Master in Computer Vision Barcelona

DCC-CVC Cluster

User's Guide

Authors

David Aldavert — aldavert@cvc.uab.cat
Jorge Ramirez — jorge.ramirez@uab.cat
Maria Vanrell — maria@cvc.uab.cat

01 December 2013

Table of Contents

Acronyms

CVC Computer Vision Center

UAB Universitat Autònoma de Barcelona

DCC Dept. de Ciències de la Computació

GE Grid Engine

OGE Oracle Grid Engine

SGE Sun Grid Engine

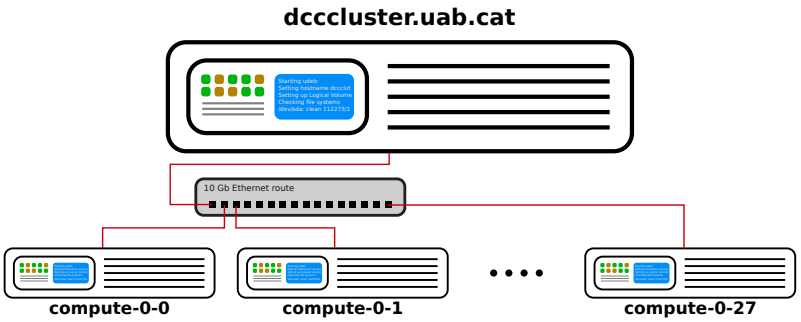
MPI Message Passing Interface

ORTE Open Run-Time Environment

1

The cluster

The DCC-CVC cluster is the computer cluster of the Dept. de Ciències de la Computació (DCC) and Computer Vision Center (CVC) at the Universitat Autònoma de Barcelona (UAB). It is a Supermicro cluster with 28 nodes under a head node and all nodes are interconnected by a Giga-Ethernet network.



All cluster nodes are running the Rocks Linux distribution* and the cluster is managed through the Oracle Grid Engine (OGE)†.

The students of the master in Computer Vision can work on a portion of this cluster (6 nodes of the cluster are reserved for students). Each student is allowed to launch up to 30 running jobs (depending on the workload of the cluster) and as many pending jobs as the cluster allows (i.e. jobs waiting for the needed resources to be available).

1.1 How to access the cluster?

You will receive by e-mail your user account access, otherwise, to get it you can contact to Jorge Ramirez at jorge.ramirez@uab.cat or at room QC/1078 at the Engineering School building. You can also reach him at (+34) 93 581 30 55.

To login the cluster you have to connect to `dccluster.uab.cat` through a Secure Shell (SSH) client. For Windows users, we suggest clients like the MobaXterm client‡ while Linux and Mac users, the client provided by the operative system.

*This distribution is based on the CentOS Linux distribution

†Previously known as Sun Grid Engine (SGE).

‡You can download the MobaXterm client from: `http://mobaxterm.mobatek.net`

The user name assigned to the master students will correspond to their NIU number. Therefore, an user with the NIU 1234567 will connect to the cluster as follows:

```
[user@localhost ~]$ ssh 1234567@dccluster.uab.cat
Password: <your password>
[1234567@dccclus ~]$
```

Users can only access to the header node (dccluster.uab.cat), while the working nodes are only accessible through the queue system provided by the Grid Engine (GE). The header node must be used only to launch experiments or punctual tasks like compiling the code. Using the header node to perform heavy tasks may halt the cluster queue system and hinder the tasks launched by other users.

1.2 How to run your jobs on the cluster?

As a first approach, you can find a short example script to perform a task queuing on the cluster.

Just using the **qsub** command you run a script on the cluster, for example:

```
[1234567@dccclus ~]$ qsub ~/example/launch.qsub
```

This job is running the Matlab program **primeaddition.m**. The output of the job is left in your home directory.

```
[1234567@dccclus ~]$ ls -la pri*
-rw-r--r-- 1 1234567 1234567  0 Nov   5 11:45 primeaddition.e161924
-rw-r--r-- 1 1234567 1234567  0 Nov   5 11:45 primeaddition.o161924
```

The **launch.qsub** script contains information needed by the GE to launch a **Matlab** program in the cluster:

Listing 1.1: Script launch.qsub

```
# SGE submission options
#$ -q fast.master.q #Queue selection
#$ -o $HOME #Output directory
#$ -e $HOME #Error directory output
#$ -cwd #Change to current working directory
#$ -N primeaddition #Name of the job
#$ -l mem_token=6G,mem_free=6G #Relevant information for process queuing
/opt/matlab/bin/matlab -singleCompThread -nodesktop -nojvm -nosplash -nodisplay <
~/example/primeaddition.m
```

The task will be initially waiting in the selected queue (i.e. **fast.master.q**) until enough resources are available, in this example, until a node with 6Gb of free memory is available. Then, the GE will reserve resources needed by the task and it will launch the task. Once the task finishes, the GE will deallocate the resources and remove the task from the queue. This process is completely transparent to the user.

A parameter that the user must specify is the queue where the job is going to run. The students of the master can launch their tasks in three different queues. The main difference between these queues is the maximum time which a task can be running. If a task exceeds this time limit, then the queue manager terminates that task. In table 1.1, you can find the time limits of the different queues available to the students.

Table 1.1: Time limits of the student queues.

Queue name	Time limit
`fast.master.q`	For jobs under 2 hours and 30 minutes time limit.
`medium.master.q`	For jobs under 2 days time limit.
`long.master.q`	For jobs under 7 days time limit.

In the next chapters of this manual, first we are going to show which are the main commands to get information about the cluster, to execute programs and check the status of the running programs. Then, we are going to show how you can maximize the use of the cluster by taking advantage of algorithm parallelization both in `Python` and in `Matlab`.

1.3 Cluster specifications

The cluster is formed by 28 computer nodes with the following specifications each one:

- 2×8 x86-64 processor cores with Hyper-Threading enabled (i.e. 32 virtual cores per node).
- 64Gb of memory
- 1Tb disk drive
- 1Gb Ethernet connection

It brings a total of 896 virtual cores with an assignment of 2Gb of memory for each core. The head node has the following specifications:

- 2×8 x86-64 processor cores with Hyper-Threading enabled
- 64Gb of memory
- 2×300Gb hard disk drives for system in RAID 1 [§] .
- 6×256Gb SSD disk drives in RAID 6 [¶] .
- 11×3Tb hard disk drives in RAID 6 [¶].
- 10Gb Etherent network connection

The network has the following specifications:

- 48 port switch of Gigabit Ethernet with 10GB uplink.
- 24 port switch of Gigabig Ethernet with 10Gb uplink.

[§]**RAID 1** comprises mirroring: Data is written identically to the two drives. This configuration reduces the reading latencies.

[¶]**RAID 6** comprises block-level striping with double distributed parity.

2

Usage

The cluster uses the Grid Engine (GE) to manage the jobs submitted to the cluster. Jobs are managed through a queue system, which holds the jobs until the cluster has enough resource to run the job, launches the jobs to the cluster nodes and manages the resources when the jobs are finished. This operations are completely transparent to the user which only has to set the resources that the job needs to run properly.

There are several ways to use the cluster. For instance, it can be used as a group of machines which work independently to run your tasks. From this perspective, the cluster of the department can be seen as 896 independent Linux machines with 2Gb of memory but with the advantage that tasks are managed automatically. Besides this basic use, the cluster can be also used to run task in parallel using several processes on the same machine (up to 32 processes using multi-threading or multi-processing) or use all working nodes of the cluster on a single task (i.e. having up to 896 parallel processes and 1.75Tb of memory available for a single task). In this chapter, we are going to show how to launch task in the cluster, how to get information about the status of our jobs and to get information about the cluster.

2.1 Launching tasks

The cluster can only be accessed by submitting batch jobs to the GE. Other clusters may allow to use interactive sessions to the GE by using commands like `qsh` *, `qlogin`* or `qrsh`*. The command to submit batch jobs is `qsub` and we can use two different ways to specify the parameters of the task: using a configuration file or passing the options directly as parameters of the `qsub` command. In this guide, we are going to use the later, since using the command line is going to be more convenient when we want to launch multiple jobs at the same time.

Original image **Transformed image**

Figure 2.1: Image pre-processing example.

2.1.1 Basic commands

As an initial task, we want to use the cluster to pre-process the images of a GRAZ02[†] database. We are going to use the ImageMagick [‡] program to maximize the contrast and sharpen the database images (see figure 2.1 for an image pre-processing example).

We can launch the task in the cluster by using **qsub** as follows:

```
[1234567@dccclus ~]$ qsub -b y -cwd -V -q fast.master.q -l mem_token=5M,\
mem_free=5M convert bike/bike_057.bmp -limit thread 1 -interpolate \
spline -resize 1024x768 -sharpen 0x1 -normalize -resize 640x480 bike/bike_057.png
Your job 171029 ("convert") has been submitted
```

First, we set the information of the task for the GE and later we set the command and the parameters run by the job. The parameter **-b y** is used to specify that the task is a binary otherwise, the cluster is configured to expect a ***Bash*** *script*. The parameter **-cwd** sets the working folder to the current folder, otherwise the GE will set the working folder at your home folder. The parameter **-V** export the environment variables to the context of the job (i.e. it exports the PATH, the route to the libraries, ...). Without this parameter, we should put the whole route to the binary **/usr/bin/convert** because the environment variable PATH would be empty for the job. The parameter **-q fast.master.q** sets the queue which the job is assigned. Besides the queue, you can also specify the node where you want to launch your job. For instance using **-q fast.master.q@compute-0-14**, will wait until enough resources are are available at the node **compute-0-14**. The last parameter **-l mem_token=5M,mem_free=5M** specifies that the task requires at least 5Mb of memory.

In the previous call, the parameter **-l mem_token=<memory>,mem_free=<memory>** is mandatory. This parameter allows to manage the memory assigned to the tasks and it is used to avoid to allocate more memory than available. This will cause memory swapping which would drastically reduce the performance of the cluster. To prevent this to happen, the cluster is configured to assign by default more memory than available, so that, an uninitialized task will indefinitely remain waiting in the queue. The variable **mem_token** sets the maximum memory necessary to run the job while **mem_free** sets the minimum amount of

[*] **qsh** submits an interactive X-windows session, **qlogin** submit an interactive login session and **qrsh** submits an interactive **rsh** session.

[†] http://www.emt.tugraz.at/~pinz/data/GRAZ_02/

[‡] More information about the ImageMagick (http://www.imagemagick.org) parameters used in the example can be found at http://www.imagemagick.org/script/command-line-options.php

memory needed to begin the job. Both variables are usually set using the same amount of memory.

Jobs must allocate as many memory as needed with parameters **mem_token** and **mem_free**, otherwise GE can allocate jobs which cause memory swapping, decreasing the overall performance of the cluster. In these cases, the system administrator can remove these under-allocated jobs and warn the user.
It is also important not to excessively overshoot while setting the memory needed by the job. In average each virtual core has up to 2Gb of memory available but properly allocating memory can allow maximize the use of virtual cores. Although overshooting is not as dangerous as underestimating the amount of memory, it also prevents to the cluster attain its maximum potential.

