Habanero - Job Examples

- Hello World
- C/C++/Fortran
- C/C++/Fortran MPI
- GPU (CUDA C/C++)
- Example of R run
- Installing R Packages on Habanero
- Local Installation
- Matlab
- Python and JULIA
- Tensorflow
- Jupyter Notebooks
- Spark

In order for the scripts in these examples to work, you will need to replace `<ACCOUNT>` with your group's account name.

Hello World

This script will print "Hello World", sleep for 10 seconds, and then print the time and date. The output will be written to a file in your current directory:

```bash
#!/bin/sh
#
# Simple "Hello World" submit script for Slurm.
#
# Replace <ACCOUNT> with your account name before submitting.
#
#SBATCH --account=<ACCOUNT>  # The account name for the job.
#SBATCH --job-name=HelloWorld  # The job name.
#SBATCH -c 1                  # The number of cpu cores to use.
#SBATCH --time=1:00           # The time the job will take to run.
#SBATCH --mem-per-cpu=1gb     # The memory the job will use per cpu core.

echo "Hello World"
sleep 10
date

# End of script
```

C/C++/Fortran

To submit a precompiled binary to run on Habanero, the script will look just as it does in the Hello World example. The difference is that you will call your executable file instead of the shell commands "echo", "sleep", and "date".

C/C++/Fortran MPI

Intel Parallel Studio

Habanero supports Intel Parallel Studio which provides a version of MPI derived from MPICH2. We encourage users to avail themselves of Intel MPI because it is faster and more modern than other versions. Also, all nodes on the cluster have Infiniband transport and that is the fabric that MPI jobs avail themselves of - which is another reason for a substantial boost of efficiency on the cluster.
To use Intel MPI, you must load the Intel module first:

```
module load intel-parallel-studio/2017
mpiexec ./myprogram
```

In order to take advantage of Habanero architecture, your program should be (re)compiled on the cluster even if you used Intel for compiling it on another cluster (like Yeti). It is important to compile with the compiler provided by the module mentioned above. Note that you may have to set additional environment variables in order to successfully compile your program.

These are the locations of the C and Fortran compilers for Intel Studio:

```
$ module load intel-parallel-studio/2017
(...) $ which mpiicc
/rigel/opt/parallel_studio_xe_2017/compilers_and_libraries_2017.0.098/linux/mpi/intel64/bin/mpiicc
$ which ifort
/rigel/opt/parallel_studio_xe_2017/compilers_and_libraries_2017.0.098/linux/bin/intel64/ifort
```

For programs written in C, use mpiicc in order to compile them:

```
$ mpiicc -o <MPI_OUTFILE> <MPI_INFILE.c>
```

The submit script below, named pi_mpi.sh, assumes that you have compiled a simple MPI program used to compute pi, (see mpi_test.c), and created a binary called pi_mpi:

```
#!/bin/sh

#SBATCH -A <ACCOUNT>
#SBATCH --time=30
#SBATCH -N 2
#SBATCH --exclusive

module load intel-parallel-studio/2017
mpiexec -bootstrap slurm ./pi_mpi

# End of script
```

The --exclusive flag will ensure that full nodes are being used in the runs (that's the reason why no memory specification is given). Each available core will give rise to another MPI thread. Without the flag, you can specify the number of tasks, or tasks per node, in order to limit the number of threads that will be created. For example, you can replace the directive containing the flag by:
#SBATCH --N 2
#SBATCH --ntasks-per-node=4

- and your MPI code will run on 8 threads, with 4 on each of the 2 nodes requested.

**Job Submission**

```bash
$ sbatch pi_mpi.sh
```

**OpenMPI**

Habanero supports also OpenMPI from the GNU family.

To use OpenMPI, you must load the following module instead:

```bash
module load openmpi/gcc/64
mpiexec myprogram
```

Your program must be compiled on the cluster. You can use the module command as explained above to set your path so that the corresponding `mpicc` will be found. Note that you may have to set additional environment variables in order to successfully compile your program.

```bash
$ module load openmpi/gcc/64
$ which mpicc
/rigel/opt/openmpi-2.0.1/bin/mpicc
```

Compile your program using `mpicc`. For programs written in C:

```bash
$ mpicc -o <MPI_OUTFILE> <MPI_INFILE.c>
```

**GPU (CUDA C/C++)**

The cluster includes two types of GPU servers: Nvidia K80s and Nvidia P100s.

- There are 14 K80 GPU servers, each with two dual K80 Tesla GPU accelerators, for a total of 4 GPU modules per server.
- There are 13 P100 servers, each with two P100 accelerators, for a total of 2 GPU modules per P100 server.

