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 Effects (1)

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.

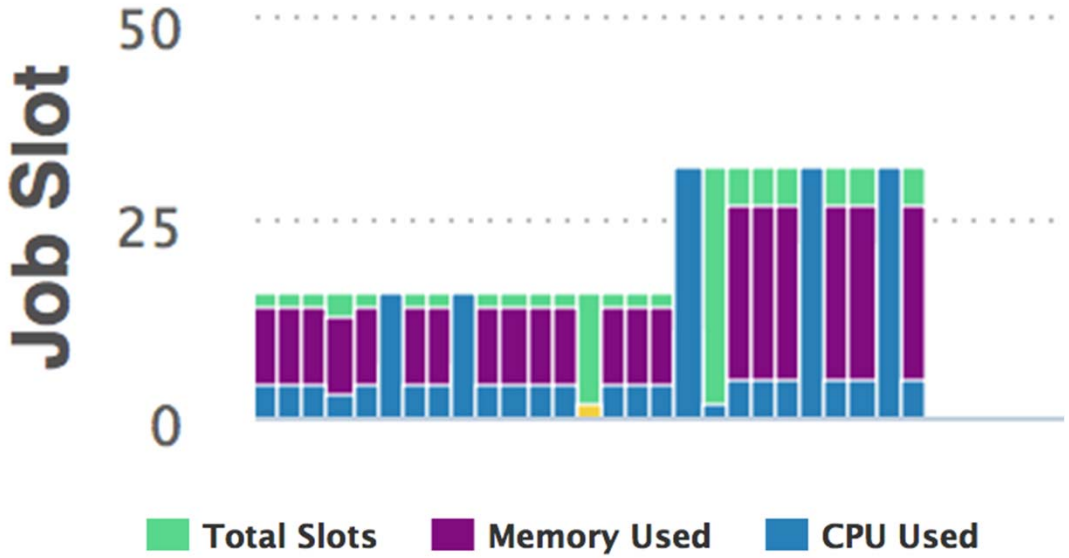
Memory Effects (2)

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

Multi-Threading

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-th

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.

Disk I/O (2)

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)

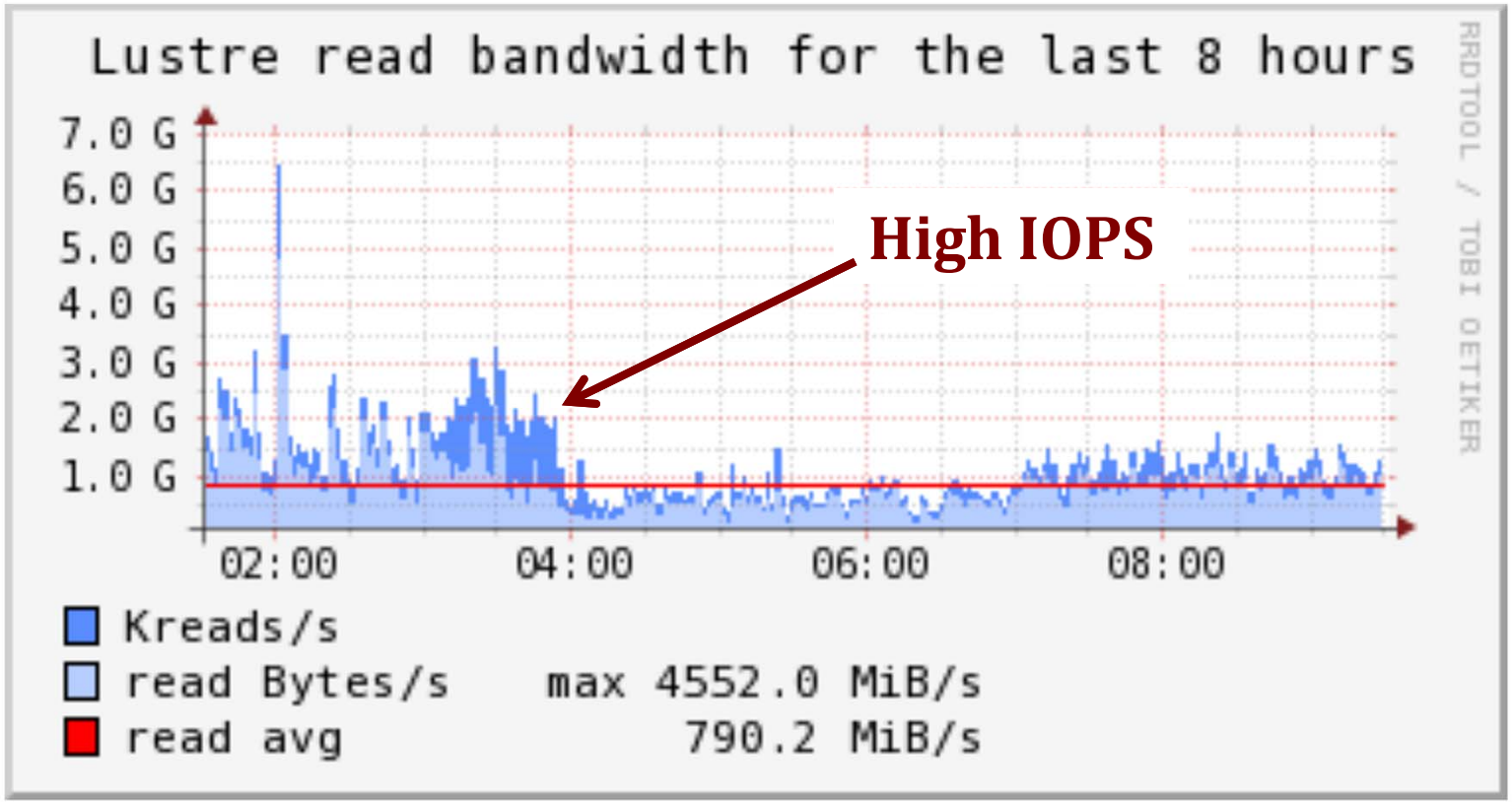
- Single disk, read 1MB, then go elsewhere on the disk; repeat
  - Time to transfer data = 1MB/150MB/s = 6.7 ms
  - Time to seek to data = 4ms (rotation) + 4ms (head) = 8 ms
  - Effective transfer speed: 1/.0147 = 68 MB/s
- RAID 8+2 disks, read 4MB, then go elsewhere; repeat
  - Time to transfer data = 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

