# Lecture 31: Introduction to Parallel I/O

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

# I/O and File Systems

- Most applications need persistent storage
  - ♦ Typical to store persistent data in *files*, accessed through input/output (I/O) features of programming language and runtime
  - ♦ Dominant implementation for persistent storage is magnetic disks
    - Tape also used for higher capacity
    - Semiconductor and other technologies used for higher performance/lower power (e.g., FLASH)

# Performance Parameters

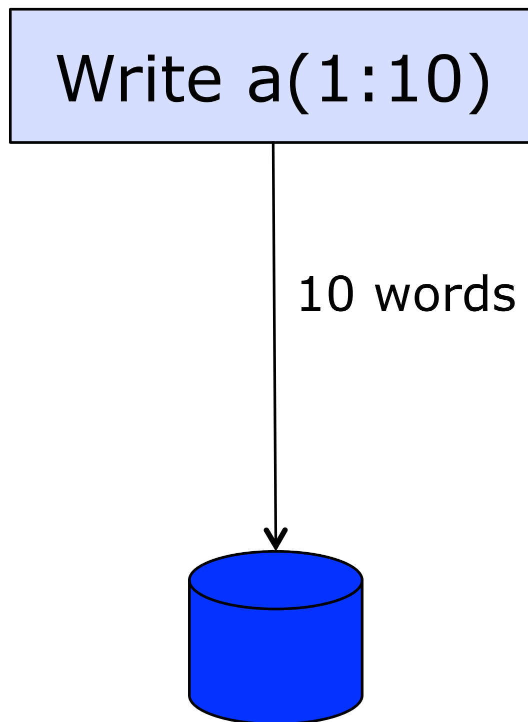
- Magnetic disks performance
  - ♦ Latency 2-10 ms (time it take the disk to spin under the read/write head)
    - 1,000x slower than internode communication
    - 10,000,000x slower than processor core
  - ♦ Bandwidth over 100MB/sec
    - But only approached for large transfers
- Performance sensitive to exact usage pattern
- Most I/O benchmarks are not representative of user I/O patterns
  - ♦ At most, useful to set a performance goal
- Most performance solutions aggregate operations in some way

# Files and File Systems

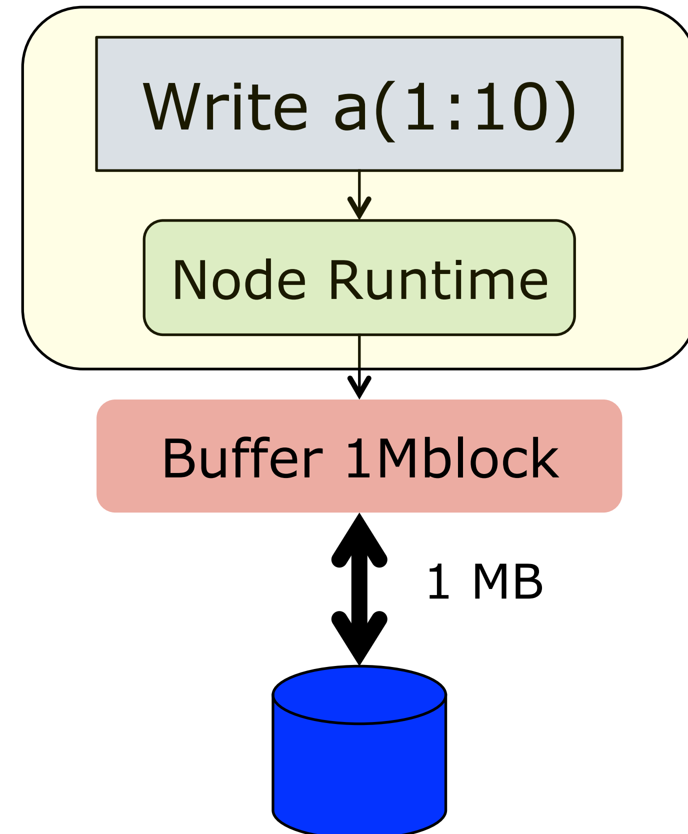
- A file is just an ordered collection of bytes
- A file system manages collections of files and properties of the files
  - Size
  - Access restrictions
  - Quotas
  - Reading and writing data
- File systems differ in the services they provide and the semantics of data access and update

