Genomics Core

VSC GC Guide

Author: Koen Herten

Version: 10

Date: 04-May-2016 12:56

Table of Contents

This is the Genomics Core (GC) guide for the usage of the High Performance Supercomputer (HPC) available at the Flemish Supercomputer Center (VSC).

This guide is an informative, possible incomplete guide ment to help Biologists and Bioinformatics in the setup of there analysis on a HPC.

1 Introduction

1.1 What is a supercomputer?

A supercomputer is a very fast and extremely parallel computer. Many of its technological properties are comparable to those of your laptop or even smartphone but there are important differences.

There is no clear agreement on the exact definition of the term ‘supercomputer’. Some say a supercomputer is a computer with at least 1% of the computing power of the fastest computer in the world. But according to this definition, there are currently only a few hundred supercomputers in the world. On www.top500.org you can find a supposed list of rankings for the fastest computers in the world.

One could take 1‰ of the performance of the fastest computer as the criterion, but it is an arbitrary criterion. Stating that a supercomputer should perform at least X trillion computations per second, is not a useful definition. Because of the fast evolution of the technology, this definition would be outdated in a matter of years. The first smartphone of a well-known manufacturer launched in 2007 had about the same computing power and more memory than the computer used to predict the weather in Europe 30 years earlier.

So what is considered as a ‘supercomputer’ is very time-bound, at least in terms of absolute compute power. So let us just agree that a supercomputer is a computer that is hundreds or thousands times faster than your smartphone or laptop.

But is a supercomputer so different from your laptop or smartphone? Yes and no. Since roughly 1975 the key word in supercomputing is parallelism. But this also applies for your PC or smartphone. PC processor manufacturers started to experiment with simple forms of parallelism at the end of the nineties. A few years later the first processors appeared with multiple cores that could perform calculations independently from each other. A laptop has mostly 2 or 4 cores and modern smartphones have 2, 4 or in some rare cases 8 cores. Although it must be added that they are a little slower than the ones on a typical laptop.

Around 1975 manufacturers started to experiment with vector processors. These processors perform the same operation to a set of numbers simultaneously. Shortly thereafter supercomputers with multiple processors working independently from each other, appeared on the market. Similar technologies are nowadays used in the processor chips of laptops and smartphones. In the eighties, supercomputer designers started to experiment with another kind

of parallelism. Several rather simple processors - this was sometimes just standard PC processors like the venerable Intel 80386 were linked together with fast networks and collaborated to solve large problems. These computers were cheaper to develop, much simpler to build, but required frequent changes to the software.

In modern supercomputers, parallelism is pushed to extremes. In most supercomputers, all forms of parallelism mentioned above are combined at an unprecedented scale and can take on extreme forms. All modern supercomputers rely on some form of vector computing or related technologies and consist of building blocks - *nodes* - uniting tens of cores and interconnecting through a fast network to a larger whole. Hence the term ‘compute cluster’ is often used.

Supercomputers must also be able to read and interpret data is ‘at a very high speed. Here the key word is also parallellism. Many supercomputers have several network connections to the outside world. Their permanent storage system consists of hundreds or even thousands of hard disks or SSDs linked together to one extremely large and extremely fast storage system. This type of technology has probably not influenced significantly the development of laptops as it would not be very practical to carry a laptop around with 4 hard drives. Yet this technology does appear to some extent in modern, fast SSD drives in some laptops and smartphones. The faster ones use several memory chips in parallel to increase their performance and it is a standard technology in almost any server storing data.

As we have already indicated to some extent in the text above, a supercomputer is more than just hardware. It also needs properly written software. or Java program you wrote during your student years will not run a 10. 000 times faster because you run it on a supercomputer. On the contrary, there is a fair chance that it won't run at all or run slower than on your PC. Most supercomputers - and all supercomputers at the VSC - use a variant of the Linux operating system enriched with additional software to combine all compute nodes in one powerful supercomputer. Due to the high price of such a computer, you're rarely the only user but will rather share the infrastructure with others.

So you may have to wait a little before your program runs. Furthermore your monitor is not directly connected to the supercomputer. Proper software is also required here with your application software having to be adapted to run well on a supercomputer. Without these changes, your program will not run much faster than on a regular PC. You may of course still run hundreds or thousands copies simultaneously, when you for example wish to explore a parameter space. This is called ‘capacity computing’.

If you wish to solve truly large problems within a reasonable timeframe, you will have to adapt your application software to maximize every form of parallellism within a modern supercomputer and use several hundreds, or even thousands, of compute cores simultaneously to solve one large problem. This is called ‘capability computing’. Of course, the problem you wish to solve has to be large enough for this approach to make sense. Every problem has an intrinsic limit to the speedup you can achieve on a supercomputer. The larger the problem, the higher speedup you can achieve.

This also implies that a software package that was cutting edge in your research area 20 years ago, is unlikely to be so anymore because it is not properly adapted to modern supercomputers, while new applications exploit supercomputers much more efficiently and subsequently generate faster, more accurate results.

To some extent this also applies to your PC. Here again you are dealing with software exploiting the parallelism of a modern PC quite well or with software that doesn't. As a ‘computational scientist’ or supercomputer user you constantly have to be open to new developments within this area. Fortunately, in most application domains, a lot of efficient software already exists which succeeds in using all the parallellism that can be found in modern supercomputers.

1.2 What is the VSC?

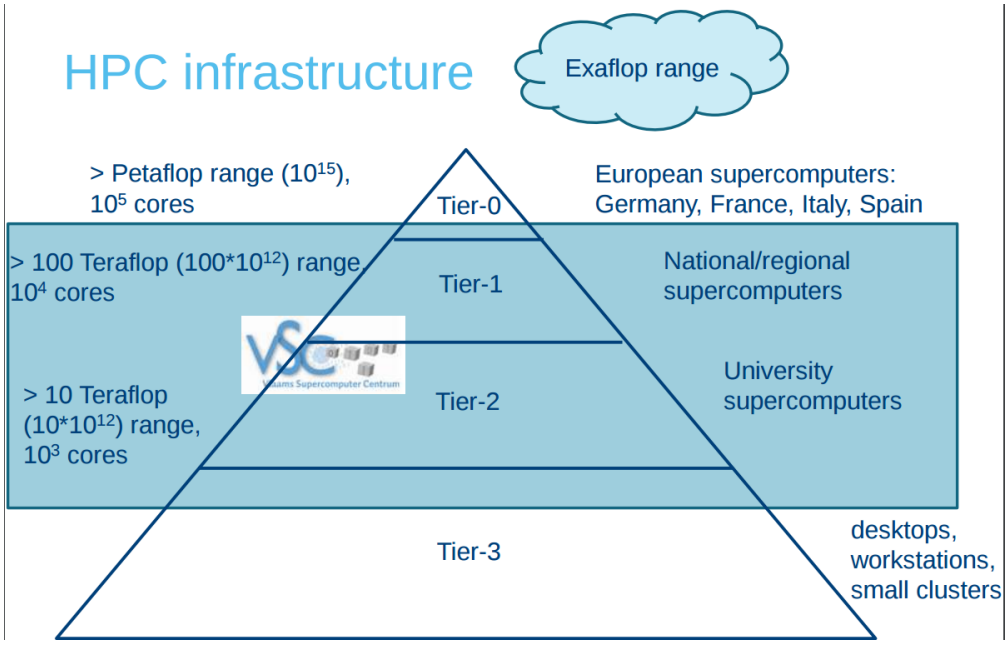
The Flemish Supercomputer Centre (VSC) is a virtual centre making supercomputer infrastructure available for both the academic and industrial world. This centre is managed by the Research Foundation - Flanders (FWO) in partnership with the five Flemish university associations.

The VSC is responsible for the development and management of High Performance Computer Infrastructure used for research and innovation.

The VSC infrastructure consists of two layers in the European multi-layer model for an integrated HPC infrastructure. Local clusters (Tier-2) at the Flemish universities are responsible for processing the mass of smaller computational tasks and provide a solid base for the HPC ecosystem. A larger central supercomputer (Tier-1) is necessary for more complicated calculations while simultaneously serving as a bridge to infrastructures at a European level.

The VSC assists researchers active in academic institutions and also the industry when using HPC through training programs and targeted advice. This offers the advantage that academia and industrialists come into contact with each other.

In addition, the VSC also works on raising awareness of the added value HPC can offer both in academic research and in industrial applications.



1.3 Basic Architecture at the VSC

This guide will only handle the Tier-2 infrastructure. Tier-1 is for big projects with heavy computing. Most bioinformatics applications do not need that kind of hardware.

The Tier-2 infrastructure has 3 big clusters:

- ThinKing
  The thin node cluster
- Cerebro
  The shared memory cluster
- Accelerators
  The GPU clusters. Since not many bioinformatics tools can use GPUs this cluster is not discussed in this guide.

More on these computing clusters can be found in the The Hardware chapter.

Besides these clusters there are some specific nodes:

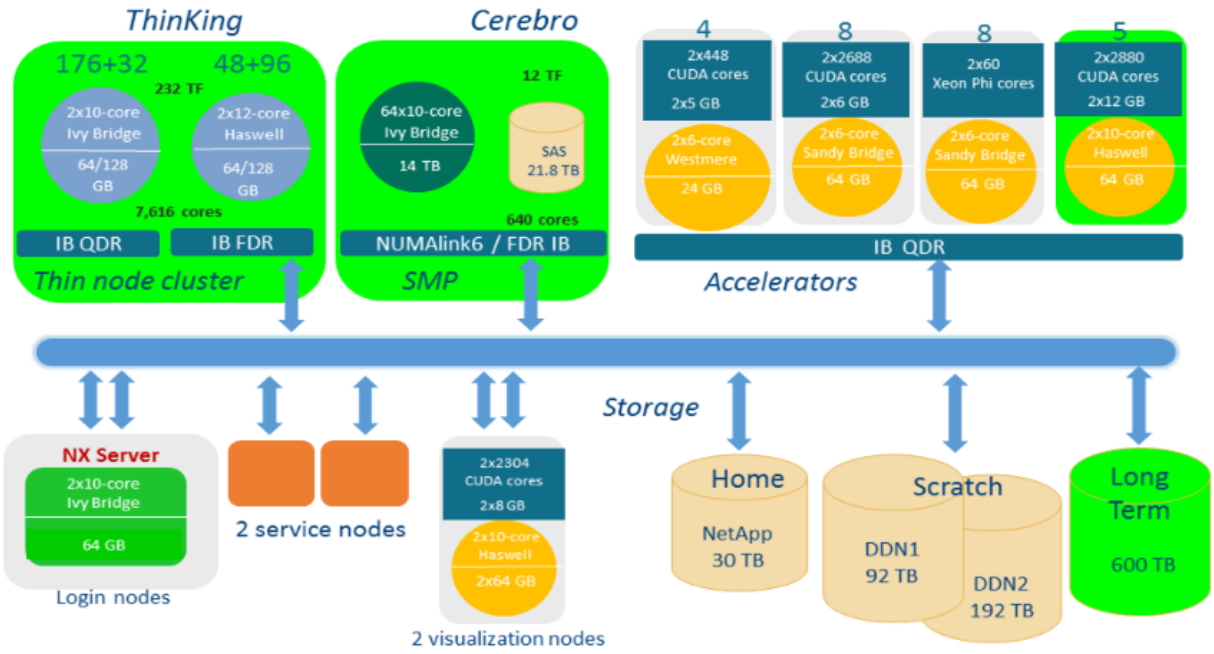
- Login nodes
  This are the nodes available to login in
- Service nodes
  These nodes are used for service, and are not usable for common users
- Visualization nodes
  These are visualization and interactive nodes. These nodes are ignored in this guide.

When logging in, the user will always work in one of the login nodes. All commands described in this guide will be executed here.

As last aspect of the VSC there is the Storage. There are 3 big storage clusters:

- The Home cluster
  This contains all users Home and Data drives
- The Scratch cluster
  This contains the site scratch, users scratch and staging partitions
- The Long Term cluster
  This is for archiving. This storage cluster is only accessible through the login nodes.

The storage and access is described in the Storage chapter.



1.4 References

All information found in this guide are interpretations found in the guides, manuals and the website of the VSC. Examples and interpretations are change to a biological/bioinformatical perspective. The Cheat Sheet and the Appendices are made for the Genomics Core by Koen Herten. Please reference both when using this information.

1.5 How do I acknowledge the VSC in publications?

When using the VSC-infrastructure for your research, you must acknowledge the VSC in all relevant publications. This will help the VSC secure funding, and hence you will benefit from it in the long run as well. It is also a contractual obligation for the VSC.

Please use the following phrase to do so in Dutch “De rekeninfrastructuur en dienstverlening gebruikt in dit werk, werd voorzien door het VSC (Vlaams Supercomputer Centrum), gefinancierd door het FWO en de Vlaamse regering – departement EWI”, or in English: “The computational resources and services used in this work were provided by the VSC (Flemish Supercomputer Center), funded by the Research Foundation - Flanders (FWO) and the Flemish Government – department EWI”.

Moreover, if you are in the KU Leuven association, you are also requested to add the relevant papers to the virtual collection "High Performance Computing" in Lirias so that we can easily generate the publication lists with relevant publications.

2 Account - Credits - Storage Request

This chapter is handling the request and the generation of an account, the request for credits and the request of storage. The use of Credits and Storage is described in there own Chapters.

2.1 Account

To log on to and use the VSC infrastructure, you need a so-called VSC account.
All VSC-accounts start with the letters "vsc" followed by a five-digit number. The first digit gives information about your home institution. There is no relationship with your name, nor is the information about the link between VSC-accounts and your name publicly accessible. Unlike your institute account, VSC accounts don't use regular fixed passwords but a key pair consisting of a public an private key because that is a more secure technique for authentication.

2.1.1 Generating Keys

The key generation can be done on both Linux, Windows and Mac. Here only the Linux way is described.

On all popular Linux distributions, the OpenSSH software is readily available, and most often installed by default. You can check whether the OpenSSH software is installed by opening a terminal and typing:

```
$ ssh -V
OpenSSH_6.9p1 Ubuntu-2ubuntu0.1, OpenSSL 1.0.2d 9 Jul 2015
```

To access the clusters and transfer your files, you will use the following commands:

- ssh: to generate the ssh keys and to open a shell on a remote machine,
- sftp: a secure equivalent of ftp,
- scp: a secure equivalent of the remote copy command rcp,
- rsync: an equivalent of scp, but the copy can resume when failed.

Usually you already have the software needed and a key pair might already be present in the default location inside your home directory:

```
$ ls ~/.ssh
authorized_keys2 id_rsa id_rsa.pub known_hosts
```

You can recognize a public/private key pair when a pair of files has the same name except for the extension ".pub" added to one of them. In this particular case, the private key is "id_rsa" and public key is "id_rsa.pub". You may have multiple keys (not necessarily in the directory "~/.ssh") if you or your operating system requires this. A popular alternative key type, instead of rsa, is dsa. However, we recommend to use rsa keys.

You will need to generate a new key pair, when:

- you don't have a key pair yet,
- you forgot the passphrase protecting your private key,
- or your private key was compromised.

