INTRODUCTION TO SUPERCOMPUTING AT JSC

Andreas Smolenko	Benedikt Steinbusch	Alexandre Strube

22 – 25 November 2021

    - Supercomputing Support

INTRODUCTION

The largest computers used for computational science have exhibited an exponential increase in the rate of basic operations they can perform since at least the 1990s. For more than a decade, this growth has been enabled, not by increasing clock speeds of individual processing units, but by assembling systems that consist of ever greater numbers of processing units. Scientific applications that are meant to run on these systems are expected to orchestrate many of these computational units to collaborate on solving a given computational problem. Building these kinds of applications is called parallel programming. Parallel programming will only be touched on briefly in this course, but Jülich Supercomputing Centre (JSC) offers several courses that teach various techniques related to the topic.

Scientists who want to run applications, be they custom made or third-party, on these systems are expected to know how to use these systems. Working through this guide will teach you how to

- access the systems available at JSC,
- navigate the file system,
- find pre-installed software,
- build your own software, and finally
- run software.

ACCESS

This chapter will teach you how to log in to one of the systems at JSC.

Getting a JSC account

A basic prerequisite to get acces to the HPC system and other services at JSC is a JSC account. If you do not already have an account (they have the form `<family name><number>`, e.g. `steinbusch1`), one can be created through JSC's user portal JuDoor (click the *Register* button).

Joining a compute time project

To be allowed to log in to an HPC system, your JSC account needs to be a member of a computing time project that has an active budget on the system. This is the case if

- you have successfully applied for test computing time for a test project and are now the principal investigator (PI) of your own project, or

- you have successfully applied for computing time during one of our calls for project proposals and are now the principal investigator (PI) of your own project, or
- you have gained access to a project either by being invited by the PI or project administrator (PA) or by being granted access upon requesting to join a project through JuDoor.

We have created a computing time project for this course with a project id of `training2126`. To join the project, log in to JuDoor and click *Join a project* under the *Projects* heading. Enter the project id and, if you want to, a message to remind the PI/PA (one of the instructors) why you should be allowed to join the project. Afterwards the PI/PA will be automatically informed about your join request and can add you to the different systems available in the project. As soon as you are unlocked for the system, the system entry will be shown on your JuDoor main page. You have to accept our Usage Agreement before you can continue with the next step.

Login procedure

Logging in to our systems is usually done through the Secure Shell (SSH) mechanism, although there are alternatives such as UNICORE and JupyterLab. Our SSH configuration uses an authentication mechanism based on public and private keys rather than passwords. A pair of public and private keys has to be generated on your personal computer. The private key has to be protected by a passphrase. The public key is then registered for access to the system through JuDoor.

NEVER SHARE YOUR *PRIVATE* KEY!!!

Several software packages can be used for logging in through SSH. The procedure is documented below for some popular choices:

- OpenSSH which is a popular choice on GNU/Linux, macOS, and other Unix-like operating systems
- PuTTY which is a popular choice on Windows

Generating a key pair with OpenSSH

OpenSSH is a set of command line tools, so open up a terminal. We suggest you start by creating a fresh pair of public and private key (a key pair). To generate a key pair enter the command below. The program asks for a passphrase. This passphrase is not used for authenticating to the remote system, but rather acts as an encryption key for the private part of the key pair stored on the local file system. In case the private key file is stolen by an attacker, they will not be able to use the key without knowing the passphrase, so make sure to use one that is hard to guess.

```
$ ssh-keygen -a 100 -t ed25519 -f ~/.ssh/id_ed25519
Generating public/private ed25519 key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/bsteinb/.ssh/id_ed25519.
Your public key has been saved in /Users/bsteinb/.ssh/id_ed25519.pub.
The key fingerprint is:
SHA256:tHin8v4j4cyVVe2BEWAinq/vlhFExupY+37s94216uA bsteinb@zam478
The key's randomart image is:
+--[ED25519 256]--+
|       .o+ o.o+. |
|      . +oo  ....|
|       o+    . ..|
|       =.o  .   .|
|      = S.oo     |
|     . +o+o      |
|      .=oo+.    .|
|       o*+oo.. oo|
|       .**+Eoo+o.|
+----[SHA256]-----+
```

If the designated output file (`~/.ssh/id_ed25519`) already exists, the program asks to overwrite it. This is probably *not* what you want, since you might be using the key contained therein. Change the output name by using the arguments `-f ~/.ssh/id_ed25519_jsc` instead of `-f ~/.ssh/id_ed25519`. If you do so, keep in mind that your keys are in a non-default location for the remainder of the course.

Print the contents of the public key to the terminal by entering:

```
$ cat ~/.ssh/id_ed25519.pub
ssh-ed25519 AAAAC3NzaC1lZDI1N [...] 6BRJMTyE4voyqJGm36P+ bsteinb@zam478
```

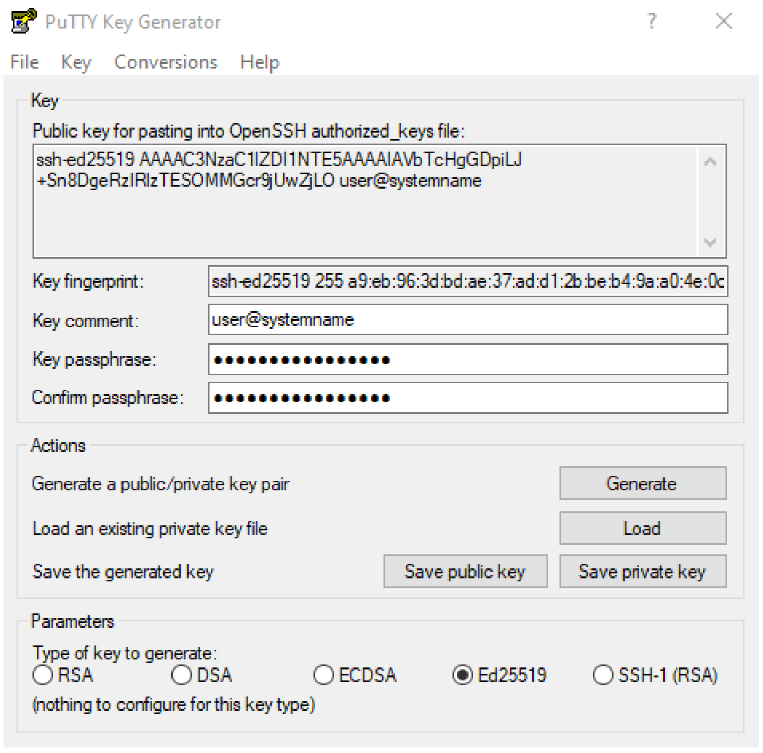
and copy it to the clipboard (do *not* copy the key above, that one is mine, yours will be different). Continue by uploading the public key to JuDoor.

Generating a key pair with PuTTY

Open `puttygen.exe` to generate a key pair. Select *Ed25519* as the key type then click *Generate* and follow the instructions of the program. Once the key has been generated, enter a strong passphrase that cannot be guessed easily. This passphrase is used to encrypt the key while it is stored on disk so that it cannot be used if it is stolen.

Click *Save private key* to save the private key to a `.ppk` file.

Now copy the contents of the field *Public key for pasting into OpenSSH authorized_keys file* to the clipboard.



Public key for pasting into OpenSSH authorized_keys file:

```
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIAVbTcHgGDpiLJ
+Sn8DgeRzIRlzTESOMMGcr9jUwZjLO user@systemname
```

Key fingerprint: ssh-ed25519 255 a9:eb:96:3d:bd:ae:37:ad:d1:2b:be:b4:9a:a0:4e:0c

Key comment: user@systemname

Key passphrase: ●●●●●●●●●●●●●●●●

Confirm passphrase: ●●●●●●●●●●●●●●●●

Generate a public/private key pair

Load an existing private key file

Save the generated key

Type of key to generate:
RSA DSA ECDSA [✓ Ed25519] SSH-1 (RSA)

(nothing to configure for this key type)

Key generation with PuTTY

Uploading the public key

Navigate to JuDoor and click on *Manage SSH-keys* next to the entry for the system you want to use under the *Systems* heading. Paste the public key into the form in the field labeled *Your public key and options string*, but do not submit yet. As a further security measure, you have to specify the systems that your log in attempts will come from. This is done via an additional `from`-clause on your public key, that can contain single IP-addresses and address ranges as well as host names and even wildcard patterns based on either of these.

