Getting the Most out of Scientific Computing Resources

Or, How to Get to Your Science Goals Faster

Chip Watson

Scientific Computing

Optimizing Batch Throughput

If you intend to run hundreds or thousands of jobs at a time, there are a couple of things to pay attention to:

1. Memory footprint
2. Disk I/O

Memory Footprint

Our batch nodes are fairly memory lean, from 0.6 to 0.9 GB per hyper-thread for most of the compute nodes, and 1.0 GB per hyper-thread for the older nodes. (The largest set of nodes have 32 GB of memory and 24 cores – 48 job slots using hyper-threading).

Consequently, if you submit a serial (one thread) batch job requesting 4 GB of memory, then you are effectively consuming 4-6 job slots on a node. If a lot of jobs like this are running, then clearly this significantly reduces the potential performance of the node – CPU cores are wasted.

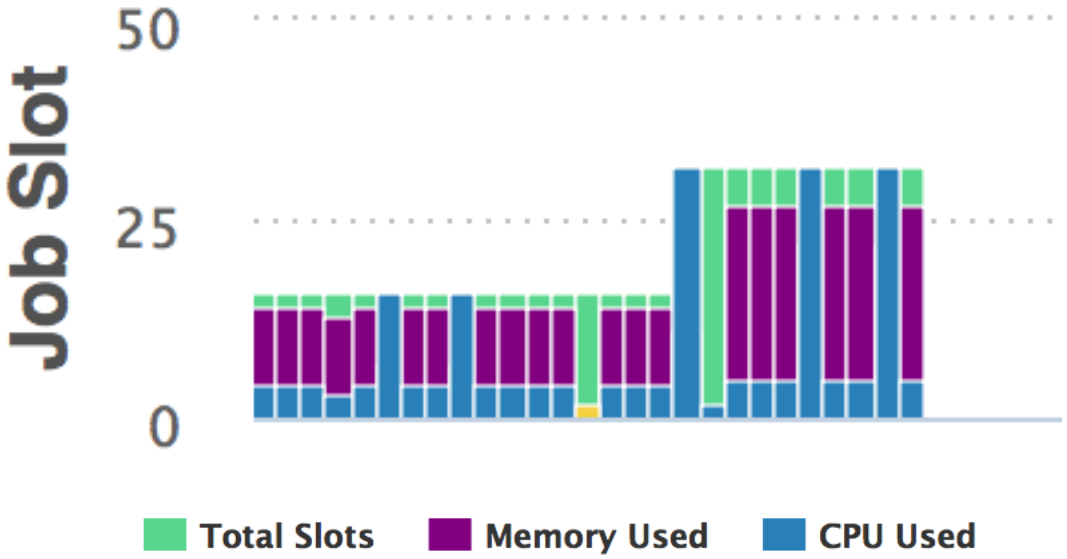
Note: using only half of the slots on a node (i.e. effectively no hyper-threading) delivers about 85% of the total performance of the node, since all physical cores will have one job, and none will be hyper-threading.

What if the job has a data dependent size, with most jobs needing < 1 GB but some needing 4 GB?

Answer: request 4 GB. We actually oversubscribe memory by some factor with this in mind, allowing the node to use virtual memory. The factor is tunable by operations, and is set to minimize page swapping. If one or two hundred 4 GB memory jobs are spread across a hundred nodes, all job slots can still be used, and nothing will be paged out to disk despite the use of virtual memory.

However, thousands of these jobs lowers the performance of the whole farm, since they will tie up physical memory and/or start to cause page-faulting to disk.

Example: CentOS 6.5 nodes



Older, non-threaded code, consuming so much memory that less than 40% of the CPU capacity is accessible.

https://scicomp.jlab.org/scicomp/#/farmNodes

The best way to shrink memory usage is to use multi-threading.

If an application is multi-threaded at the physics event level, then the memory usage typically scales like a constant (for the application size) plus a much smaller constant times the number of threads analyzing events. E.g. for an 8 thread job

memsize = 1.5 GB + 0.5 GB/thread = 5.5 GB for 8 slots

The smaller constant (0.5 in this example) represents the *average* memory use per event rather than the maximum memory use for an event. Running 9 such jobs on our newest nodes would only require 50 GB, and these nodes have 64 GB available => all cores and hyper-threads are fully utilized! Without hyper-threading we could run only ~30 jobs, thus using only 60% of the performance.

Disk I/O (1)

Jefferson Lab has ~2 PB of usable disk for the two batch systems, and ~40% of that is for Experimental Physics.

