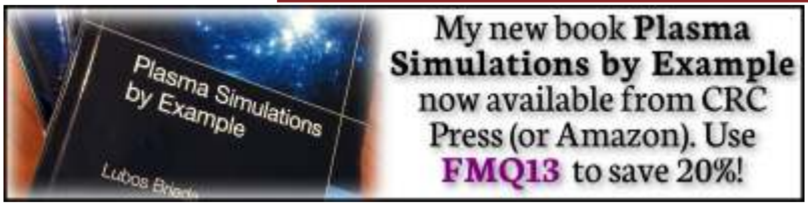
SETTING UP AN UBUNTU LINUX CLUSTER


Plasma Simulations by Example
Lubos Brieda
My new book Plasma Simulations by Example now available from CRC Press (or Amazon). Use FMQ13 to save 20%!

I recently realized that my existing Dell workstation featuring a 6-core i7-8700K CPU (12Mb cache, 3.7Ghz base, up to 4.7Ghz) with 32 Gb of RAM and an NVIDIA P1000 GPU
may not be sufficient to turn around an on-going analysis in a reasonable amount of time. I at first looked into upgrading the CPU, but the installed model already represented the peak the installed motherboard – and the power supply – could support. Hence it was time to grab another system. I settled on another Dell small form factor system (these are great, barely larger than a laptop) featuring a 10-core Intel Xeon W-1290 CPU (20 Mb cache, running nominally at 3.2Ghz but speeding up to 5.2Ghz) with 64 Gb of RAM, and another P1000 GPU. I also grabbed an 8-port Netgear gigibit Ethernet switch from NewEgg, along with some network cables. All that remained was to connect the two computers into a nano (literally as small as it can get) 2 node cluster. I put this post together to show you how to set up a similar cluster, but also to summarize what I did so I have a reference to review in the future if I decide to expand the cluster again. These instructions are for Ubuntu Linux as that is the O/S I was already using on the original system. These notes were just put together from memory so it's possible I missed some steps. Do not hesitate to leave a comment if something is missing or not working.

STEP 1. INSTALL UBUNTU

The computer already came with Ubuntu installed (this also knocks off around $100 off the Windows version), however you may want to reinstall the O/S to, for example, encrypt the hard drive.

STEP 2. CONNECT THE SWITCH

Next, connect the (two) computers to the switch. Here I made sure to use a high speed cable since apparently Ethernet cables are rated for different maximum bandwidth.

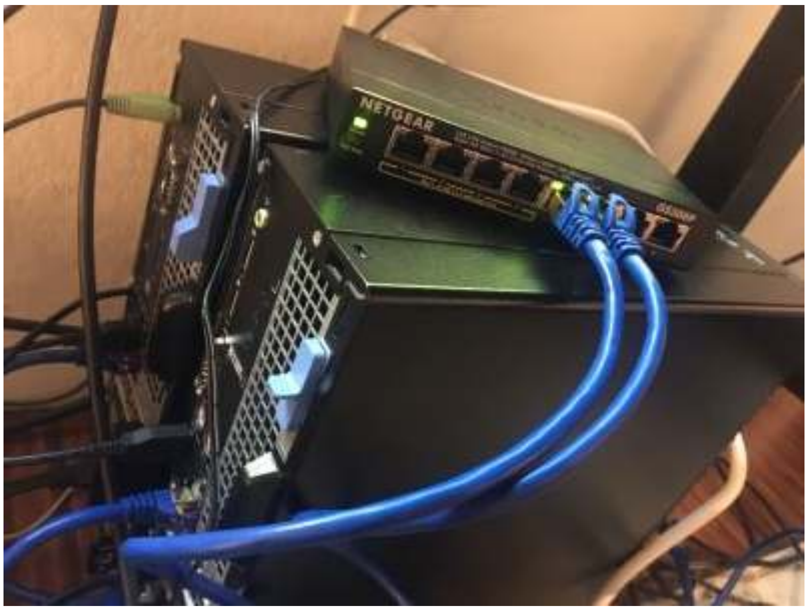
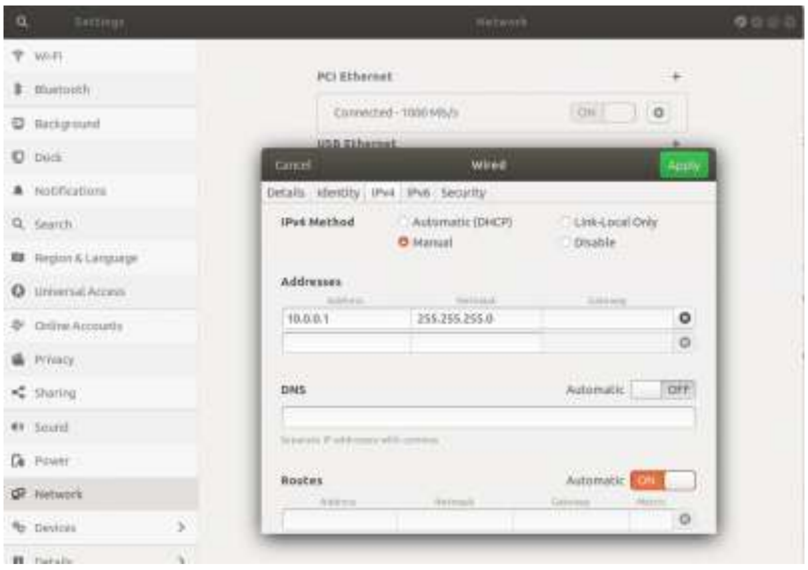


