User Guide for SLURM Scheduler on Rocky Linux 9

Welcome to the SLURM scheduler user guide. This guide is ideal for users running computational jobs requiring GPU resources for data science, deep learning, and interactive Jupyter Lab sessions. This document will help you understand how to submit and manage jobs on the SLURM scheduler, including both interactive and non-interactive jobs. We'll cover essential commands like `squeue`, `srun`, `sbatch`, and `scancel`, and we'll reference tutorial files located in `/tutorial/` outlining how to run non-interactive slurm jobs that launch interactive Jupyter Lab containers that you access from your Web Browser.

Table of Contents

Introduction

This guide is designed to help you effectively use the SLURM scheduler on our Rocky Linux 9 server equipped with two NVIDIA A100 GPUs. The Server allows you to run computational jobs, including GPU-accelerated tasks, using both interactive and non-interactive modes.

Server Resources

- **CPU:** Qty x 2, Intel(R) Xeon(R) Gold 6338 CPU @ 2.00GHz
- **Cores:** 64 Cores / 128 Threads (32C/64T per processor)

- **RAM:** 512 GB (500GB available in slurm)
- **Disk:** User disk quotas are calculated nightly based on remaining disk space
    - You can check your disk quota utilization that is shared with **all users** with: `$ quota -v`
    - Disk quotas are set high to support large computational jobs. However, unnecessary data should be backed up and removed from the server.
    - Please be mindful on your disk utilization. While the containerized environment only temporarily creates a runtime layer for your workflow any data you save on the server will be persistent. You are responsible for backing up your data, as it may be removed at any time to accommodate other users' jobs, etc. This server doesn't perform data backups.
    - **Please consider purchasing additional disks through the system administrator for the server if your research lab needs a dedicated storage space or data redundancy.**
- **Operating System:** Rocky Linux 9
- **GPUs:** 2 x NVIDIA A100
- **Scheduler:** SLURM
- **Container Runtime:** Podman (supports rootless containers)

Accessing the Server

Please note this server requires a public ssh key to be registered with the server prior to being able to connect with your siu85xxxxxxx DawgTag and Single Sign-On password. You can request that your self generated public ssh key to be registered with the School of Computing at:

https://www2.cs.siu.edu/~mbarkdoll

Once you've registered your public ssh key with the server and been assigned access then you can access the Server with your SSH account.

Use the following command to connect to a remote linux server with ssh:

```
$ ssh siu85xxxxxxx@hostname
```

Replace `siu85xxxxxxx` with your actual username and `hostname` with the server's hostname or IP address.

Running Jobs

SLURM manages job scheduling and resource allocation. You can submit jobs in two ways (non-interactive and interactive):

- **Interactive Jobs:** Use `srun` for interactive sessions.
- **Non-Interactive Jobs:** Use `sbatch` to submit job scripts.

Interactive jobs are ideal for debugging or testing, whereas non-interactive jobs allow for long-running processes typically without user intervention.

Our preferred method is the following:

- **Podman Jupyter Lab Web Browser Container Jobs:** Use `sbatch` to submit a non-interactive job script that launches an interactive jupyter lab that we'll access from our web browser.

Podman Jupyter Lab Web Browser Container Jobs

This section will guide you through submitting a non-interactive SLURM job to launch a Jupyter Lab session via your web browser. The Jupyter Lab environment will be containerized using Podman, and it supports data science and deep learning tasks.

Jupyter Lab containerized Web Browser Jobs run inside a preconfigured Podman Container running Jupyter Lab sessions that have data science and deep learning environments. You can access this Jupyter Lab from a computer connected to SIU's eduroam or VPN network.

To utilize interactive Jupyter Labs from our web browser, we'll utilize a non-interactive slurm sbatch job that automatically obtains a tcp port on the server for the end-user's slurm job. This port will allow a personalized access point from your personal device via a web browser. The slurm job will also launch the Jupyter Lab container with preconfigured environments for data science and deep learning.

Non-interactive jobs like the one about to be shown run as scripts without user intervention.

Step-by-Step Instructions:

Example: Submit a non-interactive Job Script for launching a Jupyter Lab web browser session

1. Create a Directory for Your SLURM Job

First, create a directory for storing data related to your SLURM job. This ensures a clean workspace.

```
$ mkdir -p ~/slurmjob; cd ~/slurmjob
```

2. Copy the Sample Job Script

Copy the example job script run_jupyter_shared.sbatch from the /tutorials directory to your working directory.

```
$ cp /tutorials/run_jupyter_shared.sbatch ~/slurmjob/run_jupyter_shared.sbatch
```

3. Create a Shared Directory for Jupyter Lab

Create a shared workspace directory that will be accessible inside your Jupyter Lab container.

```
$ mkdir -p ~/shared_jupyter_dir
```

Tip: After launching the SLURM job, a sample directory with example notebooks will be created at `~/shared_jupyter_dir/examples`.

4. Edit the Job Script to Match Your Needs

Modify the run_jupyter_shared.sbatch script according to your resource and time requirements.

```
# Edit our jobs configuration file:
$ nano ~/slurmjob/run_jupyter_shared.sbatch
```

In the script, make sure to adjust the following fields based on your job's requirements:

```
#!/bin/bash
#SBATCH --job-name=jupyter_gpu   # Job name
#SBATCH --gres=gpu:1             # Number of GPUs (1 or 2)
#SBATCH --cpus-per-task=4        # Number of CPU cores
#SBATCH --mem=16G                # Memory allocation (e.g., 16GB)
#SBATCH --time=04:00:00          # Maximum time (e.g., 4 hours)

# Define the shared directory path
SHARED_JUPYTER_DIR="$HOME/shared_jupyter_dir"
```

You'll need to review and edit the lines below located at the top of the file according to your needs:

- `#SBATCH --job-name=jupyter_gpu`: Job Name.
- `#SBATCH --gres=gpu:2`: Request two Graphic Processing Units (GPU).
    - Options 1 or 2 or removed for 0.
- `#SBATCH --cpus-per-task=4`: Request four CPU cores.
- `#SBATCH --mem=16G`: Request 16 GB of RAM.
- `#SBATCH --time=04:00:00`: Request 4 hours.
- `SHARED_JUPYTER_DIR="$HOME/shared_jupyter_dir"`
    - Defines what directory will be shared inside Jupyter Lab's session.
    - These files are accessible inside the Jupyter Lab Session.

> **Note:** By default, 80% of the memory allocated to your job (specified using `#SBATCH --mem=<size>`) will be assigned to shared memory (`/dev/shm`).
>
> For example, if you set `#SBATCH --mem=16G`, 12.8G of that memory will be used for shared memory. It is important to configure this line based on the specific memory needs of your job.
>
> Ensure that you allocate enough memory to account for both the shared memory and the processes that will run inside the container.