The GRAZ02 database has 1476 images and manually launching a test for each image is a tedious operation. A way to launch all the experiments at once is to generate an script which generates the cluster calls for all images:

Listing 2.1: First **Python** *launcher: launch01.py*

```
1  import os, glob
2
3  if __name__ == '__main__':
4      # Cluster parameters.
5      queue_name = 'fast.master.q'
6      memory = '5M'
7      # Route to the images.
8      images  = glob.glob('bike/*.bmp')
9      images += glob.glob('cars/*.bmp')
10     images += glob.glob('person/*.bmp')
11     images += glob.glob('none/*.bmp')
12     cmd_mask = "convert %s -limit thread 1 -interpolate spline -resize 1024x768 -sharpen
        0x1 -normalize -resize 640x480 %s"
13     print 'Launching %d tasks...' % (len(images))
14     for input_filename in images:
15         output_filename = input_filename[:-3] + 'png'
16         current_cmd = cmd_mask % (input_filename, output_filename)
17         cmd = 'qsub -b y -cwd -V -q %s -l mem_token=%s,mem_free=%s %s' % (queue_name,
       memory, memory, current_cmd)
18         print cmd
19         print os.popen(cmd).read()
```

Then, we can simply call the script to generate the 1476 jobs:

```
[1234567@dccclus ~]$ python launch01.py
Launching 1476 tasks...
qsub -b y -cwd -V -q fast.master.q -l mem_token=5M,mem_free=5M convert bike/
    bike_271.bmp -limit thread 1 -interpolate spline -resize 1024x768 -sharpen 0
    x1 -normalize -resize 640x480 bike/bike_271.png
Your job 172048 ("convert") has been submitted

qsub -b y -cwd -V -q fast.master.q -l mem_token=5M,mem_free=5M convert bike/
    bike_011.bmp -limit thread 1 -interpolate spline -resize 1024x768 -sharpen 0
    x1 -normalize -resize 640x480 bike/bike_011.png
```

```
Your job 172049 ("convert") has been submitted

qsub -b y -cwd -V -q fast.master.q -l mem_token=5M,mem_free=5M convert bike/
    bike_012.bmp -limit thread 1 -interpolate spline -resize 1024x768 -sharpen 0
    x1 -normalize -resize 640x480 bike/bike_012.png
Your job 172050 ("convert") has been submitted

qsub -b y -cwd -V -q fast.master.q -l mem_token=5M,mem_free=5M convert bike/
    bike_140.bmp -limit thread 1 -interpolate spline -resize 1024x768 -sharpen 0
    x1 -normalize -resize 640x480 bike/bike_140.png
Your job 172051 ("convert") has been submitted

...
...
...
```

The only drawback of converting the database images using this procedure is that we generate many tasks which only require a little bit of computational power. Launching too many small tasks to the cluster, can exceed the maximum number of slots available to a single user or the queue, so that, only a portion of the task will assigned to the queues while the remaining will be simply deleted. A better approach is to reduce the number of jobs by creating shell scripts which group together several commands.

2.1.2 Shell scripts

A way to join several tasks in the same job is to use shell scripts. The GE can use several shell scripts languages but when not specified it uses the default shell script language. Currently, the cluster uses **Bash** language as the default shell language. To select another language, we have to add the parameter **-S <route_to_shell>**. For instance, to apply more complex operation to the images using ImageMagick there are the "*Fred's ImageMagick Scripts* [§]". This is a collection of **Bash** scripts which combine several ImageMagick operations to achieve more complex operators. For instance, we to automatically adjust the white balance we can use the **autowhite** script:

```
[1234567@dccclus ~]$ qsub -cwd -V -q fast.master.q -l mem_token=5M,\
mem_free=5M autowhite bike/bike_057.bmp bike/bike_057.png
Your job 176109 ("autowhite") has been submitted
```

The main difference with the previous calls is that the parameter **-b y** have been removed. To apply this command to all images of the GRAZ02 database, we can use a **Python** script which group several calls together instead of launching a job for each image. Besides, we are going to use the **qsub** parameter **-t n[-m[:s]]** which allows to submit an *array job*, i.e. an array of identical tasks being differentiated only by an index number. The variable n and m define the range of the indexes assigned to each job while the variable s represents the step. The value of this variables are passed to the job through environment variables: the index of the task is in the environment variable **SGE_TASK_ID** while n, m and s can be found in the environment variables **SGE_TASK_FIRST**, **SGE_TASK_LAST** and **SGE_TASK_STEPSIZE**, respectively. Then, a possible implementation of the **Python** script used to process all database images in different chunks can be as follows:

Listing 2.2: Processing script: preProcessGraz01.py

[§] http://www.fmwconcepts.com/imagemagick/index.php

```python
import os, glob, sys

if __name__ == '__main__':
    # Get the environment variables ..............................
    task_id = int(os.environ['SGE_TASK_ID'])
    task_first = int(os.environ['SGE_TASK_FIRST'])
    task_end = int(os.environ['SGE_TASK_LAST']) + 1
    task_step = int(os.environ['SGE_TASK_STEPSIZE'])

    # Get the task ID ............................................
    tasks = range(task_first, task_end, task_step)
    number_of_jobs = len(tasks)
    job_id = tasks.index(task_id)

    # Create a list with the database image routes ...............
    images  = glob.glob('bike/*.bmp')
    images += glob.glob('cars/*.bmp')
    images += glob.glob('person/*.bmp')
    images += glob.glob('none/*.bmp')

    cmd_mask  = "convert %s -limit thread 1 -interpolate spline "
    cmd_mask += "-resize 1024x768 -sharpen 0x1 -normalize -resize 640x480 %s"
    for index in range(job_id, len(images), number_of_jobs):
        input_filename = images[index]
        output_filename = input_filename[:-3] + 'png'
        cmd = cmd_mask % (input_filename, output_filename)
        print cmd
        print os.popen(cmd).read()
        sys.stdout.flush()
```

Now, we can add the **-t 1-10** in the **qsub** call to indicate that we want an array of 10 jobs:

```
[1234567@dccclus ~]$ qsub -cwd -V -S /usr/bin/python -q fast.master.q -t 1-10 \
-l mem_token=5M,mem_free=5M preProcessGraz01.py
Your job-array 176110.1-10:1 ("preProcessGraz01.py") has been submitted
```

Note, that we also added the command **-S /usr/bin/python** to indicate to the GE that the script is written in **Python** instead of **Bash**. This will generate 10 jobs with the same ID but having a different task index:

```
[1234567@dccclus graz02]$ ls -l preProcessGraz01.py*
-rw-rw-r-- 1 1234567 1234567  1151 28 nov 12:01 preProcessGraz01.py
-rw-r--r-- 1 1234567 1234567     0 28 nov 11:57 preProcessGraz01.py.e176110.1
-rw-r--r-- 1 1234567 1234567     0 28 nov 11:57 preProcessGraz01.py.e176110.10
-rw-r--r-- 1 1234567 1234567     0 28 nov 11:57 preProcessGraz01.py.e176110.2
-rw-r--r-- 1 1234567 1234567     0 28 nov 11:57 preProcessGraz01.py.e176110.3
-rw-r--r-- 1 1234567 1234567     0 28 nov 11:57 preProcessGraz01.py.e176110.4
-rw-r--r-- 1 1234567 1234567     0 28 nov 11:57 preProcessGraz01.py.e176110.5
-rw-r--r-- 1 1234567 1234567     0 28 nov 11:57 preProcessGraz01.py.e176110.6
-rw-r--r-- 1 1234567 1234567     0 28 nov 11:57 preProcessGraz01.py.e176110.7
-rw-r--r-- 1 1234567 1234567     0 28 nov 11:57 preProcessGraz01.py.e176110.8
-rw-r--r-- 1 1234567 1234567     0 28 nov 11:57 preProcessGraz01.py.e176110.9
-rw-r--r-- 1 1234567 1234567 16754 28 nov 11:59 preProcessGraz01.py.o176110.1
-rw-r--r-- 1 1234567 1234567 16617 28 nov 11:59 preProcessGraz01.py.o176110.10
-rw-r--r-- 1 1234567 1234567 16754 28 nov 11:59 preProcessGraz01.py.o176110.2
-rw-r--r-- 1 1234567 1234567 16754 28 nov 11:59 preProcessGraz01.py.o176110.3
-rw-r--r-- 1 1234567 1234567 16754 28 nov 11:59 preProcessGraz01.py.o176110.4
-rw-r--r-- 1 1234567 1234567 16754 28 nov 11:59 preProcessGraz01.py.o176110.5
```

```
-rw-r--r-- 1 1234567 1234567 16616 28 nov 11:59 preProcessGraz01.py.o176110.6
-rw-r--r-- 1 1234567 1234567 16616 28 nov 11:59 preProcessGraz01.py.o176110.7
-rw-r--r-- 1 1234567 1234567 16616 28 nov 11:59 preProcessGraz01.py.o176110.8
-rw-r--r-- 1 1234567 1234567 16616 28 nov 11:59 preProcessGraz01.py.o176110.9
```

2.1.3 Parallel Environments

In the previous sections, we have been launching jobs which only used a single slot or process. This way, we are only using the cluster as a collection of machines where processes do not interact between them. In order to run jobs which use several slots with threads or processes which collaborate to do the job, we need to use parallel environments. The parameter to use parallel environment is `-pe <environment_name> <number_of_slots>`. In table 2.1, we can see the different parallel environments available in the cluster.

Environment Name	Description
`make`	Default environment which launches the tasks using a *round-robin* ¶ algorithm to allocated the jobs.
`mpi`	Environment which uses a *fill-up*¶ algorithm to allocate the jobs and it allows the use of Message Passing Interface (MPI) to communicate the jobs which are in the same node.
`mpich`	Environment which uses a *fill-up*¶ algorithm to allocate the jobs and it also allows the use of MPI but, unlike `mpi`, this environment allows that jobs are in different nodes.
`orte`	Environment which uses a *round-robin*¶ algorithm to allocate the jobs and it allows the use of the Open Run-Time Environment (ORTE) environment.
`pthreads`	Environment to run jobs using multi-threading. This parallel environment enforces that a job is executed in a single node.

Table 2.1: Parallel environments available at the cluster.

In order to avoid reducing the performance of the cluster, it is important that jobs allocate as many jobs as processes or threads as they are using. This is specially important in multi-threading, where some programs and libraries tend to use as many threads as possible. However, when several of those programs are running in the same machine, they allocate more threads than the number of cores of the machine, resulting in a performance drop as part of the run-time is wasted by the operative system managing the threads. Therefore, it is important to make sure that when multi-threading is used, the number of threads is limited to the number of assigned slots.

¶The *round-robin* policy assigns the jobs iterating through the nodes while the *fill-up* policy keeps assigning jobs to a machine while there are free slots.