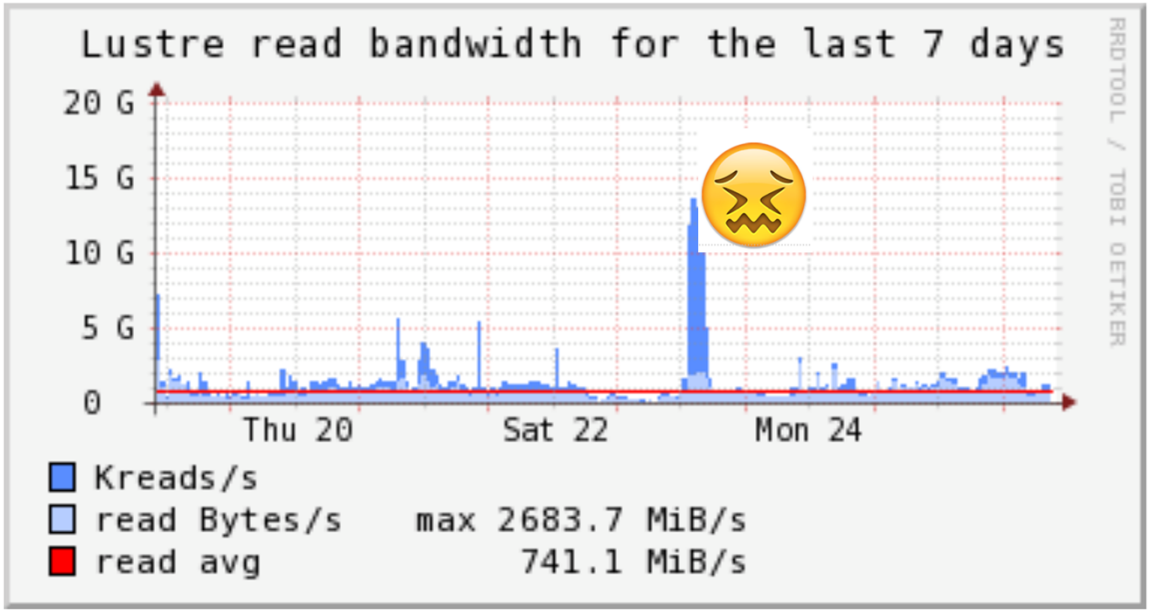
- Always read in chunks of at least 4 Mbytes if you have lots of data to read (e.g. a 1 GB file)
- Impact on your job if you do this correctly: you spend most of your time computing, and very little time waiting for I/O to complete
- Consequence of NOT doing this: you spend most of your time waiting 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!

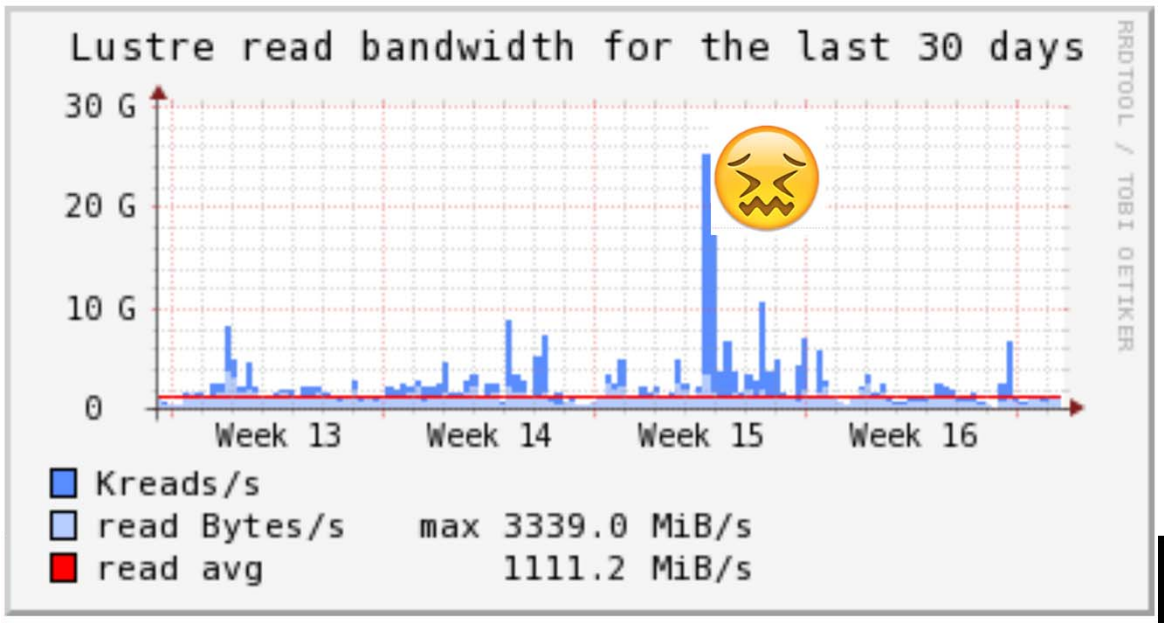
Observing GB/s vs IOPS



- 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.

