Do You Know What Your I/O Is Doing? (and how to fix it?)

William Gropp
www.cs.illinois.edu/~wgropp

Messages

- Current I/O performance is often appallingly poor
  - Even relative to what current systems can achieve
  - Part of the problem is the I/O interface semantics
- Many applications need to rethink their approach to I/O
  - Not sufficient to "fix" current I/O implementations
- HPC Centers have been complicit in causing this problem
  - By asking users the wrong question
  - By using their response as an excuse to keep doing the same thing

Just How Bad Is Current I/O Performance?

- Much of the data (and some slides) taken from "A Multiplatform Study of I/O Behavior on Petascale Supercomputers," Huong Luu, Marianne Winslett, William Gropp, Robert Ross, Philip Carns, Kevin Harms, Prabhat, Suren Byna, and Yushu Yao, presented at HPDC'15.
  - ♦ This paper has lots more data – consider this presentation a sampling
    - http://www.hpdc.org/2015/program/slides/luu.pdf
    - http://dl.acm.org/citation.cfm?doid=2749246.2749269
- Thanks to Luu, Behzad, and the Blue Waters staff and project for Blue Waters results
  - ♦ Analysis part of PAID program at Blue Waters

I/O Logs Captured By Darshan, A Lightweight I/O Characterization Tool

- Instruments I/O functions at multiple levels
- Reports key I/O characteristics
- Does not capture text I/O functions
- Low overhead → Automatically deployed on multiple platforms.
- http://www.mcs.anl.gov/research/projects/darshan/

