Ginsburg - Job Examples

- Hello World
- Running Precompiled Binaries
- C, C++, Fortran MPI
- GPU (CUDA C/C++)
- Singularity
- Installing R Packages on Ginsburg
- Local Installation
- Matlab
- Python and JULIA
- Tensorflow
- NetCDF
- Jupyter Notebooks

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.
#
#SBATCH --account=ACCOUNT        # Replace ACCOUNT with your group account name
#SBATCH --job-name=HelloWorld    # The job name
#SBATCH -c 1                     # The number of cpu cores to use (up to 32 cores per server)
#SBATCH --time=0-0:30            # The time the job will take to run in D-HH:MM
#SBATCH --mem-per-cpu=5G         # The memory the job will use per cpu core

echo "Hello World"
sleep 10
date

# End of script
```

### Running Precompiled Binaries

To submit a precompiled binary to run on Ginsburg, 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

Ginsburg supports Intel Parallel Studio which is a highly optimized compiler that builds software with the highest performance. It also supports MPI for applications that require communication between multiple nodes. All the 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/2020
mpiexec -bootstrap slurm ./myprogram
```

In order to take advantage of Ginsburg architecture, your program should be (re)compiled on the cluster even if you used Intel for compiling it on another cluster. 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:
For programs written in C, use mpiicc in order to compile them:

```bash
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`:

```bash
#!/bin/sh

#SBATCH -A ACCOUNT               # Replace ACCOUNT with your group account name
#SBATCH -N 2                     # Number of nodes
#SBATCH --mem-per-cpu=5800       # Default is 5800
#SBATCH --time=0-0:30            # Runtime in D-HH:MM
#SBATCH --ntasks-per-node=32      # Max 32 since Ginsburg has 32 cores per node

module load intel-parallel-studio/2020

mpiexec -bootstrap slurm ./pi_mpi

# End of script
```

Job Submission

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

**OpenMPI**

Ginsburg supports also OpenMPI from the GNU family.

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

```bash
#!/bin/sh

#SBATCH -A ACCOUNT               # Replace ACCOUNT with your account name
#SBATCH -N 2                     # Number of nodes
#SBATCH --mem-per-cpu=5800       # Default is 5800
#SBATCH --time=0-0:30            # Runtime in D-HH:MM
#SBATCH --ntasks-per-node=32      # Max 32 since Ginsburg has 32 cores per node

module load openmpi/gcc/64

mpiexec myprogram

# End of script
```

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
/cm/shared/apps/openmpi/gcc/64/1.10.7/bin/mpicc
```

Compile your program using mpicc. For programs written in C:
GPU (CUDA C/C++)

The cluster includes 18 Nvidia RTX 8000 nodes and 4 Nvidia V100S GPU nodes each with 2 GPU modules per server.

To use a GPU server you must specify the \texttt{--gres=gpu} option in your submit request, followed by a colon and the number of GPU modules you require (with a maximum of 2 per server).

Request a gpu, specify this in your submit script. If the colon and number are omitted, as shown below, the scheduler will request 1 GPU module.

\begin{verbatim}
#SBATCH --gres=gpu
\end{verbatim}

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 \texttt{<ACCOUNT>} with your account.

\begin{verbatim}
$ srun --pty -t 0-01:00 --gres=gpu:1 -A <ACCOUNT> /bin/bash
\end{verbatim}

Load the cuda environment module which will add cuda to your PATH and set related environment variables.

\begin{verbatim}
$ module load cuda11.1/toolkit
\end{verbatim}

You then may need to compile your program using \texttt{nvcc} if you are compiling cuda code directly.

\begin{verbatim}
$ nvcc -o <EXECUTABLE_NAME> <FILE_NAME.cu
\end{verbatim}

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

\begin{verbatim}
$ nvcc -o hello_world hello_world.cu
\end{verbatim}

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:

\begin{verbatim}
#!/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 2 gpus per GPU node)
#SBATCH --constraint=rtx8000     # You may specify rtx8000 or v100s or omit this line for either
#SBATCH --c 1                     # The number of cpu cores to use.
#SBATCH --time=0-01:00           # The time the job will take to run in D-HH:MM
#SBATCH --mem-per-cpu=5gb        # The memory the job will use per cpu core.

module load cudall.1/toolkit
./hello_world

# End of script
\end{verbatim}

\textbf{Job submission}

\begin{verbatim}
$ sbatch gpu.sh
\end{verbatim}

This program will print out "Hello World!" when run on a gpu server or print "Hello Hello" when no gpu module is found.
Ocean Climate Physics OCP GPU Partition (*For OCP members only*)

Members of OCP have access to a separate GPU partition which accesses only OCP gpu nodes. This directs jobs to first request the 4 GPU servers that OCP owns and guarantees priority access as well as allowing running up to 5 day jobs on those gpu nodes. If no OCP GPU servers are available, the scheduler will fall back to request non-OCP gpu nodes across the cluster.