A job must specify always the exact number of slots it is using and never underestimate or overestimate them. Users must be specially careful with programs like **Matlab** which are able to run multithreading but using always as many threads as cores has the system. In these cases, the system administrator will remove these jobs and warn the user.

In the previous example, we have been forcing ImageMagick to work using a single thread with **-limit thread 1**. Now, we can modify the program so that the number of threads depends on the number of slots allocated in the **qsub** call:

Listing 2.3: Processing script: preProcessGraz02.py

```
import os, glob, sys

if __name__ == '__main__':
    # Get the environment variables ..........................
    task_id = int(os.environ['SGE_TASK_ID'])
    task_first = int(os.environ['SGE_TASK_FIRST'])
    task_end = int(os.environ['SGE_TASK_LAST']) + 1
    task_step = int(os.environ['SGE_TASK_STEPSIZE'])
    nthreads = os.environ['NSLOTS']

    # Get the task ID ........................................
    tasks = range(task_first, task_end, task_step)
    number_of_jobs = len(tasks)
    job_id = tasks.index(task_id)

    # Create a list with the database image routes ..............
    images  = glob.glob('bike/*.bmp')
    images += glob.glob('cars/*.bmp')
    images += glob.glob('person/*.bmp')
    images += glob.glob('none/*.bmp')

    cmd_mask  = "convert %s -limit thread " + nthreads
    cmd_mask += " -interpolate spline -resize 1024x768"
    cmd_mask += " -sharpen 0x1 -normalize -resize 640x480 %s"
    for index in range(job_id, len(images), number_of_jobs):
        input_filename = images[index]
        output_filename = input_filename[:-3] + 'png'
        cmd = cmd_mask % (input_filename, output_filename)
        print cmd
        print os.popen(cmd).read()
        sys.stdout.flush()
```

Now, we can launch the job allocating 4 threads per job:

```
[1234567@dccclus ~]$ qsub -cwd -V -S /usr/bin/python -q fast.master.q -t 1-10 \
-l mem_token=5M,mem_free=5M -pe pthreads 4 preProcessGraz02.py
Your job-array 176114.1-10:1 ("preProcessGraz02.py") has been submitted
```

The number of threads set in the parameter **-pe pthreads 4** is stored in the environment variable **NSLOTS**, so that, the **Python** script is able to know how many threads can it use.

The GE allocates the requested resources for each used slot not for each job. For instance, resources like **mem_token** and **mem_free** are assigned to each slot, so that, the total number of memory required is obtained by multiplying the number of slots by the memory value. In the previous example, each job requeres 5Mb of memory, so each job will require 20Mb of memory.

We can also use ranges in the parallel environment parameter by using **-pe <env_name> <minimum>-<maximum>**. When ranges are used, the GE will try to use the maximum number of slots selected and decreasing that value until the task can be executed in the cluster. For instance, using a range between 2 and 8 threads,

```
[1234567@dccclus ~]$ qsub -cwd -V -S /usr/bin/python -q fast.master.q -t 1-10 \
-l mem_token=5M,mem_free=5M -pe pthreads 2-8 preProcessGraz02.py
```

the GE will first try to allocate the jobs using 8 slots. Once no more nodes are available with 8 free slots, it will try with 7 and it will keep decreasing until trying with 2 slots. Jobs which are not allocated will be waiting in the queue until at least 2 free slots are available in a machine. The **-pe** parameter also allows to use open ranges with only a minimum number of slots. For instance,

```
[1234567@dccclus ~]$ qsub -cwd -V -S /usr/bin/python -q fast.master.q -t 1-10 \
-l mem_token=5M,mem_free=5M -pe pthreads 4- preProcessGraz02.py
```

will try to the maximum number of slots for each job. Since the parallel environment is **pthreads**, the GE will try to assign 32 slots per job (i.e. the maximum number of virtual cores per node). Since it is an open range, depending on the parallel environment, you can allocate all free slots of a single job. For instance, launching a job with an open range using the parameter configuration **-pe mpich 2-**, the GE will allocate all free slots of the queue to the job.

Summarizing, parallel environments allows to assign several slots to the same job. This enables to use parallelization to speed up algorithms. However, algorithm parallelization is not straight forward and it requires to modify the algorithms in order to use methods like OpenMP, MPI or ORTE. We will see examples of programs exploiting parallelization in **Python** and **Matlab** in the next chapter.

2.2 Retrieving information

Until now, we have briefly review the use of the **qsub** command to submit jobs to the cluster. Now, let us see commands to retrieve the status of the jobs and to obtain information about the cluster.

2.2.1 Tasks information

The command to get the status of jobs and queues is **qstat**. Using **qstat** without any parameter will give us information about the your jobs in the cluster: identifier, priority, state, starting time, queue and used slots. For instance, calling **qstat** after launching the previous experiment will show us:

```
[1234567@dccclus graz02]$ qstat
job-ID  prior   name       user         state submit/start at     queue                          slots ja-task-ID
```

```
---------------------------------------------------------------------------------------------------------------
176113 0.55500 preProcess 1234567      r     11/28/2013 12:22:39 fast.master.q@compute-0-16.loc     4 1
176113 0.55500 preProcess 1234567      r     11/28/2013 12:22:39 fast.master.q@compute-0-16.loc     4 2
176113 0.55500 preProcess 1234567      r     11/28/2013 12:22:39 fast.master.q@compute-0-16.loc     4 3
176113 0.55500 preProcess 1234567      r     11/28/2013 12:22:39 fast.master.q@compute-0-16.loc     4 4
176113 0.55500 preProcess 1234567      r     11/28/2013 12:22:39 fast.master.q@compute-0-16.loc     4 5
176113 0.55500 preProcess 1234567      r     11/28/2013 12:22:39 fast.master.q@compute-0-16.loc     4 6
176113 0.55500 preProcess 1234567      r     11/28/2013 12:22:39 fast.master.q@compute-0-16.loc     4 7
176113 0.55500 preProcess 1234567      r     11/28/2013 12:22:39 fast.master.q@compute-0-16.loc     4 8
176113 0.55500 preProcess 1234567      r     11/28/2013 12:22:39 fast.master.q@compute-0-17.loc     4 9
176113 0.55500 preProcess 1234567      r     11/28/2013 12:22:39 fast.master.q@compute-0-17.loc     4 10
```

In the previous experiment, we have launched a job-array of ten tasks using 4 slots each, so that in the **slots** column we can see a 4 while the **ja-task-ID** column shows the job-array identifier assigned to each job. The state column tell us the current state of the job: **r** when the job is running, **D** while the GE is deleting it, **t** when the information is transfered to the job (while it is starting), **qw** when the job is waiting to be assigned or **E** when an error have occurred.

The parameters **-q <queue_name>** and **-u <user>** can be used to refine the results. The parameter **-q <queue_name>** shows only the jobs at the selected queue, while **-u <user>** shows the results of the specified user. Using **-u '*'** allows to see the jobs status of all users. This parameter is useful to check the workload of the cluster.

We can also use **qstat** to get further information about the jobs while using the parameter **-f**. For instance, the "full" display format **-f** used together with **-u '*'** and **-q fast.master.q** show us the jobs running by all users in the **fast.master.q** queue and the number of used and free slots in the queue:

```
[1234567@dccclus graz02]$ qstat -f -u '*' -q fast.master.q
queuename                      qtype resv/used/tot. load_avg arch          states
---------------------------------------------------------------------------------
fast.master.q@compute-0-13.loc BIP   0/32/32        0.00     linux-x64
 176117 0.55500 preProcess 1234567      r     11/28/2013 15:49:24     4 1
 176117 0.55500 preProcess 1234567      r     11/28/2013 15:49:24     4 2
 176117 0.55500 preProcess 1234567      r     11/28/2013 15:49:24     4 3
 176117 0.55500 preProcess 1234567      r     11/28/2013 15:49:24     4 4
 176117 0.55500 preProcess 1234567      r     11/28/2013 15:49:24     4 5
 176117 0.55500 preProcess 1234567      r     11/28/2013 15:49:24     4 6
 176117 0.55500 preProcess 1234567      r     11/28/2013 15:49:24     4 7
 176117 0.55500 preProcess 1234567      r     11/28/2013 15:49:24     4 8
---------------------------------------------------------------------------------
fast.master.q@compute-0-14.loc BIP   0/0/32         0.00     linux-x64
---------------------------------------------------------------------------------
fast.master.q@compute-0-15.loc BIP   0/0/32         0.00     linux-x64
---------------------------------------------------------------------------------
fast.master.q@compute-0-16.loc BIP   0/0/32         1.03     linux-x64
---------------------------------------------------------------------------------
fast.master.q@compute-0-17.loc BIP   0/0/32         5.24     linux-x64
---------------------------------------------------------------------------------
fast.master.q@compute-0-19.loc BIP   0/8/32         0.00     linux-x64
 176117 0.55500 preProcess 1234567      r     11/28/2013 15:49:24     4 9
 176117 0.55500 preProcess 1234567      r     11/28/2013 15:49:24     4 10
```

When the queue has more jobs assigned than free slots, this command will show you information of jobs pending to enter the queue at the end of the report:

```
############################################################################
 - PENDING JOBS - PENDING JOBS - PENDING JOBS - PENDING JOBS - PENDING JOBS
############################################################################
 176119 0.60500 script_gen 1234567      qw    11/28/2013 16:01:21     4
```

```
 176120 0.60500 script_gen 1234567      qw    11/28/2013 16:01:21     4
 176121 0.60500 script_gen 1234567      qw    11/28/2013 16:01:21     4
 176122 0.60500 script_gen 1234567      qw    11/28/2013 16:01:21     4
```

In order to get information of a job, we can also use the parameter **`-j <job_id>`**:

```
[1234567@dccclus graz02]$ qstat -j  176117
==============================================================
job_number:                 176117
exec_file:                  job_scripts/176117
submission_time:            Thu Nov 28 19:17:02 2013
owner:                      1234567
uid:                        501
group:                      1234567
gid:                        503
sge_o_home:                 /home/1234567
sge_o_log_name:             1234567
...
...
...
...
parallel environment:  pthreads range: 4
job-array tasks:            1-10:1
usage    1:                 cpu=00:00:01, mem=0.00000 GBs, io=0.00930, vmem=484.953M, maxvmem=484.953M
usage    2:                 cpu=00:00:01, mem=0.09426 GBs, io=0.00941, vmem=482.773M, maxvmem=482.773M
usage    3:                 cpu=00:00:01, mem=0.01317 GBs, io=0.00941, vmem=482.719M, maxvmem=482.719M
usage    4:                 cpu=00:00:02, mem=0.29113 GBs, io=0.00930, vmem=484.953M, maxvmem=484.953M
usage    5:                 cpu=00:00:01, mem=0.09786 GBs, io=0.00941, vmem=482.770M, maxvmem=482.770M
usage    6:                 cpu=00:00:01, mem=0.13931 GBs, io=0.00948, vmem=482.773M, maxvmem=482.773M
usage    7:                 cpu=00:00:01, mem=0.04921 GBs, io=0.00956, vmem=482.770M, maxvmem=482.770M
usage    8:                 cpu=00:00:01, mem=0.04921 GBs, io=0.00946, vmem=482.766M, maxvmem=482.766M
usage    9:                 cpu=00:00:02, mem=0.24637 GBs, io=0.00956, vmem=484.949M, maxvmem=484.949M
usage   10:                 cpu=00:00:02, mem=0.19074 GBs, io=0.00937, vmem=484.953M, maxvmem=484.953M
scheduling info:            (Collecting of scheduler job information is turned off)
```

An important piece of information present in the job status is in the last rows where it shows the **`usage`**. The **`usage`** shows the amount of time and memory used by the job and the **`maxvmem`** field shows the maximum amount of memory used by the job. Although it is *virtual memory*, this field can use used as a rough estimation of the memory requirements of the job and to better tune the **`mem_token`** and **`mem_free`** variables.

