FITSIO, NetCDF, HDF4 and HDF5 Performance

Some Benchmarks Results

Elena Pourmal

NCSA

Benchmark Environment (software)

- **Software**
  - **HDF4 r1.4**
  - **HDF5 1.4.2 and 1.4.2-post1 (both sequential only)**
  - **NetCDF 3.5**
  - **FITSIO version 2.2**
  - **‘System” benchmark uses** `open, write, read` **and** `close` **UNIX functions.**
- **each measurement was taken 10 times, best times were collected**

Benchmark Environment (hardware)

- 440-Mhz UltraSPARC i-Iii (Solaris 2.7)
  - 1G memory
- 2 - 550 Mhz Pentium III Xeon (Linux 2.2.18smp)
  - 1G memory
- Dual 450-Mhz Pentium II (FreeBSD 4.4*)*
  - 512 MB memory
  - SCSI-2 disk
- NCSA O2K (IRIX64)
  - http://www.ncsa.uiuc.edu/UserInfo/Resources/Hardware/Origin2000/

Benchmarks

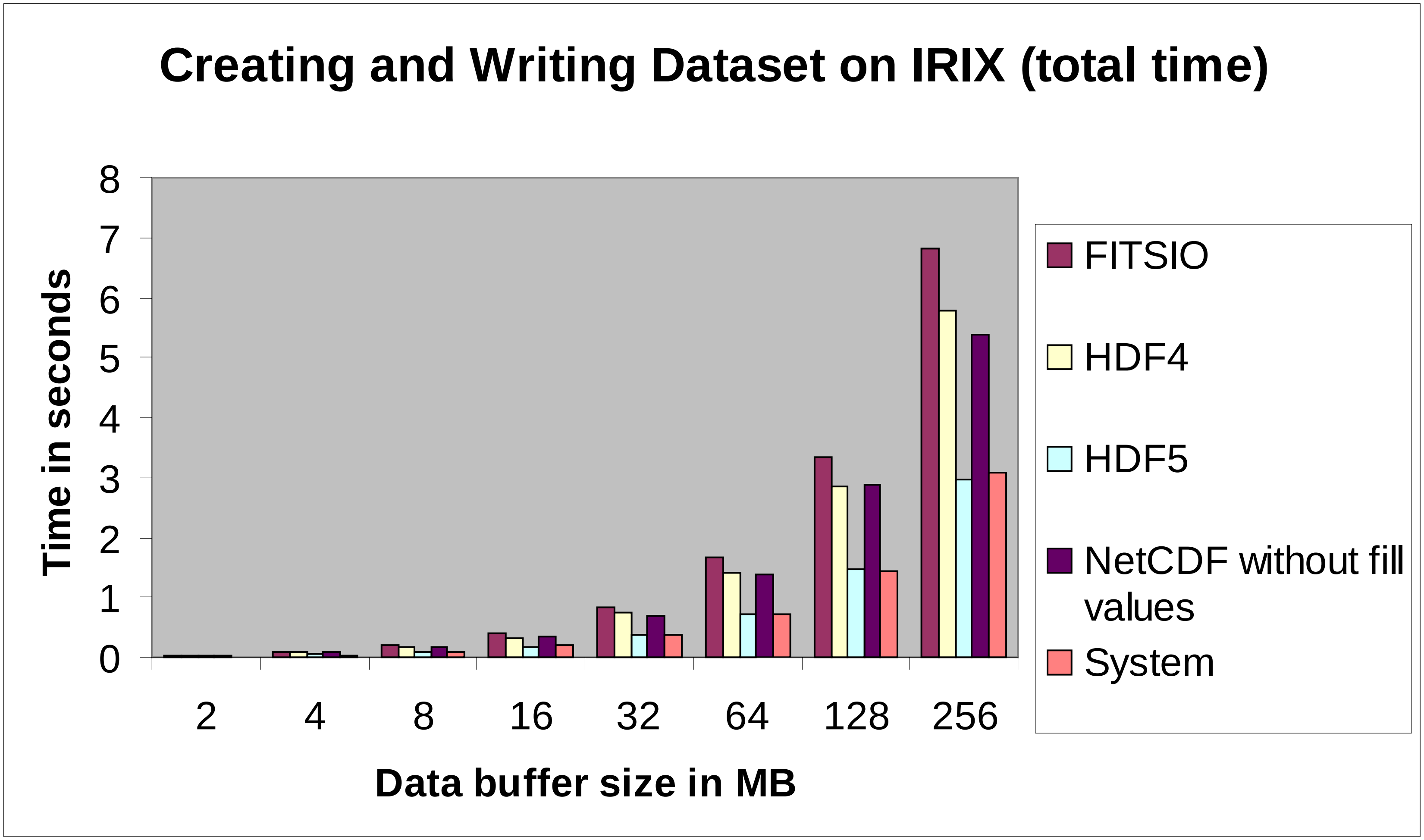
- Creating and writing contiguous dataset; sizes vary from 2MB to 512MB
- Reading contiguous dataset; sizes vary from 2MB to 256MB
- Reading contiguous hyperslab; sizes vary from 1MB to 64MB
- Reading every second element of the hyperslab; sizes of selections vary from 0.25MB to 16MB
- Creating and writing up to 1000 1MB datasets; reading back the dataset created last

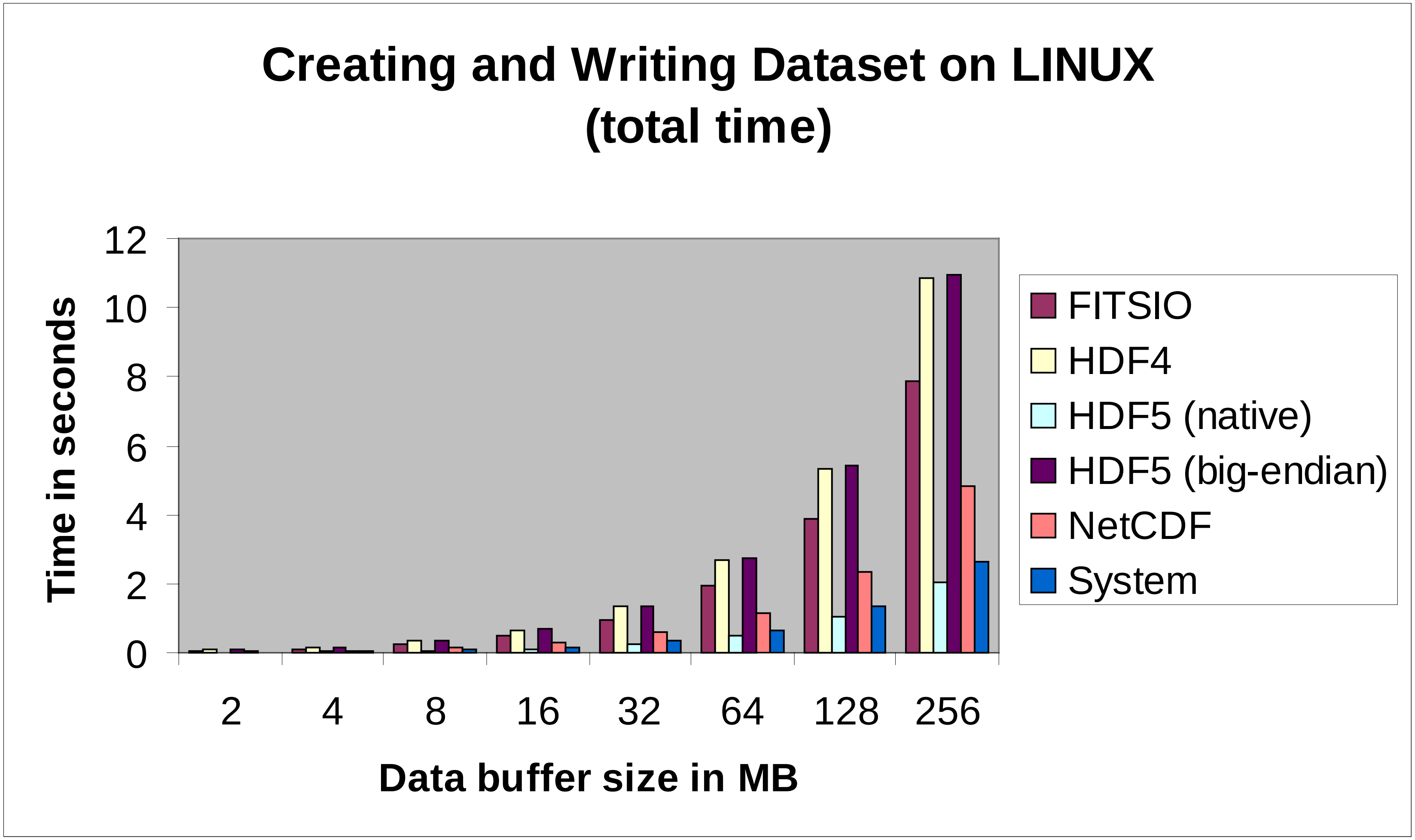
Some Remarks

- “dataset” describes array stored in the FITS, HDF4, HDF5, NetCDF and UNIX binary files, i.e. “dataset” means
  - “primary array” and “extension” for FITSIO
  - “variable” for NetCDF
  - “SDS or scientific data set” for HDF4
  - HDF5 dataset
  - raw data stored in UNIX binary file

