Introduction to Parallel Processing

Lecture 16 : Parallel I/O

10/19/2022

Professor Amanda Bienz

File Systems

- Typically, applications rely on persistent storage, aka non-volatile memory : storage that retains data after power to device is shut off
  - Think about data in main memory of program… if I shut off my computer in the middle of the program, this data is lost
  - Example of persistent storage usage :
    - SpMV - the sparse matrix downloaded from Suitesparse collection was stored on disk

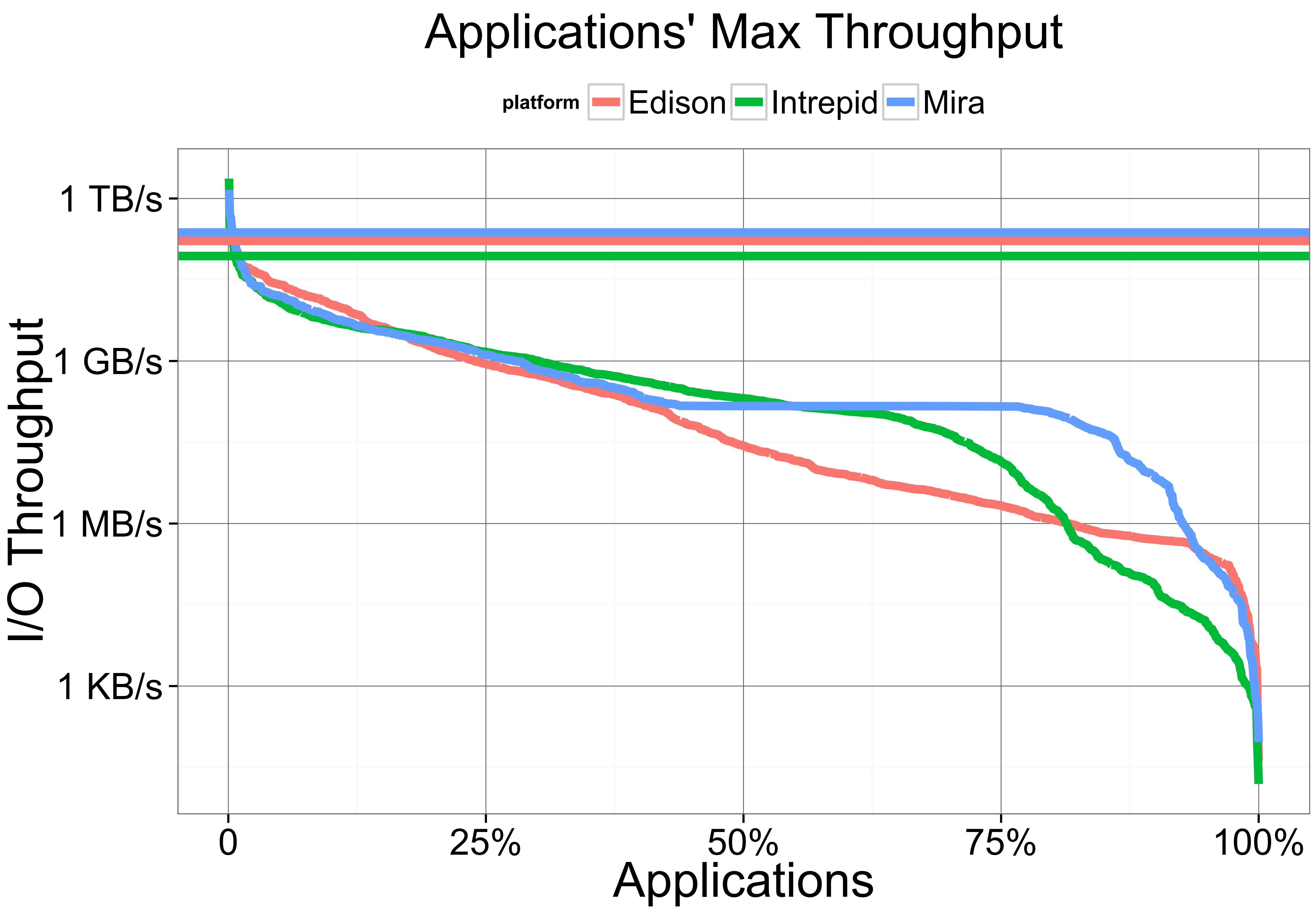
Non-Volatile Memory

- Magnetic discs are the traditional storage unit (hard disk drives, floppy disks, magnetic tape)
- There are other options for non-volatile memory
  - FLASH memory : solid state drives
  - ROM : read-only memory used for storing software that will rarely change during life of system (e.g. video games)
  - Ferroelectric RAM : random-access memory similar to DRAM but non-volatile (lower power, faster write, much faster read than FLASH)

Performance Expectations

- Magnetic disk performance (most commonly used)
    - Latency is 2-10ms
        - 1000x slower than inter-node communication!
    - Bandwidth over 100MB/sec, but only for very large transfers
- Actual user performance is often much worse, because the performance of I/O is very sensitive to the usage pattern
    - Benchmarks will look at ideal usage patterns rather than those representative of most users

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

POSIX Reads and Writes

- POSIX : Portable Operating System Interface : standards by IEEE to guarantee operating system compatibility
- What does this say about reading and writing?
  - Once a write is completed by any process, any read from any other process must see this write
  - Is this necessary for your program? Probably not… you probably will not have process 2 write something that process 0 then needs to read
  - But, this requirement will impact your performance

Open

- User gets a file description and can initialize local buffering
- Ideally, this should be very fast (and scalable)
- Problem : no interface for an exclusive access open in Unix/POSIX
  - The file system must assume other processes will access the file
  - Can't even assume that only one parallel program will access the file

Close

- User flushes data written to disk and frees the file descriptor
- But, when is the data actually written to disk?
    - This question makes benchmarks extremely difficult

Seek

- Assigns the given location to a variable, should take about 1 nanosecond
- Changes the position in the file for the next read
- Unfortunately, implementation may cause a flush of data to disk
  - Very expensive, especially when multiple processes are seeking into same file

Read or Fread

- Read is unbuffered while fread is buffered, so would expect read to be faster
    - For short data, fread is actually faster, often by several orders of magnitude, because reads are often atomic