5. Submit the SLURM Job

Submit the SLURM job using the sbatch command. This will schedule your non-interactive slurm job to run a Jupyter lab container that will be accessible via an interactive web browser connection:

```
$ sbatch ~/slurmjob/run_jupyter_shared.sbatch
```

You will receive a confirmation with the job ID:

```
Submitted batch job 5
```

6. Monitor the Job Output

Why Monitor the Job Output?

After submitting a non-interactive SLURM job to launch a Jupyter Lab session, the job output file contains critical information that you will need to access your Jupyter Lab instance, such as:

- URL: The web address you will use to connect to your Jupyter Lab session.
- Token: A unique token needed to log in to the Jupyter Lab interface.

It is crucial to monitor the job output to retrieve this information.

Common Pitfall: Missing the `cat` Command to Retrieve the URL and Token

After submitting the job, it is **crucial** to monitor the job's output file to retrieve the access URL and token required to log in to your Jupyter Lab session. Without this step, you will not be able to access your session, leading to a failed connection.

You can monitor the job's output by viewing the SLURM job log file:

```
$ cat slurm-<JobID>.out
```

Replace with your actual job ID (displayed after submitting the job).

Important: Missing the `cat` command is a common mistake. If you forget to check the output file, you won't get the URL or token needed to log in to your Jupyter Lab session. Always make sure to check the job's output as soon as the job starts (~10 seconds after submission).

Alternatively, you can view the access information using the following command:

```
$ cat access_jupyter.log
```

By closely monitoring the job output, you will be able to access your session and troubleshoot any issues that may arise, such as missing tokens or URLs.

We'll use the **later** portion of the job output file contents to obtain our **access URL** and **token**. This output takes ~10 secs after the container starts to parse this log file to obtain the early stated access token produced in the log file.

Alternatively, you can view the access_jupyter.log file for the URL and token:

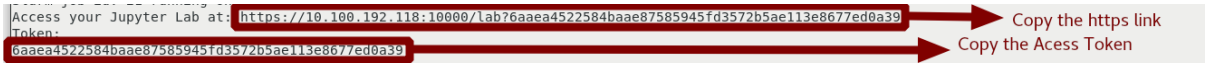
```
# Alternatively, you can just output the access url and token
$ cat access_jupyter.log
Slurm job id: 5 running on 10.100.192.118 port 10000
Access your Jupyter Lab at: https://10.100.192.118:10000/lab?
504dc9073e8fef293ab69a99908c4b9ef6b5fda9c155f08b
Token:
504dc9073e8fef293ab69a99908c4b9ef6b5fda9c155f08b
```

7. Access Jupyter Lab in Your Browser

Once you have the URL and token, follow these steps to access Jupyter Lab:

1. Copy the URL from the log output and paste it into your web browser.
2. Copy the access token from the terminal and paste it into the token prompt in the web browser.
3. Click Log in.

Here we can copy required fields from the prior terminal command's `$ cat access_jupyter.log` output:



1. Copy the **URL** from the terminals job output and paste it into a web browser.
2. Copy your **access token** from the terminal and paste it into the token prompt then select **`Log in`**.

8. Accept the Self-Signed SSL Certificate

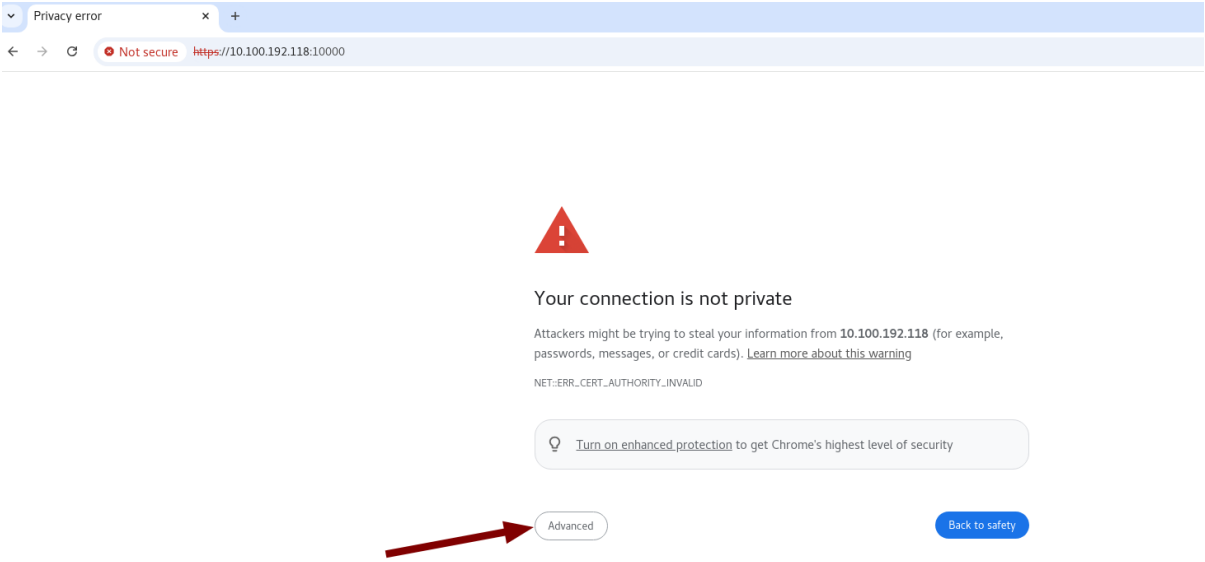
Accept the risk of a self-signed SSL certificate in popular web browsers

Since the Jupyter Lab session runs on a private network, you will encounter a self-signed SSL certificate warning. Accept this in your browser:

- Google Chrome: Click "Advanced" and then "Proceed to [website address]."
- Mozilla Firefox: Click "Advanced" and then "Accept Risk and Continue."
- Microsoft Edge: Click "Advanced" and then "Proceed to [website address]."
- Safari (Mac): Click "Show Details," then "Trust," and select the level of trust (e.g., "Always Trust").

Paste the **URL** into a web browser address bar while connected to the campus network onsite or via the VPN.

Google Chrome



a. Copy the **URL** from the terminals job output and paste it into a web browser.
b. Click through the warning: You will see a warning message indicating that the site's security certificate is invalid or untrusted. Click on "Advanced" and then "Proceed to [website address]" to continue.
c. Copy your **access token** from the terminal and paste it into the token prompt then select **`Log in`**.

Mozilla Firefox

- Navigate to the site: Open the website with the self-signed certificate.
- Click through the warning: You will see a warning message indicating that the site's security certificate is invalid or untrusted. Click on "Advanced" and then "Accept Risk and Continue".