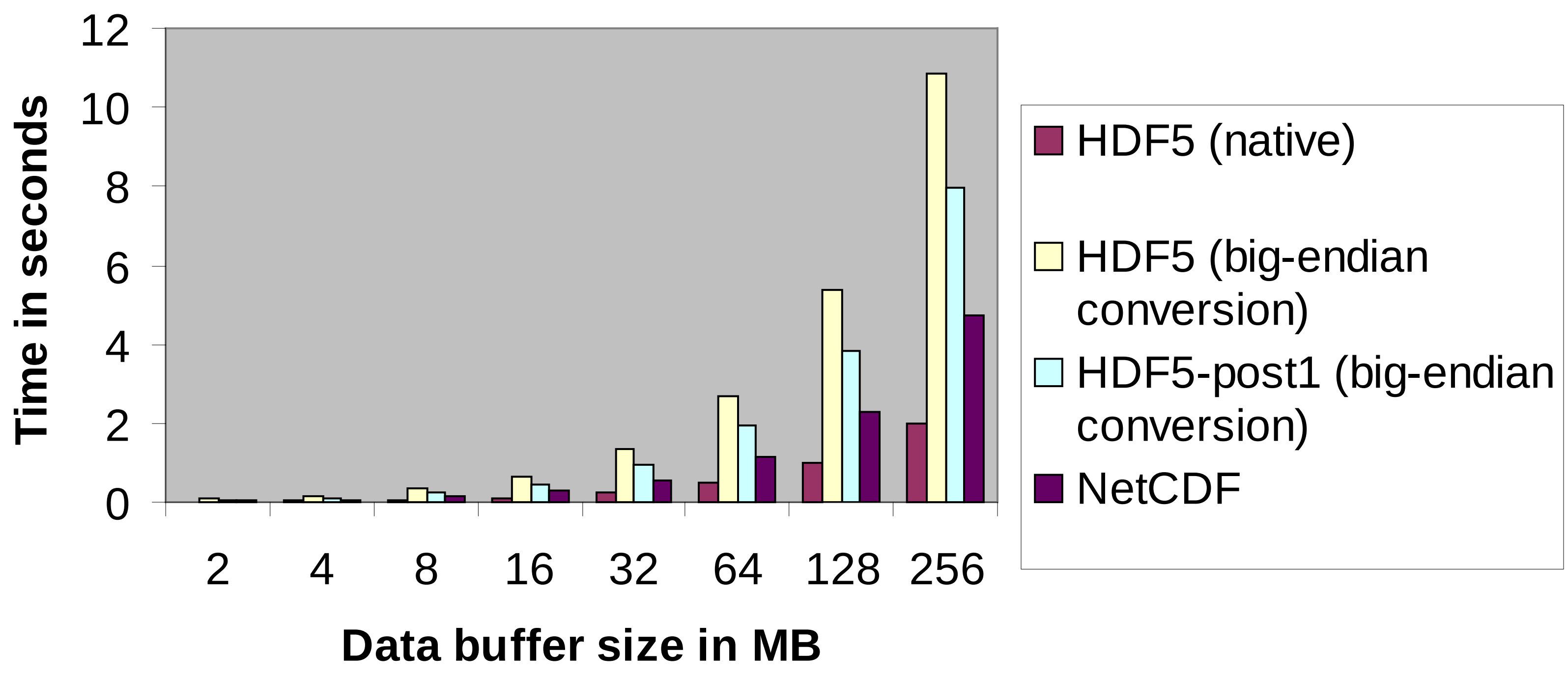
Creating and Writing Contiguous Dataset

- In this test we created a file and stored two dimensional array of short unsigned integers; size of array varied from 2MB and up to 512MB
- We measured
  - Total time to
    - create a file
    - create a dataset
    - write a dataset
    - close the dataset and the file
  - Time to write dataset only


Creating and Writing Dataset on IRIX (total time)
Time in seconds
8
7
6
5
4
3
2
1
0
2
4
8
16
32
64
128
256
Data buffer size in MB
FITSIO
HDF4
HDF5
NetCDF without fill values
System


Creating and Writing Dataset on LINUX
(total time)
Time in seconds
0
2
4
6
8
10
12
2
4
8
16
32
64
128
256
Data buffer size in MB
FITSIO
HDF4
HDF5 (native)
HDF5 (big-endian)
NetCDF
System

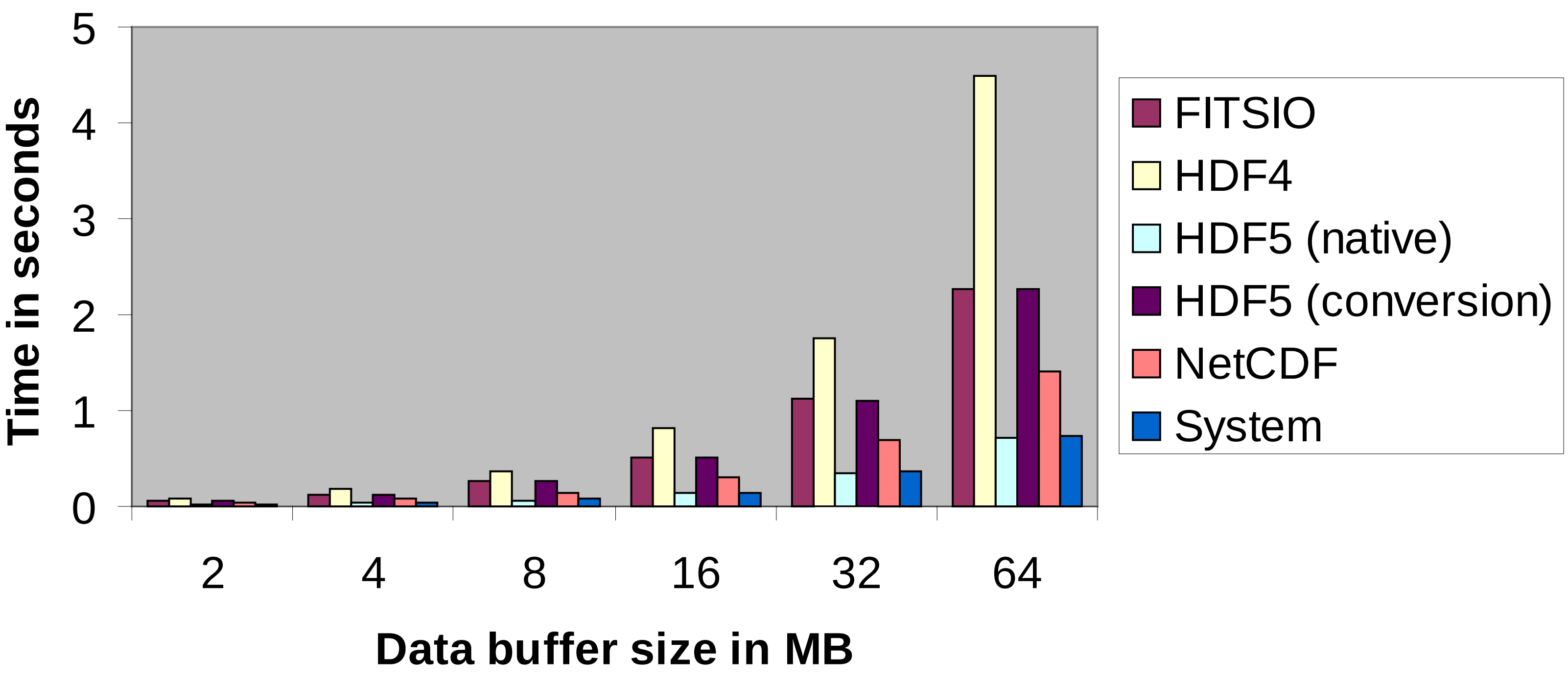
Creating and Writing Dataset on LINUX (write time)

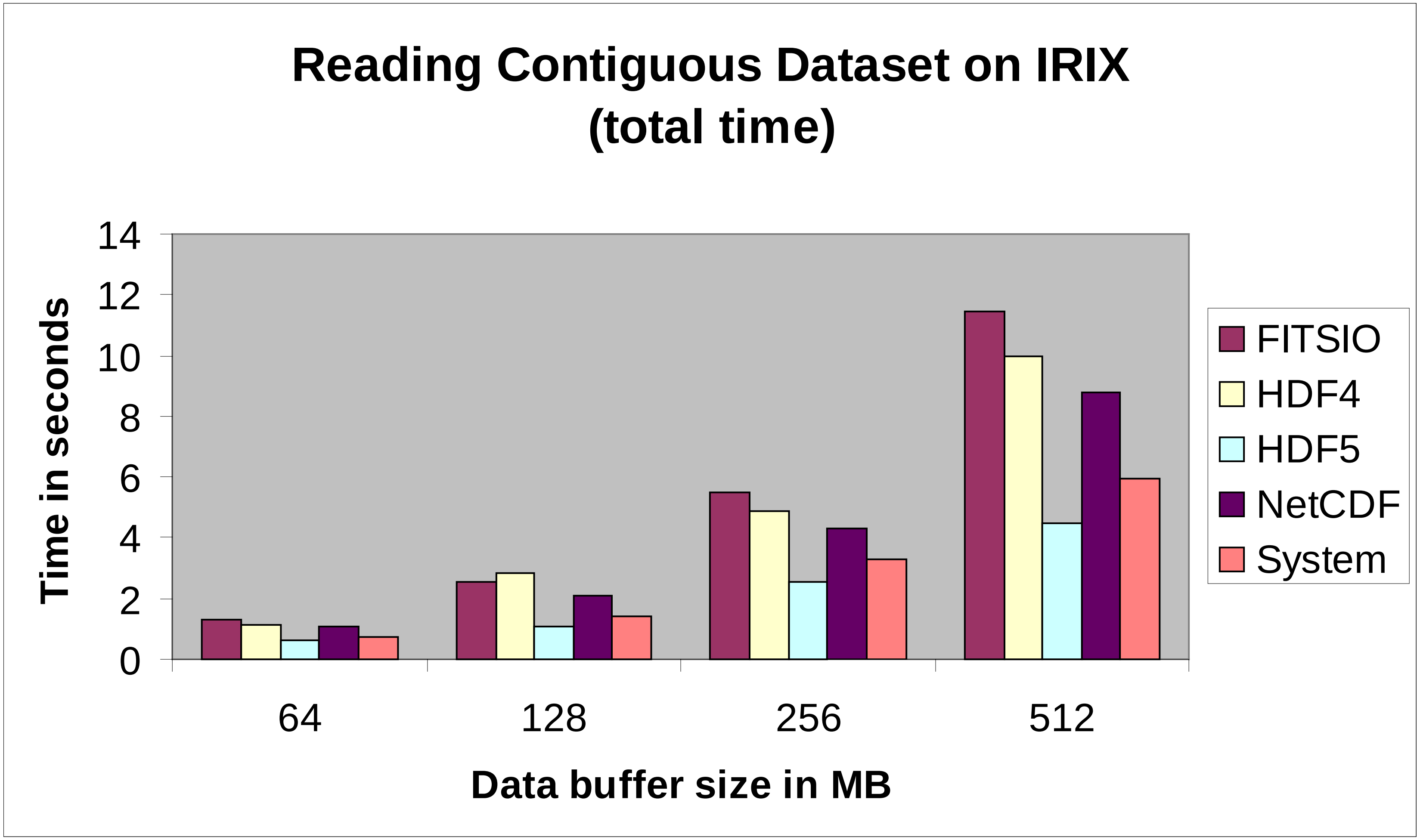


Reading Contiguous Dataset

- In this test we created two dimensional array of short unsigned integers than we read it back; size of array varied from 2MB and up to 512MB
- We measured
  - Total time to
    - open a file
    - open a dataset
    - read a dataset
    - close the dataset and the file
  - Time to read dataset only

