Benchmarking, philosophically
Why do you want to benchmark storage?

- Want a big IO500 score
- Need a number for the press release
- Management wants to know
- Need acceptance criteria for acquisition
- Need selection criteria for bids
- Estimate speedup for key workload
- Set user expectation of "fast"
Why do you want to benchmark storage?

**System Performance**
- Want a big IO500 score
- Need a number for the press release
- Management wants to know
- Need acceptance criteria for acquisition

**Application Performance**
- Need selection criteria for bids
- Estimate speedup for key workload
- Set user expectation of "fast"

**Application Performance**
- Need selection criteria for bids
- Estimate speedup for key workload
- Set user expectation of "fast"
I care about…

Why?

system performance

I should measure the biggest number

by any means necessary

subject to constraints of reality

scale: maximize number without overwhelming clients/servers

I/O sizes, layouts, clients to match app

How this shapes your approach to benchmarking
Experimental Design Benchmarking Process

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
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 \textit{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

```bash
$ srun -N 4 -n 64 ./ior -t 1m -b 64m -s 64
```

<table>
<thead>
<tr>
<th>Operation</th>
<th>Max(MiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>write</td>
<td>9539.38</td>
</tr>
<tr>
<td>read</td>
<td>492123.04</td>
</tr>
</tbody>
</table>
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

```bash
$ 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 graph]

- 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?

```bash
$ 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

```bash
$ 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!
  o Client caches and sync
  o File per proc vs. shared file
  o 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

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

Every rank writes $(1 \times 32)$ GiB total, 1 MiB at a time
(note: -s not given, so default is 1)
IOR Acceptance Tests
Lustre Bandwidth

Only 4 ppn needed

srun \(-N 960 \,-n 3840\)

srun \(-N 960 \,-n 3840\)

./ior \(-F \,-C \,-e \,-g\)

-b 4m \,-t 4m \,-s 1638

-w \,-k

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

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
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

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

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

1 $\times$ 100,000 MiB/rank
1 MiB at a time
~527 TiB total

- or -

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

Results: 3,593,657 MB/s write (max)
IOR Acceptance Tests
Lustre Bandwidth – Reads with time limit

1. Write data to files for 90 seconds
   - `-O stoneWallWearOut=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
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
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?
  o Do you want your IOPS number to reflect lock contention?
  o What are you trying to measure?

• How to cope with client page cache?
  o Read – IOR will never re-read the same transfer from cache with -C
  o Write – client caching will reorder/coalesce random writes

• Unique to IOPS - write vs. rewrite
  o Re-writing files randomly has much higher overhead
  o 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

- 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

1. Write data to files for 45 seconds
   - `-0 stoneWallWearOut=1` to generate uniform file sizes (uppercase O)
   - `-t 64m` for big bandwidth - generate big files for random read test

2. Cleanse the palate
   - `-w` and `-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 -0 stoneWallWearOut=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
Step 1. Measure metadata performance

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

2 tasks, 200000 files/directories

SUMMARY rate: (of 1 iterations)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Max</th>
<th>Min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directory creation</td>
<td>13873.461</td>
<td>13873.461</td>
</tr>
<tr>
<td>Directory stat</td>
<td>23070.131</td>
<td>23070.131</td>
</tr>
<tr>
<td>Directory rename</td>
<td>6807.317</td>
<td>6807.317</td>
</tr>
<tr>
<td>Directory removal</td>
<td>9581.667</td>
<td>9581.667</td>
</tr>
<tr>
<td>File creation</td>
<td>8798.513</td>
<td>8798.513</td>
</tr>
<tr>
<td>File stat</td>
<td>10907.440</td>
<td>10907.440</td>
</tr>
<tr>
<td>File read</td>
<td>12079.891</td>
<td>12079.891</td>
</tr>
<tr>
<td>File removal</td>
<td>14142.063</td>
<td>14142.063</td>
</tr>
<tr>
<td>Tree creation</td>
<td>2100.302</td>
<td>2100.302</td>
</tr>
<tr>
<td>Tree removal</td>
<td>236.739</td>
<td>236.739</td>
</tr>
</tbody>
</table>

Operate on 100,000 “things” per MPI process

“thing” = directories

“thing” = files

operations per second

23070.131 0.000
6807.317 0.000
9581.667 0.000
8798.513 0.000
10907.440 0.000
12079.891 0.000
14142.063 0.000
2100.302 0.000
236.739 0.000
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

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

$ mpiexec -n 2 ./mdtest -n 100000
...
2 tasks, 200000 files/directories

<table>
<thead>
<tr>
<th>Operation</th>
<th>Max</th>
<th>Min</th>
<th>Mean</th>
<th>Std Dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directory creation</td>
<td>13873.461</td>
<td>13873.461</td>
<td>13873.461</td>
<td>0.000</td>
</tr>
<tr>
<td>Directory stat</td>
<td>23070.131</td>
<td>23070.131</td>
<td>23070.131</td>
<td>0.000</td>
</tr>
<tr>
<td>Directory rename</td>
<td>6807.317</td>
<td>6807.317</td>
<td>6807.317</td>
<td>0.000</td>
</tr>
<tr>
<td>Directory removal</td>
<td>9581.667</td>
<td>9581.667</td>
<td>9581.667</td>
<td>0.000</td>
</tr>
<tr>
<td>File creation</td>
<td>8798.513</td>
<td>8798.513</td>
<td>8798.513</td>
<td>0.000</td>
</tr>
<tr>
<td>File stat</td>
<td>10907.440</td>
<td>10907.440</td>
<td>10907.440</td>
<td>0.000</td>
</tr>
<tr>
<td>File read</td>
<td>12079.891</td>
<td>12079.891</td>
<td>12079.891</td>
<td>0.000</td>
</tr>
<tr>
<td>File removal</td>
<td>14142.063</td>
<td>14142.063</td>
<td>14142.063</td>
<td>0.000</td>
</tr>
<tr>
<td>Tree creation</td>
<td>2100.302</td>
<td>2100.302</td>
<td>2100.302</td>
<td>0.000</td>
</tr>
<tr>
<td>Tree removal</td>
<td>236.739</td>
<td>236.739</td>
<td>236.739</td>
<td>0.000</td>
</tr>
</tbody>
</table>

not interesting (rank 0 only)
tickles certain key-value based file systems
may be relevant to purge
file-per-process checkpointing
Python library loading
think purging

not very interesting
Selecting which tests to run (default: all)

