Intel Educational Mindshare Initiative in Artificial Intelligence

AI and Predictive Analytics in Data-Center Environments

http://dcai.bsc.es

Session 1 – Computing Environments

In this "Hands-On" we will do some exercises showing the concepts of **Performance**, **Parallelism**, **Virtualization** and **Containers**, seen in the first session Computer Environments.

Important: All the following exercises are based in **Debian/Ubuntu**, **Bash** and **Python**, as a common operating systems and scripting interfaces found in many computing servers and data-centers. The same experiments can be also repeated in MacOS and Windows (using the latest WSL), with possible limitations, but this does not mean that there aren't other ways to achieve the same in such systems.

Exercise 1 – Performance

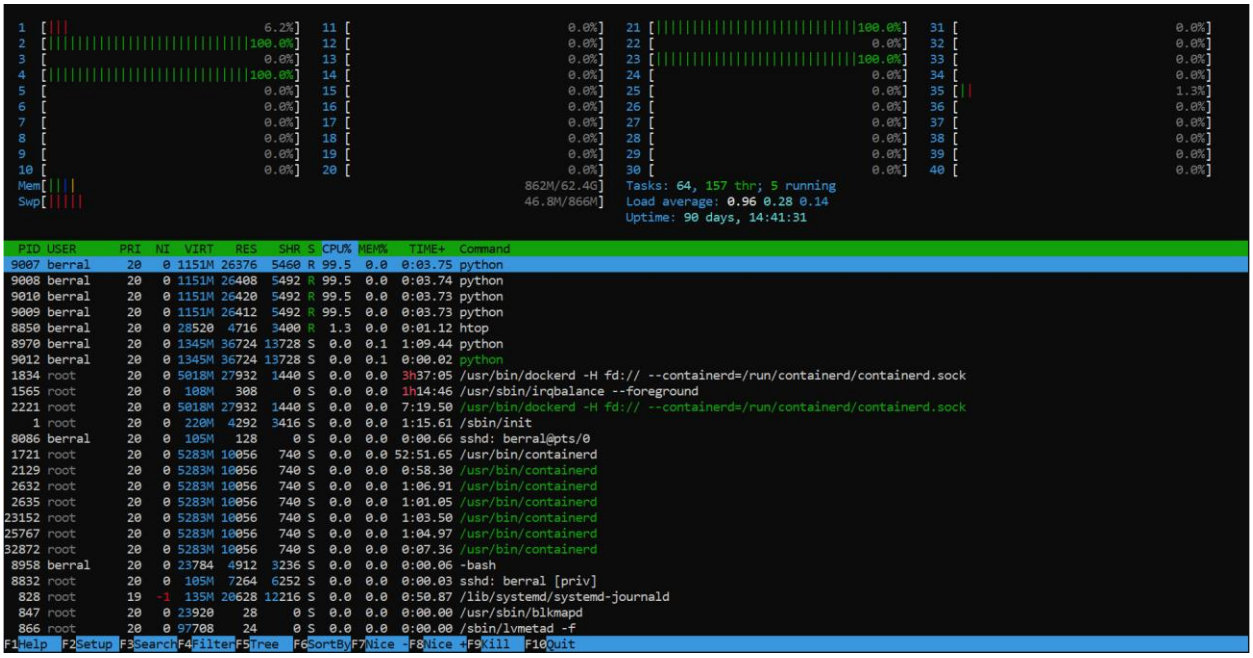
In this part we will see some applications to monitor our systems, and do some exercises to see how resources are used by applications, also how applications compete for resources among them.

Monitoring our machines

There are many applications to check the use of resources in our machine, see how many applications are in our system, and how much resources use each one. Here we recommend use **htop**, as an expanded version of the classical **top** of Unix systems. The application htop shows the usage for the different CPUs, also the list of applications and their CPU and Memory usage, with their Process ID, and time running. To install htop, you can use the package manager directly:

```
$ sudo apt install htop
```

Once starting htop from the command line, we see something like this:



On top we have a “level bar” for each CPU, indicating its level of usage from 0% to 100%. Immediately below we see a level bar for the usage of Memory, and the usage of Swap space (this is, when memory is full and pages must be sent to the disk to make room to other pages requested by the current running application, these are sent to “swap”). Here we see a machine with 40 CPUs (4 of them currently in use), aproximately 64GB of RAM and 866MB of Swap. In the inferior part we see the processes running, some of a connected user (who is running python scripts), and some more of the system running daemons and system applications. To exit just press “q”.

During this hands-on we will do exercises consisting on running things on machines, so we’ll visit htop to see what’s happening in the system.