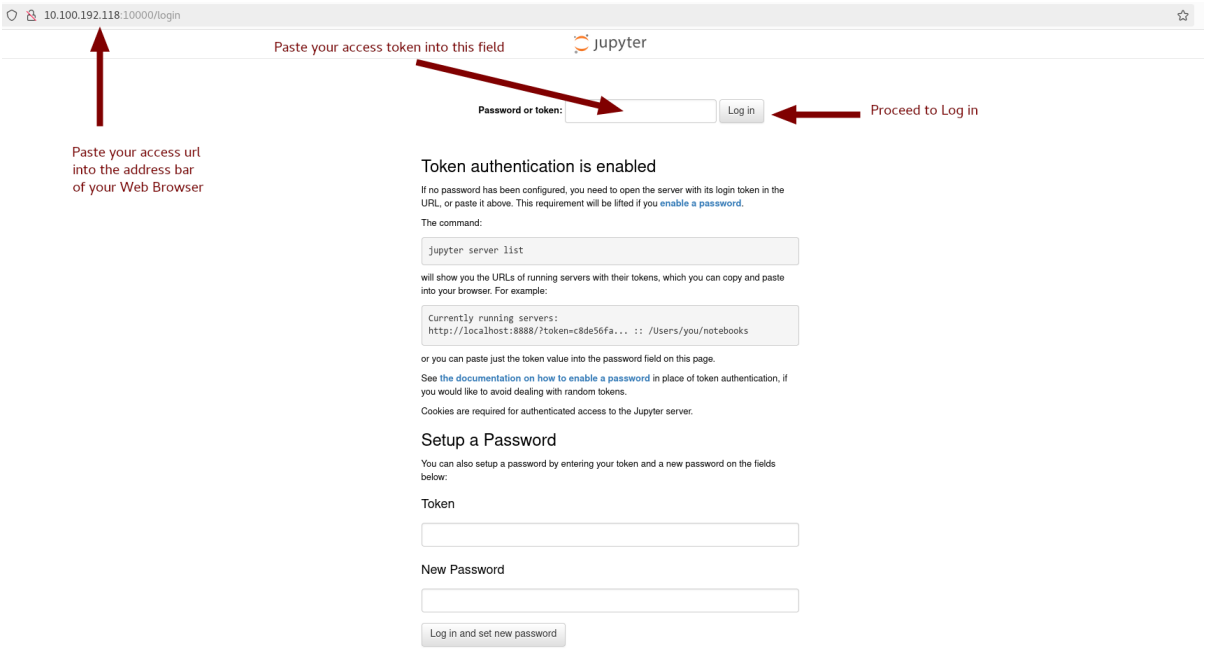
Microsoft Edge

- Navigate to the site: Open the website with the self-signed certificate.
- Click through the warning: You will see a warning message indicating that the site's security certificate is invalid or untrusted. Click on "Advanced" and then "Proceed to [website address]".

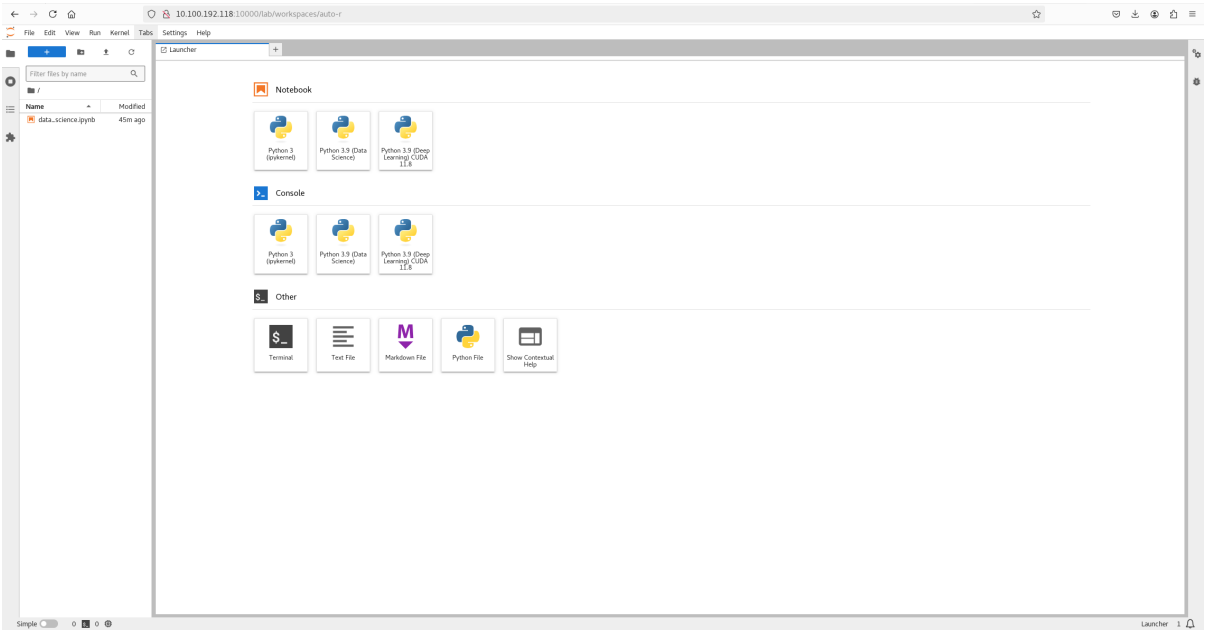
Safari (Mac)

- Navigate to the site: Open the website with the self-signed certificate.
- Click through the warning: You will see a warning message indicating that the site's security certificate is invalid or untrusted. Click on "Show Details". Trust the certificate: In the details view, click on "Trust" and select the level of trust you want to assign to the certificate (e.g., "Always Trust").

Launch a Web Browser on your remote PC connected to SIU's campus area network



1. Paste the **URL** into a web browser address bar while connected to the campus network onsite or via the VPN.
2. Accept the Self-Signed SSL Certificate. Typically, click "Advanced" and then "Accept Risk and Continue".
3. Paste your **access token** into the token prompt then select **`Log in`**.



View of Jupyter Lab after login using the access URL and token from slurm scheduler.

9. Verify the Jupyter Lab Session

Once logged in, you should see the Jupyter Lab interface, where you can start working in your containerized environment.

Notes:

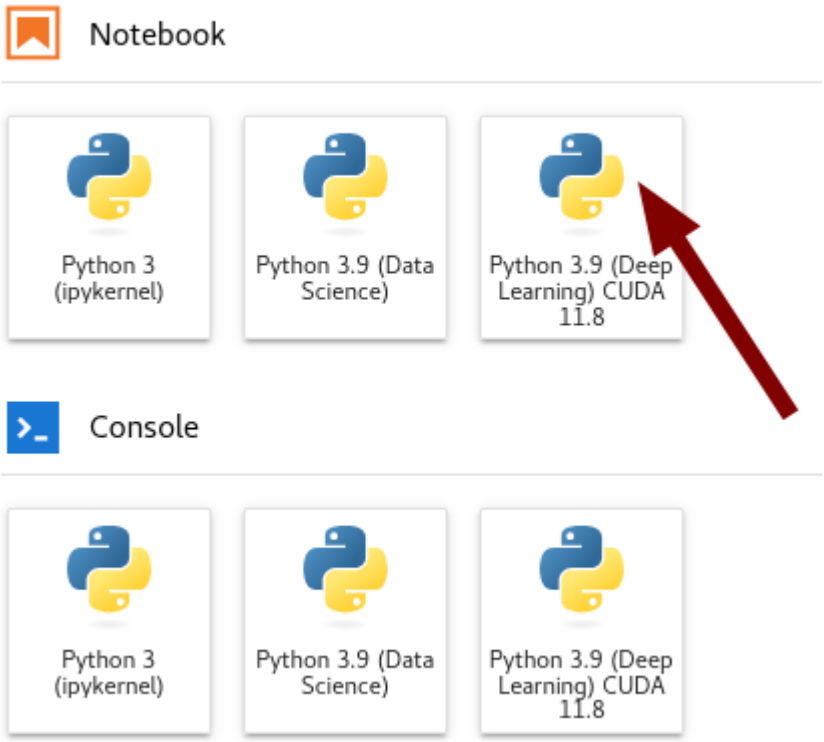
- Shared Directory: Files inside your Jupyter Lab workspace are stored in the ~/shared_jupyter_dir on the server, allowing for persistent file storage. This persistent file storage allows data storaed in the ~/shared_jupyter_dir to be accessible across different slurm jobs persistently.

