Job Scheduling and Resource Allocation Algorithm

- User Summary
- Resource limits
- Queue structure
- Job scheduling
- Charging
- Related articles

The information on this page was written for previous generations of the NCI Peak Facility. While many of the concepts are the same, this page is no longer completely accurate as scheduling and resource allocation are a continually evolving area. We are working on updated versions of this documentation.

Within NCI’s general aims of providing a world class computational resource, the management policies of the NCI attempt to satisfy the (somewhat contradictory) objectives of:

- promoting large scale parallel use of the Facility by minimizing artificial resource limits
- allowing fair access to resources for all users either independent of, or inline with, their “share” or grant
- providing near optimal turnaround for all users
- minimizing the impact of any user on other users by not allowing accidental overcommitment of resources.

To try to achieve these goals, the NCI runs an enhanced version of the PBS queueing system which features:

- detailed management and allocation of ALL system resources and all job processes
- suspend/resume based scheduling
- the flexibility of per-user or per-project limits provided by locally written software called RASH.

Virtually all access to the computational resources of the Facility is via PBS. Only limited interactive use (in the traditional Unix sense) of resources is available for development and debugging purposes. The following is a description of the job scheduling and resource allocation implemented by this version of PBS.

User Summary

To save reading the detailed description of policies and algorithms below, here is a summary of the actions users should take to make best use of (and have least conflict with) the queueing and resource allocation system on the NCI:

- make reasonably accurate resource requests when using the queueing system. Your job will NOT start until the resources you have requested are available and are not allocated to any other job. Asking for resources you do not need can only delay your job. Keep in mind that asking for less resources than your job needs will lead to the premature termination of your job!
- as a special case of the above point make reasonable requests for job timelimits since many scheduling decisions are made based on that request. For example, asking for a 50 hour timelimit for a 2 hour job could lead to the job being suspended for 5 hours (a small fraction of 50 hours) after it is started!
- take some care when running parallel jobs. There are many ways in which a parallel job request can go wrong leading to your grant being charged for 100s of hours usage that you saw no value from.

Resource limits

All jobs must have associated requests for the four main limited resources – time, processors, memory and disk. These resource request values are used in multiple ways when scheduling and running a job. The job scheduling system tracks unallocated (as opposed to currently unused) physical resources on compute nodes and will only schedule a job if the physical resources requested are free. At that point those requested resources are effectively reserved for the job. The job execution component of the queueing system provides strict limiting of the jobs resources so that it does not exceed its requests. In the case of job preemption (a job temporarily suspending other jobs – see below) resources for both suspended and running jobs must be available.

The intent is that if a job does not exceed its resource requests, it should not be excessively affected by other jobs on the system and run to completion. Sharing bandwidths (such as memory, I/O or network bandwidths) will have some minor impact on a jobs performance but the terminal depletion of static resource should not occur. The converse of this guarantee is that if a job does try to exceed its resource requests, it will be terminated. The mandatory resource limits are:

<table>
<thead>
<tr>
<th>resource</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>walltime</td>
<td>The job time limit is imposed on wall clock time. All jobs have exclusive access to those cpus assigned to the job. If a job is suspended, its wall clock time will also be suspended.</td>
</tr>
<tr>
<td>ncpus</td>
<td>The processes constituting a job are bound to a set of processors of the size requested and are not shared with any other jobs while in the running state. See the Charging section for a further discussion of this and the use of walltime instead of cputime.</td>
</tr>
<tr>
<td>vmem</td>
<td>Memory is requested in terms of total virtual memory use (i.e. the sum of “size” of concurrent processes in a job). The sum of the vmem limits of all jobs running on a node will be no greater than the physical memory of that node.</td>
</tr>
<tr>
<td>jobsfs</td>
<td>Apart from quotas on globally accessible file systems, local dynamically reservable disk space (called jobsfs on NCI NF) is also allocated and limited.</td>
</tr>
</tbody>
</table>
Upper bounds are imposed on requests for these resources as noted below. However these limits are just initial defaults – they will be varied over time with experience and can be (and are intended to be) varied on a per-user or per-project basis. Justifiable variation requests will be granted where possible.

