HPC Tutorial

Version 1.1

Author:

Geert Borstlap (UAntwerpen)

Co-authors:

Kenneth Hoste, Toon Willems, Jens Timmerman (UGent)

Geert-Jan Bex (UHasselt and KU Leuven)

Mag Selwa (KU Leuven)

Stefan Becuwe, Franky Backeljauw, Bert Tijskens, Kurt Lust (UAntwerpen)

Balázs Hajgató (VUB)

Acknowledgement: VSCentrum.be

Audience:

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

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

Contents:

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

Beginners Part		
Questions	**chapter**	**title**
What is a exactly? Can it solve my computational needs?	??	??
How to get an account?	??	??
How do I connect to the and transfer my files and programs?	??	??
How to start background jobs?	??	??
How to start jobs with user interaction?	??	??
Where do the input and output go? Where to collect my results?	??	??
Can I speed up my program by exploring parallel programming techniques?	??	??
Can I start many jobs at once?	??	??
What are the rules and priorities of jobs?	??	??

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

Advanced Part		
Questions	**chapter**	**title**
Can I compile my software on the ?	??	??
What are the optimal Job Specifications?	??	??
Do you have more examples for me?	??	??
Any more advice?	??	Good practices

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

Annex	
Title	**chapter**
??	??
??	??
??	??

Notification:

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

```
$ commands
```

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

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

"Directory" is the notation for directories (called "folders" in Windows terminology) or specific files. (e.g., "")

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

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

More support:

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

```
$ cp -r ~/
$ cd
$ ls
```

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

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

More documentation can be found at:

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

This tutorial **(Version 1.1)** is intended for users who want to connect and work on the HPC of the .

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

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

Request your appropriate version at .

Contact Information:

We welcome your feedback, comments and suggestions for improving the Tutorial (contact:).

For all technical questions, please contact the staff:

Contents

Part I

Beginner's Guide

Chapter 1

Introduction to HPC

1.1 What is HPC?

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

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

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

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

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

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

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

1.2 What is the ?

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

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

The currently consists of:

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

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

1.3 What is the not!

A computer that automatically:

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

The does not replace your desktop computer.

1.4 Is the a solution for my computational needs?

1.4.1 Batch or interactive mode?

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

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

1.4.2 Parallel or sequential programs?

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

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

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

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

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

1.4.3 What programming languages can I use?

You can use *any* programming language, *any* software package and *any* library provided it has a version that runs on Linux, specifically, on the version of Linux that is installed on the compute nodes, .

For the most common **programming languages**, a compiler is available on . Supported and common programming languages on the are C/C++, FORTRAN, Java, Perl, Python, MATLAB, R, etc.

Supported and commonly used compilers are

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

1.4.4 What operating systems can I use?

All nodes in the cluster run under , which is a specific version of . This means that all programs (executables) should be compiled for .

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

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

1.4.5 What is the next step?

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

Do not hesitate to contact the staff for any help.

Chapter 2

Getting an HPC Account

2.1 Getting ready to request an account

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

See **??** for more information on who is entitled to an account.

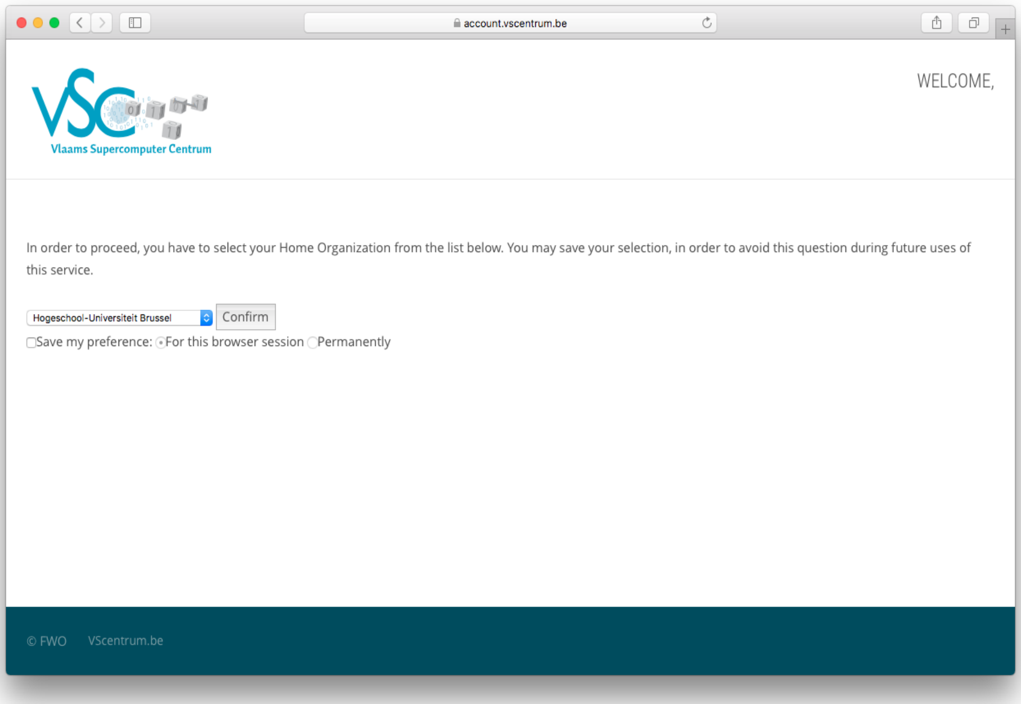
The VSC, abbreviation of Flemish Supercomputer Centre, is a virtual supercomputer centre. It is a partnership between the five Flemish associations: the Association KULeuven, Ghent University Association, Brussels University Association, Antwerp University Association and the University Colleges-Limburg.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

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

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

2.2 Applying for the account

Visit `https://account.vscentrum.be/` from within the campus network You will be redirected to our WAYF (Where Are You From) service where you have to select your “Home Organisation”.



In order to proceed, you have to select your Home Organization from the list below. You may save your selection, in order to avoid this question during future uses of this service.

Hogeschool-Universiteit Brussel | Confirm

☐ Save my preference: [✓ For this browser session] Permanently

© FWO VScentrum.be

Select ” in the dropdown box and optionaly select ’Save my preference’ and ’permanently.’

Click Confirm

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

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

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

2.2.1 Welcome e-mail

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

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

Your account should be fully active within one hour.

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

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

Kind regards,
-- The VSC administrators
```

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

2.3 Computation Workflow on the

A typical Computation workflow will be:

1. Connect to the
2. Transfer your files to the
3. Compile your code and test it
4. Create a job script
5. Submit your job
6. Wait while
   (a) your job gets into the queue
   (b) your job gets executed
   (c) your job finishes
7. Move your results

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

Chapter 3

Connecting to the HPC

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

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

3.1 First Time connection to the

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

```
$ pwd
```

Your new private home-directory is "". Here you can create your own sub-directory structure, copy and prepare your applications, compile and test them and submit your jobs on the .

```
$ cd
$ ls
Intro-HPC/
```

This directory currently contains all training material for the use of:

1. The ***Introduction to the*** .

2. Introduction to using ***perfexpert***.

More relevant training material to work with the can always be added later in this directory.

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

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

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

This directory contains:

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

```
$ cd examples
```

Tip: Typing "***cd ex<TAB>***" will generate the "***cd examples***" command. **Command-line completion** (also tab completion) is a common feature of the bash command line interpreter, in which the program automatically fills in partially typed commands.

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

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

```
$ cp -r ~/
```

You can exit the connection at anytime by entering:

```
$ exit
logout
Connection to  closed.
```

Tip: Setting your Language right:

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

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

or any other error message complaining about the locale.

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

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

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

3.2 Transfer Files to/from the

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

3.3 Modules

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

3.3.1 Environment Variables

The program environment on the is controlled by pre-defined settings, which are stored in environment (or shell) variables.

You can use shell variables to store data, set configuration options and customise the environment on the . The default shell under Scientific Linux on the is Bash (Bourne Again Shell) and can be used for the following purposes:

1. Configure look and feel of the shell.
2. Setup terminal settings depending on which terminal you're using.
3. Set the search path for running executables.
4. Set environment variables as needed by programs.
5. Set convenient abbreviations for heavily used values.
6. Run commands that you want to run whenever you log in or log out.
7. Setup aliases and/or shell function to automate tasks to save typing and time.
8. Changing the bash prompt.
9. Setting shell options.

The environment variables are typically set at login by a script, whenever you connect to the . These pre-defined variables usually impact the run time behaviour of the programs that we want to run.

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

In order to administer the active software and their environment variables, a ***"module"*** package has been developed, which:

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

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

3.3.2 Available modules

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

```
$ module av
```

or

```
$ module available
```

This will give some output such as:

This gives a full list of software packages that can be loaded. Note that modules starting with a capital letter are listed first.

3.3.3 Organisation of modules in toolchains

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

Therefore the VSC has defined so-called **toolchains** on the newer VSC-clusters. A toolchain contains a C/C++ and Fortran compiler, a MPI library and some basic math libraries for (dense matrix) linear algebra and FFT. Two toolchains are defined on most VSC systems. One, the "intel" toolchain, consists of the Intel compilers, MPI library and math libraries. The other one, the "foss" toolchain, consists of Open Source components: the GNU compilers, OpenMPI, OpenBLAS and the standard LAPACK and ScaLAPACK libraries for the linear algebra operations and the FFTW library for FFT. The toolchains are refreshed twice a year, which is reflected in their name. of the "foss" toolchain in 2014.

The toolchains are then used to compile a lot of the software installed on the VSC clusters. You can recognise those packages easily as they all contain the name of the toolchain after the version number in their name. Packages compiled with the same toolchain and toolchain version will typically work together well without conflicts.

For some clusters, additional toolchains are defined, e.g., to take advantage of specific properties of that cluster such as GPU accelerators or special interconnect features that require a vendor-specific MPI-implementation.

3.3.4 Activating and de-activating modules

A module is loaded using the following command:

This will load the most recent version of MATLAB.

For some packages, e.g., , multiple versions are installed; the load command will automatically choose the most recent version (i.e., the lexicographical last after the "/") or the default version (as set by the system administrators). However, the user can (and probably should, to avoid surprises when newer versions are installed) specify a particular version, e.g.,

Obviously, you need to keep track of the modules that are currently loaded. If you executed the two load commands stated above, you will get the following:

It is important to note at this point that other modules (e.g., intel/2014a) are also listed, although the user did not explicitly load them. This is because "Python/2.7.6-intel-2014a" depends on it (as indicated in its name), and the system administrator specified that the "intel/2014a" module should be loaded whenever the Python 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!

In fact, an easy way to check the components and version numbers of those components of a toolchain is to simply load the toolchain and then list the modules that are loaded.

To unload a module, one can use the "module unload" command. It works consistently with the load command, and reverses the latter's effect. However, the dependencies of the package are NOT automatically unloaded; the user shall unload the packages one by one. One can however unload automatically loaded modules manually, to debug some problem. When the "Python" module is unloaded, only the following modules remain:

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 in a clean state, you can use:

