Scaling in a Cluster

Overall Concept

There are two typical issues in solving a problem:

- Existing resources are not enough to handle the current problem.
- Existing resources are just enough to handle the current problem, but the problem needs to be solved at larger scales.

Adding computers to handle the first issue is called **strong scaling**.
Adding computers and increasing the size of the problem to address the second issue is called **weak scaling**.

Speedup

- How fast the program becomes when more computing resources are added.
- Improvement in performance = Reduction in run time

$$S = \frac{t_1}{t_N}$$

Scaling Limitation

- There is a limit to how much resources can be added to improve performance.
- Scalings are limited by proportion of serial (non-parallelizable) code/task within a program/workflow.
  - Strong scaling: Amdahl's Law
  - Weak scaling: Gustafson’s Law

Strong scaling: Amdahl’s Law

Let’s call *s* the proportion of code that cannot be parallelized. Hence *1-s* represents the remainder proportion of code that can be parallelized. The speedup in strong scaling (S) can then be calculated as:

$$S = \frac{1}{\left(s + \frac{1-s}{N}\right)}$$

Weak scaling: Gustafson's Law

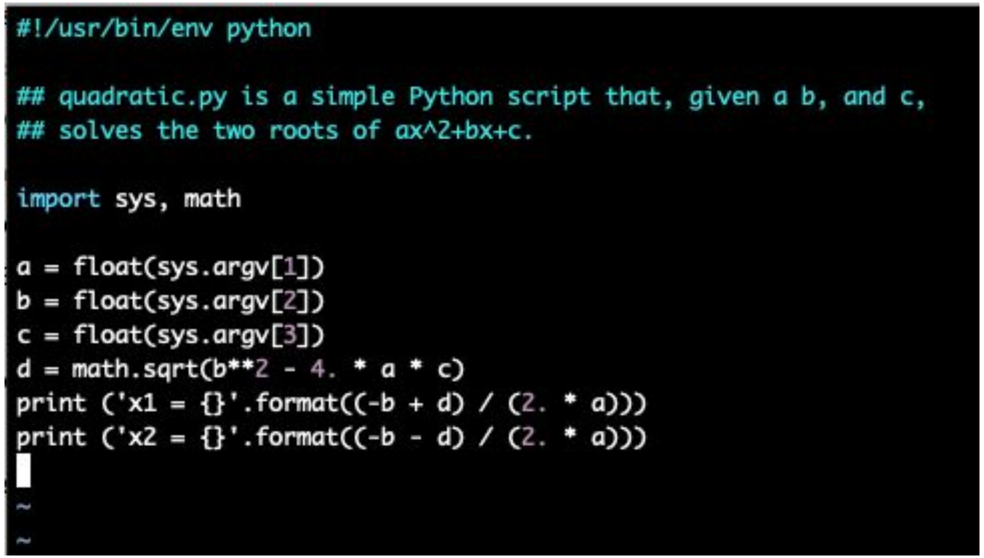
- In weak scaling we scale up the problem as well as resources, we need to consider scaled speedup (SS).
- Let's call *s* the proportion of code that cannot be parallelized. Hence *1-s* represents the remainder proportion of code that can be parallelized. The scaled speedup in strong scaling can then be calculated as:

$$SS = s + (1 - s) \times N$$

Scaling in a cluster (1)

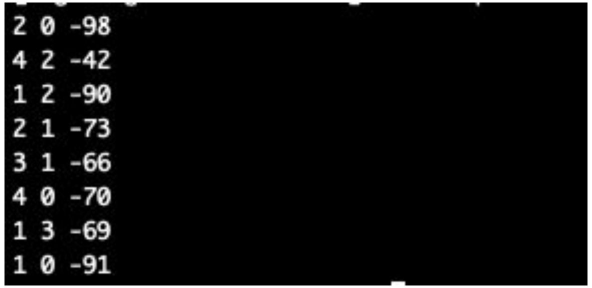
- Running multiple instances of a program in parallel
- Scheduler support: Job Array
- Module support: gnu-parallel
- No code modification is needed.

Quadratic Calculation



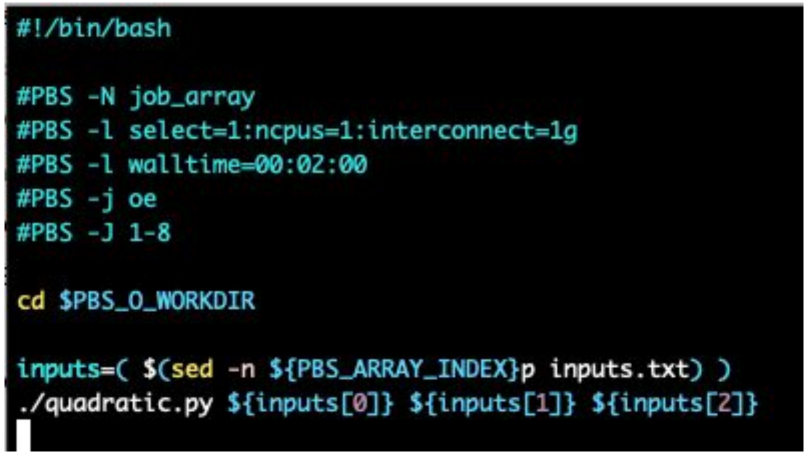
```
#!/usr/bin/env python

## quadratic.py is a simple Python script that, given a b, and c,
## solves the two roots of ax^2+bx+c.

import sys, math

a = float(sys.argv[1])
b = float(sys.argv[2])
c = float(sys.argv[3])
d = math.sqrt(b**2 - 4. * a * c)
print ('x1 = {}'.format((-b + d) / (2. * a)))
print ('x2 = {}'.format((-b - d) / (2. * a)))

~
~
```