To generate a new public/private pair, use the following command:

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/user/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/user/.ssh/id_rsa.
Your public key has been saved in /home/user/.ssh/id_rsa.pub.
```

This will ask you for a file name to store the private and public key, and a passphrase to protect your private key. It needs to be emphasized that you really should choose the passphrase wisely! The system will ask you for it every time you want to use the private key, that is every time you want to access the cluster or transfer your files.

Keys are required in the OpenSSH format.

2.1.2 Applying for an account

For most researchers from the Flemish universities (including the KULeuven), the procedure has been fully automated and works by using your institute account to request a VSC account. Open the VSC account management web site and select your "home" institution. After you log in using your institution login and password, you will be asked to upload your public key. You will get an e-mail to confirm your application. After the account has been approved by the VSC, your account will be created and you will get a confirmation e-mail.
The site account.vscentrum.be is only accessible from within your institution's domain.

2.2 Credits

The accounting system on the VSC is very similar to a regular bank. Individual users have accounts that will be charged for the jobs they run. However, the number of credits on such accounts is fairly small, so research projects will typically have one or more project accounts associated with them. Users that are project members can have their project-related jobs charged to such a project account.

There are 2 types of credits that can be requested:

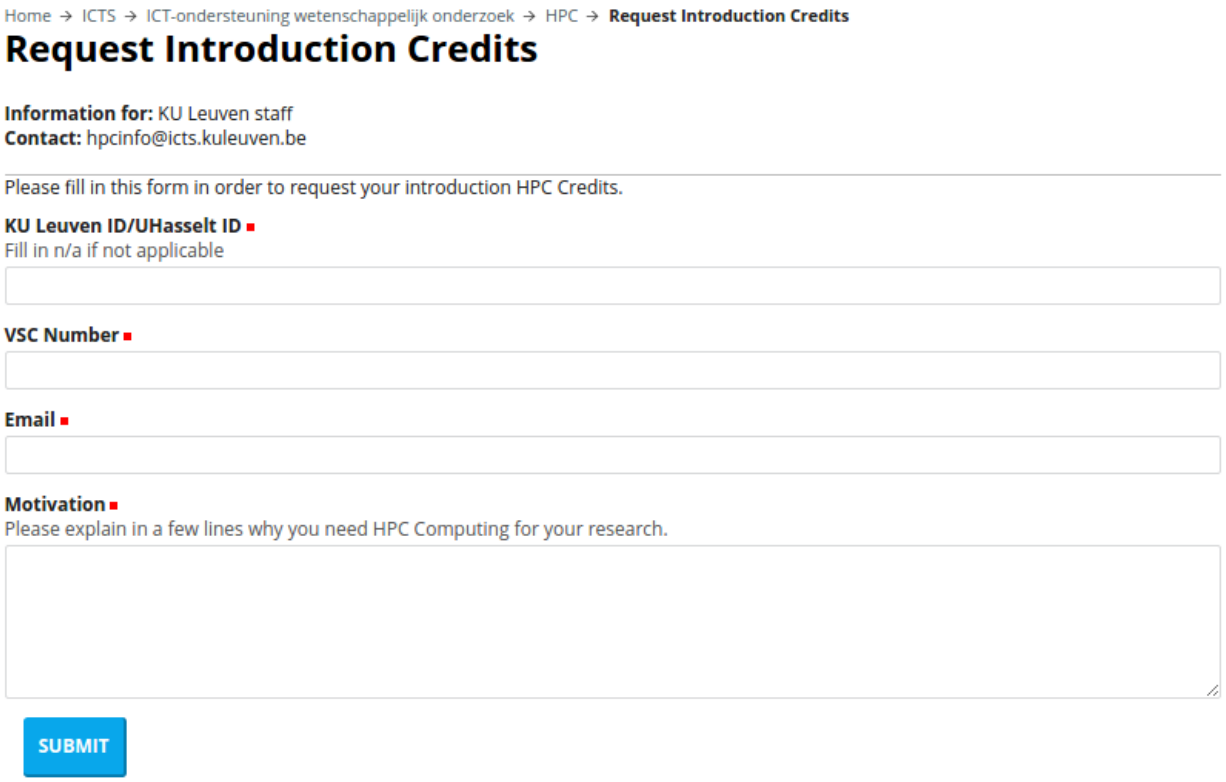
- introduction credits
- project credits

The use of the credits is discussed in the Credit Chapter.

2.2.1 Introduction credits

Introduction credits are a limited amount of free credits for test and development purposes.

They can be requested using: https://admin.kuleuven.be/icts/onderzoek/hpc/request-introduction-credits



2.2.2 Project credits

Project credits are job credits used for research.

The current prices (on 15th april 2016) are:

Credits Type	Price per 1000 credits
Introduction credits	Free
Project credits for European Projects	€ 120
Project credits with co-financing of commercial partners	€ 120
Project credits for other academic projects (internal, FWO, IWT)	€ 7

They can be requested using: https://admin.kuleuven.be/icts/onderzoek/hpc/request-project-credits. Project credits are always asked for a specific project. You will have to define a projectname first, and generate the appropriate group. Howto do this is discussed in the Account - Group Chapter.

Home → ICTS → ICT-ondersteuning wetenschappelijk onderzoek → HPC → **Request Project Credits**

Request Project Credits

Information for: KU Leuven staff
Contact: hpcinfo@icts.kuleuven.be

Please fill in this form to request your Project Credits.

For more information about conditions and pricing, please have a look at the HPC service catalog

If you have questions please contact hpcinfo@icts.kuleuven.be

KU Leuven ID ■
Responsible for the administration of the project on the HPC cluster (e.g. assignation of project members) (n/a for non K.U.L. members)

VSC Number ■
Responsible for the administration of the project on the HPC cluster (e.g. assignation of project members)

Email ■
Responsible for the administration of the project on the HPC cluster (e.g. assignation of project members)

Projectname (please create the corresponding group first) ■
The name should begin with 'p_' (KU Leuven users) or 'p_h_' (UHasselt users). This name should be the name of the group you should create first using the interface https://account.vscentrum.be/django. More info about creating groups you can find on: https://www.vscentrum.be/cluster-doc/account-management/manage-vsc-groups.

Project Description ■
Please describe your project and why you use HPC.

Number of credits ■
Estimation of how many HPC credits you need for your research project

Credit distribution over multiple years
(Optional) Describe here how you want to distribute your project credits over multiple years. (Example: Total 20.000 = 15.000 first year + 5.000 second year)

Project leader ■
K.U.Leuven ID or Full Name of the project leader (i.e. Prof. who administrates the financial budget).

Start date of project ■
Please choose the start date for the HPC calculations of your project.
-- / -- / -- -- : -- --

Financial funding ■
Please specify the source for the financial funding of your project.

Unit Number ■
unit number in the K.U.Leuven organisational chart for the billing (nummer van de organisatorische eenheid in het K.U.Leuven organigram voor de facturatie) (n/a for non K.U.L. Member)

I agree ■
Please check this checkbox if you agree to engage yourself to inform us about publications which are produced from results calculated on the HPC cluster at K.U.Leuven.

SUBMIT

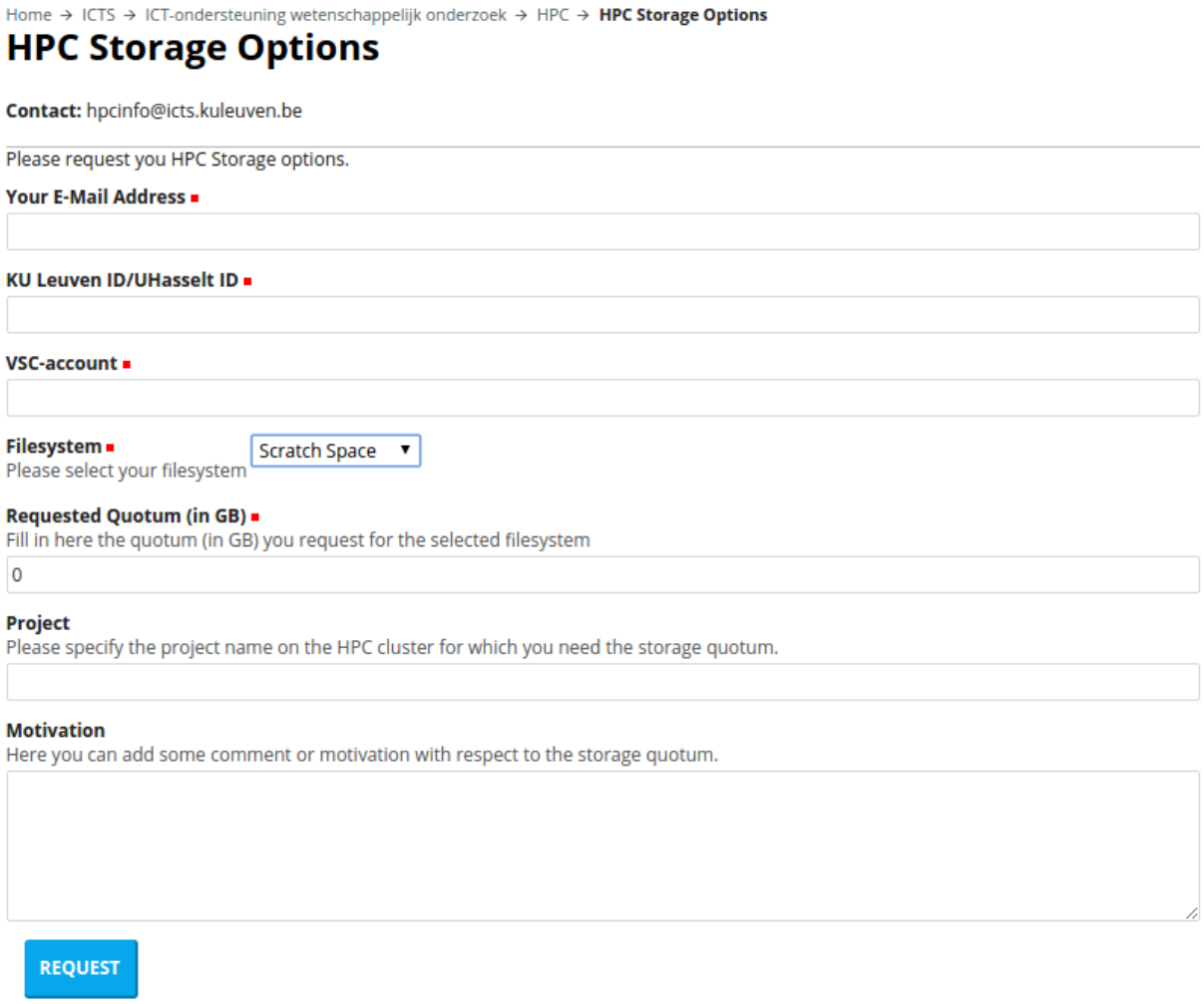
2.3 Storage

There are 3 types of storage that can be bought:

- Scratch Space
- Staging Space
- Archival Storage

These types are discussed in the Storage Chapter.

Storage can be requested via: https://admin.kuleuven.be/icts/onderzoek/hpc/hpc-storage. Storage is always asked for a specific project. You will have to define a projectname first, and generate the appropriate group. Howto do this is discussed in the Account - Group Chapter.



2.4 Billing

When you ask for credits or storage, and the request is accepted (typically within the week), an *Afsprakennota* is mailed to the person requesting it. This Nota describes who requested it, the Lab for billing, the projectname, and the requested credits or storage, with the price. If you accept this nota, you have to reply the mail with the message: *'Voor akkoord'*.

KU LEUVEN

ICTS Klanten- en servicebeheer
W. de Croylaan 52 B
3001 Heverlee

Afsprakennota AN00005456
High Performance Computing

Aanvrager
Username (u0XXXXX) / mail @kuleuven.be

Facturatie
Name of the laboratory

Beschrijving van de service

Uw Ref.: p_projectName

Service	*Prijs/Stuk*	*Aantal*	*Totaal*
High Performance Computing Credits (1.000)	7	20	€ 140,00

Totaal € 140

vsc3XXXX

Specifieke gebruiksvoorwaarden

- De connectie met de HPC-cluster gebeurt via ssh. De gebruiker moet over een ssh-client, met een publiek/privaat sleutelpaar, beschikken om toegang te krijgen tot de systemen.
- De gebruiker mag enkel commerciële software op de cluster installeren indien er aan de licentievoorwaarden van de software wordt voldaan en indien de software uitsluitend wordt gebruikt voor onderzoeksdoeleinden.
- De gebruiker is zelf verantwoordelijk voor het bewaren van zijn tussentijdse data bij langdurige jobs. Dit kan met behulp van de beschikbare KU Leuven software.
- De gebruiker dient zelf de ICTS statuspagina te raadplegen om de beschikbaarheid van de service na te kijken. Om hiervan pro-actief op de hoogte te worden gebracht, kan hij zich eventueel abonneren op de RSS feed.
- Overdracht van vervallen projectcredits kan enkel indien er nog voldoende capaciteit beschikbaar is.

3 Account and Groups

3.1 Login

The moment you get your user number, the login on the VSC is possible.

Login is simple, by opening a terminal and using:

```
$ ssh vsc3XXXX@login.hpc.kuleuven.be
Enter passphrase for key '/home/.ssh/id_rsa':
Last login: Thu Apr 14 15:51:13 2016 from bluecoat1.uzleuven.be
*********************************************
*                                           *
* Please check the following site for
*
* status messages concerning KU Leuven      *
* services (incl. HPC):                     *
*                                           *
*   http://status.kuleuven.be/              *
*                                           *
* For VSC user documentation:               *
*                                           *
*  https://www.vscentrum.be/en/user-portal  *
*                                           *
*                                           *
*********************************************
```


You will have to change vsc3XXXX to your own account number.

The ssh command (Secure Shell) creates a secure connection to the VSC. This command automatically checks your keys. If you entered a passphrase when generating the key, the passphrase will be asked. When the login was successful, the login message will be shown. After login, your current path will be your home directory (more about this in the Storage Chapter).

3.2 Groups

The VSC uses the Linux Group system. This means that users (accounts) belong to one or multiple groups. To request access and create groups, see below.
Besides users, groups are also used to control access on files and directories. Each file has one owner (a user), and belongs to one group (more about file permissions can be found in the Basic Linux Chapter).
Storage access is managed by groups (at the request of the storage, the group has to be mentioned, although the permissions can be changed).
Last the credits are managed by groups. Everyone in the group can use the credits. How to get an overview of the usage can be found in the Credits Chapter.

3.3 Account Managing

Your account is managed trough a web interface: https://account.vscentrum.be/django/.
In this interface are multiple tabs:

- View Account
- Edit Account
- View Groups
- New/Join Group
- Edit Group
- New/Join VO

3.3.1 View Account

This tab shows you an overview of your account details. You can also find the path to your home, data and scratch folder (more about this in the Storage Chapter).

3.3.2 Edit Account

This tab lets you edit your account information. You can choose your standard shell, research field or add public keys.

3.3.3 View Groups

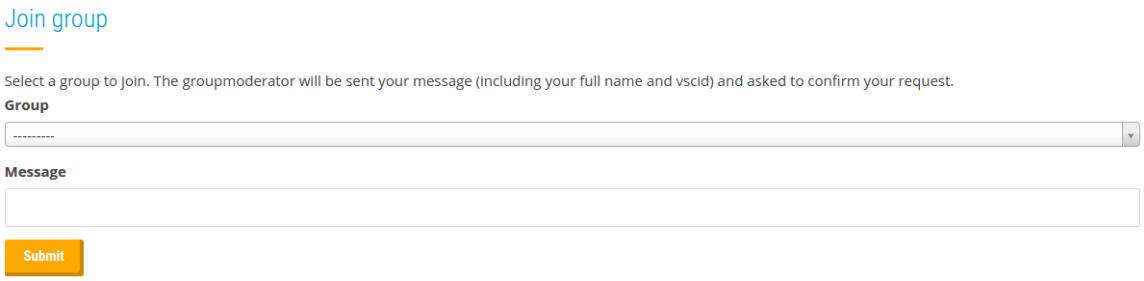
This tab gives an overview of every group you belong to. Users automatically belong to:

- leuvenall
  this is a mailing list, used for maintenance messages and information about training events
- vsc3XXXX
  (your vsc account number) All files you make belong automatically to this group. This group has only you as user, and no other users can be added.

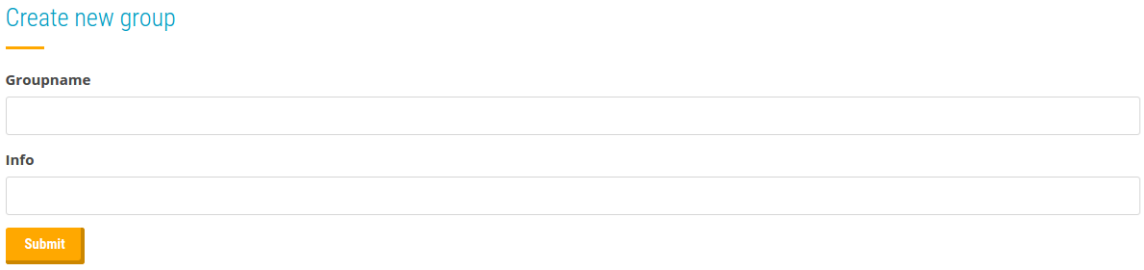
3.3.4 New/Join Group

This tab gives you 2 options:

- joining an existing group.
  Here you just select the group you wish to join, and the group moderator will get your request.



- create a new group.
  Here you can generate a new group. Groups created at the VSC on the KULeuven site, will automatically have the prefix l (from leuven). If you want to be able to use this group for requesting storage and credits, the group name has to start with p_ (p from project, and an underscore)



3.3.5 Edit Group

If you are a group moderator, you can choose which group you want to edit.
After selecting, the page will refresh. The member managing is done on this page. You can see the current members, remove them, invite users to your group, and add or remove moderators.

Edit |Groupname

General information

Group: |[redacted]
Institute: Leuven / Hasselt
Vsc_id: |[redacted]
Vsc_id_number: [redacted]
Status: active

Manage members

Members

× vsc3xxxx ([redacted]
× vsc3xxxx ([redacted]

Remove members by clicking the × in front of their names.

Invited users

× vsc3xxxx × vsc3xxxx

Invite new accounts by clicking in the field above and start typing their vsc id.
Selected members are already invited.
Remove invites by clicking the × in front of their names.

Moderators

× vsc3xxxx ([redacted]

Add moderators by clicking in the field above and start typing their name.
Remove moderators by clicking the × in front of their names.

Update

3.3.6 New/Join VO

A VO is a Virtual Organisation. This is only a feature usable for the UGent users.

4 Credits

The accounting system on the VSC is very similar to a regular bank. Individual users have accounts that will be charged for the jobs they run. However, the number of credits on such accounts is fairly small, so research projects will typically have one or more project accounts associated with them. Users that are project members can have their project-related jobs charged to such a project account.
You can request 2 types of job credits: introduction credits and project credits. Introduction credits are a limited amount of free credits for test and development purposes. Project credits are job credits used for research.
How to request these credits is discussed in the Account - Credits - Storage Request Chapter .

4.1 Checking an account balance

Since no calculations can be done without credits, it is quite useful to determine the amount of credits at your disposal. This can be done quite easily:

```
$ module load accounting
$ gbalance
```

This will provide an overview of the balance on the user's personal account, as well as on all project accounts the user has access to.

4.2 Obtaining a job quote

In order to determine the cost of a job, the user can request a quote. The gquote commands takes those options as the qsub command that are relevant for resource specification (-l, -q, -C), and/or, the PBS script that will be used to run the job. The command will calculate the maximum cost based on the resources that are requested, taking into account walltime, number of compute nodes and node type.