```
$ module purge
```

However, on some VSC clusters you may be left with a very empty list of available modules after executing "module purge". On those systems, "module av" will show you a list of modules containing the name of a cluster or a particular feature of a section of the cluster, and loading the appropriate module will restore the module list applicable to that particular system.

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

This will load the two modules as well as their dependencies.

3.3.5 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 behaviour.

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 1.12, compiled with Intel is installed on the cluster. The user loads the library using:

```
$ module load GSL
```

rather than

```
$ module load GSL/1.12
```

Everything works fine, up to the point where a new version of GSL is installed, 1.13 compiled for gcc. From then on, the user's load command will load the latter version, rather than the intended one, which may lead to unexpected problems.

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

3.3.6 Get detailed info

In order to know more about a certain package, and to know what environment variables will be changed by a certain module, try:

To get a list of all possible commands, type:

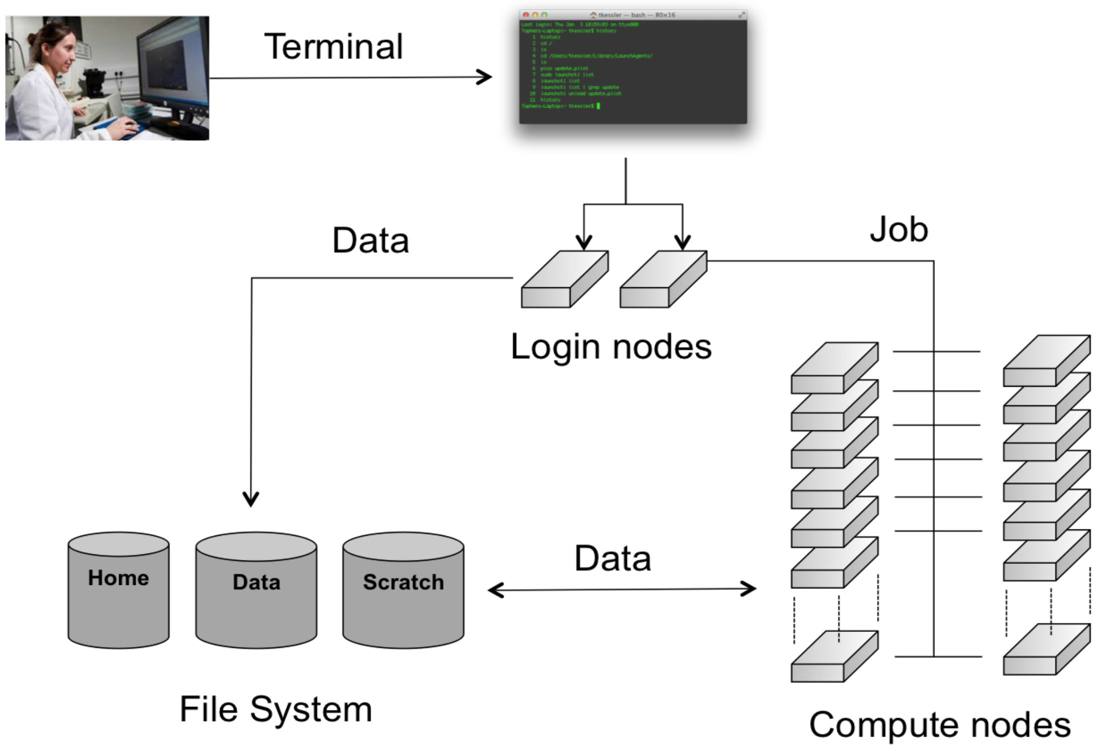
```
$ module help
```

Or to get more information about one specific module package:

Chapter 4

Running batch jobs

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



```
Last login: Thu Jan  3 18:59:03 on ttys000
Tophers-Laptop:~ tkessler$ history
    1  history
    2  cd /
    3  ls
    4  cd /Users/tkessler/Library/LaunchAgents/
    5  ls
    6  pico update.plist
    7  sudo launchctl list
    8  launchctl list
    9  launchctl list | grep update
   10  launchctl unload update.plist
   11  history
Tophers-Laptop:~ tkessler$ 
```

When you connect to the , you have access to (one of) the **login nodes** of the cluster. There you can prepare the work you want to get done on the cluster by, e.g., installing or compiling programs, setting up data sets, etc. The computations however, should not be performed on this login node. The actual work is done on the cluster's **compute nodes**. These compute nodes are managed by the job scheduling software (Moab) and a Resource Manager (TORQUE), which decides when and on which compute nodes the jobs can run. It is usually not necessary to log on to the compute nodes directly. Users can (and should) monitor their jobs periodically as they run, but do not have to remain logged in the entire time.

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

4.1 Defining and submitting your job

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

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

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

```
$ cd
$ cp -r ~/
```

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

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

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

The executable The program to execute from the end-user, together with its peripheral input files, databases and/or command options.

A configuration script (also called a job-script), which will define the computer resource requirements of the program, the required additional software packages and which will start the actual executable. The needs to know:

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

6. what executable to start, and its arguments.

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

List and check the contents with:

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

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

1. The Perl script calculates the first 30 Fibonacci numbers.

2. The job-script is actually a standard Unix/Linux shell script that contains a few extra comments at the beginning that specify directives to PBS. These comments all begin with **#PBS**.

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

On the command line, you would run this using:

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

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

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

— fibo.pbs —

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

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

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

```
$ qsub fibo.pbs
```

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

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

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

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

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

Explore the contents of the 2 new files:

```
$ more fibo.pbs.o
$ more fibo.pbs.e
```

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

4.2 Monitoring and managing your job(s)

Using the job ID that *qsub* returned, there are various ways to monitor the status of your job, e.g.,

To get the status information on your job:

```
$ qstat <jobid>
```

To show an estimated start time for your job (note that this may be very inaccurate, the margin of error on this figure can be bigger then 100% due to a sample in a population of 1.) This command is not available on all systems.

```
  $ showstart <jobid>
```

This is only a very rough estimate. Jobs may launch sooner than estimated if other jobs end faster than estimated, but may also be delayed if other higher-priority jobs enter the system.

To show the status, but also the resources required by the job, with error messages that may prevent your job from starting:

```
  $ checkjob <jobid>
```

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

```
$ qstat -n <jobid>
```

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

```
$ qdel <jobid>
```

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

```
$ qstat
 :
Job ID      Name    User      Time Use S Queue
----------- ------- --------- -------- - -----
 .... mpi      0        Q short
```

Here:

Job ID the job's unique identifier

Name the name of the job

User the user that owns the job

Time Use the elapsed walltime for the job

Queue the queue the job is in

The state S can be any of the following:

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

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

4.3 Examining the queue

As we learned above, Moab is the software application that actually decides when to run your job and what resources your job will run on. You can look at the queue by using the PBS **qstat** command or the Moab **showq** command. By default, **qstat** will display the queue ordered by **JobID**, whereas **showq** will display jobs grouped by their state ("running", "idle", or "hold") then ordered by priority. Therefore, **showq** is often more useful. Note however that at some VSC-sites, these commands show only your jobs or may be even disabled to not reveal what other users are doing.

The **showq** command displays information about active ("running"), eligible ("idle"), blocked ("hold"), and/or recently completed jobs. To get a summary:

```
$ showq -s
active jobs: 163
eligible jobs: 133
blocked jobs: 243
Total jobs:  539
```

There are 3 categories, the **active**, **eligible** and **blocked** jobs.

Active jobs are jobs that are running or starting and that consume computer resources. The amount of time remaining (w.r.t. walltime, sorted to earliest completion time) and the start time are displayed. This will give you an idea about the foreseen completion time. These jobs could be in a number of states:

- **Started** attempting to start, performing pre-start tasks
- **Running** currently executing the user application
- **Suspended** has been suspended by scheduler or admin (still in place on the allocated resources, not executing)
- **Cancelling** has been cancelled, in process of cleaning up

Eligible jobs are jobs that are waiting in the queues and are considered eligible for both scheduling and backfilling. They are all in the idle job state and do not violate any fairness policies or do not have any job holds in place. The requested walltime is displayed, and the list is ordered by job priority.

Blocked jobs are jobs that are ineligible to be run or queued. These jobs could be in a number of states for the following reasons:

- **Idle** when the job violates a fairness policy
- **Userhold** or systemhold when it is user or administrative hold

Batchhold when the requested resources are not available or the resource manager has repeatedly failed to start the job

Deferred when a temporary hold when the job has been unable to start after a specified number of attempts

Notqueued when scheduling daemon is unavailable

4.4 Specifying job requirements

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

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

4.4.1 Generic resource requirements

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

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

For the simplest cases, only the amount of maximum estimated execution time (called "walltime") is really important. Here, the job will not require more than 2 hours, 30 minutes to complete. As soon as the job would take more time, it will be "killed" (terminated) by the job scheduler. There is absolutly no harm if you *slightly* overestimate the maximum execution time.

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

The job requires no more than 4 GB of memory.

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

The job requires 5 compute nodes with two cores on each node (ppn stands for "processors per node", where processor is used to refer to individual cores).

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

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

section)[1] of the VSC website.

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

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

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

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

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

4.4.2 Available job categories (TORQUE queues)

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

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

The currently defined walltime categories for the are:

4.4.3 Node-specific properties

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

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

```
$ pbsnodes
```

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

4.5 Job output and error files

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

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

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

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

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

Inspect the generated output and error files:

```
$ cat fibo.pbs.o
...
$ cat fibo.pbs.e
...
```

4.6 E-mail notifications

4.6.1 Upon job failure

Whenever a job fails, an e-mail will be sent to the e-mail address that's connected to your <vsc-account>. This is the e-mail address that is linked to the university account, which was used during the registration process.

You can force a job to fail by specifying an unrealistic wall-time for the previous example. Lets give the "*fibo.pbs*" job just one second to complete:

```
  $ qsub -l walltime=0:00:01 fibo.pbs
```

Now, lets hope that the did not manage to run the job within one second, and you will get an e-mail informing you about this error.

