Serial Jobs (PHY1610 lecture 18)

Ramses van Zon

March 17, 2022

Today's class

- Approaches for dealing with serial jobs.
- Mini-intro to SciNet: Niagara/Teach Cluster.
- GNU parallel.
- RAMdisk.

Why Parallel Programming?

To review:

- Your desktop has many cores, as do the nodes on SciNet's clusters: i.e. Teach and Niagara clusters.
- Your code would run a lot faster if it could use all of those cores simultaneously, on the same problem.
- Or even better, use cores on many nodes simultaneously.
- So we need to adjust our programming accordingly: parallel programming.

There are several different approaches to parallel programming.

1

Serial Jobs

Serial programming

Sometimes your code is serial, meaning it only runs on a single processor, and that's as far as it's going to go. There are several reasons why we might not push the code farther:

- The problem involves a parameter study; each iteration of the parameter study is very swift; each iteration is independent; but many many need to be done.
- The algorithm is inherently serial, and there's not much that can be done about it.
- You're running a commercial code, and don't have the source code to modify.
- You're graduating in six months, and don't have time to parallelize your code.

Sometimes the best course of action is to let your code be serial, and run the serial processes in parallel. That's the topic of today's class.

A typical (computational) situation

- You have a serial code.
- Your code takes a set of parameters, either from a file or from the command line.
- The code runs in a reasonably short amount of time (*e.g.* minutes to hours).
- You have a large parameter space you want to search, which means hundreds or thousands of combinations of values of parameters.
- You'd probably like some control over your jobs, things like error checking, fault tolerance, *etc.*
- You want to run your code on SciNet: Niagara, Teach, . . .

How do we setup a set of simultaneous calculations efficiently?

We are concerned. . .

What are SciNet's concerns?

What concerns does *SciNet* have about you running serial jobs on our clusters?

- On Niagara, scheduling is done by node. A 40-core node with 188GB of RAM. Use your resources efficiently.
  - Use all of the processors on the nodes you've been given
    or
  - use all the memory you've been given efficiently (if 40 instances of your serial job won't fit in memory).

  This almost certainly means having multiple subjobs running simultaneously on your nodes.
- Don't do heavy I/O.
  - Don't try to read thousands of files.
  - Don't generate thousands of files.

Using resources efficiently helps you get more results, or get them faster. Not abusing the filesystem is everyone's responsibility.

What are your options to running in parallel?

So how might we go about running multiple instances the code (subjobs) simultaneously?

- Write a script from scratch which launches and manages the subjobs.
- For code written in Python, use multiprocessing to manage the subjobs.
- For code written in R, use the parallel R utilities to manage the subjobs.
- Use an existing script, such as GNU Parallel[1] to manage the subjobs.

We'll discuss the first and last options in this class.

[1] Tange (2011): GNU Parallel - The Command-Line Power Tool, ;login; The USENIX Magazine, February 2011:42-47.

2

Using SciNet (and other HPC) clusters

Teach cluster: login and compute nodes

- The login node, teach01, is where you develop, edit, compile, prepare and submit jobs.
- Real computations should run on the compute nodes
- The login node and compute nodes have the same architecture, operating system, and software stack.
- To run on the compute nodes, you must submit a batch job.

Storage Systems and Locations on SciNet systems

Home and scratch

You have a home and scratch directory on the system, whose locations will be given by

$HOME=/home/g/groupname/username

$SCRATCH=/scratch/g/groupname/username

Use these convenient variables!

```
teach01:~$ pwd
/home/s/scinet/rzon
teach01:~$ cd $SCRATCH
teach01:~$ pwd
/scratch/s/scinet/rzon
```

Storage Systems and Locations on SciNet systems

Home and scratch

You have a home and scratch directory on the system, whose locations will be given by

`$HOME=/home/g/groupname/username`

`$SCRATCH=/scratch/g/groupname/username`

Use these convenient variables!

```
teach01:~$ pwd
/home/s/scinet/rzon
teach01:~$ cd $SCRATCH
teach01:~$ pwd
/scratch/s/scinet/rzon
```

Project

Users from groups with a RAC allocation will also have a project directory.

`$PROJECT=/project/g/groupname/username`

Storage Limits on SciNet

location	quota	expiration time	backed up	on login	on compute
$HOME	100 GB	never	yes	yes	read-only
$SCRATCH	25 TB	2 months	no	yes	yes
$PROJECT	by allocation	never	yes	yes	yes

- Compute nodes do not have local storage, but they have a lot of memory, which you can use as if it is local disk `/dev/shm`.
- Backup means a recent snapshot, not an achive of all data that ever was.

Testing

You really should test your code before you submit it to the cluster to know if your code is correct and what kind of resources you need.