To submit to this gpu partition, ocp members must specify the partition explicitly in their submit scripts as shown below.

```
#SBATCH --partition=ocp_gpu       # Request ocp_gpu nodes first. If none are available, the scheduler will
request non-OCP gpu nodes.
#SBATCH --gres=gpu:1             # Request 1 gpu (Up to 2 gpus per GPU node)
#SBATCH --constraint=rtx8000     # You may specify rtx8000 or v100s or omit this line for either
```

If `--partition=ocp_gpu` is omitted, the scheduler will request any gpu across the cluster by default.

Singularity

Singularity is a software tool that brings Docker-like containers and reproducibility to scientific computing and HPC. Singularity has Docker container support and enables users to easily run different flavors of Linux with different software stacks. These containers provide a single universal on-ramp from the laptop, to HPC, to cloud.

Users can run Singularity containers just as they run any other program on our HPC clusters. Example usage of Singularity is listed below. For additional details on how to use Singularity, please contact us or refer to the Singularity User Guide.

Downloading Pre-Built Containers

Singularity makes it easy to quickly deploy and use software stacks or new versions of software. Since Singularity has Docker support, users can simply pull existing Docker images from Docker Hub or download docker images directly from software repositories that increasingly support the Docker format. Singularity Container Library also provides a number of additional containers.

You can use the `pull` command to download pre-built images from an external resource into your current working directory. The `docker://` uri reference can be used to pull Docker images. Pulled Docker images will be automatically converted to the Singularity container format.

This example pulls the default Ubuntu docker image from docker hub.

```
$ singularity pull docker://ubuntu
```

Running Singularity Containers

Here’s an example of pulling the latest stable release of the Tensorflow Docker image and running it with Singularity. (Note: these pre-built versions may not be optimized for use with our CPUs.)

First, load the Singularity software into your environment with:

```
$ module load singularity
```

Then pull the docker image. This also converts the downloaded docker image to Singularity format and save it in your current working directory:

```
$ singularity pull tensorflow.sif docker://tensorflow/tensorflow
```

Done. Container is at: ./tensorflow.sif

Once you have download a container, you can run it interactively in a shell or in batch mode.

Singularity - Interactive Shell
The **shell** command allows you to spawn a new shell within your container and interact with it as though it were a small virtual machine:

```sh
$ singularity shell tensorflow.sif
Singularity: Invoking an interactive shell within container...
```

From within the Singularity shell, you will see the Singularity prompt and can run the downloaded software. In this example, python is launched and tensorflow is loaded.

```
Singularity> python
>>> import tensorflow as tf
>>> print(tf.__version__)
2.4.1
>>> exit()
```

When done, you may exit the Singularity interactive shell with the "exit" command.

```
Singularity> exit
```

### Singularity: Executing Commands

The **exec** command allows you to execute a custom command within a container by specifying the image file. This is the way to invoke commands in your job submission script.

```
$ module load singularity
$ singularity exec tensorflow.sif [command]
```

For example, to run python example above using the exec command:

```
$ singularity exec tensorflow.sif python -c 'import tensorflow as tf; print(tf.__version__)'
```

### Singularity: Running a Batch Job

Below is an example of job submission script named **submit.sh** that runs Singularity. Note that you may need to specify the full path to the Singularity image you wish to run.

```bash
#!/bin/bash
# Singularity example submit script for Slurm.
#
# Replace <ACCOUNT> with your account name before submitting.
#
#SBATCH -A <ACCOUNT>           # Set Account name
#SBATCH --job-name=tensorflow  # The job name
#SBATCH -c 1                   # Number of cores
#SBATCH -t 0-0:30              # Runtime in D-HH:MM
#SBATCH --mem-per-cpu=5gb      # Memory per cpu core

module load singularity
singularity exec tensorflow.sif python -c 'import tensorflow as tf; print(tf.__version__)'
```

Then submit the job to the scheduler. This example prints out the tensorflow version.
$ sbatch submit.sh

For additional details on how to use Singularity, please contact us or refer to the Singularity User Guide.

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
#
#SBATCH -A ACCOUNT               # Replace ACCOUNT with your group account name
#SBATCH -J DeltaHedge            # The job name
#SBATCH -c 4                     # The number of cpu cores to use. Max 32.
#SBATCH -t 0-0:30                # Runtime in D-HH:MM
#SBATCH --mem-per-cpu 5gb        # 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 Ginsburg

HPC users can Install R packages locally in their home directory or group's scratch space (see below).

Local Installation

After logging in to Ginsburg, start R:

$ module load R

$ R

You can see the default library paths (where R looks for packages) by calling .libPaths():

> .libPaths()
[1] "/burg/opt/r-4.0.4/lib64/R/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 `/burg/<GROUP>/users/<UNI>/`.

```bash
$ mkdir /burg/<GROUP>/users/<UNI>/rpackages
```

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

```r
$ R
> .libPaths("/burg/<GROUP>/users/<UNI>/rpackages/")
```

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

```r
> .libPaths()
[1] "/burg/rcs/users/UNI/rpackages"    "/burg/opt/r-4.0.4/lib64/R/library"
```

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

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

Select appropriate mirror and follow install instructions.

