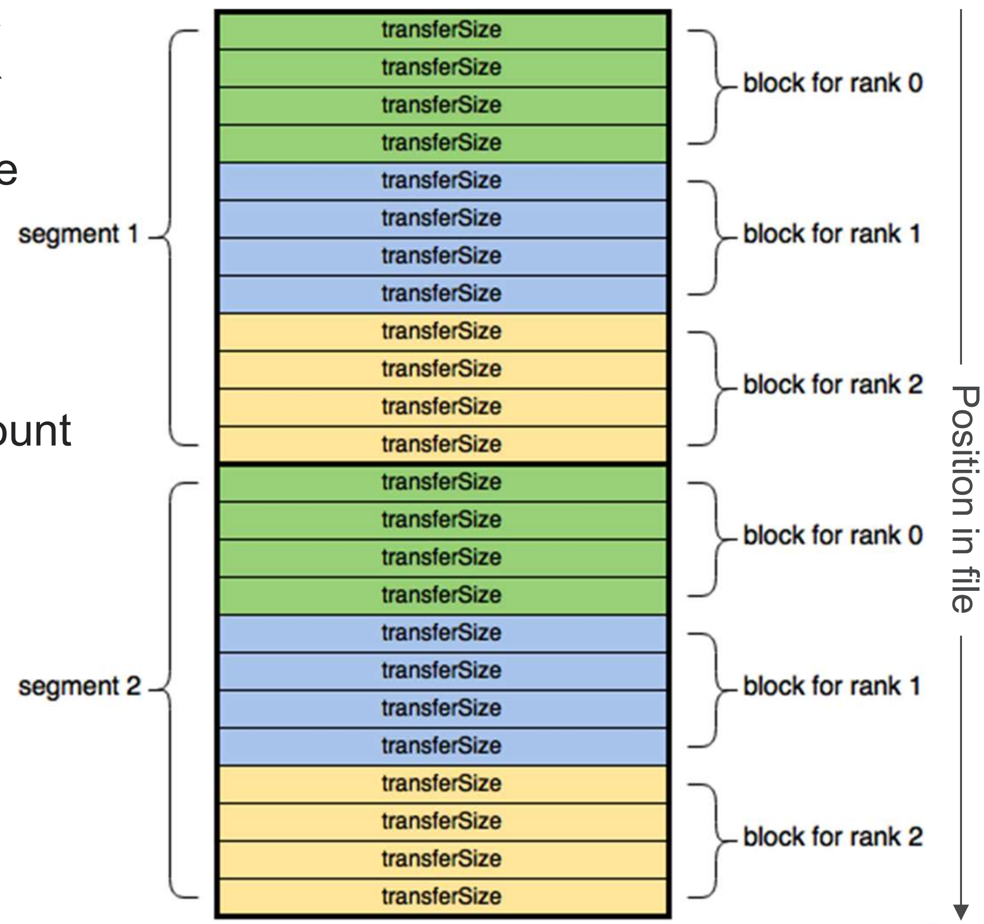
BASIC THROUGHPUT BENCHMARKING WITH IOR

Thanks to Glenn Lockwood for much of this material

THE IOR BENCHMARK

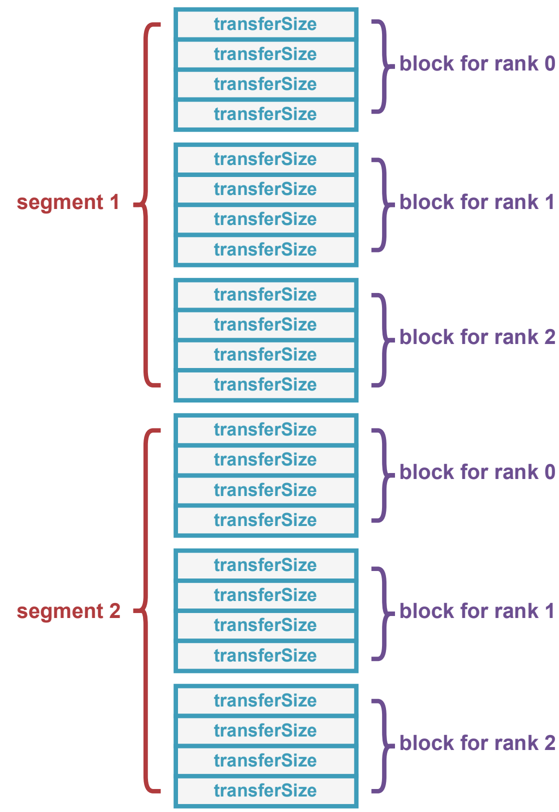
- MPI application benchmark
  – reads and writes data in configurable ways
  – I/O pattern can be interleaved or random
- Input:
  – transfer size, block size, segment count
  – interleaved or random
- Output: Bandwidth and IOPS
- Configurable backends
  – POSIX, STDIO, MPI-IO
  – HDF5, PnetCDF, S3, rados

