Job Arrays

1. Introduction

A jobs array is specified by adding a task range to `qsub` via the `-t` flag:

```
% qsub -t 1-100 model.job

Your job-array NNNNNN.1-100:1 ("model.job") has been submitted
```

The scheduler will start 100 jobs, each starting the job script file `model.job`, and pass to each job a task identifier (a number between 1 and 100) via an environment variable.

The syntax for the `-t` flag is `-t n[-m[:s]]`, namely:

`-t 1-20`	run 20 tasks, with task IDs ranging from1 to 20
`-t 10-30`	run 21 tasks, with task IDs ranging from 10 to 30
`-t 50-140:10`	run 10 tasks, with task IDs ranging from 50 to 140 by step of 10 (50, 60, ..., 140)
`-t 20`	run *one* task, with task ID 20

Each instantiation of the job will have access to the following four environment variables:

`SGE_TASK_ID`	unique ID for the specific task
`SGE_TASK_FIRST`	ID of the first task, as specified with `qsub`
`SGE_TASK_LAST`	ID of the last task, as specified with `qsub`
`SGE_TASK_STEPSIZE`	task ID step size, as specified with `qsub`

You can also limit the number of concurrent tasks with the `-tc` flag, for example:

```
% qsub -t 1-1000 -tc 100 model.job
```

will request to run 1,000 jobs, but no more than 100 running at the same time.

2. Example of a Job Script

The follow example shows how to submit a job array, using embedded directives:

Example of job script to submit a job array, using C-shell syntax

```
# /bin/csh
#
#$ -N model-1k -cwd -j y -o model.$TASK_ID.log
#$ -t 1-1000 -tc 100
#
echo + `date` $JOB_NAME started on $HOSTNAME in $QUEUE with jobID=$JOB_ID and taskID=$SGE_TASK_ID
#
set ID = model.$SGE_TASK_ID
./model -id $ID
#
echo = `date` $JOB_NAME for taskID=$SGE_TASK_ID done.
```

This example

- requests to run 1,000 models, using a task ID ranging from 1 to 1000, but limited to 100 running at the same time;
- assumes that the model computation is run with the command `./model -id <ID>`, where `<ID>` is the string `model.N` and `N` a number between 1 and 1000;
- also shows how to use the pseudo variable `TASK_ID` (not `SGE_TASK_ID`, yes, I agree, it is confusing) to give the log file of each task a different name: the output of task 123 will be `model.123.log` in the current working directory.

3. How to Convert a Task ID to a More Useful Set of Parameters

In most cases, when starting a computation you will need to convert a simple task identifier to a slew of parameters.

You can put that conversion into your code, but you may not want to do it, or can't do it because you are using some tool or package you can't modify.

Here are a few suggestions (Un*x tricks, using C-shell syntax) on how to do it:

1. You can use a separate input file for each job (task):
   If your code reads from `stdin` (standard input) you can do something like this:

   ```
   @ i = $SGE_TASK_ID
   ./domodel < input.$i
   ```

   You just need to prepare as many `input.NNN` files as cases you want to run, from `input.1` to `input.500` for example.

   If you prefer to call them `input.001` to `input.500`, you can use `awk` to reformat `$i` as follows:

   ```
   @ i = $SGE_TASK_ID
   set I = `echo $i | awk '{printf "%3.3d", $1}'`
   ./domodel < input.$I
   ```

   where `"%3.3d"` is the trick (a C-like format specifier) to convert the integer `$i` into a 3 character string `$I` with leading zeros if needed.

   The Bourne shell (`sh` or `bash`) equivalent is:

   ```
   i=$SGE_TASK_ID
   I=`echo $i | awk '{printf "%3.3d", $1}'`
   ./domodel < input.$I
   ```

2. You can use a single text file that lists a slew of parameters and extract one line, using the command `awk` (`man awk`):

   ```
   @ i = $SGE_TASK_ID
   set P = (`awk "NR==$i" parameters-list.txt`)
   ./compute $P
   ```

   This example will extract one line from the file `parameters-list.txt,` namely the line whose line number is stored in the variable `$i` (the 1st line, the 2nd line, etc). [NR stands for record number]

   It sets the variable `$P` to the content of that one line.

   You just have to create such a file with as many lines as cases of `compute` you wish to run. Each line holds the parameters that will be passed to `compute`.

   The Bourne shell (`sh` or `bash`) equivalent is:

   ```
   i=$SGE_TASK_ID
   P=`awk "NR==$i" parameters-list.txt`
   ./compute $P
   ```

3. You can write a tool (a small program or script, that I call here `mytool`) that does the conversion. You just run it to get the parameters:

   ```
   @ i = $SGE_TASK_ID
   set P = (`./mytool $i`)
   ```

or for Bourne shell (`sh` or `bash`) aficionados:

```
i=$SGE_TASK_ID
P=`./mytool $i`
```

4. You can use the shell syntax to use or manipulate the variable $SGE_TASK_ID to execute the right (set of) command(s) from the given task integer

4. How to Consolidate Small Jobs into Fewer Larger Jobs

- There is some overhead each time the GE starts a job, or a task.
- So if you need to compute let's say 5,000 similar tasks, each taking only 3 minutes,
  - it may be convenient to submit a 5,000 task job array, but
  - it will be inefficient: the system will spend 25 to 50% of its time starting and keeping track of a slew of small jobs.
- The following script illustrates a simple trick to consolidate such computations when using a job array:

Example of job array consolidation wrapper script, domodel.job, using C-shell syntax

```
# /bin/csh
#
# simple wrapper to consolidate using the step size
#
#$ -N model-1k20 -cwd -j y -o model-$TASK_ID-by-20.log
#$ -t 1-1000:20
#
echo + `date` $JOB_NAME started on $HOSTNAME in $QUEUE with jobID=$JOB_ID
#
@ iFr = $SGE_TASK_ID
@ iTo = $iFr + $SGE_TASK_STEPSIZE - 1
if ($iTo > $SGE_TASK_LAST) @ iTo = $SGE_TASK_LAST
#
echo running model.csh for taskIDs $iFr to $iTo
@ i = $iFr
while ($i <= $iTo)
  ./model.csh $i >& model-$i.log
  @ i++
end
#
echo = `date` $JOB_NAME for taskIDs $iFr to $iTo done.
```

This wrapper, that I call `domodel.job`, will run 20 models in a row, using the step size of the job array, via the `csh` script `model.csh`.

