Cluster Tools

Batch Queue Job Control

W. Trevor King

November 18, 2010

1 Submitting jobs

You can submit jobs to the batch queue for later proccessing with `qsub`. Batch queueing can get pretty fancy, so `qsub` comes with lots of options (see `man qsub`). For the most part, you can trust your sysadmin to have set up some good defaults, and not worry about setting any options explicitly. As you get used to the batch queue system, you'll want tighter control of how your jobs execute by invoking more sophisticated options yourself, but don't let that scare you off at the beginning. They are, after all, only *options*. This paper will give you a good start on the options I find myself using most often.

1.1 Simple submission

The simplest example of a job submission is:

```
$ echo "sleep 30 && echo 'Running a job...'" | qsub
2705.n0.physics.drexel.edu
```

Which submits a job executing `sleep 30 \&\& echo 'Running a job...'` to the queue. The job gets an identifying ID in the queue, which `qsub` prints to `stdout`.

You can check the status of your job in the queue with `qstat`.

```
$ qstat
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
2705.n0                   STDIN            sysadmin               0 Q batch
```

There is more information on `qstat` in Section 2.

If your job is too complicated to fit on a single line, you can save it in a sctipt and submit the script.

echo_script.sh

```
#!/bin/bash
# file: echo_script.sh
sleep 30
echo "a really,"
echo "really,"
echo "complicated"
echo "script"
```

```
$ qsub echo_script.sh
2706.n0.physics.drexel.edu
```

All the arguments discussed in later sections for the command line should have comment-style analogs that you can enter in your script if you use the script-submission approach with `qsub`.

Note that you *cannot* run executibles (non-shell-scripts) directly with `qsub`. For example

```
$ cat script.py
#!/usr/bin/python
print "hello world!"
$ qsub python script.py
```

will fail because `python` is an executible. Either use

```
$ echo python script.py | qsub
```

wrap your `python` script in a `bash` script

```
$ cat wrapper.sh
#!/bin/bash
python script.py
$ qsub wrapper.sh
```

or run your `python` script directly

```
$ qsub script.py
```

1.2 IO: Job names and working directories

You will often be interested in the `stdout` and `stderr` output from your jobs. The batch queue system saves this information for you (to the directory from which you called `qsub`) in two files *¡jobname¿*`.o`*¡jobID-number¿* and *¡jobname¿*`.e`*¡jobID-number¿*. Job IDs we have seen before, they're just the numeric part of `qsub` output (or the first field in the `qstat` output). Job IDs are assigned by the batch queue server, and are unique to each job. Job names are assigned by the job submitter (that's you) and need not be unique. They give you a method for keeping track of what job is doing what task, since you have no control over the job ID. The combined *¡jobname¿*.*¡jobID-number¿* pair is both unique (for the server) and recognizable (for the user), which is why it's used to label the output data from a given job. You control the job name by passing the `-n` *¡jobname¿* argument to `qsub`.

```
$ echo "sleep 30 && echo 'Running a job...'" | qsub -N myjob
2707.n0.physics.drexel.edu


$ qstat
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
2707.n0                   myjob            sysadmin               0 Q batch
```

Perhaps you are fine with `stdin` and `stdout`, but the default naming scheme, even with the job name flexibility, is too restrictive. No worries, `qsub` lets you specify exactly which files you'd like to use with the unsurprisingly named `-o` and `-e` options.

```
$ echo "echo 'ABC' && echo 'DEF' > /dev/stderr" | qsub -o my_out -e my_err
2708.n0.physics.drexel.edu


$ cat my_out
ABC


$ cat my_err
DEF
```

A time will come when you are no longer satified with `stdin` and `stdout` and you want to open your own files or worse, run a program! Because no sane person uses absolute paths all the time, we need to know what directory we're in so we can construct our relative paths. You might expect that your job will execute from the same directory that you called qsub from, but that is not the case. I think the reason is that that directory is not garaunteed to exist on the host that eventually runs your program. In any case, your job will begin executing in your home directory. Writing relative paths from your home directory is about as annoying as writing absolute paths, so `qsub` gives your script a nifty environment variable `PBS\_O\_WORKDIR`, which is set to the directory you called `qsub` from. Since *you* know that this directory exists on the hosts (since the home directories are NFS mounted on all of our cluster nodes), you can move to that directory yourself, using something like

```
$ echo 'pwd && cd $PBS_O_WORKDIR && pwd' | qsub
2709.n0.physics.drexel.edu
 ... time passes ...
$ cat STDIN.o2709
/home/sysadmin
/home/sysadmin/howto/cluster/pbs_queues
```

Note that if we had enclosed the echo argument in double quotes (`\"`), we would have to escape the '`\$`' symbol in our `echo` argument so that it survives the shell expansion and makes it safely into `qsub`'s input.

1.3 Long jobs

If you have jobs that may take longer than the default wall time (currently 1 hour), you will need to tell the job manager. Walltimes may seem annoying, since you don't really know how long a job will run for, but they protect the cluster from people running broken programs that waste nodes looping around forever without accomplishing anything. Therefor, your walltime doesn't have to be exactly, or even close to, your actual job execution time. Before submitting millions of long jobs, it is a good idea to submit a timing job to see how long your jobs should run for. Then set the walltime a factor of 10 or so higher. For example

```
$ echo "time (sleep 30 && echo 'Running a job...')" | qsub -j oe
2710.n0.physics.drexel.edu
 ... time passes ...
$ cat STDIN.o2710
Running a job...

real        0m30.013s
user        0m0.000s
sys        0m0.000s

$ echo "sleep 30 && echo 'Running a job...'" | qsub -l walltime=15:00
2711.n0.physics.drexel.edu

$ qstat -f | grep '[.]walltime'
```

You can set walltimes in `[[H:]M:]S` format, where the number of hours, minutes, and seconds are positive integers. I passed the `-j oe` combines both `sdtout` and `stdin` streams on `stdin` because `time` prints to `stderr`. Walltimes are only accurate on the order of minutes and above, but you probably shouldn't be batch queueing jobs that take so little time anyway.

