SLURM assignment

Xoel Mato Blanco

Slide 19

Exercise 1

Write a submit script from scratch. The script should use the following parameters:
- Uses 1 node from the research.q queue
- Creates a file (called text.txt with content: "I have written a submit script"
- Sleeps for 30 seconds
- Lists the contents of the folder
Submit your script and check the output

The slurm script was written with the necessary options included in the first lines and the commands to run worked as expected. To run the script, the following command was used:

```
sbatch exercise_1.slurm
```

Script files: exercise_1.slurm
Output files: exercise_1-output, text.txt

Exercise 2

Submit the same job from the command line (i.e. all sbatch options should be added to the command line together with a script file)

The sbatch options were removed from the script and they were added to the command used:

```
sbatch --job-name=exercise_2 --output=exercise_2-output--nodes=1 \
--partition=research.q exercise_2.slurm
```

Script files: exercise_2.slurm
Output files: exercise_2-output, text.txt

Exercise 3

Do the same without using the script file (i.e. adding a –wrap option)

The following command is equivalent to the previous scripts.

```
sbatch --job-name=exercise_3 --output=exercise_3-output --nodes=1 \
--partition=research.q --wrap="echo 'I have written a submit file' > text.txt;sleep 30;ls"
```

Output files: exercise_3-output, text.txt

Slide 28

Exercise 4

We want to sort several text files (names_0.txt...names_4.txt). Write a solution that uses SLURM job arrays.
(Hint: use the sort command from Linux to build your solution)

I solved this exercise using a redirection of the echo and sort functions to the file itself (shown below). I also added some options for the slurm array job submission. I made use of the variable *SLURM_ARRAY_TASK_ID* to sort all the files:

```
echo $(sort names_$SLURM_ARRAY_TASK_ID.txt) > names_$SLURM_ARRAY_TASK_ID.txt
```

Input files: names_[0-4].txt
Script files: exercise_4.slurm
Output files: exercise_4-output, names_[0-4].txt

Slide 32

Exercise 5

Modify Job4 and turn it into an array job. When does Job5 start now?

To make the job 4 an array job, I added the following option to the script. Besides, I slightly changed the output name so I could answer the question:

```
#SBATCH --array=1-5
#SBATCH --output=basic-job4-output%a
```

Job 5 only starts after all of the jobs in the array have been completed as I could check by the ending times of job4 outputs and the starting time of the job 5.

Script files: dependencies_array.slurm, job[1, 2, 3, 4_array, 5].slurm Output files: basic-job-output, basic-job4-output[1-5]

Exercise 6

Modify individual job scripts so that each job writes its output in a different file.

To do this, I just changed the SBATCH output option in the scripts as follows (eg. job2):

```
#SBATCH --output=basic-job2-output
```

Script files: dependencies.slurm, job[1-5].slurm
Output files: basic-job[1-5]-output

Exercise 7

Write a Python script that does the same as the previous bash script. Which approach (bash or Python script) seems easier for you?

Even though I found python consistently easier to understand, learn and use than bash, bash is more convenient to interact with command line-only tools. Since we are dealing SBATCH and we need to capture its outputs, the python script is full of calls to the *subprocess* module and string manipulations.
Apart from all of that, I run across several problems to properly use the *subprocess* module due to the use different python versions.
For a task like this one, I would rather use a bash script.

Script files: dependencies.py, job[1-5].slurm
Output files: basic-job[1-5]-output

Slide 39

Exercise 8

Write a SLURM script to run an example that uses xargs or parallel commands to parallelize a certain operation. Check that the total execution time is reduced when the operation is parallelized.