Reading Contiguous Dataset on FreeBSD
(read time only)



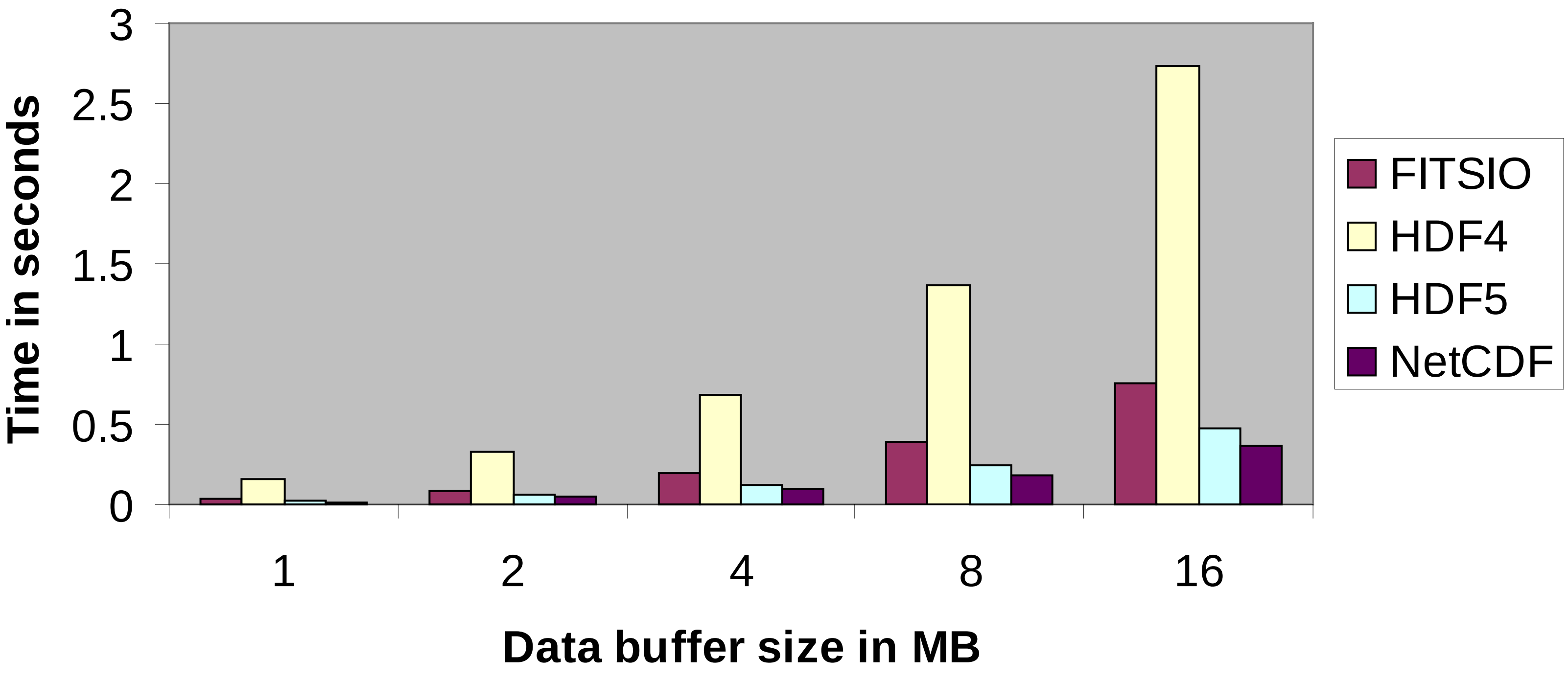


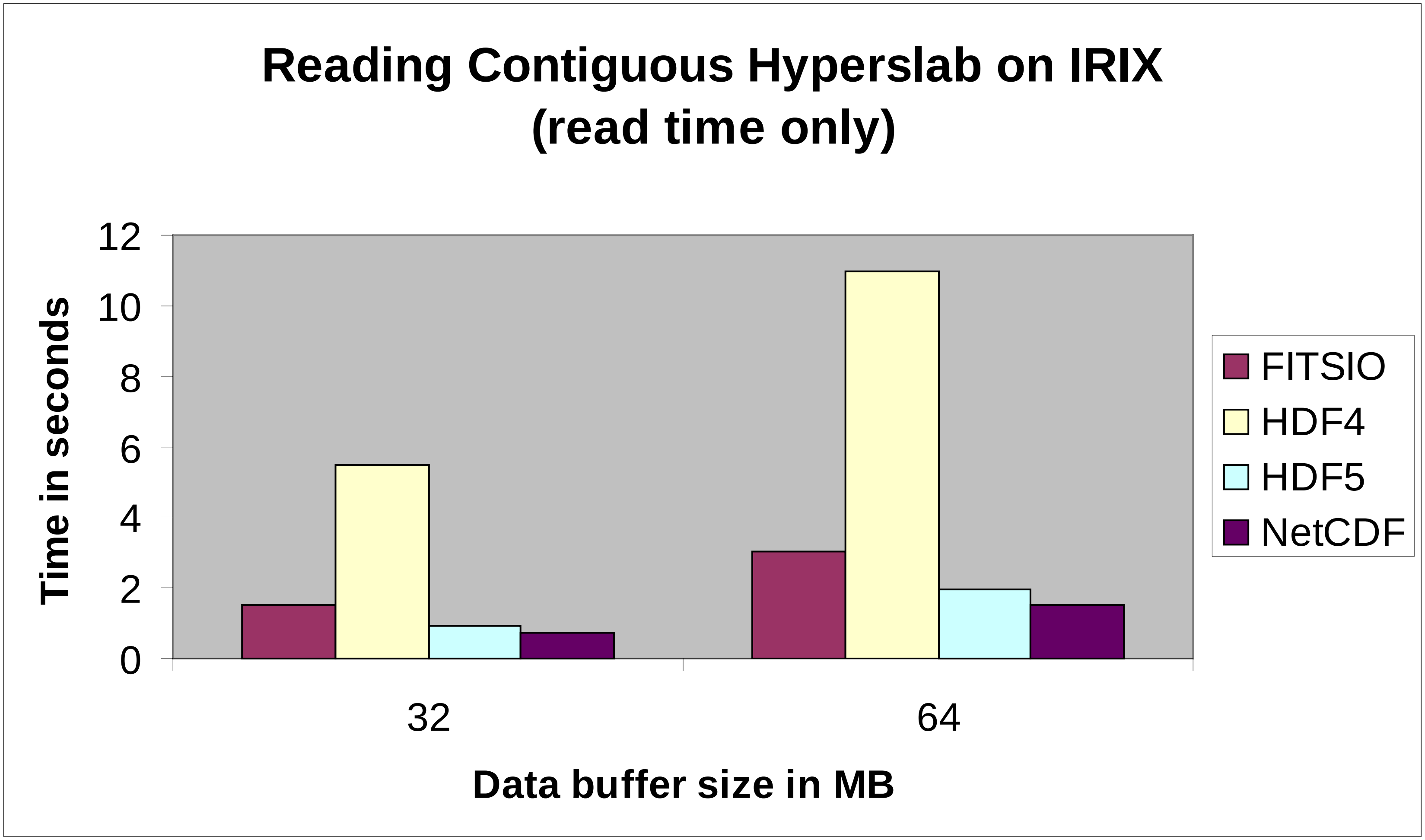
Reading Contiguous Dataset on IRIX (total time)

Reading Contiguous Hyperslab of the Dataset

- In this test we created two dimensional array of short unsigned integers and than read contiguous hyperslab of the dataset; size of the dataset was up 256 MB and size of the hyperslab varied from 1MB up to 64 MB
- We measured
  - Total time to open a file, dataset, select and read hyperslab, close the dataset and the file
  - Time to read hyperslab only

Reading Continuous Hyperslab on IRIX (read time only)



