



The HDF Group



Parallel HDF5

Quincey Koziol
Director of Core Software & HPC
The HDF Group



Parallel HDF5 Success Story

- Recent success story
 - Trillion particle simulation on hopper @ NERSC
 - 120,000 cores
 - 30TB file
 - 23GB/sec average speed with 35GB/sec peaks (out of 40GB/sec max for system)
 - <http://1.usa.gov/1sIU8uf>
- Parallel HDF5 rocks! (when used properly 😊)



Outline

- Overview of Parallel HDF5 design
- Parallel Environment Requirements
- PHDF5 Programming Model
- Examples
- Performance Analysis
- Parallel Tools
- Details of upcoming features of HDF5 (if time permits)



MPI-I/O VS. HDF5



MPI-IO

- MPI-IO is an Input/Output API
- It treats the data file as a “linear byte stream” and each MPI application needs to provide its own file and data representations to interpret those bytes



MPI-IO

- All data stored are *machine dependent* except the “external32” representation
- External32 is defined in Big Endianness
 - Little-endian machines have to do the data conversion in both read or write operations
 - 64-bit sized data types may lose information



MPI-IO vs. HDF5

- HDF5 is data management software
- It stores data and metadata according to the HDF5 data format definition
- HDF5 file is self-describing
 - Each machine can store the data in its own native representation for efficient I/O without loss of data precision
 - Any necessary data representation conversion is done by the HDF5 library automatically



OVERVIEW OF PARALLEL HDF5 DESIGN



PHDF5 Requirements

- PHDF5 should allow multiple processes to perform I/O to an HDF5 file at the same time
 - Single file image for all processes
 - Compare with one file per process design:
 - Expensive post processing
 - Not usable by different number of processes
 - Too many files produced for file system
- PHDF5 should use a standard parallel I/O interface
- Must be portable to different platforms



PHDF5 requirements

- Support Message Passing Interface (MPI) programming
- PHDF5 files compatible with serial HDF5 files
 - Shareable between different serial or parallel platforms

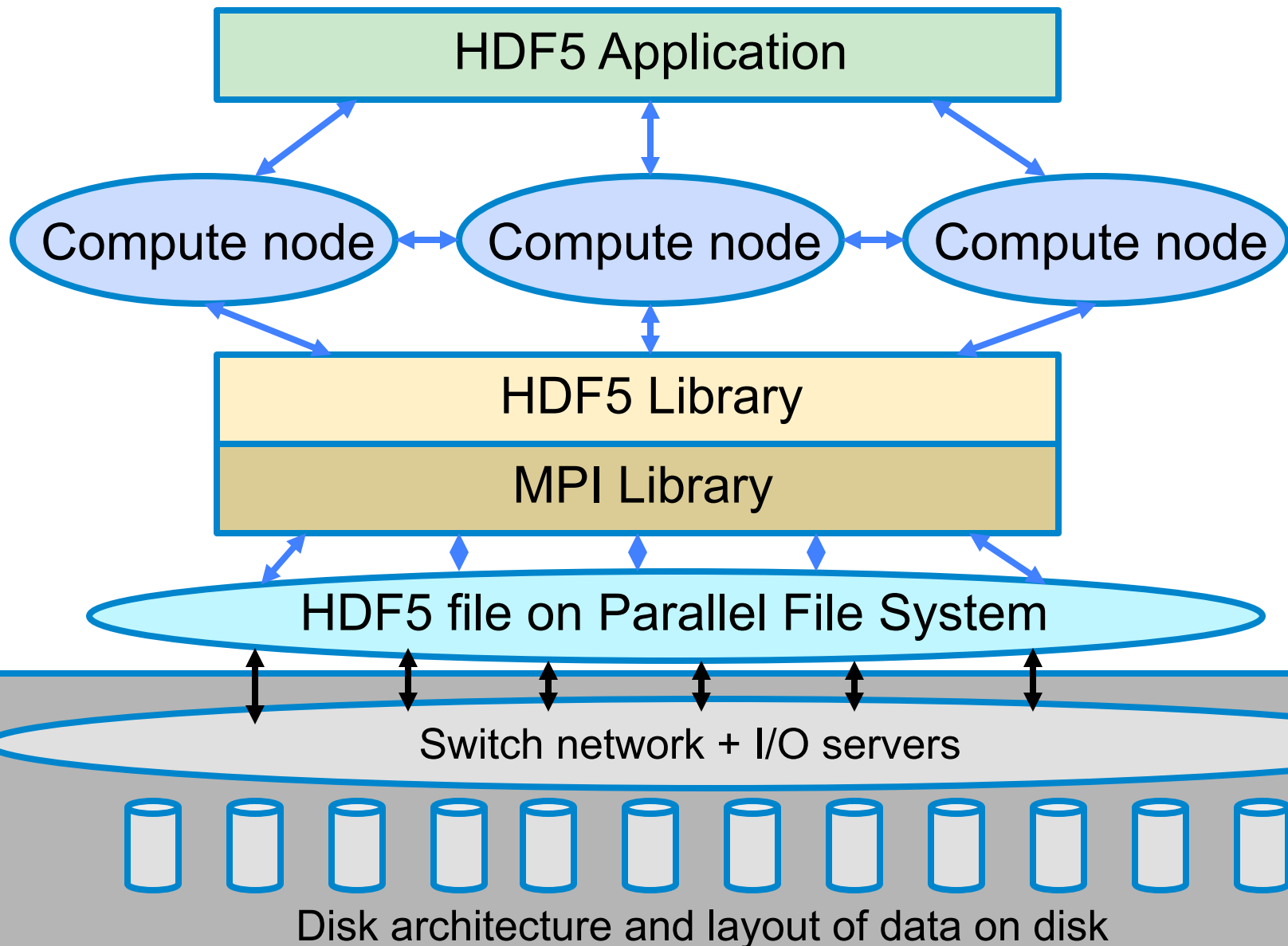


Parallel environment requirements

- MPI with MPI-IO
 - MPICH, OpenMPI w/ROMIO
 - Vendor's MPI-IO
- Parallel file system
 - IBM GPFS
 - Lustre
 - PVFS



PHDF5 implementation layers





PHDF5 CONSISTENCY SEMANTICS



Consistency Semantics

- Consistency Semantics: Rules that define the outcome of multiple, possibly concurrent, accesses to an object or data structure by one or more processes in a computer system.



PHDF5 Consistency Semantics

- PHDF5 library defines a set of consistency semantics to let users know what to expect when processes access data managed by the library.
 - When the changes a process makes are actually visible to itself (if it tries to read back that data) or to other processes that access the same file with independent or collective I/O operations



HDF5 MPI-I/O consistency semantics

- Same as MPI-I/O semantics

Process 0	Process 1
MPI_File_write_at()	
MPI_Barrier()	MPI_Barrier()
	MPI_File_read_at()

- Default MPI-I/O semantics doesn't guarantee atomicity or sequence of above calls!
- Problems may occur (although we haven't seen any) when writing/reading HDF5 metadata or raw data



HDF5 MPI-I/O consistency semantics

- MPI I/O provides atomicity and sync-barrier-sync features to address the issue
- PHDF5 follows MPI I/O
 - H5Fset_mpio_atomicity function to turn on MPI atomicity
 - H5Fsync function to transfer written data to storage device (in implementation now)
- Alternatively: We are currently working on reimplementing metadata caching for PHDF5 (using a metadata server)



HDF5 MPI-I/O consistency semantics

- For more information see “Enabling a strict consistency semantics model in parallel HDF5” linked from `H5Fset_mpi_atomicity` RM page¹

¹ <http://www.hdfgroup.org/HDF5/doc/RM/Advanced/PHDF5FileConsistencySemantics/PHDF5FileConsistencySemantics.pdf>



HDF5 PARALLEL PROGRAMMING MODEL



How to compile PHDF5 applications

- h5pcc – HDF5 C compiler command
 - Similar to mpicc
- h5pfc – HDF5 F90 compiler command
 - Similar to mpif90
- To compile:
 - % h5pcc h5prog.c
 - % h5pfc h5prog.f90



Programming restrictions

- PHDF5 opens a parallel file with an MPI 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 F90, 2003 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 HDF5 file in parallel:
 - Set up MPI-IO access template (file access property list)
 - Open File
 - Access Data
 - Close File



Moving your sequential application to the HDF5 parallel world

MY FIRST PARALLEL HDF5 PROGRAM



Example of PHDF5 C program

Parallel HDF5 program has extra calls

```
MPI_Init(&argc, &argv);
```

1. `fapl_id = H5Pcreate(H5P_FILE_ACCESS);`
2. `H5Pset_fapl_mpio(fapl_id, comm, info);`
3. `file_id = H5Fcreate(FNAME, ..., fapl_id);`
4. `space_id = H5Screate_simple(...);`
5. `dset_id = H5Dcreate(file_id, DNAME, H5T_NATIVE_INT, space_id, ...);`
6. `xf_id = H5Pcreate(H5P_DATASET_XFER);`
7. `H5Pset_dxpl_mpio(xf_id, H5FD_MPIO_COLLECTIVE);`
8. `status = H5Dwrite(dset_id, H5T_NATIVE_INT, ..., xf_id...);`

```
MPI_Finalize();
```



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>



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



Four processes writing by rows

```
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
```




Two processes writing by columns

```
HDF5 "SDS_co1.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      }  
    }  
  }  
}
```



Four processes writing by pattern

```
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
```



Four processes writing by blocks

```
HDF5 "SDS_b1k.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
```



Complex data patterns

HDF5 doesn't have restrictions on data patterns and data balance

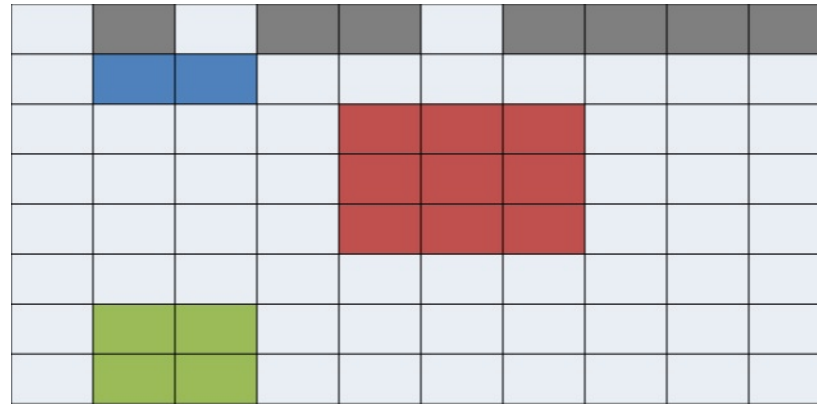
1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

1	2	3	4				
9	10	11	12				
17	18	19	20				
25	26	27	28				
				37	38	39	40
				45	46	47	48
				53	54	55	56
				61	62	63	64

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64



Examples of irregular selection



- Internally, the HDF5 library creates an MPI datatype for each lower dimension in the selection and then combines those types into one giant structured MPI datatype



PERFORMANCE ANALYSIS



Performance analysis

- Some common causes of poor performance
- Possible solutions



My PHDF5 application I/O is slow

- Raw I/O data sizes
- Independent vs. Collective I/O

“Tuning HDF5 for Lustre File Systems” by Howison, Koziol, Knaak, Mainzer, and Shalf

- ❖ Chunking and hyperslab selection
- ❖ HDF5 metadata cache
- ❖ Specific I/O system hints



