VLAAMS SUPERCOMPUTER CENTRUM | *Innovative Computing for A Smarter Flanders*

HPC Tutorial

Last updated: November 22 2018

For Windows Users

Authors:

Franky Backeljauw[5], Stefan Becuwe[5], Geert Jan Bex[3], Geert Borstlap[5], Jasper Devreker[2], Stijn De Weirdt[2], Andy Georges[2], Balázs Hajgató[1], Kenneth Hoste[2], Kurt Lust[5], Samuel Moors[1], Ward Poelmans[1], Mag Selwa[4], Álvaro Simón García[2], Bert Tijskens[5], Jens Timmerman[2], Toon Willems[2]

Acknowledgement: VSCentrum.be

[1]Free University of Brussels
[2]Ghent University
[3]Hasselt University
[4]KU Leuven
[5]University of Antwerp

Audience:

This HPC Tutorial is designed for **researchers** at the **UGent** and affiliated institutes who are in need of computational power (computer resources) and wish to explore and use the High Performance Computing (HPC) core facilities of the Flemish Supercomputing Centre (VSC) to execute their computationally intensive tasks.

The audience may be completely unaware of the HPC concepts but must have some basic understanding of computers and computer programming.

Contents:

This **Beginners Part** of this tutorial gives answers to the typical questions that a new HPC user has. The aim is to learn how to make use of the HPC.

Beginners Part		
Questions	**chapter**	**title**
What is a HPC exactly? Can it solve my computational needs?	**1**	Introduction to HPC
How to get an account?	**2**	Getting an HPC Account
How do I connect to the HPC and transfer my files and programs?	**3**	Connecting to the HPC infrastructure
How to start background jobs?	**4**	Running batch jobs
How to start jobs with user interaction?	**5**	Running interactive jobs
Where do the input and output go? Where to collect my results?	**6**	Running jobs with input/output data
Can I speed up my program by exploring parallel programming techniques?	**7**	Multi core jobs/Parallel Computing
Troubleshooting	**8**	Troubleshooting
What are the rules and priorities of jobs?	**9**	HPC Policies
FAQ	**10**	Frequently Asked Questions

The **Advanced Part** focuses on in-depth issues. The aim is to assist the end-users in running their own software on the HPC.

Advanced Part		
Questions	**chapter**	**title**
What are the optimal Job Specifications?	**11**	Fine-tuning Job Specifications
Can I start many jobs at once?	**12**	Multi-job submission
Can I compile my software on the HPC?	**13**	Compiling and testing your software on the HPC
Can I stop my program and continue later on?	**14**	Checkpointing
Do you have more program examples for me?	**15**	Program examples
Do you have more job script examples for me?	**16**	Job script examples
Any more advice?	**17**	Best Practices
Need a remote display?	**18**	Graphical applications with VNC

The **Software-specific Best Practices Part** focuses on specific programs.

Software-specific Best Practices Part		
MATLAB	**19**	MATLAB
OpenFOAM	**20**	OpenFOAM
Mympirun	**21**	Mympirun
Singularity	**22**	Singularity
SCOOP	**23**	SCOOP
Easybuild	**24**	Easybuild
HOD	**25**	Hanythingondemand (HOD)

The **Annexes** contains some useful reference guides.

Annex	
Title	**chapter**
HPC Quick Reference Guide	**A**
TORQUE options	**B**
Useful Linux Commands	**C**

Notification:

In this tutorial specific commands are separated from the accompanying text:

```
$ commands
```

These should be entered by the reader at a command line in a terminal on the UGent-HPC. They appear in all exercises preceded by a $ and printed in **bold**. You'll find those actions in a grey frame.

Button are menus, buttons or drop down boxes to be pressed or selected.

"Directory" is the notation for directories (called "folders" in Windows terminology) or specific files. (e.g., "/user/home/gent/vsc400/vsc40000")

"Text" Is the notation for text to be entered.

Tip: A "Tip" paragraph is used for remarks or tips.

More support:

Before starting the course, the example programs and configuration files used in this Tutorial must be copied to your home directory, so that you can work with your personal copy. If you have received a new VSC-account, all examples are present in an "/apps/gent/tutorials/Intro-HPC/examples" directory.

```
$ cp -r /apps/gent/tutorials/Intro-HPC/examples ~/
$ cd
$ ls
```

They can also be downloaded from the VSC website at https://www.vscentrum.be. Apart from this HPC Tutorial, the documentation on the VSC website will serve as a reference for all the operations.

Tip: The users are advised to get self-organised. There are only limited resources available at the HPC, which are best effort based. The HPC cannot give support for code fixing, the user applications and own developed software remain solely the responsibility of the end-user.

More documentation can be found at:

1. VSC documentation: https://www.vscentrum.be/en/user-portal
2. External documentation (TORQUE, Moab): http://docs.adaptivecomputing.com

This tutorial is intended for users who want to connect and work on the HPC of the **UGent**.

This tutorial is available in a Windows, Mac or Linux version.

This tutorial is available for UAntwerpen, UGent, KU Leuven, UHasselt and VUB users.

Request your appropriate version at hpc@ugent.be.

Contact Information:

We welcome your feedback, comments and suggestions for improving the HPC Tutorial (contact: hpc@ugent.be).

For all technical questions, please contact the HPC staff:

1. Website: http://www.ugent.be/hpc
2. By e-mail: hpc@ugent.be
3. In real: Directie Informatie- en Communicatietechnologie, Krijgslaan 291 Building S9, 9000 Gent
4. Follow us on Twitter: http://twitter.com/HPCUGent

Mailing-lists:

1. Announcements: hpc-announce@lists.ugent.be (for official announcements and communications)
2. Users: hpc-users@lists.ugent.be (for discussions between users)

Glossary

cluster A group of compute nodes.

compute node The computational units on which batch or interactive jobs are processed. A compute node is pretty much comparable to a single personal computer. It contains one or more sockets, each holding a single CPU. Some nodes also contain one or more GPGPUs. The compute node is equipped with memory (RAM) that is accessible by all its CPUs.

core An individual compute unit inside a CPU. A CPU typically contains one or more cores.

CPU A central processing unit. A CPU is a consumable resource. A compute node typically contains one or more CPUs.

distributed memory system Computing system consisting of many compute nodes connected by a network, each with their own memory. Accessing memory on a neighbouring node is possible but requires explicit communication.

FLOPS FLOPS is short for "Floating-point Operations Per second", i.e. the number of (floating-point) computations that a processor can perform per second.

FTP File Transfer Protocol, used to copy files between distinct machines (over a network.) FTP is unencrypted, and as such blocked on certain systems. SFTP or SCP are secure alternatives to FTP.

GPGPU A general purpose graphical processing unit. A GPGPU is a consumable resource. A GPGPU is a GPU that is used for highly parallel general purpose calculations. A compute node may contain zero, one or more GPGPUs.

grid A group of clusters.

HPC High Performance Computing, high performance computing and multiple-task computing on a supercomputer. The UGent-HPC is the HPC infrastructure at the UGent.

Infiniband A high speed switched fabric computer network communications link used in HPC.

job constraints A set of conditions that must be fulfilled for the job to start.

L1d Level 1 data cache, often called **primary cache**, is a static memory integrated with the CPU core that is used to store data recently accessed by a CPU and also data which may be required by the next operations.

L2d Level 2 data cache, also called **secondary cache**, is a memory that is used to store recently accessed data and also data, which may be required for the next operations. The goal of having the level 2 cache is to reduce data access time in cases when the same data was already accessed before..

L3d Level 3 data cache. Extra cache level built into motherboards between the microprocessor and the main memory.

LAN Local Area Network.

Linux An operating system, similar to UNIX.

LLC The Last Level Cache is the last level in the memory hierarchy before main memory. Any memory requests missing here must be serviced by local or remote DRAM, with significant increase in latency when compared with data serviced by the cache memory.

login node On HPC clusters, login nodes serve multiple functions. From a login node you can submit and monitor batch jobs, analyse computational results, run editors, plots, debuggers, compilers, do housekeeping chores as adjust shell settings, copy files and in general manage your account. You connect to these servers when want to start working on the UGent-HPC.

memory A quantity of physical memory (RAM). Memory is provided by compute nodes. It is required as a constraint or consumed as a consumable resource by jobs. Within Moab, memory is tracked and reported in megabytes (MB).

metrics A measure of some property, activity or performance of a computer sub-system. These metrics are visualised by graphs in, e.g., Ganglia.

Moab Moab is a job scheduler, which allocates resources for jobs that are requesting resources.

modules HPC uses an open source software package called "Environment Modules" (Modules for short) which allows you to add various path definitions to your shell environment.

MPI MPI stands for Message-Passing Interface. It supports a parallel programming method designed for distributed memory systems, but can also be used well on shared memory systems.

node See compute node.

node attribute A node attribute is a non-quantitative aspect of a node. Attributes typically describe the node itself or possibly aspects of various node resources such as processors or memory. While it is probably not optimal to aggregate node and resource attributes together in this manner, it is common practice. Common node attributes include processor architecture, operating system, and processor speed. Jobs often specify that resources be allocated from nodes possessing certain node attributes.

PBS, TORQUE or **OpenPBS** are Open Source resource managers, which are responsible for collecting status and health information from compute nodes and keeping track of jobs running in the system. It is also responsible for spawning the actual executable that is associated with a job, e.g., running the executable on the corresponding compute node. Client commands for submitting and managing jobs can be installed on any host, but in general are installed and used from the Login nodes.

processor A processing unit. A processor is a consumable resource. A processor can be a CPU or a (GP)GPU.

queue PBS/TORQUE queues, or "classes" as Moab refers to them, represent groups of computing resources with specific parameters. A queue with a 12 hour runtime or "walltime" would allow jobs requesting 12 hours or less to use this queue.

scp Secure Copy is a protocol to copy files between distinct machines. SCP or scp is used extensively on HPC clusters to stage in data from outside resources.

scratch Supercomputers generally have what is called scratch space: storage available for temporary use. Use the scratch filesystem when, for example you are downloading and uncompressing applications, reading and writing input/output data during a batch job, or when you work with large datasets. Scratch is generally a lot faster then the Data or Home filesystem.

sftp Secure File Transfer Protocol, used to copy files between distinct machines.

shared memory system Computing system in which all of the CPUs share one global memory space. However, access times from a CPU to different regions of memory are not necessarily uniform. This is called NUMA: Non-uniform memory access. Memory that is closer to the CPU your process is running on will generally be faster to access than memory that is closer to a different CPU. You can pin processes to a certain CPU to ensure they always use the same memory.

SSD Solid-State Drive. This is a kind of storage device that is faster than a traditional hard disk drive.

SSH Secure Shell (SSH), sometimes known as Secure Socket Shell, is a Unix-based command interface and protocol for securely getting access to a remote computer. It is widely used by network administrators to control Web and other kinds of servers remotely. SSH is actually a suite of three utilities - slogin, ssh, and scp - that are secure versions of the earlier UNIX utilities, rlogin, rsh, and rcp. SSH commands are encrypted and secure in several ways. Both ends of the client/server connection are authenticated using a digital certificate, and passwords are protected by encryption. Popular implementations include OpenSSH on Linux/Mac and Putty on Windows.

ssh-keys OpenSSH is a network connectivity tool, which encrypts all traffic including passwords to effectively eliminate eavesdropping, connection hijacking, and other network-level attacks. SSH-keys are part of the OpenSSH bundle. On HPC clusters, ssh-keys allow password-less access between compute nodes while running batch or interactive parallel jobs.

supercomputer A computer with an extremely high processing capacity or processing power.

swap space A quantity of virtual memory available for use by batch jobs. Swap is a consumable resource provided by nodes and consumed by jobs.

TLB Translation Look-aside Buffer, a table in the CPU's memory that contains information about the virtual memory pages the CPU has accessed recently. The table cross-references a program's virtual addresses with the corresponding absolute addresses in physical memory

that the program has most recently used. The TLB enables faster computing because it allows the address processing to take place independent of the normal address-translation pipeline.

walltime Walltime is the length of time specified in the job script for which the job will run on a batch system, you can visuallyse walltime as the time measured by a wall mounted clock (or your digital wrist watch). This is a computational resource.

worker node See compute node.

Contents

Part I

Beginner's Guide

Chapter 1

Introduction to HPC

1.1 What is HPC?

"**High Performance Computing**" (HPC) is computing on a "*Supercomputer*", a computer with at the frontline of contemporary processing capacity – particularly speed of calculation and available memory.

While the supercomputers in the early days (around 1970) used only a few processors, in the 1990s machines with thousands of processors began to appear and, by the end of the 20th century, massively parallel supercomputers with tens of thousands of "off-the-shelf" processors were the norm. A large number of dedicated processors are placed in close proximity to each other in a computer cluster.

A **computer cluster** consists of a set of loosely or tightly connected computers that work together so that in many respects they can be viewed as a single system.

The components of a cluster are usually connected to each other through fast local area networks ("LAN") with each *node* (computer used as a server) running its own instance of an operating system. Computer clusters emerged as a result of convergence of a number of computing trends including the availability of low cost microprocessors, high-speed networks, and software for high performance distributed computing.

Compute clusters are usually deployed to improve performance and availability over that of a single computer, while typically being more cost-effective than single computers of comparable speed or availability.

Supercomputers play an important role in the field of computational science, and are used for a wide range of computationally intensive tasks in various fields, including quantum mechanics, weather forecasting, climate research, oil and gas exploration, molecular modelling (computing the structures and properties of chemical compounds, biological macromolecules, polymers, and crystals), and physical simulations (such as simulations of the early moments of the universe, airplane and spacecraft aerodynamics, the detonation of nuclear weapons, and nuclear fusion). [1]

[1] Wikipedia: http://en.wikipedia.org/wiki/Supercomputer

1.2 What is the UGent-HPC?

The HPC is a collection of computers with Intel CPUs, running a Linux operating system, shaped like pizza boxes and stored above and next to each other in racks, interconnected with copper and fiber cables. Their number crunching power is (presently) measured in hundreds of billions of floating point operations (gigaflops) and even in teraflops.

The UGent-HPC relies on parallel-processing technology to offer UGent researchers an extremely fast solution for all their data processing needs.

The HPC currently consists of:

a set of different compute clusters. For an up to date list of all clusters and their hardware, see https://www.vscentrum.be/infrastructure/hardware/hardware-ugent.

All the nodes in the HPC run "CentOS 7.5 (delcatty, golett, phanpy, skitty, swalot, victini)" with *cpuset* support and *BLCR* modules.

Two tools perform the **job management** and **job scheduling**:

1. TORQUE: a resource manager (based on PBS);
2. Moab: job scheduler and management tools.

1.3 What the HPC infrastucture is *not*

The HPC infrastructure is *not* a magic computer that automatically:

1. runs your PC-applications much faster for bigger problems;
2. develops your applications;
3. solves your bugs;
4. does your thinking;
5. ...
6. allows you to play games even faster.

The HPC does not replace your desktop computer.

1.4 Is the HPC a solution for my computational needs?

1.4.1 Batch or interactive mode?

Typically, the strength of a supercomputer comes from its ability to run a huge number of programs (i.e., executables) in parallel without any user interaction in real time. This is what is called "running in batch mode".

It is also possible to run programs at the HPC, which require user interaction. (pushing buttons, entering input data, etc.). Although technically possible, the use of the HPC might not always be the best and smartest option to run those interactive programs. Each time some user interaction is needed, the computer will wait for user input. The available computer resources (CPU, storage, network, etc.) might not be optimally used in those cases. A more in-depth analysis with the HPC staff can unveil whether the HPC is the desired solution to run interactive programs. Interactive mode is typically only useful for creating quick visualisations of your data without having to copy your data to your desktop and back.

1.4.2 What are cores, processors and nodes?

In this manual, the terms core, processor and node will be frequently used, so it's useful to understand what they are.

Modern servers, also referred to as *(worker)nodes* in the context of HPC, include one or more sockets, each housing a multi-*core processor* (next to memory, disk(s), network cards, ...). A modern *processor* consists of multiple CPUs or *cores* that are used to execute *computations*.

1.4.3 Parallel or sequential programs?

Parallel programs

Parallel computing is a form of computation in which many calculations are carried out simultaneously. They are based on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel").

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism, with multicore computers having multiple processing elements within a single machine, while clusters use multiple computers to work on the same task. Parallel computing has become the dominant computer architecture, mainly in the form of multicore processors.

The two parallel programming paradigms most used in HPC are:

- OpenMP for shared memory systems (multithreading): on multiple cores of a single node
- MPI for distributed memory systems (multiprocessing): on multiple nodes

Parallel programs are more difficult to write than sequential ones, because concurrency introduces several new classes of potential software bugs, of which race conditions are the most common. Communication and synchronisation between the different subtasks are typically some of the greatest obstacles to getting good parallel program performance.

Sequential programs

Sequential software does not do calculations in parallel, i.e. it only uses *one single core of a single workernode*. **It does not become faster by just throwing more cores at it**: it can only use one core.

It is perfectly possible to also run purely **sequential programs** on the HPC.

Running your sequential programs on the most modern and fastest computers in the HPC can save you a lot of time. But it also might be possible to run multiple instances of your program (e.g., with different input parameters) on the HPC, in order to solve one overall problem (e.g., to perform a parameter sweep). This is another form of running your sequential programs in parallel.

1.4.4 What programming languages can I use?

You can use *any* programming language, *any* software package and *any* library provided it has a version that runs on Linux, specifically, on the version of Linux that is installed on the compute nodes, CentOS 7.5 (delcatty, golett, phanpy, skitty, swalot, victini).

For the most common **programming languages**, a compiler is available on CentOS 7.5 (delcatty, golett, phanpy, skitty, swalot, victini). Supported and common programming languages on the HPC are C/C++, FORTRAN, Java, Perl, Python, MATLAB, R, etc.

Supported and commonly used compilers are GCC and Intel.

Additional software can be installed "*on demand*". Please contact the HPC staff to see whether the HPC can handle your specific requirements.

1.4.5 What operating systems can I use?

All nodes in the HPC cluster run under CentOS 7.5 (delcatty, golett, phanpy, skitty, swalot, victini), which is a specific version of Red Hat Enterprise Linux. This means that all programs (executables) should be compiled for CentOS 7.5 (delcatty, golett, phanpy, skitty, swalot, victini).

Users can connect from any computer in the UGent network to the HPC, regardless of the Operating System that they are using on their personal computer. Users can use any of the common Operating Systems (such as Windows, macOS or any version of Linux/Unix/BSD) and run and control their programs on the HPC.

A user does not need to have prior knowledge about Linux; all of the required knowledge is explained in this tutorial.

1.4.6 What does a typical workflow look like?

A typical workflow looks like:

1. Connect to the login nodes with SSH (see section 3.1)
2. Transfer your files to the cluster (see section 3.2)

3. Optional: compile your code and test it (for compiling, see chapter 13)
4. Create a job script and submit your job (see chapter 4)
5. Get some coffee and be patient:
   (a) Your job gets into the queue
   (b) Your job gets executed
   (c) Your job finishes
6. Study the results generated by your jobs, either on the cluster or after downloading them locally.

1.4.7 What is the next step?

When you think that the HPC is a useful tool to support your computational needs, we encourage you to acquire a VSC-account (as explained in chapter 2), read chapter 3, "Setting up the environment", and explore chapters 5 to 11 which will help you to transfer and run your programs on the HPC cluster.

Do not hesitate to contact the HPC staff for any help.

Chapter 2

Getting an HPC Account

2.1 Getting ready to request an account

All users of AUGent can request an account on the HPC, which is part of the Flemish Supercomputing Centre (VSC).

See chapter 9 for more information on who is entitled to an account.

The VSC, abbreviation of Flemish Supercomputer Centre, is a virtual supercomputer centre. It is a partnership between the five Flemish associations: the Association KU Leuven, Ghent University Association, Brussels University Association, Antwerp University Association and the University Colleges-Limburg. The VSC is funded by the Flemish Government.

The UGent-HPC clusters use public/private key pairs for user authentication (rather than passwords). Technically, the private key is stored on your local computer and always stays there; the public key is stored on the HPC. Access to the HPC is granted to anyone who can prove to have access to the corresponding private key on his local computer.

2.1.1 How do SSH keys work?

- an SSH public/private key pair can be seen as a lock and a key
- the SSH public key is equivalent with a *lock*: you give it to the VSC and they put it on the *door* that gives access to your account.
- the SSH private key is like a *physical key*: you don't hand it out to other people.
- anyone who has the key (and the optional password) can unlock the *door* and log in to the account.
- the *door* to your VSC account is special: it can have multiple *locks* (SSH public keys) attached to it, and you only need to open one *lock* with the corresponding *key* (SSH private key) to open the *door* (log in to the account).

Since all VSC clusters use Linux as their main operating system, you will need to get acquainted with using the command-line interface and using the terminal.

A typical Windows environment does not come with pre-installed software to connect and run command-line executables on a HPC. Some tools need to be installed on your Windows machine first, before we can start the actual work.

2.1.2 Get PuTTY: A free telnet/SSH client

We recommend to use the PuTTY tools package, which is freely available.

You do not need to install PuTTY, you can download the PuTTY and PuTTYgen executable and run it. This can be useful in situations where you do not have the required permissions to install software on the computer you are using. Alternatively, an installation package is also available.

You can download PuTTY from the official address: `https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html`. You probably want the 64-bits version. If you can install software on your computer, you can use the "Package files", if not, you can download and use `putty.exe` and `puttygen.exe` in the "Alternative binary files" section.

The PuTTY package consists of several components, but we'll only use two:

1. **PuTTY**: *the Telnet and SSH client itself* (to login, see subsection 3.1.1)

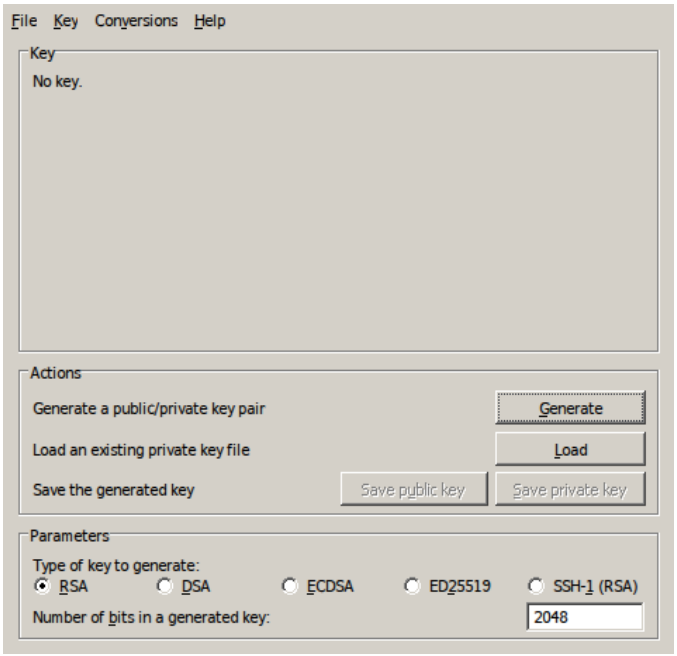
2. **PuTTYgen**: *an RSA and DSA key generation utility* (to generate a key pair, see subsection 2.1.3)

2.1.3 Generating a public/private key pair

Before requesting a VSC account, you need to generate a pair of *ssh* keys. You need 2 keys, a public and a private key. You can visualise the public key as a lock to which only you have the key (your private key). You can send a copy of your lock to anyone without any problems, because only you can open it, as long as you keep your private key secure. To generate a public/private key pair, you can use the PuTTYgen key generator.

Start ***PuTTYgen.exe*** it and follow these steps:

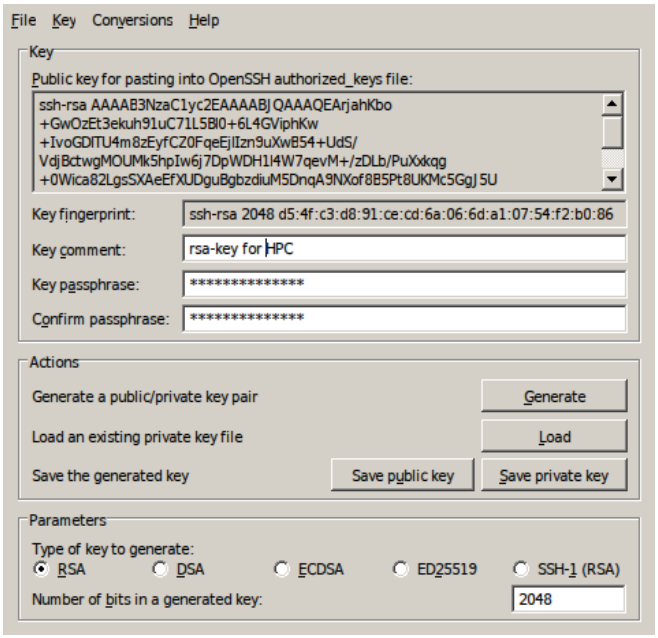
1. In Parameters (at the bottom of the window), choose "RSA" and keep the number of bits in the key to 2048.



2. Click on Generate. To generate the key, you must move the mouse cursor over the PuTTYgen window (this generates some random data that PuTTYgen uses to generate the key pair). Once the key pair is generated, your public key is shown in the field Public key for pasting into OpenSSH authorized_keys file.

3. Next, it is advised to fill in the Key comment field to make it easier identifiable afterwards.

4. Next, you should specify a passphrase in the Key passphrase field and retype it in the Confirm passphrase field. Remember, the passphrase protects the private key against unauthorised use, so it is best to choose one that is not too easy to guess but that you can still remember. Using a passphrase is not required, but we recommend you to use a good passphrase unless you are certain that your computer's hard disk is encrypted with a decent password. (If you are not sure your disk is encrypted, it probably isn't.)



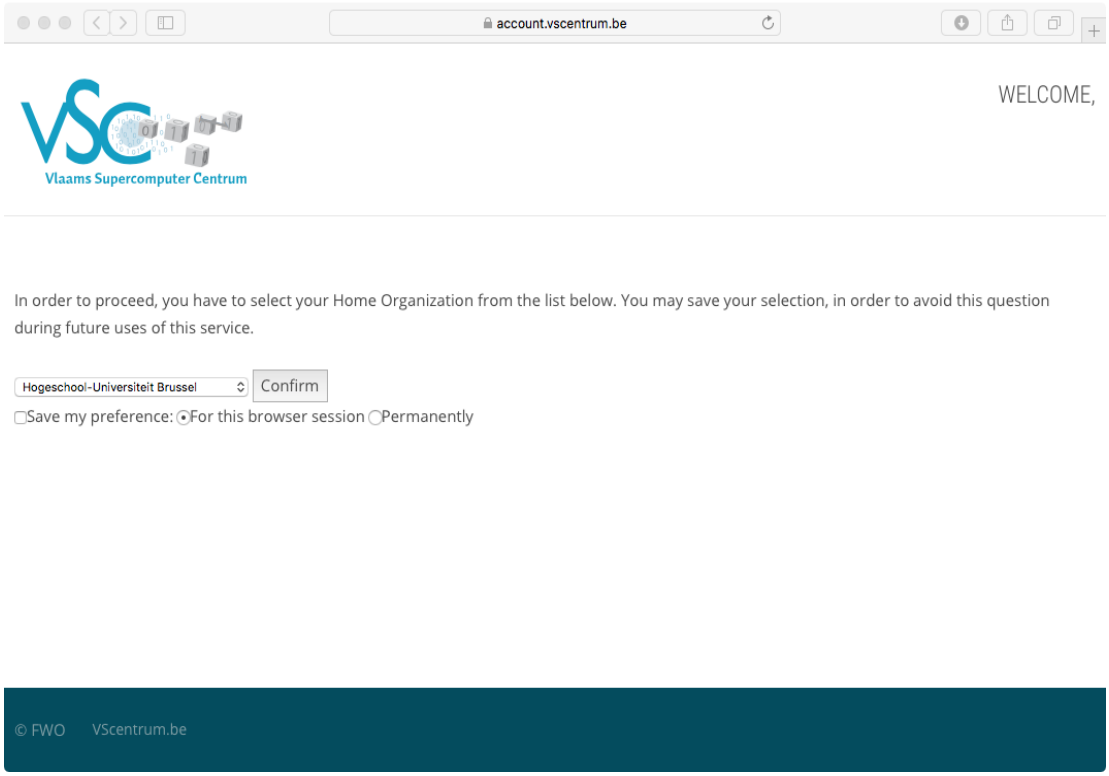
5. Finally, save both the public and private keys in a folder on your personal computer (We recommend to create and put them in the folder "C:\Users\%USERNAME%\AppData\Local\PuTTY\.ssh") with the buttons Save public key and Save private key. We recommend using the name "**id_rsa.pub**" for the public key, and "**id_rsa.ppk**" for the private key.

If you use another program to generate a key pair, please remember that they need to be in the OpenSSH format to access the HPC clusters.

2.2 Applying for the account

Visit https://account.vscentrum.be/

You will be redirected to our WAYF (Where Are You From) service where you have to select your "Home Organisation".



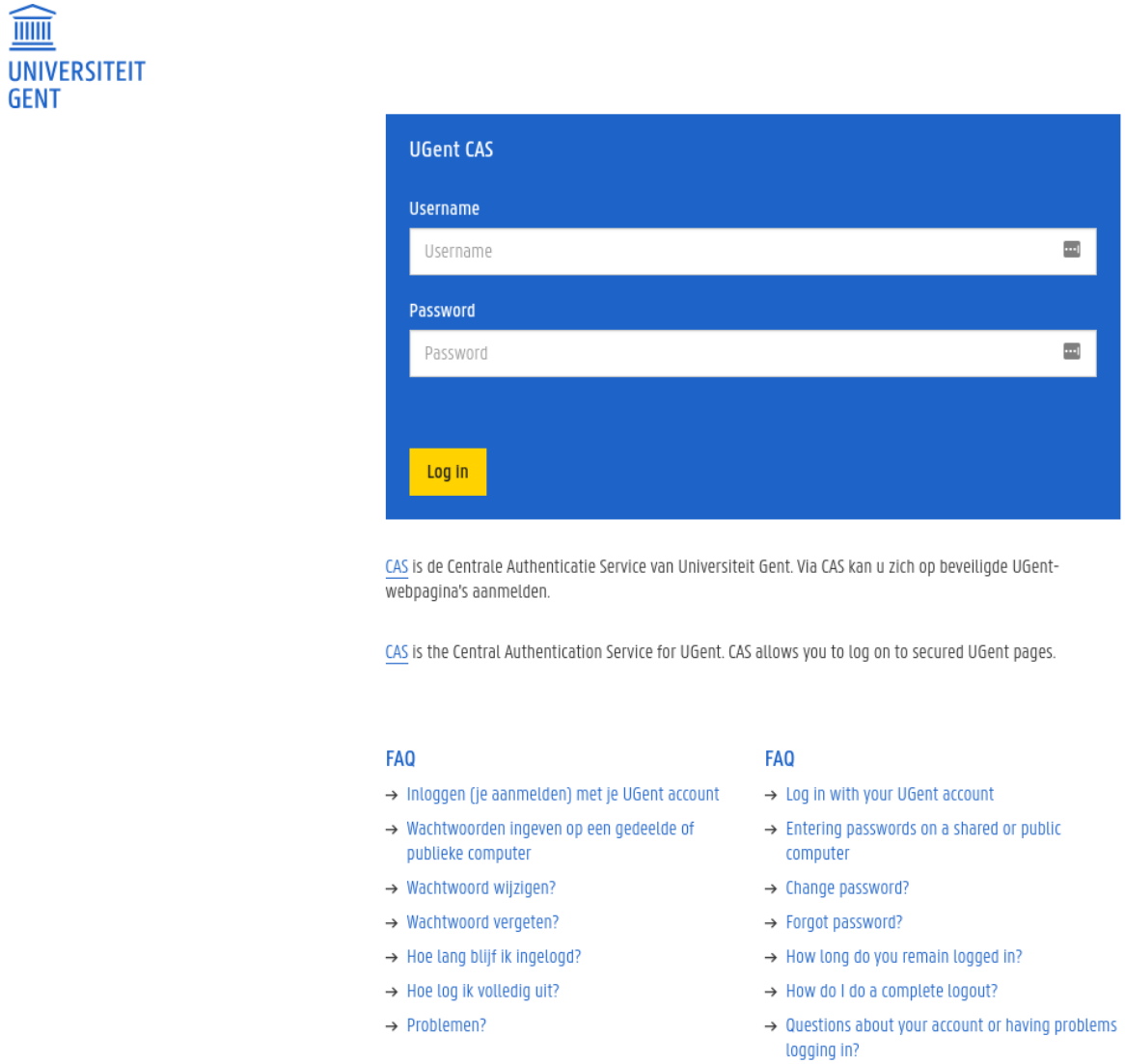
Select 'UGent' in the dropdown box and optionaly select 'Save my preference' and 'permanently.'

Click Confirm

You will now be taken to the authentication page of your institute.

You will now have to log in with CAS using your UGent account.

You either have a login name of maximum 8 characters, or a (non-UGent) email address if you are an external user. In case of problems with your UGent password, please visit: https://password.ugent.be/. After logging in, you may be requested to share your information. Click "Yes, continue".



After you log in using your UGent login and password, you will be asked to upload the file that contains your public key, i.e., the file "id_rsa.pub" which you have generated earlier.

This file should have been stored in the directory "C:\Users\%USERNAME%\AppData\Local\PuTTY\.ssh"

After you have uploaded your public key you will receive an e-mail with a link to confirm your e-mail address. After confirming your e-mail address the VSC staff will review and if applicable approve your account.

2.2.1 Welcome e-mail

Within one day, you should receive a Welcome e-mail with your VSC account details.

```
Dear (Username),
Your VSC-account has been approved by an administrator.
Your vsc-username is vsc40000

Your account should be fully active within one hour.

To check or update your account information please visit
https://account.vscentrum.be/

For further info please visit https://www.vscentrum.be/en/user-portal

Kind regards,
-- The VSC administrators
```

Now, you can start using the HPC. You can always look up your vsc id later by visiting https://account.vscentrum.be.

2.2.2 Adding multiple SSH public keys (optional)

In case you are connecting from different computers to the login nodes, it is advised to use separate SSH public keys to do so. You should follow these steps.

1. Create a new public/private SSH key pair from Putty. Repeat the process described in section 2.1.3.
2. Go to https://account.vscentrum.be/django/vo/edit
3. Upload the new SSH public key using the **Add public key** section.
4. (optional) If you lost your key, you can delete the old key on the same page. You should keep at least one valid public SSH key in your account.
5. Take into account that it will take some time before the new SSH public key is active in your account on the system; waiting for 15-30 minutes should be sufficient.

2.3 Computation Workflow on the HPC

A typical Computation workflow will be:

1. Connect to the HPC
2. Transfer your files to the HPC
3. Compile your code and test it
4. Create a job script
5. Submit your job
6. Wait while
   (a) your job gets into the queue

   (b) your job gets executed
   (c) your job finishes
7. Move your results

We'll take you through the different tasks one by one in the following chapters.

Chapter 3

Connecting to the HPC infrastructure

Before you can really start using the HPC clusters, there are several things you need to do or know:

1. You need to **log on to the cluster** using an SSH client to one of the login nodes. This will give you command-line access. The software you'll need to use on your client system depends on its operating system.
2. Before you can do some work, you'll have to **transfer the files** that you need from your desktop computer to the cluster. At the end of a job, you might want to transfer some files back.
3. Optionally, if you wish to use programs with a **graphical user interface**, you will need an X-server on your client system and log in to the login nodes with X-forwarding enabled.
4. Often several versions of **software packages and libraries** are installed, so you need to select the ones you need. To manage different versions efficiently, the VSC clusters use so-called **modules**, so you will need to select and load the modules that you need.

3.1 First Time connection to the HPC infrastucture

If you have any issues connecting to the HPC after you've followed these steps, see section 8.3 to troubleshoot.

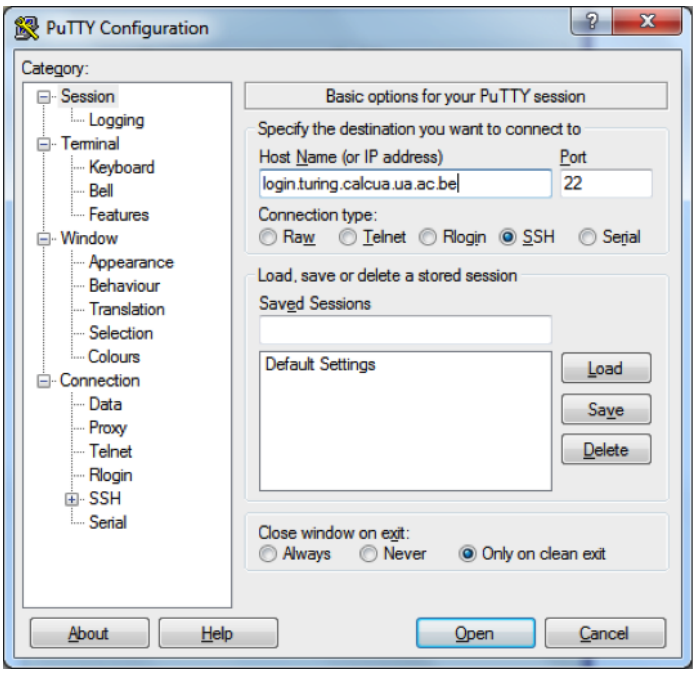
3.1.1 Open a Terminal

You've generated a public/private key pair with PuTTYgen and have an approved account on the VSC clusters. The next step is to setup the connection to (one of) the HPC.