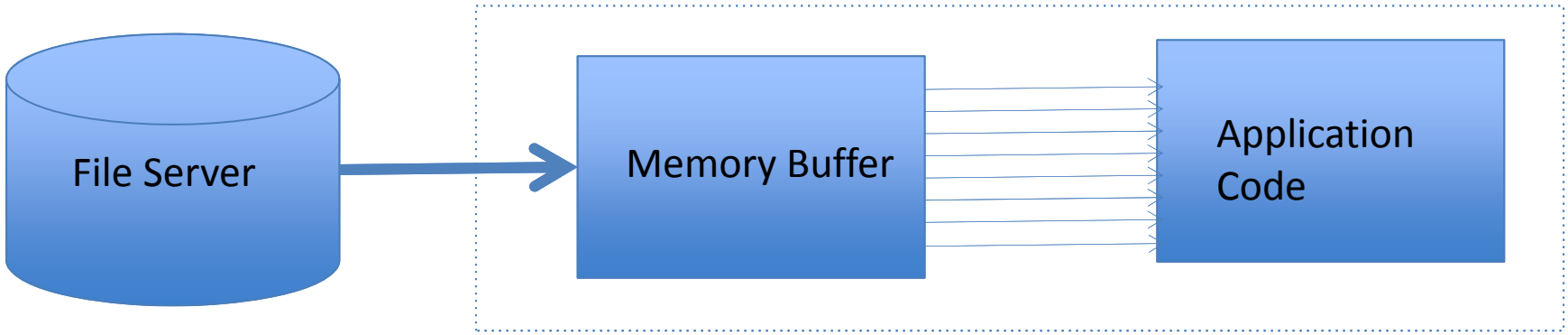
In fact, once in the past month, the entire system had to be rebooted!

How do I minimize this?

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

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 Effects (1)

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 64 GB of memory and 36 cores – 72 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.

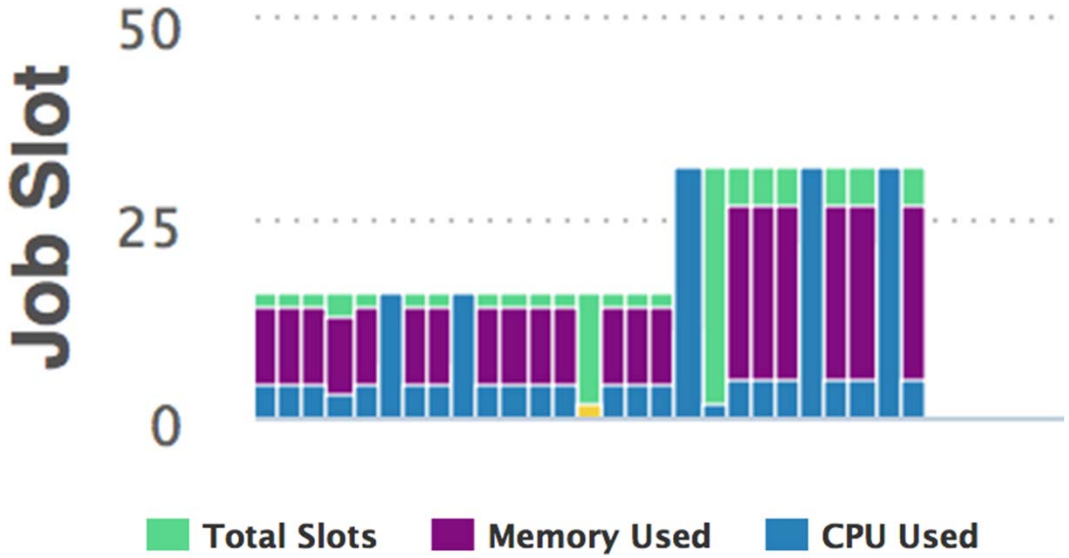
Memory Effects (2)

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

Multi-Threading

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.

Disk I/O (2)

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)

