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.

```
#!/bin/sh
#
# Simple "Hello World" submit script for Slurm.
#
# Replace ACCOUNT with your account name before submitting.
#
#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:

```
$ 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               # 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

```
$ sbatch pi_mpi.sh
```

OpenMPI

Ginsburg supports also OpenMPI from the GNU family.

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

```
#!/bin/sh
#SBATCH -A ACCOUNT               # Replace ACCOUNT with your account name
#SBATCH -N 2
#SBATCH --ntasks-per-node=32     # Max 32 since Ginsburg has 32 cores per node
#SBATCH --time=0:0:30            # Runtime in D-HH:MM

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.

```
$ 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 --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.

```
#SBATCH --gres=gpu
```

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.

```
$ module load cuda11.1/toolkit
```

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

```
$ 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 2 gpus per GPU node)
#SBATCH --constraint=rtx8000 # You may specify rtx8000 or v100s or omit this line for either
#SBATCH --c1 # 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 cuda11.1/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.
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:

$ 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.
Using GeoChemFoam in a Singularity container

GeoChemFoam is open source code, based on the OpenFoam CFD toolbox developed at the Institute of GeoEnergy Engineering, Heriot-Watt University.

After converting one of the available Docker containers to Singularity 3.x .sif, some additional steps/tweaks are needed to get all of the features working. For this tutorial we assume GeoChemFoam version 5.0. and use the Test Case 01 Species transport in a Ketton Micro-CT image tutorial. You can choose your version of Anaconda Python. Note there are some changes in the tutorial from earlier versions of GeoChemFoam, e.g., 4.8. For this tutorial we assume the name of the Singularity container is geochemfoam-5.0.sif and we'll use an interactive job with srun, to request 8 --ntasks for use with OpenMPI in the Ketton tutorial.

```
srun --pty -t 0-08:00 --ntasks 8 --mem=10gb -X -A <your-account> /bin/bash
module load singularity/3.7.1 anaconda/3-2022.05 openmpi/gcc/64/4.1.5a1
```

Singularity offers a few ways to set/pass environment variables, we'll use SINGULARITYENV_PREFEND_PATH as the container needs to know where the srun command is.

```
extport SINGULARITYENV_PREFEND_PATH=$PATH
singularity shell --bind /path/to/your-directory/runs:/home/gcfoam/works/GeoChemFoam-5.0/runs:rw geochemfoam-5.0.sif
```

--bind connects whatever is to the left of the colon, in this case, a new directory called 'runs'. The right side of the colon is a path that exists in the container. rw is read/write

All files written to the /runs directory will remain and that's the only directory that files can be written to when the container is bound.

Change the value of $HOME:

```
export HOME=/home/gcfoam
```

Copy the multiSpeciesTransportFoam directory from /home/gcfoam/works/GeoChemFoam-5.0/tutorials/transport to /path/to/your-directory/runs/, e.g.,

```
cp -a /home/gcfoam/works/GeoChemFoam-5.0/tutorials/transport /path/to/your-directory/runs/
```

Install 5 Python libraries using Python from within the container. Specify where to install these (-t option as follows is one way).

```
/usr/bin/pip3 install -t /home/gcfoam/works/GeoChemFoam-5.0/runs matplotlib numpy scikit-image numpy-stl h5py
```

Note this will populate the ~/runs directory where the results from the tutorial are so you may want to put these in a different directory.

Set 2 additional environment variables:

```
export PYTHONPATH=/home/gcfoam/works/GeoChemFoam-5.0/runs
export MPLCONFIGDIR=/home/gcfoam/works/GeoChemFoam-5.0/runs
```

Source the .bashrc file in the container:

```
source $HOME/works/GeoChemFoam-5.0/etc/bashrc
```

Run the scripts in the tutorial in your ~/runs directory.

```
cd transport/multiSpeciesTransportFoam/Ketton/
./createMesh.sh
```

To avoid seeing a couple of warnings from OpenMPI, you can add the following options to the mpirun command in the scripts (using vi) that use them: -mca btl "^openib" --mca orte_base_helpAggregate 0

For the next script, runSnappyHexMesh.sh, that would change as follows:

```
mpirun --np $NP --mca btl "^openib" --mca orte_base_helpAggregate 0 snappyHexMesh --overwrite --parallel > snappyHexMesh.out
./runSnappyHexMesh.sh
./initCaseFlow.sh
```

The next two scripts have mpirun, which you can add --mca btl "^openib" --mca orte_base_helpAggregate 0