- Token Expiry: Be mindful that the access token might expire, requiring a new session to be started if needed.

Using Jupyter Lab Web Browser

In the future, we'll produce a separate documentation file regarding instructions on using Jupyter Lab inside your Web Browser. For now, please see below is some basic use case information:

On the main launcher screen, you have the option to create a new notebook with one of the two available environments:



1. Python 3.9 (Data Science)

This environment (`env_ds`) is tailored for data science tasks and includes the following packages:

- `pandas v2.2.3` - Data manipulation and analysis (channel: conda-forge)
- `scikit-learn 1.5.2` - Machine learning tools (channel: conda-forge)
- `matplotlib v3.9.2` - Plotting and visualization (channel: conda-forge)
- `seaborn v0.13.2` - Statistical data visualization (channel: conda-forge)
- `ipykernel` - Provides the Python kernel for Jupyter

You can start this environment by selecting the **Python 3.9 (Data Science)** option from the Notebook section.

2. Python 3.9 (Deep Learning) CUDA 11.8

This environment (`env_dl`) is optimized for deep learning tasks and leverages the GPU capabilities of the server. It includes:

- `pytorch v2.4.1` & `pytorch-cuda=11.8` - Deep learning framework for GPU (channel: pytorch)
- `tensorflow v2.14.0` - Deep learning framework with cuda 11.8 gpu support (channel: conda-forge)

- `torchvision v0.19.1` & `torchaudio v2.4.1` - Computer vision and audio tools (channel: pytorch)
- `pandas v2.2.3` - Data manipulation and analysis (channel: conda-forge)
- `cudnn 8.8` - CudNN 8.8
- `cudatoolkit 11.8` - Cuda ToolKit 11.8
- `ipykernel` - Provides the Python kernel for Jupyter

To use this environment, select the **Python 3.9 (Deep Learning) CUDA 11.8** option from the Notebook section.

Using Consoles

You can also launch a new console, which provides a command-line interface to the selected environment. From the launcher screen, select the appropriate option to open:

- **Python 3 Console (Base)**: This runs from the base environment that only includes JupyterLab. This is a minimal setup suitable for general scripting and customization.
- **Python 3.9 (Data Science) Console**: This opens a console in the data science environment (`env_ds`).
- **Python 3.9 (Deep Learning) CUDA 11.8 Console**: This opens a console in the deep learning environment (`env_dl`).

If you wish to install additional packages, you can do so from any console window by activating the appropriate environment with `source /opt/conda/bin/activate my_env` (replace `env_name` with `env_ds` or `env_dl`) and then using `conda install package_name`.

Creating Your Own Conda Environment

To avoid interfering with the existing environments, you can create your own conda environment for experimentation:

1. **Open a Terminal**: Click the **Terminal** icon under the **Other** section on the main launcher.

```
# Option $_ Other: Terminal
root@4dcbecbeb0f5:/workspace# conda env list
# conda environments:
#
base                  /opt/conda
env_dl                /opt/conda/envs/env_dl
env_ds                /opt/conda/envs/env_ds

root@4dcbecbeb0f5:/workspace#
```

2. **Create a New Environment**:

```
$ conda create -n my_env python=3.9
```

Please note: The JupyterLab session needs to be refreshed after registering a new conda environment with the kernel to make the environment appear as an option.

3. **Activate Your Environment**:

```
#conda activate my_env
$ source /opt/conda/bin/activate my_env
```

4. **Install Packages**:

```
(my_env)$ conda install ipykernel
```

5. **Register the Environment with Jupyter**:

```
(my_env)$ python -m ipykernel install --user --name=my_env --display-name
'Python 3.9 my_env'
```

6. **Reload Notebook from Disk**:

   Select **File** from the menu bar then select **Reload from Disk** for the new environment to be displayed.

This will make your new environment appear as an option in the JupyterLab launcher.

Using postStart.sh for Custom Conda Environments

The **/tutorials/postStart.sh** script allows users to automatically create temporary Conda environments for their SLURM jobs. This is useful when you need a custom environment that only lasts for the duration of the SLURM job and is discarded afterward. Since the container is removed at the end of each job, any environments created during the job will not persist beyond the job's lifecycle.

How to Use postStart.sh in Your SLURM Job

1. **Prepare the postStart.sh Script:**

   The `postStart.sh` script, located in `/tutorials/`, creates and configures a custom Conda environment if it does not already exist. This script is executed after the container is launched but before Jupyter Lab starts.

   You can copy the script to your SLURM job directory:

```
$ cp /tutorials/postStart.sh ~/slurmjob/postStart.sh
```

Your SLURM job script (e.g., run_jupyter_shared.sbatch) will automatically include the postStart.sh script, if the postStart.sh file exists and run it as part of the Podman container startup process.

Please note: including a postStart.sh will cause delays in the initial JupyterLab launching.

2. **Customize the Conda Environment with postStart.sh:**

The postStart.sh script is configured to create a new Conda environment and install any packages you need. By default, it creates an environment named new_custom_env and installs pandas and scikit-learn. Here is an example of what the script looks like:

```
#!/bin/bash
# This is postStart.sh

# Load Conda environment scripts
source /opt/conda/etc/profile.d/conda.sh

# Create a new Conda environment if it doesn't exist
if ! conda info --envs | grep -q "new_custom_env"; then
    conda create -n new_custom_env python=3.9 -y
    conda install -n new_custom_env pandas scikit-learn -c conda-forge
    conda clean -ya

    # Register environment in Jupyter
    conda activate new_custom_env
    python -m ipykernel install --user --name=new_custom_env --display-name
"Python 3.9 (New Env)"
    conda deactivate
fi
```

3. **Submit Your SLURM Job:**

After editing the sbatch file to include the logic for the postStart.sh script, you can submit the job using the following command:

```
$ sbatch ~/slurmjob/run_jupyter_shared.sbatch
```

4. **Monitor Job Output:**

Check the SLURM job output log to ensure that the custom Conda environment is created correctly. Any errors or issues will be logged in the jupyter_${SLURM_JOB_ID}.log file located in the /workspace/slurmlogs/ directory.

```
$ cat /workspace/slurmlogs/jupyter_${SLURM_JOB_ID}.log
```

Advantages of Using postStart.sh:

- Ephemeral Environments: Any environment created during the job will be discarded when the container is removed, ensuring a clean workspace for the next job.
- Custom Environments: Tailor the environment to your needs by modifying postStart.sh to include any packages or configurations specific to your workflow.

