Ulysses - An Unofficial Guide Documentation

Matteo Seclì

Jul 03, 2021

Contents:

This is **The Missing Guide to Ulysses**, an unofficial how-to for the Ulysses cluster @ SISSA, from a regular user to other regular users.

Official cluster documentation: https://www.itcs.sissa.it/services/computing/hpc.

This guide is an open project, and contributions are *highly welcome*: you can report corrections or collaborate at https://github.com/matteosecli/Ulysses-doc.

Latest update: Jul 03, 2021.

CHAPTER 1

Why This Guide

A survival guide to Ulysses is long due, so I've just decided to write one based on my experience as a user in these years.

I still remember my first time on a cluster, on *this* cluster; no one ever explained me how to submit a job, why one should use a submission script, etc., so I spent the nights of an entire week trying to figure out all the bits and pieces to make some code work on the cluster (the fact that this code was highly complex and its usage had't been explained as well certainly contributed a lot).

I'm sure other people face the same situation I faced a few years ago when I first approached Ulysses; the best I can do is to offer back what I think I've learned.

This guide assumes that you know some basics of how a computer works (CPU, RAM, and that sort of stuff) and some Bash scripting.

Warning: This guide is a **personal project** and it's not officially affiliated in any way to SISSA. As such, is provided as-is and without any guarantees.

1.1 Cluster Basics

1.1.1 What is a Cluster? When and Why Should I Use It?

Using a cluster starts by having a rough knowledge of how it works.

A (computing) cluster is just a bunch of computers, called **nodes**, wired together via a high-speed connection; each of these computers is generally quite powerful itself. So, you can take advantage of a cluster in 2 scenarios.

- You have to do a super-demaning calculation, but this calculation can be split into smaller calculations that are performed at the same time by multiple nodes. All of these nodes communicate their results and/or status with each other, typically multiple times during

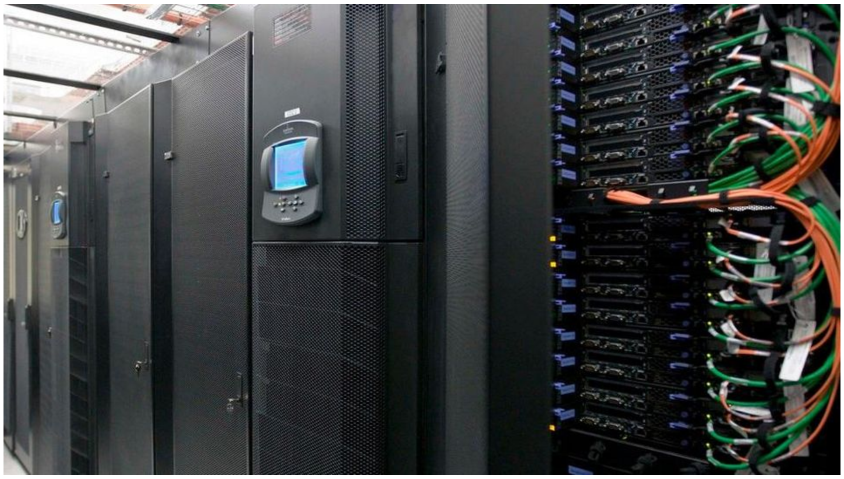


Fig. 1: A picture of (part of) Ulysses.

the calculation, via the high-speed connection that links the nodes. A typical "language" that allows nodes to communicate with each other on this high-speed connection is MPI (Message Passing Interface). This scenario is referred to as a **parallel job** or parallel calculation.

- You have to do a calculation that could in principle run fine on a powerful enough desktop workstation, but you have to *repeat* this calculation *lots* of times (maybe by varying some parameters). In this case, having a cluster is like having *lots* of workstations on which you can run many of these calculations at the same time, but these calculations don't need to communicate with each other. This scenario is referred to as a **serial job** or serial calculation.

Long story short: if you have to run *a single or few* calculations that can fit your workstation's computing capabilities, just *don't use a cluster*. You'd be better off just using your own workstation.

Note: Jobs that perform multi-core calculations are definitely implementing a form of parallelism, but if no communication is performed among different nodes then these jobs still fall under the category "serial job". The "parallel" here refers to the fact of communicating among different nodes

1.1.2 What's a Resource Manager?

A cluster is a **shared resource**. It's an important aspect to keep in mind, that makes its usage fundamentally different from using your office workstation; while you typically are the only user of your workstation, which means that you can run calculations whenever you want and for how long you want, on a cluster there are tons of users that want to run calculations at the same time. It's clear that if all the users are allowed to run whatever number of calculations all at the same time, this big boy will crash badly, **very badly**, no matter how big and powerful it is.

The solution to this problem is to employ a **resource manager**, sometimes called **queue manager**. The term "queue manager" is actually not totally correct, as these systems manage far more than just the queues, but I'd say that for our purposes the term "queue manager" gives a sufficiently intuitive idea of what this thing does.

Imagine you want to have fun and you decide to have a trip to Gardaland (an Italian amusement park). As there are multiple attractions, you have to:

- Decide whether to go to e.g. "Kung Fu Panda Master" or "Jungle Rapids" or "Blue Tornado".
- Once you decide which attraction you want to go to, you have to **queue up** for that attraction.

Fig. 2: Queues at an amusement park.

The park employee then makes sure that no one skips the queue, and assigns the visitors to the attraction's pods based on how people are grouped and on the available places. Let's say for example that a pod can host max. 6 people and that at the top of the queue there are a family of 5 people, another family of 3 people, and you (alone □). As an empty pod arrives, the queue manager lets the 5-people family on; an empty place is left, but the next in line is the 3-people family and it doesn't fit. A not-so-effective queue manager could wait for the next empty pod and let the 3-people family and you on it, but this is both a waste of time and resources for the park (filling all the empty spots means more $ for the park!) and for you, since you have to wait for the next pod despite the fact that the 5-people family left an empty spot on their pod. Instead, an effective queue manager would realize that you could fit in that empty spot and, despite you're not the next-in-line, would let you in. Or maybe the queue manager would let in a person even far behind you, because maybe that person has bought a *priority* ticket.

That's how the resource manager in a cluster works, more or less. In order for your calculation to be executed, you have to:

- Choose the right queue, called **partition** in our resource manager. The right partition depends on what you have to do; different partitions typically differ on the maximum time your calculation has got to finish, on the quantity of computing resources you can ask (e.g. how many CPUs or how much RAM), or on the kind of specific resources you can ask (e.g. some partitions are equipped on GPUs, some are not).

- Queue up for that queue. The way you explain to the cluster that you want your calculation to be queued up in that partition and that your calculation needs to use a certain kind/amount of resources is through a **job script**.

The **job script** is where all the magic happens and it's all you need to perform calculations on a cluster. In the next sections we'll go into a more detailed explanation of the job script.

> **Warning:** There's a way to ask for resources and queue for a partition even *without* a job script. This way of doing things is called an **interactive job**. There are some people that instruct new users to use the cluster only via these interactive jobs and don't explain **at all** what a job script is and why you should use it. This is **TOTAL BULLSHIT!** Unless you have a **really good and strong** reason to use an interactive job, **ALWAYS USE A JOB SCRIPT**. Not only because it forces you to be *programmatic*, but also because minimizes *a lot* the waste of computational resources.

1.2 Ulysses' Specs

Before we delve into the job script mystic magic, we have to take a look at the specs of our cluster, Ulysses. Knowing the specs of the cluster is important, as it allows you to carefully balance your computation and to choose the right partition (or "queue") for the calculation you have to perform.

1.2.1 Specs Sheet

Ulysses was originally inaugurated on September 24, 2014; it was originally based on a resource manager called TORQUE (Terascale Open-source Resource and QUEue Manager), a (not anymore) open-source resource manager derived from PBS (Portable Batch System), developed for NASA at the beginning of the 90s.

Ulysses has then been extended and a new infrastructure was made available during late 2019. This new infrastructure consists of additional nodes, an upgraded software stack and a new job scheduler, **SLURM** (Simple Linux Utility for Resource Management). After the upgrade the old nodes are used in partitions **regular1**, **long1**, **wide1** and **gpu1**, while the new nodes are available as **regular2**, **long2**, **wide2** and **gpu2**.

But let's take a closer look at the ID card of Ulysses.

> **Warning:** These info are given as they are, and they are the result of re-elaboration of public documentation [1] [2] on the cluster plus some small info I've gathered on the nodes. As this info is basically scattered around in a public form, I guess it's not classified info; if it is, however, please let me know. There is not, unfortunately, an official spec sheet of the cluster; if you have corrections or more info, please let me know.

- Name: **ulysses2**
- Operating System: SISSA Linux 7 (**CentOS**/RHEL **7.7.1908**)

- Scheduler: **SLURM 19.05.2**
- Nodes:
  - **2 x Login Nodes** (`frontend1.hpc.sissa.it`, `frontend2.hpc.sissa.it`) configured as:
    - Model: IBM System x3650 M4
    - CPU: 2 x Intel(R) Xeon(R) E5-2640 @ 2.50GHz
      - Each of the 2 CPUs has 6C/12T, but Hyper-Threading is disabled on the login nodes; so each login node has a total of 12C/12T.
    - RAM: 64 GiB
    - Disk: ???
  - **224 x Compute Nodes v1** configured as:
    - Model: IBM System x(R) iDataPlex(R) dx360 M4 Server
    - CPU: **2 x** Intel(R) Xeon(R) **E5-2680 v2 @ 2.80GHz**
      - Each of the 2 CPUs has 10C/20T. Hyper-Threading is *enabled*, so each node as a total of **20C/40T**.
    - RAM:
      - **184 nodes with 40 GiB** (2 GiB / core)
      - **24 nodes with 160 GiB** (8 GiB / core)
      - **8 nodes with 320 GiB** (16 GiB / core)
      - **8 nodes with 64 GiB** (3.2GiB / core, **GPU nodes only**)
    - GPU:
      - **8 nodes**, each with **2 x NVIDIA Tesla K20m** (each card has 5 GB GDDR5 memory).
    - Disk: ???
  - **88 x Compute Nodes v2** configured as:
    - Model: 77 x HPE ProLiant XL170r Gen9 (for CPU nodes) + 11 x HPE ProLiant XL170r Gen9 (for GPU nodes)
    - CPU: **2 x** Intel(R) Xeon(R) **E5-2683 v4 @ 2.10GHz**
      - Each of the 2 CPUs has 16C/32T. Hyper-Threading is *enabled*, so each node as a total of **32C/64T**.
    - RAM: **64 GiB** (2 GiB / core)
    - GPU:
      - **11 nodes**, each with **2 x NVIDIA Tesla P100 PCIe 16 GB** (each card has 16 GB HBM2 memory).
    - Disk: ???
- Storage:
  - Filesystem: Lustre 2.12.3
  - `/home`:

    - Total: 75.6 TiB (2 x OST with 37.8 TiB each)
    - User quota: **198 GiB (hard limit: 200 GiB)**

- `/scratch`:
    - Total: 533.2 TiB (2 x OST with 266.6 TiB each)
    - User quota: **4950 GiB (hard limit: 5000 GiB)**

1.2.2 Partitions

The partitions (queues) are then organized as follows; note that you can get detailed information about a partition via the command `scontrol show Partition=<name>`.

Table 1: Max resources you can ask for each partition.
(*): max 8 nodes. (**): max 24 nodes.

Partition	Max Nodes	Max Time (HH:MM:SS)	Max Memory per Node (MB)	Max Threads per Node	Max GPUs per Node
regular1	16	12:00:00	40000 160000 320000(*)	40	-
regular2			63500	64	
long1	8	48:00:00	40000 160000 320000	40	-
long2			63500	64	

wide1	32	08:00:00	40000
			160000(**)
			320000(*)
				40	-

wide2			63500	64	
gpu1	4	12:00:00	63500	40	2
gpu2				64	

Note: Clarification on max memory: on the the regular1, long1 and wide1 queues you can normally ask for 40000 MB max memory. However, there are also additional nodes with bigger memory. As as you can see in the *Specs Sheet*, though, there are not enough "big memory" nodes for all the possible configurations, as there are only 24 nodes with 160000 MB max memory and only 8 nodes with 320000 MB max memory. This means you have to be careful with big memory nodes if you queue jobs in regular1 or wide1. For example, it makes little sense to queue a job requiring all the 8 nodes with 320000 MB max memory in the wide queue, which in principle is useful only for a number of nodes greater than 16. Since there are only 8 nodes with 320000 MB max memory, it would make more sense to take advantage of the increased max time in the long1 queue and queue it there.

Note: Clarification on threads: since Hyper-Threading is enabled on all nodes, there are 2 threads per physical core. However, in SLURM's job script language, every thread is a CPU; this means that if you ask for "40 CPUs" in regular1 you are actually asking 40 threads, which is 20 physical cores. For a clarification on the definition on socket, core and thread take a look at the picture below.

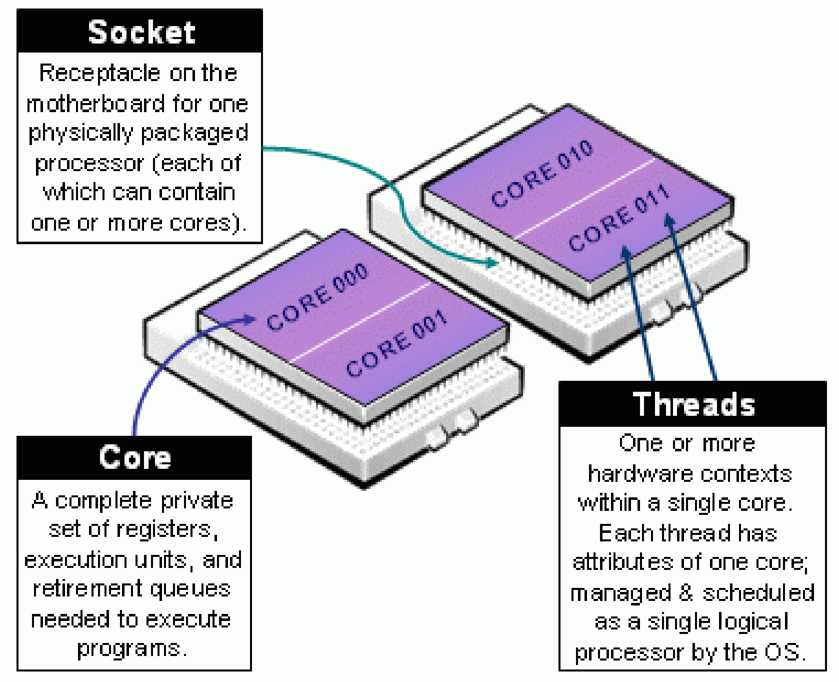

Socket
Receptacle on the motherboard for one physically packaged processor (each of which can contain one or more cores).
CORE 010
CORE 011
CORE 000
CORE 001
Core
A complete private set of registers, execution units, and retirement queues needed to execute programs.
Threads
One or more hardware contexts within a single core. Each thread has attributes of one core; managed & scheduled as a single logical processor by the OS.

Fig. 3: Definitions of Socket, Core, & Thread. From SLURM's documentation.

1.3 How To Use Ulysses

Note: This section assumes you have a basic understanding of how a cluster works (see *Cluster Basics* for a pedestrian introduction) and what are Ulysses' partitions (see *Partitions*). Having a look at the *Specs Sheet* is useful as well, even if maybe not completely necessary.

Note: The cluster has no graphical interface; all operations have to be done via a terminal (this is the norm with clusters). If you don't know what a terminal is or how to use it, you can have a quick tutorial on Ubuntu's website: https://ubuntu.com/tutorials/command-line-for-beginners. The tutorial also applies to other Linux distributions, as well as to MacOS.

1.3.1 Step 0: Get an Account

If you have a SISSA email, you can enable your account to Ulysses by writing an email to helpdesk-hpc@sissa.it.

If your name is *John Doe*, your email will probably look like `jdoe@sissa.it`; in the following, you will need to replace `username` with your equivalent of `jdoe`.

1.3.2 Step 1: Login

Warning: Ulysses used to be reachable from outside SISSA; however, since the upgraded version is still in beta, it's temporarily reachable only from SISSA's internal network. In order to enter the internal SISSA network, you have to connect to SISSA's VPN: https://www.itcs.sissa.it/services/network/internal/vpnclient. Afterwards, you can access the cluster as explained below.