Stressing our Machine and setting Quotas

Now we will see different common ways to affect the performance of our system. We will see how to **stress** a CPU, sending operations to have it fully occupied, also how to set up CPU limits for applications with **cpulimit**, setting up a maximum CPU quota on our application,

Stressing the CPU

The application **stress** is used to “just occupy” CPUs, to benchmark and test environments. When we need to see what would happen if our system had half the CPUs in use, or if an application had to compete with another application demanding all the CPU, we can lauch **stress** indicating how many CPUs we want to “occupy”. We can obtain it from the package manager:

```
$ sudo apt install stress
```

If we want to stress 1 CPU we can just run the following, while running **htop** in another terminal.

```
$ stress --cpu 1
```

We can stop "stress" by hitting CTRL+C. Stressing all the computer may make it unusable, so we must be careful when stressing it. We can set a **timeout** in case that we lose control of the terminal when overwhelming the machine:

```
$ timeout 10s stress --cpu 1
```

Now, let's create a script called "**fib.sh**" that will contain a BASH script computing the Fibonacci series. The script is the following (remember to give it execution permissions):

```
# fib.sh
a=0; b=1;

for (( i=0; i<$1; i++ ))
do
    echo -n "$a ";
    fn=$((a + b));
    a=$b; b=$fn;
done
echo ""
```

Then run it as follows, where the first parameter is the limit number for the sequence:

```
$ ./fib.sh 9999
```

With a high enough parameter, in htop we'll see how a CPU is consumed. We can measure the time it takes to compute it using **time**:

```
$ time ./fib.sh 9999
```

Now, we can stress all our CPUs before, and launch fib.sh at the same time, then check how much it takes to finish:

```
$ timeout 10s stress –cpu 32 & time ./fib.sh 99999 &
```

We will se that it took more time for fib.sh to run, as it had to compete for CPU against the stress processes. You can also try to launch several fib.sh in parallel (add "&" symbol at the end of each command), and see in htop how they compete for CPU.

Exercise 2 – Parallelism

In this part we will do some exercises on parallelizing applications on our system, and see how performance and time changes according to resources demand and availability. We will go through some examples using python and CPU parallelism libraries, and see how functions improve when prepared for parallelism.

Example using a Python script

For this example we're creating a function

First of all let's import **Numpy** and the package **multiprocessing** which handles process-level parallelism in python (as by design Python does not support thread-level parallelism in a simple way), also **datetime** to measure the execution time:

```
import numpy as np
import datetime
from multiprocessing import Pool

np.random.seed(25740565)
```

Now, we will create two vectors with some random numbers as data-set:

```
X = np.random.randn(20000)
Y = np.random.randn(20000)
```

Let's define now a "hard to execute" function that can be parallelized, and see what happens when we serialize it versus when we allow parallelism. The function will receive a Vector and a Scalar, and will sum the products of each element of the vector with the scalar:

```
def scalar_sum(X, y):
    return sum( x * y for x in X )
```

Our objective will be to use this function to make the scalar sum of X and Y, passing the vector X and each element of vector Y through the function, then add the result. If we do this serialized (also while computing the time before and after) we will have this:

```
time_start = datetime.datetime.now()
result = sum(scalar_sum(X, y) for y in Y)
time_end = datetime.datetime.now()
print(result)
print(time_end-time_start)
```

Then, we will execute a parallelized version, by indicating 4 CPUs and **mapping** our function to each tuple. We will use the function **Pool** to allocate 4 CPUs for us:

```
p = Pool(4)
```

And now, using the function **map**, we will apply our function into the following data. But before doing this, we need to modify the function a little bit, to pass everything as a single parameter (this map function requires one only input, even if it is a tuyple with our multiple inputs).

```
def scalar_sum_2(Xy):
    return sum( x * Xy[1] for x in Xy[0] )
```

Now, we prepare the data as a list of tuples `(X, y)` to be processed, and we can map our modified function to each tuple:

```
time_start = datetime.datetime.now()
tuples = ((X,y) for y in Y)
result = sum( p.map( scalar_sum_2, tuples ))
time_end = datetime.datetime.now()
print(result)
print(time_end-time_start)
```

If we open **htop** in another terminal while executing the tests, we'll see the used cores during each execution. We can see that the time was reduced by approximately 4th of the original time, counting on the minimum overhead of the system managing different concurrent processes.