### Queue structure

The queues available reflect a very simple three level hierarchy of job priorities which can be summarised by the following table (the terms “lower” and “higher” used later in this document reflect the ordering in this table):

<table>
<thead>
<tr>
<th>Queue</th>
<th>Purpose</th>
<th>Relative Charge</th>
</tr>
</thead>
<tbody>
<tr>
<td>express</td>
<td>development/testing, quick turnaround &amp; interactive</td>
<td>3</td>
</tr>
<tr>
<td>normal</td>
<td>standard production use</td>
<td>1</td>
</tr>
<tr>
<td>bonus jobs</td>
<td>use of idle cpu cycles by projects with no grant left (run in normal queue)</td>
<td>0</td>
</tr>
</tbody>
</table>

The jobs in a given queue have a higher “runpriority” than jobs in lower priority queues – resources allowing, jobs in a higher runpriority can always preempt jobs in a lower queue to run. As discussed below, preemption can also occur between jobs within a queue when certain factors allow it.

This simple queue structure is an attempt to avoid the artificial division of jobs into resource (or hardware) oriented queues. In the same vein, the system is run as one large partition and only one queue configuration is used (the commonly used day/night/holiday configurations and associated changeover points during the day are avoided).

As much as possible, the scheduling system has the complete set of resources and jobs available at all times to match up. In lieu of partitions/queues/configurations, disparate usage demands and requirements are satisfied by a flexible and configurable job scheduler.

### Fair Share scheduling

A so-called “fairshare” scheduler prioritizes jobs based on the “share” of the owning user/project or more specifically whether a user or project was exceeding or not achieving their “share” in recent history. NCI is currently introducing elements of fairshare into its scheduling algorithm. NCI (and its constituent partners) have allocated time grants for projects to be consumed in a fixed period. Projects are given a nominal share of the system by comparing those grant sizes and the relative size of the partner’s share it was allocated under. Basically, fairshare means that projects with larger grants get a larger scheduling priority but it is not proportional to grant size. For example, all projects smaller than the median sized project get the same priority. There is some onus on users/projects with large grants to continually consume their grant during the time allocation period.

### Job scheduling

The actual scheduling algorithm utilizes a combination of:

- “shuffling” of queued jobs to achieve “fair” access
- various scoring functions for both jobs and nodes to decide on “fair” scheduling actions and
- job suspension/resumption to give reasonable turnaround for all jobs.

#### Queue shuffling

Before a scheduling cycle, each job in a queue is given a score which is a measure (in some sense) of the sum of all jobs, whatever their state, above (before) the given one in the same queue and belonging to the same user or project. (The sum of the number of cpus for all higher jobs is one possible measure.) The set of all queued jobs in the queue is then sorted based on this score.

The effect is to promote the queued jobs of users/projects with no running jobs and demote those of a user/project with many (or large) running jobs. Hence the term “fair access”. Note that:

- this is very different to FIFO – the time at which you submit has no bearing on its queue position
- your jobs queue position will “jump” whenever one of your running jobs completes.

There is a fairshare component to the queue ordering.

#### Job selection

Given that sufficient non-cpu resources are available, jobs are started under two circumstances:

1. there are enough (sufficiently colocated) cpus either idle or running jobs of lower runpriority – this is a “priority-run”. In general, some lower priority jobs will need to be suspended to allow the higher priority job to start which may involve a selection process.
2. a job requiring a large number of cpus may suspend smaller jobs at the same runpriority – this is a “parallel-run” and is generally only relevant to the normal queue. The suspending job needs to be compared with all prospective suspendable jobs to determine if a parallel-run can occur. Any one of the factors discussed below can be sufficient to prohibit a preemption – the relative size of the jobs is NOT the only factor.
**Scoring factors**

In both job start circumstances, if job suspension is necessary, queued and running jobs are compared pair-wise to decide which suspensions are allowed and/or most favoured. The decision is based on a “suspension score” which is formed from the combination of a number of factors:

1. The “expansion factor”, the factor by which the predicted completion time from job start of a job being suspended is greater than the jobs requested walltime:

   \[
   \text{XFactor} = \frac{(\text{requested_walltime} + \text{projected_suspend_time})}{\text{requested_walltime}}
   \]

   An attempt is made to keep this number as close to 1.0 as possible for all jobs and generally below 1.25 (sometimes exceeding this is unavoidable).

