Module 6

Resource requirements for computing and storage

Introduction

The previous modules have discussed what the research cloud is, what a virtual machine is and how you can use it for your research. You may now ask yourself: How many instances do I require? How many CPU cores per instance? How much storage can I get, and what types of storage are available?

In this module we will discuss factors which help you determine the amount of resources (computing and storage) you require. We will take a look at the different types of storage that are available to you, which will help you decide which type is most suitable for your research purposes.

Videos

The following videos go through most of the content in this module and offer a less in-depth description of the subject than the documentation does.

https://www.youtube.com/watch?v=brAmLtNbK-w

https://www.youtube.com/watch?v=0kbL2NuF5ow

https://www.youtube.com/watch?v=JKYZJWQZK5c

Conventions

The notation throughout the training documents can be interpreted as follows:

Words in *italics* are used for names and terminology, e.g. name of a software, or name of a computing concept. It may also just emphasise a word in the traditional way. Quotations are also written in italics and are put in between quotatioin marks.

Words in **bold** are used to highlight words which identify important concepts of a paragraph, to make it easier for users to skim through the text to find a paragraph which explains a certain idea, concept or technology.

Additional information which is optional to read is displayed in *info boxes* like this one.

Important information is displayed in boxes like this one.

Definition of terms are displayed in boxes of this style.

Possibly specific **prerequisites** for reading a particular section are contained in this type of box at the beginning of a section.

Storage

The most commonly used storage systems that you are probably familiar with are *filesystem storage* (commonly deployed as *Network Attached Storage*, short *NAS*) and *block level storage*.

- *Filesystem storage* is used for transferring **individual files** over a network to or from the storage. In a file system, files are organized in a hierarchical way, so that an individual file can be located by describing the *path* to that file. Network-attached storage (NAS) is a great way to share files securely among users on a network — it works well on a local network for a large number of files, but not so well over the Internet, and managing billions of files.
- *Block storage* appears to the system and the user as **attached drives**. A *Block* is a chunk of data which may contain several files. *Block storage* is the type of storage that is required for running things like databases, or as a storage required by applications (e.g. your data analysis software) which frequently access and write files.

Both of these storage types do not efficiently scale up. However, block storage may still be essential to your research application it is required for fast and efficient access from within applications; Many traditional applications only support file access in simple block storage systems. A more *scalable* system has been designed which is better suited for Big Data demands: The *Object Storage* which is based on the principle of *filesystem storage*.

Altogether, there are *three* different types of cloud storage available to you on the NeCTAR Research Cloud:

- **On-Instance storage** — this comes with your instance and is *"ephemeral"*: any files disappear when the VM is terminated (completely shut down, or "deleted"). You may treat this storage as *block storage*. This storage is limited in size. There are two on-instance storage disks: The *primary* and *secondary* disk.
- **Volume storage** — this is persistent, expandable storage that can be attached to a VM like one or more virtual hard disks. It is independent of a VM.
- **Object storage** — individual data files can be uploaded to the object store and accessed from the VM or from anywhere via the web (e.g. using a web browser). Files are *replicated* across several physical locations in order to achieve good data integrity, protect against data loss and ensure fast access.

The storage system is shared among instances. Not all storage is created equal, and the different types of storage differ according to performance, persistence and data safety.

This summary table provides a comparison of various data storage features.

Storage service	Saved in snapshot	DATA INTEGRITY(1)	ACCESS(2)	Backed up(3)
On-instance Primary Disk	Yes	OK	BLOCK	No
On-instance Secondary Disk	No	OK	BLOCK	No
Persistent Volumes	No	Good	BLOCK	No
Object Store	No	Best	HTTP	No

1. **Saved in snapshot**: You may save the state of your instance with a "snapshot". However a snapshot does not include all types of storage.
2. **Data Integrity**: how exposed your data is to hardware errors.
3. **Access**: there are two methods of access: either as a *BLOCK* level device (like attaching a hard drive to a computer) or *HTTP* where you use a client to *GET* or *POST* objects.
4. **Backed up**: is where there is a recoverable copy of a file available after a file is updated, deleted or damaged. Backup typically refers to maintaining a completely

separate backup system, where, for example, the backup system runs nightly and takes an incremental backup of any new data.

