Northwestern University Computer Science department SLURM user guide

Cameron Churchwell
Edited by Max Morrison and Bryan Pardo

August 5, 2024

Contents

Figure 1: Default Frog

1 Introduction

This is a non-exhaustive guide with the goal of explaining how to setup efficient workflows on the Northwestern University Computer Science Department SLURM cluster.

1.1 Quick Start Code

To help new users get started as quickly as possible with a minimal setup, I have created a repository that should contain close to the bare minumum of the code you might need to configure an Apptainer container to run on our SLURM cluster for machine learning research. If the previous sentence didn't have much meaning for you, please continue to the next section.

1.2 Background Definitions

If you're new to containerized computing, you should really read these before going on.

- **SLURM** | A tool that allows the scheduling and sharing of compute resources effectively between different people and/or groups. SLURM is more efficient than simply allocating a fixed amount of resources to individuals in a virtual machine (VM) for an indefinite period. With a standard VM, if an individual is not running anything, the compute resources assigned to the VM are idle. With SLURM, the moment one user's job is done, the next user's job is started, with almost no downtime. To work with SLURM, instead of putting your project in a standard VM, your work must be put in a container.
- **Container** | A virtual machine running a virtual operating system, separate from the host machine. Containers differ from typical virtual machines in that they share more resources between each other and sometimes with the host machine. Containers package your application and its container dependencies with everything it needs to run, including: the operating system, application code, runtime, system tools, system libraries, etc. The containerization system used by our cluster is Apptainer.
- **Apptainer** | A system for creating lightweight and fast virtual machine containers for scientific computing applications. It is similar to Docker or Podman.
- **Cluster** | A collection of either virtual, logical, or physical machines (or all three) which work together for a shared purpose or purposes. In our case, the cluster is a collection of powerful server machines which we can queue tasks (e.g. your machine learning task) onto.
- **Node** | A node is one element of a network or cluster. In our case, we have both access nodes and compute nodes. The access nodes are what we interface and interact with, and the compute nodes actually run tasks for us.

1.3 Overview

To make a project and run it on the SLURM cluster you take the following steps:

1. You define an Apptainer Container (Section 2.1)
    - create mount points for external directories or files
    - install a python environment manager (micromamba or miniconda)
    - create a python environment and install packages
    - install any other dependencies
    - define container run behavior
2. You build the Apptainer image file (Section 2.2)
3. You create a run script (Section 2.3)

- This contains the shared boilerplate code for running jobs inside Apptainer sessions and inside the python environment inside that Apptainer session

4. You write batch job scripts (Section 3)
    - These scripts contain information about the job itself and the resources to allocate when the job is queued and run.
5. You run EITHER an interactive or a non-interactive SLURM job session
    - Non interactive is for longer tasks (Section 3)
    - Interactive is for short, one-off tasks and debugging

1.4 Remote Access

The SLURM cluster is, as the name suggests, a group of servers which operate together. Users interact with this group by connecting to specific access nodes via SSH(Secure Shell). If you are unfamiliar with SSH, then you may not have the background needed for this guide. Godspeed.

At the time of writing, there is a single access node, `slurm01.cs.northwestern.edu`, which users can connect to using their NetID username and password. Upon initial login, users should also upload an SSH key for easier use.

The access node can run VSCodeServer which allows you to interact with it through VSCode almost as though it was your local machine. This is incredibly helpful, but has a few downsides that have not yet been resolved, a few of which will be discussed later.

1.5 Slurm Commands

There are a few important basic SLURM commands which the user will need to know:

- squeue — Outputs the jobs currently queued, stopped, and running on compute nodes.
- srun — Starts a job which runs in the foreground. With the additional argument `--pty`, the job will be interactive.
- sbatch — Queues a non-interactive job to run in the background. Can be used to queue multiple jobs, and has a system for managing dependencies between jobs.
- scancel — Cancels a currently running non-interactive job when passed the ID of the job you want to cancel.

You may note that these are just otherwise-reasonably-named commands with the letter 's' prepended. The *queue* command shows the currently running and waiting jobs. The *run* command launches a SLURM job and executes whatever command you pass inside of that job. The *batch* command takes a SLURM job description file, explained later in this document, and launches the described job with the requested resources. The *cancel* command can be used to cancel a job by its ID.