```
./runCaseFlow.sh
```

Check the simpleFoamFlow.out file for 'SIMPLE solution converged in <X> iterations'

```
./processFlow.sh
```

```
cat poroPerm.csv
```

```
time poro perm Lpore Re UD
0 0.133986 5.71129e-12 1.84664e-05 0.0220027 0.00239467
```
The next two scripts contain mpirun:

```
./runCaseTransport.sh
./processTransport.sh
```

```
cat A_Conc.csv B_Conc.csv
time C
0 0
0.01 0.274137
0.02 0.432351
0.03 0.554637
0.04 0.652788
0.05 0.730438
0.06 0.791012
0.07 0.838068
0.08 0.87449
0.09 0.90264
0.1 0.924441
time C
0 0
0.01 0.126229
0.02 0.239035
0.03 0.345767
0.04 0.443858
0.05 0.533867
0.06 0.613993
0.07 0.681486
0.08 0.737184
0.09 0.782178
0.1 0.818111
```

To run the tutorial again make sure to run one of the delete scripts, e.g., deleteAll.sh

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

Using Couenne in a Singularity container

Couenne (Convex Over and Under ENvelopes for Nonlinear Estimation) is a branch&bound algorithm to solve Mixed-Integer Nonlinear Programming (MINLP) problems. It includes a suite of programs with several dependencies. Fortunately there is a Docker container which can be used to access these programs, e.g., bonmin, couenne, Ipopt, Cgl, and Cbc, via Singularity. You can use these sample .nl files to test with Couenne.
singularity pull docker://coinor/coin-or-optimization-suite
singularity shell coin-or-optimization-suite_latest.sif
Couenne 0.5 -- an Open-Source solver for Mixed Integer Nonlinear Optimization
Mailing list: couenne@list.coin-or.org
Instructions: http://www.coin-or.org/Couenne

```
NLP0012I   Num  Status  Obj   It   time  Location
NLP0014I   1    OPT  306.49998 22  0.004007
Couenne: new cutoff value 3.0649997900e+02 (0.009883 seconds)
Loaded instance "hs015.nl"
Constraints: 2
Variables: 2 (0 integer)
Auxiliaries: 8 (0 integer)
```

Coin0506I Presolve 29 (-1) rows, 9 (-1) columns and 64 (-2) elements
Clp0006I 0 Obj 0.25 Primal inf 473.75936 (14)
Clp0006I 13 Obj 0.31728151
Clp0032I Optimal objective 0.31728151 - 13 iterations time 0.002, Presolve 0.00
Clp0000I Optimal - objective value 0.31728151
Cbc0012I Integer solution of 306.49998 found by Couenne Rounding NLP after 0 iterations and 0 nodes (0.00 seconds)
NLP Heuristic: NLP0014I 2 OPT 306.49998 5 0.001228
solution found, obj. 306.5
Clp0000I Optimal - objective value 0.31728151
Optimality Based BT: 3 improved bounds
Probing: 2 improved bounds
Cbc0031I 1 added rows had average density of 2
Cbc0013I At root node, 4 cuts changed objective from 0.31728151 to 306.49998 in 1 passes
Cbc0014I Cut generator 0 (Couenne convexifier cuts) - 4 row cuts average 2.0 elements, 3 column cuts (3 active)
Cbc0011I Search completed - best objective 306.4999790004336, took 0 iterations and 0 nodes (0.00 seconds)
Cbc0035I Maximum depth 0, 0 variables fixed on reduced cost

couenne: Optimal

"Finished"

Linearization cuts added at root node: 30
Linearization cuts added in total: 30 (separation time: 2.4e-05s)
Total solve time: 0.003242s (0.003242s in branch-and-bound)
Lower bound: 306.5 (gap: 0.00%)  
Branch-and-bound nodes: 0
Performance of FBBT: 2.75 2ubd: 0.5 infeas: 0 runs. fix: 0 shrnk: 0.00103838
Performance of ORBBT: 0.000742s, 1 runs. fix: 0 shrnk: 6.70203

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:
### 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).

### Running LAMMPS in a Singularity Container with a GPU

Nvidia provides prebuilt Singularity LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) GPU-enabled containers in its New General Catalogue (NGC). Using the tutorial provided you can use:

**Catalogue**

$ docker://nvcr.io/hpc/lammps:29Sep2021

Here is a modified tutorial for Ginsburg, starting with an interactive session requesting one GPU, 10 GB of memory and note the extra option for the wget commands:

```bash
srun --pty --mem=10gb -t 0:2:00 --gres=gpu:1 -A <your-account> /bin/bash
module load singularity/3.7.1
wget https://lammps.sandia.gov/inputs/in.lj.txt --no-check-certificate
export BENCHMARK_DIR=$PWD
cd $BENCHMARK_DIR
chmod +x run_lammps.sh
cd $BENCHMARK_DIR
singularity run --nv -B $PWD:/host_pwd --pwd /host_pwd docker://nvcr.io/hpc/lammps:29Sep2021 ./run_lammps.sh
```

You should start seeing this response:

```
Running Lennard Jones 8x4x8 example on 1 GPUS...
LAMMPS (29 Sep 2021)
KOKKOS mode is enabled (src/KOKKOS/kokkos.cpp:97)
will use up to 1 GPU(s) per node
using 1 OpenMP thread(s) per MPI task
Lattice spacing in x,y,z = 1.6795962 1.6795962 1.6795962
```

You can combine the commands for use with directly with a Slurm interactive session as suggested in the original tutorial

```bash
srun --pty --mem=10gb -t 0:4:00 --gres=gpu:1 -A <your-account> -v --mpi=pmi2 -c 4 singularity run --nv -B $PWD:/host_pwd --pwd /host_pwd docker://nvcr.io/hpc/lammps:29Sep2021 ./run_lammps.sh
```

The same container can be used for non-GPU jobs but still requires requesting a GPU node. You can test with a tutorial that provides `polymer_plus_bridge.s.lam`. Download the .lam and initial configuration files. With srun, -v will show more verbose output.
wget https://cbrackley.github.io/simple_lammps_tutorial/lammps_tutorial/tutorial5/polymer_plus_bridges.lam
wget https://cbrackley.github.io/simple_lammps_tutorial/lammps_tutorial/tutorial5/initial_configuration.txt
srun --pty --mem=10gb -t 0-4:00 --gres=gpu:1 -A <your-account> --v --mpi=pmi2 singularity exec --nv -B /host_pwd --pwd /host_pwd docker://nvcr.io/hpc/lammps:29Sep2021 mpirun lmp -in polymer_plus_bridges.lam

Reading data file ...
 orthogonal box = (-25.000000 -25.000000 -25.000000) to (25.000000 25.000000 25.000000)
 1 by 1 by 1 MPI processor grid
 reading atoms ...
 220 atoms
 reading velocities ...
 220 velocities
 scanning bonds ...
 1 = max bonds/atom
 scanning angles ...
 1 = max angles/atom
 reading bonds ...
 199 bonds
 reading angles ...
 198 angles
 Finding 1-2 1-3 1-4 neighbors ...

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:

```bash
$ module load R
$ R
```

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

```r
> .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
> .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:
install.packages("sm", lib="/burg/<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.Azzalini
type
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("/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).

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 (see above, 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() has been deprecated by Mathworks and been replaced by parpool.

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.

**Matlab with Parallel Computing Toolbox**

Parallel Computing Toolbox (PCT™, formerly known as Distributed Computing Engine), is installed within Matlab. It lets you solve computationally and data-intensive problems using multicore processors, but limited to one compute/execute node. The compute nodes on Ginsburg have 32 CPUs, over 2 sockets, with 16 cores per socket. In order to use PCT you will have to incorporate Matlab functions such as distributed or parfor. Note in order to use all 32 CPUs the best practice is to use the --exclusive option to srun in order to assure no other users are running jobs on the requested node and it may take a while to get a free node, depending on how many users are running jobs. As in the previous section, the -c option option can be used to specify the number of cores, with 16 being the maximum.

- To configure PCT, from the Home tab, under Environment, select the down arrow next to Parallel and select Create and manage a cluster.
- This opens up Cluster Profile Manager. Click Add cluster profile and choose Local (use the cores on your machine).
- Click Set as default.
- Click Edit, which is within the Manage Profile section.
- Add a Description
- Set Number of workers to start on your local machine, with the maximum being 16.
- The rest of the fields can be left to their default value, unless you want to change them.
- Click Validate, it can take a minute to complete.