https://github.com/hpc/ior



FIRST ATTEMPT AT BENCHMARKING AN I/O PATTERN

- 120 GB/sec Lustre file system
- 4 compute nodes, 16 ppn, 200 Gb/s NIC
- Performance makes no sense
  – write performance is awful
  – read performance is mind-blowingly good




```
$ srun -N 4 -n 64 ./ior -t 1m -b 64m -s 64
...
Operation    Max(MiB)
write         9539.38
read        492123.04
```

TRY BREAKING UP OUTPUT INTO MULTIPLE FILES

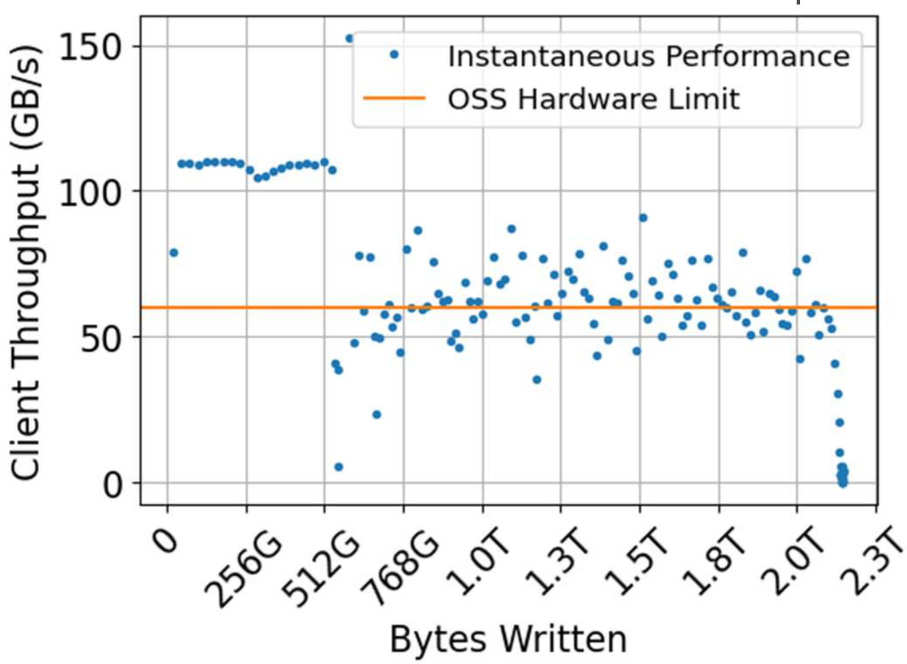
- IOR provides `-F` option to make each rank read/write to its own file instead of default single-shared-file I/O
  - Reduces lock contention within file
  - Can cause metadata load at scale
- Problem: > 400 GB/sec from 4 OSSes is faster than light

```
$ srun -N 4 -n 64 ./ior -t 1m -b 64m -s 64 -F
...
Operation   Max(MiB)
write       72852.83
read       481168.60
```

