slurm

Jakub Widawski

1/21/2020

SLURM ASSIGNMENT

Excercise 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

```
#!/bin/bash
#SBATCH --job-name=kuba
#SBATCH --partition=research.q
#SBATCH --nodes=1
#SBATCH --output=kuba_slurm_out

# write string to file
echo "I have submitted a script" > text.txt

# sleep 30 seconds
sleep 30

# list contents of folder
ls
```

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)

```
sbatch slurm_file --job-name=kuba --partition=research.q \
                  --nodes=1 --output=kuba_slurm_out
```

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

```
sbatch --wrap="echo 'I have submitted a script' > text.txt;sleep 30;ls;" --job-name=kuba --partition=researc
h.q \
                  --nodes=1 --output=kuba_slurm_out
```

Excercise 2

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) Useful SLURM variables: - SLURM_ARRAY_JOB_ID set to the job ID for an array job. - SLURM_ARRAY_TASK_ID set to the task ID inside an array job. - SLURM_ARRAY_TASK_MAX/SLURM_ARRAY_TASK_MIN maximum and minimum task IDs in an array job.

```
#!/bin/bash
#SBATCH --job-name=kuba_exc2
#SBATCH --partition=research.q
#SBATCH --array=0-4
#SBATCH --output=kuba_array.out
echo $(sort names_$SLURM_ARRAY_TASK_ID.txt) > names_$SLURM_ARRAY_TASK_ID.txt
```

Exercise 3

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

Job5 starts after all the array jobs from job4 have been completed or terminated due to the defined depedency (afterany: $jid4)

It will start relatively quicker due to job4 being modified into an array job which splits the tasks in the file.

job4.slurm

```
#!/bin/bash
#SBATCH --job-name=basic-job
#SBATCH --output=basic-job-output
#SBATCH --partition=research.q
#SBATCH --nodelist=aolin23,aolin24
#SBATCH --array=0-4
# print initial date and time
echo "Start JOB 4 at $(date)"
echo "-------------------------"
# # print name of host
hostname
echo "-------------------------"
# sleep 20 seconds
sleep 20
# print initial date and time
echo "End JOB 4 at $(date)"
```

Exercise 4

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

Main file (dependencies):

```
#!/bin/bash
# print initial date and time
echo "Start dependentjob at $(date)"
# first job - no dependencies
dia=$(date)
echo $dia
echo "starting job1"
jid1=$(sbatch --partition=research.q job1.slurm | cut -f 4 -d' ')
echo $jid1
echo "job1 done"
# multiple jobs can depend on a single job
echo "starting job2"
jid2=$(sbatch  --partition=research.q --dependency=afterok:$jid1 job2.slurm | cut -f 4 -d' ')
echo "job2 done"
echo "starting job3"
jid3=$(sbatch  --partition=research.q --dependency=afterok:$jid1 job3.slurm | cut -f 4 -d' ')
echo "job3 done"
echo "starting job4"

# a single job can depend on multiple jobs
jid4=$(sbatch  --partition=research.q --dependency=afterany:$jid2:$jid3 job4.slurm | cut -f 4 -d' ')
echo "job4 done"
echo "starting job5"
jid5=$(sbatch --partition=research.q --dependency=afterany:$jid4 job5.slurm | cut -f 4 -d' ')
echo "job5 done"
# show dependencies in squeue output:
squeue -u $USER -o "%.8A %.4C %.10m %.20E"
# print final date and time
echo "End dependent job at $(date)"
```

job1.slurm

```
#!/bin/bash
#SBATCH --job-name=basic-job
#SBATCH --output=output1.out
#SBATCH --partition=research.q
#SBATCH --nodelist=aolin23,aolin24
# print initial date and time
echo "Start JOB 1 at $(date)"
echo "-------------------------"
# print name of host
hostname
echo "-------------------------"
# sleep 10 seconds
sleep 10
# print initial date and time
echo "End JOB 1 at $(date)"
```