# Two Abstract I/O Models

- Typical user view




- A more accurate model



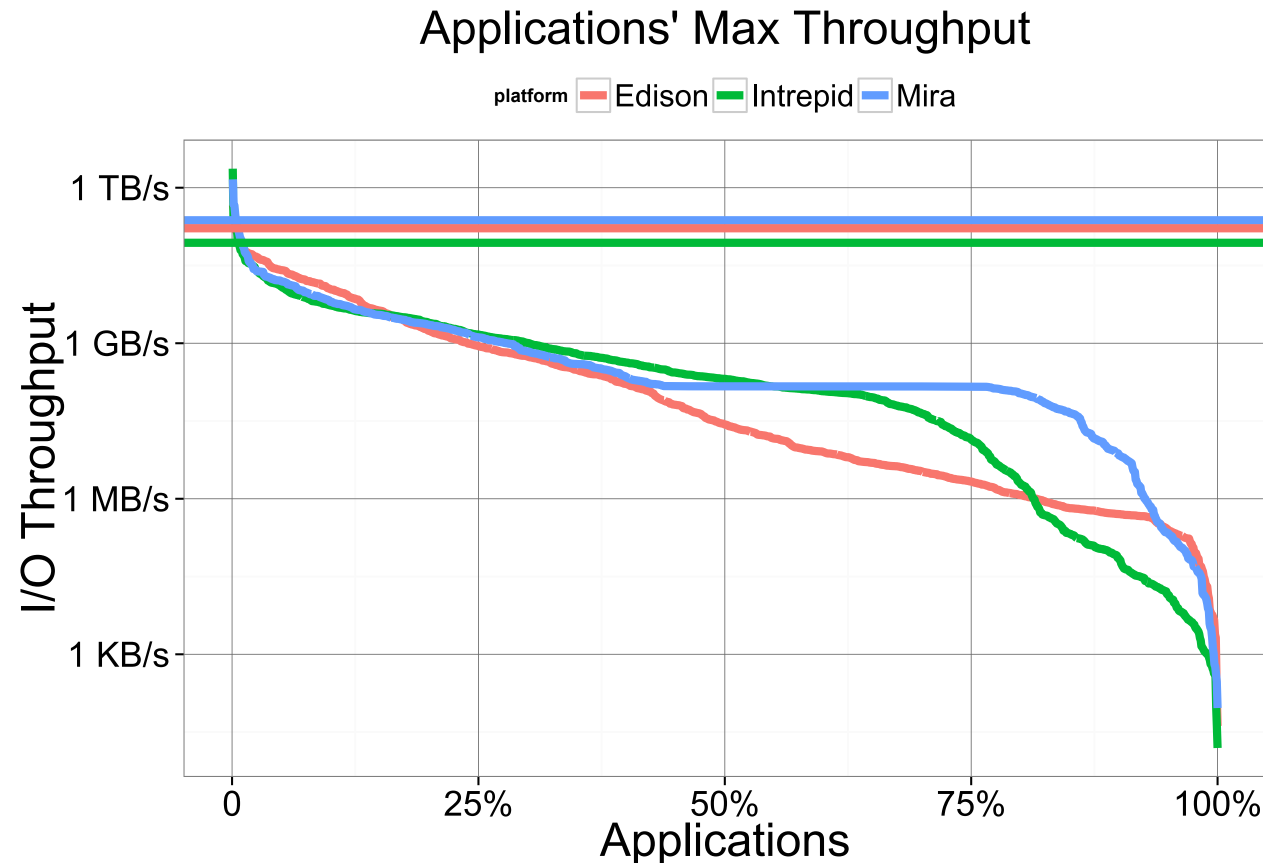
# Notes on the Abstract I/O Models

- Leaves out caching
  - Client (node) side can be hard (more later)
- Leaves out *metadata* operations
  - Does user *still* have access to the file?
  - What is the date of the last access or change?
  - How big is the file?
  - Where is the file located?

# Understanding Achieved File System Performance

- Hardware operations don't match programmer model
  - Large block reads
  - Writes are read-modify-write
- Semantics don't match most programmer's model
  - POSIX semantics for read/write to a file is similar to *sequential consistency*
  - Makes it extremely hard to implement client-side caching, metadata updates
- How much does this impact performance?