1867

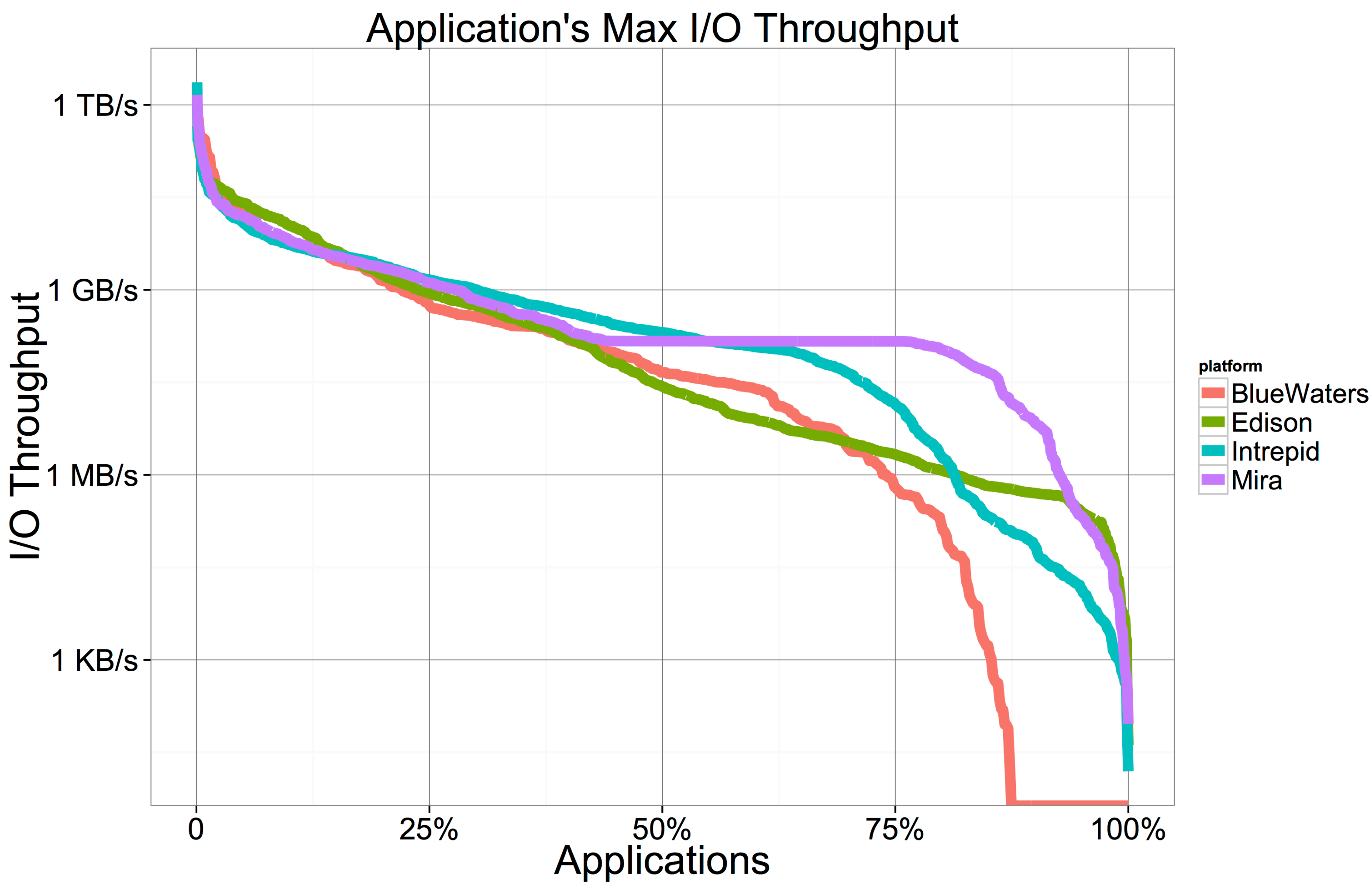
Caveats on Darshan Data

- Users can opt out
  - ♦ Not all applications recorded; typically about ½ on DOE systems
- Data saved at MPI_Finalize
  - ♦ Applications that don't call MPI_Finalize, e.g., run until time is expired and then restart from the last checkpoint, aren't covered
- About ½ of Blue Waters Darshan data not included in analysis

I/O log dataset: 4 platforms, >1M jobs, almost 7 years combined