```
  PBS Job Id:
  Job Name:   fibo.pbs
  Exec host:  /0
  Aborted by PBS Server
  Job exceeded some resource limit (walltime, mem, etc.). Job was aborted.
  See Administrator for help
```

4.6.2 Generate your own e-mail notifications

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

```
#PBS -m b
```

```
#PBS -m e
#PBS -m a
#PBS -M <your e-mail address>
```

or

```
#PBS -m abe
#PBS -M <your e-mail address>
```

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

```
$ qsub -m abe -M <your e-mail address> fibo.pbs
```

You don't have to specify the e-mail address. The system will use the e-mail address which is connected to your VSC account.

Chapter 5

Running interactive jobs

5.1 Introduction

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

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

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

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

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

5.2 Interactive jobs, without X support

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

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

```
$ hostname -f
```

This means that you're now working on the login-node "" of the cluster.

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

```
$ qsub -I
qsub: waiting for job  to start
qsub: job  ready
```

There are two things of note here.

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

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

```
$ hostname -f
```

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

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

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

You can exit the interactive session with:

```
$ exit
```

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

You can work for 3 hours by:

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

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

5.3 Interactive jobs, with graphical support

5.3.1 Software Installation

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

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

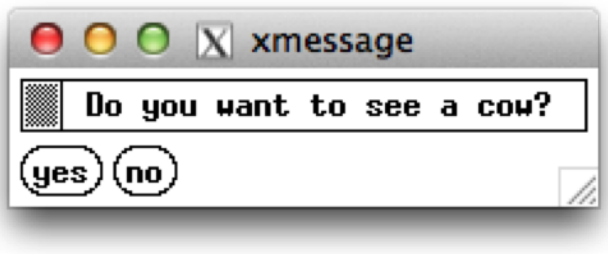
5.3.2 Run simple example

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

Now run the message program:

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

You should see the following message appearing.


Do you want to see a cow?

Click any button and see what happens.

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

5.3.3 Run your interactive application

In this last example, we will show you that you can just work on this compute node, just as if you were working locally on your desktop. We will run the Fibonacci example of the previous chapter again, but now in full interactive mode in MATLAB.

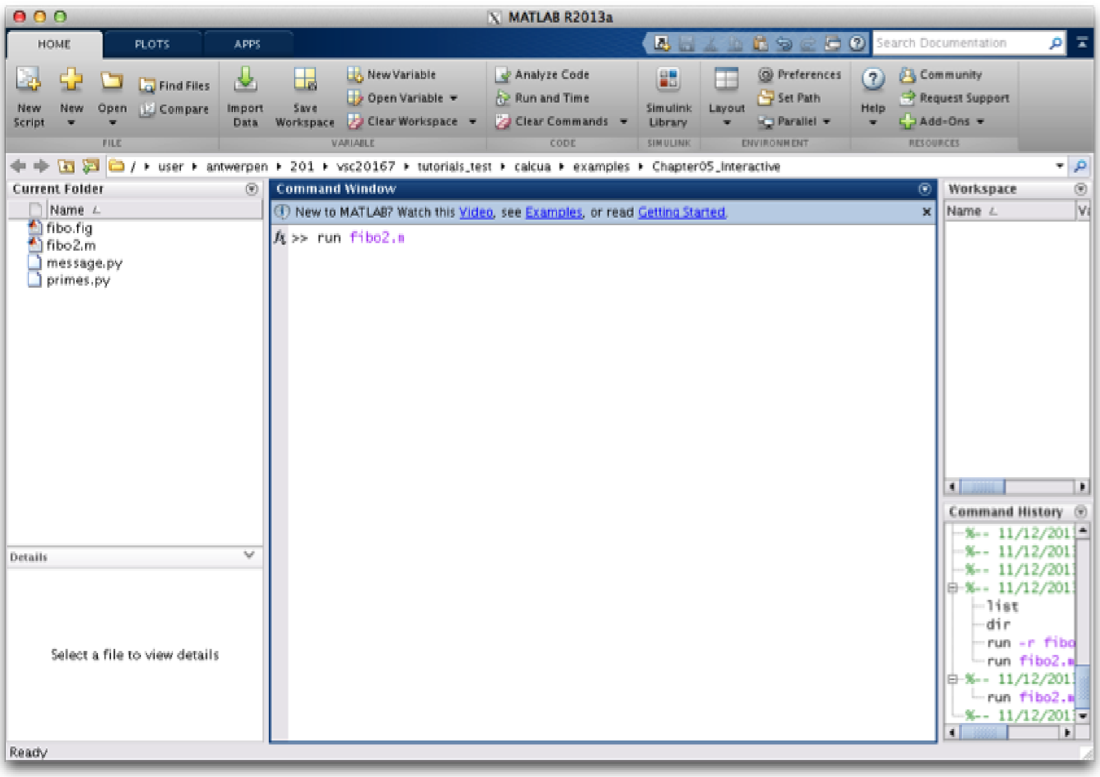
```
$ module load MATLAB
```

And start the MATLAB interactive environment:

```
$ matlab
```

And start the fibo2.m program in the command window:

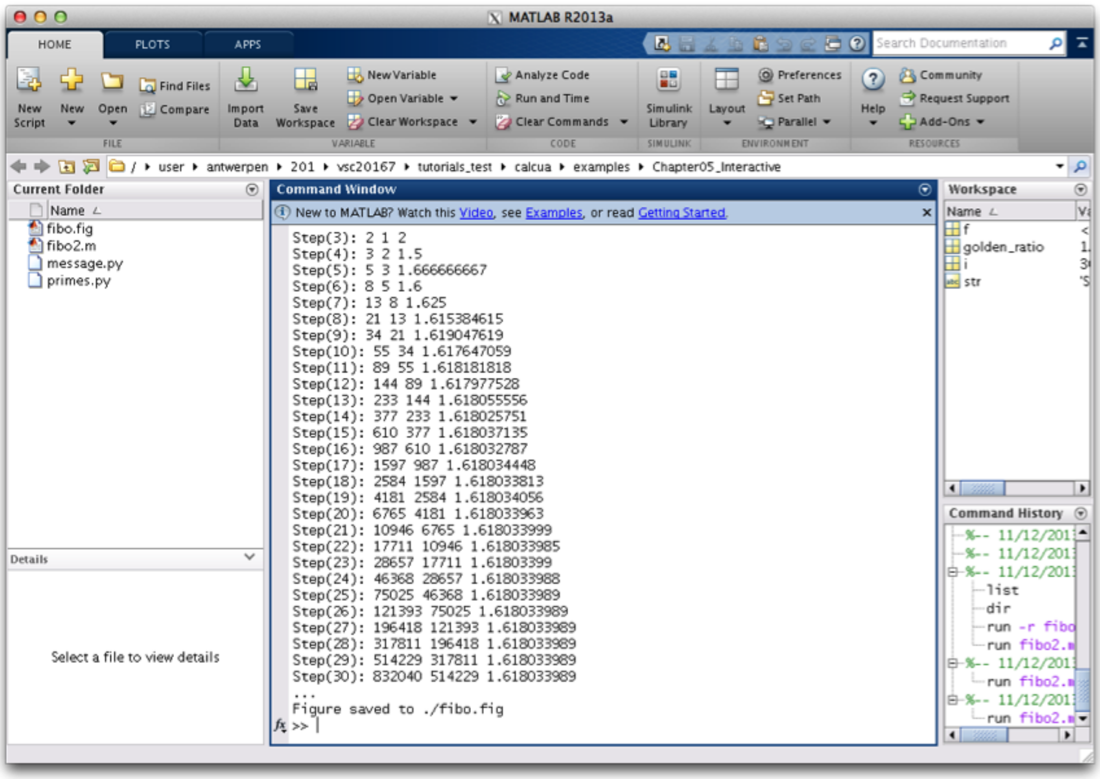
```
fx >> run fibo2.m
```



```
>> run fibo2.m
```

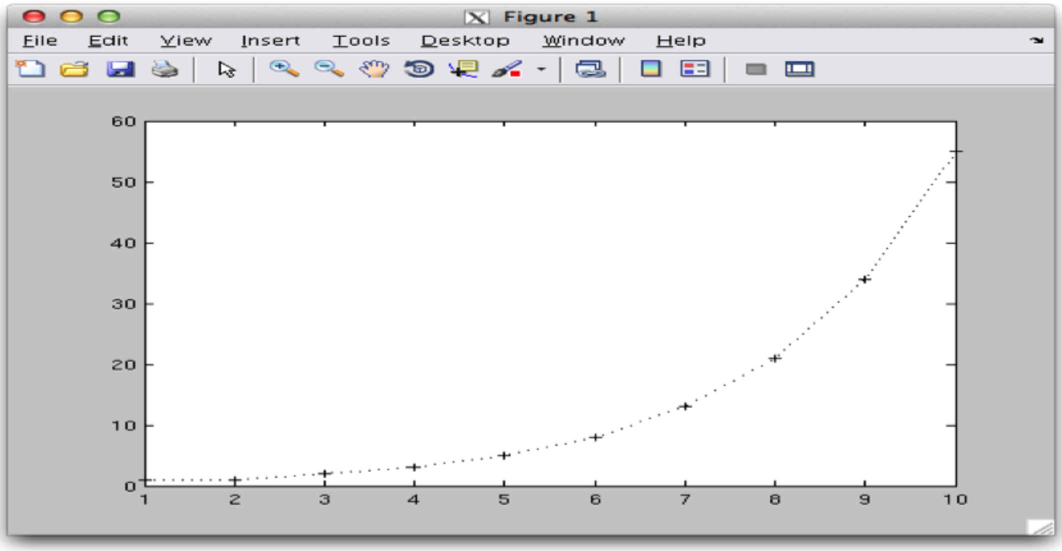
```
%-- 11/12/201
%-- 11/12/201
%-- 11/12/201
%-- 11/12/201
   list
   dir
   run -r fibo
   run fibo2.m
%-- 11/12/201
   run fibo2.m
%-- 11/12/201
```

And see the displayed calculations, . . .



```
Step(3): 2 1 2
Step(4): 3 2 1.5
Step(5): 5 3 1.666666667
Step(6): 8 5 1.6
Step(7): 13 8 1.625
Step(8): 21 13 1.615384615
Step(9): 34 21 1.619047619
Step(10): 55 34 1.617647059
Step(11): 89 55 1.618181818
Step(12): 144 89 1.617977528
Step(13): 233 144 1.618055556
Step(14): 377 233 1.618025751
Step(15): 610 377 1.618037135
Step(16): 987 610 1.618032787
Step(17): 1597 987 1.618034448
Step(18): 2584 1597 1.618033813
Step(19): 4181 2584 1.618034056
Step(20): 6765 4181 1.618033963
Step(21): 10946 6765 1.618033999
Step(22): 17711 10946 1.618033985
Step(23): 28657 17711 1.61803399
Step(24): 46368 28657 1.618033988
Step(25): 75025 46368 1.618033989
Step(26): 121393 75025 1.618033989
Step(27): 196418 121393 1.618033989
Step(28): 317811 196418 1.618033989
Step(29): 514229 317811 1.618033989
Step(30): 832040 514229 1.618033989
...
Figure saved to ./fibo.fig
fx >>
```

as well as the nice “plot” appearing:



You can work in this MATLAB GUI, and finally terminate the application by entering “**exit**” in the command window again.

```
fx >> exit
```

Chapter 6

Running jobs with input/output data

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

6.1 The current directory and output and error files

6.1.1 Default file names

First go to the directory:

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

List and check the contents with:

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

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

— file1.py —

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

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

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

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

The code of the Python script, is self explanatory:

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

Check the contents of the first job-script:

— file1a.pbs —

```
#!/bin/bash

#PBS -l walltime=00:05:00

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

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

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

Submit it:

```
$ qsub file1a.pbs
```

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

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

Some observations:

1. The file `Hello.txt` was created in the current directory.
2. The file `file1a.pbs.o` contains all the text that was written to the standard output stream ("stdout").
3. The file `file1a.pbs.e` contains all the text that was written to the standard error stream ("stderr").