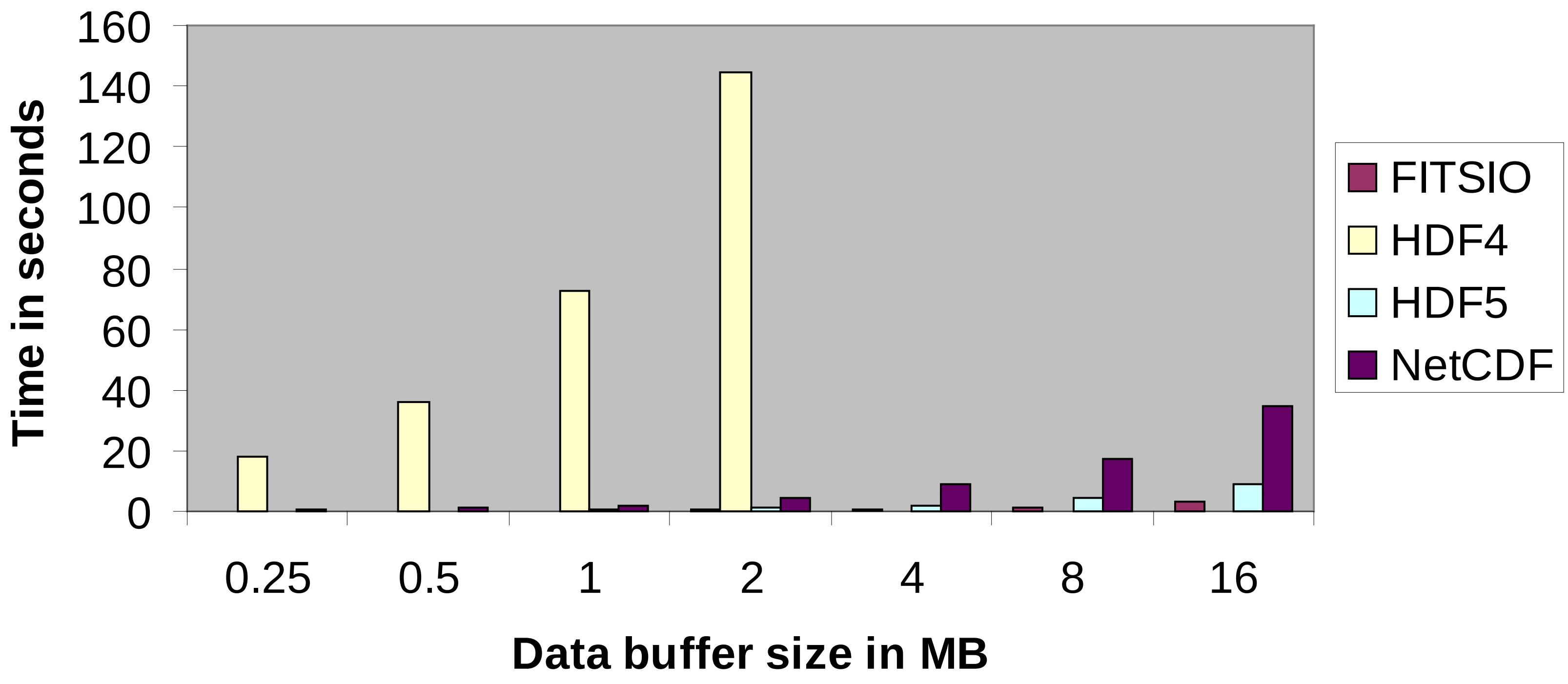


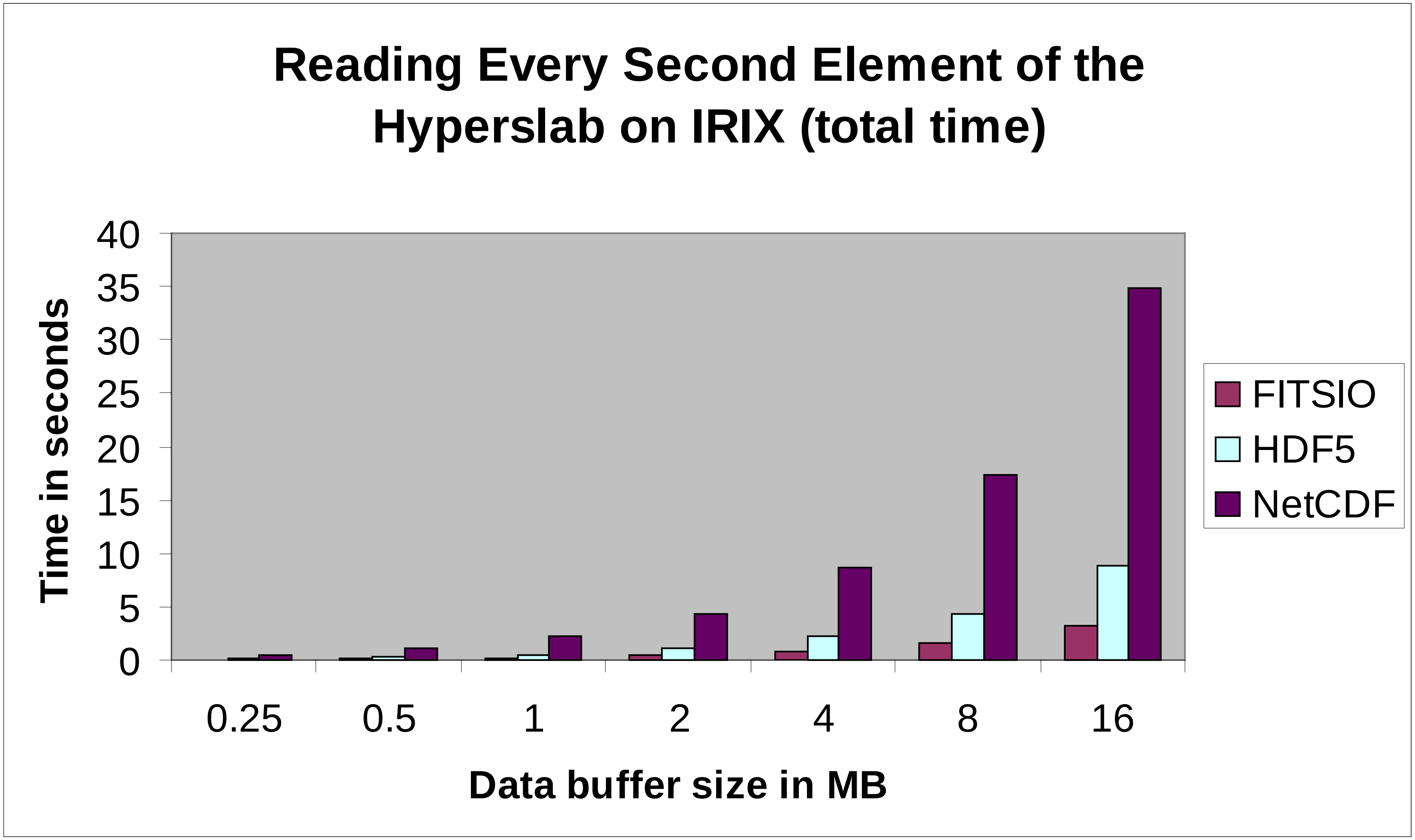
Reading Contiguous Hyperslab on IRIX
(read time only)

Reading Every Second Element in the Hyperslab

- In this test we created 256 MB two dimensional array of short unsigned integers; then we read read back every second element of the selected hyperslab
- We measured
  - Total time to open a file and dataset, select and read every second element of the hyperslab, close the file and dataset
  - Time to read selection only

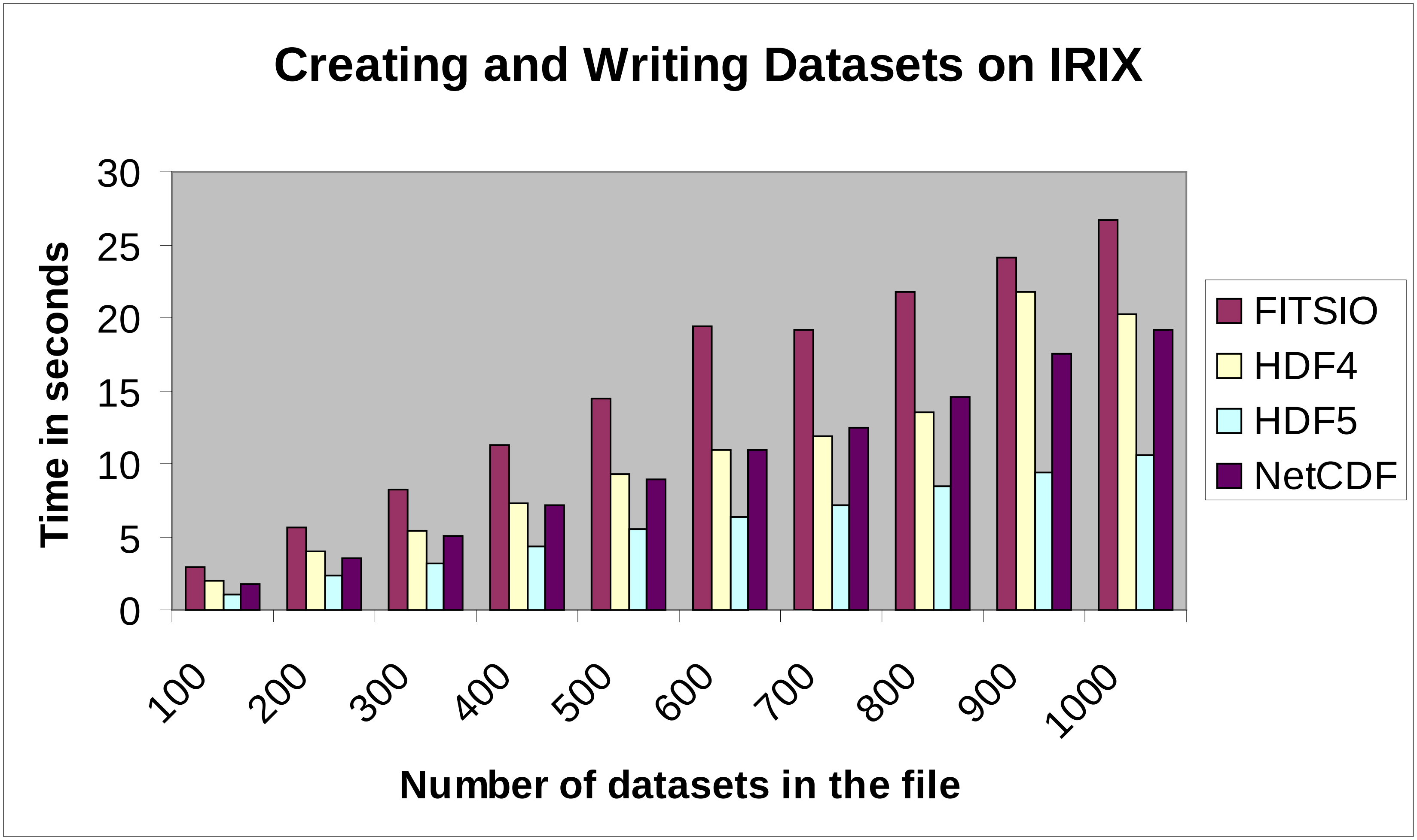
Reading Every Second Element of the Hyperslab on IRIX (total time)