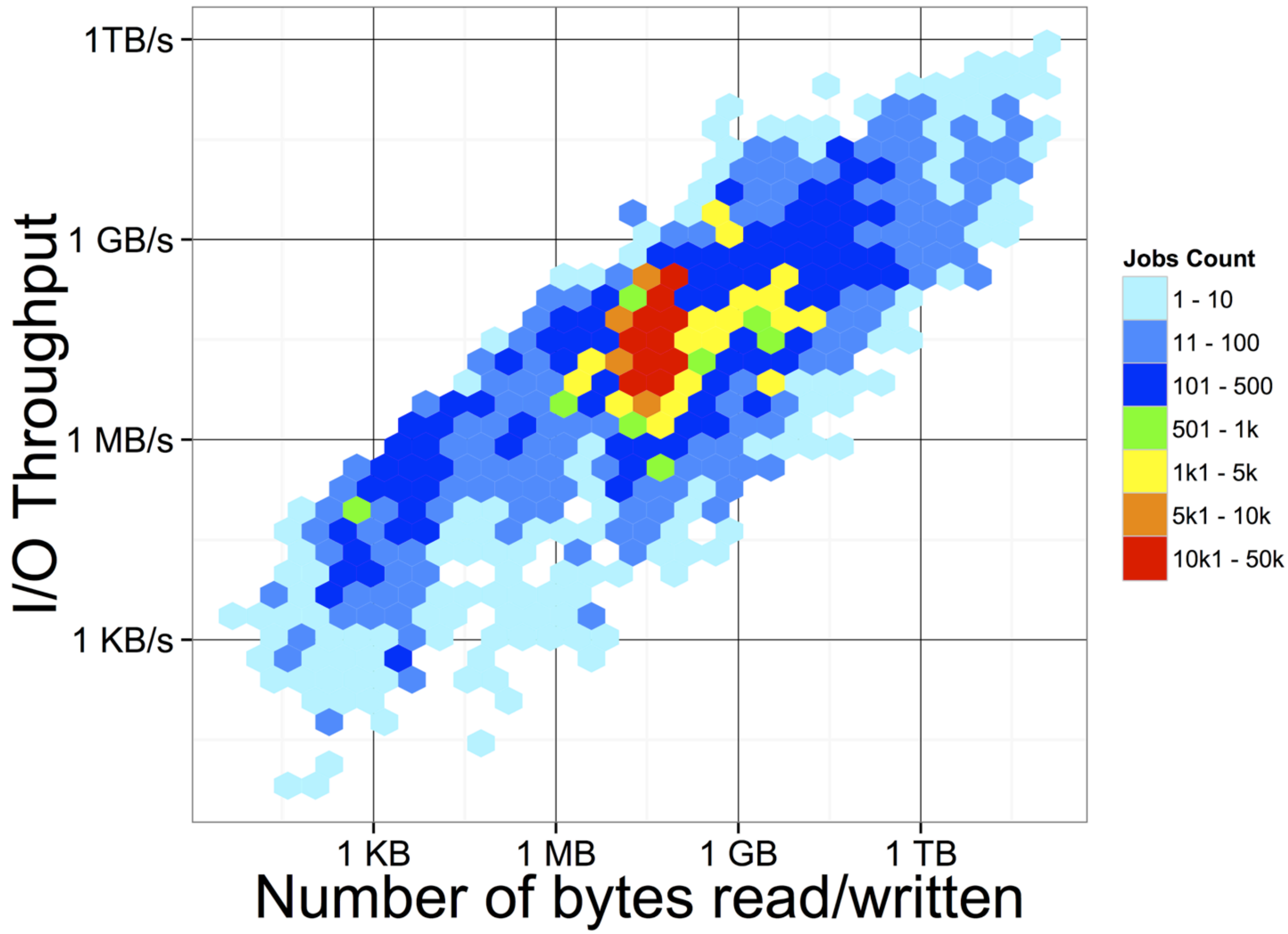
	Intrepid	Mira	Edison	Blue Waters
Architecture	BG/P	BG/Q	Cray XC30	Cray XE6/ XK7
Peak Flops	0.557 PF	10 PF	2.57 PF	13.34 PF
Cores	160K	768K	130K	792K+59K smx
Total Storage	6 PB	24 PB	7.56 PB	26.4 PB
Peak I/O Throughput	88 GB/s	240 GB/s	168 GB/s	963 GB/s
File System	GPFS	GPFS	Lustre	Lustre
# of jobs	**239K**	**137K**	**703K**	**300K**
Time period	**4 years**	**18 months**	**9 months**	**6 months**

