

A quick reference guide for working with the hawaii cluster. Code prefaced by a `$`-sign is to be interpreted as a shell command. When entering a command the `$` is not to be retyped into the terminal. Text inside `<...>` has to be modified.

## 0.1 Brief System Description

- Head node: `csg-headnode`
  - CPU: AMD EPYC 9224
  - RAM: 125 GiB
- `csg-brook{01,02}`:
  - CPU: AMD EPYC 7351P
  - RAM: 251 GiB
  - GPU: 8x Nvidia RTX 2080Ti 11 GiB
  - Interconnect: Gigabit Ethernet
- `csg-octane{01..08}`:
  - CPU: 2x Intel Xeon E5-2630
  - RAM: 94 GiB
  - Interconnect: Gigabit Ethernet
- `csg-moore`:
  - CPU: 4x Intel Xeon E5-2630
  - RAM: 251 GiB

# 1 ssh

## 1.1 Basic Login

The easiest way (not necessarily the most convenient) to connect to a remote machine is via invocation of the basic `ssh` command from the terminal. Linux, Mac, and BSD users should have `ssh` installed by default. Windows users can use the Windows Subsystem for Linux (WSL) to open a bash prompt or alternatively use the program PuTTY<sup>1</sup>. `ssh` expects a username and a hostname separated by an `@` symbol. The username corresponds to your group number, e.g. `eml03`, `gpu08`.

The cluster headnode is reachable only from the Ziti internal network directly. From the university network you have to use a proxy. For that purpose we have a gate server `zitigate.ziti.uni-heidelberg.de`. Use:

```
$ ssh <username>@zitigate.ziti.uni-heidelberg.de
```

<sup>1</sup><https://www.chiark.greenend.org.uk/~sgtatham/putty/>

to connect to it. After successfully entering your password you should be greeted by a command prompt. To close the session simply type `exit`, `Ctrl-D`, or just close your terminal emulator.

Note that this system is very restrictive and only meant to be used as a jump host. All the compute tasks should be run on the headnode and the cluster compute nodes.

To jump to the cluster headnode run the following on the gate:

```
$ ssh <username>@csg-headnode
```

Always use the head node `csg-headnode` to perform typical auxiliary tasks like compilation. The other nodes: `csg-brook{01,02}`, `csg-octane{01..08}` are reserved for compute tasks.

## 1.2 (Optional) Configuration File

Retyping the above command can become tedious over time. By using an ssh configuration file this can be sped up, however. Under Linux, Mac, and BSD you can edit (or create if it not yet exists) a config file for ssh under `$HOME/.ssh/config`. The file follows a simple syntax (see [here](#) for extensive information). A Basic example is listed below:

Listing 1: Basic ssh configuration file (on your local machine)

```
Host zitigate
    HostName zitigate.ziti.uni-heidelberg.de
    User <username>
Host headnode
    HostName csg-headnode
    User <username>
    ProxyJump zitigate
```

Now you can simply use:

```
$ ssh headnode
```

You will still be prompted for a password though. PuTTY users can just save the connection via the GUI.

## 1.3 (Optional for shell users) Login Without Password

To avoid having to reenter your password every time you log on, you can use an ssh-key for authentication. First create a new ssh-key pair via:

```
$ ssh-keygen -t ed25519
```

This will prompt you for a filename. You can call the key whatever you like. I would suggest naming it something like `eml-lecture` for easy identification. This will create two files: a private key `<your-key-file>` and a public key `<your-key-file>.pub` Next, copy the key to the server's `authorized_keys`. Because `zitigate` runs `rbash`, `exec` calls

and pipes are not allowed. This means you can *not* use the `ssh-copy-id` command to copy your key. Instead do the following, (Listing 2 shows the commands):

1. Copy the contents of your *public key* (ends with `.pub!!`) to your clipboard.
2. Log on to `zitigate` with your password.
3. If it does not exist, create a directory called `~/.ssh`
4. set the correct permissions for that directory (700)
5. create or edit a file called `authorized_keys` inside `~/.ssh` using your preferred editor. (`nano`, `vi`, and `joe` are installed. If you don't know which one to pick, choose `nano`)
6. Copy your public key from the clipboard to the editor and save the file (`<CTRL-X>` in `nano`, then press `y` to save changes if prompted).

