The University of Queensland

Research Computing Centre

Using Job Arrays

The following section has been updated but may still contain torque syntax or sample torque output.

Document Status

This update: April 7 2018 by David.Green@uq.edu.au

Introduction

For some research problems, it is necessary to submit very similar jobs multiple times to the batch system.
In this case, often the most effective approach is to use a job array to submit and manage that set of similar jobs as a single entity.

Job Array Scenarios

Note that job array sub-jobs are performed in a 'trivially parallel' mode. That is, that there can be no communication or dependency between "sub-jobs" in a job array. If they all started at the same time, it cannot matter.

Job arrays are well suited to the following scenarios

1. the same processing repeated many times for statistical purposes,
2. the same processing repeated many times with a different input parameter,
3. the same processing repeated many times with different input file(s)
4. the same processing repeated many times with different data directories

Job arrays are usually appropriate and beneficial whenever

- the individual computations are independent (i.e. all jobs could all start simultaneously without dependencies)

- there is a simple programmatic way to relate the job array index to the appropriate input parameter/file/directory

The job array contains multiple "sub-jobs" one for each value of the index parameter.
The sub-jobs run as separate batch jobs, but they are contained within the one easy to use job array entry in the batch system.

~~Job Arrays in Torque~~

~~Job arrays are submitted by using the -t option to the `qsub` command.~~
~~When the job array sub-job runs the current value of the "t parameter" is an environment variable `PBS_ARRAYID`~~
~~The index values can be non-consecutive `#PBS -t 1,11,21` and can be ranges as well `#PBS -t 1-11` etc.~~
~~There is further information about the Torque job arrays at the home of Torque !~~

Job Arrays in PBSPro

Job arrays are submitted by using the -J option to the `qsub` command.
When the job array sub-job runs the current value of the "J parameter" is an environment variable **`PBS_ARRAY_INDEX`**.
The index values can only be consecutive or fixed steps in a range `#PBS -J 1-101:25` means that the PBS_ARRAY_INDEX will have each of the 5 values 1, 26, 51, 76, 101.

There is further information about the PBSPro job arrays in

- the `man qsub` manual page
- the Tinaroo Cluster User Guide
- the UQ RCC PBSPro User Guide
- the vendor's PBSPro User Guide

Maybe you need Nimrod?

There are extreme cases of trivially parallel computation that are a test for job arrays.

Although job arrays are a great idea, they have their limits.
Some issues can arise when

- the sub-jobs do not run for very long (e.g. a few seconds)
- the sub-jobs number in the tens or hundreds of thousands

RCC supports (in fact, our Director developed) the Nimrod workload tool. It is a better tool to use whenever you have extremely large numbers of sub-jobs and/or the run time for each sub-job is tiny. The Nimrod portal will still work with the PBS batch system to acquire resources, but instead of each parameter combination being a separate batch job, the Nimrod system deploys agents into the allocated nodes. This approach reduces the administrative overhead markedly and provides a visual representation of the status of the parameter sweep.
Nimrod also provides mechanisms to perform optimisations and integration with Kepler.
If you think you may be a candidate for using Nimrod, please visit the Nimrod website

Submitting a PBSPro Job Array

Background

It is important to appreciate that in a job array,

- the **same** PBS script is used as a template for all jobs in the job array
- the resource request for the job array is the resource request for each and every element of the array
    - so if you request -l select=1:ncpus=2:mem=10GB, this applies to every sub-job in the array
- once the job arrays starts running the values of $PBS_ARRAY_INDEX are different and unique for each sub-job.

Declaring array index values

In the job array job script, the number of jobs to be submitted as part of the array is specified using the -J option.
This option takes a range of indices, in the form X-Y[:Z], where

- X is the start index,
- Y is the end index and
- Z is an optional step size.

Some examples

```
#1 2 3 4 5
#PBS -J 1-5

#2 4 6 8 10
#PBS -J 2-10:2

#10 and 20 only.
#PBS -J 10-20:10
```

How big can a job array be?