As you can see, none of the data is backed up, so you will have to do backups yourself, which is subject to Module 9. However, Object Storage has a high data integrity, which is a form of protection against data loss as well.

On-Instance storage

Each instance you start on the cloud comes with a certain amount of On-Instance storage, also called **ephemeral storage**, which appears as two separate hard disks: The *primary* and *secondary* disk. When you launch an instance (which you will do in Module 7), you will choose a *"flavor"*. The flavor determines the number of CPUs of your VM and also the sizes of the two disks on the On-Instance storage. The primary disk is saved with Snapshots which you take of the instance. The secondary disk is *not* backed up with Snapshots!

Example: In the flavor *"m1.small"*, you get a primary disk of 10GB. The secondary disk is 30GB—480GB depending on the flavor you select and your allocation. The first disk (10GB) is backed up when you take a Snapshot, whereas the second, larger disk is *not* included in the snapshot.

You can use the On-Instance storage as *block storage*, for example for your programs to read/write files to, or to operate your databases.

> On-instance storage is regarded as *ephemeral* — when you terminate your VM, or an unplanned re-start of the Hypervisor takes place in the NeCTAR Node, the data will be lost and cannot be recovered! You should treat it as **scratch space** and keep important data in either volumes or the object store.

> Not all flavors provide a secondary disk. At the time of writing, the following do:
>
> - Collection of images called *"standard-v2"*: Starting from flavor *m2.large* and bigger.
> - Collection of images called *"standard"*: Starting from flavor *m1.small* and bigger.

For more information on on-instance storage, please refer to the NeCTAR support

website

Volume storage

You may think of *Volume Storage* (also referred to as *block storage*) as a *section* of a large hard-drive which has been assigned for your use. Volume storage appears to the system and the user as *attached drives*. You will see it listed in your file explorer along with your other harddrives.

You have to *request an allocation* (see Module 5) to obtain access to Volume storage. Depending on your requirements and how you justify your needs, you may get access to very large Volumes.

Volume storage can live outside your instance. It appears as block storage which can be attached and then accessed from any of your instances. You can attach the Volume to your instance, read and write data, detach it, and make it available to another instance. Your data on a Volume is retained even when you terminate your instance. Volumes also offer a *Snapshot* feature which can be used to make convenient backups (this uses some of your overall Volume quota).

You can use the Volume storage as regular *block storage*, for example for your programs to read/write files to from within a program, or to operate your databases. You may also store all your important files on the Volumes — however don't forget to do regular backups (more about this in Module 9)!

It is also possible to boot an instance from a Volume. For more information, please refer to the NeCTAR support website.

In Module 7 we will learn how to attach and detach a Volume to an instance and access the data on it. Module 9 will discuss how to do Backups of Volumes.