Warning: `frontend1.hpc.sissa.it` is still providing access to the old version of Ulysses, but it will be soon redeployed as a login node for the new version of Ulysses. Until that happens, please use only `frontend2.hpc.sissa.it` to access Ulysses.

Open up a terminal and execute the following command (do not copy the "$", it's just there to say that we are executing a shell command):

```
$ ssh username@frontend2.hpc.sissa.it
```

Note: If you need to use graphical software on the cluster, replace `ssh` with `ssh -X`.

You should be greeted with a welcome message and the terminal should be positioned in your home folder. To check that, executing the `pwd` command should give you `/home/username`.

Note: `/home/username` is meant to store important files, *not* to run calculations, and it's backed up daily. The quota you have is 200 GiB.

To move to a place where you can run calculations, execute `cd /scratch/username`. Here you can create a new folder for you brand new HPC project via `mkdir yourfolder`. Then move to the new folder with `cd yourfolder`.

Note: `/scratch/username` is a parallel filesystem on which you can run calculations, and **it's not backed up**. The quota you have is 5 TiB.

1.3.3 Step 2: Load Your Software

It's time to load the software that you're going to use for the simulation, or to compile your own.

Step 2.1: Using Modules

Ulysses' software is organized in **modules**. ITCS has a great in-depth documentation on the modules structure, so if you're not familiar with it please take a look: https://www.itcs.sissa.it/services/modules. Here I'll just cover the basics.

In a nutshell modules are a great way of handling multiple software or multiple versions of the same software, and they work as a plug-in / plug-out tool; you plug in a certain software to make it available to the system, and when you finish you can plug it out so that it doesn't interfere with another software you have to use.

Say for example that you want to use Intel's C compiler, `icc`. This command, however, is not available by default. So you first determine what modules are available in the system, and whether Intel's compiler is among them; to do that, you use `module avail`:

```
$ module avail

-------------------------- /opt/ohpc/pub/modulefiles -------------------------
↪--
   autotools       hwloc/2.1.0              llvm5/5.0.1
   cmake/3.15.4    intel/18.0.3.222         prun/1.3
   gnu8/8.3.0      intel/19.0.4.243 (D)    singularity/3.4.1

------------------------- /opt/sissa/modulefiles/apps ------------------------
↪--
   comsol/5.5                mathematica/11.3        python3/3.6
   git/2.9.3                 mathematica/12.0 (D)    ruby/2.5
   git/2.18           (D)    matlab/2016b            sm/2.4.43
   gnuplot/5.2.4             matlab/2018a            texlive/2018
   idl/8.7                   matlab/2018b            texlive/2019    (D)
   intelpython3/3.6.8        matlab/2019a            vmd/1.9.3       (D)
   maple/2018                matlab/2019b     (D)    vmd/1.9.4a12
   maple/2019         (D)    perl/5.26               xcrysden/1.5.60

----------------------- /opt/sissa/modulefiles/compilers ---------------------
↪--
   cuda/9.0    cuda/10.0        gnu7/7.3.1
   cuda/9.1    cuda/10.1 (D)    pgi/19.4

------------------------- /opt/sissa/modulefiles/libs ------------------------
↪--
   eigen/3.3.7    fsl/6.0.1    libtensorflow_cc/1.14.0

-------------------- /usr/share/lmod/lmod/modulefiles/Core -------------------
↪--
   lmod    settarg

  Where:
   D:  Default Module
```

As you see there are 2 versions of the `intel` module: `intel/18.0.3.222` and `intel/19.0.4.243`. You can load a specific version, for example `intel/19.0.4.243`, via:

```
$ module load intel/19.0.4.243
```

If you instead use the generic module name, `intel`, the default version (marked with `(D)`) will be loaded:

```
$ module load intel
```

You can check which modules are loaded in your environment by using `module list`:

```
$ module list

Currently Loaded Modules:
  1) gcc/8   2) intel/19.0.4.243
```

For example, in this case, loading the `intel/19.0.4.243` module automatically loaded the `gcc/8` module as well.

It might happen that you're looking for a specific piece of software but you don't see it when you use `module avail`. Let's say for example that you want to use Intel's MKL with the GNU compiler. While Intel's MKL is automatically included in Intel's compiler module, it has to be loaded as an additional module when usign the GNU compiler. However, the `module avail` command above shows no traces of an `mkl` module; but trust me, it's there.

The infallible way to find modules, even the ones that might be "invisible" by default, is via `module spider`. It's a sort of internal search engine for modules; you can look for a possible `mkl` module via:

```
$ module spider mkl

-----------------------------------------------------------------------------
  mkl:
-----------------------------------------------------------------------------
    Description:
      Intel Math Kernel Library for C/C++ and Fortran

     Versions:
        mkl/18.0.3.222
        mkl/19.0.4.243

-----------------------------------------------------------------------------
  For detailed information about a specific "mkl" module (including how to␣
↪load
the modules) use the module's full name.
  For example:

     $ module spider mkl/19.0.4.243
-----------------------------------------------------------------------------
```

Ok, this tells us that the module it's there (and there are 2 available versions) but still it doesn't say how to load it. However, the message is quite explanatory: if you want to receive instructions on how to load a specific version of the module, you have to use `module spider` again with the specific version you want to load. For example, if we want to know how to load `mkl/19.0.4.243`:

```
$ module spider mkl/19.0.4.243

-----------------------------------------------------------------------------
  mkl: mkl/19.0.4.243
-----------------------------------------------------------------------------
    Description:
      Intel Math Kernel Library for C/C++ and Fortran
```

(continues on next page)

(continued from previous page)

```
   You will need to load all module(s) on any one of the lines below before␣
↪the
"mkl/19.0.4.243" module is available to load.

     gnu7/7.3.1
     gnu8/8.3.0
     pgi/19.4
```

Aha! The message is telling us that if we want to be able to load `mkl/19.0.4.243` we have to load one of the 3 listed modules first. Let's say that we choose to load `gnu8/8.3.0`; then, if we do `module avail` again after loading this module, this is what we get:

```
$ module load gnu8/8.3.0
$ module avail

------------------------- /opt/ohpc/pub/moduledeps/gnu -----------------------
↪--
   mkl/18.0.3.222    mkl/19.0.4.243 (D)

-------------------------- /opt/sissa/moduledeps/gnu8 ------------------------
↪--
   gmp/6.1.2

------------------------ /opt/ohpc/pub/moduledeps/gnu8 -----------------------
↪--
   R/3.6.1    hdf5/1.10.5    mvapich2/2.3.2    openmpi3/3.1.4    scotch/6.0.6
   gsl/2.6    metis/5.1.0    openblas/0.3.7    plasma/2.8.0      superlu/5.2.1

-------------------------- /opt/ohpc/pub/modulefiles -------------------------
↪--
   autotools           hwloc/2.1.0             llvm5/5.0.1
   cmake/3.15.4        intel/18.0.3.222        prun/1.3
   gnu8/8.3.0   (L)    intel/19.0.4.243 (D)    singularity/3.4.1

------------------------- /opt/sissa/modulefiles/apps ------------------------
↪--
   comsol/5.5                  mathematica/11.3       python3/3.6
   git/2.9.3                   mathematica/12.0 (D)    ruby/2.5
   git/2.18           (D)      matlab/2016b            sm/2.4.43
   gnuplot/5.2.4               matlab/2018a            texlive/2018
   idl/8.7                     matlab/2018b            texlive/2019    (D)
   intelpython3/3.6.8          matlab/2019a            vmd/1.9.3       (D)
   maple/2018                  matlab/2019b     (D)    vmd/1.9.4a12
   maple/2019         (D)      perl/5.26               xcrysden/1.5.60

----------------------- /opt/sissa/modulefiles/compilers ---------------------
↪--
   cuda/9.0    cuda/10.0        gnu7/7.3.1
   cuda/9.1    cuda/10.1 (D)    pgi/19.4

------------------------- /opt/sissa/modulefiles/libs ------------------------
↪--
```

(continues on next page)

(continued from previous page)

```
   eigen/3.3.7    fsl/6.0.1    libtensorflow_cc/1.14.0

-------------------- /usr/share/lmod/lmod/modulefiles/Core -------------------
↪--
   lmod    settarg

  Where:
   L:  Module is loaded
   D:  Default Module
```

Note: Loading the module `gnu8/8.3.0` has made available a whole bunch of new different modules that depend on it, in the first three sections that end with `moduledeps/gnu` or `moduledeps/gnu8`. These new sets of modules include our original target, `mkl/19.0.4.243`, but there are also other useful modules such as R, OpenMPI, the HDF5 library or various linear algebra libraries.

This way of searching and loading modules might seem convoluted and overly complicated at first sight, but it's actually a neat way to keep mental sanity when using `module avail`. Imagine if *all* the dependencies of *all* the modules were made available altogether when using `module avail`; it would be a neverending list of available modules that would make impossible to spot the one we are looking for. Instead, `module avail` shows a few "fundamental" modules; then, loading one of them often make additional modules available. In any case, if you have to look for additional modules, `module spider` is likely to become your new best friend. □

Oh, one last thing: if you want to unload a module, just use `module unload modulename` (for example `module unload intel`). To unload **all** the modules at the same time, use `module purge`.

Step 2.2: Prepare Your Software

Depending on your task, the software you need to use for you calculations might be already available as a module or you need to compile it from scratch. In this latter case, refer to the code documentation to find out all the dependencies and load the corresponding modules (or compile the dependencies as well if there are no corresponding modules); then compile the software.

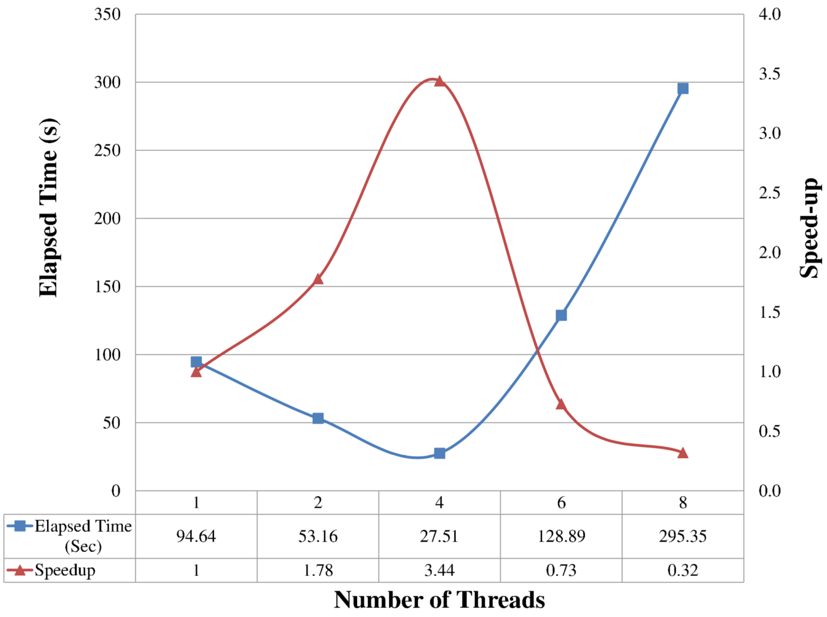
Either way, once your code compiles correctly with all the dependencies you need (and you might want to do a small 5-seconds run just to be sure that there are no missing dependencies at runtime), **note down all the modules you've loaded in the environment**. Just to be more explicit, use `module list` and copy over a file or a piece of paper, you pick one, **all the modules** that are loaded. You're gonna need this list later, so keep it somewhere you can easily find it.

1.3.4 Step 3: Define Your Workload

Every good calculation starts by carefully engineering the parameters of the simulations. Not only the "physical" parameters of the problem at hand, but also the technical parameters such as the number of CPU cores you have to use, whether or not use a GPU, the memory needed and so on.

In general, you ask yourself the following questions.

- Do you need to do a **serial MPI** or a **parallel MPI** simulation?
  - For a serial MPI calculation you use **1 node** (or **part** of a node)
  - For a parallel MPI calculation you use **1 or more nodes**
- How many CPUs (i.e. threads, i.e. 2 x number of cores in our case) do you need?
  - Not a trivial question, as parallelism doesn't always scale linearly and can saturate after a certain number of CPUs. So **don't blindly ask for all the available CPUs**; the best thing to do is to submit some small jobs that do test runs for a variable number of CPUs, and then pick the number of threads that gives the lowest elapsed time ("real" time, *not* "cpu" time!) or a similar metric. The reason is that, if your software uses multi-core parallelism, you should typically see that time decreases as you increase the number the threads, until at a certain point it stays steady or increases again due to overhead among threads. So the sweet spot for the number of threads is at the minimum of this curve.



	1	2	4	6	8
Elapsed Time (Sec)	94.64	53.16	27.51	128.89	295.35
Speedup	1	1.78	3.44	0.73	0.32

Fig. 4: Example of elapsed time and speedup vs. number of threads. Image from S.Sankaraiah et al, "Parallel Full HD Video Decoding for Multicore Architecture", doi:10.1007/978-981-4585-18-7_36. (C) Springer Science+Business Media Singapore 2014.

- (If you do parallel MPI calculations) How many MPI processes do you need?
  - You can apply the same reasoning explained above for the number of CPUs.
- How much memory (RAM) do you need?
- How much disk space do you need? Does it fit into your `/scratch` quota?
  - You can check your quota by using the `quota` command:

```
$ quota
     /home:   XXX of   198GB (hard limit =   200GB)
  /scratch:  XXXX of  4950GB (hard limit =  5000GB)
```

Please consider that your quota, particularly the `/scratch` one, is **purely theoretical**, as likely there aren't 5 TB for everybody; so you should also check the overall free space. The 5 TB limit is there so that people that need to perform calculations that store lots of data can do it, but then they should **move** this data elsewhere; the possibility to store large simulations should be considered **temporary**.

In order to check the overall free space on `/home` and `/scratch` you can do:

```
$ df -h /home /scratch
Filesystem                                  Size  Used Avail Use% Mounted␣
↪on
10.7.0.43@o2ib,10.7.0.44@o2ib:/home2         76T   11T   65T  14% /home
10.7.0.43@o2ib,10.7.0.44@o2ib:/scrt2        534T  247T  287T  46% /scratch
```

- How much time do you need?
- Do you have to do a single calculation or multiple, similar calculations that vary just by a few parameters? If yes, how many of them?

Finding the right balance between all these variables and make them fit into the limits of one or more partitions is **your job**. It might take a while to get accustomed to it, but it's practice that makes perfect; so just get your hand dirty! □

Note: In general, the less the resources you ask for the faster your calculations will start crunching numbers instead of endlessly waiting in queue. So you have to try to **minimize the resources you ask for**, while at the same time making sure that your software has the necessary amounts of CPUs, RAM, time, etc. to get the job done.

Once you are done picking up all these parameters, **note them down** somewhere; you'll need this info in the next step.

1.3.5 Step 4: Prepare and Send the Job Script

It's finally time for the long awaited job script! Hooray! □

Let's immediately jump into it and take a look. The script is heavily commented so to help you out, but don't worry if it's not all clear at first sight; there's a step-by-step explanation right after the code block.