Here is a basic example of how to launch an interactive slurm job with a single a40 GPU, 8 CPUs, and 8GB of ram, running a shell.

```
srun --pty --gres=gpu:a40:1 --cpus-per-task=8 --mem=8G /bin/bash
```

The effect of this is to give the user an interactive shell (*–pty*) on a compute node with access to a GPU and sufficient CPU power and memory to accomplish most basic tasks. This is a useful setup for debugging a program with a debugger like PDB in Python, as PDB breakpoints seem to cause jobs to crash if they are not interactive, due to not having an input stream.

Note that the above command does not launch an Apptainer session so you would not be able to run commands like python inside of the shell. To learn about how to launch Apptainer sessions on the compute nodes, please see (Section 2.3).

While *srun* can be useful for debugging, in general it is advised that you use *sbatch* to launch jobs instead of *srun*, because jobs launched with *srun* will halt if your *SSH* session dies.

2 Building and Using Apptainer Containers

In order to use third party tools like Python on the SLURM cluster, we need to install these tools inside containers. In this guide we will explain how to install a Python environment manager and python packages inside such a container.

At the time of writing, the SLURM cluster is configured to use Apptainer for containerization. This allows each user to have one or more distinct Linux environments in their container(s). This prevents users from installing or modifying packages globally on the access nodes or compute nodes and interfering with the work of other students.

Such an environment can be created by writing a definition file (.def) and then running the Apptainer build command to generate a Singularity image file (.sif) You then can use the image file like an executable to run commands inside of the virtual environment.

2.1 Definition Files

Apptainer uses Singularity Image Format (SIF) as its default container format. To create a .sif file you write a Definition (.def) files, which describes how the container should be created and what should be included inside of it. The full documentation for definition files can be found here. The definition file below is available as part of the quick-start code linked in Section 1.1. In this case, it defines an image file with Micromamba, a python environment manager like Conda, but smaller and faster. You could use Conda or another variant if you wanted by finding the relevant image on Docker Hub. If you have used Docker before, this should be a familiar process.

```
# The source of our OS is Docker
Bootstrap: docker
# Install an image containing micromamba, a miniconda replacement
From: mambaorg/micromamba

# These are arguments that get passed to the `apptainer build`
#  command and can then be used in this definition file
%arguments
    CONDA_ENV_NAME=my_environment

# This is the script that runs once after the OS has been
#  built for the image. This is where you want to install
#  any additional tools you might need (e.g. other compilers)
%post
    apt-get update && apt-get install -y \
        build-essential gcc

# This is what runs when you invoke the image file in the shell
# So `./image.sif python a.py` is equivalent to
#  `/bin/miromamba run -r ~/micromamba/ -p my_environment python a.py
# This section also will create the conda env if it does not already
#  exist in the default location in your home directory
%runscript
    #!/bin/bash
    # echo the command back for logging
    echo "Command: $*"

    # Path to python environment (probably don't change this line)
    # `{{ CONDA_ENV_NAME }}` gets replaced with the value of the
    #  conda environment, passed in at build time. Default is
    #  "my_environment" as you can see above
    directory="$HOME/micromamba/envs/{{ CONDA_ENV_NAME }}"

    # Check if the directory exists
```

```
if [ -d "$directory" ]; then
    echo "Found python environment $directory"
else
    echo "Did not find $env at $directory, creating with python 3.10"
    # create the env with python 3.10 installed and nothing else
    /bin/micromamba create -r ~/micromamba/ -n {{ CONDA_ENV_NAME }} \
        -c conda-forge -y python=3.10
fi

# run the user's command inside the conda environment
/bin/micromamba run --root-prefix ~/micromamba/ \
    --prefix {{ CONDA_ENV_NAME }} $*
```

2.2 Building a Container Image File

As previously mentioned, Apptainer uses Singularity Image Format (SIF) containers. The following command will build the image defined in *image.def* into an image called *image.sif*

```
srun --mem 1G apptainer build \
--force image.sif image.def
```

