High Throughput Computing a.k.a. Farming

Ramses van Zon

PHY1610 Winter 2026

High Throughput Computing

Sometimes your code is serial, i.e., only runs on a single processor.

There are several reasons why we might not push the code further:

- 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.

This is sometimes called Serial Farming.

A typical computational workflow

- 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: Trillium, Teach, ...

How do we setup a set of simultaneous calculations with high throughput efficiency?

We are concerned...

What are your HPC centre's concerns? Well, at SciNet:

- On our production clusters, scheduling is done by node.
  Trillium's nodes have 192 cores and with 768GB of RAM.
  - Use all of the processors on the nodes you've been given
    or
  - Use all the memory you've been given efficiently

  This almost certainly means having multiple subjobs running simultaneously.
- 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?

1. Use the scheduler's capabilities for serial farming ("job arrays").
2. Write a script from scratch which launches and manages the subjobs.
3. For code written in Python, use multiprocessing to manage the subjobs.
4. For code written in R, use the parallel R utilities to manage the subjobs.
5. Use an existing script, such as GNU Parallel[1] to manage the subjobs.

We'll discuss options 1, 2, and 5 in this class.

[1] Tange, O. (2021, March 22). GNU Parallel 20210322 ('2002-01-06'). Zenodo https://doi.org/10.5281/zenodo.4628277

Option 1: Job arrays

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=1-1000

module load gcc/14

./serial_code $SLURM_ARRAY_TASK_ID
```

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

- One jobscript for many jobs.
- 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` 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 Trillium! Each serial job always gets a whole node!

Warning 2

Not suitable for large number of short (<10min) serial jobs. Scheduling overhead would dominate.

Option 2: SIY (Script It Yourself)

Start background subjobs within the job

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=40
#SBATCH --time=1:00:00
#SBATCH --job-name=serialx40

module load gcc/14

#Run the code on 40 cores
./serial_code 1 &
./serial_code 2 &
./serial_code 3 &
...
./serial_code 39 &
./serial_code 40 &

#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 1000 cases?

Why not write your own load balancer?

A SIY attempt to balance loads could look like this.

What's wrong with this approach?

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

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=40
#SBATCH --time=1:00:00
#SBATCH --job-name=serialx40

module load gcc/14

dobysixteen() {
  while [ -n "${40}" ]
  do
    ./serial_code ${1} &
    ./serial_code ${2} &
    ./serial_code ${3} &
    ...
    ./serial_code ${40} &
    wait
    shift 40
  done
}

dobysixteen $(seq 1000)
```

Option 5: Use Existing Tools

GNU Parallel

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

- Surprisingly versatile, especially for parameters given as text.
- Gets your many cases assigned to different cores and on different nodes without much hassle.
- Invoked using the “parallel” command.

- Tange, O. (2021, March 22). GNU Parallel 20210322 (‘2002-01-06’). Zenodo https://doi.org/10.5281/zenodo.4628277

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

GNU parallel example

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=40
#SBATCH --time=1:00:00
#SBATCH --job-name=gnu-parallelx40

# load modules needed...
module load gcc/14

# Run the code on 40 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 999; echo "job 999 done"
./mycode 1000; echo "job 1000 done"
EOF
```

- The "`-j 40`" flag indicates you wish GNU parallel to run 40 subjobs at a time.
- If you can't fit 40 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.
- Each line becomes a **sub-job**

GNU parallel does not call them subjobs, just jobs, but we're already in a SLURM job.

GNU Parallel, continued

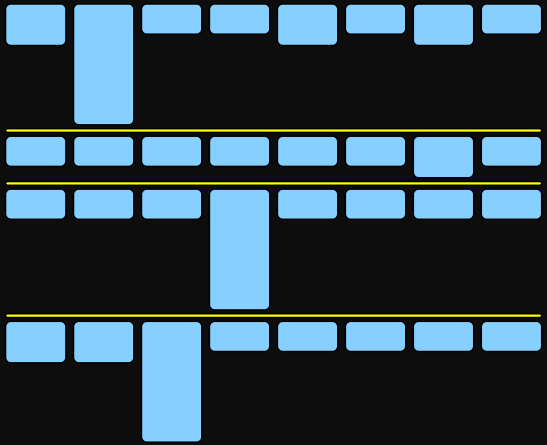
What does GNU parallel do?

- 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.*

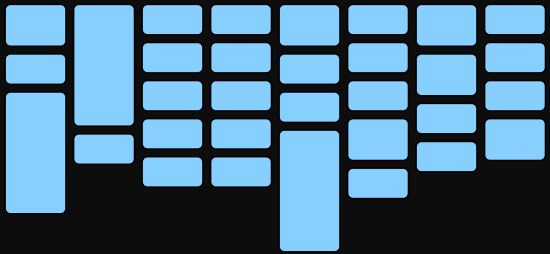
If you're running blocks of serial subjobs, just use GNU parallel!

What are the gains?

Imagine you're running 32 jobs of varying duration on 8 cores.



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 in a file.

