How to use the cluster for distributed computing in MATLAB and Python

January 17, 2017

1 When to use the cluster

The LNC has the availability of a *cluster*, which is a collection of powerful processors (CPUs) with multiple processing *cores*, together with a lot of memory (RAM) and diskspace. The cluster can be used to process data that requires a lot of RAM or fast CPUs. However, the main benefit of a cluster is that it allows users to run processes on several cores at the same time, i.e. in parallel. This can speed up your analysis, in particular when you can run the same process (i.e. analysis) on different datasets (e.g. subjects), or using different analysis parameters (e.g. frequency bands).

- *The cluster can speed up your analysis by running several analyses in parallel*
- *The analyses should only differ in a single parameter, such as datasets or analysis settings*

2 Resource manager: SLURM

The cluster can run several analyses in parallel because it has a lot of cores that can run processes independantly. However, this does mean that your analysis somehow has to be (smartly) distributed over these different cores. On top of that, memory requirements also have to be managed. For instance, if a single analysis uses a lot of memory, this will limit the number of cores that can run in parallel, since memory is a limited resource shared between cores.

To deal with these issues of distributing processes and managing resources, the cluster runs SLURM (https://slurm.schedmd.com/). SLURM runs on the clusters, which run on LINUX as an operating system. To tell SLURM how to distrubute your analyses you will therefor have to use some *tsh*, which is a basic LINUX language. In practice this will simply be a matter of editing an example.

- *The distribution of your analysis (cores/memory) is managed by SLURM*
- *SLURM uses LINUX's tsh language*

3 Determining your resources

Although SLURM is smart when it comes to managing resources for your analysis, it knows nothing *about* your analysis. Think of it as a mail service: UPS knows exactly how to deliver packages from A to B. You can ask whether a packages is in transit, when it left, or if it arrived (*when* is always just a guess). However, they know nothing about the content of the packages. In fact, you will have to weigh and measure the package yourself. If any of these are wrong, they might show up with a van that is too small, or might not be able to lift up the package. In practice, this means you should at least have a rough estimate of how much memory your analysis needs: If you ask for too much, SLURM will be

inefficient, i.e. submit less processes to a core than memory could hold, while if you not ask for enough memory, your analyses will break because they will run out of memory.

- *For SLURM to run efficiently, you will know the amount of memory that each analysis will need*

4 Jobs

In cluster lingo, every distributed analysis is called a *job*. The job that is delivered to each core consists of a (series of) commands. What those commands are depends of course on you, but they are probably include either *MATLAB* or *Python* functions. The rest of this text will describe mainly the MATLAB user scenario, but it should help you on your way for Python as well.

Imagine you told SLURM to distribute 10 jobs, each needing 10G of memory. Each job will run a MATLAB (or Python) function, e.g. *analysis.m* (or *analysis.py*). As said earlier, these analyses should only differ in a single parameter, i.e. subject number. What we have not mentioned yet, is that this has to be an integer number. This means that your script will need to take as an input argument an integer argument that will result in the analysis being applied to the right subject. E.g. *analysis(1), analysis(2), analysis(3)*, etc., should result in the analysis of subject 1, 2, 3, etc.

5 Making your script suitable for job-distribution

Before you decided to use the cluster, you might have been used to using a supervening MATLAB script to cycle through subjects by providing filenames as *strings*, such as this:

```
Function analysis-over-subjects

  subjectfile{1} = 'subject1.fif';
  subjectfile{2} = 'subject2.fif';
  subjectfile{2} = 'subject3.fif';
  for i = 1 : 3
    analysis(subjectfile{i})
  end
```

This will not run on the cluster, because the input argument for *analysis* is a string. The function does not know the mapping between the i and the filename. You will therefor need to rewrite your analysis to a function that is self-contained. In other words, the function needs to know exactly what will need to happen given only the single integer identifier, leaving out the supervening script:

```
Function analysis(subjectnr)

  subjectfile{1} = 'subject1.fif';
  subjectfile{2} = 'subject2.fif';
  subjectfile{2} = 'subject3.fif';

  filename = subjectfile{subjectnr};
  % do your analysis on this filename
```

It is possible to distribute jobs that run on more than one 'dimension', such as both subjectnumber and frequency band. However, you will need to build in this complexity by parsing the input argument yourself. E.g., 1:4 = subjectnumber 1, with frequency band 1, 2, 3 and 4, 5:10 = subjectnumber 2 with frequency band 1, 2, 3, 4, etc.