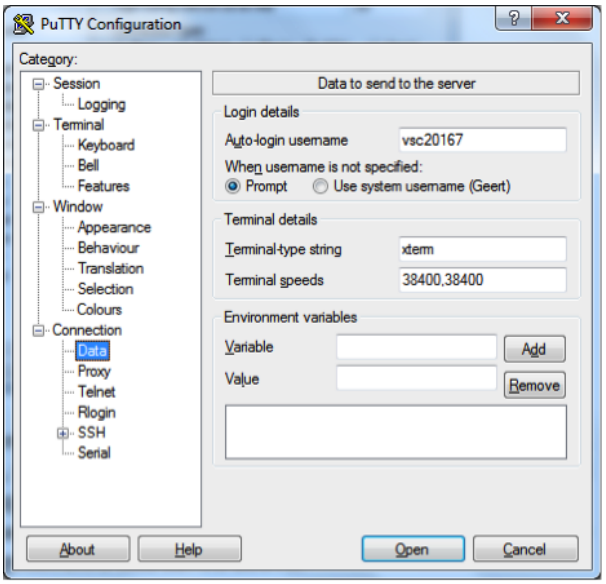
In the screenshots, we show the setup for user "***vsc20167***" to the HPC cluster via the login node "***login.hpc.ugent.be***".

1. Start the PuTTY executable `putty.exe` in your directory `C:\Program Files (x86)\PuTTY` and the configuration screen will pop up. As you will often use the PuTTY tool, we recommend adding a shortcut on your desktop.

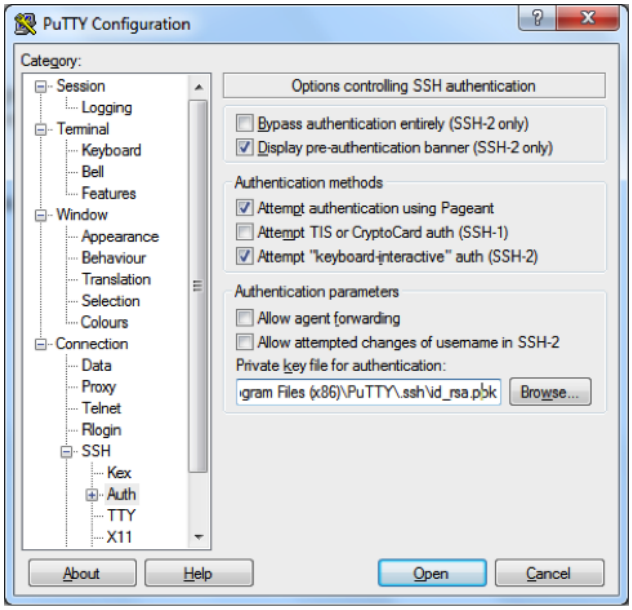
2. Within the category <***Session***>, in the field <***Host Name***>, enter the name of the login node of the HPC cluster (i.e., "***login.hpc.ugent.be***") you want to connect to.



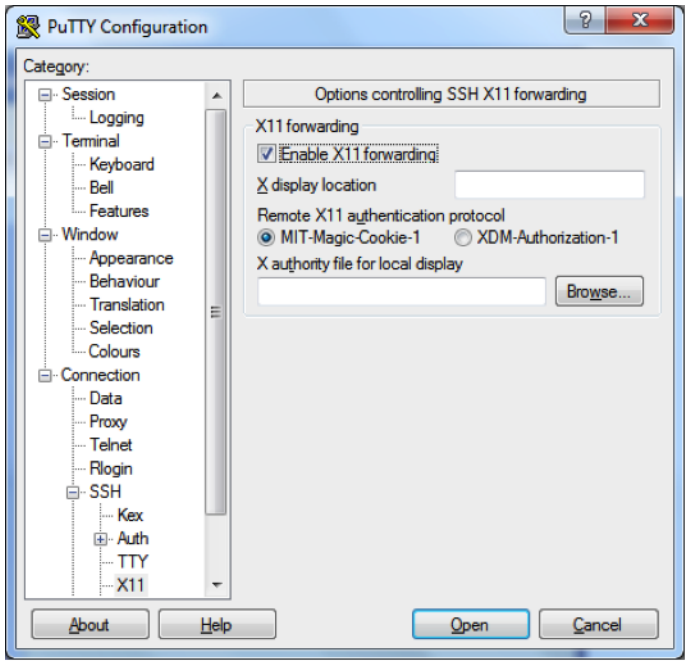
3. In the category Connection ⟩ Data, in the field Auto-login username, put in <*vsc40000*>, which is your VSC username that you have received by e-mail after your request was approved.



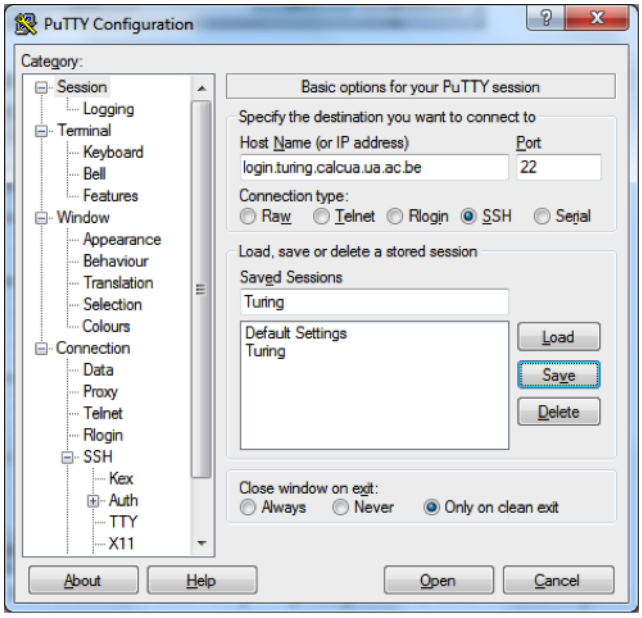
4. In the category Connection ⟩ SSH ⟩ Auth, in the field Private key file for authentication click on Browse and select the private key (i.e., "id_rsa.ppk") that you generated and saved above.



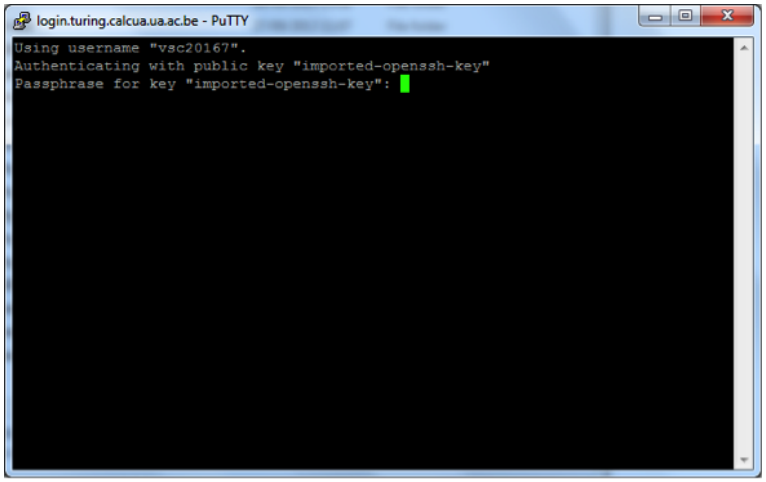
5. In the category Connection > SSH > X11, click the Enable X11 Forwarding checkbox.



6. Now go back to <Session>, and fill in "*hpcugent*" in the Saved Sessions field and press Save to store the session information.



7. Now pressing Open, will open a terminal window and asks for you passphrase.



8. If this is your first time connecting, you will be asked to verify the authenticity of the login node. Please see section 8.6 on how to do this.

9. After entering your correct passphrase, you will be connected to the login-node of the HPC.

10. To check you can now "Print the Working Directory" (pwd) and check the name of the computer, where you have logged in (hostname):

```
$ pwd
/user/home/gent/vsc400/vsc40000
$ hostname -f
gligar04.gastly.os
```

11. For future PuTTY sessions, just select your saved session (i.e. "*hpcugent*") from the list, Load it and press Open.

Congratulations, you're on the HPC infrastructure now! To find out where you have landed you can print the current working directory:

```
$ pwd
/user/home/gent/vsc400/vsc40000
```

Your new private home directory is "/user/home/gent/vsc400/vsc40000". Here you can create your own subdirectory structure, copy and prepare your applications, compile and test them and submit your jobs on the HPC.

```
$ cd /apps/gent/tutorials
$ ls
Intro-HPC/
```

This directory currently contains all training material for the ***Introduction to the HPC***. More relevant training material to work with the HPC can always be added later in this directory.

You can now explore the content of this directory with the "ls –l" (**l**ist**s** **l**ong) and the "cd" (**c**hange **d**irectory) commands:

As we are interested in the use of the ***HPC***, move further to ***Intro-HPC*** and explore the contents up to 2 levels deep:

```
$ cd Intro-HPC
$ tree -L 2
.
`-- examples
    |-- Compiling-and-testing-your-software-on-the-HPC
    |-- Fine-tuning-Job-Specifications
    |-- Multi-core-jobs-Parallel-Computing
    |-- Multi-job-submission
    |-- Program-examples
    |-- Running-batch-jobs
    |-- Running-jobs-with-input
    |-- Running-jobs-with-input-output-data
    |-- example.pbs
    `-- example.sh
9 directories, 5 files
```

This directory contains:

1. This ***HPC Tutorial*** (in either a Mac, Linux or Windows version).
2. An ***examples*** subdirectory, containing all the examples that you need in this Tutorial, as well as examples that might be useful for your specific applications.

```
$ cd examples
```

Tip: Typing `cd ex` followed by ⇥ (the Tab-key) will generate the `cd examples` command. **Command-line completion** (also tab completion) is a common feature of the bash command line interpreter, in which the program automatically fills in partially typed commands.

Tip: For more exhaustive tutorials about Linux usage, see Appendix C

The first action is to copy the contents of the HPC examples directory to your home directory,

so that you have your own personal copy and that you can start using the examples. The "-r" option of the copy command will also copy the contents of the sub-directories "*recursively*".

```
$ cp -r /apps/gent/tutorials/Intro-HPC/examples ~/
```

You will now make a doc directory in your home dir and copy the latest version of this document to this directory. We will use this later in this tutorial.

```
$ mkdir ~/docs
$ cp -r /apps/gent/tutorials/intro-HPC-windows-gent ~/
```

Go to your home directory, check your own private examples directory, ... and start working.

```
$ cd
$ ls -l
```

Upon connecting you will see a login message containing your last login time stamp and a basic overview of the current cluster utilisation.

```
Last login: Mon Sep  3 14:00:00 2018 from helios.ugent.be
STEVIN HPC-UGent infrastructure status on Mon, 03 Sep 2018 14:30:00

   cluster - full - free -  part - total - running - queued
             nodes  nodes   free   nodes    jobs      jobs
-----------------------------------------------------------------
  delcatty     72      0     52     127      N/A       N/A
    golett    168      2     16     200      N/A       N/A
    phanpy     15      0      1      16      N/A       N/A
    swalot      0      0      0     128      N/A       N/A
    skitty     69      2      0      73      N/A       N/A
   victini     83      1      3      96      N/A       N/A

For a full view of the current loads and queues see:
 http://hpc.ugent.be/clusterstate/
Updates on maintenance and unscheduled downtime can be found on
 https://www.vscentrum.be/en/user-portal/system-status
```

You can exit the connection at anytime by entering:

```
$ exit
logout
Connection to login.hpc.ugent.be closed.
```

Tip: Setting your Language right:

You may encounter a warning message similar to the following one during connecting:

```
perl: warning: Setting locale failed.
perl: warning: Please check that your locale settings:
LANGUAGE = (unset),
LC_ALL = (unset),
LC_CTYPE = "UTF-8",
LANG = (unset)
    are supported and installed on your system.
perl: warning: Falling back to the standard locale ("C").
```

or any other error message complaining about the locale.

This means that the correct "locale" has not yet been properly specified on your local machine. Try:

```
$ locale
LANG=
LC_COLLATE="C"
LC_CTYPE="UTF-8"
LC_MESSAGES="C"
LC_MONETARY="C"
LC_NUMERIC="C"
LC_TIME="C"
LC_ALL=
```

A **locale** is a set of parameters that defines the user's language, country and any special variant preferences that the user wants to see in their user interface. Usually a locale identifier consists of at least a language identifier and a region identifier.

3.2 Transfer Files to/from the HPC

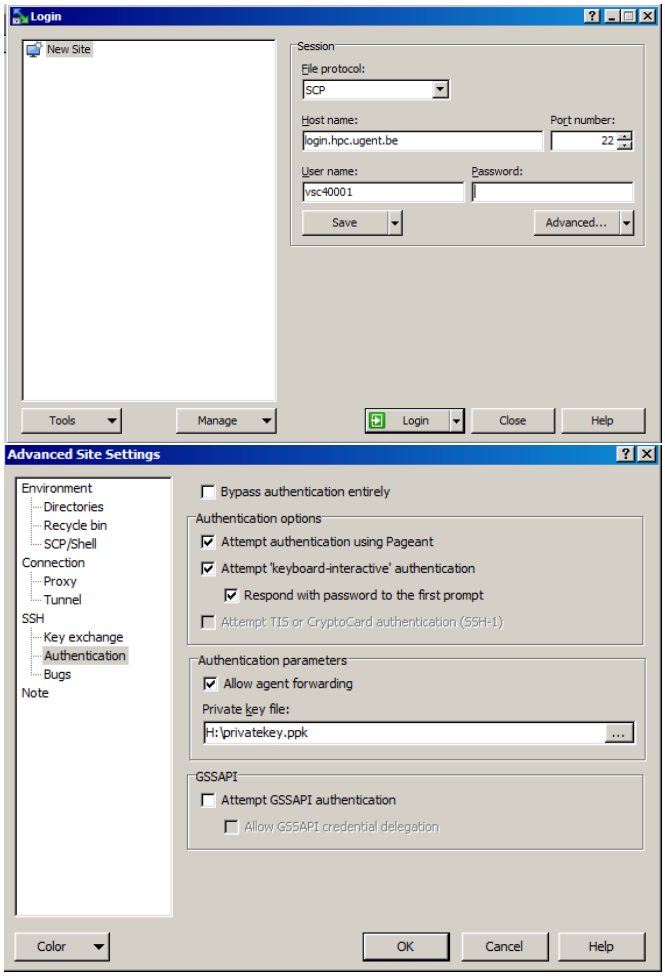
Before you can do some work, you'll have to **transfer the files** you need from your desktop or department to the cluster. At the end of a job, you might want to transfer some files back.

3.2.1 WinSCP

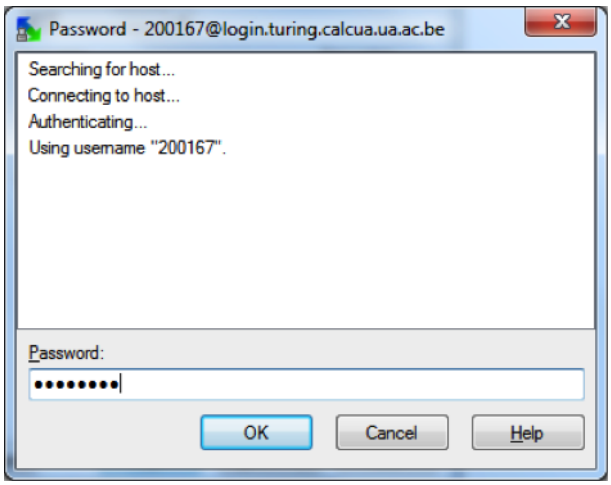
To transfer files to and from the cluster, we recommend the use of WinSCP, which is a graphical ftp-style program (but than one that uses the ssh way of communicating with the cluster rather then the less secure ftp) that is also freely available. WinSCP can be downloaded both as an installation package and as a standalone portable executable from http://www.winscp.net

To transfer your files using WinSCP,

1. Open the program
2. Fill in the necessary fields under Session
   (a) Press New.
   (b) Enter "*login.hpc.ugent.be*" in the Host name field.
   (c) Put your "*vsc-account*" in User name field.
   (d) Select SCP as the file protocol.
   (e) Note that the password field remains empty.
   (f) Click advanced.
   (g) Click Authentication (under SSH).
   (h) Select your private key in the field Private key file.



1. By pressing on the Save button, you can save the session under Stored sessions for future access.
2. Finally, when clicking on Login, you will be asked for your key passphrase.

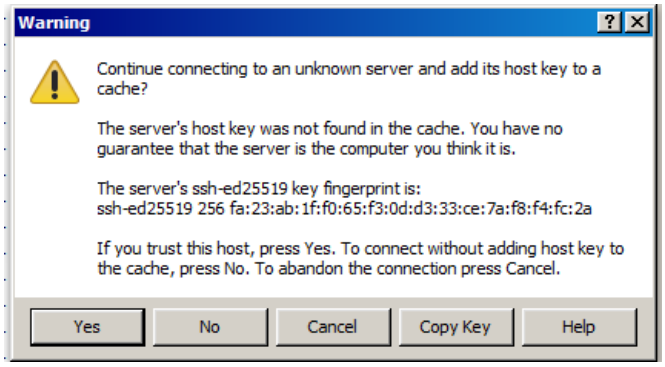


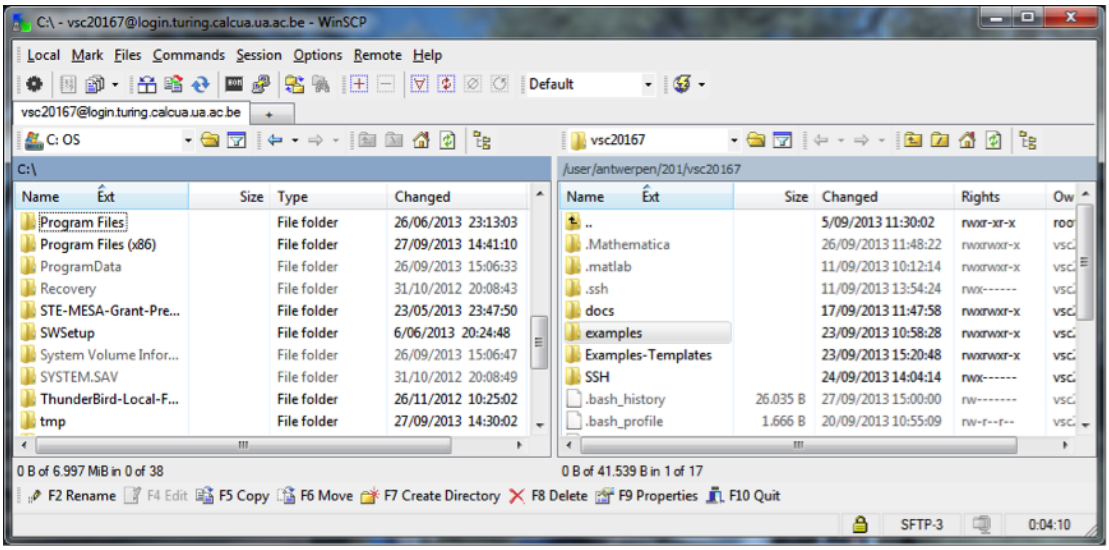
The first time you make a connection to the login node, a Security Alert will appear and you will be asked to verify the authenticity of the login node.

Make sure the fingerprint in the alert matches one of the following:

- `ssh-rsa 2048 2f:0c:f7:76:87:57:f7:5d:2d:7b:d1:a1:e1:86:19:f3`
- `ssh-ed25519 256 fa:23:ab:1f:f0:65:f3:0d:d3:33:ce:7a:f8:f4:fc:2a`
- `ssh-ecdsa 256 13:f0:11:d1:94:cb:ca:e5:ca:82:21:62:ab:9f:3f:c2`

If it does, press ***Yes***, if it doesn't, please contact hpc@ugent.be.





Now, try out whether you can transfer an arbitrary file from your local machine to the HPC and back.

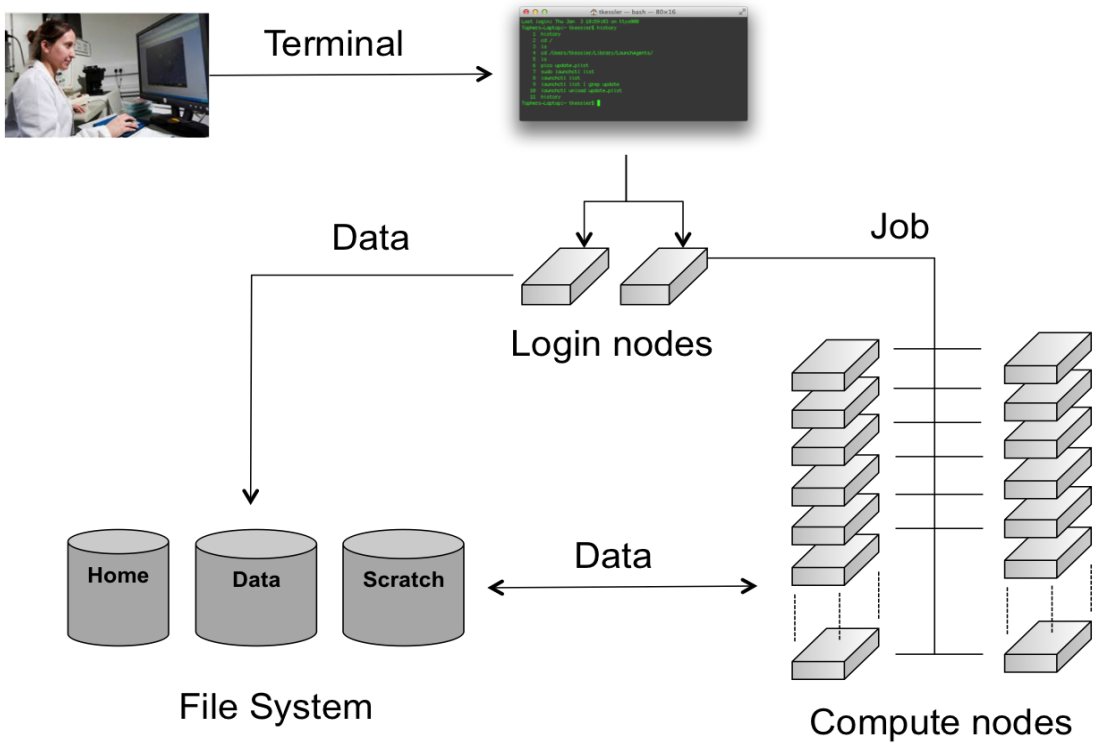
3.2.2 Fast file transfer for large datasets

See the section on `rsync` in chapter 5 of the Linux intro manual.

Chapter 4

Running batch jobs

In order to have access to the compute nodes of a cluster, you have to use the job system. The system software that handles your batch jobs consists of two pieces: the queue- and resource manager **TORQUE** and the scheduler **Moab**. Together, TORQUE and Moab provide a suite of commands for submitting jobs, altering some of the properties of waiting jobs (such as reordering or deleting them), monitoring their progress and killing ones that are having problems or are no longer needed. Only the most commonly used commands are mentioned here.



When you connect to the HPC, you have access to (one of) the **login nodes** of the cluster. There you can prepare the work you want to get done on the cluster by, e.g., installing or compiling programs, setting up data sets, etc. The computations however, should not be performed on this login node. The actual work is done on the cluster's **compute nodes**. Each compute node

contains a number of CPU **cores**. The compute nodes are managed by the job scheduling software (Moab) and a Resource Manager (TORQUE), which decides when and on which compute nodes the jobs can run. It is usually not necessary to log on to the compute nodes directly and is only allowed on the nodes where you have a job running . Users can (and should) monitor their jobs periodically as they run, but do not have to remain connected to the HPC the entire time.

The documentation in this "Running batch jobs" section includes a description of the general features of job scripts, how to submit them for execution and how to monitor their progress.

4.1 Modules

Software installation and maintenance on a HPC cluster such as the VSC clusters poses a number of challenges not encountered on a workstation or a departmental cluster. We therefore need a system on the HPC, which is able to easily activate or deactivate the software packages that you require for your program execution.

4.1.1 Environment Variables

The program environment on the HPC is controlled by pre-defined settings, which are stored in environment (or shell) variables. For more information about environment variables, see the chapter "Getting started", section "Variables" in the intro to Linux.

All the software packages that are installed on the HPC cluster require different settings. These packages include compilers, interpreters, mathematical software such as MATLAB and SAS, as well as other applications and libraries.

4.1.2 The module command

In order to administer the active software and their environment variables, the module system has been developed, which:

1. Activates or deactivates *software packages* and their dependencies.
2. Allows setting and unsetting of *environment variables*, including adding and deleting entries from list-like environment variables.
3. Does this in a *shell-independent* fashion (necessary information is stored in the accompanying module file).
4. Takes care of *versioning aspects*: For many libraries, multiple versions are installed and maintained. The module system also takes care of the versioning of software packages. For instance, it does not allow multiple versions to be loaded at same time.
5. Takes care of *dependencies*: Another issue arises when one considers library versions and the dependencies they require. Some software requires an older version of a particular library to run correctly (or at all). Hence a variety of version numbers is available for important libraries. Modules typically load the required dependencies automatically.

This is all managed with the `module` command, which is explained in the next sections.

There is also a shorter `ml` command that does exactly the same as the `module` command and is easier to type. Whenever you see a `module` command, you can replace `module` with `ml`.

4.1.3 Available modules

A large number of software packages are installed on the HPC clusters. A list of all currently available software can be obtained by typing:

```
$ module available
```

It's also possible to execute `module av` or `module avail`, these are shorter to type and will do the same thing.

This will give some output such as:

```
$ module av 2>&1 | more
--- /apps/gent/SL6/sandybridge/modules/all ---
ABAQUS/6.12.1-linux-x86_64
AMOS/3.1.0-ictce-4.0.10
ant/1.9.0-Java-1.7.0_40
ASE/3.6.0.2515-ictce-4.1.13-Python-2.7.3
ASE/3.6.0.2515-ictce-5.5.0-Python-2.7.6
...
```

Or when you want to check whether some specific software, some compiler or some application (e.g., Matlab) is installed on the HPC.

```
$ module av 2>&1 | grep -i -e "matlab"
MATLAB/2010b
MATLAB/2012b
MATLAB/2013b
```

As you are not aware of the capitals letters in the module name, we looked for a case-insensitive name with the "-i" option.

This gives a full list of software packages that can be loaded.

The casing of module names is important: lowercase and uppercase letters matter in module names.

4.1.4 Organisation of modules in toolchains

The amount of modules on the VSC systems can be overwhelming, and it is not always immediately clear which modules can be loaded safely together if you need to combine multiple programs in a single job to get your work done.

Therefore the VSC has defined so-called **toolchains**. A toolchain contains a C/C++ and Fortran compiler, a MPI library and some basic math libraries for (dense matrix) linear algebra and FFT. Two toolchains are defined on most VSC systems. One, the `intel` toolchain, consists of the Intel compilers, MPI library and math libraries. The other one, the `foss` toolchain, consists of Open Source components: the GNU compilers, OpenMPI, OpenBLAS and the standard LAPACK

and ScaLAPACK libraries for the linear algebra operations and the FFTW library for FFT. The toolchains are refreshed twice a year, which is reflected in their name.

E.g., `foss/2018a` is the first version of the `foss` toolchain in 2018.

The toolchains are then used to compile a lot of the software installed on the VSC clusters. You can recognise those packages easily as they all contain the name of the toolchain after the version number in their name (e.g. `Python/2.7.12-intel-2016b`). Only packages compiled with the same toolchain name and version can work together without conflicts.

4.1.5 Loading and unloading modules

module load

To "activate" a software package, you load the corresponding module file using the `module load` command:

```
$ module load example
```

This will load the most recent version of *example*.

For some packages, multiple versions are installed; the load command will automatically choose the default version (if it was set by the system administrators) or the most recent version otherwise (i.e., the lexicographical last after the `/`).

However, you should specify a particular version to avoid surprises when newer versions are installed:

```
$ module load secondexample/2.7-intel-2016b
```

The `ml` command is a shorthand for `module load`: `ml example/1.2.3` is equivalent to `module load example/1.2.3`.

Modules need not be loaded one by one; the two `module load` commands can be combined as follows:

```
$ module load example/1.2.3 secondexample/2.7-intel-2016b
```

This will load the two modules as well as their dependencies (unless there are conflicts between both modules).

module list

Obviously, you need to be able to keep track of the modules that are currently loaded. Assuming you have run the `module load` commands stated above, you will get the following:

```
$ module list
Currently Loaded Modulefiles:
1) example/1.2.3                                        6) imkl/11.3.3.210-iimpi
    -2016b
2) GCCcore/5.4.0                                        7) intel/2016b
3) icc/2016.3.210-GCC-5.4.0-2.26                        8) examplelib/1.2-intel
    -2016b
4) ifort/2016.3.210-GCC-5.4.0-2.26                      9) secondexample/2.7-intel
    -2016b
5) impi/5.1.3.181-iccifort-2016.3.210-GCC-5.4.0-2.26
```

You can also just use the `ml` command without arguments to list loaded modules.

It is important to note at this point that other modules (e.g., `intel/2016b`) are also listed, although the user did not explicitly load them. This is because `secondexample/2.7-intel-2016b` depends on it (as indicated in its name), and the system administrator specified that the `intel/2016b` module should be loaded whenever *this* `secondexample` module is loaded. There are advantages and disadvantages to this, so be aware of automatically loaded modules whenever things go wrong: they may have something to do with it!

module unload

To unload a module, one can use the `module unload` command. It works consistently with the `load` command, and reverses the latter's effect. However, the dependencies of the package are NOT automatically unloaded; you will have to unload the packages one by one. When the `secondexample` module is unloaded, only the following modules remain:

```
$ module unload secondexample
$ module list
Currently Loaded Modulefiles:
Currently Loaded Modulefiles:
1) example/1.2.3                                        5) impi/5.1.3.181-iccifort
    -2016.3.210-GCC-5.4.0-2.26
2) GCCcore/5.4.0                                        6) imkl/11.3.3.210-iimpi
    -2016b
3) icc/2016.3.210-GCC-5.4.0-2.26                        7) intel/2016b
4) ifort/2016.3.210-GCC-5.4.0-2.26                      8) examplelib/1.2-intel
    -2016b
```

To unload the `secondexample` module, you can also use `ml -secondexample`.

Notice that the version was not specified: there can only be one version of a module loaded at a time, so unloading modules by name is not ambiguous. However, checking the list of currently loaded modules is always a good idea, since unloading a module that is currently not loaded will *not* result in an error.

4.1.6 Purging all modules

In order to unload all modules at once, and hence be sure to start in a clean state, you can use:

```
$ module purge
```

This is always safe: the `cluster` module (the module that specifies which cluster jobs will get

submitted to) will not be unloaded (because it's a so-called "sticky" module).

4.1.7 Using explicit version numbers

Once a module has been installed on the cluster, the executables or libraries it comprises are never modified. This policy ensures that the user's programs will run consistently, at least if the user specifies a specific version. **Failing to specify a version may result in unexpected behaviour**.

Consider the following example: the user decides to use the `example` module and at that point in time, just a single version 1.2.3 is installed on the cluster. The user loads the module using:

```
$ module load example
```

rather than

```
$ module load example/1.2.3
```

Everything works fine, up to the point where a new version of `example` is installed, 4.5.6. From then on, the user's `load` command will load the latter version, rather than the intended one, which may lead to unexpected problems. See for example section 8.8.

Consider the following `example` modules:

```
$ module avail example/
example/1.2.3
example/4.5.6
```

Let's now generate a version conflict with the `example` module, and see what happens.

```
$ module av example/
example/1.2.3              example/4.5.6
$ module load example/1.2.3 example/4.5.6
Lmod has detected the following error:  A different version of the 'example'
module is already loaded (see output of 'ml').
$ module swap example/4.5.6
```

Note: A `module swap` command combines the appropriate `module unload` and `module load` commands.

4.1.8 Search for modules

With the `module spider` command, you can search for modules:

```
$ module spider example
-----------------------------------------------------------------------------
  example:
-----------------------------------------------------------------------------
    Description:
      This is just an example

     Versions:
        example/1.2.3
        example/4.5.6
-----------------------------------------------------------------------------
  For detailed information about a specific "example" module (including how to
  load the modules) use the module's full name.
  For example:

     module spider example/1.2.3
-----------------------------------------------------------------------------
```

It's also possible to get detailed information about a specific module:

```
$ module spider example/1.2.3
-----------------------------------------------------------------------------------------

  example: example/1.2.3
-----------------------------------------------------------------------------------------

    Description:
      This is just an example

    You will need to load all module(s) on any one of the lines below before the "
    example/1.2.3" module is available to load.

      cluster/delcatty
      cluster/golett
      cluster/phanpy
      cluster/swalot

    Help:

      Description
      ===========
      This is just an example


      More information
      ================
       - Homepage: https://example.com
```

4.1.9 Get detailed info

To get a list of all possible commands, type:

```
$ module help
```

Or to get more information about one specific module package:

```
$ module help example/1.2.3
---------- Module Specific Help for 'example/1.2.3' ---------------------------
  This is just an example - Homepage: https://example.com/
```

4.1.10 Save and load collections of modules

If you have a set of modules that you need to load often, you can save these in a *collection*. This will enable you to load all the modules you need with a single command.

In each `module` command shown below, you can replace `module` with `ml`.

First, load all modules you want to include in the collections:

```
$ module load example/1.2.3 secondexample/2.7-intel-2016b
```

Now store it in a collection using `module save`. In this example, the collection is named `my-collection`.

```
$ module save my-collection
```

Later, for example in a jobscript or a new session, you can load all these modules with `module restore`:

```
$ module restore my-collection
```

You can get a list of all your saved collections with the `module savelist` command:

```
$ module savelistr
Named collection list (For LMOD_SYSTEM_NAME = "CO7-sandybridge"):
  1) my-collection
```

To get a list of all modules a collection will load, you can use the `module describe` command:

```
$ module describe my-collection
1) example/1.2.3                                          6) imkl/11.3.3.210-iimpi
    -2016b
2) GCCcore/5.4.0                                          7) intel/2016b
3) icc/2016.3.210-GCC-5.4.0-2.26                          8) examplelib/1.2-intel
    -2016b
4) ifort/2016.3.210-GCC-5.4.0-2.26                        9) secondexample/2.7-intel
    -2016b
5) impi/5.1.3.181-iccifort-2016.3.210-GCC-5.4.0-2.26
```

To remove a collection, remove the corresponding file in `$HOME/.lmod.d`:

```
$ rm $HOME/.lmod.d/my-collection
```

4.1.11 Getting module details

To see how a module would change the environment, you can use the `module show` command:

```
$ module show Python/2.7.12-intel-2016b
whatis("Description: Python is a programming language that lets you work more
    quickly and integrate your systems  more effectively. - Homepage: http://python.
    org/ ")
conflict("Python")
load("intel/2016b")
load("bzip2/1.0.6-intel-2016b")
...
prepend_path(...)
setenv("EBEXTSLISTPYTHON","setuptools-23.1.0,pip-8.1.2,nose-1.3.7,numpy-1.11.1,scipy
    -0.17.1,ytz-2016.4", ...)
```

It's also possible to use the `ml show` command instead: they are equivalent.

Here you can see that the `Python/2.7.12-intel-2016b` comes with a whole bunch of extensions: `numpy`, `scipy`, ...

You can also see the modules the `Python/2.7.12-intel-2016b` module loads: `intel/2016b`, `bzip2/1.0.6-intel-2016b`, ...

If you're not sure what all of this means: don't worry, you don't have to know; just load the module and try to use the software.

4.2 Getting system information about the HPC infrastructure

4.2.1 Checking the general status of the HPC infrastructure

To check the general system state, check https://www.vscentrum.be/en/user-portal/system-status. This has information about scheduled downtime, status of the system, ...

To check how much jobs are running in what queues, you can use the `qstat -q` command:

```
$ qstat -q
Queue              Memory CPU Time Walltime Node  Run Que Lm  State
---------------- ------ -------- -------- ----  --- --- --  -----
default              --      --       --      --    0   0 --   E R
q72h                 --      --    72:00:00   --    0   0 --   E R
long                 --      --    72:00:00   --  316  77 --   E R
short                --      --    11:59:59   --   21   4 --   E R
q1h                  --      --    01:00:00   --    0   1 --   E R
q24h                 --      --    24:00:00   --    0   0 --   E R
                                                 ----- -----
                                                  337  82
```

Here, there are 316 jobs running on the `long` queue, and 77 jobs queued. We can also see that the `long` queue allows a maximum wall time of 72 hours.

4.2.2 Getting cluster state

You can check http://hpc.ugent.be/clusterstate to see information about the clusters: you can see the nodes that are down, free, partially filled with jobs, completely filled with jobs, ...

You can also get this information in text form (per cluster separately) with the `pbsmon` command:

```
$ module swap cluster/phanpy
$ pbsmon
2301 2302 2303 2304 2305 2306 2307
   _    .    X    J    _    _    _

 2308 2309 2310 2311 2312 2313 2314
    j    j    J    j    _    _    j

 2315 2316
    _    j

   J full                 : 2   |   X down                 : 1   |
   j partial              : 5   |   x down_on_error        : 0   |
   _ free                 : 7   |   . offline              : 1   |
                                |   o other                : 0   |

Node type:
 ppn=24, physmem=503.6GB, swap=20.0GB, vmem=523.6GB, local disk=1117.0GB
```

`pbsmon` only outputs details of the cluster corresponding to the currently loaded `cluster` module (see subsection 4.3.2).

It also shows details about the nodes in a cluster. In the example, all nodes have 24 cores and 503.6 GB of memory.

4.3 Defining and submitting your job

Usually, you will want to have your program running in batch mode, as opposed to interactively as you may be accustomed to. The point is that the program must be able to start and run without user intervention, i.e., without you having to enter any information or to press any buttons during program execution. All the necessary input or required options have to be specified on the command line, or needs to be put in input or configuration files.

As an example, we will run a perl script, which you will find in the examples subdirectory on the HPC. When you received an account to the HPC a subdirectory with examples was automatically generated for you.

Remember that you have copied the contents of the HPC examples directory to your home directory, so that you have your **own personal** copy (editable and over-writable) and that you can start using the examples. If you haven't done so already, run these commands now:

```
$ cd
$ cp -r /apps/gent/tutorials/Intro-HPC/examples ~/
```

First go to the directory with the first examples by entering the command:

```
$ cd ~/examples/Running-batch-jobs
```

Each time you want to execute a program on the HPC you'll need 2 things:

The executable The program to execute from the end-user, together with its peripheral input

files, databases and/or command options.