You can also download a copy of the job script here: `send_job.sh`.

```
#!/usr/bin/env bash
#
#
# ==== SLURM part (resource manager part) ===== #
#   Modify the following options based on your job's needs.
#   Remember that better job specifications mean better usage of resources,
#   which then means less time waiting for your job to start.
#   So, please specify as many details as possible.
#   A description of each option is available next to it.
#   SLURM cheatsheet:
#
#     https://slurm.schedmd.com/pdfs/summary.pdf
#
```

(continues on next page)

(continued from previous page)

```
#
# ---- Metadata configuration ----
#
#SBATCH --job-name=YourJobName       # The name of your job, you'll se it in␣
↪squeue.
#SBATCH --mail-type=ALL              # Mail events (NONE, BEGIN, END, FAIL,␣
↪ALL). Sends you an email when the job begins, ends, or fails; you can␣
↪combine options.
#SBATCH --mail-user=user@sissa.it    # Where to send the mail
#
# ---- CPU resources configuration  ----  |  Clarifications at https://slurm.
↪schedmd.com/mc_support.html
#
#SBATCH --ntasks=1                   # Number of MPI ranks (1 for MPI serial␣
↪job)
#SBATCH --cpus-per-task=40           # Number of threads per MPI rank (MAX:␣
↪2x32 cores on _partition_2, 2x20 cores on _partition_1)
#[optional] #SBATCH --nodes=1                    # Number of nodes
#[optional] #SBATCH --ntasks-per-node=1          # How many tasks on each node
#[optional] #SBATCH --ntasks-per-socket=1        # How many tasks on each␣
↪socket
#[optional] #SBATCH --ntasks-per-core=1          # How many tasks on each␣
↪core (set to 1 to be sure that different tasks run on different cores on␣
↪multi-threaded systems)
#[optional] #SBATCH --distribution=cyclic:cyclic # Distribute tasks␣
↪cyclically on nodes and sockets. For other options, read the docs.
#
# ---- Other resources configuration (e.g. GPU) ----
#
#[optional] #SBATCH --gpus=2                     # Total number of GPUs for␣
↪the job (MAX: 2 x number of nodes, only available on gpu1 and gpu2)
#[optional] #SBATCH --gpus-per-node=2            # Number of GPUs per node␣
↪(MAX: 2, only available on gpu1 and gpu2)
#[optional] #SBATCH --gpus-per-task=1            # Number of GPUs per MPI␣
↪rank (MAX: 2, only available on gpu1 and gpu2); to be used with --ntasks
#
# ---- Memory configuration ----
#
#SBATCH --mem=7900mb                 # Memory per node (MAX: 63500 on the new␣
↪ones, 40000 on the old ones); incompatible with --mem-per-cpu.
#[optional] #SBATCH --mem-per-cpu=4000mb         # Memory per thread;␣
↪incompatible with --mem
#
# ---- Partition, Walltime and Output ----
#
#[unconfig] #SBATCH --array=01-10    # Create a job array. Useful for␣
↪multiple, similar jobs. To use, read this: https://slurm.schedmd.com/job_
↪array.html
#SBATCH --partition=regular1         # Partition (queue). Avail: regular1,␣
↪regular2, long1, long2, wide1, wide2, gpu1, gpu2. Multiple partitions are␣
↪possible.
#SBATCH --time=00:05:00              # Time limit hrs:min:sec
```

(continues on next page)

(continued from previous page)

```
#SBATCH --output=%x.o%j              # Standard output log in TORQUE-style -- WARNING: %x requires a new enough SLURM. Use %j for regular jobs and %A-%a for array jobs
#SBATCH --error=%x.e%j               # Standard error  log in TORQUE-style -- WARNING: %x requires a new enough SLURM. Use %j for regular jobs and %A-%a for array jobs
#
# ==== End of SLURM part (resource manager part) ===== #
#
#
# ==== Modules part (load all the modules) ===== #
#   Load all the modules that you need for your job to execute.
#   Additionally, export all the custom variables that you need to export.
#   Example:
#
#     module load intel
#     export PATH=:/my/custom/path/:$PATH
#     export MAGMA_NUM_GPUS=2
#
#
# ==== End of Modules part (load all the modules) ===== #
#
#
# ==== Info part (say things) ===== #
#   DO NOT MODIFY. This part prints useful info on your output file.
#
NOW=`date +%H:%M-%a-%d/%b/%Y`
echo '------------------------------------------------------'
echo 'This job is allocated on '$SLURM_JOB_CPUS_PER_NODE' cpu(s)'
echo 'Job is running on node(s): '
echo  $SLURM_JOB_NODELIST
echo '------------------------------------------------------'
echo 'WORKINFO:'
echo 'SLURM: job starting at           '$NOW
echo 'SLURM: sbatch is running on      '$SLURM_SUBMIT_HOST
echo 'SLURM: executing on cluster      '$SLURM_CLUSTER_NAME
echo 'SLURM: executing on partition    '$SLURM_JOB_PARTITION
echo 'SLURM: working directory is      '$SLURM_SUBMIT_DIR
echo 'SLURM: current home directory is '$(getent passwd $SLURM_JOB_ACCOUNT | cut -d: -f6)
echo ""
echo 'JOBINFO:'
echo 'SLURM: job identifier is         '$SLURM_JOBID
echo 'SLURM: job name is               '$SLURM_JOB_NAME
echo ""
echo 'NODEINFO:'
echo 'SLURM: number of nodes is        '$SLURM_JOB_NUM_NODES
echo 'SLURM: number of cpus/node is    '$SLURM_JOB_CPUS_PER_NODE
echo 'SLURM: number of gpus/node is    '$SLURM_GPUS_PER_NODE
echo '------------------------------------------------------'
#
# ==== End of Info part (say things) ===== #
```

(continues on next page)

(continued from previous page)

```
#

# Should not be necessary anymore with SLURM, as this is the default, but you never know...
cd $SLURM_SUBMIT_DIR


# ==== JOB COMMANDS ===== #
#   The part that actually executes all the operations you want to do.
#   Just fill this part as if it was a regular Bash script that you want to
#   run on your computer.
#   Example:
#
#     echo "Hello World! :)"
#     ./HelloWorld
#     echo "Executing post-analysis"
#     ./Analyze
#     mv analysis.txt ./results/
#

# ==== END OF JOB COMMANDS ===== #


# Wait for processes, if any.
echo "Waiting for all the processes to finish..."
wait
```

Ok, let's go step-by-step.

Script Format

The script is just a regular Bash script that contains the operations you want to perform in the `JOB COMMANDS` section towards the end.

Comments, in Bash scripts, begin with a #; however, in this case, lines that begin with the comment `#SBATCH` are **special lines** that **do not** get executed anyway by the Bash script, **but are parsed by SLURM** and interpreted as **instructions for the resources** you are asking for. Lines that do not begin with that exact sequence of characters, like `#[optional] #SBATCH` or `#[unconfig] #SBATCH`, are instead *ignored* by SLURM and treated just as other regular comments.

Each `#SBATCH` directive is followed by an option, in the form `#SBATCH --option=value` (and then by another explanatory comment that I've added just for reference). These options are **the same** options supported by SLURM's `sbatch` command The complete list of `sbatch` options is available on SLURM's website: https://slurm.schedmd.com/sbatch.html; refer to it for clarifications or more advanced settings.

`sbatch` is the primary command that you use to queue a calculation. Suppose for a second that you have a regular Bash script with the list of operations you want to do with your software (i.e. containing just the `JOB COMMANDS` section, nothing else). In principle, you could send this regular script via e.g.

```
$ sbatch --ntasks=1 --cpus-per-task=40 --mem=7900mb --partition=regular1 --
↪time=00:05:00 ...(other options)... regular_script.sh
```

However remembering all these options every time you have to send a calculation is a pain in the neck, without adding the fact that you don't have a trace of all the options you've used to send a particular job or set of jobs.

The solution is to collect all these command-line options for `sbatch` in those special `#SBATCH --option=value` Bash comments; all those `--option=value` pairs will then be interpreted just as if they were given to `sbatch` on the command line.

In other words, if you augment you regular Bash script `regular_script.sh` with special `#SBATCH --option=value` lines so it becomes what we are calling `job_script.sh`, you can then send it to the queue via a simple:

```
$ sbatch job_script.sh
```

Yep, as simple as that. Of course, if you realize at the very last moment that you want to add some extra command line option, you can do that directly at the command-line level without necessarily adding it to the job script. However, if you have a stable collection of all the command-line options that you commonly use in a single place, i.e. at the beginning of a job script, you can then copy-paste that script every time you have to send a new calculation and just change the relevant bits.

The script is organized in 4 parts. Do not take it as a sacred division, you are actually free to organise it as you want; I just feel that this division may help to keep it well organized. Now we'll dive in each part in details.

SLURM Part

This is the section that contains the command-line options for `sbatch` that you want to use. It is further divided in sections, though this division it's a bit random (e.g. the output and error names maybe belong to the metadata section, but in the end you can move this options as you want).

You will need all the resource requirements you've engineered and collected in *Step 3: Define Your Workload*.

Metadata Configuration

Contains the job name and things like email configuration.

`--job-name=<value>`

A name for your job. Give a descriptive but short name.

Example:

```
#SBATCH --job-name=TestJob
```

`--mail-type=<value>`

Whether to send an email when a job begins (BEGIN), ends (END) or fails (FAIL). Other options are ALL or NONE. You can combine options by separating them with commas.

Example:

```
#SBATCH --mail-type=END,FAIL
```

`--mail-user=<value>`

The address to send messages to.

Example:

```
#SBATCH --mail-type=jdoe@sissa.it
```

CPU Resources Configuration

Contains how many MPI tasks you need, how many threads, etc.

`--ntasks=<value>`

How many MPI processes you need. Set it to 1 for "serial" jobs (i.e. that don't use MPI, but can still use OpenMP parallelism). If you are not sure, set it to *1*.

Example:

```
#SBATCH --ntasks=1
```

`--cpus-per-task=<value>`

Number of "CPUs" (i.e. **CPU threads**) per MPI task. This number is limited by each node capabilities, so the max is 64 threads = 2 x 32 cores on the partitions ending with "2" and 40 threads = 2 x 20 cores on the partitions ending with "1". The global number of CPUs (threads) that you get is equal to this number multiplied by the number MPI processes you've asked for.

Example:

```
#SBATCH --ntasks=1
```

`[optional] --nodes=<value>`

Number of nodes to ask tor. This is optional, as SLURM can use the previous two options automatically determine the number of nodes to give you.

Example:

```
#SBATCH --nodes=2
```

`[optional] --ntasks-per-[node|socket|core]=<value>`

Number of MPI processes per [node|socket|core]. These options are used to fine-tune how the MPI processes are distributed among the available resources. If you don't have a good reason to use it, just don't use it.

I think that a useful option is `--ntasks-per-core=1`, that forces separate MPI processes to run on separate cores in order to avoid possible slowdowns. This automatically happens if Hyper-Threading is disabled, but since it's enabled on Ulysses2 this option can take care of it. Note that this *doesn't disable* Hyper-Threading, and on the contrary the MPI processes can still take advantage of having 2 threads per core. The only difference is that we make sure that each pair of threads belongs to the same core.

Example:

```
#SBATCH --ntasks-per-core=1
```

Other Resources Configuration

Settings for other resources, such as GPUs.

--gpus=<value>

Total number of GPUs requested for the job. Valid only on gpu1 and gpu2; you can ask at most 2 GPUs for each node you ask.

Example:

```
#SBATCH --gpus=2
```

--gpus-per-node=<value>

Number of GPUs per node. Valid only on gpu1 and gpu2; you can ask at most 2 GPUs.

Example:

```
#SBATCH --gpus-per-node=2
```

--gpus-per-task=<value>

Number of GPUs per MPI process; intended for use in conjunction with --ntasks. Valid only on gpu1 and gpu2; you can ask at most 2 GPUs.

Example:

```
#SBATCH --gpus-per-task=2
```

Memory Configuration

Memory (RAM) resources. **Always specify the amount of RAM you need, otherwise you might encounted unexpected behaviors!**

--mem=<value>

Amount of memory per node. Refer to the table in *Partitions* for the limits. Incompatible with --mem-per-cpu.

Example:

```
#SBATCH --mem=7900mb
```

[optional] --mem-per-cpu=<value>

Amount of memory per CPU (thread). Default value: 512mb; refer to the table in *Partitions* for the limits. Incompatible with --mem.

Example:

```
#SBATCH --mem-per-cpu=4000mb
```

Partition, Walltime and Output

This section specifies the partition to use, the requested amount of time and the names of the log files.

`--partition=<value>`

Partition (queue) to send the job to. Refer to the table in *Partitions* for the details. Multiple partitions are possible, separated by commas; the job will then execute on the first partition that has available resources.

Example:

```
#SBATCH --partition=regular1,regular2
```

`--time=<value>`

The maximum time to be allocated for your job, in `HH:MM:SS` format. Default value: `01:00:00`; refer to the table in *Partitions* for the time limits.

Example:

```
#SBATCH --time=12:00:00
```

`[optional] --array=<value>`

Generates and queues multiple copies of the same job, each one having a different value of the environment variable `$SLURM_ARRAY_TASK_ID`. The value for this variable is taken from a list that you provide as input, which can be a range (e.g. `1-10`, optionally with a step e.g. `1-10:2`) or a list of values (e.g. `1,3,17`).

Useful if you have to run multiple calculations that just differ by the value of some parameters, as you can use the value of `$SLURM_ARRAY_TASK_ID` to select a parameter or a specific set of parameters for that simulation.

More info and examples: https://slurm.schedmd.com/job_array.html

Example:

```
#SBATCH --array=0-12:3    # Expands to 0, 3, 6, 9, 12
```

`[optional] --[output|error]=<value>`

Custom name for the output log and the error log. In order to use file names which are compatible with the ones use by TORQUE (the scheduler in Ulysses v1), I suggest to use this form

```
#SBATCH --output=%x.o%j
#SBATCH --error=%x.e%j
```

for regular jobs and this form

```
#SBATCH --output=%x.o%A-%a
#SBATCH --error=%x.e%A-%a
```

for array jobs. `%x` is a placeholder for the name, `%j` for the job ID, `%A` for the global array job ID and `%a` for the array index.

This is just a suggestion, you can use the names that you prefer or just don't specify them in order to use the default ones.

Modules Part

This section to be filled with **all the modules** you've collected in *Step 2: Load Your Software*. Be sure to export the relevant environment variables as well, if needed by your software.

Example (just random stuff to make my point):

```
module load intel
export PATH=:/my/custom/path/:$PATH
export MAGMA_NUM_GPUS=2
```

Info Part

You don't need to modify this part; it's just a section that prints some job info in your output file, nothing more. If you don't like it, you can safely remove it.

Job Commands Part

This is the section that contains **the actual things your calculation has to do**, so this section is completely up to you.

Before the commands, there's a line

```
cd $SLURM_SUBMIT_DIR
```

that brings the shell into the directory from which you've submitted the script. A similar line was necessary with the old TORQUE system, but it should be the default behavior with SLURM. Since you never know, best to make it sure it's actually this way.

Example (just a template of what your code might do):

```
echo "Hello World! :)"
./HelloWorld
echo "Executing post-analysis"
./Analyze
mv analysis.txt ./results/
```

Send and Monitor Your Job

At this point, you're ready to go. You can put your job in queue via (the script is named `send_job.sh`):

```
$ sbatch send_job.sh
```

The command above will return a number identifying the job, called *job ID*.

The script will be executed in the folder from where you've sent it as soon as there are available resources; you can monitor the status of your request for resources via:

```
$ squeue -j jobid
```

Or, to view all of your jobs, you can use

```
$ squeue -u username
```

You can print more job infos by adding the `-l` option; as a general tip, I also suggest you to use the screen reader `less` so not to clog your terminal:

```
$ squeue -l -u username | less
```

When you've finished taking a look, you can quit the screen reader by pressing the Q key on your keyboard.

For both running jobs and jobs that are already completed, you can print even more detailed infos via (just replace jobid with):

```
$ sacct -j jobid --format=User,JobID,Jobname,partition,state,time,start,end,
↪elapsed,MaxRss,MaxVMSize,nnodes,ncpus,nodelist
```

Note: As you've noticed, basically all SLURM commands start by "s" and are followed by a term that explain their functions. So "batch" refer to batch jobs, "queue" refer to jobs in queue, etc. In this case "acct" stands for "accounting" information.

1.3.6 Step 5: Get Your Results

Once the calculations are completed, you might want to transfer the results (or some post-processed form of the results) to your SISSA workstation or to your laptop. There are 2 command-line based ways to do it:

- Via scp
- Via sftp

ITCS has a very complete documentation on the usage of scp, so just have a look at it: https://www.itcs.sissa.it/services/network/scp/scp-index. The only caveat is that you have to replace your-SISSA-username@hostname:local_path/file_name with username@frontend2.hpc.sissa.it:path_on_the_cluster/file_name.

As for sftp, there are many tutorials online. You can for example read this one by DigitalOcean: https://www.digitalocean.com/community/tutorials/how-to-use-sftp-to-securely-transfer-files-with-a-remote-server. Again, you just have to replace sammy@your_server_ip_or_remote_hostname with username@frontend2.hpc.sissa.it.

Note: There are also more user-friendly ways to access the files on the cluster, so that you can navigate these files with the default file manager of you operating system as if they were on your workstation or on your laptop. These setups are discussed in the section *Explore Files in a User-Friendly Way*.

1.4 Examples

This section contains examples of job scripts for typical situations. Refer to *Step 4: Prepare and Send the Job Script* for details on the job script.

I'll skip all the extra comments, as well the part that prints the job info in the output file and the part that loads the modules so to focus just on the relevant bits.