1.4 Job dependencies

You will often find yourself in a situation where the execution of one job depends on the output of another job. For example, `jobA` and `jobB` generate some data, and `jobC` performs some analysis on that data. It

wouldn't do for `jobC` to go firing off as soon as there was a free node, if there was no data available yet to analyze. We can deal with *dependencies* like these by passing a `-W depend=¡dependency-list¿` option to `qsub`. The dependency list can get pretty fancy (see `man qsub`), but for the case outlined above, we'll only need `afterany` dependencies (because `jobC` should execute after jobs `A` and `B`).

Looking at the `man` page, the proper format for our dependency list is `afterany:jobid[:jobid...]`, so we need to catch the job IDs output by `qsub`. We'll just use the `bash` "back tick" command substitution operators (‘) for this.

```
$ AID=‘echo "cd \\\$PBS_O_WORKDIR && sleep 30 && echo \"we’re in\" > A_out" | qsub‘
$ BID=‘echo "cd \\\$PBS_O_WORKDIR && sleep 30 && pwd > B_out" | qsub‘
$ COND="depend=afterany:$AID:$BID -o C_out -W depend=afterany:$AID:$BID"
$ CID=‘echo "cd \\\$PBS_O_WORKDIR && cat A_out B_out" | \
>         qsub -W depend=afterany:$AID:$BID -o C_out‘
$ echo -e "A: $AID\nB: $BID\nC: $CID"
A: 2712.n0.physics.drexel.edu
B: 2713.n0.physics.drexel.edu
C: 2714.n0.physics.drexel.edu


$ qstat
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
2712.n0                   STDIN            sysadmin               0 R batch
2713.n0                   STDIN            sysadmin               0 R batch
2714.n0                   STDIN            sysadmin               0 H batch


$ cat C_out
we’re in
/home/sysadmin/howto/cluster/pbs_queues
```

Note that we have to escape the `PBS\_O\_WORKDIR` expansion twice. The back tick expansion expands `\bs\bs\bs\$PBS\ldots` to `\bs\$PBS\ldots`. The double quote expansion when evaluating the echo command expands that to `\$PBS\ldots`. When the job finally runs, we actually expand out to our working directory.

1.5 Job arrays

If you have *lots* of jobs you'd like to submit at once, it is tempting try

```
$ i=0
$ while [ $i -le 5 ]
>   do
>   JOBID=‘echo "echo ’Running a job...’" | qsub‘
>   let "i += 1"
> done
```

This does work, but it puts quite a load on the server as the number of jobs gets large. In order to allow the execution of such repeated commands the batch server provides *job arrays*. You simply pass `qsub` the `-t array\_request` option, listing the range or list of IDs for which you'd like to run your command.

```
$ echo "sleep 30 && echo ’Running job \$PBS_ARRAYID...’" | qsub -t 1-5
2721.n0.physics.drexel.edu


$ qstat
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
```

```
2721-1.n0                STDIN-1          sysadmin               0 R batch
2721-2.n0                STDIN-2          sysadmin               0 R batch
2721-3.n0                STDIN-3          sysadmin               0 R batch
2721-4.n0                STDIN-4          sysadmin               0 R batch
2721-5.n0                STDIN-5          sysadmin               0 R batch
```

One possibly tricky issue is depending on a job array. If you have an analysis job that you need to run to compile the results of your whole array, try

```
$ JOBID=‘echo "cd \\\$PBS_O_WORKDIR && sleep 30 && pwd && echo 1 > val\\\${PBS_ARRAYID}_out" | qsub -t 1-5‘
$ sleep 2 # give the Job a second to load in...
$ #echo "cd \\\$PBS_O_WORKDIR && awk ’START{s=0}{s+=\$0}END{print s}’ val*_out" | \
$ #    qsub -o sum_out -W depend=afterany:$JOBID
$ JOBNUM=‘echo $JOBID | cut -d. -f1‘
$ #echo "cd \\\$PBS_O_WORKDIR && awk ’START{s=0}{s+=\$0}END{print s}’ val*_out" | \
$ #    qsub -o sum_out -W depend=afterany:$JOBNUM
$ COND="depend=afterany"
$ i=1
$ while [ $i -le 5 ]
$ do
>   COND="$COND:$JOBNUM-$i"
>   let "i += 1"
> done
$ echo "cd \$PBS_O_WORKDIR && awk ’START{s=0}{s+=\$0}END{print s}’ val*_out" | \
>     qsub -o sum_out -W $COND
2723.n0.physics.drexel.edu

$ qstat
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
2722-1.n0                 STDIN-1          sysadmin               0 R batch
2722-2.n0                 STDIN-2          sysadmin               0 R batch
2722-3.n0                 STDIN-3          sysadmin               0 R batch
2722-4.n0                 STDIN-4          sysadmin               0 R batch
2722-5.n0                 STDIN-5          sysadmin               0 R batch
2723.n0                   STDIN            sysadmin               0 H batch

$ cat sum_out
5
```

Note that you must create any files needed by the dependent jobs *during* the early jobs. The dependent job may start as soon as the early jobs finish, *before* the `stdin` and `stdout` files for some early jobs have been written. Sadly, depending on either the returned job ID or just its numeric portion doesn't seem to work.

It is important that the jobs on which you depend are loaded into the server *before your depending job is submitted*. To ensure this, you may need to add a reasonable sleep time between submitting your job array and submitting your dependency. However, your depending job will also hang if some early jobs have *already finished* by the time you get around to submitting it.

Despite the annoyance, a single sleep per job array is cheaper than a sleep after each non-array job submission to avoid crashing the server in the `while`-loop approach. See the examples sections and `man qsub` for more details.