Inspect their contents ... and remove the files

```
$ cat Hello.txt
$ cat file1a.pbs.o
$ cat file1a.pbs.e
$ rm Hello.txt file1a.pbs.o file1a.pbs.e
```

Tip: Type "`cat H`" and press the TAB button, and it will **expand** into full filename `Hello.txt`.

6.1.2 Filenames using the name of the job

Check the contents of the job script and execute it.

— file1b.pbs —

```
#!/bin/bash

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

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

```
$ qsub file1b.pbs
```

Inspect the contents again … and remove the generated files:

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

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

6.1.3 User-defined file names

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

— file1c.pbs —

```
#!/bin/bash

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

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

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

6.2 Where to store your data on the

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

6.2.1 Pre-defined user directories

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

The following locations are available:

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

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

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

6.2.2 Your home directory ($VSC_HOME)

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

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

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

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

Furthermore, we have initially created some files/directories there (tutorial, docs, examples, examples.pbs) that accompany this manual and allow you to easily execute the provided examples.

6.2.3 Your data directory ($VSC_DATA)

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

The environment variable pointing to this directory is $VSC_DATA. This volume is shared across all clusters of the VSC. There are however no guarantees about the speed you will achieve on this volume. For guaranteed fast performance and very heavy I/O, you should use the scratch space instead.

6.2.4 Your scratch space ($VSC_SCRATCH)

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

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

Each type of scratch has its own use:

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

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

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

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

6.2.5 Pre-defined quotas

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

The amount of data (called "*Block Limits*") that is currently in use by the user ("*KB*"), the soft limits ("*quota*") and the hard limits ("*limit*") for all 3 file-systems are always displayed when a user connects to the .

Whith regards to the *file limits*, the number of files in use ("*files*"), its soft limit ("*quota*") and its hard limit ("*limit*") for the 3 file-systems are also displayed.

```
---------------------------------------------------------
Your quota is:

                   Block Limits
   Filesystem          KB      quota      limit    grace
   home            177920    3145728    3461120     none
   data          17707776   26214400   28835840     none
   scratch         371520   26214400   28835840     none

                File Limits
   Filesystem       files      quota      limit    grace
   home               671      20000      25000     none
   data            103079     100000     150000  expired
   scratch           2214     100000     150000     none

---------------------------------------------------------
```

Make sure to regularly check these numbers at log-in!

The rules are:

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

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

6.3 Writing Output files

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

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

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

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

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

6.4 Reading Input files

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

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

In this exercise, you will

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

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

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

6.5 How much disk space do I get?

6.5.1 Quota

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

1. to the amount of disk space; and

2. to the number of files

that can be made available to each individual user.

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

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

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

6.5.2 Check your quota

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

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

or on the UAntwerp clusters

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

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

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

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

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

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

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

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

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

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

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

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

```
$ du -h $VSC_HOME/*
22M /dataset01
36M /dataset02
22M /dataset03
3.5M /primes.txt
```

We also want to mention the `tree` command, as it also provides an easy manner to see which files consumed your available quotas. *Tree* is a recursive directory-listing program that produces a depth indented listing of files.

Try:

```
$ tree -s -d
```

However, we urge you to only use the `du` and `tree` commands when you really need them as they can put a heavy strain on the file system and thus slow down file operations on the cluster for all other users.

Chapter 7

Multi core jobs/Parallel Computing

7.1 Why Parallel Programming?

There are two important motivations to engage in parallel programming.

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

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

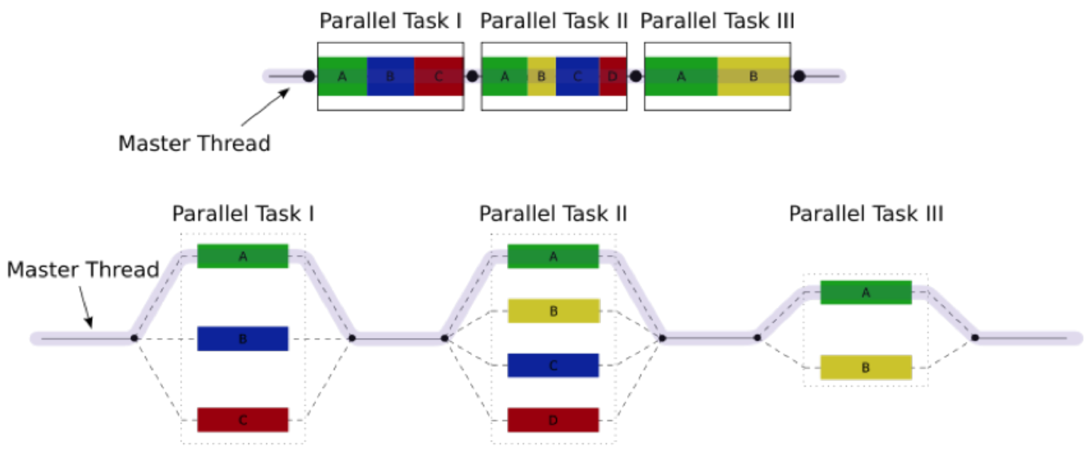
On a desktop computer, this enables a user to run multiple programs and the operating system simultaneously. For scientific computing, this means you have the ability in principle of splitting up your computations into groups and running each group on its own core.

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

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

7.2 Parallel Computing with threads

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



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

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

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

Go to the example directory:

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

Study the example first:

— T_hello.c —

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

#define NTHREADS 5

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

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

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

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

  return 1;
}
```

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

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

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

```
$ qsub T_hello.pbs

$ more T_hello.pbs.o
spawning thread 0
spawning thread 1
spawning thread 2
Hello from thread 0!
Hello from thread 1!
Hello from thread 2!
spawning thread 3
spawning thread 4
Hello from thread 3!
Hello from thread 4!
```

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

7.3 Parallel Computing with OpenMP

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

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

Here is the general code structure of an OpenMP program:

```
#include <omp.h>
main ()  {
```

```
int var1, var2, var3;
// Serial code
// Beginning of parallel section. Fork a team of threads.
// Specify variable scoping

#pragma omp parallel private(var1, var2) shared(var3)
  {
  // Parallel section executed by all threads
  // All threads join master thread and disband
  }
// Resume serial code
}
```

7.3.1 Private versus Shared variables

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

7.3.2 Parallelising for loops with OpenMP

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

— omp1.c —

```c
/*
 * VSC        : Flemish Supercomputing Centre
 * Tutorial   : Introduction to HPC
 * Description: Showcase program for OMP loops
 */
/* OpenMP_loop.c  */
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv)
{
  int i, thread_id, nloops;

#pragma omp parallel private(thread_id, nloops)
  {
    nloops = 0;

#pragma omp for
    for (i=0; i<1000; ++i)
    {
      ++nloops;
    }
    thread_id = omp_get_thread_num();
    printf("Thread %d performed %d iterations of the loop.\n", thread_id, nloops );
  }

  return 0;
}
```

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

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

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

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

7.3.3 Critical Code

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

— omp2.c —

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

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

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

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

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

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

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

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

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

7.3.4 Reduction

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

— omp3.c —

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

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

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

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

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

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

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

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

7.3.5 Other OpenMP directives

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

Some other clauses of interest are:

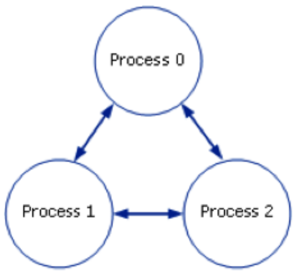
1. barrier: each thread will wait until all threads have reached this point in the code, before proceeding
2. nowait: threads will not wait until everybody is finished
3. schedule(type, chunk) allows you to specify how tasks are spawned out to threads in a for loop. There are three types of scheduling you can specify
4. if: allows you to parallelise only if a certain condition is met
5. ... and a host of others

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

7.4 Parallel Computing with MPI

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

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



Parallel Task

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

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

One context where MPI shines in particular is the ability to easily take advantage not just of multiple cores on a single machine, but to run programs on clusters of several machines. Even if you don’t have a dedicated cluster, you could still write a program using MPI that could run your program in parallel, across any collection of computers, as long as they are networked together.

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

Study the MPI-programme and the PBS-file:

— mpi_hello.c —

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

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

 #define BUFSIZE 128
 #define TAG 0

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

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

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

— mpi_hello.pbs —

```
#!/bin/bash

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

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

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

  # load the environment

module load intel

mpirun ./mpi_hello
```

and compile it:

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

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

Run the parallel program:

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

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

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

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

Chapter 8

Fine-tuning Job Specifications

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

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

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

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

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

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

This chapter shows you how to measure:

1. Walltime

2. Memory usage

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

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

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

8.1 Specifying Walltime

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

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

Test the time command:

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

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

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

The walltime can be specified in a job scripts as:

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

or on the command line

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

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

8.2 Specifying memory requirements

In many situations, it is useful to monitor the amount of memory an application is using. You need this information to determine the characteristics of the required compute node, where that application should run on. Estimating the amount of memory an application will use during execution is often non-trivial, especially when one uses third-party software.

The “eat_mem” application in the HPC examples directory just consumes and then releases memory, for the purpose of this test. It has one parameter, the amount of gigabytes of memory which needs to be allocated.

First compile the program on your machine and then test it for 1 GB:

```
  $ gcc -o eat_mem eat_mem.c
  $ ./eat_mem 1
  Consuming 1 gigabyte of memory.
```

8.2.1 Available Memory on the machine

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

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

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

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

8.2.2 Checking the memory consumption

The “Monitor” tool monitors applications in terms of memory and CPU usage, as well as the size of temporary files. Note that currently only single node jobs are supported, MPI support may be added in a future release.

To start using monitor, first load the appropriate module. Then we study the “eat_mem.c” program and compile it:

```
$ module load monitor
$ cat eat_mem.c
$ gcc -o eat_mem eat_mem.c
```

Starting a program to monitor is very straightforward; you just add the “monitor” command before the regular command line.