The following script is part of the Section 1.1 quick-start-code and shows how template variables in the build script can be used.

```
#!/bin/bash

# This part of the script was written by ChatGPT 3.5!
#  Which is nice because I don't know how to shell script and don't care
#  to learn :)

# Prompt the user for IMAGE_NAME with a default value
read -p "Enter IMAGE_NAME (default: image): " image_name
image_name=${image_name:-image}

# Prompt the user for CONDA_ENV_NAME with a default value
read -p "Enter CONDA_ENV_NAME (default: my_environment): " conda_env_name
conda_env_name=${conda_env_name:-my_environment}

# Display the entered or default values
echo "IMAGE_NAME: $image_name"
echo "CONDA_ENV_NAME: $conda_env_name"

# End ChatGPT

echo "Starting image build on compute node"

# srun runs the following command on a SLURM compute node
# `--mem 300M` allocates 300MB of ram
# `--cpus-per-task 2` allocates 2 logical CPU cores
# 'apptainer build' is the actual command we are running on the
#   compute node
#  This command takes a definition file and creates an image file
# `--force` overwrites an existing image file
# $image_name.sif is out output image file
# $image_name.def is the input file
srun \
 --mem 1G \
 --cpus-per-task 2 \
```

```
apptainer build \
--force \
--build-arg "CONDA_ENV_NAME=$conda_env_name" \
"$image_name.sif" \
"$image_name.def"
```

2.3 Run Scripts

A useful pattern is to create a run script which essentially just prepends the necessary boilerplate to commands in order to execute them inside of Apptainer. Below is an example run script. This code is available as part of the Section 1.1 quick-start-code.

```
#!/bin/bash
apptainer run --nv image.sif $*
```

This script takes any number of arguments and passes them into an Apptainer session. The `nv` option passes NVidia drivers through, allowing the use of GPUs.

Finally, the `project_name.sif` file is the image file you created in the previous step, and the `$*` means pass all arguments.

3 Example Batch Job Files

To schedule non-interactive jobs to run in the background, you simply create a job script. This is essentially a shell script but with some important metadata relating to resource allocation added as comments at the top of the file. Some good documentation can be found here. All of the following scripts use the run script from the previous section.

Here are some examples of job scripts for common tasks:

3.1 Start a Jupyter Server

```
#!/bin/bash
# Number of machines to run the job on
#SBATCH --nodes=1
# Number of processes per machine
#SBATCH --ntasks=1
# The GPU(s) to allocate. Form is gpu:<model>:<count>
#SBATCH --gres=gpu:a40:1
# The amount of ram to allocate
#SBATCH --mem=16G
# The number of logical CPU cores to allocate
#SBATCH --cpus-per-task=8
# The format of the name for the output files created for this job.
#  If this file is named jupyter.sh, the output file will look like
#  <JobID>-jupyter.sh.out
#SBATCH --output=%j-%x.out

./run.sh python -m jupyterlab --port=$1
```

Example usage:

```
> sbatch jupyter.sh 8888
Submitted batch job 8
```

You then need to port forward from the compute node to the access node, and from the access node to your local machine, which is described in the next subsection.

3.2 Port Forward

This forwards a connection from the compute node to an access node.

```
#!/bin/bash
# Number of machines to run the job on
#SBATCH --nodes=1
# Number of processes per machine
#SBATCH --ntasks=1
# The amount of ram to allocate
#SBATCH --mem=100M
# The number of logical CPU cores to allocate
#SBATCH --cpus-per-task=1
# The format of the name for the output files created for this job.
#  If this file is named ssh.sh, the output file will look like
#  <JobID>-ssh.sh.out
#SBATCH --output=%j-%x.out

ssh -N -R 0:localhost:$1 slurm01.cs.northwestern.edu
```

Example usage:

```
> sbatch ssh.sh 8888
Submitted batch job 9
> cat 9-ssh.sh.out
Allocated port 39743 for remote forward to localhost:8888
```

Then you can port forward from the access node to your host machine as you would in a normal workflow. There is almost certainly a better way of handling this, but this method works. If the ssh session dies, you can simply start a new one.

3.3 TensorBoard

TensorBoard is a tool from Google that provides visualization and tooling needed for machine learning experimentation. The following shows an example of a batch file for running TensorBoard on SLURM.

```
#!/bin/bash
# Number of machines to run the job on
#SBATCH --nodes=1
# Number of processes per machine
#SBATCH --ntasks=1
# The amount of ram to allocate
#SBATCH --mem=8G
# The number of logical CPU cores to allocate
#SBATCH --cpus-per-task=2
# The format of the name for the output files created for this job.
#  If this file is named tensorboard.sh, the output file will look like
#  <JobID>-tensorboard.sh.out
#SBATCH --output=%j-%x.out

./run.sh tensorboard --logdir repos/project/runs/ --port $1
```

Example usage (this time including the port forward):

```
> sbatch tensorboard.sh 8080
Submitted batch job 10
> sbatch ssh.sh 8080
Submitted batch job 11
> cat 11-ssh.sh.out
Allocated port 12345 for remote forward to localhost:8080
```

Then port forward 12345 to your local machine.