A batch job script , which will define the computer resource requirements of the program, the required additional software packages and which will start the actual executable. The HPC needs to know:

1. the type of compute nodes;
2. the number of CPUs;
3. the amount of memory;
4. the expected duration of the execution time (wall time: Time as measured by a clock on the wall);
5. the name of the files which will contain the output (i.e., stdout) and error (i.e., stderr) messages;
6. what executable to start, and its arguments.

Later on, the HPC user shall have to define (or to adapt) his/her own job scripts. For now, all required job scripts for the exercises are provided for you in the examples subdirectories.

List and check the contents with:

```
$ ls -l
total 512
-rw-r--r-- 1 vsc40000 193 Sep 11 10:34 fibo.pbs
-rw-r--r-- 1 vsc40000 609 Sep 11 10:25 fibo.pl
```

In this directory you find a Perl script (named "fibo.pl") and a job script (named "fibo.pbs").

1. The Perl script calculates the first 30 Fibonacci numbers.
2. The job script is actually a standard Unix/Linux shell script that contains a few extra comments at the beginning that specify directives to PBS. These comments all begin with #**PBS**.

We will first execute the program locally (i.e., on your current login-node), so that you can see what the program does.

On the command line, you would run this using:

```
$ ./fibo.pl
[0] -> 0
[1] -> 1
[2] -> 1
[3] -> 2
[4] -> 3
[5] -> 5
[6] -> 8
[7] -> 13
[8] -> 21
[9] -> 34
[10] -> 55
[11] -> 89
[12] -> 144
[13] -> 233
[14] -> 377
[15] -> 610
[16] -> 987
[17] -> 1597
[18] -> 2584
[19] -> 4181
[20] -> 6765
[21] -> 10946
[22] -> 17711
[23] -> 28657
[24] -> 46368
[25] -> 75025
[26] -> 121393
[27] -> 196418
[28] -> 317811
[29] -> 514229
```

Remark: Recall that you have now executed the Perl script locally on one of the login-nodes of the HPC cluster. Of course, this is not our final intention; we want to run the script on any of the compute nodes. Also, it is not considered as good practice, if you "abuse" the login-nodes for testing your scripts and executables. It will be explained later on how you can reserve your own compute-node (by opening an interactive session) to test your software. But for the sake of acquiring a good understanding of what is happening, you are pardoned for this example since these jobs require very little computing power.

The job script contains a description of the job by specifying the command that need to be executed on the compute node:

— fibo.pbs —

```
1 #!/bin/bash -l
2 cd $PBS_O_WORKDIR
3 ./fibo.pl
```

So, jobs are submitted as scripts (bash, Perl, Python, etc.), which specify the parameters related to the jobs such as expected runtime (walltime), e-mail notification, etc. These parameters can also be specified on the command line.

This job script that can now be submitted to the cluster's job system for execution, using the qsub (Queue SUBmit) command:

```
$ qsub fibo.pbs
123456.master15.delcatty.gent.vsc
```

The qsub command returns a job identifier on the HPC cluster. The important part is the number (e.g., "123456"); this is a unique identifier for the job and can be used to monitor and manage your job.

Your job is now waiting in the queue for a free workernode to start on.

Go and drink some coffee ... but not too long. If you get impatient you can start reading the next section for more information on how to monitor jobs in the queue.

After your job was started, and ended, check the contents of the directory:

```
$ ls -l
total 768
-rw-r--r-- 1 vsc40000 vsc40000   44 Feb 28 13:33 fibo.pbs
-rw------- 1 vsc40000 vsc40000    0 Feb 28 13:33 fibo.pbs.e123456
-rw------- 1 vsc40000 vsc40000 1010 Feb 28 13:33 fibo.pbs.o123456
-rwxrwxr-x 1 vsc40000 vsc40000  302 Feb 28 13:32 fibo.pl
```

Explore the contents of the 2 new files:

```
$ more fibo.pbs.o123456
$ more fibo.pbs.e123456
```

These files are used to store the standard output and error that would otherwise be shown in the terminal window. By default, they have the same name as that of the PBS script, i.e., "fibo.pbs" as base name, followed by the extension ".o" (output) and ".e" (error), respectively, and the job number ('123456' for this example). The error file will be empty, at least if all went well. If not, it may contain valuable information to determine and remedy the problem that prevented a successful run. The standard output file will contain the results of your calculation (here, the output of the perl script)

4.3.1 When will my job start?

In practice it's impossible to predict when your job(s) will start, since most currently running jobs will finish before their requested walltime expires, and new jobs by may be submitted by other users that are assigned a higher priority than your job(s).

The UGent-HPC clusters use a fair-share sheduling policy (see chapter 9). There is no guarantee on when a job will start, since it depends on a number of factors. One of these factors is the priority of the job, which is determined by

- historical use: the aim is to balance usage over users, so infrequent (in terms of total compute time used) users get a highter priority
- requested resources (amount of cores, walltime, memory, ...)
- time waiting in queue: queued jobs get a higher priority over time
- user limits: this avoids having a single user use the entire cluster. This means that each user can only use a part of the cluster.

Some other factors are how busy the cluster is, how many workernodes are active, the resources (e.g. number of cores, memory) provided by each workernode, ...

It might be beneficial to request less resources (e.g. not requesting all cores in a workernode), since the scheduler often finds a "gap" to fit the job into more easily.

4.3.2 Specifying the cluster on which to run

To use other clusters, you can swap the `cluster` module. This is a special module that change what modules are available for you, and what cluster your jobs will be queued in.

By default you are working on delcatty. To switch to, e.g., swalot you need to redefine the environment so you get access to all modules installed on the swalot cluster, and to be able to submit jobs to the swalot scheduler so your jobs will start on swalot instead of the default delcatty cluster.

```
$ module swap cluster/swalot
```

Note: the swalot modules may not work directly on the login nodes, because the login nodes do not have the same architecture as the swalot cluster, they have the same architecture as the delcatty cluster however, so this is why by default software works on the login nodes. See section 8.9 for why this is and how to fix this.

```
$ module avail cluster/
------------------------------------------------------------------------------
     /etc/modulefiles/vsc
    ---------------------------------------------------------------------------

   cluster/delcatty (S,L)    cluster/golett (S)    cluster/phanpy (S)    cluster/
    skitty (S)    cluster/swalot (S)    cluster/victini (S)

  Where:
   S:  Module is Sticky, requires --force to unload or purge
   L:  Module is loaded

If you need software that is not listed, request it via https://www.ugent.be/hpc/en/
    support/software-installation-request
```

As indicated in the output above, each `cluster` module is a so-called sticky module, i.e. it will not be unloaded when `module purge` (see subsection 4.1.6) is used.

The output of the various commands interacting with jobs (`qsub`, `stat`, ...) all depend on which `cluster` module is loaded.

4.4 Monitoring and managing your job(s)

Using the job ID that `qsub` returned, there are various ways to monitor the status of your job. In the following commands, replace `12345` with the job ID `qsub` returned.

```
$ qstat 12345
```

To show on which compute nodes your job is running, at least, when it is running:

```
$ qstat -n 12345
```

To remove a job from the queue so that it will not run, or to stop a job that is already running.

```
$ qdel 12345
```

When you have submitted several jobs (or you just forgot about the job ID), you can retrieve the status of all your jobs that are submitted and are not yet finished using:

```
$ qstat
master15.delcatty.gent.vsc :
Job ID      Name    User      Time Use S Queue
----------- ------- --------- -------- - -----
123456.... mpi     vsc40000 0        Q short
```

Here:

Job ID the job's unique identifier

Name the name of the job

User the user that owns the job

Time Use the elapsed walltime for the job

Queue the queue the job is in

The state S can be any of the following:

State	Meaning
Q	The job is **queued** and is waiting to start.
R	The job is currently **running**.
E	The job is currently **exiting** after having run.
C	The job is **completed** after having run.
H	The job has a user or system **hold** on it and will not be eligible to run until the hold is removed.

User hold means that the user can remove the hold. System hold means that the system or an administrator has put the job on hold, very likely because something is wrong with it. Check with your helpdesk to see why this is the case.

4.5 Examining the queue

As we learned above, Moab is the software application that actually decides when to run your job and what resources your job will run on. For security reasons, it is not possible to see what other users are doing on the clusters. As such, the PBS **qstat** command only gives information about your own jobs that are queued or running, ordered by **JobID**.

However, you can get some idea of the load on the clusters by specifying the **-q** option to the **qstat** command:

```
$ qstat -q
server: master15.delcatty.gent.vsc

Queue            Memory CPU Time Walltime Node  Run Que Lm  State
---------------- ------ -------- -------- ----  --- --- --  -----
short              --      --    11:59:59   --    2  24 --   E R
default            --      --       --      --    0   0 --   E R
debug              --      --    00:59:59   --    1   0 --   E R
long               --      --    72:00:00   --  124 453 --   E R
                                               ----- -----
                                                 127   477
```

In this example, 477 jobs are queued in the various queues whereas 127 jobs are effectively running.

4.6 Specifying job requirements

Without giving more information about your job upon submitting it with **qsub**, default values will be assumed that are almost never appropriate for real jobs.

It is important to estimate the resources you need to successfully run your program, such as the amount of time the job will require, the amount of memory it needs, the number of CPUs it will run on, etc. This may take some work, but it is necessary to ensure your jobs will run properly.

4.6.1 Generic resource requirements

The **qsub** command takes several options to specify the requirements, of which we list the most commonly used ones below.

```
$ qsub -l walltime=2:30:00
```

For the simplest cases, only the amount of maximum estimated execution time (called "walltime") is really important. Here, the job requests 2 hours, 30 minutes. As soon as the job exceeds the requested walltime, it will be "killed" (terminated) by the job scheduler. There is no harm if you *slightly* overestimate the maximum execution time. If you omit this option, the queue manager will not complain but use a default value (one hour on most clusters).

If you want to run some final steps (for example to copy files back) before the walltime kills your main process, you have to kill the main command yourself before the walltime runs out and then copy the file back. See section 16.3 for how to do this.

```
$ qsub -l mem=4gb
```

The job requests 4 GB of RAM memory. As soon as the job tries to use more memory, it will be "killed" (terminated) by the job scheduler. There is no harm if you *slightly* overestimate the requested memory.

```
$ qsub -l nodes=5:ppn=2
```

The job requests 5 compute nodes with two cores on each node (ppn stands for "processors per node", where "processors" here actually means "CPU cores").

```
$ qsub -l nodes=1:westmere
```

The job requests just one node, but it should have an Intel Westmere processor. A list with site-specific properties can be found in the next section or in the User Portal ("Available hardware"-section)[1] of the VSC website.

These options can either be specified on the command line, e.g.

```
$ qsub -l nodes=1:ppn=1,mem=2gb fibo.pbs
```

or in the job script itself using the #PBS-directive, so "fibo.pbs" could be modified to:

```
1 #!/bin/bash -l
2 #PBS -l nodes=1:ppn=1
3 #PBS -l mem=2gb
4 cd $PBS_O_WORKDIR
5 ./fibo.pl
```

Note that the resources requested on the command line will override those specified in the PBS file.

4.6.2 Available job categories (TORQUE queues)

In order to guarantee a fair share access to the computer resources to all users, only a limited number of jobs with certain walltimes are possible per user.

We therefore classify the submitted jobs in categories (confusingly also called queues), depending on the their walltime specification. A user is allowed to run up to a certain maximum number of jobs in each of these walltime categories.

The currently defined walltime categories for the HPC are:

Queue category	**Walltime**		**Max # Jobs**	
	Minimum / from (value not included)	**Maximum / to (value included)**	**Queuable**	**Runnable**
short	0	1 hour		
long	0	72 hours		
bshort	0	1 hour		
debug	0	15 minutes		

[1] URL: https://www.vscentrum.be/infrastructure/hardware

4.6.3 Node-specific properties

The following table contains some node-specific properties that can be used to make sure the job will run on nodes with a specific CPU or interconnect. Note that these properties may vary over the different VSC sites.

To get a list of all properties defined for all nodes, enter

```
$ pbsnodes
```

This list will also contain properties referring to, e.g., network components, rack number, etc.

4.7 Job output and error files

At some point your job finishes, so you may no longer see the job ID in the list of jobs when you run *qstat* (since it will only be listed for a few minutes after completion with state "C"). After your job finishes, you should see the standard output and error of your job in two files, located by default in the directory where you issued the *qsub* command.

When you navigate to that directory and list its contents, you should see them:

```
$ ls -l
total 1024
-rw-r--r-- 1 vsc40000  609 Sep 11 10:54 fibo.pl
-rw-r--r-- 1 vsc40000   68 Sep 11 10:53 fibo.pbs
-rw------- 1 vsc40000   52 Sep 11 11:03 fibo.pbs.e123456
-rw------- 1 vsc40000 1307 Sep 11 11:03 fibo.pbs.o123456
```

In our case, our job has created both output ('fibo.pbs.**o**123456') and error files ('fibo.pbs.**e**123456') containing info written to *stdout* and *stderr* respectively.

Inspect the generated output and error files:

```
$ cat fibo.pbs.o123456
...
$ cat fibo.pbs.e123456
...
```

4.8 E-mail notifications

4.8.1 Generate your own e-mail notifications

You can instruct the HPC to send an e-mail to your e-mail address whenever a job **b**egins, **e**nds and/or **a**borts, by adding the following lines to the job script `fibo.pbs`:

```
1 #PBS -m b
2 #PBS -m e
3 #PBS -m a
```

or

```
1 #PBS -m abe
```

These options can also be specified on the command line. Try it and see what happens:

```
$ qsub -m abe fibo.pbs
```

The system will use the e-mail address that is connected to your VSC account. You can also specify an alternate e-mail address with the `-M` option:

```
$ qsub -m b -M john.smith@example.com fibo.pbs
```

will send an e-mail to john.smith@example.com when the job begins.

4.9 Running a job after another job

If you submit two jobs expecting that should be run one after another (for example because the first generates a file the second needs), there might be a problem as they might both be run at the same time.

So the following example might go wrong:

```
$ qsub job1.sh
$ qsub job2.sh
```

You can make jobs that depend on other jobs. This can be useful for breaking up large jobs into smaller jobs that can be run in a pipeline. The following example will submit 2 jobs, but the second job (`job2.sh`) will be held (`H` status in `qstat`) until the first job successfully completes. If the first job fails, the second will be cancelled.

```
$ FIRST_ID=$(qsub job1.sh)
$ qsub -W depend=afterok:$FIRST_ID job2.sh
```

`afterok` means "After OK", or in other words, after the first job successfully completed.

It's also possible to use `afternotok` ("After not OK") to run the second job only if the first job exited with errors. A third option is to use `afterany` ("After any"), to run the second job after the first job (regardless of success or failure).

Chapter 5

Running interactive jobs

5.1 Introduction

Interactive jobs are jobs which give you an interactive session on one of the compute nodes. Importantly, accessing the compute nodes this way means that the job control system guarantees the resources that you have asked for.

Interactive PBS jobs are similar to non-interactive PBS jobs in that they are submitted to PBS via the command **qsub**. Where an interactive job differs is that it does not require a job script, the required PBS directives can be specified on the command line.

Interactive jobs can be useful to debug certain job scripts or programs, but should not be the main use of the UGent-HPC. Waiting for user input takes a very long time in the life of a CPU and does not make efficient usage of the computing resources.

The syntax for *qsub* for submitting an interactive PBS job is:

```
$ qsub -I <...  pbs directives ...>
```

5.2 Interactive jobs, without X support

Tip: Find the code in "~/examples/Running-interactive-jobs"

First of all, in order to know on which computer you're working, enter:

```
$ hostname -f
gligar04.gastly.os
```

This means that you're now working on the login node `gligar04.gastly.os` of the HPC cluster.

The most basic way to start an interactive job is the following:

```
$ qsub -I
qsub: waiting for job 123456.master15.delcatty.gent.vsc to start
qsub: job 123456.master15.delcatty.gent.vsc ready
```

There are two things of note here.

1. The "*qsub*" command (with the interactive -I flag) waits until a node is assigned to your interactive session, connects to the compute node and shows you the terminal prompt on that node.
2. You'll see that your directory structure of your home directory has remained the same. Your home directory is actually located on a shared storage system. This means that the exact same directory is available on all login nodes and all compute nodes on all clusters.

In order to know on which compute-node you're working, enter again:

```
$ hostname -f
node2001.delcatty.gent.vsc
```

Note that we are now working on the compute-node called "*node2001.delcatty.gent.vsc*". This is the compute node, which was assigned to us by the scheduler after issuing the "*qsub -I*" command.

Now, go to the directory of our second interactive example and run the program "primes.py". This program will ask you for an upper limit (> 1) and will print all the primes between 1 and your upper limit:

```
$ cd ~/examples/Running-interactive-jobs
$ ./primes.py
This program calculates all primes between 1 and your upper limit.
Enter your upper limit (>1): 50
Start Time:  2013-09-11 15:49:06
[Prime#1] = 1
[Prime#2] = 2
[Prime#3] = 3
[Prime#4] = 5
[Prime#5] = 7
[Prime#6] = 11
[Prime#7] = 13
[Prime#8] = 17
[Prime#9] = 19
[Prime#10] = 23
[Prime#11] = 29
[Prime#12] = 31
[Prime#13] = 37
[Prime#14] = 41
[Prime#15] = 43
[Prime#16] = 47
End Time:  2013-09-11 15:49:06
Duration:  0 seconds.
```

You can exit the interactive session with:

```
$ exit
```

Note that you can now use this allocated node for 1 hour. After this hour you will be automatically disconnected. You can change this "usage time" by explicitly specifying a "walltime", i.e., the time that you want to work on this node. (Think of walltime as the time elapsed when watching the clock on the wall.)

You can work for 3 hours by:

```
$ qsub -I -l walltime=03:00:00
```

If the walltime of the job is exceeded, the (interactive) job will be killed and your connection to the compute node will be closed. So do make sure to provide adequate walltime and that you save your data before your (wall)time is up (exceeded)! When you do not specify a walltime, you get a default walltime of 1 hour.

5.3 Interactive jobs, with graphical support

5.3.1 Software Installation

To display graphical applications from a Linux computer (such as the VSC clusters) on your machine, you need to install an X Window server on your local computer.

The X Window system (commonly known as **X11**, based on its current major version being 11, or shortened to simply **X**) is the system-level software infrastructure for the windowing GUI on Linux, BSD and other UNIX-like operating systems. It was designed to handle both local displays, as well as displays sent across a network. More formally, it is a computer software system and network protocol that provides a basis for graphical user interfaces (GUIs) and rich input device capability for networked computers.

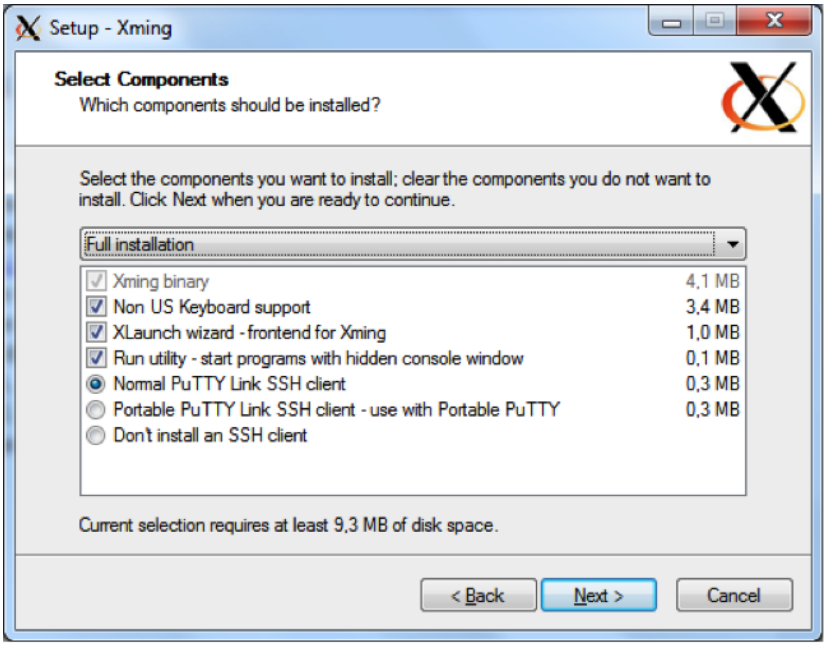
Install Xming The first task is to install the Xming software.

1. Download the Xming installer from the following address: http://www.straightrunning.com/XmingNotes/. Either download Xming from the **Public Domain Releases** (free) or from the **Website Releases** (after a donation) on the website.

2. Run the Xming setup program on your Windows desktop.

3. Keep the proposed default folders for the Xming installation.

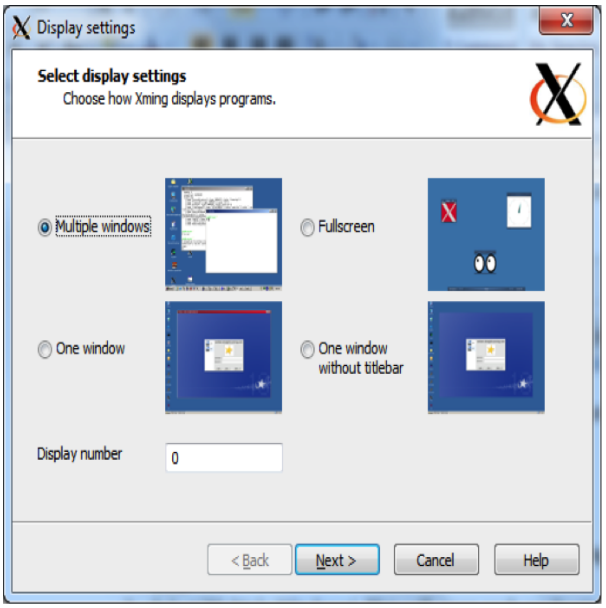
4. When selecting the components that need to be installed, make sure to select "*XLaunch wizard*" and "*Normal PuTTY Link SSH client*".



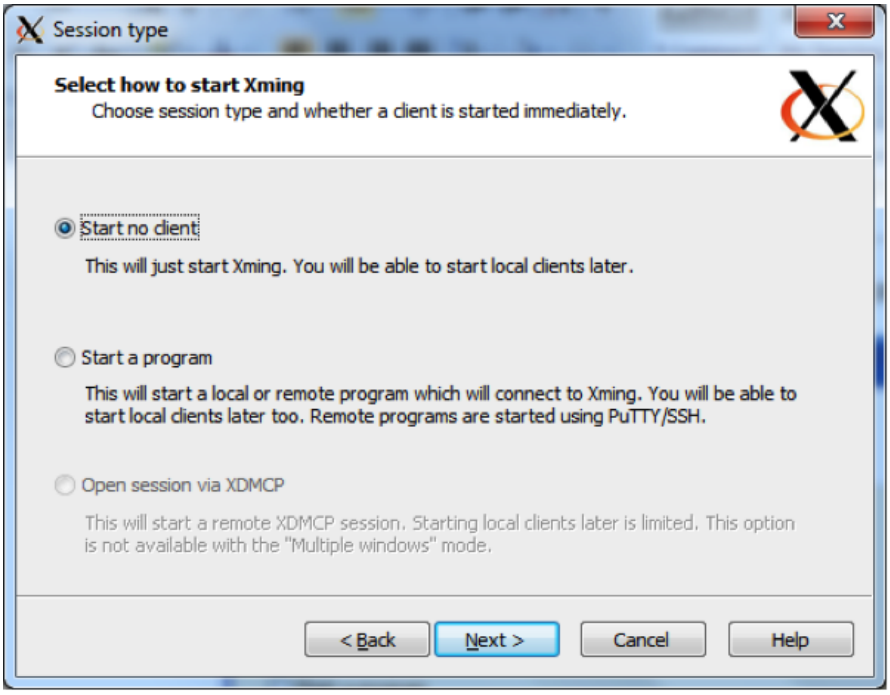
5. We suggest to create a Desktop icon for Xming and XLaunch.
6. And Install.

And now we can run Xming:

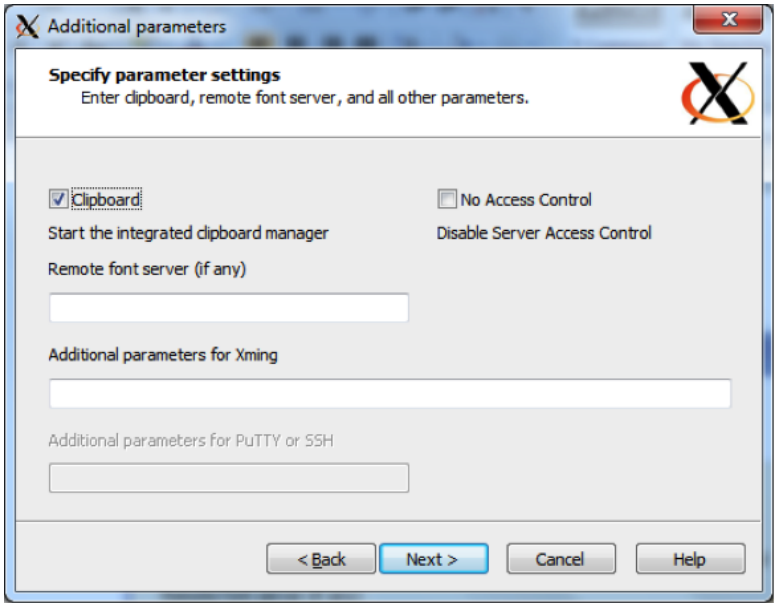
1. Select XLaunch from the Start Menu or by double-clicking the Desktop icon.
2. Select Multiple Windows. This will open each application in a separate window.



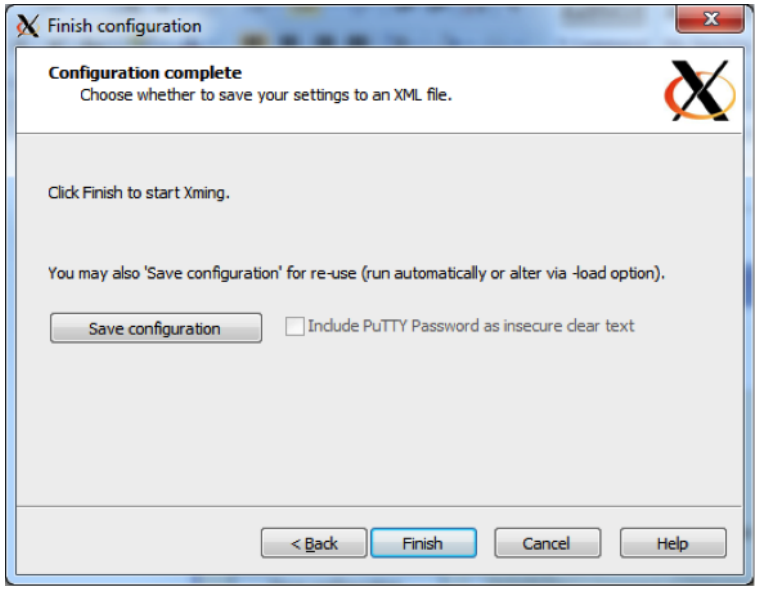
3. Select Start no client to make XLaunch wait for other programs (such as PuTTY).



4. Select Clipboard to share the clipboard.



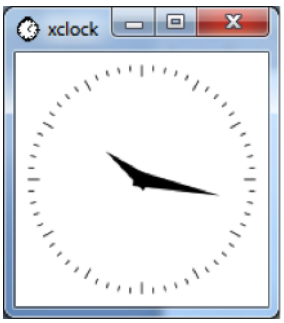
5. Finally Save configuration into a file. You can keep the default filename and save it in your Xming installation directory.



6. Now Xming is running in the background ...
   and you can launch a graphical application in your PuTTY terminal.
7. Open a PuTTY terminal and connect to the HPC.
8. In order to test the X-server, run "*xclock*". "*xclock*" is the standard GUI clock for the X Window System.

```
$ xclock
```

You should see the XWindow clock application appearing on your Windows machine. The "*xclock*" application runs on the login-node of the HPC, but is displayed on your Windows machine.



You can close your clock and connect further to a compute node with again your X-forwarding enabled:

```
$ qsub -I -X
qsub: waiting for job 123456.master15.delcatty.gent.vsc to start
qsub: job 123456.master15.delcatty.gent.vsc ready
$ hostname -f
node2001.delcatty.gent.vsc
$ xclock
```

and you should see your clock again.

SSH Tunnel In order to work in client/server mode, it is often required to establish an SSH tunnel between your Windows desktop machine and the compute node your job is running on. PuTTY must have been installed on your computer, and you should be able to connect via SSH to the HPC cluster's login node.

Because of one or more firewalls between your desktop and the HPC clusters, it is generally impossible to communicate directly with a process on the cluster from your desktop except when the network managers have given you explicit permission (which for security reasons is not often done). One way to work around this limitation is SSH tunnelling.

There are several cases where this is useful:

1. Running graphical applications on the cluster: The graphical program cannot directly communicate with the X Window server on your local system. In this case, the tunnelling is easy to set up as PuTTY will do it for you if you select the right options on the X11 settings page as explained on the page about text-mode access using PuTTY.

2. Running a server application on the cluster that a client on the desktop connects to. One example of this scenario is ParaView in remote visualisation mode, with the interactive client on the desktop and the data processing and image rendering on the cluster. This scenario is explained on this page.

3. Running clients on the cluster and a server on your desktop. In this case, the source port is a port on the cluster and the destination port is on the desktop.

Procedure: A tunnel from a local client to a specific computer node on the cluster

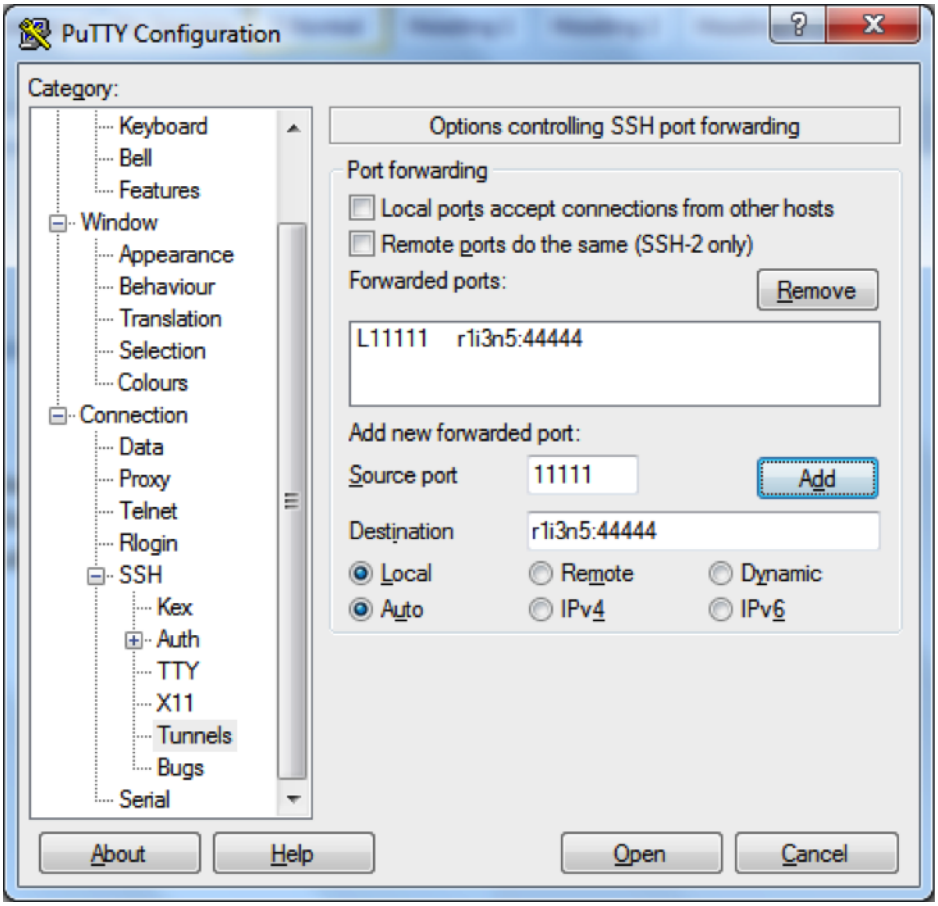
1. Log in on the login node via PuTTY.

2. Start the server job, note the compute node's name the job is running on (e.g., node2001.delcatty.gent.vsc), as well as the port the server is listening on (e.g., "54321").

3. Set up the tunnel:

   (a) Close your current PuTTY session.

   (b) In the "*Category*" pane, expand Connection » SSH, and select Tunnels as show below:



(c) In the Source port field, enter the local port to use (e.g., *5555*).

(d) In the Destination field, enter *<hostname>:<server-port>* (e.g., node2001.delcatty.gent.vsc:54321 as in the example above, these are the details you noted in the second step).

(e) Click the Add button.

(f) Click the Open button

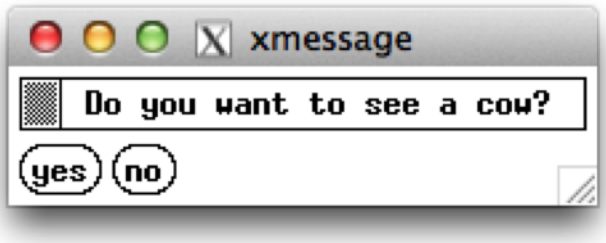
The tunnel is now ready to use.

5.3.2 Run simple example

We have developed a little interactive program that shows the communication in 2 directions. It will send information to your local screen, but also asks you to click a button.

```
$ cd ~/examples/Running-interactive-jobs
$ ./message.py
```

You should see the following message appearing.



Click any button and see what happens.

```
-----------------------
< Enjoy the day! Mooh >
-----------------------
     ^__^
     (oo)\_______
     (__)\       )\/\
         ||----w |
         ||     ||
```

Chapter 6

Running jobs with input/output data

You have now learned how to start a batch job and how to start an interactive session. The next question is how to deal with input and output files, where your standard output and error messages will go to and where that you can collect your results.

6.1 The current directory and output and error files

6.1.1 Default file names

First go to the directory:

```
$ cd ~/examples/Running-jobs-with-input-output-data
```

List and check the contents with:

```
$ ls -l
total 2304
-rwxrwxr-x 1 vsc40000  682 Sep 13 11:34 file1.py
-rw-rw-r-- 1 vsc40000  212 Sep 13 11:54 file1a.pbs
-rw-rw-r-- 1 vsc40000  994 Sep 13 11:53 file1b.pbs
-rw-rw-r-- 1 vsc40000  994 Sep 13 11:53 file1c.pbs
-rw-r--r-- 1 vsc40000 1393 Sep 13 10:41 file2.pbs
-rwxrwxr-x 1 vsc40000 2393 Sep 13 10:40 file2.py
-rw-r--r-- 1 vsc40000 1393 Sep 13 10:41 file3.pbs
-rwxrwxr-x 1 vsc40000 2393 Sep 13 10:40 file3.py
```

Now, let us inspect the contents of the first executable (which is just a Python script with execute permission).

— file1.py —

```
#!/usr/bin/env python
#
# VSC        : Flemish Supercomputing Centre
# Tutorial   : Introduction to HPC
# Description: Writing to the current directory, stdout and stderr
#
import sys

# Step #1: write to a local file in your current directory
local_f = open("Hello.txt", 'w+')
local_f.write("Hello World!\n")
local_f.write("I am writing in the file:<Hello.txt>.\n")
local_f.write("in the current directory.\n")
local_f.write("Cheers!\n")
local_f.close()

# Step #2: Write to stdout
sys.stdout.write("Hello World!\n")
sys.stdout.write("I am writing to <stdout>.\n")
sys.stdout.write("Cheers!\n")

# Step #3: Write to stderr
sys.stderr.write("Hello World!\n")
sys.stderr.write("This is NO ERROR or WARNING.\n")
sys.stderr.write("I am just writing to <stderr>.\n")
sys.stderr.write("Cheers!\n")
```

The code of the Python script, is self explanatory:

1. In step 1, we write something to the file `hello.txt` in the current directory.
2. In step 2, we write some text to stdout.
3. In step 3, we write to stderr.

Check the contents of the first job script:

— file1a.pbs —

```
#!/bin/bash

#PBS -l walltime=00:05:00

# go to the (current) working directory (optional, if this is the
# directory where you submitted the job)
cd $PBS_O_WORKDIR

# the program itself
echo Start Job
date
./file1.py
echo End Job
```

You'll see that there are NO specific PBS directives for the placement of the output files. All output files are just written to the standard paths.

Submit it:

```
$ qsub file1a.pbs
```

After the job has finished, inspect the local directory again, i.e., the directory where you executed the *qsub* command:

```
$ ls -l
total 3072
-rw-rw-r-- 1 vsc40000   90 Sep 13 13:13 Hello.txt
-rwxrwxr-x 1 vsc40000  693 Sep 13 13:03 file1.py*
-rw-rw-r-- 1 vsc40000  229 Sep 13 13:01 file1a.pbs
-rw------- 1 vsc40000   91 Sep 13 13:13 file1a.pbs.e123456
-rw------- 1 vsc40000  105 Sep 13 13:13 file1a.pbs.o123456
-rw-rw-r-- 1 vsc40000  143 Sep 13 13:07 file1b.pbs
-rw-rw-r-- 1 vsc40000  177 Sep 13 13:06 file1c.pbs
-rw-r--r-- 1 vsc40000 1393 Sep 13 10:41 file2.pbs
-rwxrwxr-x 1 vsc40000 2393 Sep 13 10:40 file2.py*
-rw-r--r-- 1 vsc40000 1393 Sep 13 10:41 file3.pbs
-rwxrwxr-x 1 vsc40000 2393 Sep 13 10:40 file3.py*
```

Some observations:

1. The file `Hello.txt` was created in the current directory.

2. The file `file1a.pbs.o123456` contains all the text that was written to the standard output stream ("stdout").

3. The file `file1a.pbs.e123456` contains all the text that was written to the standard error stream ("stderr").

Inspect their contents . . . and remove the files

```
$ cat Hello.txt
$ cat file1a.pbs.o123456
$ cat file1a.pbs.e123456
$ rm Hello.txt file1a.pbs.o123456 file1a.pbs.e123456
```

Tip: Type `cat H` and press the Tab button (looks like ⇥), and it will **expand** into `cat Hello.txt`.