To use a GPU server you must specify the `--gres=gpu` option in your submit request, followed by a colon and the number of GPU modules you require (with a maximum of 4 per server for K80s and 2 per server for P100s).

Use the `--constraint=k80` or `--constraint=p100` along with the `--gres=gpu` directive if you’d like to request a specific type of GPU (k80 or P100).

Request a K80 gpu, specify this in your submit script.
To request a **P100** gpu, specify this in your submit script:

```bash
#SBATCH --gres=gpu
#SBATCH --constraint=p100
```

Not all applications have GPU support, but some, such as MATLAB, have built-in GPU support and can be configured to use GPUs.

To build your CUDA code and run it on the GPU modules you must first set your paths so that the Nvidia compiler can be found. Please note you must be logged into a GPU node to access these commands. To login interactively to a GPU node, run the following command, replacing `<ACCOUNT>` with your account.

```
$ srun --pty -t 0-01:00 --gres=gpu:1 -A <ACCOUNT> /bin/bash
```

Load the cuda environment module which will add cuda to your PATH and set related environment variables. Note cuda 8.0 does not support gcc 6, so gcc 5 or earlier must be accessible in your environment when running nvcc.

```
$ module load gcc/4.8.5
```

Load the cuda module.

```
$ module load cuda80/toolkit
```

You then have to compile your program using `nvcc`:

```
$ nvcc -o <EXECUTABLE_NAME> <FILE_NAME.cu>
```

You can compile `hello_world.cu` sample code which can be built with the following command:

```
$ nvcc -o hello_world hello_world.cu
```

For non-trivial code samples, refer to Nvidia's CUDA Toolkit Documentation.

A Slurm script template, `gpu.sh`, that can be used to submit this job is shown below:
#!/bin/sh
#
#SBATCH --account=<ACCOUNT>      # The account name for the job.
#SBATCH --job-name=HelloWorld    # The job name.
#SBATCH --gres=gpu:1             # Request 1 gpu (Up to 4 on K80s, or up to 2 on P100s are valid).
#SBATCH -c 1                     # The number of cpu cores to use.
#SBATCH --time=1:00              # The time the job will take to run.
#SBATCH --mem-per-cpu=1gb        # The memory the job will use per cpu core.

module load cuda80/toolkit
./hello_world

# End of script

Job submission

$ sbatch gpu.sh

This program will print out "Hello World!" when run on a gpu server or print "Hello Hello" when no gpu module is found.

Example of R run

For this example, the R code below is used to generate a graph "Rplot.pdf" of a discrete Delta-hedging of a call. It hedges along a path and repeats over many paths. There are two R files required:

hedge.R
BlackScholesFormula.R

A Slurm script, hedge.sh, that can be used to submit this job is presented below:
#!/bin/sh
#hedge.sh
#Slurm script to run R program that generates graph of discrete Delta-hedging call

#Slurm directives
#
#SBATCH -A astro                     # The account name for the job.
#SBATCH -J DeltaHedge                # The job name.
#SBATCH -c 6                        # The number of cpu cores to use.
#SBATCH -t 1:00                      # The time the job will take to run.
#SBATCH --mem-per-cpu 1gb           # The memory the job will use per cpu core.

module load R

#Command to execute R code
R CMD BATCH --no-save --vanilla hedge.R routput

# End of script

Batch queue submission

$ sbatch hedge.sh

This program will leave several files in the output directory: slurm-<jobid>.out, Rplots.pdf, and routput (the first one will be empty).

Installing R Packages on Habanero

HPC users have two options for R packages:

1. Install packages locally in your user space that can be called by your programs (faster, see below).
2. E-mail hpc-support@columbia.edu and request a package to be installed on all HPC nodes (slower).

Local Installation

After logging in to Habanero, start R:

$ module load R
$ R

You can see the default library paths (where R looks for packages) by calling .libPaths():
> .libPaths()
[1] "/usr/lib64/R/library"  "/usr/share/R/library"
[3] "/usr/lib64/R/site-library"

These paths are all read-only, and so you cannot install packages to them. To fix this, we will tell R to look in additional places for packages.

Exit R and create a directory rpackages in /rigel/<GROUP>/users/<UNI>/.

$ mkdir /rigel/<GROUP>/users/<UNI>/rpackages

Go back into R and add this path to .libPaths()

$ R
> .libPaths("/rigel/<GROUP>/users/<UNI>/rpackages/")

Call .libPaths() to make sure the path has been added

> .libPaths()
[1] "/rigel/<GROUP>/users/<UNI>/rpackages/"
[2] "/usr/lib64/R/site-library"
[3] "/usr/lib64/R/library"

To install a package, such as the "sm" package, tell R to put the package in your newly created local library:

> install.packages("sm", lib="/rigel/<GROUP>/users/<UNI>/rpackages")

Select appropriate mirror and follow install instructions.

