Using Ted's Queuing System

Vale Cofer-Shabica
Stratt Group, Brown University

Modified: February 13, 2015 at 17:20

1 Introduction

Ted is a computing cluster that our group shares with several others. Currently we have 8 nodes, each of which has 8 cores, yielding a total of 64 processors available to us at any time (memory allotments are described in table 1). To use them most effectively, we must share them effectively. This document describes how to interact with ted's queuing system, which handles sharing of resources between members of the group.

In addition to this document, Margaret Doll (our system administrator) maintains a collection of useful information about working with ted at: `http://casey.brown.edu/computing/cluster/ted/`.

There are many ways to interact with the queuing system on Ted. And, while the documentation is dense, it is quite thorough. you can read it for yourself by typing:

```
% man submit
```

on the command line. You can even generate a nice pdf to print out for yourself (like the one on the resources page[1]) as follows:

```
% man -t submit | ps2pdf - - > submit.pdf
```

I suggest you read it; it has allowed me to make my research more productive. You can find a list of most available manual pages on the system by issuing the command:

```
% apropos "Grid Engine"
```

[1] `http://casey.brown.edu/grads/valecofershabica/resources/`

Node	RAM / GB
compute-0-0	4
compute-0-1	4
compute-0-2	4
compute-0-3	4
compute-0-10	16
compute-0-11	16
compute-0-12	8
compute-0-13	8

Table 1: Memory Sizes for the Stratt Group Nodes

2 Basic Usage

Suppose you've written and compiled a program called `executable`, which you'd like to run on the cluster. There are several ways you can submit a "job" to the queuing system, but the most common (in our group) is to create a shell file like the one below. A shell file lets you give the grid engine a file to execute as well as a list of options for handling the job. The file below is commented (with # signs) to explain what each option does.

```
# submit.sh
# Shell script for submission to the SGE
#!/bin/bash
#$ -j y                 # join stderr and stdout
#$ -S /bin/bash         # shell to use to execute this file
#$ -N job_name
#$ -m ae                # send an email on (e)nd and/or (a)bort
#$ -M your_name@brown.edu
#$ -o /path/to/output/directory
#$ -wd /path/to/working/directory
echo Start: $(date)
/path/to/executable
# If running an MPI job, change the last line to:
#/share/apps/bin/mpiexec -v /path/to/executable
```

It's important to use the full path (*e.g.*: `/home/username/program/executable`) for the last line as well as for the `-o` and `-wd` options. The `-wd` option is particularly important if you use any relative paths in your program. There are many other options that you can use in this file; they are described in the `submit` manual.

Once you've created a file like the one above, submitting the job is easy: use the `qsub` command as indicated below:

```
% qsub -q stratt submit.sh
% qstat
```

The second command, `qstat`, prints the status of your jobs and is a good way to check that it was submitted.

You can also specify common options for all of your jobs using a file in your home directory called `.sge_request`.

3 Parallel Jobs

The great power of the cluster comes in being able to do many things at once. Our group typically uses "batch" parallelism—executing the same program many times with different inputs. There are at least 2 ways to do this. The traditional approach (in our group) involves using MPI, the Message Passing Interface. A description of how to use MPI is beyond the scope of this document, however, what follows is a description of using it on ted. The other technique involves creating an "array job" with grid engine's own scheduler and usually involves fewer modifications to your program. The array job is *not* a universal solution. It is less flexible than MPI and if you need anything more complex than running the same program with different inputs, you may need to use MPI.

3.1 Using MPI

Xiang Sun supplied an early version of these notes.

Setup: To access the MPI library and binaries, you need to add the following lines to the `.bashrc` file in your home directory:

```
# .bashrc
export PATH=/share/apps/bin:$PATH
export LD_LIBRARY_PATH=/share/apps/lib:$LD_LIBRARY_PATH
```

You also need to create the file `.mpd.conf`, containing the following lines:

```
# .mpd.conf
MPD_USE_ROOT_MPD=1
```

Set the permissions of `.mpd.conf` as follows:

```
% chmod 600 .mpd.conf
```

and while in your home directory, run:

```
% mpdboot
% mpdtrace -l
```

And now you're ready to use MPI!

Usage: For C/C++ programs, add the following include statement to your main file:

```
#include </share/apps/openmpi/ompi/include/mpi.h>
```

And then compile with MPI:

```
% mpicxx  sourcefile.cpp -o executable
```

To run your program on N cores, modify the shell script shown above for parallel jobs (follow the comments) and submit with the command:

```
% qsub -pe stratt N shell_script
```

replacing N with the number of cores. Your job will run as before, but with up to N processors at a time!

3.2 Using an "Array Job"

This is a cheap way to set up parallelism in your code. It involves adding 2 options to your shell file and making a small modification to your code.

The two options are `-t` and `-tc`. From the manual page (edited):

> **-t n[-m[:s]]**
> Submits a so called Array Job, i.e. an array of identical tasks being differentiated only by an index number and being treated by Grid Engine almost like a series of jobs. The option argument to `-t` specifies the number of array job tasks and the index number which will be associated with the tasks. The index numbers will be exported to the job tasks via the environment variable `SGE_TASK_ID`.
>
> The task id range specified in the option argument may be a single number, a simple range of the form n-m or a range with a step size. Hence, the task id range specified by 2-10:2 would result in the task id indexes 2, 4, 6, 8, and 10, for a total of 5 identical tasks, each with the environment variable `GE_TASK_ID` containing one of the 5 index numbers.
>
> The tasks are scheduled independently and, provided enough resources exist, concurrently, very much like separate jobs. Array jobs are commonly used to execute the same type of operation on varying input data sets correlated with the task index number.
>
> **-tc max_running_tasks**
> Allow users to limit concurrent array job task execution. Parameter `max_running_tasks` specifies maximum number of simultaneously running tasks. For example we have running SGE with 10 free slots. We call `qsub -t 1-100 -tc 2 jobscript`. Then only 2 tasks will be scheduled to run even when 8 slots are free.

For example, adding the options `-t 100` and `-tc 8` to your shell file will execute your program 100 times on up to 8 cores at once. For each job the value of the variable `SGE_TASK_ID` will be different. The following example shows how to retrieve its value.

```c
/* sge_task_id_test.c */
#include <stdlib.h>
#include <stdio.h>

int main (int argc, char ** argv){
  char * taskID_string;
  int taskID;

  /* this is the key call, see the documentation with
     man 3 getenv or man 3p getenv */
  taskID_string = getenv("SGE_TASK_ID");

  // Now we make sure we got something
  if (NULL == taskID_string){
    exit(EXIT_FAILURE);
  }
  if (1 != sscanf(taskID_string,  "%d", &taskID)){
    exit(EXIT_FAILURE);
  }

  printf("The value of SGE_TASK_ID is: %d\n", taskID);

  return EXIT_SUCCESS;
}
```

Compile and execute the code with:

```
% gcc sge_task_id_test.c
% SGE_TASK_ID=1 ./a.out
The value of SGE_TASK_ID is: 1
# now try on the grid engine
% qsub -q stratt -b yes -j yes -cwd -o . -t 1:2 \
$(readlink -f a.out)
# and examine the output
% cat a.out.o*
The value of SGE_TASK_ID is: 1
The value of SGE_TASK_ID is: 2
```

Once you have the value of `SGE_TASK_ID` you can use it for anything you want, for instance indicating which in a list of initial conditions you should use in a simulation or anything else.