- Small test jobs can be run on the login nodes.

  Rule of thumb: couple of minutes, taking at most about 1-2GB of memory, couple of cores.
- You can run the the `ddt` debugger on the login nodes after `module load ddt`.
- The ddt module also gives you the `map` performance profiler.
- Short tests that do not fit on a login node, or for which you need a dedicated node or set of cores, request an interactive debug job with the debugjob command

```
teach01:~$ debugjob -n C
debugjob: Requesting 1 nodes with N tasks for 240 minutes and 0 seconds
SALLOC: Granted job allocation 202753
SALLOC: Waiting for resource configuration
SALLOC: Nodes teach35 are ready for job
teach35:~$
```

Here, C is the number of cores.

The Scheduler

- The scheduler is a program that organizes the work load on the cluster.
- You submit a request to the scheduler, and it will find the right moment for your request to run.
- It does that by looking at the resources available, your priority, times and resources requested, . . .
- It is quite a complicated process, with many variables to take into consideration.
- Even when we run 'interactively' (e.g.`debugjob`), we are requesting resources to the scheduler
- We refer to the 'resource request'+'script' as a jobs.

What we will see next is how to communicate with the scheduler and request resources to run our programs.

Submitting jobs

- Teach (as well as all other SciNet and CC systems) uses SLURM as its job scheduler.
- You submit jobs from a login node by passing a script to the sbatch command:

```
teach01:~$ cd $SCRATCH
teach01:scratch/rzon$ sbatch jobscript.sh
```

- This puts the job in the queue. It will run on the compute nodes in due course.

Submitting jobs

- Teach (as well as all other SciNet and CC systems) uses SLURM as its job scheduler.
- You submit jobs from a login node by passing a script to the sbatch command:

```
teach01:~$ cd $SCRATCH
teach01:scratch/rzon$ sbatch jobscript.sh
```

- This puts the job in the queue. It will run on the compute nodes in due course.

Keep in mind:

Submitting jobs

- Teach (as well as all other SciNet and CC systems) uses SLURM as its job scheduler.
- You submit jobs from a login node by passing a script to the sbatch command:

```
teach01:~$ cd $SCRATCH
teach01:scratch/rzon$ sbatch jobscript.sh
```

- This puts the job in the queue. It will run on the compute nodes in due course.

Keep in mind:

- You must use all requested cores.
- Maximum walltime is 24 hours.
- Jobs must write to your scratch directory (home is read-only on compute nodes).
- Compute nodes have no internet access

  Download data you need beforehand on a login node.
- Different clusters have different restrictions.

Example submission script (serial job)

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntask=1
#SBATCH --cpus-per-task=1
#SBATCH --time=1:00:00
#SBATCH --job-name serial_job
#SBATCH --output=serial_output_%j.txt
#SBATCH --mail-type=FAIL

module load gcc/9.2.0

./serial_code 1
```

```
teach01:scratch$ sbatch serial_job.sh
```

- First line indicates that this is a bash script.
- Lines starting with `#SBATCH` go to SLURM.
- sbatch reads these lines as a job request (which it gives the name `serial_job`).
- In this case, SLURM looks for one core to be used for one task, for 1 hour.
- Submit from /scratch, as /home is read-only.
- Once SLURM founds a node with an unused core, the script is run there:
  - Loads modules;
  - Sets an environment variable;
  - Runs the `serial_code` app with argument "1".

Monitoring jobs - command line

Once the job is incorporated into the queue, there are some command you can use to monitor its progress.

- "`squeue`" to show the job queue ("`squeue --me`" for just your jobs);
- "`squeue -j JOBID`" to get information on a specific job
  (alternatively, "`scontrol show job JOBID`", which is more verbose).
- "`squeue --start -j JOBID`" to get an estimate for when a job will run.
- "`jobperf JOBID`" to get an instantaneous view of the cpu+memory usage of a running job's nodes.
- "`scancel JOBID`" to cancel the job.
  "`scancel -u USERID`" to cancel all your jobs (careful!).
- "`sinfo -p compute`" to look at available nodes.
- "`sacct`" to get information on your recent jobs.

3

Running batches of serial jobs

Option 1: Job arrays

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntask=1
#SBATCH --cpus-per-task=1
#SBATCH --time=1:00:00
#SBATCH --job-name serial_job
#SBATCH --output=serial_output_%j.txt
#SBATCH --mail-type=FAIL
#SBATCH --array=1-100

module load gcc/9.2.0

./serial_code $SLURM_ARRAY_TASK_ID
```

```
teach01:scratch$ sbatch serial_array_job.sh
```

- The scheduler can be used to schedule a number of serial jobs. These may or may not run in parallel.
- Job arrays is what you would use in that case.
- The option `--array` (or `-a`) set the number of jobs.
- Each jobs of a job array gets a different value for `$SLURM_ARRAY_TASK_ID`.
- Warning: Not suitable for serial jobs on Niagara!
- Warning 2: Not suitable for large number of short serial jobs.