- Simplicity: Automatically set up and use custom environments without manual intervention every time the job runs.

By integrating the postStart.sh script into your SLURM workflow, you can streamline the setup of custom Conda environments, ensuring a consistent and efficient environment for your data science and deep learning tasks.

Workspace and File Management

The files within the container workspace are mounted to the **$HOME/shared_jupyter_dir** on the server, allowing for persistent file storage and easy access. All files you create or upload within the workspace in JupyterLab will be accessible from this path on the server.

Additional Notes for Users

- Base Environment: The base environment has minimal packages installed, providing flexibility for users to create custom setups.
- Optimized Environments: Both env_ds and env_dl environments are designed for their respective purposes (data science and deep learning), ensuring an optimized package setup.
- Resource Monitoring: It's recommended to monitor your resource usage to ensure efficient utilization of the server's capabilities.

Interactive Jobs

Interactive jobs are useful when you need a shell session with allocated resources.

Example: Request an Interactive Shell

```
$ srun --pty --gres=gpu:1 --cpus-per-task=4 --mem=16G --time=04:00:00 bash
```

The only lines we're interested in modifying are listed below:

- `--pty`: Allocate a pseudo-terminal.
- `--gres=gpu:1`: Request one GPU.
- `--cpus-per-task=4`: Request four CPU cores.
- `--mem=16G`: Request 16 GB of RAM.
- `--time=04:00:00`: Request 4 hours.

Provides an interactive bash session for four hours with the above resourses.

> **Please note: This node has multiple GPUs assigned to it and you'll need to check which GPU was assigned for your job by slurm. Running slurm jobs on a GPU not assigned to you will be detected by scripts and your job will be cancelled as well as your account suspeneded.**

To avoid this you can check which GPU on the node to utilize prior to starting any GPU computation. When in doubt you can always request the node's maximum GPUs while testing utilizing running your code on the assigned GPU or reach out to our system administrative team for advise.

Obtain the GPU assigned to use for your interactive shell

```
$ cat /tmp/cuda_visible_devices_${SLURM_JOB_ID}.txt
# Set CUDA_VISIBLE_DEVICES to your GPU IDs assigned to slurm job
$ export CUDA_VISIBLE_DEVICES=$(cat /tmp/cuda_visible_devices_${SLURM_JOB_ID}.txt)
```

This will set CUDA_VISIBLE_DEVICES environment variable for your interactive shell.

For example,

```
$ cat /tmp/cuda_visible_devices_190.txt
0,1
$ export CUDA_VISIBLE_DEVICES=$(cat /tmp/cuda_visible_devices_190.txt)
```

The above would configure CUDA_VISIBLE_DEVICES=0,1 allowing you to utilize GPU index 0 and 1. It is very important that you only run interactive GPU jobs on the GPU assigned to you. If you run processes on the misappropriate GPU it will cause other users slurm jobs to fail. Scripts have been setup to detect this activity and accounts will be suspended if this occurs.

Please reach out to our system administrative team if you have any doubts about how to utilize this with your interactive jobs.

Non-Interactive Jobs

Non-interactive jobs run scripts without user intervention.

Example: Submit a Job Script

First, create a job script or use an existing one from `/tutorials`.

```
$ sbatch /tutorials/run_jupyter_shared.sbatch
```

Job Script Breakdown

Here's what the `run_jupyter_shared.sbatch` script key points might look like:

```
#!/bin/bash
#SBATCH --job-name=jupyter_gpu
#SBATCH --gres=gpu:1
#SBATCH --cpus-per-task=4
#SBATCH --mem=16G
#SBATCH --time=04:00:00

# Define the shared directory path
SHARED_JUPYTER_DIR="$HOME/shared_jupyter_dir"
```

```
# Allocate /etc/subuid and /etc/subgid for end-user for podman's rootless
container mapping
sudo /usr/local/sbin/allocate_uid_gid.sh

echo "Starting Jupyter Notebook on port $PORT"

# Run the container with GPU support and bind the port
podman run --rm \
--device nvidia.com/gpu=all \
--security-opt=label=disable \
-p $PORT:8888 \
-v $SHARED_JUPYTER_DIR:/workspace \
-e SHELL=/bin/bash \
my_jupyter_image:latest
jupyter lab --ip=0.0.0.0 --port=8888 --no-browser --allow-root
```

- `#SBATCH` lines specify job parameters.
- `#SBATCH --gres=gpu:1`: Request one GPU (maximum 2).
- `#SBATCH --cpus-per-task=4`: Request four CPU cores per ntask.
- `#SBATCH --mem=16G`: Request 16 GB of RAM.
- `#SBATCH --time=04:00:00`: Request 4 hours
- `SHARED_JUPYTER_DIR="$HOME/shared_jupyter_dir"`
    - Defines what directory will be shared inside Jupyter Lab's session.
    - These files are accessible inside the Jupyter Lab Session.
- The `podman` command runs a containerized Jupyter Lab instance.

> **IMPORTANT:** Our examples show utilizing a preconfigured /tutorials/run_jupyter_shared.sbatch. However, if you were to create your own custom non-interactive .sbatch job outside the preferred /tutorials/run_jupyter_shared.sbatch then you'll need to take into consideration this node has multiple GPUs and you're only permitted running jobs and the one assigned to you by slurm. Slurm by default doesn't restrict per node gpu access so if you decide to ever use a customized .sbatch different from the provided run_juypter_shared.sbatch then you'll need to obtain the proper gpu on the node for any computation you run on the GPU. You can deteremine which GPU was assigned to your job by adding the following to your customize .sbatch file CUDA_VISIBLE_DEVICES=$(cat /tmp/cuda_visible_devices_${SLURM_JOB_ID}.txt) this is configured automatically for you in the preferred run_juypter_shared.sbatch example. However, if you choose to write a custom .sbatch please note that running slurm jobs on a GPU on the node that is not assigned to you will be detected by scripts and your job will be cancelled as well as your account suspeneded to prevent you from further interrupting other users jobs. Please consult with our system administration team when customize job submissions if you have any doubts.

Monitoring Jobs

Use `squeue` to view the job queue.

```
$ squeue
```

- `-u your_username`: Show jobs for your user account.

