Understanding and Measuring I/O Performance

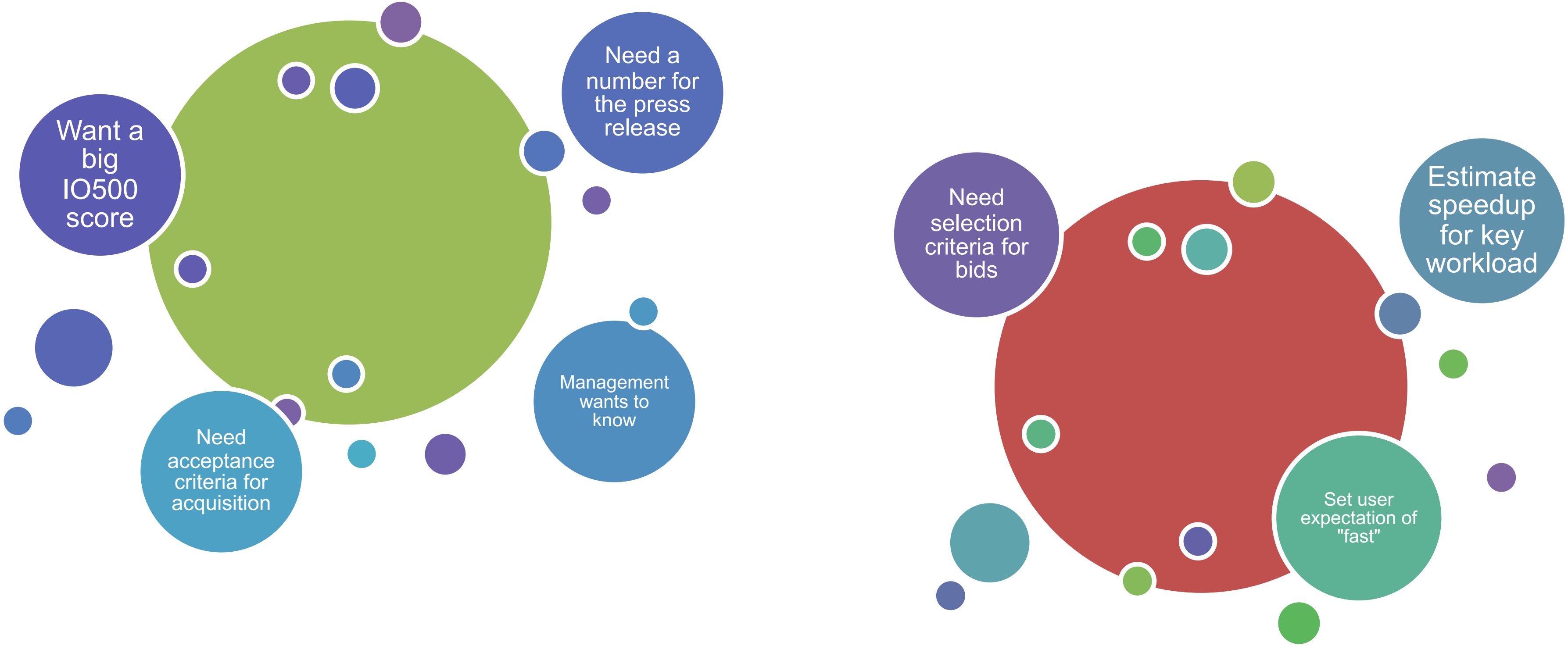
National Energy Research Scientific Computing Center
Lawrence Berkeley National Laboratory

Glenn K. Lockwood, Ph.D.
Storage Architect
Advanced Technologies Group

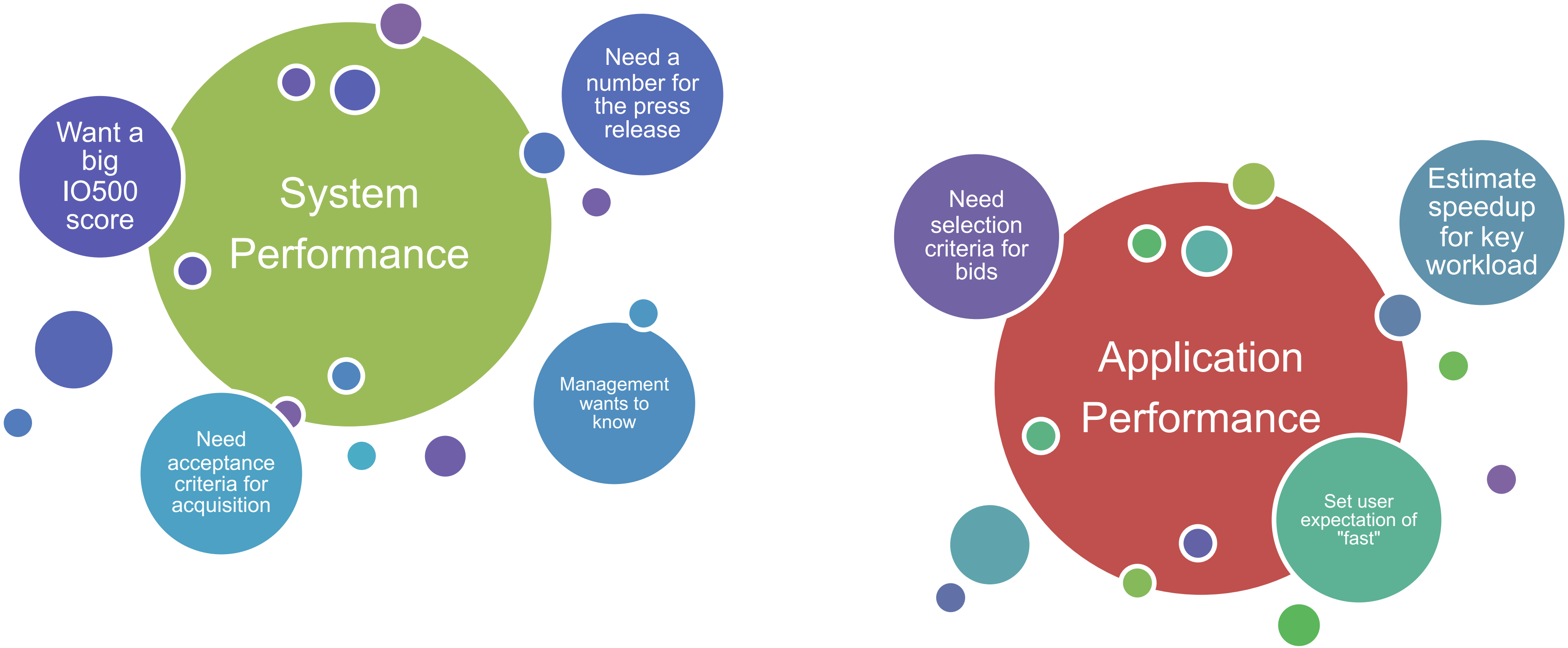
May 9, 2022

Benchmarking, philosophically

Why do you want to benchmark storage?



Why do you want to benchmark storage?



How this shapes your approach to benchmarking

Why?	I care about…	I should measure the biggest number	I should carefully select
	<u>system</u> performance	by any means necessary	scale: maximize number without overwhelming clients/servers
	<u>application</u> performance	subject to constraints of reality	I/O sizes, layouts, clients to match app

~~Experimental Design~~ Benchmarking Process

Why are you benchmarking? → **How will you benchmark?** → **What did you just measure?** → **What is the uncertainty?**

Why are you benchmarking?

- To understand system
  - Bandwidth
  - IOPS
  - Metadata
- To understand apps
  - Parallel checkpoint
  - Ensembles

How will you benchmark?

- IOR, elbencho, …?
- mdtest, md-workbench, …?
- IO500?

What did you just measure?

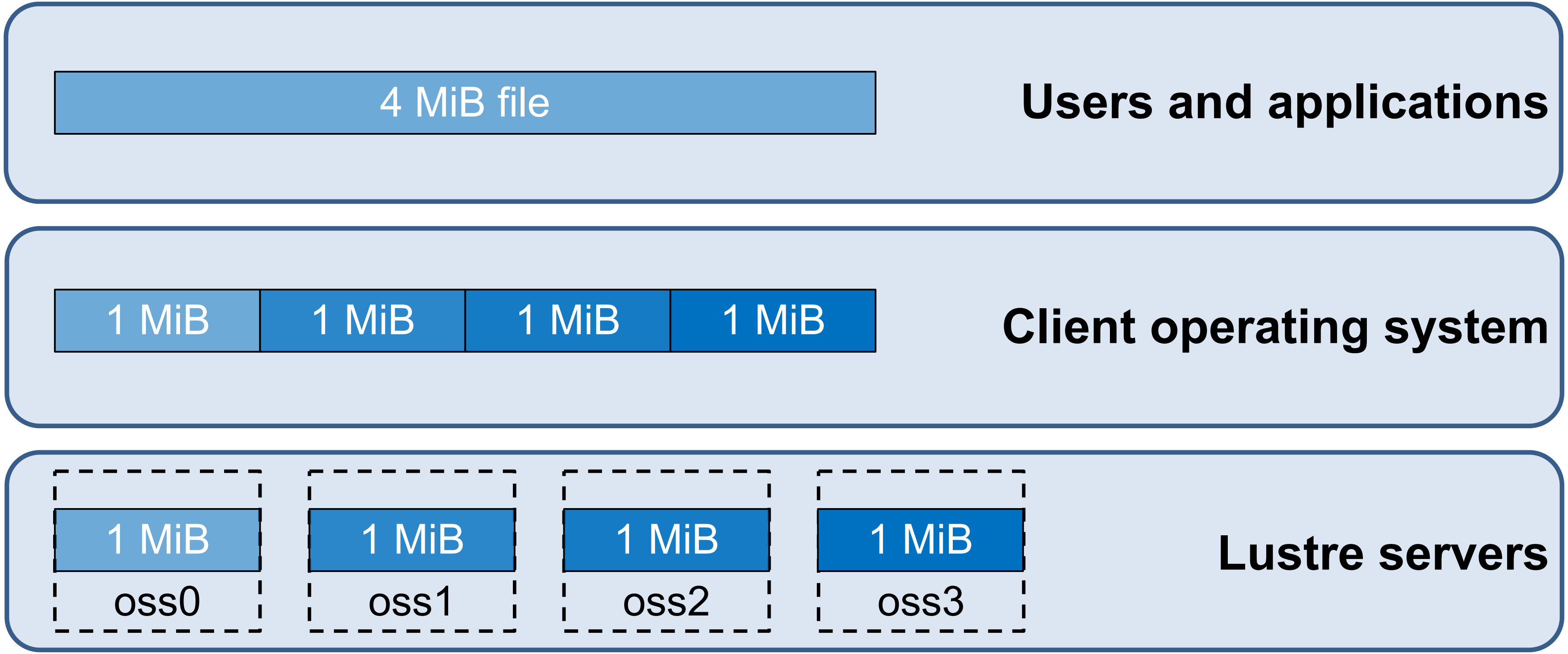
- Client DRAM?
- Network?
- OSS DRAM?
- OSS HDD/SSD?

