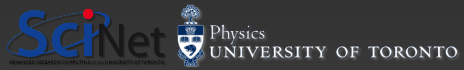
PHY1610H - Scientific Computing: Mini-intro to "SciNet": Teach/Niagara clusters Serial Jobs

Ramses van Zon and Marcelo Ponce

SciNet HPC Consortium/Physics Department
University of Toronto

March 2021



Today's class

Today we will discuss the following topics:

- 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: ie. 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: thus parallel programming.

There are several different approaches to parallel programming.

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 you just have to let your code be serial, and run the serial processes in parallel. That's the topic of today's class.

What are our assumptions?

Let us assume the following situation:

- You have a serial code.
- Your code takes a set of parameters, either from a file or (preferably) from the command line.
- The code runs in a reasonably short amount of time (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 feedback on your jobs, things like error checking, fault tolerance, *etc.*
- You want to run your code on SciNet: Niagara, Teach, ...

How do we go about performing this set of calculations efficiently?

What are SciNet's concerns?

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

- Scheduling is done by node. Each node comes with 16 cores and ∼64Gb of available memory. Use your resources efficiently.
  - Use all of the processors on the nodes you've been given continuously (load balancing), or
  - use all the memory you've been given efficiently (if 8 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 makes sense for you. Not crashing the filesystem is everyone's responsibility.

What are your options?

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

- Write a script from scratch which launches and manages the subjobs.
- If the code is written in Python, you could use ipython notebook to manage the subjobs.
- If the code is written in R, you could 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] O. Tange (2011): GNU Parallel - The Command-Line Power Tool, ;login; The USENIX Magazine, February 2011:42-47.

Niagara

- 80,960 x86-64 cores.
- 2,024 *Lenovo SD530* nodes
- Per node:
  - 40 Intel SkyLake/CascadeLake cores @2.4/2.5 GHz
  - 188 GiB RAM per node ($>$ 4 GiB per core)
  - 0 gpus, 0 harddisks
- 3.6 PFlops delivered / 6.25 PFlops theoretical. *#53 on the Jun 2018 TOP500*/currently #83
- Operating system: Linux CentOS 7.
- Burst Buffer for fast I/O

- Interconnect: InfiniBand Dragonfly+ 1:1 up to 432 nodes, 2:1 beyond that.
- Parallel share file system for home, scratch, project

SciNet

Physics
UNIVERSITY OF TORONTO

Using Niagara: Logging in

As with all SciNet and Compute Canada systems, access to Niagara is via ssh (secure shell) only.

To access SciNet systems, first open a terminal window (e.g. MobaXTerm on Windows).

Then ssh into the Niagara login nodes with your CC credentials:

```
$ ssh -Y USERNAME@niagara.scinet.utoronto.ca
```

- The Niagara login nodes are where you develop, edit, compile, prepare and submit jobs.
- These login nodes are not part of the Niagara compute cluster, but have the same architecture, operating system, and software stack.
- The optional `-Y` is needed to open windows from the Niagara command-line onto your local X server.
- To run on Niagara's compute nodes, you must submit a batch job.

Storage Systems and Locations on Niagara

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!

```
nia-login07:~$ pwd
/home/s/scinet/mponce
nia-login07:~$ cd $SCRATCH
nia-login07:mponce$ pwd
/scratch/s/scinet/mponce
```

Storage Systems and Locations on Niagara

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!

```
nia-login07:~$ pwd
/home/s/scinet/mponce
nia-login07:~$ cd $SCRATCH
nia-login07:mponce$ pwd
/scratch/s/scinet/mponce
```

Project

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

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

Storage Systems and Locations on Niagara

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!

```
nia-login07:~$ pwd
/home/s/scinet/mponce
nia-login07:~$ cd $SCRATCH
nia-login07:mponce$ pwd
/scratch/s/scinet/mponce
```

Project

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

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

Burst Buffer

Groups with heavy I/O can request access to a smaller, faster parallel file system called burst buffer.

`$BBUFFER=/bb/g/groupname/username`

Storage Limits on Niagara

location	quota	block size	expiration time	backed up	on login	on compute
$HOME	100 GB	1 MB		yes	yes	read-only
$SCRATCH	25 TB	16 MB	2 months	no	yes	yes
$PROJECT	by group allocation	16 MB		yes	yes	yes
$BBUFFER	10TB, by request	1 MB	48 hours	no	yes	yes
$ARCHIVE	by group allocation			dual-copy	no	no

- Compute nodes do not have local storage, but they have a lot of memory, which you can use as if it is local disk (`$SLURM_TMPDIR`)
- $ARCHIVE space, also called nearline storage or HPSS, is not mounted on login or compute nodes.
- Backup means a recent snapshot, not an achive of all data that ever was.