What is the file systems performance?

There are a bit more than 530 disks in the system. A single disk can move 120-220 MB/s, assuming a single new disk with a dedicated controller and a fast computer. Aggregating disks into RAID sets of 8+2 disks with two of those sets (20 disks) per controller lowers this to more like 450-1250 per controller depending upon the generation of the disks and controllers. We have 27 controllers in 20 servers, and if everything could be kept busy we might have 20 GB/s in bandwidth (read+write).

Real World Observed: much lower.

Bandwidth Constraints

- Disks spin at a constant speed (7200 rpm, 120 r/sec, 8.3ms per rotation), and data is organized into multiple platters, which are divided into cylinders, which are divided into sectors. On the outer cylinders there is room for more sectors, thus on the inner cylinders, the transfer rates are 2x-3x slower than the maximum.
- If you stop reading and restart in another location on the disk, on average you have to wait 4.1 ms for the correct sector to reach the head (rotation), plus several 4ms for the head to move to the correct sector.

Conclusion: 120 I/O operations per sec or IOPS (with qualifications)

Disk I/O (3)

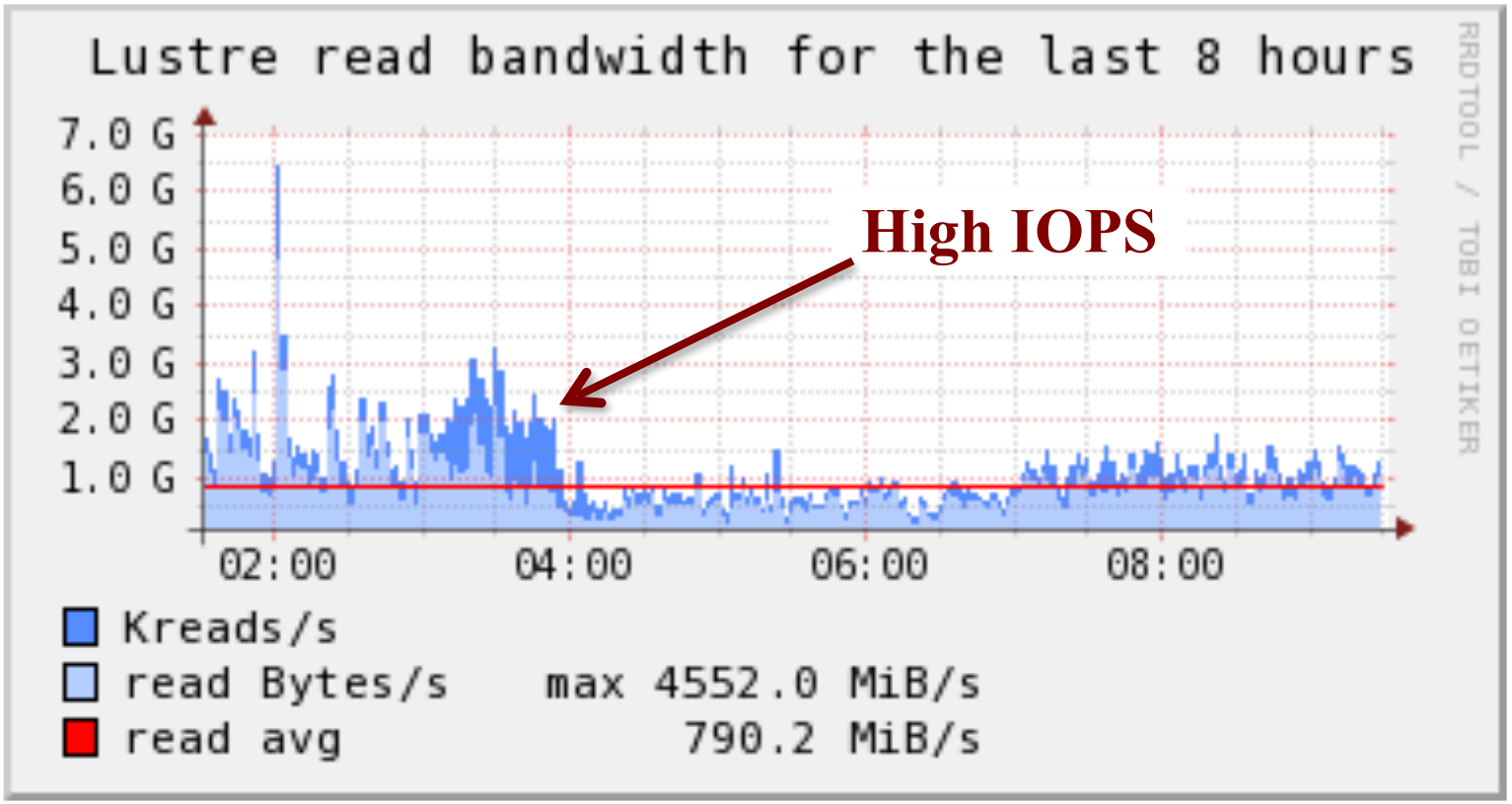
- Single disk, read 1MB, then go elsewhere on the disk; repeat
  - Time to transfer data = 1MB/150MB/s = 6.7 ms
  - Time to seek to data = 4ms (rotation) + 4ms (head) = 8 ms
  - Effective transfer speed: 1/.0147 = 68 MB/s
- RAID 8+2 disks, read 4MB, then go elsewhere; repeat
  - Time to transfer data = 512 KB / 150 MB/s = 3.4 ms (each disk in parallel)
  - Time to seek to data = 4ms (rotate) + 4ms (head) > 8 ms (one will be more)
  - Effective transfer speed: 4/.0114 MB/s ~ 350MB / s
- RAID 8+2 disks, read 32KB, repeat:
  - Time to transfer data = 4 KB / 150 MB/s = 0.0267ms (each disk)
  - Time to seek to data = 4ms (rotation) + 4ms (head) ~ 8ms
  - Effective transfer speed: .032/.0082 < 4 MB / s