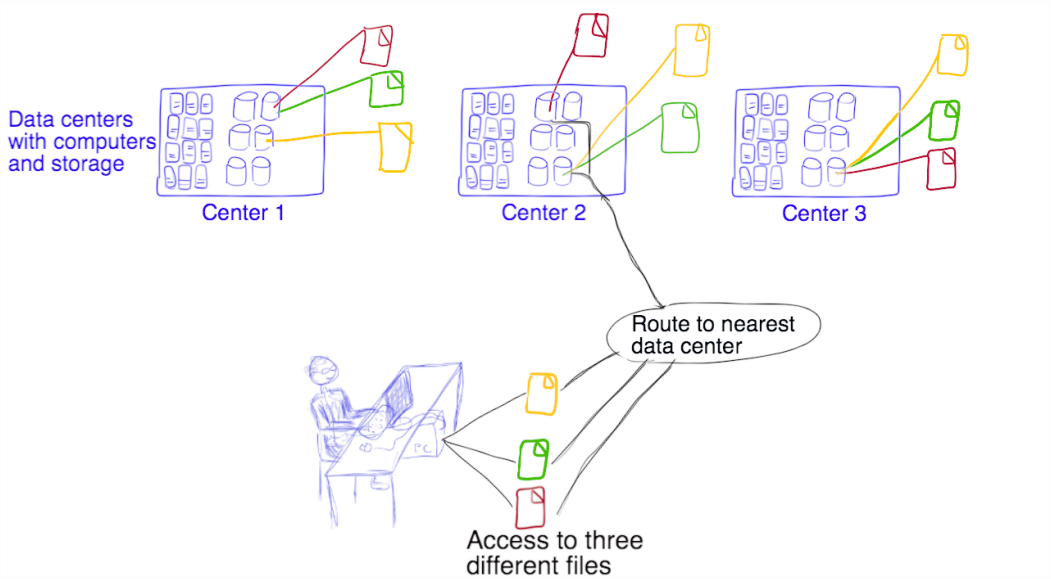
Volumes are *local services* and only instances living in the same Node where the Volume has been created can access it.

Object storage

Object storage targets the problem of data growth: As more and more data is generated, storage systems have to grow at the same pace. What if you try to expand a block-based storage system beyond multiple terabytes? You may hit limitations with the storage infrastructure, and managing a huge storage becomes very complex. A per-file object-based storage instead is easier to manage and can be distributed.

📖 An *object* in *Object Storage systems* is defined as *data* (typically a file) along with all its metadata, all bundled up as one *object*. Each object has a unique *ID* which is used by applications to retrieve the object. Unlike files in traditional hierarchical file systems, objects are stored in a *flat structure*. There is a pool of objects, and a given object is retrieved by presenting its object ID. Objects may be local or geographically distributed, but because they are in a flat structure (retrieved by their ID only), they are all retrieved exactly the same way.

Multiple copies of all object data is stored over a distributed system, while the storage still acts as one. Object Storage is highly fault tolerant through redundancy and distribution of data: If one or more nodes fail, the data can still be available — in most cases you as an end user will not even notice that a node was down. Object Storage therefore has a **great data integrity**! In most cases at least 3 copies of a file are stored across several nodes. You don't have to do anything to maintain the multiple copies, this is done automatically for you. Object Storage also scales up nicely, and it is easy to access your files from anywhere with a variety of devices.



The image above visualizes how 3 copies of your data are kept at different data centers, maybe even on different hard drives in the same data center. When you access the object store, you request a file by its ID, and it is automatically retrieved from the closest or more available data center.

❗ While Object Storage has high data integrity through geographical distribution of files, it does *not* have a "traditional" dedicated backup system — it merely provides a means to increase availability and integrity of your data by keeping multiple copies.

In the OpenStack Object Store, you may get access to large amounts of storage. Ultimately, this is limited by the storage cluster size of about 40TB. You may upload as many files as you like, as long as your allocation is not exceeded. The size of the files is not limited, however due to limits in the *http* protocol it is not recommended to upload files larger than 1GB.

The NeCTAR *trial account* has a default of 10GB object storage allocation. When you lodge an application for resources, you have to specify the amount of object storage you require, otherwise you won't receive any allocation for object storage.

Suitable use of object store

Object Storage is not a traditional file-system or real-time data storage system. It's designed for mostly static data that can be retrieved, leveraged, and then updated if necessary. It is independent of a particular instance and can be updated and used without having any instance running. It is designed to be redundant and scalable.

Think about that dataset comprised of 2GB files that you read in and analyse many times, but in general it doesn't change. Or the images you want to use on the cloud. Those are a couple examples of what's perfect for Object Storage.

In general, the object store is great for data you write once and read many times, but not suitable for applications like databases. It's the safest place to put your data on the NeCTAR Research Cloud as multiple redundant copies of your data are made, and it has great performance. You can access the object store from anywhere via the Internet, and data from Object Storage can be transferred to and from your instance with a variety of tools.