```bash
$ mpirun -n 2 ./mdtest -n 100000
...
2 tasks, 200000 files/directories
```

**SUMMARY rate: (of 1 iterations)**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directory creation</td>
<td>13873.461</td>
</tr>
<tr>
<td>Directory stat</td>
<td>23070.131</td>
</tr>
<tr>
<td>Directory rename</td>
<td>6807.317</td>
</tr>
<tr>
<td>Directory removal</td>
<td>9581.667</td>
</tr>
<tr>
<td>File creation</td>
<td>8798.513</td>
</tr>
<tr>
<td>File stat</td>
<td>10907.440</td>
</tr>
<tr>
<td>File read</td>
<td>12079.891</td>
</tr>
<tr>
<td>File removal</td>
<td>14142.063</td>
</tr>
<tr>
<td>Tree creation</td>
<td>2100.302</td>
</tr>
<tr>
<td>Tree removal</td>
<td>236.739</td>
</tr>
</tbody>
</table>

**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

```bash
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/mt0@/lus/mt1@/lus/mt2@/lus/mt3@/lus/mt4

• -d specifies output directories
  o Multiple directories can be separated by @
  o Makes mdtest evenly stripe data across many dirs
  o 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

- 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)

- z 0
  (default)

- z 1

- z 2

(out/)

(test-dir.0-0/)

(mdtest_tree.0/)

(file.0)

(file.n)

(out/)

(test-dir.0-0/)

(mdtest_tree.0/)

(file.0)

(n/2)

(file.n)

(out/)

(test-dir.0-0/)

(mdtest_tree.0/)

(file.0)

(n/3)

(file.n)

(out/)

(test-dir.0-0/)

(mdtest_tree.0/)

(file.0)

(n/3+1)

(2n/3)

(file.n)
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?
Other practical options

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

Other practical options

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

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

Create and unlink in separate runs

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

Results:
- 6,828 creates/sec (max)
- 70,546 unlinks/sec (max)
Wrapping Up
Be methodical in your approach to benchmarking

**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](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](https://github.com/hpc/ior)

- **IO500**
  - IOR, mdtest, parallel find with canned workload patterns
  - [https://github.com/IO500/io500](https://github.com/IO500/io500)
Supplemental resources

• Getting started with…
  o mdtest: https://www.glennklockwood.com/benchmarks/mdtest.html
  o elbencho: https://www.glennklockwood.com/benchmarks/elbencho.html
  o 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.