EFFECT OF PAGE CACHE ON MEASURED I/O BANDWIDTH

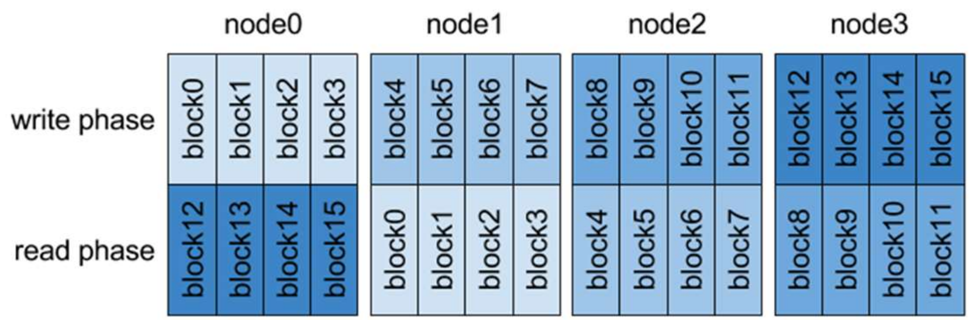
- Unused compute node memory to cache file contents
- Can dramatically affect I/O
  - <u>Writes</u>:
    - only land in local memory at first
    - reordered and sent over network later
    - `max_dirty_mb` and `max_pages_per_rpc` (Lustre)
    - `dirty_background_ratio` and `dirty_ratio` (NFS)
  - <u>Reads</u>:
    - come out of local memory if data is already there
    - read-after-write = it's already there
    - readahead also exists

Time-resolved write bandwidth
4 clients / 256 GiB DDR each
2 OSTs / 60 GB/s write spec



AVOID READING FROM CACHE WITH RANK SHIFTING

- Use `-C` to shift MPI ranks by one node before reading back
- Read performance looks reasonable
- But what about write cache?

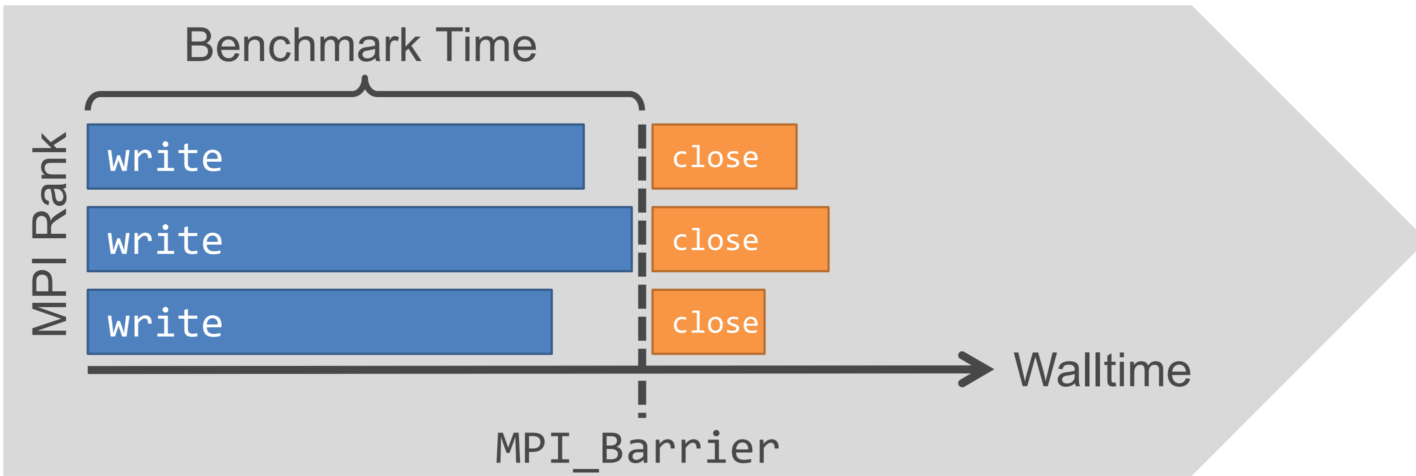



```
$ srun -N 4 -n 64 ./ior -t 1m -b 64m -s 64 -F -C
...
Operation    Max(MiB)
write        63692.33
read         28303.09
```