Test to see if package can be called:

```r
> library(sm)
Package `sm`, version 2.2-3; Copyright (C) 1997, 2000, 2005, 2007 A.W.Bowman & A.Azzalini

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:

```r
.libPaths("/burg/<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.
#!/bin/sh
#
# Simple Matlab submit script for Slurm.
#
#SBATCH -A ACCOUNT               # Replace ACCOUNT with your group account name
#SBATCH -J SimpleMLJob           # The job name
#SBATCH -c 1                     # Number of cores to use (max 32)
#SBATCH -t 0-0:30                # Runtime in D-HH:MM
#SBATCH --mem-per-cpu=5G         # The memory the job will use per cpu core

module load matlab

echo "Launching a 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 submit script (simpoiss.sh) could contain the following line:

#SBATCH -c 16

The -c flag determines the number of cores (up to 32 are allowed).

For explicit multi-threading, the users must include the following corresponding statement within their Matlab program:

parpool('local', 16)

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 Ginsburg, 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:

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

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

Save as example.py.

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

```bash
#!/bin/sh
#
# Simple "Hello World" submit script for Slurm.
#
#SBATCH --account=ACCOUNT       # Replace ACCOUNT with your group account name
#SBATCH --job-name=HelloWorld    # The job name.
#SBATCH -c 1                     # The number of cpu cores to use
#SBATCH -t 0-0:30                # Runtime in D-HH:MM
#SBATCH --mem-per-cpu=5gb        # 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:

example.sh
example.py

To submit it - please use:

```bash
$ sbatch example.sh
```

To check the output use:

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

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

```julia
println("hello world")
```

and

```bash
$ cat julia_example.sh
```
#!/bin/sh
#
# Simple "Hello World" submit script for Slurm.
#
#SBATCH --account=ACCOUNT             # Replace ACCOUNT with your group account name
#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=5gb             # 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 (*** replace ACCOUNT with your slurm group account name below):

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

$ julia
  _   _  (___)   | A fresh approach to technical computing
 () | () (___)   | Documentation: http://docs.julialang.org\nbsp;
   |   |   |   |   | Type "help" for help.
   |   |   |   |   |
   (_) | (_)| (_)| Official http://julialang.org/ release
  /   | /   |   | x86_64-pc-linux-gnu

julia>
To quit Julia use "CTRL +D"

Julia packages can be installed with this command (for example "DataFrames" package):

julia> using Pkg
julia> Pkg.add("DataFrames")
Please check this website: https://julialang.org/packages/ to see the full list of the official packages available.

Tensorflow

Tensorflow computations can use CPUs or GPUs. The default is to use CPUs which are more prevalent, but typically slower than GPUs.

Anaconda Python makes it easy to install Tensorflow, enabling your data science, machine learning, and artificial intelligence workflows. https://docs.anaconda.com/anaconda/user-guide/tasks/tensorflow/

**Tensorflow**

First, load the anaconda python module.

```
$ module load anaconda
```

You may need to run "conda init bash" to initialize your conda shell.

```
$ conda init bash
```

*** For changes to take effect, close and re-open your current shell. ***

To install the current release of CPU-only TensorFlow:

```
$ conda create -n tf tensorflow
$ conda activate tf
```

Or, to install the current release of GPU TensorFlow:

```
$ conda create -n tf-gpu tensorflow-gpu
$ conda activate tf-gpu
```

Test tensorflow

```
$ python
Python 3.7.1 (default, Dec 14 2018, 19:28:38)
>>> import tensorflow as tf
>>> print(tf.__version__)
1.13.1
```

Test tensorflow gpu support (you must be on a GPU)

```
$ python
>>> import tensorflow as tf
>>> print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU')))
```

NetCDF

NetCDF (Network Common Data Form) is an interface for array-oriented data access and a library that provides an implementation of the interface. The NetCDF library also defines a machine-independent format for representing scientific data. Together, the interface, library, and format support the creation, access, and sharing of scientific data.

To load the NetCDF Fortran Intel module:

```
$ module load netcdf-fortran-intel/4.5.3
```

To see all available NetCDF modules run:

```
$ module avail netcdf
```

<table>
<thead>
<tr>
<th>netcdf-fortran-intel/4.5.3</th>
<th>netcdf/gcc/64/gcc/64/4.7.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>netcdf-fortran/4.5.3</td>
<td>netcdf/gcc/64/gcc/64/4.7.4 (D)</td>
</tr>
</tbody>
</table>
Jupyter Notebooks
This is one way to set up and run a jupyter notebook on Ginsburg. 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.

   $ jupyter notebook --generate-config

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

   Please note that the example above specifies time limit of one hour only. That can be set to a much higher value, and in fact the default (i.e. if not specified at all) is as long as 5 days.

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.
$ hostname -i
10.43.4.206

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 Ginsburg 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@burg.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.