```
$ module load accounting
$ gquote -l nodes=3:ppn=4:ivybridge,pmem=2gb,walltime=48:00:00
```

This example has multiple parameters that can be changed:

- nodes=3:ppn=4:ivybridge
  This option request 3 nodes, 4 processors per node (ppn), of the ivybridge hardware (for an overview of the available systems, see the The Hardware Chapter).
  Note that when a node is used, all processors of this node are reserved, even if you specify less ppn then available.
- pmem=2gb
  This option requests 2 GB of memory (RAM) (for an overview of the available systems, see the The Hardware Chapter).
  This option can be used with gb and mb.
- walltime=48:00:00
  This option is the maximum time the job gets to run.

Note that when a queue is specified and no explicit walltime, the walltime used to produce the quote is the longest walltime allowed by that queue. Also note that unless specified by the user, gquote will assume the most expensive node type. This implies that the cost calculated by gquote will always be larger than the effective cost that is charged when the job finishes.

4.3 Obtaining an overview of transactions

A bank provides an overview of the financial transactions on your accounts under the form of statements. Similarly, the job accounting system provides statements that give the user an overview of the cost of each individual job. The following command will provide an overview of all transactions on all accounts the user has access to:

```
$ module load accounting
$ gstatement
```

However, it is more convenient to filter this information so that only specific projects are displayed and/or information for a specific period of time, e.g.,

```
$ gstatement -g lp_projectname -s 2015-09-01 -e 2015-09-30
```

This will show the transactions on the account for the lp_projectname group for the month September 2015.

Note that it takes quite a while to compute such statements, so please be patient.

Possible options:

- -g groupname
- -a accountname
- -u username
- -s start
- -e end
- --summarize

Very useful can be adding the '--summarize' option to the 'gstatement' command:

```
$ gstatement -g lp_projectname --summarize
################################################################################
#
# Includes fund 1558 (Account=lp_projectname)
# Includes fund 1506 (Account=default_project,User=vsc31439)
# Generated on Fri Apr 15 12:40:29 2016.
# Reporting fund activity from -Infinity to Now.
#
################################################################################
Beginning Balance:                    0
------------------ --------------------
Total Credits:                        0
Total Debits:                     -1152
------------------ --------------------
Ending Balance:                   20848
Warning: A discrepancy of                -22000 credits was
detected
between the logged transactions and the historical fund balances.
################################## Credit Summary
##################################
################################## Debit Summary
##################################
Object      Action Account         User     Machine  Amount Count
----------- ------ --------------- -------- -------- ------ -----
UsageRecord Charge lp_projectname  vsc31439 thinking   -124 25
UsageRecord Charge default_project vsc31439 cerebro  -1003 10
UsageRecord Charge lp_projectname  vsc31439 cerebro    -25 3
################################## End of Report
##################################
```

As you can see it will give you a summary of used credits (Amount), used platform (Machine), used group (for the credits) (Account) and number of jobs (Count) per user in a given timeframe for a specified project.

4.4 Reviewing job details

A statement is an overview of transactions, but provides no details on the resources the jobs consumed. However, the user may want to examine the details of a specific job. This can be done using the following command:

```
$ module load accounting
$ glsjob -J 20030021.icts-p-svcs-1
```

Note that the job ID *must* be complete, as opposed to just the job number as for most queue related tasks.

4.5 Job cost calculation

The cost of a job depends on the resources it consumes. Generally speaking, one credit buys the user one hour of walltime on one reference node. The resources that are taken into account to charge for a job are the walltime it consumed, and the number and type of compute nodes it ran on. The following formula is used:

```
(0.000278*nodes*walltime + startup)*nodetype
```

Here,

- *nodes* is the number of compute nodes the job ran on;
- *walltime* the effective duration of the job, expressed in seconds;
- *nodetype* is the factor representing the node type's performance as listed in the table below;
- *startup* is 0.1, it is added to account for the overhead of scheduling (reserved nodes can be idle for a while until the total number of nodes required are available). It is roughly equivalent to a walltime of 6 minutes.

The VSC has no reference node. The cost of computing, and which nodes are available can be found in the The Hardware Chapter.

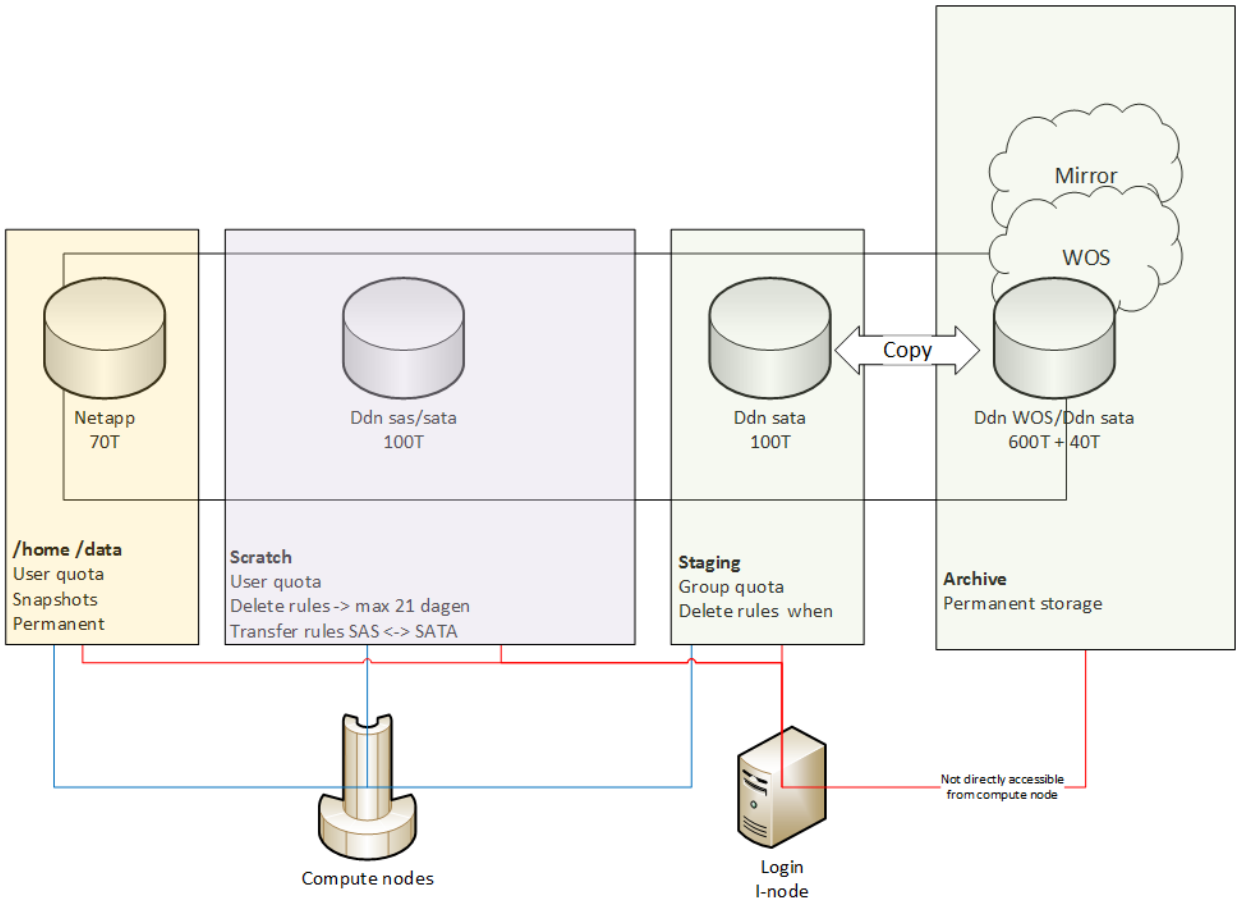
Note that charging is done for the number of compute nodes used by the job, not the number of cores. This implies that a single core job on a single node is as expensive as an 8 core job on the same single node. The rationale is that the scheduler instates a single user per node policy. Hence using a single core on a node blocks all other cores for other users' jobs. If a user needs to run many single core jobs concurrently, she is advised to use the Worker framework.

5 Storage

Storage is an important part of a cluster. But not all the storage has the same characteristics. HPC cluster storage at KU Leuven consists of 3 different storage Tiers, optimized for different usage

- NAS storage, fully back-up with snapshots for /home and /data
- Scratch storage, fast parallel filesystem
- Staging storage, to store current working projects
- Archive storage, to store large amounts of data for long time

The picture below gives a quick overview of the different components.



5.1 Home Directory

This directory is where you arrive by default when you login to the cluster. Your shell refers to it as "~" (tilde), or via the environment variable $VSC_HOME.

The data stored here should be relatively small (e.g., no files or directories larger than a gigabyte, although this is allowed), and usually used frequently. Also all kinds of configuration files are stored here, e.g., by Matlab, Eclipse, ...

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

.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're doing!
.profile	This script defines some general settings about your sessions,
.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. You could edit this file and, e.g., add "module load XYZ" if you want to automatically load module XYZ whenever you login to the cluster, although we do not recommend to load modules in your .bashrc.
.bash_history	This file contains the commands you typed at your shell prompt, in case you need them again.

This directory has a back-up.

The full path to the directory is: /user/leuven/3XX/vsc3XXXX

The environment variable is: $VSC_HOME

5.2 Data Directory

In this directory you can store all other data that you need for longer terms. The environment variable pointing to it is $VSC_DATA. There are no guarantees about the speed you'll achieve on this volume. This directory has a back-up. It can be used for higher I/O loads, but for I/O bound jobs, you might be better of using one of the 'scratch' filesystems.

The full path to the directory is: /data/leuven/3XX/vsc3XXXX

The environment variable is: $VSC_DATA

5.3 Scratch Space

To enable quick writing from your job, a few extra file systems are available on the work 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). There is no backup for these filesystems, and 'old' data may be removed automatically.

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 these are automatically cleaned 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.

Each type of scratch has his own use:

- Node scratch ($VSC_SCRATCH_NODE)
  Every node has its own scratch space, which is completely separated from the other nodes. On many cluster nodes, this space is provided by a local hard drive or SSD. Every job automatically gets its own temporary directory on this node scratch, available through the environment variable $TMPDIR. $TMPDIR is guaranteed to be unique for each job.
  Note however that when your job requests multiple cores and these cores happen to be in the same node, this $TMPDIR is shared among the cores! Also, you cannot access this space once your job has ended. And on a supercomputer, a local hard disk may not be faster than a remote file system which often has tens or hundreds of drives working together to provide disk capacity.

- Site scratch ($VSC_SCRATCH_SITE, $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.
  This scratch space is provided by a central file server that contains tens or hundreds of disks. Even though it is shared, it is usually very fast as it is very rare that all nodes would do I/O simultaneously. It also implements a parallel file system that allows a job to do parallel file I/O from multiple processes to the same file simultaneously, e.g., through MPI parallel I/O.
  For most jobs, this is the best scratch system to use.

The full path to the directory is: /scratch/leuven/3XX/vsc3XXXX

The environment variable is: $VSC_SCRATCH

Other environment variables to scratch which are not acessable from the login node are: $VSC_SCRATCH_NODE, $VSC_SCRATCH_SITE

5.4 Staging Space

The staging area is a big workspace for the storage of big data over a longer period. It is a good practice to back things up to the archive. The staging space is available from all nodes. The staging area is a part of the same hardware platform as the fast scratch filesystem, but other rules apply. Data is not deleted automatically after 21 days. When the staging area is full it will be the user's responsibility to make sure that enough space is available.

The full path to the directory is: /staging/leuven/stg_000XX

There is no environment variable.

5.5 Archive Storage

The archive is intended to store data for longer term. The storage is optimized for capacity, not for speed. The storage by default is mirrored. No deletion rules are executed on this storage. The data will be kept until the user deletes it. Data created on scratch or in the staging location which needs to be kept for longer time should be copied to the archive. The archive is only accessible through the login nodes.

The full path to the directory is: /archive/leuven/arc_000XX

There is no environment variable.

5.6 Price

Storage Type	Price	Expendable	Initial Size
Home Directory	Free	No	25GB
Data Directory	€ 12.16 per 25GB per year	Yes	75GB
Scratch Directory (personal scratch)	Free	Only temporary extensions	100GB
Staging Storage	€ 70 per 1TB per year	Yes	0
Archive Storage	€ 70 per 1TB per year	Yes	0

Shown prices are on 15/04/2016 and can change.

5.7 Moving Data

Moving data is never a good idee, when the move is aborted, data can become incomplete. Copy (with or without delete afterwards) is better to avoid incomplete files.

In Linux there are 2 widely used ways to copy:

1. cp and scp
2. rsync

The cp (copy command) is used to copy data within the file system. The scp (secure copy command) is used to copy data between servers (like upload data from a local computer to the HPC). While these commands are widely used, they have some disadvantages. The biggest one is, if the copy fails or get aborted, the copy has to start again from the beginning. The solution is the use of rsync. rsync (remote synchronization) is an advanced copy command, which can resume if a file is only partially copied, and also performs automatic checks (if the data is complete) when the copy ends. rsync can be used in the same way as cp and scp. An useful example is given here, where all content of the current directory is copied to the Home directory (see Storage) of the user.


rsync(1) --progress --copy-links -ahr * vsc3XXXX@login.hpc.kuleuven.be:/user/leuven/XXX/vsc3XXXX
a fast, versatile, remote (and local) file-copying tool
--progress
This option tells rsync to print information showing the progress of the transfer. This gives a bored user something to watch. Implies --verbose if it wasn't already specified.
While rsync is transferring a regular file, it updates a progress line that looks like this:
782448 63% 110.64kB/s 0:00:04
In this example, the receiver has reconstructed 782448 bytes or 63% of the sender's file, which is being reconstructed at a rate of 110.64 kilobytes per second, and the transfer will finish in 4 seconds if the current rate is maintained until the end.
These statistics can be misleading if rsync's delta-transfer algorithm is in use. For example, if the sender's file consists of the basis file followed by additional data, the reported rate will probably drop dramatically when the receiver gets to the literal data, and the transfer will probably take much longer to finish than the receiver estimated as it was finishing the matched part of the file.
When the file transfer finishes, rsync replaces the progress line with a summary line that looks like this:
1238099 100% 146.38kB/s 0:00:08 (xfer#5, to-check=169/396)
In this example, the file was 1238099 bytes long in total, the average rate of transfer for the whole file was 146.38 kilobytes per second over the 8 seconds that it took to complete, it was the 5th transfer of a regular file during the current rsync session, and there are 169 more files for the receiver to check (to see if they are up-to-date or not) remaining out of the 396 total files in the file-list.
-L, --copy-links
When symlinks are encountered, the item that they point to (the referent) is copied, rather than the symlink. In older versions of rsync, this option also had the side-effect of telling the receiving side to follow symlinks, such as symlinks to directories. In a modern rsync such as this one, you'll need to specify --keep-dirlinks (-K) to get this extra behavior. The only exception is when sending files to an rsync that is too old to understand -K -- in that case, the -L option will still have the side-effect of -K on that older receiving rsync.