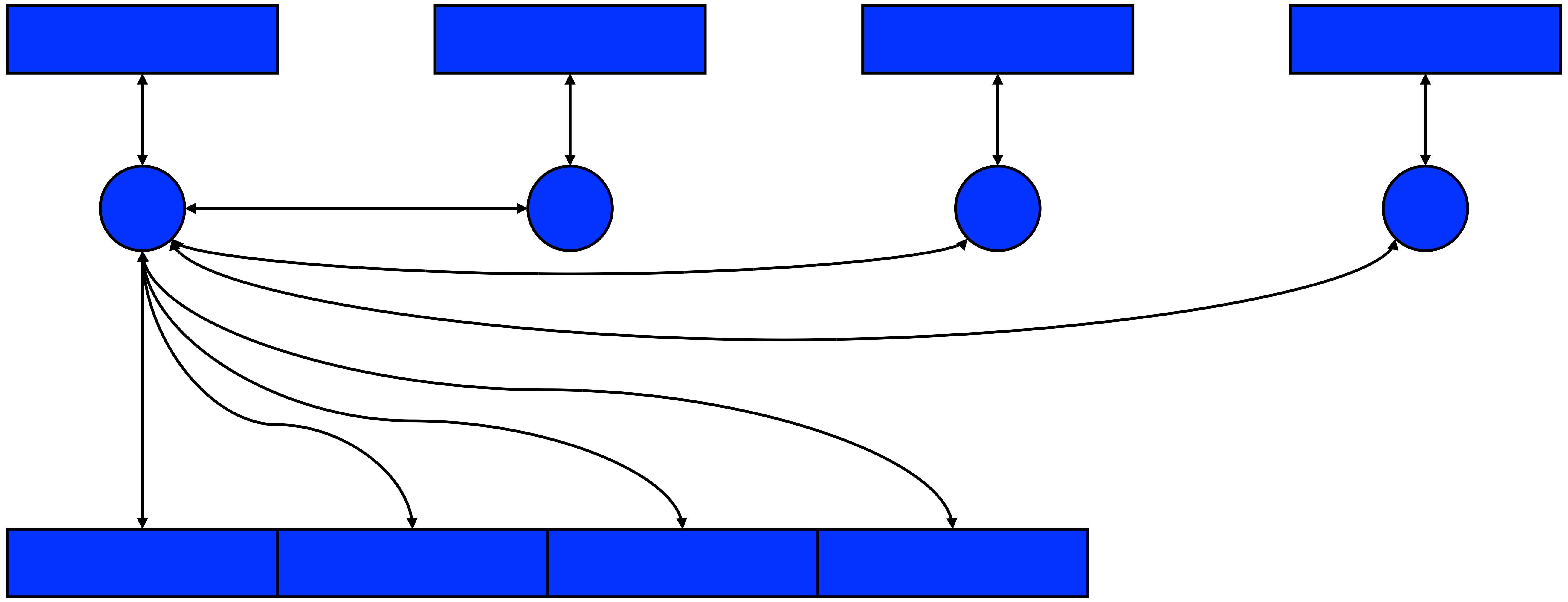
Write

- Would like to simply update part of file
- Expect to get good performance with large writes (> 1MB)
    - However, efficient writes must be aligned with hardware and file system blocks
        - Unaligned writes are much slower

What does this mean for Parallel I/O?

- There are many ways to organize I/O
- Some choices:
  - One file per program : may match workflow
  - One file per process : avoids performance and correctness bugs
  - One file per node / row / rack / etc : can help with performance and correctness bugs