Finally, we can use the command **`qstat`** to check the load of all queues by using:

```
[1234567@dccclus graz02]$ qstat -g c
CLUSTER QUEUE                   CQLOAD   USED    RES  AVAIL  TOTAL aoACDS  cdsuE
--------------------------------------------------------------------------------
all.q                             0.00      1      0    543    736      0    192
fast.master.q                     0.00     50      0    142    736      0    544
long.master.q                     0.00      0      0    192    736      0    544
medium.master.q                   0.00      0      0    192    736      0    544
```

This command shows that the queue **`all.q`** has occupied single slot and 543 available while **`fast.master.q`** has 50 occupied slots used and 142 available. This command is useful to decide to which queue launch your experiments.

2.2.2 Cluster information

The command **`qconf`** is used to configure the GE, which is beyond the use of a normal user, but it also gives information about it. The number of queues and their names can be known using the command **`qconf -sql`**. For instance, running it on the cluster will generate the following output:

```
[1234567@dccclus graz02]$ qconf -sql
all.q
fast.master.q
long.master.q
medium.master.q
```

Using the command **qconf -sq <queue_name>**, we can get information about the queue. For instance, running this command for the **fast.master.q** queue will generate the following report:

```
[1234567@dccclus graz02]$ qconf -sq fast.master.q
qname                 fast.master.q
hostlist              @allhosts
seq_no                0
load_thresholds       np_load_avg=1.75
suspend_thresholds    NONE
nsuspend              1
suspend_interval      00:05:00
priority              0
min_cpu_interval      00:05:00
processors            UNDEFINED
qtype                 BATCH INTERACTIVE
ckpt_list             NONE
pe_list               make mpi mpich orte pthreads
...
...
...
...
s_rt                  INFINITY
h_rt                  02:30:00
s_cpu                 INFINITY
h_cpu                 INFINITY
s_fsize               INFINITY
h_fsize               INFINITY
s_data                INFINITY
h_data                INFINITY
s_stack               INFINITY
h_stack               INFINITY
s_core                INFINITY
h_core                INFINITY
s_rss                 INFINITY
h_rss                 INFINITY
s_vmem                INFINITY
h_vmem                INFINITY
```

This is the general information of the queue. For example, the time limit of the **fast.master.q** queue can be checked in the variable **h_rt** or **pe_list** has the identifier of the parallel environments available in the queue.

We can also check the values assigned by default by the GE to the queue by using the command **qconf -sc**. For example, to check which is the default value assigned to the variable **mem_token**, we can run,

```
[1234567@dccclus graz02]$ qconf -sc | grep mem_token
mem_token           mtok       MEMORY      <=    YES         YES        70G      0
```

which shows us that the default value is 70 Gb. This value is set higher than the actual capacity of the cluster nodes in order to force users to set the **mem_token** parameter.

Finally, we can also use **qconf** to get information about the parallel environments. The command, **qconf -spl** will give us the list of parallel environments available and the command **qconf -sp <environment_name>** will return information about a parallel environment:

```
[1234567@dccclus graz02]$ qconf -spl
make
mpi
mpich
orte
pthreads
[1234567@dccclus graz02]$ qconf -sp pthreads
pe_name            pthreads
slots              9999
user_lists         NONE
xuser_lists        NONE
start_proc_args    /bin/true
stop_proc_args     /bin/true
allocation_rule    $pe_slots
control_slaves     FALSE
job_is_first_task  TRUE
urgency_slots      min
accounting_summary TRUE
```

We can also get information about the nodes using the command **qhost**:

```
[1234567@dccclus graz02]$ qhost
HOSTNAME                ARCH         NCPU  LOAD  MEMTOT  MEMUSE  SWAPTO  SWAPUS
----------------------------------------------------------------------------------
global                  -               -     -       -       -       -       -
compute-0-0             linux-x64      32  0.00   63.0G    1.3G  122.1G     0.0
compute-0-1             linux-x64      32  0.00   63.0G    1.4G  122.1G     0.0
compute-0-10            linux-x64      32  0.01   63.0G    1.2G  122.1G     0.0
compute-0-11            linux-x64      32  0.00   63.0G    1.2G  122.1G     0.0
compute-0-12            linux-x64      32  0.00   63.0G    1.2G  122.1G     0.0
compute-0-13            linux-x64      32  0.02   63.0G    1.2G  122.1G     0.0
compute-0-14            linux-x64      32  0.53   63.0G    1.2G  122.1G     0.0
compute-0-15            linux-x64      32  0.20   63.0G    1.2G  122.1G    2.7M
compute-0-16            linux-x64      32  0.72   63.0G    1.2G  122.1G     0.0
compute-0-17            linux-x64      32  0.74   63.0G    1.1G  122.1G    1.3M
compute-0-19            linux-x64      32  0.62   63.0G    1.2G  122.1G     0.0
compute-0-2             linux-x64      32  0.00   63.0G    1.3G  122.1G     0.0
compute-0-20            linux-x64      32  0.00   63.0G    1.2G  122.1G     0.0
compute-0-21            linux-x64      32  0.00   63.0G    1.2G  122.1G     0.0
compute-0-22            linux-x64      32  0.00   63.0G    1.2G  122.1G     0.0
compute-0-23            linux-x64      32  1.00   63.0G    2.0G  122.1G     0.0
compute-0-3             linux-x64      32  0.00   63.0G    1.2G  122.1G     0.0
compute-0-4             linux-x64      32  0.00   63.0G    1.2G  122.1G  136.0K
compute-0-5             linux-x64      32  0.00   63.0G    1.4G  122.1G     0.0
compute-0-6             linux-x64      32  0.02   63.0G    1.2G  122.1G     0.0
compute-0-7             linux-x64      32  0.00   63.0G    1.4G  122.1G     0.0
compute-0-8             linux-x64      32  0.00   63.0G    1.2G  122.1G     0.0
compute-0-9             linux-x64      32  0.00   63.0G    1.4G  122.1G     0.0
```

Using the parameters, we can filter the results or obtain more information about a nodes. For instance, to know the current value of **mem_token** on several nodes, we can use the parameter **-h <nodes>** to filter the nodes and **-F [<resources>]** to get information about **mem_token**:

```
[1234567@dccclus graz02]$ qhost -F mem_token -h compute-0-14,compute-0-15,compute-0-16,compute-0-17
HOSTNAME                ARCH         NCPU  LOAD  MEMTOT  MEMUSE  SWAPTO  SWAPUS
----------------------------------------------------------------------------------
compute-0-14            linux-x64      32  0.70   63.0G    1.2G  122.1G     0.0
    Host Resource(s):      hc:mem_token=3.000G
compute-0-15            linux-x64      32  0.27   63.0G    1.2G  122.1G    2.7M
    Host Resource(s):      hc:mem_token=53.000G
compute-0-16            linux-x64      32  0.54   63.0G    1.3G  122.1G     0.0
    Host Resource(s):      hc:mem_token=3.000G
compute-0-17            linux-x64      32  0.57   63.0G    1.3G  122.1G    1.3M
    Host Resource(s):      hc:mem_token=3.000G
```

The output shows the information about the current memory available in the selected nodes.

2.2.3 Modify pending jobs

Some parameters of jobs which are already waiting in the GE to be executed (i.e. pending jobs) can be modified using the command **qalter** For instance, if the job 176122 is waiting to be executed in the queue **fast.master.q** because there are not free slots but queue **long.master.q** has enough free slots, then we can use the command **qalter -q <queue_name> <job_id>** to change the job queue:

```
[1234567@dccclus ~]$ qalter -q long.master.q 176122
modified hard queue list of job 176122
```

Also, the **qalter** command can also be used to get information about pending jobs using **qalter -w p <job_id>**. This command allow us to know the reasons which have prevented the job to be running. For instance,

```
[1234567@dccclus ~]$ qsub -cwd -V -S /usr/bin/python -q fast.master.q -t 1-10 \
-l mem_token=5M,mem_free=5M -pe pthreads 4 preProcessGraz02.py
Your job-array 176122.1-10:1 ("preProcessGraz01.py") has been submitted
[1234567@dccclus graz02]$ qalter -w p 176122
```

```
Job 176122 cannot run in queue "all.q" because it is not contained in its hard
    queue list (-q)
Job 176122 cannot run in queue "long.master.q" because it is not contained in its
     hard queue list (-q)
Job 176122 cannot run in queue "medium.master.q" because it is not contained in
    its hard queue list (-q)
Job 176122 cannot run in PE "pthreads" because it only offers 0 slots
verification: no suitable queues
```

The messages tells that the job cannot be assigned to queues **all.q**, **long.master.q** and **medium.master.q** because the job is assigned into list **fast.master.q**. Then, it tells that in the **fast.master.q** queue there are no free slots for the **pthreads** parallel environment. Finally, it concludes that are not any queue where the job can be executed.

To remove our jobs from the GE, we can use the command **qdel <job_id>**:

```
[1234567@dccclus ~]$ qdel 176122
1234567 has registered the job-array task 176122.4 for deletion
1234567 has registered the job-array task 176122.7 for deletion
1234567 has registered the job-array task 176122.8 for deletion
1234567 has registered the job-array task 176122.9 for deletion
1234567 has deleted job 176122
```

Also, you can use **qdel '*'** to remove all your jobs from the cluster. Some jobs may remain in the queue after they have been registered for deletion. When they remain in this status for too long, you can try to force the deletion with the parameter **qdel -f <job_id>**.

2.3 Command parameter summary

Tables summarizing the parameters used with the different commands. For more information and parameters, you can check the commands manual (e.g. running **man qsub** from the command line to see the manual of the command **qsub**).

2.3.1 qsub parameters

The command **qsub** is used to launch new tasks to the cluster and set their configuration parameters.

