Parallelisation via srun and GNU parallel

bwHPC support team Hohenheim
kim-bw-projekt@uni-hohenheim.de

GNU parallel: All-purpose parallelisation

GNU parallel allows to

- run any (executable) program(s) in parallel with no communication between program instances
- Allows to run many parallelisations on a fixed number of nodes: one finishes - the next starts
- Does not have to be loaded via **module load**

Why GNU parallel?

- Not much overhead for learning standard usage (+ a bit more)
- Nice features, e.g. rerun only failed jobs

What does GNU parallel not do (well)?

- Parallelisation with (meaningful) communication between instances
- Efficient multi-node use
- Collection of output

O. Tange (2018): GNU Parallel 2018, March 2018, https://doi.org/10.5281/zenodo.1146014.
https://www.gnu.org/software/parallel/

Running in parallel with **srun** - 1st example

- Our focus is on working on the bwUniCluster – which e.g. uses SLURM as a workload manager and has GNU parallel installed. All examples are run on bwUniCluster's command line. If run elsewhere, you need to adjust some commands.

- Let's start an interactive session for 30 min w. 2 cores
  $ salloc -t 30 -p dev_single -n 2

- We will often run the executable bash script scr1 with 2 arguments

```
  GNU nano 2.9.8                    scr1

#!/bin/sh
date '+%X' #Gives the system time
sleep $((3*$1)) #Sleeps for 3x arg #1 seconds
echo "this is job $2" #Prints the text string w. arg #2 inserted
echo "I slept 3 x $1 seconds" #different text, different insert
```

Generate the executable scr1

1. Copy & paste into a (text) file scr1

```
#!/bin/sh
date '+%X'
sleep $((3*$1))
echo "this is job $2"
echo "I slept 3 x $1 seconds"
```

2. Make it executable via **$ chmod u+x scr1**

GNU parallel - main syntax

$ **parallel** [options] *executable* **:::** *arg1 arg2 arg3 ...*

Runs the executable in parallel: 1x with arg1, 1x with arg2, ...
(options are ... optional and are set via -optionname)

We start an interactive session and try this out

```
$ date '+%X'  && parallel echo ::: 10 10 &&  date '+%X'
$ date '+%X'  && parallel sleep ::: 10 10 &&  date '+%X'
```

Commands after && get executed directly after the one(s) in front - given everything in front ran successful)

GNU parallel - two options

- Instead of manually providing arguments, you can use a file (one line=one argument for parallel)

$ **parallel** -a *argfile executable* #different order than on last slide, since #provided via option -a

- For many runs in parallel (# runs > 2*ntasks):

  $ parallel -j *how_many_in_parallel* -a *argfile executable*

  - -j specifies how many jobs (at most) are run in parallel at a given time
  - if j is not specified, it runs as many jobs at the same time as there are cores

GNU parallel - multiple arguments

Combining multiple arguments with

$ parallel *executable* {1} {2} ::: *arguments1* ::: *arguments2*

runs the executable with **all combinations** of *arguments1* and *arguments2*

$ parallel --link *executable* {1} {2} ::: *args1* ::: *args2*

runs the executable with combining arg #1 from *args1* with arg #1 from *args2*, arg #2 with arg #2, ... If arguments differ in length, shorter one gets recycled

Examples:

$ parallel --link ./scr1 {1} {2} ::: 5 2 5 2 ::: 1 2 3 4
$ parallel -j 2 --link ./scr1 {1} {2} ::: 5 2 ::: 1 2 3 4
What do the commands do?

GNU parallel - more options w. bash

Helpful bash commands to use w. GNU parallel:

Let x,y,z be numbers

- seq x y z: series from x to z in steps of y
- Can be used as input arguments for GNU parallel by $(seq x y z)
- pipe operator | allows to generate arguments from other commands
- Brace expansion to mix words with numbers (see example below)
- Many more

Examples:

```
$ parallel --link ./scr1 ::: 1 2 ::: $(seq 1 10 1)
$ parallel --link ./scr1 ::: 12 ::: job{1..10}
$ ls | parallel echo "FILE IS " {}          #works also without {}
```

What do these commands do?

Using pipes etc. may lead to some trial&error (syntax needs to be met)...

GNU parallel
- some more job control options

Option (short)	Meaning
-a f1 -a f2	Run with all combinations from two input files f1 and f2 Add --link to run 1st argument w. 1st argument,...
--joblog f1 --resume-failed (--resume)	1st run: run w. --joblog to add journal log f1 (filename) 2nd run: run w. both options to rerun only the failed jobs from f1 (Alternatively for 2nd run: rerun jobs not run yet)
{#}	Instead of an argument provided after ::: or via -a, inputs the job number among all parallelised instances
--citation	Citation info, please cite it if you use it for publications

Many more: Either **$ man parallel** or
https://www.gnu.org/software/parallel/parallel.html#options

Nice tutorial: https://www.gnu.org/software/parallel/parallel_tutorial.html
(you don't need remote running via -S or -ssh, since we run via SBATCH-scripts directly on the cluster)

SLURM's built-in parallelisation: **srun**

srun allows to

- run any (executable) program(s) in parallel with no communication between program instances
- Allows to run MPI-using programs that allow communication between program instances
- Does not have to be loaded via **module load**

Why **srun**?