Sample output:

```
$ squeue
JOBID     NAME         USER ST       TIME NODES CPUS MIN_MEMO TRES_PER_NODE
STATE       START_TIME TIME_LIMIT
   81 jupyter_ siu851234567 PD       0:00     1    4      32G    gres:gpu:1
PENDING 2024-10-22T14:48    8:00:00
   82 jupyter_ siu851234567 PD       0:00     1    4      32G    gres:gpu:1
PENDING 2024-10-22T22:48    8:00:00
   79 jupyter_ siu851234568  R   15:37:57     1    4      64G    gres:gpu:1
RUNNING 2024-10-21T22:55 7-00:00:00
   80 multitas siu856562106  R    3:45:39     1    4      16G    gres:gpu:1
RUNNING 2024-10-22T10:48    4:00:00
```

In this example, the output provides information about the current SLURM job statuses, including Job ID, Job Name, User, Job State, Elapsed Time, Number of Nodes, CPU count, Memory requested, and Generic Resources (like GPUs) allocated to the job.

Field Explanations:

- **JOBID**: Unique identifier for each job.
- **NAME**: The user-defined name of the job, such as jupyter_ or multitas.
- **USER**: The user who submitted the job (e.g., siu851234567).
- **ST**: Job state. Common values include:
  - **R**: Running
  - **PD**: Pending
  - **CG**: Completing
- **TIME**: The time the job has been running or the time since submission if it's pending (format: HH:MM:SS).
- **NODES**: The number of nodes allocated to the job.
- **CPUS**: The number of CPUs requested for the job.
- **MIN_MEMO**: The minimum memory requested for the job.
- **TRES_PER_NODE**: The resources allocated per node, such as gres:gpu:1 indicating 1 GPU.
- **STATE**: The current state of the job. This could be RUNNING or PENDING with a reason.
- **START_TIME**: The estimated start time for pending jobs or the actual start time for running jobs.
- **TIME_LIMIT**: The maximum time allowed for the job, formatted as D-HH:MM:SS.

Example Breakdown:

- **Job ID 79:**
  - Running with the name jupyter_.
  - Submitted by user siu851234568.
  - Has been running for 15 hours, 37 minutes, and 57 seconds on 1 node.
  - Allocated 4 CPUs, 64 GB of memory, and 1 GPU (gres:gpu:1).
  - Estimated completion time limit is 7 days (7-00:00:00).

- **Job ID 81:**
    - Pending with the name jupyter_.
    - Submitted by user siu851234567.
    - Allocated 4 CPUs, 32 GB of memory, and 1 GPU.
    - Expected to start at 2024-10-22T14:48 and has a time limit of 8 hours (8:00:00).

> **Note: squeue has been configured with an alias to output in the following format to show your complete user id.**
>
> $ squeue -o "%.5i %.8j %.12u %.2t %.10M %.5D %.4C %.8m %.13b %.10T %.16S %.10l"

Explanation of the Format:

- %.5i: Job ID (width 5)
- %.8j: Job name (width 8, adjust as needed)
- %.12u: User name (width 12 to prevent truncation)
- %.2t: Job state (width 2)
- %.10M: Time (width 10)
- %.5D: Number of nodes (width 5)
- %.8m: Amount of memory requested (width 8, in MB/GB)
- %.13b: Generic resources requested, such as GPUs (width 13, e.g., gpu:2)
- %.10T: Job type (width 10, batch or interactive)
- %.16S: Start time of the job or estimated start time (width 16, in YYYY-MM-DD HH:MM:SS)
- %.10l: Time limit requested for the job (width 10, in D-HH:MM:SS format)

Cancelling Jobs

To cancel a job, use `scancel`. You'll want to do this once your job is completed prior to your time allocation to allow other users access to the server as soon as possible.

```
$ scancel JOBID
```

Monitoring Resource Utilization

In addition to monitoring the status of your jobs using `squeue`, it is important to track actual resource utilization to optimize future job submissions. SLURM provides several tools for this, including `sacct` and `scontrol show job`. This section will explain how to use these commands to monitor CPU, memory, and GPU usage for your jobs.

1. Monitoring Resource Usage with sacct

The `sacct` command provides detailed information about completed jobs, including CPU time, memory usage, and other resources. This can help you understand how efficiently your job used the allocated resources.

Basic Usage:

To view the resource utilization for a specific job, run:

```
$ sacct -j <JobID> --format=JobID,JobName%20,Elapsed,MaxRSS,TotalCPU,State
```

Replace `JOBID` with the actual job ID from `squeue`.

- JobID: The unique identifier for the job.
- JobName%20: The name of the job (with 20-character width for alignment).
- Elapsed: The time that the job ran.
- MaxRSS: The maximum memory used by the job (Resident Set Size).
- TotalCPU: The maximum CPU time used.
- State: The job's final state (e.g., COMPLETED, FAILED).

Example:

```
$ sacct -j 12345 --format=JobID,JobName%20,Elapsed,MaxRSS,TotalCPU,State
JobID    JobName              Elapsed      MaxRSS   TotalCPU      State
12345    my_job_name          02:30:00     2.5G     00:10:00    COMPLETED
```

In this example, you can see that the job took 2 hours and 30 minutes, used a maximum of 2.5 GB of memory, and consumed 10 minutes of CPU time. This information can help you assess whether you requested too much or too little memory and CPU.

Monitoring GPU Usage:

If your job used GPUs, you can monitor GPU usage as well:

```
$ sacct -j <JobID> --
format=JobID,JobName%20,Elapsed,MaxRSS,TotalCPU,AllocTRES%35,State
```

AllocTRES: Shows the number and type of GPUs allocated (e.g., gpu:1).

Example:

```
$ sacct -j 12345 --
format=JobID,JobName%20,Elapsed,MaxRSS,TotalCPU,AllocTRES%35,State
JobID                    JobName    Elapsed     MaxRSS   TotalCPU
AllocTRES      State
------------ -------------------- ---------- ---------- ---------- ---------------
-------------------- ----------
103                 jupyter_ADM_2   02:32:27              00:00:00
billing=8,cpu=8,mem=160G,node=1    RUNN
```

Monitoring Memory usage:

Memory utilization is crucial when running jobs that require shared memory, such as those involving containers or large datasets. SLURM jobs allocate memory with the #SBATCH --mem directive. 80% of the memory specified by --mem is reserved for shared memory (/dev/shm). The remaining 20% is used for the rest of your job's processes.

Important: Configure Memory Appropriately

It is critical to configure the --mem line in your job scripts carefully, as insufficient memory allocation can lead to job failures due to lack of shared memory.

For example:

```
#SBATCH --mem=16G               # Adjust this line according to your job's memory
requirements
```

In this case, 12.8G will be allocated for shared memory, while 3.2G will be available for other processes.

> Tip: Monitor memory usage closely using sacct or scontrol to ensure your jobs are not running out of shared memory. If you notice high memory usage or job failures due to memory constraints, adjust the --mem value in your job script accordingly.

Determining Available Memory and Setting Maximum Limits

When submitting jobs, it's crucial to understand how much memory is available on the system and to ensure that your job doesn't request more memory than the system can allocate. Proper memory configuration is key to avoiding job failures or resource overuse.

