Serial farming on Graham

Sergey Mashchenko
SHARCNET

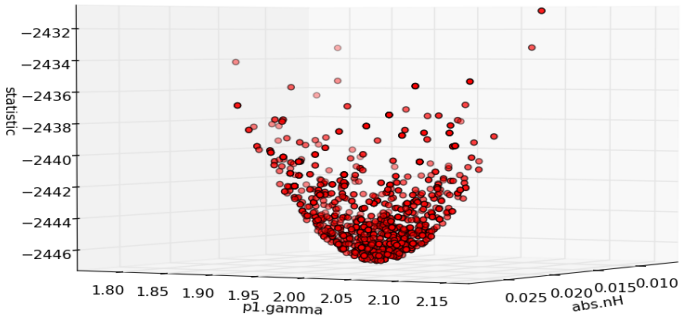
Outline

- Introduction
- Serial farming on Graham
  - Simple for loop
  - Array job
  - Bundling up
  - Whole node farming
  - Exit status processing
  - Fully automated farming with the META package
- Conclusions

Introduction

What is serial farming?

- “Farming” is running a large number of cluster jobs which are effectively independent of each other but are solving different aspects of the same problem.
  - If the jobs are serial (single cpu-core), we call it “serial farming”. One can also have “MPI farming”, “GPU farming” etc.
  - The independence means that the jobs in the farm can run in any order, concurrently or sequentially.
  - A typical example is solving problems using a Monte Carlo approach.



Advantages

- As the throughput of serial farms grows linearly with the number of jobs (due to the lack of parallel overheads), such applications have perfect (100%) scalability and are very efficient.
- The independence of jobs (and typically a small amount of resources per job) in a serial farm makes farming very scheduler-friendly.
- With serial farms, one can very easily upscale and downscale the size of the problem you are solving, without changing the code.

Challenges

- Many repetitive commands required
  - Solution: automation scripts
- Often serial farming jobs are very short (<20 min), meaning large scheduler overheads and poor efficiency
  - Solution: bundling up individual jobs into meta jobs. This also helps to bypass the limit on the number of jobs (5000 for Graham).
- Sometimes bundling up works poorly (when individual jobs have a large range of runtimes)
  - Solution: using dynamic workload balancing

Challenges (2)

- With a large number of jobs a few might fail, and it can be time consuming to track them down and re-run
  - Solution: smart scripts can track the exit status of all jobs, and resubmit those that failed or never ran
- Running more than one farm concurrently can be messy and fail-prone
  - Solution: smart scripts can handle many concurrent farms

Serial farming on Graham

Setup

- In all the examples presented here, the following simple way to describe the work for a serial farm is used
  - Each instance of a serial farm computation is called a “case”.
  - All cases are described in a text file “table.dat”, one line per case
  - Each line is a full command to be executed for each case. E.g.,

```
/home/user/bin/code arg1 arg2 arg3

code < input1.txt

cp -f ../input_${ID} input; code
```

Setup (2)

- Different scripting languages can be used to write farming scripts
- I am using Bash scripting in my examples, in part because it simplifies scripts (because in SLURM – the Graham scheduler – Bash is used internally in job scripts).
- To copy all the examples to your home directory, execute the following on Graham:

```
$ cp -pr ~syam/Serial_farming  ~
```

1. A simple For loop

```
$  cd ~/Serial_farming/1_FOR_LOOP

$  ./for_loop.sh

Submitted batch job 1595103
Submitted batch job 1595108
Submitted batch job 1595112
Submitted batch job 1595115

$ cat *.out

1; sleep 4
2; sleep 27
...
```

for_loop.sh

```
# Simplest case - using for loop to submit a serial farm
# The input file table.dat contains individual cases - one case per line

export TABLE=table.dat

# Total number of cases (= number of jobs to submit):
N_cases=$(cat "$TABLE" | wc -l)

# Submitting one job per case using the for loop:
for ((i=1; i<=$N_cases; i++))
  do
  # Using environment variable I_FOR to communicate the case number to individual jobs:
  export I_FOR=$i
  sbatch job_script.sh
  done
```

job_script.sh

```
#!/bin/bash
# Here you should provide the sbatch arguments to be used in all jobs in this serial farm
#SBATCH -t 0-00:10
#SBATCH --mem=1000
#SBATCH -A def-syam

# Extracing the $I_FOR-th line from file $TABLE:
LINE=`sed -n ${I_FOR}p "$TABLE"`

# Echoing the command (optional), with the case number prepended:
echo "$I_FOR; $LINE"

# Executing the command:
eval "$LINE"
```