2. The number of cpus requested: larger jobs have priority compared to smaller ones.

3. The number of jobs already running: jobs owned by users/projects which have a very large number of running jobs (eg. after a weekend) will be targets for suspension.

4. The fraction of the job done: jobs close to completion (that is approaching their walltime request limit) are less likely to be suspended.

5. Fairshare factor: projects with large grants do get some scheduling advantage over smaller grants.

The numerical effects of each factor will be more dramatic in extreme circumstances. For example, factor 3 has virtually no effect unless there is a large discrepancy in the number of running jobs belonging to the same user/project as the “suspender” and “suspendee” jobs. The discrepancy being large in either direction will be reflected in the score.

The functional form and contribution of each factor is designed to provide a non-linear correctional response to undesirable (i.e. unfair or imbalanced) queue scenarios.

For priority job starts, the set of running jobs giving the lowest aggregate suspension score are chosen for suspension. For parallel job starts, each pairwise suspension score must be below a cutoff and the aggregate suspension score must be below an even more stringent threshold.

---

**Job start time “window”**

Jobs are not necessarily started in (shuffled) queue order. If a job cannot be started (because of resource shortage), further jobs in the queue are considered. Even if the scheduler decides to start a job in a given scheduling cycle, that decision may be reversed later in that cycle after consideration of (larger) jobs further down the queue. If a job is bypassed in a cycle, its score will be enhanced in later cycles. In particular, the expansion factor of jobs starting “late” (or “early”) is modified to, in some sense, reflect when the job was scheduled to start.

The overall intent is to avoid any job “starving”, i.e. be stranded in the queued state for an excessive time. Eventually the jobs score will be such that it must start. The combination of time dependent job scores and job preemption allows the scheduler greater flexibility in utilizing resources and still ensure that all jobs will start within a reasonable “window” around the time when they become eligible to run.

---

**Charging**

Projects are awarded grants in terms of Service Units (SUs) which is equivalent to 1 hour of dedicated access to 1 cpu at normal priority. So the factors determining the charge incurred by a job are the wallclock time for which it was in the running state, the number of cpus requested (as opposed to used!) and the queue in which it ran.

The question arises as to whether wallclock time charging is fair since system bandwidths are shared possibly leading to degraded performance. The (not entirely satisfactory) answer is that cputime is also adversely affected by shared memory and network bandwidth (often by more than 20%) to the same degree. Wallclock time charging has the advantage of much more accurately reflecting parallel use of the system.

It is not widely recognized that dedicated cpus are often essential to efficiently supporting the fine-grained parallelism, both in message passing models (eg. MPI) and shared memory threaded models (eg. OpenMP). The “task-spinning” implicit during communication in high performance implementations coupled with a frequency of synchronization much greater than that of process context switches makes uninterrupted cpu access mandatory for reasonable performance. For example on N cpus, an OpenMP job of greater than N threads can run factors of 2 or 3 times slower than on N threads!

As an aside, this is the reason why jobs are constrained to processor sets or cpusets. These are operating system constructs to fence job processes on to specified subsets of a node cpus. Their use ensures that the user “doing the right” by requesting sufficient cpus is not affected by jobs which try to use more cpus than allocated.

(1). Note that it is possible to gain interactive access to batch nodes via interactive use under PBS.

**Related articles**

- Job Scheduling and Resource Allocation Algorithm
- Why cron jobs don’t run on Raijin
- How to improve job CPU GPU efficiency