Test to see if package can be called:

> library(sm)
Package 'sm', version 2.2-3; Copyright (C) 1997, 2000, 2005, 2007
A.W.Bowman & A.Azzalinitype
help(sm) for summary information

In order to access this library from your programs, make sure you add the following line to the top of every program:

.libPaths("/rigel/<GROUP>/users/<UNI>/rpackages/")

Since R will know where to look for libraries, a call to library(sm) will be successful (however, this line is not necessary per se for the install.packages(...) call, as the directory is already specified in it).

Matlab

Matlab (single thread)
The file linked below is a Matlab M-file containing a single function, simPoissGLM, that takes one argument (lambda).

simPoissGLM.m

A Slurm script, simpoiss.sh, that can be used to submit this job is presented below (implicitly, --cpu-per-task=1).

```bash
#!/bin/sh
#
# Simple Matlab submit script for Slurm.
#
#SBATCH -A astro                 # The account name for the job.
#SBATCH -J SimpleMLJob           # The job name.
#SBATCH -t 1:00                  # The time the job will take to run.
#SBATCH --mem-per-cpu=1gb        # The memory the job will use per cpu core.

module load matlab
echo "Launching an Matlab run"
date

#define parameter lambda
LAMBDA=10

#Command to execute Matlab code
matlab -nosplash -nodisplay -nodesktop -r "simPoissGLM($LAMBDA)" # > matoutfile

# End of script
```

Batch queue submission

```
$ sbatch simpoiss.sh
```

This program will leave several files in the output directory: slurm-<jobid>.out, out.mat, and matoutfile.

**Matlab (multi-threading)**

Matlab has built-in implicit multi-threading (even without applying its Parallel Computing Toolbox, PCT), which causes it to use several cores on the node it is running on. It consumes the number of cores assigned by Slurm. The user can activate explicit (PCT) multi-threading by specifying the number of cores desired also in the Matlab program.

The Torque submit script (simpoiss.sh) should contain the following line:

```
#SBATCH --c 6
```

The -c flag determines the number of cores (up to 24 are allowed).

For explicit multi-threading, the users must include the following corresponding statement within their Matlab program:
parpool('local', 6)

The second argument passed to parpool must equal the number specified with the ppn directive. Users who are acquainted with the use of commands like parfor need to specify explicit multi-threading with the help of parpool command above.

Note: maxNumCompThreads() is being deprecated by Mathworks. It is being replaced by parpool:

The command to execute Matlab code remains unchanged from the single thread example above.

Important note: On Yeti, where Matlab was single thread by default, it appeared that the more recent versions of Matlab took liberties to grab all the cores within a node even when fewer (or even only one) cores were specified as above. On Habanero, we believe this has been addressed by implementing a system mechanism which enforces the proper usage of the number of specified cores.

Python and JULIA

To use python you need to use:

```bash
$ module load anaconda
```

Here's a simple python program called "example.py" – it has just one line:

```python
print("Hello, World!")
```

To submit it on the Habanero Cluster, use the submit script "example.sh"

```bash
#!/bin/sh
#
# Simple "Hello World" submit script for Slurm.
#
#SBATCH --account=astro # The account name for the job.
#SBATCH --job-name=HelloWorld # The job name.
#SBATCH --c 1 # The number of cpu cores to use.
#SBATCH --time=1:00 # The time the job will take to run.
#SBATCH --mem-per-cpu=1gb # The memory the job will use per cpu core.

module load anaconda

#Command to execute Python program
python example.py

#End of script
```

If you use "ls" command you should see 2 programs:

```bash
example.sh
example.py
```
To submit it - please use:

```
$ sbatch example.sh
```

To check the output use:

```
$ cat slurm-463023.out
Hello, World!
```

Similarly, here is the "julia_example.jl" with just one line

```
$ cat julia_example.jl
println("hello world")
```

and

```
$ cat julia_example.sh

#!/bin/sh
#
# Simple "Hello World" submit script for Slurm.
#
#SBATCH --account=hblab # The account name for the job.
#SBATCH --job-name=HelloWorld # The job name.
#SBATCH -c 1 # The number of cpu cores to use.
#SBATCH --time=1:00 # The time the job will take to run.
#SBATCH --mem-per-cpu=1gb # The memory the job will use per cpu core.

module load julia

#Command to execute Python program
julia julia_example.jl

#End of script
```