Tips

1) It is convenient to use environment variables to pass arguments to the job script, e.g.:

```
export I_FOR=$i
```

2) One can use the “sed” command to extract the $i-th line of a text file $TABLE:

```
LINE=`sed -n ${i}p "$TABLE"`
```

3) Using “eval” helps to process properly semicolumns, redirections etc when executing shell commands contained in a variable:

```
eval "$LINE"
```

Tips (cont.)

4) You can kill all your jobs on the cluster using

```
$ scancel  -u user_name
```

A simple For loop summary

- A very simple way to achieve automation at the job submission stage
- Lacking all other features which are important for serial farming (no support for short job bundling, no job exit status processing, no automation for jobs query and killing, no support for running concurrent farms)

2. Array jobs

```
$  cd ~/Serial_farming/2_JOB_ARRAY

$  ./job_array.sh

Submitted batch job 1596114

$ cat *.out
1; sleep 4
2; sleep 27
...
```

job_array.sh

```
#!/bin/bash
# Using the array job feature of SLURM to submit a serial farm.
# The input file table.dat contains individual cases - one case per line
# A simple test table can be generated with
# for ((i=1; i<=4; i++)); do echo "sleep $(($RANDOM % 30))"; done > table.dat

export TABLE=table.dat

# Total number of cases (= number of jobs to submit):
N_cases=$(cat "$TABLE" | wc -l)

# Submitting an array job to the scheduler:
sbatch --array=1-$N_cases job_script.sh
```

job_script.sh

```
#!/bin/bash
# Here you should provide the sbatch arguments to be used in all jobs in this serial farm
#SBATCH -t 0-00:10
#SBATCH --mem=1000
#SBATCH -A def-syam

# Case number is given by $SLURM_ARRAY_TASK_ID environment variable:
i=$SLURM_ARRAY_TASK_ID

# Extracing the $i-th line from file $TABLE:
LINE=`sed -n ${i}p "$TABLE"`
# Echoing the command (optional), with the case number prepended:
echo "$i; $LINE"
# Executing the command:
eval "$LINE"
```

Tips (cont.)

5) In array jobs, the SLURM variable $SLURM_ARRAY_TASK_ID should be used to identify the specific case inside the job script, e.g.:

```
i=$SLURM_ARRAY_TASK_ID

LINE=`sed -n ${i}p "$TABLE"`
```

Array jobs summary

- A slightly more elegant way to automate job submission (compared to a simple for loop method).
- Each individual job from an array job is still submitted and allocated one by one internally, by SLURM.
  - The same limit on the total number of jobs (5000 on Graham)
  - The same scheduler overhead per job (meaning jobs <20 min long are very inefficient)
- All other important features of serial farming are still missing.

3. Bundling up

```
$  cd ~/Serial_farming/3_BUNDLE_UP

$  ./bundle_up.sh

Submitting 6 jobs...
Submitted batch job 1600024
Submitted batch job 1600027
...

$ cat *.out

1; sleep 3
2; sleep 9
...
```

bundle_up.sh

```
#!/bin/bash
# Bundling up multiple short cases (simplest - static - workload distribution).
# Four environment variables are passed to the job script - TABLE, N_CASES, N_JOBS, I0.
export TABLE=table.dat
# Target bundle factor (number of cases per job; can be a bit bigger for individual jobs):
N_bundle=15
# Total number of cases (= number of jobs to submit):
export N_CASES=$(cat "$TABLE" | wc -l)
# Number of jobs (rounded to the smaller integer value - this ensures that the actual bundle factor is never smaller than $N_bundle):
export N_JOBS=$(($N_CASES / $N_bundle))
echo "Submitting $N_JOBS jobs..."

for ((i=1; i<=$N_JOBS; i++))
  do
  export I0=$i
  sbatch job_script.sh
  done
```

job_script.sh

```
!/bin/bash
#SBATCH -t 0-00:10
#SBATCH --mem=1000
#SBATCH -A def-syam

# Simple way to achieve the best static workload distribution of $N_CASES cases over $N_JOBS jobs,
# with the number of cases per job equal or larger that the target $N_bundle:
for ((i=$I0; i<=$N_CASES; i=i+$N_JOBS))
  do
  LINE=`sed -n ${i}p "$TABLE"`

  echo "$i; $LINE"

  eval "$LINE"
  done
```

Tips (cont.)