The maximum size of job arrays is constrained to avoid problems with the scheduler overloading.
The maximum job array size is set in batch system and may be queried using the following command:

```
uquser@flashlite1:~> qmgr -c "p s" | grep -i array
set server max_array_size = 10000
```

So this indicates that the limit is 10,000 elements per job array.

Using a job array PBS script

An example of a simple script, 'example_job_array.pro', is shown below:

```
#Simple Job Array Example
#PBS -N ProArrayTest
#PBS -A AccountString
#PBS -l select=1:ncpus=1:mem=4GB
#PBS -l walltime=01:00:00

#The following will give me index values 10 and 20 only.
#PBS -J 10-20:10

echo "This is a job array sub-job with index value $PBS_ARRAY_INDEX in job array $PBS_ARRAY_ID"
echo $TMPDIR
echo $HOSTNAME
sleep 30
```

A job array is submitted to the batch system in exactly the same way as an ordinary job:

```
qsub example_job_array.sh
```

This would submit an array with 2 jobs, with indices 10 and 20.
Each job would be allocated 1 core for the walltime of 1 hour.
Each sub-job will print out some text, "This is a job array ..." with a different value for $PBS_ARRAY_INDEX in each sub-job.

Note that **each and every** sub-job is allocated the full resources specified at the beginning of the PBS script.

The qstat command can be used to check the state of the array and the array sub-jobs.
The job array job entry (the one with the []) will transition from Q state, to B state while any subjobs are running, to F state when all subjobs are finished.
The job array sub-jobs (the ones with an index inside the []) will transition from Q to R to X as they queue, run and exit.

```
uquser@awoonga1:~/Demo/PBSPro> qstat -at1nw

awongmgmr1:
                                                                                                  Req'd  Req'd   El
Job ID                          Username        Queue           Jobname         SessID   NDS  TSK   Memory Time  S Ti
------------------------------ --------------- --------------- --------------- -------- ---- ----- ------ ----- - --
52426[].awongmgmr1              uquser          Short           ProArrayTest         --     1     1    4gb 01:00 B  -
52426[10].awongmgmr1            uquser          Short           ProArrayTest       28070    1     1    4gb 01:00 R 00
52426[20].awongmgmr1            uquser          Short           ProArrayTest       19714    1     1    4gb 01:00 R 00
user@awoonga1:~/Demo/PBSPro>
```

Job arrays can be used with multi-node jobs.
For instance, the following resource specification would start 2 jobs at values 1 2, each with 3 complete Tinaroo nodes.

```
#Tinaroo Example
#PBS -A AccountString
#PBS -l select=3:ncpus=24,mpiprocs=24,mem=120Gb
#PBS -l walltime=100:00:00
#PBS -J 1-2

echo "Job array subjob $PBS_ARRAY_INDEX"
echo "The following is the list of unique nodes involved in this subjob:"
```

```
cat $PBS_NODEFILE | sort | uniq
```

Making Use of the PBS_ARRAY_INDEX Variable

Example 1: Printing the job index

The script above is of little use, as each job in the job array prints the same output. In order to customize the behaviour of each specific job in a job array, the variable PBS_ARRAY_INDEX is used, as shown below:

```
#A PBS Pro Example
#PBS -A AccountString
#PBS -l select=1:ncpus=1:mem=4GB
#PBS -l walltime=00:30:00

#Array index values can be a range with optional step length (if not 1)
#PBS -J 0-20:10

echo "This is job $PBS_ARRAY_INDEX"
```

Each job in the array will run their own version of the commands in the PBS script, with the environment variable $PBS_ARRAY_INDEX set to the index of the job.

You can use this value to pass as a parameter, or fetch a value from a text file (using head and tail commands), or accessing a numerical directory.

Example 2: Redirecting input file names into a program

One common use of the PBS_ARRAY_INDEX variable is to run the same command with several different inputs.