Parameter	Description
`-b <y|n>`	The job is a binary (***y***) or an script (***n***).
`-cwd`	Set as the working path the route where the **`qsub`** is called.
`-V`	Exports the environment variables to the job context.
`-q <queue_list>`	List of queues where the job can be executed. When a job can be assigned to more than a single queue, the queues are separated with comas and without spaces: **`-q fast.master.q, medium.master.q`**
`-l <resources>`	List with the resources variables specifically set for the job. In the cluster, the **`mem_token`** and **`mem_free`** must be set using this parameter.
`-S <script>`	Change the program used to execute jobs when they are scripts. In the cluster, all jobs are supposed to be written in **`Bash`** by default. Possible values are: **`/usr/bin/python`**, **`/bin/bash`**, **`/bin/csh`**, **`/bin/sh`**, ...
`-t n[-m[:s]]`	Launch a job array. When this parameter is used, the GE generates an array of jobs which go from n to m (both included) using an step s. The identifier of the jobs and the values of n, m and s can be retrieved by the job as the environment variables: **`SGE_TASK_ID`**, **`SGE_TASK_FIRST`**, **`SGE_TASK_LAST`** and **`SGE_TASK_STEPSIZE`**.
`-pe <env_name> <slots>`	Defines the parallel environment and the number of slots used to run a parallel job in the cluster. A list with the parallel environments available in the cluster can be found in table 2.1.

2.3.2 `qstat` parameters

The command **`qstat`** returns information about the jobs in the cluster.

Parameter	Description
`-q <queue_list>`	Selects the queue which information is displayed.
`-f`	Shows the "full" display format.
`-u <user>`	Selects the users which information is displayed. Use `'*'` to show the information about all users.
`-j <job_id>`	Shows information about a job.
`-g c`	Shows information of the load of the different queue (i.e. used and available slots).

2.3.3 `qalter` parameters

The command **`qalter`** modifies the parameters of jobs which are already in the cluster.

Parameter	Description
`-q <queue_name>`	Changes the queue where a pending job is assigned.
`-w p <job_id>`	Returns a report which indicates what has prevented a job to be executed.

2.3.4 `qconf` parameters

The command **`qconf`** is used to configure the cluster but standard users can use it to get information about the cluster general configuration.

Parameter	Description
`-sql`	Returns the names of the queues of the cluster.
`-sq <queue>`	Returns the configuration of the selected queue.
`-sc`	Returns information about the cluster global parameters.
`-spl`	Returns the list with the parallel environments available in the cluster.
`-sp <env_name>`	Returns the configuration of the selected parallel environment.

2.3.5 `qhost` parameters

The command **`qhost`** is used to configure the parameters of the working nodes but standard users can use it to get information about the cluster nodes.

Parameter	Description
`-F [<resources>]`	Shows the values of the different resources of each cluster node. The elements of a resources list must be separated with commas and without spaces between elements (e.g. **`-F mem_free, mem_token`**). All resource will be shown when an empty list is given (e.g. **`qhost -F`**).
`-h <host_names>`	Filters the hosts which information is shown. The node names in the list must be separated by commas and without spaces between the elements.

3

Programming

In this chapter, we are going to make a brief introduction to parallel programming. The aim of the chapter is to show the advantages of using parallel programming in a cluster. The k-Means algorithm is going to be used to show the use of the cluster into three different scenarios both in `Python` and `Matlab`: to launch independent tasks, to use multiprocessing and while using MPI.

3.1 Introduction: the *k*-Means algorithm

The k-Means algorithms is an unsupervised clustering algorithm widely used in multiple fields due to the simplicity of its implementation and the results which it obtains. The basic version of the algorithm has an straightforward implementation but it also has a high computational cost. Therefore, we decided to use this algorithm to show the advantages of using parallel programming to accelerate algorithms.

The k-Means algorithm is an iterative algorithm which at each iteration minimizes the quadratic distance between the samples and their closest centroid:

$$\arg \min_{\{\mu_1,\dots,\mu_k\}} \sum_{i=1}^{k} \sum_{\mathbf{x}_j \in S_i} \|\mathbf{x}_j - \mu_i\|^2 \tag{3.1}$$

where $\mathbf{x}_j$ is the j-th sample, μ_i is the cluster centroid and $\mathbf{S_i}$ is the set of samples belonging to the i-th cluster (i.e. the samples whom closest centroid is μ_i).

In Algorithm 1, we have the psedo-code of the k-Means algorithm used to minimize Equation 3.1. The computational cost of the algorithm mostly lies on the nearest neighbor search which has a computational cost of $O(NKd)$ where N is the number of training samples, K is the number of clusters and d is the dimensionality of the samples. The number of iterations needed to converge is somewhat related to the dimensionality of the samples, so that, as the dimensionality grows the number of iterations needed to converge also grows. Therefore, the run-time of the k-Means algorithm greatly increases as the dimensionality and the number of samples grows, which is a common scenario when the algorithm is used in a computer vision tasks.

Figure 3.1 shows a toy example where the algorithm

In figure 3.1, we can see a *toy* example where the algorithm is used to find the centroids of five Gaussian distributions. In this example, the algorithm converges to the original

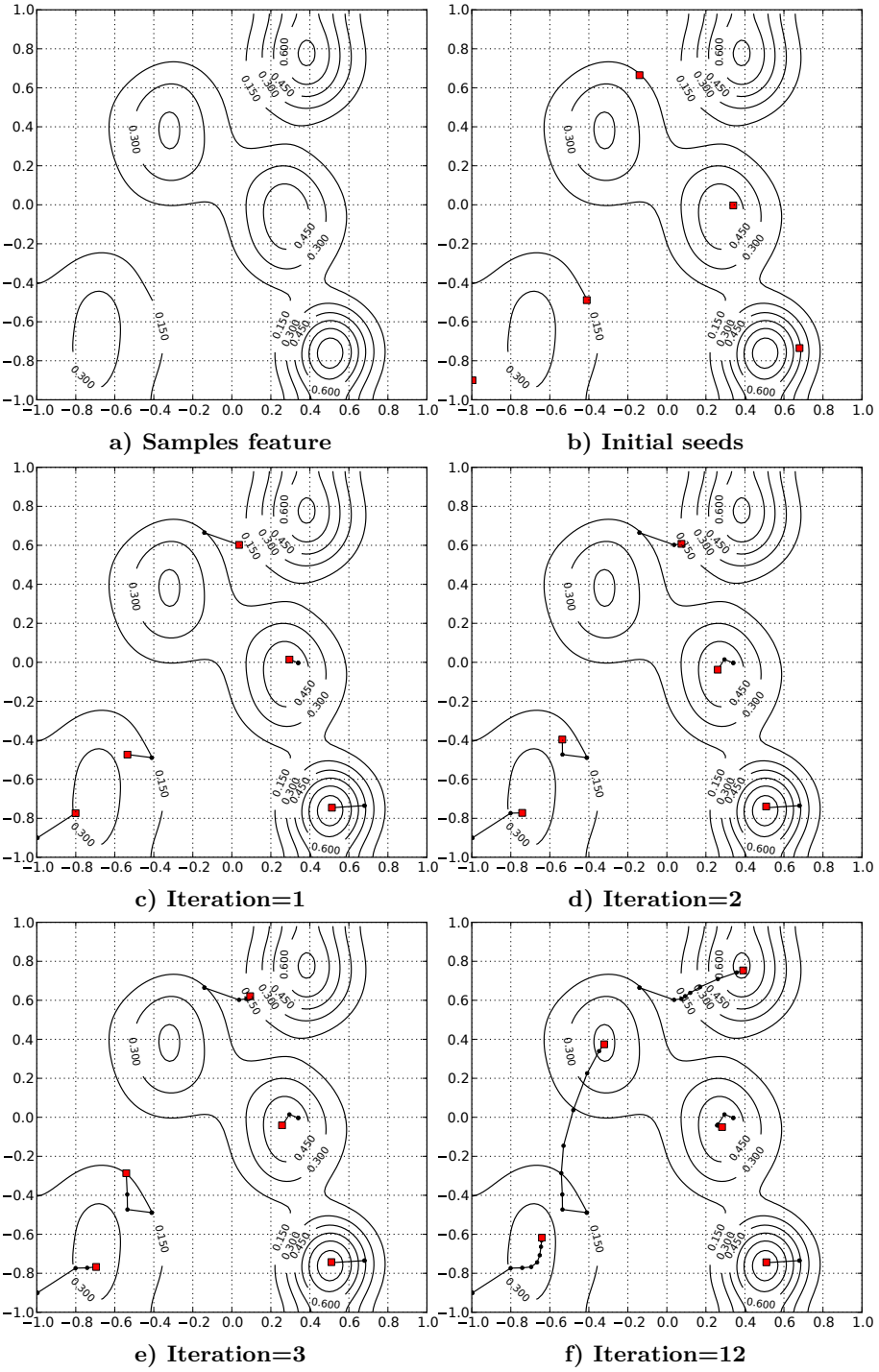


Figure 3.1: Samples generated using five Gaussian distributions: ***a)*** *Density map of the samples used to train the k-Means algorithm,* ***b)*** *initial seeds obtained using k-Means++ algorithm,* ***c-f)*** *evolution of cluster centroids at different iterations of the k-Means algorithm.*

distribution centroids. However, retrieving the original cluster centroids depends on the initial seeds and another initialization can give a completely different final results. Also, the likelihood of obtaining the original cluster centroids decreases as the number of dimensions increases. Therefore, it is usually necessary to run the k-Means algorithm several

Algorithm 1: *k-Means algorithm pseudo-code.*

input : $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$ (set of samples)
$\mathbf{C} = \{\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_K\}$ (cluster seeds)
$maximum\_iterations$ (iterations limit)
output: $\mathbf{C} = \{\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_K\}$ (cluster centroids)
$change \leftarrow true$;
$iteration \leftarrow 0$;
repeat
  $\mathbf{NC} \leftarrow 0_{k,d}$;
  $\mathbf{H} \leftarrow 0_{k,1}$;
  for $i \leftarrow 1$ **to** N **do**
    $c \leftarrow findNearestNeighbor(\mathbf{C}, \mathbf{x}_i)$;
    $\mathbf{h}_c \leftarrow \mathbf{h}_c + 1$;
    $\mathbf{nc}_c \leftarrow \mathbf{nc}_c + \mathbf{x}_i$;
  end
  $change \leftarrow false$;
  for $i \leftarrow 1$ **to** K **do**
    $\mathbf{c}_i \leftarrow \mathbf{nc}_i / \mathbf{h}_c$;
    $d \leftarrow \|\mathbf{nc}_i - \mathbf{c}_i\|^2$;
    if $d > 0$ **then**
      $change \leftarrow true$;
    end
  end
  $iteration \leftarrow iteration + 1$;
until *change = true and iteration* $\leq$ *maximum_iterations*;

times in order to validate the obtained results. Besides, the number of k-Means is usually set using a validation process where different k values are evaluated. The validation of the k parameters, the random initialization and the computational cost at high dimensional spaces, makes it specially important to accelerate the algorithm and to be able to run several experiments in parallel.