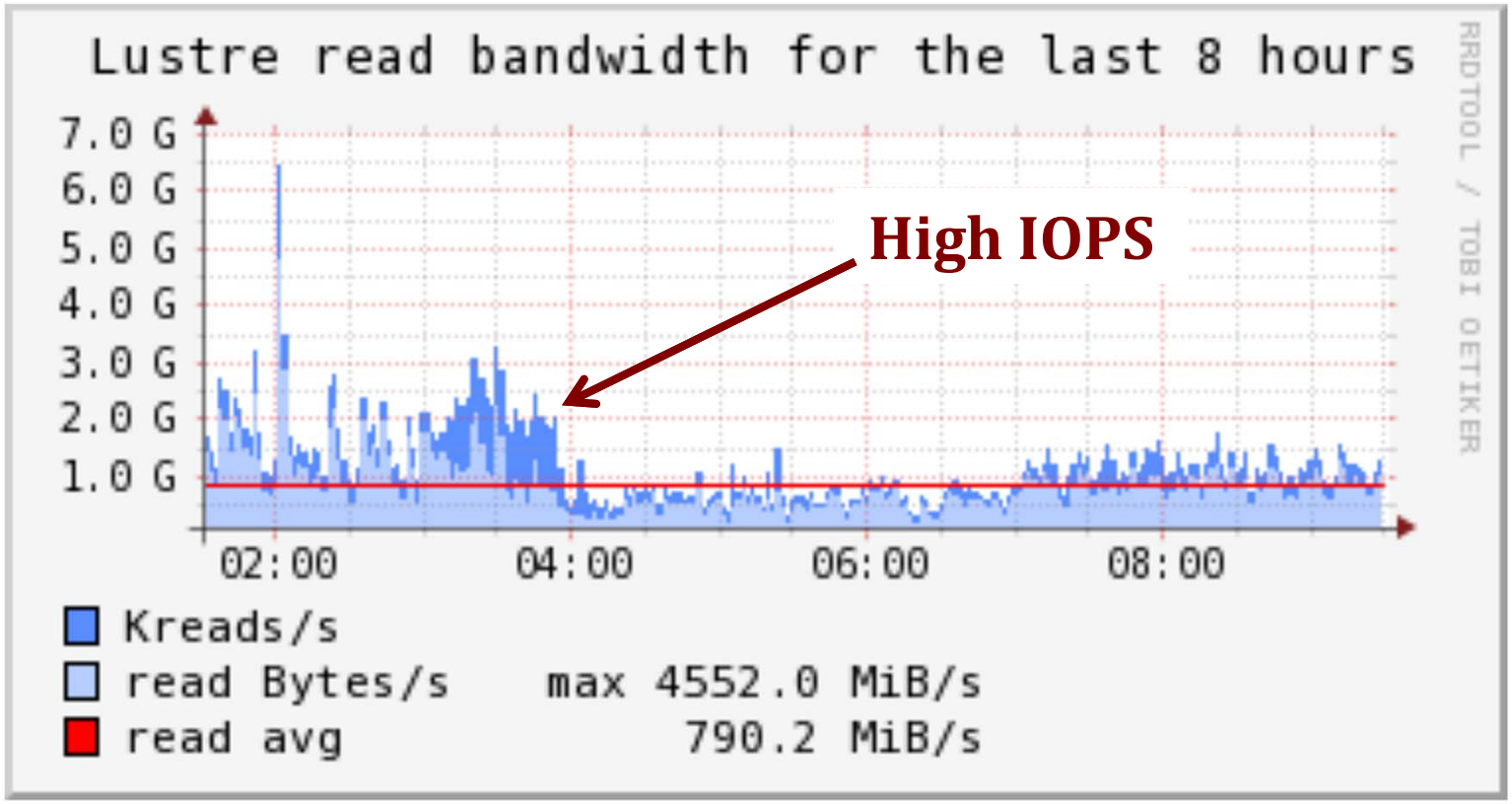
- Single disk, read 1MB, then go elsewhere on the disk; repeat
  - Time to transfer data = 1MB/150MB/s = 6.7 ms
  - Time to seek to data = 4ms (rotation) + 4ms (head) = 8 ms
  - Effective transfer speed: 1/.0147 = 68 MB/s
- RAID 8+2 disks, read 4MB, then go elsewhere; repeat
  - Time to transfer data = 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

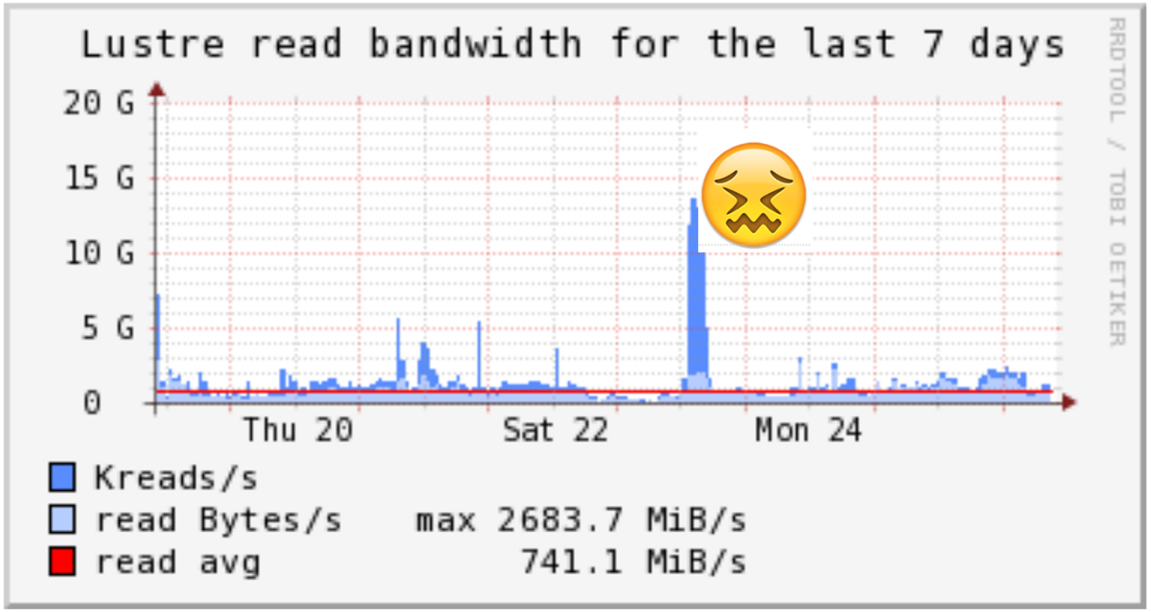
- Always read in chunks of at least 4 Mbytes if you have lots of data to read (e.g. a 1 GB file)
- Impact on your job if you do this correctly: you spend most of your time computing, and very little time waiting for I/O to complete
- Consequence of NOT doing this: you spend most of your time waiting 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!