Checking Available Memory on the System

Before submitting a job, you can check the total and available memory on the compute nodes by using the following command:

```
$ free -h
```

This command will show an output similar to:

```
$ free -h
               total        used        free      shared  buff/cache   available
Mem:           502Gi        46Gi        52Gi        13Gi       420Gi       456Gi
Swap:          4.0Gi       0.0Ki       4.0Gi
```

- total: The total memory installed on the node.
- used: The memory currently in use.
- free: The memory currently available for new jobs.

- shared: Memory used by shared processes.
- available: The memory available to be allocated for new processes or jobs.

How to Determine the Maximum Memory for Your Job

To determine the maximum memory your job should request, you need to consider several factors:

- **Available memory on the node**: As shown in the output of free -h, focus on the available memory to decide how much can be used by your job without risking running out of memory.
- **Multiple jobs running**: If multiple jobs are running on the node, ensure that your job doesn't consume too much memory, which might impact other jobs or cause the node to run out of memory.
- **Percentage of total memory**: As a best practice, do not request more than 80-90% of the total available memory on a node for your job, especially if other jobs might be running.

Example:

If the available memory is **502GiB**, and you plan to submit a job that requires heavy use of shared memory (e.g., a machine learning model), you should configure your job to use a portion of that memory:

```
#SBATCH --mem=401G               # Allocates 401G, with 321.2G used for /dev/shm
(shared memory)
```

In this example:

- **401G** is the total memory requested.
- **80%** of this (321.2G) will be used for shared memory (/dev/shm).
- The remaining **20%** (80.32G) will be used for other processes.

> **Important**: Avoid requesting more than **90% of the available memory**. Leaving some memory free ensures that the system can handle background processes and helps prevent job crashes due to lack of memory.

2. Monitoring Job Details with scontrol show job

For real-time monitoring of job details, `scontrol show job` provides detailed information about the job while it is running. It also displays resource usage after the job completes.

Basic Usage

```
$ scontrol show job <JobID>
```

This will output detailed information about the job's configuration and current status, including memory usage, CPU utilization, job start time, and more.

Key Fields to Look For:

- **JobState**: The current state of the job (e.g., RUNNING, COMPLETED).
- **ReqMem**: The amount of memory requested.

- **AveCPU**: Average CPU usage during the job's execution.
- **TRES**: Tracks the number of CPUs, GPUs, memory, and other resources allocated.

Example:

```
$ scontrol show job 12345
JobId=12345 JobName=my_job_name
   JobState=RUNNING Reason=None Dependency=(null)
   ReqMem=16G AllocMem=16G
   AveCPU=00:30:00
   TRES=cpu=4,mem=16G,gres/gpu=1
   StartTime=2024-10-08T10:00:00 EndTime=2024-10-08T14:00:00
   Nodes=node01
```

In this example, you can see that the job has requested 16 GB of memory and is running on one GPU and four CPUs. The average CPU usage is displayed as well, helping you evaluate if the requested resources match the actual usage.

3. Tips for Optimizing Resource Requests

Monitoring actual resource usage is crucial for optimizing future job submissions. Here are some tips:

- **Memory**: If MaxRSS is consistently lower than the memory requested (ReqMem), you can request less memory in future jobs. Conversely, if your job approaches or exceeds MaxRSS, you may need to request more memory.
- **CPU**: Check the AveCPU to see if the CPUs you requested are fully utilized. If the AveCPU is much lower than the requested number of CPUs, you can reduce the number of CPUs for future jobs.
- **GPU**: If using GPUs, check the AllocTRES field to ensure the number of GPUs requested matches your job's actual GPU usage.

4. Visualizing Job Performance (Optional)

For users who prefer visual feedback, the seff command (if available) provides an easier way to view job efficiency:

```
$ seff <JobID>
```

Example Output:

```
$ seff 12345
Job ID: 12345
Job Name: my_job_name
Cluster: your_cluster
User/Group: user/group
State: COMPLETED (exit code 0)
Cores: 4
```

```
CPU Utilized: 00:10:00
CPU Efficiency: 12.50% of 01:20:00 core-walltime
Memory Utilized: 2.5 GB
Memory Efficiency: 15.63% of 16.00 GB
```

This command provides a quick overview of your job's efficiency, showing how much of the requested memory and CPU were actually used, making it easier to adjust resource requests in the future.

By understanding how to monitor and analyze resource usage for your jobs, you can optimize your SLURM job submissions to avoid wasting resources, reduce job queue times, and ensure more efficient computations.

This section provides a detailed breakdown of how to monitor job performance and resource utilization, making it easier for users to understand their SLURM jobs and optimize resource requests accordingly.

5. Monitor GPU Usage

1. Enable GPU Monitoring

To track the GPU usage during your job, you can enable GPU memory logging. This is particularly useful for research purposes or optimizing resource allocation for future jobs.

In the job script (`run_jupyter_gpumonitoring_shared.sbatch`) GPU memory logging is enabled by default, but you will find options to enable or disable GPU memory logging.

To enable GPU memory logging, set the following flag to true:

```
ENABLE_GPU_LOGGING=true
```

To disable GPU memory logging, set the flag to false:

```
ENABLE_GPU_LOGGING=false
```

You can also choose whether to log each GPU's memory usage in separate files or to aggregate the logs into a single file:

Choose Separate or Aggregated Logging:

- To log each GPU's memory usage in separate files, set:

```
SEPARATE_LOGS=true
```

- To log all GPUs' memory usage into a single file, set:

```
SEPARATE_LOGS=false
```

2. Why Enable GPU Monitoring?

Enabling GPU memory logging is highly recommended for all users running GPU-intensive tasks, such as deep learning, because it provides invaluable insights into resource usage:

- **Resource optimization**: Detailed memory usage logs allow you to optimize future job submissions by requesting the right amount of GPU resources. This reduces the risk of underutilizing or overloading GPUs, helping you avoid resource wastage and job delays.
- **Performance insight**: GPU logs give you a clear understanding of how well your job is utilizing GPU resources. This can help identify performance bottlenecks and guide adjustments for more efficient execution of data science and deep learning tasks.
- **Documentation for research**: For research and reproducibility purposes, GPU memory logs provide critical information about how your job performed in terms of resource consumption. This can be useful for research documentation and sharing with collaborators.

Without this information, you risk over-allocating resources and missing optimization opportunities. Make sure to enable GPU monitoring if you're aiming for peak performance and efficient resource use in your computational jobs.

3. View GPU Logs

After the job is complete, you can review the GPU memory usage logs stored in your job directory:

For separate GPU logs:

```
$ cat gpu_usage_log_gpu0-${SLURM_JOB_ID}.out
$ cat gpu_usage_log_gpu1-${SLURM_JOB_ID}.out
```

For aggregated logs:

```
$ cat gpu_usage_log-${SLURM_JOB_ID}.out
```

To view a summary of GPU memory usage (maximum and average), you can review the summary log:

```
$ cat gpu_summary_slurm-${SLURM_JOB_ID}.out
```

Example output:

```
GPU 0 Maximum memory used: 16280 MiB
GPU 0 Average memory used: 8100 MiB
```

```
GPU 1 Maximum memory used: 16280 MiB
GPU 1 Average memory used: 8200 MiB
```

This summary provides key metrics such as maximum and average memory used for each GPU.