Swift

Swift is the component that provides object storage for the *OpenStack* framework which is used in the NeCTAR Research Cloud. With your credentials (which you can download from the Dashboard) you can request *Swift* to reserve and create storage (called *containers* or *buckets*) and upload/download files.

The object store can be accessed via the Dashboard, which provides a UI for the *swift* component. There are also other graphical *clients* which can be used to access the store, as we shall see in Module 7. It is also possible to use a *command line client* to access the object store, which is subject to Module 10.

> **Security Warning:** Swift does NOT provide encryption of the data it stores. When you upload/download data to/from the object store, this will happen without encryption. If you have sensitive data that requires encryption you must encrypt the data files *before* upload.

In Module 7 we will learn a few ways how to connect to your object storage and copy objects to and from our object store. Module 8 will introduce a few tools that can be used for data encryption, and Module 10 will discuss the more advanced use of command line tools to access the object store.

For more information, please refer to the NeCTAR support website.

Volume (block) vs Object storage

In summary, the main differences between *volume* and *object* store are:

File Access
Volume or *Block* storage can be seen as what we know as a partition of a harddrive (or a whole hard drive), containing a lot of files in one continuous block. Files are accessed with traditional file access methods. In the *Object Storage*, files may be spread over several hard-drives. An *object* is accessed using a *http* based interface.

Distribution
Object Storage is a per-file based storage system which stores each file at several locations — you won't know where exactly the individual files are stored, they may even not all be on the same harddrive. In contrast, *Volume Storage* is one block of storage which contains *several* files, located at *one* location.

Recommendations

In summary, here are a few recommendations on how you may use the different storage types:

- Use On-Instance Storage as "scratch space" and/or to keep copies/clones of data (e.g. websites) which are mainly stored elsewhere. Because On-instance storage is *ephemeral*, you will lose access to your data when an unplanned restart of the hypervisor takes place, or after you terminate your instance. Therefore, the main motivation to use this storage should be to have it readily available on the instance without requiring extra Volumes, or using up your Volume storage quota for it.

- Use Volumes or the Object Store for all data which should survive an instance termination (or an unplanned restart of hypervisor). You may need *Volumes* for data access from your programs which require block storage, for example to read files from a disk and write results onto it. If you have control over the source code of your program, you may instead consider to use *object storage* instead of Volumes; APIs for many programming languages are available (more information in Module 10 and the OpenStack SDKs website).
- Use the Object Store for individual files which you want to have easy access to from many locations, or which you would like to easily share with collaborators.
- For very large data demands, you may also consider using Volumes.

> Especially with Volume storage, but also Object Storage (albeit with less urgency), you should **back up your data** at regular intervals, for example at important stages of its life cycle. If you use On-Instance storage for data which is *not* a copy of a dataset mainly kept elsewhere, backing up is extremely important. Module 9 will guide you through a few options to back up and restore your data.

How many cores do I need?

Virtual machine (VM) instances in the Research Cloud are available in “Standard” sizes:

- Small (1 core), Medium (2 cores), Large (4 cores), Extra-large or XL (8 cores), XXL (16 cores)
- Per core, a VM gets 4 GB of memory (RAM) and 30 GB of On-Instance Storage (On-Instance secondary drive).
- Example: an XL VM instance has 8 processing cores, 32 GB RAM and 240 GB of On-Instance Storage.