Specifying a `from`-clause is relatively easy if you have access to a system with a fixed IP-address or an IP-address that changes dynamically, but comes from a range of addresses that can be specified concisely. This is typically the case for systems which are connected to university or research centre networks (even via VPN when working from home). For example, systems connected to the network of Forschungszentrum Jülich will be assigned an IP-address from the range `134.94.0.0/16`, so a valid `from`-clause would be `from="134.94.0.0/16"`. Other institutions will use different address ranges, you should be able to find these out from your institutions network operations centre.

Sometimes, patterns based on host names will work better than those based on IP addresses. For example, Forschungszentrum Jülich assigns host names that end in either `fz-juelich.de` or `kfa-juelich.de`, so a valid `from`-clause could also be `from="*.fz-juelich.de,*.kfa-juelich.de"` (notice how multiple patterns can be combined with a comma in between). Once again, the host names assigned by other institutions will be different. To some extent, this scheme also works for home internet access. Internet providers typically assign IP addresses dynamically drawing from fragmented pools that are hard to specify completely in terms of address ranges, but they often assign host names which follow a pattern that can be found out. The command `nslookup <your IP>` will tell you the host name assigned to your system by the provider (find out your IP either from the JuDoor key upload form or by asking a search engine "what is my ip"). This host name might look something like `2909a2-ip.nrw.provider.net`. Chop name components off the beginning and replace them with `*` to come up with a pattern, e.g. `*.nrw.provider.net`.

Add your `from`-clause in front of the public key you already pasted into the form. The result should be something like:

```
from="134.94.0.0/16" ssh-ed25519 AAAA [...]
```

Then click *Start upload of SSH keys*. It will take some time for the key you uploaded to JuDoor to be synched to the actual system. Eventually though, you will be able to log in. Once again, we have instructions for

- OpenSSH
- PuTTY

Logging in with OpenSSH

To log in with OpenSSH, enter the following command:

```
$ ssh -i ~/.ssh/id_ed25519 <account name>@<system name>.fz-juelich.de
```

(Remember to change the location of the key `~/.ssh/id_ed25519` if you saved it to a non-default location.) For example, if I wanted to log in to JUWELS Cluster it would be:

```
$ ssh steinbusch1@juwels-cluster.fz-juelich.de
```

The following table lists the host names of login nodes for the different systems. Pick the one you want to use.

System	Login node host name
JURECA DC and Booster	`jureca.fz-juelich.de`
JUWELS Cluster	`juwels-cluster.fz-juelich.de`

System	Login node host name
JUWELS Booster	`juwels-booster.fz-juelich.de`
JUSUF	`jusuf.fz-juelich.de`

When connection for the first time, OpenSSH will prompt you to confirm the server key fingerprint:

```
The authenticity of host 'jusuf.fz-juelich.de (134.94.0.184)' can't be established.
ECDSA key fingerprint is SHA256:tuswM7JtVcWNS5wRCVIfv1h4uRHReHIN77C4zTYaPjs.
Are you sure you want to continue connecting (yes/no/[fingerprint])?
```

JSC publishes SSH fingerprints for its systems through JuDoor. You can find them on the page you used to upload your public key. Either compare the keys or, in newer versions of OpenSSH, you can paste the fingerprint from JuDoor into the prompt to confirm it.

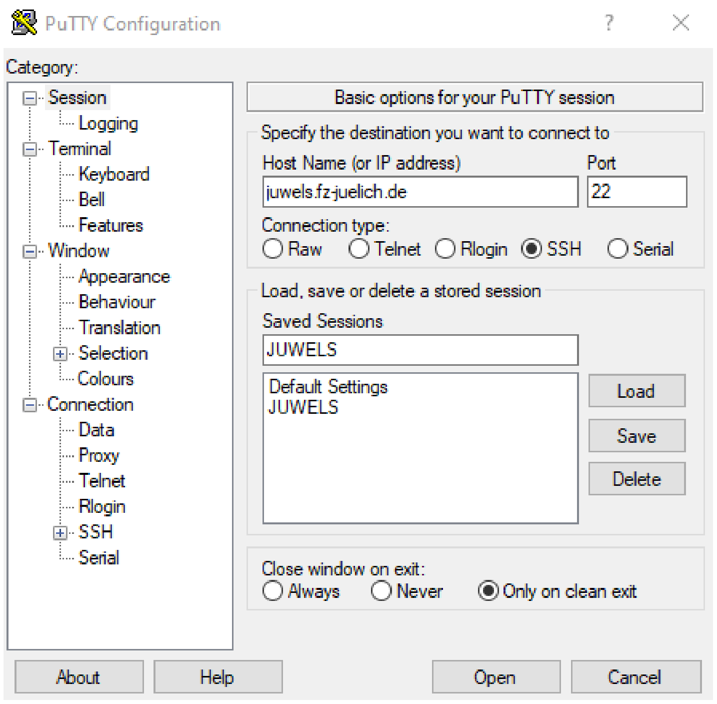
Then you should see an informational message (the *message of the day*, MOTD) followed by a shell prompt similar to the following:

```
********************************************************************************
*  Welcome to                                                                  *
*       _ _   ___        _______ _     ____                                    *
*      | | | | \ \      / / ____| |   / ___|     Juelich Wizard                *
*   _  | | | | |\ \ /\ / /|  _| | |   \___ \        for                        *
*  | |_| | |_| | \ V  V / | |___| |___ ___) |      European Leadership         *
*   \___/ \___/   \_/\_/  |_____|_____|____/         Science                   *
*                                                                              *
********************************************************************************
                                                          2020-11-19T14:00+0200
 ### Status information JUWELS ###

Known issues:     https://apps.fz-juelich.de/jsc/hps/juwels/known-issues.html

********************************************************************************
steinbusch1@jwlogin01:~ $
```

Once you have logged in successfully, you can continue with Unix shell basics.

Logging in with PuTTY

Launch `putty.exe` to log in. Set the *Host name* for the system you want to connect to, e.g. `juwels-cluster.fz-juelich.de`.



Basic options for your PuTTY session

Specify the destination you want to connect to

Host Name (or IP address): juwels.fz-juelich.de

Port: 22

Connection type: Raw Telnet Rlogin [✓ SSH] Serial

Load, save or delete a stored session

Saved Sessions: JUWELS

Default Settings
JUWELS

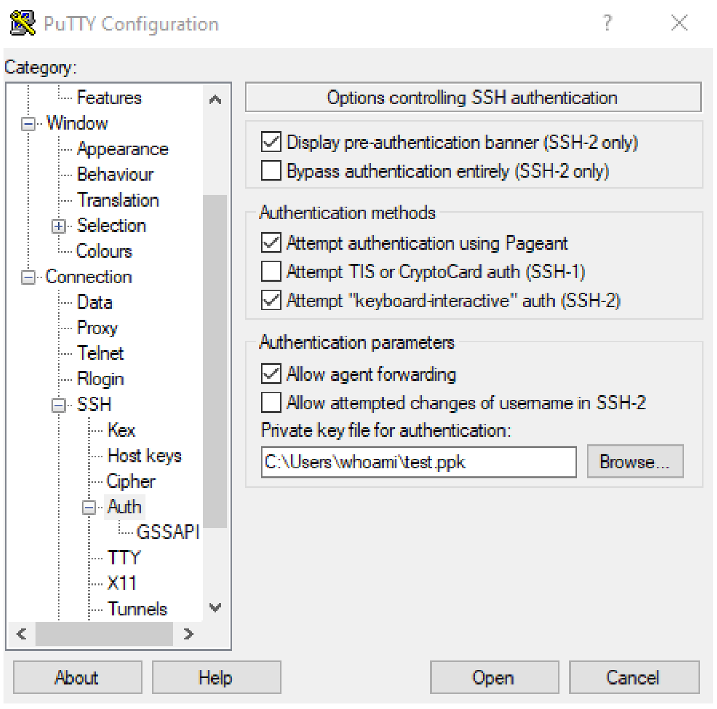
Close window on exit: Always Never [✓ Only on clean exit]

PuTTY session configuration

The following table lists the host names of login nodes for the different systems. Pick the one you want to use.