6 The job identifier

We have almost covered all the important steps. However, we need to explain how to create and extract the input argument for your function. This might be a little complex but we will do it step-by-step. We just covered the third step, but still need to discuss the first two:

1. Create the job identifiers on the cluster front.
2. Extracts the identifier from the LINUX environment (on each core) and call your analysis script with that identifier
3. Have your MATLAB analysis script execute the right analysis based on the identifier (on each core).

6.1 Creating the job identifier

We will look a some full examples later, but for now let's focus on understanding the job identifier. You will create a SLURM script that has a line such as the first one below. This will tell SLURM to create 26 jobs, with an identifier going from 1 to 26. On the next line it tells SLURM to then to let MATLAB execute a (MATLAB) function called *launch-batch.m*.

```
#SBATCH --array=1-26
matlab -nodesktop -r launch-batch.m
```

6.2 Extract the job identifier

Note that we have not let SLURM start our analysis script directly. Instead, we let it execute an intermediate MATLAB script. We use this intermediate script to extract the job identifier, and to pass it to our main analysis script. The job identifier is stored in the LINUX environment, which is different for each job. We extract the job identifier from the LINUX environment using the *getenv* command. Note that this identifier will be a string, so we need to convert it to an integer using *str2num* before passing it to our analysis script.

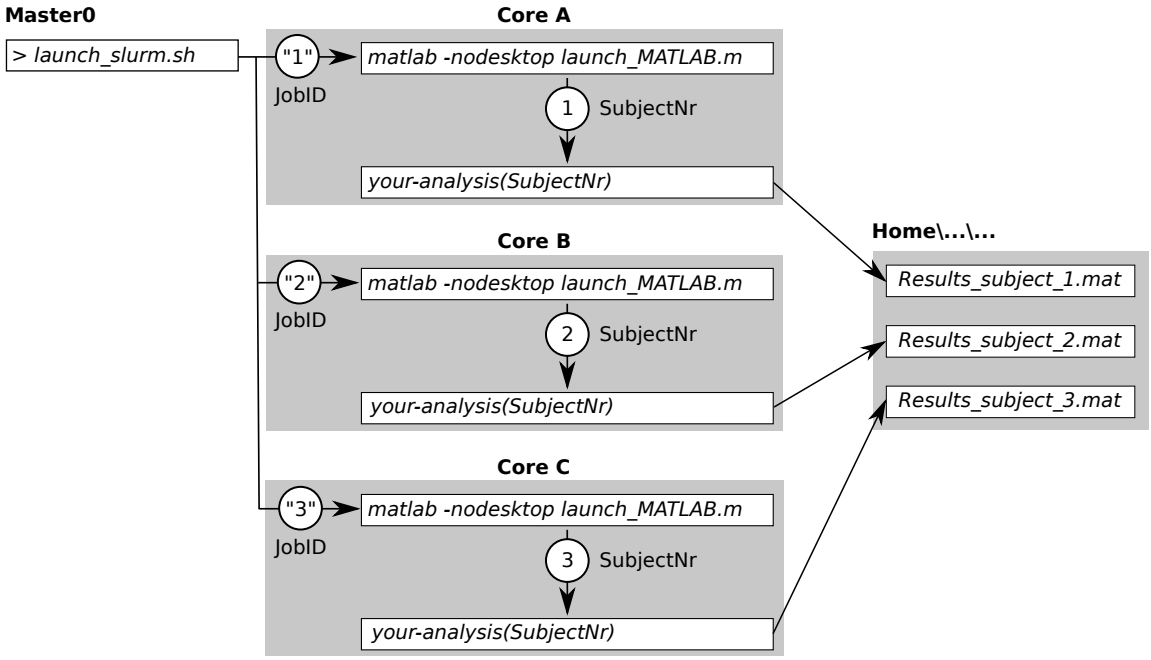
```
taskID = getenv('SLURM_ARRAY_TASK_ID');
subjectnumber = str2num(taskID);

your_main_analysis_script(subjectnumber);
```

It is probably possible to have your analysis script extract the job identifier and not use this intermediate step. However, by doing it in the way described, you will not need to adapt all your analysis scripts, but rather call them as you would do on your own computer. Also, using Python is a lot easier, since all commands and functions can be called from the command line and easily bashed.