6.1.2 Filenames using the name of the job

Check the contents of the job script and execute it.

— file1b.pbs —

```
#!/bin/bash

#   Specify the "name" of the job
#PBS -N my_serial_job

cd $PBS_O_WORKDIR
echo Start Job
date
./file1.py
echo End Job
```

```
$ qsub file1b.pbs
```

Inspect the contents again ... and remove the generated files:

```
$ ls
Hello.txt  file1a.pbs  file1c.pbs  file2.pbs  file3.pbs  my_serial_job.e123456
file1.py*  file1b.pbs  file2.py*   file3.py*  my_serial_job.o123456
$ rm Hello.txt my_serial_job.*
```

Here, the option "`-N`" was used to explicitly assign a name to the job. This overwrote the JOBNAME variable, and resulted in a different name for the *stdout* and *stderr* files. This name is also shown in the second column of the "`qstat`" command. If no name is provided, it defaults to the name of the job script.

6.1.3 User-defined file names

You can also specify the name of *stdout* and *stderr* files explicitly by adding two lines in the job script, as in our third example:

— file1c.pbs —

```
#!/bin/bash

# redirect standard output (-o) and error (-e)
#PBS -o stdout.$PBS_JOBID
#PBS -e stderr.$PBS_JOBID

cd $PBS_O_WORKDIR
echo Start Job
date
./file1.py
echo End Job
```

```
$ qsub file1c.pbs
$ ls
```

6.2 Where to store your data on the HPC

The HPC cluster offers their users several locations to store their data. Most of the data will reside on the shared storage system, but all compute nodes also have their own (small) local disk.

6.2.1 Pre-defined user directories

Three different pre-defined user directories are available, where each directory has been created for different purposes. The best place to store your data depends on the purpose, but also the size and type of usage of the data.

The following locations are available:

Variable	Description
Long-term storage slow filesystem, intended for smaller files	
$VSC_HOME	For your configuration files and other small files, see §6.2.2. The default directory is `/user/gent/xxx/vsc40000`. The same file system is accessible from all sites, i.e., you'll see the same contents in $VSC_HOME on all sites.
$VSC_DATA	A bigger "workspace", for **datasets**, results, logfiles, etc. see §6.2.3. The default directory is `/data/gent/xxx/vsc40000`. The same file system is accessible from all sites.
Fast temporary storage	
$VSC_SCRATCH_NODE	For **temporary** or transient data on the local compute node, where fast access is important; see §6.2.4. This space is available per node. The default directory is `/tmp`. On different nodes, you'll see different content.
$VSC_SCRATCH	For **temporary** or transient data that has to be accessible from all nodes of a cluster (including the login nodes) The default directory is `/scratch/gent/xxx/vsc40000`. This directory is cluster- or site-specific: On different sites, and sometimes on different clusters on the same site, you'll get a different directory with different content.
$VSC_SCRATCH_SITE	Currently the same as $VSC_SCRATCH, but could be used for a scratch space shared across all clusters at a site in the future. See §6.2.4.
$VSC_SCRATCH_GLOBAL	Currently the same as $VSC_SCRATCH, but could be used for a scratch space shared across all clusters of the VSC in the future. See §6.2.4.
$VSC_SCRATCH_CLUSTER	The scratch filesystem closest to this cluster.
$VSC_SCRATCH_PHANPY	A separate (smaller) shared scratch filesystem, powered by SSDs. This scratch filesystem is intended for very I/O-intensive workloads.

Since these directories are not necessarily mounted on the same locations over all sites, you should always (try to) use the environment variables that have been created.

We ellaborate more on the specific function of these locations in the following sections.

Note: `$VSC_SCRATCH_KYUKON` and `$VSC_SCRATCH` are the same directories ("kyukon" is the name of the storage cluster where the default shared scratch filesystem is hosted).

For documentation about VO directories, see subsection 6.7.5.

6.2.2 Your home directory ($VSC_HOME)

Your home directory is where you arrive by default when you login to the cluster. Your shell refers to it as "~" (tilde), and its absolute path is also stored in the environment variable $VSC_HOME. Your home directory is shared across all clusters of the VSC.

The data stored here should be relatively small (e.g., no files or directories larger than a few megabytes), and preferably should only contain configuration files. Note that various kinds of configuration files are also stored here, e.g., by MATLAB, Eclipse, ...

The operating system also creates a few files and folders here to manage your account. Examples are:

File or Directory	Description
.ssh/	This directory contains some files necessary for you to login to the cluster and to submit jobs on the cluster. Do not remove them, and do not alter anything if you don't know what you are doing!
.bash_profile	When you login (type username and password) remotely via ssh, .bash_profile is executed to configure your shell before the initial command prompt.
.bashrc	This script is executed every time you start a session on the cluster: when you login to the cluster and when a job starts.
.bash_history	This file contains the commands you typed at your shell prompt, in case you need them again.

6.2.3 Your data directory ($VSC_DATA)

In this directory you can store all other data that you need for longer terms (such as the results of previous jobs, ...). It is a good place for, e.g., storing big files like genome data.

The environment variable pointing to this directory is $VSC_DATA. This volume is shared across all clusters of the VSC. There are however no guarantees about the speed you will achieve on this volume. For guaranteed fast performance and very heavy I/O, you should use the scratch space instead. If you are running out of quota on your $VSC_DATA filesystem you can request a VO. See section 6.7 on how to do this.

6.2.4 Your scratch space ($VSC_SCRATCH)

To enable quick writing from your job, a few extra file systems are available on the compute nodes. These extra file systems are called scratch folders, and can be used for storage of temporary and/or transient data (temporary results, anything you just need during your job, or your batch

of jobs).

You should remove any data from these systems after your processing them has finished. There are no guarantees about the time your data will be stored on this system, and we plan to clean these automatically on a regular base. The maximum allowed age of files on these scratch file systems depends on the type of scratch, and can be anywhere between a day and a few weeks. We don't guarantee that these policies remain forever, and may change them if this seems necessary for the healthy operation of the cluster.

Each type of scratch has its own use:

Node scratch ($VSC_SCRATCH_NODE). Every node has its own scratch space, which is completely separated from the other nodes. On some clusters, it will be on a local disk in the node, while on other clusters it will be emulated through another file server. In many cases, it will be significantly slower than the cluster scratch as it typically consists of just a single disk. Some **drawbacks** are that the storage can only be accessed on that particular node and that the capacity is often very limited (e.g., 100 GB). The performance will depend a lot on the particular implementation in the cluster. In many cases, it will be significantly slower than the cluster scratch as it typically consists of just a single disk. However, if that disk is local to the node (as on most clusters), the performance will not depend on what others are doing on the cluster.

Cluster scratch ($VSC_SCRATCH). To allow a job running on multiple nodes (or multiple jobs running on separate nodes) to share data as files, every node of the cluster (including the login nodes) has access to this shared scratch directory. Just like the home and data directories, every user has its own scratch directory. Because this scratch is also available from the login nodes, you could manually copy results to your data directory after your job has ended. Also, this type of scratch is usually implemented by running tens or hundreds of disks in parallel on a powerful file server with fast connection to all the cluster nodes and therefore is often the fastest file system available on a cluster.
You may not get the same file system on different clusters, i.e., you may see different content on different clusters at the same intitute.

Site scratch ($VSC_SCRATCH_SITE). At the time of writing, the site scratch is just the same volume as the cluster scratch, and thus contains the same data. In the future it may point to a different scratch file system that is available across all clusters at a particular site, which is in fact the case for the cluster scratch on some sites.

Global scratch ($VSC_SCRATCH_GLOBAL). At the time of writing, the global scratch is just the same volume as the cluster scratch, and thus contains the same data. In the future it may point to a scratch file system that is available across all clusters of the VSC, but at the moment of writing there are no plans to provide this.

6.2.5 Pre-defined quotas

Quota is enabled on these directories, which means that the amount of data you can store there is limited. This holds for both the total size of all files as well as the total number of files that can be stored. The system works with a soft quota and a hard quota. You can temporarily exceed the soft quota, but you can never exceed the hard quota. The user will get warnings as soon as he exceeds the soft quota.

To see your a list of your current quota, visit the VSC accountpage: https://account.vscentrum.be. VO moderators can see a list of VO quota usage per member of their VO via https://account.vscentrum.be/django/vo/.

The rules are:

1. You will only receive a warning when you have reached the soft limit of either quota.
2. You *will* start losing data and get I/O errors when you reach the hard limit. In this case, data loss will occur since nothing can be written anymore (this holds both for new files as well as for existing files), until you free up some space by removing some files. Also note that you *will not* be warned when data loss occurs, so keep an eye open for the general quota warnings!
3. The same holds for running jobs that need to write files: when you reach your hard quota, jobs will crash.

We do realise that quota are often observed as a nuisance by users, especially if you're running low on it. However, it is an essential feature of a shared infrastructure. Quota ensure that a single user cannot accidentally take a cluster down (and break other user's jobs) by filling up the available disk space. And they help to guarantee a fair use of all available resources for all users. Quota also help to ensure that each folder is used for its intended purpose.

6.3 Writing Output files

Tip: Find the code of the exercises in "`~/examples/Running-jobs-with-input-output-data`"

In the next exercise, you will generate a file in the $VSC_SCRATCH directory. In order to generate some CPU- and disk-I/O load, we will

1. take a random integer between 1 and 2000 and calculate all primes up to that limit;
2. repeat this action 30.000 times;
3. write the output to the "`primes_1.txt`" output file in the $VSC_SCRATCH-directory.

Check the Python and the PBS file, and submit the job: Remember that this is already a more serious (disk-I/O and computational intensive) job, which takes approximately 3 minutes on the HPC.

```
$ cat file2.py
$ cat file2.pbs
$ qsub file2.pbs
$ qstat
$ ls -l
$ echo $VSC_SCRATCH
$ ls -l $VSC_SCRATCH
$ more $VSC_SCRATCH/primes_1.txt
```

6.4 Reading Input files

Tip: Find the code of the exercise "`file3.py`" in

"`~/examples/Running-jobs-with-input-output-data`".

In this exercise, you will

1. Generate the file "`primes_1.txt`" again as in the previous exercise;
2. open the this file;
3. read it line by line;
4. calculate the average of primes in the line;
5. count the number of primes found per line;
6. write it to the "`primes_2.txt`" output file in the $VSC_SCRATCH-directory.

Check the Python and the PBS file, and submit the job:

```
$ cat file3.py
$ cat file3.pbs
$ qsub file3.pbs
$ qstat
$ ls -l
$ more $VSC_SCRATCH/primes_2.txt
...
```

6.5 How much disk space do I get?

6.5.1 Quota

The available disk space on the HPC is limited. The actual disk capacity, shared by all users, can be found on the "Available hardware" page on the website. (https://www.vscentrum.be/infrastructure/hardware) As explained in §6.2.5, this implies that there are also limits

1. to the amount of disk space; and
2. to the number of files

that can be made available to each individual HPC user.

The quota of disk space and number of files for each HPC user is:

Volume	Max. disk space	Max. # Files
HOME	3 GB	20000
DATA	25 GB	100000
SCRATCH	25 GB	100000

Tip: The first action to take when you have exceeded your quota is to clean up your directories. You could start by removing intermediate, temporary or log files. Keeping your environment clean will never do any harm.

Tip: Users can request for additional quota, which can be granted in duly justified cases. Please contact the UGent HPC team staff.

6.5.2 Check your quota

The "`show_quota`" command has been developed to show you the status of your quota in a readable format:

```
$ show_quota
VSC_DATA:    used 81MB (0%)  quota 25600MB
VSC_HOME:    used 33MB (1%)  quota 3072MB
VSC_SCRATCH:   used 28MB (0%)  quota 25600MB
VSC_SCRATCH_GLOBAL: used 28MB (0%)  quota 25600MB
VSC_SCRATCH_SITE:   used 28MB (0%)  quota 25600MB
```

or on the UAntwerp clusters

```
$ module load scripts
$ show_quota.py
VSC_DATA:    used 81MB (0%)  quota 25600MB
VSC_HOME:    used 33MB (1%)  quota 3072MB
VSC_SCRATCH:   used 28MB (0%)  quota 25600MB
VSC_SCRATCH_GLOBAL: used 28MB (0%)  quota 25600MB
VSC_SCRATCH_SITE:   used 28MB (0%)  quota 25600MB
```

With this command, you can follow up the consumption of your total disk quota easily, as it is expressed in percentages. Depending of on which cluster you are running the script, it may not be able to show the quota on all your folders. E.g., when running on the tier-1 system Muk, the script will not be able to show the quota on $VSC_HOME or $VSC_DATA if your account is a KU Leuven, UAntwerpen or VUB account.

Once your quota is (nearly) exhausted, you will want to know which directories are responsible for the consumption of your disk space. You can check the size of all subdirectories in the current directory with the "`du`" (**Disk Usage**) command:

```
$ du
256 ./ex01-matlab/log
1536 ./ex01-matlab
768 ./ex04-python
512 ./ex02-python
768 ./ex03-python
5632
```

This shows you first the aggregated size of all subdirectories, and finally the total size of the current directory "." (this includes files stored in the current directory).

If you also want this size to be "human readable" (and not always the total number of kilobytes), you add the parameter "-h":

```
$ du -h
256K ./ex01-matlab/log
1.5M ./ex01-matlab
768K ./ex04-python
512K ./ex02-python
768K ./ex03-python
5.5M .
```

If the number of lower level subdirectories starts to grow too big, you may not want to see the information at that depth; you could just ask for a summary of the current directory:

```
$ du -s
5632 .
$ du -s -h
5.5M .
```

If you want to see the size of any file or top-level subdirectory in the current directory, you could use the following command:

```
$ du -s -h *
1.5M ex01-matlab
512K ex02-python
768K ex03-python
768K ex04-python
256K example.sh
1.5M intro-HPC.pdf
```

Finally, if you don't want to know the size of the data in your current directory, but in some other directory (e.g., your data directory), you just pass this directory as a parameter. The command below will show the disk use in your home directory, even if you are currently in a different directory:

```
$ du -h $VSC_HOME/*
22M /user/home/gent/vsc400/vsc40000/dataset01
36M /user/home/gent/vsc400/vsc40000/dataset02
22M /user/home/gent/vsc400/vsc40000/dataset03
3.5M /user/home/gent/vsc400/vsc40000/primes.txt
```

6.6 Groups

Groups are a way to manage who can access what data. A user can belong to multiple groups at a time. **Groups can be created and managed without any interaction from the system administrators**.

Please note that changes are not instantaneous: it may take about an hour for the changes to propagate throughout the entire HPC infrastructure.

To change the group of a directory and it's underlying directories and files, you can use:

```
$ chgrp -R groupname directory
```

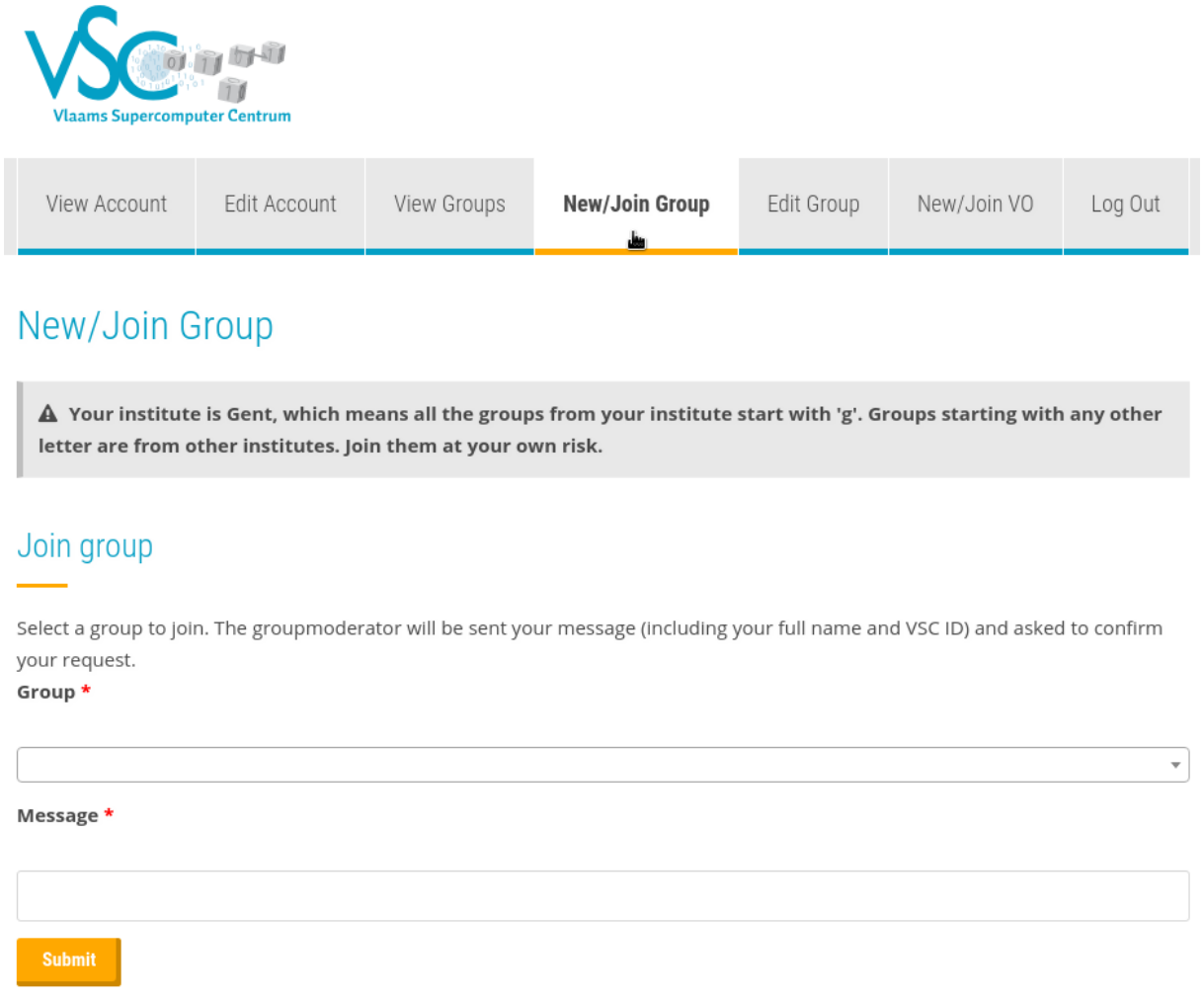
6.6.1 Joining an existing group

1. Get the group name you want to belong to.

2. Go to https://account.vscentrum.be/django/group/new and fill in the section named "Join group". You will be asked to fill in the group name and a message for the moderator of the group, where you identify yourself. This should look something like Figure 6.6.1.

3. After clicking the submit button, a message will be sent to the moderator of the group, who will either approve or deny the request. You will be a member of the group shortly after the group moderator approves your request.

Figure 6.1: Joining a group.



6.6.2 Creating a new group

1. Go to https://account.vscentrum.be/django/group/new and scroll down to the section "Request new group". This should look something like Figure 6.6.2.

2. Fill out the group name. This cannot contain spaces.

3. Put a description of your group in the "Info" field.

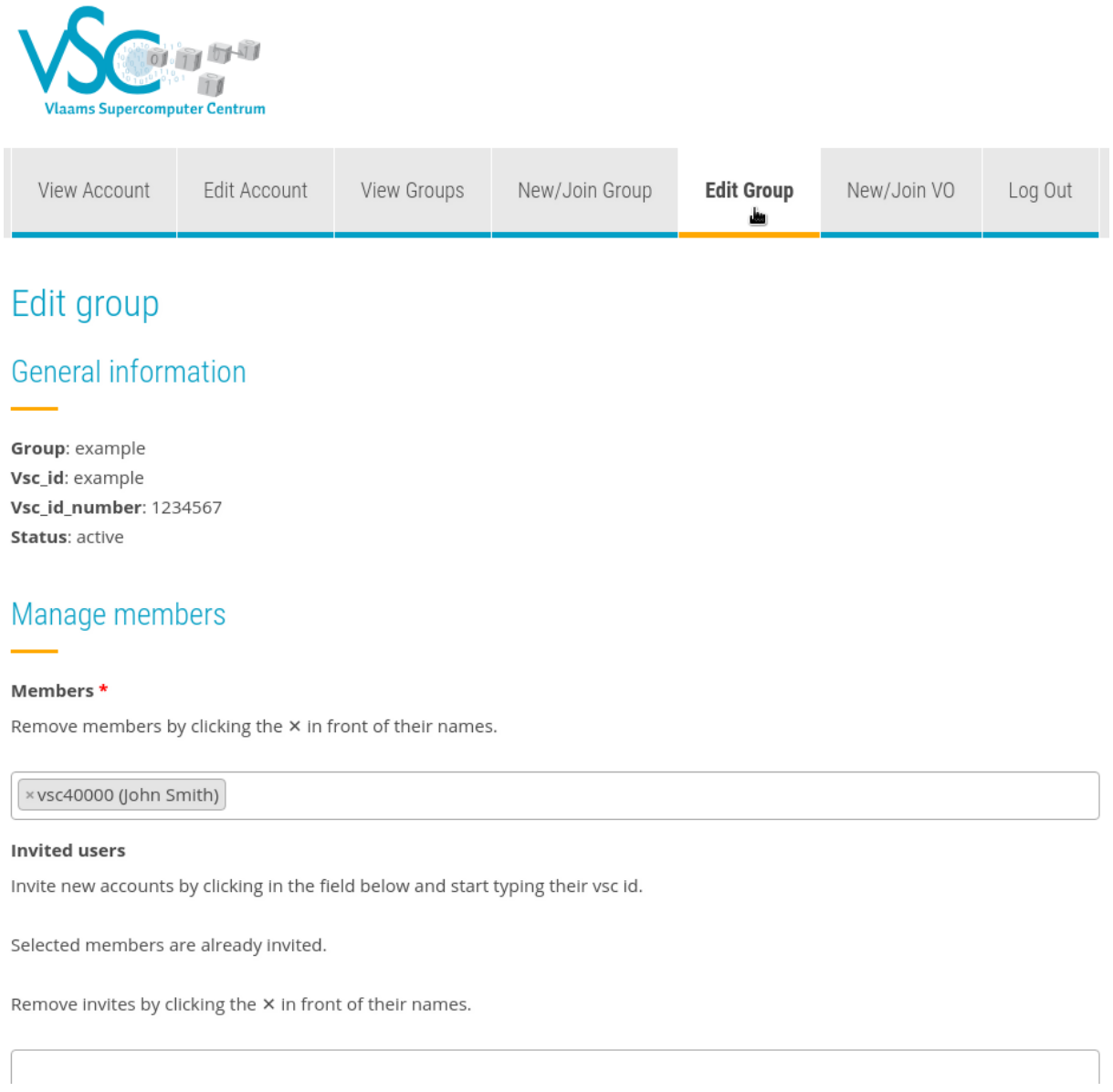
4. You will now be a member and moderator of your newly created group.

Figure 6.2: Creating a new group.

6.6.3 Managing a group

Group moderators can go to `https://account.vscentrum.be/django/group/edit` to manage their group (see Figure 6.6.3). Moderators can invite and remove members. They can also promote other members to moderator and remove other moderators.

Figure 6.3: Creating a new group.



6.6.4 Inspecting groups

You can get details about the current state of groups on the HPC infrastructure with the following command (`example` is the name of the group we want to inspect):

```
$ getent group example
example:*:1234567:vsc40001,vsc40002,vsc40003
```

We can see that the VSC id number is 1234567 and that there are three members in the group: `vsc40001`, `vsc40002` and `vsc40003`.

6.7 Virtual Organisations

A Virtual Organisation (VO) is a special type of group. You can only be a member of one single VO at a time (or not be in a VO at all). Being in a VO allows for larger storage quota to be obtained (but these requests should be well-motivated).

6.7.1 Joining an existing VO

1. Get the VO id of the research group you belong to (this id is formed by the letters `gvo`, followed by 5 digits).
2. Go to https://account.vscentrum.be/django/vo/join and fill in the section named "Join VO". You will be asked to fill in the VO id and a message for the moderator of the VO, where you identify yourself. This should look something like Figure 6.7.1.
3. After clicking the submit button, a message will be sent to the moderator of the VO, who will either approve or deny the request.

6.7.2 Creating a new VO

1. Go to https://account.vscentrum.be/django/vo/new and scroll down to the section "Request new VO". This should look something like Figure 6.7.2.
2. Fill why you want to request a VO.
3. Fill out the both the internal and public VO name. These cannot contain spaces, and should be 8-10 characters long. For example, `genome25` is a valid VO name.
4. Fill out the rest of the form and press submit. This will send a message to the HPC administrators, who will then either approve or deny the request.
5. If the request is approved, you will now be a member and moderator of your newly created VO.

6.7.3 Requesting more storage space

If you're a moderator of a VO, you can request additional quota for the VO and its members.

1. Go to https://account.vscentrum.be/django/vo/edit and scroll down to "Request additional quota". See Figure 6.7.3 to see how this looks.

Figure 6.4: Joining a VO.

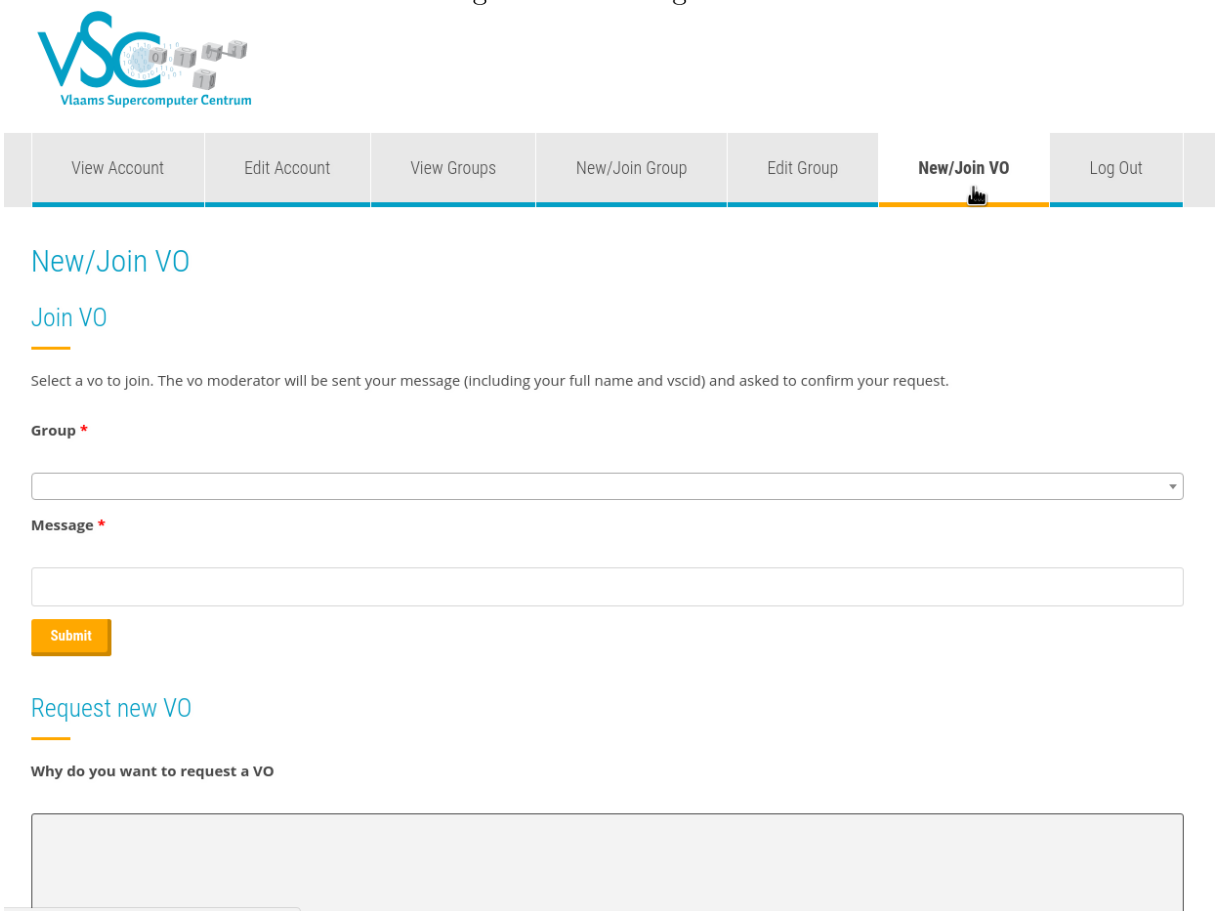


Figure 6.5: Creating a new VO.

2. Fill out how much *additional* storage you want. In the screenshot below, we're asking for 500 GiB extra space for `VSC_DATA`, and for 1 TiB extra space on `VSC_SCRATCH_KYUKON`.
3. Add a comment explaining why you need additional storage space and submit the form.
4. An HPC administrator will review your request and approve or deny it.

Figure 6.6: Requesting additional quota for the VO and its members.

6.7.4 Setting per-member VO quota

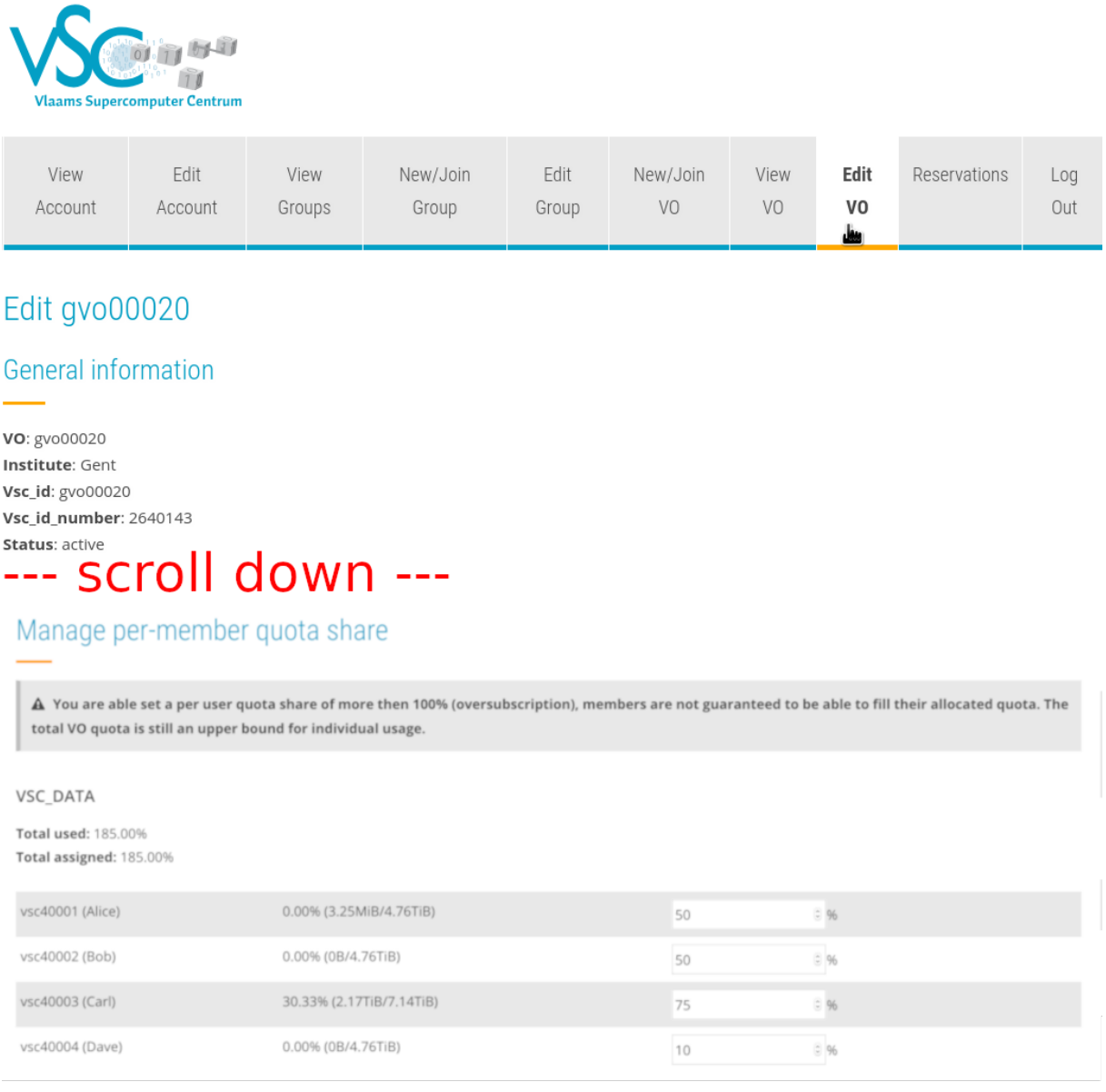
VO moderators can tweak how much of the VO quota each member can use. By default, this is set to 50% for each user, but the moderator can change this: it is possible to give a particular user more than half of the VO quota (for example 80%), or significantly less (for example 10%).

Note that the total percentage can be above 100%: the percentages the moderator allocates per user are *the maximum percentages* of storage users can use.

1. Go to https://account.vscentrum.be/django/vo/edit and scroll down to "Manage per-member quota share". See Figure 6.7.4 to see how this looks.

2. Fill out how much percent of the space you want each user to be able to use. Note that the total can be above 100%. In the screenshot below, there are four users. Alice and Bob can use up to 50% of the space, Carl can use up to 75% of the space, and Dave can only use 10% of the space. So in total, 185% of the space has been assigned, but of course only 100% can actually be used.

Figure 6.7: Setting per-member quota.



6.7.5 VO directories

When you're a member of a VO, there will be some additional directories on each of the shared filesystems available:

VO scratch (`$VSC_SCRATCH_VO`) A directory on the shared *scratch* filesystem shared by the members of your VO, where additional storage quota can be provided (see subsection 6.7.3). You can use this as an alternative to your personal `$VSC_SCRATCH` directory (see subsection 6.2.4).

VO data (`$VSC_DATA_VO`) A directory on the shared *data* filesystem shared by the members of your VO, where additional storage quota can be provided (see subsection 6.7.3). You can use this as an alternative to your personal `$VSC_DATA` directory (see subsection 6.2.3).

If you put `_USER` after each of these variable names, you can see your personal folder in these filesystems. For example: `$VSC_DATA_VO_USER` is your personal folder in your VO data filesystem (this is equivalent to `$VSC_DATA_VO/$USER`), and analogous for `$VSC_SCRATCH_VO_USER` .

Chapter 7

Multi core jobs/Parallel Computing

7.1 Why Parallel Programming?

There are two important motivations to engage in parallel programming.

1. Firstly, the need to decrease the time to solution: distributing your code over C cores holds the promise of speeding up execution times by a factor C. All modern computers (and probably even your smartphone) are equipped with multi-core processors capable of parallel processing.

2. The second reason is problem size: distributing your code over N nodes increases the available memory by a factor N, and thus holds the promise of being able to tackle problems which are N times bigger.

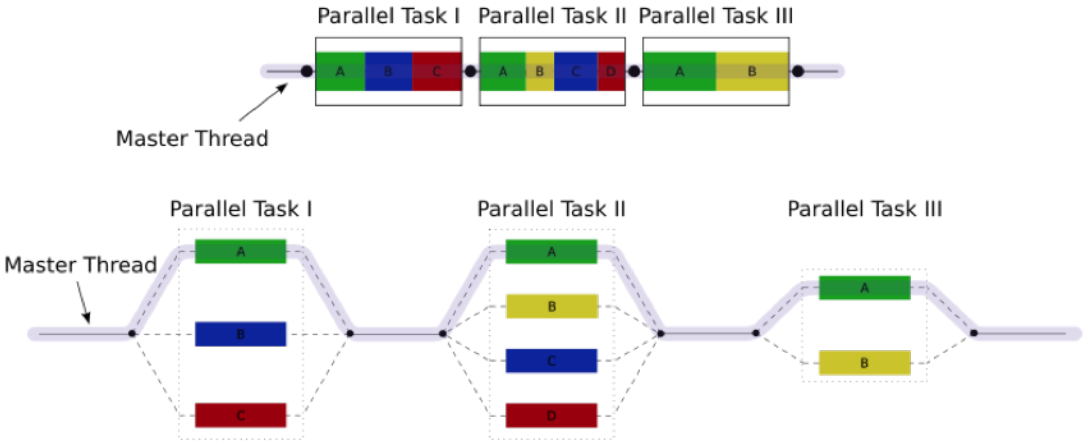
On a desktop computer, this enables a user to run multiple programs and the operating system simultaneously. For scientific computing, this means you have the ability in principle of splitting up your computations into groups and running each group on its own core.

There are multiple different ways to achieve parallel programming. The table below gives a (non-exhaustive) overview of problem independent approaches to parallel programming. In addition there are many problem specific libraries that incorporate parallel capabilities. The next three sections explore some common approaches: (raw) threads, OpenMP and MPI.

Parallel programming approaches		
Tool	**Available language bindings**	**Limitations**
Raw threads pthreads, boost:: threading, ...	Threading libraries are available for all common programming languages	Threads are limited to shared memory systems. They are more often used on single node systems rather than for HPC. Thread management is hard.
OpenMP	Fortran/C/C++	Limited to shared memory systems, but large shared memory systems for HPC are not uncommon (e.g., SGI UV). Loops and task can be parallelised by simple insertion of compiler directives. Under the hood threads are used. Hybrid approaches exist which use OpenMP to parallelise the work load on each node and MPI (see below) for communication between nodes.
Lightweight threads with clever scheduling, Intel TBB, Intel Cilk Plus	C/C++	Limited to shared memory systems, but may be combined with MPI. Thread management is taken care of by a very clever scheduler enabling the programmer to focus on parallelisation itself. Hybrid approaches exist which use TBB and/or Cilk Plus to parallelise the work load on each node and MPI (see below) for communication between nodes.
MPI	Fortran/C/C++, Python	Applies to both distributed and shared memory systems. Cooperation between different nodes or cores is managed by explicit calls to library routines handling communication routines.
Global Arrays library	C/C++, Python	Mimics a global address space on distributed memory systems, by distributing arrays over many nodes and one sided communication. This library is used a lot for chemical structure calculation codes and was used in one of the first applications that broke the PetaFlop barrier.
Scoop	Python	Applies to both shared and distributed memory system. Not extremely advanced, but may present a quick road to parallelisation of Python code.