2 Querying

You can get information about currently running and queued jobs with `qstat`. In the examples in the other sections, we've been using bare `qstat`s to get information about the status of jobs in the queue. You get information about a particular command with

```
$ JOBID=‘echo "sleep 30 && echo ’Running a job...’" | qsub‘
$ sleep 2 && qstat $JOBID
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
2724.n0                   STDIN            sysadmin               0 R batch
```

and you can get detailed information on a every command (or a particular one, see previous example) with the `-f` (full) option.

```
$ JOBID=‘echo "sleep 30 && echo ’Running a job...’" | qsub‘
$ sleep 2 && qstat -f
Job Id: 2725.n0.physics.drexel.edu
    Job_Name = STDIN
    Job_Owner = sysadmin@n0.physics.drexel.edu
    job_state = R
    queue = batch
    server = n0.physics.drexel.edu
    Checkpoint = u
    ctime = Thu Jun 26 13:58:54 2008
    Error_Path = n0.physics.drexel.edu:/home/sysadmin/howto/cluster/pbs_queues
        /STDIN.e2725
    exec_host = n8/0
    Hold_Types = n
    Join_Path = n
    Keep_Files = n
    Mail_Points = a
    mtime = Thu Jun 26 13:58:55 2008
    Output_Path = n0.physics.drexel.edu:/home/sysadmin/howto/cluster/pbs_queue
        s/STDIN.o2725
    Priority = 0
    qtime = Thu Jun 26 13:58:54 2008
    Rerunable = True
    Resource_List.nodect = 1
    Resource_List.nodes = 1
    Resource_List.walltime = 01:00:00
    session_id = 18020
    Variable_List = PBS_O_HOME=/home/sysadmin,PBS_O_LANG=en_US.UTF-8,
        PBS_O_LOGNAME=sysadmin,
        PBS_O_PATH=/home/sysadmin/bin:/usr/local/bin:/usr/local/sbin:/usr/bin
        :/usr/sbin:/bin:/sbin:/usr/X11R6/bin:/usr/local/maui/bin:/home/sysadmi
        n/script:/home/sysadmin/bin:.,PBS_O_MAIL=/var/mail/sysadmin,
        PBS_O_SHELL=/bin/bash,PBS_SERVER=n0.physics.drexel.edu,
        PBS_O_HOST=n0.physics.drexel.edu,
        PBS_O_WORKDIR=/home/sysadmin/howto/cluster/pbs_queues,
        PBS_O_QUEUE=batch
    etime = Thu Jun 26 13:58:54 2008
    start_time = Thu Jun 26 13:58:55 2008
    start_count = 1
```

The `qstat` command gives you lots of information about the current state of a job, but to get a history you should use the `tracejob` command.

```
$ JOBID=‘echo "sleep 30 && echo ’Running a job...’" | qsub‘
$ sleep 2 && tracejob $JOBID

Job: 2726.n0.physics.drexel.edu

06/26/2008 13:58:57  S    enqueuing into batch, state 1 hop 1
06/26/2008 13:58:57  S    Job Queued at request of sysadmin@n0.physics.drexel.edu, owner = sysadmin@n0.physics.drexel.edu, job n
06/26/2008 13:58:58  S    Job Modified at request of root@n0.physics.drexel.edu
06/26/2008 13:58:58  S    Job Run at request of root@n0.physics.drexel.edu
06/26/2008 13:58:58  S    Job Modified at request of root@n0.physics.drexel.edu
```

You can also get the status of the queue itself by passing `-q` option to `qstat`

```
$ qstat -q

server: n0

Queue            Memory CPU Time Walltime Node  Run Que Lm  State
---------------- ------ -------- -------- ----  --- --- --  -----
batch              --      --       --      --    2   0 --   E R
                                                ----- -----
                                                    2     0
```

or the status of the server with the `-B` option.

```
$ qstat -B
Server             Max   Tot   Que   Run   Hld   Wat   Trn   Ext Status
----------------   ---   ---   ---   ---   ---   ---   ---   --- ----------
n0.physics.drexe     0     2     0     2     0     0     0     0 Active
```

You can get information on the status of the various nodes with `qnodes` (a symlink to `pbsnodes`). The output of `qnodes` is bulky and not of public interest, so we will not reproduce it here. For more details on flags you can pass to `qnodes`/`pbsnodes` see `man pbsnodes`, but I haven't had any need for fancyness yet.

3 Altering and deleting jobs

Minor glitches in submitted jobs can be fixed by altering the job with `qalter`. For example, incorrect dependencies may be causing a job to hold in the queue forever. We can remove these invalid holds with

```
$ JOBID=‘echo "sleep 30 && echo ’Running a job...’" | qsub -W depend=afterok:3‘
$ qstat && qalter -h n $JOBID && echo "hold removed" && qstat
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
2725.n0                   STDIN            sysadmin               0 R batch
2726.n0                   STDIN            sysadmin               0 R batch
2727.n0                   STDIN            sysadmin               0 H batch
hold removed
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
2725.n0                   STDIN            sysadmin               0 R batch
2726.n0                   STDIN            sysadmin               0 R batch
2727.n0                   STDIN            sysadmin               0 Q batch
```

`qalter` is a Swiss-army-knife-style command, since it can change many aspects of a job. The specific hold-release case above could also have been handled with the `qrls` command. There are a number of `q*` commands which provide a

If you decide a job is beyond repair, you can kill it with `qdel`. For obvious reasons, you can only kill your own jobs, unless your an administrator.

```
$ JOBID=‘echo "sleep 30 && echo ’Running a job...’" | qsub‘
$ qdel $JOBID
$ echo "deleted $JOBID" && qstat
deleted 2728.n0.physics.drexel.edu
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
2725.n0                   STDIN            sysadmin               0 R batch
2726.n0                   STDIN            sysadmin               0 R batch
2727.n0                   STDIN            sysadmin               0 R batch
```

