The HDF Group

HDF5 Application Tuning

Part 1: There is more than one way to skin a cat(fish)

Gerd Heber

Setting

- Practical advice for beginners
- Work with a “simple” problem for clarity
- Show three tools that will be sufficient most of the time
- A lot of material
  - Go w/ the flow and ignore the parts that seem irrelevant / unclear
  - Focus on possibilities and remember Murphy’s law
  - Try it with your code!
- Part 1 (in a sandbox), Part 2 (on a “real” cluster)
- It’s all about ***method*** (and resources)

A Simple Problem

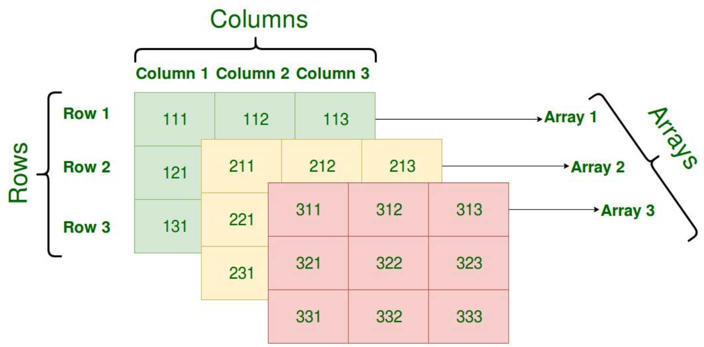
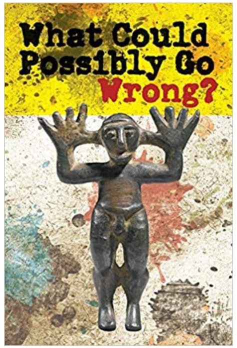


Figure: GeeksForGeeks

Writing multiple 2D array variables over time:

```
ACROSS P processes arranged in a R x C process grid
    FOREACH step 1 .. S
        FOREACH count 1 .. A
            CREATE a double ARRAY of size [X,Y] | [R*X,C*Y] (strong | weak)
            (WRITE | READ) the ARRAY (to | from) an HDF5 file
        END
    END
END
```

S(teps) = 20, A(rrays) = 500, X = 100, Y = 200 (See adios_iotest)



Missing Information

- How are the array variables represented in HDF5?
  - 2D, 3D, 4D datasets
  - Are the extents known a priori?
  - How are the dimensions ordered?
  - Groups?
- How (order) is the data written and is the data read the same way?
- What’s that storage layout?
  - How many physical files?
  - Contiguous or chunked, etc.
  - Is the data compressible?
- What’s the file system or data store?
- Collective vs. independent MPI-IO
- ...

The “Raw HDF5” Dilemma

What many users want

`#include “hdf5.h”`

Turtles all the way down

Basic Combinations (24)

- Six griddings
  - `/step=[0..10]/array=[0..499]` `Dataset {100, 200}`
  - `/array=[0..499]/step=[0..19]` `Dataset {100, 200}`
  - `/step=[0..20]` `Dataset {500, 100, 200}`
  - `/array=[0..499]` `Dataset {20, 100, 200}`
  - `/dataset` `Dataset {20, 500, 100, 200}`
  - `/dataset` `Dataset {500, 20, 100, 200}`
- Two layouts
  - Contiguous or chunked
- Two MPI modes
  - Collective or independent
- ...

Environment

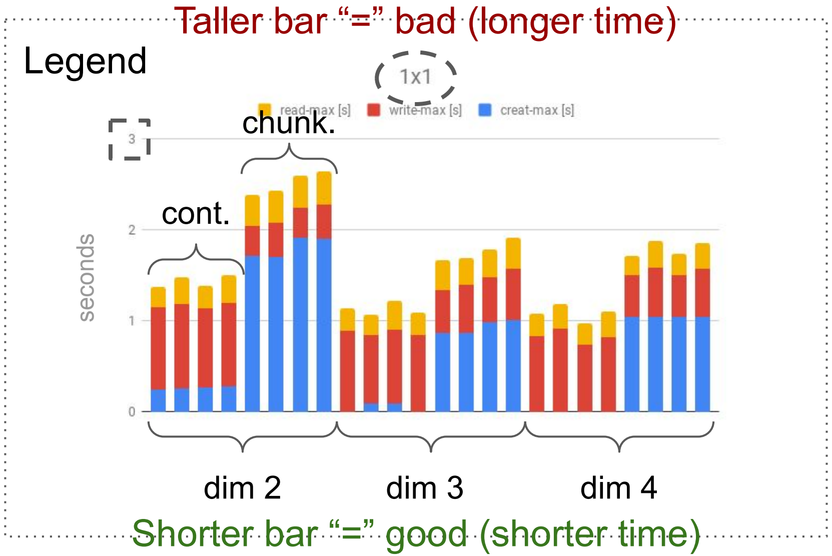
- Google Cloud (other fine choices are available: Amazon, Microsoft, Oracle, ...)
- `n2-standard-8` instance (8 vCPU, 32 GB RAM)
- Ubuntu 20.04 LTS
- Two local NVMe attached SSDs, single 750 GB MD RAID 0 volume
  - Write BW w/ 1 MB blocksize: 819 MB/s
  - Random read BW w/ 4 KB blocksize: 1,474 MB/s
  - Random write BW w/ 4 KB blocksize: 664 MB/s
- See the scripts in the `gcloud` folder

