MODI Userguide

Rasmus Munk

January 2022

Contents

1 MODI 1

2 Architecture 2

3 Node Specifications 2

4 Special Directories 4

4.1 erda_mount 4

4.2 modi_images 4

4.3 modi_mount 5

5 SLURM Specifications and Job Runtimes 5

6 Getting Started with SLURM 6

6.1 What is running? 7

6.2 Selecting a Partition 9

7 Singularity Images 10

8 Installing Custom Packages 11

9 Examples 13

9.1 SLURM Hello World Job 14

9.2 SLURM Job with Singularity 16

9.3 MPI SLURM Job via Singularity 17

9.4 Test Image Dependencies Support 19

10 Further Support 21

1 MODI

MPI Oriented Development and Investigation (MODI) is a SLURM based cluster of 8 compute nodes that can be used to submit batch jobs and scale an individual application to a parallel and distributed execution via OpenMPI. MODI is designated to be a small HPC sandbox, this means that it is not designed to be running jobs with extensive runtimes, e.g. several weeks. Instead it is meant as a staging ground for testing, benchmarking, and developing HPC like applications. To enforce this, the system is configured to limit the allowed runtime that a job can execute for, see Section 5 for specifics on this. In summary, it means that on MODI your jobs are subjugated to a maximum runtime and an associated priority for each SLURM queue partition. This has the effect that a job that requires several days of computation is a low priority job, which implies that it could potentially be rescheduled/restarted if it uses resources that a subsequent higher priority job requires.

2 Architecture

An overview of the MODI architecture can be seen in Figure 1. As shown here, the notebook itself interacts with the compute nodes via the standard SLURM commands. An overview over the most common ones can be found at SLURM User Quickstart and SLURM Commands. However, in our setup there are various limitations that restrict certain features. This includes the use of **srun** and **sacct** which currently are not supported. Beyond the basic SLURM usage, the handling of file sharing between the notebook and the compute nodes is handled via the **Shared NFS Storage**. This share is used to share resources between the notebook's file system and the SLURM compute nodes. Further details about this share setup and the special notebook directories will be presented in Section 4.

3 Node Specifications

An overview of the MODI slurm nodes specification can be found in Table 1.

Hostname	CPU	Cores	Memory	Node-Node Bandwidth	Node-Node RDMA
modi000.science	AMD EPYC 7501 @ 2Gz	2 x 32	256GB	25Gbit RoCE	1.2 μs
modi001.science	AMD EPYC 7501 @ 2Gz	2 x 32	256GB	25Gbit RoCE	1.2 μs
modi002.science	AMD EPYC 7501 @ 2Gz	2 x 32	256GB	25Gbit RoCE	1.2 μs
modi003.science	AMD EPYC 7501 @ 2Gz	2 x 32	256GB	25Gbit RoCE	1.2 μs
modi004.science	AMD EPYC 7501 @ 2Gz	2 x 32	256GB	25Gbit RoCE	1.2 μs
modi005.science	AMD EPYC 7501 @ 2Gz	2 x 32	256GB	25Gbit RoCE	1.2 μs
modi006.science	AMD EPYC 7501 @ 2Gz	2 x 32	256GB	25Gbit RoCE	1.2 μs
modi007.science	AMD EPYC 7501 @ 2Gz	2 x 32	256GB	25Gbit RoCE	1.2 μs

Table 1: MODI Nodes Specifications

As illustrated here, all nodes have identical hardware specs. In terms of software, each node is configured with CentOS 7, SLURM 18.08.4, OpenMPI

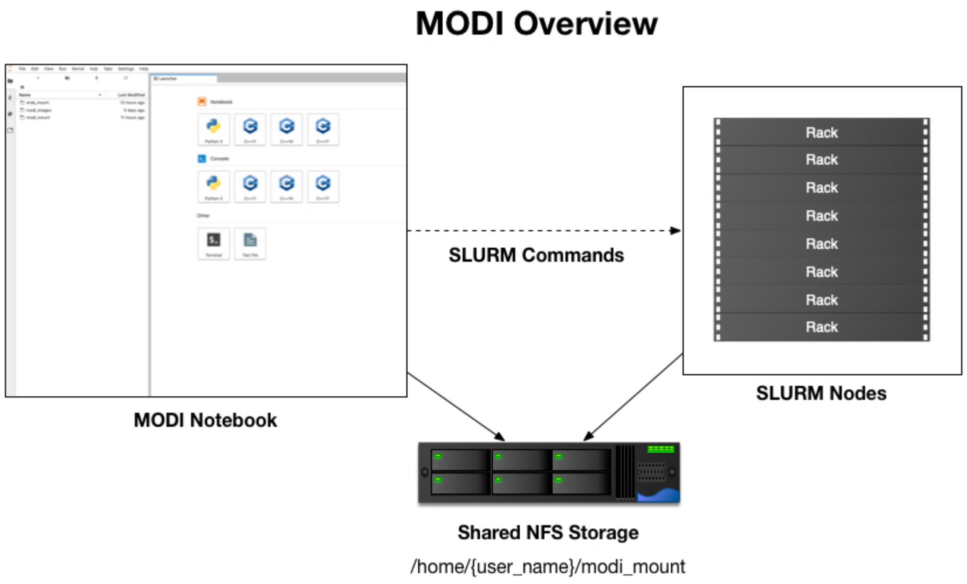

Figure 1: MODI Overview

2.1.1 and the most recent version of Singularity as per our update schedule.