3.2 Sequential implementation

As a first version of the k-Means algorithm, we are going to implement a simple sequential version of the algorithm both in `Matlab` and in `Python`. The sequential version of the algorithm is designed so that it is easy to launch multiple jobs into the cluster using different parameter configurations (e.g. change the initial seeds or the number of clusters generated). This is the most straightforward way to use the cluster and it is very useful for algorithm which can be easily divided into independent tasks (e.g. multiclass classifiers learning using a one-vs-all strategy) or testing different sets of parameters (e.g. tunning the parameters of bag-of-visual-words framework).

In both versions of the algorithm, we calculate the Euclidean distance using the dot product in order to take a advantage of the matrix operators which are implemented using efficient libraries (both `Python` and `Matlab` use the `LAPACK` and `ATLAS` libraries to calculate most matrix operations). Then, the Euclidean distance can be calculated using the dot-product as:

$$\|\mathbf{x_j} - \mu_\mathbf{i}\|^2 = \|\mathbf{x_j}\|^2 + \|\mu_i\|^2 - 2\langle \mathbf{x}_j, \mu_i \rangle$$

Therefore, we are going to pre-calculate the L2-norm of the samples and centroids, so that, the Euclidean distance can be calculated as the addition of the centroids and sample norms minus the product between the sample and the centroids matrix.

3.2.1 Python version

The k-Means algorithm can be divided in two different parts: first, assign each sample to the closest centroid and calculate the center of masses of each cluster. The second part updates the cluster centroids and calculates the maximum update distance. Therefore, we decided two create two different functions two implement this parts. The function `calculateCentroids` calculates the new cluster centroids given the centroids, the samples and the normals of the samples:

Listing 3.1: function `calculateCentroids` in `kmeansTutorial.py`

```
1  def calculateCentroids(centroids, data, normals):
2      nsamples, ndimensions = data.shape
3      nclusters = centroids.shape[0]
4      centroid_normals = (centroids ** 2).sum(axis=1)
5      new_centroids = numpy.zeros((nclusters, ndimensions))
6      histogram = [0] * nclusters
7      # Assign each sample to their closest centroid.
8      for j in range(nsamples):
9          distances = centroid_normals + normals[j] - 2 * numpy.dot(centroids, data[j, :])
10         selected, distance = min(enumerate(distances), key = operator.itemgetter(1))
11         new_centroids[selected, :] += data[j, :]
12         histogram[selected] += 1
13     return new_centroids, histogram
```

This function returns the accumulated samples and the frequency histogram of the new centroids instead of directly returning the averaged centroids, so that, the function can be reused later. These accumulated samples and frequency histograms together with the current centroids are given to the function `updateCentroids` which averages the centroids and calculates the maximum update distance:

Listing 3.2: function `updateCentroids` in `kmeansTutorial.py`

```
1  def updateCentroids(centroids, new_centroids, histogram):
2      nclusters = centroids.shape[0]
3      update = 0
4      for j in range(nclusters):
5          new_centroids[j, :] /= histogram[j]
6          distance = sum((centroids[j, :] - new_centroids[j, :]) ** 2)
7          update = max(update, distance)
8          centroids[j, :] = new_centroids[j, :]
9      return update
```

Using this two functions, the sequential version of the algorithm can be simply implemented as a loop which calls both functions until the algorithm converges:

Listing 3.3: function `kmeans` in `kmeansTutorial.py`

```
1  def kmeans(data, seeds, maximum_iterations = 1000):
2      current_time = time.time()
3      nsamples, ndimensions = data.shape
4      nclusters = seeds.shape[0]
5      normals = (data ** 2).sum(axis=1)
6      centroids = seeds
```

```
7       for iteration in range(maximum_iterations):
8           # Assign each sample to the closest centroid and calculate the mean centroid.
9           new_centroids, histogram = calculateCentroids(centroids, data, normals)
10          # Calculate the maximum distance between the previous and the current centroids.
11          update = updateCentroids(centroids, new_centroids, histogram)
12          elapsed = time.time() - current_time
13          print "[ %09.3f ] Iteration %06d: update=%e" % (elapsed, iteration, update)
14          if update == 0:
15              break # Stop when the centroids have not changed.
16      print "[ %09.3f ] DONE" % (time.time() - current_time)
17      return centroids
```

Finally, we can create a script in **Python** to call the k-Means algorithm from the command line, so that, we can easily use the **qsub** command to launch the algorithm in the cluster using different parameter configurations:

Listing 3.4: launcher file **kmeans_basic_tutorial.py**

```
1  #!/bin/env python
2  import numpy, sys, cPickle
3  sys.path.append('.') # Make sure that kmeansTutorial.py is found.
4  import kmeansTutorial as tut
5
6  if __name__ == '__main__':
7      # Get parameters from command-line.
8      samples_filename = sys.argv[1]
9      seeds_filename = sys.argv[2]
10     nclusters = int(sys.argv[3])
11     cluster_filename = sys.argv[4]
12     # Load information from disk.
13     samples = tut.load(samples_filename)
14     seeds = tut.load(seeds_filename)[:nclusters, :]
15     # k-Means algorithm.
16     clusters = kmeans(seeds, seeds)
17     # Save the obtained clusters to disk.
18     tut.save(cluster_filename, clusters)
```

At the beginning of the script, we have added the line **sys.path.append('.')** to add the current folder to the library path of the node's **Python**, otherwise calling the script directly with a **qsub** can generate a **ImportError** exception while loading your own libraries (e.g. while loading the library kmeansTutorial.py). Since the script receives the samples filename, seeds filename, number of clusters and the destination cluster filename as arguments, we can easily launch experiments with different parameter configurations with the following command:

```
[1234567@dccclus graz02]$ qsub -S /usr/bin/python -cwd -V -q fast.master.q \
-l mem_free=1.5G,mem_token=1.5G kmeans_basic_tutorial.py data/samples002.data \
data/seeder002.data 16 data/clusters002.data
Your job 294536 ("kmeans_basic_tutorial.py") has been submitted
```

The file **kmeans_basic_tutorial.py.o294536** contains the output generated by the algorithm:

```
[1234567@dccclus python_kmeans]$ cat kmeans_basic_tutorial.py.o294536
[ 00000.357 ] Iteration 000000: update=3.434782e+00
[ 00000.642 ] Iteration 000001: update=3.701591e-02
[ 00000.927 ] Iteration 000002: update=1.131685e-02
[ 00001.211 ] Iteration 000003: update=5.155033e-03
[ 00001.496 ] Iteration 000004: update=3.664949e-03
```

```
[ 00001.780 ] Iteration 000005: update=2.957448e-03
[ 00002.065 ] Iteration 000006: update=2.124332e-03
[ 00002.350 ] Iteration 000007: update=1.786910e-03
...
...
...
[ 00036.935 ] Iteration 000129: update=1.597055e-05
[ 00037.217 ] Iteration 000130: update=0.000000e+00
[ 00037.217 ] DONE
```

3.2.2 Matlab version

We have implemented the k-Means algorithm in **Matlab** as in **Python**, we have two auxiliary functions to calculate the new centroids and the maximum distance update and a main function which simply calls both functions until the algorithm converges:

Listing 3.5: function `kmeans` *in* `kmeansBasicTutorial.m`

```
function [centroids] = kmeansBasicTutorial(data, seeds, maximum_iterations)
    [nsamples, ndimensions] = size(data);
    tic;
    centroids = seeds(:, :);
    nclusters = size(centroids, 1);
    normals = sum(data .* data, 2);
    time_first_iteration = tic;
    for iteration = 1:maximum_iterations,
        % Assign each sample to their closest centroid.
        [new_centroids, histogram] = calculateCentroids(centroids, data, normals);
        % Update the accumulated centroids and calculate the maximum update distance.
        [update, centroids] = updateCentroids(centroids, new_centroids, histogram)
        disp(sprintf('[ %09.3f ] Iteration %06d: update=%e', toc, iteration, update));
        if update == 0,
            break
        end
    end
    average_time = toc(time_first_iteration) / (iteration + 1);
    disp(sprintf('[ %09.3f ] Average time per iteration: %f', toc, average_time));
```

Then, we have implemented the **calculateCentroids** function as a local function in the **kmeansBasicTutorial.m** as:

Listing 3.6: function `calculateCentroids` *in* `kmeansBasicTutorial.m`

```
function [new_centroids, histogram] = calculateCentroids(centroids, data, normals)
    [nsamples, ndimensions] = size(data);
    nclusters = size(centroids, 1);
    centroid_normals = sum(centroids .* centroids, 2);
    new_centroids = zeros(nclusters, ndimensions);
    histogram = zeros(nclusters, 1);
    for j = 1:nsamples,
        current = centroid_normals + normals(j) - 2 * (centroids * data(j, :)');
        [distance, selected] = min(current);
        new_centroids(selected, :) = new_centroids(selected, :) + data(j, :);
        histogram(selected) = histogram(selected) + 1;
    end
```

And, the **updateCentroids** function as:

Listing 3.7: function `updateCentroids` *in* `kmeansBasicTutorial.m`

```
function [update, centroids] = updateCentroids(centroids, new_centroids, histogram)
    nclusters = size(centroids, 1);
    update = 0;
    for j = 1:nclusters,
        new_centroids(j, :) = new_centroids(j, :) / histogram(j);
        update = max(update, sum((centroids(j, :) - new_centroids(j, :)) .^ 2));
        centroids(j, :) = new_centroids(j, :);
    end
```

Finally, we have created a **Bash** script to call the **Matlab** function with different parameter configurations. Since the number of **Matlab** instructions needed to load the information and call the k-Means algorithm is small, the **Bash** script embeds the **Matlab** instructions in the **Matlab** call:

Listing 3.8: file **kmeansTutorialBasic.sh**

```
#!/bin/bash
export MATLAB=/opt/matlab/
export PATH=${MATLAB}:${PATH}
$MATLAB/bin/matlab -singleCompThread -nojvm -nodisplay -nosplash -nodesktop -r "\
load('$1', 'samples');\
load('$2', 'centroids');\
clusters = kmeansBasicTutorial(samples, centroids(:$3, :), 1000);\
save('$4', 'clusters');\
exit;"
```

We can launch the **Matlab** version of the algorithm in the cluster using a similar command as in the **Python** version:

```
[1234567@dccclus graz02]$ qsub -cwd -V -q fast.master.q -l mem_free=1.5G,\
mem_token=1.5G kmeansTutorialBasic.sh data/samples/samples_100K_64_0000.mat \
data/clusters/seeder_100K_64_0000.mat 16 data/clusters/clusters_100K_64_0000.mat
Your job 294538 ("kmeans_basic_tutorial.py") has been submitted
```

The output of the algorithm is in the file **kmeansTutorialBasic.sh.o294538**:

```
[1234567@dccclus matlab_kmeans]$ cat kmeansTutorialBasic.sh.o294538

                            < M A T L A B (R) >
                  Copyright 1984-2012 The MathWorks, Inc.
                   R2012a (7.14.0.739) 64-bit (glnxa64)
                              February 9, 2012

To get started, type one of these: helpwin, helpdesk, or demo.
For product information, visit www.mathworks.com.

[ 00001.360 ] Iteration 000001: update=3.644334e+00
[ 00002.521 ] Iteration 000002: update=1.749069e-02
[ 00003.643 ] Iteration 000003: update=3.250518e-03
[ 00004.772 ] Iteration 000004: update=1.309559e-03
[ 00005.892 ] Iteration 000005: update=9.951048e-04
[ 00007.007 ] Iteration 000006: update=7.457801e-04
[ 00008.125 ] Iteration 000007: update=6.870820e-04
...
...
...
[ 00806.736 ] Iteration 000727: update=2.642367e-07
[ 00807.842 ] Iteration 000728: update=0.000000e+00
```

```
[ 00807.843 ] Average time per iteration: 1.108096
```

3.2.3 Launching randomized algorithms

The files of the previous **Matlab** and **Python** examples can be found in the folders **~/example/matlab_kmeans** and **~/example/python_kmeans**, respectively. In both cases, in those folders you can find scripts used to generate the random samples and the seeds as well as the scripts used to launch multiple experiments in the cluster.