Software and Libraries

Once you are on one of the login nodes, what software is already installed?

- Other than essentials, all installed software is made available using module commands.
- These set environment variables (`PATH`, etc.)
- Allows multiple, conflicting versions of a given package to be available.
- module spider shows the available software.

Software and Libraries

Once you are on one of the login nodes, what software is already installed?

- Other than essentials, all installed software is made available using module commands.
- These set environment variables (PATH, etc.)
- Allows multiple, conflicting versions of a given package to be available.
- module spider shows the available software.

```
nia-login07:~$ module spider
----------------------------------------
The following is a list of the modules cur
----------------------------------------
  CCEnv: CCEnv
  NiaEnv: NiaEnv/2018a
  anaconda2: anaconda2/5.1.0
  anaconda3: anaconda3/5.1.0
  autotools: autotools/2017
    autoconf, automake, and libtool
  boost: boost/1.66.0
  cfitsio: cfitsio/3.430
  cmake: cmake/3.10.2 cmake/3.10.3

  ...
```

Software and Libraries, continued

- `module load <module-name>`
  use particular software
- `module purge`
  remove currently loaded modules

- `module spider` (or `module spider <module-name>`)
  list available software packages
- `module avail`
  list loadable software packages
- `module list`
  list loaded modules

Software and Libraries, continued

- `module load <module-name>`
  use particular software
- `module purge`
  remove currently loaded modules

- `module spider` (or `module spider <module-name>`)
  list available software packages
- `module avail`
  list loadable software packages
- `module list`
  list loaded modules

On Niagara, there are really two software stacks:

Software and Libraries, continued

- `module load <module-name>`
  use particular software
- `module purge`
  remove currently loaded modules

- `module spider` (or `module spider <module-name>`)
  list available software packages
- `module avail`
  list loadable software packages
- `module list`
  list loaded modules

On Niagara, there are really two software stacks:

1. A Niagara software stack tuned and compiled for this machine. This stack is available by default, but if not, can be reloaded with

```
module load NiaEnv
```

Software and Libraries, continued

- `module load <module-name>`
  use particular software
- `module purge`
  remove currently loaded modules

- `module spider` (or `module spider <module-name>`)
  list available software packages
- `module avail`
  list loadable software packages
- `module list`
  list loaded modules

On Niagara, there are really two software stacks:

1. A Niagara software stack tuned and compiled for this machine. This stack is available by default, but if not, can be reloaded with

```
module load NiaEnv
```

2. The same software stack available on Compute Canada's heterogeneous clusters Graham, Cedar and Beluga:

```
module load CCEnv arch/avx512 StdEnv
```

The `StdEnv` module loads the same default modules as available on Graham/Cedar/Beluga.

Compiling on Niagara

- Suppose you have to compile your own C, C++ or Fortran code.
- Not a problem: Niagara has GNU compilers as well as Intel compilers installed in modules.
- MPI? Not a problem either: Niagara has openmpi and intelmpi libraries as modules.
- We recommend that you use the intel compilers with openmpi libraries.
  Use the `-march=native` *or* `-xhost` *compilation flags to get the most out of Niagara's cpus. . . .*

Example

```
nia-login07:~$ module load intel/2020u4 gsl/2.5
nia-login07:~$ ls
main.c module.c
nia-login07:~$ icc -c -O3 -xHost -o main.o main.c
nia-login07:~$ icc -c -O3 -xHost -o module.o module.c
nia-login07:~$ icc  -o main module.o main.o -lgsl -mkl
nia-login07:~$ ./main
```

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, request an interactive debug job with the debugjob command

```
nia-login07:~$ debugjob N
```

where N is the number of nodes. The duration of your interactive debug session can be at most one hour, can use at most N=4 nodes, and each user can only have one such session at a time.

SciNet's Niagara/Teach-cluster Scheduler

- on a HPC cluster, we don't run programs as we will do in our own computers
- 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 indeed a quite complicated process, many variables to take into consideration
- plus the continuous dynamics of the cluster...
- even when we run 'interactively' (`debugjob, salloc ...`), we are requesting resources to the scheduler
- we refer to the 'request' (processes/programs to run) as *jobs*
- the scheduler keeps everything in shape by organizing *queues* for the jobs to run
- what we will see next is how to comunicate with the scheduler and request resources to run our programs

Submitting jobs

- Niagara uses SLURM as its job scheduler.
- You submit jobs from a login node by passing a script to the sbatch command:

```
nia-login07:~$ sbatch jobscript.sh
```

- This puts the job in the queue. It will run on the compute nodes in due course.
- Jobs will run under their group's RRG allocation, or, if the group has none, under a RAS (or "default") allocation.