Listing 2: Authorize the key

```
$ ssh <username>@zitigate.ziti.uni-heidelberg.de
$ mkdir -pv ~/.ssh
$ chmod 700 ~/.ssh
$ nano ~/.ssh/authorized_keys
--> Copy key to file, close with <CTRL-X>, confirm with <y>
$ chmod 600 ~/.ssh/authorized_keys
```

Make sure you do not overwrite your teammates' keys!

You can instruct `ssh` to use your key automatically by editing your `ssh-config`:

Listing 3: `ssh` config with automatic login


```
Host zitigate
    HostName zitigate.ziti.uni-heidelberg.de
    User <username>
    IdentityFile <path-to-your-keyfile>
Host headnode
    HostName csg-headnode
    User <username>
    IdentityFile <path-to-your-keyfile>
    ProxyJump zitigate
```

You should now be able to automatically log in via:

```
$ ssh headnode
```

## 2 Slurm

You don't have direct access to the cluster's compute nodes. Instead, Slurm is used as a workload manager. Slurm handles your jobs and distributes them among the compute-nodes according to your specification. Slurm takes care of allocating resources and scheduling jobs. For more information on Slurm see [here](#).

There are several so-called partitions: `exercise-eml`, `exercise-hpdc`, `exercise-gpu`,  `exercise-coco`. Please use the partition associated with your lecture. There is also a default job time limit of 12h for EML and 4h for GPU/HPDC/COCO. Usually, if your Job takes longer than the limit something is going wrong.

Use the `sinfo` command to see the available nodes and their status.

### 2.1 (Basic) Running Jobs

You can start jobs by using the `srun` command:

```
$ srun [-p <partition>] [-w <node>] --pty -- <command>
```

Please choose the appropriate partition for your lecture (`exercise-eml`, `exercise-gpu`, `exercise-hpdc`). To request specific nodes, use `-w` which accepts a single node or a comma separated list of node names.

Tip: use `sinfo` and `squeue` to see if a node is currently blocked by another group.

### 2.2 Consumable Resources

Resources like GPUs and CPU cores can be requested using the `--gres` option. Check the message of the day when logging in for more details.

The following will command will allocate a GPU.

```
srun --gres=gpu:1 <...>
```

More CPU cores can also be requested.

```
srun --cpus-per-task 8 <...>
```

### 2.3 Canceling Jobs

If you want to terminate a job, e.g. due to a deadlock or other error, use the `squeue` command to get its job ID. To see only jobs started by you provide your username with the `u` option:

```
$ squeue -u <username>
```

Once you have identified the job's ID use:

```
$ scancel <jobid>
```

to stop the job.

## 2.4 (Advanced) Running Jobs

Slurm supports special shell scripts called sbatch files. These work like normal shell scripts but may contain special instructions for Slurm that are prefaced by `#SBATCH`. The corresponding command is `sbatch`:

```
$ sbatch <batch-file>
```

This is especially handy for starting multiple jobs with different launch parameters with one command. See also [this](#) very good summary of how to use `srun` and `sbatch`.

## 2.5 Conda – Software Environment Management EML

We use *conda* to manage different versions of software available. Use

```
$ conda activate eml
```

to enable the default PyTorch environment used for the EML lecture.