After you finish creating those two files, if you use "ls" command you should see:

```
julia_example.jl
julia_example.sh
```
To submit it use:

```
$ sbatch julia_example.sh
Submitted batch job 463030
```

To check the output

```
$ cat slurm-463030.out
hello world
```

Julia Interactive Session Usage:

Step 1 >> start an interactive session (** use "astro" if you are a member of "astro" group, otherwise use your group name):

```
$ srun --pty -t 0-04:00 -A astro /bin/bash
$ module load julia/0.5.1
$ julia julia_example.jl
hello world

$ julia
  _ _ _(_)_ _ | A fresh approach to technical computing
 ( | | (_) | Documentation: http://docs.julialang.org
 | _ _ _(_|_ | Type "?help" for help.
 | | | | | | / `._ | |
 | | | | | | ( | | | Version 0.5.1 (2017-03-05 13:25 UTC)
| | | | | | | | | Official http://julialang.org/ release
|___/ x86_64-pc-linux-gnu

julia>
```

To quit Julia use "CTRL +D"

Tensorflow

The following describes is how you can import tensorflow on GPU node.

First, run an interactive job requesting a GPU node

```
$ srun --pty -t 0-02:00:00 --gres=gpu:1 -A <group_name> /bin/bash
```

Load modules:
$ module load cuda80/toolkit cuda80/blas cudnn/5.1
$ module load tensorflow-gpu/anaconda2-4.4.0/1.2.1

Start python and test tensorflow:

$ python
Python 2.7.13 |Anaconda custom (64-bit)| (default, Dec 20 2016, 23:09:15)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux2

>>> import tensorflow as tf
>>> hello = tf.constant('Hello, TensorFlow!')
>>> sess = tf.Session()

>>> print(sess.run(hello))
Hello, TensorFlow!

The following describes is how you can import tensorflow on GPU P100 node.

First, run an interactive job requesting a P100 gpu node

$ srun --pty -t 0-02:00:00 --constraint=p100 -A <group_name> /bin/bash

Load modules:

$ module load cuda90/toolkit
$ module load cuda90/blas
$ module load cudnn/7.0.5

****
Default anaconda module comes with python 3.5.2
$ module load anaconda

If you never installed tensorflow-gpu package please use this command once:
$ pip install upgrade tensorflow-gpu --user

Start python and test tensorflow:
$ python

Python 3.5.2 |Anaconda 4.2.0 (64-bit)| (default, Jul 2 2016, 17:53:06)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.

>>> import tensorflow as tf
>>> print(tf.__version__)
1.8.0
>>> hello = tf.constant('Hello, TensorFlow!')

>>> sess = tf.Session()
totalMemory: 15.89GiB freeMemory: 14.61GiB
name: Tesla P100-PCIE-16GB major: 6 minor: 0 memoryClockRate(GHz): 1.3285

>>> print(sess.run(hello).decode())
Hello, TensorFlow!

Jupyter Notebooks
This is one way to set up and run a jupyter notebook on Habanero. As your notebook will listen on a port that will be accessible to anyone logged in on a submit node you should first create a password.

Creating a Password
The following steps can be run on the submit node or in an interactive job.
1. Load the anaconda python module.

   $ module load anaconda

2. If you haven’t already done so, initialize your jupyter environment.
3. Start a python or ipython session.

$ ipython

4. Run the password hash generator. You will be prompted for a password, prompted again to verify, and then a hash of that password will be displayed.

In [1]: from notebook.auth import passwd; passwd()
Enter password:
Verify password:
Out[1]: 'sha1:60bdb1:306fe0101ca73be2429edbab0935c545'

5. Cut and paste the hash into ~/.jupyter/jupyter_notebook_config.py

(Important: the following line in the file is commented out by default so please uncomment it first)

c.NotebookApp.password = 'sha1:60bdb1:306fe0101ca73be2429edbab0935c545'

Setting the password will prevent other users from having access to your notebook and potentially causing confusion.

**Running a Jupyter Notebook**

1. Log in to the submit node. Start an interactive job.

$ srun --pty -t 0-01:00 -A <ACCOUNT> /bin/bash

2. Get rid of XDG_RUNTIME_DIR environment variable

$ unset XDG_RUNTIME_DIR

3. Load the anaconda environment module.

$ module load anaconda

4. Look up the IP of the node your interactive job is running on.
5. Start the jupyter notebook, specifying the node IP.

   $ jupyter notebook --no-browser --ip=10.43.4.206

6. Look for the following line in the startup output to get the port number.

   The Jupyter Notebook is running at: http://10.43.4.206:8888/

7. From your local system, open a second connection to Habanero that forwards a local port to the remote node and port. Replace UNI below with your uni.