Submitting jobs

- Niagara uses SLURM as its job scheduler.
- You submit jobs from a login node by passing a script to the sbatch command:

```
nia-login07:~$ sbatch jobscript.sh
```

- This puts the job in the queue. It will run on the compute nodes in due course.
- Jobs will run under their group's RRG allocation, or, if the group has none, under a RAS (or "default") allocation.

Keep in mind:

Submitting jobs

- Niagara uses SLURM as its job scheduler.
- You submit jobs from a login node by passing a script to the sbatch command:

```
nia-login07:~$ sbatch jobscript.sh
```

- This puts the job in the queue. It will run on the compute nodes in due course.
- Jobs will run under their group's RRG allocation, or, if the group has none, under a RAS (or "default") allocation.

Keep in mind:

- Scheduling is by node, so in multiples of 40-cores. *Use all cores!*
- Maximum walltime is 24 hours.
- Jobs must write to your scratch or project directory (home is read-only on compute nodes).
- Compute nodes have no internet access.
  → Download data you need beforehand on a login node.

SLURM nomenclature

SLURM has a somewhat different way of referring to things like MPI processes and threads tasks.

Term	Meaning	SLURM	Scheduler option(s)
·**job**	Scheduled piece of work for which specific resources were requested	**job**	`sbatch,salloc,debugjo`
·**node**	Basic computing component with several cores (40) that share memory	**node**	`--nodes -N`
·**MPI process**	One of a group of programs using MPI for parallel computing	**task**	`--ntasks -n` `--ntasks-per-node`
·**physical core**	A fully-functional independent execution unit	**billing**	
·**logical cpu**	An execution unit that the operating system can assign work to	**cpu**	`--ncpus-per-task`
·**thread**	One of multiple simultaneous execution paths in a program that share memory		`--ncpus-per-task` and `OMP_NUM_THREADS`
·**hyperthreading**	Hardware-assisted overloading of cores		

Hyperthreading: Logical CPUs vs. cores

- Hyperthreading, a technology that leverages more of the physical hardware by pretending there are twice as many logical cores than real once, is enabled on Niagara.
- So the OS and scheduler see 80 logical cores.
- 80 logical cores vs. 40 real cores typically gives about a 5-10% speedup (YMMV).

Hyperthreading: Logical CPUs vs. cores

- Hyperthreading, a technology that leverages more of the physical hardware by pretending there are twice as many logical cores than real once, is enabled on Niagara.
- So the OS and scheduler see 80 logical cores.
- 80 logical cores vs. 40 real cores typically gives about a 5-10% speedup (YMMV).

Because Niagara is scheduled by node, hyperthreading is actually fairly easy to use:

- Ask for a certain number of nodes N for your jobs.
- You know that you get 40xN cores, so you will use (at least) a total of 40xN MPI processes or threads.
  (mpirun, srun, and the OS will automaticallly spread these over the real cores)
- But you should also test if running 80xN MPI processes or threads gives you any speedup.
- Regardless, your usage will be counted as 40xNx(walltime in years).

Example submission script (OpenMP)

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --cpus-per-task=40
#SBATCH --time=1:00:00
#SBATCH --job-name openmp_job
#SBATCH --output=openmp_output_%j.txt
#SBATCH --mail-type=FAIL

module load intel/2018.2
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

./openmp_example
  # or 'srun ./openmp_example'
```

```
nia-login07:scratch$ sbatch openmp_job.sh
```

Example submission script (OpenMP)

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --cpus-per-task=40
#SBATCH --time=1:00:00
#SBATCH --job-name openmp_job
#SBATCH --output=openmp_output_%j.txt
#SBATCH --mail-type=FAIL

module load intel/2018.2
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

./openmp_example
  # or 'srun ./openmp_example'


nia-login07:scratch$ sbatch openmp_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 `openmp_job`).
- In this case, SLURM looks for one node with 40 cores to be run inside one task, for 1 hour.
- Submit from /scratch, as /home is read-only.
- Once it founds a node,the script is run:
  - Loads modules;
  - Sets an environment variable;
  - Runs the `openmp_example` application.

Example submission script (MPI)

