# Job Arrays & rslurm for Parallelism

Steven Finch, sfinch@mit.edu

Research Computing Specialist, Sloan Technology Services

Two methods to submit parallel R batch jobs to the Engaging HPC cluster include

- using a SLURM job array
- using a package from within R (even interactively) called rslurm.

We will illustrate the former via a small example (5 parallel jobs) and the latter via a large example (more than 60 parallel jobs). Both examples involve Monte Carlo simulation of a simple asymmetric random walk on the one-dimensional integer lattice with reflection at the origin. Thus the walk never strays into the negative integers. Let the number of time steps be $N=10^6$. Assuming that the upward probability $p$ is less than the downward probability $q$, what can be said about the expected value of the walk maximum? Likewise, what can be said about the associated uncertainty? For the sake of concreteness, assign $p=1/3$ and $q=2/3$. The average maximum is very small relative to $N$; in fact, it is only on the order of $\log(N)$. Our aim is to numerically estimate the average maximum as precisely as possible, necessitating a sizeable sample of synthetic realizations. Theory fails here; experimentation is the only available approach.

Here are the files needed for the first example:

Walk_initial.sh, Walk_initial.R, Walk_final.sh, Walk_final.R

In the initial shell script, the line

```
#SBATCH --array=1-5
```

directs SLURM to create an array of the first five positive integers (which must match an array used in the final R program). One of SLURM's built-in variables

```
${SLURM_ARRAY_TASK_ID}
```

appears in the srun line of the initial shell script; this accesses the specific task ID of the current task in the job array (1 for the first task, 2 for the second, etc.) The lines

```
#SBATCH --output=Walk_initial_%a_out.txt
#SBATCH --error=Walk_initial_%a_err.txt
```

utilize a shorthand representation "%a" for accessing SLURM_ARRAY_TASK_ID (1, 2, etc.) in labeling output/error files appropriately. Note also the use of "Rscript" rather than simply "R" in the srun command. The initial R program begins with important lines

```
args <- commandArgs(TRUE)
job.id <- strtoi(args[1], base=10L)
```

that acquire the task ID from SLURM (for the sake of the TXT output) and ends with another important line

```
saveRDS(c(mu1, mu2), file = paste("Walk",job.id,".rds",sep=''))
```

that constructs the RDS file containing sample mean/mean-square for the current task. Type the command

sbatch Walk_initial.sh

at the UNIX prompt to commence the five tasks. When finished, you should possess five error files (all empty), five output files (each containing 40 datapoints, as well as moment estimates) and five RDS files (containing only moment estimates). Observe that our R function rnd.Walk() actually provides the data, which contrasts with the R function rnd_W() to be defined via the rslurm-based method shortly.

Do not apply the final shell script until all fifteen files are in place:

sbatch Walk_final.sh

The final shell script is straightforward (containing no job arrays); the final R program begins with important lines

```
f <- function(job.id) readRDS(paste("Walk",job.id,".rds",sep=''))

vec <- t(sapply(1:5, f))
```

that assimilate each of the five RDS files and write the sample moments to a 5×2 matrix. After averaging, the results should be roughly 17.6 for the mean and 1.8 for the standard deviation. In words, a reflected random walk (with $p$=1/3 and $q$=2/3) starting at 0 typically does not pass through 19, even after $10^6$ time steps. Your estimates might be poor because the sample size was only 200. Increasing drastically the numbers 5 (task count) and 40 (data count) will help improve quality, but we believe that rslurm offers more convenient mechanisms for doing this.

Here are the files needed for the second example:

W_initial.sh, W_initial.R, W_final.sh, W_final.R

Neither shell script requires special comment. The initial R program, however, contains critical lines

```
pars <- data.frame(k = 1:500)

sjob <- slurm_apply(rnd_W, pars, jobname = "test_W", nodes = 250,
              slurm_options = list("cpus-per-task" = "1", "mem" = "2G"))

saveRDS(sjob, file = "sjobW.rds")
```

which direct SLURM to distribute the computation among various Engaging nodes through use of the "pars" dataframe and a vital function slurm_apply within rslurm. It would further be possible to pass parameters to the parallel processing via "pars" (e.g., different values $N$, $p$, $q$ for different scenarios), but we won't demonstrate this. While the data count is 40 as previously, the task count is 250, leading to a sample size of 10000. Engaging seems to allow 60 or slightly more tasks to run simultaneously; hence most of the tasks are temporarily held in a queue before staggered entry. A new folder _rslurm_test_W is created containing 250 output files (all empty) and 250 RData files. Observe that our R function rnd_W() provides only the moments, not the data, thus the RData files are all tiny. Other files (params.RData, slurm_run.R, submit.sh inside the folder and files W_initial_err.txt, W_initial_out.txt outside) are also created but do not concern us. The truly essential file for subsequent work is sjobW.rds, as the critical lines:

```
sjob <- readRDS("sjobW.rds")

res <- get_slurm_out(sjob, outtype = "table")
```

of the final R program indicate. I believe that rslurm's ability to neatly wrap up the multiplicity of parallel processing "loose ends", as well as the convenience it offers of being able to perform everything interactively within R or R Studio, makes this tool a strong candidate for any analyst's use.

Please feel free to write to me (sfinch@mit.edu) with questions and suggestions on how to improve this tutorial.