Input files for 8 quadratic functions



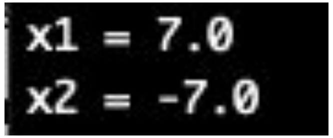
```
2 0 -98
4 2 -42
1 2 -90
2 1 -73
3 1 -66
4 0 -70
1 3 -69
1 0 -91
```

Job Array



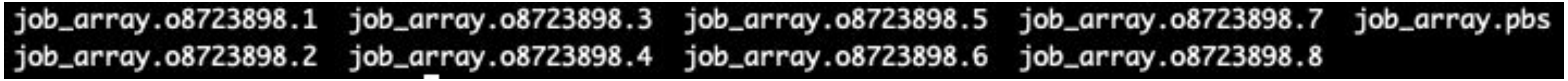
```
#!/bin/bash

#PBS -N job_array
#PBS -l select=1:ncpus=1:interconnect=1g
#PBS -l walltime=00:02:00
#PBS -j oe
#PBS -J 1-8

cd $PBS_O_WORKDIR

inputs=( $(sed -n ${PBS_ARRAY_INDEX}p inputs.txt) )
./quadratic.py ${inputs[0]} ${inputs[1]} ${inputs[2]}
```

Output in one job



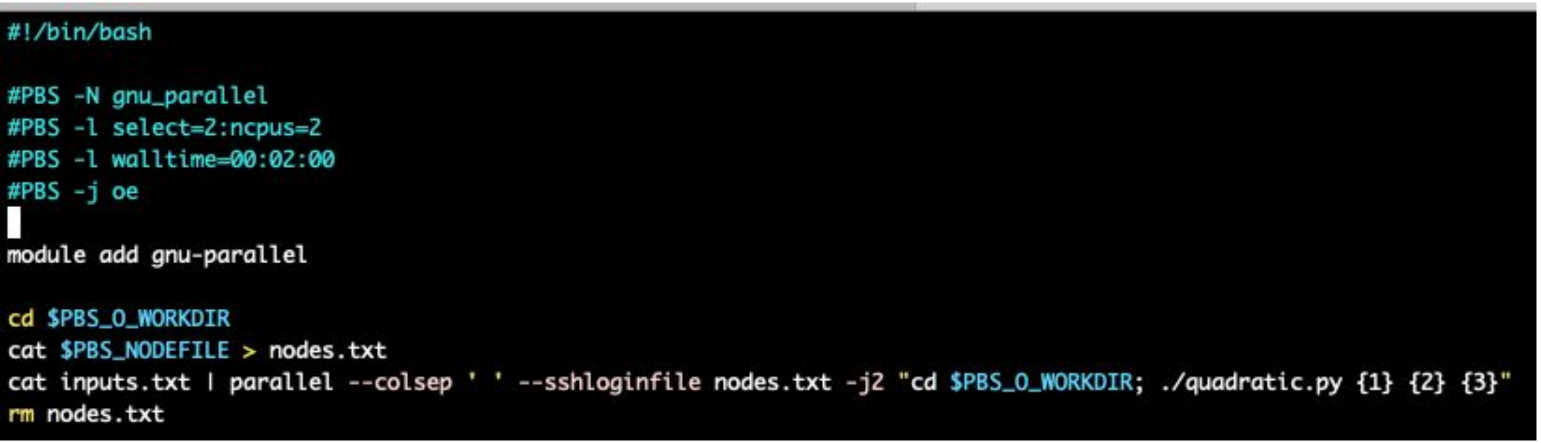
```
x1 = 7.0
x2 = -7.0
```

List of jobs’ output files from the job array submission

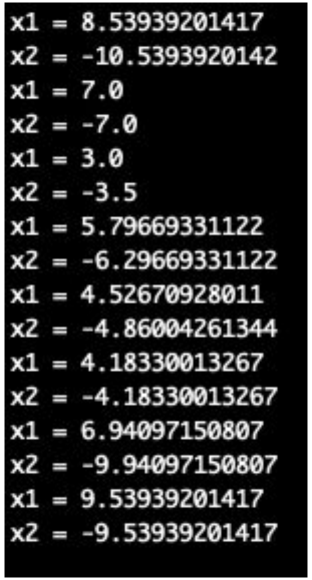


```
job_array.o8723898.1  job_array.o8723898.3  job_array.o8723898.5  job_array.o8723898.7  job_array.pbs
job_array.o8723898.2  job_array.o8723898.4  job_array.o8723898.6  job_array.o8723898.8
```

GNU-Parallel



```
#!/bin/bash

#PBS -N gnu_parallel
#PBS -l select=2:ncpus=2
#PBS -l walltime=00:02:00
#PBS -j oe

module add gnu-parallel

cd $PBS_O_WORKDIR
cat $PBS_NODEFILE > nodes.txt
cat inputs.txt | parallel --colsep ' ' --sshloginfile nodes.txt -j2 "cd $PBS_O_WORKDIR; ./quadratic.py {1} {2} {3}"
rm nodes.txt
```



```
x1 = 8.53939201417
x2 = -10.5393920142
x1 = 7.0
x2 = -7.0
x1 = 3.0
x2 = -3.5
x1 = 5.79669331122
x2 = -6.29669331122
x1 = 4.52670928011
x2 = -4.86004261344
x1 = 4.18330013267
x2 = -4.18330013267
x1 = 6.94097150807
x2 = -9.94097150807
x1 = 9.53939201417
x2 = -9.53939201417
```

Output from 8 quadratic functions within one job.