6) Choose the N_bundle value such that the bundled job length would get larger than ~20 minutes, and don't forget to modify the “#SBATCH -t“ argument accordingly.
7) In SLURM, job scripts are just BASH scripts, so one can use any BASH commands there (for loops etc.).
8) “for (i=thread_ID; i<M; i+=N_threads)” is the classical way to achieve the best static workload balancing when computing M cases in N_threads concurrent threads/processes.

Bundling Up summary

- Be careful – bad table.dat or N_bundle can result in a huge number of jobs
- In addition to job submission automation, this approach introduces the second important serial farming feature which is needed when job runtimes are short (<20 minutes).
- One can dramatically improve throughput of very short jobs (<5 minutes).
- Also, the 5000 jobs limit can be circumvented.

4. Whole node farming

$ cd ~/Serial_farming/4_WHOLE_NODE

$ ./whole_node.sh

Submitted batch job 1601528

$ cat out.*

1; sleep 3
33; sleep 9
65; sleep 2
...

whole_node.sh

```
#!/bin/bash
# Serial farm running on a whole node (also using an optional
# bundling up - for short jobs).
# The input file table.dat containts individual cases - one case per line
# One environment variable is passed to the job script - TABLE.

export TABLE=table.dat

sbatch job_script.sh
```

job_script.sh

```
# Total memory (for all processes)
#SBATCH --mem=100G
# These switches ensure a whole node reservation:
#SBATCH --nodes=1
#SBATCH --tasks-per-node=32

N_cases=$(cat "$TABLE" | wc -l)

# Number of threads = number of cpu cores on the node:
N_threads=$SLURM_TASKS_PER_NODE

# Running $N_threads concurrently on the node:
for ((i=1; i<=$N_threads; i++))
  do
  ./single_thread.sh $i $N_cases $N_threads &> out.$i &
  done

# Without "wait", the job would finish as soon as all the threads are launched.
# "wait" forces the job to wait until all the background processes are done.
wait
```

single_thread.sh

```
#!/bin/bash
# Task performed by a single thread.  $TABLE is an environment variable.

thread_ID=$1
N_cases=$2
N_threads=$3

# Simple way to achieve the best static workload distribution of $N_cases cases over $N_threads threads:
for ((i=$thread_ID; i<=$N_cases; i=i+$N_threads))
  do
  LINE=`sed -n ${i}p "$TABLE"`

  echo "$i; $LINE"

  eval "$LINE"
  done
```

Tips (cont.)

9) Running multiple serial processes on a whole node is accomplished by
   (a) Requesting a whole node (sbatch switches --nodes=N and --tasks-per-node=32)
   (b) Sending all the processes to background with “&”
   (c) Adding “wait” command at the end of the job script

Whole Node summary

- Often whole node (“by-node”) allocation is faster than by-core allocation, so this method can be more efficient.
- Also, this method is better if the jobs need shared access to the node-local fast SSD drive (at /tmp).
- Otherwise, this method is less flexible, as only 32 cpu cores can participate in a farm.

5. Exit status processing

```
$  cd ~/Serial_farming/5_STATUS

$  ./submit_status.sh

Submitted batch job 1601528


$ cat out.*
1; sleep 3
33; sleep 9
65; sleep 2
...
```

submit_status.sh

```
#!/bin/bash
# Capturing the exit status of individual cases, with bundling up.

export TABLE=table.dat
# Target bundle factor (number of cases per job; can be a bit bigger for individual jobs):
N_bundle=15
# Total number of cases (= number of jobs to submit):
export N_CASES=$(cat "$TABLE" | wc -l)
# Number of jobs (rounded to the smaller integer value - this ensures that the actual bundle factor is never smaller than $N_bundle):
export N_JOBS=$(($N_CASES / $N_bundle))

echo "Submitting $N_JOBS jobs..."

\rm status.*

for ((i=1; i<=$N_JOBS; i++))
  do
  export I0=$i
  sbatch job_script.sh
  done
```

job_script.sh

```
for ((i=$I0; i<=$N_CASES; i=i+$N_JOBS))
  do
  # Extracting the $i-th line from file $TABLE:
  LINE=`sed -n ${i}p "$TABLE"`

  # Echoing the command (optional), with the case number prepended:
  echo "$i $LINE"

  # Executing the command (if using multiple commands the code execution should be the last one):
  eval "$LINE"

  # Exit status of the code:
  STATUS=$?

  # Storing the ID and the exit status for this case in a separate for each job status.* file:
  echo $i $STATUS >> status.$I0
  done
```

Tips (cont.)

