



GNU Parallel Exercises

Note: when we show commands, we put them in back ticks, `command`. Don't type the ticks!

Note: the GNU parallel examples assume we are using the bash shell. If you are using another shell you may need to check that the shell commands still work

- 01 First examples of GNU Parallel

- To get started, lets run some easy examples
- First, try just the command 'parallel' - if the command is not found, you can ask us for help installing it. On the SCRTP task farm or cluster you'll need to load the module
- https://blogs.warwick.ac.uk/researchsoftware/entry/gnu_parallel_without/ gives instructions for a manual install but this is *not usually needed*

Info:

We use the 'echo' command, which is a program which repeats whatever text we feed to it, to stand in for some actual program we'd be trying to run. Later we'll show a simple "real" program for you to build on

Try:

Based on the examples from the slides, can you produce a command to print (echo) the numbers from 1 to 5?

Or to print every combination of 3 digits 0 and 1 (i.e. 000, 001, 010 etc)

In the bash shell, we can produce a range from x to y using `{x..y}` (note the two dots. Try ``echo {1..10}`` or ``echo {a..z}`` and check this works. Can you combine this with parallel to get the numbers 1 to 100 as arguments?

- 02 Proving things run in parallel

- Lets check that we really are running in parallel and see how to limit the number of jobs
- Parallel doesn't start jobs literally at once, there is a tiny delay, so for very short jobs they may not actually run simultaneously. We'll use the `sleep` command to mimic a longer running job

Info:

The `time` command can be used before any other command to show how long it took to run

Try:

Try the command `parallel sleep ::: 1 2 3`` How long do you expect it to take to run? How long does it take?

The '-j' flag to parallel limits how many jobs can run at once. How long will this take? `parallel -j 2 sleep ::: 3 3 3``

Parallel generally starts jobs in order. Why will `parallel -j 2 sleep ::: 2 4 6`` probably take longer than `parallel -j 2 sleep ::: 6 4 2``

Aside - the previous example shows one of the easiest tricks in parallel job scheduling - start the long things first! If it's not obvious why, try drawing a few scheduling pictures like we showed on the slides

- 03 A simple real program

- If you know how to write a program which takes command line arguments, or can write one using the information in the appendix, write something simple now that takes a number and does something depending on it
- If not you can use the supplied script, `sleep_script.sh` (bash assumed), also printed at the end of this document. This takes 0 or 1 numbers as arguments and sleeps for that number of seconds

Info:

You may have to set the “execute” permission on the script using `chmod u+x sleep_script.sh` to allow it to run, and you’ll need to use `./sleep_script.sh` with the leading `./` when you run it

Try:

Replace the echo and sleep command we used above with your program and try a few parallel commands

Try running this `parallel -j 10 ./sleep_script.sh ::: 9 8 3 6 1 2 10 4 7 5` Assuming you have 10 or more cores available, what common task is this, in a very silly fashion, performing on its arguments?

- 04 Generating Arguments

- So far our argument lists have been short and we just type them. There are two main ways to get more sophisticated - generate the argument lists programmatically, or read them from a file

Info:

You can “pipe” or redirect output from a shell command to a file using `>` such as `echo Hello World > tmp.txt` This can be a handy way to generate argument files quickly

Try:

In the bash shell, we can produce a range from x to y as part of a command using `{x..y}` (note the two dots. Try `echo {1..10}` or `echo {a..z}` and check this works. Can you combine this with parallel to get the numbers 1 to 100 as arguments?

There is also a command to generate a sequence of numbers, for example the even numbers from 2 to 20 like `seq 2 2 20`. We can “pipe” the output of a command like this in parallel like this: `seq 2 2 20 | parallel echo Number {}` Try some other commands on the left - as long as they produce a list of items parallel should understand

Create an input file for parallel using a command such as seq. If the file is named "in.txt" then you can feed it to parallel using `parallel -a in.txt echo`` for example. Try this out.

- 05 Parallel Power

- We've shown the basic features of parallel, but there it can do more sophisticated things with arguments, and has lots of options to control job starting, stopping, logging, and repeating failed jobs
- Check the docs - they are very good for a change https://www.gnu.org/software/parallel/parallel_tutorial.html#input-sources

Try:

The progress flag can be very handy, and is worth seeing. Try this `parallel --progress -j 16 sleep ::: 9 8 3 6 1 2 10 4 7 5``

Note that if your jobs produce output, you'll get progress mixed in with it

You can also specify a delay between starting jobs with the delay flag in seconds. This is useful if your jobs do a lot of IO when they start, or access some shared resource, and starting a lot at once would be troublesome

Lastly for now, you can specify that a job taking longer than x seconds should be stopped with the timeout flag

Appendix - command line arguments by language

Command line arguments usually get split up on whitespace. For complicated programs, you will want to name your arguments, and use something like "myprogram a=1 b=2" and your program would need to divide up the name from the '=' sign and the value. For now, lets just show how to pass simple values and assume their order tells us everything we need to know

- **Bash or shell** - use the variables ``$n`` in your script where n goes from 1 to the number of arguments you supply.
- `$0` is the name of the script that has been run

- Strictly you should check how many args were given using ``$#`` but for simple examples we just make sure to supply the right amount
- **Python**
 - Import ``argv`` from ``sys`` (e.g. ``from sys import argv``) (see C below for the reason for the name)
 - `argv` is then a list containing command line arguments - the name of the running script as the 0th element, just like in bash
 - Check the number of arguments with ``len(argv)``
 - Note that the arguments are all strings, no matter what values you pass to them, so you will need to turn them into numbers if you want to (e.g. ``int(argv[1])``)
- **Matlab** (note that starting Matlab is not quick, so running very short examples wont be very fun)
 - For a flat script file, here called "test.m", you can run in hands-off batch mode using ``matlab -nodisplay -batch "test"``
 - The easiest way to pass arguments to this is to define them as part of the command like ``matlab -nodisplay -batch "x=2;test"`` and the use `x` in your script as though it was defined within it.
 - As part of a parallel command it looks like ``parallel matlab -nodisplay -batch "x={};test" :: 1 2``
- **C and C++**
 - Your main program should be either ``int main()`` or ``int main(int argc, char* argv[])``
 - The latter has two arguments, `argc` is the count of the number of arguments, `argv` is an array of strings
 - You can use functions like `atoi` and `atof` to get integer or float values from strings
 - In C++ there are fancier ways, and the best option depends on how you are going to parse the values. C++17 adds a "from_chars" function which has sophisticated parsing ability and is worth checking out if you want more than simple ints
- **Fortran**

- Since F2003 (which we very strongly recommend you use!) you can handle command line arguments very easily
- `command_argument_count()` return the number of arguments
- You can get each one as a string like `CALL get_command_argument(i, arg)` where `i` is the number and `arg` is a string to fill with the argument
- Converting from string to int or float is done using `read`, like ``read(arg, *, IOSTAT=ierr) val`` where `val` is a value of the type you want, `arg` is the argument, the `'*'` is a format, here default, and `ierr` is an integer to be filled with an error status

Appendix - super simple sleepy script

```
#!/bin/bash
if [ $# -ne 0 ]; then
    sleep $1
    echo Slept for $1
fi
```