Grid Engine Configuration Recipes

Dave Love
2013-08-30

Table of Contents

This is a somewhat-random collection of commonly-required configuration recipes. It is written mainly from the point of view of high performance computing clusters, and some of the configuration suggestions may not be relevant in other circumstances or for old versions of gridengine. See also the other howto documents. Suggestions for additions/corrections are welcome (to `d.love @ liverpool.ac.uk`).

Script Execution

Unix behaviour

Set `shell_start_mode` to `unix_behavior` in your queue configurations to get the normally-expected behaviour of starting job scripts with the shell specified in the initial `#!` line.

Modules environment

A side-effect of `unix_behaviour` is usually not getting the normal login environment, specifically not with the `module` command available for those sites that use environment modules. At least for use with the `bash` shell, add the following to the site sge_request file to avoid users having to source `modules.sh` etc. in job scripts, assuming sessions from which jobs are submitted have modules available:

```
-v module -v MODULESHOME -v MODULEPATH
```

This may not work with other shells.

Parallel Environments

Heterogeneous/Isolated Node Groups

Suppose you have various sets of compute nodes over which you want to run parallel jobs, but each job must be confined to a specific set. Possible reasons are that you have

- significantly heterogeneous nodes (different CPU speeds, architectures, core numbers, etc.),
- groups of nodes which have restricted access, e.g. dedicated to a user group, controlled by an ACL,
- or islands of connectivity on your MPI fabric(s) which are either actually isolated or have slow communication boundaries over switch layers.

Then you'll want to define multiple parallel environments and host groups. There will typically be one PE and one host group (with possibly an ACL) for each node type or island. The PEs will all be the same, unless you want a different fixed `allocation_rule` for each, but with different names. The names need to be chosen so that you can conveniently use wildcard specifications for them. Normally the names will all have the same name base, e.g. `mpi-`.... As an example, for different architectures, with different numbers of cores which get exclusive use for any tightly integrated MPI:

```
$ qconf -sp mpi-8
pe_name            mpi-8
slots              99999
user_lists         NONE
xuser_lists        NONE
start_proc_args    NONE
stop_proc_args     NONE
allocation_rule    8
control_slaves     TRUE
job_is_first_task  FALSE
urgency_slots      min
accounting_summary FALSE
qsort_args         NONE
$ qconf -sp mpi-12
pe_name            mpi-12
slots              99999
user_lists         NONE
xuser_lists        NONE
start_proc_args    NONE
stop_proc_args     NONE
allocation_rule    12
control_slaves     TRUE
job_is_first_task  FALSE
urgency_slots      min
accounting_summary FALSE
qsort_args         NONE
```

with corresponding host groups `@quadcore` and `@hexcore` for each type of dual-processor box. Those PEs are assigned one-to-one to host groups, ensuring that jobs can't run across the groups (since parallel jobs are always granted a unique PE by the scheduler, whereas they can be split across queues).

```
$ qconf -sq parallel
...
seq_no     2,[@quadcore=3],[@hexcore-eth=4],...
...
pe_list    NONE,[@quadcore=make mpi-8 smp],[@hexcore=make mpi-12 smp],...
...
slots      0,[@quadcore=8],[@hexcore=12],...
...
```

Now the PE naming comes in useful, since you can submit to a wildcarded PE, `-pe mpi-*`, if you're not fussy about the PE you actually get. See Wildcarded PEs for the next step.

Suppose you want to retain the possibility of running across all the PEs (assuming they're not isolated). Then you can define an extra PE, say `allmpi`, which isn't matched by the wildcard.

Note SGE 8.1.1+ allows general PE wildcards (actually patterns), as documented, fixing the bug which meant that only `*` was available in older versions. The correct treatment might be useful with such configurations, e.g. selecting `mpi-[1-4]`.

JSVs

See jsv(1) and jsv_script_interface(3) for documentation on job submission verifiers, as well as the examples in `$SGE_ROOT/util/resources/jsv/`.

See also the Resource Reservation section.