In those examples, you can see that the randomized algorithms usually have a parameter to initialize the random seeder. For instance, the **Matlab** seeder and samples generators scripts begin with

```
s = RandStream('mcg1607', 'Seed', random_seed);
RandStream.setGlobalStream(s);
```

which sets the seed of the random pseudo-number generator of the **Matlab**. The first line sets the type of algorithm and initial random seed while, the second line sets the newly generated seed as the seed for the whole **Matlab**. In **Python**, it is done similarly by setting the initial seed with **random.seed(random_seed)** or **numpy.random.seed(random_seed)**, depending which random seeder is initialized.

Both in the **Matlab** and **Python** cases, the random seeds values are generated by the scripts which launch the experiments in the cluster. These scripts give the generated initial random seeds to the cluster experiments as another parameter. This is done because random pseudo-generators are usually initialized using the system date. Therefore, it is possible that multiple jobs simultaneously launched to the cluster end up obtaining the same pseudo-random generator. This is an undesired scenario in certain tasks like seeding the initial centroids where multiple k-Means++ algorithms launched at the same time may generate the exact same set of initial centroids when this is not taken into account.

3.3 Multi-processing implementation

In the previous section, we have presented the basic version of the k-Means algorithm which uses only a single thread/process to run the algorithm. Since the main bottleneck of the algorithm is to assign each sample to its nearest centroid, a simple way to reduce the computational time is to calculate the nearest centroid concurrently. Both **Python** and **Matlab** can natively use multiprocessing but not multithreading. Multiprocessing and multithreading are similar paradigms with the major difference that in multithreading the threads share the memory space while in multiprocessing processes are in separated memory spaces. In multithreading all variables and objects are in the same memory space and they are accessible by all threads. Therefore, the programmer has to ensure that the code is thread-safe, so that, the code only manipulates data structures in a manner which guarantees the safe execution of multiple threads at the same time. For instance, by ensuring that a value which is being read by a thread cannot be modified by any other thread. In contrast, multiprocessing does not share the same memory space so that structures from a process are not accessible from another process. Therefore, the communication between processes is not as straightforward as it is in the multithreading and methods like pipes, queues or shared memory maps are used to share data between processes. The use of such structures has the disadvantage that there may be some overheat while transferring data between processes. However, it has the advantage that

the programmer does not has to ensure that the processes are thread-safe and methods like mutex, locks or semaphores are not needed.

Since in the default `Matlab` configuration the number of threads cannot be controlled, always disable it using the parameter `-singleCompThread` when calling `Matlab`.
Jobs must **allocate as many slots as processes or threads the task needs**, otherwise the GE can allocate too many jobs in a node, decreasing the overall performance of the cluster. In these cases, the system administrator can remove these under-allocated jobs and warn the user.

Although `Matlab` and `Python` does not support multithreading natively, multithreading can still be used in both of them by using external libraries. Actually, some `Python` packages and `Matlab` *mex* libraries which are implemented in other languages like C/C++ or Fortran, allow to run their algorithms using multithreading. When using those libraries, it is important that the **job allocates as many slots as threads are used** by the library, so that, it is important to be able to select how many threads the program is going to use. For instance, `Matlab` uses standard libraries like LAPACK or ATLAS to do most matrix operations. These libraries support multithreading and `Matlab` is configured by default to use as many threads as possible. This provokes that `Matlab` allocates as many threads as cores available in a cluster node when multithreading is enabled. Usually, this leads to use more threads than the number of cores available in a node which decreases the overall performance of that node since part of the resources are diverted to the operative system to manage the threads. This reduction in performance can be quite important and it can result in the algorithm running slower than without using multithreading. Therefore, the parameter `-singleCompThread` which forces `Matlab` to run in a single thread must be always used while launching a `Matlab` job.

3.3.1 Python version

The multiprocessing is added to the `calculateCentroids` function in order to concurrently calculate the closest centroid of each sample. A new main function called `kmeansMP` is created from the previous `kmeans` function with the only differences that it has a new parameter to specify the number of processes used and that it calls the multiprocessing function `calculateCentroidsMP` to calculate the new cluster centroids:

Listing 3.9: function `calculateCentroidsMP` in `kmeansTutorial.py`

```
1  import multiprocessing as mt
2
3  def calculateCentroidsMP(centroids, data, normals, number_of_processes):
4      global shared
5      nsamples, ndimensions = data.shape
6      nclusters = centroids.shape[0]
7      centroid_normals = (centroids ** 2).sum(axis=1)
8      # Initialize structures.
9      shared = [data, normals, centroids, centroid_normals]
10     processes = mt.Pool(number_of_processes)
11     new_centroids = numpy.zeros((nclusters, ndimensions))
12     histogram = [0] * nclusters
```

```
13      # Assign each sample concurrently.
14      selected_centroid = processes.map(nearestNeighbor, range(nsamples))
15      # Assign each sample to their closest centroid.
16      for j in range(nsamples):
17          new_centroids[selected_centroid[j], :] += data[j, :]
18          histogram[selected_centroid[j]] += 1
19      processes.terminate()
20      return (new_centroids, histogram)
```

The function uses the **Python** package **multiprocessing** to implement the concurrency. At each iteration, it creates a pool of processes which calculate the index of the closest centroid for each sample. The indexes are calculated using an auxiliary function:

Listing 3.10: function `nearestNeighbor` *in* `kmeansTutorial.py`

```
1 def nearestNeighbor(j):
2     distances = shared[3] + shared[1][j] - 2 * numpy.dot(shared[2], shared[0][j, :])
3     selected, distance = min(enumerate(distances), key = operator.itemgetter(1))
4     return selected
```

This function is passed to the **map** function of the **processes** object. This **map** works like the standard **Python map** function but it divides the elements of the processed object between the processes of the pool. To reduce the communication between processes, the variable **shared** is declared global and shared between all processes.

The new **kmeansMP** function can be directly called using the script **kmeans_mp_tutorial.py** in the examples folder. The new script uses the environment variable **NSLOTS** to set the number of processes used by the concurrent k-Means algorithm. Therefore, we can easily control the number of processes used by our algorithm using the parallel environment options of the **qsub** call:

```
[1234567@dccclus graz02]$ qsub -S /usr/bin/python -cwd -V -q fast.master.q \
-l mem_free=1.0G,mem_token=1.0G -pe pthreads 4 kmeans_mp_tutorial.py \
data/samples003.data data/seeder003.data 128 data/clusters003.data
```

The **qsub** call uses the parallel environment **pthreads** to ensure that all processes are allocated in the same node. Using different number of processes in **-pe pthreads <number_of_threads>**, we can evaluate the relationship between the computational time and the number of processes used:

Number of processes	1	2	4	8	16	32
Iteration	32.94	16.81	9.75	6.01	3.79	3,18

Table 3.1: Runtime of the k-Means algorithm and time per iteration using different number of processes in **Python**.

As can be seen in table 3.1, the computational cost is reduced as more processes are used. Note that the reduction is not linear to the number of processes. Partially because as more processes are used more computational time is diverted to manage this processes but also because CPU resources are shared between processes and half of the nodes available are obtained from hyper-threading rather than being actual cores. For instance, the use of multiprocessing can increase the number of misses of the cache if not implemented carefully.

3.3.2 Matlab version

In the **Matlab** version, we made a similar modification of the algorithm. A new k-Means function named **kmeansMPTutorial** has been created from the original **kmeansTutorial** function and a new auxiliary function is used to calculate the new cluster centroids:

Listing 3.11: auxiliary function `calculateCentroidsMP` *in* `kmeansMPTutorial.m`

```
function [new_centroids, histogram] = calculateCentroidsMP(centroids, data, normals)
    [nsamples, ndimensions] = size(data);
    nclusters = size(centroids, 1);
    centroid_normals = sum(centroids .* centroids, 2);
    new_centroids = zeros(nclusters, ndimensions);
    histogram = zeros(nclusters, 1);
    selected_centroids = zeros(nsamples, 1);
    parfor j = 1:nsamples,
        % Search the closest centroid for each data point...
        distances = centroid_normals + normals(j) - 2 * (centroids * data(j, :)');
        [distance, selected] = min(distances);
        selected_centroids(j) = selected;
    end
    for j = 1:nsamples,
        % ...and accumulate the data point to the selected new centroid.
        selected = selected_centroids(j);
        new_centroids(selected, :) = new_centroids(selected, :) + data(j, :);
        histogram(selected) = histogram(selected) + 1;
    end
```

The main difference with the previous implementation is that the function calculates the closest centroid index and accumulates the new centroids in two separated loops. The first loop uses the instruction **parfor** instead of a standard **for**. This instruction executes the loop concurrently, so that, the iterations are divided between the worker processes of the parallel pool. The **parfor** has some constrains compared to a regular **for** and **Matlab** automatically generates an exception when the code cannot be processed in parallel. The use of **parfor** has the advantage that it makes concurrency almost transparent to the programmer. However, it has the drawback that lacks flexibility. For instance, **Matlab** usually makes as many copies of the matrices used inside the **parfor** as worker processes used. Therefore, **parfor** usually requires more memory than the memory actually needed.

In order to use the instruction **parfor**, a parallel pool has to be initialized. In an standard machine, the environment is initialized using the commands **matlabpool open <number_of_processes>** and **matlabpool close**. In the cluster, this commands are implemented using the instructions **POOL_OPEN** and **POOL_CLOSE**. **POOL_OPEN** create as many workers as processes assigned to the parallel environment of the job while **POOL_CLOSE** closes the parallel pool (i.e. stops the **Matlab** workers).

To use **POOL_OPEN** and **POOL_CLOSE**, **Matlab** needs to use the Java Virtual Machine. Therefore, the argument **-nojvm** cannot be used while calling **Matlab**.

