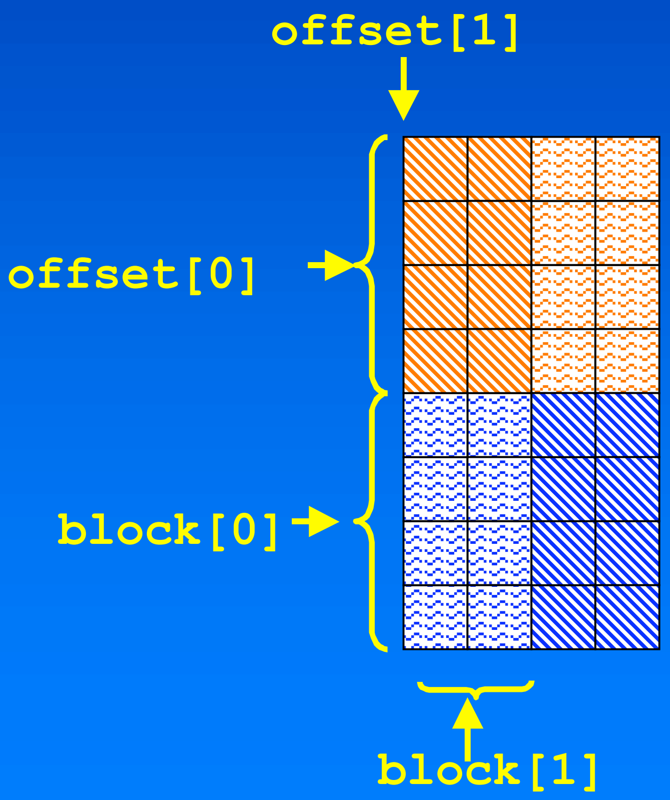
```
HDF5 "SDS_chnk.h5" {
GROUP "/" {
   DATASET "IntArray" {
      DATATYPE  H5T_STD_I32BE
      DATASPACE  SIMPLE { ( 8, 4 ) / ( 8, 4 ) }
      DATA {
         1, 1, 2, 2,
         1, 1, 2, 2,
         1, 1, 2, 2,
         1, 1, 2, 2,
         3, 3, 4, 4,
         3, 3, 4, 4,
         3, 3, 4, 4,
 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

Example 4
Writing dataset by chunks

Memory



File



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

C 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];
}
```

Performance Tuning in HDF5

Two Sets of Tuning Knobs

- File level knobs
  - Apply to the entire file
- Data transfer level knobs
  - Apply to individual dataset read or write

File Level Knobs

- H5Pset_meta_block_size
- H5Pset_alignment
- H5Pset_fapl_split
- H5Pset_cache
- H5Pset_fapl_mpio

H5Pset_meta_block_size

- Sets the minimum metadata block size allocated for metadata aggregation.
- Aggregated block is usually written in a single write action
- Default is 2KB
- *Pro:*
  - Larger block size reduces I/O requests
- *Con:*
  - Could create "holes" in the file and make file bigger

H5Pset_meta_block_size

- When to use:
- File is open for a long time and
  - A lot of objects created
  - A lot of operations on the objects performed
  - As a result metadata is interleaved with raw data
  - A lot of new metadata (attributes)

H5Pset_alignment

- Sets two parameters
  - Threshold
    - Minimum size of object for alignment to take effect
    - Default 1 byte
  - Alignment
    - Allocate object at the next multiple of alignment
    - Default 1 byte
- Example: (threshold, alignment) = (1024, 4K)
  - All objects of 1024 or more bytes starts at the boundary of 4KB

H5Pset_alignment Benefits

- In general, the default (no alignment) is good for single process serial access since the OS already manages buffering.
- For some parallel file systems such as GPFS, an alignment of the disk block size improves I/O speeds.
- *Con: File may be bigger*

H5Pset_fapl_split

- HDF5 splits to two files
  - Metadata file for metadata
  - Rawdata file for raw data (array data)
  - Two files represent one logical HDF5 file
- *Pro:* Significant I/O improvement if
  - metadata file is stored in Unix file systems (good for many small I/O)
  - raw data file is stored in Parallel file systems (good for large I/O).