Example Data Science and Machine Learning Notebooks

By default the /tutorials/run_jupyter_gpumonitoring_shared.sbatch slurm job will copy example jupyter notebooks to your ~/shared_jupyter_dir/examples which are shared to your containerized JupyterLab web session.

You can access these examples inside JupyterLab's web session /exampels directory:

```
$ tree
├── examples
│   ├── data_science
│   │   ├── matplotlib_visualizations.ipynb
│   │   ├── pandas_basics.ipynb
│   │   └── scikit_learn_intro.ipynb
│   ├── data_science.ipynb
│   ├── deep_learning
│   │   ├── periodic_checkpointing
│   │   │   ├── PyTorchSaveBasedOnTime.ipynb
│   │   │   ├── PyTorchSaveEachEpoch.ipynb
│   │   │   ├── TensorFlowSaveBasedOnTime.ipynb
│   │   │   └── TensorFlowSaveEachEpoch.ipynb
│   │   ├── pytorch_gpu_example.ipynb
│   │   ├── tensorflow_gpu_example.ipynb
│   │   ├── torchvision_image_classification.ipynb
│   │   └── torchvision_image_classificationv2.ipynb
│   ├── deep_learning.ipynb
│   └── scripts
│       ├── script_name.sh
│       └── slurm_remaining_time.sh
```

Changing Job Priorities

You can adjust the priority of your pending jobs.

```
$ scontrol update jobid=JOBID priority=PRIORITY_VALUE
```

- `JOBID`: The ID of your job.
- `PRIORITY_VALUE`: An integer; higher values increase priority.

Note: Regular users have limited ability to change priorities. Contact your administrator (mabarkdoll@siu.edu) if necessary. Please plan accordingly as responses and adjustments take place during working hours and are subject to availability.

Using Screen Sessions

What is Screen?

`screen` is a terminal multiplexer that allows you to run multiple terminal sessions within a single window or SSH session. It provides a way to start and manage multiple shells and processes simultaneously. You can detach from these sessions (leaving them running in the background) and reattach later, allowing you to maintain long-running processes or work across multiple sessions without losing progress when you disconnect.

In this example, screen is used to run an interactive srun session on a SLURM-managed compute node. Running an interactive job via srun often involves waiting for resource allocation, performing computations, or executing commands that may take a long time to complete.

By using screen, you can:

- **Run the interactive session in the background:** You can start the session and then "detach" from it, allowing the `srun` command and any related processes to continue running without requiring you to stay logged into the terminal.
- **Reattach to the session at any time:** If you need to check the progress or interact with the session again, you can "reattach" to it easily.
- **Prevent interruptions:** If your SSH connection drops or you need to close your terminal, screen ensures that the interactive session continues to run on the server without being interrupted.

In summary, `screen` allows you to start an interactive `srun` session on a SLURM node and have the flexibility to disconnect and reconnect to it as needed, providing robustness for long-running tasks or instances when you may need to step away from your terminal.

You can reattach to an existing `screen` session to obtain a bash prompt during your scheduling computation time allotment.

Creating a Screen Session

Before starting your job, you can create a new screen session:

```
$ screen -S my_screen_session
```

This will start a screen session named `my_screen_session`.

Run an Interactive `srun`

Within the screen session, run the interactive `srun` command to allocate resources on a SLURM node. Here's an example:

```
$ srun --pty --partition=debug --ntasks=1 --cpus-per-task=4 --mem=8G bash
```

- Adjust the resources (--ntasks, --cpus-per-task, --mem, --gres) based on your needs.

- When using --gres please refer to the earlier interactive job section and only using GPUs assigned to your job.

Wait for SLURM to allocate resources. Once a node is allocated, you will be dropped into an interactive shell on the compute node.

Detaching from a Screen Session

To detach from the session without terminating it, press:

```
Ctrl + A, then D
```

Reattaching to a Screen Session

To reattach to your screen session:

```
$ screen -r my_screen_session
```

If you have multiple screen sessions, list them with:

```
$ screen -ls
```

And then attach to the desired session using its ID:

```
$ screen -r SESSION_ID
```

End the Interactive `srun` Session

To end the interactive srun session, simply type:

```
$ exit
```

Finally, make sure that you terminate your screen session with:

```
$ exit
```

This will close the screen session completely and cleanup unused resources on the server!

Monitoring Your Job in a Screen Session

If you started your job within a screen session using `srun`, you can check if it is running or pending by using:

```
$ squeue
```

If the job is still in the queue, it will show up with its status. If the job has started, you can reattach to the screen session to view its progress.

Scheduling Jobs Within Specific Time Ranges

You can specify when a job becomes eligible to run.

Example: Schedule a Job to Start on Monday at 8 AM

```
sbatch --begin=now+2days run_your_job.sbatch
```

- `--begin=now+2days`: Start the job two days from now.

Alternatively, specify an exact time:

```
sbatch --begin=2023-10-02T08:00:00 run_your_job.sbatch
```

- `--begin=YYYY-MM-DDThh:mm:ss`: Schedule for a specific date and time.

Limit Job to Run Between Specific Hours

Use a job script with a time limit and schedule accordingly.

```
#SBATCH --time=09:00:00  # 9 hours max runtime
```

Submit the job with a specific start time.

```
sbatch --begin=2023-10-02T08:00:00 run_your_job.sbatch
```

Using Tutorial Files in /tutorials

The `/tutorials` directory contains sample slurm job scripts and Dockerfiles. We'll primarily use just the sample slurm job script from this directory when submitting slurm non-interactive sbatch jobs.

Example Files

- **run_jupyter_shared.sbatch**: Submits a Jupyter Lab job.
- **Dockerfile**: Builds a custom container image.

- Please consult with system administrator for custom docker images so that these can be shared from a shared image store location to save on disk space. This allows us to have a master container image for multiple user's jobs.

How to Use

1. Navigate to the `/tutorials` directory.

```
cd /tutorials
```

2. Review the job script.

```
less run_jupyter_shared.sbatch
```

3. Submit the job.

   To see what the following non-interactive slurm job does please see section:

   Podman Jupyter Lab Web Browser Jobs

```
# First copy the .sbatch file to a location within your homedir:
$ mkdir -p ~/myjobdir
$ cd ~/myjobdir/
$ cp /tutorials/run_jupyter_shared.sbatch ~/myjobdir/
# Non-interactive slurm job submission example see:
# Podman Jupyter Lab Web Browser Jobs above for details
$ sbatch run_jupyter_shared.sbatch
```

Additional Resources

- **SLURM Documentation:** https://slurm.schedmd.com/documentation.html
- **Podman Documentation:** https://podman.io/docs

For Support

Feel free to reach out to:

Michael Allen Barkdoll
Computer System Architecture Specialist
mbarkdoll@cs.siu.edu