Code overview

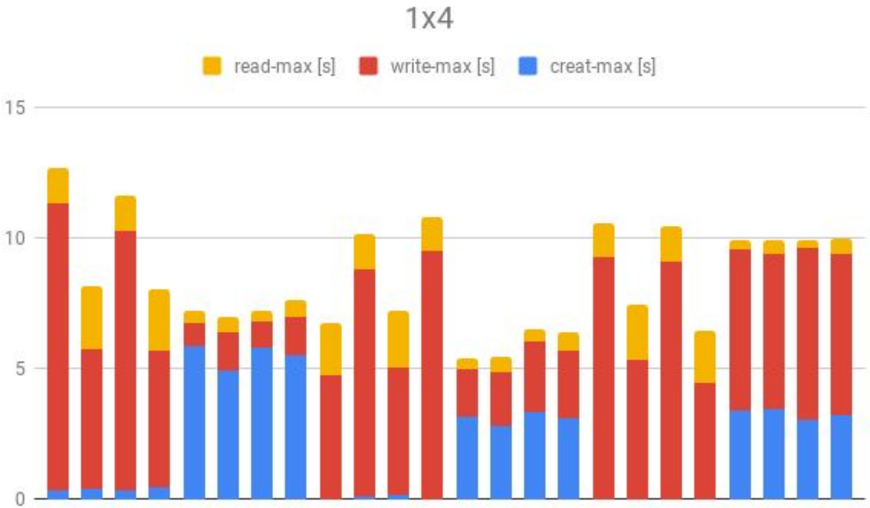
- Code can be found on GitHub
- Basic structure
    - Read and parse configuration from INI file
    - Create HDF5 file
    - Write phase
    - Close HDF5 file
    - Open HDF5 file
    - Read phase
    - Close HDF5 file
    - Write CSV output file w/ timings

Baseline

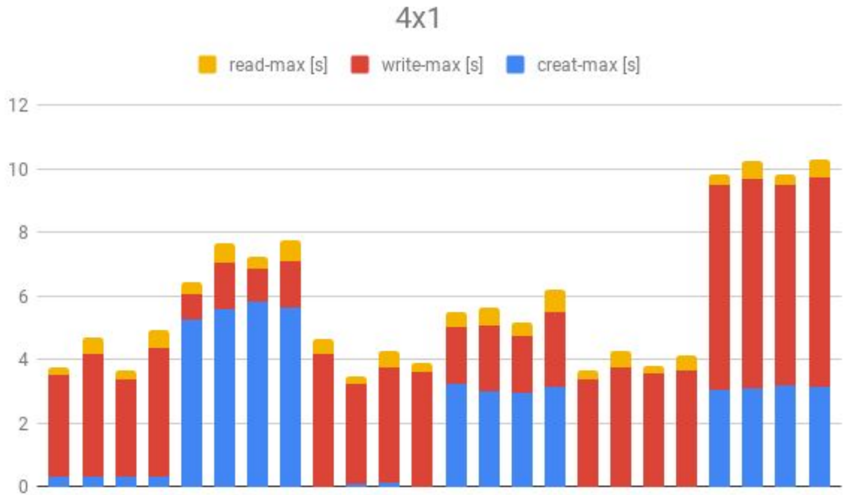
- Run **24 parameter configurations**
- Weak scaling
  - Each process writes 500*100*200*8 (~ 80 MB) per step (20 steps)
- Single processor, 4 processor grids: 1 x 4, 2 x 2, 4 x 1
- Measure times for dataset creation, write, and read



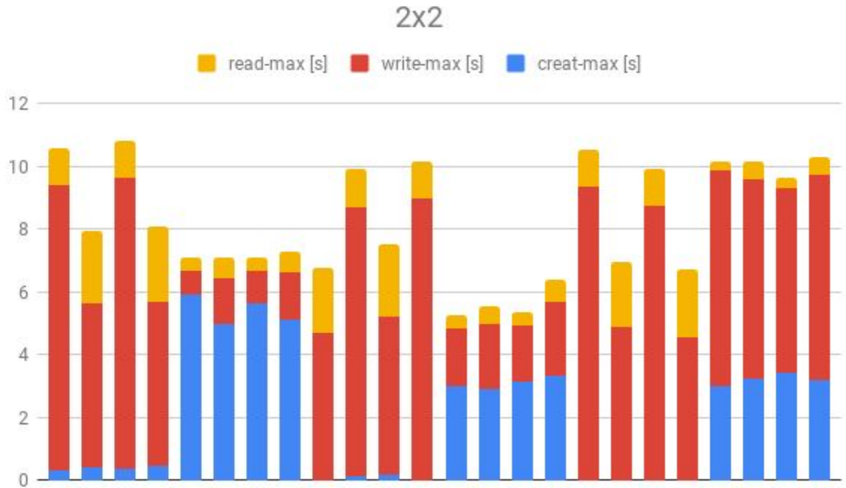
Taller bar "=" bad (longer time)
Legend
1x1
read-max [s]
write-max [s]
creat-max [s]
seconds
cont.
chunk.
dim 2
dim 3
dim 4
Shorter bar "=" good (shorter time)



1x4
read-max [s]
write-max [s]
creat-max [s]



4x1
read-max [s]
write-max [s]
creat-max [s]