4 Special Directories

When you launch a notebook, it will be spawned into your personal home directory. In this location, you will have a set of preset directories created for you the first time you start a notebook. These include the **erda_mount**, **modi_images**, and **modi_mount**. The purpose of each of these will be explained in the following sections.

4.1 erda_mount

The **erda_mount** directory is where your peronal ERDA home is mounted. This can be verified by executing an **ls ~/erda_mount** which will produce a list of it's current contents. The mount itself is provided by ERDA's standard SSHFS support, therefore the I/O rate is limited by the bandwidth available between the MODI and ERDA systems. Which means that at any point in time the access speed can fluctuate depending on the current usage. A quick test of this is shown in Listing 1 with a single measured contemporary rate of 46.9 MB/s write and 69.8 MB/s read.

```
# Write Test
wlp630_ku_dk@d89000877b60:~/erda_mount$ time \
    dd if=/dev/zero of=~/erda_mount/write_test bs=1M count=2000
2000+0 records in
2000+0 records out
2097152000 bytes (2.1 GB, 2.0 GiB) copied, 44.7334 s, 46.9 MB/s
real 0m44.847s
user 0m0.008s
sys 0m2.328s

# Read Test
wlp630_ku_dk@d89000877b60:~/erda_mount$ time \
    dd if=~/erda_mount/write_test of=/tmp/read_test bs=1M count=2000
2000+0 records in
2000+0 records out
2097152000 bytes (2.1 GB, 2.0 GiB) copied, 30.0353 s, 69.8 MB/s
real 0m30.275s
user 0m0.002s
sys 0m3.599s
```

Listing 1: I/O speed test to ERDA

4.2 modi_images

The **~/modi_images** contains a set of pre-built Singularity images that can be utilized when executing SLURM jobs that require special libraries that are not preinstalled on the compute nodes. At the time of writing, the directory contains the two images, i.e. the **hpc-notebook-latest.sif** and **slurm-notebook-**

latest.sif images as shown in Listing 2. Why these are provided will be explained in Section 7 and their usage throughout Section 9.

```
wlp630_ku_dk@adc3f840e849:~$ ls modi_images/*.sif
modi_images/hpc-notebook-latest.sif \
modi_images/slurm-notebook-latest.sif
```

Listing 2: MODI Singularity Images

4.3 modi_mount

Lastly the **~/modi_mount** directory, is a NFS share that is mounted on every SLURM compute node as shown in Figure 1. This means that any file that the individual node needs to have access to as part of job execution, needs to be located in this directory. If not, the nodes will be unable to load the necessary files. This also means that any output generated by a job needs to be placed in this directory or it won't be retrievable upon job completion. Thus, Slurm will output the results in a standard slurm-xxxx.out file in the directory from which the job was executed. You either need to move the job file into the **~/modi_mount** directory and submit it from there, or utilize the **-o** flag of the **sbatch** command to specify an outfile in this directory, such as **~/modi_mount/results.out**. Additionally, the **modi_mount** directory is currently limited to a maximum of 50 GB per user. Any data that is written beyond this limit will be refused with a "Disk quota exceeded" return message.

5 SLURM Specifications and Job Runtimes

MODI is configured with 3 SLURM partitions that the user can submit their jobs to. This includes the *devel*(the default), *short*, and *long* partitions, in turn each of these partitions have a maximum job runtime limit of 20 minutes, 48 hours, one week, or one month. Jobs that exceed the partitions time limit will be cancelled upon such a violation. In addition, each partition is configured with an associated priority. This priority defines in which turn the individual submitted jobs will be scheduled by the SLURM batch queue. The order of priority is tied to the time each partition allows the jobs to execute for, meaning that the shorter the time limit the higher priority, i.e. *modi_devel*, *modi_short*, *modi_long*, and *modi_max*.

If a job is submitted to a higher priority partition that requires resources that a lower priority partition job is currently using, the lower priority job will be requeued to that partition's batch queue. This means that the lower priority job will be *restarted* once the required resources are available.

6 Getting Started with SLURM

All SLURM jobs in this section are scheduled to the default *modi_devel** partition.