10) Read the value of “$?” variable to find out the status of the previous command.

11) Many research codes return a good status (0) even if there was a problem. One can add more intelligence to the job_script.sh file to handle your code specifics.

- For example, if your code is expected to create a non-zero file (say, “output.dat”) in the course of a normal run, you can use that to fine-tune the STATUS variable in job_script.sh:

```
STATUS=$?
if test ! -s output.dat
  then
  STATUS=1
  fi
```

Status Processing summary

- This approach allows capturing of code exit status for all the cases in your serial farm. This should greatly simplify the task of identifying failed cases.
- The next obvious improvement would be to have the ability to automatically re-submit all the cases which failed, and also those which never ran. This will be accomplished in our next (and final) example – META scripts.

6. META package

```
$  mkdir ~/bin

$  mv ~/Serial_farming/6_META/bin/*  ~/bin

$  export PATH=~/bin:$PATH

$  cd ~/Serial_farming/6_META
```

Customize the files table.dat, job_script.sh, and single_case.run in ~/Serial_farming/6_META to your specific case.

Overview

- META is a package which covers all the essential serial farming needs:
  - Automates farm submission, query, killing, analysis and re-submission of the cases which failed or never ran.
  - Supports both long (> 20 minutes; one case per job) and short (<20 minutes; multiple cases per job) case runtimes.
  - In the multiple cases per job mode, uses dynamic workload balancing between running jobs (great when cases' runtimes vary a lot)
  - Any number of independent farms can be run on the cluster concurrently.
  - Has some advanced features.

List of META commands

- submit.run N : Submits the farm either in a one case per job mode (N=-1) or a multiple cases per job mode (N>0 is the number of the meta jobs to run)
- query.run : Prints a one line summary of the farm's state
- list.run : Lists all queued and running jobs of the farm
- kill.run : Kills the farm
- prune.run : Kills all queued jobs of the farm
- Status.run : Lists exit statuses (one line per case). If “-f” switch is used, places all failed cases at the bottom.
- resubmit.run N : Resubmits all cases which failed or never ran.
- clean.run : deletes all files/subdirectories in the farm directory except for table.dat, job_script.sh, and single_case.run.

job_script.sh

```
#!/bin/bash
# Here you should provide the sbatch arguments to be used in all jobs in this
# serial farm
# It has to contain the runtime switch (either -t or --time):
#SBATCH -t 0-00:10
#SBATCH --mem=1000
#SBATCH -A def-syam

# Don't change this line:
task.run
```

single_case.run

```
# ++++++++++++++++++++++  This part can be customized:  ++++++++++++++++++++++++
#  Here:
#  $ID contains the case id from the original table
#  $COMM is the line corresponding to the case $ID in the original table, without the ID field

mkdir -p RUN$ID
cd RUN$ID

# Executing the command (if using multiple commands the code execution should be the last one):
eval "$COMM" &> out.log
# Exit status of the code:
STATUS=$?

cd ..
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Some details

- One should create a separate directory for each farm you want to run:

```
$  cp -pr ~/Serial_farming/6_META  ~/Farm1
$  cp -pr ~/Serial_farming/6_META  ~/Farm2
   ...
```

- All META commands should be executed inside the corresponding farm's directory.
- The only obligatory SBATCH switch in job_script.sh is “#SBATCH -t”. Additional switches can be supplied via optional command line arguments to submit.run:

```
$  submit.run  N  [additional sbatch arguments]
```

Conclusions

- Serial farming can be messy, time consuming, and error prone if proper tools are not used.
- This webinar presented a few different examples of farming scripts, starting from a simplest for loop job submission automation, and ending in the all-in-one META package.
- The script examples and tips provided should help you to become a seasoned serial farmer, without spending too much time and efforts.
- Questions? Email me to syam@sharcnet.ca