1.4.1 Serial Jobs

Serial Job – Without OpenMP

Typical situation for a serial (non-MPI) job. On a cluster without Hyper-Threading 1 CPU is enough, but since on Ulysses we have Hyper-Threading we can ask for 2 (virtual) CPUs in such a way that they correspond to 1 physical core.

Listing 1: Serial job asking for 2 threads (bounded to **1 physical core**), 32 GB of memory and 10 hours on `regular1`. The output and error filenames are in TORQUE style.

```
#!/usr/bin/env bash
#
#SBATCH --job-name=Serial_Job
#SBATCH --mail-type=ALL
#SBATCH --mail-user=jdoe@sissa.it
#
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=2
#SBATCH --ntasks-per-core=1
#
#SBATCH --mem=32000mb
#
#SBATCH --partition=regular1
#SBATCH --time=10:00:00
#SBATCH --output=%x.o%j
#SBATCH --error=%x.e%j
#

## YOUR CODE GOES HERE (load the modules and do the calculations)
## Sample code:

# Make sure it's the same module used at compile time
module load intel

# Run calculation
./my_program.x
```

Serial Job – With OpenMP

If you are confident that your software implements OpenMP parallelism, you can ask for more threads.

Listing 2: Serial job asking for 20 (OpenMP) threads, 20 GB of memory and 10 hours on either `regular1` or `regular2`. The output and error filenames are in TORQUE style.

```
#!/usr/bin/env bash
#
```

(continues on next page)

(continued from previous page)

```
#SBATCH --job-name=Serial_Job_OpenMP
#SBATCH --mail-type=ALL
#SBATCH --mail-user=jdoe@sissa.it
#
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=20
#
#SBATCH --mem=20000mb
#
#SBATCH --partition=regular1,regular2
#SBATCH --time=10:00:00
#SBATCH --output=%x.o%j
#SBATCH --error=%x.e%j
#

## YOUR CODE GOES HERE (load the modules and do the calculations)
## Sample code:

# Make sure it's the same module used at compile time
module load intel

# Optionally set the number of OpenMP threads as you want
# Example:     export OMP_NUM_THREADS=$((${SLURM_CPUS_PER_TASK}/2))

# Print info on current number of OpenMP threads
"Using "$(($OMP_NUM_THREADS))" OpenMP threads."

# Run calculation
./my_program.x
```

Note: You can add the option `#SBATCH --ntasks-per-core=1` to make sure that 20 CPUs correspond to actual 10 physical cores.

1.4.2 Parallel Jobs

MPI Job – Without OpenMP

This is an example of a pure MPI job. Again, we ask for 2 CPUs per task but we make sure they correspond to 1 physical core.

Listing 3: Parallel job asking for 32 MPI tasks, each with 2 threads (bounded to **1 physical core**) and 1.98 GB of memory for a total of 12 hours on `regular2`. The output and error filenames are in TORQUE style.

```
#!/usr/bin/env bash
#
#SBATCH --job-name=Parallel_Job
#SBATCH --mail-type=ALL
```

(continues on next page)

(continued from previous page)

```
#SBATCH --mail-user=jdoe@sissa.it
#
#SBATCH --ntasks=32
#SBATCH --cpus-per-task=2
#SBATCH --ntasks-per-core=1
#
#SBATCH --mem-per-cpu=990mb
#
#SBATCH --partition=regular2
#SBATCH --time=12:00:00
#SBATCH --output=%x.o%j
#SBATCH --error=%x.e%j
#

## YOUR CODE GOES HERE (load the modules and do the calculations)
## Sample code:

# Make sure it's the same module used at compile time
module load intel

# Run MPI calculation
mpirun -np $SLURM_NTASKS ./my_program.x
```

MPI Job – With OpenMP

This is an example of a mixed MPI/OpenMP job, with each MPI process having 8 threads at its disposal.

Listing 4: Parallel job asking for 16 MPI tasks, each with 8 (OpenMP) threads and 7.92 GB of memory for a total of 12 hours on `regular2`. The output and error filenames are in TORQUE style.

```
#!/usr/bin/env bash
#
#SBATCH --job-name=Parallel_Job_OpenMP
#SBATCH --mail-type=ALL
#SBATCH --mail-user=jdoe@sissa.it
#
#SBATCH --ntasks=16
#SBATCH --cpus-per-task=8
#
#SBATCH --mem-per-cpu=990mb
#
#SBATCH --partition=regular2
#SBATCH --time=12:00:00
#SBATCH --output=%x.o%j
#SBATCH --error=%x.e%j
#

## YOUR CODE GOES HERE (load the modules and do the calculations)
```

(continues on next page)

(continued from previous page)

```
## Sample code:

# Make sure it's the same module used at compile time
module load intel

# Optionally set the number of OpenMP threads as you want
# Example:     export OMP_NUM_THREADS=$((${SLURM_CPUS_PER_TASK}/2))

# Print info on current number of OpenMP threads
"Using "$(($OMP_NUM_THREADS))" OpenMP threads."

# Run MPI calculation
mpirun -np $SLURM_NTASKS ./my_program.x
```

Note: You can add the option `#SBATCH --ntasks-per-core=1` to make sure that 8 CPUs correspond to actual 4 physical cores.

Note: These parameters were specifically tailored for nodes in `regular2`. Each of the nodes in this queue has 64 CPUs and 63.5 GB of memory, so asking for 8 CPUs/task and 990 MB per CPU means that 8 tasks are able to saturate a node. In fact, 990 MB x 8 CPUs x 8 tasks = 63360 MB (the max on these nodes is 63500 MB) and 8 CPUs x 8 tasks = 64 CPUs (equal to the max). So, SLURM would normally allocate 2 nodes to the job; however, if you prepare the script in this way, SLURM could also "fill" the holes left by other jobs and start your job sooner, instead of waiting to have 2 full nodes to employ for your calculation.

This is the same example, but with an explicit setup to ask for 2 full nodes.

Listing 5: Parallel job asking for 16 MPI tasks distributed among 2 nodes, each task with 8 (OpenMP) threads and 7.92 GB of memory for a total of 12 hours on `regular2`. The output and error filenames are in TORQUE style.

```
#!/usr/bin/env bash
#
#SBATCH --job-name=Parallel_Job_OpenMP
#SBATCH --mail-type=ALL
#SBATCH --mail-user=jdoe@sissa.it
#
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=8
#SBATCH --cpus-per-task=8
#SBATCH --exclusive
#
#SBATCH --mem=0
#
#SBATCH --partition=regular2
#SBATCH --time=12:00:00
#SBATCH --output=%x.o%j
#SBATCH --error=%x.e%j
```

(continues on next page)

(continued from previous page)

```
#

## YOUR CODE GOES HERE (load the modules and do the calculations)
## Sample code:

# Make sure it's the same module used at compile time
module load intel

# Optionally set the number of OpenMP threads as you want
# Example:     export OMP_NUM_THREADS=$((${SLURM_CPUS_PER_TASK}/2))

# Print info on current number of OpenMP threads
"Using "$(($OMP_NUM_THREADS))" OpenMP threads."

# Run MPI calculation
mpirun -np $SLURM_NTASKS ./my_program.x
```

Note: The option `--mem=0` is a shorthand for asking all the available memory on each node. The `--exclusive` option makes sure that the nodes are not shared among other running jobs (even if these particular settings already saturate all the 2 nodes).

1.4.3 Job Arrays

Job arrays are a handy way to send multiple jobs that vary e.g. just by some parameters of the calculation. SLURM's documentation has a very well-written page on job arrays, I suggest you to take a look for more details and examples: https://slurm.schedmd.com/job_array.html. Here I'll just show a couple of examples.

A job array is specified via the `--array=` option (see *Partition, Walltime and Output*), that takes a range of integers as `<value>`. This range can be specified as an interval, e.g. `1-10` (numbers from 1 to 10), or as a sequence, e.g. `3,5,23`, or both, e.g. `1-5,13` (numbers from 1 to 5, then 13).

For example, if one uses the option `--array=1-5,13`, then SLURM will generate 6 different jobs, each of one containing the following environment variables:

Environment Variable	Value
`$SLURM_ARRAY_TASK_ID`	One of the following: `1,2,3,4,5,13`.
`$SLURM_ARRAY_TASK_COUNT`	6 (number of jobs in the array)
`$SLURM_ARRAY_TASK_MAX`	13 (max of given range)
`$SLURM_ARRAY_TASK_MIN`	1 (min of given range)

In other words, each of these 6 different jobs will have a variable `$SLURM_ARRAY_TASK_ID` containing one (and only one) of the numbers given to `--array`. This variable can then be used to generate one or more parameters of the simulation, in a way that's completely up to you.

Note: Array ranges can additionally be specified with a step. For example, to generate multiples of 3 up to 21, you can use `--array=0-21:3`.

Note: You can also specify a maximum number of jobs in that array that are allowed to run at the same time. For example, `--array=1-20%4` generates 20 jobs but only 4 of them are allowed to run at the same time.

Serial Job Array

Listing 6: Serial job array asking for 2 threads (bounded to 1 physical core), 990 MB of memory and 6 hours for each of the 32 jobs, on `regular2`. The output and error filenames are in TORQUE style.

```
#!/usr/bin/env bash
#
#SBATCH --job-name=Array_Job
#SBATCH --mail-type=ALL
#SBATCH --mail-user=jdoe@sissa.it
#
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=2
#SBATCH --ntasks-per-core=1
#
#SBATCH --mem-per-cpu=990mb
#
#SBATCH --array=1-32
#SBATCH --partition=regular2
#SBATCH --time=06:00:00
#SBATCH --output=%x.o%A-%a
#SBATCH --error=%x.e%A-%a
#

## YOUR CODE GOES HERE (load the modules and do the calculations)
## Sample code:

# Make sure it's the same module used at compile time
module load intel

# Calculate the parameter of the calculation based on the array index,
# e.g. in this case as 5 times the array index
PARAM=$((${SLURM_ARRAY_TASK_ID}*5))

# Run calculation
./my_program.x $PARAM
```

Note: This workload is based on the specifics of the regular2 nodes. With these numbers you should be able to occupy even just a single node, if it’s available; but hey, nonetheless you are running 32 calculations at the same time! □

1.4.4 Dependencies

SLURM allows the jobs dependency adding a simple flag to the `sbatch` command

```
$ sbatch --dependency=<type:jobid[:jobid][,type:jobid[:jobid]]> ...
```

Command	Meaning
`after:jobid`	job can begin after the specified jobs have started
`afterany:jobid`	job can begin after the specified jobs have terminated
`afternotok:jobid`	job can begin after the specified jobs have failed
`afterok:jobid`	job can begin after the specified jobs have run to completion with an exit code of zero (see the user guide for caveats).
`singleton`	jobs can begin execution after all previously launched jobs with the same name and user have ended.

A command that I found useful is the following. Suppose you have `n` jobs, each dependent from the previous one, and that for each `i`-th job you have a script called `job_i.sh`. Then you can use the following command to queue them all at once:

```
ID=($(sbatch job_1.sh ))
for i in $(seq 2 n ); do
   ID=($(sbatch --dependency=<type>:${ID[3]} job_$i.sh ))
done
```

where <type> is one of `after`, `afterany`, `afternotok` or `afterok`.

1.5 Extra Tips

1.5.1 Run Interactive Jobs

You can ask for a Bash terminal on one or more nodes and then execute all the commands you want. This kind of job is called an **interactive job**.

Interactive jobs are not resource efficient, so their usage should be **limited** to those cases in which it's not possible to use a job script; for example, you might need to interactively debug a code running on GPUs.

As an example, this command asks for 1 CPU on a node either in regular1 or in regular2, for 30 minutes:

```
$ srun --partition=regular1,regular2 --time=00:30:00 --ntasks=1 --cpus-per-
↪task=1 --pty bash -i
```

`srun` supports a set of options similar to `sbatch`; for more info, refer to the manual page: https://slurm.schedmd.com/srun.html.

1.5.2 Run Quick, High-Priority Jobs

You can run a single, **really high-priority** job (e.g. for debugging) by using the `fastlane` QoS (Quality of Service), i.e.

```
$ sbatch --qos=fastlane send_job.sh
```

Warning: Using the *fastlane* QoS is **not free**, in the sense that you will be "punished" by lowering the proprity of your subsequent jobs. So, use it **only if you really need it!**

You can get a list of the available QoS's via:

```
$ sacctmgr show qos format=name,priority
      Name   Priority
---------- ----------
    normal          0
  fastlane   10000000
      wide          0
      long          0
 priority1     500000
```

1.5.3 Disable Hyper-Threading

If you want to **totally disable** Hyper-Threading, you can use

```
$ sbatch --hint=nomultithread --cpu-bind=cores send_job.sh
```

1.5.4 Automatic Login

If you're on a **trusted computer**, you can avoid entering your password every time you login in Ulysses.

First, generate an SSH keypair via:

```
$ ssh-keygen
```

Then, upload your credentials to Ulysses:

```
$ ssh-copy-id username@frontend2.hpc.sissa.it
```

You'll be asked for your password for the last time. □

You can further shorten the login procedure by opening (or creating) the file `~/.ssh/config` and adding the following lines (replace `username` with your SISSA username and `sissacluster2` with the name you prefer):

```
Host sissacluster2
   User username
   HostName frontend2.hpc.sissa.it
   IdentityFile ~/.ssh/id_rsa
   ServerAliveInterval 120
   ServerAliveCountMax 60
```

Then, in order to login, you will just need

```
$ ssh sissacluster2
```

An even shorter way to login is then to open or create the file `~/.bash_profile` and, at the end, add the following line (replace `cluster2` with some name you like):

```
alias cluster2='ssh sissacluster2'
```

At this point, logging in to Ulysses becomes a matter of executing the command

```
$ cluster2
```

in a terminal (you might need to close and reopen the terminal, first).

1.5.5 Explore Files in a User-Friendly Way

Warning: As long as Ulysses 2 is in beta, you still have to be under VPN!

You can read and write files on the cluster without necessarily using `scp` or `sftp` from the command line, so that you can navigate files as if they were on a normal folder on your computer.

There are two main ways to do that:

- by using the SFTP protocol though you system's native file browser
- by using SSHFS

SFTP - Linux

Basically every major modern file browser has SFTP support. This guide uses Nautilus (the default file browser) from a recent version of Ubuntu; it should be similar in other distributions.