Another example

Let's define another function to test the distribution of data processing. We define now a function that does a pairwise sum between two vectors, and we will use the same generated random data for it:

```
def pairwise_sum(X, Y):
    return sum( x*y for x in X for y in Y )
```

Now we run it sequentially, and check in **htop** how a single core is fully used while **pairwise_sum** is running:

```
time_start = datetime.datetime.now()
result = pairwise_sum(X, Y)
time_end = datetime.datetime.now()
print(result)
print(time_end-time_start)
```

And now, we apply a parallel version mapping the function to the inputs. Notice that we need to change again the function to allow one single parameter), then prepare the data as tuples of pairs to be multiplied. And finally add the results:

```
def pairwise_sum_2(Xy):
    return sum(x * Xy[1] for x in Xy[0])

p = Pool(4)

time_start = datetime.datetime.now()
```

```
tuples = ((X, y) for y in Y)
result = sum(p.map(pairwise_sum_2, tuples ))
time_end = datetime.datetime.now()
print(result)
print(time_end-time_start)
```

From here we see that by preparing the data properly we can easily parallelize its processing. With all of this we just see that some serialized operations will improve their execution time by mapping functions.

Exercise 3 – Virtualization

In this part we're installing a VM Hypervisor (the program managing and runnng virtual machines) and an application to retrieve virtual machine images, with base systems and frameworks already installed, from Internet repositories.

VirtualBox & Vagrant

VirtualBox is a hypervisor application that allows us to create, manage and run virtual machines. It comes for many systems like Linux, MacOS and Windows, and lets us to create VMs for using other operating systems inside, or just run isolated systems. Also Vagrant is an application that allows us to manage VM images, download them from Internet repositories or even get updated versions.

Installing VirtualBox & Vagrant
The first step is installing VirtualBox and Vagrant in our system. You can install **Vagrant** & **Virtualbox** using the package manager (e.g. apt for Debian/Ubuntu):

```
$ sudo apt install virtualbox vagrant
```

It is possible that virtualbox asks additional permissions and drivers for properly running. By default, the package is enough.

Creating and initializing the virtual environment
Using the VirtualBox Manager we can create new virtual machines and VM images, but now we're going to use Vagrant to download an already prepared VM image containing Debian. By using Vagrant we don't need to interact directly with VirtualBox, as it has commands to retrieve and manage the VMs with VirtualBox in the background.

Before creating the VM we need to define the location where our virtual environment will be donwloaded and deployed. E.g. we create the folders "virtual_envs" and "IntelAI-BigDL" in our "home" directory, and go in:

```
$ mkdir -p ~/virtual_envs/IntelAI-BigDL
$ cd ~/virtual_envs/IntelAI-BigDL
```

Once we are inside the folder we can initialize the virtual environment with **vagrant init** indicating the image you want to use for the VM:

```
$ vagrant init debian/stretch64
```

The identifier “debian/stretch64” is obtained from the Vagrant repository website, containing a 64bit Debian Stretch version. The init command initializes the environment, indicating that that folder will contain that image.

Starting the virtual environment
Now, we can start the VM using **vagrant up**, and the first time we run it Vagrant will download the image and leave it ready in the directory.

```
$ vagrant up
```

Now we have the VM image downloaded and an instance of it (our VM) running in our system.

Connecting to the VM
As the VM is already started, we can connect to it from our local machine through SSH using the command **vagrant ssh**:

```
$ vagrant ssh
```

Now we are INSIDE the VM, and everything we do here is isolated from our “physical system”. As a VM, changes that we do inside are persistent, and when shuting down the machine, changes will remain stored in our copy of the VM image.

To exit the session and return to our local shell simply type **ctrl-d** or type **exit**, and we are back on the “physical domain”.

Stopping the VM
We can stop the VM by just entering and shutting it down with a **halt now** command, as in any Linux system, or we can stop it from outside the VM using **vagrant halt**:

```
$ vagrant halt
```

The **vagrant halt** command will send a halt command to the VM. It also can be suspended or powered off directly. But better shut it down gracefully. Once stopped, we can restart the VM again using **vagrant up**.

Deleting the VM
We can decide to remove the VM if we’re not using it anymore, by deleting it by calling **vagrant destroy**:

```
$ vagrant destroy
```

That will delete the VM files and the VM image, with any modifications and files that we created inside. Notice that as a “machine”, the VM can be configured with a network connection to put and retrieve files from the VM. VirtualBox creates by default a virtual network where VMs are connected, so you can use **scp** (ssh + copy) to put and pull files from the VM, or even share a “physical directory” with the VM to share and preserve files. Check the documentation here on Shared Folders.

Exercise 4 - Containers

In this part we're installing Docker, a Container Manager, already able to retrieve container images, with base systems and frameworks already installed, from Internet repositories.

Docker as Container Engine

Docker is a containers engine that allows us to create containers, retrieve them from Internet repositories also manage them.

