Distributed Computing

Step 1: Create a new user

Create new user accounts with the same username in all the machines to keep things simple. The following command is needed to run in all nodes and host computers.

```
$ sudo adduser username
```

To give access to the newly created user account as sudo, we have to run the following command.

```
$ sudo usermod -aG sudo username
```

And right after that, log in with your newly created account

```
$ su - username
```

Step 2: Configure your `hosts` file

To add multiple nodes in a host computer, we have to write the name and IP Addresses of all the node's computers manually in the ***/etc/hosts*** file.

```
username@master:~$ cat /etc/hosts
username@master:~$ sudo vim /etc/hosts
```

```
127.0.0.1          localhost
127.0.1.1          master
192.168.0.222      worker1
192.168.0.225      worker2
```

Step 3: Install SSH

Next, we need to install an SSH server on all the systems using the following command:

```
username@master:~$ sudo apt install openssh-server
```

Step 4. Setup password-less access

Next, we need to enable password-less SSH access. To do so, we have to run the following command only in the master node.

```
username@master:~$ ssh-keygen -t rsa -b 4096
```

This command will create two files in your ~/.ssh directory: id_rsa and id_rsa.pub. The latter is the *public key*. The contents of this file need to be copied to the ~/.ssh/authorized-keys file on the *remote* machine. From the command line, you may want to first use cat to display the file contents.

```
username@master:~$ cat ~/.ssh/id_rsa.pub
```

```
--
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAACAQDSg3DUv2O8mvUIhta2J6aoXyq71Q9Ld0Ez1exO1M+OGO
NH...cvzQ== username@master
```

Now, add the generated key to each of the other computers. In our case, we have the worker1 and worker2 machines.

```
username@master:~$ ssh-copy-id worker1 #ip-address may also be used
```

or,

```
username@master:~$ ssh-copy-id worker2 #ip-address may also be used
```

Alternative way to add the key:

```
username@master:~$ scp ~/.ssh/authorized_keys worker1:~/.ssh/
username@master:~$ scp ~/.ssh/authorized_keys worker2:~/.ssh/
```

scp is a very useful program for copying files over SSH.

Then login to the remote machine (you will be prompted for a password),

```
username@master:~$ ssh worker1     #ip-address may also be used
```

```
Welcome to Ubuntu 20.04.2 LTS (beaver-osp1-ellaria X31) (GNU/Linux
5.0.0-1065-oem-osp1 x86_64)
```

or,

```
username@master:~$ ssh worker2    #ip-address may also be used
```

```
Welcome to Ubuntu 20.04.2 LTS (beaver-osp1-ellaria X31) (GNU/Linux
5.0.0-1065-oem-osp1 x86_64)
```

If everything worked, you will be logged in without being prompted for a password.

Step 5. Install MPI

Next, we need to install MPI on all the machines (master, worker1, worker2,...). Message Passing Interface (MPI) is the standard communication protocol used for distributed parallel processing. We have to run the following command in all the master and worker nodes.

```
username@master:~$ sudo apt install libopenmpi-dev
```

Step 6. Test MPI

With MPI now installed, we next need to make sure it is working. There are two parts to this. First, let’s make sure the MPI commands are available:

```
username@master:~$ mpic++
```

```
g++: fatal error: no input files
compilation terminated.
```

This command confirmed that we can launch commands using mpirun.

```
username@master:~$ mpirun -np 2 hostname
```

```
master
master
```

Next comes the real test: verifying that MPI is able to launch processes across the network interface. Run

```
username@master:~$ mpirun -np 5 -host worker1:2 -host wroker2:2 -host master:1
hostname
```

```
worker1
worker1
worker2
worker2
master
```

If you get a similar output, congratulations, you now have a basic cluster capable of launching MPI jobs on multiple systems!

Step 7. Set up the network file system

There is a reason we have so far used the hostname command: it is available by default on all systems. With MPI, it is important to remember that we are essentially only using a network connection to allow multiple running jobs to communicate with each other. Each job is however running on its own computer, with access to its own hard drive.

This specifically means that the command to launch needs to exist on all computers and in the same location. Let's say you put together a simple python code in ***print.py***:

```
print("Hello world!")
```

We can compile and run the program *locally* using:

```
username@master:~$ mpirun -np 3 python3 ./print.py
Hello world!
Hello world!
Hello world!
```

However, if we try to run the program *remotely*, it fails:

```
username@master:~$ mpirun -np 3 -host worker1:3 ./print.py
--------------------------------------------------------------------------
mpirun was unable to launch the specified application as it could not
access
or execute an executable:
```

This is because there is no ***print.py*** executable in the home directory on the worker1 harddrive. We could use scp or rsync to copy it there:

```
username@master:~$ rsync -rtv ./print.py worker1:
sending incremental file list
print.py
sent 122,887 bytes  received 35 bytes  245,844.00 bytes/sec
…
```

```
Then we can run the file on the worker1 drive.
```

```
username@master:~$ mpirun -np 3 -host worker1:3 ./print.py
```

```
Hello world!
Hello world!
Hello world!
```

But, as you can imagine this can get quite annoying especially if the program produces results that need to be copied back for analysis. A much better solution is to set up a network drive. This is surprisingly easy on Linux. You first need to install **NFS (network file system)** kernel and common programs:

```
username@master:~$ sudo apt install nfs-kernel-server nfs-common
```

You next need to create a mount point for the shared directory. I decided to share the entire home directory on mini. Now, since the paths need to be identical on all MPI processors, I created a symbolic link on the master that points to my home directory,

```
username@master:~$ sudo ln -s /home/username /nfs
username@master:~$ ls -la /nfs
```

```
lrwxrwxrwx 1 root root 21 Sep 21 13:48 nfs -> /home/username/
```

Next, again on the server, add the following line to /etc/exports

```
username@master:~$ sudo vim /etc/exports
```

```
…

/home/username    worker1(rw)
```

This command gives worker1 read-write access to the specified folder.

Now, in the remote client (worker1), create a directory named /nfs. This is for creating the /nfs mount point on client machines:

```
username@worker1:~$ sudo mkdir /nfs
```

Next update /etc/fstab to include the line at last:

```
username@worker1:~$ sudo vim /etc/fstab
```

…
…

```
master:/home/username       /nfs      nfs defaults 0 0
```

To process fstab, run:

```
username@worker1:~$ sudo mount -a
```

If everything went well, you should be able to navigate to the folder and see the contents of your home directory on the server:

```
username@worker1:~$ cd /nfs
username@worker1:/nfs$ ls -la
```

```
total 220
drwxr-xr-x 80 user user    12288 Sep  7 14:17 .
drwxr-xr-x 25 root root    4096 Aug 27 06:19 ..
-rw-------  1 user user    186 Aug 19  2019
2019-08-19-18-54-17.002-VBoxSVC-22509.log
drwxr-xr-x  3 user user    4096 Apr 28 17:10 .altera.quartus
...
```

Finally, let’s see if this worked. Make sure to navigate to the /nfs path on the server, and next try to run the program,

```
username@master:/nfs$ mpirun -np 5 -host worker1:3 -host master:2 python3
./print.py
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

Congratulations! You have successfully set up a cluster of computers and ran a python program on the master node and the worker nodes combined.