As presented at SLURM User Quickstart, there are a couple of basic commands that can be used to get an overview of the cluster. This includes the supported **sinfo**, **squeue**, and **scontrol**. **Sinfo** as shown in Listing 3 outputs the available partitions (modi_devel*, modi_short, modi_long), their current availability e.g. **up** or **down**, the maximum time a job can run before it is automatically terminated, the number of associated nodes and their individual state, **idle** here means that 8 nodes are available to process jobs. As Listing 3 also shows, after requesting resources for a job via **salloc**, the subsequent state of one of the nodes has changed to **mix**, meaning that currently some resources on the node is being consumed while others are still idle. Other possible node states include down*, draining*, drained*, fail, etc. A full account of the **sinfo** options and outputs, including individual state explanations can be found at sinfo. If a particular node ever gets stuck in an unavailable state such as down* or fail, please raise the issue as instructed in Section 10.

```
wlp630_ku_dk@adc3f840e849:~$ sinfo
PARTITION AVAIL TIMELIMIT NODES STATE NODELIST
modi_devel* up 20:00 8 idle modi[000-007]
modi_short up 2-00:00:00 8 idle modi[000-007]
modi_long up 7-00:00:00 8 idle modi[000-007]

# Request job resources
wlp630_ku_dk@adc3f840e849:~$ salloc
salloc: Granted job allocation {JOB_ID_NUMBER}

wlp630_ku_dk@348bdb8f3a56:~$ sinfo
PARTITION AVAIL TIMELIMIT NODES STATE NODELIST
modi_devel* up 20:00 1 mix modi000
modi_devel* up 20:00 7 idle modi[001-007]
modi_short up 2-00:00:00 1 mix modi000
modi_short up 2-00:00:00 7 idle modi[001-007]
modi_long up 7-00:00:00 1 mix modi000
modi_long up 7-00:00:00 7 idle modi[001-007]

# Cancel job allocation to release node
wlp630_ku_dk@adc3f840e849:~$ scancel {JOB_ID_NUMBER}
wlp630_ku_dk@adc3f840e849:~$ sinfo
PARTITION AVAIL TIMELIMIT NODES STATE NODELIST
modi_devel* up 20:00 8 idle modi[000-007]
modi_short up 2-00:00:00 8 idle modi[000-007]
modi_long up 7-00:00:00 8 idle modi[000-007]
```

Listing 3: Sinfo output

6.1 What is running?

To get an overview of the current queued jobs, the **squeue** command is particularly helpful, especially when combined with **sinfo**. An example of this can be seen in Listing 4. Here the queue is first empty, hereafter we submit a number of MPI based simulations via the **slurm_job.sh** job script. The first submission includes a job that requires two nodes and should execute 128 tasks in total across these two nodes. Then another job is scheduled that requires 256 tasks on four nodes. From this, a call to **squeue** shows what we expect, i.e. that six nodes (modi[000-005]) are currently in alloc mode for job execution and two (modi[006-007]) are still fully available. To utilize these last nodes, three similar jobs are scheduled. The final call to **squeue** then illustrates the inevitable, that the first of the last three jobs (Job 3) is correctly running on the modi[006-007] nodes, and that the two additional jobs are currently in a Pending state where they are either awaiting Resources to be available (Job 5) or a higher Priority job has to be scheduled before it can claim job resources (Job 4).

```
wlp630_ku_dk@adc3f840e849:~$ squeue
JOBID PARTITION NAME USER ST TIME NODES NODELIST(REASON)
wlp630_ku_dk@adc3f840e849:~$ squeue
# Submit a number of MPI jobs to allocate every node
# A single node can process 64 tasks at a time
wlp630_ku_dk@348bdb8f3a56:~/modi_mount/module4/ShallowWater$ sbatch \
    -N 2 --ntasks 128 slurm_job.sh
Submitted batch job 1
wlp630_ku_dk@348bdb8f3a56:~/modi_mount/module4/ShallowWater$ sbatch \
    -N 4 --ntasks 256 slurm_job.sh
Submitted batch job 2

wlp630_ku_dk@348bdb8f3a56:~/modi_mount/module4/ShallowWater$ squeue
JOBID PARTITION NAME USER ST TIME NODES NODELIST(REASON)
2 modi_deve slurm_jo wlp630_k R 0:02 4 modi[002-005]
1 modi_deve slurm_jo wlp630_k R 0:22 2 modi[000-001]

# Node overview
wlp630_ku_dk@348bdb8f3a56:~/modi_mount/module4/ShallowWater$ sinfo
PARTITION AVAIL TIMELIMIT NODES STATE NODELIST
modi_devel* up 20:00 6 alloc modi[000-005]
modi_devel* up 20:00 2 idle modi[006-007]
modi_short up 2-00:00:00 6 alloc modi[000-005]
modi_short up 2-00:00:00 2 idle modi[006-007]
modi_long up 7-00:00:00 6 alloc modi[000-005]
modi_long up 7-00:00:00 2 idle modi[006-007]

wlp630_ku_dk@348bdb8f3a56:~/modi_mount/module4/ShallowWater$ sbatch \
    -N 2 --ntasks 128 slurm_job.sh
Submitted batch job 3
wlp630_ku_dk@348bdb8f3a56:~/modi_mount/module4/ShallowWater$ sbatch \
    -N 2 --ntasks 128 slurm_job.sh
Submitted batch job 4
wlp630_ku_dk@348bdb8f3a56:~/modi_mount/module4/ShallowWater$ sbatch \
    -N 2 --ntasks 128 slurm_job.sh
Submitted batch job 5

wlp630_ku_dk@348bdb8f3a56:~/modi_mount/module4/ShallowWater$ squeue
JOBID PARTITION NAME USER ST TIME NODES NODELIST(REASON)
5 modi_deve slurm_jo wlp630_k PD 0:00 2 (Resources)
4 modi_deve slurm_jo wlp630_k PD 0:00 2 (Priority)
3 modi_deve slurm_jo wlp630_k R 0:03 2 modi[006-007]
2 modi_deve slurm_jo wlp630_k R 0:35 4 modi[002-005]
1 modi_deve slurm_jo wlp630_k R 0:55 2 modi[000-001]

wlp630_ku_dk@348bdb8f3a56:~/modi_mount/module4/ShallowWater$ sinfo
PARTITION AVAIL TIMELIMIT NODES STATE NODELIST
modi_devel* up 20:00 8 alloc modi[000-007]
modi_short up 2-00:00:00 8 alloc modi[000-007]
```

```
modi_long up 7-00:00:00 8 alloc modi[000-007]
```