System	**Login node host name**
JURECA DC and Booster	`jureca.fz-juelich.de`
JUWELS Cluster	`juwels-cluster.fz-juelich.de`
JUWELS Booster	`juwels-booster.fz-juelich.de`
JUSUF	`jusuf.fz-juelich.de`

Navigate to *Connection* > *SSH* > *Auth* and under *Private key file for authentication:* select the key you just generated.



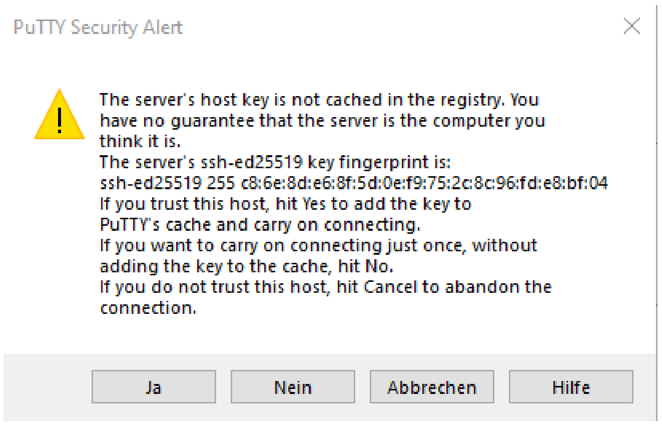
☑ Display pre-authentication banner (SSH-2 only)
☐ Bypass authentication entirely (SSH-2 only)

Authentication methods
☑ Attempt authentication using Pageant
☐ Attempt TIS or CryptoCard auth (SSH-1)
☑ Attempt "keyboard-interactive" auth (SSH-2)

Authentication parameters
☑ Allow agent forwarding
☐ Allow attempted changes of username in SSH-2
Private key file for authentication:
C:\Users\whoami\test.ppk

PuTTY auth configuration

If you want to save this configuration, you can navigate back to the *Session* screen to give the session a name and save it. Now click *Open* to connect. When you connect for the first time, PuTTY will display a dialog like the following:



The server's host key is not cached in the registry. You have no guarantee that the server is the computer you think it is.
The server's ssh-ed25519 key fingerprint is:
ssh-ed25519 255 c8:6e:8d:e6:8f:5d:0e:f9:75:2c:8c:96:fd:e8:bf:04
If you trust this host, hit Yes to add the key to PuTTY's cache and carry on connecting.
If you want to carry on connecting just once, without adding the key to the cache, hit No.
If you do not trust this host, hit Cancel to abandon the connection.

PuTTY security alert

This is not an error, but a security feature. The server key fingerprint displayed in the dialog has to be verified by comparing it to the known good fingerprint. JSC publishes SSH fingerprints for its systems through JuDoor. You can find them on the page you used to upload your public key.

Once you have logged in successfully, you can continue with Unix shell basics.

JupyterLab

Alternatively, you can use JupyterLab to log in. The authentication credentials are the same as for JuDoor. Once you have logged in, you need to create a JupyterLab instance by clicking *Add New JupyterLab*. On the next screen you must select which system you want to log in to, what project to use for accounting and what part of the system you want to log in to (more about this later), login nodes are the right choice for the moment. Startup of JupyterLab may take a while, but once it is done, you can launch a terminal running a shell on the system of your choice inside the browser. To do so, click *File* > *New* > *Terminal* and you should see a shell prompt similar to this:

```
[steinbusch1@jrl06 ~]$
```

Further reading

Our online documentation has more information on accessing the systems. It provides further examples of `from`-clauses, discusses configuration of SSH clients to set up short-cuts and gives hints for troubleshooting. If you want more details, you can find the documentation for our various systems here:

- JUWELS documentation: Access
- JURECA documentation: Access
- JUSUF documentation: Access

UNIX SHELL BASICS

Whether you log in via OpenSSH or PuTTY or opening a Terminal in JupyterLab, you will be interacting with the system through a Unix shell. Unix shells are text based interfaces that prompt the user to input commands and display the result of executing those commands back to the user. The underlying concepts (the file system, executing programs, etc.) are probably familiar to you, but the text based interface can seem daunting at first. This section will teach you how to accomplish essential tasks on a Unix shell. If you are already familiar with this kind of interface, you may want to skip ahead to the section describing the environment.

Like many operating systems, Unix provides an abstraction for storage media called a file system. Data of various types (text, images, executable code, etc.) is stored in files which can be organized in a tree-like hierarchy of directories that starts at a single root (the “root directory”). Objects in the file system (files or directories) are addressed using strings of characters called “paths” that list the directories one has to traverse to get to an object plus the objects name. The slash / serves as the separator between elements of a path and cannot itself appear in file or directory names. Some examples for paths are:

```
/etc
/usr/bin/env
```

```
/home/bsteinb/Documents
```

These paths are all "absolute paths", meaning, they describe the location of an objects in relation to the root directory (which is represented by a single slash `/`):

- `etc` is a directory that is found inside of the root directory
- `env` is a file found in the directory `bin` which itself is found in the directory `usr` inside the root directory
- `Documents` is a directory in `bsteinb` which is a directory in `home` which is a directory in the root directory

Since absolute paths can become unwieldy in deep directory hierarchies, Unix also allows relative paths. To this end, every program (including the shell you are using) is executed in a "working directory" (which can be changed during the execution of the program). Relative path specifications are then interpreted in relation to this working directory. They are written without the initial slash `/`. Some examples for relative paths are:

```
Documents
bin/env
../etc/crontab
```

With a working directory of `/usr`, `bin/env` refers to `/usr/bin/env`, just like the absolute path above. The path component `..` above has a special function. It refers to the parent (the containing directory) of a file system object and can appear in both relative and absolute paths. So `../etc/crontab` refers to a file `crontab` in a directory `etc` that can be found in the parent directory of the current working directory. `/home/bsteinb/../janedoe/.bashrc` can be simplified to `/home/janedoe/.bashrc`.

To find out the current working directory of the shell you are using, type:

```
$ pwd
```

The output should be something like:

```
/p/home/jusers/steinbusch1/juwels
```

which is the "home directory" associated with your account on the system. To list the contents of the working directory, execute the `ls` command:

```
$ ls
```

If you are working with a fresh user account, the output of this command might be empty, because there are no files (or only hidden files) in your home directory. To make `ls` display the hidden files as well, add the optional argument `-a`:

```
$ ls -a
```

The output should now be non-empty and contain files and directories with names that start with the period `.`. In Unix, whether a file system item is hidden or not is determined by the first character in its name being the period `.`.

`ls -a` is our first example of a more complex command invocation. It starts with the name of a command (so far, we have seen pwd and `ls`) followed by a list of arguments (here `-a`), all separated by spaces. `ls` can be used to list the contents of any directory, by specifying the path of the directory in the last position. To list the items in the `/etc` directory, type:

```
$ ls /etc
```

Most commands and the list of arguments they accept are documented in the Unix manual pages. They can be accessed through a command – `man` – that takes as its only argument the name of the manual page you want to read. For most commands there is a manual page with the same name as the command. To read the manual page for `ls`, type:

```
$ man ls
```

You can scroll through the manual page using the arrow keys. When you are done reading, close the manual by pressing `q` on the keyboard. To find manual pages for a specific topic, you can use the `apropos` command which searches the library of manual pages for a given keyword.