What is the uncertainty?

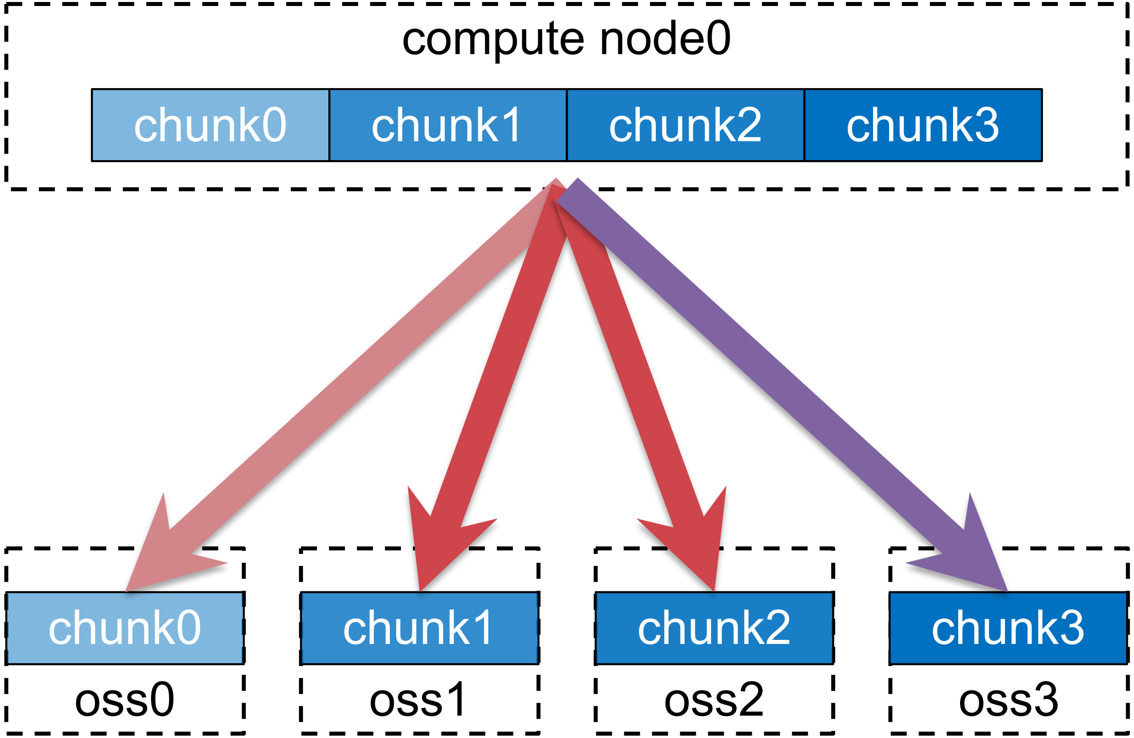
- Standard deviation?
- Multimodality?
- Distribution shape?

Lustre performance in a nutshell

Lustre in principle



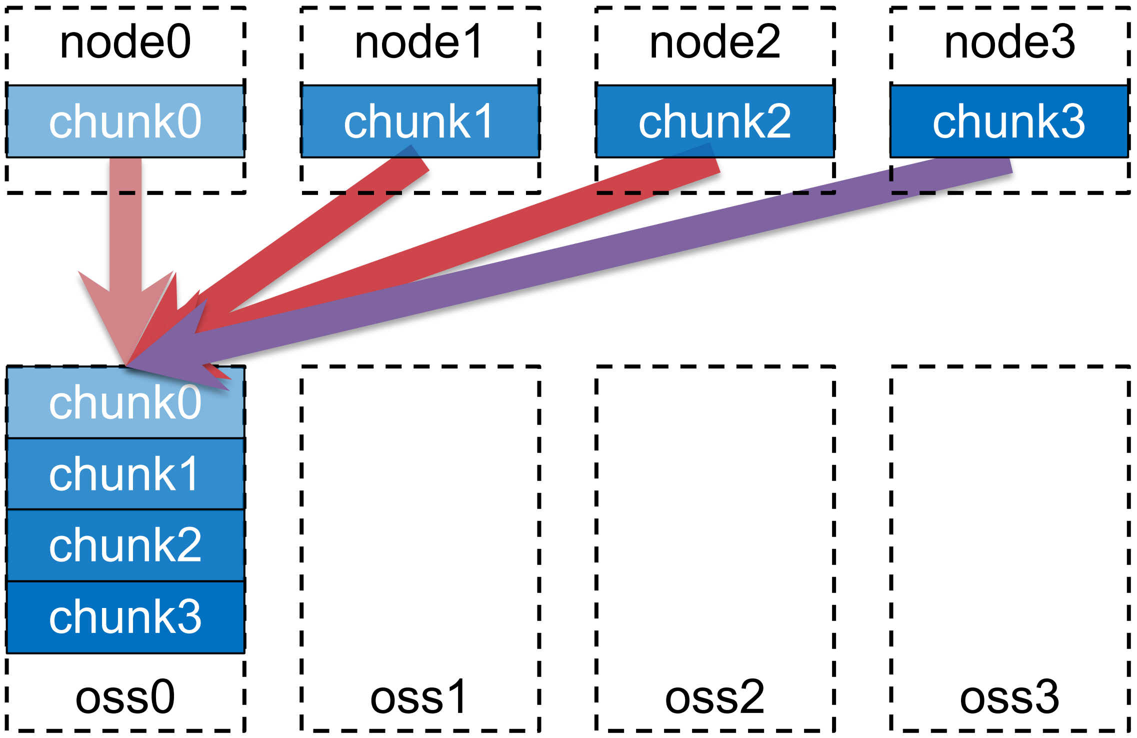
The speed of light still applies



One compute node can't talk to every storage server at full speed

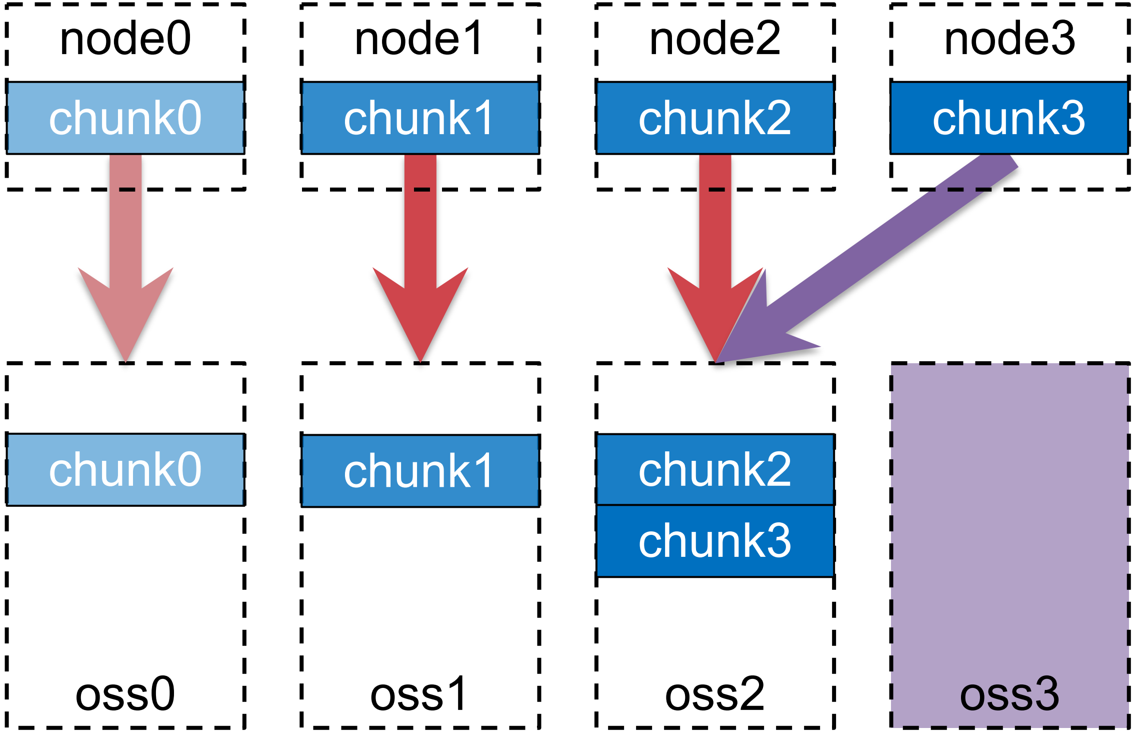
Why dd is *not* useful for testing performance