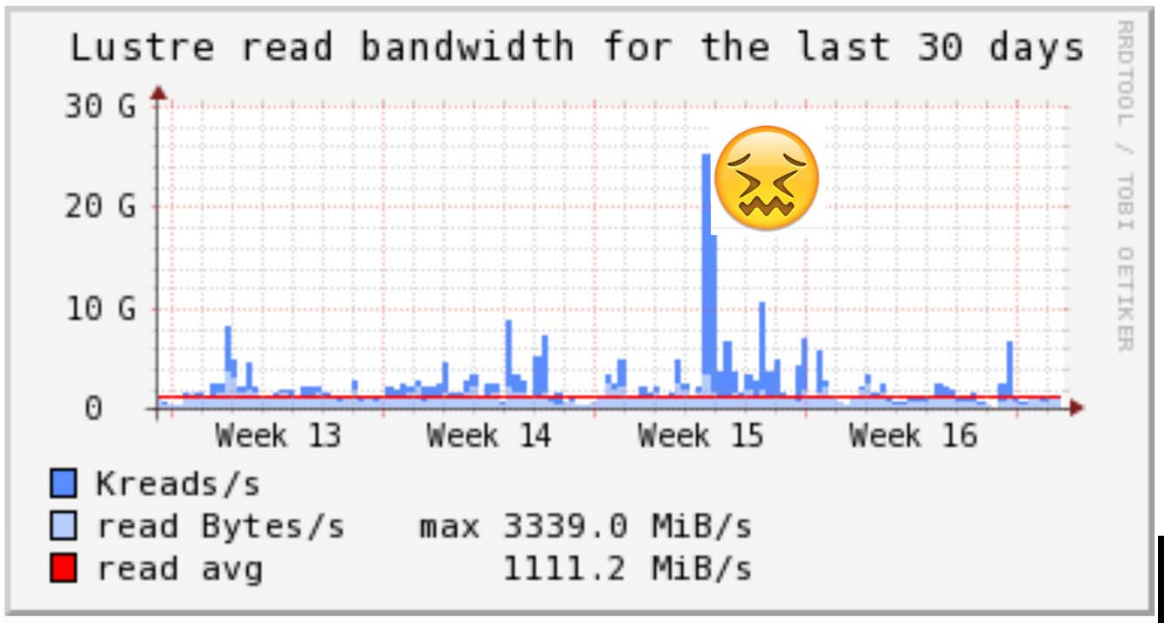
Observing GB/s vs IOPS



- 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.

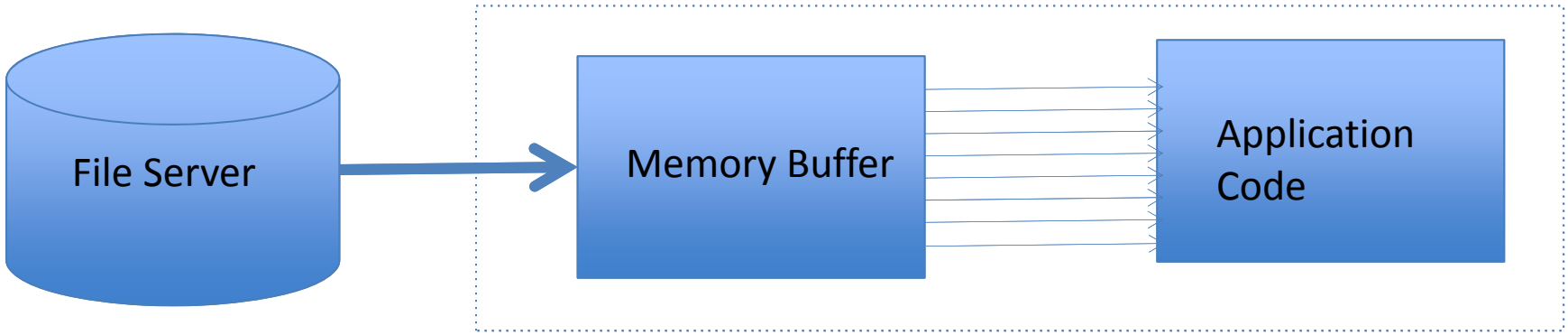
In fact, once in the past month, the entire system had to be rebooted!

How do I minimize this?

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

COMPUTE AND STORAGE FOR THE FARM AT JLAB

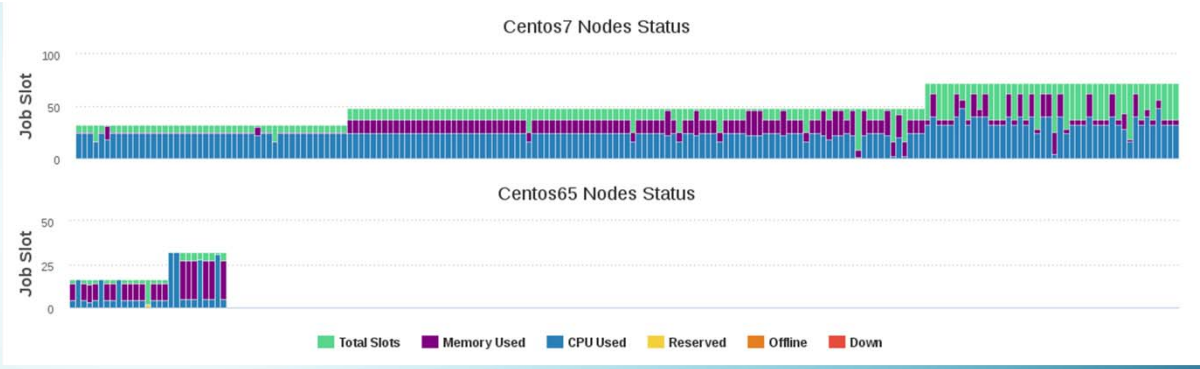
KURT J. STROSAHL

SCIENTIFIC COMPUTING, THOMAS JEFFERSON NATIONAL ACCELERATOR FACILITY

OVERVIEW

- Compute nodes to process data and run simulations
  - Single and multi-thread capable
  - Linux (CentOS)
- On disk space and tape storage
  - Disk space for immediate access
  - Tape space for long term storage
- Interactive nodes
  - Code testing
  - Review of data
- Infiniband network
  - Provides high speed data transfer between nodes and disk space

COMPUTE NODES



- Run CentOS Linux (based off of Red Hat)
  - CentOS 7.2 primarily (10,224 total job slots)
  - CentOS 6.5 being phased out (544 total job slots)
- Many software libraries available
  - Python
  - Java
  - Gcc
  - fortran
- Span several generations of hardware
  - Newer systems have more memory, larger local disks, faster processors

FILE SYSTEMS

- CNI provided
  - /group - a space for groups to put software and some files, backup up by CNI
  - /home - your home directory, backed up by CNI
- Cache - write through cache
- Volatile - acts as a scratch space
- Work – unmanaged outside of quotas / reservations

/CACHE

- A write-through cache (files get written to tape)
- 450 TB of total space
- Resides on the Lustre file system
- Usage is managed by software
  - Quota: Limit that can be reached when luster usage on a whole is low
  - Reservation: Space that will always be available
  - Groups are allowed to exceed these limits if other groups are under their limits.

