What happened to my job?

Cluster scheduling in (some) detail

What’s a cluster?

- Login nodes: which you normally interact with
- Compute nodes: controlled by scheduler, no external network
- Admin and storage nodes: hopefully invisible

So you submit your job to the scheduler, which like some inscrutable nature god, whimsically chooses compute nodes to run it eventually (if ever)...

Job environment

What does the scheduler need to run a job?

- Command: the program and its parameters
- stdin/stdout/stderr files
- Runtime and memory limits
- Number and type of cores, nodes (optional: grouping)
- GPU, other resources like licenses

Actually, submitting a job is just a database insert

All the scheduler does is collect some parameters when you submit a job.

But it starts considering the job for execution - comparing it against all the other jobs, to see which should go next.

The main goal is to avoid overcommitting any compute node: running more jobs than can be supported by the cores or memory (or GPUs, etc)

Scheduler cycle

- Prioritize jobs (dark magic here)
- Sort by descending priority
- Repeat: See if the highest priority job can start
  - This means scanning all nodes
  - Evaluate which resources (cores, memory, gpus, etc) are free
  - Whether the job in question can start given available resources

Starting a job means marking as busy the resources it will use, then starting user process(es) on the node(s) chosen. Most schedulers have a special agent on each node that does this, and also monitors the processes, waiting for them to end. And killing them if they exceed their allocated resources...

Simple schedule (single 4-core, ignoring memory)

Job 1: 1 core, 3 units of time

Job 2: 4 cores, 4 units of time

Job 3: 1 core, 3 units of time

Problems

- Smaller job may start opportunistically
- Larger jobs wait longer (priority inversion)
  - This is exponentially bad, since using twice the resources will normally result in waiting more than twice as long (probably 4x as long).
  - Not really “large”, but rather “picky” - any job with constrained resources will wait longer than jobs with easer-to-satisfy resources.
  - In a cluster with mixed serial and parallel jobs, the former have a structural advantage.
- Longer jobs make everything worse

Schedules and resources

- For every node, track which cores are in use and how much memory
- Scheduler may start jobs on idle resources
- By default, jobs start “opportunistically”
- Scheduler may have a future plan, too
- Call this a “forward reservation” - a planned commitment of resources that hasn’t started yet.
- Reservations create “bubbles”: free resources before the reservation
- “Backfill” may fill these bubbles with sufficiently short and small jobs

Planning is hard work

Scheduler will only make a limited number of forward reservations

- They are conditional on the **current** priority of pending jobs
  - New jobs may be higher priority, so require a whole new schedule
  - Jobs that end early also require re-doing the schedule
  - Pending jobs may also change priority
- Main benefit is to provide “picky” jobs an opportunity to avoid starving due to opportunism by less picky jobs

Remember priority (dark magic)?

- RAC jobs get a big priority boost
- SN groups have a small priority factor derived from the group’s usage over the past two months.
- Mainly priority is dependent on fairshare
  - Concept: cpu usage rate **target**
  - Fairshare target is compared to recent consumption
  - If usage is below target, Fairshare provides a priority boost
  - If above, the effective priority is decreased
  - Usage is based on a time window, so is eventually forgotten
  - Usage also decays exponentially (eg .9 after every 12 hours)

Job problems

- Running out of time
- Running out of memory
  - Usually means failed allocations, so can be handled by the program
  - This can happen at process startup
  - Didn’t quite get the parameters right
- Executable or library issues
  - Sqsub propagates environment variables to the job
  - If shared libraries work on a login node, they should work on a compute node
- Filesystem issues
  - Can’t do anything about it if you remove or overwrite the executable
  - Or rename directories, change permissions, etc

Resource limits

The linux kernel only enforces RLIMIT_AS (per-process virtual address space) and RLIMIT_CPUTIME. Walltime limit has to be enforced by the scheduler. And memory and cputime have to be summed across nodes

Jobs that exceed resource limits are killed.

RLIMIT_AS means that if the program uses too much memory, allocations will fail: the process won't be killed. (Many programs will crash when an allocation fails, though.)

IO in the job context

We mount the same filesystems on login, development and compute nodes (except for /archive). So the job's processes should be able to find all the files they need.

Except for /tmp, which is always node-specific. So /tmp on a login node is a different filesystem on a compute node: files don't get copied for you. But it's great for transient files you use during a job, since you aren't competing with other nodes for filesystem performance.

Final comments

- Never use MPI or threaded if you can use serial
- Avoid overestimating memory usage (it won’t make the program faster)
- Try to provide accurate time limits (helps for reservations)
- Avoid making multi-node jobs “picky” (job requiring 4 whole nodes will wait longer than one requiring 48 cores in any layout)
- Keep jobs queued, if you can - don’t wait to submit