```
$ monitor ./eat_mem 3
time (s) size (kb) %mem %cpu
Consuming 3 gigabyte of memory.
5  252900 1.4 0.6
10  498592 2.9 0.3
15  743256 4.4 0.3
20  988948 5.9 0.3
25  1233612 7.4 0.3
30  1479304 8.9 0.2
35  1723968 10.4 0.2
40  1969660 11.9 0.2
45  2214324 13.4 0.2
50  2460016 14.9 0.2
55  2704680 16.4 0.2
60  2950372 17.9 0.2
65  3167280 19.2 0.2
70  3167280 19.2 0.2
75  9264  0 0.5
80  9264  0 0.4
```

Whereby:

1. The first column shows you the elapsed time in seconds. By default, all values will be displayed every 5 seconds.
2. The second column shows you the used memory in kb. We note that the memory slowly increases up to 3 GB (= 3072 KB), and is released again.
3. The third column shows the memory utilisation, expressed in percentages of the full available memory. At full memory consumption, 19.2% of the memory was being used by our application. With the *"free"* command, we have previously seen that we had a node of 16 GB in this example. 3 GB is indeed more or less 19.2% of the full available memory.
4. The fourth column shows you the CPU utilisation, expressed in percentages of a full CPU load. As there are no computations done in our exercise, the value remains very low (i.e., 0.2%).

Monitor will write the CPU usage and memory consumption of simulation to standard error.

This is the rate at which monitor samples the program's metrics. 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 test1.log eat_mem 2
Consuming 2 gigabyte of memory.
$ cat test1.log
```

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 test2.log -n 12 eat_mem 4
Consuming 4 gigabyte of memory.
```

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

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

```
$ monitor -d 1 ./eat_mem
Consuming 3 gigabyte of memory.
```

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

Alternative options to monitor the memory consumption are the "*top*" or the "*htop*" command.

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

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

```
$ top
$ htop
```

8.2.3 Setting the memory parameter

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

<u>Sequential or single-node applications:</u>

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

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

or on the command line

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

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

<u>Parallel or multi-node applications:</u>

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

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

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

or on the command line

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

In this example, you request 8 GB of memory in total on the node.

8.3 Specifying processors requirements

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

8.3.1 Number of processors

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

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

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

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

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

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

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

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

or with equivalent parameters on the command line

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

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

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

to request all cores of a node, or

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

to request half of them.

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

8.3.2 Monitoring the CPU-utilisation

The previously used "monitor" tool also shows the overall CPU-load. The "eat_cpu" program performs a multiplication of 2 randomly filled a 1500×1500 matrices and is just written to consumes a lot of "*cpu*".

We first load the monitor modules, study the "eat_cpu.c" program and compile it:

```
  $ module load monitor
  $ cat eat_cpu.c
  $ gcc -o eat_cpu eat_cpu.c
```

And then start to monitor the *eat_ cpu* program:

```
  $ monitor -d 1 ./eat_cpu
  time  (s) size (kb) %mem %cpu
  1  52852  0.3 100
  2  52852  0.3 100
  3  52852  0.3 100
  4  52852  0.3 100
  5  52852  0.3  99
  6  52852  0.3 100
  7  52852  0.3 100
  8  52852  0.3 100
```

We notice that it the program keeps its CPU nicely busy at 100%.

Some processes spawn one or more sub-processes. In that case, the metrics shown by monitor are aggregated over the process and all of its sub-processes (recursively). The reported CPU usage is the sum of all these processes, and can thus exceed 100%.

Some (well, since this is a 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 × 100%.

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

```
$ htop
```

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

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

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

If ***htop*** reports that your program is taking 75% CPU on a certain processor, it means that 75% of the samples taken by top found your process active on the CPU. The rest of the time your application was in a wait. (It is important to remember that a CPU is a discrete state machine. It really can be at only 100%, executing an instruction, or at 0%, waiting for something to do. There is no such thing as using 45% of a CPU. The CPU percentage is a function of time.) However, it is likely that your application's rest periods include waiting to be dispatched on a

CPU and not on external devices. That part of the wait percentage is then very relevant to understanding your overall CPU usage pattern.

8.3.3 Fine-tuning your executable and/or job-script

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

But how can you maximise?

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

8.4 The system load

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

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

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

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

8.4.1 Optimal load

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

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

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

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

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

1. When you are running **computational intensive applications**, one application per processor will generate the optimal load.

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

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

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

8.4.2 Monitoring the load

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

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

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

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

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

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

8.4.3 Fine-tuning your executable and/or job-script

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

But how can you maximise?

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

And then check again.

8.5 Checking File sizes & Disk I/O

8.5.1 Monitoring File sizes during execution

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

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

We first load the monitor modules, study the “eat_disk.c” program and compile it:

```
$ module load monitor
$ cat eat_disk.c
$ gcc -o eat_disk eat_disk.c
```

The *monitor* tool provides an option (-f) to display the size of one or more files:

```
$ monitor -f $VSC_SCRATCH/test.txt ./eat_disk
time (s) size (kb) %mem %cpu
5  1276  0 38.6 168820736
10  1276  0 24.8 238026752
15  1276  0 22.8 318767104
20  1276  0 25 456130560
25  1276  0 26.9 614465536
30  1276  0 27.7 760217600
...
```

Here, the size of the file "*test.txt*" in directory $VSC_SCRATCH will be monitored. Files can be specified by absolute as well as relative path, and multiple files are separated by ",".

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

Several actions can be taken, to avoid storage problems:

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

8.6 Specifying network requirements

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

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

The parameter to add in your job-script would be:

```
#PBS -l ib
```

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

```
#PBS -l gbe
```

8.7 Some more tips on the Monitor tool

8.7.1 Command Lines arguments

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.

8.7.2 Exit Code

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 terminates 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.

8.7.3 Monitoring a running process

It is also possible to "attach" monitor to a program or process that is already running. One simply determines the relevant process ID using the ps command, e.g., 18749, and starts monitor:

```
  $ monitor -p 18749
```

Note that this feature can be (ab)used to monitor specific sub-processes.

Part II

Advanced Guide

Chapter 9

Multi-job submission

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

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

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

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

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

It can handle many small jobs determined by:

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

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

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

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

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

9.1 The worker Framework: Parameter Sweeps

First go to the right directory:

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

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

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

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

weather– par_sweep/weather —

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

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

weather_p01.pbs– par_sweep/weather_p01.pbs —

```
#!/bin/bash

#PBS -l nodes=1:ppn=8
#PBS -l walltime=01:00:00

cd $PBS_O_WORKDIR
./weather -t 20 -p 1.05 -v 4.3
```

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

To submit the job, the user would use:

```
$ qsub weather_p01.pbs
```

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

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

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

In order to make our PBS generic, the PBS file can be modified as follows:

weather.pbs– par_sweep/weather.pbs —

```
#!/bin/bash

#PBS -l nodes=1:ppn=8
#PBS -l walltime=04:00:00

cd $PBS_O_WORKDIR
./weather -t $temperature -p $pressure -v $volume

# # This script is submitted to the cluster with the following 2 commands:
# module load worker
# wsub -data data.csv -batch weather.sh
```

Note that:

1. the parameter values 20, 1.05, 4.3 have been replaced by variables $temperature, $pressure and $volume respectively, which were being specified on the first line of the "*data.csv*" file;
2. the number of processors per node has been increased to 8 (i.e., ppn=1 is replaced by ppn=8);
3. the walltime has been increased to 4 hours (i.e., walltime=00:15:00 is replaced by walltime=04:00:00).

The walltime is calculated as follows: one calculation takes 15 minutes, so 100 calculations take 1500 minutes on one CPU. However, this job will use 8 CPUs, so the 100 calculations will be done in 1500/8 = 187.5 minutes, i.e., 4 hours to be on the safe side.

The job can now be submitted as follows:

```
$ module load worker
$ wsub -batch weather.pbs -data data.csv
total number of work items: 41
```

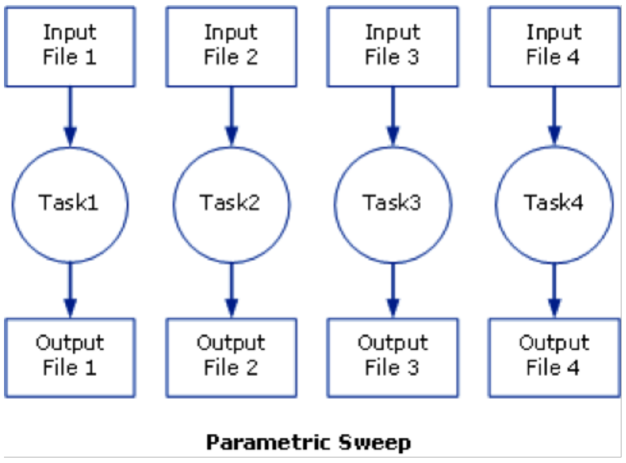
Note that the PBS file is the value of the -batch option. The weather program will now be run for all 100 parameter instances – 8 concurrently – until all computations are done. A computation for such a parameter instance is called a work item in Worker parlance.

9.2 The Worker framework: Job arrays

First go to the right directory:

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

As a simple example, assume you have a serial program called *myprog* that you want to run on various input files *input[1-100]*.



The following bash script would submit these jobs all one by one:

```
#!/bin/bash
for i in `seq 1 100`; do
   qsub -o output $i -i input $i myprog.pbs
done
```

This, as said before, could be disturbing for the job scheduler.

Alternatively, TORQUE provides a feature known as *job arrays* which allows the creation of multiple, similar jobs with only **one qsub** command. This feature introduced a new job naming convention that allows users either to reference the entire set of jobs as a unit or to reference one particular job from the set.

Under TORQUE, the *-t range* option is used with qsub to specify a job array, where *range* is a range of numbers (e.g., *1-100* or *2,4-5,7*).

The details are

1. a job is submitted for each *number* in the range;

2. individuals jobs are referenced as *jobid-number*, and the entire array can be referenced as *jobid* for easy killing etc.; and

3. each jobs has *PBS_ARRAYID* set to its *number* which allows the script/program to specialise for that job

The job could have been submitted using:

```
$ qsub -t 1-100 my_prog.pbs
```

The effect was that rather than 1 job, the user would actually submit 100 jobs to the queue system. This was a popular feature of TORQUE, but as this technique puts quite a burden on the scheduler, it is not supported by Moab (the current job scheduler).

To support those users who used the feature and since it offers a convenient workflow, the "worker framework" implements the idea of "job arrays" in its own way.

A typical job-script for use with job arrays would look like this:

job_array.pbs– job_array/job_array.pbs —

```
#!/bin/bash -l
#PBS -l nodes=1:ppn=1
#PBS -l walltime=00:15:00
cd $PBS_O_WORKDIR
INPUT_FILE="input_${PBS_ARRAYID}.dat"
OUTPUT_FILE="output_${PBS_ARRAYID}.dat"
my_prog -input ${INPUT_FILE}  -output ${OUTPUT_FILE}
```

In our specific example, we have prefabricated 100 input files in the "./input" subdirectory. Each of those files contains a number of parameters for the "test_set" program, which will perform some tests with those parameters.

Input for the program is stored in files with names such as input_1.dat, input_2.dat, . . . , input_100.dat in the ./input subdirectory.