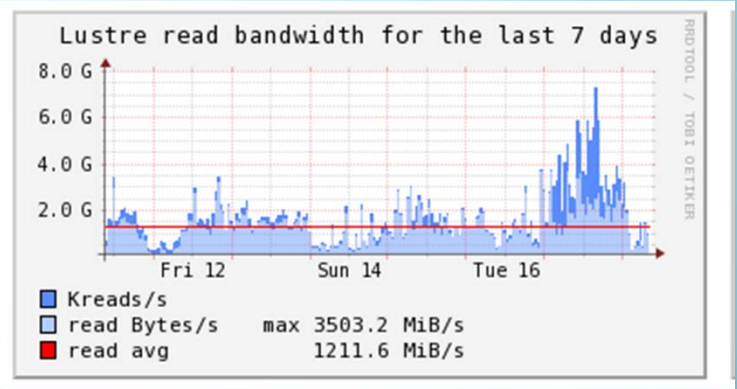
/VOLATILE

- 200TB of space total
- Uses a separate set of quotas and reservations
- Files are not automatically moved to tape
- Files are deleted if they have not been accessed in six months
- When a project reaches its quota files are deleted based on a combination of age and use (oldest, least used first).
- If lustre use is critical files are deleted using the same

/WORK

- Exists on both the lustre file system and independent file servers (zfs)
- Has quotas and reservations
- Is otherwise managed by users
- Non-lustre work space is for operations that are heavy in small i/o (code compilation), databases, and operations that require file locking.
- Is not backed up automatically
  - Can be backed up manually using the jasmine commands

LUSTRE FILE SYSTEM



- A clustered, parallel, file system
  - Clustered: Many file servers are grouped together to provide one "namespace"
  - Parallel: Each file server performs its own reads and writes to clients
  - A metadata system keeps track of file locations, coordinates
- Luster has a quota system, but it applies to the whole lustre file system and not subdirectories.
  - You can check this quota using: lfs quota –g <group name>
  - This quota is a sum of all quotas (volatile, cache, work) and serves as a final break against a project overrunning lustre
- Best performance comes from large reads and writes
- Uses ZFS to provide data integrity
  - Copy on Write (COW)
  - Data is checksummed as it is written, and that checksum is then stored for later verification

TAPE STORAGE

- IBM tape library, with LTO4 to LTO6 tapes
- Long term storage
- Is accessed by bringing files from tape to disk storage
  - Jasmine commands request specific files to be sent or retrieved from tape
  - Can be accessed through auger and SWIF submission scripts
- Maximum file size is 20GB

AUGER, OR HOW YOU USE THE BATCH FARM

- Auger provides an intelligent system to coordinate data and jobs.
- Takes either plain text or xml input files
- describe what files are needed for a job
- What resources a job is going to need
  - Memory, number of processors (threads), on-node disk space, time needed
- What to do with output files
- Full details of the auger files, and the options available can be found at https://scicomp.jlab.org/docs/auger_command_files

LINKS

- https://scicomp.jlab.org/
  - System status
  - Where your jobs are
  - Status of tape requests
  - Important news items
- https://scicomp.jlab.org/docs/FarmUsersGuide
  - Documentation of commands available

SWIF

SCIENTIFIC WORKFLOW INDEFATIGABLE FACTOTUM

WHY?

- Optimize tape access
- Easily cancel / modify / retry jobs
- Specify inter-job dependencies
- Script-friendly
- Simplify batch system

TERMS

- Workflow
  - Named container for batch jobs
  - Must be unique per-user
- Job
  - Named collection of specifications for launching job
- Attempt
  - One iteration of running job on batch farm
  - Possibly many attempts per job
- Documentation
  - https://scicomp.jlab.org/docs/swif

CREATE A WORKFLOW

- Create an empty workflow, then add jobs:
  - `swif create "My Workflow"`
  - `swif add-job "My Workflow" …`
- Create a full workflow defined by JSON file
  - `swif import -file my-workflow.json`

ADD A JOB

- `swif add-job "My Workflow" -project gluex -track one_pass [options] … command`
- Create a new job in workflow named 'My Workflow', for project 'gluex' using track 'one_pass' optionally defining
  - Resource limits (wall time, disk bytes, ram bytes, cpu cores)
  - Job tags (for your own book-keeping)
  - Input and output files
  - Conditions (antecedent jobs or external semaphore)
  - Job phase
- Will dispatch once workflow is active and all conditions are resolved

```
swif add-job “my workflow”
    -name run1_pass2
    -project gluey
    -track reconstruction
    -os centos7
    -ram 8G
    -disk 10G
    -time 4h
    -tag run 1
    -tag pass 2
    -tag file_number 456
    -input myfile1 /mss/halle/gluey/f123
    -input myfile2 /volatile/halle/blah
    -output result /mss/halle/gluey/recon/r1p2.out
    -output meta /home/bobdobbs/r1p2.meta
    -antecedent run0_pass2
    -condition file:///volatile/halle/cond/x12
    -phase 2
    -shell /bin/zsh
    -cores 8
    -stdout /volatile/halle/myjob.out
    -stderr /volatile/halle/myjob.err
    /home/bobdobbs/scripts/job-launch -some args -for script
```

START WORKFLOW

```
# swif run "My Workflow" [ options ]
```

Start dispatching and scheduling jobs.