job2.slurm

```
#!/bin/bash
#SBATCH --job-name=basic-job
#SBATCH --output=output2.out
#SBATCH --partition=research.q
#SBATCH --nodelist=aolin23,aolin24
# print initial date and time
echo "Start JOB 2 at $(date)"
echo "-------------------------"
# print name of host
hostname
echo "-------------------------"
sleep 20
# print initial date and time
echo "End JOB 2 at $(date)"
```

job3.slurm

```
#!/bin/bash
#SBATCH --job-name=basic-job
#SBATCH --output=output3.out
#SBATCH --partition=research.q
#SBATCH --nodelist=aolin23,aolin24
# print initial date and time
echo "Start JOB 3 at $(date)"
echo "-------------------------"
# print name of host
hostname
echo "-------------------------"
# sleep 10 seconds
sleep 10
```

job4.slurm

```
#!/bin/bash
#SBATCH --job-name=basic-job
#SBATCH --output=output3.out
#SBATCH --partition=research.q
#SBATCH --nodelist=aolin23,aolin24
#SBATCH --array=0-4
# print initial date and time
echo "Start JOB 4 at $(date)"
echo "-------------------------"
# print name of host
hostname
echo "-------------------------"
# sleep 10 seconds
sleep 10
# print initial date and time
echo "End JOB 4 at $(date)"
```

job5.slurm

```
#!/bin/bash
#SBATCH --job-name=basic-job
#SBATCH --output=output5.out
#SBATCH --partition=research.q
#SBATCH --nodelist=aolin23,aolin24
# print initial date and time
echo "Start JOB 5 at $(date)"
echo "-------------------------"
# print name of host
hostname
echo "-------------------------"
# sleep 10 seconds
sleep 10
# print initial date and time
echo "End JOB 5 at $(date)"
```

Exercise 5

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

*Before running this file please add minconda to make sure the dependencies are satisfied:

```
$ module add miniconda/3
```

Python main file (dependencies):

```
#!/bin/env python3
from datetime import datetime
import subprocess
import re

print("Start dependent job at: ", datetime.today())

# first job - no dependencies
dia=datetime.today()
print(dia)
print("Starting job1")


jid1 = subprocess.check_output("sbatch --partition=research.q job1.slurm".split())
jid1 = ''.join(re.findall(r'\d', str(jid1)))
print(jid1)

jid2 = subprocess.check_output(f"sbatch  --partition=research.q --dependency=afterok:{jid1} job2.slurm".spli
t())
jid2 = ''.join(re.findall(r'\d', str(jid2)))

jid3 = subprocess.check_output(f"sbatch  --partition=research.q --dependency=afterok:{jid1} job3.slurm".spli
t())
jid3 = ''.join(re.findall(r'\d', str(jid3)))

jid4 = subprocess.check_output(f"sbatch  --partition=research.q --dependency=afterany:{jid2}:{jid3} job4.slu
rm".split())
jid4 = ''.join(re.findall(r'\d', str(jid4)))

jid5 = subprocess.check_output(f"sbatch  --partition=research.q --dependency=afterany:{jid4} job5.slurm".spl
it())
jid4 = ''.join(re.findall(r'\d', str(jid5)))

user = subprocess.check_output(["whoami"]).decode("utf-8").strip()
queueinfo = subprocess.check_output(["squeue", "-u", user, "-o", '"%.8A %.4C %.10m %.20E"']).decode("utf-8")
print(queueinfo.replace('"', ''))

# print final date and time
print(f"End dependent job at {datetime.today()}")
```

Exercise 6

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.