Listing 4: squeue and sinfo usage

Additional information and explanations about the possible output and states can be found at squeue. Furthermore, basic examples of how to configure and submit simple bash or MPI based job scripts can be found in Section 9.

6.2 Selecting a Partition

As shown in Section 6.1, you can get an overview of the available partitions and their current state by using the sinfo command, an example of this can be seen in Listing 5.

```
wlp630_ku_dk@6155c12973e5:~$ sinfo
PARTITION AVAIL TIMELIMIT NODES STATE NODELIST
modi_devel* up 15:00 1 mix modi000
modi_devel* up 15:00 7 idle modi[001-007]
modi_short up 2-00:00:00 1 mix modi000
modi_short up 2-00:00:00 7 idle modi[001-007]
modi_long up 7-00:00:00 1 mix modi000
modi_long up 7-00:00:00 7 idle modi[001-007]
modi_max up 31-00:00:0 1 mix modi000
modi_max up 31-00:00:0 7 idle modi[001-007]
```

Listing 5: sinfo partition overview

This information can then be used to specify which of the partitions your particular job should be executed in. When using sbatch on the commandline, –**partition/-p** can be used. An example of how this can be used to execute a job on the *modi_short* partition can be seen below in Listing 6:

```
wlp630_ku_dk@6155c12973e5:~/modi_mount/python_hello_world$ sbatch \
    --partition modi_short slurm_job.sh
Submitted batch job 2083
wlp630_ku_dk@6155c12973e5:~/modi_mount/python_hello_world$ squeue
JOBID PARTITION NAME USER ST TIME NODES NODELIST(REASON)
2083 modi_shor slurm_jo wlp630_k R 0:00 1 modi000
609 modi_shor run.sh zsk578_a R 16:55 1 modi000
```

Listing 6: sbatch partition selection

Another way to specify the partition, is to specify it inside the script that is executed with sbatch. An example of this can be seen below in Listing 7:

```
wlp630_ku_dk@6155c12973e5:~/modi_mount/python_hello_world$ cat slurm_job.sh
#!/bin/bash
#SBATCH --partition=modi_short

srun ~/modi_mount/python_hello_world/run.sh
```

Listing 7: sbatch file partition selection

7 Singularity Images

Singularity, as stated in the output from the **singularity help** command, is a Linux container platform designed for HPC environments that enables the mobility of computing on both an application and environment level. This means that we can support a set of prebuilt environments for a wide range of applications that can be executed in an isolated runtime environment. This enables us to configure our compute nodes with a basic installation without having to install special dependencies or maintain the subsequent state of custom packages directly on our MODI SLURM nodes. This has the benefit that we can be quite flexible in terms of supporting many different dependencies separately without risking typical issues such as version conflicts between the different dependencies. However, this does come with the administrative cost of having to continuously manage, update, and test these image environments.