The speed of light still applies



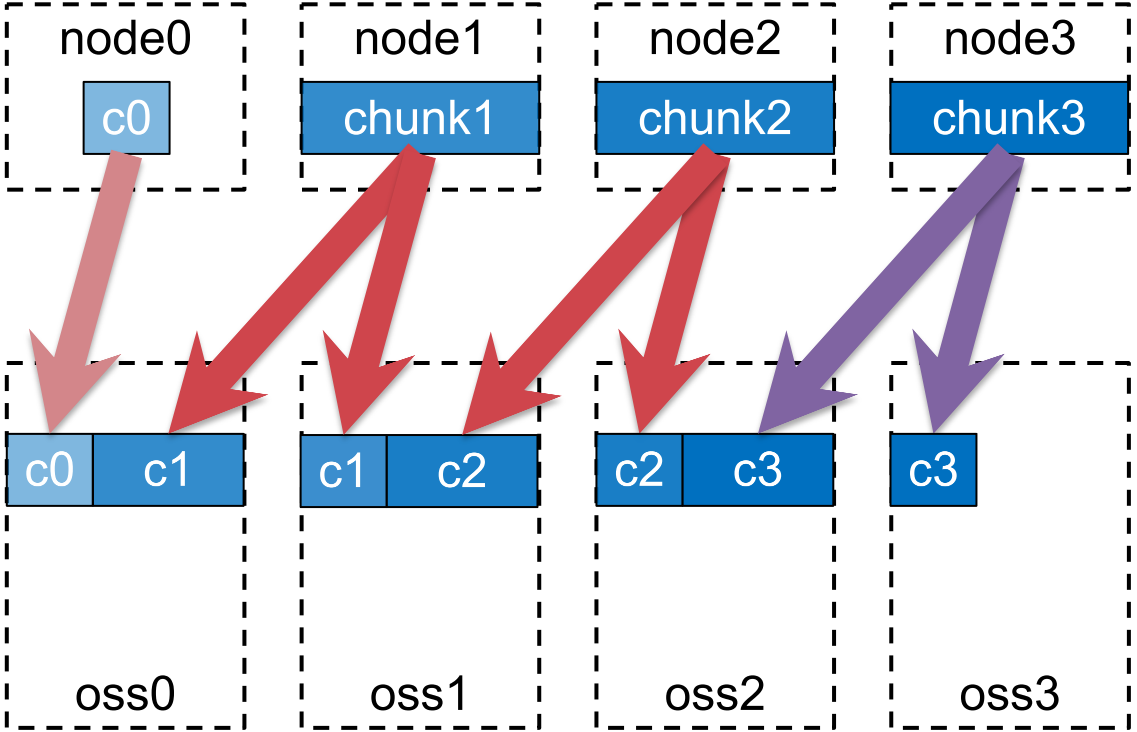
One storage server can't talk to every compute node at full speed

The speed of light still applies



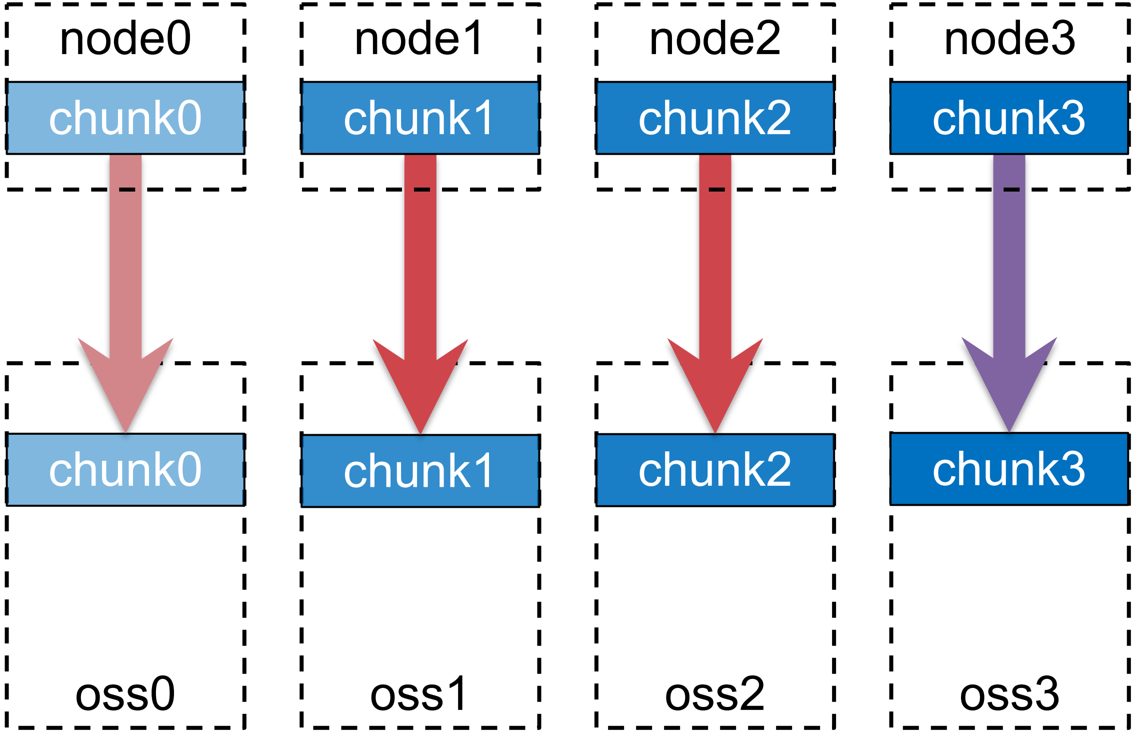
Accidental imbalance caused by a server failure

The speed of light still applies



Accidental imbalance caused by misalignment

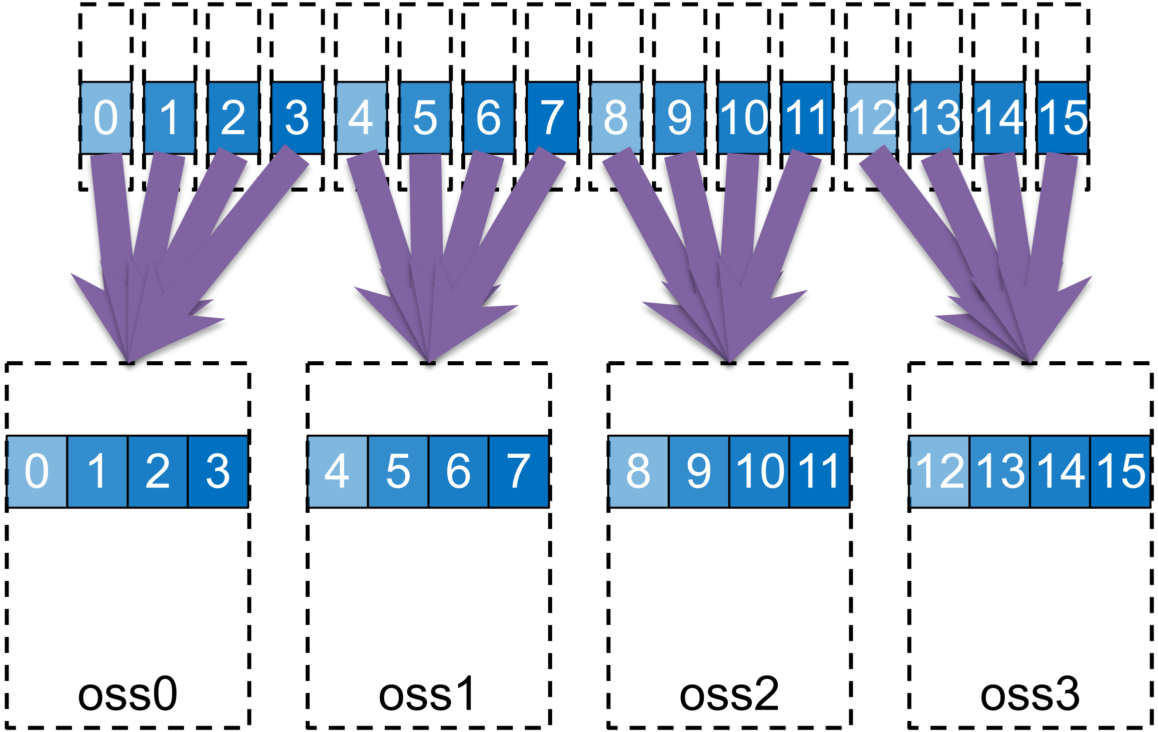
The speed of light still applies



Overall goals when doing I/O to a PFS:

- Each client *and server* handle the same data volume
- Work around gotchas specific to the PFS implementation

The speed of light still applies



Overall goals when doing I/O to a PFS:

- Each client *and server* handle the same data volume
- Work around gotchas specific to the PFS implementation