Slurm file, no xargs used:

```
#!/bin/bash
#SBATCH --job-name=no_xargs
#SBATCH --partition=research.q
#SBATCH --output=no_xargs_slurm.out

IFS=" "

if [ ! -d "no_xargs_results" ]
then
    mkdir no_xargs_results
fi

for file in ./references/sequence*
do
    file_no_path="${file##*/}"
    echo "Received file: $file_no_path"
    bwa index -p ./no_xargs_results/${file_no_path%.*} $file 2>/dev/null;
done
```

Using xargs:

```
#!/bin/bash
#SBATCH --job-name=xargs
#SBATCH --partition=research.q
#SBATCH --output=xargs_slurm.out

IFS=" "

if [ ! -d "xargs_results" ]
then
    mkdir xargs_results
fi

echo -e "1\n2\n3" | xargs -n 1 -P 3 -I {} bash -c "echo 'Received file: sequence{}.fasta'; bwa index -p ./x
args_results/sequence{} ./references/sequence{}.fasta 2>/dev/null;"
```

We run the each of the versions 5 times and compute runtimes using slurms' "sacct" as shown below:

```
sacct --format=jobid,jobname,nnodes,ncpus,elapsed,state -u biom-2-10 -S2020-01-26-23:35 -E2020-01-26-23:59 -
s CD --allocations
```

```
biom-2-10@aolin-login:~$ sacct --format=jobid,jobname,nnodes,ncpus,elapsed,state
 -u biom-2-10 -S2020-01-26-23:35 -E2020-01-26-23:59 -s CD --allocations
       JobID    JobName   NNodes      NCPUS    Elapsed      State
------------ ---------- -------- ---------- ---------- ----------
45719          no_xargs        1          2   00:00:21  COMPLETED
45720          no_xargs        1          2   00:00:21  COMPLETED
45722          no_xargs        1          2   00:00:19  COMPLETED
45723          no_xargs        1          2   00:00:20  COMPLETED
45724          no_xargs        1          2   00:00:18  COMPLETED
45725             xargs        1          2   00:00:13  COMPLETED
45726             xargs        1          2   00:00:13  COMPLETED
45727             xargs        1          2   00:00:13  COMPLETED
45728             xargs        1          2   00:00:13  COMPLETED
45730             xargs        1          2   00:00:13  COMPLETED
```

Mean program speed over 5 runs:

- no xargs = 19.8
- xargs = 13

Relative performance improvement:

19.8 / 13 ~ 1.52

Using xargs the program was around 1.52 times faster.

Exercise 7

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)

A. 2 nodes, 2 tasks, 1 core:

```
Sat Feb  1 23:39:45 CET 2020
Running prime number generator program on 2 nodes with 2 tasks, each with 1 cores.
01 February 2020 11:39:46 PM

PRIME_MPI
  C/MPI version

  An MPI example program to count the number of primes.
  The number of processes is 2

         N        Pi          Time

         1         0        0.005757
         2         1        0.031392
         4         2        0.000156
         8         4        0.000164
        16         6        0.000135
        32        11        0.000157
        64        18        0.000138
       128        31        0.000159
       256        54        0.000168
       512        97        0.000245
      1024       172        0.000551
      2048       309        0.001586
      4096       564        0.006211
      8192      1028        0.017322
     16384      1900        0.055468
     32768      3512        0.152627
     65536      6542        0.529593
    131072     12251        1.993189
    262144     23000        7.522270

PRIME_MPI - Master process:
  Normal end of execution.

01 February 2020 11:39:56 PM
```

B. 1 node, 1 task, 2 cores:

```
Sat Feb  1 23:38:01 CET 2020
Running prime number generator program on 1 nodes with 1 tasks, each with 2 cores.
01 February 2020 11:38:01 PM

PRIME_MPI
  C/MPI version

  An MPI example program to count the number of primes.
  The number of processes is 1

         N        Pi          Time

         1         0        0.000005
         2         1        0.000000
         4         2        0.000000
         8         4        0.000000
        16         6        0.000001
        32        11        0.000001
        64        18        0.000003
       128        31        0.000010
       256        54        0.000032
       512        97        0.000111
      1024       172        0.000393
      2048       309        0.001376
      4096       564        0.005027
      8192      1028        0.018212
     16384      1900        0.067680
     32768      3512        0.250177
     65536      6542        0.934416
    131072     12251        3.116518
    262144     23000       11.333289

PRIME_MPI - Master process:
  Normal end of execution.

01 February 2020 11:38:17 PM
```