```
-a, --archive
       This is equivalent to -rlptgoD. It is a quick way of saying you want recursion and want to
       preserve almost everything (with -H being a notable omission). The only exception to the above
       equivalence is when --files-from is specified, in which case -r is not implied.

       Note that -a does not preserve hardlinks, because finding multiply-linked files is expensive. You
       must separately specify -H.

-h, --human-readable
       Output numbers in a more human-readable format. This makes big numbers output using larger units,
       with a K, M, or G suffix. If this option was specified once, these units are K (1000), M
       (1000*1000), and G (1000*1000*1000); if the option is repeated, the units are powers of 1024
       instead of 1000.

-r, --recursive
       This tells rsync to copy directories recursively. See also --dirs (-d).

       Beginning with rsync 3.0.0, the recursive algorithm used is now an incremental scan that uses much
       less memory than before and begins the transfer after the scanning of the first few directories
       have been completed. This incremental scan only affects our recursion algorithm, and does not
       change a non-recursive transfer. It is also only possible when both ends of the transfer are at
       least version 3.0.0.

       Some options require rsync to know the full file list, so these options disable the incremental
       recursion mode. These include: --delete-before, --delete-after, --prune-empty-dirs, and
       --delay-updates. Because of this, the default delete mode when you specify --delete is now
       --delete-during when both ends of the connection are at least 3.0.0 (use --del or --delete-during
       to request this improved deletion mode explicitly). See also the --delete-delay option that is a
       better choice than using --delete-after.

       Incremental recursion can be disabled using the --no-inc-recursive option or its shorter --no-i-r
       alias.
```

```
Local:  rsync [OPTION...] SRC... [DEST]

Access via remote shell:
  Pull: rsync [OPTION...] [USER@]HOST:SRC... [DEST]
  Push: rsync [OPTION...] SRC... [USER@]HOST:DEST

Access via rsync daemon:
  Pull: rsync [OPTION...] [USER@]HOST::SRC... [DEST]
        rsync [OPTION...] rsync://[USER@]HOST[:PORT]/SRC... [DEST]
  Push: rsync [OPTION...] SRC... [USER@]HOST::DEST
        rsync [OPTION...] SRC... rsync://[USER@]HOST[:PORT]/DEST
```

5.7.1 Check Storage Usage

There are 2 easy ways to check the disk usage:

- du
  where you can check the size per directory
- df
  where you can check the size of a partition (ideal for staging and archive)

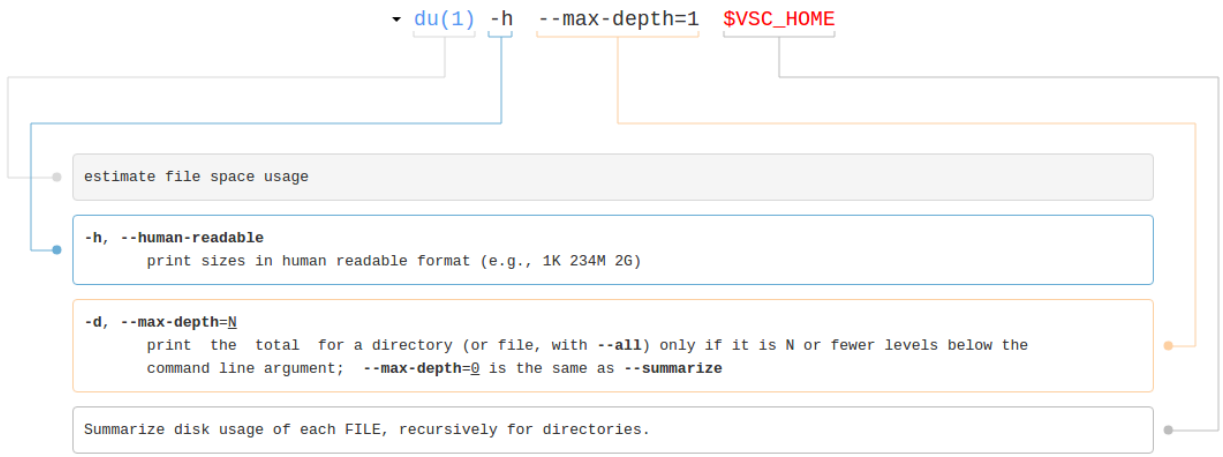
du

du is a standard linux command to check the size of the content of a directory. For example executed in the home directory:

```
$ du -h --max-depth=1 $VSC_HOME
20K    /user/leuven/314/vsc31439/.ssh
8.0K    /user/leuven/314/vsc31439/.vim
28K    /user/leuven/314/vsc31439/intel
12K    /user/leuven/314/vsc31439/.gnome2
3.2M    /user/leuven/314/vsc31439/.cache
4.1M    /user/leuven/314/vsc31439/.mozilla
4.0K    /user/leuven/314/vsc31439/Desktop
8.0K    /user/leuven/314/vsc31439/.parallel
8.0K    /user/leuven/314/vsc31439/.oracle_jre_usage
7.4M    /user/leuven/314/vsc31439
```

The output shows 2 columns: in the first the size of the file or directory, in the second the file or directory name.

The du command explained:



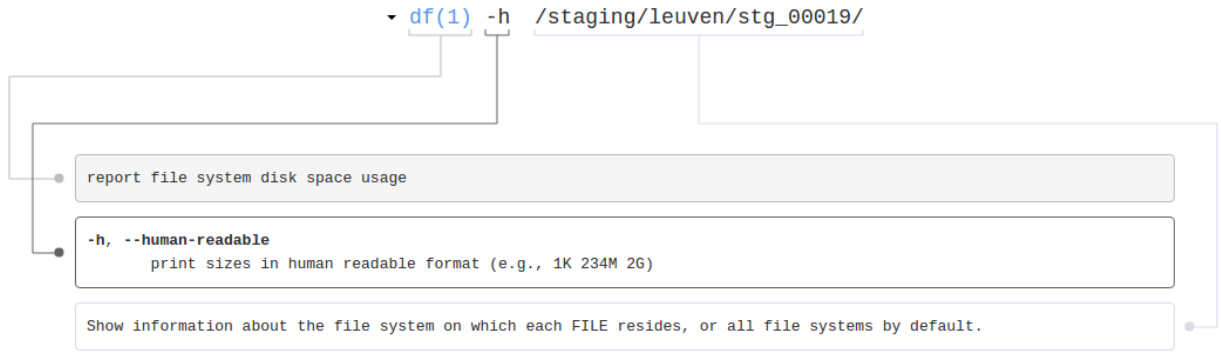
df

df is a standard linux command to check the size of a partition. This command is ideal to check the size of the staging or archive for a project. As example on a staging partition:

```
$ df -h /staging/leuven/stg_000XX/
Filesystem      Size  Used Avail Use% Mounted on
/dev/vol_ddn2   5.0T  1.1T  4.0T  22% /ddn1/vol1
```

The output shows multiple columns including: the mounting location (FileSystem), the current size (Size), the used size (Used), the free space (Avail) and the usage in percentage (Use%).

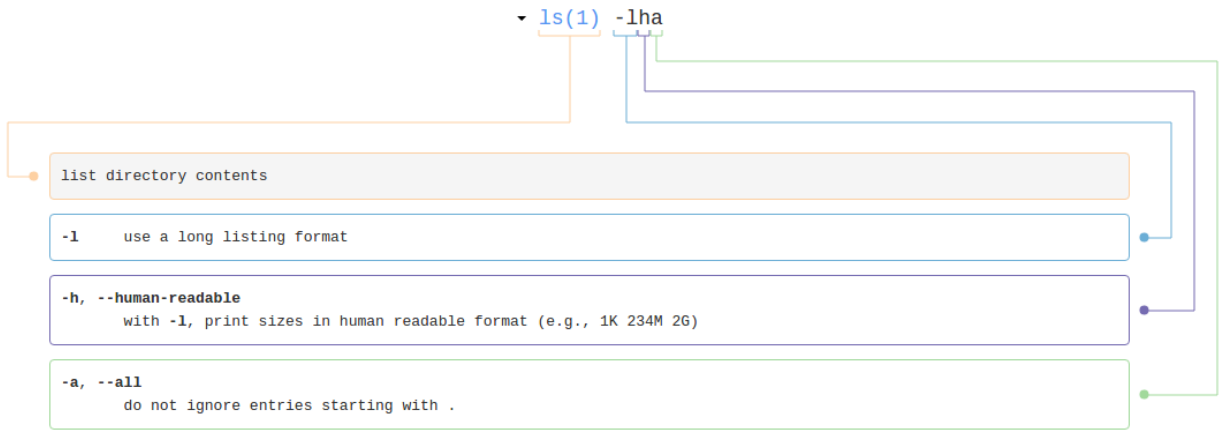
The df command explained:



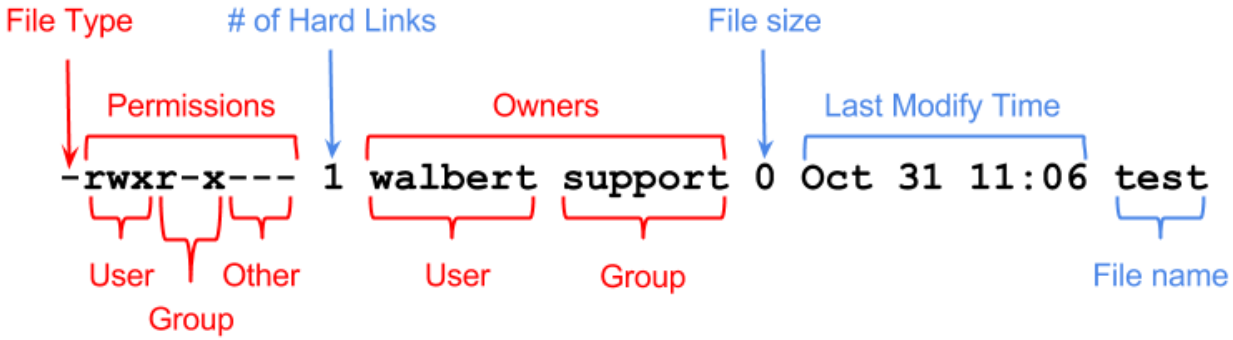
5.8 File Permission

File permissions can easy be checked with the ls command.

```
$ ls -lha
```



The output of this command is ordered like:



Where the most important fields are:

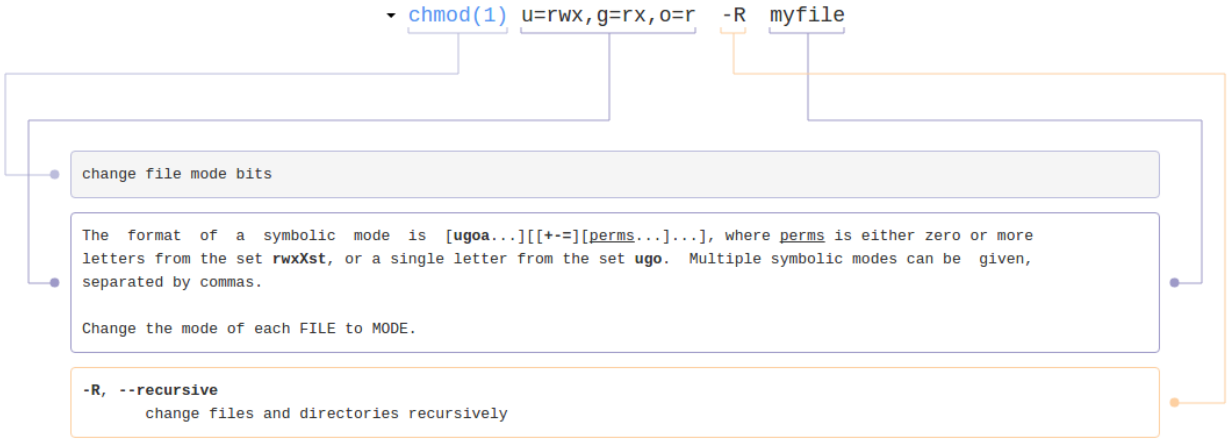
- File Type
  This field is - for a file, d for a directory
- Permissions
  permissions are given as r (read), w (write), x (execute) or - this permission is not available (also see image below).
- User
  The name of the owner
- Group
  The name of the group (in the group permission field)
- File size
  The size of the file, in case of a directory this is NOT the size of the directory. This can be measured using du
- Last Modify Time
  The last date and time of the last modification
- File name
  The name of the file or directory

5.8.1 chmod

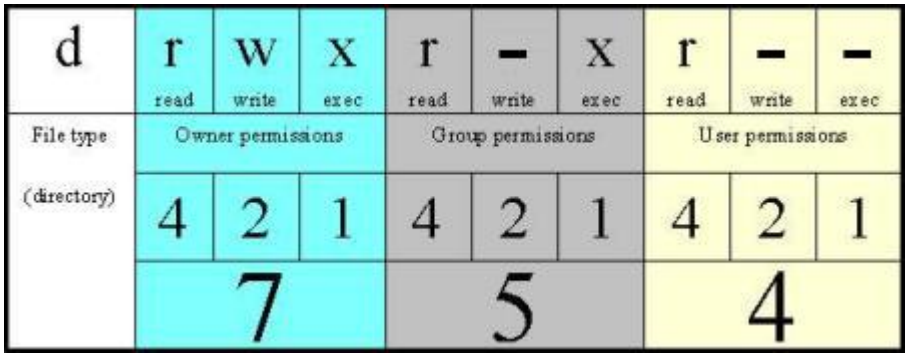
Permissions of files and directories can be changed using chmod. There are 2 possibilities to change permissions:

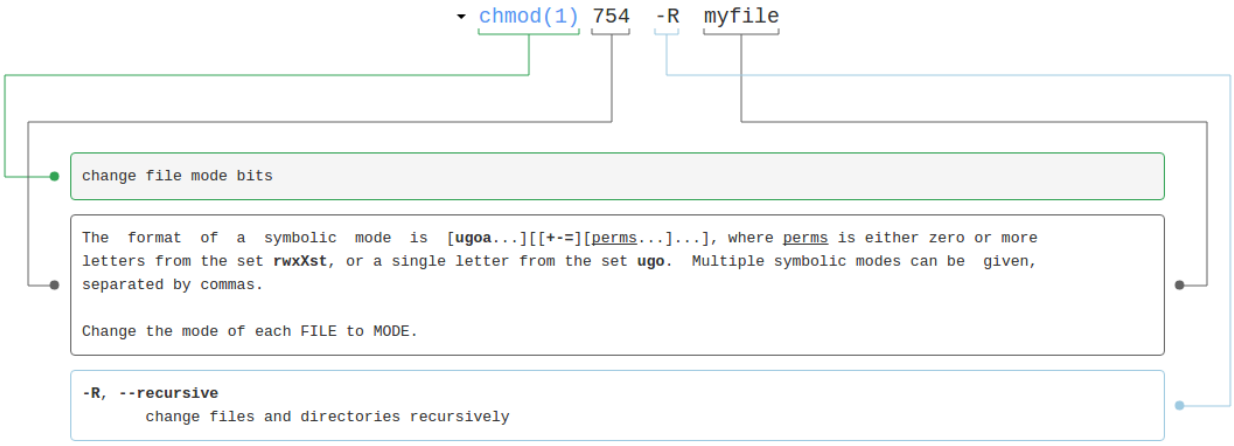
```
$ chmod u=rwx,g=rx,o=r -R myfile
$ chmod 754 -R myfile
```

The first option is to specify the user (u), group (g) or others (o) equals the permissions they get read (r), write (w) or execute (x). Note that write permission means change, but also delete.



Another method can be the use of the binary option. The next images explain this feature.

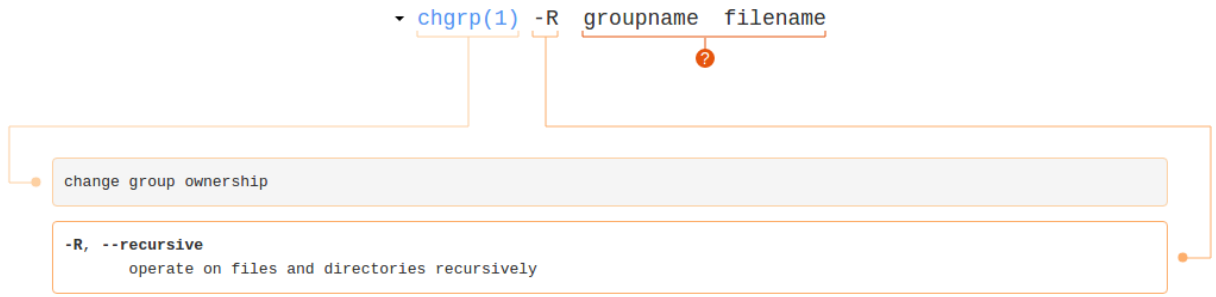




5.8.2 chgrp

In the Linux system files belong to groups. On the HPC, files automatically belongs to the user group. However, this group will always count 1 member: the user. When data has to be shared, but not with everyone, a group should be created (how to create groups and add members can be found in Account and Groups). After data is generated, the group access will need to change (with chmod), as well as the group (with chgrp). Note, this can only be done when the user belongs to the group.

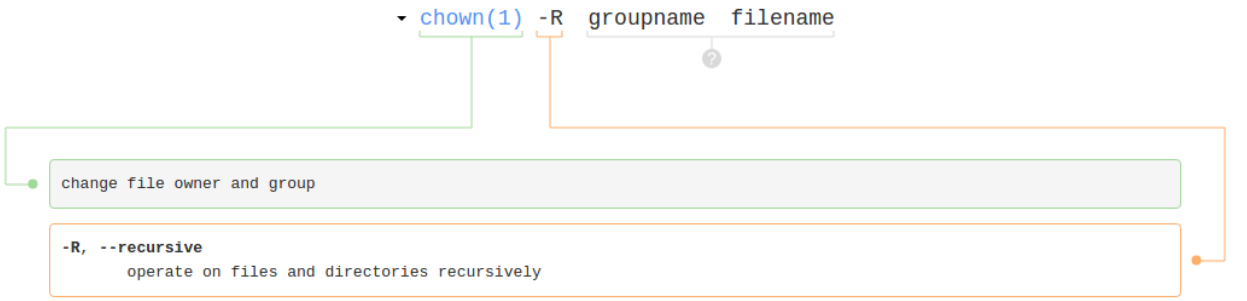
```
$ chgrp -R groupname filename
```



5.8.3 chown

As last, the owner of the file can be changed as well. Note, this can only be done if the new user is in the same group as the file.