INDEPENDENT VS. COLLECTIVE RAW DATA I/O



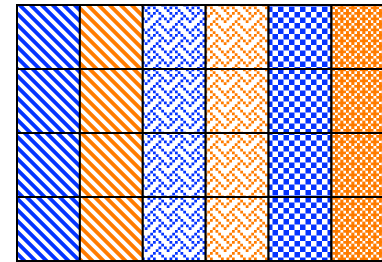
Collective vs. independent calls

- MPI definition of collective calls:
 - All processes of the communicator must participate in calls in the right order. E.g.,
 - Process1 Process2
 - call A(); call B(); call A(); call B(); ****right****
 - call A(); call B(); call B(); call A(); ****wrong****
- Independent means not collective 😊
- Collective is not necessarily synchronous, nor must require communication



Independent vs. collective access

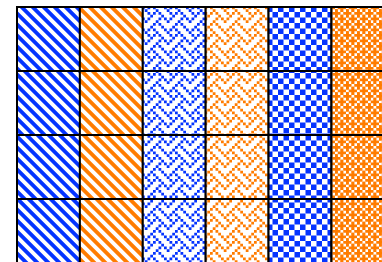
- User reported independent data transfer mode was much slower than the collective data transfer mode
- Data array was tall and thin: 230,000 rows by 6 columns



⋮

230,000 rows

⋮





Debug Slow Parallel I/O Speed(1)

- Writing to one dataset
 - Using 4 processes == 4 columns
 - HDF5 datatype is 8-byte doubles
 - 4 processes, 1000 rows == $4 \times 1000 \times 8 = 32,000$ bytes
- % mpirun -np 4 ./a.out 1000
 - Execution time: 1.783798 s.
- % mpirun -np 4 ./a.out 2000
 - Execution time: 3.838858 s.
- Difference of 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-debug --enable-parallel ...`
 - This allows the tracing of MPIIO I/O calls in the HDF5 library.
- E.g., to trace
 - `MPI_File_read_xx` and `MPI_File_write_xx` calls
 - `% setenv H5FD_mpio_Debug "rw"`



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
```

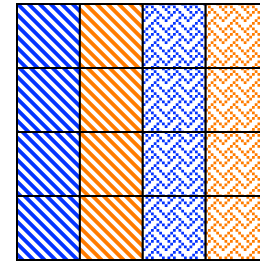
```
...
```

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



Independent calls are many and small

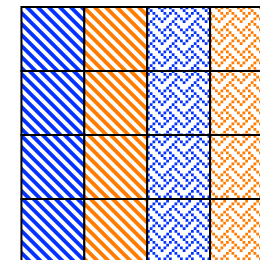
- Each process writes one element of one row, skips to next row, write one element, so on.
- Each process issues 230,000 writes of 8 bytes each.



⋮

230,000 rows

⋮





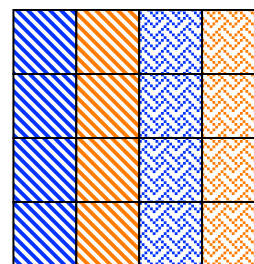
Debug slow parallel I/O speed (4)

```
% 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.
```




Use collective mode or chunked storage

- Collective I/O will combine many small independent calls into few but bigger calls
- Chunks of columns speeds up too



:

:

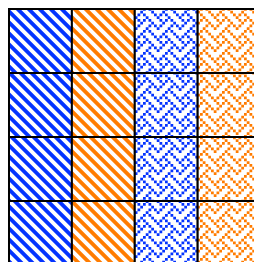
:

230,000 rows

:

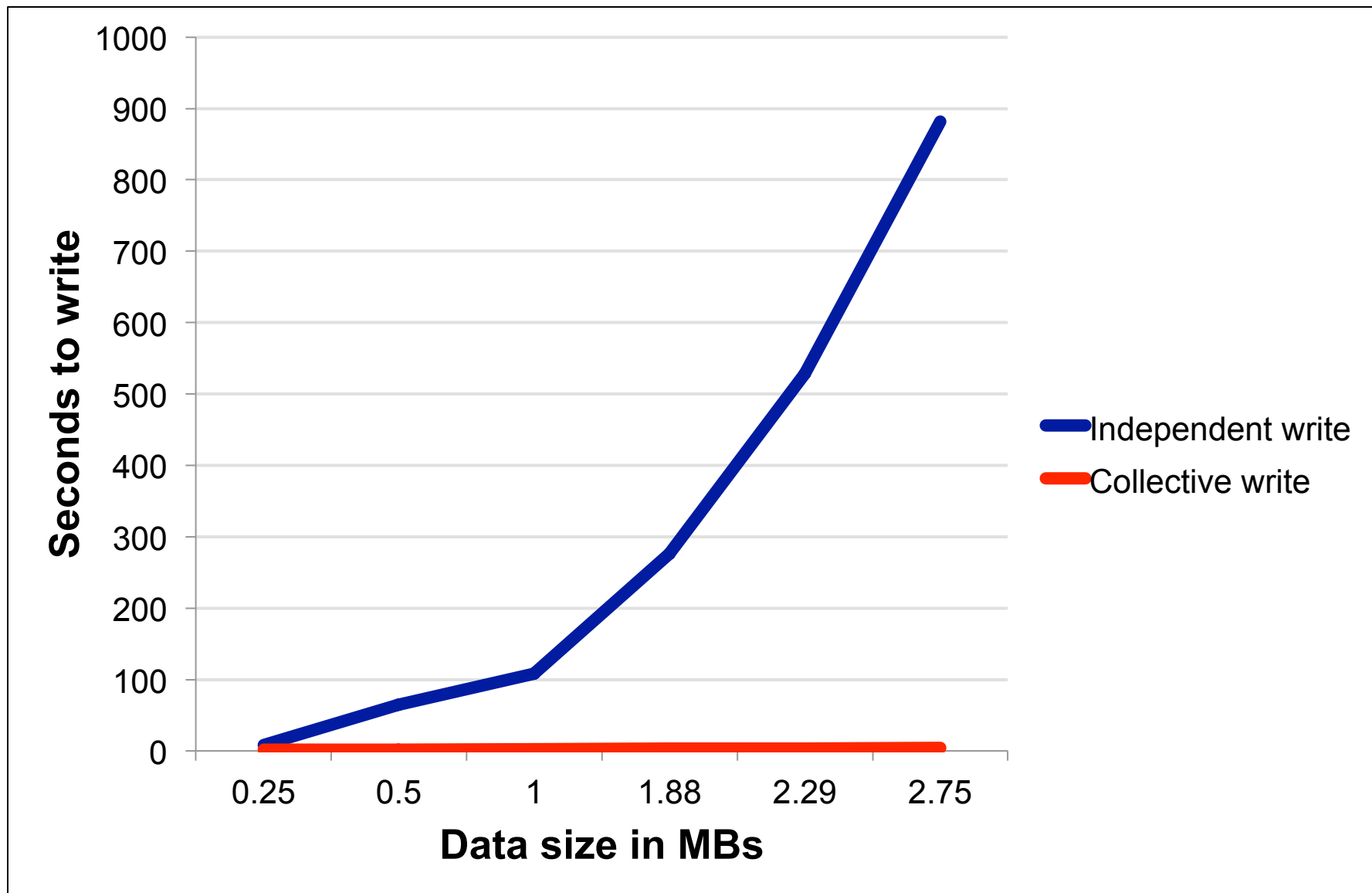
:

:





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 user asks for it:
 - Data conversion
 - Compressed Storage
 - Chunking Storage:
 - When the chunk is not selected by a certain number of processes

Enabling Collective Parallel I/O with HDF5