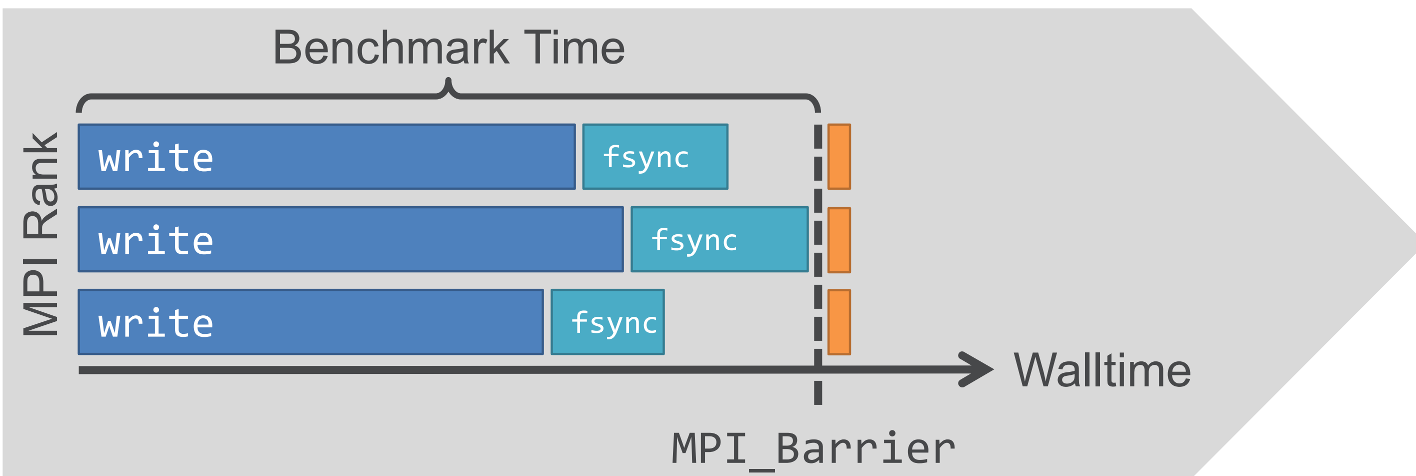
FORCE SYNC TO ACCOUNT FOR WRITE CACHE EFFECTS

- Default: benchmark timer stops when last write completes
- Desired: benchmark timer stops when all data reaches OSSes
  - Use `-e` option to force `fsync(2)` and write back all "dirty" (modified) pages
  - Measures time to write data to durable media—not just page cache
- Without fsync, `close(2)` operation may include hidden sync time

```
$ srun -N 4 -n 64 ./ior -t 1m -b 64m -s 64 -F -C -e
...
Operation   Max(MiB)
write       70121.02
read        30847.85
```