Figure 1. Two Dell small factor workstations connected into a mini cluster

STEP 3. SET UP NETWORK ADDRESSES

Now that the switch is connected, we need to assign IP addresses. I used to administer a Linux cluster back in my AFRL days, but I didn't actually set up the hardware. This was done by the vendor. This was my first time ever actually using a switch and as such, I had no idea if the IP addresses get assigned automatically or not. Perhaps there is a way to do it automatically, but I found that just setting a static IP works just fine. Here I just used the Ubuntu GUI interface. I assigned `10.0.0.1` on my original i7 system (called `mini`) and `10.0.0.2` on the new Xeon system.



PCI Ethernet: Connected - 1000 Mb/s, ON

IPv4 Method: Automatic (DHCP) [✓ Manual] Link-Local Only Disable

Addresses

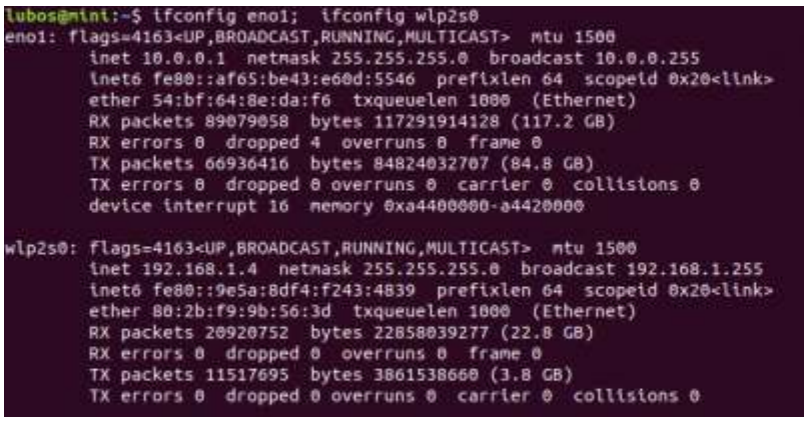
Address	Netmask	Gateway
10.0.0.1	255.255.255.0	

DNS: Automatic OFF

Routes: Automatic ON

Figure 2. IP address is assigned manually

We can check the assigned IP address using `ifconfig`. Figure 3 shows the IP address on the Ethernet interface (the switch) as well as the IP address assigned to my wireless connection.



```
lubos@mint:~$ ifconfig eno1;  ifconfig wlp2s0
eno1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.0.1  netmask 255.255.255.0  broadcast 10.0.0.255
        inet6 fe80::af65:be43:e60d:5546  prefixlen 64  scopeid 0x20<link>
        ether 54:bf:64:8e:da:f6  txqueuelen 1000  (Ethernet)
        RX packets 89079058  bytes 117291914128 (117.2 GB)
        RX errors 0  dropped 4  overruns 0  frame 0
        TX packets 66936416  bytes 84824032707 (84.8 GB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 16  memory 0xa4400000-a4420000

wlp2s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.4  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::9e5a:8df4:f243:4839  prefixlen 64  scopeid 0x20<link>
        ether 80:2b:f9:9b:56:3d  txqueuelen 1000  (Ethernet)
        RX packets 20920752  bytes 22858039277 (22.8 GB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 11517695  bytes 3861538660 (3.8 GB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Figure 3. We can check the assigned address using `ifconfig`

STEP 4. INSTALL SSH

Next, we need to install SSH server on both systems. This may already be installed, but if not, you will want to run

```
$ sudo apt install openssh-server
```

I am not going to get into the detail of SSH configuration, but you may at least want to review the configuration options for SSH and SSHD (the server deamon) in