7.2 Parallel Computing with threads

Multi-threading is a widespread programming and execution model that allows multiple threads to exist within the context of a single process. These threads share the process' resources, but are able to execute independently. The threaded programming model provides developers with a useful abstraction of concurrent execution. Multi-threading can also be applied to a single process to enable parallel execution on a multiprocessing system.



This advantage of a multithreaded program allows it to operate faster on computer systems that have multiple CPUs or across a cluster of machines — because the threads of the program naturally lend themselves to truly concurrent execution. In such a case, the programmer needs to be careful to avoid race conditions, and other non-intuitive behaviours. In order for data to be correctly manipulated, threads will often need to synchronise in time in order to process the data in the correct order. Threads may also require mutually exclusive operations (often implemented using semaphores) in order to prevent common data from being simultaneously modified, or read while in the process of being modified. Careless use of such primitives can lead to deadlocks.

Threads are a way that a program can spawn concurrent units of processing that can then be delegated by the operating system to multiple processing cores. Clearly the advantage of a multithreaded program (one that uses multiple threads that are assigned to multiple processing cores) is that you can achieve big speedups, as all cores of your CPU (and all CPUs if you have more than one) are used at the same time.

Here is a simple example program that spawns 5 threads, where each one runs a simple function that only prints "Hello from thread".

Go to the example directory:

```
$ cd ~/examples/Multi-core-jobs-Parallel-Computing
```

Study the example first:

— T_hello.c —

```c
/*
 * VSC        : Flemish Supercomputing Centre
 * Tutorial   : Introduction to HPC
 * Description: Showcase of working with threads
 */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NTHREADS 5

void *myFun(void *x)
{
  int tid;
  tid = *((int *) x);
  printf("Hello from thread %d!\n", tid);
  return NULL;
}

int main(int argc, char *argv[])
{
  pthread_t threads[NTHREADS];
  int thread_args[NTHREADS];
  int rc, i;

  /* spawn the threads */
  for (i=0; i<NTHREADS; ++i)
    {
      thread_args[i] = i;
      printf("spawning thread %d\n", i);
      rc = pthread_create(&threads[i], NULL, myFun, (void *) &thread_args[i]);
    }

  /* wait for threads to finish */
  for (i=0; i<NTHREADS; ++i) {
    rc = pthread_join(threads[i], NULL);
  }

  return 1;
}
```

And compile it (whilst including the thread library) and run and test it on the login-node:

```
$ module load GCC
$ gcc -o T_hello T_hello.c -lpthread
$ ./T_hello
spawning thread 0
spawning thread 1
spawning thread 2
Hello from thread 0!
Hello from thread 1!
Hello from thread 2!
spawning thread 3
spawning thread 4
Hello from thread 3!
Hello from thread 4!
```

Now, run it on the cluster and check the output:

```
$ qsub T_hello.pbs
123456.master15.delcatty.gent.vsc
$ more T_hello.pbs.o123456
spawning thread 0
spawning thread 1
spawning thread 2
Hello from thread 0!
Hello from thread 1!
Hello from thread 2!
spawning thread 3
spawning thread 4
Hello from thread 3!
Hello from thread 4!
```

Tip: If you plan engaging in parallel programming using threads, this book may prove useful: *Professional Multicore Programming: Design and Implementation for C++ Developers. Cameron Hughes and Tracey Hughes. Wrox 2008.*

7.3 Parallel Computing with OpenMP

OpenMP is an API that implements a multi-threaded, shared memory form of parallelism. It uses a set of compiler directives (statements that you add to your code and that are recognised by your Fortran/C/C++ compiler if OpenMP is enabled or otherwise ignored) that are incorporated at compile-time to generate a multi-threaded version of your code. You can think of Pthreads (above) as doing multi-threaded programming "by hand", and OpenMP as a slightly more automated, higher-level API to make your program multithreaded. OpenMP takes care of many of the low-level details that you would normally have to implement yourself, if you were using Pthreads from the ground up.

An important advantage of OpenMP is that, because it uses compiler directives, the original serial version stays intact, and minimal changes (in the form of compiler directives) are necessary to turn a working serial code into a working parallel code.

Here is the general code structure of an OpenMP program:

```
1  #include <omp.h>
2  main ()  {
3  int var1, var2, var3;
4  // Serial code
5  // Beginning of parallel section. Fork a team of threads.
6  // Specify variable scoping
7
8  #pragma omp parallel private(var1, var2) shared(var3)
9    {
10   // Parallel section executed by all threads
11   // All threads join master thread and disband
12   }
13 // Resume serial code
14 }
```

7.3.1 Private versus Shared variables

By using the private() and shared() clauses, you can specify variables within the parallel region as being **shared**, i.e., visible and accessible by all threads simultaneously, or **private**, i.e., private to each thread, meaning each thread will have its own local copy. In the code example below for parallelising a for loop, you can see that we specify the thread_id and nloops variables as private.

7.3.2 Parallelising for loops with OpenMP

Parallelising for loops is really simple (see code below). By default, loop iteration counters in OpenMP loop constructs (in this case the i variable) in the for loop are set to private variables.

— omp1.c —

```
1  /*
2   * VSC        : Flemish Supercomputing Centre
3   * Tutorial   : Introduction to HPC
4   * Description: Showcase program for OMP loops
5   */
6  /* OpenMP_loop.c  */
7  #include <stdio.h>
8  #include <omp.h>
9
10 int main(int argc, char **argv)
11 {
12   int i, thread_id, nloops;
13
14 #pragma omp parallel private(thread_id, nloops)
15   {
16     nloops = 0;
17
18 #pragma omp for
19     for (i=0; i<1000; ++i)
20     {
21       ++nloops;
22     }
23     thread_id = omp_get_thread_num();
24     printf("Thread %d performed %d iterations of the loop.\n", thread_id, nloops );
25   }
26
27   return 0;
28 }
```

And compile it (whilst including the "*openmp*" library) and run and test it on the login-node:

```
$ module load GCC
$ gcc -fopenmp -o omp1 omp1.c
$ ./omp1
Thread 6 performed 125 iterations of the loop.
Thread 7 performed 125 iterations of the loop.
Thread 5 performed 125 iterations of the loop.
Thread 4 performed 125 iterations of the loop.
Thread 0 performed 125 iterations of the loop.
Thread 2 performed 125 iterations of the loop.
Thread 3 performed 125 iterations of the loop.
Thread 1 performed 125 iterations of the loop.
```

Now run it in the cluster and check the result again.

```
$ qsub omp1.pbs
$ cat omp1.pbs.o*
Thread 1 performed 125 iterations of the loop.
Thread 4 performed 125 iterations of the loop.
Thread 3 performed 125 iterations of the loop.
Thread 0 performed 125 iterations of the loop.
Thread 5 performed 125 iterations of the loop.
Thread 7 performed 125 iterations of the loop.
Thread 2 performed 125 iterations of the loop.
Thread 6 performed 125 iterations of the loop.
```

7.3.3 Critical Code

Using OpenMP you can specify something called a "critical" section of code. This is code that is performed by all threads, but is only performed **one thread at a time** (i.e., in serial). This provides a convenient way of letting you do things like updating a global variable with local results from each thread, and you don't have to worry about things like other threads writing to that global variable at the same time (a collision).

— omp2.c —

```c
/*
 * VSC        : Flemish Supercomputing Centre
 * Tutorial   : Introduction to HPC
 * Description: OpenMP Test Program
 */
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[])
{
  int i, thread_id;
  int glob_nloops, priv_nloops;
  glob_nloops = 0;

  // parallelize this chunk of code
  #pragma omp parallel private(priv_nloops, thread_id)
  {
    priv_nloops = 0;
    thread_id = omp_get_thread_num();

    // parallelize this for loop
    #pragma omp for
    for (i=0; i<100000; ++i)
    {
      ++priv_nloops;
    }

    // make this a "critical" code section
    #pragma omp critical
    {
      printf("Thread %d is adding its iterations (%d) to sum (%d), ", thread_id,
   priv_nloops, glob_nloops);
      glob_nloops += priv_nloops;
      printf("total is now %d.\n", glob_nloops);
    }
  }
  printf("Total # loop iterations is %d\n", glob_nloops);
  return 0;
}
```

And compile it (whilst including the "*openmp*" library) and run and test it on the login-node:

```
$ module load GCC
$ gcc -fopenmp -o omp2 omp2.c
$ ./omp2
Thread 3 is adding its iterations (12500) to sum (0), total is now 12500.
Thread 7 is adding its iterations (12500) to sum (12500), total is now 25000.
Thread 5 is adding its iterations (12500) to sum (25000), total is now 37500.
Thread 6 is adding its iterations (12500) to sum (37500), total is now 50000.
Thread 2 is adding its iterations (12500) to sum (50000), total is now 62500.
Thread 4 is adding its iterations (12500) to sum (62500), total is now 75000.
Thread 1 is adding its iterations (12500) to sum (75000), total is now 87500.
Thread 0 is adding its iterations (12500) to sum (87500), total is now 100000.
Total # loop iterations is 100000
```

Now run it in the cluster and check the result again.

```
$ qsub omp2.pbs
$ cat omp2.pbs.o*
Thread 2 is adding its iterations (12500) to sum (0), total is now 12500.
Thread 0 is adding its iterations (12500) to sum (12500), total is now 25000.
Thread 1 is adding its iterations (12500) to sum (25000), total is now 37500.
Thread 4 is adding its iterations (12500) to sum (37500), total is now 50000.
Thread 7 is adding its iterations (12500) to sum (50000), total is now 62500.
Thread 3 is adding its iterations (12500) to sum (62500), total is now 75000.
Thread 5 is adding its iterations (12500) to sum (75000), total is now 87500.
Thread 6 is adding its iterations (12500) to sum (87500), total is now 100000.
Total # loop iterations is 100000
```

7.3.4 Reduction

Reduction refers to the process of combining the results of several sub-calculations into a final result. This is a very common paradigm (and indeed the so-called "map-reduce" framework used by Google and others is very popular). Indeed we used this paradigm in the code example above, where we used the "critical code" directive to accomplish this. The map-reduce paradigm is so common that OpenMP has a specific directive that allows you to more easily implement this.

— omp3.c —

```c
/*
 * VSC        : Flemish Supercomputing Centre
 * Tutorial   : Introduction to HPC
 * Description: OpenMP Test Program
 */
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[])
{
  int i, thread_id;
  int glob_nloops, priv_nloops;
  glob_nloops = 0;

  // parallelize this chunk of code
  #pragma omp parallel private(priv_nloops, thread_id) reduction(+:glob_nloops)
  {
    priv_nloops = 0;
    thread_id = omp_get_thread_num();

    // parallelize this for loop
    #pragma omp for
    for (i=0; i<100000; ++i)
    {
      ++priv_nloops;
    }
    glob_nloops += priv_nloops;
  }
  printf("Total # loop iterations is %d\n", glob_nloops);
  return 0;
}
```

And compile it (whilst including the "*openmp*" library) and run and test it on the login-node:

```
$ module load GCC
$ gcc -fopenmp -o omp3 omp3.c
$ ./omp3
Total # loop iterations is 100000
```

Now run it in the cluster and check the result again.

```
$ qsub omp3.pbs
$ cat omp3.pbs.o*
Total # loop iterations is 100000
```

7.3.5 Other OpenMP directives

There are a host of other directives you can issue using OpenMP.

Some other clauses of interest are:

1. barrier: each thread will wait until all threads have reached this point in the code, before

proceeding

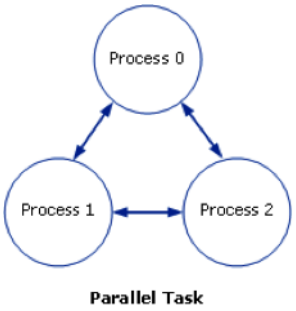
2. nowait: threads will not wait until everybody is finished
3. schedule(type, chunk) allows you to specify how tasks are spawned out to threads in a for loop. There are three types of scheduling you can specify
4. if: allows you to parallelise only if a certain condition is met
5. ... and a host of others

Tip: If you plan engaging in parallel programming using OpenMP, this book may prove useful: *Using OpenMP - Portable Shared Memory Parallel Programming.* By Barbara Chapman Gabriele Jost and Ruud van der Pas Scientific and Engineering Computation. 2005.

7.4 Parallel Computing with MPI

The Message Passing Interface (MPI) is a standard defining core syntax and semantics of library routines that can be used to implement parallel programming in C (and in other languages as well). There are several implementations of MPI such as Open MPI, Intel MPI, M(VA)PICH and LAM/MPI.

In the context of this tutorial, you can think of MPI, in terms of its complexity, scope and control, as sitting in between programming with Pthreads, and using a high-level API such as OpenMP. For a Message Passing Interface (MPI) application, a parallel task usually consists of a single executable running concurrently on multiple processors, with communication between the processes. This is shown in the following diagram:



The process numbers 0, 1 and 2 represent the process rank and have greater or less significance depending on the processing paradigm. At the minimum, Process 0 handles the input/output and determines what other processes are running.

The MPI interface allows you to manage allocation, communication, and synchronisation of a set of processes that are mapped onto multiple nodes, where each node can be a core within a single CPU, or CPUs within a single machine, or even across multiple machines (as long as they are networked together).

One context where MPI shines in particular is the ability to easily take advantage not just of multiple cores on a single machine, but to run programs on clusters of several machines. Even if

you don't have a dedicated cluster, you could still write a program using MPI that could run your program in parallel, across any collection of computers, as long as they are networked together.

Here is a "Hello World" program in MPI written in C. In this example, we send a "Hello" message to each processor, manipulate it trivially, return the results to the main process, and print the messages.

Study the MPI-programme and the PBS-file:

— mpi_hello.c —

```c
/*
 * VSC        : Flemish Supercomputing Centre
 * Tutorial   : Introduction to HPC
 * Description: "Hello World" MPI Test Program
 */
#include <stdio.h>
#include <mpi.h>

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

 #define BUFSIZE 128
 #define TAG 0

 int main(int argc, char *argv[])
 {
   char idstr[32];
   char buff[BUFSIZE];
   int numprocs;
   int myid;
   int i;
   MPI_Status stat;
   /* MPI programs start with MPI_Init; all 'N' processes exist thereafter */
   MPI_Init(&argc,&argv);
   /* find out how big the SPMD world is */
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   /* and this processes' rank is */
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);

   /* At this point, all programs are running equivalently, the rank
      distinguishes the roles of the programs in the SPMD model, with
      rank 0 often used specially... */
   if(myid == 0)
   {
     printf("%d: We have %d processors\n", myid, numprocs);
     for(i=1;i<numprocs;i++)
     {
       sprintf(buff, "Hello %d! ", i);
       MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD);
     }
     for(i=1;i<numprocs;i++)
     {
       MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD, &stat);
       printf("%d: %s\n", myid, buff);
     }
   }
   else
   {
     /* receive from rank 0: */
     MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &stat);
     sprintf(idstr, "Processor %d ", myid);
     strncat(buff, idstr, BUFSIZE-1);
     strncat(buff, "reporting for duty", BUFSIZE-1);
     /* send to rank 0: */
     MPI_Send(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD);
   }

   /* MPI programs end with MPI Finalize; this is a weak synchronization point */
   MPI_Finalize();
   return 0;
 }
```

— mpi_hello.pbs —

```
#!/bin/bash

#PBS -N mpihello
#PBS -l walltime=00:05:00

  # assume a 40 core job
#PBS -l nodes=2:ppn=20

  # make sure we are in the right directory in case writing files
cd $PBS_O_WORKDIR

  # load the environment

module load intel

mpirun ./mpi_hello
```

and compile it:

```
$ module load intel
$ mpiicc -o mpi_hello mpi_hello.c
```

mpiicc is a wrapper of the Intel C++ compiler icc to compile MPI programs (see the chapter on compilation for details).

Run the parallel program:

```
$ qsub mpi_hello.pbs
$ ls -l
total 1024
-rwxrwxr-x 1 vsc40000 8746 Sep 16 14:19 mpi_hello*
-rw-r--r-- 1 vsc40000 1626 Sep 16 14:18 mpi_hello.c
-rw------- 1 vsc40000    0 Sep 16 14:22 mpi_hello.o123456
-rw------- 1 vsc40000  697 Sep 16 14:22 mpi_hello.o123456
-rw-r--r-- 1 vsc40000  304 Sep 16 14:22 mpi_hello.pbs
$ cat mpi_hello.o123456
0: We have 16 processors
0: Hello 1! Processor 1 reporting for duty
0: Hello 2! Processor 2 reporting for duty
0: Hello 3! Processor 3 reporting for duty
0: Hello 4! Processor 4 reporting for duty
0: Hello 5! Processor 5 reporting for duty
0: Hello 6! Processor 6 reporting for duty
0: Hello 7! Processor 7 reporting for duty
0: Hello 8! Processor 8 reporting for duty
0: Hello 9! Processor 9 reporting for duty
0: Hello 10! Processor 10 reporting for duty
0: Hello 11! Processor 11 reporting for duty
0: Hello 12! Processor 12 reporting for duty
0: Hello 13! Processor 13 reporting for duty
0: Hello 14! Processor 14 reporting for duty
0: Hello 15! Processor 15 reporting for duty
```

The runtime environment for the MPI implementation used (often called mpirun or mpiexec)

spawns multiple copies of the program, with the total number of copies determining the number of process *ranks* in MPI_COMM_WORLD, which is an opaque descriptor for communication between the set of processes. A single process, multiple data (SPMD = Single Program, Multiple Data) programming model is thereby facilitated, but not required; many MPI implementations allow multiple, different, executables to be started in the same MPI job. Each process has its own rank, the total number of processes in the world, and the ability to communicate between them either with point-to-point (send/receive) communication, or by collective communication among the group. It is enough for MPI to provide an SPMD-style program with MPI_COMM_WORLD, its own rank, and the size of the world to allow algorithms to decide what to do. In more realistic situations, I/O is more carefully managed than in this example. MPI does not guarantee how POSIX I/O would actually work on a given system, but it commonly does work, at least from rank 0.

MPI uses the notion of process rather than processor. Program copies are *mapped* to processors by the MPI runtime. In that sense, the parallel machine can map to 1 physical processor, or N where N is the total number of processors available, or something in between. For maximum parallel speedup, more physical processors are used. This example adjusts its behaviour to the size of the world N, so it also seeks to scale to the runtime configuration without compilation for each size variation, although runtime decisions might vary depending on that absolute amount of concurrency available.

Tip: mpirun does not always do the optimal core pinning and requires a few extra arguments to be the most efficient possible on a given system. At Ghent we have a wrapper around `mpirun` called `mympirun`. See chapter 21 for more information.

You will generally just start an MPI program on the UGent-HPC by using `mympirun` instead of `mpirun -n <nr of cores> <--other settings> <--other optimisations>`

Tip: If you plan engaging in parallel programming using MPI, this book may prove useful: *Parallel Programming with MPI. Peter Pacheo. Morgan Kaufmann. 1996.*

Chapter 8

Troubleshooting

8.1 Walltime issues

If you get from your job output an error message similar to this:

```
=>> PBS: job killed: walltime <value in seconds> exceeded limit <value in seconds>
```

This occurs when your job did not complete within the requested walltime. See section 11.1 for more information about how to request the walltime. It is recommended to use *checkpointing* if the job requires **72 hours** of walltime or more to be executed.

8.2 Out of quota issues

Sometimes a job hangs at some point or it stops writing in the disk. These errors are usually related to the quota usage. You may have reached your quota limit at some storage endpoint. You should move (or remove) the data to a different storage endpoint (or request more quota) to be able to write to the disk and then resubmit the jobs. Another option is to request extra quota for your VO to the VO moderator/s. See section 6.2.1 and section 6.2.5 for more information about quotas and how to use the storage endpoints in an efficient way.

8.3 Issues connecting to login node

If you are confused about the SSH public/private key pair concept, maybe the key/lock analogy in subsection 2.1.1 can help.

If you have errors that look like:

```
vsc40000@login.hpc.ugent.be: Permission denied
```

or you are experiencing problems with connecting, here is a list of things to do that should help:

1. Keep in mind that it an take up to an hour for your VSC account to become active after it has been *approved*; until then, logging in to your VSC account will not work.

2. Please double/triple check your VSC login ID. It should look something like *vsc40000*: the letters `vsc`, followed by exactly 5 digits. Make sure it's the same one as the one on https://account.vscentrum.be/.

3. You previously connected to the HPC from another machine, but now have another machine? Please follow the procedure for adding additional keys in section 2.2.2. You may need to wait for 15-20 minutes until the SSH public key(s) you added become active.

4. Make sure you are using the private key (not the public key) when trying to connect: If you followed the manual, the private key filename should end in `.ppk` (not in `.pub`).

5. If you have multiple private keys on your machine, please make sure you are using the one that corresponds to (one of) the public key(s) you added on https://account.vscentrum.be/.

6. Please do not use someone else's private keys. You must never share your private key, they're called *private* for a good reason.

If you are using PuTTY and get this error message:

```
server unexpectedly closed network connection
```

it is possible that the PuTTY version you are using is too old and doesn't support some required (security-related) features.

Make sure you are using the latest PuTTY version if you are encountering problems connecting (see subsection 2.1.2). If that doesn't help, please contact hpc@ugent.be.

If you've tried all applicable items above and it doesn't solve your problem, please contact hpc@ugent.be and include the following information:

Please create a log file of your SSH session by following the steps in this article and include it in the email.

8.4 Security warning about invalid host key

If you get a warning that looks like the one below, it is possible that someone is trying to intercept the connection between you and the system you are connecting to. Another possibility is that the host key of the system you are connecting to has changed.

You will need to verify that the fingerprint shown in the dialog matches one of the following fingerprints:

- `ssh-rsa 2048 2f:0c:f7:76:87:57:f7:5d:2d:7b:d1:a1:e1:86:19:f3`
- `ssh-ed25519 256 fa:23:ab:1f:f0:65:f3:0d:d3:33:ce:7a:f8:f4:fc:2a`
- `ssh-ecdsa 256 13:f0:11:d1:94:cb:ca:e5:ca:82:21:62:ab:9f:3f:c2`

Do not click "Yes" until you verified the fingerprint. Do not press "No" in any case..

If it the fingerprint matches, click "Yes".

If it doesn't (like in the example) or you are in doubt, take a screenshot, press "Cancel" and contact hpc@ugent.be.

8.5 DOS/Windows text format

If you get errors like:

```
$ qsub fibo.pbs
qsub:  script is written in DOS/Windows text format
```

It's probably because you transferred the files from a Windows computer. Please go to the section about `dos2unix` in chapter 5 of the intro to Linux to fix this error.

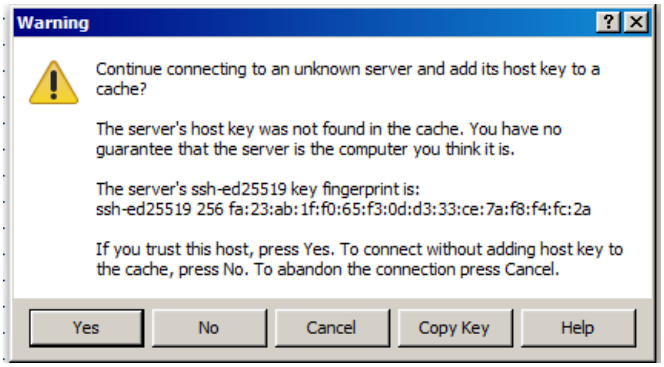
8.6 Warning message when first connecting to new host

The first time you make a connection to the login node, a Security Alert will appear and you will be asked to verify the authenticity of the login node.

Make sure the fingerprint in the alert matches one of the following:

- `ssh-rsa 2048 2f:0c:f7:76:87:57:f7:5d:2d:7b:d1:a1:e1:86:19:f3`
- `ssh-ed25519 256 fa:23:ab:1f:f0:65:f3:0d:d3:33:ce:7a:f8:f4:fc:2a`
- `ssh-ecdsa 256 13:f0:11:d1:94:cb:ca:e5:ca:82:21:62:ab:9f:3f:c2`

If it does, press ***Yes***, if it doesn't, please contact hpc@ugent.be.



8.7 Memory limits

To avoid jobs allocating too much memory, there are memory limits in place by default. It is possible to specify higher memory limits if your jobs require this.

8.7.1 How will I know if memory limits are the cause of my problem?

If your program fails with a memory-related issue, there is a good chance it failed because of the memory limits and you should increase the memory limits for your job.

Examples of these error messages are: `malloc failed`, `Out of memory`, `Could not allocate memory` or in Java: `Could not reserve enough space for object heap` . Your program can also run into a `Segmentation fault` (or `segfault`) or crash due to bus errors.

You can check the amount of virtual memory (in Kb) that is available to you via the `ulimit -v` command *in your job script*.

8.7.2 How do I specify the amount of memory I need?

See subsection 4.6.1 to set memory and other requirements, see section 11.2 to finetune the amount of memory you request.

8.8 Module conflicts

Modules that are loaded together must use the same toolchain version: it is impossible to load two versions of the same module. In the following example, we try to load a module that uses the `intel-2018a` toolchain together with one that uses the `intel-2017a` toolchain:

```
$ module load Python/2.7.14-intel-2018a
$ module load HMMER/3.1b2-intel-2017a
Lmod has detected the following error:  A different version of the 'intel' module is
     already loaded (see output of 'ml').
You should load another 'HMMER' module for that is compatible with the currently
    loaded version of 'intel'.
Use 'ml avail HMMER' to get an overview of the available versions.

If you don't understand the warning or error, contact the helpdesk at hpc@ugent.be
While processing the following module(s):

    Module fullname          Module Filename
    ---------------          ---------------
    HMMER/3.1b2-intel-2017a  /apps/gent/CO7/haswell-ib/modules/all/HMMER/3.1b2-intel
    -2017a.lua
```

This resulted in an error because we tried to load two different versions of the `intel` module.

To fix this, check if there are other versions of the modules you want to load that have the same version of common dependencies. You can list all versions of a module with `module avail`: for `HMMER`, this command is `module avail HMMER`.

Another common error is:

```
$ module load cluster/swalot
Lmod has detected the following error:  A different version of the 'cluster' module
    is already loaded (see output of 'ml').

If you don't understand the warning or error, contact the helpdesk at hpc@ugent.be
```

This is because there can only be one `cluster` module active at a time. The correct command is `module swap cluster/swalot`. See also subsection 4.3.2.

8.9 Running software that is incompatible with host

When running software provided through modules (see section 4.1), you may run into errors like:

```
$ module swap cluster/golett

The following have been reloaded with a version change:
  1) cluster/delcatty => cluster/golett

$ module load Python/2.7.14-intel-2018a
$ python

Please verify that both the operating system and the processor support Intel(R)
    MOVBE, F16C, FMA, BMI, LZCNT and AVX2 instructions.
```

or errors like:

```
$ module swap cluster/golett

The following have been reloaded with a version change:
  1) cluster/delcatty => cluster/golett

$ module load Python/2.7.14-foss-2018a
$ python
Illegal instruction
```

When we swap to a different cluster, the available modules change so they work for that cluster. That means that if the cluster and the login nodes have a different CPU architecture, software loaded using modules might not work.

If you want to test software on the login nodes, make sure the `cluster/delcatty` module is loaded (with `module swap cluster/delcatty`, see subsection 4.3.2), since the login nodes and delcatty have the same CPU architecture.

If modules are already loaded, and then we swap to a different cluster, all our modules will get reloaded. This means that all current modules will be unloaded and then loaded again, so they'll work on the newly loaded cluster. Here's an example of how that would look like:

```
$ module load Python/2.7.14-intel-2018a
$ module swap cluster/swalot

Due to MODULEPATH changes, the following have been reloaded:
  1) GCCcore/6.4.0                    5) Tcl/8.6.8-GCCcore-6.4.0           9)
    iccifort/2018.1.163-GCC-6.4.0-2.28    13) impi/2018.1.163-iccifort-2018.1.163-
    GCC-6.4.0-2.28    17) ncurses/6.0-GCCcore-6.4.0
  2) GMP/6.1.2-GCCcore-6.4.0          6) binutils/2.28-GCCcore-6.4.0      10) ifort
    /2018.1.163-GCC-6.4.0-2.28       14) intel/2018a
              18) zlib/1.2.11-GCCcore-6.4.0
  3) Python/2.7.14-intel-2018a        7) bzip2/1.0.6-GCCcore-6.4.0        11) iimpi
    /2018a                           15) libffi/3.2.1-GCCcore-6.4.0
  4) SQLite/3.21.0-GCCcore-6.4.0      8) icc/2018.1.163-GCC-6.4.0-2.28    12) imkl
    /2018.1.163-iimpi-2018a           16) libreadline/7.0-GCCcore-6.4.0

The following have been reloaded with a version change:
  1) cluster/delcatty => cluster/swalot
```

This might result in the same problems as mentioned above. When swapping to a different cluster, you can run `module purge` to unload all modules to avoid problems (see subsection 4.1.6).

Chapter 9

HPC Policies

Stub chapter

Chapter 10

Frequently Asked Questions

10.1 When will my job start?

See the explanation about how jobs get prioritized in subsection 4.3.1.

10.2 Can I share my account with someone else?

NO. You are not allowed to share your VSC account with anyone else, it is strictly personal. See https://helpdesk.ugent.be/account/en/regels.php. If you want to share data, there are alternatives (like a shared directories in VO space, see section 6.7).

10.3 Can I share my data with other HPC users?

Yes, you can use the `chmod` or `setfacl` commands to change permissions of files so other users can access the data. For example, the following command will enable a user named "otheruser" to read the file named `dataset.txt`. See

```
$ setfacl -m u:otheruser:r dataset.txt
$ ls -l dataset.txt
-rwxr-x---+ 2 vsc40000 mygroup      40 Apr 12 15:00 dataset.txt
```

For more information about `chmod` or `setfacl`, see the section on chmod in chapter 3 of the Linux intro manual.

10.4 Can I use multple different SSH key pairs to connect ot my VSC account?

Yes, and this is recommendend when working from different computers. Please see subsection 2.2.2 on how to do this.

10.5 I want to use software that is not available on the clusters yet

Please fill out the details about the software and why you need it in this form: https://www.ugent.be/hpc/en/support/software-installation-request. When submitting the form, a mail will be sent to hpc@ugent.be containing all the provided information. The HPC team will look into your request as soon as possible you and contact you when the installation is done or if further information is required.

Part II

Advanced Guide

Chapter 11

Fine-tuning Job Specifications

As HPC system administrators, we often observe that the HPC resources are not optimally (or wisely) used. For example, we regularly notice that several cores on a computing node are not utilised, due to the fact that one sequential program uses only one core on the node. Or users run I/O intensive applications on nodes with "slow" network connections.

Users often tend to run their jobs without specifying specific PBS Job parameters. As such, their job will automatically use the default parameters, which are not necessarily (or rarely) the optimal ones. This can slow down the run time of your application, but also block HPC resources for other users.

Specifying the "optimal" Job Parameters requires some knowledge of your application (e.g., how many parallel threads does my application uses, is there a lot of inter-process communication, how much memory does my application need) and also some knowledge about the HPC infrastructure (e.g., what kind of multi-core processors are available, which nodes have Infiniband).

There are plenty of monitoring tools on Linux available to the user, which are useful to analyse your individual application. The HPC environment as a whole often requires different techniques, metrics and time goals, which are not discussed here. We will focus on tools that can help to optimise your Job Specifications.

Determining the optimal computer resource specifications can be broken down into different parts. The first is actually determining which metrics are needed and then collecting that data from the hosts. Some of the most commonly tracked metrics are CPU usage, memory consumption, network bandwidth, and disk I/O stats. These provide different indications of how well a system is performing, and may indicate where there are potential problems or performance bottlenecks. Once the data have actually been acquired, the second task is analysing the data and adapting your PBS Job Specifications.

Another different task is to monitor the behaviour of an application at run time and detect anomalies or unexpected behaviour. Linux provides a large number of utilities to monitor the performance of its components.

This chapter shows you how to measure:

1. Walltime
2. Memory usage

3. CPU usage
4. Disk (storage) needs
5. Network bottlenecks

First, we allocate a compute node and move to our relevant directory:

```
$ qsub -I
$ cd ~/examples/Fine-tuning-Job-Specifications
```

11.1 Specifying Walltime

One of the most important and also easiest parameters to measure is the duration of your program. This information is needed to specify the *walltime*.

The *time* utility **executes** and **times** your application. You can just add the time command in front of your normal command line, including your command line options. After your executable has finished, **time** writes the total time elapsed, the time consumed by system overhead, and the time used to execute your executable to the standard error stream. The calculated times are reported in seconds.

Test the time command:

```
$ time sleep 75
real 1m15.005s
user 0m0.001s
sys 0m0.002s
```

It is a good practice to correctly estimate and specify the run time (duration) of an application. Of course, a margin of 10% to 20% can be taken to be on the safe side.

It is also wise to check the walltime on different compute nodes or to select the "slowest" compute node for your walltime tests. Your estimate should appropriate in case your application will run on the "slowest" (oldest) compute nodes.

The walltime can be specified in a job scripts as:

```
#PBS -l walltime=3:00:00:00
```

or on the command line

```
$ qsub -l walltime=3:00:00:00
```

It is recommended to always specify the walltime for a job.

11.2 Specifying memory requirements

In many situations, it is useful to monitor the amount of memory an application is using. You need this information to determine the characteristics of the required compute node, where that

application should run on. Estimating the amount of memory an application will use during execution is often non-trivial, especially when one uses third-party software.

11.2.1 Available Memory on the machine

The first point is to be aware of the available free memory in your computer. The "*free*" command displays the total amount of free and used physical and swap memory in the system, as well as the buffers used by the kernel. We also use the options "-m" to see the results expressed in Mega-Bytes and the "-t" option to get totals.

```
$ free -m -t
                 total   used   free  shared  buffers  cached
Mem:             16049   4772  11277       0      107     161
-/+ buffers/cache:       4503  11546
Swap:            16002   4185  11816
Total:           32052   8957  23094
```

Important is to note the total amount of memory available in the machine (i.e., 16 GB in this example) and the amount of used and free memory (i.e., 4.7 GB is used and another 11.2 GB is free here).

It is not a good practice to use swap-space for your computational applications. A lot of "swapping" can increase the execution time of your application tremendously.

11.2.2 Checking the memory consumption

To monitor the memory consumption of a running application, you can use the "*top*" or the "*htop*" command.

top provides an ongoing look at processor activity in real time. It displays a listing of the most CPU-intensive tasks on the system, and can provide an interactive interface for manipulating processes. It can sort the tasks by memory usage, CPU usage and run time.

htop is similar to top, but shows the CPU-utilisation for all the CPUs in the machine and allows to scroll the list vertically and horizontally to see all processes and their full command lines.

```
$ top
$ htop
```

11.2.3 Setting the memory parameter

Once you gathered a good idea of the overall memory consumption of your application, you can define it in your job script. It is wise to foresee a margin of about 10%.

Sequential or single-node applications:

The maximum amount of physical memory used by the job can be specified in a job script as:

```
#PBS -l mem=4gb
```

or on the command line

```
$ qsub -l mem=4gb
```

This setting is ignored if the number of nodes is not 1.

Parallel or multi-node applications:

When you are running a parallel application over multiple cores, you can also specify the memory requirements per processor (pmem). This directive specifies the maximum amount of physical memory used by any process in the job.

For example, if the job would run four processes and each would use up to 2 GB (gigabytes) of memory, then the memory directive would read:

```
#PBS -l pmem=2gb
```

or on the command line

```
$ qsub -l pmem=2gb
```

(and of course this would need to be combined with a CPU cores directive such as nodes=1:ppn=4). In this example, you request 8 GB of memory in total on the node.

11.3 Specifying processors requirements

Users are encouraged to fully utilise all the available cores on a certain compute node. Once the required numbers of cores and nodes are decently specified, it is also good practice to monitor the CPU utilisation on these cores and to make sure that all the assigned nodes are working at full load.

11.3.1 Number of processors

The number of core and nodes that a user shall request fully depends on the architecture of the application. Developers design their applications with a strategy for parallelisation in mind. The application can be designed for a certain fixed number or for a configurable number of nodes and cores. It is wise to target a specific set of compute nodes (e.g., Westmere, Harpertown) for your computing work and then to configure your software to nicely fill up all processors on these compute nodes.

The */proc/cpuinfo* stores info about your CPU architecture like number of CPUs, threads, cores, information about CPU caches, CPU family, model and much more. So, if you want to detect how many cores are available on a specific machine:

```
$ less /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 23
model name      : Intel(R) Xeon(R) CPU  E5420  @ 2.50GHz
stepping        : 10
cpu MHz         : 2500.088
cache size      : 6144 KB
...
```

Or if you want to see it in a more readable format, execute:

```
$ grep processor /proc/cpuinfo
processor : 0
processor : 1
processor : 2
processor : 3
processor : 4
processor : 5
processor : 6
processor : 7
```

Remark: Unless you want information of the login nodes, you'll have to issue these commands on one of the workernodes. This is most easily achieved in an interactive job, see the chapter on Running interactive jobs.