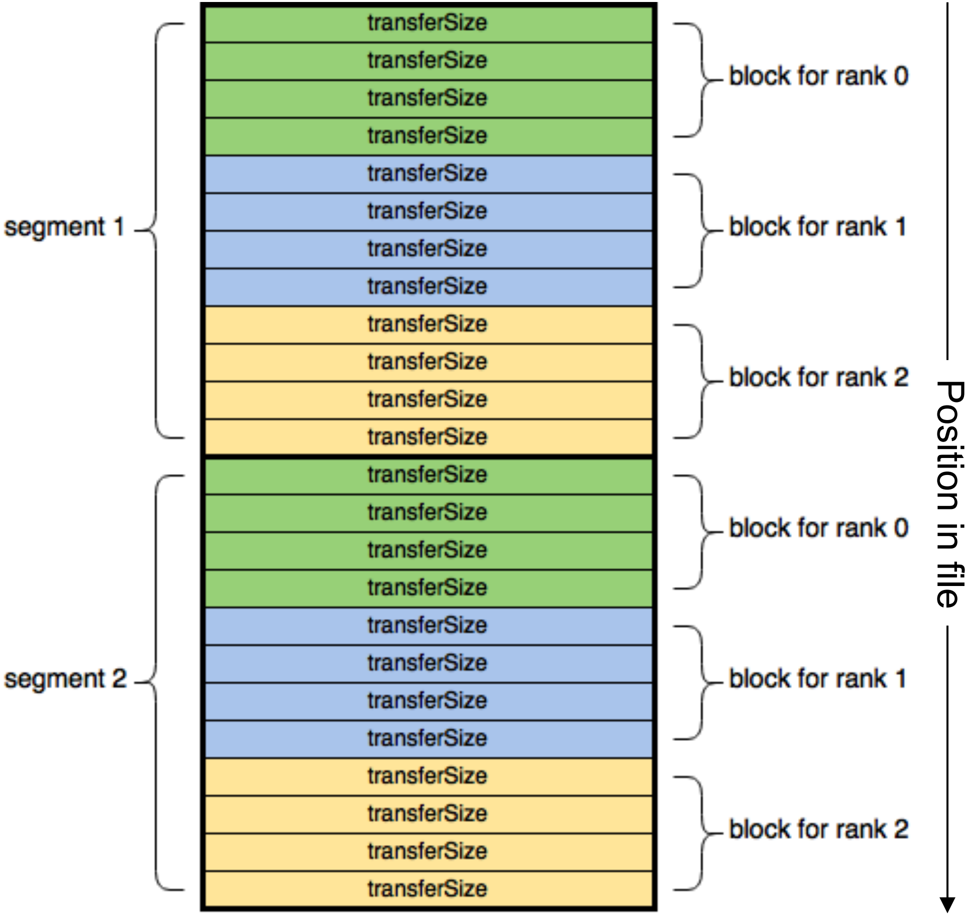
Measuring bandwidth

with IOR

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

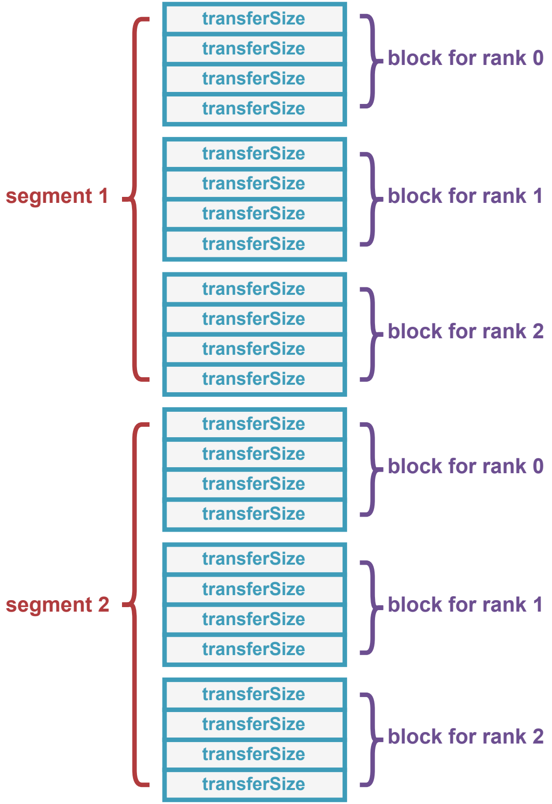
https://github.com/hpc/ior



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

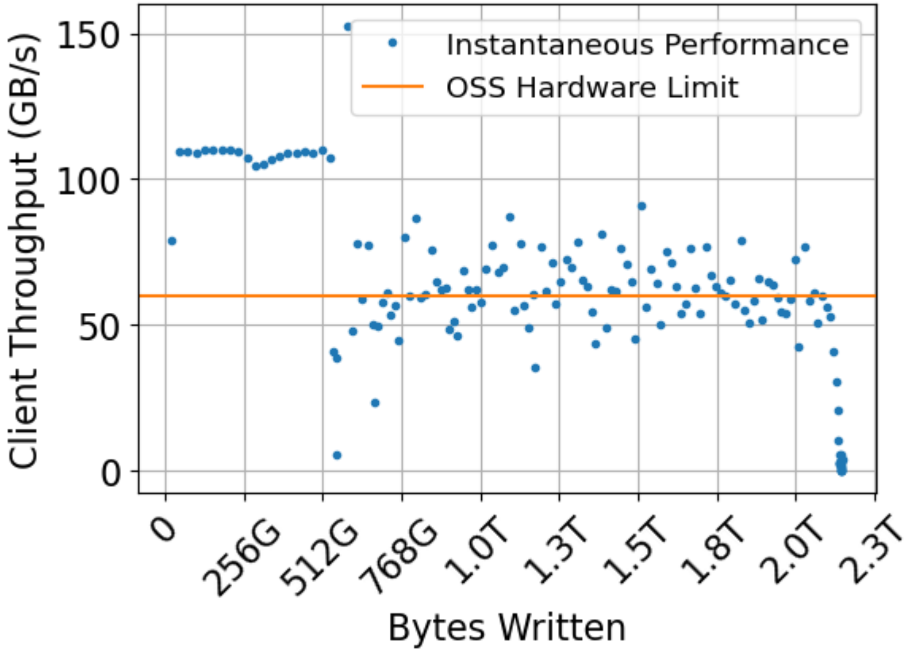
- IOR provides `-F` option to make each rank read/write to its own file instead of default single-shared-file I/O
  - Reduces lock contention within file
  - Can cause metadata load at scale
- Problem: > 400 GB/sec from 4 OSSes is faster than light

```
$ 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

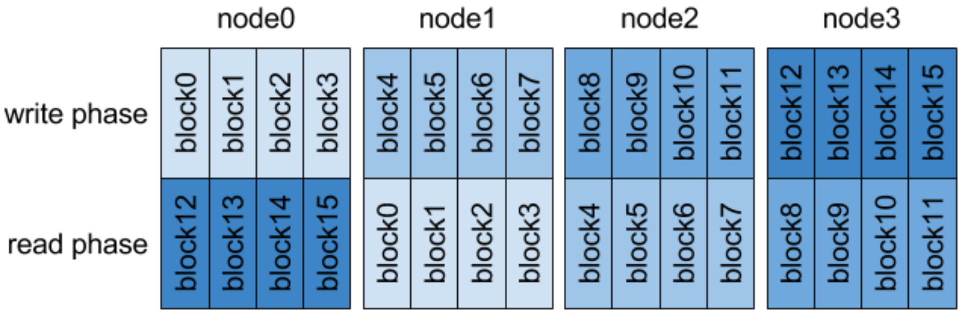
- Unused compute node memory to cache file contents
- Can dramatically affect I/O
  - Writes:
    - only land in local memory at first
    - reordered and sent over network later
    - `max_dirty_mb` and `max_pages_per_rpc`
  - Reads:
    - come out of local memory if data is already there
    - read-after-write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

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?

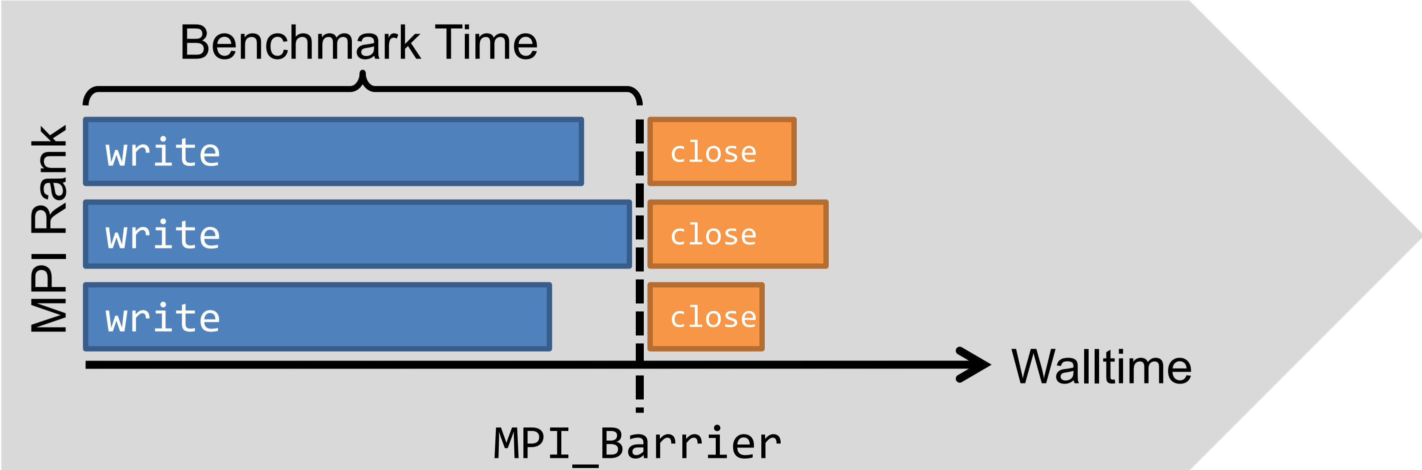



```
$ srun -N 4 -n 64 ./ior -t 1m -b 64m -s 64 -F -C
...
Operation    Max(MiB)
write        63692.33
read         28303.09
```

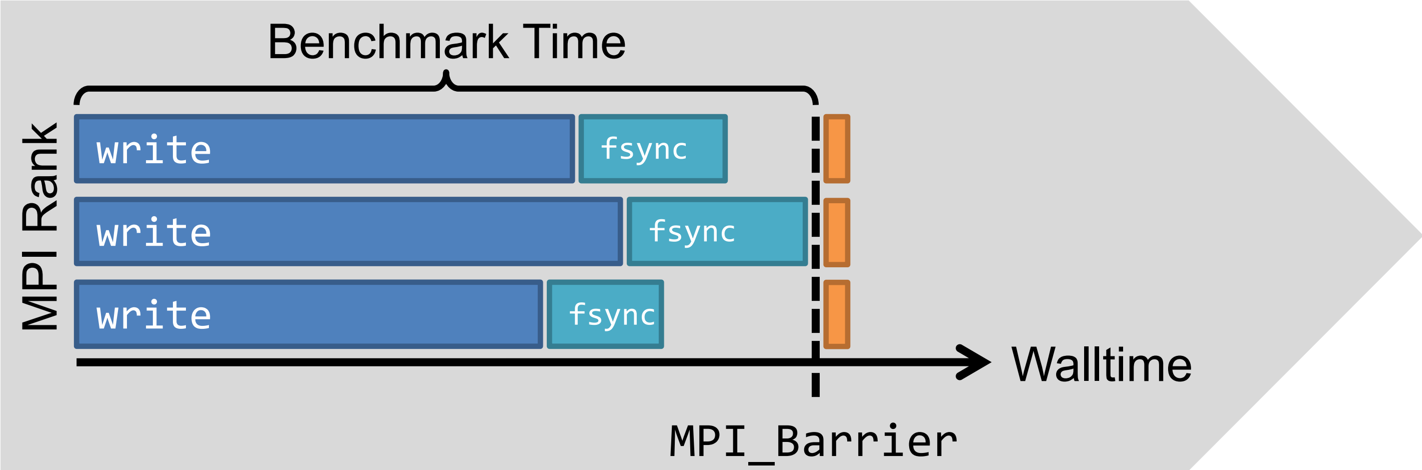
Force sync to account for write cache effects

- 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
```