```
teach01:scratch$ cat subjobs.in
./mycode 1; echo "job 1 done"
./mycode 2; echo "job 2 done"
./mycode 3; echo "job 3 done"
...
./mycode 999; echo "job 999 done"
./mycode 1000; echo "job 1000 done"
teach01:scratch$
```

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=40
#SBATCH --time=1:00:00
#SBATCH --job-name=gnu-parallelx40

# load modules needed...
module load gcc/14

# Run the code on 40 cores.
parallel -j $SLURM_TASKS_PER_NODE < subjobs.in
```

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

GNU Parallel with a list of files

Example:

```
$ find . -name '*.csv' | parallel -j 40 grep -l JFK
```

This accomplishes the following:

- The linux command `find` lists files with the extension `csv` in the current directory and its subdirectory.
- parallel, which is the GNU parallel command, divides up the list of filenames up.
- For each filename, it executes the linux `grep` command for each filename to find files containing `JFK`.
- It runs 4 subjobs at the same time, because of the `-j4` parameter. Without this parameter, GNU parallel uses as many cores as there are.

GNU Parallel with parameter lists

GNU parallel can take parameters on the command line too:

For instance:

```
$ parallel -j 2 echo {} ::: 1 2 3 4
```

might yield:

```
1
2
4
3
```

(with 2 cases done in parallel)

GNU parallel can take several sets of parameters, and run them in every combination.

For instance:

```
$ parallel -j 2 echo {1} {2} {3} ::: 1 2 ::: A B ::: x y
```

yields:

```
1 A x
1 A y
1 B x
1 B y
2 A x
2 A y
2 B x
2 B y
```

GNU Parallel using a database

- GNU parallel can create entries for each subjob in a database.
- Then it can run those, filling in the job particularities.
- Using e.g. sqlite, this looks as follows:

```
$ parallel --sqlmaster sqlite3:///db.sq/tbl  echo ::: 1 2 ::: A B ::: x y
```

This stores the jobs in the file `db.sq`, in table `tbl`.

```
$ parallel --sqlworker sqlite3:///db.sq/tbl
```

This executes the jobs in the database.

- Keeps track of runtime, completion, parameters.
- Can restart where it left off.

GNU Parallel's restart capability

It can do restarts without a proper database too, using a "log".

- `--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.

  For this to work the original GNU parallel job must have had a `--joblog` option.
- `--pipe`, splits stdin into chunks given to the stdin of each subjob.

Review the man page for parallel, or review the program's webpage, for a full list of options.

Ramdisk

Ramdisk - Local I/O

Running 40 (or 192) programs in the same directory can overload the shared file system. Instead:

- You can use upto 70% of the node's RAM as local disk.
- Accessible from `/dev/shm/` only on local node.
- On Teach and Trillium, jobs get a ramdisk folder through the environment variable `$SLURM_TMPDIR` (on other clusters this may be local disk)
- Much faster than real disk.
- Ramdisk requires you to stage your data in/out.
- Sacrifices programs RAM space.

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks=40
#SBATCH --time=1:00:00
#SBATCH --job-name=ramdisk
#SBATCH --output=ramdisk_%j

# load required modules
module load gcc/14

# copy application, and data from the data/*.in
WORK_DIR=$SLURM_TMPDIR
cp $SLURM_SUBMIT_DIR/mycode $WORK_DIR
cp $SLURM_SUBMIT_DIR/data/*.in $WORK_DIR
cd $WORK_DIR

# run subjobs
parallel -j 40 ./mycode \< {}.in \> {}.out ::: {1..1000}

$ 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.
- Ramdisk (`/dev/shm` or `$SLURM_TMPDIR`) available as fast local node storage.

References

- https://docs.scinet.utoronto.ca/index.php/Teach
- https://docs.alliancecan.ca/wiki/Trillium_Quickstart
- https://docs.scinet.utoronto.ca/index.php/Running_Serial_Jobs_on_Niagara