In order to specify the number of nodes and the number of processors per node in your job script, use:

```
#PBS -l nodes=N:ppn=M
```

or with equivalent parameters on the command line

```
$ qsub -l nodes=N:ppn=M
```

This specifies the number of nodes (nodes=N) and the number of processors per node (ppn=M) that the job should use. PBS treats a processor core as a processor, so a system with eight cores per compute node can have ppn=8 as its maximum ppn request. You can also use this statement in your job script:

```
#PBS -l nodes=N:ppn=all
```

to request all cores of a node, or

```
#PBS -l nodes=N:ppn=half
```

to request half of them.

Note that unless a job has some inherent parallelism of its own through something like MPI or OpenMP, requesting more than a single processor on a single node is usually wasteful and can impact the job start time.

11.3.2 Monitoring the CPU-utilisation

This could also be monitored with the ***htop*** command:

```
$ htop
```

```
  1  [|||   11.0%]   5  [||     3.0%]     9  [||     3.0%]   13 [       0.0%]
  2  [|||||100.0%]   6  [       0.0%]     10 [       0.0%]   14 [       0.0%]
  3  [||     4.9%]   7  [||     9.1%]     11 [       0.0%]   15 [       0.0%]
  4  [||     1.8%]   8  [       0.0%]     12 [       0.0%]   16 [       0.0%]
  Mem[||||||||||||||||59211/64512MB]     Tasks: 323, 932 thr; 2 running
  Swp[||||||||||||      7943/20479MB]     Load average: 1.48 1.46 1.27
                                          Uptime: 211 days(!), 22:12:58

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
22350 vsc00000   20   0 1729M 1071M   704 R 98.0  1.7 27:15.59 bwa index
 7703 root        0 -20 10.1G 1289M 70156 S 11.0  2.0 36h10:11 /usr/lpp/mmfs/bin
27905 vsc00000   20   0  123M  2800  1556 R  7.0  0.0  0:17.51 htop
```

The advantage of htop is that it shows you the cpu utilisation for all processors as well as the details per application. A nice exercise is to start 4 instances of the "cpu_eat" program in 4 different terminals, and inspect the cpu utilisation per processor with monitor and htop.

If ***htop*** reports that your program is taking 75% CPU on a certain processor, it means that 75% of the samples taken by top found your process active on the CPU. The rest of the time your application was in a wait. (It is important to remember that a CPU is a discrete state machine. It really can be at only 100%, executing an instruction, or at 0%, waiting for something to do. There is no such thing as using 45% of a CPU. The CPU percentage is a function of time.) However, it is likely that your application's rest periods include waiting to be dispatched on a CPU and not on external devices. That part of the wait percentage is then very relevant to understanding your overall CPU usage pattern.

11.3.3 Fine-tuning your executable and/or job script

It is good practice to perform a number of run time stress tests, and to check the CPU utilisation of your nodes. We (and all other users of the HPC) would appreciate that you use the maximum of the CPU resources that are assigned to you and make sure that there are no CPUs in your node who are not utilised without reasons.

But how can you maximise?

1. Configure your software. (e.g., to exactly use the available amount of processors in a node)

2. Develop your parallel program in a smart way.

3. Demand a specific type of compute node (e.g., Harpertown, Westmere), which have a specific number of cores.

4. Correct your request for CPUs in your job script.

11.4 The system load

On top of the CPU utilisation, it is also important to check the **system load**. The system **load** is a measure of the amount of computational work that a computer system performs.

The system load is the number of applications running or waiting to run on the compute node. In a system with for example four CPUs, a load average of 3.61 would indicate that there were, on average, 3.61 processes ready to run, and each one could be scheduled into a CPU.

The load averages differ from CPU percentage in two significant ways:

1. "*load averages*" measure the trend of processes waiting to be run (and not only an instantaneous snapshot, as does CPU percentage); and
2. "*load averages*" include all demand for all resources, e.g. CPU and also I/O and network (and not only how much was active at the time of measurement).

11.4.1 Optimal load

What is the "*optimal load*" rule of thumb?

The load averages tell us whether our physical CPUs are over- or under-utilised. The **point of perfect utilisation**, meaning that the CPUs are always busy and, yet, no process ever waits for one, is **the average matching the number of CPUs**. Your load should not exceed the number of cores available. E.g., if there are four CPUs on a machine and the reported one-minute load average is 4.00, the machine has been utilising its processors perfectly for the last 60 seconds. The "100% utilisation" mark is 1.0 on a single-core system, 2.0 on a dual-core, 4.0 on a quad-core, etc. The optimal load shall be between 0.7 and 1.0 per processor.

In general, the intuitive idea of load averages is the higher they rise above the number of processors, the more processes are waiting and doing nothing, and the lower they fall below the number of processors, the more untapped CPU capacity there is.

Load averages do include any processes or threads waiting on I/O, networking, databases or anything else not demanding the CPU. This means that the optimal *number of applications* running on a system at the same time, might be more than one per processor.

The "**optimal number of applications**" running on one machine at the same time depends on the type of the applications that you are running.

1. When you are running **computational intensive applications**, one application per processor will generate the optimal load.
2. For **I/O intensive applications** (e.g., applications which perform a lot of disk-I/O), a higher number of applications can generate the optimal load. While some applications are reading or writing data on disks, the processors can serve other applications.

The optimal number of applications on a machine could be empirically calculated by performing a number of stress tests, whilst checking the highest throughput. There is however no manner in the HPC at the moment to specify the maximum number of applications that shall run per core dynamically. The HPC scheduler will not launch more than one process per core.

The manner how the cores are spread out over CPUs does not matter for what regards the load. Two quad-cores perform similar to four dual-cores, and again perform similar to eight single-cores. It's all eight cores for these purposes.

11.4.2 Monitoring the load

The **load average** represents the average system load over a period of time. It conventionally appears in the form of three numbers, which represent the system load during the last **one**-, **five**-, and **fifteen**-minute periods.

The **uptime** command will show us the average load

```
$ uptime
10:14:05 up 86 days, 12:01, 11 users, load average: 0.60, 0.41, 0.41
```

Now, start a few instances of the "*eat_ cpu*" program in the background, and check the effect on the load again:

```
$ ./eat_cpu&
$ ./eat_cpu&
$ ./eat_cpu&
$ uptime
10:14:42 up 86 days, 12:02, 11 users, load average: 2.60, 0.93, 0.58
```

You can also read it in the **htop** command.

11.4.3 Fine-tuning your executable and/or job script

It is good practice to perform a number of run time stress tests, and to check the system load of your nodes. We (and all other users of the HPC) would appreciate that you use the maximum of the CPU resources that are assigned to you and make sure that there are no CPUs in your node who are not utilised without reasons.

But how can you maximise?

1. Profile your software to improve its performance.
2. Configure your software (e.g., to exactly use the available amount of processors in a node).
3. Develop your parallel program in a smart way, so that it fully utilises the available processors.
4. Demand a specific type of compute node (e.g., Harpertown, Westmere), which have a specific number of cores.
5. Correct your request for CPUs in your job script.

And then check again.

11.5 Checking File sizes & Disk I/O

11.5.1 Monitoring File sizes during execution

Some programs generate intermediate or output files, the size of which may also be a useful metric.

Remember that your available disk space on the HPC online storage is limited, and that you have environment variables which point to these directories available (i.e., *$VSC_DATA*, *$VSC_SCRATCH* and *$VSC_DATA*). On top of those, you can also access some temporary storage (i.e., the /tmp directory) on the compute node, which is defined by the *$VSC_SCRATCH_NODE* environment variable.

It is important to be aware of the sizes of the file that will be generated, as the available disk space for each user is limited. We refer to section 6.5 on "Quotas" to check your quota and tools to find which files consumed the "quota".

Several actions can be taken, to avoid storage problems:

1. Be aware of all the files that are generated by your program. Also check out the hidden files.
2. Check your quota consumption regularly.
3. Clean up your files regularly.
4. First work (i.e., read and write) with your big files in the local /tmp directory. Once finished, you can move your files once to the VSC_DATA directories.
5. Make sure your programs clean up their temporary files after execution.
6. Move your output results to your own computer regularly.
7. Anyone can request more disk space to the HPC staff, but you will have to duly justify your request.

11.6 Specifying network requirements

Users can examine their network activities with the htop command. When your processors are 100% busy, but you see a lot of red bars and only limited green bars in the htop screen, it is mostly an indication that they loose a lot of time with inter-process communication.

Whenever your application utilises a lot of inter-process communication (as is the case in most parallel programs), we strongly recommend to request nodes with an "Infiniband" network. The Infiniband is a specialised high bandwidth, low latency network that enables large parallel jobs to run as efficiently as possible.

The parameter to add in your job script would be:

```
#PBS -l ib
```

If for some other reasons, a user is fine with the gigabit Ethernet network, he can specify:

```
#PBS -l gbe
```

Chapter 12

Multi-job submission

A frequent occurring characteristic of scientific computation is their focus on data intensive processing. A typical example is the iterative evaluation of a program over different input parameter values, often referred to as a "*parameter sweep*". A **Parameter Sweep** runs a job a specified number of times, as if we sweep the parameter values through a user defined range.

Users then often want to submit a large numbers of jobs based on the same job script but with (i) slightly different parameters settings or with (ii) different input files.

These parameter values can have many forms, we can think about a range (e.g., from 1 to 100), or the parameters can be stored line by line in a comma-separated file. The users want to run their job once for each instance of the parameter values.

One option could be to launch a lot of separate individual small jobs (one for each parameter) on the cluster, but this is not a good idea. The cluster scheduler isn't meant to deal with tons of small jobs. Those huge amounts of small jobs will create a lot of overhead, and can slow down the whole cluster. It would be better to bundle those jobs in larger sets. In TORQUE, an experimental feature known as "*job arrays*" existed to allow the creation of multiple jobs with one *qsub* command, but is was not supported by Moab, the current scheduler.

The "**Worker framework**" has been developed to address this issue.

It can handle many small jobs determined by:

parameter variations i.e., many small jobs determined by a specific parameter set which is stored in a .csv (comma separated value) input file.

job arrays i.e., each individual job got a unique numeric identifier.

Both use cases often have a common root: the user wants to run a program with a large number of parameter settings, and the program does not allow for aggregation, i.e., it has to be run once for each instance of the parameter values.

However, the Worker Framework's scope is wider: it can be used for any scenario that can be reduced to a **MapReduce** approach.[1]

[1] MapReduce: 'Map' refers to the map pattern in which every item in a collection is mapped onto a new value by applying a given function, while "reduce" refers to the reduction pattern which condenses or reduces a collection of previously computed results to a single value.

12.1 The worker Framework: Parameter Sweeps

First go to the right directory:

```
$ cd ~/examples/Multi-job-submission/par_sweep
```

Suppose the program the user wishes to run the "*weather*" program, which takes three parameters: a temperature, a pressure and a volume. A typical call of the program looks like:

```
$ ./weather -t 20 -p 1.05 -v 4.3
T: 20  P: 1.05  V: 4.3
```

For the purpose of this exercise, the weather program is just a simple bash script, which prints the 3 variables to the standard output and waits a bit:

weather– par_sweep/weather —

```
1 #!/bin/bash
2 # Here you could do your calculations
3 echo "T: $2  P: $4  V: $6"
4 sleep 100
```

A job script that would run this as a job for the first parameters (p01) would then look like:

weather_p01.pbs– par_sweep/weather_p01.pbs —

```
1 #!/bin/bash
2
3 #PBS -l nodes=1:ppn=8
4 #PBS -l walltime=01:00:00
5
6 cd $PBS_O_WORKDIR
7 ./weather -t 20 -p 1.05 -v 4.3
```

When submitting this job, the calculation is performed or this particular instance of the parameters, i.e., temperature = 20, pressure = 1.05, and volume = 4.3.

To submit the job, the user would use:

```
$ qsub weather_p01.pbs
```

However, the user wants to run this program for many parameter instances, e.g., he wants to run the program on 100 instances of temperature, pressure and volume. The 100 parameter instances can be stored in a comma separated value file (.csv) that can be generated using a spreadsheet program such as Microsoft Excel or RDBMS or just by hand using any text editor (do **not** use a word processor such as Microsoft Word). The first few lines of the file "*data.csv*" would look like:

```
$ more data.csv
temperature, pressure, volume
293, 1.0e5, 107
294, 1.0e5, 106
295, 1.0e5, 105
296, 1.0e5, 104
297, 1.0e5, 103
...
```

It has to contain the names of the variables on the first line, followed by 100 parameter instances in the current example.

In order to make our PBS generic, the PBS file can be modified as follows:

weather.pbs– par_sweep/weather.pbs —

```
1  #!/bin/bash
2
3  #PBS -l nodes=1:ppn=8
4  #PBS -l walltime=04:00:00
5
6  cd $PBS_O_WORKDIR
7  ./weather -t $temperature -p $pressure -v $volume
8
9  # # This script is submitted to the cluster with the following 2 commands:
10 # module load worker/1.6.8-intel-2018a
11 # wsub -data data.csv -batch weather.pbs
```

Note that:

1. the parameter values 20, 1.05, 4.3 have been replaced by variables $temperature, $pressure and $volume respectively, which were being specified on the first line of the "*data.csv*" file;
2. the number of processors per node has been increased to 8 (i.e., ppn=1 is replaced by ppn=8);
3. the walltime has been increased to 4 hours (i.e., walltime=00:15:00 is replaced by walltime=04:00:00).

The walltime is calculated as follows: one calculation takes 15 minutes, so 100 calculations take 1500 minutes on one CPU. However, this job will use 8 CPUs, so the 100 calculations will be done in 1500/8 = 187.5 minutes, i.e., 4 hours to be on the safe side.

The job can now be submitted as follows (to check which `worker` module to use, see subsection 4.1.7):

```
$ module load worker/1.6.8-intel-2018a
$ wsub -batch weather.pbs -data data.csv
total number of work items: 41
123456.master15.delcatty.gent.vsc
```

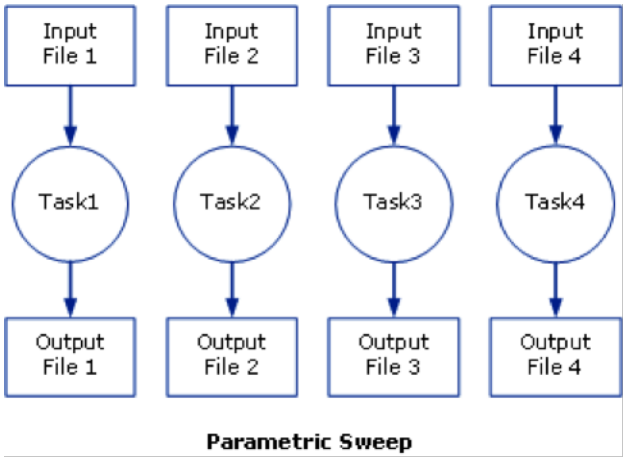
Note that the PBS file is the value of the -batch option. The weather program will now be run for all 100 parameter instances – 8 concurrently – until all computations are done. A computation for such a parameter instance is called a work item in Worker parlance.

12.2 The Worker framework: Job arrays

First go to the right directory:

```
$ cd ~/examples/Multi-job-submission/job_array
```

As a simple example, assume you have a serial program called *myprog* that you want to run on various input files *input[1-100]*.



The following bash script would submit these jobs all one by one:

```
1 #!/bin/bash
2 for i in `seq 1 100`; do
3    qsub -o output $i -i input $i myprog.pbs
4 done
```

This, as said before, could be disturbing for the job scheduler.

Alternatively, TORQUE provides a feature known as *job arrays* which allows the creation of multiple, similar jobs with only **one qsub** command. This feature introduced a new job naming convention that allows users either to reference the entire set of jobs as a unit or to reference one particular job from the set.

Under TORQUE, the *-t range* option is used with qsub to specify a job array, where *range* is a range of numbers (e.g., *1-100* or *2,4-5,7*).

The details are

1. a job is submitted for each *number* in the range;
2. individuals jobs are referenced as *jobid-number*, and the entire array can be referenced as *jobid* for easy killing etc.; and
3. each jobs has *PBS_ARRAYID* set to its *number* which allows the script/program to specialise for that job

The job could have been submitted using:

```
$ qsub -t 1-100 my_prog.pbs
```

The effect was that rather than 1 job, the user would actually submit 100 jobs to the queue system. This was a popular feature of TORQUE, but as this technique puts quite a burden on the scheduler, it is not supported by Moab (the current job scheduler).

To support those users who used the feature and since it offers a convenient workflow, the "worker framework" implements the idea of "job arrays" in its own way.

A typical job script for use with job arrays would look like this:

job_array.pbs– job_array/job_array.pbs —

```
1 #!/bin/bash -l
2 #PBS -l nodes=1:ppn=1
3 #PBS -l walltime=00:15:00
4 cd $PBS_O_WORKDIR
5 INPUT_FILE="input_${PBS_ARRAYID}.dat"
6 OUTPUT_FILE="output_${PBS_ARRAYID}.dat"
7 my_prog -input ${INPUT_FILE}  -output ${OUTPUT_FILE}
```

In our specific example, we have prefabricated 100 input files in the "./input" subdirectory. Each of those files contains a number of parameters for the "test_set" program, which will perform some tests with those parameters.

Input for the program is stored in files with names such as input_1.dat, input_2.dat, ..., input_100.dat in the ./input subdirectory.

```
$ ls ./input
...
$ more ./input/input_99.dat
This is input file \#99
Parameter #1 = 99
Parameter #2 = 25.67
Parameter #3 = Batch
Parameter #4 = 0x562867
```

For the sole purpose of this exercise, we have provided a short "test_set" program, which reads the "input" files and just copies them into a corresponding output file. We even add a few lines to each output file. The corresponding output computed by our "*test_set*" program will be written to the "*./output*" directory in output_1.dat, output_2.dat, ..., output_100.dat. files.

test_set– job_array/test_set —

```
#!/bin/bash

# Check if the output Directory exists
if [ ! -d "./output" ] ; then
  mkdir ./output
fi

#   Here you could do your calculations...
echo "This is Job_array #" $1
echo "Input File : " $3
echo "Output File: " $5
cat ./input/$3 | sed -e "s/input/output/g" | grep -v "Parameter" > ./output/$5
echo "Calculations done, no results" >> ./output/$5
```

Using the "worker framework", a feature akin to job arrays can be used with minimal modifications to the job script:

test_set.pbs– job_array/test_set.pbs —

```
#!/bin/bash -l
#PBS -l nodes=1:ppn=8
#PBS -l walltime=04:00:00
cd $PBS_O_WORKDIR
INPUT_FILE="input_${PBS_ARRAYID}.dat"
OUTPUT_FILE="output_${PBS_ARRAYID}.dat"
./test_set ${PBS_ARRAYID} -input ${INPUT_FILE}  -output ${OUTPUT_FILE}
```

Note that

1. the number of CPUs is increased to 8 (ppn=1 is replaced by ppn=8); and

2. the walltime has been modified (walltime=00:15:00 is replaced by walltime=04:00:00).

The job is now submitted as follows:

```
$ module load worker/1.6.8-intel-2018a
$ wsub -t 1-100 -batch test_set.pbs
total number of work items: 100
123456.master15.delcatty.gent.vsc
```

The "*test_set*" program will now be run for all 100 input files – 8 concurrently – until all computations are done. Again, a computation for an individual input file, or, equivalently, an array id, is called a work item in Worker speak.

Note that in contrast to TORQUE job arrays, a worker job array only submits a single job.

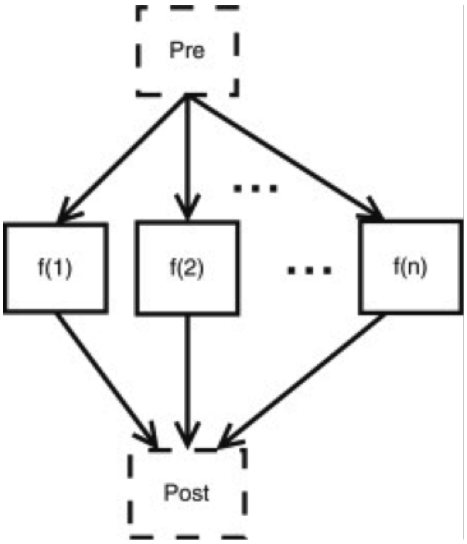
```
$ qstat
Job id          Name          User      Time   Use S Queue
--------------- ------------- --------- ---- ----- - -----
123456.master15.delcatty.gent.vsc  test_set.pbs  vsc40000          0 Q

And you can now check the generated output files:
$ more ./output/output_99.dat
This is output file #99
Calculations done, no results
```

12.3 MapReduce: prologues and epilogue

Often, an embarrassingly parallel computation can be abstracted to three simple steps:

1. a preparation phase in which the data is split up into smaller, more manageable chunks;
2. on these chunks, the same algorithm is applied independently (these are the work items); and
3. the results of the computations on those chunks are aggregated into, e.g., a statistical description of some sort.



The Worker framework directly supports this scenario by using a prologue (pre-processing) and an epilogue (post-processing). The former is executed just once before work is started on the work items, the latter is executed just once after the work on all work items has finished. Technically, the master, i.e., the process that is responsible for dispatching work and logging progress, executes the prologue and epilogue.

```
$ cd ~/examples/Multi-job-submission/map_reduce
```

The script "pre.sh" prepares the data by creating 100 different input-files, and the script "post.sh" aggregates (concatenates) the data.

First study the scripts:

pre.sh– map_reduce/pre.sh —

```
#!/bin/bash

# Check if the input Directory exists
if [ ! -d "./input" ] ; then
  mkdir ./input
fi

# Just generate all dummy input files
for i in {1..100};
do
  echo "This is input file #$i" >  ./input/input_$i.dat
  echo "Parameter #1 = $i" >>  ./input/input_$i.dat
  echo "Parameter #2 = 25.67" >>  ./input/input_$i.dat
  echo "Parameter #3 = Batch" >>  ./input/input_$i.dat
  echo "Parameter #4 = 0x562867" >>  ./input/input_$i.dat
done
```

post.sh– map_reduce/post.sh —

```
#!/bin/bash

# Check if the input Directory exists
if [ ! -d "./output" ] ; then
  echo "The output directory does not exist!"
  exit
fi

# Just concatenate all output files
touch all_output.txt
for i in {1..100};
do
  cat ./output/output_$i.dat >> all_output.txt
done
```

Then one can submit a MapReduce style job as follows:

```
$ wsub -prolog pre.sh -batch test_set.pbs -epilog post.sh -t 1-100
total number of work items: 100
123456.master15.delcatty.gent.vsc
$ cat all_output.txt
...
$ rm -r -f ./output/
```

Note that the time taken for executing the prologue and the epilogue should be added to the job's total walltime.

12.4 Some more on the Worker Framework

12.4.1 Using Worker efficiently

The "Worker Framework" is implemented using MPI, so it is not restricted to a single compute nodes, it scales well to multiple nodes. However, remember that jobs requesting a large number of nodes typically spend quite some time in the queue.

The "Worker Framework" will be effective when

1. work items, i.e., individual computations, are neither too short, nor too long (i.e., from a few minutes to a few hours); and,
2. when the number of work items is larger than the number of CPUs involved in the job (e.g., more than 30 for 8 CPUs).

12.4.2 Monitoring a worker job

Since a Worker job will typically run for several hours, it may be reassuring to monitor its progress. Worker keeps a log of its activity in the directory where the job was submitted. The log's name is derived from the job's name and the job's ID, i.e., it has the form `<jobname>.log<jobid>`. For the running example, this could be `run.pbs.log123456`, assuming the job's ID is 123456. To keep an eye on the progress, one can use:

```
$ tail -f run.pbs.log123456
```

Alternatively, `wsummarize`, a Worker command that summarises a log file, can be used:

```
$ watch -n 60 wsummarize run.pbs.log123456
```

This will summarise the log file every 60 seconds.

12.4.3 Time limits for work items

Sometimes, the execution of a work item takes long than expected, or worse, some work items get stuck in an infinite loop. This situation is unfortunate, since it implies that work items that could successfully execute are not even started. Again, the Worker framework offers a simple and yet versatile solution. If we want to limit the execution of each work item to at most 20 minutes, this can be accomplished by modifying the script of the running example.

```
1 #!/bin/bash -l
2 #PBS -l nodes=1:ppn=8
3 #PBS -l walltime=04:00:00
4 module load timedrun/1.0
5 cd $PBS_O_WORKDIR
6 timedrun -t 00:20:00 weather -t $temperature  -p $pressure  -v $volume
```

Note that it is trivial to set individual time constraints for work items by introducing a parameter, and including the values of the latter in the CSV file, along with those for the temperature, pressure and volume.

Also note that "timedrun" is in fact offered in a module of its own, so it can be used outside the Worker framework as well.

12.4.4 Resuming a Worker job

Unfortunately, it is not always easy to estimate the walltime for a job, and consequently, sometimes the latter is underestimated. When using the Worker framework, this implies that not all work items will have been processed. Worker makes it very easy to resume such a job without having to figure out which work items did complete successfully, and which remain to be computed. Suppose the job that did not complete all its work items had ID "445948".

```
$ wresume -jobid 123456
```

This will submit a new job that will start to work on the work items that were not done yet. Note that it is possible to change almost all job parameters when resuming, specifically the requested resources such as the number of cores and the walltime.

```
$ wresume -l walltime=1:30:00 -jobid 123456
```

Work items may fail to complete successfully for a variety of reasons, e.g., a data file that is missing, a (minor) programming error, etc. Upon resuming a job, the work items that failed are considered to be done, so resuming a job will only execute work items that did not terminate either successfully, or reporting a failure. It is also possible to retry work items that failed (preferably after the glitch why they failed was fixed).

```
$ wresume -jobid 123456 -retry
```

By default, a job's prologue is not executed when it is resumed, while its epilogue is. "wresume" has options to modify this default behaviour.

12.4.5 Further information

This how-to introduces only Worker's basic features. The wsub command has some usage information that is printed when the -help option is specified:

```
$ wsub -help
### usage: wsub  -batch <batch-file>          \
#                [-data <data-files>]        \
#                [-prolog <prolog-file>]     \
#                [-epilog <epilog-file>]     \
#                [-log <log-file>]           \
#                [-mpiverbose]               \
#                [-dryrun] [-verbose]        \
#                [-quiet] [-help]            \
#                [-t <array-req>]            \
#                [<pbs-qsub-options>]
#
#   -batch <batch-file>   : batch file template, containing variables to be
#                           replaced with data from the data file(s) or the
#                           PBS array request option
#   -data <data-files>    : comma-separated list of data files (default CSV
#                           files) used to provide the data for the work
#                           items
#   -prolog <prolog-file> : prolog script to be executed before any of the
#                           work items are executed
#   -epilog <epilog-file> : epilog script to be executed after all the work
#                           items are executed
#   -mpiverbose           : pass verbose flag to the underlying MPI program
#   -verbose              : feedback information is written to standard error
#   -dryrun               : run without actually submitting the job, useful
#   -quiet                : don't show information
#   -help                 : print this help message
#   -t <array-req>        : qsub's PBS array request options, e.g., 1-10
#   <pbs-qsub-options>    : options passed on to the queue submission
#                           command
```

Chapter 13

Compiling and testing your software on the HPC

All nodes in the HPC cluster are running the "CentOS 7.5 (delcatty, golett, phanpy, skitty, swalot, victini)" Operating system, which is a specific version of Red Hat Enterprise Linux. This means that all the software programs (executable) that the end-user wants to run on the HPC first must be compiled for CentOS 7.5 (delcatty, golett, phanpy, skitty, swalot, victini). It also means that you first have to install all the required external software packages on the HPC.

Most commonly used compilers are already pre-installed on the HPC and can be used straight away. Also many popular external software packages, which are regularly used in the scientific community, are also pre-installed.

13.1 Check the pre-installed software on the HPC

In order to check all the available modules and their version numbers, which are pre-installed on the HPC enter:

```
$ module av 2>&1 | more
--- /apps/gent/SL6/sandybridge/modules/all ---
ABAQUS/6.12.1-linux-x86_64
AMOS/3.1.0-ictce-4.0.10
ant/1.9.0-Java-1.7.0_40
ASE/3.6.0.2515-ictce-4.1.13-Python-2.7.3
ASE/3.6.0.2515-ictce-5.5.0-Python-2.7.6
...
```

Or when you want to check whether some specific software, some compiler or some application (e.g., Matlab) is installed on the HPC.

```
$ module av 2>&1 | grep -i -e "matlab"
MATLAB/2010b
MATLAB/2012b
MATLAB/2013b
```

As you are not aware of the capitals letters in the module name, we looked for a case-insensitive name with the "-i" option.

When your required application is not available on the HPC please contact any HPC member. Be aware of potential "License Costs". "Open Source" software is often preferred.

13.2 Porting your code

To **port** a software-program is to translate it from the operating system in which it was developed (e.g., Windows 7) to another operating system (e.g., Red Hat Enterprise Linux on our HPC) so that it can be used there. Porting implies some degree of effort, but not nearly as much as redeveloping the program in the new environment. It all depends on how "portable" you wrote your code.

In the simplest case the file or files may simply be copied from one machine to the other. However, in many cases the software is installed on a computer in a way, which depends upon its detailed hardware, software, and setup, with device drivers for particular devices, using installed operating system and supporting software components, and using different directories.

In some cases software, usually described as "portable software" is specifically designed to run on different computers with compatible operating systems and processors without any machine-dependent installation; it is sufficient to transfer specified directories and their contents. Hardware- and software-specific information is often stored in configuration files in specified locations (e.g., the registry on machines running MS Windows).

Software, which is not portable in this sense, will have to be transferred with modifications to support the environment on the destination machine.

Whilst programming, it would be wise to stick to certain standards (e.g., ISO/ANSI/POSIX). This will ease the porting of your code to other platforms.

Porting your code to the CentOS 7.5 (delcatty, golett, phanpy, skitty, swalot, victini) platform is the responsibility of the end-user.

13.3 Compiling and building on the HPC

Compiling refers to the process of translating code written in some programming language, e.g., Fortran, C, or C++, to machine code. Building is similar, but includes gluing together the machine code resulting from different source files into an executable (or library). The text below guides you through some basic problems typical for small software projects. For larger projects it is more appropriate to use makefiles or even an advanced build system like CMake.

All the HPC nodes run the same version of the Operating System, i.e. CentOS 7.5 (delcatty, golett, phanpy, skitty, swalot, victini). So, it is sufficient to compile your program on any compute node. Once you have generated an executable with your compiler, this executable should be able to run on any other compute-node.

A typical process looks like:

1. Copy your software to the login-node of the HPC
2. Start an interactive session on a compute node;

3. Compile it;
4. Test it locally;
5. Generate your job scripts;
6. Test it on the HPC
7. Run it (in parallel);

We assume you've copied your software to the HPC The next step is to request your private compute node.

```
$ qsub -I
qsub: waiting for job 123456.master15.delcatty.gent.vsc to start
```

13.3.1 Compiling a sequential program in C

Go to the examples for chapter 13 and load the foss module:

```
$ cd ~/examples/Compiling-and-testing-your-software-on-the-HPC
$ module load foss
```

We now list the directory and explore the contents of the "*hello.c*" program:

```
$ ls -l
total 512
-rw-r--r-- 1 vsc40000 214 Sep 16 09:42 hello.c
-rw-r--r-- 1 vsc40000 130 Sep 16 11:39 hello.pbs*
-rw-r--r-- 1 vsc40000 359 Sep 16 13:55 mpihello.c
-rw-r--r-- 1 vsc40000 304 Sep 16 13:55 mpihello.pbs
```

— hello.c —

```
1  /*
2   * VSC        : Flemish Supercomputing Centre
3   * Tutorial   : Introduction to HPC
4   * Description: Print 500 numbers, whilst waiting 1 second in between
5   */
6  #include "stdio.h"
7  int main( int argc, char *argv[] )
8  {
9    int i;
10   for (i=0; i<500; i++)
11   {
12     printf("Hello #%d\n", i);
13     fflush(stdout);
14     sleep(1);
15   }
16 }
```

The "hello.c" program is a simple source file, written in C. It'll print 500 times "Hello #<num>", and waits one second between 2 printouts.

We first need to compile this C-file into an executable with the gcc-compiler.

First, check the command line options for *"gcc" (GNU C-Compiler)*, then we compile and list the contents of the directory again:

```
$ gcc -help
$ gcc -o hello hello.c
$ ls -l
total 512
-rwxrwxr-x 1 vsc40000 7116 Sep 16 11:43 hello*
-rw-r--r-- 1 vsc40000  214 Sep 16 09:42 hello.c
-rwxr-xr-x 1 vsc40000  130 Sep 16 11:39 hello.pbs*
```

A new file "hello" has been created. Note that this file has "execute" rights, i.e., it is an executable. More often than not, calling gcc – or any other compiler for that matter – will provide you with a list of errors and warnings referring to mistakes the programmer made, such as typos, syntax errors. You will have to correct them first in order to make the code compile. Warnings pinpoint less crucial issues that may relate to performance problems, using unsafe or obsolete language features, etc. It is good practice to remove all warnings from a compilation process, even if they seem unimportant so that a code change that produces a warning does not go unnoticed.

Let's test this program on the local compute node, which is at your disposal after the "qsub –I" command:

```
$ ./hello
Hello #0
Hello #1
Hello #2
Hello #3
Hello #4
...
```

It seems to work, now run it on the HPC

```
$ qsub hello.pbs
```

13.3.2 Compiling a parallel program in C/MPI

```
$ cd ~/examples/Compiling-and-testing-your-software-on-the-HPC
```

List the directory and explore the contents of the "*mpihello.c*" program:

```
$ ls -l
total 512
total 512
-rw-r--r-- 1 vsc40000 214 Sep 16 09:42 hello.c
-rw-r--r-- 1 vsc40000 130 Sep 16 11:39 hello.pbs*
-rw-r--r-- 1 vsc40000 359 Sep 16 13:55 mpihello.c
-rw-r--r-- 1 vsc40000 304 Sep 16 13:55 mpihello.pbs
```

— mpihello.c —

```c
/*
 * VSC        : Flemish Supercomputing Centre
 * Tutorial   : Introduction to HPC
 * Description: Example program, to compile with MPI
 */
#include <stdio.h>
#include <mpi.h>

main(int argc, char **argv)
{
  int node, i, j;
  float f;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &node);

  printf("Hello World from Node %d.\n", node);
  for (i=0; i<=100000; i++)
    f=i*2.718281828*i+i+i*3.141592654;

  MPI_Finalize();
}
```

The "mpi_hello.c" program is a simple source file, written in C with MPI library calls.

Then, check the command line options for *"mpicc" (GNU C-Compiler with MPI extensions)*, then we compile and list the contents of the directory again:

```
$ mpicc -help
$ mpicc -o mpihello mpihello.c
$ ls -l
```

A new file "hello" has been created. Note that this program has "execute" rights.

Let's test this program on the "login"-node first:

```
$ ./mpihello
Hello World from Node 0.
```

It seems to work, now run it on the HPC.

```
$ qsub mpihello.pbs
```

13.3.3 Compiling a parallel program in Intel Parallel Studio Cluster Edition

We will now compile the same program, but using the Intel Parallel Studio Cluster Edition compilers. We stay in the examples directory for this chapter:

```
$ cd ~/examples/Compiling-and-testing-your-software-on-the-HPC
```

We will compile this C/MPI -file into an executable with the Intel Parallel Studio Cluster Edition. First, clear the modules (purge) and then load the latest "intel" module:

```
$ module purge
$ module load intel
```

Then, compile and list the contents of the directory again. The Intel equivalent of mpicc is mpiicc.

```
$ mpiicc -o mpihello mpihello.c
$ ls -l
```

Note that the old "mpihello" file has been overwritten. Let's test this program on the "login"-node first:

```
$ ./mpihello
Hello World from Node 0.
```

It seems to work, now run it on the HPC.

```
$ qsub mpihello.pbs
```

Note: The AUGent only has a license for the Intel Parallel Studio Cluster Edition for a fixed number of users. As such, it might happen that you have to wait a few minutes before a floating license becomes available for your use.

Note: The Intel Parallel Studio Cluster Edition contains equivalent compilers for all GNU compilers. Hereafter the overview for C, C++ and Fortran compilers.

	Sequential Program		**Parallel Program (with MPI)**	
	GNU	**Intel**	**GNU**	**Intel**
C	gcc	icc	mpicc	mpiicc
C++	g++	icpc	mpicxx	mpiicpc
Fortran	gfortran	ifort	mpif90	mpiifort

Chapter 14

Checkpointing

14.1 Why checkpointing?

If you want to run jobs that require wall time than the maximum wall time per job and/or want to avoid you lose work because of power outages or system crashes, you need to resort to checkpointing.

14.2 What is checkpointing?

Checkpointing allows for running jobs that run for weeks or months, by splitting the job into smaller parts (called subjobs) which are executed consecutively. Each time a subjob is running out of requested wall time, a snapshot of the application memory (and much more) is taken and stored, after which a subsequent subjob will pick up the checkpoint and continue.

14.3 How to use checkpointing?

Using checkpointing is very simple: just use `csub` instead of `qsub` to submit a job.

The `csub` command creates a wrapper around your job script, to take care of all the checkpointing stuff.

In practice, you (usually) don't need to adjust anything, except for the command used to submit your job.

Checkpointing does not require any changes to the application you are running, and should support most software.

14.4 Usage and parameters

An overview of the usage and various command line parameters is given here.

14.4.1 Submitting a job

Typically, a job script is submitted with checkpointing support enabled by running:

```
csub -s job_script.sh
```

The `-s` flag specifies the job script to run.

14.4.2 Caveat: don't create local directories

One important caveat is that the job script (or the applications run in the script) should *not* create its own local temporary directories, because those will not (always) be restored when the job is restarted from checkpoint.

14.4.3 PBS directives

Most PBS directives (`#PBS ...` specified in the job script will be ignored. There are a few exceptions however, i.e. `# PBS -N <name>` (job name) and all `-l` directives (`# PBS -l`), e.g. `nodes`, `ppn`, `vmem` (virtual memory limit), etc. Controlling other job parameters (like requested walltime per sub-job) should be specified on the `csub` command line.

14.4.4 Getting help

Help on the various command line parameters supported by `csub` can be obtained using `-h` or `--help`.