In addition, it also introduces the additional complexity that the user programs to be executed on the MODI cluster need to first verify that the required dependencies (i.e. header files, shared libraries, python packages, Rscripts, etc) are either directly part of the basic CentOS 7 installation on the SLURM nodes, or that they are provided in one of the prebuilt Singularity images located in the **~/modi_images** directory as shown in Listing 2 and explained in 4.2. Both of these options can be tested by simply executing the job with the basic examples as shown in Section 9.1 and 9.4. In both cases, it is enough to verify that the job can execute on one of the compute nodes since they are uniformally configured.

The images that we provide on MODI are available both in their latest and previous built versions on DockerHub and the generating source on nbi-jupyter-docker-stacks, which also provides instructions on how to build an individual image on your local machine.

If it is discovered that particular dependencies are missing from the provided images there are two options fix this. Either follow the instructions as presented on the source's GitHub page to both build, include, and test the required change directly with the source, with an explanation of why it should be accepted. Meaning that you should explain why a particular change should be included by default in every users session.

Upon an acceptance, the change will then be included in the next update of images. The other approach is to get in contact with us through the ERDA ticket system as presented in Section 10 and explain the wanted change to the image. In both cases we will consider the requests on a case by case basis.

8 Installing Custom Packages

Custom packages have to be installed as part of the actual SLURM job. To accomplish this you need to define the installation part in the SLURM script that you submit to the underlying scheduler. In addition, this installation has to be executed inside one of the provided Singularity images as explained in Section 7 and shown in Section 9.2. To avoid this turning into a spaghetti structure, and maintaining sanity, we recommend that your implementation is split into two scripts. The first being a script that defines which Singularity image and path to the second script that defines the installation and job execution. An example of the two script structure and how they can be executed can be seen in Listing 8 and 9. In this example, the tardis package is installed and afterwards executed.

```
#!/bin/bash

$srun singularity exec ~/modi_images/hpc-notebook-latest.sif \
~/modi_mount/tardis/run_tardis.sh
```

Listing 8: slurm_job.sh (script one)

```
#!/bin/bash

# Defines where the package should be installed.
# Since the modi_mount directory content is
# available on each node, we define the package(s) to be installed
# here so that the node can find it once the job is being executed.
export CONDA_PKGS_DIRS=~/modi_mount/conda_dir

# Activate conda in your PATH
# This ensures that we discover every conda environment
# before we try to activate it.
source $CONDA_DIR/etc/profile.d/conda.sh

# As per https://tardis-sn.github.io/tardis/installation.html
# We download and install the tardis environment
wget https://raw.githubusercontent.com/tardis-sn \
    /tardis/master/tardis_env3.yml
conda env create -f tardis_env3.yml
conda activate tardis

# Afterwards we clone and install the tardis package itself
# If supported, this could also have been a regular pip install
git clone https://github.com/tardis-sn/tardis.git
cd tardis
python setup.py install

# Run your application in the current directory
python3 tardis_app.py
```

Listing 9: run_tardis.sh (script two)

A more complex example of installing custom packages can be seen in Listing 10 and 11. Here we install the deeplabcut package, which we subsequently execute as a defined Notebook with the papermill package. This is useful because papermill allows you to execute your existing Notebooks in a SLURM job. Furthermore, example 11 also highlights how you can customize whether the conda environment you aim to activate already exists or not.

```
#!/bin/bash
singularity exec ~/modi_images/hpc-notebook-latest.sif \
    ~/modi_mount/deeplabcut/run_deeplabcut.sh
```

Listing 10: slurm_job.sh (script one)

```
#!/bin/bash

# Defines where the package should be installed.
# Since the modi_mount directory content is
# available on each node, we define the package(s) to be installed
# here so that the node can find it once the job is being executed.
export CONDA_PKGS_DIRS=~/modi_mount/conda_dir

# Activate conda in your PATH
# This ensures that we discover every conda environment
# before we try to activate it.
source $CONDA_DIR/etc/profile.d/conda.sh

# Either activate the existing environment
# or create a new one
conda activate DLC
if [ $? != 0 ]; then
    conda create -n DLC -y python=3.8
    conda activate DLC
fi

# Install the packages into the conda environment that was
activated.
pip3 install -q deeplabcut==2.2rc3 tensorflow papermill ipykernel
# Ensure that the Jupyter kernel is available for papermill.
python3 -m ipykernel install --user --name=DLC

# Transform and execute the deeplabcut.ipynb notebook
# in the created kernel and put the results in
# the deeplabcut.result.ipynb output file
papermill -k DLC deeplabcut.ipynb deeplabcut.result.ipynb
```

Listing 11: run_deeplabcut.sh (script two)

To execute either of these two examples, the ‘slurm_job.sh‘ has to be submitted to the SLURM queue via the **sbatch** command as highlighted in the **Hello World** example 9.1.

9 Examples

In this section a couple of examples on how to use the system will be presented. This includes how to get a simple batch job working, how to scale it to run on multiple nodes, how submit jobs to be executed in a Singularity image environment and how to test whether a particular image has the required dependency to execute a particular program.