```
$ ls ./input
...
$ more ./input/input_99.dat
This is input file \#99
Parameter #1 = 99
Parameter #2 = 25.67
Parameter #3 = Batch
Parameter #4 = 0x562867
```

For the sole purpose of this exercise, we have provided a short "test_set" program, which reads the "input" files and just copies them into a corresponding output file. We even add a few lines to each output file. The corresponding output computed by our "*test_set*" program will be written to the "*./output*" directory in output_1.dat, output_2.dat, . . . , output_100.dat. files.

test_set– job_array/test_set —

```
#!/bin/bash

# Check if the output Directory exists
if [ ! -d "./output" ] ; then
  mkdir ./output
fi

#   Here you could do your calculations...
echo "This is Job_array #" $1
echo "Input File : " $3
echo "Output File: " $5
cat ./input/$3 | sed -e "s/input/output/g" | grep -v "Parameter" > ./output/$5
echo "Calculations done, no results" >> ./output/$5
```

Using the "worker framework", a feature akin to job arrays can be used with minimal modifications to the job-script:

test_set.pbs– job_array/test_set.pbs —

```
#!/bin/bash -l
#PBS -l nodes=1:ppn=8
#PBS -l walltime=04:00:00
cd $PBS_O_WORKDIR
INPUT_FILE="input_${PBS_ARRAYID}.dat"
OUTPUT_FILE="output_${PBS_ARRAYID}.dat"
./test_set ${PBS_ARRAYID} -input ${INPUT_FILE}  -output ${OUTPUT_FILE}
```

Note that

1. the number of CPUs is increased to 8 (ppn=1 is replaced by ppn=8); and

2. the walltime has been modified (walltime=00:15:00 is replaced by walltime=04:00:00).

The job is now submitted as follows:

```
$ module load worker
$ wsub -t 1-100 -batch test_set.pbs
total number of work items: 100
```

The "*test_set*" program will now be run for all 100 input files – 8 concurrently – until all computations are done. Again, a computation for an individual input file, or, equivalently, an array id, is called a work item in Worker speak.

Note that in contrast to TORQUE job arrays, a worker job array only submits a single job.

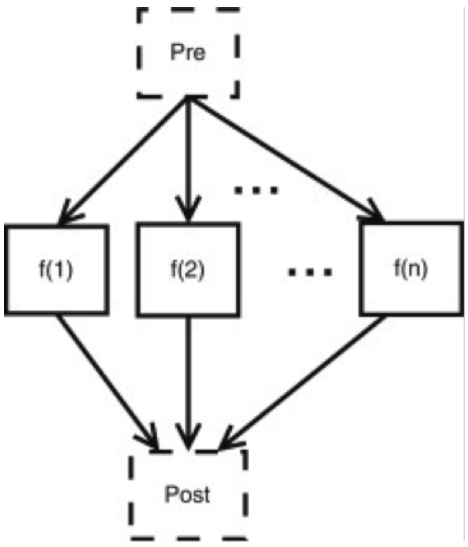
```
$ qstat
Job id          Name          User      Time   Use S Queue
--------------- ------------- --------- ---- ----- - -----
  test_set.pbs            0 Q

And you can now check the generated output files:
$ more ./output/output_99.dat
This is output file #99
Calculations done, no results
```

9.3 MapReduce: prologues and epilogue

Often, an embarrassingly parallel computation can be abstracted to three simple steps:

1. a preparation phase in which the data is split up into smaller, more manageable chunks;
2. on these chunks, the same algorithm is applied independently (these are the work items); and
3. the results of the computations on those chunks are aggregated into, e.g., a statistical description of some sort.



The Worker framework directly supports this scenario by using a prologue (pre-processing) and an epilogue (post-processing). The former is executed just once before work is started on the work items, the latter is executed just once after the work on all work items has finished. Technically, the master, i.e., the process that is responsible for dispatching work and logging progress, executes the prologue and epilogue.

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

The script “pre.sh” prepares the data by creating 100 different input-files, and the script “post.sh” aggregates (concatenates) the data.

First study the scripts:

pre.sh– map_reduce/pre.sh —

```
#!/bin/bash

# Check if the input Directory exists
if [ ! -d "./input" ] ; then
  mkdir ./input
fi

# Just generate all dummy input files
for i in {1..100};
do
  echo "This is input file #$i" >  ./input/input_$i.dat
  echo "Parameter #1 = $i" >>  ./input/input_$i.dat
  echo "Parameter #2 = 25.67" >>  ./input/input_$i.dat
  echo "Parameter #3 = Batch" >>  ./input/input_$i.dat
  echo "Parameter #4 = 0x562867" >>  ./input/input_$i.dat
done
```

post.sh– map_reduce/post.sh —

```
#!/bin/bash

# Check if the input Directory exists
if [ ! -d "./output" ] ; then
  echo "The output directory does not exist!"
  exit
fi

# Just concatenate all output files
touch all_output.txt
for i in {1..100};
do
  cat ./output/output_$i.dat >> all_output.txt
done
```

Then one can submit a MapReduce style job as follows:

```
$ wsub -prolog pre.sh -batch test_set.pbs -epilog post.sh -t 1-100
total number of work items: 100

$ cat all_output.txt
...
$ rm -r -f ./output/
```

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

9.4 Some more on the Worker Framework

9.4.1 Using Worker efficiently

The “Worker Framework” is implemented using MPI, so it is not restricted to a single compute nodes, it scales well to multiple nodes. However, remember that jobs requesting a large number of nodes typically spend quite some time in the queue.

The “Worker Framework” will be effective when

1. work items, i.e., individual computations, are neither too short, nor too long (i.e., from a few minutes to a few hours); and,
2. when the number of work items is larger than the number of CPUs involved in the job (e.g., more than 30 for 8 CPUs).

9.4.2 Monitoring a worker job

Since a Worker job will typically run for several hours, it may be reassuring to monitor its progress. Worker keeps a log of its activity in the directory where the job was submitted. The log’s name is derived from the job’s name and the job’s ID, i.e., it has the form <jobname>.log<jobid>. For the running example, this could be “run.pbs.log”, assuming the job’s ID is . To keep an eye on the progress, one can use:

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

Alternatively, “wsummarize”, a Worker command that summarises a log file, can be used:

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

This will summarise the log file every 60 seconds.

9.4.3 Time limits for work items

Sometimes, the execution of a work item takes long than expected, or worse, some work items get stuck in an infinite loop. This situation is unfortunate, since it implies that work items that could successfully execute are not even started. Again, the Worker framework offers a simple and yet versatile solution. If we want to limit the execution of each work item to at most 20 minutes, this can be accomplished by modifying the script of the running example.

```
#!/bin/bash -l
#PBS -l nodes=1:ppn=8
#PBS -l walltime=04:00:00
module load timedrun/1.0
cd $PBS_O_WORKDIR
timedrun -t 00:20:00 weather -t $temperature  -p $pressure  -v $volume
```

Note that it is trivial to set individual time constraints for work items by introducing a parameter, and including the values of the latter in the CSV file, along with those for the temperature, pressure and volume.

Also note that “timedrun” is in fact offered in a module of its own, so it can be used outside the Worker framework as well.

9.4.4 Resuming a Worker job

Unfortunately, it is not always easy to estimate the walltime for a job, and consequently, sometimes the latter is underestimated. When using the Worker framework, this implies that not all work items will have been processed. Worker makes it very easy to resume such a job without having to figure out which work items did complete successfully, and which remain to be computed. Suppose the job that did not complete all its work items had ID “445948”.

```
$ wresume -jobid
```

This will submit a new job that will start to work on the work items that were not done yet. Note that it is possible to change almost all job parameters when resuming, specifically the requested resources such as the number of cores and the walltime.

```
$ wresume -l walltime=1:30:00 -jobid
```

Work items may fail to complete successfully for a variety of reasons, e.g., a data file that is missing, a (minor) programming error, etc. Upon resuming a job, the work items that failed are considered to be done, so resuming a job will only execute work items that did not terminate either successfully, or reporting a failure. It is also possible to retry work items that failed (preferably after the glitch why they failed was fixed).

```
$ wresume -jobid -retry
```

By default, a job’s prologue is not executed when it is resumed, while its epilogue is. “wresume” has options to modify this default behaviour.

9.4.5 Further information

This how-to introduces only Worker’s basic features. The wsub command has some usage information that is printed when the -help option is specified:

```
$ wsub -help
### usage: wsub  -batch <batch-file>          \
#                [-data <data-files>]         \
#                [-prolog <prolog-file>]      \
#                [-epilog <epilog-file>]      \
#                [-log <log-file>]            \
#                [-mpiverbose]                \
#                [-dryrun] [-verbose]         \
#                [-quiet] [-help]             \
#                [-t <array-req>]             \
#                [<pbs-qsub-options>]
#
#   -batch <batch-file>   : batch file template, containing variables to be
#                           replaced with data from the data file(s) or the
#                           PBS array request option
#   -data <data-files>    : comma-separated list of data files (default CSV
#                           files) used to provide the data for the work
#                           items
#   -prolog <prolog-file> : prolog script to be executed before any of the
#                           work items are executed
#   -epilog <epilog-file> : epilog script to be executed after all the work
#                           items are executed
#   -mpiverbose           : pass verbose flag to the underlying MPI program
#   -verbose              : feedback information is written to standard error
#   -dryrun               : run without actually submitting the job, useful
#   -quiet                : don't show information
#   -help                 : print this help message
#   -t <array-req>        : qsub's PBS array request options, e.g., 1-10
#   <pbs-qsub-options>    : options passed on to the queue submission
#                           command
```

Chapter 10

Compiling and testing your software on the HPC

All nodes in the cluster are running the “” Operating system, which is a specific version of RedHat-Linux. This means that all the software programs (executable) that the end-user wants to run on the first must be compiled for . It also means that you first have to install all the required external software packages on the .

Most commonly used compilers are already pre-installed on the and can be used straight away. Also many popular external software packages, which are regularly used in the scientific community, are also pre-installed.

10.1 Check the pre-installed software on the

In order to check all the available modules and their version numbers, which are pre-installed on the enter:

When your required application is not available on the please contact any member. Be aware of potential “License Costs”. “Open Source” software is often preferred.

10.2 Porting your code

To **port** a software-program is to translate it from the operating system in which it was developed (e.g., Windows 7) to another operating system (e.g., on our) so that it can be used there. Porting implies some degree of effort, but not nearly as much as redeveloping the program in the new environment. It all depends on how “portable” you wrote your code.

In the simplest case the file or files may simply be copied from one machine to the other. However, in many cases the software is installed on a computer in a way, which depends upon its detailed hardware, software, and setup, with device drivers for particular devices, using installed operating system and supporting software components, and using different directories.

In some cases software, usually described as “portable software” is specifically designed to run on different computers with compatible operating systems and processors without any machinedependent installation; it is sufficient to transfer specified directories and their contents. Hardware- and software-specific information is often stored in configuration files in specified locations (e.g., the registry on machines running MS Windows).

Software, which is not portable in this sense, will have to be transferred with modifications to support the environment on the destination machine.

Whilst programming, it would be wise to stick to certain standards (e.g., ISO/ANSI/POSIX). This will ease the porting of your code to other platforms.

Porting your code to the platform is the responsibility of the end-user.

10.3 Compiling and building on the

Compiling refers to the process of translating code written in some programming language, e.g., Fortran, C, or C++, to machine code. Building is similar, but includes gluing together the machine code resulting from different source files into an executable (or library). The text below guides you through some basic problems typical for small software projects. For larger projects it is more appropriate to use makefiles or even an advanced build system like CMake.

All the nodes run the same version of the Operating System, i.e. . So, it is sufficient to compile your program on any compute node. Once you have generated an executable with your compiler, this executable should be able to run on any other compute-node.

A typical process looks like:

1. Copy your software to the login-node of the
2. Start an interactive session on a compute node;
3. Compile it;
4. Test it locally;
5. Generate your job-scripts;
6. Test it on the
7. Run it (in parallel);

We assume you've copied your software to the The next step is to request your private compute node.