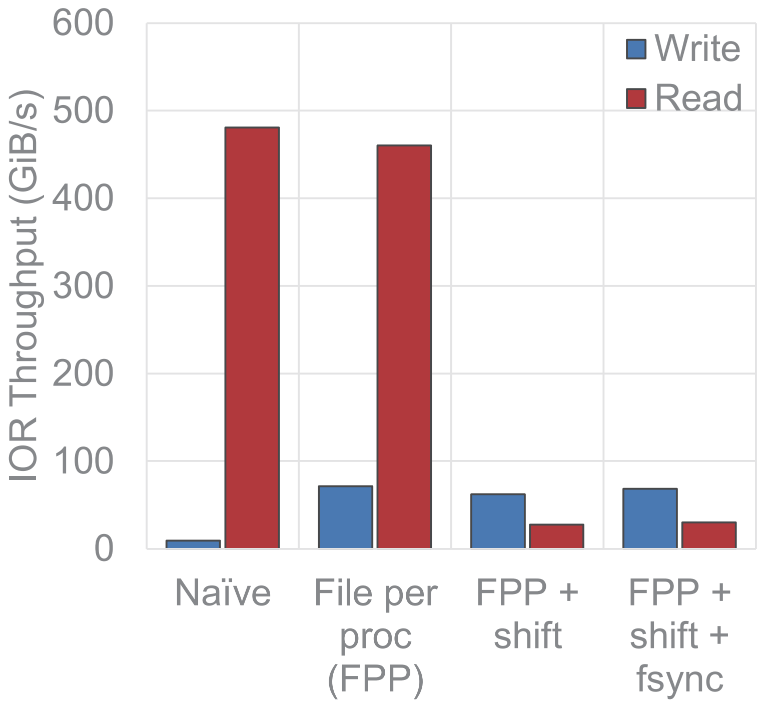


By default, benchmark may appear to hang at the end when files are being closed



With -e / fsync, time to write dirty pages to file system is included

MEASURING BANDWIDTH CAN BE COMPLICATED



- 100x difference from same file system!
  - Client caches and sync
  - File per proc vs. shared file
  - Usual Lustre stuff (e.g., striping)
- For system benchmarking, start with `-F -C -e`
- Page cache is not part of POSIX!
  - Every file system does it differently
  - Understanding its effects requires expert knowledge

IOR ACCEPTANCE TESTS
LUSTRE BANDWIDTH

Only 4 ppn needed

```
$ srun -N 960 -n 3840 ./ior -F -C -e -g -b 4m -t 4m -s 1638 -w -k
$ srun -N 960 -n 3840 ./ior -F -C -e -g -b 4m -t 4m -s 1638 -r
```

Standard args:

`-F` File-per-process
`-C` Shift ranks
`-e` include fsync(2) time

Every rank writes 4 MiB × 1,638
4 MiB at a time
Total ~25 TB

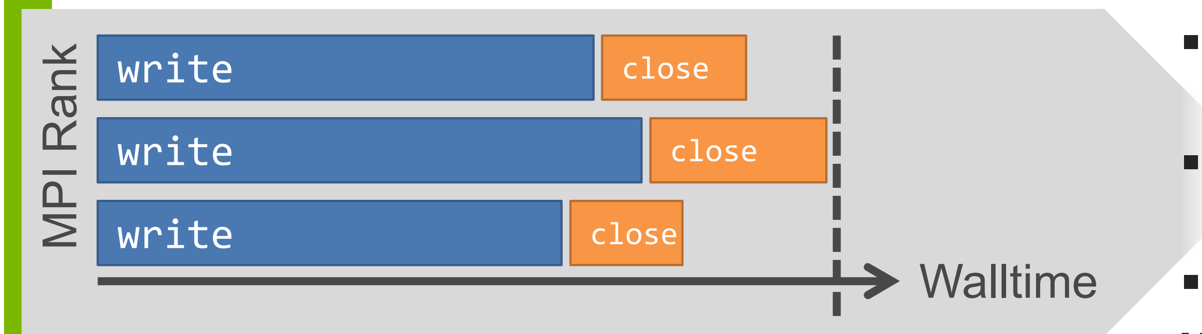
`-w` Perform write benchmark only
`-k` Don't delete written files
`-r` Perform read benchmark only

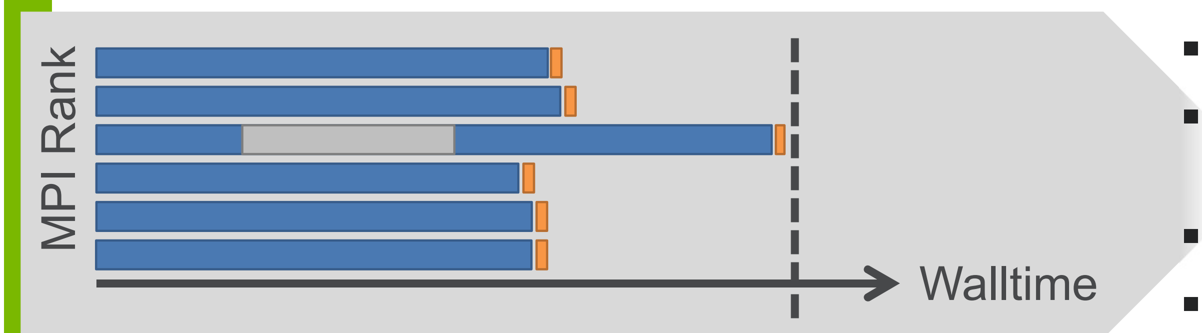
Separate sruns drop client caches

Results:

- 751,709 MB/s write (max)
- 678,256 MB/s read (max)

RUNNING AFOUL OF "WIDE" BENCHMARKING





How much `-b`/`-s`?