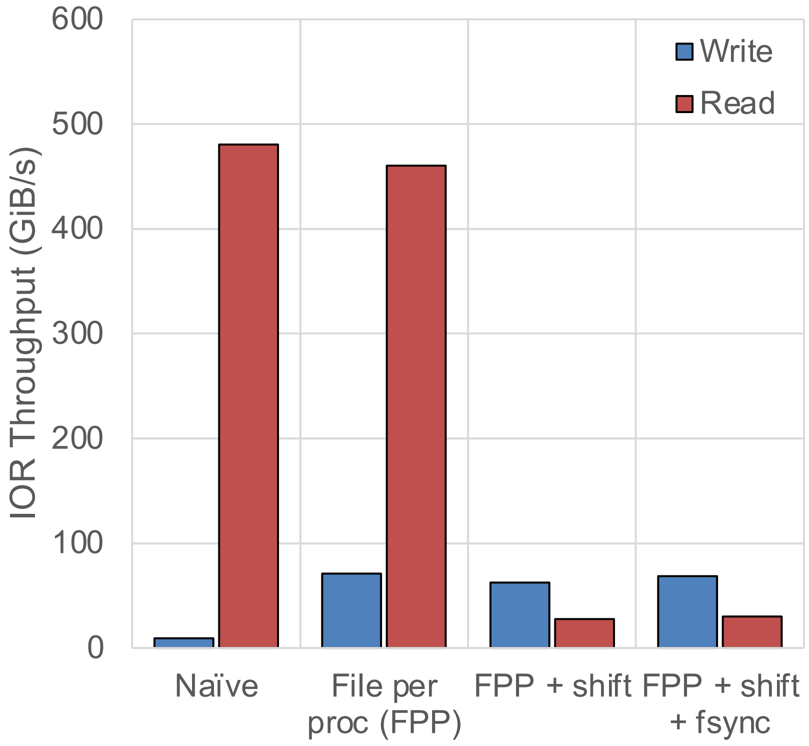
By default, benchmark may appear to hang at the end when files are being closed



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

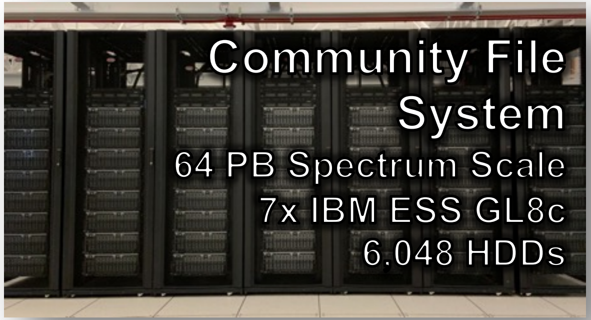
Measuring bandwidth can be complicated

- 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`



IOR Acceptance Tests

Spectrum Scale Bandwidth



8 ppn used

```
$ srun -N 51 -n 408 ./ior -F -C -e -b 32g -t 1m
```

Standard args:

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

Every rank writes (1 × 32) GiB total, 1 MiB at a time
(note: -s not given, so default is 1)

Results:

- 193,717 MB/s write (max)
- 162,753 MB/s read (max)

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

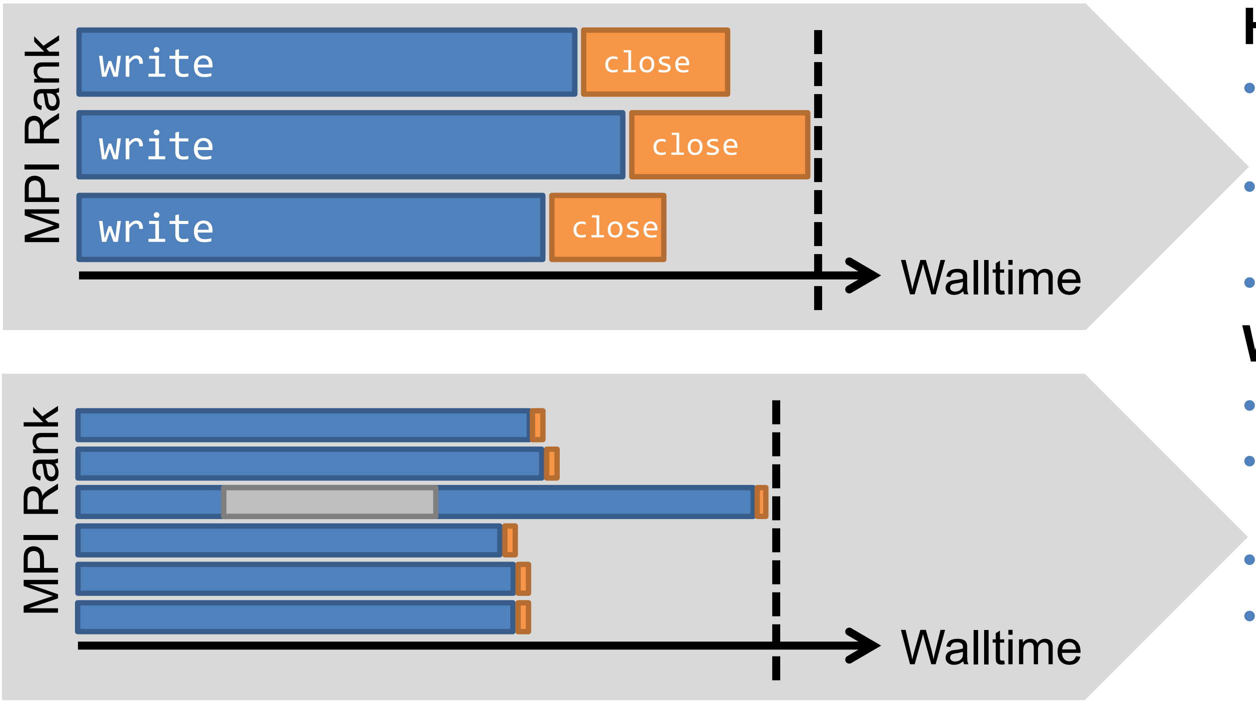
-w Perform write benchmark only
-k Don't delete written files
-r Perform read benchmark only

Separate sruns drop client caches

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

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

Running afoul of “wide” benchmarking



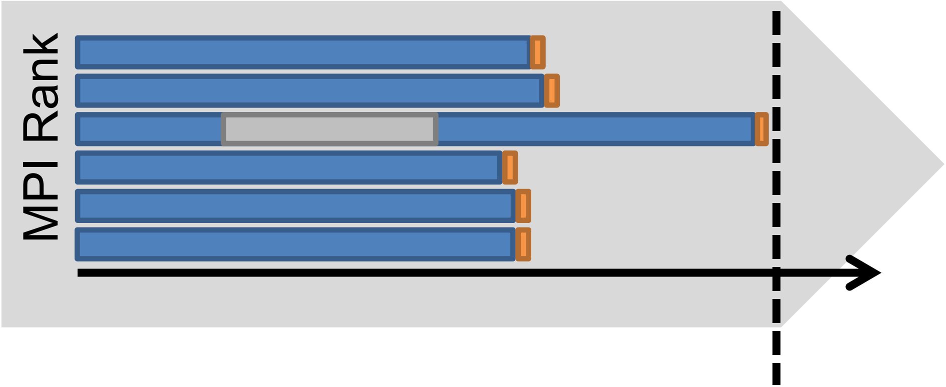
How much -b/-s?

- More is better: overrun cache effects
- More is worse: increase likelihood of hiccup
- Glenn’s goal: 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

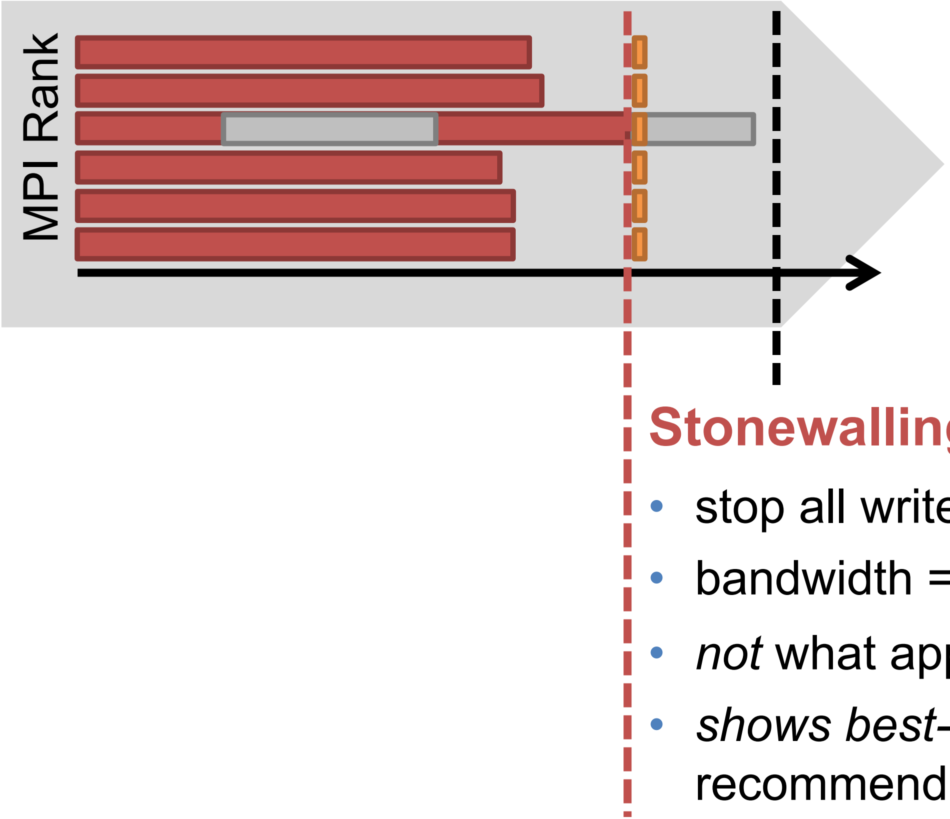
Stonewalling to reduce penalty of stragglers



Default behavior

- all ranks write same total bytes
- timer stops when slowest rank finishes

Stonewalling to reduce penalty of stragglers



Default behavior

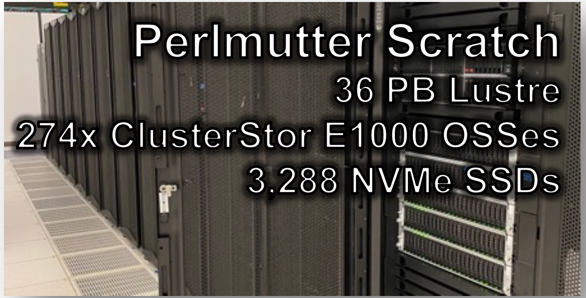
- all ranks write same total bytes
- timer stops when slowest rank finishes

Stonewalling (`-D 30`)

- stop all writes after 30 sec, add up bytes written
- bandwidth = total bytes written / 30 sec [+ fsync time]
- *not* what apps do – apps don't give up if I/O is slow!
- *shows best-case* *system* *capability despite hiccups* – recommended for system acceptance

IOR Acceptance Tests

Lustre Bandwidth – Writes with time limit



4 ppn used

```
$ srun -N 1382 -n 5528 ./ior -F -C -e -g -b 1m -t 1m -s 100000 -D 45 -w
```