Once validation completes you can use the example for Use Distributed Arrays to Solve Systems of Linear Equations with Direct Methods, you can paste the following into Matlab's Editor and click Run. A graph labeled System of Linear Equations with Full Matrices will pop up. In the bottom left corner under ondemand (Folder) you will see Starting parallel pool and in the Command Window you will see Connected to the parallel pool (number of workers: ##). ## is the number of cores you configured earlier. Assuming you opened Matlab in the background, you can also run the top -i command to see the number of CPUs being used during the calculation.
Matlab with Parallel Server

Matlab 2020b and 2022b on Ginsburg now have access to Parallel Server, and the toolbox is installed. The first time you run Matlab, it can take a few minutes to fully open, especially over WiFi. In order to use Parallel Server, a Cluster Profile needs to be created to use the Slurm job scheduler. You will need to request the number of nodes desired as well and may need to increase the amount of memory desired. With an interactive job requesting two nodes start with:

```bash
srun --pty -t 0-04:00  --nodes=2 --mem=10gb -A <your-account> /bin/bash
```

**Step One**

Using the Configure for Slurm MathWorks tutorial as a guide:

2. Create a new profile in the Cluster Profile Manager by selecting Add Cluster Profile > Slurm.
3. With the new profile selected in the list, click Rename and edit the profile name something informative for future use, e.g., InstallTest. Press Enter.
4. In the Properties tab, provide settings for the following fields:
   a. Set the Description field to something informative, e.g., For testing installation.
   b. Set the JobStorageLocation to the location where you want job and task data to be stored, e.g., /burg/home/<your-directory>.
   i. Note: JobStorageLocation should not be shared by parallel computing products running different versions; each version on your cluster should have its own JobStorageLocation.
5. Set the NumWorkers field to the number of workers you want to run the validation tests on. This should be not be more than what is specified by -nodes in the interactive job request, i.e., srun.
6. Set the ClusterMatlabRoot to the installation location of the MATLAB version, i.e., /burg/opt/matlab/R2020b or /burg/opt/matlab/R2022b.
7. Within ADDITIONAL SLURM PROPERTIES add -A <your account-name> (replace <your account-name> accordingly).
8. Click Done to save your cluster profile.

**Step Two**

In this step you verify your cluster profile, and thereby your installation. You can specify the number of workers to use when validating your profile. If you do not specify the number of workers in the Validation tab, then the validation will attempt to use as many workers as the value specified by the NumWorkers property on the Properties tab. You can specify a smaller number of workers to validate your configuration without occupying the whole cluster.

1. If it is not already open, start the Cluster Profile Manager from the MATLAB desktop. On the Home tab, in the Environment area, select Parallel > Create and Manage Clusters.
2. Select your cluster profile in the listing.
3. Click Validation tab.
4. Use the checkboxes to choose all tests, or a subset of the validation stages, and specify the number of workers to use when validating your profile.
5. Click Validate. Note when the Parallel pool test (parpool) starts running, the screen flips back to Matlab, and in the very bottom left status bar, you will see Starting Parallel Pool on the profile name you created in Step 1.
6. The Validation Results tab shows the output as shown in the MathWorks tutorial.

```matlab
n = 1e3;
A = randi(100,n,n);
ADist = distributed(A);
b = sum(A,2);
bDist = sum(ADist,2);
xEx = ones(n,1);
xDistEx = ones(n,1,'distributed');
x = A;
err = abs(xEx-x);
xDist = ADistDist;
errDist = abs(xDistEx-xDist);

figure
subplot(2,1,1)
semilogy(err,'o');
title('System of Linear Equations with Full Matrices');
ylabel('Absolute Error');
xlim([10e-17,10e-13])
subplot(2,1,2)
semilogy(errDist,'o');
title('System of Linear Equations with Distributed Full Matrices');
ylabel('Absolute Error');
xlim([10e-17,10e-13])
```
7. If your validation passed, you now have a valid profile that you can use in other parallel applications. You can make any modifications to your profile appropriate for your applications, such as NumWorkersRange, AttachedFiles, AdditionalPaths, etc.

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:

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

Save as example.py.

To submit it on the Ginsburg Cluster, use the submit script "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 -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:

```
$ 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=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 "Is"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
_ _ _(_)_ | 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

```bash
netcdf-fortran-intel/4.5.3  netcdf/gcc/64/gcc/64/4.7.3
netcdf-fortran/4.5.3        netcdf/gcc/64/gcc/64/4.7.4 (D)
```
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.

Running FSL from a Mac with multiple monitors

Running FSL on a Mac with multiple monitors causes the application to open too large to view. One of the developers/maintainers of FSL provided RCS with a PKL file ("A PKL file is a serialized object created with the pickle module in Python 2.x. It contains binary strings representing an object used in a Python project"). A copy is located in /burg/rcs/projects/files/config.pkl. In order for it to work you need to put it in your $HOME/.config/fsleyes/ directory. This has been preset to open at a reasonable size on the leftmost monitor.