```
/* 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);
```



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

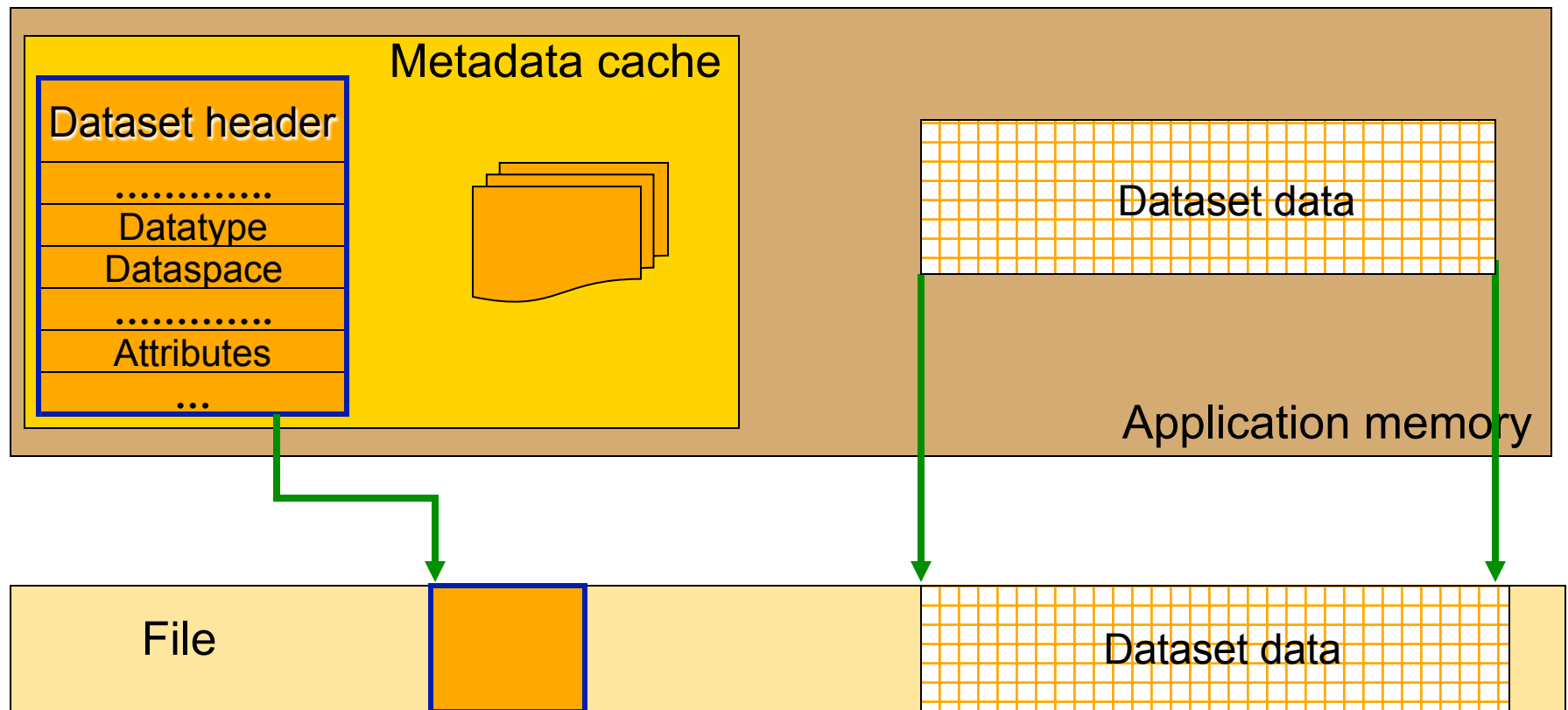


EFFECT OF HDF5 STORAGE



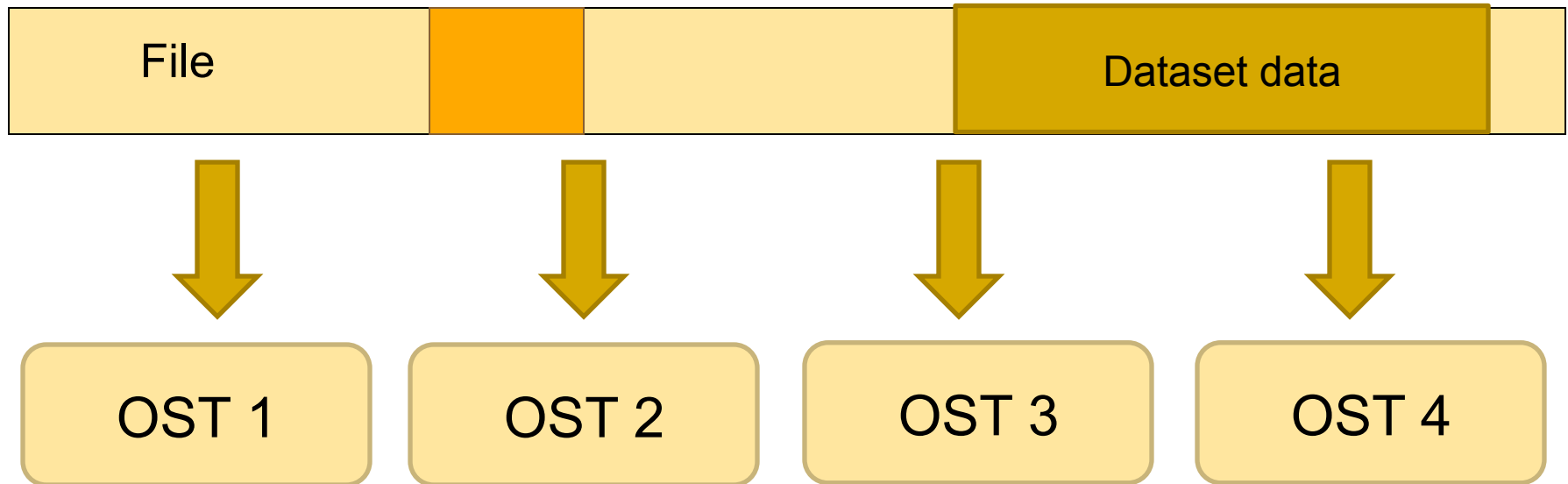
Contiguous storage

- Metadata header separate from dataset data
- Data stored in one contiguous block in HDF5 file





On a parallel file system



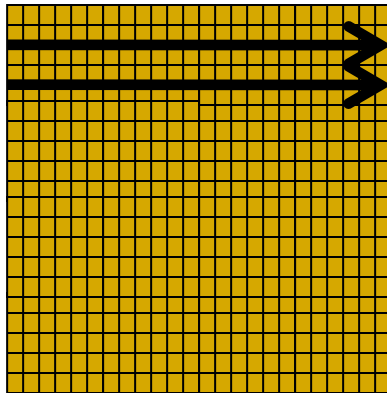
The file is striped over multiple OSTs depending on the stripe size and stripe count that the file was created with.



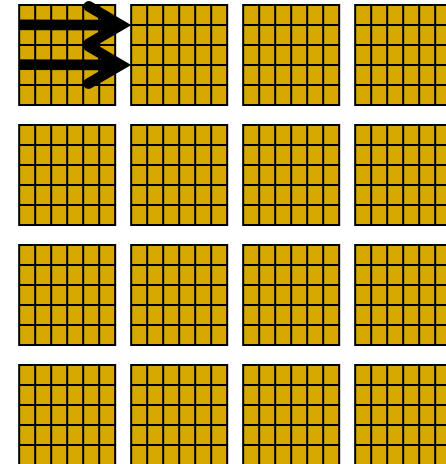
Chunked storage

- Data is stored in chunks of predefined size
 - Two-dimensional instance may be referred to as data tiling
- HDF5 library writes/reads the whole chunk

Contiguous



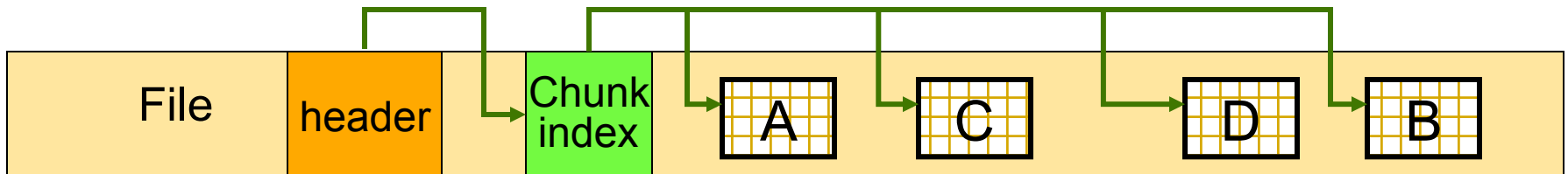
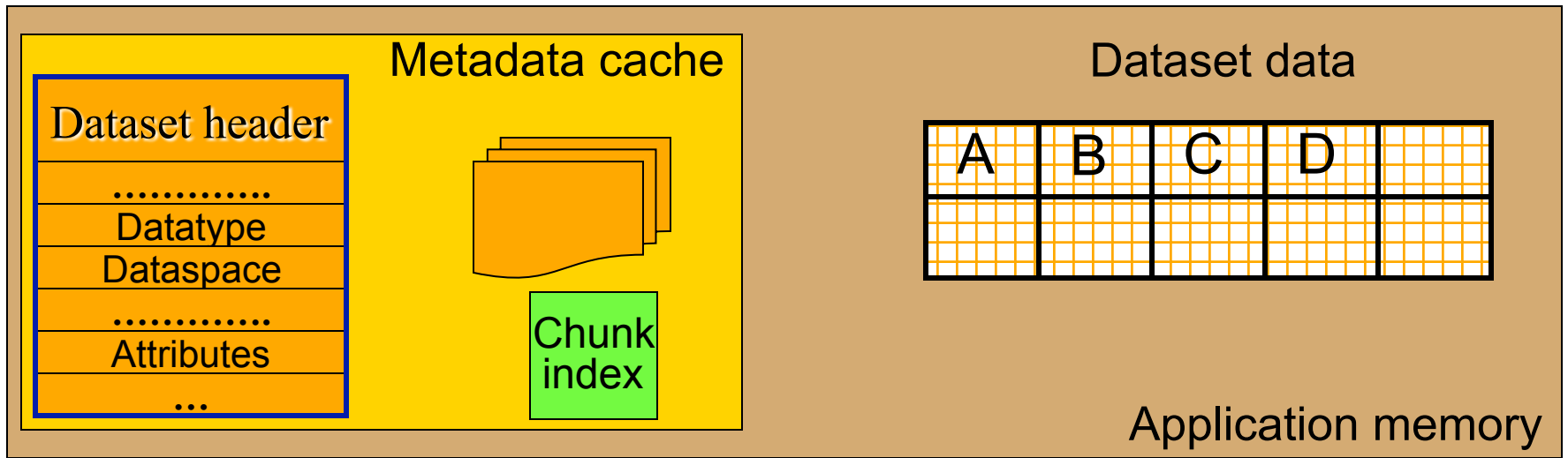
Chunked





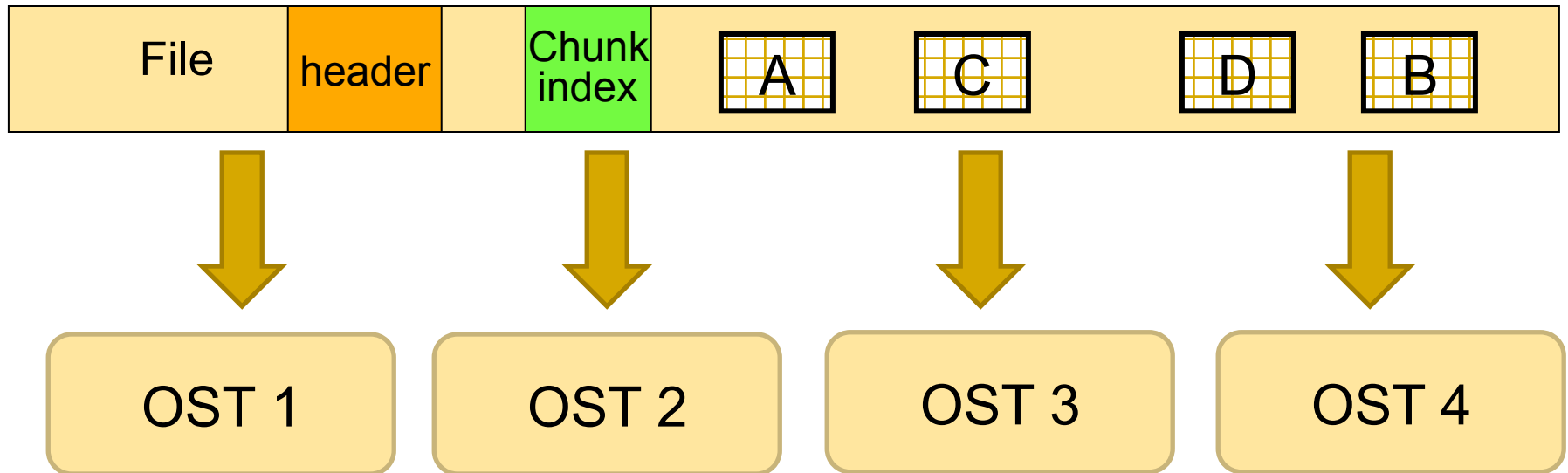
Chunked storage (cont.)

- Dataset data is divided into equally sized blocks (chunks).
- Each chunk is stored separately as a **CONTIGUOUS** block in HDF5 file.





On a parallel file system

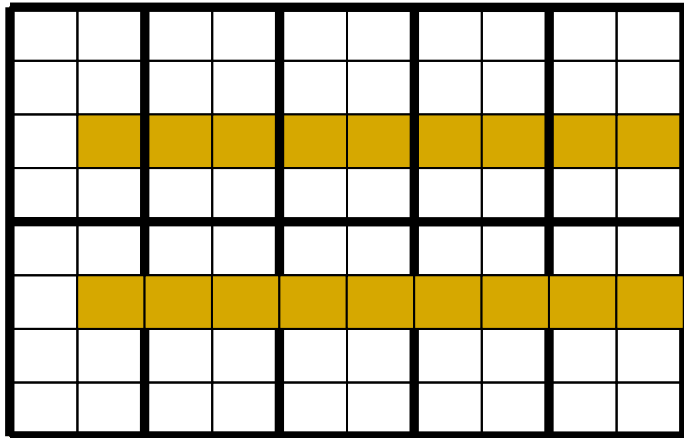


The file is striped over multiple OSTs depending on the stripe size and stripe count that the file was created with

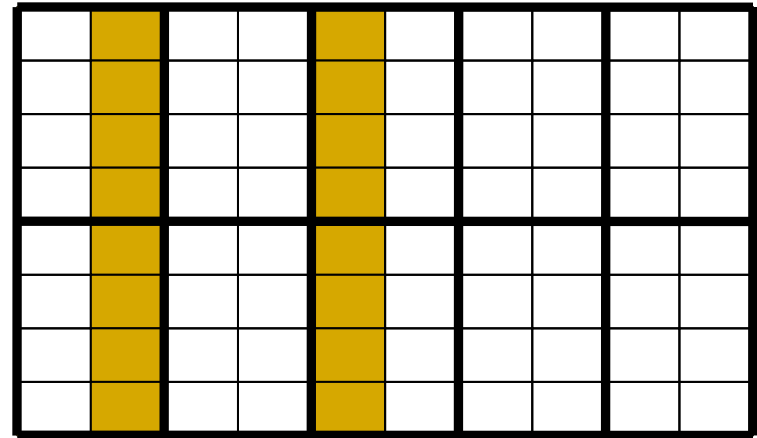


Which is better for performance?

- It depends!!
- Consider these selections:



- If contiguous: 2 seeks
- If chunked: 10 seeks

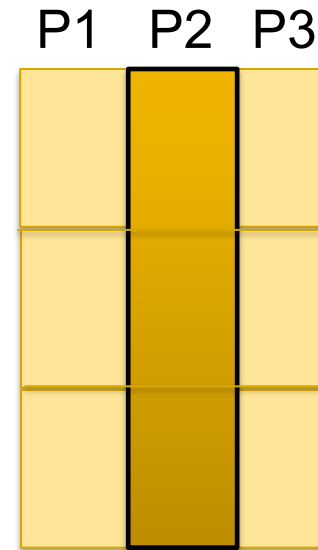
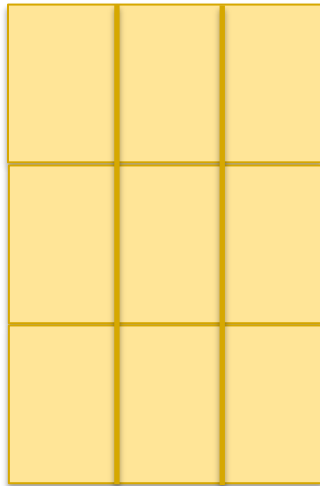


- If contiguous: 16 seeks
- If chunked: 4 seeks

Add to that striping over a Parallel File System, which makes this problem very hard to solve!



Chunking and hyperslab selection

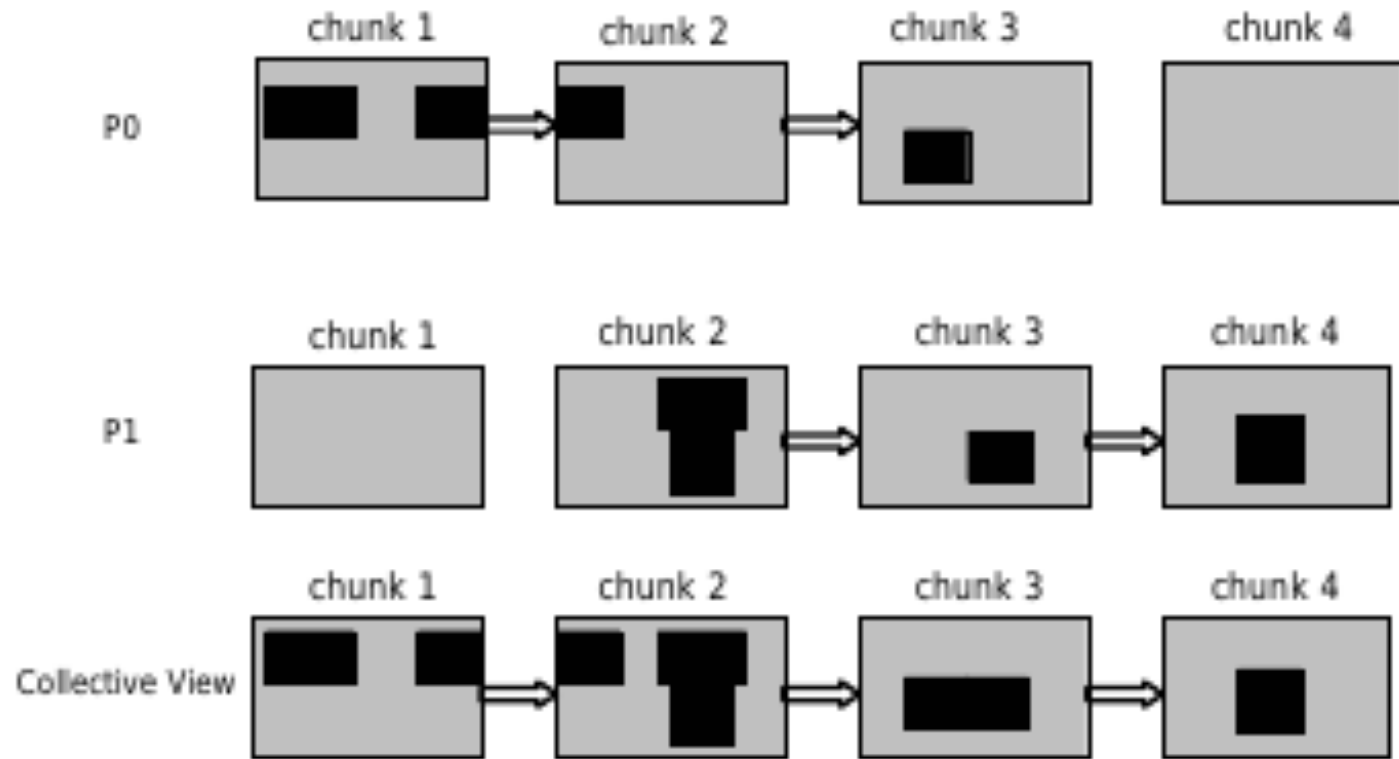


- When writing or reading, try to use hyperslab selections that coincide with chunk boundaries.
- If not possible, HDF5 provides some options



Parallel I/O on chunked datasets

- Multiple options for performing I/O when collective:
 - Operate on all chunks in one collective I/O operation: “Linked chunk I/O”
 - Operate on each chunk collectively: “Multi-chunk I/O”
 - Break collective I/O and perform I/O on each chunk independently (also in “Multi-chunk I/O” algorithm)

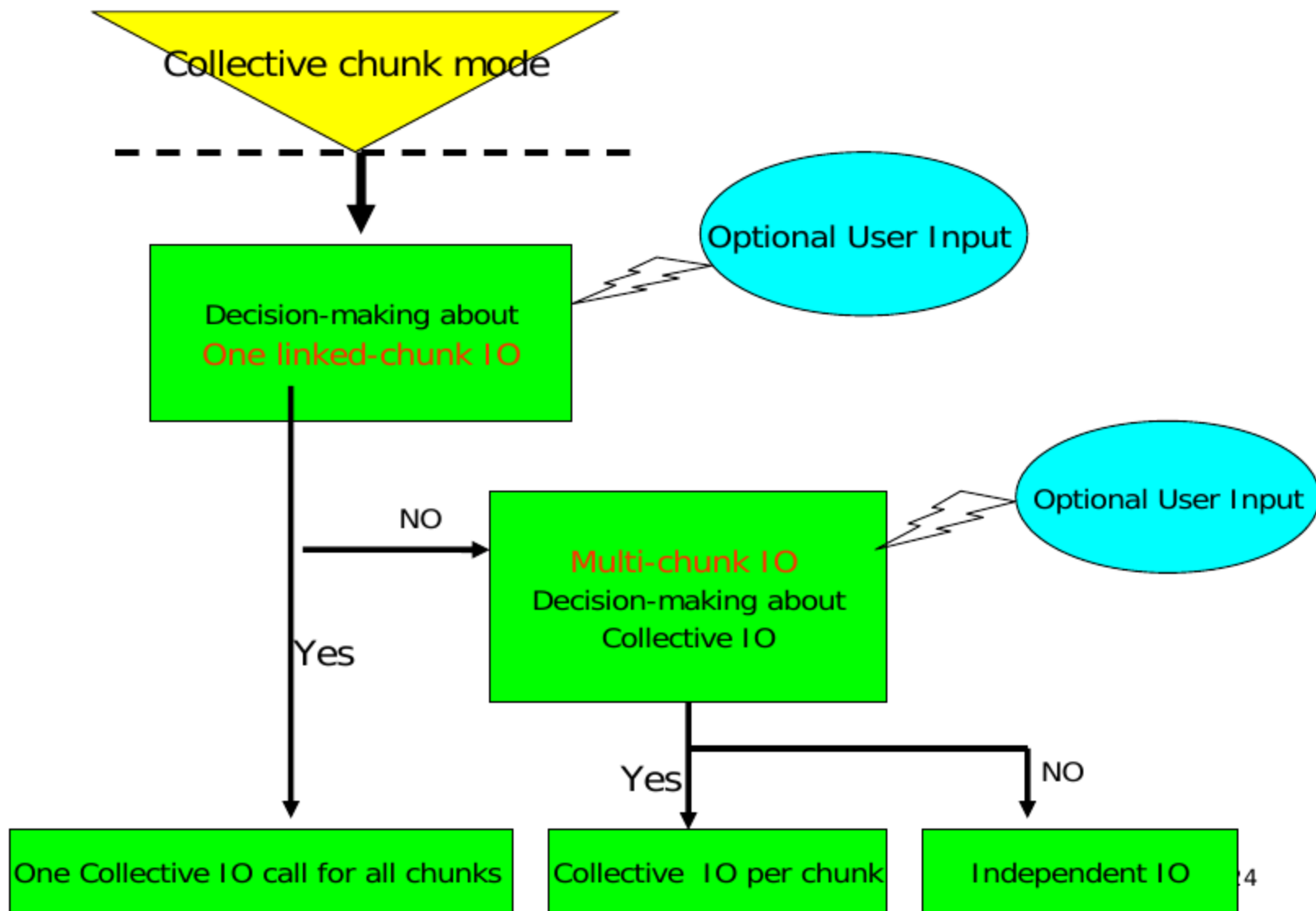


- One MPI Collective I/O Call



Multi-chunk I/O

- Collective I/O per chunk
- Determine for each chunk if enough processes have a selection inside to do collective I/O
- If not enough, use independent I/O





EFFECT OF HDF5 METADATA CACHE



PHDF5 and Metadata

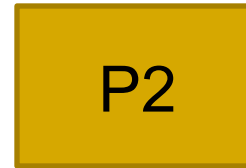
- Metadata operations:
 - Creating/removing a dataset, group, attribute, etc...
 - Extending a dataset's dimensions
 - Modifying group hierarchy
 - etc ...
- All operations that modify metadata are collective, i.e., all processes have to call that operation:
 - If you have 10,000 processes running your application, and one process needs to create a dataset, **ALL** processes must call H5Dcreate to create 1 dataset.



Space allocation