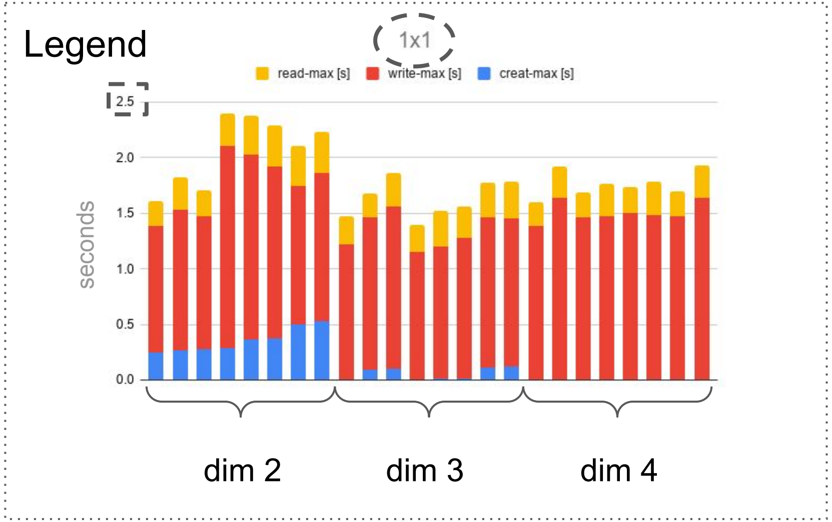
2x2
read-max [s]
write-max [s]
creat-max [s]

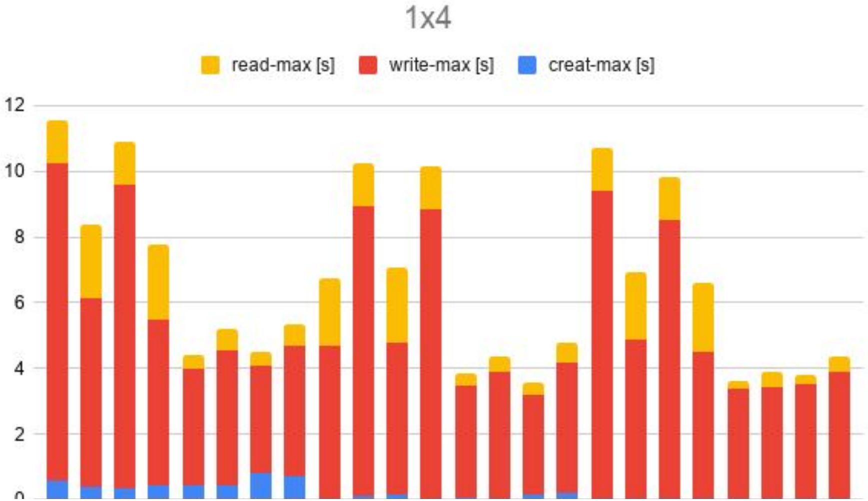
Observations

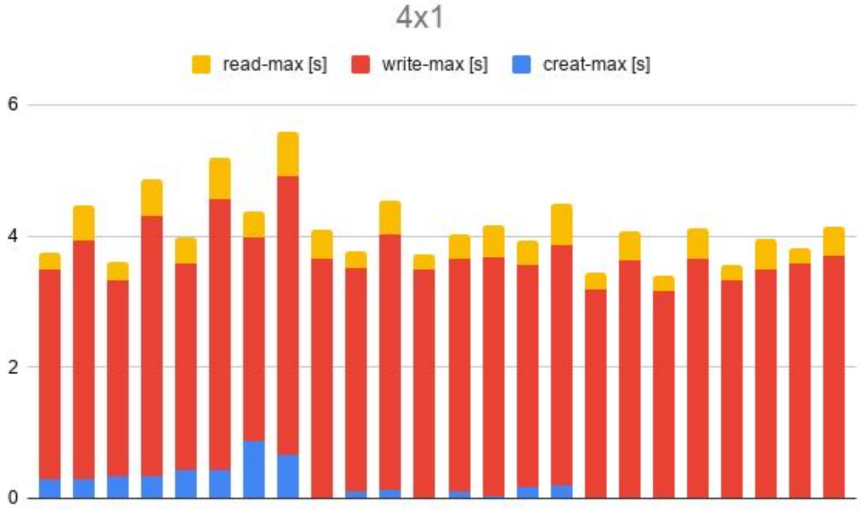
- 3-4x variability in performance
- Dataset creation overhead when using chunking
  - Most pronounced for two-dimensional datasets
    - Even for contiguous layout
- Process topology matters (?)
- There are limits to what we can get out of “user instrumentation”

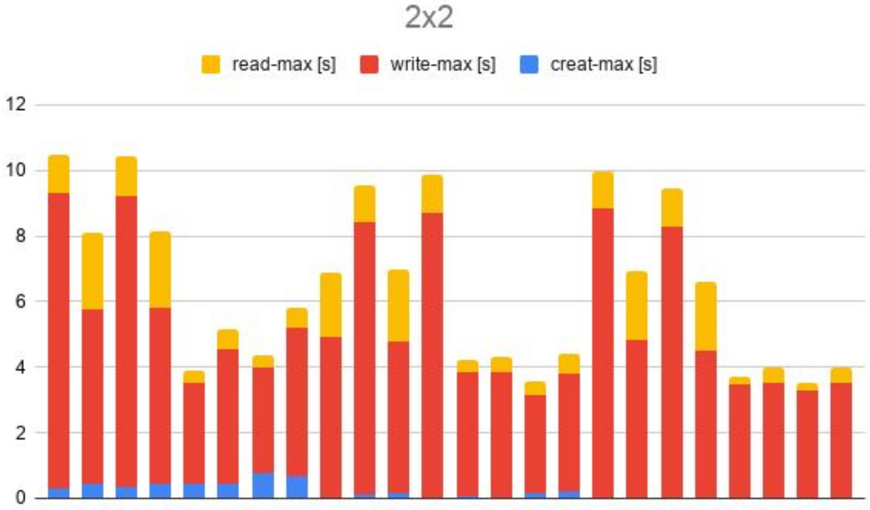
Reducing the “Dataset Creation Overhead”

- “Low-hanging fruit”
- HDF5 dataset creation
  - Storage allocation
    - When
  - Dataset initialization
    - Y/N, when, what
  - (Metadata management)
- Library defaults (may not be what you expect)
- Functions `H5Pset_alloc_time`, `H5Pset_fill_time`, `H5Pset_fill_value`
- In our example, there’s no need for initialization
  - Let’s try `H5Pset_fill_time(dcpl, H5D_FILL_TIME_NEVER)` !


