MOOSE Installation Instructions for DCC

Duke University

December 6, 2020

1.0.2

Casper Versteeg
Gary Hu
Roozbeh Rezakhani

Contents

Revision Table

Changes	Author(s)	Version
[2020/07/10]—Created document from pinned post on Slack	Versteeg, Casper, Hu, Gary, Rezakhani, Roozbeh	1.0.0
[2020/07/14]—more useful tricks	Hu, Gary	1.0.1
[2020/12/06]—Added note on binding issue. Fixed script.	Versteeg, Casper	1.0.2

1 Introduction

These are some basic instructions on installing MOOSE on the Duke Compute Cluster[1] (DCC). Hopefully, they are detailed enough such that users with limited Linux experience are able to get up and running with minimal assistance. However, if something is unclear, or if issues arise while following these instructions, feel free to reach out for assistance on the `#cluster` channel on Slack.

2 Installing

2.1 Preliminary setup

Upon logging in to DCC (using `ssh <net_id>@dcc-slogin.oit.duke.edu`) you'll need to load the modules required to compile PETSc, LibMesh and MOOSE. You can find what modules are available to you (as installed by DCC admins, ready for you to use), with `module avail`.

Note 2.1: *(purging)*
This may not be necessary, but you might also want to run `module purge` to unload any modules that may be loaded by default. To view the currently loaded modules, run `module list`, and you should see the output `No Modulefiles Currently Loaded.`.

Next, you will have to load the following modules:

```
module load cmake/3.10.2
module load Python/3.6.4
module load GCC/7.4.0
module load MPICH/3.2.1
```

2.2 Installing MOOSE

From here on, you can follow the HPC install instructions[2] on the MOOSE website. The section "PREFIX Setup" is only required for installation, so you will not need to add any of those variables to your `.bashrc`. You should be able to `copy + paste` the code in the "PETSc" section without any issues.

[1] https://rc.duke.edu/dcc
[2] https://mooseframework.org/getting_started/installation/hpc_install_moose.html

Note 2.2: *(complete instructions)*
You may also find a complete set of instructions in Appendix B, to eliminate any ambiguity.

You can create a MOOSE profile per the instructions (by creating a `.moose_profile`), however you can also simply add those lines to your `.bashrc` to avoid creating a new profile. By adding them to your `.bashrc` your session will be configured to use MOOSE every time you log in to DCC. This is really a matter of preference—there should not be any issues one way or another.

Note 2.3: *(automatically loading the MOOSE environment on login)*
f you choose to maintain a `.moose_profile` per the instructions, you should add the following lines to your `.moose_profile` so that the modules required by the MOOSE environment are also loaded:

```
module load cmake/3.10.2
module load Python/3.6.4
module load GCC/7.4.0
module load MPICH/3.2.1
```

you can then add the following lines to your `.bashrc` so that the MOOSE profile (and the corresponding modules) are automatically sourced:

```
# moose environment
if type "module" &> /dev/null; then
  source $HOME/.moose_profile
fi
```

2.3 Installing Raccoon

If you will be using Gary Hu's Raccoon app[3], you may stop following the HPC install instructions at the ”Obtaining and Building MOOSE” section, since Raccoon includes MOOSE as a submodule. To get Raccoon set up, just do the usual `git` dance:

[3] https://hugary1995.github.io/raccoon/install/index.html

```
mkdir ~/projects && cd ~/projects
git clone https://github.com/hugary1995/raccoon.git
cd raccoon
git checkout master
git submodule --init
```

Since we did not install MOOSE using Conda like you would on your local machine, section 3 "Compile libMesh (Optional)" is—in fact—*not optional*.

Note 2.4: *(compiling LibMesh)*
When compiling LibMesh, you may be hindered by the limited memory available on the DCC login node. As a means of mitigating this issue, you can specify the compile method as follows:

```
METHOD=opt ./update_and_rebuild_libmesh.sh --with-boost
```

After compliling LibMesh, you should be able to compile Raccoon, and run tests, without any issues.