Standard args:

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

1 × 100,000 MiB/rank
1 MiB at a time
~527 TiB total

-or-

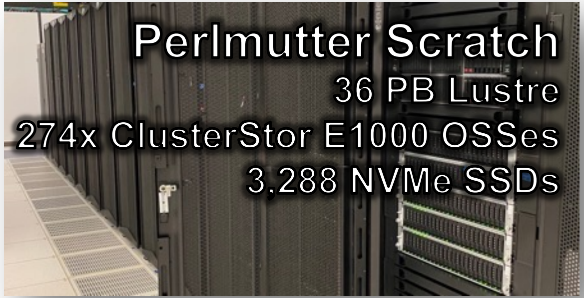
for 45 seconds (`-D 45`),
whichever happens first

`-w` Write-only benchmark

Results: 3,593,657 MB/s write (max)

IOR Acceptance Tests

Lustre Bandwidth – Reads with time limit




```
$ srun -N 1382 -n 5528 ./ior -F -C -e -g -b 1m -t 1m -s 100000 -D 90 -w -k -O stoneWallingWearOut=1

$ srun -N 1382 -n 5528 ./ior -F -C -e -g -b 1m -t 1m -s 100000 -D 30 -r
```

1. Write data to files for 90 seconds
   - `-O stoneWallingWearOut=1` : every rank writes the same amount of bytes even if stonewalling (`-D 90`) cuts the run short
   - Generates uniform files - avoid EOF when ranks are shifted and read back
2. Read back files for 30 seconds

Results: 4,003,761 MB/s read (max)

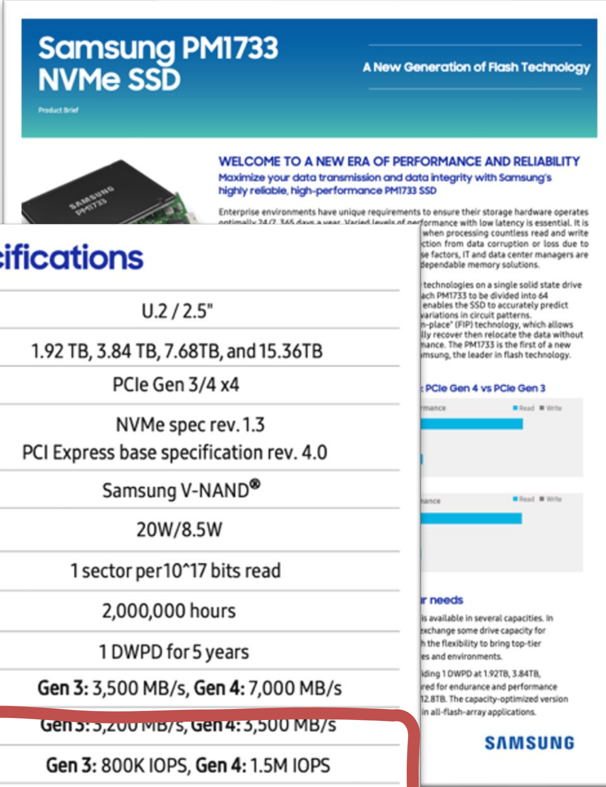
Measuring IOPS

with IOR

Measuring random I/O performance - IOPS

I/O Operations Per Second

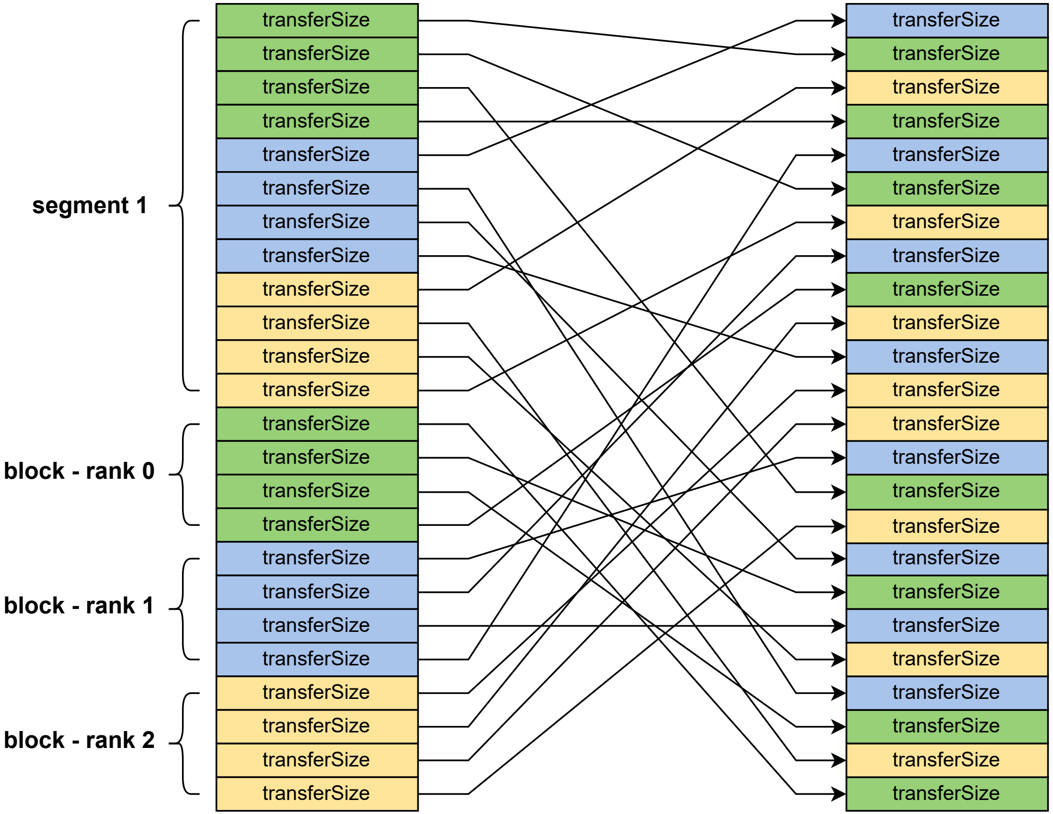
- Move smallest unit of storage from/to arbitrary location
- “smallest” usually 4 KiB (memory page in Linux)
- 1 IOP = 4 KiB I/Os per sec from random offsets
- Historically block-level



Samsung PM1733 specifications

Form factor	U.2 / 2.5"
Capacity	1.92 TB, 3.84 TB, 7.68TB, and 15.36TB
Host interface	PCIe Gen 3/4 x4
Spec Compliance	NVMe spec rev. 1.3 PCI Express base specification rev. 4.0
NAND flash memory	Samsung V-NAND®
Power consumption (Active/Idle)	20W/8.5W
Uncorrectable Bit Error Rate (UBER)	1 sector per 10^17 bits read
Mean Time Between Failure (MTBF)	2,000,000 hours
Endurance	1 DWPD for 5 years
Sequential read	**Gen 3:** 3,500 MB/s, **Gen 4:** 7,000 MB/s
Sequential write	**Gen 3:** 3,200 MB/s, **Gen 4:** 3,500 MB/s
Random read	**Gen 3:** 800K IOPS, **Gen 4:** 1.5M IOPS
Random write	**Gen 3 & 4:** Up to 135,000 IOPS

Switching IOR from “interleaved” to “random”



`-z` randomizes order of each transfer (`-t`)

- `-b` and `-s` still set dataset size
- `-t 4K` sets size of each read/write

Lessons learned still apply

Plus a new gotcha for IOPS

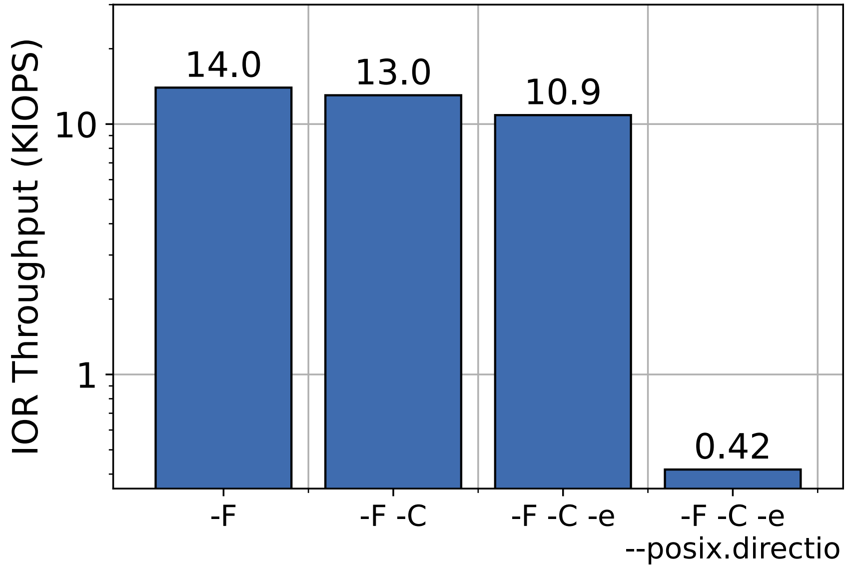
- File per process (`-F`) or shared file?
  - Do you want your IOPS number to reflect lock contention?
  - What are you trying to measure?
- How to cope with client page cache?
  - Read – IOR will never re-read the same transfer from cache with `-C`
  - Write – client caching will reorder/coalesce random writes
- Unique to IOPS - write vs. rewrite
  - Re-writing files randomly has much higher overhead
  - Consider RAID read-modify-write impacts

Overcoming caches for random writes

- `--posix.odirect`
  - forces file I/O to bypass page cache entirely
  - reduces apparent write IOPS
- Which is “true performance?”
  - True random writes are rare
  - Random, direct I/O is rarer
- *Application performance* should include write-back
- *System performance* is better measured with `O_DIRECT`

