Configuring Snakemake, BigDataScript, and WDL / Cromwell for LSF

Many (all?) bioinformatics groups use cloud or cluster computing to handle grunt work such as sequence alignment. They use scheduling systems such as Sun Grid Engine and LSF to submit jobs to the cluster. But, it's becoming more common to use one of many modern pipelining tools. These pipelining tools abstract away the details of job submission, getting rid of boilerplate that would otherwise appear every time you build a pipeline. This makes your code **much easier to read.** (They also have secondary benefits -- some have map-reduce-like constructs for simple parallelization and many can recover automatically from interruptions without re-doing completed steps.)

Here at UMass, we use LSF to submit jobs. I've configured a number of modern pipelining tools on our cluster, some of which more or less natively support LSF and others that I had to coax. Here's how I did it.

Underlying pattern

In each case, you first download and install the pipelining tool. It will have a wrapper script for jobs to specify how it interfaces with your cluster's job manager. The trick is to find out where that is and modify it correctly.

Cromwell

Cromwell is the muscle behind the Broad Institute's wonderfully readable Workflow Description Language (WDL). To install Cromwell, I got the download and did my best to implement the LSF interface based on these instructions. Everything almost worked, but I hit a snag where Cromwell lacked permissions to execute the shell scripts that it creates. If you read that thread, you'll see how this issue was eventually resolved: it turns out if you use `bash myscript.sh` instead of `./myscript.sh`, you won't need execution permissions for the script itself.

Here is the job submission/monitoring file that I used to interface between Cromwell and LSF.

```
include required(classpath("application"))
backend {
    default = LSF
        providers {
            LSF {
             actor-factory = "cromwell.backend.impl.sfs.config.ConfigBackendLifecycle
ActorFactory"
             config {
                concurrent-job-limit = 16

                 runtime-attributes = """
                 Int cpu
                 Int nthreads
                 Float? memory_mb
                 """

                 submit = """
                 bsub \
                 -J ${job_name} \
                 -cwd ${cwd} \
                 -R rusage[mem=${memory_mb}] \
                 -n ${nthreads} \
                 -W ${cpu} \
                 -o ${out} \
                 -e ${err} \
                 " bash ${script} "
                 """

                 kill = "bkill ${job_id}"
                 check-alive = "bjobs ${job_id}"
                 job-id-regex = "Job <(\\d+)>.*"
             }
            }
        }
}
```

Room for improvement

Looking back at this with more experience, I am tempted to set the concurrent job limit much higher -- maybe 1000 or 2000. With LSF, my understanding is that you might as well send in lots of jobs, and if there's not enough cores at the moment, it will just leave some of them pending. I wouldn't send in 1,000,000 at once because my intuition is that sad things could happen, but in the past year I've seen that LSF can handle 1000 or so in the queue with no issue. According to the Cromwell devs, that parameter is not meant to be a full-scale

solution, so you can keep an eye out for what's developed in the future.

BigDataScript

BigDataScript (BDS) is a pipelining tool that imitates the look and feel of Java. It is used, among other places, by Anshul Kundaje's ENCODE-compliant ChIP-seq and ATAC-seq pipelines.

When I first tried BDS, it already had some Perl scripts for LSF job submission and monitoring. But, there were some errors I had to work through. The process is detailed in issue #33, and you can get the final result here.

To install BDS and run it on LSF:

1. Install the download for your OS and then do `export PATH=$PATH:$HOME/.bds`.
2. Move or symlink the scripts from BigDataScript/config/clusterGeneric_LSF/ into `BigDataScript/config/clusterGeneric/`.
3. When you invoke your pipeline, pass in the option `-s generic`. This tells BDS to look in `BigDataScript/config/clusterGeneric/` for job submission and monitoring, so it will find the interface to LSF.

Room for improvement

Does anyone who's reading know how to avoid overwriting `clusterGeneric/`? Can you just point BDS somewhere else?

Snakemake

By the time I sslithered into Ssnakemake, there was already a nice demo for LSF use here on Kamil Slowkowski's blog. Instead of writing a whole script, you can just pass a prefix in when you call Snakemake. Slowkowski's example is

```
snakemake --jobs 999 --cluster 'bsub -q normal -R "rusage[mem=4000]"'
```

To make this more flexible, for instance if you don't always need 4000M of memory or you don't always want to specify the job queue named `normal`, you can add some Snakemake wildcards.

```
snakemake --jobs 999 --cluster 'bsub -n {cluster.n} -R {cluster.resources} -q {cluste
r.queue}'
```

Just make sure your wildcards follow the same format as mine: `cluster.<thing>`. This has to do with how the actual values get filled in, which is explained below. First, here is the fully flexible job-submission command that I use.

```
snakemake \
    --snakefile my_Snakefile \
    --cluster-config cluster_config.json \
    --jobs 999 \
       --cluster \"bsub \
        -J {cluster.name} \
        -W {cluster.time} \
        -R {cluster.resources} \
        -n {cluster.n} \
        -e {cluster.err} \
        -o {cluster.output} \
    \"
```

Notice that I added an extra option here: `--cluster-config cluster_config.json`. That's because in Snakemake, you don't put the computational requirements inline as you would with WDL or BDS. They have to be in another file called the cluster config file. Details are here, but I'll give a brief intro below to keep the post self-contained.