4 Example 1: jobs depending on jobs

As an example of a medium-complexity compound job, consider my single-proccessor Monte Carlo with two layers of post processing batch submission script. The simulation program is the compiled binary `sawsim` which models the unfolding of a multi-domain protein. It takes command line arguments specifying model parameters and experimental conditions and prints the forces at which the various domains unfold to `stdout`. I want to run with a certain set of parameters, but sweep one of the parameters (pulling velocity) through a number of different values. For each basic-parameter–pulling-velocity combination, I want to run the simulation 400 times, to get statistically significant output. The first stage of the post-processing gathers the output of all the `sawsim` jobs into a single list of unfolding forces at each velocity. The second stage of the post-processing generates histograms of unfolding forces for each velocity and a file with pulling speed vs. unfolding force data for plotting.

run_vel_dist.sh

```
#!/bin/bash
#
# run_vel_dist.sh
#
# run N simulations each at a series of pulling speeds V
# (non-velocity parameters are copied from the get_vel_dist.sh command line)
# for each speed, save all unfolding data data_$V, and
#  generate a histogram of unfolding forces hist_$V
# also generate a file v_dep with the mean unfolding force vs velocity

# We can't depend on the entire array in one shot, so
# batch executions into a single array instance
BN=100 # sawsim executions per array job instance
N=50   # instances in a single array
Vs=".2e-6 .5e-6 1e-6 2e-6 5e-6"
SAFTEY_SLEEP=2 # seconds to sleep between spawning array job and depending on it
               # otherwise dependencies don't catch

STAMP=`date +"%Y.%m.%d.%H.%M.%S"`
mkdir v_dep-$STAMP || exit 1
cd v_dep-$STAMP
echo "Date "`date` > v_dep_notes
echo "Run$ sawsim $* -v \$V" >> v_dep_notes
echo "for V in $*" >> v_dep_notes

Is="" # build a list of allowed 'i's, since 'for' more compact than 'while'
i=1
while [ $i -le $N ]
  do
  Is="$Is $i"
  let "i += 1"
done

vdone_condition="afterany"
VJOBIDS=""
for V in $Vs
do
  # run N sawtooth simulations
  idone_condition="afterany"
  cmd="cd \$PBS_O_WORKDIR
         i=1
         while [ \$i -le $BN ]; do
           ~/sawsim/sawsim $* -v $V > run_$V-\$PBS_ARRAYID-\$i
           let \"i += 1\"
         done"
  JOBID=`echo "$cmd" | qsub -h -t 1-$N -N "sawsim" -o run_$V.o -e run_$V.e` || exit 1
  #  "3643.abax.physics.drexel.edu" --> "3643"
  ARRAYID=`echo "$JOBID" | sed "s/\\([0-9]*\\)\\([.].*\\)/\\1/g"`
```

```
  for i in $Is; do idone_condition="$idone_condition:$ARRAYID-$i"; done

  sleep $SAFTEY_SLEEP
  # once they're done, compile a list of all unfolding forces for each speed
  JOBID=`echo "cd \\\$PBS_O_WORKDIR && cat run_$V-* | grep -v '#' > data_$V" |\
      qsub -h -W depend="$idone_condition" -N sawComp -o data_$V.o -e data_$V.e` \
        || exit 1
  vdone_condition="$vdone_condition:$JOBID"
  VJOBIDS="$VJOBIDS $JOBID"
  for i in $Is; do qrls "$ARRAYID-$i"; done
done
# once we have force lists for each velocity, make histograms and F(V) chart
echo "Final job :"
echo "cd \$PBS_O_WORKDIR && ../vel_dist_graph.sh $Vs" | \
    qsub -W depend="$vdone_condition" -N sawsimGraph || exit 1
for job in $VJOBIDS; do qrls $job; done
cd ..

exit 0
```

We start off by generating and `cd`ing into a new directory, since we'll be generating *lots* of files. Then we loop through our velocities, creating a new job array (`sawsim`) executing `sawsim` with the appropriate parameters.

Job arrays are still young in Torque (our batch queue manager), and we can't depend on the job array as a whole. Because the number of jobs we can depend on simultaneously is limited (command line size?), we bundle several `sawsim` executions into a single array job instance. This allows us to keep the number of executions up without generating huge dependency lists for the analysis jobs. We add job array with a user hold (`-h`) to ensure that the jobs don't begin executing immediately and exit before we start our analysis jobs. We also sleep for a few seconds after adding the job, because sometimes the array takes a bit to 'deploy' into the queue (symptom: out-of-order job IDs in the queue).

The first postprocessing stage (`sawComp`) gathers the output from the array job for it's particular velocity, strips comment lines, and dumps the output into a central file (`data\_\$V`). We set this job to depend on each of the array job instances at that velocity (since we can't depend on the entire array job). We also start *this* job on hold, so that it doesn't exit before we add the second layer analysis job. After adding the job, we release the user holds on the array job instances to let them get going (another place where access by array job ID would be nice...).

The final analysis stage involves enough steps that it has it's own script, but nothing in that script involves the batch queue system, so it is not reproduced here. We depend on all the first stage analysis jobs, since we need the consolidated `data\_\$V` files for the script. There is no need to hold this job, since nothing depends on it. After adding the job, we release the *user* holds on the first-stage analysis jobs. The *system* holds due to their dependencies remain until they are satisfied as the `sawsim` jobs complete.

5 Utilities

After playing around on the cluster for a while, I've developed some utilities to integrate job submission with the rest of my workflow. So far, I have written `dup\_env` for setting up an environment like the one you called qsub from, `qcmd` for submitting a single command as a job, and `qcmds` for submitting lots of one-line jobs and waiting until they complete.

5.1 dup_env

When `qsub` starts your job on a compute node, it sets up your environmental variables for you. Unfortunately, it doesn't do this by cloning your old environment, for example your new `PATH` will come from `/etc/bash.bashrc` (I think?). Your old `PATH` get's stored in `PBS\_O\_PATH` (see `man qsub`). `dup\_env` just loops through your current environment and overrides `qsub`'s decisions and reinstalls your original enviroment. There may be, of course, some exceptions, since you don't want to set your `HOST` to the host you called `qsub` from. Remember to place `dup\_env` somewhere where your restricted `qsub` path will find it.