   $ ssh -L 8080:10.43.4.206:8888 UNI@habanero.rcs.columbia.edu

8. Open a browser session on your desktop and enter the URL ‘localhost:8080’ (i.e. the string within the single quotes) into its search field. You should now see the notebook.

**Spark**

Spark is a fast and general-purpose cluster computing framework for large-scale data processing. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports cyclic data flow and in-memory computing.

For a short overview of how Spark runs on clusters, refer to the Spark Cluster Mode Overview.

The spark-slurm script launches Spark in standalone cluster mode and is integrated with the cluster scheduler to automatically set up a spark mini-cluster using the nodes allocated to your job.

To use the script, you must first launch a job that allocates at least 2 nodes.

The script performs the following steps:

- Launches a spark master process.
- Launches a spark worker process on each allocated node, pointing each one to the master process.
- Sets various default environment variables some of which can be overridden.

The spark-slurm script on Habanero is slightly modified version of the github spark-slurm script.

**Set your environment:**

Before running spark-slurm, JAVA_HOME and the spark environment must be set:
To run spark within an interactive job allocation of 3 nodes (replacing `<account>` with your account):

```
$ salloc -N 3 -A <account> --cpus-per-task 24 --mem=120G
```

Run spark-slurm to launch the spark cluster

```
$ spark-slurm
```

Or, if you’d like to save the console output to a log file:

```
$ spark-slurm > ~/.spark/spark-${SLURM_JOB_ID}.log &
```

To view the log file:

```
$ less ~/.spark/spark-${SLURM_JOB_ID}.log
```

After spark-slurm successfully started a spark cluster, look for the line starting with `starting master:` ... and use that URL for your spark-shell or spark-submit scripts.

So to get the spark master URL from a log file:

```
$ awk '/master:/ {print $NF}' ${SLURM_JOB_ID}.log
spark://10.43.4.220:7077
```

You can then submit spark jobs to using this information.

**Running spark as a non-interactive batch job**

Example submit script spark-submit.sh:
#!/bin/bash
#SBATCH --time=01:00:00
#SBATCH --account=<your_account>
#SBATCH --nodes=3
#SBATCH --mem=120G
#SBATCH --cpus-per-task=24
#SBATCH --mail-user=<your_email>
#SBATCH --mail-type=ALL
module load spark
export JAVA_HOME=/usr
SPARK_LOG=~/.spark/spark-${SLURM_JOB_ID}.log
spark-slurm > $SPARK_LOG &
sleep 20
sparkmaster=`awk '/master:/ {print $NF}' $SPARK_LOG`
echo sparkmaster="$sparkmaster"
spark-submit --master $sparkmaster
$SPARK_HOME/examples/src/main/python/wordcount.py $SPARK_HOME/README.md

Submit

$ sbatch spark-submit.sh

The console/log will also indicate the master WebUI port. To determine which port it was started on:

$ grep UI spark-9463479.log | grep Success
2018-10-20 09:42:53 INFO  Utils:54 - Successfully started service 'MasterUI' on port 8082.

To connect to the spark master WebUI, you can launch google-chrome from a login node. You will need to use Xwindows. If using Windows operating system, install and run Xming and then use Putty and enable SSH Xwindows forward before connecting.

Run chrome in the Xwindows session.

$ google-chrome &

This should bring up a new browser window which is running on the login node. This is necessary since you cannot directly connect to the compute node's internal network from your personal computer. In that browser, load the URL for the master WebUI, for example:

Example URL only, replace with actual master node and actual port as shown in log file:

node220:8082
Use `spark-submit` to submit spark programs to the spark cluster.

You can then submit spark jobs to using this information.

Submit spark wordcount program to spark cluster.

```bash
$ sparkmaster=spark://10.43.4.220:7077
$ spark-submit --master ${sparkmaster}
$SPARK_HOME/examples/src/main/python/wordcount.py $SPARK_HOME/README.md

View help
$ spark -h

If needed, you may optionally set the number of spark executor cores or executor memory available by supplying flags to spark-submit. Here's a submit spark program example, specifying total executor cores and memory:

```bash
$ spark-submit --total-executor-cores 48 --executor-memory 5G --master
${sparkmaster} $SPARK_HOME/examples/src/main/python/wordcount.py
$SPARK_HOME/README.md
```

When you exit the job allocation or when your job ends, your spark master and slave processes will be killed.