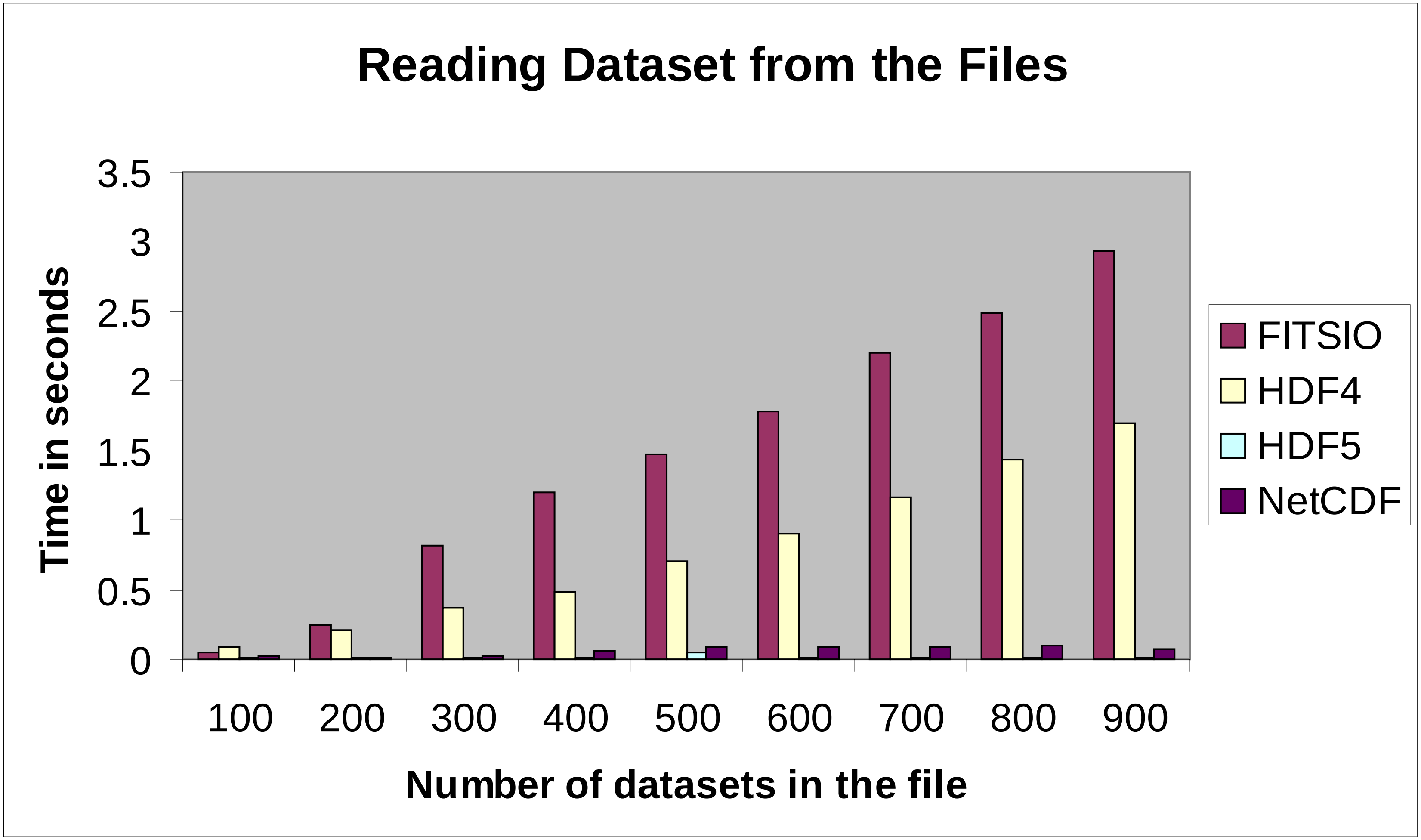



Reading Every Second Element of the Hyperslab on IRIX (total time)
Time in seconds
0
5
10
15
20
25
30
35
40
0.25
0.5
1
2
4
8
16
Data buffer size in MB
FITSIO
HDF5
NetCDF

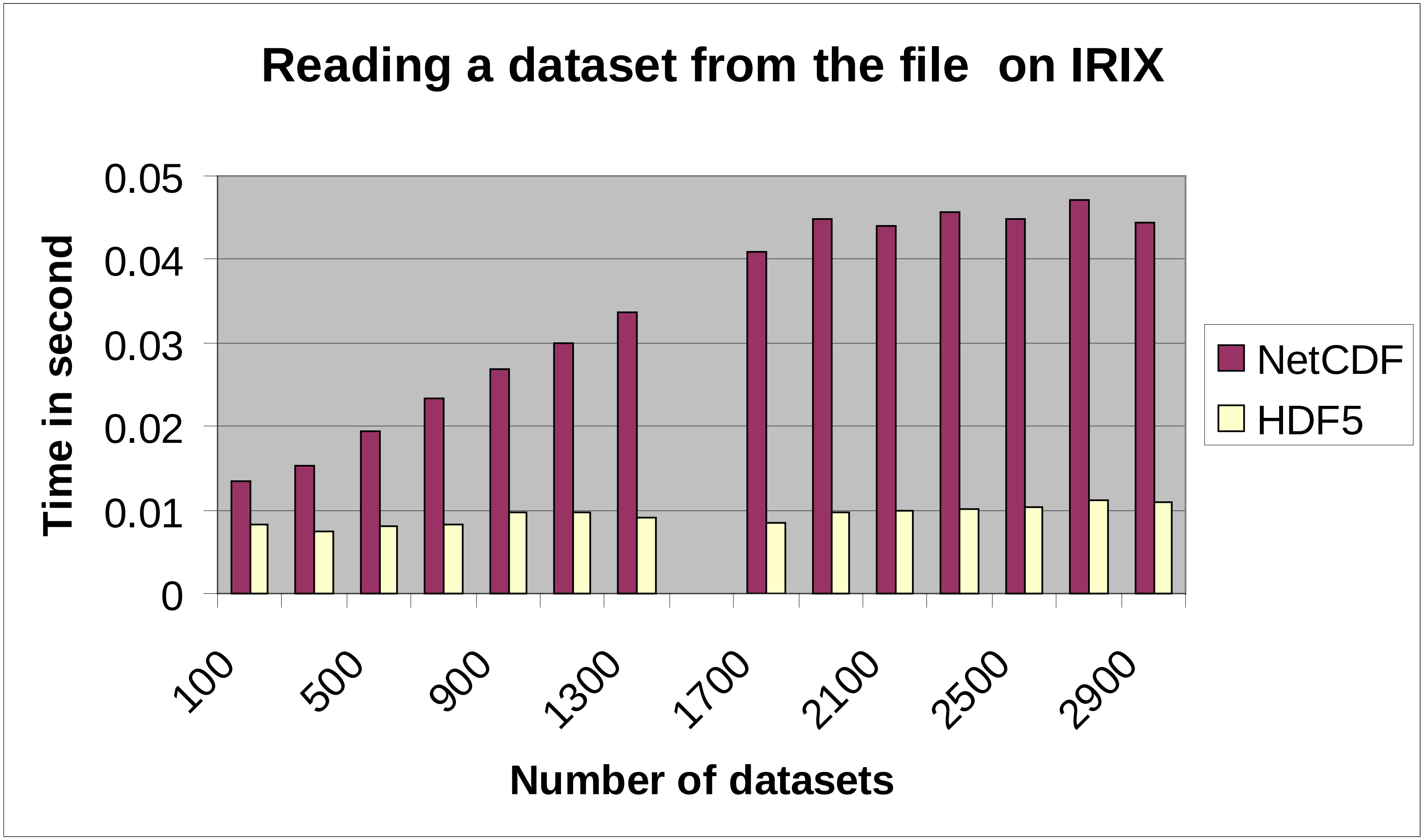
Creating and Writing Multiple Datasets

- **In this test we created up to 1000 1MB two dimensional datasets of short unsigned integers; then we read the last created dataset**
- **We measured**
  - **Time to**
    - **create a file**
    - **create and write N datasets**
    - **close all datasets and the file**
  - **Time to open the file, read N-th dataset and close the file**


Creating and Writing Datasets on IRIX
Time in seconds
0
5
10
15
20
25
30
100
200
300
400
500
600
700
800
900
1000
Number of datasets in the file
FITSIO
HDF4
HDF5
NetCDF



Reading Dataset from the Files


Reading a dataset from the file on IRIX
Time in second
0.05
0.04
0.03
0.02
0.01
0
100
500
900
1300
1700
2100
2500
2900
Number of datasets
NetCDF
HDF5

Summary

- HDF5 is 2-6 times faster when performs native write/read
- HDF5 needs some tuning when datatype conversion is used
- When subsetting is used, HDF5 performs about the same as FITSIO and NetCDF, and 2-6 times faster than HDF4
- HDF5 is an order of magnitude faster in accessing datasets within the file with many objects