Legend
1x1
read-max [s]
write-max [s]
creat-max [s]
seconds
2.5
2.0
1.5
1.0
0.5
0.0
dim 2
dim 3
dim 4


1x4
read-max [s]
write-max [s]
creat-max [s]
12
10
8
6
4
2
0


4x1
read-max [s]
write-max [s]
creat-max [s]
6
4
2
0

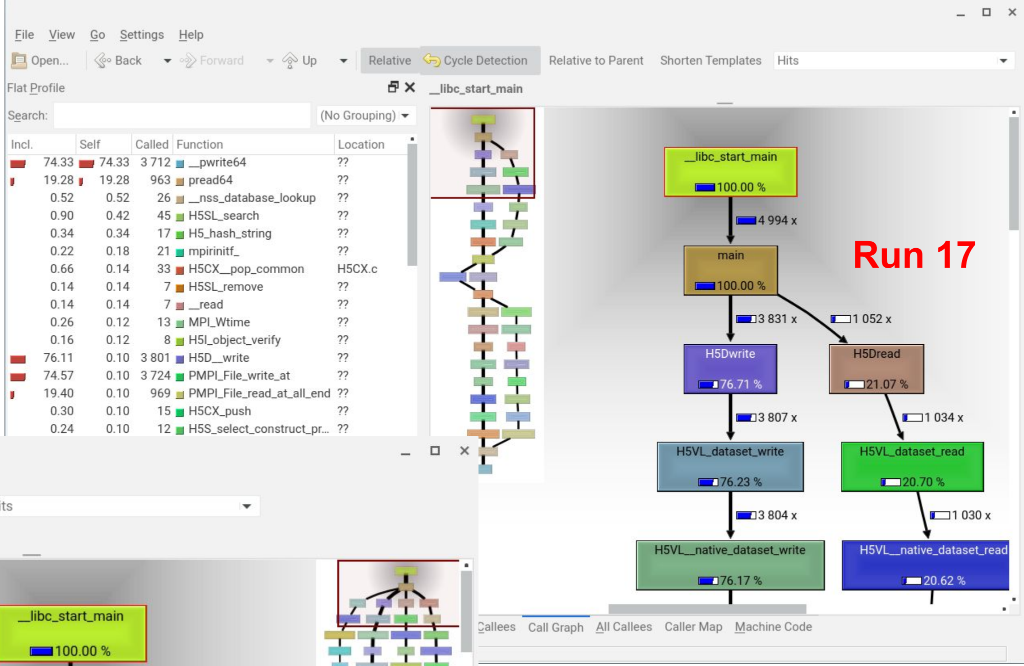

2x2
read-max [s]
write-max [s]
creat-max [s]
12
10
8
6
4
2
0

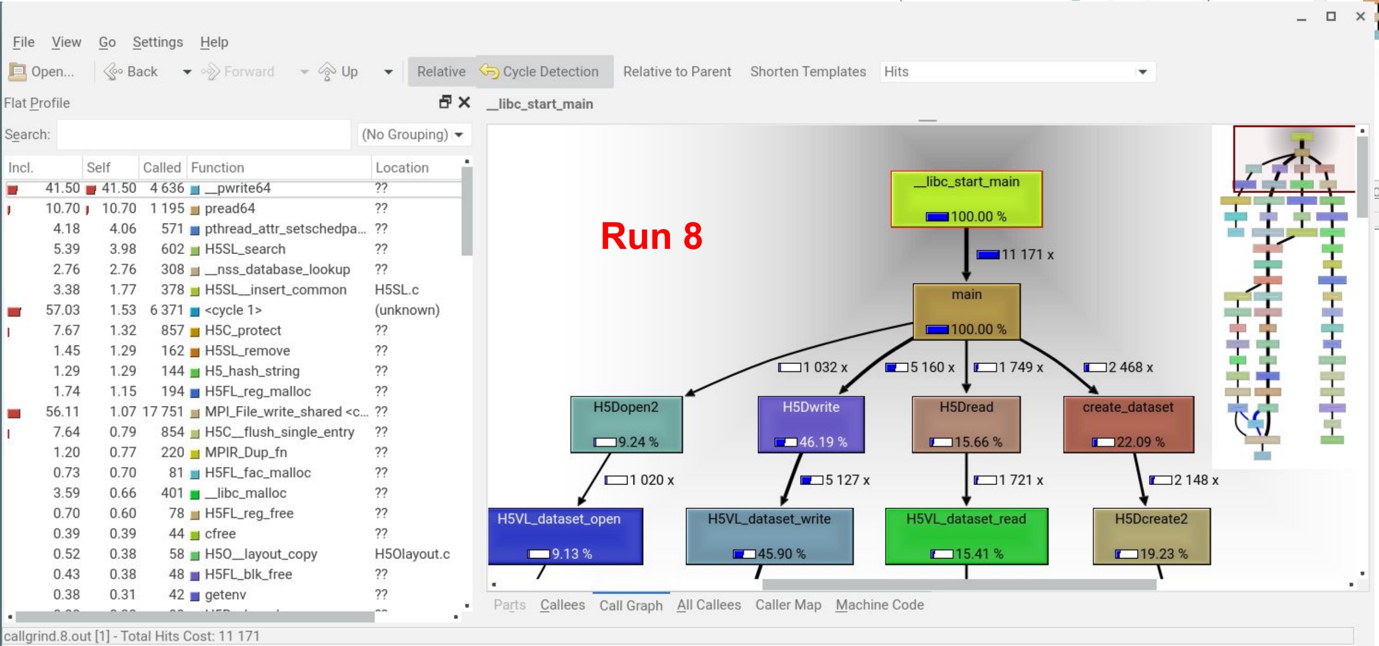
Next steps

- Relatively low variability w/ 4 x 1 process grid
- There’s not much else these charts can tell us 😟
- ⇒ Bring in tools
  - Gperftools
  - Darshan
  - Recorder
  - (TAU)
- Part 2
  - Run w/ a parallel file system!
  - Explore strong scaling!

Gperftools

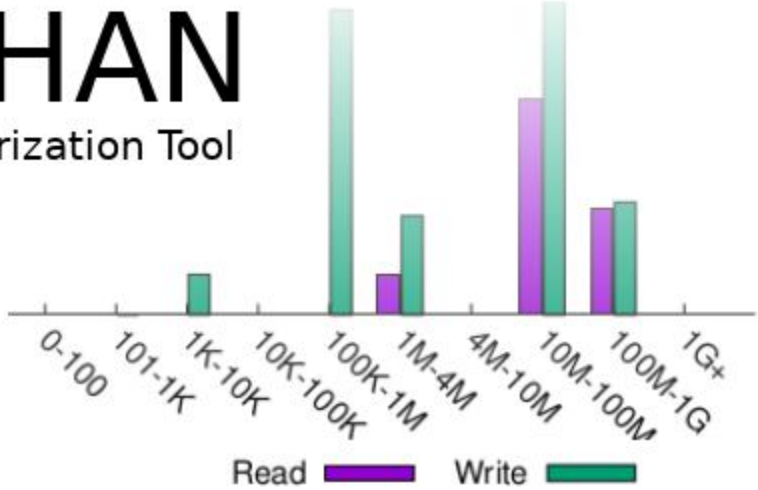
- Source on [GitHub](#)
- Use w/ [KCachegrind](#)
- See [`profiler.sh`](#)