```
$ more /etc/ssh/ssh_config
$ more /etc/ssh/sshd_config
```

If not already set by default, you may want to at least disable root login. You can make sure sshd is running using

```
user@mini:~$ sudo systemctl status sshd
● ssh.service - OpenBSD Secure Shell server
   Loaded: loaded (/lib/systemd/system/ssh.service; enabled; vendor preset: enabled)
   Active: active (running) since Fri 2020-08-14 22:16:59 PDT; 3 weeks 2 days ago
```

STEP 5. SETUP PASSWORD-LESS ACCESS

Next, we need to enable password-less SSH access. There is more info on this step at **this link** (and many others), but essentially, you want to run

```
$ ssh-keygen -t rsa -b 4096
```

and accept all default settings. It is generally recommended to use a password to encrypt the private key, but I did not do that here, since this key is used only on my internal network. 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

```
user@mini:~$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAACAQDSg3DUv2O8mvUIhta2J6aoXyq7lQ9Ld0Ez1exOlM+OGONH...cvzQ==
```

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

```
user@mini:~$ ssh 10.0.0.2
Welcome to Ubuntu 18.04.2 LTS (beaver-osp1-ellaria X31) (GNU/Linux 5.0.0-1065-oem-osp1 x86_64)
```

Highlight the entire `cat` output and copy to clipboard. Then use `vi` to append the entire `cat` output to the end of the file (use *i* to enter insert mode, then *[esc]* and *:wg[enter]* to write out)

```
user@xeon:~$ vi ~/.ssh/authorized_keys
[i], [right click / paste]
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAACAQDSg3DUv208mvUIhta2J6aoXyq7lQ9Ld0Ez1ex0lM+OGONH...cvzQ==
[esc], [:wq], [enter]
```

There are other ways of doing this using command line, but this visual approach is what works for me.

Next repeat by creating a new private key on the new system and copy its public key to the other machine. As you can imagine, doing so for a large cluster would be rather difficult. Therefore, you may want to just use the same private key on all systems. You can then just copy the `authorized_keys` file to all nodes,

```
user@mini:~$ scp ~/.ssh/authorized_keys 10.0.0.2:~/.ssh/
```

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

Finally, to avoid having to type in IP addresses, you may want to add the host names to your `/etc/hosts` files.

```
user@mini:~$ more /etc/hosts
127.0.0.1       localhost
127.0.1.1       mini
10.0.0.2        xeon
```

You can then test your remote login by using

```
user@mini:~$ ssh xeon
Welcome to Ubuntu 18.04.2 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 password.

Step 6. Install MPI

Next, we need to install MPI. Message Passing Interface is the standard communication protocol used for distributed parallel processing. If you are not familiar with MPI, I suggest you check out my online course on code parallelization or review Chapter 9 in my book on plasma simulations.

There are several implementations of the MPI standard, with the two most popular being MPICH and OpenMPI. I historically used MPICH, but for this project, found OpenMPI much easier to setup. Despite both systems running Ubuntu 18, the binary versions of MPICH installed off the Ubuntu distro `apt` server would not "talk to each other". I next tried to compile MPICH from source, but this resulted in compilation errors on one of the systems – again strange given both were running the same O/S (although this is not quite true, the new Xeon had the Ubuntu as installed by Dell, while I have since reinstalled Ubuntu from scratch on the older workstation). OpenMPI worked like charm.

```
user@xeon:~$ sudo apt install libopenmpi-dev
```

Step 7. 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:

```
user@xeon:~$ mpic++
g++: fatal error: no input files
compilation terminated.
```

Despite the above command failing due to not providing any input files, we at least confirmed that the `mpic++` compiler (just a wrapper for `g++`) is available.

Back on my older workstation, I next ran

```
user@mini:~$ mpirun -np 2 hostname
mini
mini
```

This command confirmed that we can launch commands using `mpirun`. Next comes the real test: verifying that MPI is able to launch processes across the network interface. Run

```
user@mini:~$ mpirun -np 2 -host 10.0.0.2:2 hostname
xeon
xeon
```

This particular syntax is specific to OpenMPI, with MPICH using slightly different command line arguments (check the documentation). Here we are specifying that we want to run the command `hostname` on host with IP address `10.0.0.2` and that this system has "capacity" of 2 computational slots (`:2`). With the command above, we launch two copies of `hostname` on the `xeon` node from the `mini` system. We can also run

```
user@mini:~$ mpirun -np 3 -host 10.0.0.2:2 -host 10.0.0.1 hostname
mini
xeon
xeon
```

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

Assuming you added the remote IP address to your `/etc/hosts` file, the above command is identical to

```
user@mini:~$ mpirun -np 3 -host xeon:2 -host mini hostname
mini
xeon
xeon
```