- Allocating space at the file's EOA is very simple in serial HDF5 applications:
 - the EOA value begins at offset 0 in the file
 - when space is required, the EOA value is incremented by the size of the block requested.
- Space allocation using the EOA value in parallel HDF5 applications can result in a race condition if processes do not synchronize with each other:
 - multiple processes believe that they are the sole owner of a range of bytes within the HDF5 file.
- Solution: Make it Collective

- Consider this case, where 2 processes want to create a dataset each.



H5Dcreate(D1)

H5Dcreate(D2)

Each call has to allocate space in file to store the dataset header.

Bytes 4 to 10 in the file are free

Bytes 4 to 10 in the file are free

Conflict!

P1

P2

H5Dcreate(D1)

H5Dcreate(D1)

Allocate space in file to store the dataset header.
Bytes 4 to 10 in the file are free.
Create the dataset.

H5Dcreate(D2)

H5Dcreate(D2)

Allocate space in file to store the dataset header.
Bytes 11 to 17 in the file are free.
Create the dataset.



Metadata cache

- To handle synchronization issues, all HDF5 operations that could potentially modify the metadata in an HDF5 file are required to be collective
 - A list of those routines is available in the HDF5 reference manual (<http://www.hdfgroup.org/HDF5/doc/RM/CollectiveCalls.html>)
- If those operations are not collective, how can each process manage its Metadata Cache?
 - Do not have one, i.e. always access metadata directly from disk
 - Disastrous for performance as metadata is usually very small



Managing the metadata cache

- All operations that **modify** metadata in the HDF5 file are collective:
 - All processes will have the same dirty metadata entries in their cache (i.e., metadata that is inconsistent with what is on disk).
 - Processes are not required to have the same clean metadata entries (i.e., metadata that is in sync with what is on disk).
- Internally, the metadata cache running on process 0 is responsible for managing changes to the metadata in the HDF5 file.
 - All the other caches must retain dirty metadata until the process 0 cache tells them that the metadata is clean (i.e., on disk).



Example

- Metadata Cache is clean for all processes:

P0	P1	P2	P3
E1	E1	E4	E12
E2	E7	E6	E32
E3	E8	E1	E1
E4	E2	E5	E4



Example

- All processes call H5Gcreate that modifies metadata entry E3 in the file:

P0	P1	P2	P3
E3	E3	E3	E3
E1	E1	E4	E12
E2	E7	E6	E32
E4	E8	E1	E1



Example

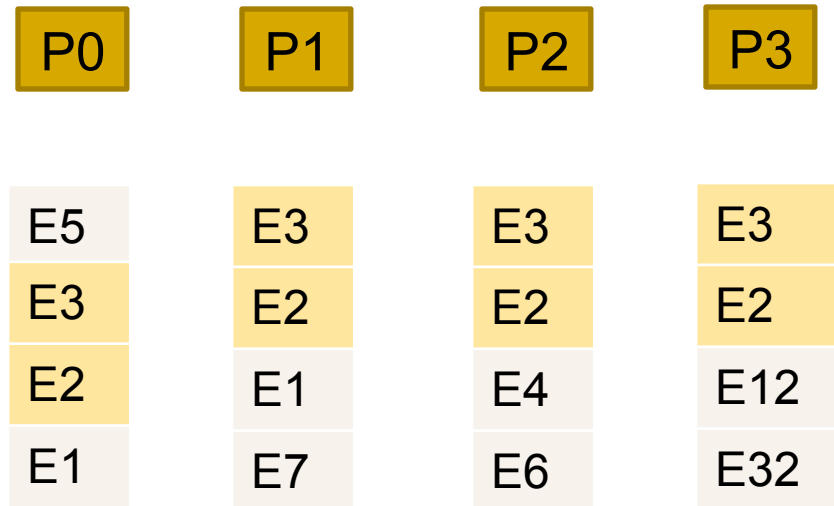
- All processes call H5Dcreate that modifies metadata entry E2 in the file:

P0	P1	P2	P3
E3	E3	E3	E3
E2	E2	E2	E2
E1	E1	E4	E12
E4	E7	E6	E32



Example

- Process 0 calls H5Dopen on a dataset accessing entry E5





Flushing the cache

- Initiated when:
 - The size of dirty entries in cache exceeds a certain threshold
 - The user calls a flush
- The actual flush of metadata entries to disk is currently implemented in two ways:
 - Single Process (Process 0) write
 - Distributed write



Single Process (Process 0) write

- All processes enter a synchronization point.