14.4.5 Local files (`-pre` / `-post`)

The `--pre` and `--post` parameters control whether local files are copied or not. The job submitted using `csub` is (by default) run on the local storage provided by a particular workernode. Thus, no changes will be made to the files on the shared storage.

If the job script needs (local) access to the files of the directory where `csub` is executed, `--pre` flag should be used. This will copy all the files in the job script directory to the location where the job script will execute.

If the output of the job (`stdout`/`stderr`) that was run, or additional output files created by the job in its working directory are required, the `--post` flag should be used. This will copy the entire job working directory to the location where `csub` was executed, in a directory named `result.<jobname>`. An alternative is to copy the interesting files to the shared storage at the end of the job script.

14.4.6 Running on shared storage (`-shared`)

If the job needs to be run on the shared storage, `--shared` should be specified. You should enable this option by default, because it makes the execution of the underlying `csub` script more robust: it doesn't have to copy from/to the local storage on the workernode. When enabled, the

job will be run in a subdirectory of `$VSC_SCRATCH/chkpt`. All files produced by the job will be in `$VSC_SCRATCH/chkpt/<jobid>/` while the job is running.

Note that if files in the directory where the job script is located are required to run the job, you should also use `--pre`.

14.4.7 Job wall time (`-job_time`, `-chkpt_time`)

To specify the requested wall time per subjob, use the `--job-time` parameter. The default setting is 10 hours per (sub)job. Lowering this will result in more frequent checkpointing, and thus more (sub)jobs.

To specify the time that is reserved for checkpointing the job, use `--chkpt_time`. By default, this is set to 15 minutes which should be enough for most applications/jobs. *Don't change this unless you really need to.*

The total requested wall time per subjob is the sum of both `job_time` and `chkpt_time`.

If you would like to time how long the job executes, just prepend the main command in your job script with the time command `time`, e.g.:

```
time main_command
```

The `real` time will not make sense, as it will also include the time passed between two checkpointed subjobs. However, the `user` time should give a good indication of the actual time it took to run your command, even if multiple checkpoints were performed.

14.4.8 Resuming from last checkpoint (`-resume`)

The `--resume` option allows you to resume a job from the last available checkpoint in case something went wrong (e.g. accidentally deleting a (sub)job using `qdel`, a power outage or other system failure, ...).

Specify the job name as returned after submission (and as listed in `$VSC_SCRATCH/chkpt`). The full job name consists of the specified job name (or the script name if no job name was specified), a timestamp and two random characters at the end, for example `my_job_name.20180102_133755.Hk` or `script.sh.20180102_133755.Hk`.

Note: When resuming from checkpoint, you can change the wall time resources for your job using the `--job_time` and `--chkpt_time` options. This should allow you to continue from the last checkpoint in case your job crashed due to an excessively long checkpointing time.

In case resuming fails for you, please contact hpc@ugent.be, and include the output of the `csub --resume` command in your message.

14.5 Additional options

14.5.1 Array jobs (`-t`)

`csub` has support for checkpointing array jobs with the `-t <spec>` flag on the `csub` command line. This behaves the same as `qsub`, see section 12.2.

14.5.2 Pro/epilogue mimicing (`-no_mimic_pro_epi`)

The option `--no_mimic_pro_epi` disables the workaround currently required to resolve a permissions problem when using actual Torque prologue/epilogue scripts. Don't use this option unless you really know what you are doing.

14.5.3 Cleanup checkpoints (`-cleanup_after_restart`)

Specifying this option will make the wrapper script remove the checkpoint files after a successful job restart. This may be desirable in cause you are short on storage space.

Note that we don't recommend setting this option, because this way you won't be able to resume from the last checkpoint when something goes wrong. It may also prevent the wrapper script from reattempting to resubmit a new job in case an infrequent known failure occurs. So, don't set this unless you really need to.

14.5.4 No cleanup after job completion (`-no_cleanup_chkpt`)

Specifying this option will prevent the wrapper script from cleaning up the checkpoints and related information once the job has finished. This may be useful for debugging, since this also preserves the `stdout/stderr` of the wrapper script.

Don't set this unless you know what you are doing.

Chapter 15

Program examples

Go to our examples:

```
$ cd ~/examples/Program-examples
```

Here, we just have put together a number of examples for your convenience. We did an effort to put comments inside the source files, so the source code files are (should be) self-explanatory.

1. 01_Python
2. 02_C_C++
3. 03_Matlab
4. 04_MPI_C
5. 05a_OMP_C
6. 05b_OMP_FORTRAN
7. 06_NWChem
8. 07_Wien2k
9. 08_Gaussian
10. 09_Fortran
11. 10_PQS

The above 2 OMP directories contain the following examples:

C Files	Fortran Files	Description
omp_hello.c	omp_hello.f	Hello world
omp_workshare1.c	omp_workshare1.f	Loop work-sharing
omp_workshare2.c	omp_workshare2.f	Sections work-sharing
omp_reduction.c	omp_reduction.f	Combined parallel loop reduction
omp_orphan.c	omp_orphan.f	Orphaned parallel loop reduction
omp_mm.c	omp_mm.f	Matrix multiply
omp_getEnvInfo.c	omp_getEnvInfo.f	Get and print environment information
omp_bug1.c omp_bug1fix.c omp_bug2.c omp_bug3.c omp_bug4.c omp_bug4fix omp_bug5.c omp_bug5fix.c omp_bug6.c	omp_bug1.f omp_bug1fix.f omp_bug2.f omp_bug3.f omp_bug4.f omp_bug4fix omp_bug5.f omp_bug5fix.f omp_bug6.f	Programs with bugs and their solution

Compile by any of the following commands:

C:	icc -openmp omp_hello.c -o hello pgcc -mp omp_hello.c -o hello gcc -fopenmp omp_hello.c -o hello
Fortran:	ifort -openmp omp_hello.f -o hello pgf90 -mp omp_hello.f -o hello gfortran -fopenmp omp_hello.f -o hello

Be invited to explore the examples.

Chapter 16

Job script examples

16.1 Single-core job

Here's an example of a single-core job script:

— single_core.sh —

```
1  #!/bin/bash
2  #PBS -N count_example          ## job name
3  #PBS -l nodes=1:ppn=1          ## single-node job, single core
4  #PBS -l walltime=2:00:00       ## max. 2h of wall time
5  module load Python/3.6.4-intel-2018a
6  # copy input data from location where job was submitted from
7  cp $PBS_O_WORKDIR/input.txt $TMPDIR
8  # go to temporary working directory (on local disk) & run
9  cd $TMPDIR
10 python -c "print(len(open('input.txt').read()))" > output.txt
11 # copy back output data, ensure unique filename using $PBS_JOBID
12 cp output.txt $VSC_DATA/output_${PBS_JOBID}.txt
```

1. Using `#PBS` header lines, we specify the resource requirements for the job, see Appendix B for a list of these options

2. A module for `Python 3.6` is loaded, see also section 4.1

3. We stage the data in: the file `input.txt` is copied into the "working" directory, see chapter 6

4. The main part of the script runs a small Python program that counts the number of characters in the provided input file `input.txt`

5. We stage the results out: the output file `output.txt` is copied from the "working directory" (`$TMPDIR`|) to a unique directory in `$VSC_DATA`. For a list of possible storage locations, see subsection 6.2.1.

16.2 Multi-core job

Here's an example of a multi-core job script that uses `mympirun`:

— multi_core.sh —

```
#!/bin/bash
#PBS -N mpi_hello             ## job name
#PBS -l nodes=2:ppn=all       ## 2 nodes, all cores per node
#PBS -l walltime=2:00:00      ## max. 2h of wall time
module load intel/2017b
module load vsc-mympirun      ## We don't use a version here, this is on purpose
# go to working directory, compile and run MPI hello world
cd $PBS_O_WORKDIR
mpicc mpi_hello.c -o mpi_hello
mympirun ./mpi_hello
```

An example MPI hello world program can be downloaded from https://github.com/hpcugent/vsc-mympirun/blob/master/testscripts/mpi_helloworld.c.

16.3 Running a command with a maximum time limit

If you want to run a job, but you are not sure it will finish before the job runs out of walltime and you want to copy data back before, you have to stop the main command before the walltime runs out and copy the data back.

This can be done with the `timeout` command. This command sets a limit of time a program can run for, and when this limit is exceeded, it kills the program. Here's an example job script using `timeout`:

— timeout.sh —

```
#!/bin/bash
#PBS -N timeout_example
#PBS -l nodes=1:ppn=1         ## single-node job, single core
#PBS -l walltime=2:00:00      ## max. 2h of wall time

# go to temporary working directory (on local disk)
cd $TMPDIR
# This command will take too long (1400 minutes is longer than our walltime)
# $PBS_O_WORKDIR/example_program.sh 1400 output.txt

# So we put it after a timeout command
# We have a total of 120 minutes (2 x 60) and we instruct the script to run for
# 100 minutes, but timeout after 90 minute,
# so we have 30 minutes left to copy files back. This should
#  be more than enough.
timeout -s SIGKILL 90m $PBS_O_WORKDIR/example_program.sh 100 output.txt
# copy back output data, ensure unique filename using $PBS_JOBID
cp output.txt $VSC_DATA/output_${PBS_JOBID}.txt
```

The example program used in this script is a dummy script that simply sleeps a specified amount of minutes:

— example_program.sh —

```
1  #!/bin/bash
2  # This is an example program
3  # It takes two arguments: a number of times to loop and a file to write to
4  # In total, it will run for (the number of times to loop) minutes
5
6  if [ $# -ne 2 ]; then
7      echo "Usage: ./example_program amount filename" && exit 1
8  fi
9
10 for ((i = 0; i < $1; i++ )); do
11     echo "${i} => $(date)" >> $2
12     sleep 60
13 done
```

Chapter 17

Best Practices

17.1 General Best Practices

1. Before starting you should always check:
   - Are there any errors in the script?
   - Are the required modules loaded?
   - Is the correct executable used?
2. Check your computer requirements upfront, and request the correct resources in your batch job script.
   - Number of requested cores
   - Amount of requested memory
   - Requested network type
3. Check your jobs at runtime. You could login to the node and check the proper execution of your jobs with, e.g., `top` or `vmstat`. Alternatively you could run an interactive job (`qsub -I`).
4. Try to benchmark the software for scaling issues when using MPI or for I/O issues.
5. Use the scratch file system (`$VSC_SCRATCH_NODE`, which is mapped to the local /tmp) whenever possible. Local disk I/O is always much faster as it does not have to use the network.
6. When your job starts, it will log on to the compute node(s) and start executing the commands in the job script. It will start in your home directory `$VSC_HOME`, so going to the current directory with `cd $PBS_O_WORKDIR` is the first thing which needs to be done. You will have your default environment, so don't forget to load the software with `module load`.
7. Submit your job and wait (be patient) ...
8. Submit small jobs by grouping them together. See chapter 12 for how this is done.

9. The runtime is limited by the maximum walltime of the queues. For longer walltimes, use checkpointing.
10. Requesting many processors could imply long queue times. It's adivised to only request the resources you'll be able to use.
11. For all multi-node jobs, please use a cluster that has an "Infiniband" interconnect network.
12. And above all, do not hesitate to contact the HPC staff at hpc@ugent.be. We're here to help you.

Chapter 18

Graphical applications with VNC

Virtual Network Computing is a graphical desktop sharing system that enables you to interact with graphical software running on the HPC infrastructure from your own computer.

Please carefully follow the instructions below, since the procedure to connect to a VNC server running on the HPC infrastructure is not trivial, due to security constraints.

18.1 Starting a VNC server

First login on the login node (see section 3.1), then start `vncserver` with:

```
$ vncserver -geometry 1920x1080 -localhost
You will require a password to access your desktops.

Password:<enter a secure password>
Verify:<enter the same password>
Would you like to enter a view-only password (y/n)? n
A view-only password is not used

New 'gligar04.gastly.os:6 (vsc40000)' desktop is gligar04.gastly.os:6

Creating default startup script /user/home/gent/vsc400/vsc40000/.vnc/xstartup
Creating default config /user/home/gent/vsc400/vsc40000/.vnc/config
Starting applications specified in /user/home/gent/vsc400/vsc40000/.vnc/xstartup
Log file is /user/home/gent/vsc400/vsc40000/.vnc/gligar04.gastly.os:6.log
```

When prompted for a password, make sure to enter a secure password: if someone can guess your password, they will be able to do anything with your account you can!

Note down the details in bold: the hostname (in the example: `gligar04.gastly.os`) and the (partial) port number (in the example: `6`).

It's important to remember that VNC sessions are permanent. They survive network problems and (unintended) connection loss. This means you can logout and go home without a problem (like the terminal equivalent `screen` or `tmux`). This also means you don't have to start `vncserver` each time you want to connect.

18.2 List running VNC servers

You can get a list of running VNC servers on a node with

```
$ vncserver -list
TigerVNC server sessions:

X DISPLAY #     PROCESS ID
:6                  30713
```

This only displays the running VNC servers on **the login node you run the command on**.

To see what login nodes you are running a VNC server on, you can run the `ls .vnc/*.pid` command in your home directory: the files shown have the hostname of the login node in the filename:

```
$ cd $HOME
$ ls .vnc/*.pid
.vnc/gligar04.gastly.os:6.pid
.vnc/gligar05.gastly.os:8.pid
```

This shows that there is a VNC server running on `gligar04.gastly.os` on port 5906 and another one running `gligar05.gastly.os` on port 5908 (see also subsection 18.3.1).

18.3 Connecting to a VNC server

The VNC server runs on a **specific login node** (in the example above, on `gligar04.gastly.os`).

In order to access your VNC server, you will need to set up an SSH tunnel from your workstation to this login node (see subsection 18.3.3).

Login nodes are rebooted from time to time. You can check that the VNC server is still running in the same node by executing `vncserver -list` (see also section 18.2). If you get an empty list, it means that there is no VNC server running on the login node.

To set up the SSH tunnel required to connect to your VNC server, you will need to port forward the VNC port to your workstation.

The *host* is `localhost`, which means "your own computer": we set up an SSH tunnel that connects the VNC port on the login node to the same port on your local computer.

18.3.1 Determining the source/destination port

The *destination port* is the port on which the VNC server is running (on the login node), which is **the sum of `5900` and the partial port number** we noted down earlier (`6`); in the running example, that is `5906`.

The *source port* is the port you will be connecting to with your VNC client on your workstation. Although you can use any (free) port for this, we strongly recommend to use the **same value as the destination port**.

So, in our running example, both the source and destination ports are `5906`.

18.3.2 Picking an intermediate port to connect to the right login node

In general, you have no control over which login node you will be on when setting up the SSH tunnel from your workstation to `login.hpc.ugent.be` (see subsection 18.3.3).

If the login node you end up on is a different one than the one where your VNC server is running (i.e., `gligar05.gastly.os` rather than `gligar04.gastly.os` in our running example), you need to create a ***second* SSH tunnel** on the login node you are connected to, in order to "patch through" to the correct port on the login node where your VNC server is running.

In the remainder of these instructions, we will assume that we are indeed connected to a different login node. Following these instructions should always work, even if you happen to be connected to the correct login node.

To set up the second SSH tunnel, you need to **pick an (unused) port on the login node you are connected to**, which will be used as an *intermediate* port.

Now we have a chicken-egg situation: you need to pick a port before setting up the SSH tunnel from your workstation to `gligar04.gastly.os`, but only after starting the SSH tunnel will you be able to determine whether the port you picked is actually free or not...

In practice, if you **pick a** random **number between** 10000 **and** 30000, you have a good chance that the port will not be used yet.

We will proceed with 12345 as intermediate port, but **you should pick another value that other people are not likely to pick**. If you need some inspiration, run the following command on a Linux server (for example on a login node): `echo $RANDOM` (but do not use a value lower than 1025).

18.3.3 Setting up the SSH tunnel(s)

Setting up the first SSH tunnel from your workstation to login.hpc.ugent.be

First, we will set up the SSH tunnel from our workstation to login.hpc.ugent.be.

Use the settings specified in the sections above:

- *source port*: the port on which the VNC server is running (see subsection 18.3.1);
- *destination host*: `localhost`;
- *destination port*: use the intermediate port you picked (see subsection 18.3.2)

See section 5.3.1 for detailed information on how to configure PuTTY to set up the SSH tunnel, by entering the settings in the Source port and Destination fields in Connection > SSH > Tunnels.

With this, we have forwarded port `5906` on our workstation to port `12345` on the login node we are connected to.

Again, do *not* use `12345` as destination port, as this port will most likely be used by somebody else already; replace it with a port number you picked yourself, which is unlikely to be used already (see subsection 18.3.2).

Checking whether the intermediate port is available

Before continuing, it's good to check whether the intermediate port that you have picked is actually still available (see subsection 18.3.2).

You can check using the following command (**do not forget to replace `12345` the value you picked for your intermediate port**):

```
$ netstat -an | grep -i listen | grep tcp | grep 12345
$
```

If you see no matching lines, then the port you picked is still available, and you can continue.

If you see one or more matching lines as shown below, **you must disconnect the first SSH tunnel, pick a different intermedate port, and set up the first SSH tunnel again using the new value**.

```
$ netstat -an | grep -i listen | grep tcp | grep 12345
tcp 0 0 0.0.0.0:12345 0.0.0.0:* LISTEN
tcp6 0 0 :::12345 :::* LISTEN
$
```

Setting up the second SSH tunnel to the correct login node

In the session on the login node you created by setting up an SSH tunnel from your workstation to `login.hpc.ugent.be`, you now need to set up the second SSH tunnel to "patch through" to the login node where your VNC server is running (`gligar04.gastly.os` in our running example, see section 18.1).

To do this, run the following command:

```
$ ssh -L 12345:localhost:5906 gligar04.gastly.os
$ hostname
gligar04.gastly.os
```

With this, we are forwarding port `12345` on the login node we are connected to (which is referred to as `localhost`) through to port `5906` on our target login node (`gligar04.gastly.os`).

Combined with the first SSH tunnel, port `5906` on our workstation is now connected to port `5906` on the login node where oru VNC server is running (via the intermediate port `12345` on the login node we ended up one with the first SSH tunnel).

Do not forget to change the intermediate port (`12345`), destination port (`5906`), and hostname of the login node (`gligar04.gastly.os`) in the command shown above!

As shown above, you can check again using the `hostname` command whether you are indeed connected to the right login node. If so, you can go ahead and connect to your VNC server (see subsection 18.3.4).

18.3.4 Connecting using a VNC client

You can download a free VNC client from https://sourceforge.net/projects/turbovnc/files/. You can download the latest version by clicking the top-most folder that has a version number in it that doesn't also have beta in the version. Then download a file that looks like TurboVNC64-2.1.2.exe (the version number can be different, but the 64 should be in the filename) and execute it.

Now start your VNC client and connect to localhost:5906. **Make sure you replace the port number 5906 with your own destination port** (see subsection 18.3.1).

When prompted for a password, use the password you used to setup the VNC server.

When prompted for default or empty panel, choose default.

If you have an empty panel, you can reset your settings with the following commands:

```
$ xfce4-panel -quit ; pkill xfconfd
$ mkdir ~/.oldxfcesettings
$ mv ~/.config/xfce4 ~/.oldxfcesettings
$ xfce4-panel
```

18.4 Stopping the VNC server

The VNC server can be killed by running

```
vncserver -kill :6
```

where 6 is the port number we noted down earlier. If you forgot, you can get it with vncserver -list (see section 18.2).

18.5 I forgot the password, what now?

You can reset the password by first stopping the VNC server (see section 18.4), then removing the .vnc/passwd file (with rm .vnc/passwd) and then starting the VNC server again (see section 18.1).

Part III

Software-specific Best Practices

Chapter 19

MATLAB

19.1 Why is the MATLAB compiler required?

The main reason behind this alternative way of using MATLAB is licensing: only a limited number of MATLAB sessions can be active at the same time. However, once the MATLAB program is compiled using the MATLAB compiler, the resulting stand-alone executable can be run without needing to contact the license server.

Note that a license is required for the MATLAB Compiler, see https://nl.mathworks.com/help/compiler/index.html. If the `mcc` command is provided by the MATLAB installation you are using, the MATLAB compiler can be used as explained below.

Only a limited amount of MATLAB sessions can be active at the same time because there are only a limited amount of MATLAB research licences available on the UGent MATLAB license server. If each job would need a license, licenses would quickly run out.

19.2 How to compile MATLAB code

Compiling MATLAB code can only be done from the login nodes, because only login nodes can access the MATLAB license server, workernodes on clusters can not.

To access the MATLAB compiler, the `MATLAB` module should be loaded first. Make sure you are using the same `MATLAB` version to compile and to run the compiled MATLAB program.

```
$ module avail MATLAB
----------------------/apps/gent/delcatty/modules/all----------------------
   MATLAB/2016b    MATLAB/2017b    MATLAB/2018a (D)
$ module load MATLAB/2018a
```

After loading the `MATLAB` module, the `mcc` command can be used. To get help on `mcc`, you can run `mcc -?`.

To compile a standalone application, the `-m` flag is used (the `-v` flag means verbose ouput). To show how `mcc` can be used, we use the `magicsquare` example that comes with MATLAB.

First, we copy the `magicsquare.m` example that comes with MATLAB to `example.m`:

```
$ cp $EBROOTMATLAB/extern/examples/compiler/magicsquare.m example.m
```

To compile a MATLAB program, use `mcc -mv`:

```
$ mcc -mv example.m
Opening log file:  /user/home/gent/vsc400/vsc40000/java.log.34090
Compiler version: 6.6 (R2018a)
Dependency analysis by REQUIREMENTS.
Parsing file "/user/home/gent/vsc400/vsc40000/example.m"
        (Referenced from: "Compiler Command Line").
Deleting 0 temporary MEX authorization files.
Generating file "/user/home/gent/vsc400/vsc40000/readme.txt".
Generating file "run\_exampe.sh".
```

19.2.1 Libraries

To compile a MATLAB program that *needs a library*, you can use the `-I library_path` flag. This will tell the compiler to also look for files in *library_path*.

It's also possible to use the `-a path` flag. That will result in all files under the *path* getting added to the final executable.

For example, the command `mcc -mv example.m -I examplelib -a datafiles` will compile `example.m` with the MATLAB files in `examplelib`, and will include all files in the `datafiles` directory in the binary it produces.

19.2.2 Memory issues during compilation

If you are seeing Java memory issues during the compilation of your MATLAB program on the login nodes, consider tweaking the default maximum heap size (128M) of Java using the `_JAVA_OPTIONS` environment variable with:

```
$ export _JAVA_OPTIONS="-Xmx64M"
```

The MATLAB compiler spawns multiple Java processes, and because of the default memory limits that are in effect on the login nodes, this might lead to a crash of the compiler if it's trying to create to many Java processes. If we lower the heap size, more Java processes will be able to fit in memory.

Another possible issue is that the heap size is too small. This could result in errors like:

```
Error: Out of memory
```

A possible solution to this is by setting the maximum heap size to be bigger:

```
$ export _JAVA_OPTIONS="-Xmx512M"
```

19.3 Multithreading

MATLAB can only use the cores in a single workernode (unless the Distributed Computing toolbox is used, see https://nl.mathworks.com/products/distriben.html).

The amount of workers used by MATLAB for the parallel toolbox can be controlled via the `parpool` function: `parpool(16)` will use 16 workers. It's best to specify the amount of workers, because otherwise you might not harness the full compute power available (if you have too few workers), or you might negatively impact performance (if you have too much workers). By default, MATLAB uses a fixed number of workers (12).

You should use a number of workers that is equal to the number of cores you requested when submitting your job script (the `ppn` value, see subsection 4.6.1). You can determine the right number of workers to use via the following code snippet in your MATLAB program:

— parpool.m —

```
1 % specify the right number of workers (as many as there are cores available in the
      job) when creating the parpool
2 c = parcluster('local')
3 pool = parpool(c.NumWorkers)
```

See also the parpool documentation.

19.4 Java output logs

Each time MATLAB is executed, it generates a Java log file in the users home directory. The output log directory can be changed using:

```
$ MATLAB_LOG_DIR=<OUTPUT_DIR>
```

where <OUTPUT_DIR> is the name of the desired output directory. To create and use a temporary directory for these logs:

```
# create unique temporary directory in $TMPDIR (or /tmp/$USER if $TMPDIR is not
    defined)
# instruct MATLAB to use this directory for log files by setting $MATLAB_LOG_DIR
$ export MATLAB_LOG_DIR=$(mktemp -d -p $TMPDIR:-/tmp/$USER)
```

```
$ rm -rf $MATLAB_LOG_DIR
```

19.5 Cache location

When running, MATLAB will use a cache for performance reasons. This location and size of this cache can be changed trough the `MCR_CACHE_ROOT` and `MCR_CACHE_SIZE` environment variables.

The snippet below would set the maximum cache size to 1024MB and the location to `/tmp/`

testdirectory.

```
$ export MATLAB_CACHE_ROOT=/tmp/testdirectory
$ export MATLAB_CACHE_SIZE=1024M
```

So when MATLAB is running, it can fill up to 1024MB of cache in /tmp/testdirectory.

19.6 MATLAB job script

All of the tweaks needed to get MATLAB working have been implemented in an example job script. This job script is also available on the HPC.

— jobscript.sh —

```
1  #!/bin/bash
2  #PBS -l nodes=1:ppn=1
3  #PBS -l walltime=1:0:0
4  #
5  # Example (single-core) MATLAB job script
6  # see http://hpcugent.github.io/vsc_user_docs/
7  #
8
9  # make sure the MATLAB version matches with the one used to compile the MATLAB
       program!
10 module load MATLAB/2018a
11
12 # use temporary directory (not $HOME) for (mostly useless) MATLAB log files
13 # subdir in $TMPDIR (if defined, or /tmp otherwise)
14 export MATLAB_LOG_DIR=$(mktemp -d -p  ${TMPDIR:-/tmp})
15
16 # configure MATLAB Compiler Runtime cache location & size (1GB)
17 # use a temporary directory in /dev/shm (i.e. in memory) for performance reasons
18 export MCR_CACHE_ROOT=$(mktemp -d -p /dev/shm)
19 export MCR_CACHE_SIZE=1024MB
20
21 # change to directory where job script was submitted from
22 cd $PBS_O_WORKDIR
23
24 # run compiled example MATLAB program 'example', provide '5' as input argument to
       the program
25 # $EBROOTMATLAB points to MATLAB installation directory
26 ./run_example.sh $EBROOTMATLAB 5
```

Chapter 20

OpenFOAM

In this chapter, we outline best practices for using the centrally provided OpenFOAM installations on the VSC HPC infrastructure.

20.1 Different OpenFOAM releases

There are currently three different sets of versions of OpenFOAM available, each with its own versioning scheme:

- OpenFOAM versions released via http://openfoam.com: `v3.0+`, `v1706`
  - see also http://openfoam.com/history/
- OpenFOAM versions released via https://openfoam.org: `v4.1`, `v5.0`
  - see also https://openfoam.org/download/history/
- OpenFOAM versions released via http://wikki.gridcore.se/foam-extend: `v3.1`

Make sure you know which flavor of OpenFOAM you want to use, since there are important differences between the different versions w.r.t. features. If the OpenFOAM version you need is not available yet, see section 10.5.

20.2 Documentation & training material

The best practices outlined here focus specifically on the use of OpenFOAM on the VSC HPC infrastructure. As such, they are intented to augment the existing OpenFOAM documentation rather than replace it. For more general information on using OpenFOAM, please refer to:

- OpenFOAM websites:
  - https://openfoam.com

  - https://openfoam.org
  - http://wikki.gridcore.se/foam-extend

- OpenFOAM user guides:
  - https://www.openfoam.com/documentation/user-guide
  - https://cfd.direct/openfoam/user-guide/
- OpenFOAM C++ source code guide: https://cpp.openfoam.org
- tutorials: https://wiki.openfoam.com/Tutorials
- recordings of "*Introduction to OpenFOAM*" training session at UGent (May 2016): https://www.youtube.com/playlist?list=PLqxhJj6bcnY9RoIgzeF6xDh5L9bbeK3BL

Other useful OpenFOAM documentation:

- https://github.com/ParticulateFlow/OSCCAR-doc/blob/master/openFoamUserManual_PFM.pdf
- http://www.dicat.unige.it/guerrero/openfoam.html

20.3 Preparing the environment

To prepare the environment of your shell session or job for using OpenFOAM, there are a couple of things to take into account.

20.3.1 Picking and loading an `OpenFOAM` module

First of all, you need to pick and load one of the available `OpenFOAM` modules. To get an overview of the available modules, run '`module avail OpenFOAM`'. For example:

```
$ module avail OpenFOAM

------------------ /apps/gent/CO7/sandybridge/modules/all ------------------
   OpenFOAM/v1712-foss-2017b     OpenFOAM/4.1-intel-2017a
   OpenFOAM/v1712-intel-2017b    OpenFOAM/5.0-intel-2017a
   OpenFOAM/2.2.2-intel-2017a    OpenFOAM/5.0-intel-2017b
   OpenFOAM/2.2.2-intel-2018a    OpenFOAM/5.0-20180108-foss-2018a
   OpenFOAM/2.3.1-intel-2017a    OpenFOAM/5.0-20180108-intel-2017b
   OpenFOAM/2.4.0-intel-2017a    OpenFOAM/5.0-20180108-intel-2018a
   OpenFOAM/3.0.1-intel-2016b    OpenFOAM/6-intel-2018a            (D)
   OpenFOAM/4.0-intel-2016b
```

To pick a module, take into account the differences between the different OpenFOAM versions w.r.t. features and API (see also section 20.1). If multiple modules are available that fulfill your requirements, give preference to those providing a more recent OpenFOAM version, and to the ones that were installed with a more recent compiler toolchain; for example, prefer a module that includes `intel-2018b` in its name over one that includes `intel-2018a`.

To prepare your environment for using OpenFOAM, load the `OpenFOAM` module you have picked;

```
$ module load OpenFOAM/4.1-intel-2017a
```

20.3.2 Sourcing the `$FOAM_BASH` script

OpenFOAM provides a script that you should `source` to further prepare the environment. This script will define some additional environment variables that are required to use OpenFOAM. The `OpenFOAM` modules define an environment variable named `FOAM_BASH` that specifies the location to this script. Assuming you are using `bash` in your shell session or job script, you should always run the following command after loading an `OpenFOAM` module:

```
$ source $FOAM_BASH
```

20.3.3 Defining utility functions used in tutorial cases

If you would like to use the `getApplication`, `runApplication`, `runParallel`, `cloneCase` and/or `compileApplication` functions that are used in OpenFOAM tutorials, you also need to `source` the `RunFunctions` script:

```
$ source $WM_PROJECT_DIR/bin/tools/RunFunctions
```

Note that this needs to be done **after** sourcing `$FOAM_BASH` to make sure `$WM_PROJECT_DIR` is defined.

20.3.4 Dealing with floating-point errors

If you are seeing `Floating Point Exception` errors, you can undefine the `$FOAM_SIGFPE` environment variable that is defined by the `$FOAM_BASH` script as follows:

```
$ unset $FOAM_SIGFPE
```

Note that this only prevents OpenFOAM from propogating floating point exceptions, which then results in terminating the simulation. However, it does not prevent that illegal operations (like a division by zero) are being executed; if `NaN` values appear in your results, floating point errors are occuring.

As such, **you should *not* use this in production runs**. Instead, you should track down the root cause of the floating point errors, and try to prevent them from occuring at all.

20.4 OpenFOAM workflow

The general workflow for OpenFOAM consists of multiple steps. Prior to running the actual simulation, some *pre-processing* needs to be done:

- generate the mesh;
- decompose the domain into subdomains using `decomposePar` (only for parallel OpenFOAM simulations);

After running the simulation, some *post-processing* steps are typically performed:

- reassemble the decomposed domain using `reconstructPar` (only for parallel OpenFOAM simulations, and optional since some postprocessing can also be done on decomposed cases);
- evaluate or further process the simulation results, either visually using ParaView (for example, via the `paraFoam` tool; use `paraFoam -builtin` for decomposed cases) or using command-line tools like `postProcess`; see also https://cfd.direct/openfoam/user-guide/postprocessing.

Depending on the size of the domain and the desired format of the results, these pre- and post-processing steps can be run either before/after the job running the actual simulation, either on the HPC infrastructure or elsewhere, or as a part of the job that runs the OpenFOAM simulation itself.

Do make sure you are using the same OpenFOAM version in each of the steps. Meshing can be done sequentially (i.e., on a single core) using for example `blockMesh`, or in parallel using more advanced meshing tools like `snappyHexMesh`, which is highly recommended for large cases. For more details, see https://cfd.direct/openfoam/user-guide/mesh/.

One important aspect to keep in mind for 'offline' pre-processing is that the domain decomposition needs to match the number of processor cores that are used for the actual simulation, see also subsection 20.5.3.

For post-processing you can either download the simulation results to a local workstation, or do the post-processing (interactively) on the HPC infrastructure, for example on the login nodes or using an interactive session on a workernode. This may be interesting to avoid the overhead of downloading the results locally.

20.5 Running OpenFOAM in parallel

For general information on running OpenFOAM in parallel, see https://cfd.direct/openfoam/user-guide/running-applications-parallel/.

20.5.1 The `-parallel` option

When running OpenFOAM in parallel, **do not forget to specify the `-parallel` option**, to avoid running the same OpenFOAM simulation N times, rather than running it once using N processor cores.

You can check whether OpenFOAM was run in parallel in the output of the main command: the OpenFOAM header text should only be included *once* in the output, and it should specify a value different than '`1`' in the `nProcs` field. Note that most pre- and post-processing utilities like `blockMesh`, `decomposePar` and `reconstructPar` can not be run in parallel.

20.5.2 Using `mympirun`

It is highly recommended to use the `mympirun` command when running parallel OpenFOAM simulations rather than the standard `mpirun` command; see chapter 21 for more information on

mympirun.

See section 21.1 for how to get started with mympirun.

To pass down the environment variables required to run OpenFOAM (which were defined by the $FOAM_BASH script, see section 20.3) to each of the MPI processes used in a parallel OpenFOAM execution, the $MYMPIRUN_VARIABLESPREFIX environment variable must be defined as follows, prior to running the OpenFOAM simulation with mympirun:

```
$ export MYMPIRUN_VARIABLESPREFIX=WM_PROJECT,FOAM,MPI
```

Whenever you are instructed to use a command like mpirun -np <N> ..., use mympirun ... instead; mympirun will automatically detect the number of processor cores that are available (see also section 21.2).

20.5.3 Domain decomposition and number of processor cores

To run OpenFOAM in parallel, you must decompose the domain into multiple subdomains. Each subdomain will be processed by OpenFOAM on one processor core.

Since mympirun will automatically use all available cores, you need to make sure that the number of subdomains matches the number of processor cores that will be used by mympirun. If not, you may run into an error message like:

```
number of processor directories = 4 is not equal to the number of processors = 16
```

In this case, the case was decomposed in 4 subdomains, while the OpenFOAM simulation was started with 16 processes through mympirun. To match the number of subdomains and the number of processor cores used by mympirun, you should either:

- adjust the value for numberOfSubdomains in system/decomposeParDict (and adjust the value for n accordingly in the domain decomposition coefficients), and run decomposePar again; or
- submit your job requesting exactly the same number of processor cores as there are subdomains (see the number of processor* directories that were created by decomposePar)

See section 21.2 to control the number of process mympirun will start.

This is interesting if you require more memory per core than is available by default. Note that the decomposition method being used (which is specified in system/decomposeParDict) has significant impact on the performance of a parallel OpenFOAM simulation. Good decomposition methods (like metis or scotch) try to limit communication overhead by minimising the number of processor boundaries.

To visualise the processor domains, use the following command:

```
$ mympirun foamToVTK -parallel -constant -time 0 -excludePatches '(".*.")'
```

and then load the VTK files generated in the VTK folder into ParaView.

20.6 Running OpenFOAM on a shared filesystem

OpenFOAM is known to significantly stress shared filesystems, since a lot of (small) files are generated during an OpenFOAM simulation. Shared filesystems are typically optimised for dealing with (a small number of) large files, and are usually a poor match for workloads that involve a (very) large number of small files (see also http://www.prace-ri.eu/IMG/pdf/IO-profiling_with_Darshan-2.pdf).

Take into account the following guidelines for your OpenFOAM jobs, which all relate to input parameters for the OpenFOAM simulation that you can specify in `system/controlDict` (see also https://cfd.direct/openfoam/user-guide/controldict).

- instruct OpenFOAM to write out results at a reasonable frequency, **certainly *not* for every single time step**; you can control this using the `writeControl`, `writeInterval`, etc. keywords;
- consider only retaining results for the last couple of time steps, see the `purgeWrite` keyword;
- consider writing results for only part of the domain (e.g., a line of plane) rather than the entire domain;
- if you do not plan to change the parameters of the OpenFOAM simulation while it is running, **set `runTimeModifiable` to `false`** to avoid that OpenFOAM re-reads each of the `system/*Dict` files at every time step;
- if the results per individual time step are large, consider setting `writeCompression` to `true`;

For modest OpenFOAM simulations where a single workernode suffices, consider using the local disk of the workernode as working directory (accessible via `$VSC_SCRATCH_NODE`), rather than the shared `$VSC_SCRATCH` filesystem. **Certainly do not use a subdirectory in `$VSC_HOME` or `$VSC_DATA` as working directory for OpenFOAM simulations**, since these shared filesystems are too slow for these type of workloads.

For large parallel OpenFOAM simulations on the UGent Tier-2 clusters, consider using the alternative shared scratch filesystem `$VSC_SCRATCH_PHANPY` (see subsection 6.2.1).

These guidelines are especially important for large-scale OpenFOAM simulations that involve more than a couple of dozen of processor cores.

20.7 Using own solvers with OpenFOAM

See https://cfd.direct/openfoam/user-guide/compiling-applications/.