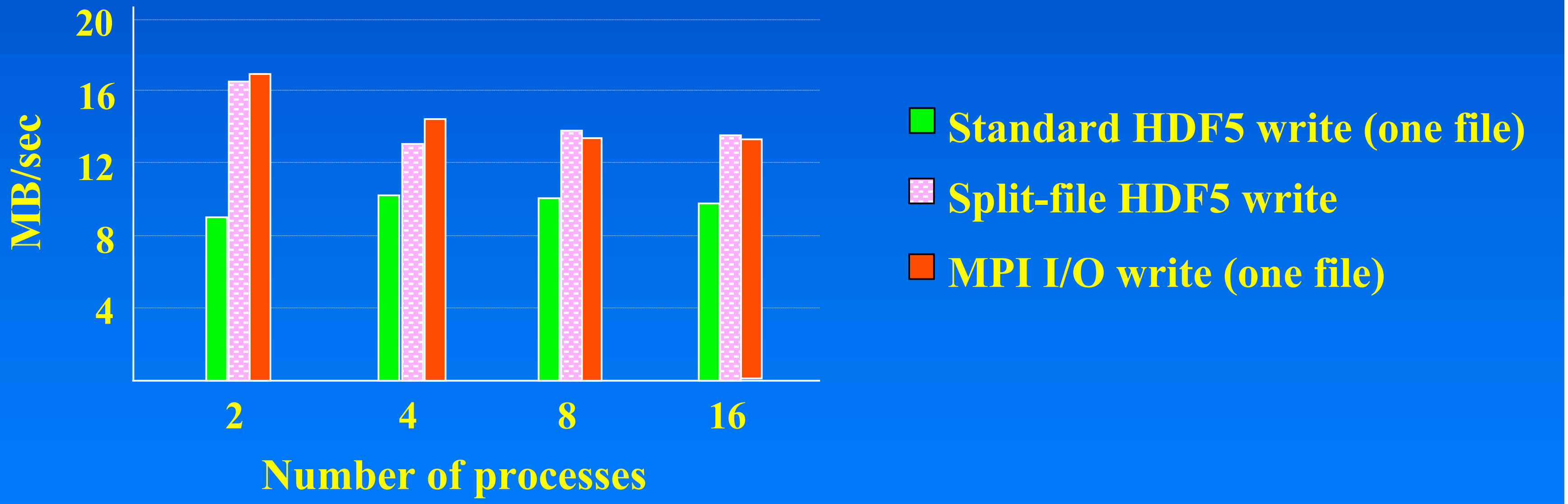
H5Pset_fapl_split

- *Con:*
  - Both files should be "kept together" for integrity of the HDF5 file
  - Can be a potential problem when files are moved to another platform or file system

Write speeds of
Standard vs. Split-file HDF5 vs. MPI-IO

Results for ASCI Red machine at Sandia National Laboratory

- **Each process writes 10MB of array data**



H5Pset_cache

- Sets:
  - The number of elements (objects) in the meta data cache
  - The number of elements, the total number of bytes, and the preemption policy value (default is 0.75) in the raw data chunk cache

H5Pset_cache (cont.)

- Preemption policy:
  - _ Chunks are stored in the list with the most recently accessed chunk at the end
  - _ Least recently accessed chunks are at the beginning of the list
  - _ X*100% of the list is searched for the fully read/written chunk; X is called preemption value, where X is between 0 and 1
  - _ If chunk is found then it is deleted from cache, if not then first chunk in the list is deleted

H5Pset_cache (cont.)

- The right values of N
  - May improve I/O performance by controlling preemption policy
  - 0 value forces to delete the "oldest" chunk from cache
  - 1 value forces to search all list for the chunk that will be unlikely accessed
  - Depends on application access pattern

Chunk Cache Effect by H5Pset_cache

- Write one integer dataset 256x256x1024 (256MB)
- Using chunks of 256x16x1024 (16MB)
- Two tests of
  - _Default chunk cache size (1MB)
  - _Set chunk cache size 16MB

Chunk Cache
Time Definitions

- Total
  - Time to open file, write dataset, close dataset and close file
- Dataset write
  - Time to write the whole dataset
- Chunk write
  - Time to write a chunk
- User time/System time
  - Total Unix user/system time of test

Chunk Cache Size Results

Cache buffer size (MB)	Chunk write time (sec)	Dataset write time (sec)	Total time (sec)	User time (sec)	System time (sec)
1	132.58	2450.25	2453.09	14.00	2200.10
16	0.376	7.83	8.27	6.21	3.45