- Process 0 writes all the dirty entries to disk while other processes wait and do nothing
- Process 0 marks all the dirty entries as clean
- Process 0 broadcasts the cleaned entries to all processes that marks them as clean too



Distributed write

- All processes enter a synchronization point.
- Process 0 broadcasts the metadata that needs to be flushed to all processes
- Using a distributed algorithm each determines what part of the metadata cache entries it needs to write, and writes them to disk independently
- All processes mark the flushed metadata as clean



PARALLEL TOOLS



Parallel tools

- h5perf
 - Performance measuring tool showing I/O performance for different I/O APIs



h5perf

- An I/O performance measurement tool
- Tests 3 File I/O APIs:
 - POSIX I/O (open/write/read/close...)
 - MPI-I/O (MPI_File_{open,write,read,close})
 - HDF5 (H5Fopen/H5Dwrite/H5Dread/H5Fclose)
- An indication of I/O speed upper limits



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



UPCOMING FEATURES IN HDF5



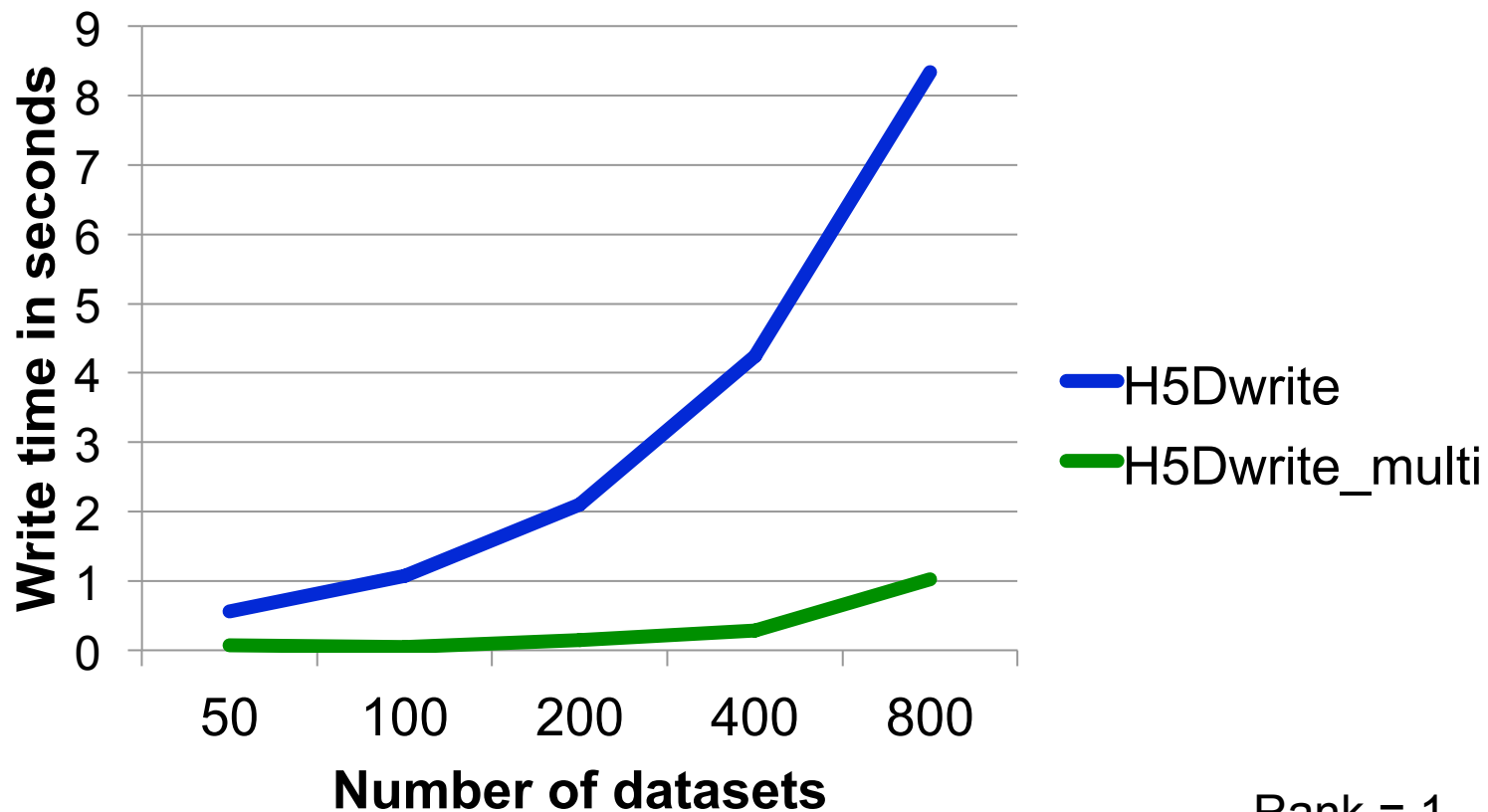
PHDF5 Improvements in Progress

- Multi-dataset read/write operations
 - Allows single collective operation on multiple datasets
 - Similar to PnetCDF “write-combining” feature
 - `H5Dmulti_read/write(<array of datasets, selections, etc>)`
 - Order of magnitude speedup (see next slides)



H5Dwrite vs. H5Dwrite_multi

Chunked floating-point datasets

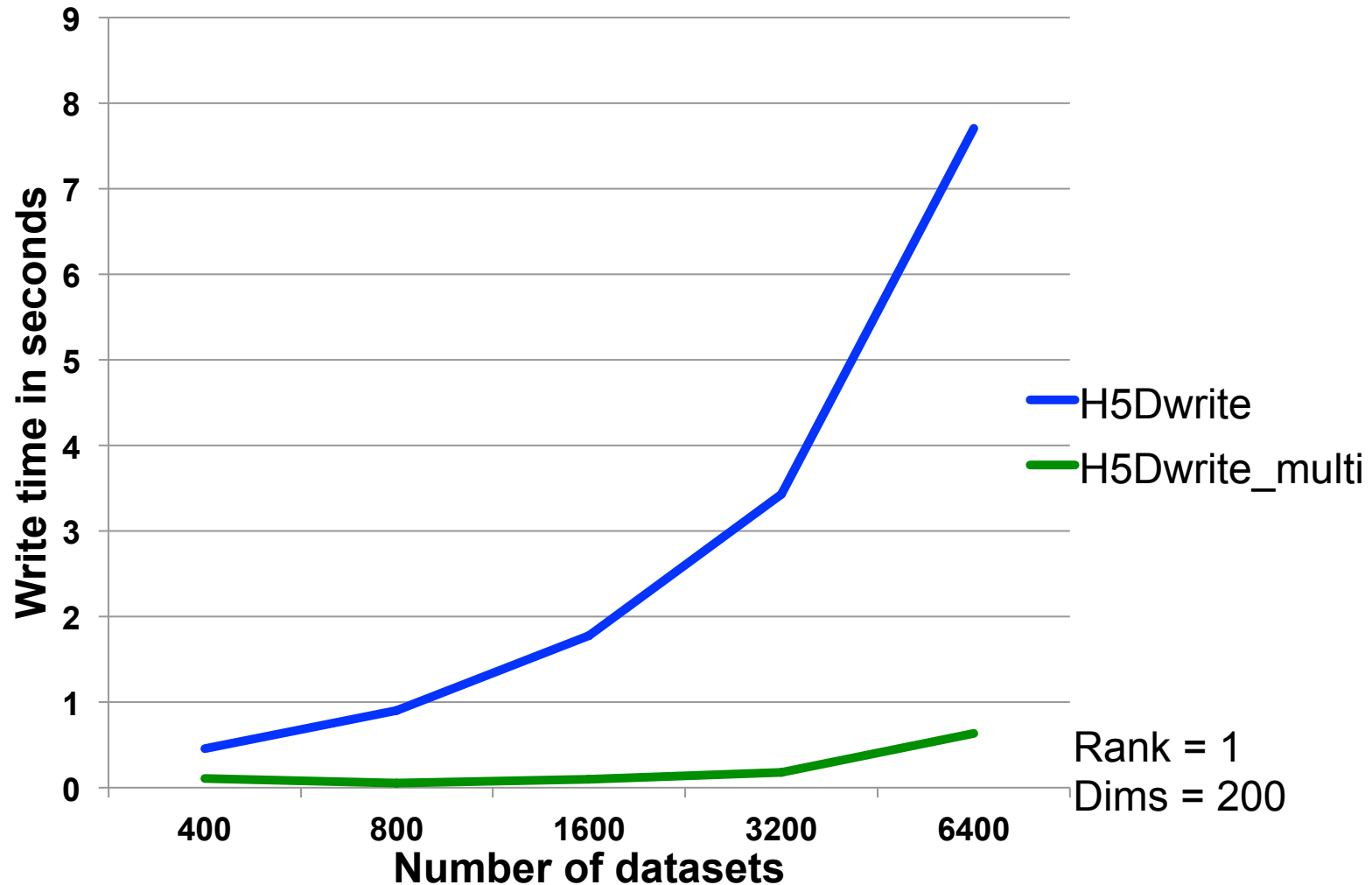


Rank = 1
Dims = 200
Chunk size = 20



H5Dwrite vs. H5Dwrite_multi

Contiguous floating-point datasets





PHDF5 Improvements in Progress

- Avoid file truncation
 - File format currently requires call to truncate file, when closing
 - Expensive in parallel (`MPI_File_set_size`)
 - Change to file format will eliminate truncate call



PHDF5 Improvements in Progress

- Collective Object Open
 - Currently, object open is independent
 - All processes perform I/O to read metadata from file, resulting in I/O storm at file system
 - Change will allow a single process to read, then broadcast metadata to other processes
- Virtual Object Layer (VOL)
- I/O Autotuning



VIRTUAL OBJECT LAYER (VOL)

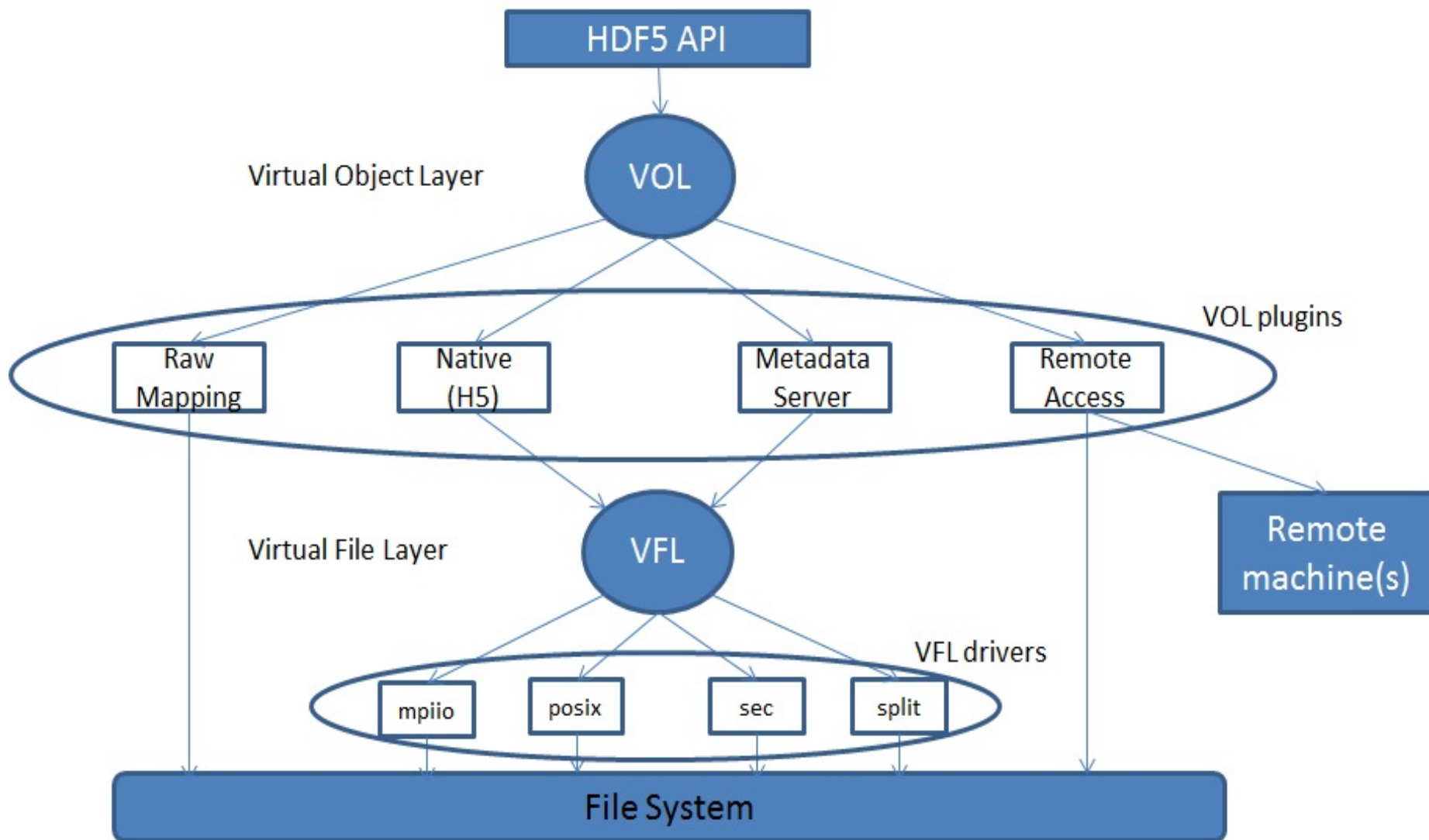


Virtual Object Layer (VOL)

- Goal
 - Provide an application with the HDF5 data model and API, but allow different underlying storage mechanisms
- New layer below HDF5 API
 - Intercepts all API calls that can touch the data on disk and routes them to a VOL plugin
- Potential VOL plugins:
 - Native HDF5 driver (writes to HDF5 file)
 - Raw driver (maps groups to file system directories and datasets to files in directories)
 - Remote driver (the file exists on a remote machine)



Virtual Object Layer





Why not use the VFL?

- VFL is implemented below the HDF5 abstract model
 - Deals with blocks of bytes in the storage container
 - Does not recognize HDF5 objects nor abstract operations on those objects
- VOL is layered right below the API layer to capture the HDF5 model



Sample API Function Implementation