~/bin/dup_env

```
#!/bin/bash
#
# override Torque qsub's strange environment shelving behavior by copying all
# your precious environmental variables back to where they should be.
#
# usage: source dup_env

while read VARSTR
  do
  VAR=${VARSTR%=*} # strip everything after & including the =
  VAL=${VARSTR#$VAR=} # strip everying up & including to the =
  #echo "parsed '$VARSTR' to '$VAR' & '$VAL'"
  if [ "${VAR:0:6}" == "PBS_O_" ]
      then
      ORIGVAR=${VAR#PBS_O_} # strip PBS_O_ from $VAR
      if [ "${!ORIGVAR}" != "$VAL" ] && [ "$ORIGVAR" != "HOST" ]
          then
          #echo "changing $ORIGVAR from '${!ORIGVAR}' to '$VAL'"
          export $ORIGVAR=$VAL
      fi
  fi
done < <(printenv)
```

5.2 qcmd

I often have scripts with a few big, slow jobs at the beginning, generating some data that I process in the remainder of the script. Rather than chew up the processors on the master node, I'd rather bump them out to the compute nodes. To make this alteration as painless as possible, I've written `qcmd` which submits the command in your current environment, waits for the job to finish, and outputs the job's `stdout`/`stderr`/exit-status as it's own. You can carry on exactly as if the job executed on your local machine. The script works by listening at your mailbox for the job completion email from the queue manager, but I talk about all that in the script itself.

~/script/qcmd

```
#!/bin/bash
#
# Run commands through a PBS queue without having to remember anything. :p
#
# This script keeps stdin and stdout open until the PBS job returns, monitoring
# your email for the appropriate job-completion message from the PBS server.
# This lets you bump a long-executing task onto a compute node, while keeping
# the lightweight part of your processing on the master node, for easier
# connection and dependency checking.
#
# Downsides to the current implementation:
# Blocks on email with a subshell (the piped input while loop) and tail
# running, so counting the script itself, that's 3 processes on the head node
# for each PBS job you're waiting for.  While that's not a really big deal for
# a few dozen jobs, it eats up memory fairly quickly for thousands of jobs.
```

```
#
# Another drawback is that you can only append to your $MAIL file while this is
# running.  I'm using a .procmailrc file to deflect PBS-related mail to a
# seperate mailbox, so I can still edit my system mailbox while this script is
# running, but that's one more thing you'd have to set up...
#
# Yet another drawback is that you can't run qsub from a screen session, but
# that is just qsub in general.  Sigh.
#
# The solution to the memory problem is to have a single script handle all
# the spawning and waiting for multiple commands.  Take a look at 'qcmds'
#
# usage: qcmd [-w[[H:]M:]S] command [arguments]
# where -w optionally sets the wall time
# for example:
#   $ ls .
#   file
#   $ qcmd cp file file2 && ls .
#   file
#   file2
#   $ diff file file2
#   (none)
# or
#   $ qcmd pwd '&&' echo \$PBS_O_WORKDIR
#   /home/sysadmin/script
#   /home/sysadmin/script
# and to prove we're running through qsub (2nd hostname is on calling system)
#   $ qcmd hostname && hostname
#   n8
#   n0
#
# Warning:
# This script uses process substitution which is a non-POSIX bash feature.
#
# command line arguments
#          |
#          v
#      addjob() ---(submit job with qsub)---> Job-Queue
#          |                                      |
#          +----+                          (job-complete email)
#               |                                 |
#       while:  v                                 |
#         checkfornew()                           v
#         getnextmail() <---(tail -f $MAIL)--- Mailbox

DEBUG=0
MAIL=$HOME/.mailspool/completed
# I have a ~/.procmailrc filter forwarding my PBS mail bodies to this $MAIL
# If you don't, comment the line out so you monitor your system $MAIL.

JOB_OUTSTANDING=  # Store the job id of our uncompleted job
MAIL_BODY=""      # Store the return of getnextmail()

if [ $DEBUG -eq 1 ]
    then
    DEBUGFILE=qcmd.$$
    > $DEBUGFILE # clear $DEBUGFILE if it existed before (Warning: clobber)
else
    DEBUGFILE=/dev/null
fi

# functions for job spawning

addjob ()
{
```

```
    CMMD=$*
    echo running: $CMMD >> $DEBUGFILE
    SCRIPT="cd \$PBS_O_WORKDIR && source dup_env && $CMMD"
    # dup_env is in ~/bin even though it's a script
    # since qsub creates it's own evironment, and moves X -> PBS_O_X
    # for example, PATH -> PBS_O_PATH.
    # who knows why it does this...
    JOBID=`echo $SCRIPT | qsub -mae $WALLTIME_OPTION || exit 1`
    # -mae : Send mail on abort or execute
    #JOBNAME=STDIN # the qsub default for scripts piped into qsub
    echo spawner: started $JOBID >> $DEBUGFILE
    JOB_OUTSTANDING="$JOBID"
    echo "add new depend:  $JOBID" >> $DEBUGFILE
    return 0
}

# functions for the job checking loop

# look for completion message bodies along the lines of:
# PBS Job Id: 206.n0.abax.physics.drexel.edu
# Job Name:   STDIN
# ...
#                                 <-- blank line
#
# could also poll on ls.
# neither 'ls' or 'tail -f' busy loops seem to take measurable processor time.
getnextmail() # blocking, fd 3 is tail -f $MAIL output
{
    BODY=""
    DONE=0
    INJOB=0
    echo "block on mail" >> $DEBUGFILE
    while [ $DONE -eq 0 ] && read LINE <&3
      do
      if [ "${LINE:0:11}" == "PBS Job Id:" ]
          then
          # we're reading an email about a job.
          #echo "in a job" >> $DEBUGFILE
          INJOB=1
      fi
      if [ $INJOB -eq 1 ]
          then
          #echo "getting mail: $LINE" >> $DEBUGFILE
          if [ "${#LINE}" -eq 0 ]
              then
              #echo "matched blank line" # we're leaving the email about our job.
              #echo "got mail" >> $DEBUGFILE
              DONE=1
              break
          fi
          BODY=`echo -e "$BODY\n$LINE"`
      fi
    done
    echo "returning mail" >> $DEBUGFILE
    MAIL_BODY="$BODY"
    return 0
}

check4done()
{
    JOBID=$1
    i=0
    echo -n "were we waiting on $JOBID? " >> $DEBUGFILE
    if [ "$JOBID" == "$JOB_OUTSTANDING" ]
        then
```

```
        # Sometimes the email comes in BEFORE STDOUT and STDERR were written
        # stupid Torque...
        JOBNUM=`echo "$JOBID" | sed 's/[.].*//'`
        JOBNAME=STDIN # the qsub default for scripts piped into qsub
        STDOUT=$JOBNAME.o$JOBNUM
        STDERR=$JOBNAME.e$JOBNUM
        while [ ! -e "$STDOUT" ] || [ ! -e "$STDERR" ]
          do
          sleep 0
        done
        # end stupid Torque hack

        echo "yes" >> $DEBUGFILE
        return 0 # job complete :)
    fi
    echo "no" >> $DEBUGFILE
    return 1 # not one of our completing jobs
}

printjoboutput()
{
    JOBID=$1
    shift
    MAIL_BODY=$*

    JOBNUM=`echo "$JOBID" | sed 's/[.].*//'`
    JOBNAME=STDIN # the qsub default for scripts piped into qsub
    STDOUT=$JOBNAME.o$JOBNUM
    STDERR=$JOBNAME.e$JOBNUM

    cat $STDOUT
    cat $STDERR >&2
    rm -f $STDOUT $STDERR

    return 0
}


echo "spawning job" >> $DEBUGFILE

if [ ${1:0:2} == "-w" ]
then
    WALLTIME=${1:2}
    WALLTIME_OPTION="-l walltime=$WALLTIME"
    shift
fi

addjob $*

# use process substitution to keep tail running between reads, see 'man bash'
exec 3< <(tail -f $MAIL --pid $$) # open the MAIL file for reading on fd 3
# $$ expands to this script's PID
# --pid $$ sets up tail to die when this script exits, see 'man tail'

echo "loop on outstanding job" >> $DEBUGFILE

# email checking loop, in the forground
while [ -n "$JOB_OUTSTANDING" ]
  do
  getnextmail
  JOBID=`echo "$MAIL_BODY" | sed -n 's/PBS Job Id: *//p'`
  if check4done "$JOBID"
      then
      printjoboutput $JOBID "$MAIL_BODY"
      EXIT=`echo "$BODY" | sed -n 's/Exit_status=//p'`
```

```
      unset JOB_OUTSTANDING
  fi
done

# the tail dies automatically because of the --pid argument
# we'll leave the tail file descriptor open just in case
# tail tries to print something else before it dies.

echo "qcmd complete" >> $DEBUGFILE

exit $EXIT
```