For instance, if a user has files 'input_1.csv', 'input_2.csv', 'input_3.csv' and their program 'myApp' in their personal /30days folder, the following script could be submitted to run the myApp with each of the 3 input files:

```
#Syntax PBSPro
#PBS -A AccountString
#PBS -N NumberedFiles
#PBS -l select=1:ncpus1:mem=4GB
#PBS -l walltime=00:30:00
#PBS -J 1-3

cd /30days/$USER

./myApp < input_${PBS_ARRAY_INDEX}.csv
```

This assumes that myApp is a program which takes a csv file as standard input.

As all three jobs in the job array can run at the same time, this is a much better option than, for instance, looping over each input file within myApp itself.

Example 3: Reading the PBS_ARRAY_INDEX environment variable from within a program

As PBS_ARRAY_INDEX is an environment variable, it is possible to grab this value from within a running program or script. As an example, the R script 'myApp.R' below gets the index of a specific job and uses that index to construct the input file for that job:

```
# read job array index into 'index'
index=Sys.getenv("PBS_ARRAY_INDEX")

# construct the input file name
filename=paste("input_", index, ".csv", sep="")

# read in the input
myData=read.csv(filename)

# [do something with the data here]
```

As each job has its own set of environment variables, only a single PBS script, such as the one below, would be required to run three instances of myApp.R with three different input files.

```
#PBS -A AccountString
#PBS -N PassInIndex
#PBS -l select=1:ncpus=1:mem=4GB
#PBS -l walltime=00:30:00
#PBS -J 1-3
```

```
cd /30days/$USER
module load R

CMD BATCH myApp.R
```

Note that the function for grabbing the value of environment variables will vary from language to language. For instance, this function is getenv('variable_name') in MATLAB. In python, environment variables are stored under os.environ['variable_name'].

For a list of all PBS environment variables available to a job, add the command

```
env | grep PBS
```

to the PBS script of that job.

Example 4: Reading a (long) list of arbitrary input file or folder names

The 'head' and 'tail' commands can be used in combination with the PBS_ARRAY_INDEX variable to select the file/folder name at a particular index in the list of files in a user's working directory. This is useful if the input files do not have regular names such as input_1, input_2 etc.

The Nth filename or folder name in the present working directory can be selected and assigned to a variable THIS_ONE using

```
THIS_ONE=`ls -1 | head -n +N | tail -1`
```

(N must be a positive integer value)

The following PBS script reads a single file name in the 'inputs' folder of the user's working directory per job, and sends that file name to the standard input of 'myApp'. It is assumed there are 10 input files in that folder, and no other files.

```
#PBS -A AccountString
#PBS -N GetByFilename
#PBS -l select=1:ncpus=1:mem=4GB
#PBS -l walltime=00:30:00
#PBS -J 1-10

cd $PBS_O_WORKDIR

# get the filename in the inputs folder at position PBS_ARRAY_INDEX
filename=`ls -1 ./inputs | tail -n +${PBS_ARRAY_INDEX} | head -1`

# use that file as an input to myApp
myApp < ./inputs/$filename
```

Example 5: Reading a row of input parameters

This example is similar to reading a filename in the example above.
Instead of a filename, we will get a single row of text from the "big input data file" which will contain input parameters.

```
>head -2 big_input_data_file
1,47.5,17.2,5,black
2,66.6,11.1,3,orange
```

You can write those parameters into a file in $TMPDIR that has a generic name, then your job array subjobs will all be reading from the same filename (but it will be a different file location because of the different values of $TMPDIR)

```
#PBS -A AccountString
#PBS -N GetByParamsLine
#PBS -l select=1:ncpus=1:mem=4GB
#PBS -l walltime=00:30:00
#PBS -J 1-10

cd $PBS_O_WORKDIR

# get the filename in the inputs folder at position PBS_ARRAY_INDEX
cat big_input_data_file | tail -n +${PBS_ARRAY_INDEX} | head -1 > $TMPDIR/generic_input_single_line_filename

# use that generic input filename  as an input to myApp
myApp < $TMPDIR/generic_input_single_line_filename

# or you may be able to encode the generic filename into your application as long as it has access to the value of $
```

Example 6: Iterating over multiple parameters in a column

This is similar to the previous example except that the input parameters are columnar, so we have to "stride" through the file 3 lines at a time.