20.8 Example OpenFOAM job script

Example job script for `damBreak` OpenFOAM tutorial (see also https://cfd.direct/openfoam/user-guide/dambreak):

— OpenFOAM_damBreak.sh —

```
#!/bin/bash
#PBS -l walltime=1:0:0
#PBS -l nodes=1:ppn=4
# check for more recent OpenFOAM modules with 'module avail OpenFOAM'
module load OpenFOAM/6-intel-2018a
source $FOAM_BASH
# purposely not specifying a particular version to use most recent mympirun
module load vsc-mympirun
# let mympirun pass down relevant environment variables to MPI processes
export MYMPIRUN_VARIABLESPREFIX=WM_PROJECT,FOAM,MPI
# set up working directory
# (uncomment one line defining $WORKDIR below)
#export WORKDIR=$VSC_SCRATCH/$PBS_JOBID  # for small multi-node jobs
#export WORKDIR=$VSC_SCRATCH_PHANPY/$PBS_JOBID  # for large multi-node jobs
export WORKDIR=$VSC_SCRATCH_NODE/$PBS_JOBID  # for single-node jobs
mkdir -p $WORKDIR
# damBreak tutorial, see also https://cfd.direct/openfoam/user-guide/dambreak
cp -r $FOAM_TUTORIALS/multiphase/interFoam/laminar/damBreak/damBreak $WORKDIR
cd $WORKDIR/damBreak
echo "working directory: $PWD"
# pre-processing: generate mesh
echo "start blockMesh: $(date)"
blockMesh &> blockMesh.out
# pre-processing: decompose domain for parallel processing
echo "start decomposePar: $(date)"
decomposePar &> decomposePar.out
# run OpenFOAM simulation in parallel
# note:
#  * the -parallel option is strictly required to actually run in parallel!
#    without it, the simulation is run N times on a single core...
#  * mympirun will use all available cores in the job by default,
#    you need to make sure this matches the number of subdomains!
echo "start interFoam: $(date)"
mympirun --output=interFoam.out interFoam -parallel
# post-processing: reassemble decomposed domain
echo "start reconstructPar: $(date)"
reconstructPar &> reconstructPar.out
# copy back results, i.e. all time step directories: 0, 0.05, ..., 1.0
export RESULTS_DIR=$VSC_DATA/results/$PBS_JOBID
mkdir -p $RESULTS_DIR
cp -a *.out [0-9.]* $RESULTS_DIR
echo "results copied to $RESULTS_DIR at $(date)"
# clean up working directory
cd $HOME
rm -rf $WORKDIR
```

Chapter 21

Mympirun

`mympirun` is a tool to make it easier for users of HPC clusters to run MPI programs with good performance. We strongly recommend to use `mympirun` instead of `impirun`.

In this chapter, we give a high-level overview. For a more detailed description of all options, see the vsc-mympirun README.

21.1 Basic usage

Before using `mympirun`, we first need to load its module:

```
$ module load mympirun
```

As an exception, we don't specify a version here. The reason is that we want to ensure that the latest version of the `mympirun` script is always used, since it may include important bug fixes or improvements.

The most basic form of using `mympirun` is `mympirun [mympirun options] your_program [your_program options]`.

For example, to run a program named `example` and give it a single argument (`5`), we can run it with `mympirun example 5`.

21.2 Controlling number of processes

There are four options you can choose from to control the number of processes `mympirun` will start. In the following example, the program `mpi_hello` prints a single line: `Hello world from processor <node> ...` (the sourcecode of `mpi_hello` is available in the vsc-mympirun repository).

By default, `mympirun` starts one process per *core* on every node you assigned. So if you assigned 2 nodes with 16 cores each, `mympirun` will start $2 \cdot 16 = 32$ test processes in total.

21.2.1 `--hybrid`/`-h`

This is the most commonly used option for controlling the number of processing.

The `--hybrid` option requires a positive number. This number specifies the number of processes started on each available physical *node*. It will ignore the number of available *cores* per node.

```
$ echo $PBS_NUM_NODES
2
$ mympirun -hybrid 2 ./mpi_hello
Hello world from processor node2157.delcatty.os, rank 1 out of 4 processors
Hello world from processor node2158.delcatty.os, rank 3 out of 4 processors
Hello world from processor node2158.delcatty.os, rank 2 out of 4 processors
Hello world from processor node2157.delcatty.os, rank 0 out of 4 processors
```

21.2.2 Other options

There's also `--universe`, which sets the exact amount of processes started by `mympirun`; `--double`, which uses double the amount of processes it normally would; and `--multi` that does the same as `--double`, but takes a multiplier (instead of the implied factor 2 with `--double`).

See vsc-mympirun README for a detailed explanation of these options.

21.3 Dry run

You can do a so-called "dry run", which doesn't have any side-effects, but just prints the command that `mympirun` would execute. You enable this with the `--dry-run` flag:

```
$ mympirun -dry-run ./mpi_hello
mpirun ... -genv I_MPI_FABRICS shm:dapl ... -np 16 ... ./mpi_hello
```

21.4 FAQ

21.4.1 `mympirun` seems to ignore its arguments

For example, we have a simple script (`./hello.sh`):

```
1 #!/bin/bash
2 echo "hello world"
```

And we run it like `mympirun ./hello.sh --output output.txt`.

To our surprise, this doesn't output to the file `output.txt`, but to standard out! This is because `mympirun` expects the program name and the arguments of the program to be its last arguments. Here, the `--output output.txt` arguments are passed to `./hello.sh` instead of to `mympirun`. The correct way to run it is:

```
$ mympirun -output output.txt ./hello.sh
```

21.4.2 I have other problems/questions

Please don't hesitate to contact hpc@ugent.be.

Chapter 22

Singularity

22.1 What is Singularity?

Singularity is an open-source computer program that performs operating-system-level virtualization (also known as containerisation).

One of the main uses of Singularity is to bring containers and reproducibility to scientific computing and the high-performance computing (HPC) world. Using Singularity containers, developers can work in reproducible environments of their choosing and design, and these complete environments can easily be copied and executed on other platforms.

For more general information about the use of Singularity, please see the official documentation at `https://www.sylabs.io/docs/`.

This documentation only covers aspects of using Singularity on the UGent-HPC infrastructure.

22.2 Before using Singularity

In order to use Singularity on the UGent-HPC infrastructure, you must be a member of the `gsingularity` group. See subsection 6.6.1 for how join this group.

Please take into account that someone from the UGent HPC team will need to accept your request, and it may take a couple of minutes for the accepted request gets trickled down into the UGent-HPC systems.

22.3 Restrictions on image location

Some restrictions have been put in place on the use of Singularity. This is mainly done for performance reasons and to avoid that the use of Singularity impacts other users on the system.

The Singularity image file must be located on either one of the scratch filesystems, the local disk of the workernode you are using or `/dev/shm`. The centrally provided `singularity` command will refuse to run using images that are located elsewhere, in particular on the `$VSC_HOME`, `/apps` or `$VSC_DATA` filesystems.

In addition, this implies that running containers images provided via a URL (e.g., `shub://...` or `docker://...`) will not work.

If these limitations are a problem for you, please let us know via hpc@ugent.be.

22.4 Available filesystems

All HPC-UGent shared filesystems will be readily available in a Singularity container, including the home, data and scratch filesystems, and they will be accessible via the familiar `$VSC_HOME`, `$VSC_DATA*` and `$VSC_SCRATCH*` environment variables.

22.5 Singularity Images

22.5.1 Creating Singularity images

Creating new Singularity images or converting Docker images requires admin privileges, which is obviously not available on the UGent-HPC infrastructure.

As an alternative, you can either:

- install Singularity on a local Linux system you have administrator privileges on, create you Singularity images there and upload them to the UGent-HPC infrastructure
- create Singularity images via Singularity Hub (https://www.singularity-hub.org/), and download them on the UGent-HPC infrastructure

We strongly recommend the use of Singularity Hub, see https://singularity-hub.org/ for more information.

22.5.2 Converting Docker images

For more information on converting existing Docker images to Singularity images, see https://www.sylabs.io/guides/2.6/user-guide/singularity_and_docker.html. Note that this can also be done via Singularity Hub, see https://singularity-hub.org/.

22.6 Execute our own script within our container

Copy testing image from `/apps/gent/tutorials/Singularity` to `$VSC_SCRATCH`:

```
$ cp /apps/gent/tutorials/Singularity/CentOS7_EasyBuild.img $VSC_SCRATCH/singularity
```

Create a job script like:

```
1 #!/bin/sh
2
3 #PBS -o singularity.output
```

```
#PBS -e singularity.error
#PBS -l nodes=1:ppn=1
#PBS -l walltime=12:00:00


singularity exec $VSC_SCRATCH/singularity/CentOS7_EasyBuild.img ~/my_script.sh
```

Create an example `myscript.sh`:

```
#!/bin/bash

# prime factors
factor 1234567
```

22.7 Tensorflow example

We already have a Tensorflow example image, but you can also convert the Docker image (see https://hub.docker.com/r/tensorflow/tensorflow) to a Singularity image yourself

Copy testing image from `/apps/gent/tutorials` to `$VSC_SCRATCH`:

```
$ cp /apps/gent/tutorials/Singularity/Ubuntu14.04_tensorflow.img $VSC_SCRATCH
```

```
#!/bin/sh
#
#
#PBS -o tensorflow.output
#PBS -e tensorflow.error
#PBS -l nodes=1:ppn=4
#PBS -l walltime=12:00:00
#

singularity exec $VSC_SCRATCH/Ubuntu14.04_tensorflow.img python ~/linear_regression.
    py
```

You can download `linear_regression.py` from the official Tensorflow repository.

22.8 MPI example

It is also possible to execute MPI jobs within a container, but the following requirements apply:

- Mellanox IB libraries must be available from the container (install the `infiniband-diags`, `libmlx5-1` and `libmlx4-1` OS packages)
- Use modules within the container (install the `environment-modules` or `lmod` package in your container)
- Load the required module(s) before `singularity` execution.
- Set `C_INCLUDE_PATH` variable in your container if it is required during compilation time (`export C_INCLUDE_PATH=/usr/include/x86_64-linux-gnu/:$C_INCLUDE_PATH` for Debian flavours)

Copy the testing image from `/apps/gent/tutorials/Singularity` to `$VSC_SCRATCH`

```
$ cp /apps/gent/tutorials/Singularity/Debian8_UGentMPI.img $VSC_SCRATCH/singularity
```

For example to compile an MPI example:

```
$ module load intel
$ singularity shell $VSC_SCRATCH/singularity/Debian8_UGentMPI.img
$ export LANG=C
$ export C_INCLUDE_PATH=/usr/include/x86_64-linux-gnu/:$C_INCLUDE_PATH
$ mpiicc ompi/examples/ring_c.c -o ring_debian
$ exit
```

Example MPI job script:

```
1  #!/bin/sh
2
3  #PBS -N mpi
4  #PBS -o singularitympi.output
5  #PBS -e singularitympi.error
6  #PBS -l nodes=2:ppn=15
7  #PBS -l walltime=12:00:00
8
9  module load intel vsc-mympirun
10 mympirun --impi-fallback singularity exec $VSC_SCRATCH/singularity/Debian8_UGentMPI.
       img ~/ring_debian
```

Chapter 23

SCOOP

SCOOP (Scalable COncurrent Operations in Python) is a distributed task module allowing concurrent parallel programming on various environments, from heterogeneous grids to supercomputers. It is an alternative to the worker framwork, see chapter 12.

It can used for projects that require lots of (small) tasks to be executed.

The `myscoop` script makes it very easy to use SCOOP, even in a multi-node setup.

23.1 Loading the module

Before using `myscoop`, you first need to load the `vsc-mympirun-scoop` module. We don't specify a version here (this is an exception, for most other modules you should, see subsection 4.1.7) because newer versions might include important bug fixes or performance improvements.

```
$ module load vsc-mympirun-scoop
```

23.2 Write a worker script

A Python worker script implements both the main program, and (typically) the small task function that needs to be executed a large amount of times.

This is done using the functionality provided by the `scoop` Python module, for example `futures.map` (see also https://scoop.readthedocs.org/).

First, the necessary imports need to be specified:

```
1 import sys
2 from scoop import futures
```

A Python function must be implemented for the core task, for example to compute the square of a number:

```
1 def square(x):
2     return x*x
```

The main function then applies this simple function to a range of values specified as an argument. Note that it should be guarded by a conditional (`if __name__ == "=__main__"`) to make sure it is only executed when executing the script (and not when importing from it):

```
1 if __name__ == "__main__":
2
3     # obtain n from first command line argument
4     n = int(sys.argv[1])
5
6     # compute the square of the first n numbers, in parallel using SCOOP
      functionality
7     squares = futures.map(square, range(n))  # note: returns an iterator
8
9     print("First %d squares: %s" % (n, list(squares)))
```

23.3 Executing the program

To execute the Python script implementing the task and main function in a SCOOP environment, specify to the `python` command to use the `scoop` module:

```
$ python -m scoop squares.py 10000
```

23.4 Using `myscoop`

To execute the SCOOP program in an multi-node environment, where workers are spread across multiple physical systems, simply use `myscoop`: just specify the name of the Python module in which the SCOOP program is implemented, and specify further arguments on the command line.

You will need to make sure that the path to where the Python module is located is listed in `$PYTHONPATH`.

This is an example of a complete job script executing the SCOOP program in parallel in a multi-node job (i.e. 2 nodes with 8 cores each):

— squares_jobscript.pbs —

```
#!/bin/bash

#PBS -l nodes=2:ppn=8

module load vsc-mympirun-scoop

# change to directory where job was submitted from
cd $PBS_O_WORKDIR

# assume squares.py is in current directory
export PYTHONPATH=.:$PYTHONPATH

# compute first 10k squares, in parallel on available cores
myscoop --scoop-module=squares 10000
```

Note that you don't need to specify how many workers need to be used; the `myscoop` command figures this out by itself. This is because `myscoop` is a wrapper around `mympirun` (see chapter 21). In this example, 16 workers (one per available core) will be execute the 10000 tasks one by one until all squares are computed.

To run the same command on the local system (e.g. a login node for testing), add the `--sched =local` option to `myscoop`.

23.5 Example: calculating π

A more practical example of a worker script is one to compute π using a Monte-Carlo method (see also https://scoop.readthedocs.org/en/0.6/examples.html#computation-of).

The `test` function implements a tiny task that is be executed `tries` number of times by each worker. Afterwards, the number of successful tests is determined using the Python `sum` function, and an approximate value of π is computed and returned by `calcPi` so the main function can print it out.

— picalc.py —

```
from math import hypot
from random import random
from scoop import futures

NAME = 'SCOOP_piCalc'

# A range is used in this function for python3. If you are using python2,
# an xrange might be more efficient.
try:
    range_fn = xrange
except:
    range_fn = range


def test(tries):
    return sum(hypot(random(), random()) < 1 for i in range_fn(tries))

# Calculates pi with a Monte-Carlo method. This function calls the function
# test "n" times with an argument of "t". Scoop dispatches these
# functions interactively accross the available resources.
def calcPi(workers, tries):
    expr = futures.map(test, [tries] * workers)
    piValue = 4. * sum(expr) / float(workers * tries)
    return piValue


if __name__ == '__main__':
    import sys
    nr_batches = 3000
    batch_size = 5000

    # Program name and two arguments
    if len(sys.argv) == 3:
        try:
            nr_batches = int(sys.argv[1])
            batch_size = int(sys.argv[2])
        except ValueError as ex:
            sys.stderr.write("ERROR: Two integers expected as arguments: %s\n" % ex)
            sys.exit(1)
    elif len(sys.argv) != 1:
        sys.stderr.write("ERROR: Expects either zero or two integers as arguments.\n
    ")
        sys.exit(1)

    print("PI=%f (in nr_batches=%d,batch_size=%d)" % (calcPi(nr_batches, batch_size)
    , nr_batches, batch_size))
```

— picalc_job_script.pbs —

```
#!/bin/bash
#PBS -l nodes=2:ppn=16

module load vsc-mympirun-scoop

# change to directory where job was submitted from
cd $PBS_O_WORKDIR

# assume picalc.py is in current directory
export PYTHONPATH=.:$PYTHONPATH

# run 10k batches/workers with a batch size of 5000
myscoop --scoop-module=picalc 10000 5000
```

Chapter 24

Easybuild

24.1 What is Easybuild?

You can use EasyBuild to build and install supported software in your own VSC account, rather than requesting a central installation by the HPC support team.

EasyBuild (https://easybuilders.github.io/easybuild) is the software build and installation framework that was created by the HPC-UGent team, and has recently been picked up by HPC sites around the world. It allows you to manage (scientific) software on High Performance Computing (HPC) systems in an efficient way.

24.2 When should I use Easybuild?

For general software installation requests, please see section 10.5. However, there might be reasons to install the software yourself:

- applying custom patches to the software that only you or your group are using
- evaluating new software versions prior to requesting a central software installation
- installing (very) old software versions that are no longer eligible for central installation (on new clusters)

24.3 Configuring EasyBuild

Before you use EasyBuild, you need to configure it:

24.3.1 Path to sources

This is where EasyBuild can find software sources:

```
$ export EASYBUILD_SOURCEPATH=$VSC_DATA/easybuild/sources:/apps/gent/source
```

- the first directory `$VSC_DATA/easybuild/sources` is where EasyBuild will (try to) automatically download sources if they're not available yet
- `/apps/gent/source` is the central "cache" for already downloaded sources, and will be considered by EasyBuild before downloading anything

24.3.2 Build directory

This directory is where EasyBuild will build software in. To have good performance, this needs to be on a fast filesystem.

```
$ export EASYBUILD_BUILDPATH=$TMPDIR:-/tmp/$USER
```

On cluster nodes, you can use the fast, in-memory `/dev/shm/$USER` location as a build directory.

24.3.3 Software install location

This is where EasyBuild will install the software (and accompanying modules) to.

For example, to let it use `$VSC_DATA/easybuild`, use:

```
$ export EASYBUILD_INSTALLPATH=$VSC_DATA/easybuild/$VSC_OS_LOCAL/
    $VSC_ARCH_LOCAL$VSC_ARCH_SUFFIX
```

Using the `$VSC_OS_LOCAL`, `$VSC_ARCH` and `$VSC_ARCH_SUFFIX` environment variables ensures that your install software to a location that is specific to the cluster you are building for.

Make sure you **do not build software on the login nodes**, since the loaded `cluster` module determines the location of the installed software. Software built on the login nodes may not work on the cluster you want to use the software on (see also section 8.9).

To share custom software installations with members of your VO, replace `$VSC_DATA` with `$VSC_DATA_VO` in the example above.

24.4 Using EasyBuild

Before using EasyBuild, you first need to load the `EasyBuild` module. We don't specify a version here (this is an exception, for most other modules you should, see subsection 4.1.7) because newer versions might include important bug fixes.

```
module load EasyBuild
```

24.4.1 Installing supported software

EasyBuild provides a large collection of readily available software versions, combined with a particular toolchain version. Use the `--search` (or `-S`) functionality to see which different 'easycon-

figs' (build recipes, see http://easybuild.readthedocs.org/en/latest/Concepts_and_Terminology.html#easyconfig-files) are available:

```
$ eb -S example-1.2
CFGS1=/apps/gent/CO7/sandybridge/software/EasyBuild/3.6.2/lib/python2.7/site-
   packages/easybuild_easyconfigs-3.6.2-py2.7.egg/easybuild/easyconfigs
 * $CFGS1/e/example/example-1.2.1-foss-2018a.eb
 * $CFGS1/e/example/example-1.2.3-foss-2018b.eb
 * $CFGS1/e/example/example-1.2.5-intel-2018a.eb
```

For readily available easyconfigs, just specify the name of the easyconfig file to build and install the corresponding software package:

```
$ eb example-1.2.1-foss-2018a.eb --robot
```

24.4.2 Installing variants on supported software

To install small variants on supported software, e.g. a different software version, or using a different compiler toolchain, use the corresponding `--try-X` options:

To try to install `example v1.2.6`, based on the easyconfig file for `example v1.2.5`:

```
$ eb example-1.2.5-intel-2018a.eb --try-software-version=1.2.6
```

To try to install example v1.2.5 with a different compiler toolchain:

```
$ eb example-1.2.5-intel-2018a.eb --robot --try-toolchain=intel,2018b
```

24.4.3 Install other software

To install other, not yet supported, software, you will need to provide the required easyconfig files yourself. See https://easybuild.readthedocs.org/en/latest/Writing_easyconfig_files.html for more information.

24.5 Using the installed modules

To use the modules you installed with EasyBuild, extend `$MODULEPATH` to make them accessible for loading:

```
$ module use $EASYBUILD_INSTALLPATH/modules/all
```

It makes sense to put this `module use` command and all `export` commands in your `.bashrc` login script. That way you don't have to type these commands every time you want to use EasyBuild or you want to load modules generated with EasyBuild. See also the section on `.bashrc` in the "Beyond the basics" chapter of the intro to Linux.

Chapter 25

Hanythingondemand (HOD)

Hanythingondemand (or HOD for short) is a tool to run a Hadoop (Yarn) cluster on a traditional HPC system.

25.1 Documentation

The official documentation for HOD version 3.0.0 and newer is available at https://hod.readthedocs.org/en/latest/. The slides of the 2016 HOD training session are available at http://users.ugent.be/~kehoste/hod_20161024.pdf.

This chapter only covers how HOD is installed on the UGent-HPC infrastructure.

25.2 Using HOD

Before using HOD, you first need to load the `hod` module. We don't specify a version here (this is an exception, for most other modules you should, see subsection 4.1.7) because newer versions might include important bug fixes.

```
$ module load hod
```

25.2.1 Compatibility with login nodes

The `hod` modules are constructed such that they can be used on the UGent-HPC login nodes, regardless of which `cluster` module is loaded (this is not the case for software installed via modules in general, see section 8.9).

As such, you should experience no problems if you swap to a different cluster module before loading the `hod` module and subsequently running |hod|.

For example, this will work as expected:

```
$ module swap cluster/swalot
$ module load hod
$ hod
hanythingondemand - Run services within an HPC cluster
usage: hod <subcommand> [subcommand options]
Available subcommands (one of these must be specified!):
    batch           Submit a job to spawn a cluster on a PBS job controller, run a
    job script, and tear down the cluster when it's done
    clean           Remove stale cluster info.
...
```

Note that also modules named `hanythingondemand/*` are available. These should however not be used directly, since they may not be compatible with the login nodes (depending on which cluster they were installed for).

25.2.2 Standard HOD configuration

The `hod` module will also put a basic configuration in place for HOD, by defining a couple of `$HOD_*` environment variables:

```
$ module load hod
$ env | grep HOD | sort
HOD_BATCH_HOD_MODULE=hanythingondemand/3.2.2-intel-2016b-Python-2.7.12
HOD_BATCH_WORKDIR=$VSC_SCRATCH/hod
HOD_CREATE_HOD_MODULE=hanythingondemand/3.2.2-intel-2016b-Python-2.7.12
HOD_CREATE_WORKDIR=$VSC_SCRATCH/hod
```

By defining these environment variables, we avoid that you have to specify `--hod-module` and `--workdir` when using `hod batch` or `hod create`, since they are strictly required.

If you want to use a different parent working directory for HOD, it suffices to either redefine `$HOD_BATCH_WORKDIR` and `$HOD_CREATE_WORKDIR`, or to specify `--workdir` (which will override the corresponding environment variable).

Changing the HOD module that is used by the HOD backend (i.e. using `--hod-module` or redefining `$HOD_*_HOD_MODULE`) is strongly discouraged.

25.2.3 Cleaning up

After HOD clusters terminate, their local working directory and cluster information is typically not cleaned up automatically (for example, because the job hosting an interactive HOD cluster submitted via `hod create` runs out of walltime).

These HOD clusters will still show up in the output of `hod list`, and will be marked as `<job-not-found>`.

You should occasionally clean this up using `hod clean`:

```
$ module list
Currently Loaded Modulefiles:
  1) cluster/delcatty(default)   2) pbs_python/4.6.0            3) vsc-base/2.4.2
              4) hod/3.0.0-cli

$ hod list
Cluster label   Job ID                                          State
    Hosts
example1               123456.master15.delcatty.gent.vsc        <job-not-found> <
    none>

$ hod clean
Removed cluster localworkdir directory /user/scratch/gent/vsc400/vsc40000/hod/hod/
    123456.master15.delcatty.gent.vsc for cluster labeled example1
Removed cluster info directory /user/home/gent/vsc400/vsc40000/.config/hod.d/
    wordcount for cluster labeled example1

$ module swap cluster/swalot
$ hod list
Cluster label   Job ID                            State           Hosts
example2               98765.master19.swalot.gent.vsc        <job-not-found> <none>

$ hod clean
Removed cluster localworkdir directory /user/scratch/gent/vsc400/vsc40000/hod/hod
    /98765.master19.swalot.gent.vsc for cluster labeled example2
Removed cluster info directory /user/home/gent/vsc400/vsc40000/.config/hod.d/
    wordcount for cluster labeled example2
```

Note that **only HOD clusters that were submitted to the currently loaded `cluster` module will be cleaned up**.

25.3 Getting help

If you have any questions, or are experiencing problems using HOD, you have a couple of options:

- Subscribe to the HOD mailing list via https://lists.ugent.be/wws/info/hod, and contact the HOD users and developers at hod@lists.ugent.be.
- Contact the UGent HPC team via hpc@ugent.be
- Open an issue in the `hanythingondemand` GitHub repository, via https://github.com/hpcugent/hanythingondemand/issues.

Appendix A

HPC Quick Reference Guide

Remember to subsitute the usernames, login nodes, file names, ... for your own.

Login	
Login	`ssh vsc40000@login.hpc.ugent.be`
Where am I?	`hostname`
Copy to HPC	`scp foo.txt vsc40000@login.hpc.ugent.be:`
Copy from HPC	`scp vsc40000@login.hpc.ugent.be:foo.txt .`
Setup ftp session	`sftp vsc40000@login.hpc.ugent.be`

Modules	
List all available modules	`module avail`
List loaded modules	`module list`
Load module	`module load example`
Unload module	`module unload example`
Unload all modules	`module purge`
Help on use of module	`module help`

Jobs	
Submit job with job script `script.pbs`	`qsub script.pbs`
Status of job with ID 12345	`qstat 12345`
Show compute node of job with ID 12345	`qstat -n 12345`
Delete job with ID 12345	`qdel 12345`
Status of all your jobs	`qstat`
Detailed status of your jobs + a list nodes they are running on	`qstat -na`
Submit Interactive job	`qsub -I`

Disk quota	
Check your disk quota	See `https://account.vscentrum.be`
Disk usage in current directory (`.`)	`du -h .`

Worker Framework	
Load worker module	`module load worker/1.6.8-intel-2018a` Don't forget to specify a version. To list available versions, use `module avail worker/`
Submit parameter sweep	`wsub -batch weather.pbs -data data.csv`
Submit job array	`wsub -t 1-100 -batch test_set.pbs`
Submit job array with prolog and epilog	`wsub -prolog pre.sh -batch test_set.pbs -epilog post.sh -t 1-100`

Appendix B

TORQUE options

B.1 TORQUE Submission Flags: common and useful directives

Below is a list of the most common and useful directives.

Option	**System type**	**Description**
-k	All	Send "stdout" and/or "stderr" to your home directory when the job runs **#PBS -k o** or **#PBS -k e** or **#PBS -koe**
-l	All	Precedes a resource request, e.g., processors, wallclock
-M	All	Send an e-mail messages to an alternative e-mail address **#PBS -M me@mymail.be**
-m	All	Send an e-mail address when a job **b**egins execution and/or **e**nds or **a**borts **#PBS -m b** or **#PBS -m be** or **#PBS -m ba**
mem	Shared Memory	Specifies the amount of memory you need for a job. **#PBS -l mem=80gb**
mpiprocs	Clusters	Number of processes per node on a cluster. This should equal number of processors on a node in most cases. **#PBS -l mpiprocs=4**
-N	All	Give your job a unique name **#PBS -N galaxies1234**
-ncpus	Shared Memory	The number of processors to use for a shared memory job. **#PBS ncpus=4**
-r	All	Control whether or not jobs should automatically re-run from the start if the system crashes or is rebooted. Users with check points might not wish this to happen. **#PBS -r n** **#PBS -r y**
select	Clusters	Number of compute nodes to use. Usually combined with the mpiprocs directive **#PBS -l select=2**

-V	All	Make sure that the environment in which the job **runs** is the same as the environment in which it was **submitted.** #**PBS -V**
Walltime	All	The maximum time a job can run before being stopped. If not used a default of a few minutes is used. Use this flag to prevent jobs that go bad running for hundreds of hours. Format is HH:MM:SS #**PBS -l walltime=12:00:00**

B.2 Environment Variables in Batch Job Scripts

TORQUE-related environment variables in batch job scripts.

```
1  # Using PBS - Environment Variables:
2  # When a batch job starts execution, a number of environment variables are
3  # predefined, which include:
4  #
5  #      Variables defined on the execution host.
6  #      Variables exported from the submission host with
7  #                -v (selected variables) and -V (all variables).
8  #      Variables defined by PBS.
9  #
10 # The following reflect the environment where the user ran qsub:
11 # PBS_O_HOST    The host where you ran the qsub command.
12 # PBS_O_LOGNAME Your user ID where you ran qsub.
13 # PBS_O_HOME    Your home directory where you ran qsub.
14 # PBS_O_WORKDIR The working directory where you ran qsub.
15 #
16 # These reflect the environment where the job is executing:
17 # PBS_ENVIRONMENT       Set to PBS_BATCH to indicate the job is a batch job,
18 #         or to PBS_INTERACTIVE to indicate the job is a PBS interactive job.
19 # PBS_O_QUEUE   The original queue you submitted to.
20 # PBS_QUEUE     The queue the job is executing from.
21 # PBS_JOBID     The job's PBS identifier.
22 # PBS_JOBNAME   The job's name.
```

IMPORTANT!! All PBS directives MUST come before the first line of executable code in your script, otherwise they will be ignored.

When a batch job is started, a number of environment variables are created that can be used in the batch job script. A few of the most commonly used variables are described here.

Variable	**Description**
PBS_ENVIRONMENT	set to PBS_BATCH to indicate that the job is a batch job; otherwise, set to PBS_INTERACTIVE to indicate that the job is a PBS interactive job.
PBS_JOBID	the job identifier assigned to the job by the batch system. This is the same number you see when you do *qstat*.
PBS_JOBNAME	the job name supplied by the user

PBS_NODEFILE	the name of the file that contains the list of the nodes assigned to the job . Useful for Parallel jobs if you want to refer the node, count the node etc.
PBS_QUEUE	the name of the queue from which the job is executed
PBS_O_HOME	value of the HOME variable in the environment in which *qsub* was executed
PBS_O_LANG	value of the LANG variable in the environment in which *qsub* was executed
PBS_O_LOGNAME	value of the LOGNAME variable in the environment in which *qsub* was executed
PBS_O_PATH	value of the PATH variable in the environment in which *qsub* was executed
PBS_O_MAIL	value of the MAIL variable in the environment in which *qsub* was executed
PBS_O_SHELL	value of the SHELL variable in the environment in which *qsub* was executed
PBS_O_TZ	value of the TZ variable in the environment in which *qsub* was executed
PBS_O_HOST	the name of the host upon which the *qsub* command is running
PBS_O_QUEUE	the name of the original queue to which the job was submitted
PBS_O_WORKDIR	the absolute path of the current working directory of the *qsub* command. This is the most useful. Use it in every job script. The first thing you do is, cd $PBS_O_WORKDIR after defining the resource list. This is because, pbs throw you to your $HOME directory.
PBS_O_NODENUM	node offset number
PBS_O_VNODENUM	vnode offset number
PBS_VERSION	Version Number of TORQUE, e.g., TORQUE-2.5.1
PBS_MOMPORT	active port for mom daemon
PBS_TASKNUM	number of tasks requested
PBS_JOBCOOKIE	job cookie
PBS_SERVER	Server Running TORQUE

Appendix C

Useful Linux Commands

C.1 Basic Linux Usage

All the HPC clusters run some variant of the "Red Hat Enterprise Linux" operating system. This means that, when you connect to one of them, you get a command line interface, which looks something like this:

```
vsc40000@ln01[203] $
```

When you see this, we also say you are inside a "shell". The shell will accept your commands, and execute them.

ls	Shows you a list of files in the current directory
cd	Change current working directory
rm	Remove file or directory
nano	Text editor
echo	Prints its parameters to the screen

Most commands will accept or even need parameters, which are placed after the command, separated by spaces. A simple example with the "echo" command:

```
$ echo This is a test
This is a test
```

Important here is the "$" sign in front of the first line. This should not be typed, but is a convention meaning "the rest of this line should be typed at your shell prompt". The lines not starting with the "$" sign are usually the feedback or output from the command.

More commands will be used in the rest of this text, and will be explained then if necessary. If not, you can usually get more information about a command, say the item or command "ls", by trying either of the following:

```
$ ls -help
$ man ls
$ info ls
```

(You can exit the last two "manuals" by using the "q" key.) For more exhaustive tutorials about Linux usage, please refer to the following sites: http://www.linux.org/lessons/

http://linux.about.com/od/nwb_guide/a/gdenwb06.htm

C.2 How to get started with shell scripts

In a shell script, you will put the commands you would normally type at your shell prompt in the same order. This will enable you to execute all those commands at any time by only issuing one command: starting the script.

Scripts are basically non-compiled pieces of code: they are just text files. Since they don't contain machine code, they are executed by what is called a "parser" or an "interpreter". This is another program that understands the command in the script, and converts them to machine code. There are many kinds of scripting languages, including Perl and Python.

Another very common scripting language is shell scripting. In a shell script, you will put the commands you would normally type at your shell prompt in the same order. This will enable you to execute all those commands at any time by only issuing one command: starting the script.

Typically in the following examples they'll have on each line the next command to be executed although it is possible to put multiple commands on one line. A very simple example of a script may be:

```
1 echo "Hello! This is my hostname:"
2 hostname
```

You can type both lines at your shell prompt, and the result will be the following:

```
$ echo "Hello!  This is my hostname:"
Hello! This is my hostname:
$ hostname
gligar04.gastly.os
```

Suppose we want to call this script "foo". You open a new file for editing, and name it "foo", and edit it with your favourite editor

```
$ nano foo
```

or use the following commands:

```
$ echo "echo Hello!  This is my hostname:" > foo
$ echo hostname >> foo
```

The easiest ways to run a script is by starting the interpreter and pass the script as parameter. In case of our script, the interpreter may either be "sh" or "bash" (which are the same on the cluster). So start the script:

```
$ bash foo
Hello! This is my hostname:
gligar04.gastly.os
```

Congratulations, you just created and started your first shell script!

A more advanced way of executing your shell scripts is by making them executable by their own, so without invoking the interpreter manually. The system can not automatically detect which

interpreter you want, so you need to tell this in some way. The easiest way is by using the so called "shebang"-notation, explicitly created for this function: you put the following line on top of your shell script "#!/path/to/your/interpreter".

You can find this path with the "which" command. In our case, since we use bash as an interpreter, we get the following path:

```
$ which bash
/bin/bash
```

We edit our script and change it with this information:

```
1 #!/bin/bash
2 echo "Hello! This is my hostname:"
3 hostname
```

Note that the "shebang" must be the first line of your script! Now the operating system knows which program should be started to run the script.

Finally, we tell the operating system that this script is now executable. For this we change its file attributes:

```
$ chmod +x foo
```

Now you can start your script by simply executing it:

```
$ ./foo
Hello! This is my hostname:
gligar04.gastly.os
```

The same technique can be used for all other scripting languages, like Perl and Python.

Most scripting languages understand that lines beginning with "#" are comments, and should be ignored. If the language you want to use does not ignore these lines, you may get strange results ...

C.3 Linux Quick reference Guide

C.3.1 Archive Commands

tar	An archiving program designed to store and extract files from an archive known as a tar file.
tar -cvf foo.tar foo/	compress the contents of foo folder to foo.tar
tar -xvf foo.tar	extract foo.tar
tar -xvzf foo.tar.gz	extract gzipped foo.tar.gz

C.3.2 Basic Commands

ls	Shows you a list of files in the current directory
cd	Change the current directory
rm	Remove file or directory
mv	Move file or directory
echo	Display a line or text
pwd	Print working directory
mkdir	Create directories
rmdir	Remove directories

C.3.3 Editor

emacs	
nano	Nano's ANOther editor, an enhanced free Pico clone
vi	A programmers text editor

C.3.4 File Commands

cat	Read one or more files and print them to standard output
cmp	Compare two files byte by byte
cp	Copy files from a source to the same or different target(s)
du	Estimate disk usage of each file and recursively for directories
find	Search for files in directory hierarchy
grep	Print lines matching a pattern
ls	List directory contents
mv	Move file to different targets
rm	Remove files
sort	Sort lines of text files
wc	Print the number of new lines, words, and bytes in files

C.3.5 Help Commands

man	Displays the manual page of a command with its name, synopsis, description, author, copyright etc.

C.3.6 Network Commands

hostname	show or set the system's host name
ifconfig	Display the current configuration of the network interface. It is also useful to get the information about IP address, subnet mask, set remote IP address, netmask etc.
ping	send ICMP ECHO_REQUEST to network hosts, you will get back ICMP packet if the host responds. This command is useful when you are in a doubt whether your computer is connected or not.

C.3.7 Other Commands

logname	Print user's login name
quota	Display disk usage and limits
which	Returns the pathnames of the files that would be executed in the current environment
whoami	Displays the login name of the current effective user

C.3.8 Process Commands

&	In order to execute a command in the background, place an ampersand (&) on the command line at the end of the command. A user job number (placed in brackets) and a system process number are displayed. A system process number is the number by which the system identifies the job whereas a user job number is the number by which the user identifies the job
at	executes commands at a specified time
bg	Places a suspended job in the background
crontab	crontab is a file which contains the schedule of entries to run at specified times
fg	A process running in the background will be processed in the foreground
jobs	Lists the jobs being run in the background
kill	Cancels a job running in the background, it takes argument either the user job number or the system process number
ps	Reports a snapshot of the current processes
top	Display Linux tasks

C.3.9 User Account Commands

chmod	Modify properties for users
chown	Change file owner and group