```
$ chown -R username filename
```



5.8.4 Summary

Storage Type	Path	Environment Variable	Standard Size	Usage
Home	/user/leuven/3XX/vsc3XXXX	$VSC_HOME	25GB	Important data, like configuration files. Is private.
Data	/data/leuven/3XX/vsc3XXXX	$VSC_DATA	75GB	Important data, biger data. Is private.
Scratch	/scratch/leuven/3XX/vsc3XXXX	$VSC_SCRATCH	100GB	Temporary data, will be deleted within 21 days.
Node Scratch		$VSC_SCRATCH_NODE	machine dependeble, min 150GB	Temporary data, while job is running. Can not be accessed from the login node. Data is lost at the jobs end.
Staging	/staging/leuven/stg_000XX		project dependable, per 1TB	Storage for project files, while still working at the project.
Archive	/archive/leuven/arc_000XX		project dependable, per 1TB	Backup for project files (or staging), long term storage. Only accessible from the login node.

6 The Module System

6.1 Software stack

Software installation and maintenance on HPC infrastructure such as the VSC clusters poses a number of challenges not encountered on a workstation or a departemental cluster. For many libraries and programs, multiple versions have to installed and maintained as some users require specific versions of those. And those libraries or executable sometimes rely on specific versions of other libraries, further complicating the matter.

The way Linux finds the right executable for a command, and a program loads the right version of a library or a plug-in, is through so-called environment variables. These can, e.g., be set in your shell configuration files (e.g., .bashrc), but this requires a certain level of expertise. Moreover, getting those variables right is tricky and requires knowledge of where all files are on the cluster. Having to manage all this by hand is clearly not an option.

The VSC clusters deal with this in the following way. First, the concept of a toolchain is defined on most of the newer clusters. They consist of a set of compilers, MPI library and basic libraries that work together well with each other, and then a number of applications and other libraries compiled with that set of tools and thus often dependent on those. Tool chains are used based on the Intel and GNU compilers, and are refresh twice a year, leading to version numbers like 2014a, 2014b or 2015a for the first and second refresh of a given year. Some tools are installed outside a toolchain, e.g., additional versions requested by a small group of users for specific experiments, or tools that only depend on basic system libraries. Second, the module system is used to manage the environment variables and all dependencies and possible conflicts between various programs and libraries., and that is what this chapter focuses on.

6.2 Basic use of the module system

Many software packages are installed as modules. These packages include compilers, interpreters, bioinformatical software such as Bowtie and samtools, as well as other applications and libraries. This is managed with the `module` command.

To view a list of available software packages, use the command `module av`. The output will look similar to this:

```
$ module av
----- /apps/leuven/thinking/2014a/modules/all ------
Autoconf/2.69-GCC-4.8.2
Autoconf/2.69-intel-2014a
Automake/1.14-GCC-4.8.2
Automake/1.14-intel-2014a
BEAST/2.1.2
...
pyTables/2.4.0-intel-2014a-Python-2.7.6
timedrun/1.0.1
worker/1.4.2-foss-2014a
zlib/1.2.8-foss-2014a
zlib/1.2.8-intel-2014a
```

This gives a list of software packages that can be loaded. Some packages in this list include `intel-2014a` or `foss-2014a`in their name. These are packages installed with the 2014a versions of the toolchains based on the Intel and GNU compilers respectively. The other packages do not belong to a particular toolchain. The name of the packages also includes a version number (right after the /) and sometimes other packages they need.

A module is loaded using the command `module load` with the name of the package. E.g., with the above list of modules,

```
$ module load BEAST
```

will load the `BEAST/2.1.2` package.

For some packages, e.g., `zlib` in the above list, multiple versions are installed; the `module load` command will automatically choose the lexicographically last, which is typically, but not always, the most recent version. In the above example,

```
$ module load zlib
```

will load the module `zlib/1.2.8-intel-2014a`. This may not be the module that you want if you're using the GNU compilers. In that case, the user should specify a particular version, e.g.,

```
$ module load zlib/1.2.8-foss-2014a
```

Obviously, the user needs to keep track of the modules that are currently loaded. If he executed the two load commands stated above, he will get something very similar to:

```
$ module list
Currently Loaded Modulefiles:
  1) /thinking/2014a
  2) Java/1.7.0_51
  3) icc/2013.5.192
  4) ifort/2013.5.192
  5) impi/4.1.3.045
  6) imkl/11.1.1.106
  7) intel/2014a
  8) beagle-lib/20140304-intel-2014a
  9) BEAST/2.1.2
 10) GCC/4.8.2
 11) OpenMPI/1.6.5-GCC-4.8.2
 12) gompi/2014a
 13) OpenBLAS/0.2.8-gompi-2014a-LAPACK-3.5.0
 14) FFTW/3.3.3-gompi-2014a
 15) ScaLAPACK/2.0.2-gompi-2014a-OpenBLAS-0.2.8-LAPACK-3.5.0
 16) foss/2014a
 17) zlib/1.2.8-foss-2014a
```

It is important to note at this point that, e.g., `icc/2013.5.192` is also listed, although it was not loaded explicitly by the user. This is because `BEAST/2.1.2` depends on it, and the system administrator specified that the `intel` toolchain module that contains this compiler should be loaded whenever the `BEAST` 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!

To unload a module, one can use the `module unload` command. It works consistently with the `load` command, and reverses the latter's effect. One can however unload automatically loaded modules manually, to debug some problem.

```
$ module unload BEAST
```

Notice that the version was not specified: the module system is sufficiently clever to figure out what the user intends. However, checking the list of currently loaded modules is always a good idea, just to make sure...

In order to unload all modules at once, and hence be sure to start with a clean slate, use:

```
$ module purge
```

It is a good habit to use this command in PBS scripts, prior to loading the modules specifically needed by applications in that job script. This ensures that no version conflicts occur if the user loads module using his `.bashrc` file.

Finally, modules need not be loaded one by one; the two 'load' commands can be combined as follows:

```
$ module load  BEAST/2.1.2  zlib/1.2.8-foss-2014a
```

This will load the two modules and, automatically, the respective toolchains with just one command.

To get a list of all available module commands, type:

```
$ module help
```

6.2.1 Getting even more software

The list of software available on a particular cluster can be unwieldingly long and the information that `module av` produces overwhelming. Therefore the administrators may have chose to only show the most relevant packages by default, and not show, e.g., packages that aim at a different cluster, a particular node type or a less complete toolchain. Those additional packages can then be enabled by loading another module first. On ThinKing, the KU Leuven cluster, the most complete and most used toolchains are the 2014a versions. Hence only the list of packages in those releases of the `intel` and `foss` (GNU) toolchain are shown at the time. Yet

```
$ module av
```

returns at the end of the list:

```
...
-------------------------- /apps/leuven/etc/modules/
--------------------------
cerebro/2014a  K20Xm/2014a    phi/2014a
ictstest/2014a M2070/2014a    thinking/2014a
```

shows modules specifically for the thin node cluster ThinKing, the SGI shared memory system Cerebro, two types of NVIDIA GPU nodes and the Xeon Phi nodes. Loading one of these will show the appropriate packages in the list obtained with`module av`. E.g.,

```
$ module load cerebro/2014a
```

will make some additional modules available for Cerebro, including two additional toolchains with the SGI MPI libraries to take full advantage of the interconnect of that machine.

6.2.2 Explicit version numbers

As a rule, 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 behavior.

Consider the following example: the user decides to use the GSL library for numerical computations, and at that point in time, just a single version 1.15, compiled with the foss toolchain is installed on the cluster. The user loads the library using:

```
$ module load GSL
```

rather than

```
$ module load GSL/1.15-foss-2014a
```

Everything works fine, up to the point where a new version of GSL is installed, e.g., 1.16 compiled with both the `intel` and the `foss` toolchain. From then on, the user's load command will load the latter version, rather than the one he intended, which may lead to unexpected problems.

7 The Hardware

The selection of the system to use determines the cost and run time of the job. The calculation of how many credits a job will take is shown in the Credits chapter, and therefor it is not discussed here. This chapter is about when to select which kind of cluster/partition for which job, and is therefore almost completely hardware related.

The table below shows the most important hardware differences between all partitions. The VSC has 2 main clusters (actually there are also the GPU nodes, but currently there are no GPU bioinformatics applications). These 2 clusters, called ThinKing and Cerebro, are the most used for bioinformatics applications. Most applications can run on both setups, but not both setups are interesting. Structural and hardware wise, there are big differences between ThinKing and Cerebro, and this has an effect to the bioinformatics applications.

7.1 ThinKing

ThinKing is a cluster with thin, powerful nodes (King of the Thin nodes). There are 2 partitions: the Haswell and the Ivybridge. Both partitions are named after there CPU type. The Ivybridge is the 'older' version of the CPU-types. This also means that the Ivybridge is a bit slower. Both Ivybridge as Haswell come in a 64GB and 128GB RAM version. Note that the OS also needs place to work and be stored, therefore a good practice is to reserve 4GB for this (however the complete 4GB will not be used by the OS). Third, the Ivybridge has only 20 cores, while the Haswell has 24 cores. This means that the Haswell can run 4 more threads per node then the Ivybridge. An other difference is the internal disk or the node scratch. The Ivybridge has more node scratch then the Haswell. As last, the Haswell is more expensive (6.68 vs 4.76 credits /hour).

So which partition to use for a task? As a start the Haswell is faster, has more cores (so more threads), but a smaller node scratch. The Ivybridge is a bit slower, has less cores, but a bigger scratch. So for speed operations, Haswell is the winner. But most bioinformatic applications are not about speed, they are about big datasets, I/O. Both ThinKing partitions share the same hardware to access the storage, so are equally fast. As of I/O is always slower then computing, the advantage of the fast Haswell is lost over the Ivybridge. This means that the only big difference between the 2 is node scratch and number of threads. A script does not have to use the node scratch, and can use the users scratch (see the Storage chapter). Therefore the only factor is the number of cores. It is clear that the Ivybridge (4.76 credits/hour) is cheaper, when the number of cores is 20 or below. Haswell (6.68 credits/hour) is cheaper for 21 to 24 cores, because you need 2 Ivybridge nodes to accomplish this. But for even more cores, Ivybridge will always be the cheapest.

As a result: the Haswell partition is the most powerful partition, but since bioinformatics is about big datasets (and a lot of I/O), the Ivybridge will almost always be more interesting. Typical tasks performed on the ThinKing cluster are: read mapping, read counting, variant calling, ...

7.2 Cerebro

Cerebro is a cluster with shared memory (Cerebro, Spanish for brain). There are 2 partitions: Smp1 and Smp2. The only difference between these 2 is the usable memory: 250GB vs 124GB per node. The advantage of this cluster is that within a partition, you can combine the memory between nodes. This means, if you have a program with only 1 thread, but need 500GB RAM, you reserve 2 nodes on Smp1, and use the complete 500GB for only this thread. This cluster uses the same hardware to access the storage as ThinKing. A node has only 10 cores, so running tasks that can run on ThinKing are expensive here.

As a result: Cerebro can combine the RAM over nodes (shared), so task with high memory usage can run here. Typical tasks performed on the Cerebro cluster are: assemblies.

7.3 Summary

The table gives a hardware overview of the different clusters and their partitions.

Cluster	Partition	CPU-Type	#nodes	#cores (threads) per node	Internal disk (Node Scratch)	Useable Memory (RAM) per node	#credits /hour
ThinKing	Ivybridge	Ivybridge	208	20	250GB	60-124GB	4.76
ThinKing	Haswell	Haswell	144	24	150GB	60-124GB	6.68
Cerebro	Smp1	Ivybridge	48	10	shared 10TB	shared 250GB (max 11.77 TB)	3.45
Cerebro	Smp2	Ivybridge	16	10	shared 10TB	shared 124GB (max 1.79 TB)	3.45

Typical tasks run on the different clusters/partitions:

Cluster	Partition	Task Description	Task Example
ThinKing	Ivybridge	Memory low jobs, with lots of I/O	Alignment, Read Mapping, Variant Calling, Read Counting, ...
ThinKing	Haswell	Memory low jobs, low I/O, high computing power needed	Model Calculations, Discovering a new Star, ...
Cerebro	Smp1 and Smp2	High memory jobs, computing power less important	De Novo Assemblies, Reference-based Assemblies, ...

8 Portable Batch System

> **Portable Batch System** (or simply **PBS**) is the name of computer software that performs job scheduling. Its primary task is to allocate computational tasks, i.e., batch jobs, among the available computing resources. It is often used in conjunction with UNIX cluster environments. PBS is supported as a job scheduler mechanism by several meta schedulers ...
>
> -- by Wikipedia

8.1 PBS script

Every bash (.sh) script can be turned into a PBS (.pbs) script. A typical script needs the following adjustments:

1. the PBS headers should be included
2. the modules containing the software needs to be loaded
3. is optimization possible by using a scratch node
4. check if the results are stored in a data or staging directory
5. scratch node clean-up.

8.1.1 PBS headers

The PBS headers are added after the first bash script line (#!/bin/bash). These headers contain the parameters that can be given to the start command (see the Start Basic Jobs chapter). A good practice is to write the variables in the script, so at chairing, reusing or rerunning all needed options are included. Here is an example of how the PBS header can look like (ThinKing example):

```
#!/bin/bash -l
#PBS -l walltime=12:00:00
#PBS -l mem=100gb
#PBS -l nodes=1:ppn=20:ivybridge
#PBS -M mail@mail.com
#PBS -m aeb
#PBS -N jobname
#PBS -A lp_projectname
```

Each of these lines is an optional parameter when the script is launched.

```
#PBS -l walltime=12:00:00
```

This defines the walltime of the job. Here this is 12 hours. The walltime is the longest possible time the program can run. When this time is exceeded, the job is aborted.

```
#PBS -l mem=100gb
```

This defines the needed memory (RAM) for the job. Here 100gb. For information on the possible values, see the The Hardware chapter.

```
#PBS -l nodes=1:ppn=20:ivybridge
```

Here the needed CPU is described. This job needs 1 node with 20 cores of the Ivybridge type. Check the The Hardware chapter for a description of all available hardware.