To change the working directory of your shell and all commands you invoke subsequently, use the `cd` command:

```
$ cd /
```

This will take you to the root directory. If you now execute `ls` without specifying a path, it should show you all items in the root directory, e.g.:

```
$ ls
arch   bin   dev  gpfs  lib    media  opt  proc  run   selinux  sys  usr
arch2  boot  etc  home  lib64  mnt    p    root  sbin  srv      tmp  var
```

Invoking `cd` without an argument takes you back to your home directory:

```
$ cd
$ pwd
/p/home/jusers/steinbusch1/juwels
```

Alternatively, the path to your home directory is also availably as the value of an “environment variable”. Environment variables map names (strings) to values (also strings) and can be seen as implicit input to commands while arguments on the command line are explicit inputs. The name of the environment variable that contains the path to your home directory is `HOME`. Its value can be inspected using the `printenv` command:

```
$ printenv HOME
/p/home/jusers/steinbusch1/juwels
```

The `printenv` command asks the environment for the value of the variable `HOME` (using the `getenv` function) and prints it to the terminal. In some situations it makes sense, to use the value of environment variables as explicit arguments to a command (e.g. if you want to `cd` to the value of `HOME`). This is supported by a shell mechanism called "variable expansion": mention the name of a variable, prefixed by the dollar sign `$` in a command line and the shell will substitute the value of the variable and pass that as an argument to the command:

```
$ cd $HOME
$ pwd
/p/home/jusers/steinbusch1/juwels
```

The `env` command can be used to inspect the environment. When invoked without any arguments it prints a list of all variables currently defined and their values.

```
$ env
[...]
HOME=/p/home/jusers/steinbusch1/juwels
[...]
```

`pwd`, `cd` and `ls` let you navigate the file system. The following commands can be used to make modifications to the file system. First is `mkdir` which allows you to make a directory:

```
mkdir <directory_path>
```

To create an empty file at a given location, use:

```
touch <file_path>
```

In your home directory, create two directories and a file:

```
$ mkdir dir1 dir2
$ touch dir1/file1
```

You can use `ls` to confirm that you have created two directories next to each other, one of which contains an empty file.

```
$ ls
dir1 dir2
$ ls dir1
file1
$ ls dir2
```

Files and directories can be moved, copied and deleted with the commands:

```
$ mv <source_path> <destination_path>
$ cp -r <source_path> <destination_path>
$ rm -r <path>
```

Make a copy of `dir1` and check that it also contains `file1`.

```
$ cp -r dir1 dir3
$ ls dir3
file1
```

Move the copy of the file into `dir2`.

```
$ mv dir3/file1 dir2
$ ls dir2
file1
$ ls dir3
```

Finally, remove all three directories.

```
$ rm -r dir1 dir2 dir3
$ ls
```

Lastly, we will mention one way of editing text files: the `nano` editor. To open a file in `nano`, type:

```
$ module load nano
$ nano <file_path>
```

(The `module` command will be explained in detail later on.)

To insert something into the file, just start typing. Save your changes by pressing *CTRL-O*. Exit the editor by pressing *CTRL-X*. The bottom part of the terminal will display more functions which can be reached using certain key bindings. Interaction with the editor, such as specifying a file name when saving, will also happen here.

ENVIRONMENT

Now that you know about the basic Unix commands, this section will teach you about some of the peculiarities of the environment on the systems at JSC.

Active project

The first point to talk about is the *active project*. You already know about accounts and computing time projects and by this point you should be a member of at least one project to have access to one of our systems. However, in general, a single user account can be a member of multiple computing time ("C") projects (and also data projects ("D")) at the same time. You can see the projects that you are currently a member of in your user profile on JuDoor, or, if you are logged in to one of the HPC systems, you can use the `jutil` command:

```
$ jutil user projects
     project    unixgroup      PI-uid project-type budget-accounts
```

```
------------ ------------ ----------- ------------ ---------------
       hello        hello    hellopi1            D               -
      chello       chello    hellopi1            C           hello
  training00   training00      coach2            C      training00
```

Certain system resources, like file system space and compute time, are associated with the projects that you are a member of. Performing actions that consume these resources, storing files or running a computation, have to be counted against the resource pool available to the project. This is done by storing files in certain locations or specifying a compute time budget when running computations. It is possible to explicitly specify a project, each time one of these actions is performed. For brevity's sake, one can also make one of the projects the "active project" and then all actions performed in the remainder of the session will implicitly be performed in the context of that project. This can also be done through the `jutil` command:

```
$ jutil env activate -p training2126 -A training2126
```

Now `training2126` is the active project. Any computational jobs will be accounted against its budget and the special file system locations associated with it can be reached through certain environment variables. More about that in the next section.

Hint: In case you are working on different compute budgets we recommend to set the budget explicitly as it is described later in the document to avoid using the "wrong" budget for a specific simulation job.

File system points of interest

Every user account on the systems has a home directory (reachable through the `HOME` environment variable) where the user can store his personal files. However, there is a limit on the volume of data and also the number of files that can be stored in this directory. Furthermore, the file system performance in `HOME` is reduced. It is recommended to use `HOME` only for configuration files. More storage space is granted to computing time projects. At least two directories are created for each project:

- a `PROJECT` directory, that can store medium amounts of data, offers modest performance and is backed up regularly, and
- a `SCRATCH` directory, that offers high I/O bandwidth, should be used for input and output of computations (however, no back up is performed and files that have not been touched in 90 days get deleted automatically).

Data projects have access to other storage locations, e.g. the tape based `ARCHIVE` for long term storage of results.

The path of these directories is available as the value of environment variables of the form `<directory>_<project>`, e.g. `PROJECT_training2126` or

SCRATCH_training2126. If you have activated a project in the previous section, you will also have environment variables that are just PROJECT and SCRATCH that point to the respective directories of the active project.

Print the contents of PROJECT_training2126 and PROJECT:

```
$ printenv PROJECT_training2126
/p/project/training2126
$ printenv PROJECT
/p/project/training2126
```

Change into that directory and see what is already there:

```
$ cd $PROJECT_training2126
$ ls
```

Inside the PROJECT directory, make a directory to contain the files that you work on. In order to avoid collisions, use your account name as the name of the directory (the USER environment variable contains your user name):

```
$ mkdir $USER
```

There is more information on file system points of interest in the documentation.

Further reading

Our online documentation has more information on the system environment. It describes further file systems covering more specialised use cases and discusses transferring files to and from the systems via SSH and Git. If you want more details, you can find the documentation for our various systems here:

- JUWELS documentation: Environment
- JURECA documentation: Environment
- JUSUF documentation: Environment

SOFTWARE MODULES

HPC centres will usually make some effort to provide software that is commonly used for scientific purposes. This includes compilers, parallel programming libraries like MPI, numerical libraries, and even complete simulation programs. These software packages form a hierarchy of dependencies (simulation programs use the numerical and parallel programming libraries and all of it is compiled with a certain compiler). Towards the bottom of this hierarchy, packages tend to be interchangeable (several compilers for C or Fortran, several libraries implement the MPI standard) and some of the higher up packages perform better for example when compiled with a certain compiler. Therefore, it makes sense to offer a range of software packages that implement low level functions and then build a software landscape upon each combination of those low level packages. The two lowest levels in this hierarchy, compiler and MPI library together form a “toolchain”. To help keep the complexity of accessing these different collections of software in check, JSC uses a combination of EasyBuild and Lmod to build software and make it available as software modules. During a log in session, modules can be loaded and unloaded using the `module` command to use the software that is provided by them. When you log in, a set of default modules is loaded for you, e.g. on JUWELS:

```
$ module list

Currently Loaded Modules:
  1) GCCcore/.9.3.0 (H)   3) binutils/.2.34 (H)
  2) zlib/.1.2.11   (H)   4) StdEnv/2020

  Where:
   H:  Hidden Module
```