Options:

- `-phaselimit N`
  - Only dispatch jobs with phase <= N
- `-errorlimit N`
  - Stop dispatching/scheduling jobs if num errors exceeds N
- `-joblimit N`
  - Dispatch at most N jobs
  - (This feature is currently disabled, will be re-enabled soon)

PAUSE WORKFLOW

```
# swif pause “My Workflow” [ -now ]
```

This will prevent new jobs from being dispatched or scheduled.

Options:

- -now
  - Cancel and recall any currently running or scheduled job attempts
- (Use swif run to resume running)

CANCEL WORKFLOW

```
# swif cancel "My Workflow" [ options ]
```

This will cancel any running job attempts and delete the workflow.

Options:

- `-delete`
  - Also delete the workflow
- `-discard-tape-files`
  - Also delete any tape files produced by constituent jobs
- `-discard-disk-files`
  - Also delete any disk files produced by constituent jobs

FREEZE / UNFREEZE WORKFLOW

- `# swif freeze "My Workflow"`
  - Prevent any further actions on workflow (i.e. make read-only)

- `# swif unfreeze "My Workflow"`
  - Undo effects of swif freeze

CHECK WORKFLOW STATUS

```
# swif status “My Workflow”
```

Display summary info about workflow

Options:

- `-problems`
  - Display jobs with problems
- `-jobs`
  - Display status of all jobs
- `-display [ xml | json ]`
  - Format output in xml or json

```
[larrieu@scdm1 larrieu]$ /site/bin/swif status calib
workflow_id                  = 9787
workflow_name                = calib
workflow_user                = beattite
jobs                         = 4502
succeeded                    = 4244
problems                     = 257
dispatched                   = 1
auger_active                 = 1
problem_types                = AUGER-INPUT-FAIL,SWIF-USER-NON-ZERO,AUGER-TIMEOUT,AUGER-
FAILED
problem_auger_timeout        = 13
problem_auger_failed         = 2
problem_auger_input_fail     = 156
problem_swif_user_non_zero   = 86
attempts                     = 4502
create_ts                    = 2017-05-16 21:41:10.0
update_ts                    = 2017-05-18 10:23:49.0
current_ts                   = 2017-05-18 14:49:42.0
```

```
[larrieu@scdm1 larrieu]$ /site/bin/swif status calib -problems
job_run_id          = 8106563
job_id              = 5748346
job_options_id      = 1751
workflow_id         = 9787

problem_type_name   = AUGER-INPUT-FAIL
workflow_name       = calib
job_name            = calib_Run031003_010
auger_id            = 38050490
auger_state         = FAILED
auger_vmem_kb       = 0
auger_wall_sec      = 5
copy_uri            = /cache/halld/RunPeriod-2017-01/rawdata/Run031003/hd_rawdata_031003_010.evio
copy_error          = /bin/cp: cannot stat '/cache/halld/RunPeriod-2017-
01/rawdata/Run031003/hd_rawdata_031003_010.evio': No such file or directory

job_run_id          = 8106565
job_id              = 5748348
job_options_id      = 1751
workflow_id         = 9787

problem_type_name   = SWIF-USER-NON-ZERO
workflow_name       = calib
job_name            = calib_Run031004_218
exitcode            = 134
auger_id            = 38050505
auger_state         = SUCCESS
auger_vmem_kb       = 5122972
auger_wall_sec      = 3817
```

JOB LIFECYCLE

- Starts in pending list
- On dispatch, moves to run list
- From run list:
  - Success list
  - Problem queue
- From problem queue:
  - Run list (retry / modify)
  - Fail list (abandon)

RETRY PROBLEM JOBS

- `# swif retry-jobs "My Workflow"`
- Move jobs from problem queue back into pending list
- Options:
  - `-names`
    - Select jobs by name
  - `-problems`
    - Select all jobs with specified problem(s)
  - `-regexp`
    - Selection will be matched against regular expression

EXAMPLE

- Resubmit all problem jobs that failed with system or unknown error.
- `swif retry-jobs “my workflow”`

  `-problems SWIF-SYSTEM-ERROR AUGER-UNKNOWN`

MODIFY JOB OPTIONS

- Change some parameters of jobs
  - RAM
  - Disk
  - Time
  - CPU Cores
- Set new value or add/subtract from current value

EXAMPLE

- Double the memory of jobs with id ‘123’ and ‘456’, halve time, add four cores.
- `swif modify-jobs "my workflow"`

```
-ram mult 2
-time mult 0.5
-cores add 4
123 456
```

EXAMPLE

- Subtract an hour from the requested time for all jobs, add 1GB ram.
- `swif modify-jobs "my workflow"`

```
-time add -1h
-ram add 1gb
-names
-regexp '.*'
```

EXAMPLE

- Request 8 cores for all unresolved problem jobs that failed owing to Auger timeout or resource limits.
- `swif modify-jobs “my workflow”`

```
-cores set 8
-problems AUGER-TIMEOUT AUGER-OVER-RLIMIT
```

ABANDON JOBS

- Move problem jobs to failure list
- Cancel running jobs, move to failure list
- Cancel undispatched jobs, move to cancel list

EXAMPLE

- Abandon all jobs named with trailing numbers in the range [100-300)
- `swif abandon-jobs “my workflow”`

```
-names
-regexp '.*_[12][0-9]{2}'
```

JLab Auger

Auger is the interface to JLab's data analysis cluster ("the farm")

- Controls batch job submissions
- Manages input/output from jobs
- Provides details on job status
- Gathers job statistics

- Code written in-house at JLab
- Connects to open source PBS/Torque/Maui resource manager and scheduler
- Underlies JLab's SWIF workflow tool (Chris' talk)

JLab Auger (2)

Auger commands

- jsub – submits job described by text or xml file
- jobstat – checks job statistics
- jkill – ends running jobs
- jobinfo – displays running job information
- farmhosts – displays farm node information

Auger commands reside in /site/bin, available from all JLab CUE (Common User Environment) systems.