SFTP should be installed by default. If for some reason it's not, fire up a terminal and install it via

```
$ sudo apt install gvfs-backends
```

Then, open the file browser and click on "Other Locations" on the sidebar:

You should see a "Connect to Server" field at the bottom of the window; fill it with the following SFTP address

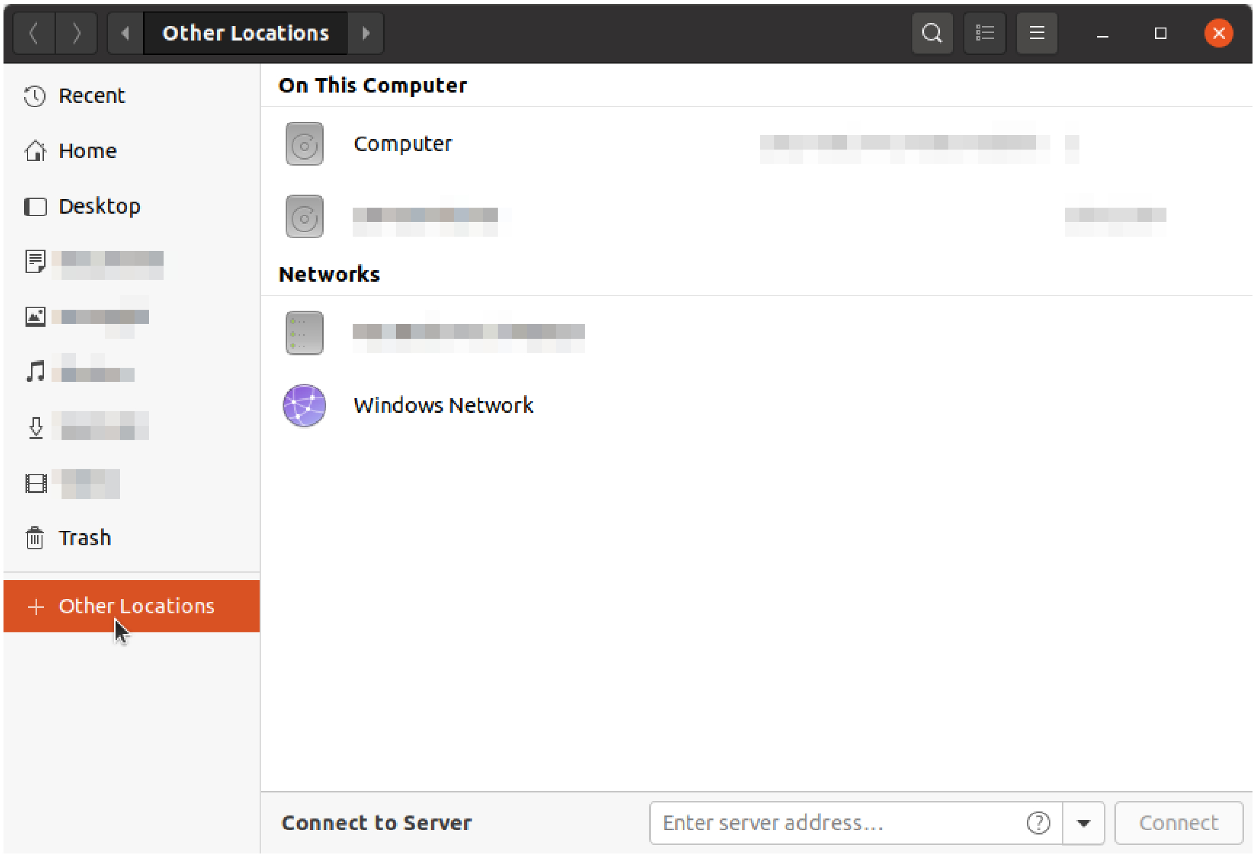
```
sftp://user@frontend2.hpc.sissa.it/
```

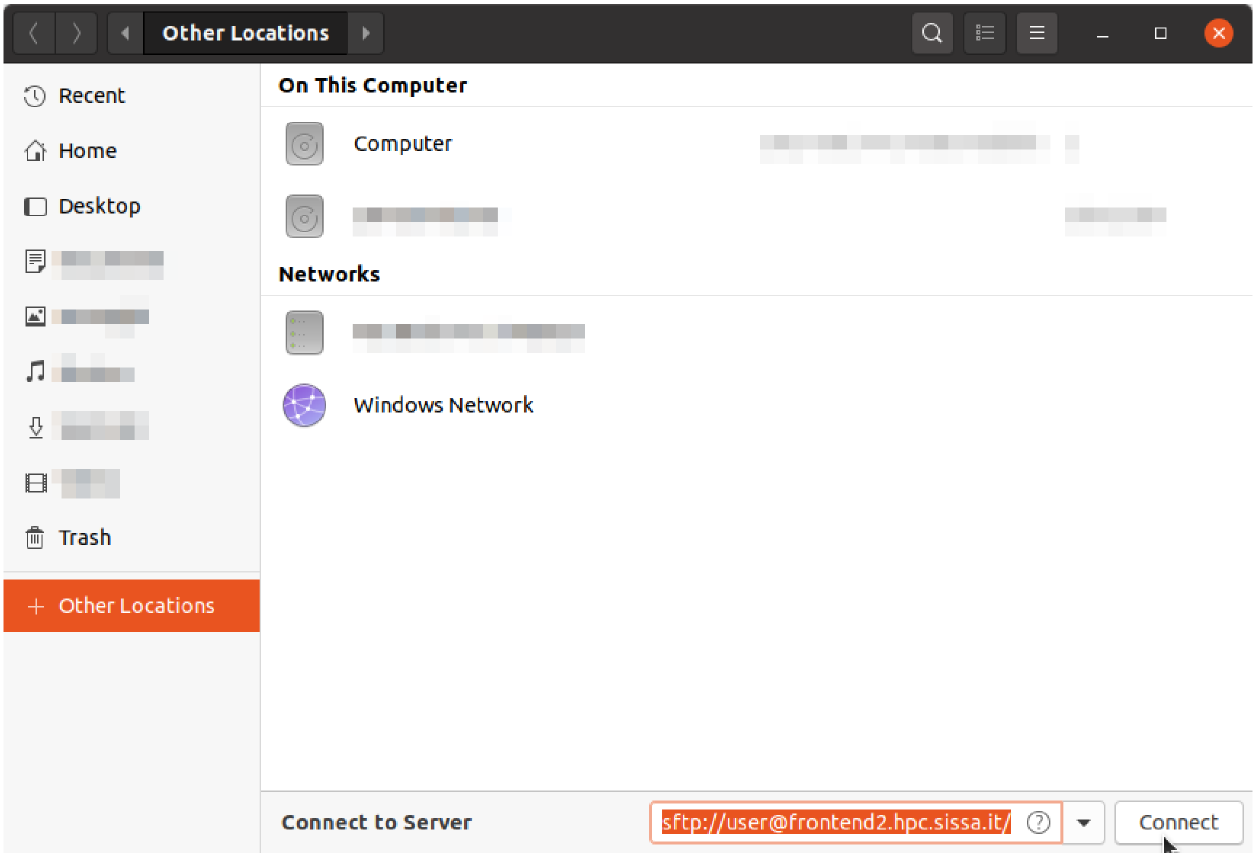
where `user` is your SISSA username. Then, click "Connect":

You might see a message warning about connecting to the server for the first time. If that's the case, you can click on "Log in anyway".

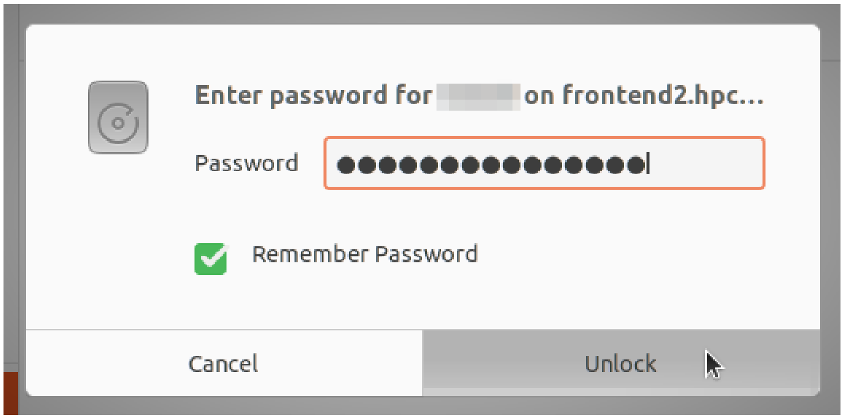
Then, just insert your password and check the box "Remember Password" so that you don't have to insert it next time:

At this point, you should see the Ulysses' folder structure.





sftp://user@frontend2.hpc.sissa.it/

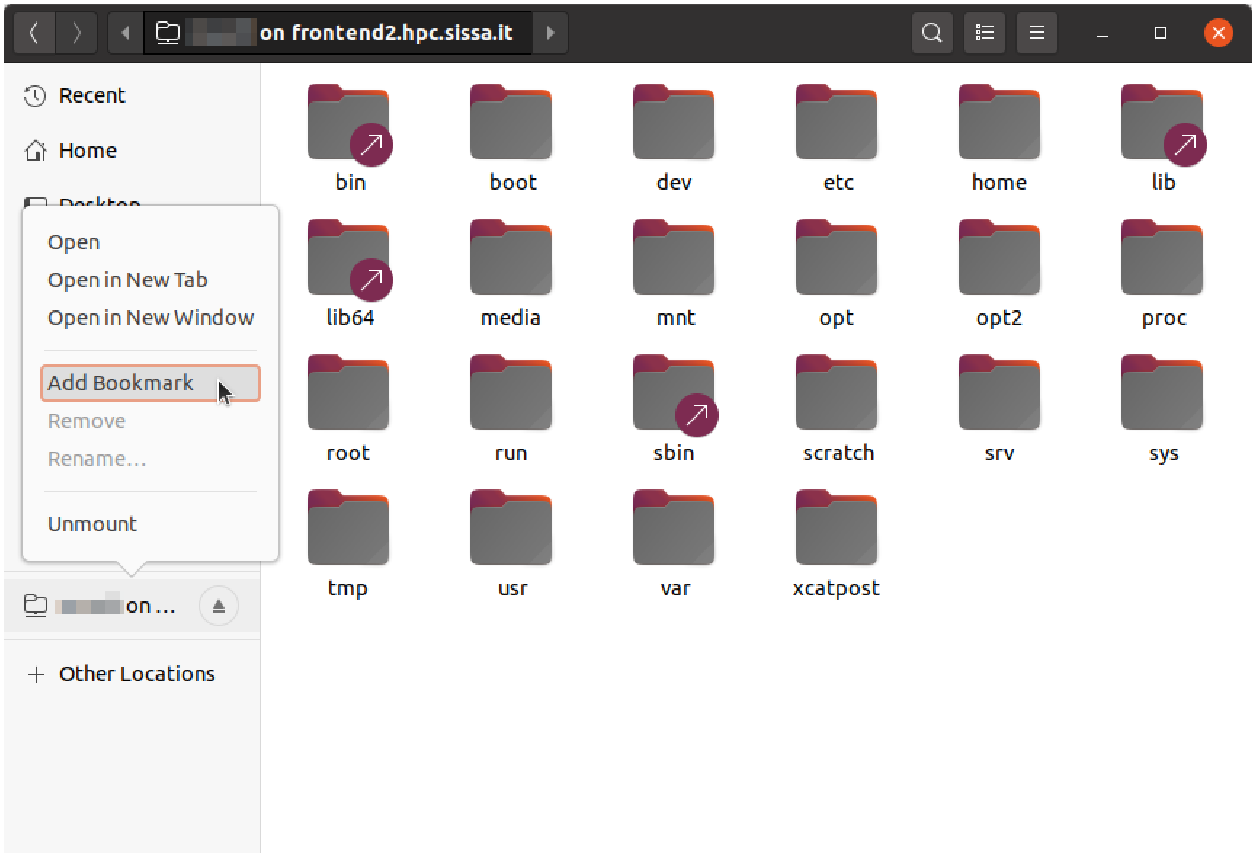


Enter password for [redacted] on frontend2.hpc…

Password ●●●●●●●●●●●●●●●●

☑ Remember Password

First of all let’s create a bookmark so you don’t have to repeat this procedure next time. Just go to the entry in the left sidebar that says *user on frontend2.hpc.sissa.it*, right-click on it and select “Add Bookmark”.

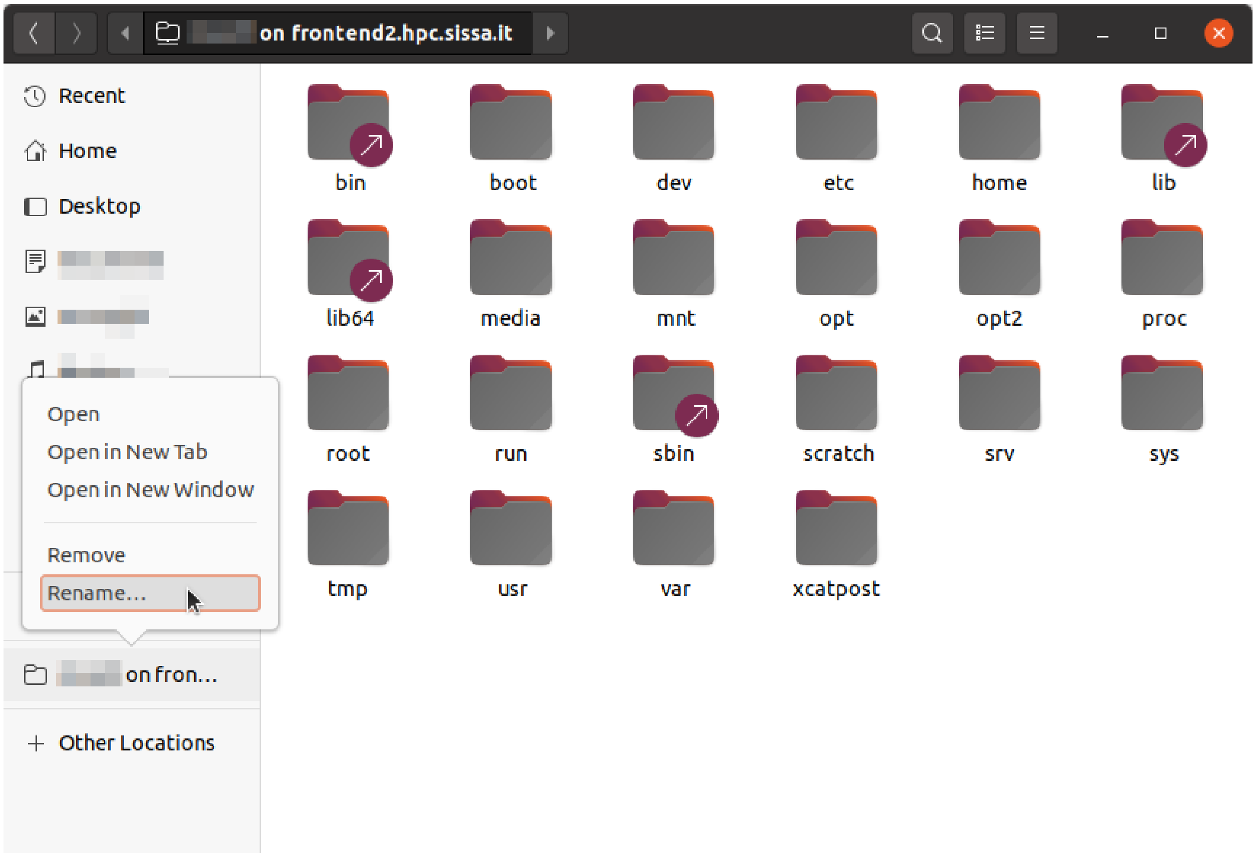


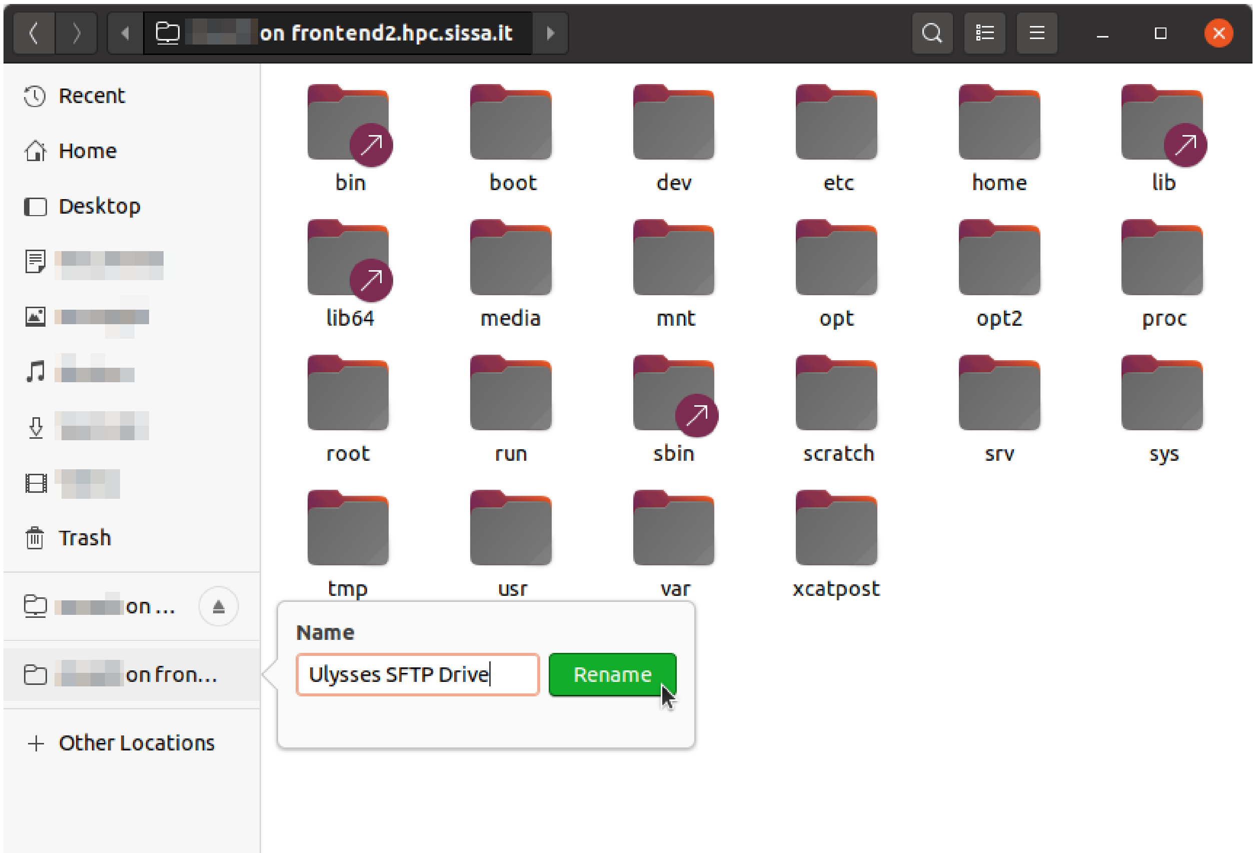
You should see a new entry with the same name in the left sidebar; just right-click on this new entry and select “Rename”:

Give it the name you want, for example “Ulysses SFTP Drive”:

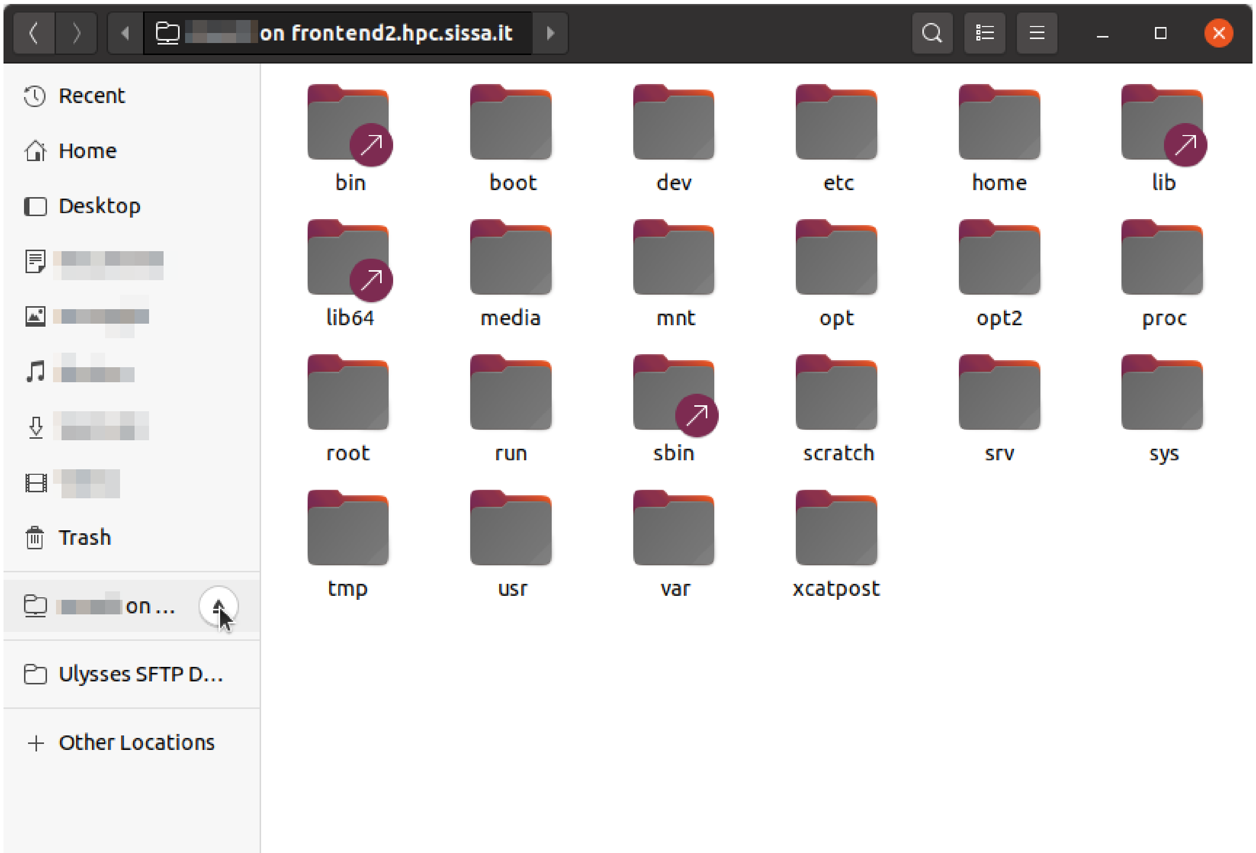
Then, the next time you want to access the cluster storage, just click on the “Ulysses SFTP Drive” bookmark in the left panel.

When you’ve finished you can safely unmount the cluster’s storage by clicking on the “Eject” icon:





Name: Ulysses SFTP Drive

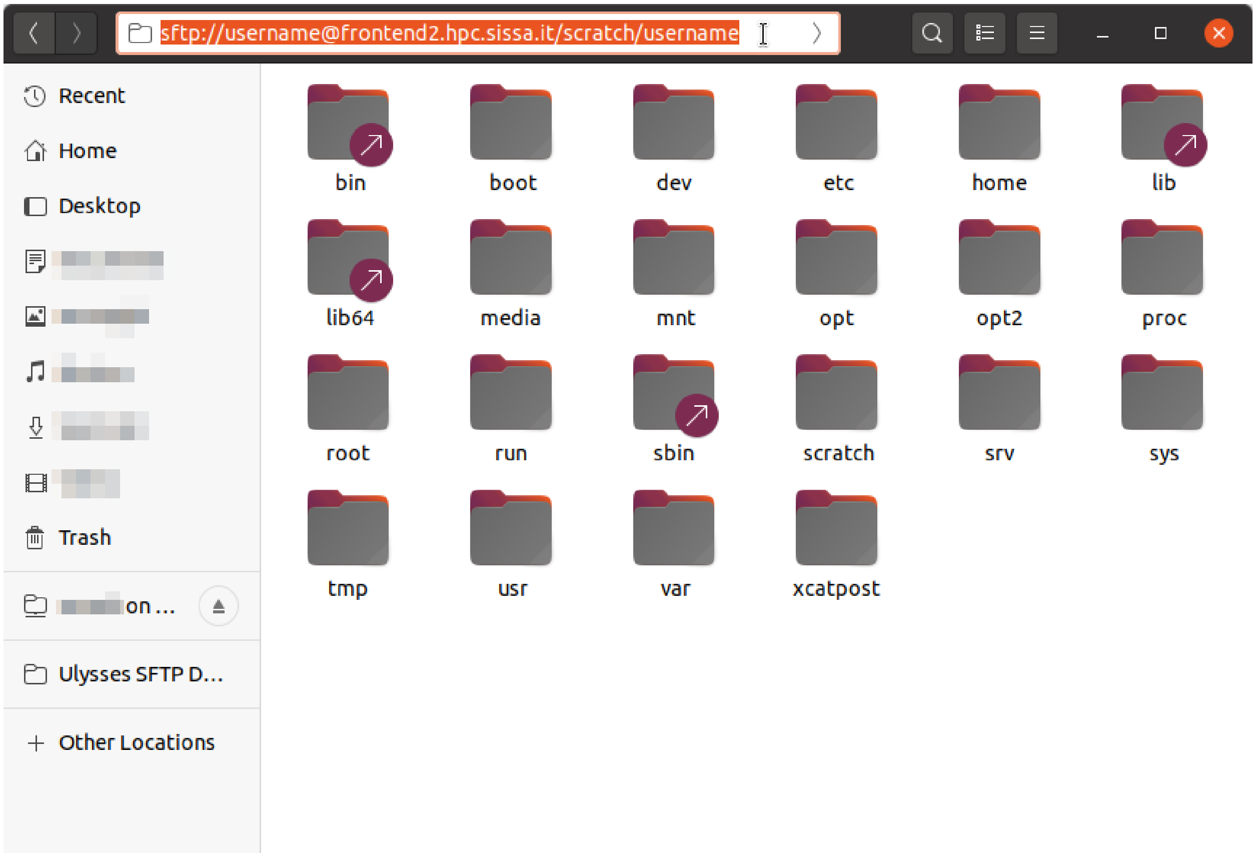


Note: The bookmark brings you by default in your home folder. If you want to change directory without navigating the full tree (which can be slow), just press "Ctrl + L" and add the path you want to go to at the end of the address. For example, if you want to visit the `/scratch/username` folder on Ulysses, where `username` is your SISSA username, you can put the following address in the top bar `sftp://username@frontend2.hpc.sissa.it/scratch/username`:

Warning: Mounting a folder via the SFTP protocol doesn't make it behave *exactly* as a local folder. For example, you cannot open a *local* terminal in it. However, you should be able to right-click into the folder and see an option "Open in Remote Terminal" that opens a **remote** terminal in that folder, i.e. connects to Ulysses via SSH and moves the terminal to that folder. This means that you cannot copy/paste or move files from Ulysses to your local computer via the terminal; you can do that only via the graphical interface. For a more flexible solution that allows you to open *local* terminals, you might want to take a look at the section *SSHFS – Linux*.

SFTP - MacOS

Finder does not natively support SFTP. You have to resort to a third-party solution or use SSHFS (see the section *SSHFS – MacOS*).



sftp://username@frontend2.hpc.sissa.it/scratch/username

SFTP – Windows

Windows Explorer does not natively support SFTP. You have to resort to a third-party solution or use SSHFS (see the section *SSHFS – Windows*), though even SSHFS requires installing additional software.

SSHFS – Linux

SSHFS allows you to use a local folder as an entry point (called "mount" point) for a remote folder, i.e. it allows you to show the contents of a remote folder `B` as if they were into a local folder `A`.

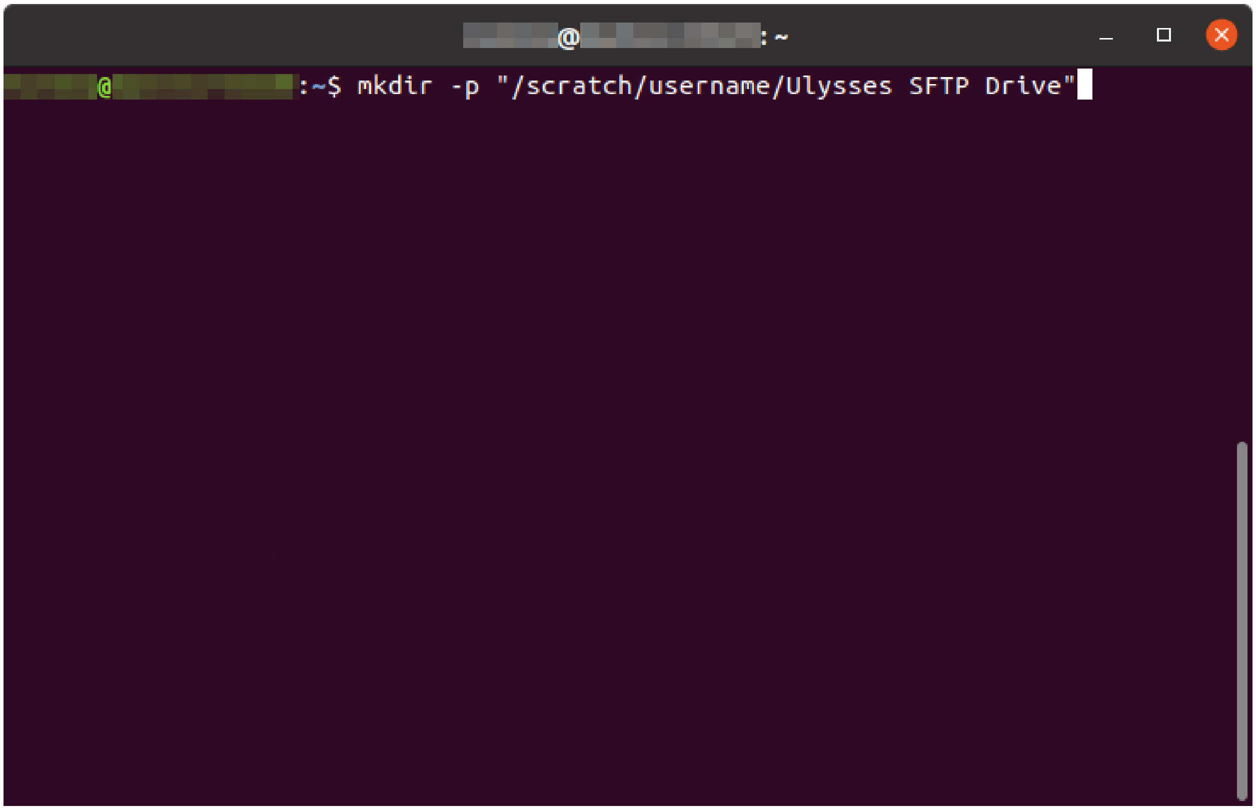
Again, we'll see Ubuntu as an example; the procedure, however, should be similar on other systems as well.

First of all, if you don't have SSHFS, install it via

```
$ sudo apt install sshfs
```

Then, create the folder that you want to use as a mount point. This folder can be in whatever place you like: in "Documents", in "Home", etc.; the only requirement is that you have to have the permissions to create the folder. On SISSA workstations, I suggest you to create a subfolder in the */scratch/username/* folder of your workstation; **don't** use the home folder!

You can create the folder either via the graphical file browser or via the terminal, for example I decided to make a folder `` `"/scratch/username/Ulysses SFTP Drive": ``



```
[redacted]@[redacted]:~$ mkdir -p "/scratch/username/Ulysses SFTP Drive"
```

Then, you can "connect" this local folder to the cluster storage via the following command:

```
$ sshfs username@frontend2.hpc.sissa.it:/ /scratch/username/Ulysses\ SFTP\␣
↪Drive -o reconnect -o fsname="Ulysses SFTP Drive"
```

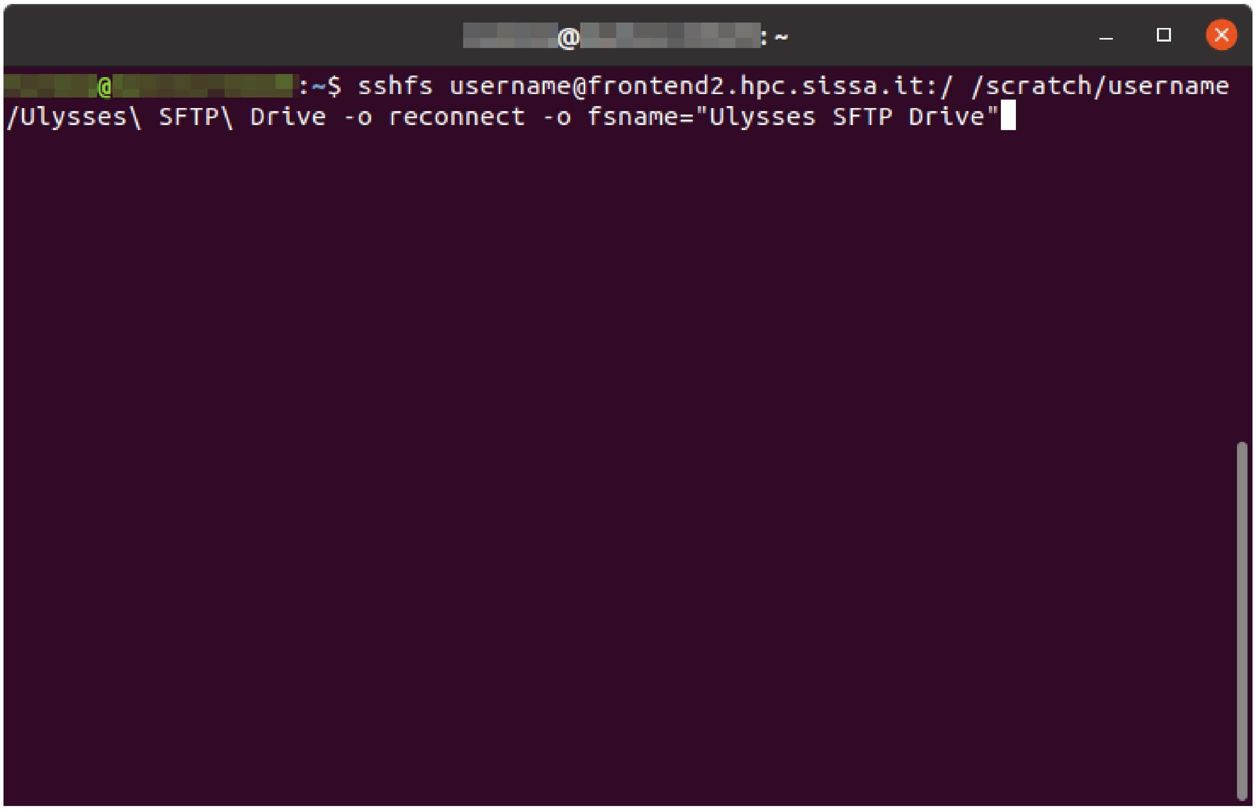
where `username` is your SISSA username, `/scratch/username/Ulysses\ SFTP\ Drive` has to be replaced with the path of the folder you've created, and `"Ulysses SFTP Drive"` is just a name that you can change as you want:

At this point you should see a device called "Ulysses SFTP Drive" in the left panel of the fole browser. Right-click on it and select "Add Bookmark", so that after you eject it you can just click on the bookmark to connect again.