To see what other modules can currently be loaded, type:

```
$ module avail

-------------------------- Core packages ---------------------------
   Advisor/2020_update3
   Autotools/20200321
   Autotools/20200321                                 (D)

   [...]

   unzip/6.0
   xpra/4.0.4-Python-3.8.5
   zsh/5.8

---------------------------- Compilers -----------------------------
   GCC/9.3.0                          NVHPC/20.9-GCC-9.3.0  (g)
   Intel/2020.2.254-GCC-9.3.0         NVHPC/20.11-GCC-9.3.0 (g,D)
   NVHPC/20.7-GCC-9.3.0       (g)     NVHPC/21.1-GCC-9.3.0  (g)

----------------- User-based install configuration -----------------
   UserInstallations/easybuild

  Where:
   S:        Module is Sticky, requires --force to unload or purge
   g:        built for GPU
   L:        Module is loaded
   Aliases:  Aliases exist: foo/1.2.3 (1.2) means that "module load foo/1.2" will load
foo/1.2.3
   D:        Default Module

Use "module spider" to find all possible modules and extensions.
Use "module keyword key1 key2 ..." to search for all possible modules matching
any of the "keys".
```

The available modules are grouped into categories:

- Core packages, which are independent of the choice of toolchain
- Compilers, which are the first ingredient of a toolchain
- Archictectures, that can be used to load software for different processor architectures, this category does not exist on all systems

Go ahead and load a compiler:

```
$ module load GCC
```

If you now run `module avail` again, you will notice two additional software categories:

```
$ module avail

------------ MPI runtimes available for GNU compilers -------------
[...]

--------------- Packages compiled with GNU compilers ---------------
[...]
```

These contain modules that depend on (or were built with) the GCC module that you just loaded. Loading one of the available MPI modules will complete your choice of a toolchain and make more software available:

```
$ module load OpenMPI
$ module avail

------------------------- OpenMPI settings -------------------------
   mpi-settings/CUDA-low-latency    mpi-settings/CUDA (L,D)

--------- Packages compiled with OpenMPI and GCC compilers ---------
[...]
```

If you are looking for a particular piece of software that you know the name of, rather than rummaging through all the toolchains, you can use the `module spider` subcommand, as the output of `module avail` suggests:

```
$ module spider LAMMPS

----------------------------------------------------------------------------
  LAMMPS:
----------------------------------------------------------------------------
    Description:
      LAMMPS is a classical molecular dynamics code, and an acronym for
      Large-scale Atomic/Molecular Massively Parallel Simulator. LAMMPS has
      potentials for solid-state materials (metals, semiconductors) and
      soft matter (biomolecules, polymers) and coarse-grained or mesoscopic
      systems. It can be used to model atoms or, more generically, as a
      parallel particle simulator at the atomic, meso, or continuum scale.
      LAMMPS runs on single processors or in parallel using message-passing
      techniques and a spatial-decomposition of the simulation domain. The
      code is designed to be easy to modify or extend with new
      functionality.

     Versions:
        LAMMPS/24Dec2020-CUDA
        LAMMPS/24Dec2020
        LAMMPS/29Oct2020-CUDA
        LAMMPS/29Oct2020

----------------------------------------------------------------------------
  For detailed information about a specific "LAMMPS" package (including how to load the
modules) use the module's full name.
  Note that names that have a trailing (E) are extensions provided by other modules.
  For example:

     $ module spider LAMMPS/29Oct2020
----------------------------------------------------------------------------
```

Loading the LAMMPS module with OpenMPI loaded fails:

```
$ module load LAMMPS
Lmod has detected the following error:  These module(s) or
extension(s) exist but cannot be loaded as requested: "LAMMPS"
  Try: "module spider LAMMPS" to see how to load the module(s).
```

`module spider` with a specific module version provides details on how the module can be loaded:

```
$ module spider LAMMPS/24Dec2020

----------------------------------------------------------------------------
  LAMMPS: LAMMPS/24Dec2020
----------------------------------------------------------------------------
    Description:
      LAMMPS is a classical molecular dynamics code, and an acronym for
      Large-scale Atomic/Molecular Massively Parallel Simulator. LAMMPS has
      potentials for solid-state materials (metals, semiconductors) and
      soft matter (biomolecules, polymers) and coarse-grained or mesoscopic
      systems. It can be used to model atoms or, more generically, as a
      parallel particle simulator at the atomic, meso, or continuum scale.
      LAMMPS runs on single processors or in parallel using message-passing
      techniques and a spatial-decomposition of the simulation domain. The
      code is designed to be easy to modify or extend with new
      functionality.


    You will need to load all module(s) on any one of the lines below before the
"LAMMPS/24Dec2020" module is available to load.

      GCC/9.3.0  ParaStationMPI/5.4.7-1
      Intel/2020.2.254-GCC-9.3.0  ParaStationMPI/5.4.7-1

    Help:
      Description
      ===========
      LAMMPS is a classical molecular dynamics code, and an acronym
```

```
for Large-scale Atomic/Molecular Massively Parallel Simulator. LAMMPS has
potentials for solid-state materials (metals, semiconductors) and soft matter
(biomolecules, polymers) and coarse-grained or mesoscopic systems. It can be
used to model atoms or, more generically, as a parallel particle simulator at
the atomic, meso, or continuum scale. LAMMPS runs on single processors or in
parallel using message-passing techniques and a spatial-decomposition of the
simulation domain. The code is designed to be easy to modify or extend with new
functionality.


More information
================
 - Homepage: https://lammps.sandia.gov/
 - Site contact: a.kreuzer@fz-juelich.de
```

The problem is that LAMMPS is only available in toolchains which include ParaStationMPI. It is not necessary to reload the entire toolchain, it is enough to reload the MPI runtime:

```
$ module load ParaStationMPI
$ module load LAMMPS
```

Specific modules can be unloaded again using the `module unload` command. To unload (almost) all modules and start with a fresh environment, use `module purge`.

JURECA and JUWELS consist of multiple system modules (as opposed to software modules) based on different compute technologies (AMD and Intel CPUs on JURECA, CPUs and GPUs on JUWELS). The software we provide on these systems is also split into different hierarchies, one per system module. On JUWELS, which uses different login nodes for the different system modules, the correct software collection is loaded automatically based on which login node you use. On JURECA, which uses one set of login nodes for both system modules, the default software collection is the one for the DC module. To access software for JURECA Booster, you have to use:

```
$ module load Architecture/jurecabooster
```

The `module` command is part of the Lmod software package. It comes with its own help document which you can access by running `module help` and a user guide is available online.

Further reading

Our online documentation has more information on software modules. It lists the basic tool chains (compiler + communication library + math library) available on our systems and discusses using older software stages. If you want more details, you can find the documentation for our various systems here:

- JUWELS documentation: Software Modules
- JURECA documentation: Software Modules

- JUSUF documentation: Software Modules

CUSTOM SOFTWARE

For some, the software that is made available via the module system is enough to do their daily work. Others will want to bring their own software to the systems. This chapter will teach you how to run software distributed as source code for both compiled programming languages and scripting languages.

Compiled languages

For the three most common compiled languages in scientific computing, C, C++, and Fortran, the basic workflow is very similar. Open the file `hellompi.c` in the `nano` editor (or a different editor of your choice). (`nano` is available as a module, if you want to use it, type `module load nano`.)

```
$ nano hellompi.c
```

Paste the following listing into the file, save and close the editor.

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[]) {
  MPI_Init(&argc, &argv);

  int r, s;
  MPI_Comm_rank(MPI_COMM_WORLD, &r);
  MPI_Comm_size(MPI_COMM_WORLD, &s);
  printf("hello from process %d of %d\n", r, s);

  MPI_Finalize();
}
```

Once you have a compiler and an MPI library loaded (e.g. `module load GCC OpenMPI`), the file can be compiled as follows:

```
$ mpicc -std=c11 -o hellompi hellompi.c
```

We will explain how to run the program in a later chapter.

A lot of software is not compiled and installed by invoking the compiler directly, but by using a build system. GNU `make` is installed from the operating system package sources and GNU `autotools` as well as `CMake` are available as modules. More exotic build systems are also available, as are compilers for other languages like Go or Rust.

Scripting languages

Scripting languages have become more popular in scientific computing recently. Modules are available for Python and Julia.

Python

The Python interpreter can be loaded as a module as well as the `mpi4py` package that allows you to use MPI from your Python programs.