```
hid_t H5Dcreate2 (hid_t loc_id, const char *name,
hid_t type_id, hid_t space_id, hid_t lcpl_id, hid_t
dcpl_id, hid_t dapl_id) {
/* Check arguments */

    ...

/* call corresponding VOL callback for H5Dcreate */
    dset_id = H5_VOL_create (TYPE_DATASET, ...);
/*
    Return result to user (yes the dataset is created,
    or no here is the error)
*/
    return dset_id;
}
```



Work in progress: VOL

CONSIDERATIONS



VOL Plugin Selection

- Use a pre-defined VOL plugin:

```
hid_t fapl = H5Pcreate(H5P_FILE_ACCESS);  
H5Pset_fapl_mds_vol(fapl, ...);  
hid_t file = H5Fcreate("foo.h5", ..., ..., fapl);  
H5Pclose(fapl);
```

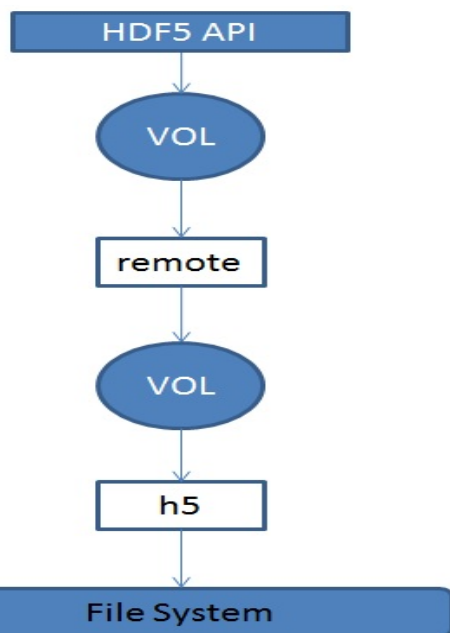
- Register user defined VOL plugin:

```
H5VOLregister (H5VOL_class_t *cls)  
H5VOLunregister (hid_t driver_id)  
H5Pget_plugin_info (hid_t plist_id)
```




Interchanging and Stacking Plugins

- Interchanging VOL plugins
 - Should be a valid thing to do
 - User's responsibility to ensure plugins coexist
- Stacking plugins

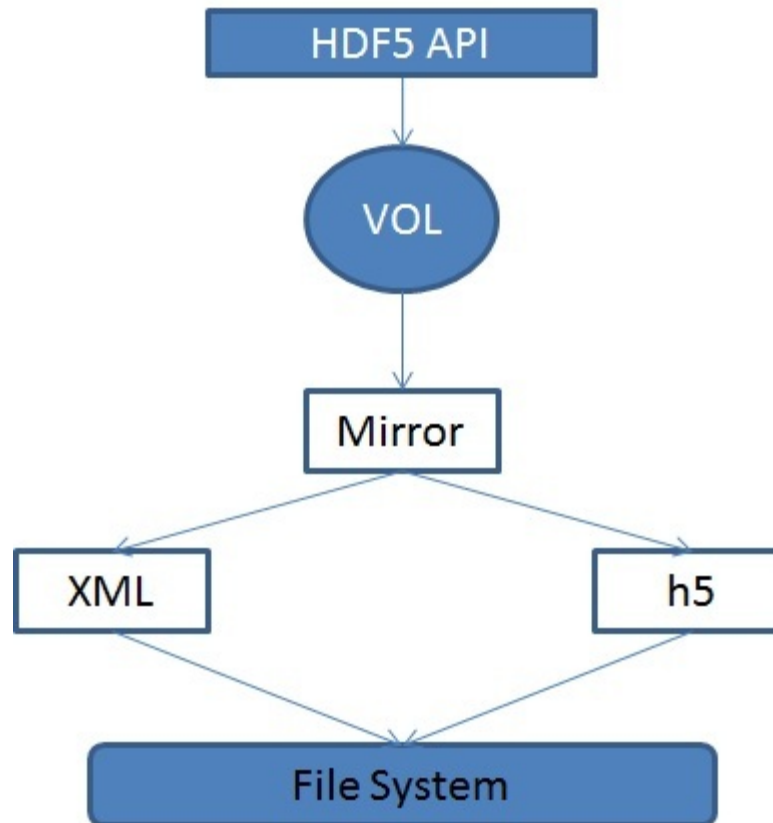


- Stacking should make sense.
- For example, the first VOL plugin in a stack could be a statistics plugin, that does nothing but gather information on what API calls are made and their corresponding parameters.

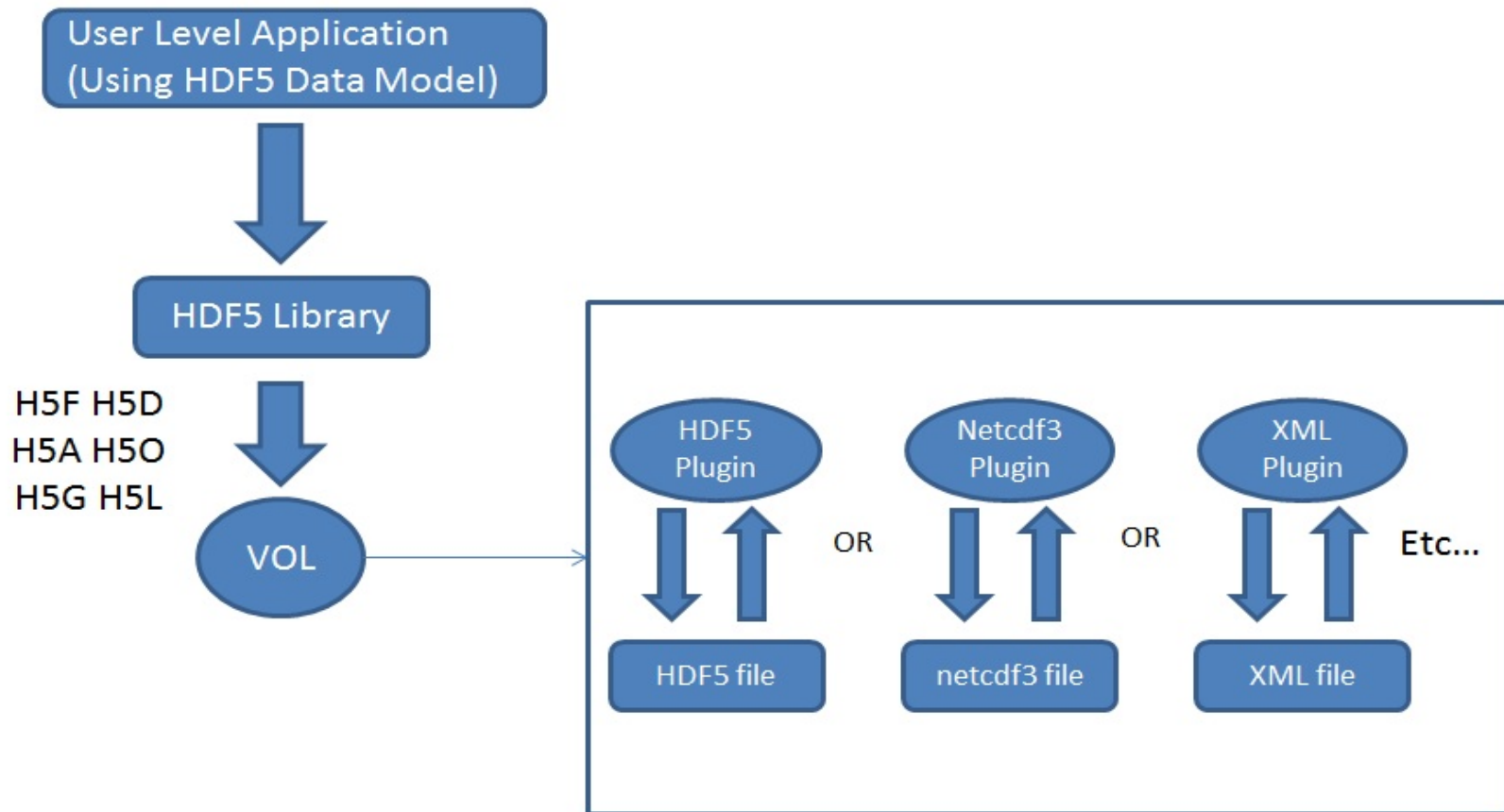


Mirroring

- Extension to stacking
- HDF5 API calls are forwarded through a mirror plugin to two or more VOL plugins

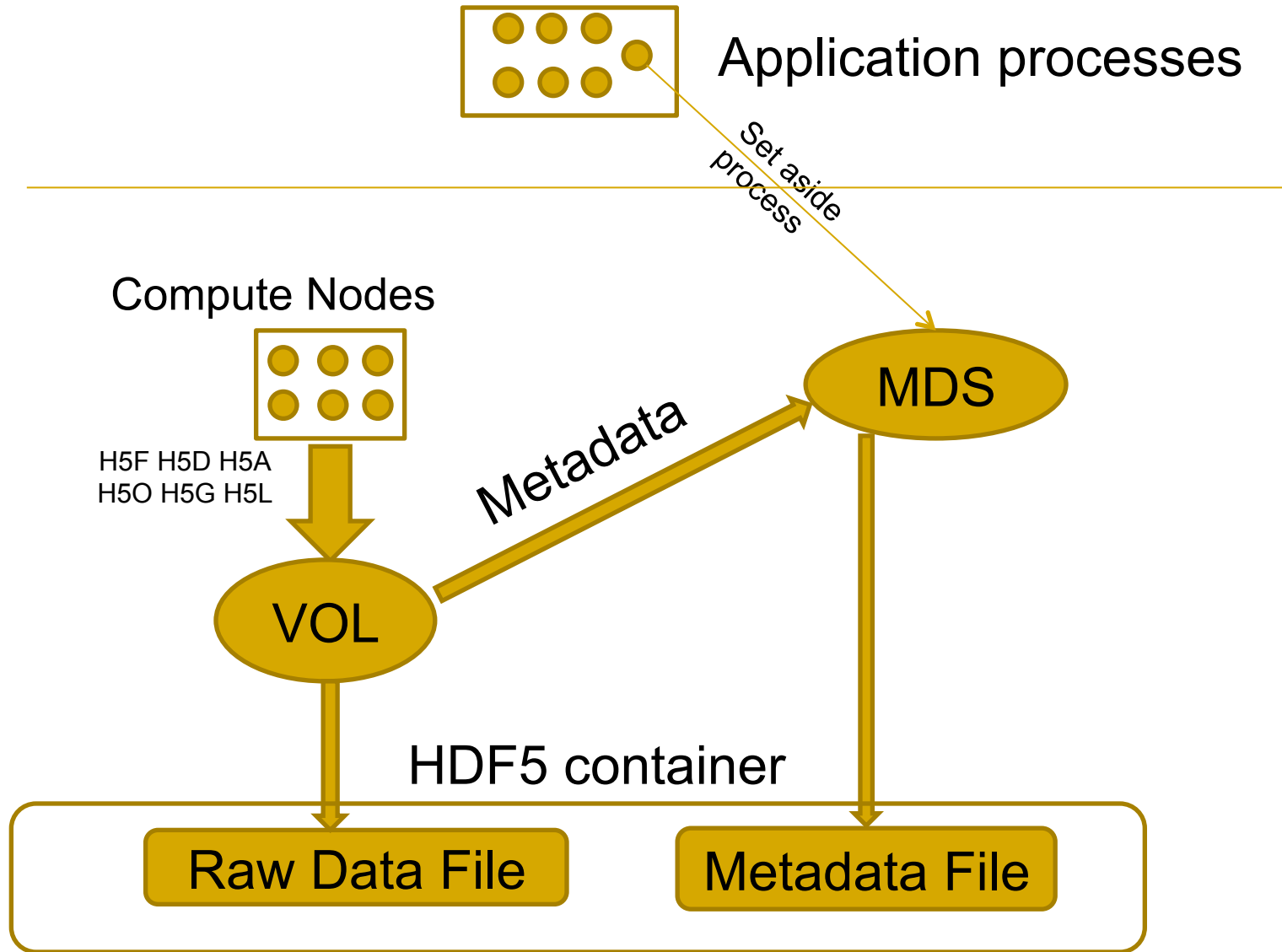


- Different File Format plugins





Sample Plugins: Metadata Server





Raw Plugin

- The flexibility of the virtual object layer provides developers with the option to abandon the single file, binary format like the native HDF5 implementation.
- A “raw” file format could map HDF5 objects (groups, datasets, etc ...) to file system objects (directories, files, etc ...).
- The entire set of raw file system objects created would represent one HDF5 container.
- Useful to the PLFS package (<http://institute.lanl.gov/plfs/>)



Remote Plugin

- A remote VOL plugin would allow access to files located remotely.
- The plugin could have an HDF5 server module located where the HDF5 file resides and listens to incoming requests from a remote process.
- Use case: Remote visualization
 - Large, remote datasets are very expensive to migrate to the local visualization system.
 - It would be faster to just enable *in situ* visualization to remotely access the data using the HDF5 API.



Implementation

- VOL Class
 - Data structure containing general variables and a collection of function pointers for HDF5 API calls
- Function Callbacks
 - API routines that potentially touch data on disk
 - H5F, H5D, H5A, H5O, H5G, H5L, and H5T



Implementation

- We will end up with a large set of function callbacks:
 - Lump all the functions together into one data structure OR
 - Have a general class that contains all common functions, and then children of that class that contain functions specific to certain HDF5 objects OR
 - For each object have a set of callbacks that are specific to that object (This is design choice that has been taken).



Filters

- Need to keep HDF5 filters in mind
- Where is the filter applied, before or after the VOL plugin?
 - Logical guess now would be before, to avoid having all plugins deal with filters



Current status of VOL

- ?



Research Focus -

AUTOTUNING



Autotuning Background

- Software Autotuning:
 - Employ empirical techniques to evaluate a set of alternative mappings of computation kernels to an architecture and select the mapping that obtains the best performance.
- Autotuning Categories:
 - Self-tuning library generators such as ATLAS, PhiPAC and OSKI for linear algebra, etc.
 - Compiler-based autotuners that automatically generate and search a set of alternative implementations of a computation
 - Application-level autotuners that automate empirical search across a set of parameter values proposed by the application programmer



HDF5 Autotuning

- Why?
 - Because the dominant I/O support request at NERSC is poor I/O performance, many/most of which can be solved by enabling Lustre striping, or tuning another I/O parameter
 - *Scientists shouldn't have to figure this stuff out!*
- Two Areas of Focus:
 - Evaluate techniques for autotuning HPC application I/O
 - File system, MPI, HDF5
 - Record and Replay HDF5 I/O operations



Autotuning HPC I/O

- Goal: Avoid tuning each application to each machine and file system
 - Create I/O autotuner library that can inject “optimal” parameters for I/O operations on a given system
- Using Darshan* tool to create wrappers for HDF5 calls
 - Application can be dynamically linked with I/O autotuning library
 - No changes to application or HDF5 library
- Using several HPC applications currently:
 - VPIC, GCRM, Vorpai

* - <http://www.mcs.anl.gov/research/projects/darshan/>



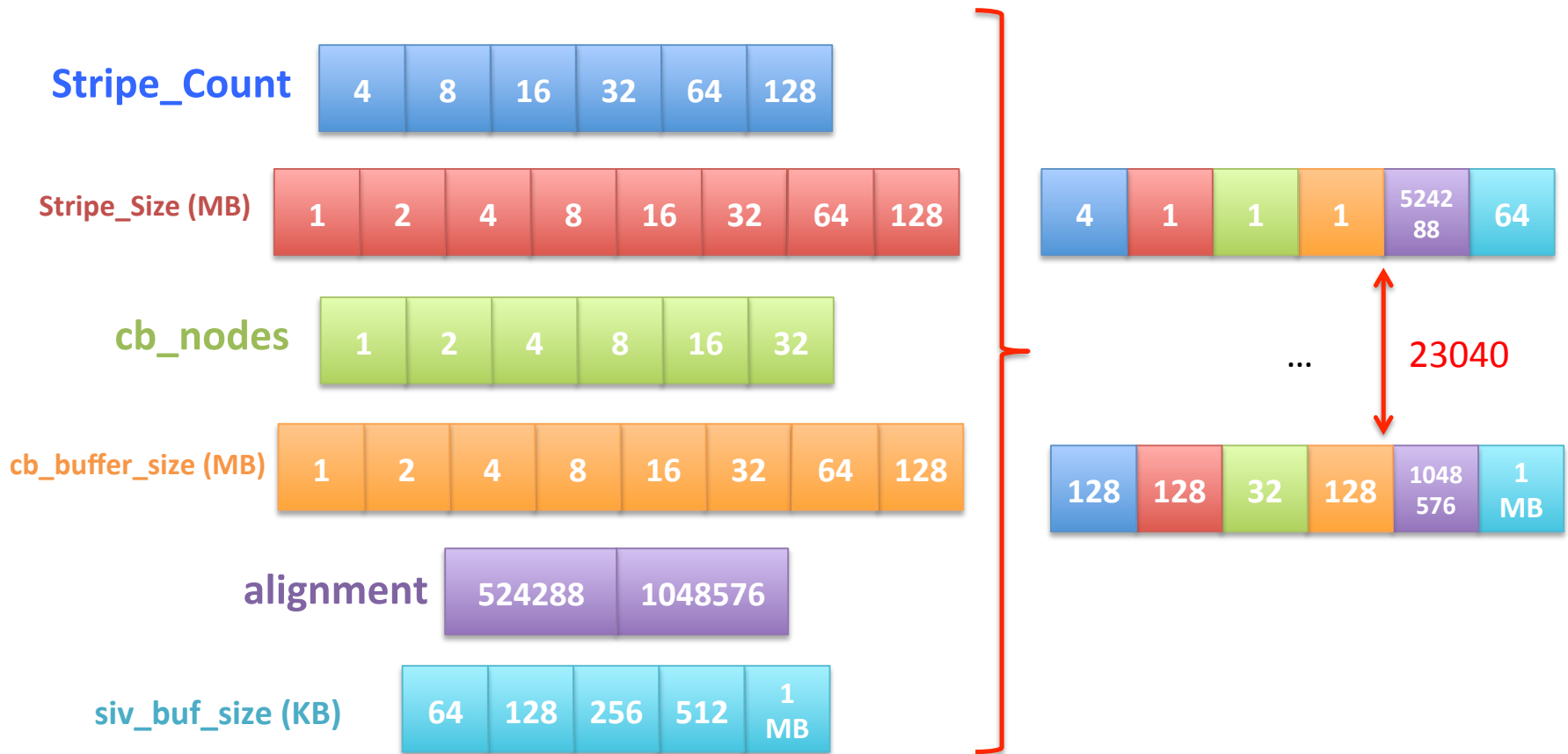
Autotuning HPC I/O

- Initial parameters of interest
 - File System (Lustre): stripe count, stripe unit
 - MPI-I/O: Collective buffer size, coll. buffer nodes
 - HDF5: Alignment, sieve buffer size



Autotuning HPC I/O

The whole space visualized





Autotuning HPC I/O

- Autotuning Exploration/Generation Process:
 - Iterate over running application many times:
 - Intercept application's I/O calls
 - Inject autotuning parameters
 - Measure resulting performance
 - Analyze performance information from many application runs to create configuration file, with best parameters found for application/machine/file system



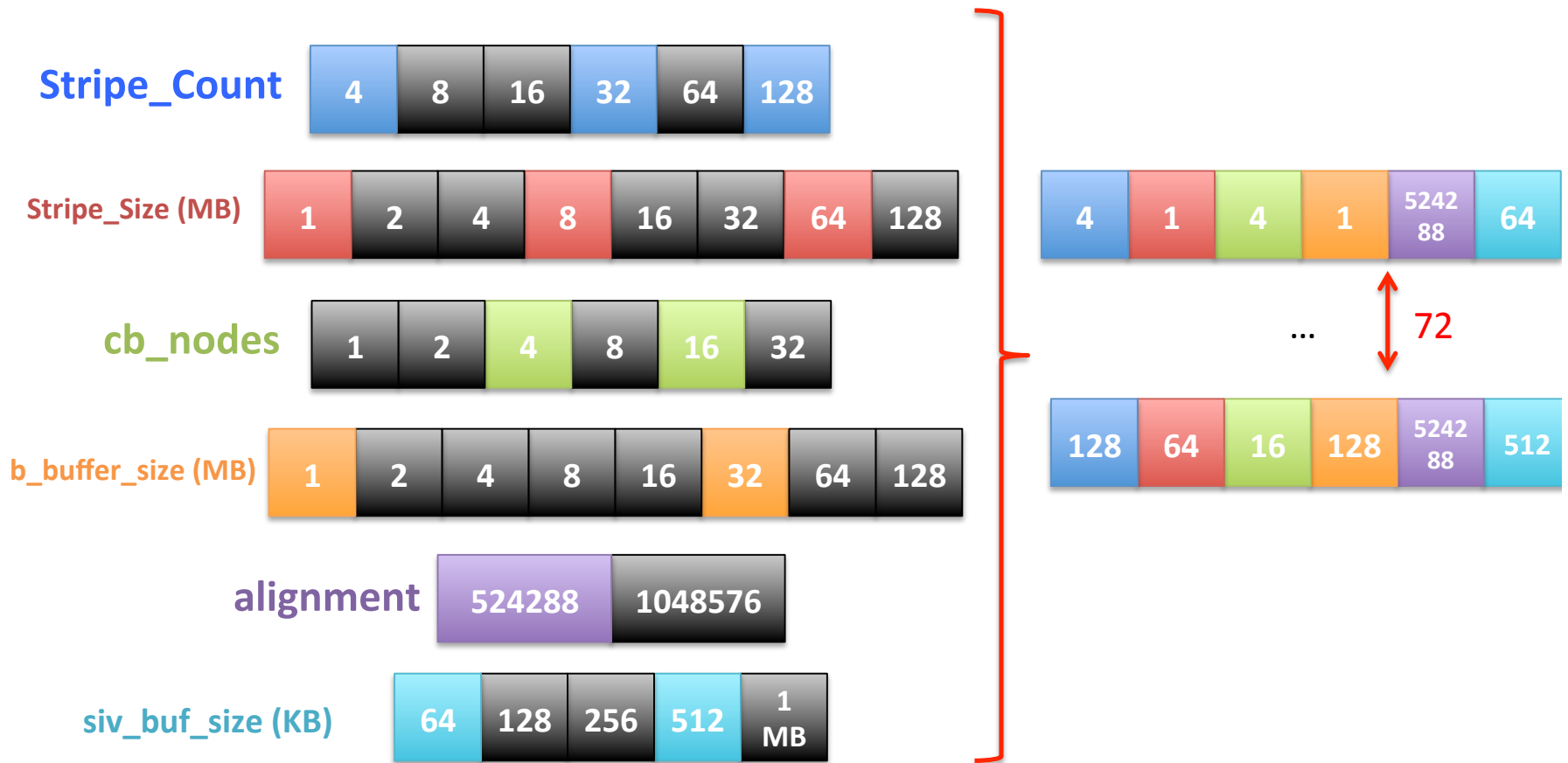
Autotuning HPC I/O

- Using the I/O Autotuning Library:
 - Dynamically link with I/O autotuner library
 - I/O autotuner library automatically reads parameters from config file created during exploration process
 - I/O autotuner automatically injects autotuning parameters as application operates



Autotuning HPC I/O

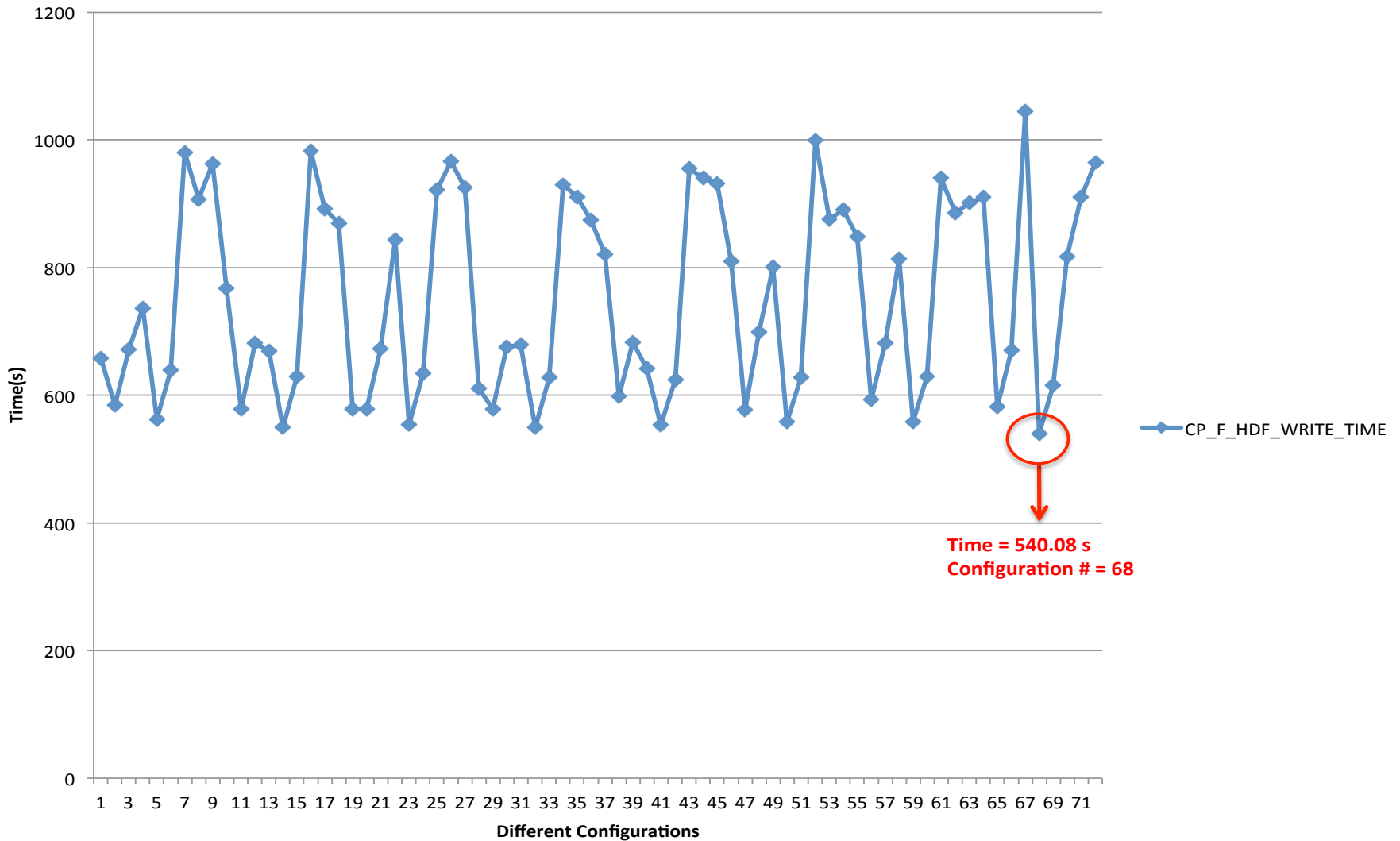
Smaller set of space visualized





Autotuning HPC I/O

Result of Running Our Script using 72 Configuration files on 32 Cores/1 Node of Ranger





Configuration #68