Very Low I/O Throughput Is The Norm

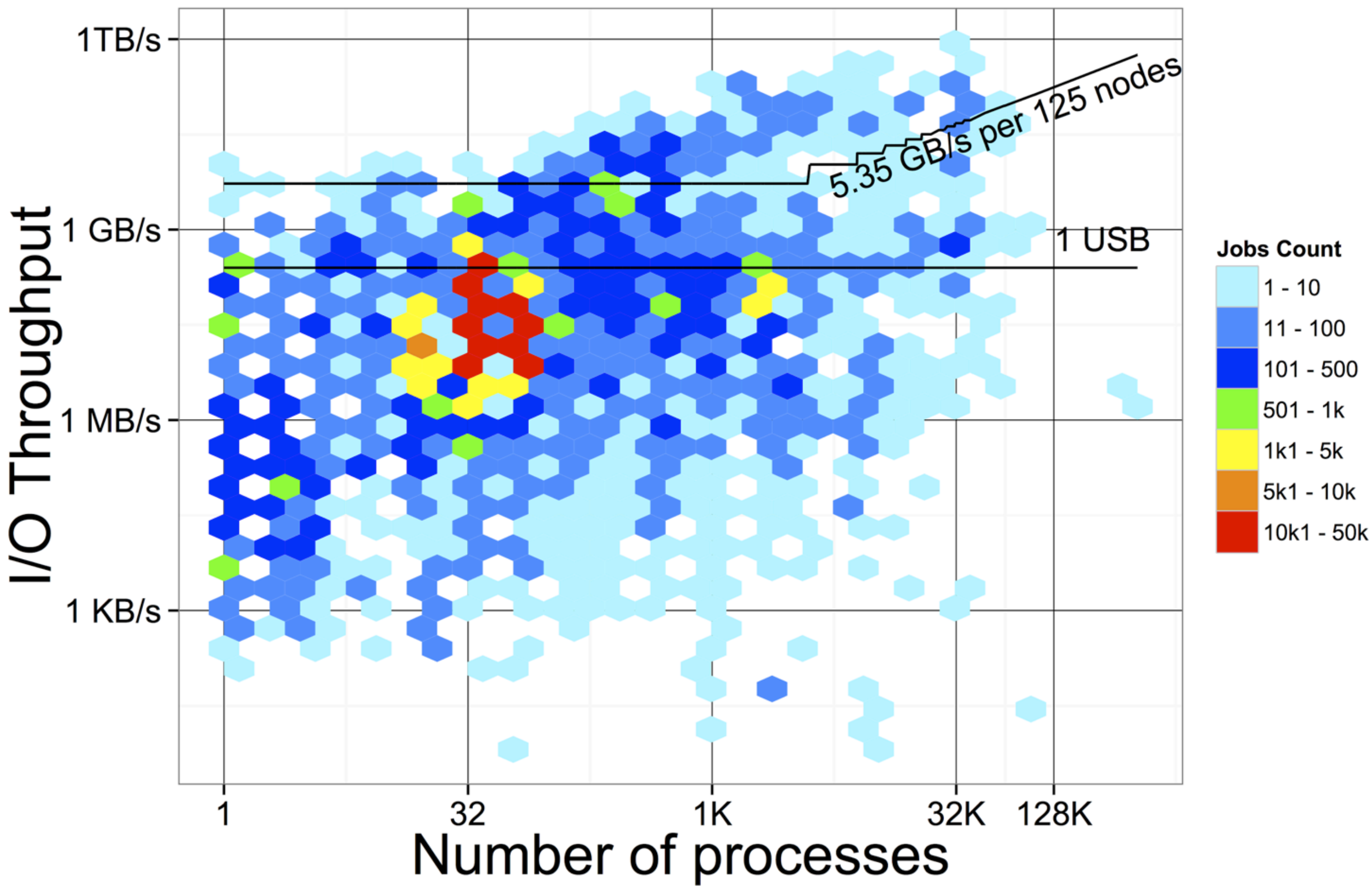


PARALLEL@ILLINOIS

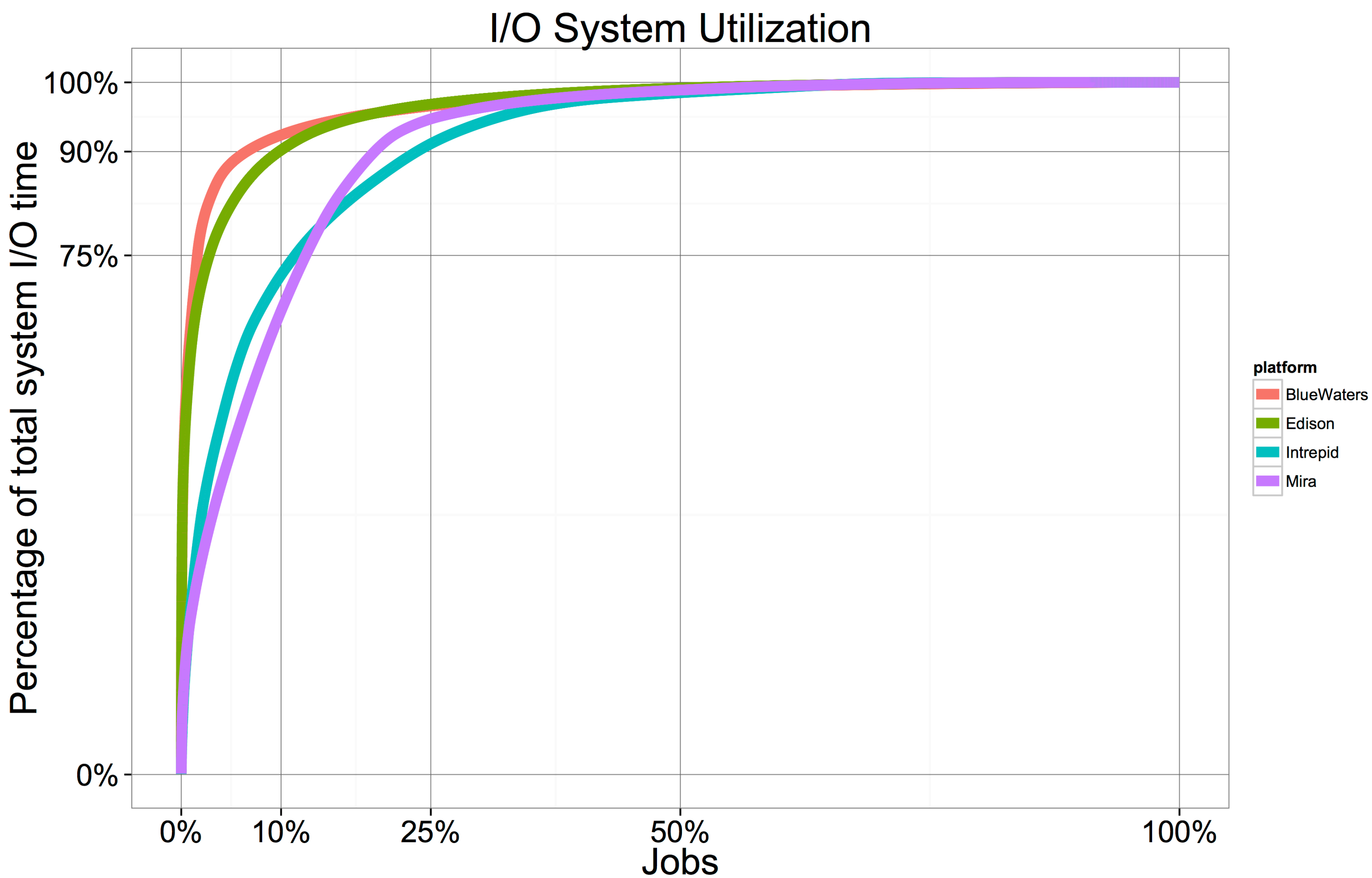
Most Jobs Read/Write Little Data (Blue Waters data)