C. 1 node, 2 tasks, 2 cores:

```
Sat Feb  1 23:38:49 CET 2020
Running prime number generator program on 1 nodes with 2 tasks, each with 2 cores.
01 February 2020 11:38:49 PM

PRIME_MPI
  C/MPI version

  An MPI example program to count the number of primes.
  The number of processes is 2

         N        Pi          Time

         1         0        0.000052
         2         1        0.000003
         4         2        0.000002
         8         4        0.000002
        16         6        0.000002
        32        11        0.000002
        64        18        0.000004
       128        31        0.000010
       256        54        0.000032
       512        97        0.000111
      1024       172        0.000385
      2048       309        0.001377
      4096       564        0.005013
      8192      1028        0.018203
     16384      1900        0.067647
     32768      3512        0.250110
     65536      6542        0.934565
    131072     12251        3.118034
    262144     23000       11.343938

PRIME_MPI - Master process:
  Normal end of execution.

01 February 2020 11:39:05 PM
```

D. 2 nodes, 2 tasks, 2 cores:

```
Sat Feb  1 23:39:08 CET 2020
Running prime number generator program on 2 nodes with 2 tasks, each with 2 cores.
01 February 2020 11:39:12 PM

PRIME_MPI
  C/MPI version

  An MPI example program to count the number of primes.
  The number of processes is 2

         N        Pi          Time

         1         0        0.006084
         2         1        0.000173
         4         2        0.000148
         8         4        0.000173
        16         6        0.000143
        32        11        0.000155
        64        18        0.000137
       128        31        0.000152
       256        54        0.000173
       512        97        0.000293
      1024       172        0.000617
      2048       309        0.001929
      4096       564        0.006659
      8192      1028        0.021825
     16384      1900        0.065197
     32768      3512        0.162618
     65536      6542        0.530293
    131072     12251        1.995300
    262144     23000        7.757183

PRIME_MPI - Master process:
  Normal end of execution.

01 February 2020 11:39:23 PM
```

E. 2 nodes, 4 tasks, 2 cores:

```
Sat Feb  1 23:34:14 CET 2020
Running prime number generator program on 2 nodes with 4 tasks, each with 2 cores.
01 February 2020 11:34:17 PM

PRIME_MPI
  C/MPI version

  An MPI example program to count the number of primes.
  The number of processes is 4

        N        Pi          Time

        1         0        0.006951
        2         1        0.000173
        4         2        0.000145
        8         4        0.000175
       16         6        0.000140
       32        11        0.000144
       64        18        0.000127
      128        31        0.000146
      256        54        0.000131
      512        97        0.000194
     1024       172        0.000307
     2048       309        0.001072
     4096       564        0.003836
     8192      1028        0.012821
    16384      1900        0.043444
    32768      3512        0.119051
    65536      6542        0.350974
   131072     12251        1.323173
   262144     23000        4.958100

PRIME_MPI - Master process:
  Normal end of execution.

01 February 2020 11:34:23 PM
Sat Feb  1 23:34:24 CET 2020
```

Comments:

- With two tasks and two cores, increasing number of used nodes from 1 to 2 lead to relative improvement of ~1.46 (11.34s vs 7.76s) (C vs D)
- Using the same number of cores (2) and further multiplying number of tasks (4) (E) leads to further performance incerease, with a relative improvements (2 nodes 4 tasks 2 cores):

~2.28 Over 1 node, 2 task, 2 cores (11.34s vs 4.96s) (C vs E)

~1.56 Over 2 nodes, 2 tasks, 2 cores (7.76s vs 4.96s) (D vs E)

- With the same number of nodes and tasks, increasing number of cores does not lead to any performance improvemments (A, B)