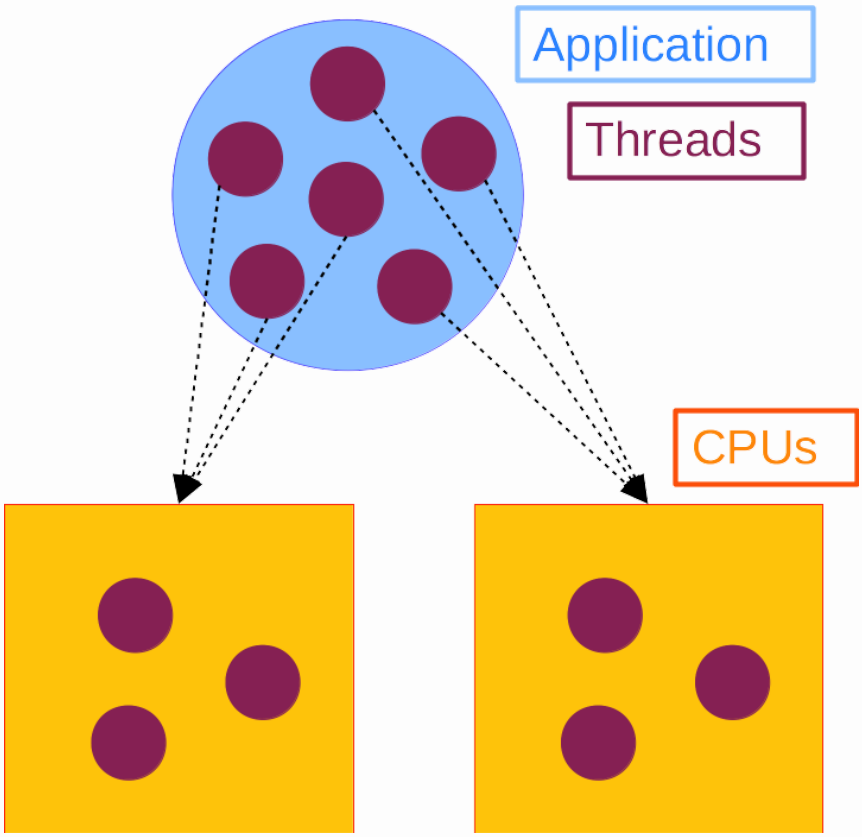
You may be unsure how many CPU cores you need for your instance: Which flavor to choose?

You will have to take a closer look at your application to find out how much you can benefit from several cores.

Some applications are **single-threaded**, which means they only use one core anyway. In this case, you will not benefit from launching an instance with several cores.

However many applications are **multi-threaded**, and support a certain number of *threads* (you may regard a thread as a working process that can run on its own core). The documentation of your application should reveal more information about that.

Example:
Say your application supports 6 threads. You may then select up to the same number of cores — with 6 cores, the application will run at maximum performance. If you choose more cores, you will not experience any performance gains. It is ok to select a lower number of cores: say, you choose 2 cores while your application supports 6 threads. In this case, the workload of the 6 threads will be distributed across the 2 cores, and your application will run *one third of its maximum performance* (2 cores / 6 threads).



Use of OpenMP

OpenMP is an API that can be used for multi-thread handling on *shared memory systems*.

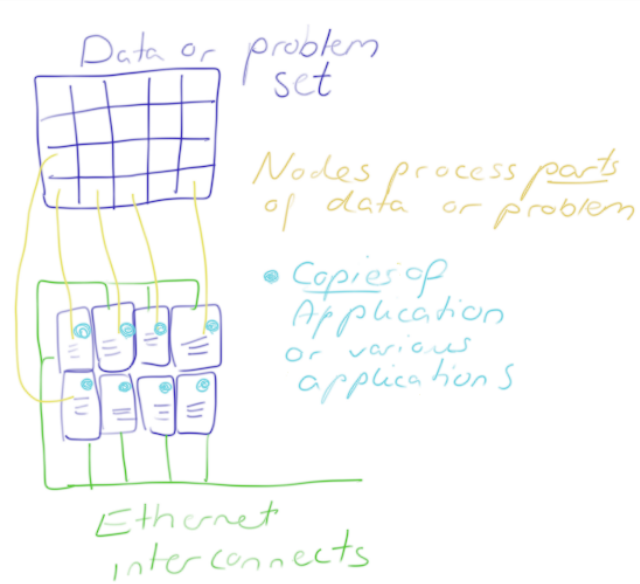
> A computer (or a virtual machine) with several CPUs is a **shared memory system** when all CPUs access the *same* memory.