I created the following slurm script that will use bwa to index 8 fastq files (they are exactly the same). It calls a script that runs the indexing one at a time, and it also runs it in parallel using xargs commands. It will also output the performance statistics for both operations:

```
#!/bin/bash
#SBATCH --job-name=exercise_8
#SBATCH --output=exercise_8-output
#SBATCH --partition=research.q
#SBATCH --nodelist=aolin23,aolin24
#SBATCH --exclusive
#SBATCH --ntasks=8

perf stat bash bwa_example.sh
rm example*.fastq.*
echo -e "1\n2\n3\n4\n5\n6\n7\n8" | perf stat xargs -n1 -P0 -I{} /usr/bin/bwa index example{}.fastq
rm example*.fastq.*
```

The result shows that there is an **important decrease in the time spent when using parallelisation**: from 1,4s to 0,52s:

```
 Performance counter stats for 'bash bwa_example.sh':

            39,20 msec task-clock:u              #    0,028 CPUs utilized
                0      context-switches:u        #    0,000 K/sec
                0      cpu-migrations:u          #    0,000 K/sec
             3212      page-faults:u             #    0,082 M/sec
         12157129      cycles:u                  #    0,310 GHz                      (91,50%)
         56208559      stalled-cycles-frontend:u #  462,35% frontend cycles idle     (93,77%)
         52283613      stalled-cycles-backend:u  #  430,07% backend cycles idle      (55,40%)
          7920348      instructions:u            #    0,65  insn per cycle
                                                 #    7,10  stalled cycles per insn  (78,90%)
          1792566      branches:u                #   45,728 M/sec                    (73,74%)
           119171      branch-misses:u           #    6,65% of all branches          (86,29%)

      1,400822778 seconds time elapsed

      0,005535000 seconds user
      0,049577000 seconds sys


 Performance counter stats for 'xargs -n1 -P0 -I{} /usr/bin/bwa index example{}.fastq':

            37,63 msec task-clock:u              #    0,072 CPUs utilized
                0      context-switches:u        #    0,000 K/sec
                0      cpu-migrations:u          #    0,000 K/sec
             2972      page-faults:u             #    0,079 M/sec
         10663343      cycles:u                  #    0,283 GHz                      (87,31%)
         54839688      stalled-cycles-frontend:u #  514,28% frontend cycles idle     (85,04%)
         54782220      stalled-cycles-backend:u  #  513,74% backend cycles idle      (57,09%)
          6233328      instructions:u            #    0,58  insn per cycle
                                                 #    8,80  stalled cycles per insn  (81,34%)
          1216223      branches:u                #   32,320 M/sec                    (81,03%)
           112733      branch-misses:u           #    9,27% of all branches          (90,23%)

      0,521315462 seconds time elapsed
```

```
     0,011980000 seconds user
     0,044646000 seconds sys
```

Input files: example[1-8].fastq
Script files: exercise_8.slurm, bwa_example.sh
Output files: exercise_8-output

Exercise 9

Complete the SLURM script provided to run the MPI application that computes prime numbers. Execute it with different configurations regarding number of nodes, number of tasks and number of tasks per node and see the performance variations (you could also try out the –ntasks-per-node option)

I tried out different options for this job. All of them were performed in –exclusive mode. The selected configurations were the following (processes running wew extracted from the output):

```
settings <- read.csv('./files/settings.csv')
settings
```

```
##   id_run nodes tasks.per.node ntasks cpus.per.task processes.running
## 1      1     1              1      1             1                 1
## 2      2     1              2      2             1                 2
## 3      3     2              1      2             1                 2
## 4      4     1              4      4             1                 4
## 5      5     1              4      4             2                 4
## 6      6     2              4      8             2                 8
```

Below, the time spent for the prime numbers found are presented below (only the last one is shown). Each of the rows corresponds to a differently set-up job.

```
results <- read.csv('./files/results.csv')
matrix(results[,19], nrow=6)
```

```
##           [,1]
## [1,] 11.333716
## [2,] 11.336937
## [3,]  7.516307
## [4,]  3.972397
## [5,]  5.959970
## [6,]  2.983815
```

From the results, we can see that the performance increase expected from 1 process to 2 processes is not aparent (compare run 1 and run 2).
However, there is an important increase in performance when using more than one node even if the number of tasks and processes running are exactly the same (compare run 2 and 3).
If we increase the number of tasks (see run 3, 5 and 6) and therefore, the number of processes running, the performance increases are obvious. Nevertheless, using more cpus-per-task surprisingly yielded worse results than using only one cpu per task (compare run 4 and 5).
From my data, I would increase the number of nodes, tasks per node and number of tasks to use as many parallel processes as possible.