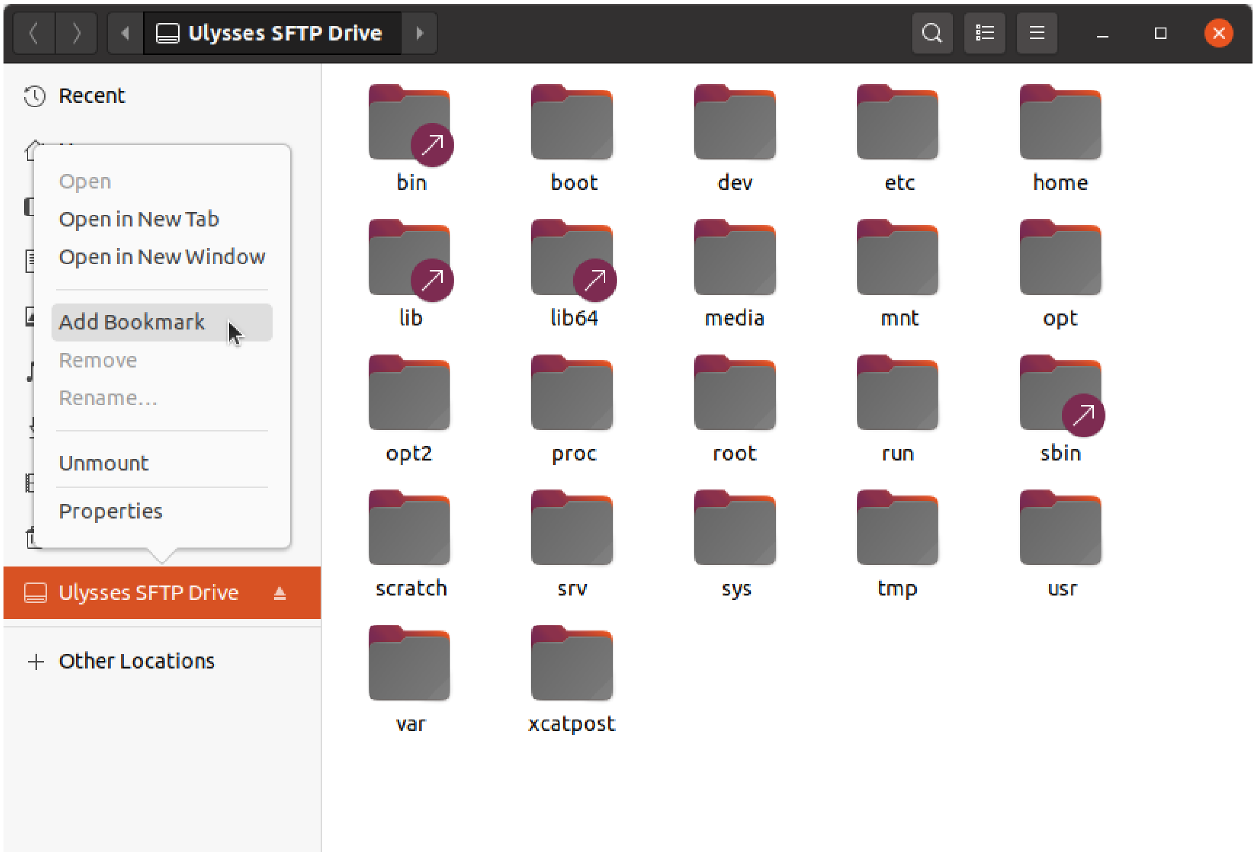
Note: If you want to change directory without navigating the full tree (which can be slow), just press "Ctrl + L" and add the remote path you want to go to **after** the path of the local folder. For example, if you want to visit the `/home/username` folder on Ulysses, where `username` is your SISSA username, and the local mount point is `"/scratch/username/Ulysses SFTP Drive"`, you have to use the following address in the top bar `/scratch/username/Ulysses SFTP Drive/home/username`:

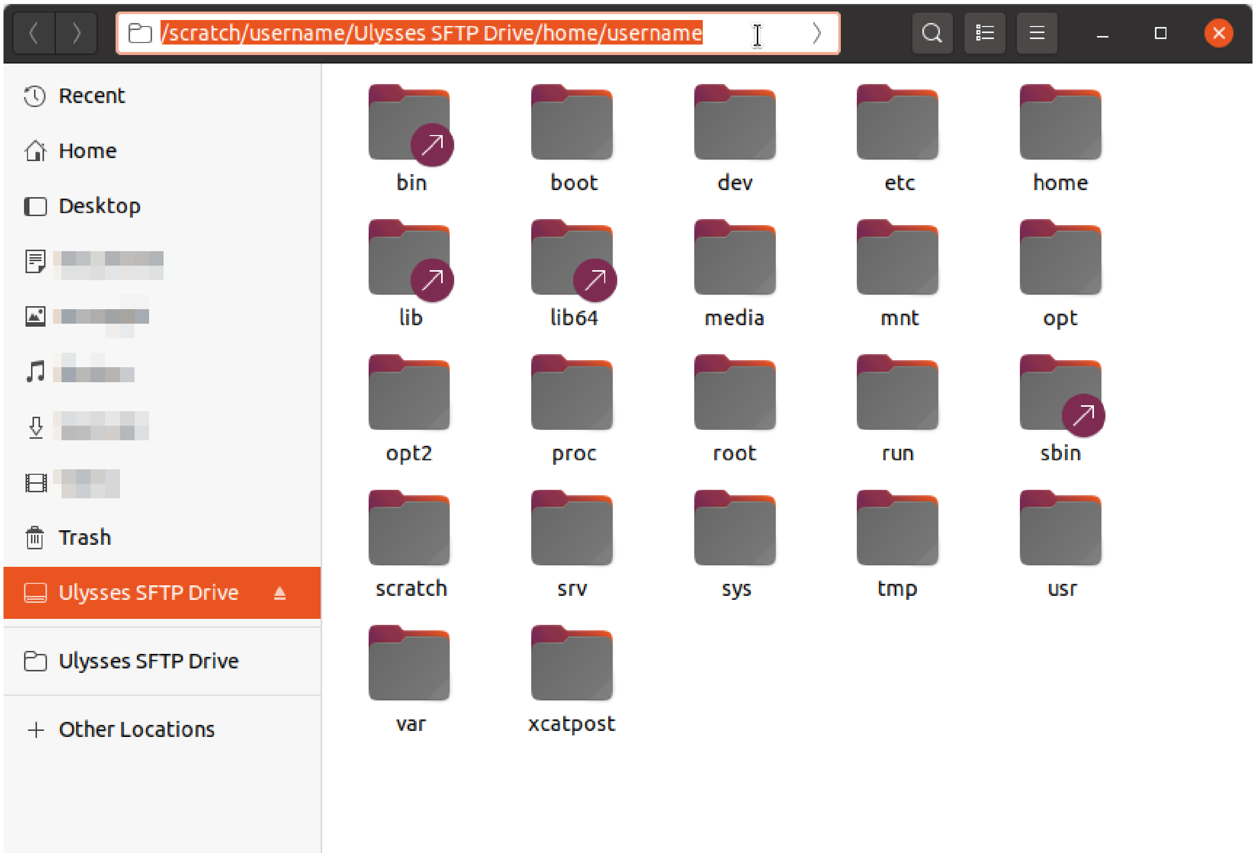
Warning: Unmounting the volume via the "Eject" icon might not work (it complains about permissions). It's a known bug; if it happens, you can unmount the volume by opening a terminal and using

```
$ fusermount -u "/scratch/username/Ulysses SFTP Drive"
```



```
[redacted]@[redacted]: ~
[redacted]@[redacted]:~$ sshfs username@frontend2.hpc.sissa.it:/ /scratch/username
/Ulysses\ SFTP\ Drive -o reconnect -o fsname="Ulysses SFTP Drive"
```





/scratch/username/Ulysses SFTP Drive/home/username

where `"/scratch/username/Ulysses SFTP Drive"` has to be replace with your local mount point.

SSHFS - MacOS

The steps for MacOS are similar to the ones outlined in *SSHFS - Linux* (have a read), with the notable exception that **you don't have to create the local folder**.

To install SSHFS on MacOS, first install HomeBrew. Then, fire up a terminal and execute

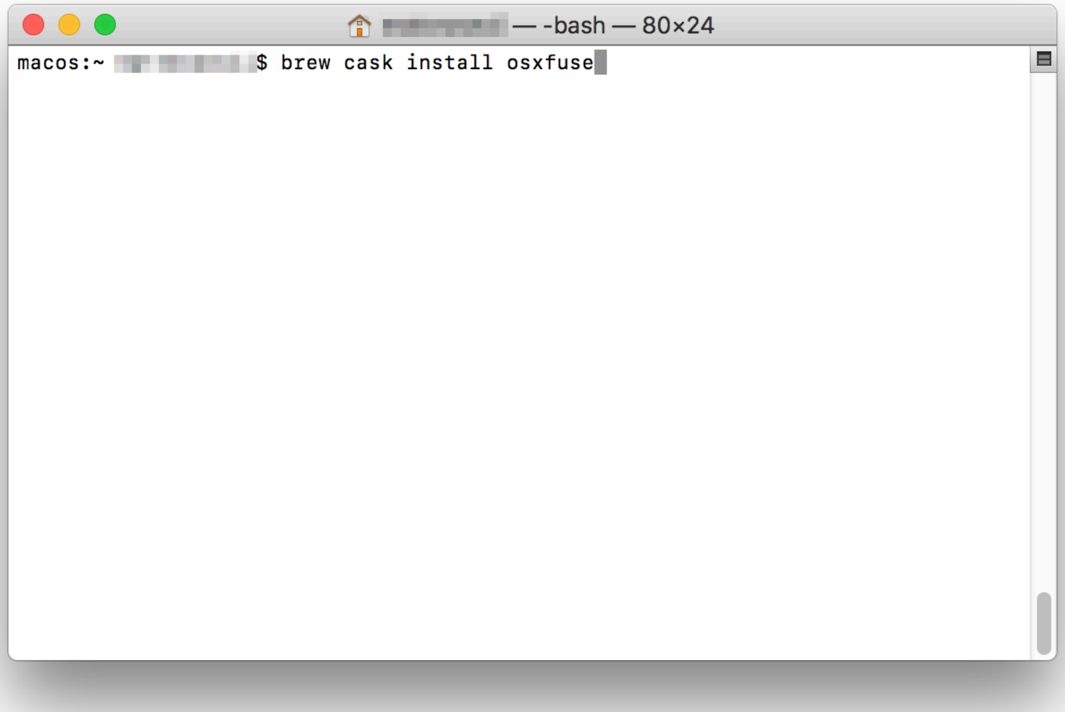
```
$ brew cask install osxfuse
```

Then reboot, open up a terminal again and install sshfs via:

```
$ brew install sshfs
```

You can finally mount Ulysses' storage, for example in :ref:`/Volumes/Ulysses\ SFTP\ Drive`, by executing in a terminal:

```
$ sshfs username@frontend2.hpc.sissa.it:/ /Volumes/Ulysses\ SFTP\ Drive -o local -o volname="Ulysses SFTP Drive" -o auto_cache,reconnect,defer_permissions,noappledouble
```



```
macos:~ [redacted]$ brew cask install osxfuse
```

Note: Answer yes to the fingerprint message and insert your password when asked.

Note: In order to avoid repeating this command every time, you can put it in a Bash script and just execute the Bash script when you want to connect.

The first time you use `sshfs` you might get a message like this one:

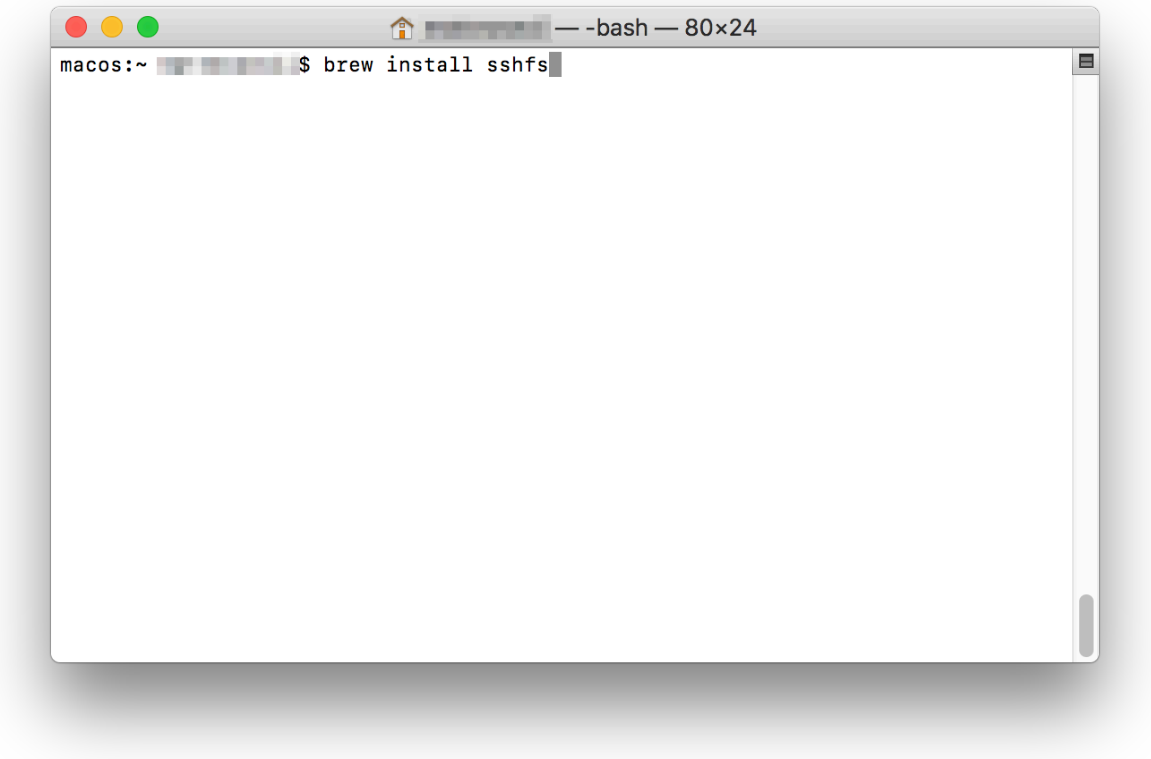
Just click on "Open System Preferences" to get to this screen

and then click on "Allow" at the bottom of the window. After you've done this, **repeat** the `sshfs` command again and you should be fine.

You can find the drive on your desktop or on "My Computer":

Note: You can skip the manual navigation of the tree in a way similar to the "Ctrl + L" command described for Ubuntu. In MacOS, you have to use "Command + Maiusc + G" and insert the address (e.g. referring to the same example above) `/Volumes/Ulysses SFTP Drive/home/username`.

Unmounting should work by just right-clicking on the drive and select "Eject", or by using the "Eject" icon in the Finder. If that doesn't happen or if you receive an error about the position already existing when you use the `sshfs` command again, you can manually force-unmount via the following terminal command:



[redacted] — -bash — 80×24

```
macos:~ [redacted]$ brew install sshfs
```

```
$ umount -f /Volumes/Ulysses\ SFTP\ Drive
```

Note that the command is *umount*, **not** *unmount*.