# Few Applications Get Even 1% of I/O Performance



**A Multiplatform Study of I/O Behavior on Petascale Supercomputers,** Luu, Winslett, Gropp, Ross, Carns, Harms, Prabhat, Byna, Yao. HPDC'15, to appear

PARALLEL@ILLINOIS

# POSIX Read/Write Semantics

- Once a write completes (at any process), any read, *from any other process*, must see that write
- Basic operations have requirements that are often not understood and can impact performance
  - In other words, few applications in scientific computing require this strong ordering of read and write operations between processes, but it impacts the performance and stability of the file system for everyone

# POSIX Read and Write

- Read and Write are atomic
- No assumption on the number of processes (or their relationship to each other) that have a file open for reading and writing
- Consider this case
- Process 1<br>read a<br>…<br>read b

  Process 2<br>…<br>write b

PARALLEL@ILLINOIS

# POSIX Read and Write

- Reading a large block containing both a and b (Caching data) and using that data to perform the second read without going back to the original file is ***incorrect***
- This requirement of read/write results in the over specification of the interface in many applications codes
  - Few applications require strong synchronization of read/write
- There are similar mismatches with other operations…

# Open

- User's model is that this gets a file descriptor and (perhaps) initializes local buffering
- Should be fast and (more importantly) scalable operation
- Problem: no Unix (or POSIX) interface for "exclusive access open"
  - File system must assume that other processes will open the file
  - Since the file system doesn't know anything about a parallel program, there's no case for when a unique job (rather than process) opens the file

# Close

- User's model is that this flushes the last data written to disk (if they think about that) and relinquishes the file descriptor
- When is data written out to disk?
  - On close?
  - Sometime later?
  - Never?
- Example:
  - Unused physical memory pages used as disk cache (at the server)
  - Combined with Uninterruptible Power Supply, data may ***never*** appear on disk
  - Makes it extremely hard to write a robust I/O benchmark

# Seek

- User's model is that this assigns the given location to a variable and takes about 1 nanosecond
- Changes position in file for "next" read
- May interact with implementation to cause data to flush data to disk (clear all caches)
  - In that case very expensive, particularly when multiple processes are seeking into the same file

# Read/Fread

- Users expect read (unbuffered) to be faster than fread (buffered)  (rule: buffering is bad, particularly when done by the user)
  - Reverse true for short data (often by several orders of magnitude)
  - User thinks reason is “System calls are expensive”
  - Real culprit is atomic nature of read
- Fortran since F77 requires *unique open* (Section 12.3.2, lines 44-45)
  - Fortran spec designed for performance
  - Still doesn’t help for parallel programs