5.3 qcmds

With three processes open for each waiting job, `qcmd` scales pretty poorly. For scripts where you need a bit more muscle to handle several tens to thousands of jobs, I wrote `qcmds`. `qcmds` is almost identical to `qcmd`, except that where you had one `qcmd` instance per job, now you have one `qcmds` instance per *set* of jobs. With a whole bunch of jobs and only one `stdout`, `qcmds` needed a seperate interface. You just pipe your endline ('`\bs n`') delimited jobs into `qcmds stdin`, and out the `stdout` comes a list of completing jobs, in a tab delimited list for easy processing. Run the examples suggested in the script comments to get a feel for how these work.

~/script/qcmds

```
#!/bin/bash
#
# Run commands through a PBS queue without having to remember anything. :p
#
# This is an attempt to scale qcmd functionality to multiple commands.
# The goal is to be able to spawn a bunch of PBS jobs from the command line or
# a script, and then wait until they all finish before carrying on.  Basically
# the bash 'wait' functionality for background commands, except for PBS jobs
# spawned by your particular process.
#
# This script doesn't bother with keeping stdin and stdout open for each job,
# since that could be a lot of file descriptors.  It also doesn keep a process
# going for each job.  Instead, we fire up qcmds in the backround attached to
# an input and output fifo.  New job commands get piped into the input fifo,
# and job completion information gets piped out the output fifo.  When all the
# jobs that qcmds has started have completed, qcmds closes it's output fifo.
#
# qcmds uses endlines ('\n') as it's job-delimiters when reading it's input,
# so you can only use single-line jobs.  They can be long lines though ;).
#
# Simple usage would be:
#   echo 'sleep 5 && echo hi there' | qcmds
# Common usage would be:
#   #!/bin/bash
#   i=0
#   $QCMDS=$((while [ $i -lt 1000 ]
#             do
#               echo "sleep 10 && echo $i"
#               let "i += 1"
#             done
#            ) | qcmds)
#   wait # wait for qcmds tail to finish
#   exit 0
#
# Downsides to the current implementation:
# You can only append to your $MAIL file while this is running.  I'm using a
# .procmailrc file to deflect PBS-related mail to a seperate mailbox, so I can
# still edit my system mailbox while this script is running, but that's one
```

```
# more thing you'd have to set up...
#
# Another drawback is that you can't run qsub from a screen session, but that
# is just qsub in general.  Sigh.
#
# For small numbers of jobs where the scripting overhead of a seperate process
# and fifos seems excessive, take a look at the more memory intensive 'qcmd'.
#
# For a nice introduction to bash arrays, see
# http://tldp.org/LDP/abs/html/arrays.html
#
# see ~/script/.test/t_fifo_readline for a demonstration of the `threading'
#
# Warning:
# This script uses process substitution which is a non-POSIX bash feature.
#
#        stdin
#          |
#    (list of jobs)
#          |
#          v
#      addjob() ---(submit job with qsub)---> Job-Queue
#          |                                      |
#          +----+                          (job-complete email)
#               |                                 |
#       while:  v                                 |
#         checkfornew()                           v
#         getnextmail() <---(tail -f $MAIL)--- Mailbox

DEBUG=0
MAIL=$HOME/.mailspool/completed
# I have a ~/.procmailrc filter forwarding my PBS mail bodies to this $MAIL
# If you don't, comment the line out so you monitor your system $MAIL.

JOBS_OUTSTANDING=( ) # Store job ids that haven't completed yet
NUM_OUTSTANDING=0
MAIL_BODY=""         # Store the return of getnextmail()

if [ $DEBUG -eq 1 ]
    then
    DEBUGFILE=qcmds.$$
    > $DEBUGFILE # clear $DEBUGFILE if it existed before (Warning: clobber)
else
    DEBUGFILE=/dev/null
fi

# functions for the job spawning subshell

addjob ()
{
    CMMD=$*
    echo running: $CMMD >> $DEBUGFILE
    SCRIPT="cd \$PBS_O_WORKDIR && source dup_env && $CMMD"
    # dup_env is in ~/bin even though it's a script
    # since qsub creates it's own evironment, and moves X -> PBS_O_X
    # for example, PATH -> PBS_O_PATH.
    # who knows why it does this...
    JOBID=`echo $SCRIPT | qsub -mae || exit 1`
    # -mae : Send mail on abort or execute
    #JOBNAME=STDIN # the qsub default for scripts piped into qsub
    echo spawner: started $JOBID >> $DEBUGFILE
    echo $JOBID # >> $SPAWN_TO_CHECK
    return 0
}
```

```