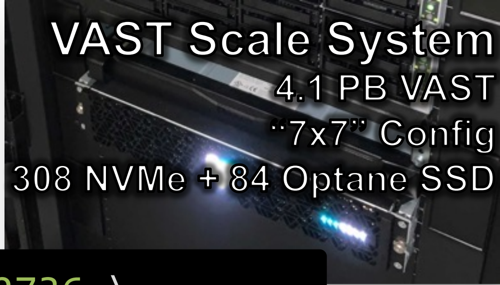
IOR Write IOPS tests

4 clients, 16 ppn
274x NVMe OSTs



IOR Hero Test

VAST IOPS – Writes with time limit



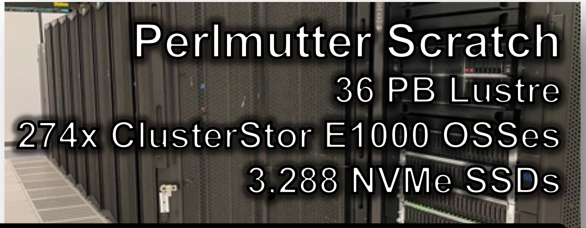

```
$ srun -N 32 -n 4096 ./ior -F -C -e -g -b 1m -t 4k -s 372736 \
                         -D 45 -w -z --posix.odirect -l random
```

- Using `--posix.odirect`
  - Force random pattern to cross network without reordering
  - Oversubscribed clients (128 ppn) to make up for `O_DIRECT` loss
- Other parameters are standard
  - `-D 45` for stonewalling
  - `-w` is write-only test (and no `-k` means don’t bother keeping files after)
  - `-z` for random instead of interleaved

Results: 640,713 IOPS write (max)

IOR Hero Test

Lustre IOPS – Reads with time limit




```
$ srun -N 1024 -n 32768 ./ior -F    -e -g -b 64g -t 64m       -D 90 -w -k -O stoneWallingWearOut=1

$ srun -N 1024 -n 32768 ./ior -F    -e -g -b 64g -t 64m       -D 45 -w    -o tempfiles.dat

$ srun -N 1024 -n 32768 ./ior -F -C -e -g -b  1g -t  4k -s 20 -D 45 -r -k -z
```

1. Write data to files for 45 seconds
   - `-O stoneWallingWearOut=1` to generate uniform file sizes (uppercase O)
   - `-t 64m` for big bandwidth - generate big files for random read test
2. Cleanse the palate
   - write (`-w`) and discard (no `-k`) data to flush out caches (clients + servers) just in case
   - `-o tempfiles.dat` specifies name of files IOR creates (lowercase o)
   - don't want to overwrite dataset generated in Step 1
3. Read back files for 45 seconds
   - Avoid EOF by sizing `-b` and `-s` to match file sizes generated in step 1 from `-D 90 -O stoneWallingWearOut=1`

Results: 117,349,729 IOPS read (max)

Measuring metadata performance

mdtest

The mdtest benchmark

- MPI application benchmark – included with IOR
- Performs metadata operations on configurable directory hierarchies
- Input:
  - # files/dirs to test
  - how deep/wide tree should be
- Output: metadata ops per second
- Configurable backends
  - POSIX, STDIO, MPI-IO
  - same support as IOR

https://github.com/hpc/ior/releases

- `tree.0/`
  - `tree.1/`
    - `tree.3/`
      - `tree.7/`
        - `file.0`
        - `file.1`
      - `tree.4/`
        - `file.2`
        - `file.3`
    - `tree.2/`
      - `tree.5/`
        - `file.4`
        - `file.5`
      - `tree.6/`
        - `file.6`
        - `file.7`

Step 1. Measure metadata performance

```
$ mpirun -n 2 ./mdtest -n 100000
...

2 tasks, 200000 files/directories

SUMMARY rate: (of 1 iterations)
   Operation                    Max          Min
   ---------                    ---          ---
   Directory creation     13873.461    13873.461
   Directory stat         23070.131    23070.131    23070.131      0.000
   Directory rename        6807.317     6807.317     6807.317      0.000
   Directory removal       9581.667     9581.667     9581.667      0.000
   File creation           8798.513     8798.513
   File stat              10907.440    10907.440
   File read              12079.891    12079.891
   File removal           14142.063    14142.063    14142.063      0.000
   Tree creation           2100.302     2100.302     2100.302      0.000
   Tree removal             236.739      236.739      236.739      0.000
```

Operate on 100,000 “things” per MPI process

“thing” = directories

“thing” = files

operations per second

Step 2. Figure out what it’s really doing

What mdtest does:

1. Create directory tree
2. Test directory performance
   1. create
   2. stat
   3. rename
   4. unlink
3. Test file performance
   1. create and write
   2. stat
   3. read and close
   4. unlink
4. Destroy directory tree

MPI_Barrier()

What mdtest tells you:

- how fast one operation *and nothing else* can be sustained
- bulk-synchronous performance (think: file-per-process checkpoint)
- cost of different metadata operations

What mdtest does not tell you:

- compile/untar/python performance
- when a file system will tip over
- how laggy a file system will feel

What these numbers represent

```
$ mpirun -n 2 ./mdtest -n 100000
...

2 tasks, 200000 files/directories

SUMMARY rate: (of 1 iterations)
   Operation                      Max            Min
   ---------                      ---            ---
   Directory creation       13873.461      13873.461      13873.461
   Directory stat           23070.131      23070.131      23070.131
   Directory rename          6807.317       6807.317       6807.317
   Directory removal         9581.667
   File creation             8798.513
   File stat                10907.440      10907.440      10907.440
   File read                12079.891      12079.891
   File removal             14142.063
   Tree creation             2100.302
   Tree removal               236.739        236.739
```

not very interesting

tickles certain key-value based file systems

may be relevant to purge

file-per-process checkpointing

file system walking

Python library loading

think purging

not interesting (rank 0 only)

Selecting which tests to run (default: all)

```
$ mpirun -n 2 ./mdtest -n 100000
...

2 tasks, 200000 files/directories

SUMMARY rate: (of 1 iterations)
   Operation                      Max
   ---------                      ---
   Directory creation        13873.461
   Directory stat            23070.131
   Directory rename           6807.317
   Directory removal          9581.667
   File creation              8798.513
   File stat                 10907.440
   File read                 12079.891
   File removal              14142.063
   Tree creation              2100.302
   Tree removal                236.739
```

Option	Effect
-D	Run only **directory** tests
-F	Run only **file** tests
-C	Run only **create** phase
-T	Run only **stat** phase
-E	Run only **read** phase
-r	Run only **removal** phase

mdtest Acceptance Tests

Lustre on HDDs - No DNE

32 ppn

```
$ srun -N 1620 -n 51840 ./mdtest -n 20 -F -C -T -r
```

Create 20 files/dirs per MPI rank

- `-F` Only run file tests
- `-C` Only run create phase
- `-T` Only run stat phase
- `-r` Only run removal phase

Results:

- 45,945 creates/sec (max)
- 147,502 stats/sec (max)
- 28,213 unlinks/sec (max)

So don't run:

- directory tests
- read phase

mdtest Acceptance Tests

Lustre on HDDs - DNE Phase 1


```
$ srun -N 1620 -n 51840 ./mdtest -n 20 -F -C -T -r -u \
    -d /lus/mdt0@/lus/mdt1@/lus/mdt2@/lus/mdt3@/lus/mdt4
```

- `-d` specifies output directories
  - Multiple directories can be separated by `@`
  - Makes mdtest evenly stripe data across many dirs
  - Can repeat directories if, e.g., one MDT is “better” than others

Results:

- 112,349 creates/sec (max)
- 453,902 stats/sec (max)
- 111,286 unlinks/sec (max)

mdtest Acceptance Tests

Lustre on NVMe - DNE Phase 2


```
$ lfs mkdir -c 4 -D /lus/striped
$ srun -N 64 -n 1024 ./mdtest -n 2441 -F -C -r -d /lus/striped
```

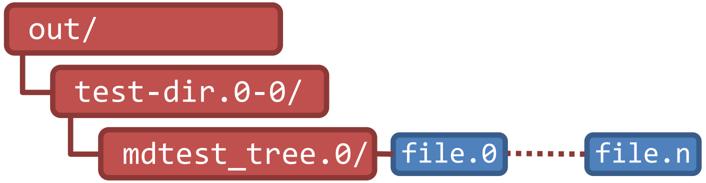
- Create striped metadata directory (`/lus/striped`)
- Use 2,441 files per MPI process
- Run only file tests (`-F`), create (`-C`) and unlink (`-r`) phases
- Work in our newly created striped dir (`-d /lus/striped`)

Results:

- 217,396 creates/sec (max)
- 187,845 unlinks/sec (max)

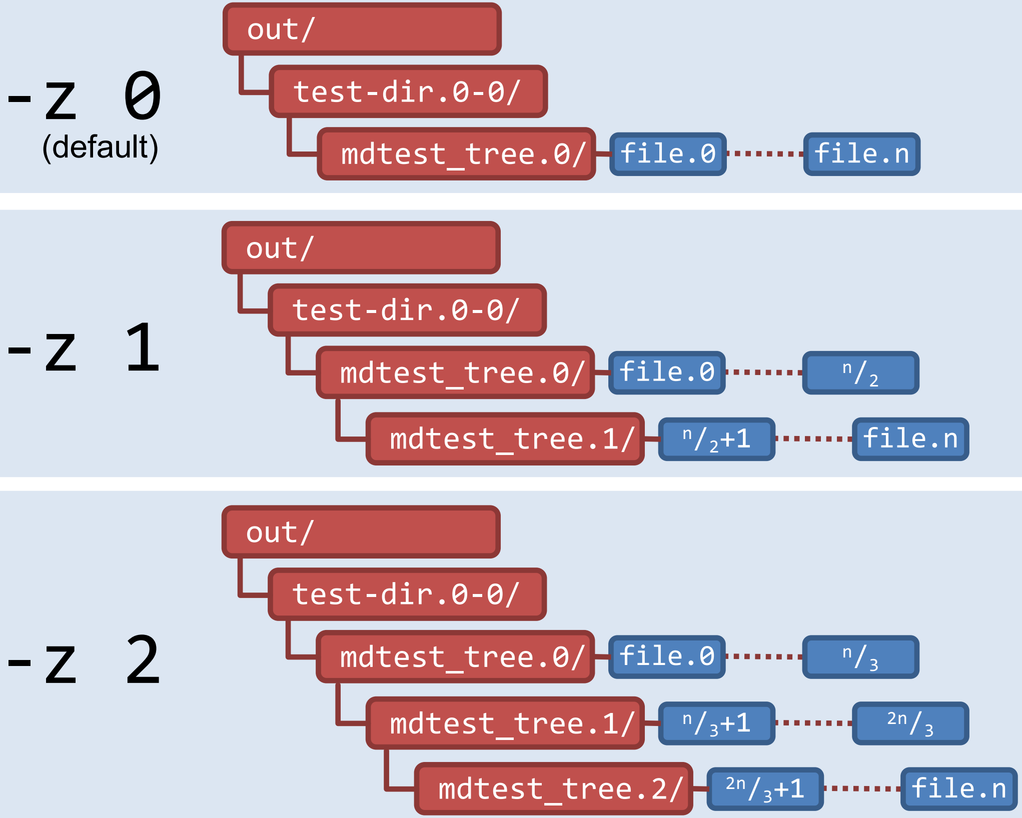
Controlling the directory hierarchy

- Depth factor (`-z`) controls depth
- Branching factor (`-b`) controls breadth
- Files are either
  - spread evenly throughout every directory (default)
  - spread evenly at deepest directories (leaf mode (`-L`))
- Default
  - zero depth (`-z 0`), zero breadth (`-b 1`)
  - dumps all files into one giant directory