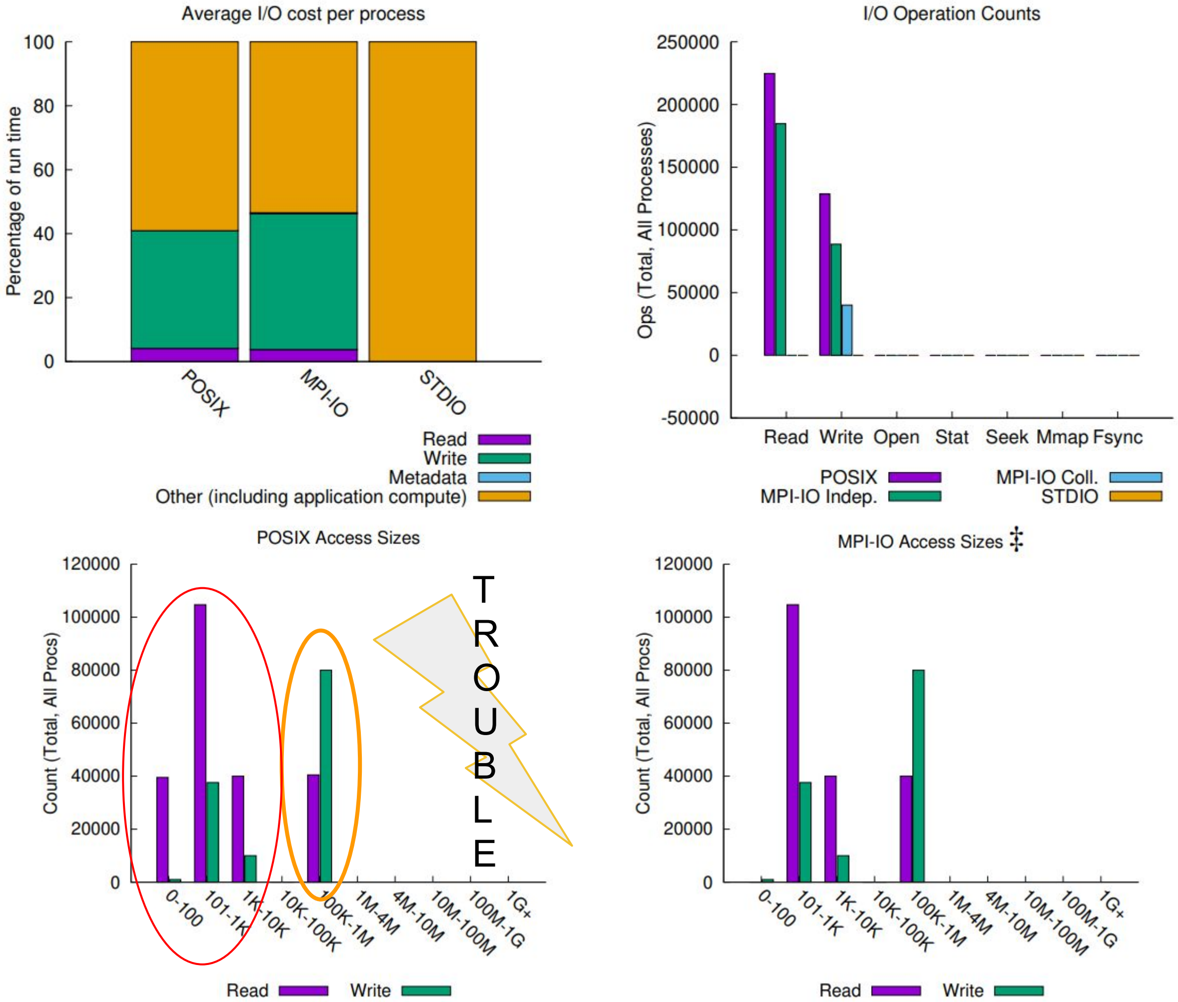
Darshan

DARSHAN
HPC I/O Characterization Tool



- Source on [GitLab]
  - Use the latest version (3.2.1) or build from source
- Runtime + utilities
- Performance counters
  - Grouped by modules (POSIX, MPI-IO, HDF5, etc.)
  - Record IDs for files, HDF5 datasets, etc.
  - `MPIIO_SIZE_READ_AGG_1K_10K`, `POSIX_RW_SWITCHES`, `H5D_REGULAR_HYPERSLAB_SELECTS`, etc.
- Tools for parsing and summarization
- Customize for the relevant counters
- Don’t miss the Python module
  - See the example by Alexandar Jelenak (HDF Group)!

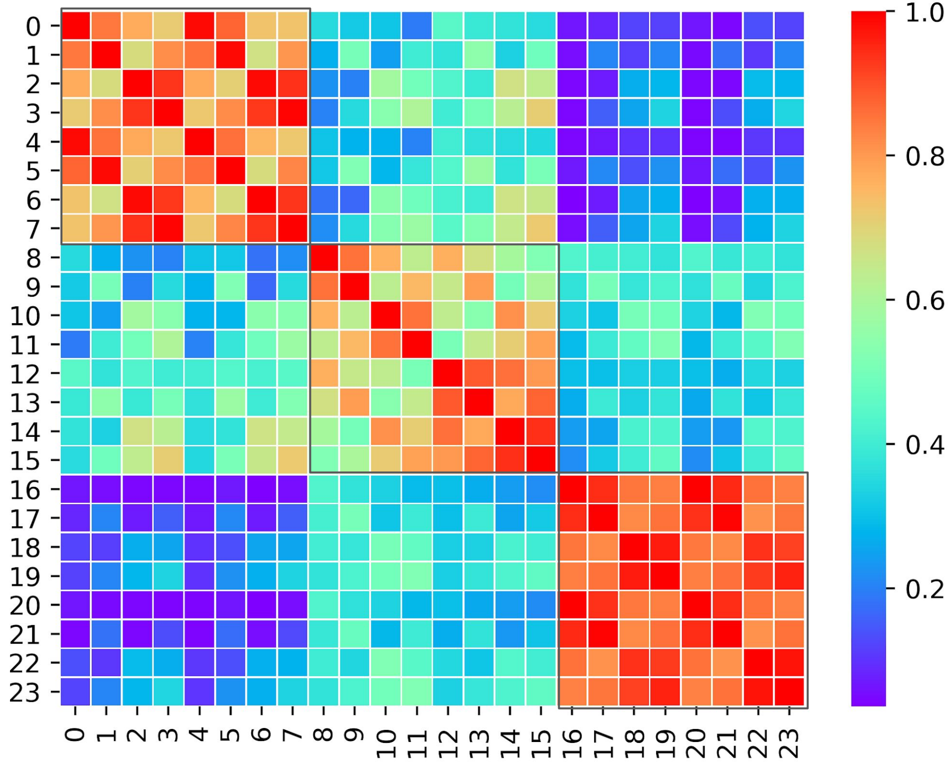
4x1 Run 7



Average I/O cost per process
Percentage of run time
100
80
60
40
20
0
POSIX
MPI-IO
STDIO
Read
Write
Metadata
Other (including application compute)
I/O Operation Counts
Ops (Total, All Processes)
250000
200000
150000
100000
50000
0
-50000
Read Write Open Stat Seek Mmap Fsync
POSIX
MPI-IO Indep.
MPI-IO Coll.
STDIO
POSIX Access Sizes
Count (Total, All Procs)
120000
100000
80000
60000
40000
20000
0
0-100 101-1K 1K-10K 10K-100K 100K-1M 1M-4M 4M-10M 10M-100M 100M-1G 1G+
Read
Write
T R O U B L E
MPI-IO Access Sizes ‡
Count (Total, All Procs)
120000
100000
80000
60000
40000
20000
0
0-100 101-1K 1K-10K 10K-100K 100K-1M 1M-4M 4M-10M 10M-100M 100M-1G 1G+
Read
Write

Application I/O Similarity

- Construct an I/O *signature* from Darshan counters
- Calculate the similarity of different runs or applications
- Research by Neeraj Rajesh (IIT)
- See his SC20 poster!
- Figure shows the baseline (dis-)similarity
- Are you thinking about redecorating your kitchen or bathroom?