OpenMP uses a set of compiler directives (within the C code) that generate a multi-threaded version of your code at compile time.

You can think of OpenMP as a slightly more automated way to create multi-threaded programs than when low-level thread libraries as *pthread* were used. As such, also OpenMP programs gain maximum performance when it is given the same amount of cores as it uses threads (the cores may also be distributed over *several CPUs* of the same machine).

Can I benefit from several VMs?

In Module 4, we have discussed *horizontal scalability*. Horizontal scalability entails that you may run several instances of your application on separate virtual machines, all processing different parts of your data, or processing the same data in a different way (e.g. with different parameters).

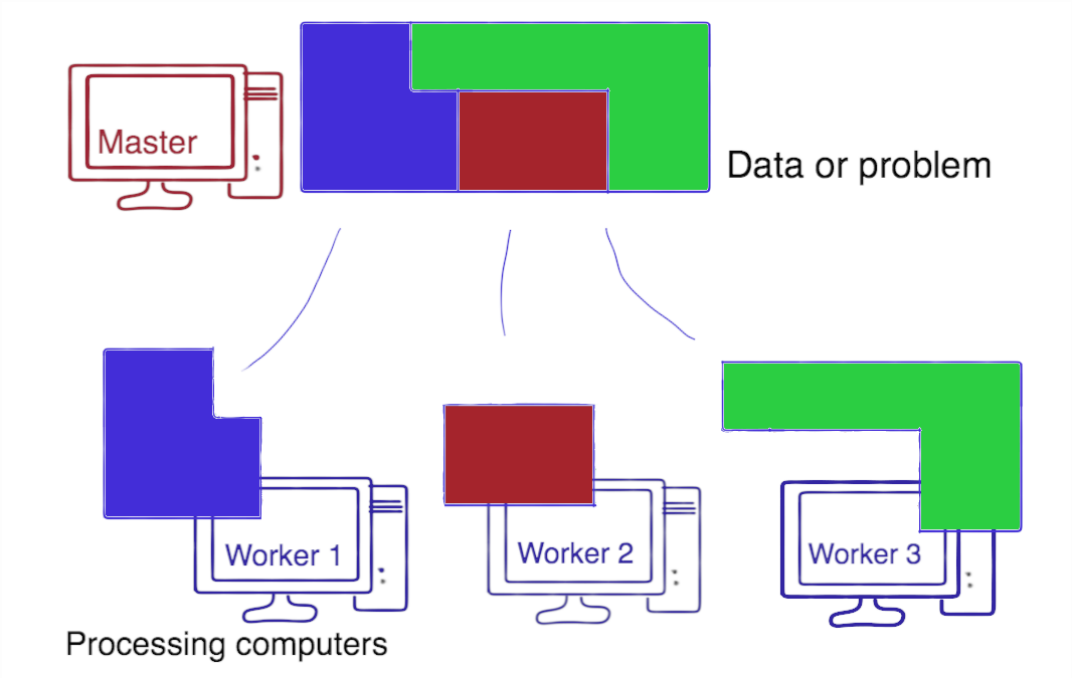


The Cloud is suitable for such a scenario. Distributing your problem over multiple instances is certainly an advanced topic as it requires knowledge about parallel programming. However this section will give a brief overview of the possibilities with the Cloud and the key factors to look out for in your application. For more details, we refer to related Literature.

Data or problem partitioning

If your data or your problem set can be split into separate units for processing, you may dispatch the processing to several instances. For example, if you can split the data set into several chunks, then you could send one chunk to one instance each, where each instance runs the same program to analyse a chunk in the same fashion; when all instances are finished, results are summarized. Or if your problem can be split into several operations (which may or may not use the same data), for example different operations to perform on a dataset, each operational task can then be dispatched to one instance each.