Extra Tip. On MacOS, you can customize the appearance of the mounted drive. For example you can create or download an icon that you like and let's say that you've stored it as `/Users/YOURNAME/.local/share/icons/sissa_drive.icns`. Then, in order to use it for your drive, add the following option to the `sshfs` command above:

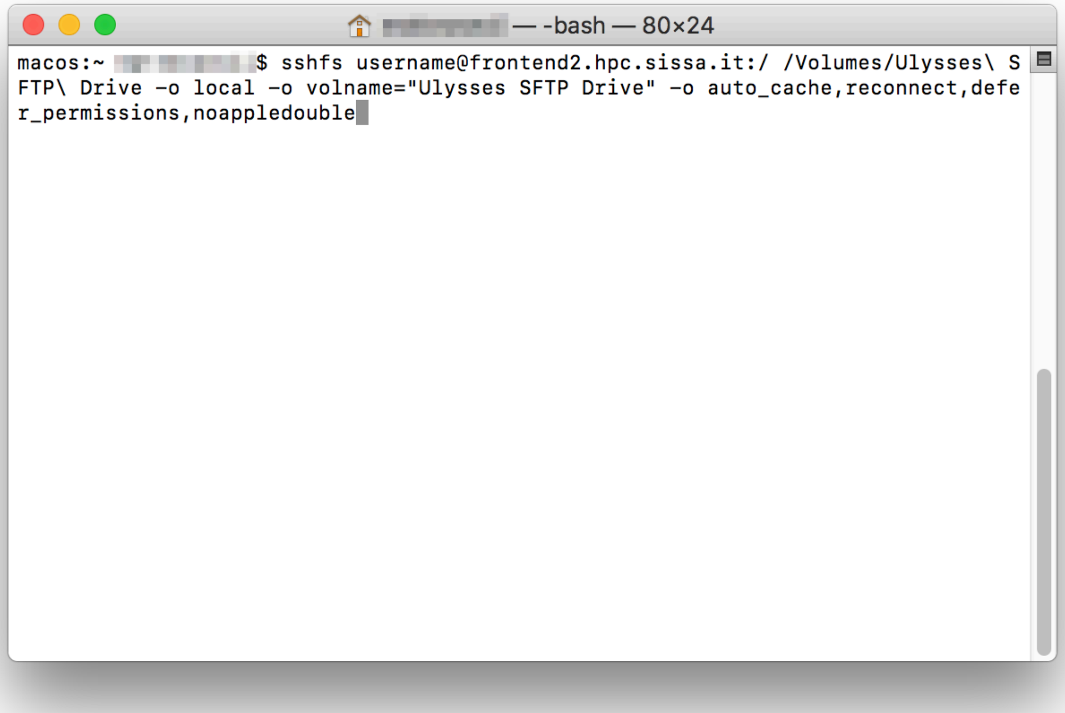
```
$ -o modules=volicon,iconpath=/Users/YOURNAME/.local/share/icons/sissa_drive.
↪icns
```

SSHFS – Windows

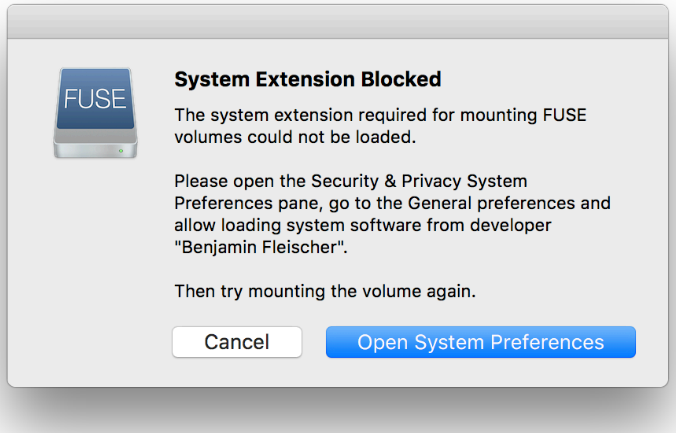
To make SSHFS work in Windows, we need to install some third-party drivers. I've tested the open-source solution below and it seems to work fine on Windows 10.

- First, install WinFsp. Keep all the default options.
- Then, install the SSHFS-Win as well. If you receive a warning about the computer being protected by Windows, click on "More info" and then on "Run anyway". Keep all the default options.

At this point, open Windows File Explorer, right-click on "This PC" in the left panel and then "Map Network Drive". Choose a drive letter to map to Ulysses (I've chosen `U:` as "Ulysses" ▯) and use the following address:



```
macos:~ [redacted]$ sshfs username@frontend2.hpc.sissa.it:/ /Volumes/Ulysses\ S
FTP\ Drive -o local -o volname="Ulysses SFTP Drive" -o auto_cache,reconnect,defe
r_permissions,noappledouble
```

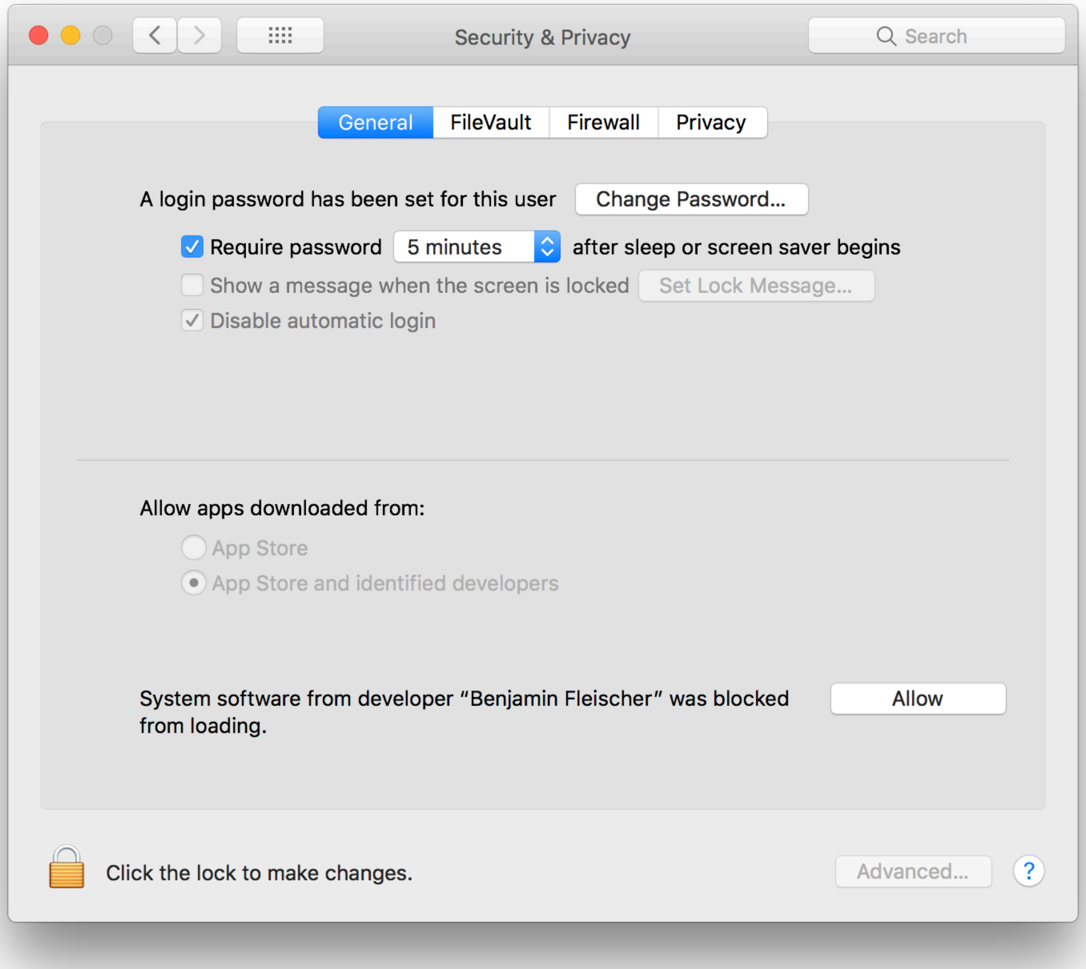


System Extension Blocked

The system extension required for mounting FUSE volumes could not be loaded.

Please open the Security & Privacy System Preferences pane, go to the General preferences and allow loading system software from developer "Benjamin Fleischer".

Then try mounting the volume again.



A login password has been set for this user

☑ Require password 5 minutes after sleep or screen saver begins

□ Show a message when the screen is locked

☑ Disable automatic login

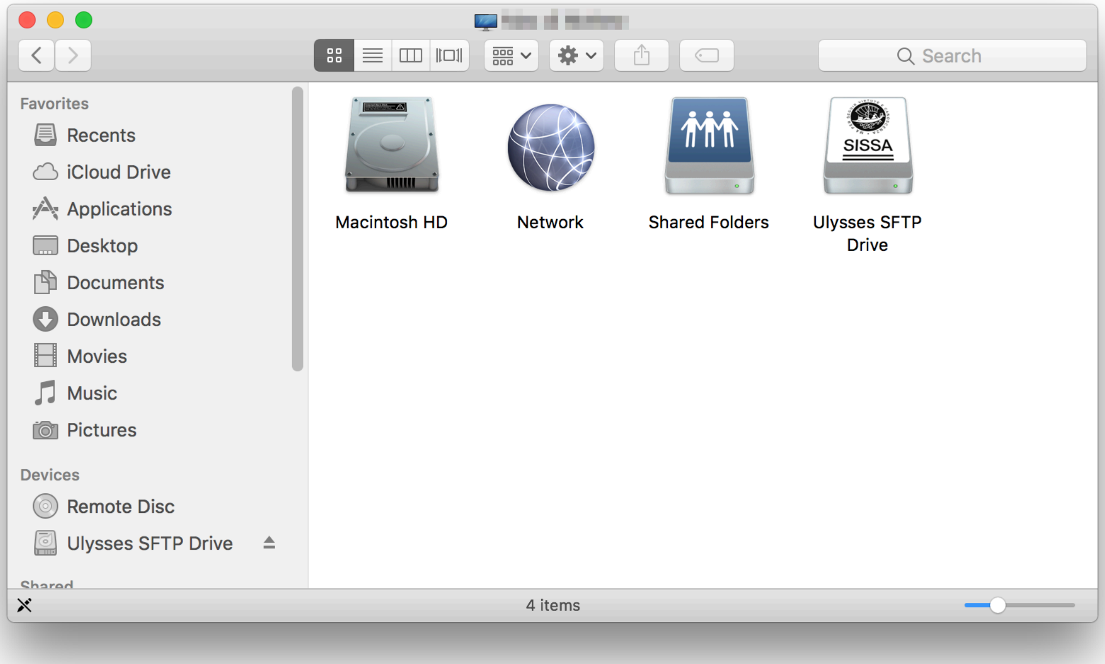
Allow apps downloaded from:

○ App Store

● App Store and identified developers

System software from developer “Benjamin Fleischer” was blocked from loading.

Click the lock to make changes.



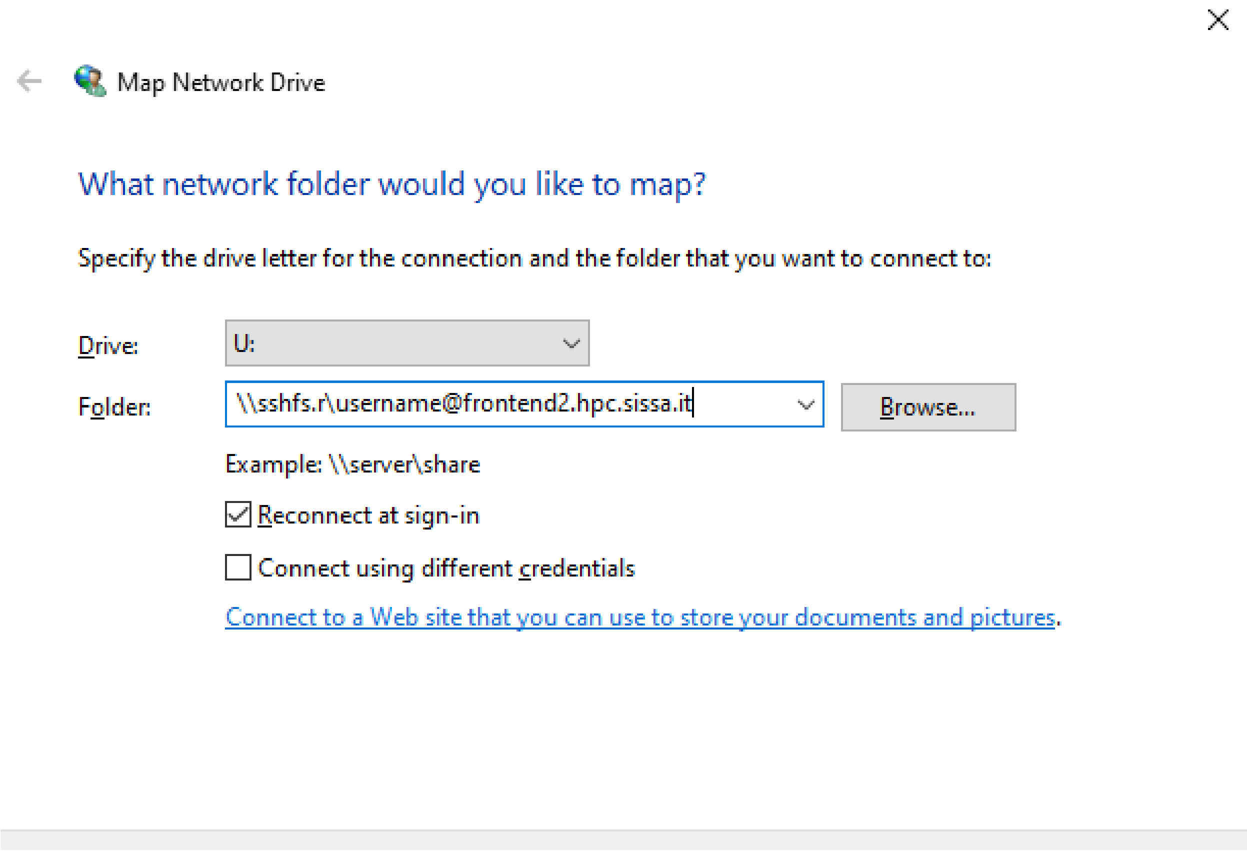
```
\\sshfs.r\username@frontend2.hpc.sissa.it
```

where `username` is your SISSA username. You can choose whether to reconnect at sign-in or not.

You will be prompted with user/pass:

and then you should be set! □

Warning: On Windows, folder paths use a \ instead of the / used on Linux and MacOS!



Map Network Drive

What network folder would you like to map?

Specify the drive letter for the connection and the folder that you want to connect to:

Drive: U:

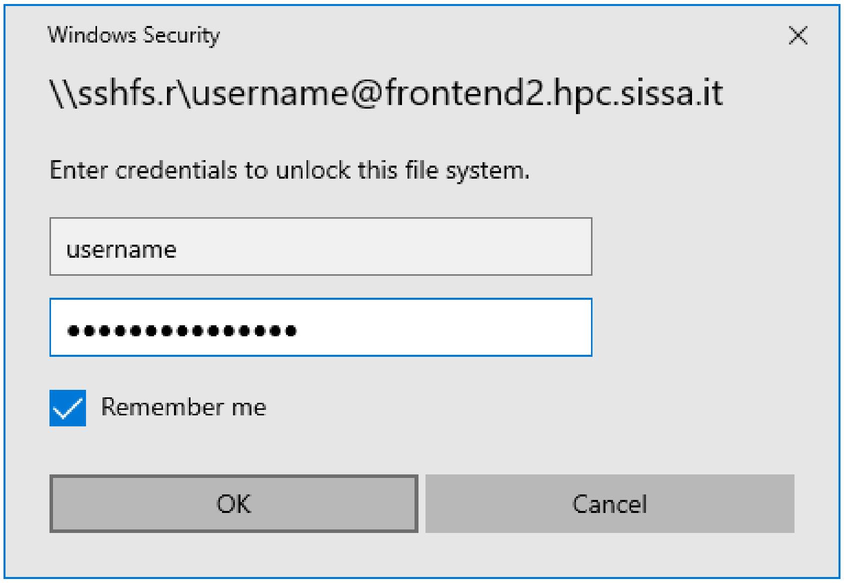
Folder: \\sshfs.r\username@frontend2.hpc.sissa.it

Example: \\server\share

☑ Reconnect at sign-in

□ Connect using different credentials

Connect to a Web site that you can use to store your documents and pictures.



Windows Security

\\sshfs.r\username@frontend2.hpc.sissa.it

Enter credentials to unlock this file system.

username

●●●●●●●●●●●●●●●

☑ Remember me