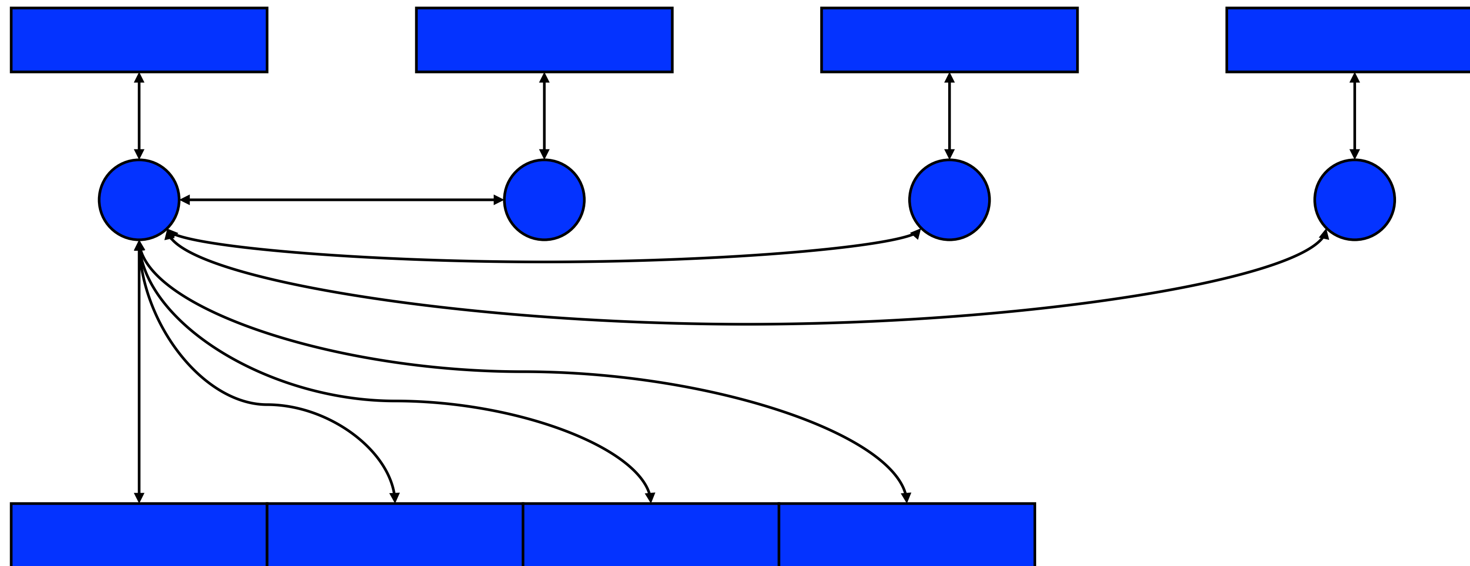
# Write

- Users expect write to simply update that part of the file
  - ♦ Often expect buffering (fwrite) even with write, or caching by the runtime
- Users expect to get good performance with large writes
  - ♦ E.g., write more than 1MB
  - ♦ But efficient writes must be *aligned* with hardware/file system blocks
    - Unaligned writes can be much slower
    - May use general data path instead of special write-full-block path

# What Does This Mean For I/O From Parallel Programs?

- Many ways to organize I/O
  - ♦ Should be considered in the context of the entire application workflow, not just one program
- Several natural choices
  - ♦ One file per program
    - May match workflow, other tools
  - ♦ One file per process
    - Avoids performance and correctness bugs in the File system
  - ♦ One file per node/row/rack/…
    - Workarounds for performance and correctness issues

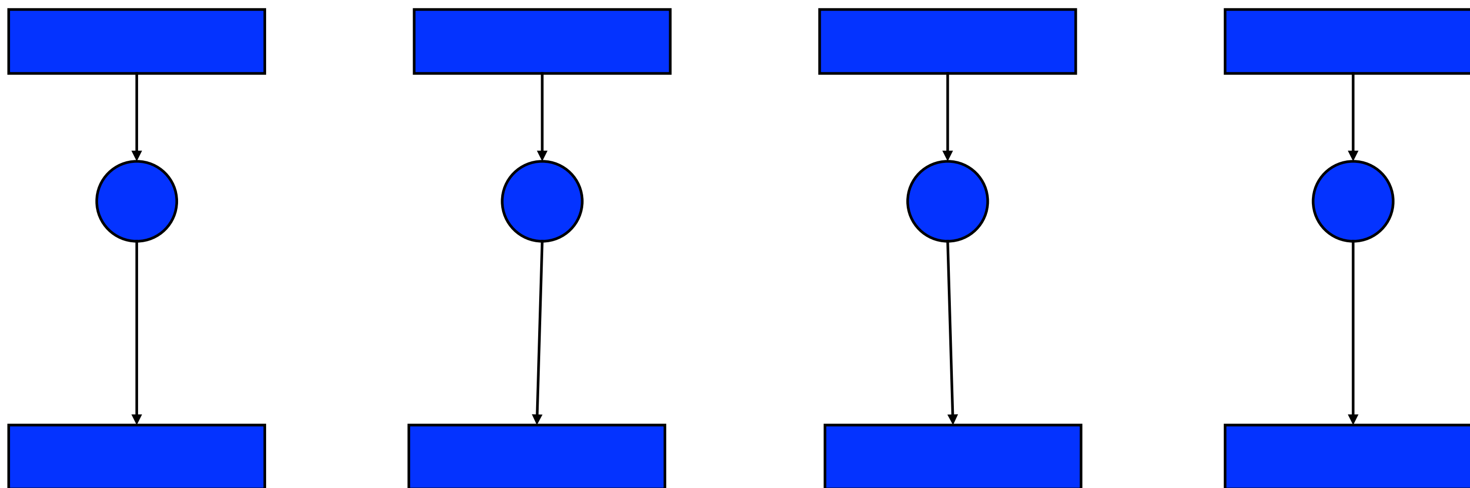
# Non-Parallel I/O



- Non-parallel
- Performance worse than sequential
- Often legacy from before application was parallelized