Wildcarded PEs

When you would use a wildcarded PE as above, for convenience and abstraction, you can use a JSV to write the wildcard pattern. This JSV fragment from `jsv_on_verify` in Bourne shell re-writes `-pe mpi` to `-pe mpi-*`:

```
if [ $(jsv_is_param pe_name) = true ]; then
pe=$(jsv_get_param pe_name)
...
case $pe in
    mpi)
    jsv_set_param pe_name "$pe-*"
    pe="$pe-*"
    modified=1
    ...
```

Suppose you want to retain the possibility of running across all the PEs (assuming the groups aren't isolated). Then you can define an extra PE, say `allmpi`, which isn't re-written by the JSV.

Checking for Windows-style line endings in job scripts

Users sometimes transfer job scripts from MS Windows systems in binary mode, so that they end up with CRLF line endings, which typically fail, often with a rather obscure failure to execute if `shell_start_mode` is set to `unix_behavior`. The following fragment from `jsv_on_verify` in a shell script JSV prevents submitting such job scripts

```
# Avoid having to use literal ^M
ctrlM=$(printf "\15")
...
jsv_on_verify () {
...
  cmd=$(jsv_get_param CMDNAME)
  case $(jsv_get_param b) in y|yes) binary=y;; esac
  [ "$cmd" != NONE -a "$cmd" != STDIN -a "$binary" != y ] &&
      [ -f "$cmd" ] &&
      grep -q "$ctrlM\$" "$cmd" &&
      # Can't use multi-line messages, unfortunately.
      jsv_reject "\
  Script has Windows-style line endings; transfer in text mode or use dos2unix"
...
```

Scheduling Policies

See sched_conf(5) for detailed information on the scheduling configuration.

Host Group Ordering

To change scheduling so that hosts in different host groups are preferentially used in some defined order, set `queue_sort_method` to `seqno`:

```
$ qconf -ssconf
...
queue_sort_method seqno
...
```

and define the ordering in the relevant queue(s) as required with `seq_no`):

```
$ qconf -sq ...
...
seq_no 10,[@group1=4],[@group2=3],...
...
```

It is possible to use `seqno`, for instance, to schedule serial jobs preferentially to one 'end' of the hosts and parallel jobs to the other ‘end’.

Fill Up Hosts

To schedule preferentially to hosts which are already running a job, as opposed to the default of roughly round robin according to the load level, change the `load_formula`:

```
$ qconf -ssconf
...
queue_sort_method load
load_formula slots
...
```

assuming the `slots` consumable is defined on each node.

Reasons for compacting jobs onto a few nodes as possible include avoiding fragmentation (so that parallel jobs which require complete nodes have a better chance of being fitted in), or being able to power down complete unused nodes.

Note Scheduling is done according to load values reported at scheduling time, without lookahead, so that it only takes effect over time.

Since the load formula is used to determine scheduling when hosts are equal according to `queue_sort_method`, you can schedule to the preferred host groups by `seqno` as above, and still compact jobs onto the nodes using `slots` in the load formula, as above, i.e. with this configuration:

```
$ qconf -ssconf
...
queue_sort_method seqno
load_formula slots
...
```

Avoiding Starvation (Resource Reservation/Backfilling)

To avoid “starvation” of larger, higher-priority jobs by smaller, lower-priority ones (i.e. the smaller ones always run in front of the larger ones) enable resource reservation by setting `max_reservation` to a reasonable value (maybe around 100), and arrange that relevant jobs are submitted with `-R y`, e.g. using a JSV. Here is a JSV fragment suitable for client side use, to add reservation to jobs over a certain size, assuming that PE slots is the only relevant resource:

```
    if [ $(jsv_is_param pe_name) = true ]; then
        pe=$(jsv_get_param pe_name)
        pemin=$(jsv_get_param pe_min)
...
        # Check for an appropriate pe_min with no existing reservation.
        if [ $(jsv_is_param R) = false ]; then
            if [ $pemin -ge $pe_min_reserve ]; then
                jsv_set_param R y
                modified=1
            fi
        fi
```

Note For “backfilling” (shorter jobs can fill the gaps before reservations) to work properly with jobs which do not specify an `h_rt` value at submission, the scheduler `default_duration` must be set to a value other then the default `infinity`, e.g. to the longest runtime you allow.

To monitor reservations, set `MONITOR=1` in sched_conf(5) `params` and use qsched(1) after running `process_scheduler_log`; `qsched -a` summarizes all the current reservations.