```
<Parameters>
  <High_Level_IO_Library>
    <sieve_buf_size> 524280 </sieve_buf_size>
    <alignment> 262144,524288 </alignment>
    <!-- H5Pset_alignment function gets 2 args: (Threshold, Alignment) -
  </High_Level_IO_Library>

  <Middleware_Layer>
    <cb_buffer_size> 134217728 </cb_buffer_size>
    <cb_nodes> 16 </cb_nodes>
  </Middleware_Layer>

  <Parallel_File_System>
    <striping_factor> 32 </striping_factor>
    <striping_unit> 8388608 </striping_unit>
  </Parallel_File_System>
</Parameters>
```



Autotuning in HDF5

- “Auto-Tuning of Parallel IO Parameters for HDF5 Applications”, Babak Behzad, et al, poster @ SC12
- “Taming Parallel I/O Complexity with Auto-Tuning”, Babak Behzad, et al, SC13



Autotuning HPC I/O

- Remaining research:
 - Determine “speed of light” for I/O on system and use that to define “good enough” performance
 - Entire space is too large to fully explore, we are now evaluating genetic algorithm techniques to help find “good enough” parameters
 - How to factor out “unlucky” exploration runs
 - Methods for avoiding overriding application parameters with autotuned parameters



Thank You!

Questions?