Installing Docker
The first step is installing Docker in our system. We can use the package manager for this:

```
$ sudo apt install docker.io
```

The docker group
Once installed we need to create a group for docker, to be used by container instances with the proper permissions. The thing is that the Docker daemon binds to a Unix socket instead of a TCP port, which is by default owned by **root**, and other users can only access it using **sudo**. Because of this, the Docker daemon itself always run as **root**, requiring superuser permissions for anyone to use it. But, if you want to avoid having to type **sudo** before every Docker command, you can create a group called **docker** and add all required users to it, so the Docker daemon grants access to all users in this group to the socket it uses.

We can create the group as any other Unix group:

```
$ sudo groupadd docker
```

Once the group is created we can add users to it as we typically do with Unix users: (just replace `<USER>` with the corresponding username)

```
$ sudo usermod -aG docker <USER>
```

After a user is added we might need to log out and log in again, so group memberships are updated in the new session.

Important: The docker group grants privileges equivalent to the root user, and that may be a security problem for many systems. For details on how this impacts security in your system, see Docker Daemon Attack Surface. There are alternative container managers to Docker with limited user privileges, like Singularity, with no superuser privileges to run or to its containers, but with limited capabilities due not being superuser.

Creating and running a container
We can retrieve and start an available container on the repository using the **docker run command**. This command pulls the indicated container image, allocates the required resources and starts an instance of the container. Docker maintains a local repository of images, and it first looks in our local repo before checking the Internet repos. If it is not able to find the specified image in local, it will look for it in the available image repositories. E.g. we are (downloading, the first time, and) initiating an instance of the

"latest Debian" in the repository, named "debian-container", initiating an "interactive terminal" (opening a "/bin/bash" terminal in this case), and ordering to "remove" the instance once we close the interactive session:

```
$ docker run -it --rm --name="debian-container" debian:latest
/bin/bash
```

This example will start an instance of Debian and connect us to it, then when leaving it will destroy the instance (not the original image, being stored in the repository). Notice that any change done into the container will be lost when removing it. Luckily for us, containers usually mount some of our directories into the container, so we can share and save our files there.

The options shown here are the most usual when running containers. This is what they do:

- `-i`: Indicates the daemon to keep the Standard Input of the container open so you can interact with it via command line.
- `-t`: Allocates a pseudo-TTY to the container.
- `--rm`: Automatically deletes the container when it stops.
- `--name <name>`: Assigns the name <name> to the container.

The **/bin/bash** argument indicates what will be executed on container startup. If instead of starting a bash session, we want to run a script or application inside of it directly, we can indicate it there. Notice that the application must be accessible to the container, and this means installed by default in the container image, or accessible in the directories shared with the container.

Once we are connected to a container via command line, we can exit it normally with **ctrl-d** (stopping the container if the shell is the only thing running on it), using exit if we are in a bash session, or detach from it keeping the current shell by typing **ctrl-p ctrl-q**.

Listing all existing containers & images

If we start containers in a non interactive mode, just running them and indicating which script or command we want to run instead of bash, the container will just run in background. To list all running containers we can use the command **docker ps**:

```
$ docker ps
```

This will show the containers running. If we want to list **all** the containers in our machine (the running ones and the already stopped or finished) we can use the **–a** option:

```
$ docker ps -a
```

We can also list the container images available in our local repository, with the ones we downloaded previously and the ones we created from a "container receipe" (instructions to download a base system and programs to be installed, creating a new container image). We can use the **docker images** command (an alias of **docker image ls**):

```
$ docker images
```

Running commands & scripts on containers

Once we have a container running (here in a non interactive session, just in background), we can send command and scripts to run from our command line, by using the **docker exec** command. For example, if we wanted to start a new shell in a running container we can order the container to execute /bin/bash:

```
$ docker exec -it <container-name> /bin/bash
```

Notice that as we are running bash as an "interactive session", we use the **–it** option. If we were launching a non interactive application or a non interactive bash script we would not need the **–it**.

Attaching and detaching from containers
If we have a container running with a shell available, we can connect to it using the **docker attach** command:

```
$ docker attach <container-name>
```

This shell should have been started with the container or using docker run or exec, so it will keep the options we indicated when running **docker run** or **docker exec**. E.g. if we didn't add **-t** it will not have a TTY when we attach to it. Also, we can detach from a container using **ctrl-p ctrl-q**, or closing the shell with **ctrl-d** or **exit**.

Stopping a container
To stop a running container we can just use the **docker stop** command:

```
$ docker stop <container-name>
```

And remember that if we ran the container with the **--rm** option, stopping it will also delete the container instance. Additionally, we can use the command **docker start** to start again a stopped container.

Deleting a container
Once we have a container stopped, we can delete it using the **docker rm** command:

```
$ docker rm <container-name>
```

After deleting it, we can create new fresh instance of the container image using the previous command **run**.

Reminder

All these exercises are to show the basic concepts of those technologies. To have a better understanding and discover all the capabilities of those methods and applications, check the reference manuals and play with new examples.