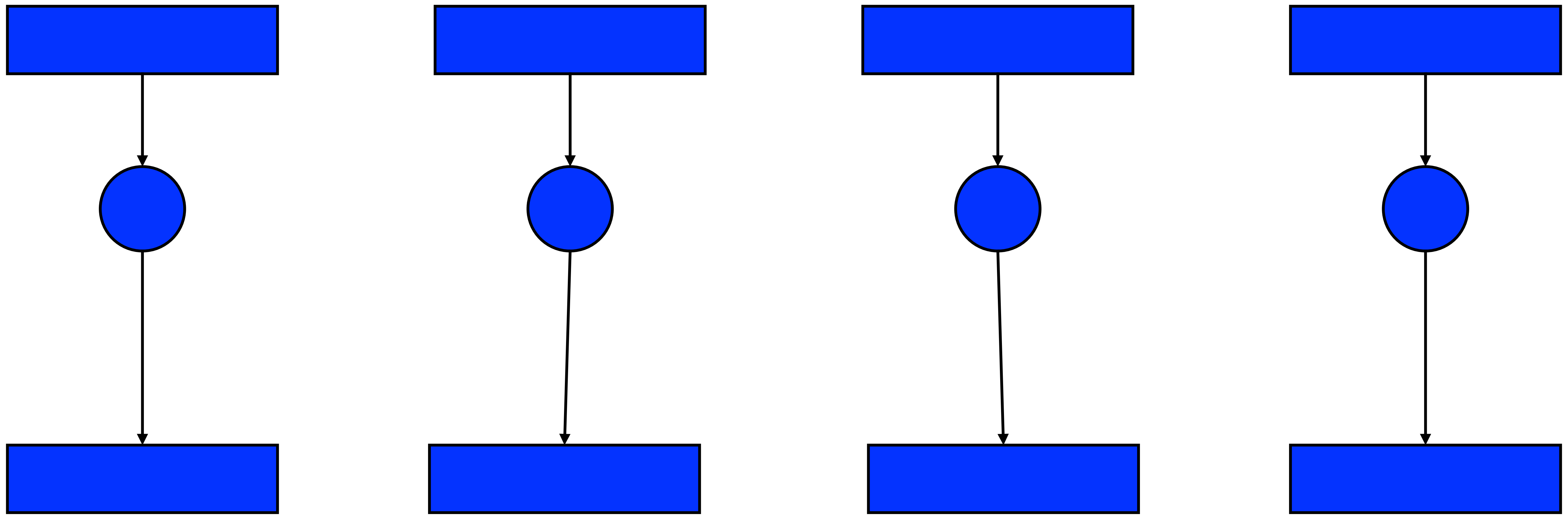
Non-Parallel I/O



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

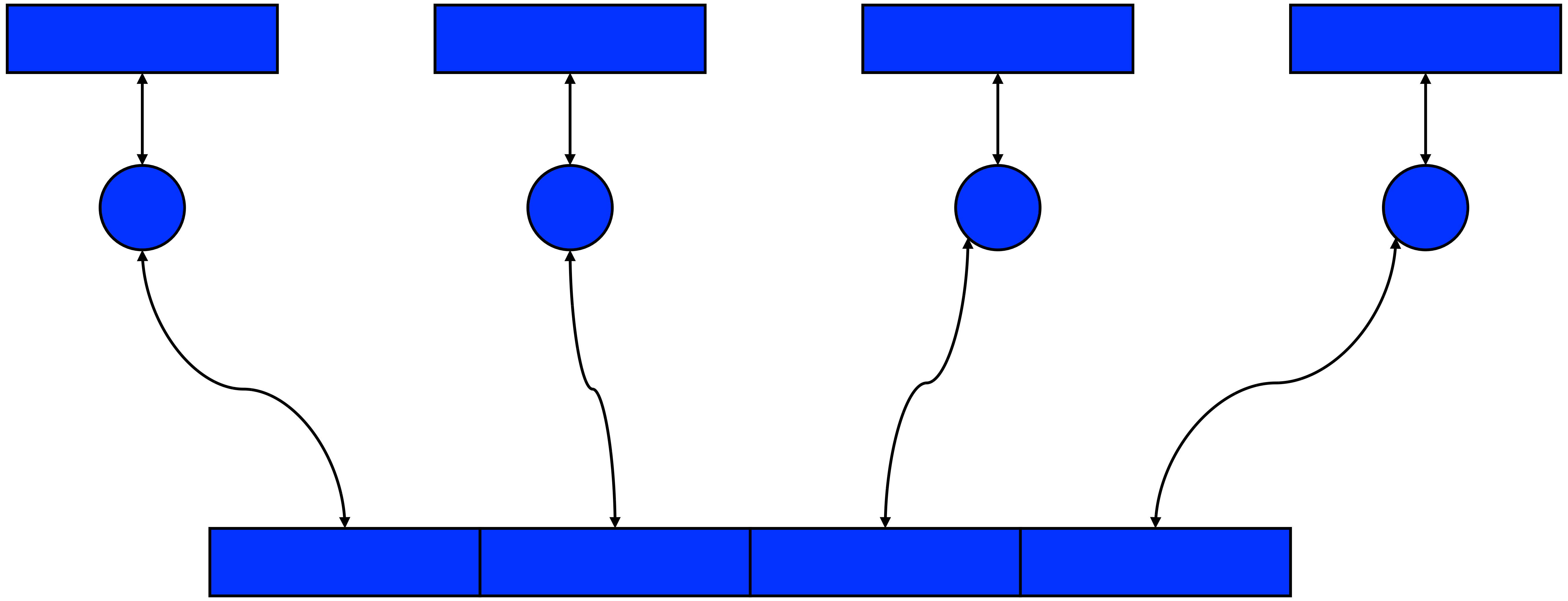
Independent Parallel I/O

- 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)