# functions for the job checking loop

check4new() # blocks for < 1 second, fd 3 is addjob() output
{
    read -t1 JOBID <&3 || return 1 # nothing new.
    # add job to our outstanding list
    JOBNUM=‘echo "$JOBID" | sed ’s/[.].*//’‘
    JOBS_OUTSTANDING[$JOBNUM]=1
    let "NUM_OUTSTANDING += 1"
    echo "add new depend:  $JOBID" >> $DEBUGFILE
    # extra space ’:  $J’ to align with addjob DEBUG message
    return 0
}


# look for completion message bodies along the lines of:
# PBS Job Id: 206.n0.abax.physics.drexel.edu
# Job Name:   STDIN
# ...
#                                 <-- blank line
#
# could also poll on ls.
# neither ’ls’ or ’tail -f’ busy loops seem to take measurable processor time.
getnextmail() # blocking, fd 4 is tail -f $MAIL output
{
    BODY=""
    DONE=0
    INJOB=0
    echo "block on mail" >> $DEBUGFILE
    while [ $DONE -eq 0 ] && read LINE <&4
      do
      if [ "${LINE:0:11}" == "PBS Job Id:" ]
          then
          # we’re reading an email about a job.
          #echo "in a job" >> $DEBUGFILE
          INJOB=1
      fi
      if [ $INJOB -eq 1 ]
          then
          #echo "getting mail: $LINE" >> $DEBUGFILE
          if [ "${#LINE}" -eq 0 ]
              then
              #echo "matched blank line" # we’re leaving the email about our job.
              #echo "got mail" >> $DEBUGFILE
              DONE=1
              break
          fi
          BODY=‘echo -e "$BODY\n$LINE"‘
      fi
    done
    echo "returning mail" >> $DEBUGFILE
    MAIL_BODY="$BODY"
    return 0
}

check4done()
{
    JOBID=$1
    JOBNUM=‘echo "$JOBID" | sed ’s/[.].*//’‘
    echo -n "were we waiting on $JOBID? " >> $DEBUGFILE
    if [ -n "${JOBS_OUTSTANDING[$JOBNUM]}" ]
        then
        echo "yes" >> $DEBUGFILE
        # Sometimes the email comes in BEFORE STDOUT and STDERR were written
        # stupid Torque...
```

```
        JOBNAME=STDIN # the qsub default for scripts piped into qsub
        STDOUT=$JOBNAME.o$JOBNUM
        STDERR=$JOBNAME.e$JOBNUM
        while [ ! -e "$STDOUT" ] || [ ! -e "$STDERR" ]
          do
          sleep 0
        done
        # end stupid Torque hack
        # remove the outstanding entry from the array.
        unset JOBS_OUTSTANDING[$JOBNUM]
        let "NUM_OUTSTANDING -= 1"
        return 0 # job complete :)
    fi
    echo "no" >> $DEBUGFILE
    return 1 # not one of our completing jobs
}

printjobinfo()
{
    JOBID=$1
    shift
    MAIL_BODY=$*

    JOBNUM=`echo "$JOBID" | sed 's/[.].*//'`
    JOBNAME=STDIN # the qsub default for scripts piped into qsub
    STDOUT=$JOBNAME.o$JOBNUM
    STDERR=$JOBNAME.e$JOBNUM

    HOST=`echo "$BODY" | sed -n 's/Exec host: *//p'`
    EXIT=`echo "$BODY" | sed -n 's/Exit_status=//p'`
    WALLTIME=`echo "$BODY" | sed -n 's/resources_used.walltime=//p'`
    SESSION=`echo "$BODY" | sed -n 's/session_id=//p'`

    echo -e "$JOBID\t$JOBNUM\t$EXIT\t$STDOUT\t$STDERR\t$HOST\t$SESSION\t$WALLTIME"
    echo -e "$JOBID\t$JOBNUM\t$EXIT\t$STDOUT\t$STDERR\t$HOST\t$SESSION\t$WALLTIME" >> $DEBUGFILE

    return 0
}

nothing()
{
    return 0
}

echo "start the spawning subshell" >> $DEBUGFILE

# job-spawning subshell, in a subshell so we can't access it's variables
# we send this script's stdin into the subshell as it's stdin (< /dev/stdin)
# and we open the subshell's output for reading on file descriptor 3
# with 'exec 3< <(subshell cmmsds)' (process substitution, see 'man bash').
exec 3< <(
    echo "spawner: about to read" >> $DEBUGFILE
    while read LINE
    do
      addjob $LINE
    done
    exit 0
) < /dev/stdin

# use process substitution to keep tail running between reads, see 'man bash'
exec 4< <(tail -f $MAIL --pid $$) # open the MAIL file for reading on fd 4
# $$ expands to this script's PID
# --pid $$ sets up tail to die when this script exits, see 'man tail'

echo "loop on outstanding jobs" >> $DEBUGFILE
```

```
# email checking loop, in the forground
check4new # make sure there is an outstanding job first...
while [ $NUM_OUTSTANDING -gt 0 ]
  do
  getnextmail
  while check4new; do nothing; done;  # clean out the pipe
  JOBID=‘echo "$MAIL_BODY" | sed -n ’s/PBS Job Id: *//p’‘
  if check4done "$JOBID"
      then
      printjobinfo $JOBID "$MAIL_BODY"
  fi
  echo "still $NUM_OUTSTANDING job(s) outstanding" >> $DEBUGFILE
done

wait

echo "cleanup" >>  $DEBUGFILE

exec 3>&-   # close the connection from the job-spawning subshell

# the tail dies automatically because of the --pid argument
# we’ll leave the tail file descriptor open just in case
# tail tries to print something else before it dies.

echo "qcmds complete" >> $DEBUGFILE

exit 0
```