7 Recap

So to recap, one you have a MATLAB function that can take an integer value to analyse a particular subject or parameter you create two scripts to distribute it over cores: a bash script to launch SLURM, and a MATLAB script to extract and pass through the unique job identifier. Ofcourse you will also have to make sure your analysis scripts writes the results. Given that your home or project directly will be mapped on the same path for each core, this is easy.



8 Example SLURM launch script

In the example below note the other recommended functionality. You will need to specify an ouput directory and name for error messages and output messages. A job-name will make it easier to find your job between others. Finally, it is important to specify the two lines with *share* and *mem-per-cpu*. The latter takes as input the number of kilobytes memory that will be reserved for the task. When this is below the amount of memory per CPU (say <20Gig), SLURM is able to run jobs in parallel over its several cores. Also note that MATLAB is started using the *-nodesktop* option because you will not be able to use a graphical interface. Finally, the loading of the MATLAB module is needed to add (a specific version of) MATLAB on the path.

```
#!/bin/bash
#SBATCH --job-name=WANDER
#SBATCH --output=/home/swhitmarsh/slurm/WANDER%A_%a.out
#SBATCH --error=/home/swhitmarsh/slurm/WANDER%A_%a.err

#SBATCH --array=1-26

#SBATCH --share
#SBATCH --mem-per-cpu=2000

module load matlabR2016b

cd /home/swhitmarsh/WANDER/scripts

matlab -nodesktop -r launch_WANDER_batch
```

Once you made the shell script such as the one above, and saved it as e.g. *launch-bash.sh*, you can run it by using *sbatch* :

```
sbatch launch-bash.sh
```

9 Example intermediate MATLAB script

This intermediate MATLAB script is used to setup the path for MATLAB, as well as specifying some other parameters that can be easily changed in different runs of the analysis.

```
function launch_WANDER_batch

addpath('/home/swhitmarsh/WANDER/scripts/');
addpath('/home/swhitmarsh/WANDER/fieldtrip/');

taskID = getenv('SLURM_ARRAY_TASK_ID');
isubject = str2num(taskID);

force = 1;
timing = 'cue';
rootpath = 0;
restingstate = 0;

WANDER_ERF(isubject,force,timing,rootpath);
```

10 Example MATLAB analysis script

Well, this one depends fully on you of course.

```
function ERF = WANDER_ERF(isubject,force,timing,rootpath)

% do analysis on subjectnumber isubject
```

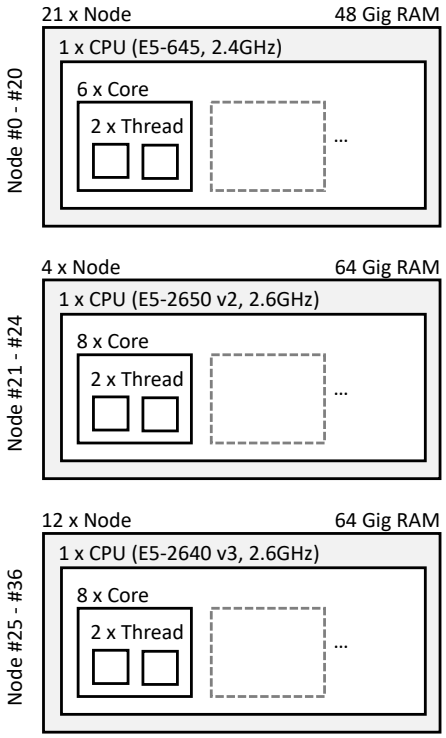
11 The resources

The server contains 37 nodes (not all operational at this time) that have either 48Gb, or 64Gb RAM (See figure below). Each node has several cores and each core allows for two parallel processes (threads). This architecture neccitates some careful considerations on the use of CPU versus RAM.

12 Monitoring and deleting jobs

You can use the command *squeue* to monitor all the jobs that are currently running. The first column will give you an identifier (number) which you can use to delete the job, using *scancel*. E.g.:

```
scancel 2666
```



13 Further notes

- Please note that creation of figures is not supported and will crash your job.