3.4 Training (with GPU)

The following batch file can be used to deploy a machine learning training session.

```
#!/bin/bash
# Number of machines to run the job on
#SBATCH --nodes=1
# Number of processes per machine
#SBATCH --ntasks=1
# The GPU(s) to allocate. Form is gpu:<model>:<count>
#SBATCH --gres=gpu:a40:1
# The amount of ram to allocate
#SBATCH --mem=16G
# The number of logical CPU cores to allocate
#SBATCH --cpus-per-task=8
# The format of the name for the output files created for this job.
#  If this file is named train.sh, the output file will look like
#  <JobID>-train.sh.out
#SBATCH --output=%j-%x.out

./run.sh python -m project.train --dataset mnist --gpus 0 \
--config "/repos/project/config/$1.yaml"
```

Example usage:

```
> sbatch train.sh default_config
Submitted batch job 12
```

4 Files and Storage

The local filesystem is passed through to the compute nodes when a job is started. You should have been notified of a proper directory to store your projects for your lab/group. Only some parts of the local file system are passed through to Apptainer images, and both your home directory as well as your lab's allocation should be passed through, so you should not need to worry about bind mounts or other such techniques.

You can check the status of your group's allocation via the command `quota`. Here is an example:

```
> quota -gs -g pardo
```

This outputs the storage quota and usage for the 'pardo' group. You should ensure that your files are owned by the group you belong to, both so that you can collaborate with your colleagues/peers as well as so that the drive space used for these files is properly counted towards your disk quota.

5 Useful Workflows

There are various useful tricks and workflows that can make working with this setup easier. Here are a few examples:

5.1 Job Output

One useful trick is to include the following line...

```
#SBATCH --output=%j-%x.out
```

...in your batch job files. When a non-interactive job runs, its output is written to a file instead of the standard output and error streams. This line writes that output to a file of the form

```
JobID-JobFilenameWithExtension.out
```

For example:

```
120-train.sh.out
```

This makes it easy to tell what a job was and to tab complete the full name of the output file based on just the job ID.

To see the full output of a job you can simply use cat:

```
> cat 120-train.sh.out
```

To follow it and see the output in real time, you can use tail:

```
> tail -f 120-train.sh.out
```

5.2 Run any Python File

The following batch job will run any python script you pass it, so long as it is mounted into the apptainer image.

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --mem=4G
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=2
#SBATCH --output=%j-%x.out

./run.sh python $*
```

Example usage:

```
> sbatch run-python.sh /repos/module_name/random_task.py
```

This can be helpful for running scripts that you only need to run a few times and for which the resources allocated are not as relevant.

5.3 Debugging GPU Jobs

The following script can be used to run a batch job script interactively for debugging

```
#!/bin/bash
srun --pty --gres=gpu:a40:1 --mem=8G --cpus-per-task=4 $*
```

Example usage:

```
> run-gpu.sh train.sh buggy_config
```

5.4 Setup Scripts

It can be helpful to have a script which will install and download everything from scratch. Here is an example:

```
#!/bin/bash
#SBATCH --nodes 1
#SBATCH --mem 8G
#SBATCH --ntasks 1
#SBATCH --cpus-per-task 4
#SBATCH --output=%j-%x.out

apptainer build --force image.sif image.def
./run.sh python -m module_name.data.download --datasets vctk
./run.sh python -m module_name.data.augment --datasets vctk
./run.sh python -m module_name.partition --datasets vctk
```

This will build the Apptainer image and then run a download, augmentation, and partition script. Note that preprocessing is not included here, as that might require a GPU and it would not be good to tie up a GPU to simply sit idle while downloading is happening. In general you want to only request the resources you need.

6 Chaining Jobs

You can chain jobs when you submit them so that they will run sequentially assuming no errors. Here is an example:

```
> sbatch download.sh
Submitted batch job 13
> sbatch --dependency=afterok:13 preprocess.sh greyscale_pixels
Submitted batch job 14
> sbatch --dependency=afterok:13 partition.sh train test valid
Submitted batch job 15
> sbatch --dependency=afterok:14:15 train.sh convnet
Submitted batch job 16
```

Here downloading will start immediately, and preprocessing and partitioning will only start once the download script has finished successfully (with exit code 0). Note that both preprocessing and partitioning can depend on downloading. Then, training will only start once both the partitioning and the preprocessing have finished successfully. It is also possible to convert this into a script as well for easy repeating.

More general documentation on job dependencies can be found here.