3 Running

DCC uses a task scheduling system called SLURM[4], which manages job priority and queues for the various researchers using DCC. SLURM is typically quite good at what it does, but often may not provide you with helpful information if your job is terminated prematurely. To gain some confidence that your job will run on DCC, you can create an interactive session on the login node—essentially giving you DCC-grade horsepower in the login node's terminal you have access to. As a shortcut, you can add the following handy-dandy bash function to your `.bashrc`:

```
# interactive testing
InteractiveSession () {
  srun --account=dolbowlab -p dolbowlab --pty -N 1 -n $1 -c $2 --mem $3
   ↪ --time=5- bash -i
}
```

[4] https://slurm.schedmd.com/documentation.html

To trial your job on e.g. 1 node, 35 processors, and 200GB RAM (`$1` , `$2` , and `$3` arguments in the function), do the following in your terminal (remember to `source ~/.bashrc` before using the command):

```
InteractiveSession 1 35 200G
```

You may find it useful to keep a bash script in the folder from which you run your jobs on DCC. Here is an example script:

```
1  #!/bin/bash
2  #SBATCH -N 1
3  #SBATCH --ntasks=35
4  #SBATCH --job-name=aMeaningfulTitle
5  #SBATCH --partition=dolbowlab
6  #SBATCH --mem-per-cpu=10G
7  #SBATCH -o outputFileName
8
9  export SLURM_CPU_BIND=none
10
11 echo "Start: $(date)"
12 echo "cwd: $(pwd)"
13
14 mpirun <path_to_executable> -i <path_to_input_file>
15
16 echo "End: $(date)"
```

Note 3.1: *(MPI)*

You will not have to use the syntax `mpirun -n <#cpu> ...` . SLURM will automatically assign CPUs to your job based on the flags such as `--ntasks` .

Note 3.2: *(CPU Bind)*

There was a recent issue that caused CPU usage to be compressed to a single CPU. Basically, all the 35 tasks created by line 3 in the script above would be bound to only one CPU. DCC admins Tom Milledge[a] and Victor Orlikowski[b] recommended adding `export SLURM_CPU_BIND=none` , as in line 9 above, after the SLURM directives.

[a] tom.milledge@duke.edu
[b] vjo@duke.edu

When you wish to submit your job to the SLURM queue, simply use the `sbatch` command as follows:

```
sbatch <bash_script_name>
```

and just wait for it to finish...

Finally, you can check the status of your job with the `squeue` command, which has many formatting options. You can get a lot of useful information with the following:

```
squeue -u <netid> -o "%.18i %.9P %.8j %.8u %.8T %.10M %.12l %.6D %.18R %.5C
↪ %.10c"
```

Note 3.3: *(following the output)*
ou can use `tail` to continuously follow the output, e.g. `tail -f <outputFileName>`.

Note 3.4: *(navigate output)*
Your output file may be huge. If you use VIM to view this file, you can use `shift + G` to take you to the end of the file.

4 Some Final Notes...

Finally, it is worth pointing out that your home folder (i.e. `echo $HOME`) only has 10GB of storage space, which will seriously limit the size of the jobs you'll be able to run. SLURM will just kill your task if you run out of space in your home folder. Instead, you should probably run your jobs from the group directory:

```
  /hpc/group/dolbowlab/<net_id>
```

For convenience, you may want to set an environment variable in your `.bashrc` to access this folder quickly:

```
export JDDIR="/hpc/group/dolbowlab/<net_id>"
```

Then quickly jump to your folder with `cd $JDDIR`.

If you are OK with slower latency, and back up your work often enough, you can also create a directory for yourself in the `/work` folder, which has 450TB of storage[5]. However, this directory is not backed up, and any files older than 75 days will be purged automatically.