Say the same program needs to be run for combinations of initial x, y and z values.
The input file, 'parameters.txt', with 12 rows, could look like:

```
1.0
1.0
1.0
2.0
1.0
1.0
1.0
2.0
1.0
2.0
2.0
1.0
```

If the same program needs to be run multiple times for every combination of a number of input parameters, these parameter combinations can be 'flattened' and described by a single index. In the previous example we put all parameters for a single run on the same line. Here we put them all in a single column.

The head and tail commands can then be used in a similar way as above to grab the appropriate parameter values in each job but we will jump and fetch values 3 at a time:

```
#PBS -A AccountString
#PBS -N OneD23D
#PBS -l select=1:ncpus=1:mem=4GB
#PBS -l walltime=00:30:00
#PBS -J 1-12:3

cd $PBS_O_WORKDIR

# get the parameter combination at line PBS_ARRAY_INDEX in parameters.txt
parameters=`cat parameters.txt | tail -n +${PBS_ARRAY_INDEX} | head -3`

# split the three parameters into an array
parameterArray=($parameters)

# the value of each parameter is accessible using ${parameterArray[i]}
echo "x: ${parameterArray[0]}"
echo "y: ${parameterArray[1]}"
echo "z: ${parameterArray[2]}"
```

Naming Conventions

The job array identifiers are of the form id_number[].server_id, e.g. `44937[].awongmgmr1`. This refers to the entire array.
The ids of individual jobs in a job array also include the index of the job, e.g. `44937[1].awongmgmr1`. This refers to just the first sub-job.
Note that the server name extension, `.awongmgmr1`, is not required to specify the jobID because you are not working in a multi-server environment.

Job names, as well as the names of job array objects, are all assigned the same value, specified by the -N flag to `qsub` command or in the PBS script header.

Standard output file names are of the form [job_name].o[job_id].[job_index], e.g. 'myJob.o432192.2' for the second job in the job array 'myJob'. The corresponding standard error file would be 'myJob.e432192.2'.

Querying the Status of Jobs

In the normal output of qstat, only job array objects, rather than individual jobs in an array, will be shown. To view the individual jobs in job arrays, add the '-t' flag to qstat. To show only job array objects, add the '-J' flag to qstat. The status of a job array object is 'Q' when all jobs in the array are queued, 'B' when at least one job is running, and 'F' when finished. To view the percentage of jobs completed in an array, add the '-p' flag to qstat.

Modifying Submitted Jobs

Jobs in a job array which are still queued can be modified by using the qalter command on the job array object. For

instance, to reduce the walltime of queued jobs in job array '432189[]' to one hour, use the command:

```
qalter -l walltime=01:00:00 432189[]
```

Note that if some jobs in this array were already running, those jobs would be unaffected by the qalter command. It is also not possible to use qalter on individual jobs in a job array.

Using the qdel command with a job array object will delete all jobs in the job array from the queue, including any running jobs. It is possible to delete individual subjobs from a job array. Such jobs will be represented by the 'X' status in qstat.

Rerunning Individual Jobs from an Array

The same PBS script can be used to rerun single jobs in a job array (for instance if a job in the array failed) by changing the range of the -J argument to include only that job. For instance, to rerun the job with index 50, use:

```
#PBS -J 50:50
```

Staggering Job Start Times

This should not be required as batch system has been configured to stagger the starts of swarms of jobs.

For large job arrays, jobs are usually started in groups of up to thirty jobs at once. This could cause an overload on the storage system, if, for instance, every one of those jobs acessed the user's home directory at the same time. The sleep function can be used to stagger the start time of each job, thus making it less likely that all I/O will happen at the same time. This can be done by adding the line

```
sleep `expr $RANDOM % 60`
```

as the first command in a job. This will cause every job in the job array to wait a random number of seconds (up to one minute) before executing later commands.