Parallel HDF5 Performance

Albert Cheng
NCSA

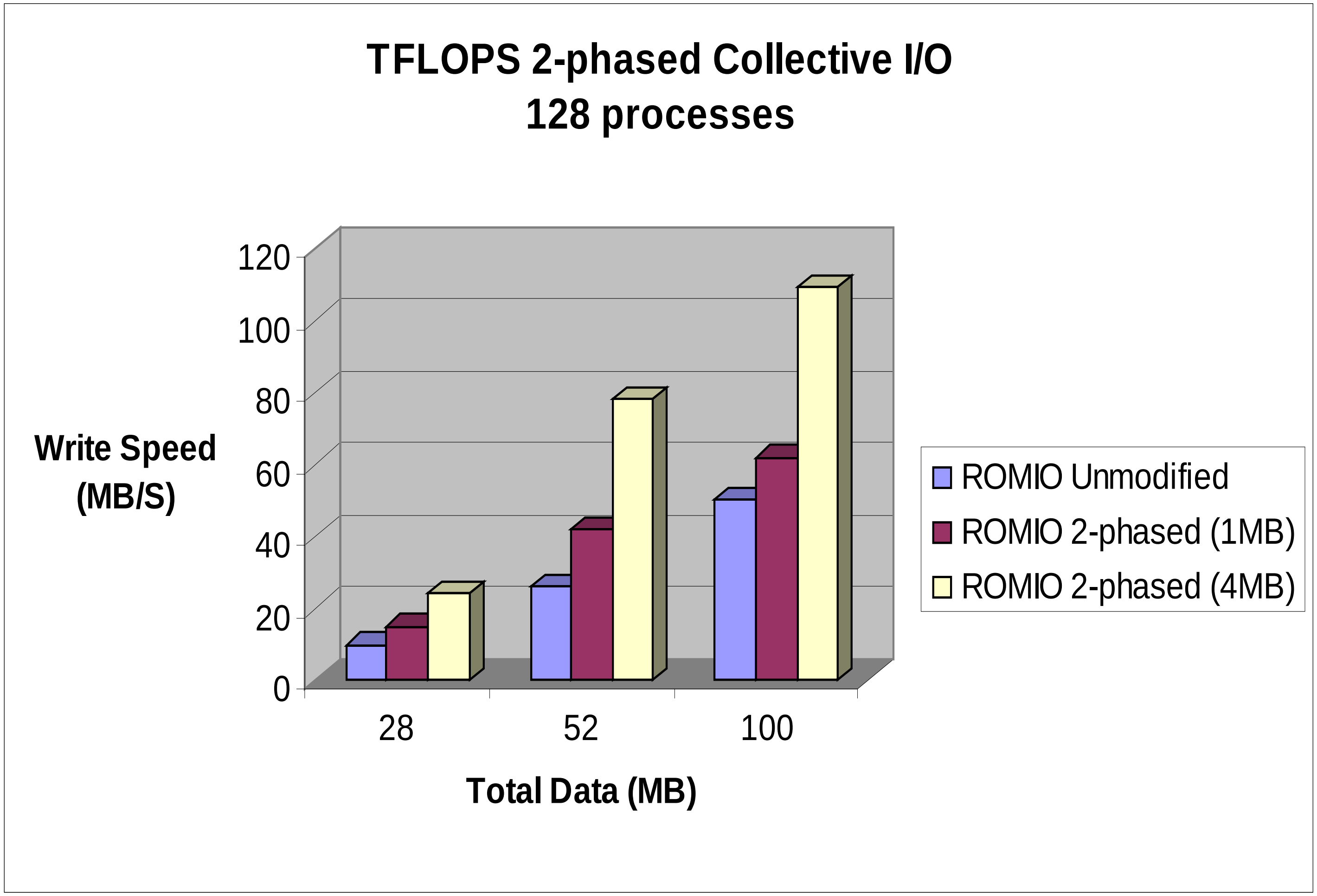
SNL Tflops
PHDF5 Collective I/O

- Romio, as is, does not do 2-phased collective I/O, even when requested, if data are not interleaved
- Modified Romio to do 2-phased I/O if requested
- Test
  - 128 processes
  - Used 1 & 4 MB collection buffer sizes

PHDF5 2-Phased
Collective I/O Numbers

PHDF5, ROMIO, 2-phase collective I/0 performance numbers			
Romio (1.2.2.1) VS modified Romio to force 2-phas collective I/O			
128 Processes			
Total Data (MB)	ROMIO Unmodified	ROMIO 2-phased (1MB)	ROMIO 2-phased (4MB)
MB	MB/S	MB/S	MB/S
28	10	15	24
52	26	42	78
100	50	62	109

2-phased Collective I/O Chart



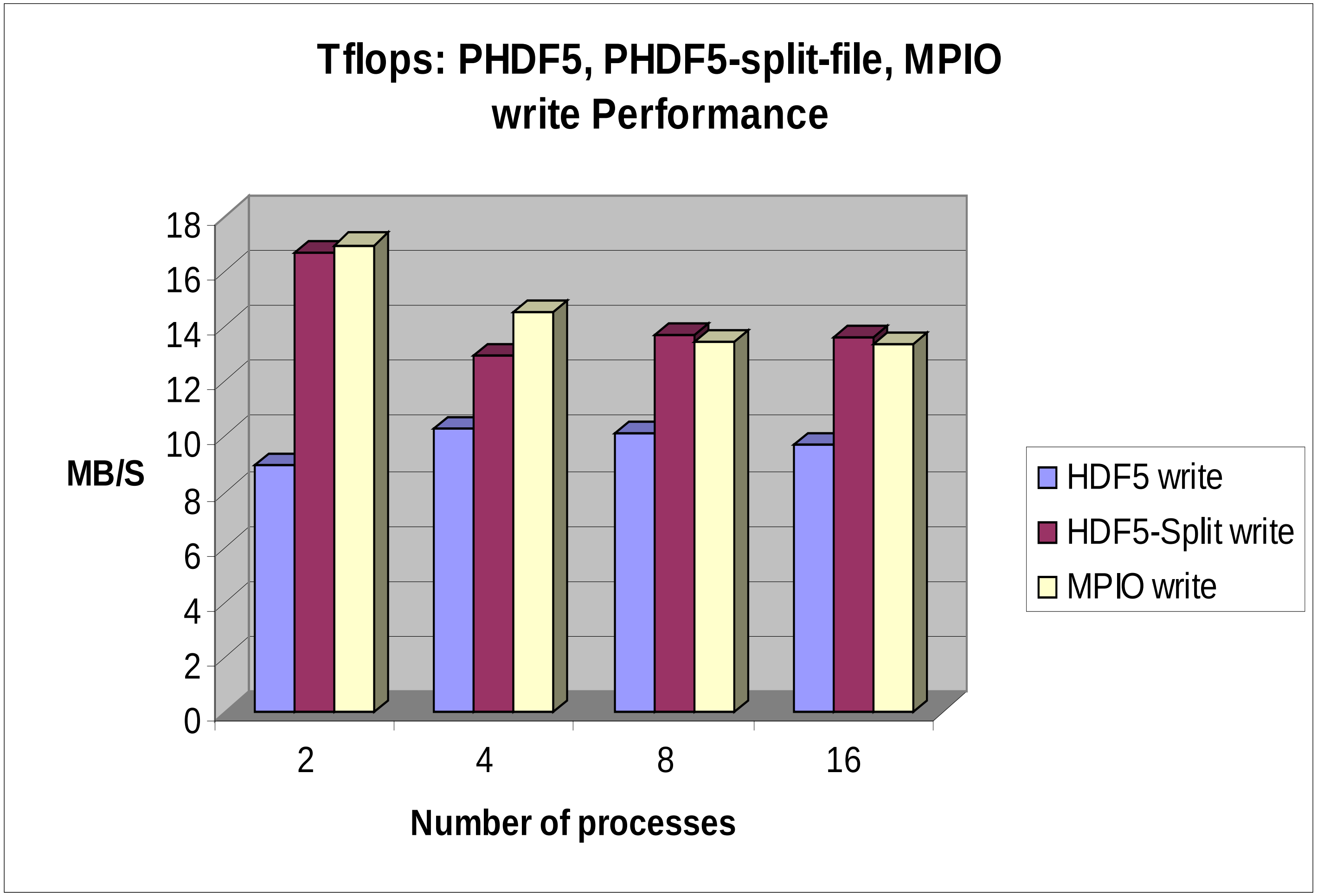
2-phased Collective I/O Remarks

- Improvement for collective I/O even for just 1MB collection buffer
- Can be invokde by the MPI-INFO object parameter when setting up the MPIO-access for H5Fopen.

Split-file for Tflops Write Performance

- **Each process writes 10 blocks, each is 1MB big, in round robin**
- **Number of processes: 2, 4, 8, 16**
- **Write via**
  - **MPI-IO**
  - **PHDF5 to one PFS file**
  - **PHDF5 split meta-file to UFS and raw-file to PFS files**

Tflops: HDF5 Split-file Improvement



Split File for Tflops Remarks

- Split-file big improvement
  - Match up with MPIO speed
  - Could it be alignment?

SAF Performance

Larry Schoof

Sandia National Laboratories

Outline

- Current applications of parallel I/O
- Parallel I/O issues
- Performance considerations
- End-to-end parallel SAF benchmark

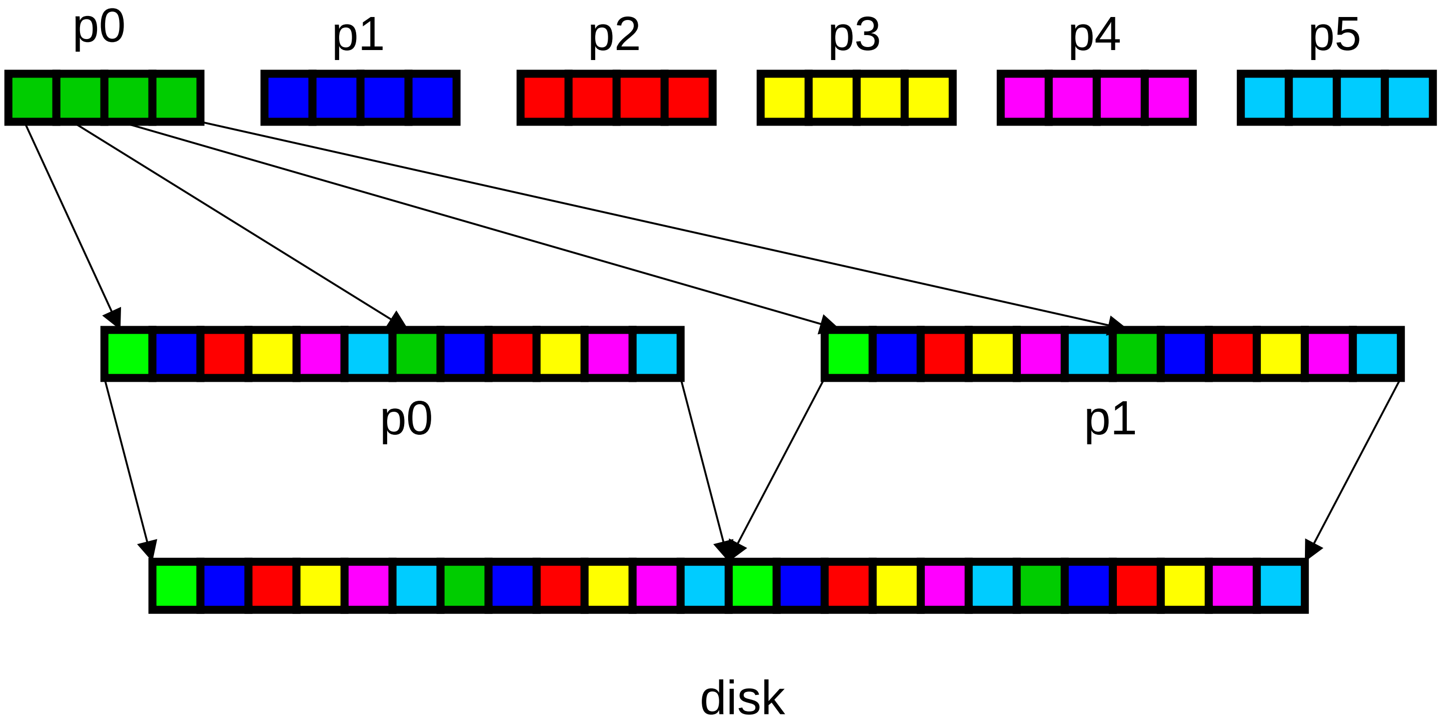
Current Applications

- Most current applications use EPIO
  - file per processor
  - read/write access pattern results in small I/O requests
- Naïve parallel implementation (1 file / processor)
  - 10M element 3d mesh
  - 10 fields
  - 1000 time steps; flush every time step
  - 1000 processors
  - aggregate dataset size = 400 GB
  - BUT…
  - individual file size -- 400 GB / 1000 processors = 400 MB
  - I/O request size -- 400 MB / 1000 time steps = 400 KB I/O requests!
  - many I/O requests (e.g., metadata) are ~10 KB
- Many HPC I/O subsystems peak at >> 1 MB requests