Option 2: Start background subjobs within the job

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=16
#SBATCH --time=1:00:00
#SBATCH --job-name=serialx16

module load gcc/9.2.0

#Run the code on 16 cores
./serial_code 1 &
./serial_code 2 &
./serial_code 3 &
...
./serial_code 15 &
./serial_code 16 &

#Tell the script to wait, or all
#the subjobs get killed immediately.
wait
```

If your subjobs all take the same amount of time, there's nothing in principle wrong with this submission script.

- Does it use all the cores?
  Yes.
- Will any cores be wasting time not running?
  Only if the subjobs take varying amounts of time.

Option 2: Start background subjobs within the job

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=16
#SBATCH --time=1:00:00
#SBATCH --job-name=serialx16

module load gcc/9.2.0

#Run the code on 16 cores
./serial_code 1 &
./serial_code 2 &
./serial_code 3 &
...
./serial_code 15 &
./serial_code 16 &

#Tell the script to wait, or all
#the subjobs get killed immediately.
wait
```

If your subjobs all take the same amount of time, there's nothing in principle wrong with this submission script.

- Does it use all the cores?
  Yes.
- Will any cores be wasting time not running?
  Only if the subjobs take varying amounts of time.

But if your subjobs take variable amounts of time, then the longer jobs will keep running, while other cores do nothing.

We need to balance the load.

Also, what if there are 100 cases to do?

Why not write your own load balancer?

A write-your-own attempt to balance loads could look like this.

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=16
#SBATCH --time=1:00:00
#SBATCH --job-name=serialx16

module load gcc/9.2.0

dobysixteen() {
  while [ -n "${16}" ]
  do
    ./serial_code ${1} &
    ./serial_code ${2} &
    ./serial_code ${3} &
    ...
    ./serial_code ${16} &
    wait
    shift 16
  done
}

dobysixteen $(seq 100)
```

Why not write your own load balancer?

A write-your-own attempt to balance loads could look like this.

What's wrong with this approach?

- Reinventing the wheel,
- More code to maintain/debug,
- No load balancing,
- No job control,
- No error checking,
- No fault tolerance,
- Edge cases.

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=16
#SBATCH --time=1:00:00
#SBATCH --job-name=serialx16

module load gcc/9.2.0

dobysixteen() {
  while [ -n "${16}" ]
  do
    ./serial_code ${1} &
    ./serial_code ${2} &
    ./serial_code ${3} &
    ...
    ./serial_code ${16} &
    wait
    shift 16
  done
}

dobysixteen $(seq 100)
```

GNU Parallel

GNU parallel solves the problem of managing blocks of subjobs of differing duration.

- Written in *perl*.
- But surprisingly versatile, especially for text input.
- Gets your many cases assigned to different cores and on different nodes without much hassle.
- Invoked using the “parallel” command.

- O.~Tange, “GNU Parallel - The Command-Line Power Tool’ ’ ;login: **36** (1), 42-47 (2011)

https://www.gnu.org/software/parallel/parallel_tutorial.html

GNU parallel example

Notes about our example:

- Load the gnu-parallel module within your script.
- The "-j 16" flag indicates you wish GNU parallel to run 16 subjobs at a time.
- If you can't fit 16 subjobs onto a node due to memory constraints, specify a different value for the "-j" flag.
- Put all the commands for a given subjob onto a single line.

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=16
#SBATCH --time=1:00:00
#SBATCH --job-name=gnu-parallelx16