Interactive user systems are **ifarm.jlab.org**

JLab Auger (3)

Auger requires specifying a valid project, track, and command

- Project - select appropriate from

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

- Track – describes job type, submits to appropriate queue

Track	Batch Queue	Description
debug	priority	For debugging use
reconstruction	prod64	Reconstruction of raw data
analysis	prod64	Analysis jobs
one_pass	prod64	Combined reconstruction/analysis
simulation	prod64	Simulation jobs
test	idle	Test run of code
theory	longJob	Running long theory jobs

JLab Auger (4)

Auger options include specifying other than defaults for

OS
COMMAND_COPY
JOBNAME
MAIL
TIME
OPTIONS
INPUT_FILES
SINGLE_JOB
MULTI_JOBS
OTHER_FILES
INPUT_DATA
OUTPUT_DATA
OUTPUT_TEMPLATE
CPU
DISK_SPACE
MEMORY

Details for these options are at https://scicomp.jlab.org/docs/text_command_file

JLab Auger (5)

Examples

Text configuration file

ifarml> **jsub <config file>** f

```
PROJECT: MyProject
TRACK: MyTrack
JOBNAME: MyJob
COMMAND: ls
```

XML configuration file

ifarml> **jsub -xml <XML config file>** f

```
<Request>
 <Email email="user@jlab.org" request="false" job="true"/>
 <Project name="MyProject"/>
 <Track name="MyTrack"/>
 <Name name="MyJob"/>
 <Command><![CDATA[
   ls
   ]]></Command>

<Job>
</Job>
</Request>
```

JLab Auger (6)

Standard out and standard errors

- stored in your ~/.farm_out directory
- named <job-id>.out , <job-id>.err

Exit codes

- User defined exit codes are passed by Auger as the final job exit code; if the job is killed by PBS for out of resource or other, the code is 128 + the signal number

- Exit code of 0 means the job completed successfully

JLab Auger (7)

Fairshare and Priorities

Auger Account	FS percentage
halla	10%
hallb	20%
hallc	10%
halld	60%
other	10%

Physics Computing Coordinators can request adjustments based on current priorities

JLab Auger (8)

Useful tips…

- When you need resources other than defaults, request enough but not too much, especially
  - MEMORY – wastes resources if overspecified
  - WALLTIME – if the job is short, tell the scheduler; likely it will be able to fit small jobs in sooner
- Only one user runs jobs on a node at a time – use the whole machine because it's all yours
- Run multi-threaded jobs – more efficient than the same number of single-threaded jobs doing the same work
- I/O usage – Lustre likes large files (Chip's talk)
- Use SWIF, JLab's workflow tool (Chris' talk)

Jlab Auger (9)

Viewing information about jobs

http://scicomp.jlab.org

From the command line,

```
ifarm> jobstat -u ton
JOB_ID   USER    STAT  QUEUE     EXEC_HOST   JOB_NAME        SUBMIT_TIME   CPU_TIME
WALLTIME  ACCOUNT
38065376  ton     Q     prod64    --          ...eplay_1_1898   May 17 12:18  --        --        halla
38065379  ton     Q     prod64    --          ...eplay_2_1899   May 17 12:18  --        --        halla
38065381  ton     Q     prod64    --          ...eplay_3_1885   May 17 12:18  --        --        halla
38065383  ton     Q     prod64    --          ...eplay_4_1889   May 17 12:18  --        --        halla
ifarm> ls ~/.farmout
-rw-r--r--   1 philpott scicomp   117 Feb  2 12:00 FarmJob.33168932.out
-rw-r--r--   1 philpott scicomp   438 Feb  2 12:00 FarmJob.33168932.err
-rw-r--r--   1 philpott scicomp   117 Feb  2 12:05 FarmJob.33169006.out
-rw-r--r--   1 philpott scicomp    22 Feb  2 12:05 FarmJob.33169006.err
```

JASMine

JASMine

- Jefferson Lab's Asynchronous Storage Manager
- Interface between disk and tape storage
- In house written software
- Used with Auger and SWIF for efficient tape I/O
- Manages tape library and data
  - currently ~12 PB
  - IBM TS3500 tape robot with 14 LTO drives
  - stores all "raw" CEBAF beam data, with a duplicate copy in the vault, plus all other "production" data from processing on the farm

JASMine (2)

JASMine commands

- jput – put files to tape
- jget – get files from tape
- jqueue – info on read and write tape jobs
- jcancel – cancel jasmine jobs

File system /mss – stub files of the tape data

```
ifarm> ls -alL /mss
drwxrwxr-x   9 larrieu epics      4096 Jul 27  2016 accel
dr-xr-xr-x   6 jasmine scicomp    4096 Jan 10  1997 ccc
dr-xr-xr-x  62 jasmine nobody     4096 Feb 18 02:53 clas
drwxrwxr-x   9 clas12  clas12     4096 Jan 26 10:37 clas12
drwxrwxr-x   4 pavel   muscn      4096 Mar 22  2016 ehsq
dr-xr-xr-x   3 jasmine nobody     4096 Nov 11  2014 eic
dr-xr-xr-x  60 jasmine nobody     4096 Feb 15 11:03 halla
dr-xr-xr-x  10 jasmine nobody     4096 Feb  7 14:11 hallb
dr-xr-xr-x  40 jasmine nobody     4096 Mar  6 08:24 hallc
dr-xr-xr-x  20 jasmine nobody     4096 Jan  3 15:39 halld
dr-xr-xr-x 369 jasmine nobody    12288 Mar 30 16:36 home
drwxrwxr-x  36 jasmine lattice-1  4096 Apr 20 03:37 lattice
```