- More is <u>better</u>: overrun cache effects
- More is <u>worse</u>: increase likelihood of hiccup
- Preference: run for 30-60s

What is realistic for you?

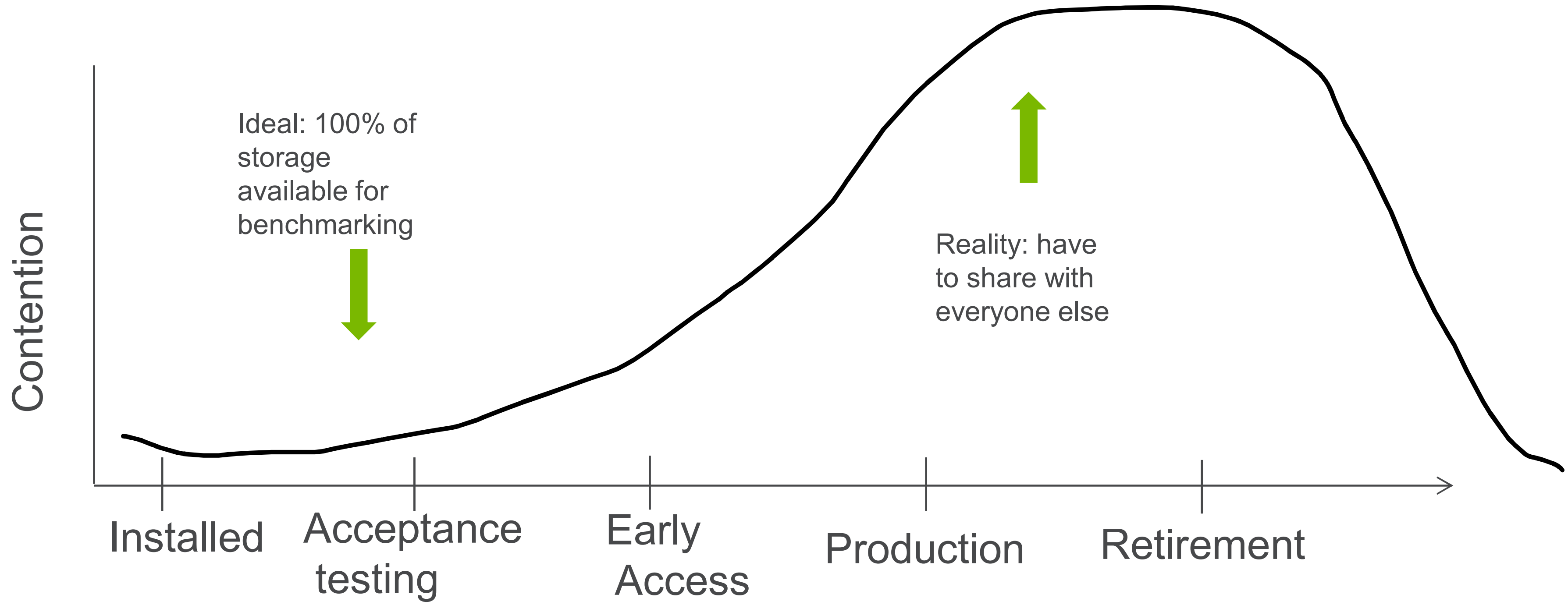
- do you want the big number?
- do you want to emulate user experience?
- small = fast
- big = realistic

EXAMPLE SYSTEM: ALCF POLARIS

- 560 nodes;
  - 2.8 GHz AMD EPYC Milan 7543P 32 core CPU
  - 512 GB of DDR4 RAM
  - 2 local 1.6TB of SSDs in RAID0
- Storage: ALCF production file systems
  - /home: Lustre (8 OST, 4 MDT)
  - /grand: Lustre: (160 OST, 40 MDT, 650 GB/sec)
  - /eagle: Lustre: (160 OST, 40 MDT, 650 GB/sec)
  - /theta-fs0: legacy Lustre
  - /theta-fs1: legacy GPFS
- Network: Slingshot 10 (soon to be slingshot 11)