```
$ qsub -I
qsub: waiting for job  to start
```

10.3.1 Compiling a sequential program in C

Go to the examples for **??** and load the foss module:

```
$ cd ~/examples/Compiling-and-testing-your-software-on-the-HPC
$ module load foss
```

We now list the directory and explore the contents of the "*hello.c*" program:

```
$ ls -l
total 512
-rw-r--r-- 1  214 Sep 16 09:42 hello.c
-rw-r--r-- 1  130 Sep 16 11:39 hello.pbs*
-rw-r--r-- 1  359 Sep 16 13:55 mpihello.c
-rw-r--r-- 1  304 Sep 16 13:55 mpihello.pbs
```

— hello.c —

```c
/*
 * VSC        : Flemish Supercomputing Centre
 * Tutorial   : Introduction to HPC
 * Description: Print 500 numbers, whilst waiting 1 second in between
 */
#include "stdio.h"
int main( int argc, char *argv[] )
{
  int i;
  for (i=0; i<500; i++)
  {
    printf("Hello #%d\n", i);
    fflush(stdout);
    sleep(1);
  }
}
```

The "hello.c" program is a simple source file, written in C. It'll print 500 times "Hello #<num>", and waits one second between 2 printouts.

We first need to compile this C-file into an executable with the gcc-compiler.

First, check the command line options for *"gcc" (GNU C-Compiler)*, then we compile and list the contents of the directory again:

```
$ gcc -help
$ gcc -o hello hello.c
$ ls -l
total 512
-rwxrwxr-x 1  7116 Sep 16 11:43 hello*
-rw-r--r-- 1   214 Sep 16 09:42 hello.c
-rwxr-xr-x 1   130 Sep 16 11:39 hello.pbs*
```

A new file "hello" has been created. Note that this file has "execute" rights, i.e., it is an executable. More often than not, calling gcc – or any other compiler for that matter – will provide you with a list of errors and warnings referring to mistakes the programmer made, such as typos, syntax errors. You will have to correct them first in order to make the code compile. Warnings pinpoint less crucial issues that may relate to performance problems, using unsafe or obsolete language features, etc. It is good practice to remove all warnings from a compilation process, even if they seem unimportant so that a code change that produces a warning does not go unnoticed.

Let's test this program on the local compute node, which is at your disposal after the "qsub –I" command:

```
$ ./hello
Hello #0
Hello #1
Hello #2
Hello #3
Hello #4
...
```

It seems to work, now run it on the

```
$ qsub hello.pbs
```

10.3.2 Compiling a parallel program in C/MPI

```
$ cd ~/examples/Compiling-and-testing-your-software-on-the-HPC
```

List the directory and explore the contents of the "*mpihello.c*" program:

```
$ ls -l
total 512
total 512
-rw-r--r-- 1  214 Sep 16 09:42 hello.c
-rw-r--r-- 1  130 Sep 16 11:39 hello.pbs*
-rw-r--r-- 1  359 Sep 16 13:55 mpihello.c
-rw-r--r-- 1  304 Sep 16 13:55 mpihello.pbs
```

— mpihello.c —

```c
/*
 * VSC        : Flemish Supercomputing Centre
 * Tutorial   : Introduction to HPC
 * Description: Example program, to compile with MPI
 */
#include <stdio.h>
#include <mpi.h>

main(int argc, char **argv)
{
  int node, i, j;
  float f;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &node);

  printf("Hello World from Node %d.\n", node);
  for (i=0; i<=100000; i++)
    f=i*2.718281828*i+i+i*3.141592654;

  MPI_Finalize();
}
```

The "mpi_hello.c" program is a simple source file, written in C with MPI library calls.

Then, check the command line options for *"mpicc" (GNU C-Compiler with MPI extensions)*, then we compile and list the contents of the directory again:

```
$ mpicc -help
$ mpicc -o mpihello mpihello.c
$ ls -l
```

A new file "hello" has been created. Note that this program has "execute" rights.

Let's test this program on the "login"-node first:

```
$ ./mpihello
Hello World from Node 0.
```

It seems to work, now run it on the .

```
$ qsub mpihello.pbs
```

10.3.3 Compiling a parallel program in Intel Parallel Studio Cluster Edition

We will now compile the same program, but using the Intel Parallel Studio Cluster Edition compilers. We stay in the examples directory for this chapter:

```
$ cd ~/examples/Compiling-and-testing-your-software-on-the-HPC
```

We will compile this C/MPI -file into an executable with the Intel Parallel Studio Cluster Edition. First, clear the modules (purge) and then load the latest "intel" module:

```
$ module purge
$ module load intel
```

Then, compile and list the contents of the directory again. The Intel equivalent of mpicc is mpiicc.

```
$ mpiicc -o mpihello mpihello.c
$ ls -l
```

Note that the old "mpihello" file has been overwritten. Let's test this program on the "login"-node first:

```
$ ./mpihello
Hello World from Node 0.
```

It seems to work, now run it on the .

```
$ qsub mpihello.pbs
```

Note: The only has a license for the Intel Parallel Studio Cluster Edition for a fixed number of users. As such, it might happen that you have to wait a few minutes before a floating license becomes available for your use.

Note: The Intel Parallel Studio Cluster Edition contains equivalent compilers for all GNU compilers. Hereafter the overview for C, C++ and Fortran compilers.

	Sequential Program		**Parallel Program (with MPI)**	
	GNU	**Intel**	**GNU**	**Intel**
C	gcc	icc	mpicc	mpiicc
C++	g++	icpc	mpicxx	mpiicpc
Fortran	gfortran	ifort	mpif90	mpiifort

Chapter 11

Program examples

Go to our examples:

```
$ cd ~/examples/Program-examples
```

Here, we just have put together a number of examples for your convenience. We did an effort to put comments inside the source files, so the source code files are (should be) self-explanatory.

1. 01_Python
2. 02_C_C++
3. 03_Matlab
4. 04_MPI_C
5. 05a_OMP_C
6. 05b_OMP_FORTRAN
7. 06_NWChem
8. 07_Wien2k
9. 08_Gaussian
10. 09_Fortran
11. 10_PQS

The above 2 OMP directories contain the following examples:

C Files	**Fortran Files**	**Description**
omp_hello.c	omp_hello.f	Hello world
omp_workshare1.c	omp_workshare1.f	Loop work-sharing
omp_workshare2.c	omp_workshare2.f	Sections work-sharing
omp_reduction.c	omp_reduction.f	Combined parallel loop reduction
omp_orphan.c	omp_orphan.f	Orphaned parallel loop reduction
omp_mm.c	omp_mm.f	Matrix multiply
omp_getEnvInfo.c	omp_getEnvInfo.f	Get and print environment information
omp_bug1.c omp_bug1fix.c omp_bug2.c omp_bug3.c omp_bug4.c omp_bug4fix omp_bug5.c omp_bug5fix.c omp_bug6.c	omp_bug1.f omp_bug1fix.f omp_bug2.f omp_bug3.f omp_bug4.f omp_bug4fix omp_bug5.f omp_bug5fix.f omp_bug6.f	Programs with bugs and their solution

Compile by any of the following commands:

C:	icc -openmp omp_hello.c -o hello pgcc -mp omp_hello.c -o hello gcc -fopenmp omp_hello.c -o hello
Fortran:	ifort -openmp omp_hello.f -o hello pgf90 -mp omp_hello.f -o hello gfortran -fopenmp omp_hello.f -o hello

Be invited to explore the examples.

Chapter 12

Good Practices

1. Before starting you should always check:
   (a) Are there any errors in the script?
   (b) Are the required modules loaded?
   (c) Is the correct executable used?
2. Check your computer requirements upfront, and request the correct resources in your PBS configuration script.
   (a) Number of requested cores
   (b) Amount of requested memory
   (c) Requested network type
3. Check your jobs at runtime. You could login to the node and check the proper execution of your jobs with, e.g., “top” or ”vmstat”. Alternatively you could run an interactive job (“qsub -I”).
4. Try to benchmark the software for scaling issues when using MPI or for I/O issues.
5. Use the scratch file system ($VSC_SCRATCH_NODE which is mapped to the local /tmp) whenever possible. Local disk I/O is always much faster as it does not have to use the network.
6. When your job starts, it will log on to the compute node(s) and start executing the commands in the job script. It will start in your home directory ($VSC_HOME), so going to the current directory with the “cd $PBS_O_WORKDIR” is the first thing which needs to be done. You will have your default environment, so don’t forget to load the software with “module load”.
7. In case your job not running, use “checkjob”. It will show why your job is not yet running. Sometimes commands might timeout with an overloaded scheduler.
8. Submit your job and wait (be patient) . . .
9. Submit small jobs by grouping them together. The “Worker Framework” has been designed for these purposes.

10. The runtime is limited by the maximum walltime of the queues. For longer walltimes, use checkpointing.
11. Requesting many processors could imply long queue times.
12. For all parallel computing, request to use "Infiniband".
13. And above all ... do not hesitate to contact the staff. We're here to help you.

12.1 Windows / Unix

Important note: the PBS file on the has to be in UNIX format, if it is not, your job will fail and generate rather weird error messages.

If necessary, you can convert it using

```
$ dos2unix file.pbs
```

Appendix A

Quick Reference Guide

Login	
Login	ssh <vsc-account>@
Where am I?	hostname
Copy to	scp foo.txt <vsc-account>@:
Copy from	scp <vsc-account>@:foo.txt .
Setup ftp session	sftp <vsc-account>@

Modules	
List all available modules	module av
List loaded modules	module list
Load module	module load <name>
Unload module	module unload <name>
Unload all modules	module purge
Help on use of module	module help

Jobs	
Submit Job	qsub <script.pbs>
Status of the job	qstat <jobid>
Possible start time (not available everywhere)	showstart <jobid>
Check job (not available everywhere)	checkjob <jobid>
Show compute node	qstat -n <jobid>
Delete job	qdel <jobid>
Status of all your jobs	qstat
Show all jobs on queue (not available everywhere)	showq
Submit Interactive job	qsub -I

Disk quota	
Check your disk quota	mmlsquota
Check disk quota nice	show_quota.py
Local disk usage	du -h .
Overall disk usage	df -a

Worker Framework	
Load worker module	module load worker
Submit parameter sweep	wsub -batch weather.pbs -data data.csv
Submit job array	wsub -t 1-100 -batch test_set.pbs
Submit job array with prolog and epilog	wsub -prolog pre.sh -batch test_set.pbs -epilog post.sh -t 1-100

Appendix B

TORQUE options

B.1 TORQUE Submission Flags: common and useful directives

Below is a list of the most common and useful directives.

Option	**System type**	**Description**
-k	All	Send “stdout” and/or “stderr” to your home directory when the job runs **#PBS -k o** or **#PBS -k e** or **#PBS -koe**
-l	All	Precedes a resource request, e.g., processors, wallclock
-M	All	Send an e-mail messages to an alternative e-mail address **#PBS -M me@mymail.be**
-m	All	Send an e-mail address when a job **b**egins execution and/or **e**nds or **a**borts **#PBS -m b** or **#PBS -m be** or **#PBS -m ba**
mem	Shared Memory	Specifies the amount of memory you need for a job. **#PBS -l mem=80gb**
mpiprocs	Clusters	Number of processes per node on a cluster. This should equal number of processors on a node in most cases. **#PBS -l mpiprocs=4**
-N	All	Give your job a unique name **#PBS -N galaxies1234**
-ncpus	Shared Memory	The number of processors to use for a shared memory job. **#PBS ncpus=4**
-r	All	Control whether or not jobs should automatically re-run from the start if the system crashes or is rebooted. Users with check points might not wish this to happen. **#PBS -r n** **#PBS -r y**
select	Clusters	Number of compute nodes to use. Usually combined with the mpiprocs directive **#PBS -l select=2**

-V	All	Make sure that the environment in which the job **runs** is the same as the environment in which it was **submitted.** **#PBS -V**
Walltime	All	The maximum time a job can run before being stopped. If not used a default of a few minutes is used. Use this flag to prevent jobs that go bad running for hundreds of hours. Format is HH:MM:SS **#PBS -l walltime=12:00:00**

B.2 Environment Variables in Batch Job Scripts

TORQUE-related environment variables in batch job scripts.