Small random I/O reduces bandwidth by 80x !!!

Conclusion

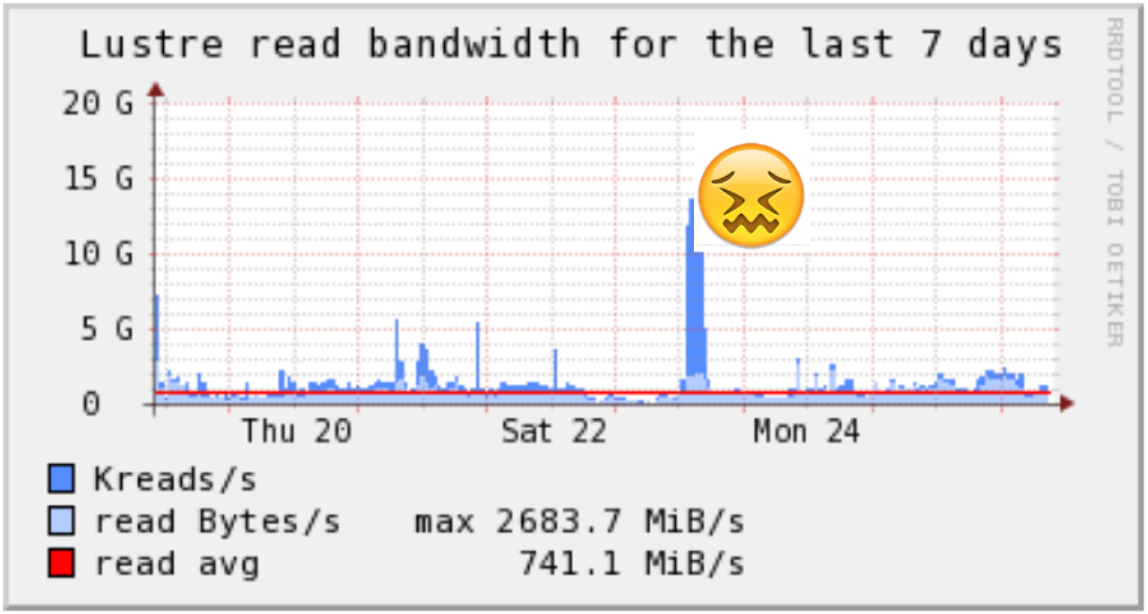
- Always read in chunks of at least 4 Mbytes if you have lots of data to read (e.g. a 1 GB file)
- Impact on your job if you do this correctly: you spend most of your time computing, and very little time waiting for I/O to complete
- Consequence of NOT doing this: you spend most of your time waiting on I/O, and therefore at least wasting memory doing nothing, and if you have a lot of jobs, you are also wasting CPU slots
- One user running 100s of badly behaving jobs lowers the throughput for ALL of the Farm and LQCD!
- WHEN WE CATCH YOU DOING THIS, WE WILL IMPOSE LIMITS ON THE #JOBS YOU CAN RUN!