If you use the Raccoon app, its `examples` folder is not tracked by git and is ideal to serve as your work directory. To utilize larger storage space in either the dolbowlab group directory or the DCC work space, create a folder at your desired location and soft link it to your home directory. To do so (using the dolbowlab group directory), first create an `examples` folder:

```
mkdir /hpc/group/dolbowlab/<net_id>/examples
```

Then soft link it to your home directory:

```
ln -s /hpc/group/dolbowlab/<net_id>/examples ~/projects/raccoon/examples
```

Then you can use the `examples` directory the same way as you can locally, while also enjoying a larger storage space.

[5]Temporarily increased to 650TB due to increased remote computing demand during the COVID-19 pandemic

Appendices

A Horsepower

If you are curious, you can find out how much horsepower we have in our node with `scontrol show node dcc-dolbowlab-01`, which will print something like:

```
NodeName=dcc-dolbowlab-01 Arch=x86_64 CoresPerSocket=1
  CPUAlloc=35 CPUTot=70 CPULoad=35.34
  AvailableFeatures=(null)
  ActiveFeatures=(null)
  Gres=mathematica:8
  NodeAddr=dcc-dolbowlab-01 NodeHostName=dcc-dolbowlab-01 Version=19.05.4
  OS=Linux 3.10.0-1062.9.1.el7.x86_64 #1 SMP Mon Dec 2 08:31:54 EST 2019
  RealMemory=741580 AllocMem=204800 FreeMem=625894 Sockets=70 Boards=1
  State=MIXED ThreadsPerCore=1 TmpDisk=25587 Weight=1 Owner=N/A MCS_label=N/A
  Partitions=dolbowlab
  BootTime=2019-12-19T11:26:28 SlurmdStartTime=2019-12-19T11:24:08
  CfgTRES=cpu=70,mem=741580M,billing=70
  AllocTRES=cpu=35,mem=200G
  CapWatts=n/a
  CurrentWatts=0 AveWatts=0
  ExtSensorsJoules=n/s ExtSensorsWatts=0 ExtSensorsTemp=n/s
```

This essentially tells you the `dcc-dolbowlab-01` node has 70 CPUs in one node, with a total of 700GB of memory.

B Complete Instructions

```
module load cmake/3.10.2
module load Python/3.6.4
module load GCC/7.4.0
module load MPICH/3.2.1

export STACK_SRC=`mktemp -d /tmp/stack_temp.XXXXXX`
export PACKAGES_DIR=$HOME/moose-compilers

```

```
cd $STACK_SRC
curl -L -O
↪ http://ftp.mcs.anl.gov/pub/petsc/release-snapshots/petsc-3.11.4.tar.gz
tar -xf petsc-3.11.4.tar.gz -C .

cd $STACK_SRC/petsc-3.11.4

./configure \
--prefix=$PACKAGES_DIR/petsc-3.11.4 \
--with-debugging=0 \
--with-ssl=0 \
--with-pic=1 \
--with-openmp=1 \
--with-mpi=1 \
--with-shared-libraries=1 \
--with-cxx-dialect=C++11 \
--with-fortran-bindings=0 \
--with-sowing=0 \
--download-hypre=1 \
--download-fblaslapack=1 \
--download-metis=1 \
--download-ptscotch=1 \
--download-parmetis=1 \
--download-superlu_dist=1 \
--download-scalapack=1 \
--download-mumps=1 \
--download-slepc=1 \
PETSC_DIR=`pwd` PETSC_ARCH=linux-opt

make PETSC_DIR=$STACK_SRC/petsc-3.11.4 PETSC_ARCH=linux-opt all
make PETSC_DIR=$STACK_SRC/petsc-3.11.4 PETSC_ARCH=linux-opt install
make PETSC_DIR=$PACKAGES_DIR/petsc-3.11.4 PETSC_ARCH="" test

if [ -d "$STACK_SRC" ]; then rm -rf "$STACK_SRC"; fi

mkdir ~/projects
cd ~/projects
git clone https://github.com/hugary1995/raccoon.git
```

```
cd raccoon
git checkout master
git submodule update --init

cd moose/scripts
METHOD=opt ./update_and_rebuild_libmesh.sh --with-boost

cd ../..
make -j N
./run_test -j N
```