Simple-Job-Array-Howto

From GridWiki

Contents

Array jobs for clusters running SGE

Last modified by: --Dag 16:51, 5 June 2006 (EDT) This document is based on a PDF written by Kevin Thornton, modified and reproduced on this Wiki with his permission.

The problem

A common problem is that you have a large number of jobs to run, and they are largely identical in terms of the command to run. For example, you may have 1000 data sets, and you want to run a single program on them, using the cluster. The naive solution is to somehow generate 1000 shell scripts, and submit them to the queue. This is not efficient, neither for you nor for the head node.

Array jobs are the solution

There is an alternative on SGE systems – array jobs. The advantages are:

1. You only have to write one shell script
2. You don’t have to worry about deleting thousands of shell scripts, etc.
3. If you submit an array job, and realize you’ve made a mistake, you only have

one job id to qdel, instead of figuring out how to remove 100s of them.
4. You put less of a burden on the head node.

In fact, there are no disadvantages that I'm aware of. Submitting an array job to do 1000 computations is entirely equivalent to submitting 1000 separate scripts, but much less work for you.

The basic commands

In this section, I assume that you prefer the bash shell. To review, a basic SGE job using bash may look like the following:

```
#!sh
#$ -S /bin/bash
~/programs/program -i ~/data/input -o ~/results/output
```

Now, let's complicate things. Assume you have input files input.1, input.2, . . . , input. 10000, and you want the output to be placed in files with a similar numbering scheme. You could use perl to generate 10000 shell scripts, submit them, then clean up the mess later. Or, you could use an array job. The modification to the previous shell script is simple:

```
#!sh
#$ -S /bin/bash
# Tell the SGE that this is an array job, with "tasks" to be numbered 1 to 10000
#$ -t 1-10000
# When a single command in the array job is sent to a compute node,
# its task number is stored in the variable SGE_TASK_ID,
# so we can use the value of that variable to get the results we want:
~/programs/program -i ~/data/input.$SGE_TASK_ID -o ~/results/output.$SGE_TASK_ID
```

That's it. When the above script is submitted, it will find available nodes, and jobs will execute in order of the task IDs specified by the -t option. Also, the array job is subject to all the fair queueing rules. The above script is entirely equivalent to submitting 10000 scripts, but without the mess.

A more complex example

This is a modification of the above which only runs the program if the output file is not present. Please note that these sorts of checks in bash are whitespace-sensitive, which is a common cause of errors. In particular, in the if statement, all spaces must be one single space, regardless of what the typesetting shows:

```
#!sh
#$ -S /bin/bash
# Tell the SGE that this is an array job, with "tasks" to be numbered 1 to 10000
#$ -t 1-10000
# When a single command in the array job is sent to a compute node,
# its task number is stored in the variable SGE_TASK_ID,
# so we can use the value of that variable to get the results we want:
if [ ! -e ~/results/output.$SGE_TASK_ID ]
then
~/programs/program -i ~/data/input.$SGE_TASK_ID -o ~/results/output.$SGE_TASK_ID
fi
```

Pulling data from the i^{th} line of a file

Let's say you have a list of numbers in a file, one number per line. For example, the numbers could be random number seeds for a simulation. For each task in an array job, you want to get the ith line from the file, where i equals SGE_TASK_ID, and use that value as the seed. This is accomplished by using the unix head and tail commands. (Read the man pages for those commands – don't ask me.)

```
#!sh
#$ -S /bin/bash
#$ -t 1-10000
SEEDFILE=~/data/seeds
SEED=$(cat $SEEDFILE | head -n $SGE_TASK_ID | tail -n 1)
~/programs/simulation -s $SEED -o ~/results/output.$SGE_TASK_ID
```

You can use this trick for all sorts of things. For example, if your jobs all use the same program, but with very different command-line options, you can list all the options in the file, one set per line, and the exercise is basically the same as the above, and you only have two files to handle (or 3, if you have a perl script generate the file of command-lines).

As an alternative to using cat, head and tail, the unix sed command could also be used directly (again, see the man pages and experiment a bit).

```
#!/bin/sh
#$ -S /bin/bash -t 1-10000
SEEDFILE=~/data/seeds
SEED=$(sed -n -e "$SGE_TASK_ID p" $SEEDFILE)
~/programs/simulation -s $SEED -o ~/results/output.$SGE_TASK_ID
```

In this example, the '-n' option suppresses all output except that which is explicitly printed (on the line equal to SGE_TASK_ID).

What if you number files from 0 instead of 1?

The -t option will not accept 0 as part of the range, i.e. #$ -t 0-99 is invalid, and

will generate an error. However, I often label my input files from 0 to n − 1. That's easy to deal with:

```
#!sh
#$ -S /bin/bash
# Tell the SGE that this is an array job, with "tasks" to be numbered 1 to 10000
#$ -t 1-10000
let i=$SGE_TASK_ID-1
if [ ! -e ~/results/output.$i ]
then
~/programs/program -i ~/data/input.$i -o ~/results/output.$i
fi
```

Example: R Scripts with Grid Engine Job Arrays

All of the above applies to well-behaved, interactive program. However, sometimes you need to use R to analyze your data. In order to do this, you have to hardcode file names into the R script, because these scripts are not interactive. This is a royal pain. However, there is a solution that makes use of HERE documents in bash. HERE documents also exist in perl, and an online tutorial for them in bash is at http://www.tldp.org/LDP/abs/html/here-docs.html. The short of it is that a HERE document can represent a skeleton document at the end of a shell script. Let's concoct an example. You have 100 data files, labeled data.1 to data.10. Each file contains a single column of numbers, and you want to do some calculation for each of them, using R. Let's use a HERE document:

```
#!sh
#$ -S /bin/bash
#$ -t 1-10
WORKDIR=/Users/jl566/testing
INFILE=$WORKDIR/data.$SGE_TASK_ID
OUTFILE=$WORKDIR/data.$SGE_TASK_ID.out
# See comment below about paths to R
PATHTOR=/common/bin
if [ -e $OUTFILE ]
then
rm -f $OUTFILE
fi
# Below, the phrase "EOF" marks the beginning and end of the HERE document.
# Basically, what's going on is that we're running R, and suppressing all of
# it's output to STDOUT, and then redirecting whatever's between the EOF words
# as an R script, and using variable substitution to act on the desired files.
$PATHTOR/R --quiet --no-save > /dev/null <<EOF
x<-read.table("$INFILE")
write(mean(x\$V1),"$OUTFILE")
EOF
```

So now you can use the cluster to analyze your data – just write the R script within the HERE document, and go from there. As I've only just figured this out, some caveats are necessary. If anyone experiments and figures out something neat, let me know. Be aware of the following:

1. In my limited experience, indenting is important for HERE documents. In particular, it seems that the beginning and end (i.e. both lines containing the term EOF in the above example), must be aligned with the left-hand edge of the buffer (i.e. not indented at all). So, if you use a HERE document in a conditional or control statement, be mindful of this.
2. In the mean command, I escaped the dollar sign with a backslash. In my limited experiments, both mean(x\$V1) and mean(x$V1) seem to work. However, escaping the dollar sign for the read.table command prevents the variable substitution from occurring in the shell, causing R to fail, because the input file named $INFILE cannot be found. In other words, escaping in that context causes the HERE doc to pass $INFILE as a string literal to R, rather than the value stored in the shell variable.
3. This is more useful than just array jobs on an SGE system. If you know bash well enough, you can write a shell script that takes a load of arguments, and processes them with a HERE document. This solves a major limitation with R scripts themselves. You can do the same in perl, too, on your workstation, but you must use a shell language on the cluster.