Parallel I/O Issues

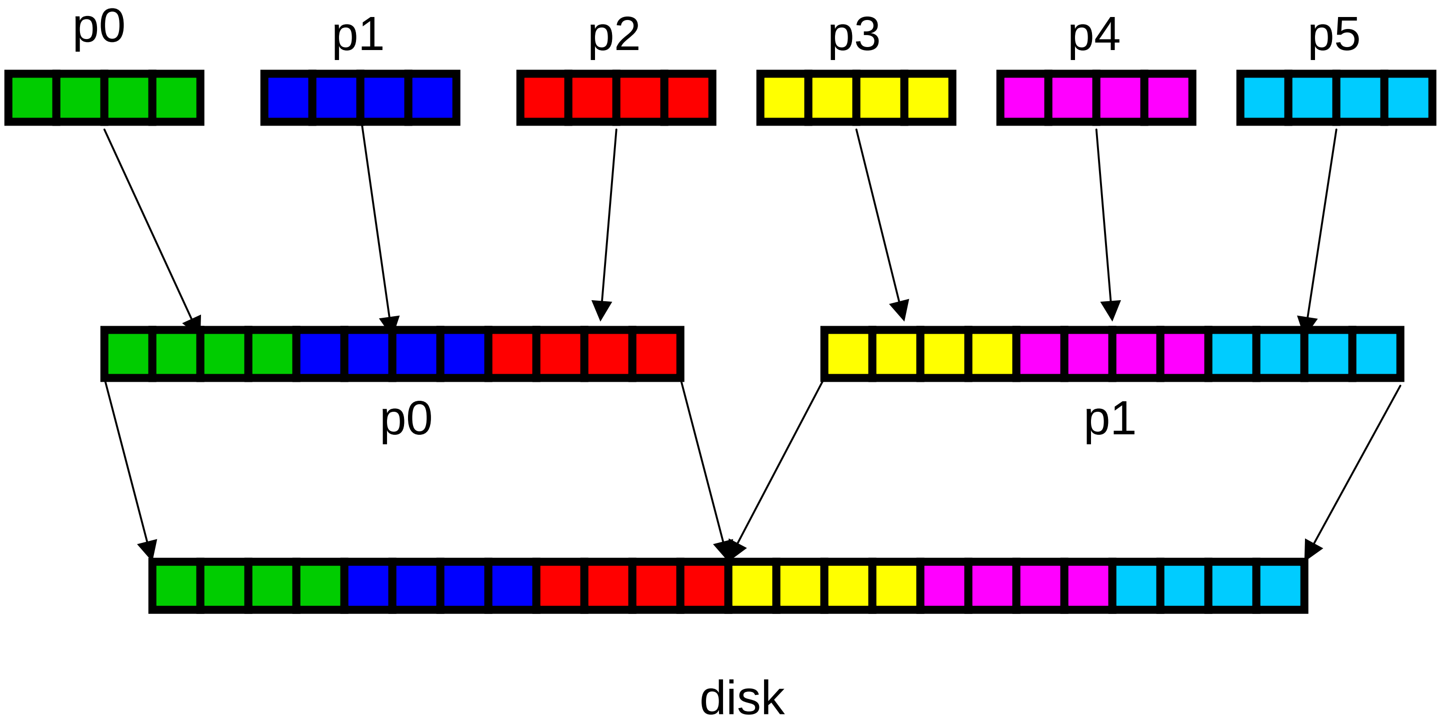
- Distinguish parallel I/O from parallel data constructs
  - parallel I/O
    - EPIO vs.. collective parallel I/O
    - distributed or global parallel file system
  - parallel data constructs
    - local vs. global information
    - local/global mapping (node- or element-based decomposition)
- Separate raw data from metadata in function calls
- Consider file system characteristics
  - UFS vs. PFS (large block read/write vs. small data access)
- Allow flexibility in file specification
  - what data goes to which file
  - separated by time slice, processor, mesh part, etc.
- Parallel I/O *and* parallel data constructs must be implicit part of data model, not appendages
- Aggregate data into large buffers

2-Phase I/O



- Interleaving (ROMIO does this)

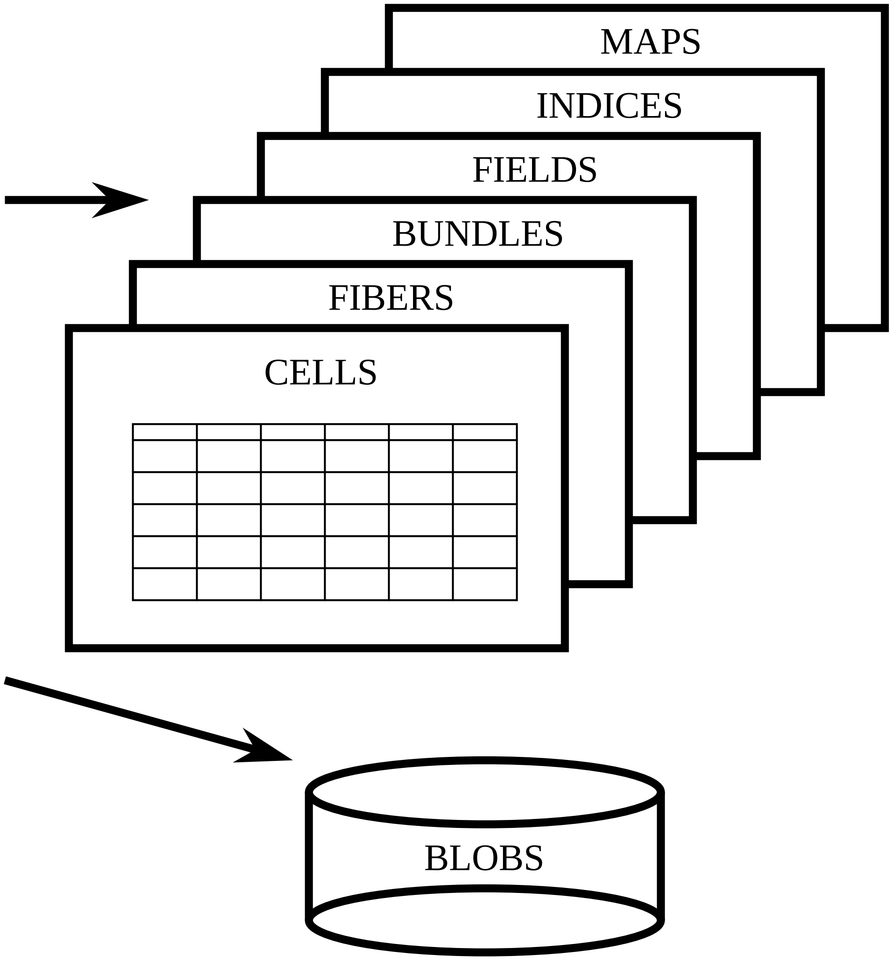
2-Phase I/O



- Aggregation (ROMIO doesn’t do this!); useful for
  - filling I/O buffers
  - moving data to processors that have better connectivity

SAF Performance Considerations

- **Light data (metadata)**
  - memory resident
  - data is local (private) or global (shared) across processors
  - VBT manipulates
- **Heavy data (raw data)**
  - file resident
  - no transformations unless requested
  - passed through to HDF



SAF Performance Considerations (cont.)

- **Allow choice of when to perform transformations**
  - local/global remapping on write (during simulation) or on read (during visualization)
- **Minimize transformations; transform data only when client requests it**
  - "hub and spoke" paradigm is not optimal
    - units, binary data representation (e.g., XDR) primitive node-ordering, etc.
  - requires description of data (via metadata)
- **Multi-layer approach; what is the function of each layer; sometimes there are decisions**
  - local/global remapping -- MPI-IO has this functionality (MPI_Type_indexed / MPI_File_set_view), but we chose to do it in SAF

End-to-end Parallel SAF Client

- **Purpose**
  - create a parallel client to test performance of SAF implementation; test all layers
  - simulate the I/O of parallel analysis process (mesh generation, domain decomposition, physics code, visualization)
  - create, write, read arbitrarily large sets of SAF data