So instead of running 1,000 three-minute-long jobs, it will run 50 one-hour-long jobs.

You can, of course, adjust the step size accordingly, including setting it to 1 for cases when individual models lead to long computations.

The script `model.csh` is simply:

C-shell script model.csh called by domodel.job

```
#!/bin/csh
#
set TID = $1
echo + `date` model.csh started for taskID=$TID
#
./model -id $TID
#
echo = `date` model.csh for taskID=$TID done.
```

but it can be as complex as you may need/want it to be. (BTW, there is nothing C-shell or Bourne shell syntax specific in this example)

The file `model.csh` must be authorized to be executed with the command:

```
% chmod +x model.csh
```

You can use a bash script, or any other valid Linux command in place of the line `./model.csh`.

5. Rules for Submitting Job Arrays that use Parallel Environments (like MPI)

⚠️ While one can submit a job array that uses a parallel environment (`-pe` and `-t,` or parallel job arrays), one ***must*** use a the following approach to avoid a race condition specific to SGE.

- By default, SGE makes a local copy of each job script on the compute nodes it runs on.
- Parallel job arrays should avoid this to prevent a race condition, where for a small fraction of the tasks the scheduler starts the script *before* it is copied, hence some tasks fails to start.

- The output of `qstat -j 9616234` will show something like this:
  `error reason  11:          03/24/2016 11:09:11 [10464:63260]: unable to find job file "/opt/gridengine/default/spool/compute-2-2/job_scripts/9416234"`
  and the SGE reporting file will list:
  `job never ran -> schedule it again`
- The output of `qstat -f -explain E | grep QERROR` will show something like this:
  `queue mThC.q marked QERROR as result of job 9616234's failure at host compute-1-2.local`
  leaving a queue entry in Error state.

How to Write a Parallel Job Array

1. Do not use embedded directive (😷).
2. Write a script (`sh`,`csh`, `perl`, `python`, etc) with the required steps, as for a job script.
3. Make that script executable (`chmod +x`), you can use the `#!` mechanism to specify the interpreter (aka shebang).
4. Write a file with the `qsub` command and all the options that you would otherwise put as embedded directives.
5. Pass the `-b y` option to `qsub` and specify the full path of the script to execute.
6. Source that file to submit the parallel job array.
7. ⚠️ Do not modify the executable script file while the job array is running.

Example

The following job script with embedded directives must be broken into two files:

one job script with embedded directives

demo.job

```
#--------
#$ -q mThC.q
#$ -pe orte 20
#$ -l mres=4G,h_data=4G,h_vmem=4G
#$ -cwd -y j -N demo -o demo.$TASK_ID.log
#$ -t 1-100
#-------
module load some/thing
#-------
echo + `date` job $JOB_NAME started in $QUEUE
with jobID=$JOB_ID on $HOSTNAME
echo + taskID=$SGE_TASK_ID
echo + NSLOTS=$NSLOTS distributed over:
cat $PE_HOSTFILE
#
mpirun -np $NSLOTS crunch -i input.$SGE_TASK_ID -
o output.$SGE_TASK_ID
#
echo = `date` job $JOB_NAME done
```

is replaced by two files, a qsub_XXX.sou and a XXX.sh

qsub_demo.sou

```
qsub \
 -q mThC.q \
 -pe orte 20 \
 -l mres=4G,h_data=4G,h_vmem=4G \
 -cwd -y j -N demo -o 'demo.$TASK_ID.log' \
 -t 1-100 \
 -b y $PWD/demo.sh
```

⚠️ no spaces after the '\'

demo.sh

```
#!/bin/sh
# any embedded directives here will be ignored
#-------
source /etc/profile.d/modules.sh
module load some/thing
#-------
echo + `date` job $JOB_NAME started in $QUEUE with
jobID=$JOB_ID on $HOSTNAME
echo + taskID=$SGE_TASK_ID
echo + NSLOTS=$NSLOTS distributed over:
cat $PE_HOSTFILE
#
mpirun -np $NSLOTS crunch -i input.$SGE_TASK_ID -o
output.$SGE_TASK_ID
#
echo = `date` job $JOB_NAME done
```

 This can be any type of executable script, but if you use:

- `#!/bin/sh` or `#!/bin/bash`, you'll need "`source /etc/profile.d/modules.sh`" to access modules,
- `#!/bin/sh -l` or `#!/bin/bash -l`, module will be defined, but the script will read `~/.profile`,
- `#!/bin/csh` or `#!/bin/tcsh`, module will be defined,
- `#!/bin/csh -f` or `#!/bin/tcsh -f`, you'll need `source /etc/profile.d/modules.csh` to access modules.
   Don't you love Linux? (It's all in the man pages, tho).

Before submitting the job array, make sure the script is executable:

```
chmod +x demo.sh
```

To submit the job array, simply source the `qsub_XXX.sou` file:

```
source qsub_demo.sou
```

You can edit the `qsub_demo.sou` to submit more tasks, but do not modify the executable script file while the job array is running.

Last updated 26 Jun 2018 SGK.