I/O Thruput vs Relative Peak



I/O Time Usage Is Dominated By A Small Number Of Jobs/Apps



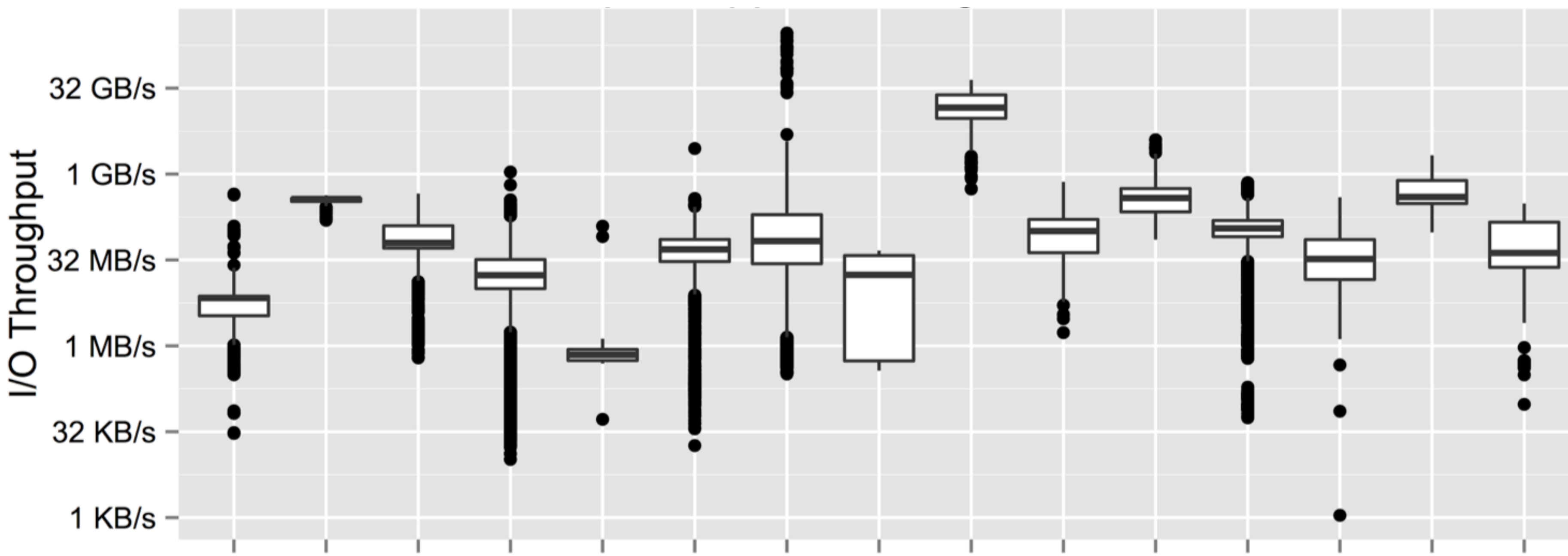
PARALLEL@ILLINOIS

Improving the performance of the top 15 apps can save a lot of I/O time

	Platform I/O time percent	Percent of platform I/O time saved if min thruput = 1 GB/s
Mira	83%	32%
Intrepid	73%	31%
Edison	70%	60%
Blue Waters	75%	63%

Top 15 apps with largest I/O time (Blue Waters)

- Consumed 1500 hours of I/O time (75% total system I/O time)



PARALLEL@ILLINOIS

What Are Some of the Problems?