Fair Share

It is often required to provide a fair share of resources in some sense, whereby heavier users get reduced priority. There are two SGE policies for this. The *share tree* policy assigns priorities based on historical usage with a specified lifetime, and the *functional* policy only takes current usage into account, i.e. is similar to the share tree with a very short decay time. (It isn't actually possible to use the share tree with a lifetime less than one hour.) Use one of the other, but not both to avoid confusion.

With both methods, ensure that the default scheduler parameters are changed so that `weight_ticket` is a lot larger than `weight_urgency` and `weight_priority` or set the latter two to zero if you don't need them. Otherwise it is possible to defeat the fair share by submitting with a high priority (`-p`) or with resources with a high urgency attached. See sge_priority(5) for details.

You may also want to set `ACCT_RESERVED_USAGE` in `execd_params` to use effectively 'wall clock' time in the accounting that determines the shares.

Functional

For simple use of the functional policy, add

```
weight_tickets_functional 10000
```

to the default scheduling parameters (`qconf -msconf`) and define a non-zero `fshare` for each user (`qconf -muser`). If you use `enforce_user auto` in the configuration,

```
auto_user_fshare 1000
```

could be used to set up automatically-created users (new ones only).

Warning `enforce_user auto` implies not using CSP security, which typically is not wise.

Share Tree

See share_tree(5).

To make a simple tree, use `qconf -Astree` with a file with contents similar to:

```
id=0
name=Root
type=0
shares=1
childnodes=1
id=1
name=default
type=0
shares=1000
childnodes=NONE
```

and give the share tree policy a high weight, like (`qconf -msconf`):

```
weight_tickets_share 10000
```

If you have auto-creation of users (see the warning above), you probably want to ensure that they are preserved with:

```
auto_user_delete_time 0
```

The share tree usage decays with a half-life of 7 days by default; modify `halftime` (specified in hours) to change it.

Resource Management

Slot Limits

You normally want to prevent over-subscription of cores on execution hosts by limiting the slots allocated on a host to its core (or actually processor) count — where "processors" might mean hardware threads. There are multiple ways of doing so, according to taste, administrative convenience, and efficiency.

If you only have a single queue, you can get away with specifying the slot counts in the queue definition (`qconf -`

mconf), e.g. by host group

```
   slots 0,[@hexcore=12],[@quadcore=8]...
```

but with multiple queues on the same hosts, you may need to avoid over-subscription due to contributions from each queue.

An easy way for an inhomogeneous cluster is with the following RQS (with qconf -arqs), although it may lead to slow scheduling in a large cluster:

```
{
   Name          host-slots
   description   restrict slots to core count
   enabled       true
   limit         hosts {*} to slots=$num_proc
}
```

This would probably be the best solution if num_proc, the processor count, is variable by turning hardware threads on and off.

Alternatively, with a host group for each hardware type, you can use a set of limits like

```
   limit         hosts {@hexcore} to slots=12
   limit         hosts {@quadcore} to slots=8
```

which will avoid the possible scheduling inefficiency of the $num_proc dynamic limit.

Finally, and possibly the most foolproof way in normal situations is to set the complex on each host, e.g.

```
$ for n in 8 16; do qconf -mattr exechost complex_values slots=$n \
   `qconf -sobjl exechost load_values "*num_proc=$n*"`; done
```

Memory Limits

Normally it is advisable to prevent jobs swapping. To do so, make the h_vmem complex consumable, and give it a default value that is (probably slightly less than) the lowest memory/core that you have on execution hosts, e.g.:

```
$ qconf -sc | grep h_vmem
h_vmem                h_vmem        MEMORY       <=       YES          YES         2000m
```

(See complex(5) and the definition of memory_specifier.)

Also set h_vmem to an appropriate value on each execution host, leaving some head-room for system processes, e.g. (with bash-style expansion):

```
$ qconf -mattr exechost complex_values h_vmem=31.3G node{1..32}
```

Then single-process jobs can't over-subscribe memory on the hosts—at least the jobs on their own—and multi-process ones can't over-subscribe long term (see below).

Jobs which need more than the default (2000m per slot above) need to request it at job submission with -l h_vmem=..., and may end up under-subscribing hosts' slots to get enough memory in total.