For instance, to launch the multiprocessing k-Means algorithm we can create the following **Bash** script:

Listing 3.12: **Bash** *script file* `kmeansTutorialMP.sh`.

```
#!/bin/bash
export MATLAB=/etc/matlab/
export PATH=${MATLAB}:${PATH}
$MATLAB/bin/matlab -singleCompThread -nodisplay -nosplash -nodesktop -r "\
POOL_OPEN;\
load('$1', 'samples');\
load('$2', 'centroids');\
clusters = kmeansMPTutorial(samples, centroids(1:$3, :), 1000);\
save('$4', 'clusters');
POOL_CLOSE;\
exit;"
```

The code which uses multiprocessing must be always between the **POOL_OPEN** and **POOL_CLOSE**. Also, **Matlab** needs the Java Virtual Machine to run the parallel toolbox, so that, the call does not uses the **-nojvm**. Then we can select the number of processes used by the **Matlab** code using the parallel environment parameter of the **qsub** command:

```
[1234567@dccclus matlab_kmeans]$ qsub -cwd -V -q long.master.q \
-l mem_token=1.0G,mem_free=1.0G -pe pthreads 4 kmeansTutorialMP.sh \
data/samples/samples_100K_128_0000.mat data/clusters/seeder_100K_128_0000.mat \
16 data/clusters/clusters_100K_128_0000.mat
Your job 220882 ("kmeansTutorialMP.sh") has been submitted
```

Like in the **Python** version, we can launch several **Matlab** jobs using different number of processes in order to evaluate the computational cost of the algorithm as different number of processes are used. The computational cost per iteration can be seen in table 3.2. The main difference is that the number of processes in **Matlab** is limited to 12. Therefore, even more processes are allocated in the parallel environment call, only twelve will be used and the remaining slots will not be used. There is also a significant difference in the computational cost, but **Matlab** and **Python** experiments are not using exactly the configuration, so that, it is not possible to directly compare both implementations using these results. Besides the code must be further optimized, specially in the Python case, in order to fairly compare both languages.

Number of processes	1	2	4	6	8	10	12
Iteration	7.55	4.25	2.40	2.34	2.03	1.38	1.33

Table 3.2: Runtime of the k-Means algorithm and time per iteration using different number of processes in **Matlab**.

3.4 Message Passing Interface implementation

Using a multiprocessing approach to run the algorithms has the advantage that processes are not limited to run in the same machine, so that an algorithm is not limited to the maximum resources available in a single node machine. This is important to surpass the 32 processes limit of the cluster and, most importantly, to be able to deal with larger amounts of data. In order to run an algorithm into multiple machines, we need to communicate the different processes to coordinate their actions and to send data between them. In our case, the cluster is configured to use the MPI framework to communicate the processes.

Since **Matlab** is not fully integrated with the GE of the cluster and it cannot use the parallel environments **mpich** and **mpi**, we are only going to show an example using **Python**.

The package **mpi4py** allows to easily use the MPI framework to communicate the processes. In this version of the k-Means algorithm multiple processes which can be on different nodes are going to separately processes the samples in order to speed up the algorithm. Each process is only going to load and process a portion of the samples, i.e. a process only manages a certain amount of samples but it does not has access to the full set of samples. The actions of the processes is managed by a main process which coordinates the working processes and manages cluster centroids. The following script initializes the MPI k-Means algorithm:

Listing 3.13: main `Python` *script* `kmeans_mpi_tutorial.py`

```
#!/bin/env python
import numpy, cPickle, sys
import kmeansTutorial as tut
from mpi4py import MPI

if __name__ == '__main__':
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank() # MPI identifier of the process.

    # Load the list of sample files loaded by each process.
    f = open(sys.argv[1], 'r')
    mpi_samples_filenames = f.readlines()
    f.close()

    # Load the samples of the current process.
    if sys.argv[1].count('/') > 0:
        route = sys.argv[1][:sys.argv[1].rindex('/') + 1]
    else:
        route = ''
    samples = tut.load(route + mpi_samples_filenames[rank].replace('\n', ''))
    # Only the main process loads the seeds.
    if rank == 0:
        seeds = tut.load(sys.argv[2])
    else:
        seeds = None

    # Call the k-Means algorithm.
    cluster = tut.kmeansMPI(comm, samples, seeds)
    # Save the obtained clusters to disk.
    if rank == 0:
        tut.save(sys.argv[3], cluster)
```

The script first gets the MPI object with `comm = MPI.COMM_WORLD`. This object is used to get information about the MPI framework, to synchronize the different processes and to communicate the processes by merging objects from different processes, broadcasting an object to all processes or enable the direct communication between processes. In the script, we only use the function `Get_rank()` to get the identifier of the process. This identifier ranges from 0 to the number of processes of the parallel environment. Therefore, we use this identifier to select which chunk of samples the process must load. It is also used to identify the main process, so that, only process with rank 0 loads the seeds and saves the resulting clusters to disk.

The function `kmeansMPI` implements the MPI version of the algorithm. The algorithm uses the same functions `calculateCentroids` and `updateCentroids` of the sequential version of the algorithm with the main difference that some parts are only executed by the process with rank 0:

Listing 3.14: Function `kmeansMPI` *at* `kmeansTutorial.py`

```
def kmeansMPI(comm, data, seeds):
    mpi_size = comm.Get_size() # Number of MPI processes.
    mpi_rank = comm.Get_rank() # Identifier of the current process.
    nsamples, ndimensions = data.shape
    begin_time = time.time()

    centroids = seeds if mpi_rank == 0 else None
    normals = (data ** 2).sum(axis=1)

    time_first_iteration = time.time()
    for iteration in range(1000):
        # Broadcast the centroids to each process
        centroids = comm.bcast(centroids, root=0)
        new_centroids, histogram = calculateCentroids(centroids, data, normals)

        # Gather the new centroids and histograms from all
        # processes into the 0 process.
        new_centroids = comm.gather(new_centroids, root = 0)
        histogram = comm.gather(histogram, root = 0)

        solution_found = False
        if mpi_rank == 0: # ONLY PROCESS WITH IDENTIFIER 0
            # Collapse the results into a single object.
            for r in range(1, mpi_size):
                new_centroids[0] += new_centroids[r]
                histogram[0] = map(lambda x, y: x + y, histogram[0], histogram[r])
            new_centroids = new_centroids[0]
            histogram = histogram[0]

            update = updateCentroids(centroids, new_centroids, histogram)
            elapsed_time = time.time() - begin_time
            info = (elapsed_time, iteration + 1, update)
            print "[ %09.3f ] Iteration %06d: Maximum update distance = %e" % info
            # Stop when the distance between centrods is zero.
            if update == 0:
                solution_found = True

        # Broadcast the finish flag to all nodes
        solution_found = comm.bcast(solution_found, root=0)
        if solution_found:
            break

    if mpi_rank == 0:
        average_time = float(time.time() - time_first_iteration) / float(iteration + 1)
        info = (time.time() - begin_time, average_time)
        print "[ %09.3f ] Average time per iteration: %f" % info
        return centroids
```

The algorithm uses the commands **`comm.bcast`** and **`comm.gather`** to send information between processes. The command **`comm.bcast`** *broadcasts* the centroids from process with rank 0 (i.e. the main process) to all processes of the MPI framework. The command **`comm.gather`** does the contrary and it *gathers* the objects **`new_centroids`** and **`histograms`** from all processes and returns them into a list in the main process. Both commands freeze the process until it is executed by all processes, so that, they also synchronize the execution of the different processes. Once the **`new_centroids`** and **`histograms`** are gathered, the main process merges them and updates the cluster centroids. The final broadcast is used to finish all processes when the algorithm has converged, otherwise only the main process will

finish while all working processes will be frozen waiting that the main process broadcasts the centroids.

To use MPI, the python script **kmeans_mpi_tutorial.py** must be run using **mpirun**. The following command can be used to launch MPI jobs to the cluster:

```
[1234567@dccclus python_kmeans]$  qsub -b y -cwd -V -q long.master.q \
-l mem_token=1.8G,mem_free=1.8G -pe mpich 50 mpirun ./kmeans_mpi_tutorial.py \
data/mpi/list_100K_64.data data/mpi/seeder_1K_64_0000.data \
data/mpi/cluster_1K_64_0000.data
Your job 294555 ("mpirun") has been submitted
```

This **qsub** will create a job which needs 50 slots and 90Gb of memory. If we check the status of the **long.master.q** queue where we launched the job, we will obtain the following information:

```
[1234567@dccclus python_kmeans]$ qstat -q long.master.q
job-ID  prior   name       user         state submit/start at     queue                          slots ja-task-ID
-----------------------------------------------------------------------------------------------------------------
 294555 0.60500 mpirun     1234567      r     01/13/2013 16:26:18 long.master.q@compute-0-14.loc    50
```

The queue which the **qstat** command returns as the location of the job is where the first job has been launched, in this case in node **compute-0-14**. However, checking the status of all **long.master.q** queue nodes, we can see that the job is distributed between two different machines:

```
[1234567@dccclus python_kmeans]$ qstat -q long.master.q -f
queuename                      qtype resv/used/tot. load_avg arch          states
---------------------------------------------------------------------------------
long.master.q@compute-0-13.loc BIP   0/18/32        16.87     linux-x64
 294555 0.60500 mpirun     1234567      r     01/13/2013 16:26:18    18
---------------------------------------------------------------------------------
long.master.q@compute-0-14.loc BIP   0/32/32        29.97     linux-x64
 294555 0.60500 mpirun     1234567      r     01/13/2013 16:26:18    32
---------------------------------------------------------------------------------
long.master.q@compute-0-15.loc BIP   0/0/32         0.00      linux-x64
---------------------------------------------------------------------------------
long.master.q@compute-0-16.loc BIP   0/0/32         0.00      linux-x64
---------------------------------------------------------------------------------
long.master.q@compute-0-17.loc BIP   0/0/32         0.00      linux-x64
---------------------------------------------------------------------------------
long.master.q@compute-0-19.loc BIP   0/0/32         0.00      linux-x64
```

The **qsub** stores the output of the algorithm in the file **mpirun.o294555** and file **mpirun.po294555** contains the node where each process is running. Like in the previous versions of the algorithm, we can launch the algorithm using different number of processes to check the evolution of the computational cost of the algorithm.

Number of processes	16	32	64	128
Iteration	375,29	336,53	185,82	96,33

Table 3.3: Runtime of the MPI version of the k-Means algorithm and time per iteration using different number of processes.

Although the MPI version of the algorithm is able to deal with more samples, we decided that the experiment is small enough to be carried out into a single node when needed. The experiment has 12.8 million samples of 128 dimensions and the k-Means algorithm calculates 1024 clusters. The computational cost per iteration of the algorithm

can be found in table 3.3. As we can see, the computational cost nearly halves as the number of processes is doubled in all cases but the first. In the 16 and 32 slots cases all job processes were running in the same node, so the 32 slots job were using hyper-threading instead of actual cores. This may be the reason why the computational cost only slightly reduces.