9.1 SLURM Hello World Job

First we will get a range of nodes to output the string "Hello World" to an output file. The first example will get a single node to accomplish this. Starting in your home directory i.e. in the Jupyter Terminal.

```
wlp630_ku_dk@669ffda64cbc:/some/other/directory/path$ cd
wlp630_ku_dk@669ffda64cbc:~$
```

Listing 12: Go to your home directory

In this location you have the mentioned directories.

```
wlp630_ku_dk@669ffda64cbc:~$ ls -l
total 8
drwxr-xr-x. 1 501 501 4096 May 27 10:51 erda_mount
drwxr-xr-x. 4 root root 102 May 25 11:07 modi_images
drwxr-xr-x. 2 wlp630_ku_dk users 4096 May 27 12:08 modi_mount
```

Listing 13: List of home directories

To make our life easy in terms of managing where the output should be produced, we will move into the **~/modi_mount** directory and create the **hello_world.sh** job file.

```
wlp630_ku_dk@669ffda64cbc:~$ cd modi_mount
wlp630_ku_dk@669ffda64cbc:~/modi_mount$ vi hello_world.sh
#!/bin/bash
echo "Hello␣World"
```

Listing 14: Create job file

In the same location, run the following command to submit the file as a SLURM job to be executed by a now.

```
wlp630_ku_dk@669ffda64cbc:~/modi_mount$ sbatch hello_world.sh
Submitted batch job {JOB_ID_NUMBER}
```

Listing 15: Submit the hello_world.sh file

After this, there will immediately be an output file with a default name of **slurm-{JOB_ID_NUMBER}** present in the same directory as from which you executed the **sbatch** command. Initially, this will have a size of 0 bytes and have zero content. However, as the job produces stdout strings they will be appended into this file. In this instance, this should produce the following.

```
wlp630_ku_dk@669ffda64cbc:~/modi_mount$ cat \
    slurm-{JOB_ID_NUMBER}.out
Hello World
```

Listing 16: Retrieve job output

This was produced by one of the **modi00[0-7]** nodes as highlighted in Section 6. To get information on which node executed the job, we can execute the system provided **hostname** command to retrieve this, e.g.

```
wlp630_ku_dk@669ffda64cbc:~/modi_mount$ vi echo_hostname.sh
#!/bin/bash
hostname
```

Listing 17: Know who executed the job

If we resubmit and retrieve the result, we should get.

```
wlp630_ku_dk@669ffda64cbc:~/modi_mount$ cat \
    slurm-{JOB_ID_NUMBER}.out
modi00{MODI_NODE_NUMBER}.science
```

Listing 18: Hostname of executing node

Additionally, if we want to specify how many nodes that should be allocated to this job, the **-N** flag can used. However, as indicated in Listing 19, the **sbatch** command is only responsible for allocation of nodes to the job and will not launch additional tasks per node. Instead, the **srun** command is responsible for doing this, and as shown in Listing 20 we need to prepend the task with the **srun** command. This will execute the command on the additional allocated nodes to the particular job.

```
wlp630_ku_dk@d89000877b60:~/modi_mount$ sbatch -N 8 echo_hostname.sh
Submitted batch job {JOB_ID_NUMBER}
wlp630_ku_dk@d89000877b60:~/modi_mount$ cat \
    slurm-{JOB_ID_NUMBER}.out
modi00{MODI_NODE_NUMBER}.science
```

Listing 19: Job not scaled

```
wlp630_ku_dk@669ffda64cbc:~/modi_mount$ cat echo_hostname.sh
#!/bin/bash
srun hostname

wlp630_ku_dk@d89000877b60:~/modi_mount$ sbatch -N 8 \
    echo_hostname.sh
Submitted batch job {JOB_ID_NUMBER}
wlp630_ku_dk@d89000877b60:~/modi_mount$ cat \
    slurm-{JOB_ID_NUMBER}.out
modi000.science
modi001.science
modi004.science
modi002.science
modi006.science
modi005.science
```

```
modi007.science
modi003.science
```

Listing 20: Launch additional tasks via srun

Please refer to the sbatch man page and srun man page man pages for further information about available flags and options.