Each process is limited by the system to the requested memory (see setrlimit(2)), and attempts to allocate more will fail. If it is a stack allocation, the program will typically die; if it is an attempt to malloc(3) too much, well-written programs should report an allocation failure. Also, the qmaster tracks the total memory accounted to the job, and will kill it if allocated memory exceeds the total requested.

These mechanisms are not ideal in the case of MPI-style jobs, in particular. The rlimit applied is the h_vmem request multiplied by the slot count for the job on the host, but it is for each process in the job—the limit does not apply to the process tree as a whole. This means that MPI processes, for instance, can over-subscribe in the PDC_INTERVAL before the execd notices, and out-of-memory system errors may still occur if a job can fill swap fast enough. Future use of memory control groups will help address this on Linux. However, if you happen to use Open MPI 1.7 or later, you could use a JSV to set OMPI_MCA_opal_set_max_sys_limits consistent with h_vmem.

Note Killing by qmaster due to the memory limit may occur spuriously, at least under Linux, if the execd over-accounts memory usage. Older SGE versions, and possibly newer ones on old Linux versions, use the value of `VmSize` that Linux reports (see `proc(5)`); that includes cached data, and takes no account of sharing. The current SGE uses a more accurate value if possible (see `execd_params` `USE_SMAPS`). Also, if a job maps large files into memory (see `mmap(2)`), that may cause it to fail due to the rlimit, since that counts the memory mapped data, at least under Linux. A future version of SGE is expected to provide control over using the rlimit.

Note Suspended jobs contribute to the `h_vmem` consumed on the host, which may need to be taken into account if you allow jobs to preempt others by suspension.

Note Setting `h_vmem` can cause trouble with programs using `pthreads(7)`, typically appearing as a segmentation violation. This is apparently because the pthreads runtime (at least on GNU/Linux) defines a per-thread stack from the `h_vmem` limit. The solution is to specify a reasonable value for `h_stack` in addition; typically a few 10s to 100 or so of MB is a good value, but may depend on the program.

Note There is also an issue with recent OpenJDK Java. It allegedly tries to allocate 1/4 of physical memory for the heap initially by default, which will fail with typical `h_vmem` on recent systems. The (only?) solution is to use `java -Xmx`*N* explicitly, with *N* derived from `h_vmem`.

Licence Tokens

For managing Flexlm licence tokens, see Olesen's method. This could be adapted to similar systems, assuming they can be interrogated suitably. There's also the licence juggler for multiple locations.

Killing Detached Processes

If any of a job's processes detach themselves from the process tree under the shepherd, they are not killed directly when the job terminates. Use `ENABLE_ADDGRP_KILL` to turn on finding and killing them at job termination. It will probably be on by default in a future version.

Core Binding

Binding processes to cores (or 'CPU affinity') is normally important for performance on 'modern' systems (in the mainstream at least since the SGI Origin). Assuming cores are not over-subscribed, a good default (since SGE 8.0.0c) is to set a default in sge_request(5) of

```
-binding linear:slots
```

The allocated binding is accessible via `SGE_BINDING` in the job's environment, which can be assigned directly to `GOMP_CPU_AFFINITY` for the benefit of the GNU OpenMP implementation, for instance.

If you happen to use Open MPI, a good default in `openmpi-mca-params.conf` (at least for Open MPI 1.6), matching the SGE `-binding` is:

```
orte_process_binding = core
```

Binding is the default in Open MPI 1.8, but to sockets, not cores, in most cases. That can be changed with, say,

```
hwloc_base_bind_to_core = 1
```

Open MPI will respect the core binding from SGE, i.e. it won't try to bind to cores outside `SGE_BINDING`.

Registered Memory Limit for openib etc.

To use Infiniband via openib, processes need to be able to 'register' a large amount of memory, and the limit for jobs is typically too small by default (see `ulimit -l`). It is typically necessary to add `H_MEMORYLOCKED=infinity` to `execd_params` (see sge_conf(5)). This may be necessary with other transports too.

Administration

Maintenance Periods

Rejecting Jobs

In case you want to drain the system, adding `$SGE_ROOT/util/resources/jsv/jsv_reject_all.sh` as a server JSV will reject all jobs at submission with a suitable message.