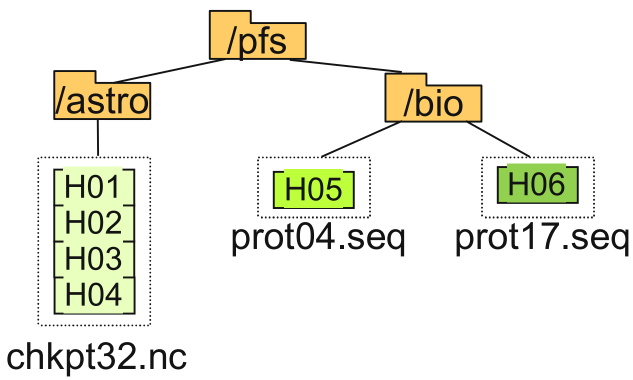
More Polaris information: https://docs.alcf.anl.gov/polaris

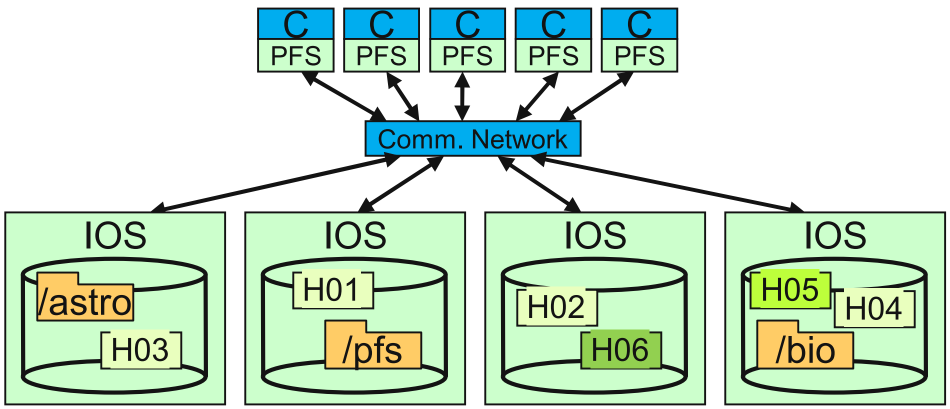
CONTENTION IN BENCHMARKIG



TUNING: STRIPE COUNT

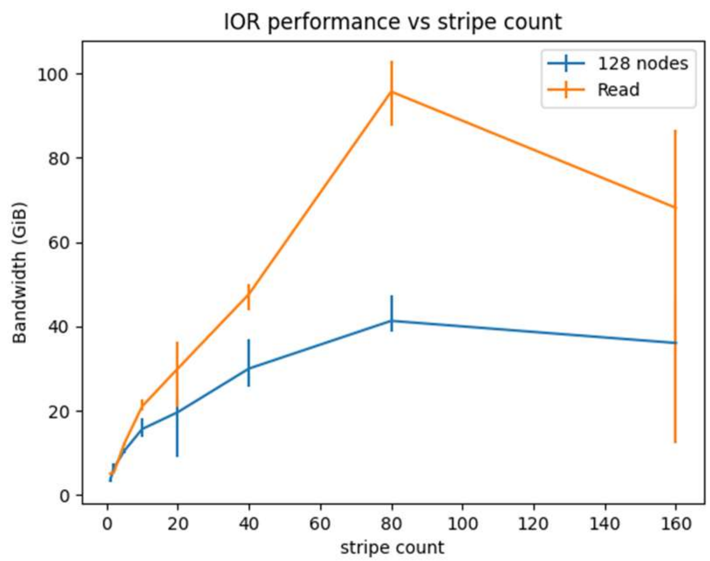
- Parallel file system
  - Parallel in terms of clients
  - Parallel in terms of servers
- File system will hide details, but defaults and details matter for performance