9.2 SLURM Job with Singularity

To begin with, we will submit a basic job as in the Hello World 9.1 example, but in this instance we will execute the binary inside a Singularity runtime environment. An example of this can be seen in Listing 21. Here, the **echo "Hello World"** command is executed within the environment provided by the **~/modi_images/slurm-notebook-latest.sif** image.

```
wlp630_ku_dk@adc3f840e849:~/modi_mount$ vi hello_world.sh
#!/bin/bash
singularity exec ~/modi_images/slurm-notebook-latest.sif \
    echo "Hello␣World"

wlp630_ku_dk@adc3f840e849:~/modi_mount$ sbatch hello_world.sh
Submitted batch job {JOB_ID_NUMBER}
wlp630_ku_dk@adc3f840e849:~/modi_mount$ cat \
    slurm-{JOB_ID_NUMBER}.out
Hello World
```

Listing 21: Singularity Hello World

The difference here can be further illustrated by retrieving the operating system that the image provides, as shown in Listing 22. We can see that instead of being the native "NAME="CentOS Linux" OS, we are now executing inside an Ubuntu environment. The reason for this difference is that the images we provide inherit the base configuration from the official Jupyter team's images, which uses the Ubuntu distribution for images.

```
wlp630_ku_dk@adc3f840e849:~/modi_mount$ cat os_release.sh
#!/bin/bash
singularity exec ~/modi_images/slurm-notebook-latest.sif \
    cat /etc/os-release

wlp630_ku_dk@adc3f840e849:~/modi_mount$ sbatch os_release.sh
Submitted batch job {JOB_ID_NUMBER}
wlp630_ku_dk@adc3f840e849:~/modi_mount$ cat \
    slurm-{JOB_ID_NUMBER}.out
NAME="Ubuntu"
VERSION="18.04.1␣LTS␣(Bionic␣Beaver)"
ID=ubuntu
```

```
ID_LIKE=debian
PRETTY_NAME="Ubuntu␣18.04.1␣LTS"
VERSION_ID="18.04"
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/
␣␣␣␣terms-and-policies/privacy-policy"
VERSION_CODENAME=bionic
UBUNTU_CODENAME=bionic
```

Listing 22: Singularity Ubuntu Environment

If the to be scheduled application requires additional dependencies that are not by default available on the MODI SLURM nodes, the job will fail. To resolve this, the provided Singularity images can be used to support custom dependencies, a further explanation about how this is accomplished can be found in Section 7. However, it is not a given that the prebuilt images will provide the required dependencies. The steps presented in section 9.4 are applicable to verify that.

9.3 MPI SLURM Job via Singularity

To submit an MPI job, the simplest approach to ensure compatibility on the SLURM nodes is to execute the MPI program inside one of the provided Singularity images. This is especially important when dealing with a program that has shared library dependencies.

Furthermore, in relation to shared library dependencies, it is also recommended that the compilation itself takes place in the same notebook image that is used to schedule the job. Meaning, that if the dependencies are provided by the ucphhpc/slurm-notebook image (which is therefore used for the job execution), it is recommended that the compilation of the program takes place in the same MODI notebook image.

For instance, if we want to test the simple C Hello World MPI program in Listing 23 by executing it within the ucphhpc/slurm-notebook image. We can simply attempt to compile and execute it within the notebook terminal in a spawned Slurm Notebook on MODI as shown in Listing 24.

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    // setup size
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    // setup rank
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // get name
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // output combined id
    printf("Hello world from processor %s, "
           "rank %d out of %d processors\n",
           processor_name, world_rank, world_size);
    MPI_Finalize();
}
```

Listing 23: main.c

```
# Figure out if the required non standard header file mpi.h
# is present in the image
wlp630_ku_dk@adc3f840e849:~/modi_mount$ find /usr \
    -type f \
    -name mpi.h \
    /usr/lib/x86_64-linux-gnu/openmpi/include/mpi.h \
    | grep include/mpi.h

# Compile the main.c source in ~/
# with both including the required header
# and link against the shared library libmpi.so
wlp630_ku_dk@adc3f840e849:~$ gcc main.c \
    -I/usr/lib/x86_64-linux-gnu/openmpi/include \
    -L/usr/lib/x86_64-linux-gnu/openmpi/lib \
    -lmpi \
    -o main

# Execute the output file
wlp630_ku_dk@adc3f840e849:~$ ./main
Hello world from processor adc3f840e849, rank 0 out of 1 processors
```

Listing 24: Test main.c support

As the output shows in Listing 24, the ucphhpc/slurm-notebook image is able to both compile and execute the main.c program on MODI. This means that we should be able to execute it across the SLURM nodes by replicating the approach in section 9.2. Namely, creating and submitting a SLURM job script as shown in Listing 25. From the result we can see that the program was successfully executed within the image on each of the nodes.