```
# Using PBS - Environment Variables:
# When a batch job starts execution, a number of environment variables are
# predefined, which include:
#
#      Variables defined on the execution host.
#      Variables exported from the submission host with
#                -v (selected variables) and -V (all variables).
#      Variables defined by PBS.
#
# The following reflect the environment where the user ran qsub:
# PBS_O_HOST    The host where you ran the qsub command.
# PBS_O_LOGNAME Your user ID where you ran qsub.
# PBS_O_HOME    Your home directory where you ran qsub.
# PBS_O_WORKDIR The working directory where you ran qsub.
#
# These reflect the environment where the job is executing:
# PBS_ENVIRONMENT       Set to PBS_BATCH to indicate the job is a batch job,
#         or to PBS_INTERACTIVE to indicate the job is a PBS interactive job.
# PBS_O_QUEUE   The original queue you submitted to.
# PBS_QUEUE     The queue the job is executing from.
# PBS_JOBID     The job's PBS identifier.
# PBS_JOBNAME   The job's name.
```

IMPORTANT!! All PBS directives MUST come before the first line of executable code in your script, otherwise they will be ignored.

When a batch job is started, a number of environment variables are created that can be used in the batch job script. A few of the most commonly used variables are described here.

Variable	**Description**
PBS_ENVIRONMENT	set to PBS_BATCH to indicate that the job is a batch job; otherwise, set to PBS_INTERACTIVE to indicate that the job is a PBS interactive job.
PBS_JOBID	the job identifier assigned to the job by the batch system. This is the same number you see when you do *qstat*.
PBS_JOBNAME	the job name supplied by the user

PBS_NODEFILE	the name of the file that contains the list of the nodes assigned to the job . Useful for Parallel jobs if you want to refer the node, count the node etc.
PBS_QUEUE	the name of the queue from which the job is executed
PBS_O_HOME	value of the HOME variable in the environment in which *qsub* was executed
PBS_O_LANG	value of the LANG variable in the environment in which *qsub* was executed
PBS_O_LOGNAME	value of the LOGNAME variable in the environment in which *qsub* was executed
PBS_O_PATH	value of the PATH variable in the environment in which *qsub* was executed
PBS_O_MAIL	value of the MAIL variable in the environment in which *qsub* was executed
PBS_O_SHELL	value of the SHELL variable in the environment in which *qsub* was executed
PBS_O_TZ	value of the TZ variable in the environment in which *qsub* was executed
PBS_O_HOST	the name of the host upon which the *qsub* command is running
PBS_O_QUEUE	the name of the original queue to which the job was submitted
PBS_O_WORKDIR	the absolute path of the current working directory of the *qsub* command. This is the most useful. Use it in every job-script. The first thing you do is, cd $PBS_O_WORKDIR after defining the resource list. This is because, pbs throw you to your $HOME directory.
PBS_O_NODENUM	node offset number
PBS_O_VNODENUM	vnode offset number
PBS_VERSION	Version Number of TORQUE, e.g., TORQUE-2.5.1
PBS_MOMPORT	active port for mom daemon
PBS_TASKNUM	number of tasks requested
PBS_JOBCOOKIE	job cookie
PBS_SERVER	Server Running TORQUE

Appendix C

Useful Linux Commands

C.1 Basic Linux Usage

All the clusters run some variant of the “” operating system. This means that, when you connect to one of them, you get a command line interface, which looks something like this:

```
@ln01[203] $
```

When you see this, we also say you are inside a “shell”. The shell will accept your commands, and execute them.

ls	Shows you a list of files in the current directory
cd	Change current working directory
rm	Remove file or directory
joe	Text editor
echo	Prints its parameters to the screen

Most commands will accept or even need parameters, which are placed after the command, separated by spaces. A simple example with the “echo” command:

```
$ echo This is a test
This is a test
```

Important here is the “$” sign in front of the first line. This should not be typed, but is a convention meaning “the rest of this line should be typed at your shell prompt”. The lines not starting with the “$” sign are usually the feedback or output from the command.

More commands will be used in the rest of this text, and will be explained then if necessary. If not, you can usually get more information about a command, say the item or command “ls”, by trying either of the following:

```
$ ls -help
$ man ls
$ info ls
```

(You can exit the last two “manuals” by using the “q” key.) For more exhaustive tutorials about Linux usage, please refer to the following sites: `http://www.linux.org/lessons/` `http://linux.about.com/od/nwb_guide/a/gdenwb06.htm`

C.2 How to get started with shell scripts

In a shell script, you will put the commands you would normally type at your shell prompt in the same order. This will enable you to execute all those commands at any time by only issuing one command: starting the script.

Scripts are basically non-compiled pieces of code: they are just text files. Since they don’t contain machine code, they are executed by what is called a “parser” or an “interpreter”. This is another program that understands the command in the script, and converts them to machine code. There are many kinds of scripting languages, including Perl and Python.

Another very common scripting language is shell scripting. In a shell script, you will put the commands you would normally type at your shell prompt in the same order. This will enable you to execute all those commands at any time by only issuing one command: starting the script.

Typically in the following examples they’ll have on each line the next command to be executed although it is possible to put multiple commands on one line. A very simple example of a script may be:

```
echo "Hello! This is my hostname:"
hostname
```

You can type both lines at your shell prompt, and the result will be the following:

```
$ echo "Hello!  This is my hostname:"
Hello! This is my hostname:
$ hostname

```

Suppose we want to call this script “foo”. You open a new file for editing, and name it “foo”, and edit it with your favourite editor

```
\ifgent
$ nano foo
\else
$ vi foo
\fi
```

or use the following commands:

```
$ echo "echo Hello!  This is my hostname:" > foo
$ echo hostname >> foo
```

The easiest ways to run a script is by starting the interpreter and pass the script as parameter. In case of our script, the interpreter may either be “sh” or “bash” (which are the same on the cluster). So start the script:

```
$ bash foo
Hello! This is my hostname:
```

Congratulations, you just created and started your first shell script!

A more advanced way of executing your shell scripts is by making them executable by their own, so without invoking the interpreter manually. The system can not automatically detect which interpreter you want, so you need to tell this in some way. The easiest way is by using the so called "shebang"-notation, explicitly created for this function: you put the following line on top of your shell script "#!/path/to/your/interpreter".

You can find this path with the "which" command. In our case, since we use bash as an interpreter, we get the following path:

```
$ which bash
/bin/bash
```

We edit our script and change it with this information:

```bash
#!/bin/bash
echo "Hello! This is my hostname:"
hostname
```

Note that the "shebang" must be the first line of your script! Now the operating system knows which program should be started to run the script.

Finally, we tell the operating system that this script is now executable. For this we change its file attributes:

```
$ chmod +x foo
```

Now you can start your script by simply executing it:

```
$ ./foo
Hello! This is my hostname:
```

The same technique can be used for all other scripting languages, like Perl and Python.

Most scripting languages understand that lines beginning with "#" are comments, and should be ignored. If the language you want to use does not ignore these lines, you may get strange results . . .

C.3 Linux Quick reference Guide

C.3.1 Archive Commands

tar	An archiving program designed to store and extract files from an archive known as a tar file.
tar -cvf foo.tar foo/	compress the contents of foo folder to foo.tar
tar -xvf foo.tar	extract foo.tar
tar -xvzf foo.tar.gz	extract gzipped foo.tar.gz

C.3.2 Basic Commands

ls	Shows you a list of files in the current directory
cd	Change the current directory
rm	Remove file or directory
mv	Move file or directory
echo	Display a line or text
pwd	Print working directory
mkdir	Create directories
rmdir	Remove directories

C.3.3 Editor

emacs	
nano	Nano's ANOther editor, an enhanced free Pico clone
vi	A programmers text editor

C.3.4 File Commands

cat	Read one or more files and print them to standard output
cmp	Compare two files byte by byte
cp	Copy files from a source to the same or different target(s)
du	Estimate disk usage of each file and recursively for directories
find	Search for files in directory hierarchy
grep	Print lines matching a pattern
ls	List directory contents
mv	Move file to different targets
rm	Remove files
sort	Sort lines of text files
wc	Print the number of new lines, words, and bytes in files

C.3.5 Help Commands

man	Displays the manual page of a command with its name, synopsis, description, author, copyright etc.

C.3.6 Network Commands

hostname	show or set the system's host name
ifconfig	Display the current configuration of the network interface. It is also useful to get the information about IP address, subnet mask, set remote IP address, netmask etc.
ping	send ICMP ECHO_REQUEST to network hosts, you will get back ICMP packet if the host responds. This command is useful when you are in a doubt whether your computer is connected or not.

C.3.7 Other Commands

logname	Print user's login name
quota	Display disk usage and limits
which	Returns the pathnames of the files that would be executed in the current environment
whoami	Displays the login name of the current effective user

C.3.8 Process Commands

&	In order to execute a command in the background, place an ampersand (&) on the command line at the end of the command. A user job number (placed in brackets) and a system process number are displayed. A system process number is the number by which the system identifies the job whereas a user job number is the number by which the user identifies the job
at	executes commands at a specified time
bg	Places a suspended job in the background
crontab	crontab is a file which contains the schedule of entries to run at specified times
fg	A process running in the background will be processed in the foreground
jobs	Lists the jobs being run in the background
kill	Cancels a job running in the background, it takes argument either the user job number or the system process number
ps	Reports a snapshot of the current processes
top	Display Linux tasks

C.3.9 User Account Commands

chmod	Modify properties for users
chown	Change file owner and group