- POSIX I/O has a strong consistency model
  - Hard to cache effectively
  - Applications need to transfer block-aligned and sized data to achieve performance
  - Complexity adds to fragility of file system, the major cause of failures on large scale HPC systems
- Files as I/O objects add metadata "choke points"
  - Serialize operations, even with "independent" files
  - Do you know about O_NOATIME ?
- Burst buffers will *not* fix these problems – must change the semantics of the operations
- "Big Data" file systems have very different consistency models and metadata structures, designed for their application needs
  - Why doesn't HPC?
    - There have been some efforts, such as PVFS, but the **requirement** for POSIX has **held up** progress

Remember

- POSIX is not just "open, close, read, and write" (and seek …)
  - That's (mostly) syntax
- POSIX includes strong semantics if there are concurrent accesses
  - Even if such accesses never occur
- POSIX also requires consistent metadata
  - Access and update times, size, …

No Science Application Code Needs POSIX I/O

- Many are single reader or single writer
  - Eventual consistency is fine
- Some are disjoint reader or writer
  - Eventual consistency is fine, but must handle non-block-aligned writes
- Some applications use the file system as a simple data base
  - Use a data base – we know how to make these fast and reliable
- Some applications use the file system to implement interprocess mutex
  - Use a mutex service – even MPI point-to-point
- A few use the file system as a bulletin board
  - May be better off using RDMA
  - Only need release or eventual consistency
- *Correct* Fortran codes do not require POSIX
  - Standard requires unique open, enabling correct and aggressive client and/or server-side caching
- MPI-IO would be better off without POSIX

Part 2: What Can We Do About it?

- Short run
  - What can we do now?
- Long run
  - How can we fix the problem?

Short Run

- Diagnose
  - Case study. Code "P"
- Avoid serialization (really!)
  - Reflects experience with bugs in file systems, including claiming to be POSIX but not providing correct POSIX semantics
- Avoid cache problems
  - Large block ops; aligned data
- Avoid metadata update problems
  - Limit number of processes updating information about files, even implicitly

Case Study

- Code P:
  - ♦ Logically Cartesian mesh
  - ♦ Reads ~1.2GB grid file
    - Takes about 90 minutes!
  - ♦ Writes similar sized files for time steps
    - Only takes a few minutes (each)!
- System I/O Bandwidth is ~ 1TB/s peak; ~5 GB/sec per (groups of 125) nodes

Serialized Reads

- "Sometime in the past only this worked"
  - File systems buggy (POSIX makes system complex)
- Quick fix: allow 128 concurrent reads
  - One line fix (if (mod(i,128) == 0)) in front of Barrier
  - About 10x improvement in performance
    - Takes about 10 minutes to read file

What’s Really Wrong?

- Single grid file (in easy-to-use, canonical order) requires each process to read multiple short sections from file
- I/O system reads large blocks; only a small amount of each can be used when each process reads just its own block
  - ♦ For high performance, must read and use entire blocks
  - ♦ Can do this by having different processes read blocks, then shuffle data to the processes that need it
- Easy to accomplish using a few lines of MPI (MPI_File_set_view, MPI_File_read_all)

Fixing Code P

- Developed simple API for reading arbitrary blocks within an n-D mesh
  - ♦ 3D tested; expected use case
  - ♦ Can position beginning of n-D mesh anywhere in file
- Now ~3 seconds to read file
  - ♦ 1800x faster than original code
  - ♦ Sounds good, but is still <1GB/s
  - ♦ Similar test on BG/Q 200x faster
- Writes of time steps now the top problem
  - ♦ Somewhat faster by default (caching by file system is slightly easier)
  - ♦ Roughly 10 minutes/timestep
  - ♦ MPI_File_write_all should have similar benefit as read

Long Run

- Rethink I/O API, especially semantics
  - May keep open/read/write/close, but add API to select more appropriate semantics
    - Maintains correctness for legacy codes
    - Can add improved APIs for new codes
    - New architectures (e.g., "burst buffers") unlikely to implement POSIX semantics

Final Thoughts

- Users often unaware of how poor their I/O performance is
  - They've come to expect awful
- Collective I/O can provide acceptable performance
  - Single file approach often most convenient for workflow; works with arbitrary process count
- Single file per process can work
  - But at large scale, metadata operations can limit performance
- Antiquated HPC file system semantics make systems fragile and perform poorly
  - Past time to reconsider in requirements; should look at "big data" alternatives

Thanks!

- Especially Huong Luu, Babak Behzad
- Code P I/O: Ed Karrels
- Funding from:
  - NSF
  - Blue Waters
- Partners at ANL, LBNL; DOE funding