```
# First move the binary into the ~/modi_mount directory
# so the SLURM nodes will have access to it.
wlp630_ku_dk@adc3f840e849:~$ mv main modi_mount/

# Create job script file
wlp630_ku_dk@adc3f840e849:~/modi_mount$ vi job.sh
#!/bin/bash
singularity exec ~/modi_images/slurm-notebook-latest.sif \
    ./main

# Schedule 10 tasks on each node
wlp630_ku_dk@adc3f840e849:~/modi_mount$ sbatch -N 8 --tasks 80 job.sh
Submitted batch job {JOB_ID_NUMBER}
# Check queue
wlp630_ku_dk@adc3f840e849:~/modi_mount$ squeue
JOBID PARTITION NAME USER ST TIME NODES NODELIST(REASON)
{JOB_ID_NUMBER} modi job.sh wlp630_k R 0:00 8 modi[000-007]

wlp630_ku_dk@adc3f840e849:~/modi_mount$ cat \
    slurm-{JOB_ID_NUMBER}.out
...
Hello world from processor modi000.science, rank 4 out of 80 processors
Hello world from processor modi001.science, rank 10 out of 80 processors
Hello world from processor modi002.science, rank 22 out of 80 processors
Hello world from processor modi003.science, rank 31 out of 80 processors
Hello world from processor modi004.science, rank 41 out of 80 processors
Hello world from processor modi005.science, rank 51 out of 80 processors
Hello world from processor modi006.science, rank 63 out of 80 processors
Hello world from processor modi007.science, rank 73 out of 80 processors
...
```

Listing 25: Submit MPI job

9.4 Test Image Dependencies Support

There are several options to verify that a particular image has the necessary dependencies/configuration to execute a particular program.

For environment verification we recommend testing the application support directly inside the spawner MODI notebook itself, or by testing the image environment locally on your personal system.

On MODI, the simplest approach is to follow the steps in Section 9.3 on one of the available Singularity images in the **~/modi_images** directory and simply replace the MPI aspect with the dependencies that are to be tested.

To test it locally a couple of prerequisites have to be met. This includes either having Docker or Singularity installed on your host system before you can proceed.

Then, you either need to pull a version of the image to be tested from DockerHub, or build a local version directly from the source nbi-jupyter-docker-stacks.

It should be noted that if you build it yourself, the installed versions within the built image may differ from the official version on MODI. The reason being that any released version of the image reflects the up to date software versions available at the time of the build. Therefore, it is best to use prebuilt versions when testing your application to ensure compatibility.

An example of how the prebuilt image can be downloaded via either Docker or Singularity can be seen in Listing 26 and 27. Further explanations and documentation on these commands can be found at Docker and Singularity.

```
# Docker pull
docker pull ucphhpc/slurm-notebook
```

Listing 26: Docker pull image to your own machine

```
# Singularity pull
singularity pull docker://ucphhpc/slurm-notebook
```

Listing 27: Singularity pull image to your own machine

Upon having the particular image prepared, the next steps include spawning a bash shell inside the image environment, mounting the application source that is to be tested within the environment, optionally compiling the source into a binary, and executing the prepared program. Examples of this can be seen in Listings 28 and 29

```
# Start an image environment and mount the source mpi_test
# directory into the /root/ path and change the workdir to /root
docker run -w /root -it \
    --mount type=bind,src=$(pwd)/mpi_test,dst=/root/mpi_test \
    ucphhpc/slurm-notebook bash

# List directories within the image's /root path
root@dfb16d84c340:/root# ls
mpi_test
# Change to the mpi_test directory
root@feb0bd58b791:/root# cd mpi_test/
# Since it's a C source we need to compile it and attempt to execute it
root@feb0bd58b791:/root/mpi_test# gcc main.c -o main
main.c:4:10: fatal error: mpi.h: No such file or directory
 #include <mpi.h>
          ^~~~~~~
compilation terminated.

# Since we can't find the header file from the default path,
# we can try and search for it in the system.
root@feb0bd58b791:/root/mpi_test# find / -type f -name mpi.h
/usr/lib/x86_64-linux-gnu/openmpi/include/mpi.h
# Include the header and link the shared mpi library
root@feb0bd58b791:/root/mpi_test# gcc main.c \
    -I/usr/lib/x86_64-linux-gnu/openmpi/include -lmpi -o main
```

```
# Execute binary
root@feb0bd58b791:/root/mpi_test# ./main
Hello world from processor feb0bd58b791, rank 0 out of 1 processors
```

Listing 28: Docker mount and execute program

```
# Start a bash shell within the container image environement
# Since with Singularity you share the filesystem with the
# actual host, you simply need to spawn the shell from the location
# of the mpi_test directory
root@hostname:~# singularity exec slurm-notebook-latest.sif bash
# List the directories from the current location
# Here the mpi_test directory should be included
root@hostname:~# ls
mpi_test

# Next follow the same steps as in Listing 19
root@hostname:~/mpi_test# cd mpi_test/
root@hostname:~/mpi_test# gcc main.c \
    -I/usr/lib/x86_64-linux-gnu/openmpi/include -lmpi -o main
# Execute binary
root@hostname:~/mpi_test# ./main
Hello world from processor hostname, rank 0 out of 1 processors
```

Listing 29: Singularity mount and execute program

10 Further Support

If any issue, question or request arises while using the MODI system, please contact either support@erda.dk or info@erda.dk to get in touch with us.