SciNet Setup

This guide is intended to assist with initial setup on the SciNet Niagara supercomputer, and to show how to use it with `galpy`. For more detailed information on using Niagara, please see the SciNet documentation: https://docs.scinet.utoronto.ca/index.php/Niagara_Quickstart.

1 Accessing Niagara

1. First, you'll need a Compute Canada account. If you don't already have one, speak with Jo about getting registered.

2. Log in to Compute Canada at https://ccdb.computecanada.ca/security/login.

3. Navigate to **My Account > Apply for a consortium account**, and apply for a SciNet account. You should be granted access within two business days.

4. After receiving a confirmation email that your account has been granted, you may now `ssh` into the Niagara login nodes using your Compute Canada user name and password:

   ```
   $ ssh USERNAME@niagara.scinet.utoronto.ca
   ```

5. You may now add an SSH key and set up your Linux environment as usual. For further instruction, see the SciNet Documentation at https://docs.scinet.utoronto.ca/index.php/SSH_keys or the **Setup and SSH keys** section of the group server page at http://astro.utoronto.ca/~bovy/group/compute_servers.html.

2 Installing and Using Software

Many software modules, including Python and the necessary compilers to install `galpy`, are pre-installed on Niagara, however they are not accessible by default. Instead, you must explicitly load each software module that you want to use at any given time. The command to load a module is:

```
$ module load <module-name>
```

To view a list of currently loaded modules, type:

```
$ module list
```

To view a list of modules that are currently available to load, type:

```
$ module avail
```

To view the full list of software installed on Niagara, type:

```
$ module spider
```

Finally, to clear all currently loaded modules, type:

```
$ module purge
```

Below are the step-by-step instructions for installing `galpy` for use on Niagara. For more general information on loading software, see the SciNet documentation.

2.1 Installing `galpy`

1. We will first set up an Anaconda virtual environment to install our own Python modules:

   ```
   $ module load python/3.6.4-anaconda5.1.0
   $ conda create -n galpyEnv python=3.6
   ```

2. Now, activate the new environment and install the Scientific Python suite:

   ```
   $ source activate galpyEnv
   $ pip install numpy, scipy, matplotlib, astropy
   ```

3. `galpy` should be installed from its source files for use with SciNet. Create a directory to hold the repository, and download the source using `git`:

   ```
   $ module load git
   $ git clone https://github.com/jobovy/galpy.git
   ```

4. Now, navigate to the repository. To install galpy, load the necessary compiler and `gsl`, then run `setup.py` in development mode:

   ```
   $ module load gcc gsl
   $ python setup.py develop
   ```

5. Whenever you want to use `galpy`, you must first reactivate the Python virtual environment in which it is installed and load the `gsl` module. You can save a combination of modules to streamline this process. Simply load the modules you wish to save and do:

   ```
   $ module save MYMODULES
   ```

   where `MYMODULES` is the name that you give to the combination of modules. To later reload this combination, do:

   ```
   $ module restore MYMODULES
   ```

3 Submitting Jobs

Niagara uses the Slurm Workload Manager for scheduling compute jobs. When requesting compute time on Niagara, you must send all commands through Slurm. This section provides a minimal example of submitting a job using Slurm. More detailed examples, and information about limits for submitting jobs, can be found in the SciNet documentation.

Below is an example of a job submission script `example_job.sh` to Slurm.

```
1  #!/bin/bash
2  #SBATCH --nodes=1
3  #SBATCH --ntasks=80
4  #SBATCH --time=1:00:00
5  #SBATCH --job-name=example_job
6  #SBATCH --output=example_output.txt
7  #SBATCH --mail-type=ALL
8  #SBATCH --mail-user=email@gmail.com
9
10 module restore MYMODULES
11 source activate myEnv
12
13 cd $SLURM_SUBMIT_DIR
14
15 python myJobScript.py
```

This script does the following:

- The first line indicates that the script is a Bash script.
- The lines that begin with `#SBATCH` are Slurm parameters.
  - Here, we have requested to run a job on 1 compute node for 1 hour. All jobs are submitted in multiples of nodes, and each node has 40 cores.
  - We have also indicated that our node will run 80 tasks or processes in parallel. Here, we are using hyperthreading to split each of the 40 physical cores into 2 logical cores. For certain applications, this can improve efficiency.
  - The console output of the program will be saved to `example_output.txt` in the same directory as the submission script.
  - Finally, we have also requested to receive an email when the job starts, finishes, or fails.
- We then restore a combination of modules necessary to run our script, and activate a Python virtual environment.
- Finally, we `cd` to the directory containing the submission script and run a Python script.

This script can be submitted to Slurm using the `sbatch` command:

```
$ sbatch example_job.sh
```

You can view the status of your submitted jobs by typing:

```
$ squeue -u USERNAME
```

Once the job makes it through the queue, it will be assigned an entire node which it will have full control over. As such, your jobs should make use of the entire node. Note that you can only write to the `$SCRATCH` directory from the compute nodes – your `$HOME` directory is read-only from the compute nodes.

As a general rule of thumb, your jobs will get through the queue faster if you split them up into smaller chunks that can run quickly – for instance, instead of submitting a single job to run for 24 hours, it might be faster to submit 12 jobs each taking 2 hours.

For further instructions on submitting jobs, including information on debugging, job limits, and automatically resubmitting jobs, see the Niagara Quickstart page and the SciNet FAQ.

4 Running Orbit Integration Suites

Orbit integration with multiprocessing in `galpy` can be accomplished using the `Orbits` class (which, as of December 2018, is available in the `orbits` branch). This class uses the same syntax as the standard `Orbit` class, but is capable of storing multiple orbits simultaneously and integrating them in parallel. For example, given a list of initial conditions `init` for multiple orbits, they can be integrated in parallel using 80 cores as follows:

```
1 import numpy as np
2 from galpy.orbit import Orbits
3 from galpy.potential import MWPotential2014
4
5 t = np.linspace(0, 1, 100)
6 o = Oribts(vxvv=init)
7 o.integrate(t, MWPotential2014, numcores=80)
```

The integrated coordinates of the orbits can then be extracted and saved to a file.

When running very large integration suites on SciNet with millions of orbits, the memory requirements will quickly become unmanageable. As such, the integration must be broken up into batches. You could accomplish this by splitting up your initial conditions into many batches, and integrating each in a separate job submission to Slurm. Alternatively, you could submit larger jobs to Slurm, but only load a portion of the data at any given time. You can write your own tools to do this, or optionally you can use the tool available at https://github.com/mwbub/multiorbit. This tool accepts the initial conditions and integration parameters, and will automatically split

up the data into chunks before integrating. It will then save the results of the integration in a `.fits` file. It is also capable of restarting the integration in case of an interruption, which can be useful for running integration suites that take longer than the maximum allowed compute time on SciNet. To use the tool as in the above example, do the following:

```
from multiorbit.integrate import integrate_chunks

filename = '<path-to-dir>/myFile.fits'  # Save the output here
integrate_chunks(t, MWPotential2014, filename, vxvv=init,
                 numcores=80, chunk_size=1000000)
```

This will save only the final time step of the integration. To save every time step, add `save_all=True` to the call to `integrate_chunks`. Once the integration is complete, the output file can be accessed using any program capable of reading `.fits` files. For example, using `astropy`:

```
from astropy.table import Table

filename = '<path-to-dir>/myFile_0.fits'  # Files are numbered
                                          # by time step
Table.read(filename, format='fits')
```

The data columns can then be accessed using the `astropy.table.Table` syntax.

This guide was written by Mathew Bub in December 2018.