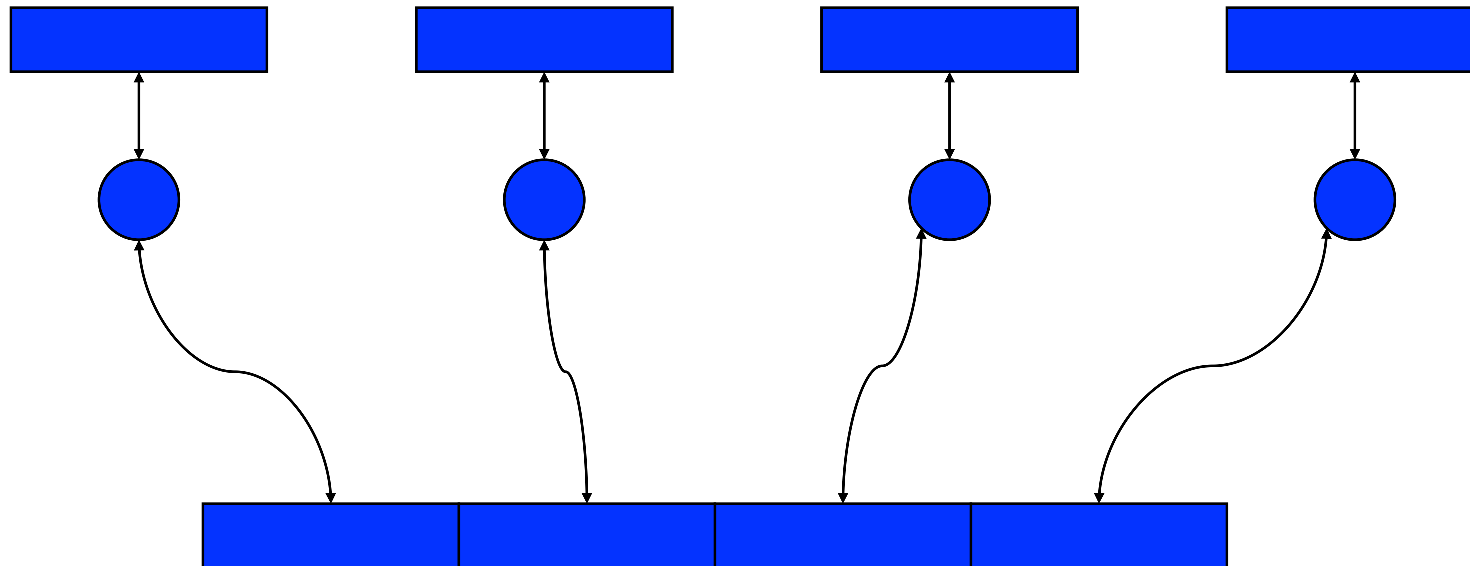
# Independent Parallel I/O

- Each process writes to a separate file



- Pro: parallelism
- Con: lots of small files to manage
- Either legacy from before MPI or done to address file system issues

# Parallel I/O – Single File



- Parallel
- Performance can be great, good, bad, or terrible (even worse than sequential)
- Depends on correct implementation of concurrent updates in file (all too rare)

# Other Approaches

- File systems with different consistency semantics
  - NFS (esp version 3) – essentially no consistency; safe for serial access only
  - PVFS – defines non-overlapping writes
  - HDFS – Parallel access to immutable files
- Other models of persistent data objects besides files
  - Databases
  - Object stores

# Other Approaches: Hardware

- The large performance gap between disks and memory makes performance difficult
- New memory designs offering intermediate performance and cost
  - Non-volatile RAM (NVRAM)
  - Burst buffers (smooth out I/O performance demands with buffer to absorb writes)
    - Note useful only if POSIX semantics abandoned

# Asking the Right Question

- Do you want Unix or Fortran I/O?
  - Even with a significant performance penalty?
- Do you want to change your program?
  - Even to another portable version with faster performance?
  - Not even for a factor of 40???
- User "requirements" can be misleading

# Readings

- **Darshan**: P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, R. Ross, Understanding and improving computational science storage access through continuous characterization, ACM Trans. on Storage, 7(3):8, 2011
  - Darshan is a tool to observe I/O usage by applications in HPC; current state-of-the-art in gathering data on HPC application I/O behavior