```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --ntasks=80
#SBATCH --time=1:00:00
#SBATCH --job-name mpi_job
#SBATCH --output=mpi_output_%j.txt
#SBATCH --mail-type=FAIL

module load intel/2018.2
module load openmpi/3.1.0

mpirun ./mpi_example
' # or  srun ./mpi_example'
```

```
nia-login07:scratch$ sbatch mpi_job.sh
```

Example submission script (MPI)

```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --ntasks=80
#SBATCH --time=1:00:00
#SBATCH --job-name mpi_job
#SBATCH --output=mpi_output_%j.txt
#SBATCH --mail-type=FAIL

module load intel/2018.2
module load openmpi/3.1.0

mpirun ./mpi_example
' # or  srun ./mpi_example'
```

```
nia-login07:scratch$ sbatch mpi_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 `mpi_job`)
- In this case, SLURM looks for 2 nodes with 40 cores on which to run 80 tasks, for 1 hour.
- Submit from /scratch, so output can be written.
- Once it found nodes, the script is run:
  - Loads modules;
  - Runs the `mpi_example` application.

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 -u $USER` 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 -i 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.

If you're not sure . . .

We've gone over this quickly, quite intentionally.

- There are lots of details that we've skipped over.
- If you haven't taken the "Intro to SciNet" class, we recommend you do.
- If you've never run on SciNet, read the SciNet User Tutorial and the GPC quickstart guide.
- Almost certainly your question is answered on the docs.scinet.utoronto.ca.
- If all else fails, email us (support@scinet.utoronto.ca).

`https://docs.scinet.utoronto.ca`

Now back to serial

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? Not if all the subjobs take the same amount of time.

But if your subjobs take variable amounts of time this approach isn't going to work.

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=40
#SBATCH --time=1:00:00
#SBATCH --job-name=serialx8
# load modules needed...
module load intel/2020u4 gsl/2.5

cd $SLURM_SUBMIT_DIR
# Run the code on 8 cores.
(cd jobdir1; ../mycode) &
(cd jobdir2; ../mycode) &
(cd jobdir3; ../mycode) &
.  .  .
(cd jobdir38; ../mycode) &
(cd jobdir39; ../mycode) &
(cd jobdir40; ../mycode) &
# Tell the script to wait, or all
# the subjobs get killed immediately.
wait
```

Why not roll your own?

Suppose I can only fit 4 subjobs simultaneously on a node. What's wrong with this approach?

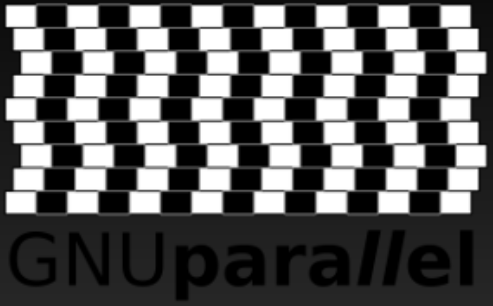
- Reinventing the wheel,
- More code to maintain/debug,
- No load balancing,
- No job control,
- No error checking,
- No fault tolerance,
- No multi-node jobs.

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --time=5:00:00
#SBATCH --job-name=badserialx4
# load modules needed...
module load intel/2020u4 gsl/2.5

cd $SLURM_SUBMIT_DIR
function dobyfour() {
    while [ -n "$1" ]
    do
        ./variable_time_code $1 &
        ./variable_time_code $2 &
        ./variable_time_code $3 &
        ./variable_time_code $4 &
        wait
        shift 4
    done
}
dobyfour $(seq 100)
```

GNU Parallel

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



- Basically a perl script.
- 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.

1. O. Tange, "GNU Parallel - The Command-Line Power Tool" ;login: **36** (1), 42-47 (2011)
2. `http://www.gnu.org/software/parallel/parallel_tutorial.html`

SciNet Physics UNIVERSITY OF TORONTO

GNU parallel example

Notes about our example:

- Load the gnu-parallel module within your script.
- The "-j 8" flag indicates you wish GNU parallel to run 8 subjobs at a time.
- If you can't fit 8 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=80
#SBATCH --time=1:00:00
#SBATCH --job-name=gnu-parallelx80

cd $SLURM_SUBMIT_DIR
# load modules needed...
module load intel/2020u4 gsl/2.5
module load gnu-parallel