The instance which splits the problem set and dispatches the tasks is called the "Master", while the instances to which tasks are dispatched are called "*Workers*". A software framework like *MapReduce* can be used to split and dispatch your problem set.



MapReduce is a framework for processing parallelizable problems across huge datasets using a large number of computers (nodes). MapReduce processes and generates large data sets with a parallel, distributed algorithm on a *cluster* (a collection of compute nodes). — Source, and more details: Wikipedia

Detailed discussion of MapReduce frameworks or similar programming models is not within the scope of this course. Instead, please refer to related Literature.

For use with OpenStack, you may use *Apache Hadoop* through the Sahara Project. *Apache Hadoop* is an industry standard and widely adopted MapReduce implementation. The aim of the Sahara project is to enable users to easily provision and manage Hadoop clusters on OpenStack. Using Sahara is not part of this course, we refer to the on-line documentation at this place instead.

Note: The use of *Sahara* in the NeCTAR Cloud is ***currently not available yet***, but it will be coming soon.

When using *Sahara*, you don't launch the instances manually on the Dashboard as we will do in Module 7. You specify *templates* for the virtual machines, and Sahara will launch the instances for you and set them up so they

can be used with your application.

Whether you benefit from this model, depends on the problem that you are trying to solve. There are limitations and trade-offs:

- Into how many pieces can the data or problem be split? This limits the number of instances you can use.
- You should also consider the communication demands for distributing the problem to the workers: if you chop the problem or data into very small pieces, does it still pay off to distribute it to many compute nodes, considering communication demands for the distribution?
- The time to split the dataset is also a factor: You will *not* benefit from splitting the problem if the process of splitting, distributing to workers and then summarizing the results takes longer than solving the problem itself on only one instance.
- A limitation will also be the number of virtual machine instances that you have been granted in your resource allocation.

Auto-scaling with OpenStack Heat

Sometimes, you just need to *replicate* your resources at times when heavy workloads are experienced. For example, if you are deploying a webserver, and at a certain time lots of users access your website and create a lot of traffic and workload on your server, you can *scale up* and create a second (or third, etc.) webserver with the same configuration. The workload can then be distributed across the identical servers, and users don't experience delays any more.

The OpenStack framework (used by NeCTAR to run the Cloud) provides a means to *orchestrate* your resources: The **Heat** project. You specify your resources in a human-readable text file, which is called a "template". *Heat* then manages the entire lifecycle of your infrastructure and applications, from creating the resources until deletion when your application has finished. The templates provide a lot of options to specify and configure your resources. It therefore has a bit of a learning curve, but it is worth the effort: After you have defined the template, you may launch your services within a matter of minutes, without having to go through the whole setup process again. You can repeatedly use the template to start up your services — with the template, you have *automated* the process of setting up the resources you require for a particular research application.

A *Heat* template can include the instances you want to launch (and which image they

should be launched from), volumes to create, security groups to use, and more. It may also include applications to be installed on the servers, e.g. the webserver application.

Most importantly for this section, Heat also provides an **autoscaling service**: You may include a “scaling group” in the template. This group is controlled by a set of *“alarms”* you define. For example, an alarm may be triggered if the CPU-workload exceeds a certain percentage. In which case you specify to launch an additional server. Heat takes care of this type of autoscaling for you, according to what you specify in the template.

A detailed discussion of *Heat* is not part of this tutorial, instead please refer to the OpenStack Heat wiki and the NeCTAR support website which includes a lot of useful information and links.

Use of Message Passing Interface (MPI)

The Message Passing Interface (MPI) is a technique which is widely spread in parallel programming. MPI is a standard defining syntax and semantics of library routines that can be used to implement parallel programming in C and other languages. There are several implementations of MPI such as Open MPI, MPICH2 and LAM/MPI.

With MPI, you can spread the processing of your application over several instances. Communication between nodes is handled in different ways. *Open MPI* facilitates secure communication between nodes via *ssh*.