5.4 qcleanmail

Both `qcmd` and `qcmds` listen for job-completion emails from the queue manager, and these can fill up your inbox pretty quickly. I wrote this little script to run through my mailboxes and delete all PBS-related messages.

~/script/qcleanmail

```
#!/bin/bash
#
# Running my qcmd implementation on your system mailbox can quickly fill it up
# with PBS junk.  This scans through the box and deletes any messages from the
# PBS admin.
#
# from man procmail:
# Procmail can also be invoked to postprocess an already filled system mailbox.
# ...
#
# This script is almost verbatim from ’man procmail’

ORIGMAILS="$MAIL $HOME/.mailspool/completed"
FILTER=$HOME/.mailspool/mailfilter/qcleanmail.proc

clean ()
{
    ORIGMAIL=$1
    if cd $HOME &&
        test -s $ORIGMAIL &&
        lockfile -r0 -l1024 .newmail.lock 2>/dev/null
        then
        umask 077
        trap "rm -f .newmail.lock" 1 2 3 13 15
        echo "cleaning $ORIGMAIL"
        # lock the system maildir, and extract all the messages
```

```
        lockfile -l1024 -ml
        cat $ORIGMAIL >>.newmail &&
        cat /dev/null >$ORIGMAIL
        lockfile -mu
        # lock released, now new messages can arrive as they normally do

        # process the mail we copied out with our filter
        # mail that should be in $ORIGMAIL will be appended as procmail notices...
        formail -s procmail $FILTER <.newmail &&
          rm -f .newmail
          rm -f .newmail.lock
    fi
    return 0
}

for M in $ORIGMAILS
  do
  clean $M
done

exit 0
```

The referenced `procmail` filter is:

~/.mailspool/mailfilter/qcleanmail.proc

```
# qcleanmail.proc
#
# remove mail from PBS queue administrator from your system mailbox
#
# see man procmail, procmailrc, and procmailex
# there is also a good procmail tutorial at
# http://userpages.umbc.edu/~ian/procmail.html
#
# procmail uses sed-like regexps

MAILDIR=$HOME/.mailspool   #$MAIL
LOGFILE=$MAILDIR/mailfilter/qcleanmail.log
SHELL=/bin/bash
HOST=`hostname -f`
PBS_ADM=adm@abax[.]physics[.]drexel[.]edu

# trash anything from satifying ALL of
# From adm@abax.physics.drexel.edu,
# Envelope-to: $USER@$HOST
# To: $USER@$HOST
# Subject: PBS JOB [0-9]*.$HOST
# lock the mail file while the processing is being carried out
:0:                                  # the last colon means use a lockfile
* ^From $PBS_ADM
* ^To: $USER@$HOST
* ^Subject: PBS JOB [0-9]*$HOST
/dev/null                            # write matching messages to /dev/null
```

In order to avoid filling up my system mailox with PBS junk between calls to `qcleanmail`, I use the following `\$HOME/.procmailrc` filter to deflect PBS messages to `\$HOME/.mailspool/completed`:

~/.procmailrc

```
# .procmailrc
#
# log recipt of mail from PBS queue administrator from your system mailbox
#
# see man procmail, procmailrc, and procmailex
# there is also a good procmail tutorial at
```

```
# http://userpages.umbc.edu/~ian/procmail.html
#
# procmail uses sed-like regexps

DEFAULT=$MAIL
MAILDIR=$HOME/.mailspool
LOGFILE=$MAILDIR/mailfilter/procmailrc.log
SHELL=/bin/bash
COMPLETED=$MAILDIR/completed

# log job id's and exit status for anything satifying ALL of
# From adm@abax.physics.drexel.edu,
# Envelope-to: sysadmin@n0.abax.physics.drexel.edu
# To: sysadmin@n0.abax.physics.drexel.edu
# Subject: PBS JOB [0-9]*.n0.abax.physics.drexel.edu
# lock the mail file while the processing is being carried out
:0 b:    # b only use the body, (c pass copy to other rules), : use a lockfile
* ^From adm@abax[.]physics[.]drexel[.]edu
* ^To: sysadmin@n0[.]abax[.]physics[.]drexel[.]edu
* ^Subject: PBS JOB [0-9]*[.]n0[.]abax[.]physics[.]drexel[.]edu
| cat >> $COMPLETED
```