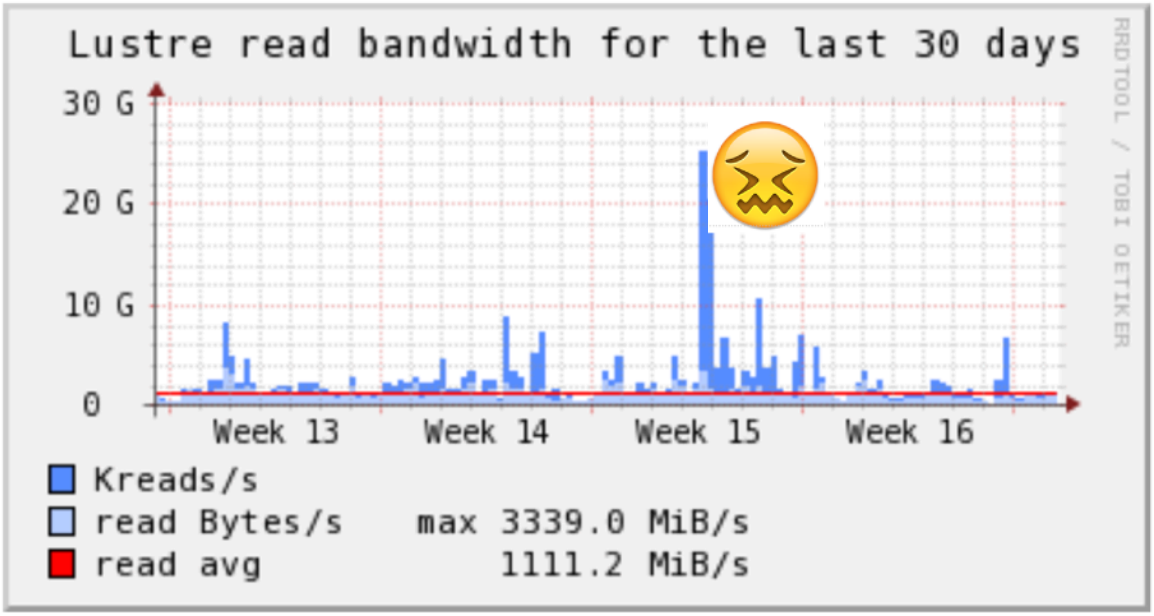
Observing GB/s vs IOPS



- We routinely monitor the stress that high IOPS causes on the system. Dark blue is bad.

Extreme Impact





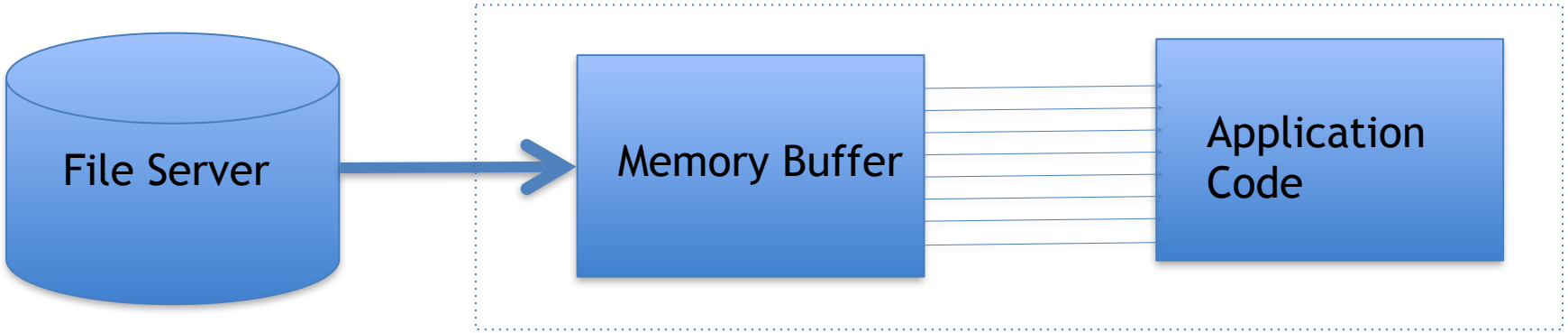
Spikes like those at the left put such a strain on the system that the whole file system becomes non-responsive.

In fact, once in the past month, the entire system had to be rebooted!

What if my I/O only needs 4 Kbyte of data for each event?

Use Buffered I/O. This can be made transparent to the rest of your software

- Buffer software reads 4 MB
- Application reads 4 KB, satisfied from the buffer, which automatically refills as needed



C Example

C++ Example

Java Example