The MPI interface allows the programmer to manage allocation, communication, and synchronization of a set of processes that are distributed over multiple *Nodes*. Such *Nodes* may be:

- the individual **cores** in one CPU, or
- several **CPUs** in the *same* machine, or even
- **separate machines** which are connected in a network.

If your application uses MPI to support parallelism, you may benefit from several instances. However, your better choice may be to go for a HPC solution (see discussion in Module 4). MPI performs poorly on common OpenStack networks because communication between the nodes is slow.

You can run MPI programs across OpenStack instances as long as your instances are allowed to communicate with each other over their fixed IPs, and your instances’ firewall settings allow traffic across the ports that your MPI implementation is using. There is however one exception: If your MPI application uses *broadast* or *multicast* in

any form, it can not run on OpenStack. This is because some clouds do not support broadcast or multicast in any form (at the time of writing, neither Amazon EC2 nor OpenStack do).

A detailed discussion of MPI is outside the scope of this course—setting up an MPI program on the cloud requires previous experience and in-depth knowledge of MPI and the application to run. Please refer to related Literature.

Summary: Using several instances

In summary, you may use several instances, but it depends on the application whether this is beneficial.

- You may **split the problem or data set** and distribute it, for example using *MapReduce* frameworks.
- You may configure **auto-scaling** for your resources with *Heat* in order to adapt to current workload demands.
- You may be able to run your **MPI** application on the NeCTAR Research Cloud.

Distributing your application over several instances is an advanced topic which was only described on a high level in this course. For more details, please refer to the related Literature.

Summary

And… we're done! Ready for more?

This module has discussed factors which help determine the amount of resources (computing and storage) which may be required. Different types of storage that are available have been introduced, which should help you decide which type is most suitable for your research purposes. We have also discussed key factors which help determine how many cores you should choose for your instance. You may have found that it is worth doing a bit more reading to learn more about using several instances (e.g. using MapReduce style frameworks).

You now have all the knowledge required to get started with using the NeCTAR research cloud: You know what a virtual machine is and what is required to keep it secure; you know the types of storage available to you, and you have at least a rough idea about how many cores, instances and which types of storage you will require.

Before you continue, you may take a brief look at the related literature for this Module.

Continue with Module 7 to start getting some hands-on experience with launching and

using virtual machines.

Literature

1. Parallel Programming
    - Pattern: Patterns for Parallel Programming, Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill, Addison-Wesley, 2005
    - CDK5: Distributed Systems, Concepts and Design, George Coulouris, Jean Dollimore, and Tim Kindberg, Gordon Blair, Addison-Wesley, 2011
    - Enterprise Integration Patterns
    - A good tutorial on parallel computing: Introduction to Parallel Computing by Blaise Barney, Lawrence Livermore National Laboratory
    - Another good on-line tutorial: Parallel programming in C by Paul Gribble at Western University in London, Ontario, Canada
    - Free eBook on parallel programming: Programming on Parallel Machines by Norm Matloff, University of California
2. Grid and Cloud
    - Grid-D: Grid Computing for Developers, Vladimir Silva, Charles River Media, Inc., 2006
    - Grid-2: The Grid 2, 2nd Ed, Ian Foster and Carl Kesselman, Morgan Kaufmann, 2003
    - Cloud: Cloud Application Architectures, George Reese, O'Reilly, 2009
3. OpenMP
    - OpenMP: Using OpenMP, Barbara Chapman et.al., The MIT Press, 2007
    - The OpenMP API Specification for Parallel Programming
4. MPI
    - MPI: Parallel Programming with MPI, Peter S. Pacheco, Morgan kaufmann, 1997
    - The Message Passing Interface (MPI) Standard
    - mpiJava Home Page

5. MapReduce
    - MapReduce: Data-Intensive Text Processing with MapReduce, Jimmy Lin and Chris Dyer, Morgan and Claypool, 2010
    - Hadoop: Hadoop: The Definitive Guide, MapReduce for the Cloud, Tom White, O'Reilly, 2009
    - Apache Hadoop
    - Sahara Project

Continue with Module 7.