Recorder

- A multi-level I/O *tracing* and trace data analysis tool
  - Multi-level: HDF5, MPI-IO, POSIX I/O
- Source on [GitHub]
- Function statistics (count, timing)
- Per rank access patterns
- Accessed offsets by rank and time
- Data hazards (RAW, WAR, WAW)
- …
- Nice HTML reports
- Examples

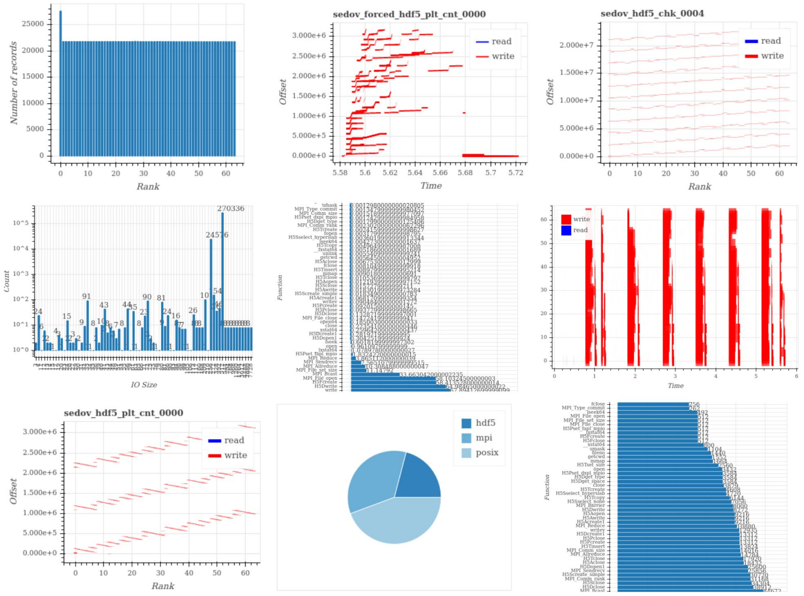
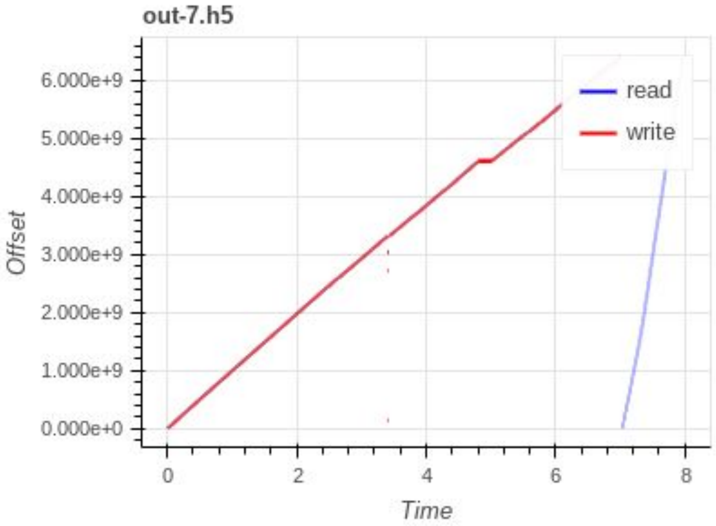
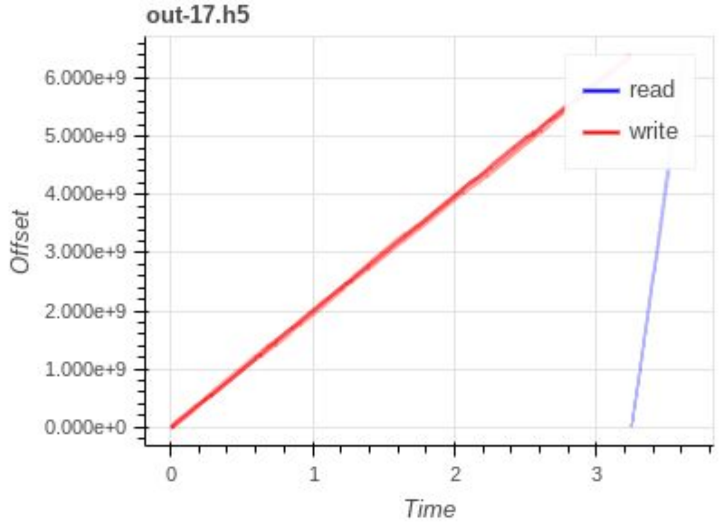
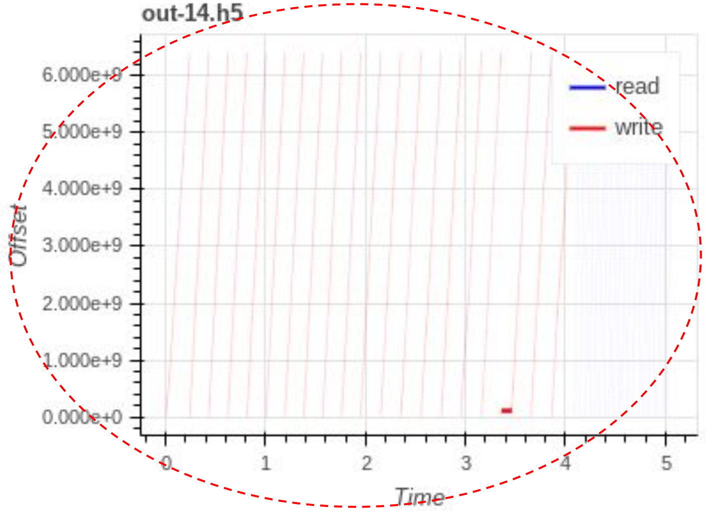
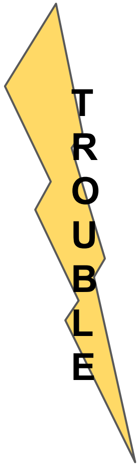