```
#PBS -M mail@mail.com
#PBS -m aeb
```

These 2 options belong together, and describes the possible reporting. The -m is the option for when mails should be send. Possible options are: when the job begins (b), when it ends (e) and when it is aborted (a). The -M option specifies the mail adres.

```
#PBS -N jobname
```

This option shows the jobname. Note that this jobname is shown when a qstat command is ran. This name is also visible to other users.

```
#PBS -A lp_projectname
```

This option is used for the billing, when this is missing the default project (users introduction credits) will be selected. More on this billing in the Start Basic Jobs chapter.

8.1.2 Load modules

When a job is started on a node, this node has no modules loaded, except for the standard tools (standard linux commands). Therefor every software that is used in the script has to be loaded. This is possible by loading the correct module. For example:

```
$ module load zlib
```

More about modules can be found in the The Module System chapter.

8.1.3 Scratch And Result Storage

The execution of software often generates output, and many temporary results. When the program ends, the output must be stored on a save place (this should be the data or staging directory). To be sure everything is kept, the script can write the output directly to the staging /data directory. However this can reduce speed, and generate a lot of extra, unneeded data. A solution for this can be the use of the scratch storage. In this case the output and temporary files will be written to the faster scratch. Before the end of the script, the correct needed file(s) should be copied to the data/staging directory, and the scratch should be cleaned.

Using the scratch during execution of a script, speeds up the run time. But at the end, the result has to be copied or moved to the right data/staging directory. A good practice is to remove your working directory on the scratch, especially when batch jobs are used.

9 Start Basic Jobs

9.1 Job Submission

9.1.1 Job Submission on ThinKing

A job can be started using the qsub command, for example:

```
$ qsub run-job.pbs
```

At least, when the PBS script is defined as described in the Portable Batch System chapter. All variables described in this chapter are also available as parameters for this job. At the start, these PBS variables will be parsed, so the system knows which node type it has to reserve. If the account to be charged has insufficient credits for the job, the user receives a warning at this point. Just prior to job execution, a reservation will be made on the specified project's account, or the user's personal account if no project was specified. When the user checks her balance at this point, she will notice that it has been decreased with an amount equal to, or less than that provided by gquote. The latter may occur when the node type is determined when the reservation is made, and the node type is less expensive than that assumed by gquote. If the relevant account has insufficient credits at this point, the job will be deleted from the queue. When the job finishes, the account will effectively be charged. The balance of that account will be equal or larger after charging. The latter can occur when the job has taken less walltime than the reservation was made for. This implies that although quotes and reservations may be overestimations, users will only be charged for the resources their jobs actually consumed.

When a job is submitted, the job id will be returned. If this job id starts with a 2, it is submitted to the ThinKing cluster. This job ID is always unique.

9.1.2 Job Submission on Cerebro

Cerebro uses the same submission system as ThinKing. The only difference is that before a job can be submitted, the cerebro module has to be loaded.

```
$ module load cerebro/2014a
$ qsub run-job.pbs
```

When a job is submitted, the job id will be returned. If this job id starts with a 3, it is submitted to the Cerebro cluster. This job ID is always unique.

9.2 Job Progress

A submitted job can be checked in multiple ways.

9.2.1 qstat

```
$ qstat -u vsc3XXXX
```

This command shows all scheduled jobs for the given user.

For Cerebro, the cerebro module has to be loaded. For going back to ThinKing, the cerebro module has to be unloaded.

```
$ module load cerebro/2014a
$ module unload cerebro
```

9.2.2 showstart

The showstart command shows an estimation of the job start. The result gives an estimation when the job will start (here in 0 seconds, thus the job just started. This can be negative when the job is running). It also describes that the job reserved 10 processors for 20 days. The best partition is the smp1 partition (on Cerebro).

```
$ showstart 30017323.hpc-p-svcs-5
job 30017323 requires 10 procs for 20:00:00:00
Estimated Rsv based start in                00:00:00 on Mon Apr
18 16:29:10
Estimated Rsv based completion in        20:00:00:00 on Sun May
8 16:29:10
Best Partition: smp1
NOTE:  job is running
```

9.2.3 checkjob

Another usefull command is checkjob. This command gives an overview every parameter and all used resources.

```
$ checkjob 30017323.hpc-p-svcs-5
job 30017323
AName: jobname
State: Running
Creds:  user:vsc3XXXX  group:vsc3XXXX  account:lp_projectname
class:q21d  qos:normal
WallTime:   00:02:43 of 20:00:00:00
SubmitTime: Mon Apr 18 16:28:56
  (Time Queued  Total: 00:00:14  Eligible: 00:00:14)
StartTime: Mon Apr 18 16:29:10
TemplateSets:  DEFAULT
Total Requested Tasks: 10
Req[0]  TaskCount: 10  Partition: smp1
Memory >= 1024M  Disk >= 0  Swap >= 0
Dedicated Resources Per Task: PROCS: 1  MEM: 24G
NodeSet=ONEOF:FEATURE:[NONE]
Allocated Nodes:
[smp1-31:10]
SystemID:   Moab
SystemJID:  30017323
Notification Events: JobFail  Notification Address: mail@mail.com
StartCount:     1
Partition List: smp1
Flags:          BACKFILL,RESTARTABLE
```

```
Attr:           BACKFILL,checkpoint
Variables:      UsageRecord=4872632
StartPriority:  2084
PE:             46.13
Reservation '30017323' (-00:02:43 -> 19:23:57:17  Duration: 20:00:
00:00)
```

The qstat command shows all running and scheduled jobs. As parameter it needs -u and a username.

```
$ qstat -u vsc3XXXX
hpc-p-svcs-5:

Req'd       Req'd       Elap
Job ID                  Username    Queue    Jobname
SessID  NDS   TSK   Memory      Time    S   Time
----------------------- ----------- -------- ----------------
------ ----- ------ --------- --------- - ---------
30017323.hpc-p-svcs-5   vsc3XXXX    q21d     jobname
197071     1     10       1gb 480:00:00 R  00:04:16
```

9.2.4 pbstop

This more advanced command shows all installed nodes, and there status. When your job is not starting, you could check if all nodes are in use. The screen below can be exited by pressing q.

```
$ pbstop
```

```
Usage Totals: 45/640 Procs, 45/64 Nodes, 1/1 Jobs
Running
16:34:34
Node States:    27 free                  37 job-exclusive
 visible CPUs: 0,1,2,3
          1 2 3 4 5 6 7 0   1 2 3 4 5 6 7 0
          --------------------------------
   smp2-0 .... .... .... .... .... .... .... ....   .... .... .... .... .... .... .... ....
   smp1-0 .... .... .... .... .... .... @@@@ @@@@   @@@@ @@@@ @@@@ @@@@ @@@@ @@@@ @@@@ @@@@
  smp1-16 @@@@ @@@@ @@@@ @@@@ @@@@ @@@@ @@@@ @@@@   @@@@ @@@@ @@@@ @@@@ @@@@ @@@@ @@@@ @@@@
  smp1-32 .... .... .... .... .... @@@@ @@@@ @@@@   @@@@ @@@@ @@@@ @@@@ @@@@ @@@@ @@@@ @@@@
          --------------------------------
      Job#    Username Queue    Jobname          CPUs/Nodes S Elapsed/Requested
  v = 30017323 vsc3XXXX q21d    jobname             ?/1     R 00:05:04/480:00:0
[?] unknown  [@] busy  [*] down  [.] idle  [%] offline  [!] other
```

⚠ The job scheduler job rule is first comes first serves, but lets you skip a part of the queue if a certain job cannot be runned. Let me explain: Lets say the cluster only has 10 nodes. Job A is requested for 4 nodes for 10 hours. No job is running, so Job A is started. After 1 hour Job B is requested for 10 nodes for 2 hours. There are only 6 nodes idle, so Job B has to wait until Job A finishes (worst case, only in 9 hours). After another hour Job C is requested. Job A is still running (if not Job B would already been launched). Job C only request 2 nodes. Now the tricky part: Job A still has a requested walltime of another 8 hours. If the requested walltime of Job C is 8 hours or larger, it will have to wait at the back of the queue, and have to wait until job B ends. If the requested walltime of Job C is less, Job C would theoretically end before Job A, and so not infecting the start time of Job B. So Job C will be launched.

9.3 Stop/kill/delete a job

Sometimes a running job has to be cancelled. This can happen for various reasons like: wrong project to be billed, launched to the wrong cluster, ... Cancelling a job is easy, when the job id is known. Note that when a job is cancelled before running, no credits will be lost. When a job was running, the resources used while running will be accounted.

```
qdel 30017323.hpc-p-svcs-5
```

10 Parallel Jobs

Sometimes a list of samples has to be processed on exactly the same way. If the basic jobs would be used, the user can use multi-threading, but has to adjust each pbs script (see Portable Batch System) for each sample. A neat solution for this editing option is the parameter variation. The parameter variation uses the worker framework.

10.1 Parameter variations

Suppose the program the user wishes to run is 'map' (this program does not exist, it is just an example) that takes three parameters, a sample (-s), a reference (-r) and a read length (-l). A typical call of the program looks like:

```
$ map -s sample1 -r homo_sapiens -l 100
```

The program will write its results to standard output. A PBS script (say run.pbs) that would run this as a job would then look like:

```
#!/bin/bash -l
#PBS -l nodes=1:ppn=1
#PBS -l walltime=00:15:00
cd $VSC_SCRATCH
map -s sample1 -r homo_sapiens -l 100
```

When submitting this job, the calculation is performed or this particular instance of the parameters, i.e., sample = sample1, reference = homo_sapiens, and read length = 100. To submit the job, the user would use:

```
$ qsub run.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 samples, references and read lengths. To this end, the PBS file can be modified as follows:

```
#!/bin/bash -l
#PBS -l nodes=1:ppn=8
#PBS -l walltime=04:00:00
cd $VSC_SCRATCH
map -s $sample  -r $reference  -l $length
```

Note that

1. the parameter values sample1, homo_sapiens, 100 have been replaced by variables $sample, $reference and $length respectively;
2. the *number of processors per node* has been increased to 8 (i.e., ppn=1 is replaced by ppn=8); and
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 1,500 minutes on one CPU. However, this job will use 7 CPUs (1 is reserved for delegating work), so the 100 calculations will be done in 1,500/7 = 215 minutes, i.e., 4 hours to be on the safe side. Note that starting from version 1.3, a dedicated core is no longer required for delegating work when using the `-master` flag. This is however not the default behavior since it is implemented using features that are not standard. This implies that in the previous example, the 100 calculations would be completed in 1,500/8 = 188 minutes.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 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.txt would look like:

```
sample,reference,length
sample1,homo_sapiens,100
sample2,homo_sapiens,50
sample2,mus_musculus,50
sample3,homo_sapiens,100
sample3,homo_sapiens,50
...
```

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

The job can now be submitted as follows:

```
$ module load worker/1.5.0-intel-2014a
$ wsub -batch run.pbs -data data.csv
```


Note that the PBS file is the value of the -batch option . The map program will now be run for all 100 parameter instances—7 concurrently—until all computations are done. A computation for such a parameter instance is called a work item in Worker parlance.

10.2 Job Arrays

Although job arrays are available, the use of parameter variation is easier, and recommended.

10.3 MapReduce: Prologues and Epilogues

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 chuncks;
2. on these chuncks, the same algorithm is applied independently (these are the work items); and
3. the results of the computations on those chuncks are aggregated into, e.g., a statistical description of some sort.

The Worker framework directly supports this scenario by using a prologue and an epilogue. 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 prologue and epilogue are executed by the master, i.e., the process that is responsible for dispatching work and logging progress.

Suppose that 'split-data.sh' is a script that prepares the data by splitting it into 100 chuncks, and 'distr.sh' aggregates the data, then one can submit a MapReduce style job as follows:

```
$ wsub -prolog split-data.sh -batch run.pbs -epilog distr.sh -
data data.csv
```


Note that the time taken for executing the prologue and the epilogue should be added to the job's total walltime.

10.4 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.log445948', assuming the job's ID is 445948. To keep an eye on the progress, one can use:

```
$ tail -f run.pbs.log445948
```

Alternatively, a Worker command that summarizes a log file can be used:

```
$ watch -n 60 wsummarize run.pbs.log445948
```

This will summarize the log file every 60 seconds.

10.5 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 are not even started. Again, a simple and yet versatile solution is offered by the Worker framework. 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.

```
#!/bin/bash -l
#PBS -l nodes=1:ppn=8
#PBS -l walltime=04:00:00
module load timedrun/1.0.1
cd $VSC_SCRATCH
timedrun -t 00:20:00 map -s $sample  -r $reference  -l $length
```

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 sample, reference and length.

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.

10.6 Multithreaded work items

When a cluster is configured to use CPU sets, using Worker to execute multithreaded work items doesn't work by default. Suppose a node has 20 cores, and each work item runs most efficiently on 4 cores, then one would expect that the following resource specification would work:

```
$ wsub -l nodes=10:ppn=5 -W x=nmatchpolicy=exactnode -batch run.
pbs -data my_data.csv
```

This would run 5 work items per node, so that each work item would have 4 cores at its disposal. However, this will not work when CPU sets are active since the four work item threads would all run on a single core, which is detrimental for application performance, and leaves 15 out of the 20 cores idle. Simply adding the -threaded option will ensure that the behavior and performance is as expected:

```
 $ wsub -l nodes=10:ppn=5 -batch run.pbs -data my_data.csv -
threaded
```


Note however that using multihreaded work items may actually be less efficient than single threaded execution in this setting of many work items since the thread management overhead will be accumulated.

Also note that this feature is new since Worker version 1.5.x.

11 Interactive Node

Interactive jobs are jobs that asks user input. These jobs run with a direct access to the node. This should be used only for short jobs (this has a limited execution time). An example of jobs are: compiling new software, matlab, R, ...

11.1 Regular interactive jobs, without X support

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

```
vsc30001@login1:~> qsub -I
qsub: waiting for job 20030021.icts-p-svcs-1 to start
qsub: job 20030021.icts-p-svcs-1 ready
vsc30001@r2i2n15:~>
```

11.2 Interactive jobs with X support

Before starting an interactive job with X support, you have to make sure that you have logged in to the cluster with X support enabled. If that is not the case, you won't be able to use the X support inside the cluster either!

The easiest way to start a job with X support is:

```
vsc30001@login1:~> qsub -X -I
qsub: waiting for job 20030021.icts-p-svcs-1 to start
qsub: job 20030021.icts-p-svcs-1 ready
vsc30001@r2i2n15:~>
```

12 Monitoring memory, CPU usage and temporary file size of programs

Estimating the amount of memory an application will use during execution is often non trivial, especially when one uses third-party software. However, this information is valuable, since it helps to determine the characteristics of the compute nodes a job using this application should run on. Although the tool presented here can also be used to support the software development process, better tools are almost certainly available. Note that currently only single node jobs are supported, MPI support may be added in a future release.

12.1 Monitoring a program

To start using monitor, first load the appropriate module:

```
$ module load monitor
```

Starting a program, e.g., simulation, to monitor is very straightforward

```
$ monitor simulation
```

monitor will write the CPU usage and memory consumption of simulation to standard error. Values will be displayed every 5 seconds. This is the rate at which monitor samples the program's metrics.

12.1.1 Log file

Since monitor's output may interfere with that of the program to monitor, it is often convenient to use a log file. The latter can be specified as follows:

```
$ monitor -l simulation.log simulation
```

For long running programs, it may be convenient to limit the output to, e.g., the last minute of the programs execution. Since monitor provides metrics every 5 seconds, this implies we want to limit the output to the last 12 values to cover a minute:

```
$ monitor -l simulation.log -n 12 simulation
```

Note that this option is only available when monitor writes its metrics to a log file, not when standard error is used.

12.1.2 Modifying the sample resolution

The interval at which monitor will show the metrics can be modified by specifying delta, the sample rate:

```
$ monitor -d 60 simulation
```

monitor will now print the program's metrics every 60 seconds. Note that the minimum delta value is 1 second.

12.1.3 File sizes

Some programs use temporary files, the size of which may also be a useful metric. monitor provides an option to display the size of one or more files:

```
$ monitor -f tmp/simulation.tmp,cache simulation
```

Here, the size of the file simulation.tmp in directory tmp, as well as the size of the file cache will be monitored. Files can be specified by absolute as well as relative path, and multiple files are separated by ','.

12.1.4 Programs with command line options

Many programs, e.g., matlab, take command line options. To make sure these do not interfere with those of monitor and vice versa, the program can for instance be started in the following way:

```
$ monitor -delta 60 -- matlab -nojvm -nodisplay computation.m
```

The use of '--' will ensure that monitor does not get confused by matlab's '-nojvm' and '-nodisplay' options.