IOR AND STRIPE COUNT

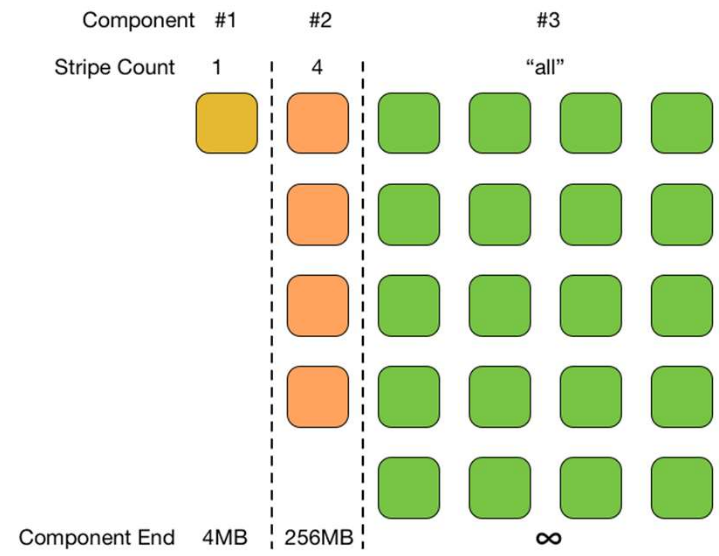
- Default stripe size is 1
  - Why? Most files small: optimizing for common case
- "All the servers" doesn't seem to hurt performance here
  - lfs setstripe -1 /path/to/file
- Could go further with "overstriping"
  - Didn't work on Polaris: investigating
- "Where's my bandwidth?"
  - 128 nodes (network links) here
  - Shared file (so I can experiment with stripe count) means lustre locking overhead/coordination



https://github.com/radix-io/io-sleuthing/tree/main/examples/striping

PROGRESSIVE FILE LAYOUT

- Introduced in Lustre-2.10
- Not on by default at ALCF
- Small files land on one server
- Larger files get more parallelism
- Most files small
- Most data lives in big files



USING PFL

- Example:
  - `lfs setstripe -E4M -c1 -E256M -c4 -E-1 -c-1 /grand/radix-io/scratch/robl/pfl/testfile`
- Confirming it worked:
  - `lfs getstripe /grand/radix-io/scratch/robl/pfl/testfile`
- Lots of detail here: focus on “lcme_extent.e_start” and “lcme_extent.e_end”

```
/grand/radix-io/scratch/robl/pfl/testfile
  lcm_layout_gen:    3
  lcm_mirror_count:  1
  lcm_entry_count:   3
    lcme_id:             1
    lcme_mirror_id:      0
    lcme_flags:          init
    lcme_extent.e_start: 0
    lcme_extent.e_end:   4194304
      lmm_stripe_count:  1
      lmm_stripe_size:   1048576
      lmm_pattern:       raid0
      lmm_layout_gen:    0
      lmm_stripe_offset: 119
      lmm_objects:
      - 0: { l_ost_idx: 119, l_fid: [0x100770000:0x12343c1:0x0] }

    lcme_id:             2
    lcme_mirror_id:      0
    lcme_flags:          0
    lcme_extent.e_start: 4194304
    lcme_extent.e_end:   268435456
      lmm_stripe_count:  4
      lmm_stripe_size:   1048576
      lmm_pattern:       raid0
      lmm_layout_gen:    0
      lmm_stripe_offset: -1

    lcme_id:             3
    lcme_mirror_id:      0
    lcme_flags:          0
    lcme_extent.e_start: 268435456
    lcme_extent.e_end:   EOF
      lmm_stripe_count:  -1
      lmm_stripe_size:   1048576
      lmm_pattern:       raid0
      lmm_layout_gen:    0
      lmm_stripe_offset: -1
```

I/O PERFORMANCE FROM LUSTRE

- 1: Please I am begging you stripe your data across more than one server
- 2: There are some other things you could do, too, but please see step 1
  - Lockahead
  - Overstriping
  - File per process (ugh, fine, but only “break glass in case of emergency”)
    - How will you manage thousands of files?