You may be prompted on first use to run

```
$ conda init <bash|zsh>
```

depending on your login shell. This will modify your `.bashrc/.zshrc` file to load the appropriate paths and will change your prompt to show the currently loaded environment.

```
(base) csg-headnode% conda activate eml  
(eml) csg-headnode%
```

## 2.6 Spack – Software Environment Management GPU/HPDC

Additionally we use *Spack* to manage all non-Python software, especially CUDA installations. Specifically for CUDA development, there exists a special environment. Spack environment work similar to conda. Use

```
$ spack env activate cuda  
$ spack load cuda@<version>
```

To activate the CUDA environment and load a specific version of CUDA. Installed versions are `cuda@11.8.0` `cuda@12.1.1` `cuda@12.4.0` `cuda@12.5.0`. Now tools like `nvcc`, `cuda-gdb`, `cuobjdump` etc. are available.

# 3 Editing and File Transfer

There are multiple ways to get your programs and results to and from the cluster.

## 3.1 Editing Locally

You can edit your files on the head-node directly from the terminal. The editors [\(neo\)vim](#) and [nano](#) are preinstalled. If you are unfamiliar with `vim`, `nano` should be more straight forward.

```
$ nano <file>
```

## 3.2 scp

[scp](#) is both the name of the protocol and program used for copying files between remote machines using ssh. It works exactly like its local counterpart `cp`. If you want to copy a file from your local pc to the cluster use:

```
$ scp <file> headnode:<path-to-destination>
```

This also works in the other direction. If you want to copy a file back from the headnode use:

```
$ scp headnode:<path-to-file> <file>
```

NOTE: if you want to copy a folder do not forget to use the `-r` flag to recursively copy subdirectories and files.

## 3.3 rsync

A more sophisticated option to copy files is [rsync](#). `rsync` not only copies files, but also keeps track of changes like a version control system. It will not re-copy edited files but simply send a diff to the remote host which will update the file accordingly. Please read the documentation for usage. A basic example:

```
$ rsync -r --update <folder> headnode:<path-to-destination>
```

## 3.4 sshfs

You can also mount your development folder directly to your file system using [sshfs](#). This has the added benefit that you can edit all of your files directly in your favorite editor or IDE on your local machine. Depending on your network connection this can be flaky at times. You may want to pass the `-o reconnect` option to automatically reconnect if the connection is lost.

First create a mount-point for the remote file system, e.g.:

```
$ mkdir mnt-remote
```

Then mount the remote file system:

```
$ sshfs headnode:<path-to-folder> mnt-remote
```

Again consult the linked wiki article for more information on the options etc.

### 3.5 Using VS Code with SSH Extension

Visual Studio Code (VS Code) provides an SSH extension that enables you to work directly on the cluster by connecting to it via SSH. This allows you to use all the features of VS Code, including extensions and version control, as if you were working locally. Follow these steps to set it up:

1. Install the SSH extension in VS Code. You can find it by searching for "Remote - SSH" in the Extensions Marketplace.
2. Open the Command Palette in VS Code by pressing **Ctrl+Shift+P** (or **Cmd+Shift+P** on macOS) and select **Remote-SSH: Connect to Host...**
3. Enter the SSH command that connects to the cluster head node, for example: (if you have set up the config file as described in Section 1.2 you can choose the `csg-headnode` alias in a dropdown menu)

```
ssh -J <username>@zitigate.ziti.uni-heidelberg.de <  
→ username>@csg-headnode
```

4. If this is your first time connecting, you may need to add the host to your `known_hosts` file.
  5. After connecting, VS Code will open a new window connected to the remote system. You can navigate to the folder you want to work in using the **File Explorer** panel.
  6. You can now open, edit, and save files directly on the cluster. Use the integrated terminal within VS Code to execute commands on the remote machine as needed.
- Note:** If you set up SSH keys as described in Section 1.3, VS Code will automatically use them, so you won't need to enter your password each time.

This method is especially useful if you prefer to work in a GUI environment and benefit from VS Code's code completion and debugging features. For more information, refer to the [official documentation on the VS Code website](#).

### 3.6 Windows

Windows users can follow [this](#) tutorial on how to use ssh and scp from the Windows Powershell.