12.1.5 Subprocesses and multicore programs

Some processes spawn one or more subprocesses. In that case, the metrics shown by monitor are aggregated over the process and all of its subprocesses (recursively). The reported CPU usage is the sum of all these processes, and can thus exceed 100 %. Some (well, since this is a HPC cluster, we hope most) programs use more than one core to perform their computations. Hence, it should not come as a surprise that the CPU usage is reported as larger than 100 %. When programs of this type are running on a computer with n cores, the CPU usage can go up to n x 100 %.

12.1.6 Exit codes

Monitor will propagate the exit code of the program it is watching. Suppose the latter ends normally, then monitor's exit code will be 0. On the other hand, when the program terminates abnormally with a non-zero exit code, e.g., 3, then this will be monitor's exit code as well. When monitor has to terminate in an abnormal state, for instance if it can't create the log file, its exit code will be 65. If this interferes with an exit code of the program to be monitored, it can be modified by setting the environment variable MONITOR_EXIT_ERROR to a more suitable value.

13 Glossary

archive	The archive is long time storage.
bash	Bash is a Unix shell and command language
Cerebro	The shared memory cluster of the VSC
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.
core	A core is the unit that read and execute program instructions . It is part of the CPU.
CPU	A central processing unit (CPU) is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logical, control and input/output (I/O) operations specified by the instructions.
flops	FLOPS or flops (floating-point operations per second) is a measure of computer performance , useful in fields of scientific calculations that make heavy use of floating-point calculations. For such cases it is a more accurate measure than the generic instructions per second .
Haswell	A type of fast CPUs, with 12 cores per CPU. In Thinking combined per 2 in a node.
HPC	High Performance Computing/Computer or supercomputer is a computer with a high-level computational capacity compared to a general-purpose computer. Performance of a supercomputer is measured in floating-point operations per second (FLOPS) instead of million instructions per second (MIPS).
I/O	Input/Output (I/O) is the communication between an information processing system , such as a computing node , and the outside world, possibly a human, another information processing system or the storage.
Ivybridge	

	A type of fast CPUs, with 10 cores per CPU. A node in Cerebro has 1 CPU, a node in ThinKing has 2.
node	A node is a basic unit used in computer science. Nodes are devices or data points on a larger network.
pbs	Portable Batch System (or simply PBS) is the name of computer software that performs job scheduling. Its primary task is to allocate computational tasks, i.e., batch jobs, among the available computing resources. It is often used in conjunction with UNIX cluster environments. PBS is supported as a job scheduler mechanism by several meta schedulers.
RAM	Random Access Memory (RAM) is a form of computer data storage. It is the fastest storage to access by the CPU.
schedular	Scheduling is the method by which work specified by some means is assigned to resources that complete the work. The work may be virtual computation elements such as threads or processes , which are in turn scheduled onto hardware resources such as processors (CPU).
scratch	The scratch is the temporary storage for programs that are executing.
sh	A Unix shell is a command-line interpreter or shell that provides a traditional Unix-like command line user interface . Users direct the operation of the computer by entering commands as text for a command line interpreter to execute, or by creating text scripts of one or more such commands (sh script).
shared memory	Shared memory is memory that may be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies. Shared memory is an efficient means of passing data between programs. Depending on context, programs may run on a single processor or on multiple separate processors.
staging	The staging is storage for on going projects.
thread	A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler .
ThinKing	The Thin node cluster of the VSC

VSC	Vlaamse Supercomputing Centrum

14 Cheat Sheet

A quick overview of all available commands and variables.

14.1 Account and Groups

Account and groups are managed through: https://account.vscentrum.be/django/

```
#login to the VSC (vsc3XXXXX is your user id)
$ ssh vsc3XXXX@login.hpc.kuleuven.be
```

14.2 Credits

```
#load the accounting module
$ module load accounting
#check how many credits you have left
$ gbalance
#ask a quote for this resources
$ gquote -l nodes=3:ppn=4:ivybridge,pmem=2gb,walltime=48:00:00
#get an overview of all jobs
$ gstatement
#get an overview of all jobs in the month september for the
lp_projectname
$ gstatement -g lp_projectname -s 2015-09-01 -e 2015-09-30
#get infromation about a job
$ glsjob -J jobID
```

14.3 Storage

Storage Type	Path	Environment Variable	Standard Size	Usage
Home	/user/leuven /3XX /vsc3XXXX	$VSC_HOME	25GB	Important data, like configuration files. Is private.
Data	/data/leuven /3XX /vsc3XXXX	$VSC_DATA	75GB	Important data, biger data. Is private.
Scratch	/scratch /leuven/3XX /vsc3XXXX	$VSC_SCRATCH	100GB	Temporary data, will be deleted within 21 days.
Node Scratch		$VSC_SCRATCH_NODE	machine dependeble, min 150GB	Temporary data, while job is running. Can not be accessed from the login node. Data is lost at the jobs end.
Staging	/staging /leuven /stg_000XX		project dependable, per 1TB	Storage for project files, while still working at the project.
Archive	/archive /leuven /arc_000XX		project dependable, per 1TB	Backup for project files (or staging), long term storage. Only accessible from the login node.

```
#get the size of a directory
$ du -h --max-depth=1 directoryname
#get the size of a partition
$ df -h directoryname
#change the permissions of a file
$ chmod u=rwx,g=rx,o=r -R myfile
$ chmod 754 -R myfile
#change the group of a file
$ chgrp -R groupname filename
#change the owner of a file
$ chown -R username filename
```

14.4 The Module System

```
#give a list of available modules
$ module av
#load a module
$ module load modulename
$ module load modulename/moduleversion
#show a list of all loaded modules
$ module list
#unload a module
$ module unload modulename
#unload all modules at once
$ module purge
#show the help
$ module help
```

14.5 The Hardware

Cluster	Partition	CPU-Type	#nodes	#cores (threads) per node	Internal disk (Node Scratch)	Useable Memory (RAM) per node	#credits /hour
ThinKing	Ivybridge	Ivybridge	208	20	250GB	60-124GB	4.76
ThinKing	Haswell	Haswell	144	24	150GB	60-124GB	6.68
Cerebro	Smp1	Ivybridge	48	10	shared 10TB	shared 250GB (max 11.77 TB)	3.45
Cerebro	Smp2	Ivybridge	16	10	shared 10TB	shared 124GB (max 1.79 TB)	3.45

Typical tasks run on the different clusters/partitions:

Cluster	Partition	Task Description	Task Example
ThinKing	Ivybridge	Memory low jobs, with lots of I/O	Alignment, Read Mapping, Variant Calling, Read Counting, ...
ThinKing	Haswell	Memory low jobs, low I/O, high computing power needed	Model Calculations, Discovering a new Star, ...
Cerebro	Smp1 and Smp2	High memory jobs, computing power less important	De Novo Assemblies, Reference-based Assemblies, ...

14.6 Portable Batch System

A typical pbs script header:

```
#!/bin/bash -l
#PBS -l walltime=12:00:00
#PBS -l mem=100gb
#PBS -l nodes=1:ppn=20:ivybridge
#PBS -M mail@mail.com
#PBS -m aeb
#PBS -N jobname
#PBS -A lp_projectname
module load modulename modulename2
```

14.7 Start Basic Jobs

```
#submit a job
$ qsub job.pbs
#show all requested/running jobs for the user
$ qstat -u vsc3XXXX
#show the estimated start for a job
$ showstart jobID
#check the status of a job
$ checkjob jobID
#show the status of the cluster
$ pbstop
```

14.8 Parallel Jobs

```
#load the worker module
$ module load worker/1.5.0-intel-2014a
#launch a batch job with the parameter variation
$ wsub -batch job.pbs -data data.csv
$ wsub -prolog beforejob.sh -batch job.pbs -epilog afterjob.sh -
data data.csv
$ wsub -l nodes=10:ppn5 -batch job.pbs -data data.csv -threaded
#show the process of a parallel hob
$ tail -f job.pbs.logXXXXXX
$ watch -n number_of_seconds job.pbs.logXXXXXX
```

An example of a parallel job, where each job gets a limited time.

```
#!/bin/bash -l
#PBS -l nodes=1:ppn=8
#PBS -l walltime=04:00:00
module load timedrun/1.0.1
cd $VSC_SCRATCH
timedrun -t 00:20:00 map -s $sample  -r $reference  -l $length
```

An example of the data file, belonging to the above pbs script. Note that the variables in the scripts are the heading in the file.

```
sample,reference,length
sample1,homo_sapiens,100
sample2,homo_sapiens,50
sample2,mus_musculus,50
sample3,homo_sapiens,100
sample3,homo_sapiens,50
...
```

14.9 Interactive Nodes

```
$ qsub -I
$ qsub -X -I
```

15 Appendix A: Exercise

15.1 Exercise 0

Login to the HPC of the VSC. This exercise is trivial, and is already tackled at the Linux Basic course.

15.2 Exercise 1

Since no tasks can be performed without raw data, and results should be kept for longer periodes, storage is an important feature of a cluster.

Therefor the first exercise is to check the different environment variables, and find there complete path.

Storage Type	Environment Variable	Path
Home	$VSC_HOME	
Data	$VSC_DATA	
Scratch	$VSC_SCRATCH	
Node Scratch	$VSC_SCRATCH_NODE	
Staging		/staging/leuven/stg_000XX
Archive		/archive/leuven/arc_000XX

Make a tutorial directory in your data directory. Copy the tutorial data from /staging/leuven /stg_00019/workshop to the tutorial directory.

15.3 Exercise 2

Computing costs money, therefor the VSC uses a credit based system for the accounting.

1. Get an overview of your current balance.
2. Get an overview of all your transactions of the last month.
3. Ask a quote for 5 minutes computing on 1 node, 5 processors of the ivybridge type.
4. Ask a quote for 20 minutes computing on 2 nodes, 4 processors of the haswell processor type, using 100Gb of memory.
5. Ask for both the ivybridge and the haswell processor a quote for 1 hour of computing.

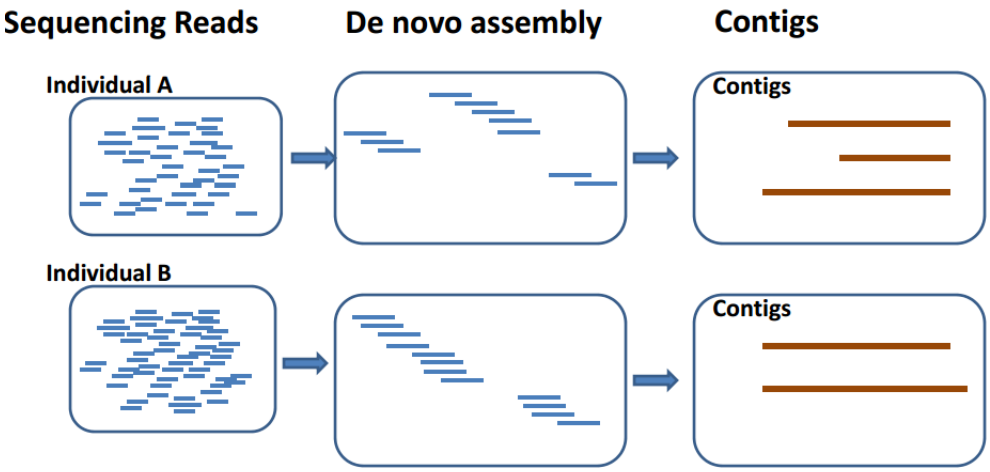
15.4 Exercise 3

This exercise uses the paths.pbs script.

1. Open the paths.pbs script, and check if the pbs headers and the script purpose is clear.
2. Ask a quote based on the pbs header.
3. Start the job on thinking
4. Check the start time and status (2 different commands) of the job.
5. Check the output of the job. Can you access all paths?

15.5 Exercise 4

Up till now the exercises were ment only as an introduction. Here we start with a more real-life example of a bioinformatics project: a *de novo* assembly of a small circular genome. The basic principle is: reads are put together if they overlap, resulting in a contig. In this case, we have a circular genome, with only 1 chromosome. So in the ultimate case, we end with one contig.



ABySS will be used in this exercise. ABySS is a widely used de Bruijn graph assembler. Since circular genomes are hard to assemble (since there is no clear choose where to cut the genome to make it linear), a large kmer size has to be chosen (so it can split the genome easily in one or multiple contigs).

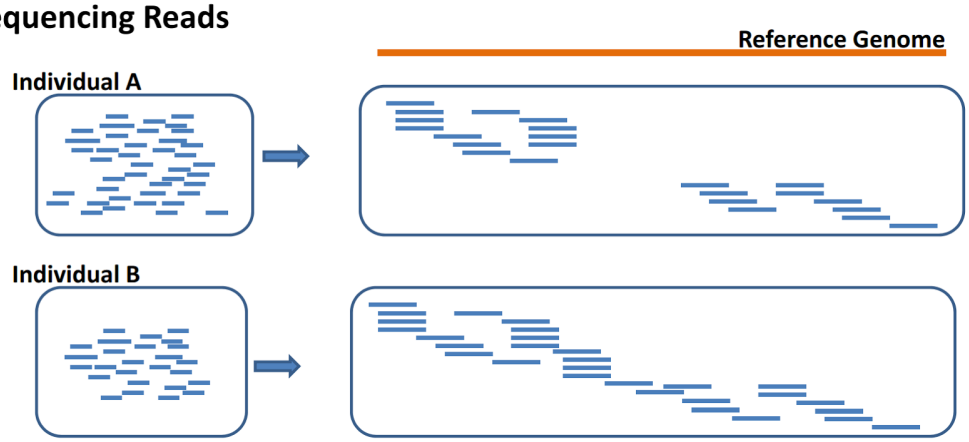
1. Since *de novo* assembly generally uses a lot of memory, this task has to run on cerebro. Load the cerebro module.
2. Check if ABySS is available on cerebro. If not, load the 2015a versions. Load ABySS.
3. Open the abyss.pbs script, and check if everything is alright to run.
4. Run the abyss.pbs script. Note that the returned job ID start with a 3.
5. Check the output of the assembly.

15.6 Exercise 5

The next exercise is best to run on thinking.

1. List all loaded modules
2. Unload or Purge these modules
3. Load the thinking module

In this exercise we are going to map multiple samples against our new assembled reference genome. Mapping is the process where you compare your sequences against a reference in order to get the location of your sequence on the reference genome. BWA and Bowtie are examples of widely used mapping tools. In this exercise Bowtie will be used.



Assume a population study of 19 individuals (same origin as the sample of exercise 4). If the tasks are split, 20 tasks are needed: the indexing of the genome by the mapping tool (is only needed once, and can be reused), the mapping of the samples (19 samples, so 19 tasks). If the number of individuals would rise more, the number of scripts to run will get to high to be manageable. Therefor the solution is to use parallel jobs.

1. Parallel jobs can exists of 3 parts: a prolog, the batch, and an epilogue. Which part is suitable for which part of the job?

Task type	Prolog, batch or epilogue?
Mapping of a sample	
Cleanup of the scratch or temporary directories	

Task type	Prolog, batch or epilogue?
Creating of the reference index needed by the mapping tool	

2. Open the batch script.
    a. Change the PBS header so 20 cores are used for the batch. Extra: Why do we use 20 cores while there are only 19 individuals to map?
    b. Check the name of the variable for the sample name.
    c. Is the input and output path correct?
    d. Which directory path is used to store the reference genome?
3. Open the prolog script.
    a. What happens in this script?
    b. Check if the new assembled reference genome can be accessed.
    c. Check the path where the reference genome is stored.
4. Open the epilogue script
    a. What happens in this script?
    b. Why is this needed?
5. A parallel job also needs a data file. In this case the data-file contains as header the name of the variable, and per row a new sample name.
    a. Can you recreate this sample file?
6. Start the job on thinking
7. Check the mapping statistics (especially the one of the used data with the assembly (previous exercise), since this can give an indication of the assembly quality).