- You can control how the program instances are distributed across cores/nodes
- Allows mpi use, i.e. It is the built-in SLURM machinery to run mpi programs
- SLURM allows a lot of reporting

Where does **srun** need help?

- It starts all jobs in parallel - workarounds needed if you want to run more than the requested number of tasks
- Collection of output (when running independent instances)

SLURM: https://slurm.schedmd.com/overview.html
srun man page: https://slurm.schedmd.com/srun.html

srun basics - run one program in parallel

- While **srun** can also request resources alongside the program call, we use it here only **within a SBATCH** script (or analogously within resources allocated via salloc)

- General syntax: $ **srun** *srun-options executable executable-args/opts*

- Simplest way to run it:
  $ srun -n *no_tasks executable [args_ex1]*

  Runs an executable file (via ***./executable_1,*** via program path) n times in parallel (optionally with arguments args_ex1)
  - *Use this e.g. when collecting all output from simulations in the same file in parallel*
  - *Or for a multi-threaded program that runs on multiple cores*

Example: **$ srun -n** *2 ./scr1 5 1*

srun - multiple executables and/or multiple arguments

- Run multiple instances of srun with one command, separated by colons :

  $ srun *srun-opt1 exec1 exec1-args* **:** *srun-opt2 exec2 exec2-args* **: ...**

  e.g.

  $ srun -n 4 exec1 : -n 3 exec2 : -n 2 exec 3

  runs exec1 4 times, exec2 3 times and exec3 2 times (all in parallel). However, each instance runs on a separate node (produces an error if not enough nodes allocated).

- Better way to run it, works on a single node:

  $ srun -n *no_jobs* --multiprog *input_file*

The input file is a text file, where each line shows, separated by a blank space,

- Which task to run with these arguments (indicated by number 0,1,...,n-1)
- Program to be run
- Program parameters

srun - the next example

Run scr1 w. two different sets of arguments

```
$ srun -n 1 ./scr1 5 1 : -n 1 ./scr1 2 2
```

What does the command do? Does it run on our resources? Why not?

- Exercise: Do the exact same via --multiprog.

See the next slide for an example file for --multiprog

An example of a --multiprog input file

conf_multiprog.txt

```
0 ./scr1 5 1
1 ./scr1 5 2
2 ./scr1 5 3
3 ./scr1 2 4
4 ./scr1 3 5
5 ./scr1 5 6
6 ./scr1 1 7
7 ./scr1 5 8
8 ./scr1 3 9
9 ./scr1 2 10
```

- If multiple tasks have the same arguments, you can separate them via comma an/or specify a range *from-to*
  Example: **0,3-5 ./fun1 7**
  runs ./fun1 w. argument 7 as tasks 0,3,4,5
- Setting * as task number runs all tasks not specified in lines above it with the arguments in that line

srun - some more job control options

Option (short)	Meaning
--exclusive	Only allocate memory matching to the number of cores used for srun
-N *min-max*	Spread job over at least min nodes, at most max nodes
--spread-job	Spreads jobs across nodes in a balanced manner
-r *offset*	Start distributing process instances on node *offset* of the current allocation (default value 0)
-c *tasks-per*	Each invoked process instance gets *tasks-per* cores allocated
-v [-v ...]	shows details of how srun distributes the jobs. Adding it multiple times increases the show information.
--gpus-per-task=*number*	Take *number* of GPUs per invoked process

All entries in italics are integer-valued

Full (long) list: https://slurm.schedmd.com/srun.html *worth looking at, e.g --distribution to have precise control over how tasks are distributed*

A word on output files from running many tasks via GNU parallel

- GNU parallel allows to run many tasks, which possibly all write an output into a file

- Writing many files (e.g. tens of thousands, millions) within a workspace is not ideal for the memory system underlying the workspaces (and really bad on $HOME)

- Solution (within SBATCH or salloc): Write the ouput files to the temporary directory
  - Each salloc or sbatch job has its own temporal directory while it exists
  - Its path is stored in the variable **$TMP** (within the job allocation)
  - Write output there, and then move it back at the end of the SBATCH script/interactive session

```
$ echo $TMP
$ cd $TMP
$ echo "THIS IS THE TMP: " $TMP > temp1.txt
$ cp temp1.txt $HOME/temp1.txt
```

Combining srun and GNU parallel to (comfortably) run on multiple nodes

- Run many jobs in parallel by GNU parallel, but wrap the executable into a srun to use srun's job control

Example (needs two cores to run, so you'd need a new resource call):

```
$ parallel -j 8 --link srun --exclusive -N 1  -n 1 -r {1} ./scr1 {2} {#} ::: 0 1 ::: 6 1
1 1 1 1 1 6 7 1 2 1 2 5 6 7 8 1 2 3 1 2 4 7
```

Runs 24 instances of exec1 with different parameters, at most 8 at one time, each one on one core/one node. Each second instance runs on node 2, all others on node 1. All instances get only default memory per node allocated

IMPORTANT!! You need to set --exclusive, otherwise the first instance gets all resources allocated