# load modules needed...
module load gcc/9.2.0
module load gnu-parallel

# Run the code on 80 cores.
parallel -j $SLURM_TASKS_PER_NODE <<EOF
./mycode 1; echo "job 1 done"
./mycode 2; echo "job 2 done"
./mycode 3; echo "job 3 done"
...
./mycode 99; echo "job 99 done"
./mycode 100; echo "job 100 done"
EOF
```

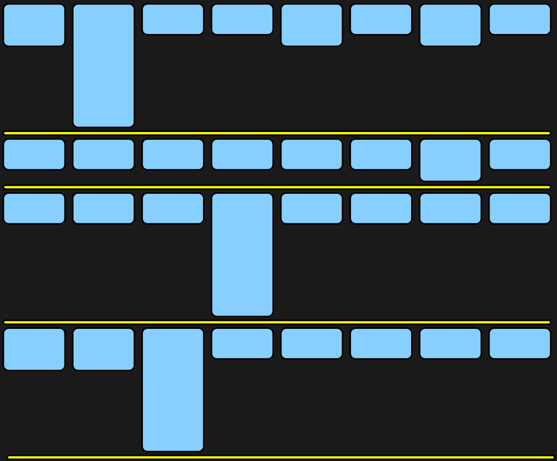
GNU Parallel, continued

What does GNU parallel do?

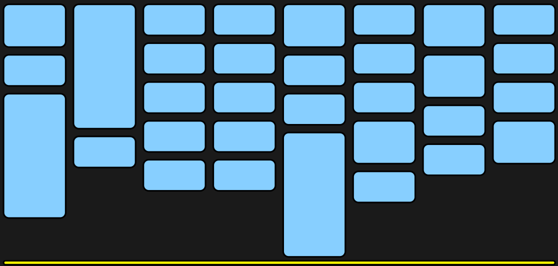
- GNU parallel assigns subjobs to the processors.
- As subjobs finish it assigns new subjobs to the free processors.
- It continues to do assign subjobs until all subjobs in the subjob list are assigned.
- Consequently there is built-in load balancing!
- You can use GNU parallel across multiple nodes as well.
- It can also log a record of each subjob, including information about subjob duration, exit status, *etc.*

If you're running blocks of serial subjobs, just use GNU parallel!

What are the gains?



17 hours
42% utilization



10 hours
72% utilization

GNU parallel example 2

Sometimes it's easiest to just create a list that holds all of the subjob commands.

```
teach01:scratch$ cat subjobs
./mycode 1; echo "job 1 done"
./mycode 2; echo "job 2 done"
./mycode 3; echo "job 3 done"
...
./mycode 99; echo "job 99 done"
./mycode 100; echo "job 100 done"
teach01:scratch$
```

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=16
#SBATCH --time=1:00:00
#SBATCH --job-name=gnu-parallelx16

# load modules needed...
module load gcc/9.2.0
module load gnu-parallel

# Run the code on 80 cores.
parallel -j $SLURM_TASKS_PER_NODE < subjobs
```

Use the `--no-run-if-empty` flag to indicate that empty lines in the subjob list file should be skipped.

GNU parallel options

Some commonly used arguments for GNU parallel:

- `--jobs NUM`, sets the number of simultaneous subjobs. By default parallel uses the number of cores on the node (16 on Teach). Same as `-j NUM`.
- `--joblog LOGFILE`, causes parallel to output a record for each completed subjob. The records contain information about subjob duration, exit status, and other goodies.
- `--resume`, when combined with `--joblog`, continues a full GNU parallel job that was killed prematurely.
- `--pipe`, splits stdin into chunks given to the stdin of each subjob.

GNU Parallel, more options

GNU parallel has a ton of optional arguments. We've barely scratched the surface.

- There are specialized ways of passing in combinations of arguments to functions.
- There are ways to modify arguments to functions on the fly.
- There are specialized ways of formatting output.
- Review the man page for parallel, or review the program's webpage, for a full list of options.

https://docs.scinet.utoronto.ca/index.php/Running_Serial_Jobs_on_Niagara

Ramdisk - Local I/O

- Can use upto 70% of RAM as local disk, (~44GB on Teach) on a regular node.
- Accessible from /dev/shm/ only on local node.
- Much faster than “spinning’ ’ disk.
- Requires you to stage your data in/out.
- Sacrifices programs RAM space.

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks=16
#SBATCH --time=1:00:00
#SBATCH --job-name=ramdisk

# load modules needed...
module load gcc/9.2.0

# copy data in
mkdir /dev/shm/$USER/workdir
cp $SLURM_SUBMIT_DIR/* /dev/shm/$USER/workdir
cd /dev/shm/$USER/workdir

# run subjobs (could use GNU Parallel as well)
for ((i=1;i<=16;i++)); do
    ./executable < $i.in > $i.out &
done
wait

$ copy data out
tar cf $SLURM_SUBMIT_DIR/out.tar *.out
```

Summary on Serial Jobs

- Be aware of the features of your code, and the details of the hardware where you will run it.
- If you need to run serial jobs on a cluster with multicore architecture, be sure to run them in batches, so as to use your nodes efficiently.
- Unless your jobs all take the same amount of time, don't try to write your own serial-job management code.
- Use GNU-Parallel to manage your serial jobs.
- More details on GNU-Parallel, can be found in this
- Ramdisk (`/dev/shm`) available to local node.

References

- https://docs.scinet.utoronto.ca/index.php/Teach
- https://docs.scinet.utoronto.ca/index.php/Niagara_Quickstart
- https://docs.scinet.utoronto.ca/index.php/Running_Serial_Jobs_on_Niagara