15.7 HomeWork

The samples used in Exercise 5, contains out of 2 population. Population 1 is a subset of real sequenced data at the Genomics Core, while Population 2 is simulated data (has a constant quality) from the NCBI reference genome. In population studies variants that only occur in one of the populations are interesting. FreeBayes is an easy variant calling tool for populations, since it can call variants in multiple samples at the same time. Write a pbs script around this command, and execute this on the HPC.

```
$GENOME_DIR="/data/leuven/314/vsc31439/tutorial/denovo
/abyss_output";
$FREEBAYES_OPTIONS="-m 20 -q 15 --use-duplicate-reads --ploidy 2";
$OUTPUT_DIR="/data/leuven/314/vsc31439/tutorial/freebayes";
mkdir -p $OUTPUT_DIR;
bamfiles="";
for i in `ls -1 -d $SCRATCH_DIR/bams/*bam`;
do
        bamfiles="$bamfiles $i";
done
freebayes --fasta-reference $GENOME_DIR/genome.fa
$FREEBAYES_OPTIONS $bamfiles > $OUTPUT_DIR/freebayes.vcf;
```

How many variants do you find? And how many of these variants are interesting to discriminate the populations?

16 Appendix B: Solutions

16.1 Exercise 0

Login to the HPC of the VSC. This exercise is trivial, and is already tackled at the Linux Basic course.

```
$ ssh vsc3XXXX@login.hpc.kuleuven.be
```

16.2 Exercise 1

Since no tasks can be performed without raw data, and results should be kept for longer periodes, storage is an important feature of a cluster.

Therefor the first exercise is to check the different environment variables, and find there complete path.

```
$ cd $VSC_HOME
$ pwd
/user/leuven/3XX/vsc3XXXX
$ cd $VSC_DATA
$ pwd
/data/leuven/3XX/vsc3XXXX
$ cd $VSC_SCRATCH
$ pwd
/scratch/leuven/3XX/vsc3XXXX
$ cd $VSC_SCRATCH_NODE
/node_scratch
```

Storage Type	Environment Variable	Path
Home	$VSC_HOME	/user/leuven/3XX/vsc3XXXX
Data	$VSC_DATA	/data/leuven/3XX/vsc3XXXX
Scratch	$VSC_SCRATCH	/scratch/leuven/3XX/vsc3XXXX
Node Scratch	$VSC_SCRATCH_NODE	/node_scratch NOTE: this is only the scratch of the login node!
Staging		/staging/leuven/stg_000XX
Archive		/archive/leuven/arc_000XX

Make a tutorial directory in your data directory. Copy the tutorial data from /staging/leuven /stg_00019/workshop to the tutorial directory.

```
$ cd $VSC_DATA
$ mkdir tutorial
$ cd tutorial
$ rsync -ahr --progress /staging/leuven/stg_00019/workshop/* .
```

16.3 Exercise 2

Computing costs money, therefor the VSC uses a credit based system for the accounting.

1. Get an overview of your current balance.

```
$ module load accounting
$ gbalance
```

2. Get an overview of all your transactions of the last month.

```
$ gstatement -s 2016-05-01 -e 2016-05-31
```

3. Ask a quote for 5 minutes computing on 1 node, 5 processors of the ivybridge type.

```
$ gquote -l nodes=1:ppn=5:ivybridge,walltime=0:05:00
0.20
```

4. Ask a quote for 20 minutes computing on 2 nodes, 4 processors of the haswell processor type, using 100Gb of memory.

```
$ gquote -l nodes=2:ppn=4:haswell,pmem=100gb,walltime=0:20:00
4.45
```

5. Ask for both the ivybridge and the haswell processor a quote for 1 hour of computing.

```
$ gquote -l nodes=1:ppn=1:ivybridge,walltime=1:00:00
4.76
$ gquote -l nodes=1:ppn=1:haswell,walltime=1:00:00
6.68
```

16.4 Exercise 3

This exercise uses the paths.pbs script.

1. Open the paths.pbs script, and check if the pbs headers and the script purpose is clear.

```
#multiple methods possible, to view:
$ cat paths.pbs
$ less paths.pbs
$ more paths.pbs
#to edit:
$ nano paths.pbs
$ vim paths.pbs
```

paths.pbs

```
#!/bin/bash -l
#PBS -l walltime=00:05:00
#PBS -l mem=6gb
#PBS -l nodes=1:ppn=1:ivybridge
#PBS -M koen.herten@kuleuven.be
#PBS -m aeb
#PBS -N paths
#PBS -A default_project
echo $VSC_HOME >> $VSC_DATA/tutorial/paths.txt;
echo $VSC_DATA >> $VSC_DATA/tutorial/paths.txt;
echo $VSC_SCRATCH >> $VSC_DATA/tutorial/paths.txt;
echo $VSC_SCRATCH_NODE >> $VSC_DATA/tutorial/paths.txt;
sleep 240;
```

2. Ask a quote based on the pbs header.

```
$ cat paths.pbs | grep "#PBS"
#PBS -l walltime=00:05:00
#PBS -l mem=6gb
#PBS -l nodes=1:ppn=1:ivybridge
#PBS -m aeb
#PBS -N paths
#PBS -A default_project
$ gquote -l nodes=1:ppn=1:ivybridge,pmem=6gb,walltime=00:05:00
```

3. Start the job on thinking

```
$ qsub paths.pbs
##returns the job ID
```

4. Check the start time and status (2 different commands) of the job.

```
$ showstart jobID
$ checkjob jobID
$ qstat -u vsc3XXXX
```

5. Check the output of the job. Can you access all paths?

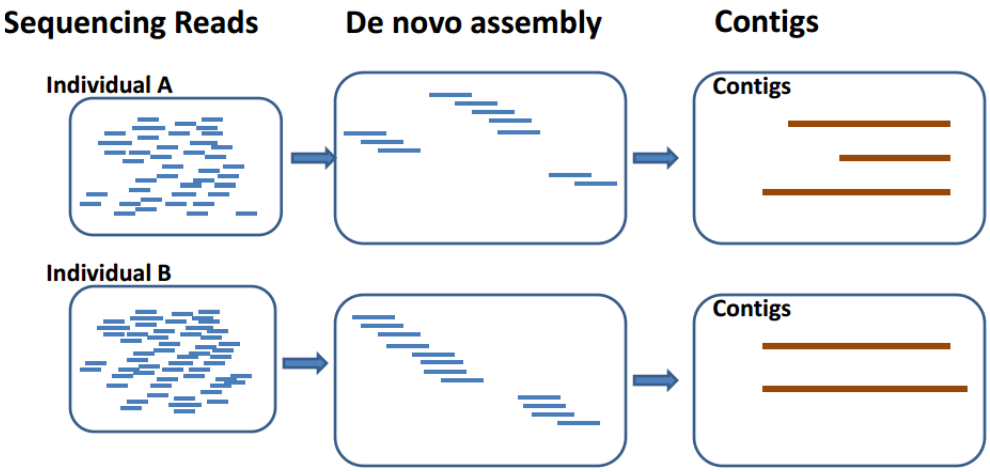
```
$ cat paths.txt
#idem as in Exercise 1
```

paths.txt

```
/user/leuven/314/vsc31439
/data/leuven/314/vsc31439
/scratch/leuven/314/vsc31439
/node_scratch
```

16.5 Exercise 4

Up till now the exercises were ment only as an introduction. Here we start with a more real-life example of a bioinformatics project: a *de novo* assembly of a small circular genome. The basic principle is: reads are put together if they overlap, resulting in a contig. In this case, we have a circular genome, with only 1 chromosome. So in the ultimate case, we end with one contig.



ABySS will be used in this exercise. ABySS is a widely used de Bruijn graph assembler. Since circular genomes are hard to assemble (since there is no clear choose where to cut the genome to make it linear), a large kmer size has to be chosen (so it can split the genome easily in one or multiple contigs).

1. Since *de novo* assembly generally uses a lot of memory, this task has to run on cerebro. Load the cerebro module.

```
$ module load cerebro/2014a
```

2. Check if ABySS is available on cerebro. If not, load the 2015a versions. Load ABySS.

```
$ module av
$ module use /apps/leuven/thinking/2015a/modules/all
$ module av
$ module load ABySS/1.9.0-intel-2015a-Python-2.7.9
```

3. Open the abyss.pbs script, and check if everything is alright to run.

```
```

```
#multiple methods possible, to view:
$ cat abyss.pbs
$ less abyss.pbs
$ more abyss.pbs
#to edit:
$ nano abyss.pbs
$ vim abyss.pbs
```

4. Run the abyss.pbs script. Note that the returned job ID start with a 3.

```
$ qsub abyss.pbs
#Returns a job ID, this should start with a 3
```

5. Check the output of the assembly.

```
#If the job is done, the abyss.o******** and abyss.e********
files can be checked for errors
#the output of the job is found in:
$ ls $VSC_DATA/tutorial/denovo/abyss_output
coverage.hist     genome_k31-1.path  genome_k31-2.fa
genome_k31-3.fa         genome_k31-stats      genome_k31-stats.
tab
genome_k31-1.dot  genome_k31-2.dot   genome_k31-2.path
genome_k31-bubbles.fa  genome_k31-stats.csv  genome_k31-
unitigs.fa
genome_k31-1.fa   genome_k31-2.dot1  genome_k31-3.dot
genome_k31-indel.fa    genome_k31-stats.md
```

part of the reference genome

```
$ head genome_k31-unitigs.fa | awk '{for(i=0; i<length($0);
i=i+50){print substr($0,i,50);}}' | head
>0 5405 53099
CCTCCAAGATTTGGAGGCATGAAAACATACAATTGGGAGGGTGTCAATCC
CTGACGGTTATTTCCTAGACAAATTAGAGCCAATACCATCAGCTTTACCG
TCTTTCCAGAAATTGTTCCAAGTATCGGCAACAGCTTTATCAATACCATG
AAAAATATCAACCACACCAGAAGCAGCATCAGTGACGACATTAGAAATAT
CCTTTGCAGTAGCGCCAATATGAGAAGAGCCATACCGCTGATTCTGCGTT
TGCTGATGAACTAAGTCAACCTCAGCACTAACCTTGCGAGTCATTTCTTT
GATTTGGTCATTGGTAAAATACTGACCAGCCGTTTGAGCTTGAGTAAGCA
TTTGGCGCATAATCTCGGAAACCTGCTGTTGCTTGGAAAGATTGGTGTTT
TCCATAATAGACGCAACGCGAGCAGTAGACTCCTTCTGTTGATAAGCAAG
```

16.6 Exercise 5

The next exercise is best to run on thinking.

1. List all loaded modules

```
$ module list
```

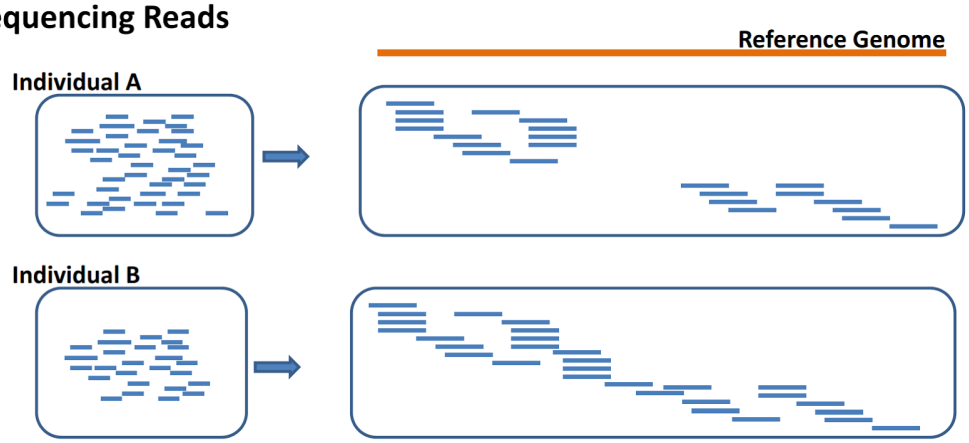
2. Unload or Purge these modules

```
$ module unload cerebro
$ module purge
```

3. Load the thinking module

```
$ module load thinking
```

In this exercise we are going to map multiple samples against our new assembled reference genome. Mapping is the process where you compare your sequences against a reference in order to get the location of your sequence on the reference genome. BWA and Bowtie are examples of widely used mapping tools. In this exercise Bowtie will be used.



Assume a population study of 19 individuals (same origin as the sample of exercise 4). If the tasks are split, 20 tasks are needed: the indexing of the genome by the mapping tool (is only needed once, and can be reused), the mapping of the samples (19 samples, so 19 tasks). If the number of individuals would rise more, the number of scripts to run will get to high to be manageable. Therefor the solution is to use parallel jobs.

1. Parallel jobs can exists of 3 parts: a prolog, the batch, and an epilogue. Which part is suitable for which part of the job?

Task type	Prolog, batch or epilogue?
Mapping of a sample	batch
Cleanup of the scratch or temporary directories	epilogue
Creating of the reference index needed by the mapping tool	prolog

2. Open the batch script.
    a. Change the PBS header so 20 cores are used for the batch. Extra: Why do we use 20 cores while there are only 19 individuals to map?

```
$ nano bowtie_batch.pbs
$ vim bowtie_batch.pbs
##CHANGE:
#PBS -l nodes=1:ppn=10:ivybridge
##TO
#PBS -l nodes=1:ppn=20:ivybridge
```

    The worker module that manages all the jobs are running on 1 core, so 19 remain for all other jobs.

    b. Check the name of the variable for the sample name. $SAMPLE
    c. Is the input and output path correct? both $VSC_DATA/tutorial/mapping
    d. Which directory path is used to store the reference genome? $VSC_SCRATCH/genome/genome is used, but this is the prefix for bowtie, the path is $VSC_SCRATCH/genome/

3. Open the prolog script.
    a. What happens in this script? The created genome in exercise 4 is copied to the $VSC_SCRATCH, and is renamed to genome.fa. The needed bowtie2 index is created.
    b. Check if the new assembled reference genome can be accessed.

```
$ ls $VSC_DATA/tutorial/denovo/abyss_output/
```

```
#output must contain genome_k31-unitigs.fa
```

c. Check the path where the reference genome is stored.
The path in the prolog script is $VSC_SCRATCH/genome/genome, and this has to be the same as in point 2d.

4. Open the epilogue script

a. What happens in this script?
The directory of the genome on the scratch is removed.

b. Why is this needed?
You need to clean to scratch for the next user.

5. A parallel job also needs a data file. In this case the data-file contains as header the name of the variable, and per row a new sample name.

a. Can you recreate this sample file?

```
$ echo "SAMPLE" > samples.csv
$ for i in `ls -1 -d *gz`; do echo $i | sed 's/.R1.fastq.
gz//g'; done >> samples.csv
```

6. Start the job on thinking

```
#You have to load the worker module.
module use /apps/leuven/thinking/2015a/modules/all
module load worker/1.6.4-intel-2015a

#Since the epilogue removes the genome from the users scratch
dir, this can be ran without, and the scratch dir can be
checked later
$ wsub -prolog bowtie_prolog.sh -batch bowtie_batch.pbs -data
samples.csv
#The clean start, with the epilogue:
$ wsub -prolog bowtie_prolog.sh -batch bowtie_batch.pbs -
epilog bowtie_epilog.sh -data samples.csv
```

7. Check the mapping statistics (especially the one of the used data with the assembly (previous exercise), since this can give an indication of the assembly quality).

```
$ cat $VSC_DATA/tutorial/mapping/test_assembly_data.bowtie2.
log
2693 reads; of these:
  2693 (100.00%) were unpaired; of these:
    9 (0.33%) aligned 0 times
```

```
    2680 (99.52%) aligned exactly 1 time
    4 (0.15%) aligned >1 times
99.67% overall alignment rate
```

16.7 HomeWork

The samples used in Exercise 5, contains out of 2 population. Population 1 is a subset of real sequenced data at the Genomics Core, while Population 2 is simulated data (has a constant quality) from the NCBI reference genome. In population studies variants that only occur in one of the populations are interesting. FreeBayes is an easy variant calling tool for populations, since it can call variants in multiple samples at the same time. Write a pbs script around this command, and execute this on the HPC.

```
$GENOME_DIR="/data/leuven/314/vsc31439/tutorial/denovo
/abyss_output";
$FREEBAYES_OPTIONS="-m 20 -q 15 --use-duplicate-reads --ploidy 2";
$OUTPUT_DIR="/data/leuven/314/vsc31439/tutorial/freebayes";
mkdir -p $OUTPUT_DIR;
bamfiles="";
for i in `ls -1 -d $SCRATCH_DIR/bams/*bam`;
do
        bamfiles="$bamfiles $i";
done
freebayes --fasta-reference $GENOME_DIR/genome.fa
$FREEBAYES_OPTIONS $bamfiles > $OUTPUT_DIR/freebayes.vcf;
```

How many variants do you find? And how many of these variants are interesting to discriminate the populations?

```
 $ grep -v "#" freebayes.m20.q15.useduplicates.ploidy2.vcf | wc -l
5
```