Down Time

If you want jobs to stay queued, there are two approaches to avoid starting ones that might run into a maintenance period, assuming you enforce a runtime limit and the maintenance won't start any sooner than that period: a calendar and an advance reservation.

Calendar

You can define a calendar for the shutdown period and attach it to all your queues, e.g.

```
# qconf -scal shutdown
calendar_name    shutdown
year             6.9.2013-9.9.2013=off
week             NONE
# qconf -mattr queue calendar shutdown serial parallel
root@head modified "serial" in cluster queue list
root@head modified "parallel" in cluster queue list
```

Note To get the scheduler to look ahead to the calendar, you need to enable resource reservation (issue #493) but that reservation may interact poorly with calendars (issue #722), but it's not clear whether this is still a problem.

Advance reservation

Define a fake PE with `allocation_rule 1` and access only by the `admin` ACL, say, and attach it to all your hosts, possibly via a new queue if you already have a complex `pe_list` setup:

```
$ qconf -sp all
slots              99999
user_lists         admin
...
allocation_rule    1
...
$ qconf -sq shutdown
qname                 shutdown
hostlist              @allhosts
...
pe_list               all
...
```

Now you can make an advance reservation (assuming `max_advance_reservations` allows it, and you're in `arusers` as well as `admin`):

```
$ qrsub -l exclusive -pe all $(qselect -pe all|wc -l) -a 201309061200 -d 201309091200
```

Rebooting execution hosts

To reboot execution hosts, you need to ensure they're empty and avoid races with job submission. Thus, submit a job which requires exclusive access to the host and then does a reboot. Since you want to avoid root being able to run jobs for security reasons, use `sudo(1)` with appropriate settings to allow password-less executions of the commands by the appropriate users. You want to comment out `Defaults requiretty` from `/etc/sudoers`, add `!requiretty` to the relevant policy line, or use `-pty y` on the job submission. It is cleanest to shut down the execd before the reboot.

The job submission parameters will depend on what is allowed to run on the hosts in question, but assuming you can run SMP jobs on all hosts (some might not be allowed serial jobs), a suitable job might be

```
qsub -pe smp 1 -R y -N boot-$1 -l h=$node,exclusive -p 1024 -l h_rt=60 -j y <<EOF
/usr/bin/sudo /sbin/service sgeexecd.ulgbc5 softstop
/usr/bin/sudo /sbin/reboot
EOF
```

where `$node` is the host in question, and we try to ensure the job runs early by using a resource reservation and a high priority.

Broken/For-testing Hosts

Administrative Control

A useful tactic for dealing with hosts which are broken, or possibly required for testing and not available to normal users is to make a host group for them, say `@testing` (`qconf -ahgrp testing`), and restrict access to it only to admin users with an RQS rule like

```
   limit users {!@admin} hosts {@testing} to slots=0
```

It can also be useful to have a host-level string-valued complex (say `comment` or `broken`) with information on the breakage, say with a URL pointing to your monitoring/ticketing system. A utility script can look after adding to the host group, setting the complex and, for instance, assigning downtime (in Nagios' terms) for the host in your monitoring system.

Alternatively the RQS could control access on the basis of the `broken` complex rather than using host group separately.

A monitoring system like Nagios (which has hooks for such actions and is allowed admin access to the SGE system) can set the status as above when it detects a problem.

Using a restricted host group or complex is more flexible than disabling the relevant queues on the host, as sometimes recommended; that stops you running test jobs on them and can cause confusion if queues are disabled for other reasons.

Using Alarm States

As an alternative to explicitly restricting access as above, one can put a host into an alarm state to stop it getting jobs. This can be done by defining an appropriate complex and a load formula involving it, along with a suitable load sensor. The sensor executes periodic tests, e.g. using existing frameworks, and sets the load value high via the complex if it detects an error. However, since it takes time for the load to be reported, jobs might still get scheduled for a while after the problem occurs.

Running tests could also be done in the prolog potentially to set the queue into an error state before trying to run the job. However, that is queue-specific, and the prolog only runs on parallel jobs' master node.

Last updated 2015-01-08 10:57:12 GMT