Chunk Cache Size Summary

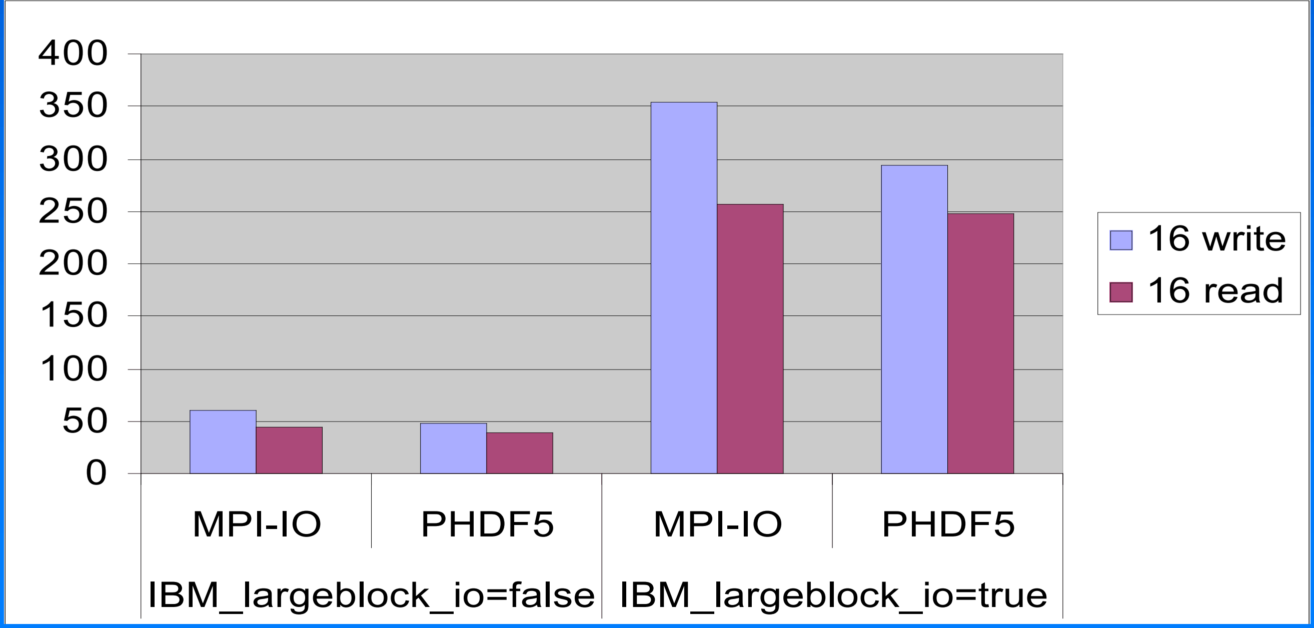
- Big chunk cache size improves performance
- Poor performance mostly due to increased system time
  - Many more I/O requests
  - Smaller I/O requests

I/O Hints via H5Pset_fapl_mpio

- MPI-IO hints can be passed to the MPI-IO layer via the Info parameter of H5Pset_fapl_mpio
- Examples
  - Telling Romio to use 2-phases I/O speeds up collective I/O in the ASCI Red machine
  - Setting IBM_largeblock_io=true speeds up GPFS write speeds

Effects of I/O Hints IBM_largeblock_io

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



Effects of I/O Hints
IBM_largeblock_io

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

		IBM_largeblock_io=false		IBM_largeblock_io=true	
Tasks		MPI-IO	PHDF5	MPI-IO	PHDF5
16	write	60	48	354	294
16	read	44	39	256	248

Data Transfer Level Knobs

- H5Pset_buffer
- H5Pset_sieve_buf_size

H5Pset_buffer

- Sets size of the internal buffers used during data transfer
- Default is 1 MB
- Pro:
  - Bigger size improves performance
- Con:
  - Library uses more memory

H5Pset_buffer

- When should be used:
  - Datatype conversion
  - Data gathering-scattering (e.g. checker board dataspace selection)

H5Pset_sieve_buf_size

- Sets the size of the data sieve buffer
- Default is 64KB
- Sieve buffer is a buffer in memory that holds part of the dataset raw data
- During I/O operations data is replaced in the buffer first, then one big I/O request occurs

H5Pset_sieve_buf_size

- Pro:
  - Bigger size reduces I/O requests issued for raw data access
- Con:
  - Library uses more memory
- When to use:
  - Data scattering-gathering (e.g. checker board)
  - Interleaved hyperslabs

Parallel I/O Benchmark Tool

- h5perf
  - Benchmark test I/O performance
- Four kinds of API
  - Parallel HDF5
  - MPI-IO
  - Native parallel (e.g., gpfs, pvfs)
  - POSIX (open, close, lseek, read, write)

Useful Parallel HDF Links

- Parallel HDF information site
  - http://hdf.ncsa.uiuc.edu/Parallel_HDF/
- Parallel HDF mailing list
  - hdfparallel@ncsa.uiuc.edu
- Parallel HDF5 tutorial available at
  - http://hdf.ncsa.uiuc.edu/HDF5/doc/Tutor