You typically write one cluster config file for each Snakefile. The cluster config file is structured as a JSON dictionary that contains one level of (sub-)dictionaries. In the outer dictionary, keys correspond to task names in the corresponding Snakefile. There's an extra for defaults, which uses the reserved key `__default__`. In each inner dictionary, you specify keys that you can specify, and these correspond to the wildcards in the LSF submission command. Here's an example.

```
{
"__default__" :
    {
        "time"      : "1:00",
        "n"         : "1",
        "resources" : "\"rusage[mem=4000] span[hosts=1]\"",
        "name"      : "snake_{rule}_{wildcards}",
        "output"    : "logs_{rule}_{wildcards}.out",
        "err"       : "logs_{rule}_{wildcards}.err"
    },

    "some_memory_intensive_task" :
    {
        "resources" : "\"rusage[mem=30000] span[hosts=1]\""
    },

    "a_quick_task" :
    {
        "time" : "0:10"
    }
}
```

The cluster config file has to be valid JSON or YAML. To avoid hassle when you write your own:

1. Don't use tab characters.
2. Check the line endings for missing commas every time you edit the file.

You may notice that my cluster config file has lines that refer to `{rule}` and `{wildcards}`. Those allow Snakemake to cross-reference the cluster config file with the corresponding Snakefile. For example, look at the stderr output file. This is ultimately fed to LSF's `-e` argument in the submission script. To get to the submission script, it comes from the wildcard `{cluster.err}`, and so in the example `cluster_config.json` file, it comes from value at the `err` key. This value is, by default, `"logs_{rule}_{wildcards}.err"`. Thus, Snakemake will look in the Snakefile, and if it's submitting a job for a rule called `align` with wildcards describing the file to be aligned -- say `S1_L001_R1.fastq` -- then this is the final error log name.

```
logs_align_S1_L001_R1.fastq
```

This gets dumped straight into Snakemake's working directory when the job finishes. If you want things a little more organized, you can make two changes.

1. Change `"logs_{rule}_{wildcards}.err"` to `"logs/{rule}/{wildcards}.err"`.

2. Early on your Snakefile, make the folder `logs/{rule}/` for each rule.

Don't do step 1 without step 2. LSF will complain that it can't find the spot where you want the error file, and very probably **it will instead email you the stderr output for each job submitted under that rule.** Unless you want to clog the outgoing email queue and look silly in front of the sysadmins, take care of this first and test it with a small job. If you want to be extra cautious here, there are also other ways to tell LSF not to email you.

Room for improvement

The process above gets cumbersome, especially writing a new cluster config for every snakefile. There is actually now a better way. Maybe I will dig into this and write another tutorial if I do more with Snakemake in the future.

Martian

Martian is an awesome open-source pipelining tool built by the folks at 10X Genomics. When I installed it as part of cellranger, it worked for LSF out of the box! But, I still ended up tinkering with the configuration, because there were a couple of weeks over the summer when one of the job queues on the UMass cluster was very heavily used, and I wanted an option to avoid that queue. I'll describe this in the context of 10X cellranger, because I've never used Martian elsewhere. First, find the Martian installation inside cellranger and go to the job templates.

```
cd /path/to/your/bioinformatics/tools/cellranger-2.1.0/martian/???
```

Copy the LSF one. I named mine after the queue I wanted to avoid.

```
cp config_lsf.txt config_lsf_nolong.txt
```

Edit it. I changed the walltime from `-W 10:00` to `-W 4:00` to shorten my jobs and make them eligible for a quicker queue. (Quit quagmire queues. Quicker queues compute quite quickly.) This could cause problems if cellranger sends in a big job that takes longer than 4 hours, but in my experience it always submits many small jobs.

Crucial step: Martian does not yet know that your new config file exists. Tell it where to look by *carefully* modifying `cluster_configurations?????.json`. It's a JSON dictionary with entries that look like this (here's the default one for LSF).

???

Modify the name and the file it points to.

To test out your new configuration, you can invoke `cellranger count` as usual, but with `--jobmode lsf_nolong`.

```
PLS ADD EXAMPLE HERE THX
```

Takeaways

If you've reached this post, it's probably because you have downloaded a pipelining tool and you need to configure it. But, if you're still choosing which one to use, here are my picks. For absolute beginners or for simple tasks, I recommend WDL -- it keeps things very simple and clean. For an experienced programmer who needs something very flexible and who is willing to risk a little more chaos, I might suggest Snakemake.

Disclaimers

You're free to reuse the code snippets I provide here, but it's given without any guarantees on performance or behavior. In particular, *I know nothing about security* and my solutions may contain loopholes.