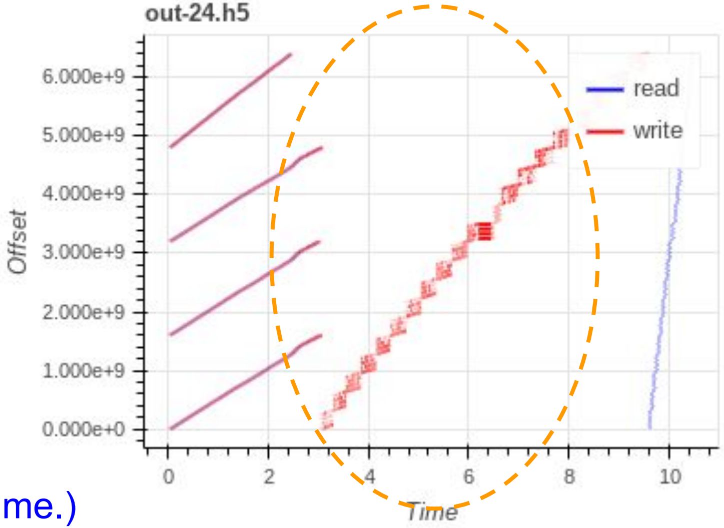
Figure by Chen Wang (UIUC)


out-7.h5
read
write
Offset
Time
6.000e+9
5.000e+9
4.000e+9
3.000e+9
2.000e+9
1.000e+9
0.000e+0
0
2
4
6
8


out-17.h5
read
write
Offset
Time
6.000e+9
5.000e+9
4.000e+9
3.000e+9
2.000e+9
1.000e+9
0.000e+0
0
1
2
3


out-14.h5
read
write
Offset
Time
6.000e+9
5.000e+9
4.000e+9
3.000e+9
2.000e+9
1.000e+9
0.000e+0
0
1
2
3
4
5


TROUBLE


out-24.h5
read
write
Offset
Time
6.000e+9
5.000e+9
4.000e+9
3.000e+9
2.000e+9
1.000e+9
0.000e+0
0
2
4
6
8
10

(Next time.)

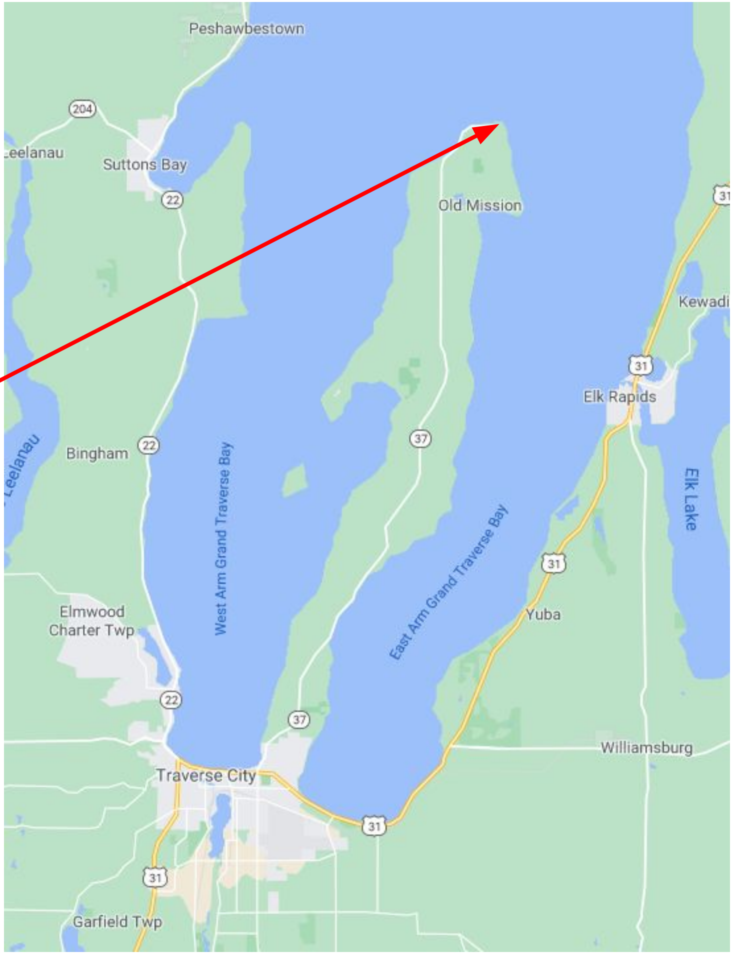
Summary

- Know what your system's capabilities are
  - It's easy to get off on the wrong foot
  - Find a *quiet corner* where you are in control and there's no queue, e.g., cloud
- Do back-of-the-envelope calculations and have expectations
  - Spot bugs/trouble
- Keep the number of turtles/variables as small as possible
- Start with single process analysis!
- Use the right tool(s) for the job
- Multiple perspectives; too much or too little information begets confusion
- Know when you've exhausted available information

Next Time

- In depth analysis of Darshan results and Recorder plots
- The next turtle/hurdle: MPI-IO and parallel file systems
- Strong scaling


M
37
END


Peshawbestown
204
Leelanau
Suttons Bay
22
Old Mission
31
Kewadi
31
Elk Rapids
Bingham
22
37
Leelanau
West Arm Grand Traverse Bay
East Arm Grand Traverse Bay
Elk Lake
31
Yuba
Elmwood
Charter Twp
22
37
Williamsburg
Traverse City
31
31
Garfield Twp