```
$ module load Python mpi4py
```

Edit a file `hellompi.py`:

```
($ module load nano)
$ nano hellompi.py
```

And paste the following content into it, then save and exit the editor.

```
from mpi4py import MPI

r = MPI.COMM_WORLD.rank
s = MPI.COMM_WORLD.size

print(f"hello from process {r} of {s}")
```

We will explain how to run the program in a later chapter.

More Python packages are available as modules. For scientific computing, the `SciPy-Stack` collection is especially interesting.

ACCOUNTING

General information

Each computing time project has been granted a certain amount of compute time (core hours) on an HPC system. This budget is split monthly over the runtime of a project so that a regular project that runs for 12 months has 1/12 of the total amount of the granted core-h available each month. To allow further flexibility we have established a “3-month-window”: Core hours that have not been used in the previous month can be used in the current month and will be lost in the next month if they are not used in the current month. Whereby in the current month you can also use the quota of the next month but with a decreased priority of the submitted jobs. The priority will be further decreased if you have used up even the quota of the next month.

Job accounting

Users are charged for complete nodes they occupy, regardless of the number of CPUs used since the requested compute nodes for your application are not shared among users. The comute time used for one job will be accounted by the following formula: `#nodes * #AvailableCoresPerNode * walltime`.

Jobs that run on nodes equipped with GPUs are charged in the same way. Independent of the usage of the GPUs the available cores on the host CPU node are taken into account.

Detailed information of each job can be found in KontView which is accessible via the button 'show extended statistics' for each project in Judoor.

Alternatively, you can execute the following command on the login nodes to query your CPU quota usage: `jutil user cpuquota`. Further information can be found in the "Accounting" chapter of the corresponding System Documentation.

RUNNING JOBS

Up to now, you have been working on the log in nodes of the system. These nodes are set aside for working interactively on tasks that are needed to prepare your computations, such as compiling your applications, moving input data into place, and writing configuration files for your programs. Since the number of log in nodes for each system is small and they are shared between all users, we ask you to keep the resource consumption on these systems as low as possible. Building software should be restricted to using only a few processes in parallel, simulations and post-processing jobs should be run on the compute nodes. Use the who command to see who else is logged in to the log in node you are currently using:

```
$ who
steinbusch1 pts/71       2021-03-11 09:51 (pool-148-54.vpn.kfa-juelich.de)
[...]
$ who | wc -l
59
```

Unlike the log in nodes, users are not given free access to the compute nodes at any time. Instead they form a pool of resources managed by the resource manager software. Due to our collaboration with the company Partec we use "psslurm" which is based on Slurm and optimized for our systems to manage these resources. To run a computation on the compute nodes, you have to specify to the resource manager what amount of resources you need and for which duration. Once the resources have become available, you will be allowed to execute programs on them. Two modes of operation are possible:

- interactive mode where programs can be run on the allocated resources from a shell, possibly repeatedly, and
- batch mode where a shell script describing the commands to run as part of a

computation is handed off to the resource manager for asynchronous execution.

Interactive mode

One-shot

The `srun` command is used to execute commands on a set of allocated resources. If no resources are currently allocated, `srun` can infer from its command line arguments what resources are needed, request them from the resource manager and defer the execution of the associated commands until the resources are available. After the associated commands have been run, the resources are relinquished and running further commands will have to ask for resources again. This one-shot mode can be useful when you want to interactively run a few quick jobs with varying sets of resources allocated for them. Run the `hostname` command to see how `srun` will run commands on different nodes than the log in nodes. On JURECA DC, JUWELS Cluster and JUSUF, use this command:

```
$ hostname
jwlogin08.juwels
$ srun -A training2126 --reservation hands-on hostname
srun: job 3472578 queued and waiting for resources
srun: job 3472578 has been allocated resources
jwc00n075.juwels
```

For the boosters, JURECA Booster and JUWELS Booster, there are a few differences: On both boosters, the name of the reservation is `hands-on-booster` instead of `hands-on`. Furthermore, to submit to JURECA Booster, you have to specify a non-default partition (more about partitions below):

```
$ hostname
jrlogin02.jureca
$ srun -A training2126 --reservation hands-on --partition booster hostname
srun: job 9792359 queued and waiting for resources
srun: job 9792359 has been allocated resources
jrc6617.jureca
```

To submit to JUWELS Booster, you want to be logged in to the Booster login nodes and you have to specify the number of GPUs you want to use:

```
$ hostname
jwlogin24.juwels
$ srun -A training2126 --reservation hands-on --gres gpu:4 hostname
srun: job 4575092 queued and waiting for resources
srun: job 4575092 has been allocated resources
jwb0053.juwels
```

Please keep these differences in mind if you are using one of the Boosters, they will not be repeated in further examples.

Invocations of the `srun` command have the following syntax:

```
$ srun <srun options...> <program> <program options...>
```

Above we have seen four `srun` options:

- `-A` (short for `--account`) to charge the resources consumed by the computation to the budget allotted to this course (if you have used `jutil env activate -A training2126` earlier on, you do not need this)
- `--reservation` to use nodes which have been set aside for this course. As before, to work on Booster modules, you have to replace `hands-on` with `hands-on-booster`.
- `--partition` specifies which set of compute nodes to request resources from. We typically group nodes of the same hardware type into a partition.
- `--gres` specifies additional resources, other than compute nodes, in this case the presence of four GPUs in the compute nodes.

For the `<program>` we used `hostname` with no arguments of its own.

To run more parallel instances of a program, increase the number of Slurm *tasks* using the `-n` option to `srun`:

```
$ srun --label -A training2126 --reservation hands-on -n 10 hostname
srun: job 3472812 queued and waiting for resources
srun: job 3472812 has been allocated resources
8: jwc00n002.juwels
9: jwc00n002.juwels
0: jwc00n002.juwels
1: jwc00n002.juwels
6: jwc00n002.juwels
3: jwc00n002.juwels
5: jwc00n002.juwels
2: jwc00n002.juwels
7: jwc00n002.juwels
4: jwc00n002.juwels
```

If you do not tell Slurm that your commands are multi-threaded (`hostname` is not), it will assume each task only needs a single CPU core and pack as many as possible into a node. Note also the `--label` option to `srun` which prefixes every line of output by a number that identifies the task that generated the output.

Running more tasks than will fit on a single node will allocate two nodes and split the tasks between nodes:

```
$ srun --label -A training2126 --reservation hands-on -n 100 hostname
srun: job 3473040 queued and waiting for resources
srun: job 3473040 has been allocated resources
 0: jwc00n007.juwels
[...]
50: jwc00n008.juwels
```

```
[...]
```

Allocations always contain entire nodes exclusively. So your jobs should request a number of tasks that is divisible by the number of tasks which can fit on a node to avoid losing parts of your budget.

You can now also use `srun` to run the `hellompi` program introduced in the previous section on deploying custom software:

```
$ srun -A training2126 --reservation hands-on -n 5 ./hellompi
srun: job 3471349 queued and waiting for resources
srun: job 3471349 has been allocated resources
hello from process 4 of 5
hello from process 0 of 5
hello from process 3 of 5
hello from process 1 of 5
hello from process 2 of 5
```

Interlude: Partitions

The systems at JSC typically provide more than one pool of resources, called *partitions*. The resources in the different partitions might have diferent hardware characteristics or cater to different use cases.

Unless you are using JURECA Booster, the previous examples were run on the default partition of the system you are using, `batch` on JUWELS Cluster and JUSUF Cluster, `booster` on JUWELS Booster and `dc-cpu` on JURECA. You can find out what partitions the different systems have in the documentation for JURECA, JUWELS, and JUSUF.

Of particular interest are the development partitions on each system (look for `devel` in their name). These consist of a small number of nodes which are set aside to prioritise small and short jobs which are typically run as part of development work on your application rather than production use of the system.

Try running the previous two examples using `hostname` on the development partition of your system by specifying it through `srun`'s `-p` option. Remove the `--reservation` option, because the reservation does not include nodes from the development partition.

We will have a look at other partitions later.

Interactive allocation

If, instead of requesting resources anew everytime you want to run a command on the compute nodes, you want to hold on to a specific set of resources and quickly dispatch a series of commands to run on them, you can use the `salloc` command in combination with `srun`. To do so, you specify the amount of resources you will need for your computations when calling `salloc`. `salloc` will request these resources from the resource manager and block until they are available. Then it will launch a new shell for you from which you can call `srun`, possibly multiple times, to dispatch commands onto the allocated resources.

In the previous section you took a task-centric approach to requesting resources by using the `-n` command line argument to `srun` to specify a number of tasks you want to run. This approach also works with `salloc` – in fact the way you specify resources is mostly the same between all different modes Slurm supports. However, since the number of CPU cores is always rounded up to the next multiple of the number of CPU cores in a single node, it might make sense to take a hardware centric approach to requesting resources. Using the `-N` command line argument, you can request a number of nodes from the resource manager (remember to specify `--partition booster` for JURECA Booster or `--gres gpu:4` for the JUWELS Booster):

```
$ salloc -A training2126 --reservation hands-on -N 1
salloc: Pending job allocation 3475519
salloc: job 3475519 queued and waiting for resources
salloc: job 3475519 has been allocated resources
salloc: Granted job allocation 3475519
salloc: Waiting for resource configuration
salloc: Nodes jwc00n014 are ready for job
$
```

At the new shell prompt, you can use `srun` to run commands without having to specify resources again:

```
$ srun hostname
jwc00n014.juwels
```

By default, Slurm assumes that your program is single-threaded, but still only launches one task per allocated node. This can be changed by specifying the CPUs per task with the `-c` argument.

```
$ srun -c 1 hostname
jwc00n014.juwels
[...]
jwc00n014.juwels
```

If you want to run several commands on a node without having to go through `srun` each time, you can use `srun` to launch a shell on the node:

```
$ srun --pty --cpu-bind=none /bin/bash
$ hostname
jwc00n014.juwels
$ exit
```

When using `srun` in one-shot mode, your account is charged for the time it takes to run the associated command. With `salloc` your account is charged for the duration of time you spend in the shell launched by `salloc` (and commands launched by that shell). Once you are done with the allocated resources, do not forget to exit from the shell:

```
$ exit
salloc: Relinquishing job allocation 3475519
salloc: Job allocation 3475519 has been revoked.
$ printenv SLURM_JOB_ID
$
```

If the `printenv SLURM_JOB_ID` prints a number, then you are still inside the allocation.

Batch mode

If the system is relatively quiet and you are asking for a small amount of resources (or working on the `devel` partitions), `salloc` or one-shot `srun` should allow you to work with the system more or less interactively. Large production jobs on the other hand might have to wait an uncomfortably long time for resources and so running them interactively is not really convenient. Imagine you `salloc` a large number of nodes and while you wait you decide to go have lunch. If the allocation comes through while you are away you will still be charged for the resources even if they idle.

Also, if the systems were only used interactively, resource utilization would drop off in the late hours of the evening and ramp up in the mornings.

To enable better resource utilization and allow users to schedule jobs asynchronously, Slurm offers a batch mode through the `sbatch` command. It too requests resources from the resource manager, but unlike `salloc` which presents you with an interactive shell prompt from which you can call `srun`, `sbatch` runs commands from a shell script (the "job script") without needing user intervention. The resources can be specified as command line arguments to `sbatch`, same as with `salloc` and `srun`, but can also be described in the job script. Open a new shell script in the editor:

```
($ module load nano)
$ nano testjob.sh
```

And enter the following script:

```
#!/bin/bash
#SBATCH --account=training2126
#SBATCH --reservation=hands-on
#SBATCH --nodes=2
#SBATCH --cpus-per-task=1
#SBATCH --output=mpi-out.%j
#SBATCH --error=mpi-err.%j
#SBATCH --time=00:05:00

module load GCC ParaStationMPI

srun ./hellompi
```

Remember to specify the `booster` partition for JURECA Booster or the `gpu:4` `gres` for JUWELS Booster.

Then save the script and submit it for execution with:

```
$ sbatch testjob.sh
Submitted batch job 3476793
```

After the first line (the shebang line) the script contains specially formatted comments that act like arguments to `sbatch`. These arguments are written in their long form. Previously, you used the short form (e.g. `-N` is the same as `--nodes`). After the block of comments come regular shell comands. Inside the job script, we use the `module` command to make the software modules needed by the job programs available (here the compiler with its runtime libraries and an MPI library). The tasks are once again created using the `srun` command which works the same as before.

The job created by `sbatch` has to wait in a queue until the necessary resources become available. Use the `squeue` command to inspect the queue:

```
$ squeue -u $USER
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
           3476793     batch testjob. steinbus PD       0:00      2 (Priority)
```

You might have to wait for a while, but eventually your job will be run. While your job is pending in the queue or already running you can execute another command to retrieve further information about your job:

```
$ scontrol show job <JOBID>
```

Once it is running, you will find two files next to the job script, `mpi-err.XXXXXXX` and `mpi-out.XXXXXXX` where `X` are decimal digits. These contain what was written to the standard error and output streams by your job.

Affinity and multi-threading

Computers today are typically equipped with multi-core CPUs which can work on multiple streams of instructions at the same time. The operating system is in charge of deciding which program gets to use which CPU core at a given point in time. Usually, it will let those programs which need access to resources run wherever resources are available, meaning one and the same program can end up using different CPU cores at different points in time. On a desktop machine this is not a problem. In fact it is a good thing, since we typically run far more programs than we have CPU cores available.

In an HPC setting things are different in that the workloads are adapted to use a number of processes or threads which matches the number of CPU cores (normally, you will have `n_processes x n_threads = n_nodes x n_CPU_cores_per_node`). If there is exactly one process or thread per CPU core, it would be wasteful to shuffle them around between different CPU cores. In order to avoid this shuffling, the resource manager assigns to the processes that it spawns an *affinity mask*. An affinity mask is a set of numbers identifying the CPU cores a process is allowed to use. By default, Slurm assumes that the processes you create are single threaded and gives each process access to a single CPU core. Allocate a node for playing around with this mechanism:

```
$ salloc -A training2126 --reservation hands-on -N 1
salloc: Pending job allocation 3499694
salloc: job 3499694 queued and waiting for resources
salloc: job 3499694 has been allocated resources
salloc: Granted job allocation 3499694
salloc: Waiting for resource configuration
salloc: Nodes jwc00n001 are ready for job
```

Use the `numactl` command to inspect the affinity masks created by Slurm:

```
$ srun --label numactl --show
0: policy: default
0: preferred node: current
0: physcpubind: 0
0: cpubind: 0
0: nodebind: 0
0: membind: 0 1
```

The identifiers of accessible CPU cores are listed in `physcpubind`. Here, the single process that is created has access to a single CPU core, `0`. Now, confirm that different processes will get access to different CPU cores:

```
$ srun --label -n 3 numactl --show
2: policy: default
2: preferred node: current
2: physcpubind: 1
2: cpubind: 0
2: nodebind: 0
2: membind: 0 1
1: policy: default
1: preferred node: current
1: physcpubind: 24
1: cpubind: 1
```

```
1: nodebind: 1
1: membind: 0 1
0: policy: default
0: preferred node: current
0: physcpubind: 0
0: cpubind: 0
0: nodebind: 0
0: membind: 0 1
```

The three processes get access to CPU cores `0`, `1`, and `24` respectively. If your processes are not single-threaded, you will have to give them access to more CPU cores (otherwise all threads will run on the same CPU core). This can be done using Slurm's `--cpus-per-task` parameter, or `-c`:

```
$ srun --label -c 24 numactl --show
1: policy: default
1: preferred node: current
1: physcpubind: 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
1: cpubind: 1
1: nodebind: 1
1: membind: 0 1
0: policy: default
0: preferred node: current
0: physcpubind: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
0: cpubind: 0
0: nodebind: 0
0: membind: 0 1
2: policy: default
2: preferred node: current
2: physcpubind: 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
2: cpubind: 0
2: nodebind: 0
2: membind: 0 1
3: policy: default
3: preferred node: current
3: physcpubind: 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
3: cpubind: 1
3: nodebind: 1
3: membind: 0 1
```

Note how once you specify the number of CPU cores per task, Slurm switches its behavior from creating one process per node to filling the node with as many processes as possible. Each process gets access to 24 different CPU cores.

Copy the following small program into a file `hellohybrid.c`:

```
#include <stdio.h>
#include <mpi.h>
#include <omp.h>

int main(int argc, char* argv[]) {
  MPI_Init(&argc, &argv);

  int r, s;
  MPI_Comm_rank(MPI_COMM_WORLD, &r);
  MPI_Comm_size(MPI_COMM_WORLD, &s);
  #pragma omp parallel
  if (!omp_get_thread_num())
    printf(
      "hello from process %d of %d, using %d threads\n",
      r, s, omp_get_num_threads()
    );

  MPI_Finalize();
}
```

And compile it with:

```
$ mpicc -fopenmp -o hellohybrid hellohybrid.c
```

Now run the program:

```
$ srun ./hellohybrid
hello from process 0 of 1, using 1 threads
```

Again, using default settings, Slurm creates a single process and restricts it to a single CPU core. The OpenMP run time library queries the number of CPU cores accessible to the process and creates just as many threads (here only one). If you specify a number of CPU cores per process this changes:

```
$ srun -c 24 ./hellohybrid
hello from process 2 of 4, using 24 threads
hello from process 0 of 4, using 24 threads
hello from process 3 of 4, using 24 threads
hello from process 1 of 4, using 24 threads
```

Once more, Slurm fills the node with four processes having appropriate affinity masks. The OpenMP run time figures out that each process is allowed to use 24 CPU cores and creates a team of threads to fill those CPU cores.

JSC Affinity Tools

Since we are using psslurm we have implemented a few options different than the default in Slurm. For this reason we are offering two tools that can help you to understand the process affinity on our systems:

1. The command line executable: `psslurmgetbind`
2. An online pinning tool

Further information can be found in the “Processor Affinity” chapter of the corresponding System Documentation.

Further reading

Our online documentation has more information on working with the resource manager. It has detailed lists with the hardware available in various partitions as well as job limits. Also, it discusses advanced topics like multiple job steps, dependency chains and heterogeneous jobs. If you want more details, you can find the documentation for our various systems here:

- JUWELS documentation: Batch system
- JURECA documentation: Batch system
- JUSUF documentation: Batch system

You can also have a look at the official Slurm documentation.

USING GPUS

All systems at JSC have nodes which are accelerated by General Purpose Graphics Processing Units (GPGPUs or just GPUs). Since they GPUs are all made by NVIDIA, using them is accomplished through their CUDA SDK. CUDA is available as a module:

```
$ module load CUDA
```

To demonstrate how to compile and run a program that uses GPUs, we will use one of the examples included in CUDA:

```
$ cp -r $EBROOTCUDA/samples/0_Simple/simpleMPI $PROJECT_training2126/$USER
$ cd $PROJECT_training2126/$USER/simpleMPI
$ make
/p/software/juwels/stages/2020/software/psmpi/5.4.7-1-GCC-9.3.0/bin/mpicxx -
I../../common/inc    -o simpleMPI_mpi.o -c simpleMPI.cpp
/p/software/juwels/stages/2020/software/CUDA/11.0/bin/nvcc -ccbin g++ -I../../common/inc
-m64    -gencode arch=compute_35,code=sm_35 -gencode arch=compute_37,code=sm_37 -gencode
arch=compute_50,code=sm_50 -gencode arch=compute_52,code=sm_52 -gencode
arch=compute_60,code=sm_60 -gencode arch=compute_61,code=sm_61 -gencode
arch=compute_70,code=sm_70 -gencode arch=compute_75,code=sm_75 -gencode
arch=compute_80,code=sm_80 -gencode arch=compute_80,code=compute_80 -o simpleMPI.o -c
simpleMPI.cu
nvcc warning : The 'compute_35', 'compute_37', 'compute_50', 'sm_35', 'sm_37' and 'sm_50'
architectures are deprecated, and may be removed in a future release (Use -Wno-
deprecated-gpu-targets to suppress warning).
/p/software/juwels/stages/2020/software/psmpi/5.4.7-1-GCC-9.3.0/bin/mpicxx    -o
simpleMPI simpleMPI_mpi.o simpleMPI.o  -
L/p/software/juwels/stages/2020/software/CUDA/11.0/lib64 -lcudart
mkdir -p ../../bin/x86_64/linux/release
mkdir: cannot create directory ‘../../bin’: Permission denied
make: *** [Makefile:377: simpleMPI] Error 1
```

You can ignore the error at the end. There should now be an executable called `simpleMPI` inside the `simpleMPI` directory. To run the program, use `srun` like before:

```
$ srun -A training2126 -p <gpu partition> --gres gpu:4 -N 1 -n 4 ./simpleMPI
srun: job 3490053 queued and waiting for resources
srun: job 3490053 has been allocated resources
Running on 4 nodes
Average of square roots is: 0.667305
PASSED
```

You have to specify a partition that contains nodes equipped with GPUs, `-p develgpus` for JUWELS and JUSUF, `-p dc-gpu-devel` for JURECA, or `-p develbooster` for JUWELS Booster, and you have to specify how many GPUs you want those nodes to have, `--gres gpu:4` (or `--gres gpu:1` on JUSUF).

GPU Affinity

On systems with more than one GPU per node, a choice presents itself of which GPU should be visible to which application task. This is controlled through the environment variable `CUDA_VISIBLE_DEVICES`, which can be set to a comma separated list of integers identifying devices to make visible to a task. You can manually define this variable before running your tasks with `srun`. If the variable is not defined by you, `srun` will provide a default:

- for jobs with a single task (`-n 1`) all devices will be visible `CUDA_VISIBLE_DEVICES=0,1,2,3`
- for all other jobs, only a single device will be visible per task, with the same device being visible to multiple tasks if there are more tasks than GPUs

Note: The behavior described above is currently only implemented on JURECA. On all other systems, only a single GPU is visible by default, even for jobs with a single task. This is a known issue which we expect to be resolved soon.

Further reading

Our online documentation has more information on software modules. It lists the basic tool chains (compiler + communication library + math library) available on our systems and discusses using older software stages. If you want more details, you can find the documentation for our various systems here:

- JUWELS documentation: GPU Computing
- JURECA documentation: GPU Computing
- JUSUF documentation: GPU Computing

USEFUL LINKS

System Documentation

JSC offers documentation for the production systems:

- JUWELS
- JURECA
- JUSUF

Job Reporting

The Job Reporting service gives you access to PDF reports which contain certain performance metrics that the system automatically collects about your jobs. It also includes an overview over the system utilization and queue. You can access the Job Reporting service for the different systems here:

- JUWELS
- JURECA
- JUSUF

Apply for Computing Time

The JSC web site describes how to apply for computing time.

JSC Course Programme

JSC offers many courses throughout the year covering topics such as parallel programming, machine learning, and visualization. Please have a look at the course programme on the JSC web site.

Supercomputing Support

Our high-level support team supports the users in case of problems on our systems, e.g. porting of the application, parallelisation and performance issues as well as usage of the HPC system. So if you are having a question, you cannot sort out by yourself, by working through this document or by having a look into the documentation, just drop a mail to sc@fz-juelich.de.