# Run the code on 80 cores.
parallel -j $SLURM_TASKS_PER_NODE <<EOF
cd jobdir1; ../mycode; echo "job 1 done"
cd jobdir2; ../mycode; echo "job 2 done"
cd jobdir3; ../mycode; echo "job 3 done"
.
.
.
cd jobdir199; ../mycode; echo "job 199 done"
cd jobdir200; ../mycode; echo "job 200 done"
EOF
```

GNU Parallel, continued

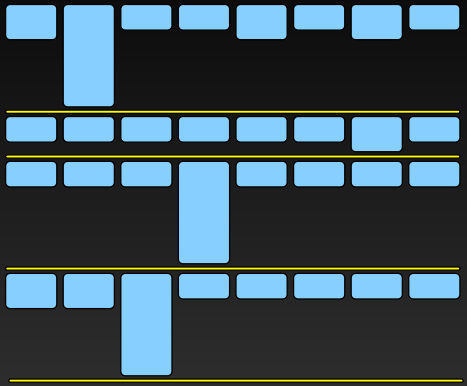
What does GNU parallel do?

- 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 more than 8 subjobs per node (hyperthreading).
- 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!

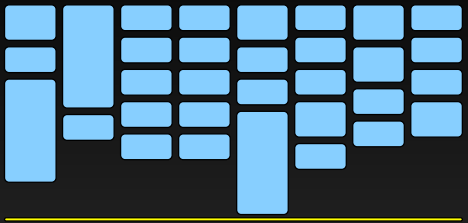
Backfilling

Without GNU parallel:



17 hours
42% utilization

With GNU parallel:



10 hours
72% utilization

GNU parallel example 2

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

```
user@gpc-f103n084-ib0:~>
user@gpc-f103n084-ib0:~> cat subjobs
cd jobdir1; ../mycode; echo "job 1 done"
cd jobdir2; ../mycode; echo "job 2 done"
cd jobdir3; ../mycode; echo "job 3 done"
⋮
cd jobdir198; ../mycode; echo "job 198 done"
cd jobdir199; ../mycode; echo "job 199 done"
cd jobdir200; ../mycode; echo "job 200 done"
user@gpc-f103n084-ib0:~>
```

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=80
#SBATCH --time=1:00:00
#SBATCH --job-name=gnu-parallelx80

# load modules needed...
module load intel/2020u4 gsl/2.5
module load gnu-parallel

cd $SLURM_SUBMIT_DIR
# Run the code on 80 cores.
parallel -j $SLURM_TASKS_PER_NODE \
    --no-run-if-empty < subjobs
```

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

GNU parallel syntax

Some commonly used arguments for GNU parallel:

- `--jobs NUM`, sets the number of simultaneous subjobs. By default parallel uses the number of virtual cores (32/80 on Tech/Niagara nodes). Same as `-j N`.
- `--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 multi-node example

Notes about using multiple nodes:

- By default parallel only knows about the head node. Tell it about the other nodes using `--sshloginfile $PBS_NODEFILE`.
- The non-head nodes also don't know where they should be working, thus `--workdir $PWD`.
- This setup will run 40 jobs simultaneously, and only makes sense if, say, several hundred jobs are in your subjobs file.
- Remember that the fewer nodes you request, the more likely your job is to run.

```
#!/bin/bash
# SLURM submission script for multiple serial jobs
      on multiple Niagara nodes
#
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=40
#SBATCH --time=12:00:00
#SBATCH --job-name gnuparallel-multinode-example

# DIRECTORY TO RUN - $SLURM_SUBMIT_DIR is the
     directory from which the job was submitted
cd $SLURM_SUBMIT_DIR

# Turn off implicit threading in Python, R
export OMP_NUM_THREADS=1

module load gnu-parallel/20180322

HOSTS=$(scontrol show hostnames $SLURM_NODELIST |
     tr '\n' ,)

parallel --env OMP_NUM_THREADS,PATH,
     LD_LIBRARY_PATH --joblog slurm-$SLURM_JOBID.
     log -j $SLURM_NTASKS_PER_NODE -S $HOSTS --wd
     $PWD "cd␣serialjobdir{}␣&&␣./doserialjob{}"
     ::: {001..800}
```

GNU Parallel, more options

GNU parallel has a tonne 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 --cpus-per-task=40
#SBATCH --time=1:00:00
#SBATCH --job-name=ramdisk

# load modules needed...
module load intel/2020u4 gsl/2.5

mkdir /dev/shm/$USER/workdir
cp $SLURM_SUBMIT_DIR/* /dev/shm/$USER/workdir
cd /dev/shm/$USER/workdir
for ((i=1;i<=40;i++)); do
    ./executable < $i.in > $i.out &
done
wait
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 tech talk.
- 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`