End-to-end Parallel SAF Client

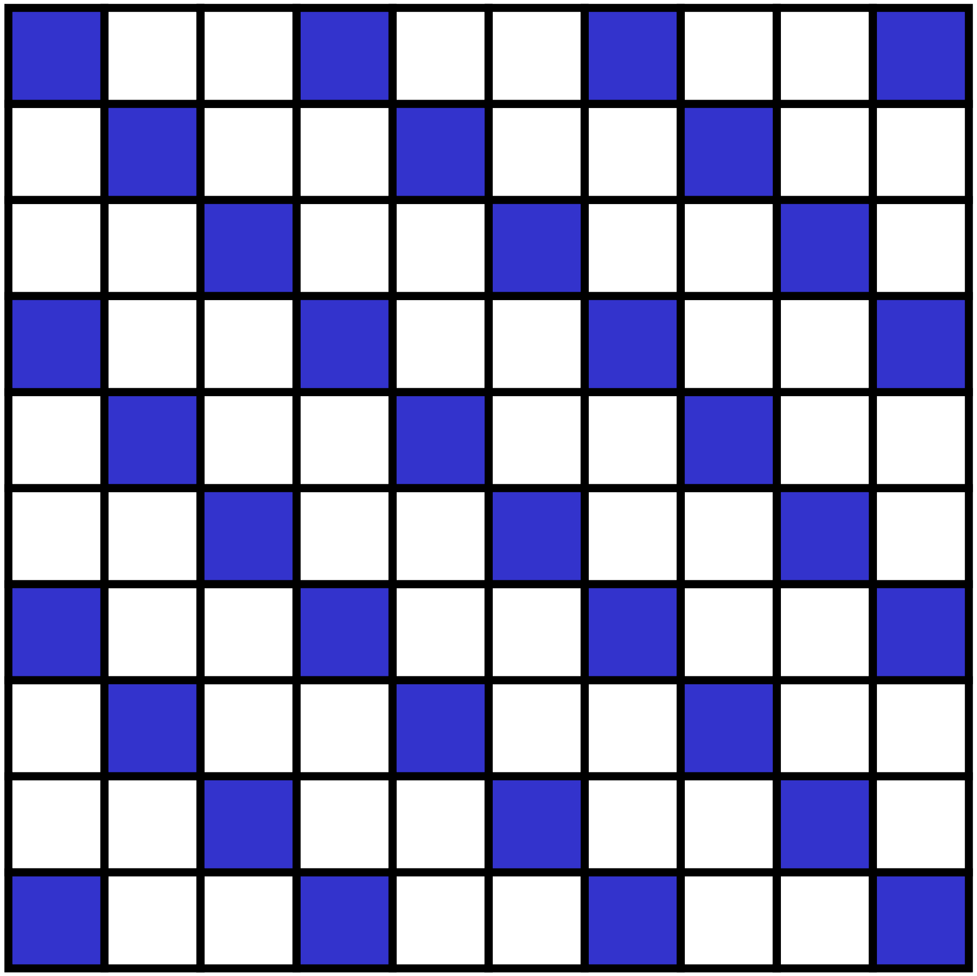
- **Description**
  - **create mesh (serial)**
  - **decompose mesh (serial)**
  - **write mesh and decomposition (serial)**
  - **read mesh (parallel)**
  - **write mesh (parallel)**

create + write

read + write

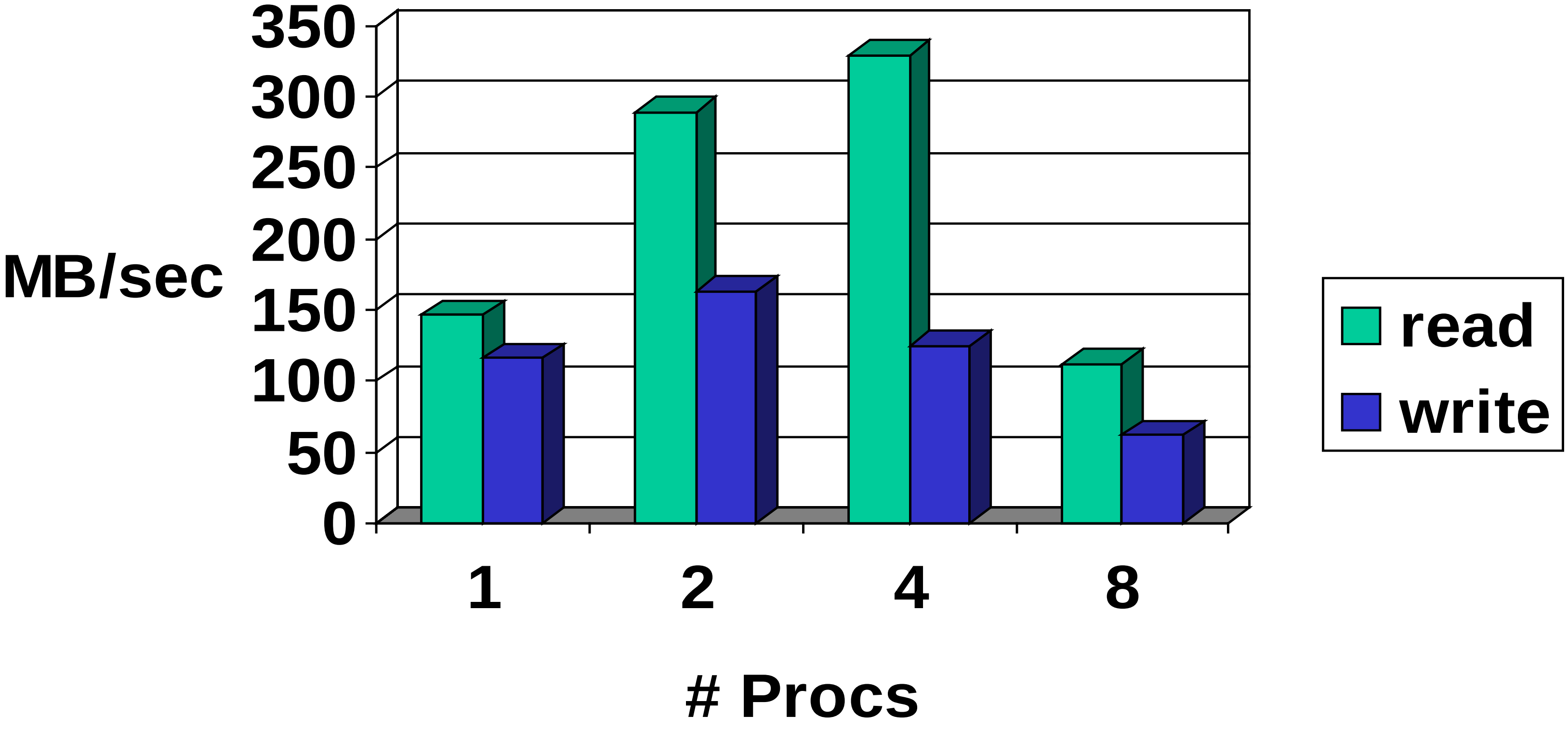
- **Parameters**
  - **number of (processor) domains**
  - **size of mesh**
  - **number of fields**
  - **file mode (use of master and supplemental files)**

Processor-local mesh



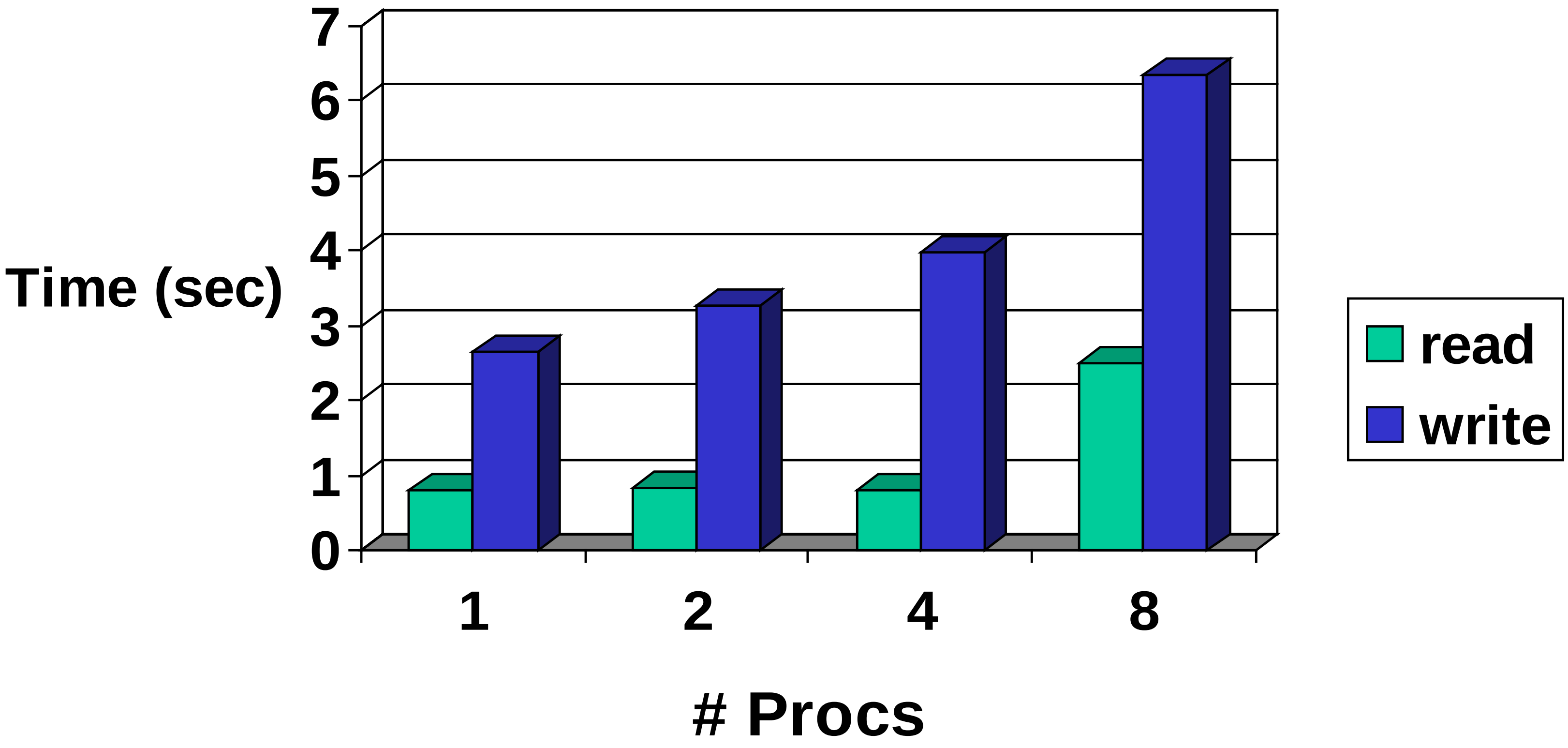
SAF Benchmark Results (preliminary)

SGI O2K 16 proc
1M elem / 10 fields



SAF Benchmark Results (preliminary)

SGI O2K 16 proc
1M elem/10 fields



Performance Summary

- Provide many “knobs” to turn
  - what transforms to perform
  - when to perform them
  - flexibility in file specification (what data to which file)
  - aggregation options (e.g., 2-phase I/O)
- Don’t use “hub and spoke” paradigm
- Parallel data constructs must be implicit part of data model
- Separate “light data” from “raw data”
- Most parallel I/O implementations in current applications are naïve