Changing depth factor

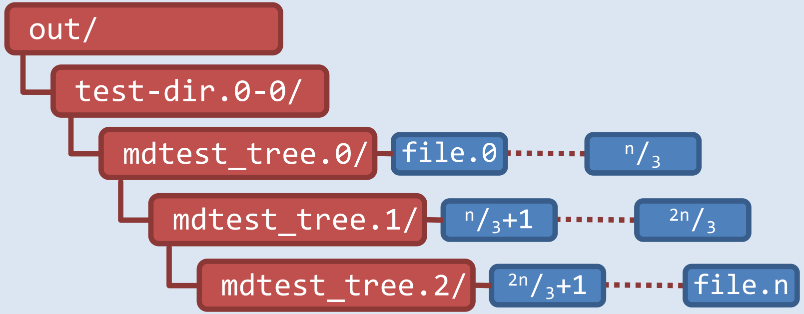
- Creates skinny trees
- Always the same file count in each dir
- Rounds n down if not evenly divisible by (depth+1)



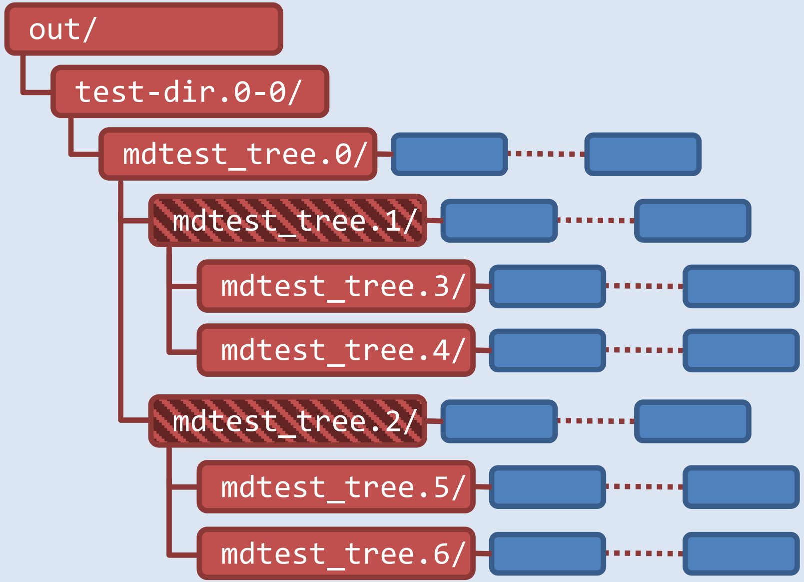
Changing branching factor

- Exponential branching
  - Files evenly spread, rounded down
  - Need high number of files (-n) to get lots of files/dir
- Realistic fs complexity
- Realistic workload?
  - Parallel file transfers?
  - Anything else?

-z 2



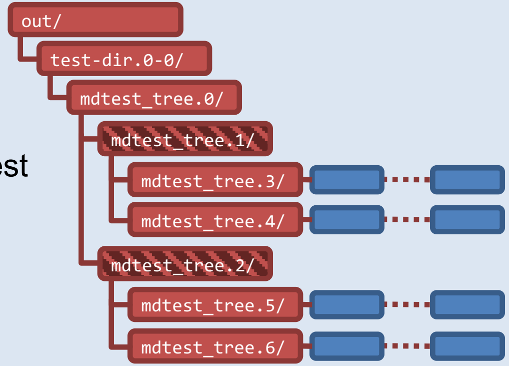
-z 2 -b 2

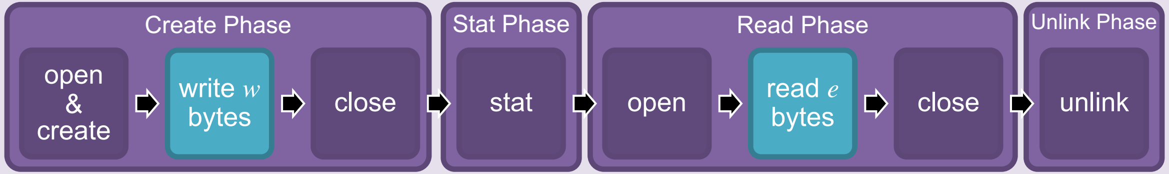


Other practical options

Leaf mode (`-L`)

- `-L -z 2 -b 2`
- Create files at deepest directories only
- Closer to some real datasets





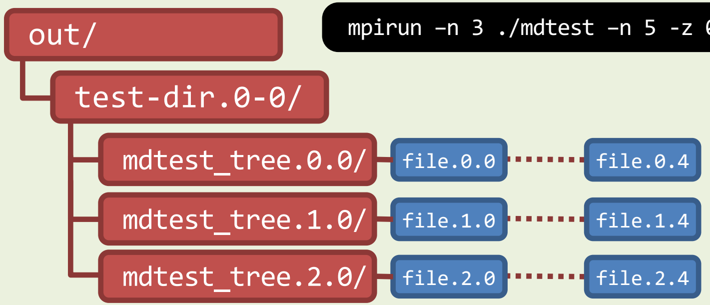
Perform I/O to files

- `-w` and `-e` write/read to each file
- `-w 4096 -e 4096` to create and read 4 KiB files
- Try using with DoM

Directory-per-MPI rank (`-u`)

- Each MPI proc makes own directory for its files within tree
- Reduces directory locking
- Like file-per-proc in IOR

```
mpirun -n 3 ./mdtest -n 5 -z 0 -b 1 -u
```



mdtest Acceptance Tests

Lustre on NVMe - Purge performance


```
$ srun -N 1 -n 32 ./mdtest -n 93750 -F -C -u -d /lus/mdt0@/lus/mdt1 \
    -z 7 -b 3 -w 1048576

$ srun -N 1 -n 32 ./mdtest -n 93750 -F -r -u -d /lus/mdt0@/lus/mdt1 \
    -z 7 -b 3
```

Create 7-deep, 3-wide tree to approximate messiness of user scratch directories

Create 1 MiB files to unlink - because purging empty files is not realistic

Create and unlink in separate runs

Results:

- 6,828 creates/sec (max)
- 70,546 unlinks/sec (max)

Wrapping Up

Be methodical in your approach to benchmarking

Why are you benchmarking?	How will you benchmark?	What did you just measure?	What is the uncertainty?

Why are you benchmarking?

- To understand system
  - Bandwidth
  - IOPS
  - Metadata
- To understand apps
  - Parallel checkpoint
  - Ensembles

How will you benchmark?

- IOR, elbencho, …?
- mdtest, md-workbench, …?
- IO500?

What did you just measure?

- Client DRAM?
- Network?
- OSS DRAM?
- OSS HDD/SSD?

What is the uncertainty?

- Standard deviation?
- Multimodality?
- Distribution shape?

Other benchmarks worth considering

- **elbencho**
  - does similar to IOR+mdtest for non-MPI environments
  - reports performance in realtime while benchmark is running
  - https://github.com/breuner/elbencho
- **md-workbench**
  - workload analogous to compilation
  - messy, incoherent, small-file create/write/stat/read/unlink
  - https://github.com/hpc/ior
- **IO500**
  - IOR, mdtest, parallel find with canned workload patterns
  - https://github.com/IO500/io500

Supplemental resources

- **Getting started with…**
  - IOR: https://glennklockwood.blogspot.com/2016/07/basics-of-io-benchmarking.html
  - mdtest: https://www.glennklockwood.com/benchmarks/mdtest.html
  - elbencho: https://www.glennklockwood.com/benchmarks/elbencho.html
  - md-workbench: https://www.glennklockwood.com/benchmarks/md-workbench.html
- **Example acceptance test and hero run parameters:** https://www.glennklockwood.com/benchmarks/ior-results.html

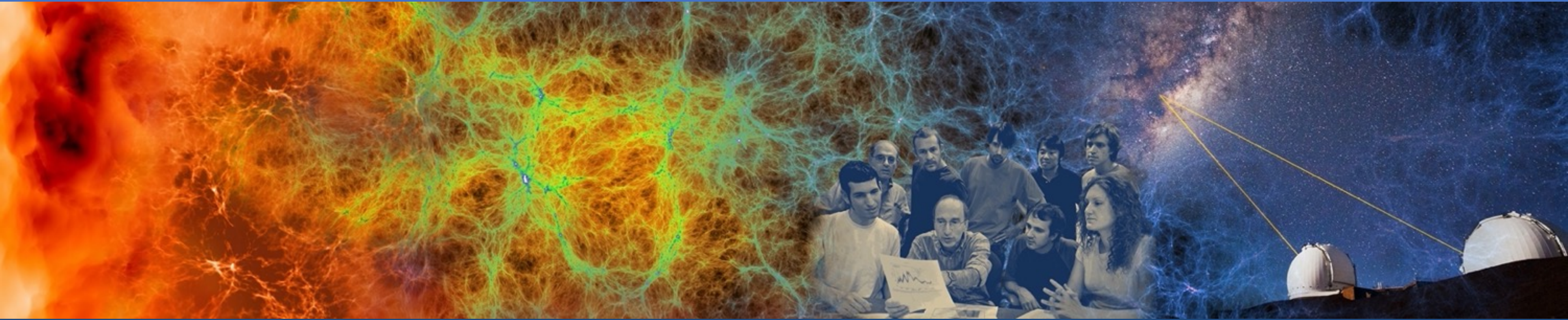
Thank you!

Special thanks to

Doug Petesch, John Fragalla, and Bill Loewe

(all of HPE/Cray) for many of the examples and insights presented.

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-05CH11231. This research used resources and data generated from resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.



NERSC BERKELEY LAB U.S. DEPARTMENT OF ENERGY Office of Science