Now, since specifying hosts on the command line can get annoying, we can also create a hostfile,

```
user@mini:~$ more ~/hosts
10.0.0.2 slots=10
10.0.0.1 slots=6
```

Here we indicate that new Xeon system has 10 available CPU cores while my legacy workstation has 6 cores. OpenMPI launches jobs in order of the specified resources. Hence, the simulation will first start filling up the new Xeon system before moving to the “mini”. The new system is faster, and is also used solely for computations, hence it makes sense to put it to use first. We use the file by specifying a `hostfile` option,

```
user@mini:~$ mpirun -np 11 -hostfile ~/hosts hostname
mini
xeon
xeon
xeon
xeon
xeon
xeon
xeon
xeon
xeon
xeon
```

As expected, we launched 10 copies on the Xeon with the remaining one deployed on the local system.

With the current setup, we can launch only up to 16 cores. However, let’s say we wanted to run a simulation that requires more cores. This will obviously make individual jobs run at sub 100% CPU usage, but could be useful for code testing. But we get an error message is we try to use more than 16 cores:

```
user@mini:~$ mpirun -np 20 -hostfile ~/hosts hostname
--------------------------------------------------------------------------
There are not enough slots available in the system to satisfy the 20 slots
that were requested by the application:
  hostname
```

This is behavior is unique to OpenMPI. MPICH just starts reusing the available resources by default. We can get this same behavior with OpenMPI using the `oversubscribe` flag,

```
user@mini:~$ mpirun -np 20 -hostfile ~/hosts -oversubscribe hostname
mini
mini
...
xeon
xeon
```

Step 8. Set up 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 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 MPI code:

```
#include <iostream>
#include <sstream>
#include <mpi.h>

int main(int n_args, char *args[])
{
 MPI_Init(&n_args,&args);

 int mpi_size;
 int mpi_rank;
 char proc_name[MPI_MAX_PROCESSOR_NAME];
 int len;

 MPI_Comm_size(MPI_COMM_WORLD,&mpi_size);
 MPI_Comm_rank(MPI_COMM_WORLD,&mpi_rank);
 MPI_Get_processor_name(proc_name,&len);

 std::stringstream ss;
 ss<<"I am "<<mpi_rank<<" of "<<mpi_size<<" on "<<proc_name<<std::endl;
 std::cout<<ss.str();

 MPI_Finalize();

 return 0;
}
```

We can compile and run the program *locally* using

```
user@mini:~$ mpic++ mpi_test.cpp -o mpi_test
user@mini:~$ mpirun -np 3 ./mpi_test
I am 0 of 3 on mini
I am 1 of 3 on mini
I am 2 of 3 on mini
```

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

```
user@mini:~$ mpirun -np 3 -host xeon:3 ./mpi_test
--------------------------------------------------------------------------
mpirun was unable to launch the specified application as it could not access
or execute an executable:
```

This is because there is no `mpi_test` executable in the home directory on the `xeon` harddrive. We could use `scp` or `rsync` to copy it there:

```
user@mini:~$ rsync -rtv ./mpi_test xeon:
sending incremental file list
mpi_test
sent 122,887 bytes  received 35 bytes  245,844.00 bytes/sec
total size is 122,776  speedup is 1.00

user@mini:~$ mpirun -np 3 -host xeon:3 ./mpi_test
I am 0 of 3 on xeon
I am 1 of 3 on xeon
I am 2 of 3 on xeon
```

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. I used the following guide, but essentially, you first need to install NFS (network file system) kernel and common programs:

```
user@mini:~$ 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 `mini` that points to my home directory,

```
user@mini:/$ sudo ln -s /nfs /home/user
user@mini:/$ ls -la /nfs
lrwxrwxrwx 1 root root 12 Aug 18 13:48 nfs -> /home/user/
```

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

```
user@mini:/$ more /etc/exports
 ...
/home/user      xeon(rw)
```

This command gives `xeon` read-write access to the specified folder. Then on the remote client (`xeon`), we start by creating the `/nfs` mount point. Next update `/etc/fstab` to include

```
user@xeon:~$ sudo mkdir /nfs
user@xeon:~$ more /etc/fstab
...
10.0.0.1:/home/user       /nfs      nfs defaults 0 0
```

Then run

```
user@xeon:~$ sudo mount -a
```

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

```
user@xeon:~$ cd /nfs
```

```
user@xeon:/nfs$ ls -la
total 60084
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
...
```

It's possible that instead of a user name, you will see numbers such as 1001 for the file owners. This is what happened to me when doing this setup, and it also prevent me from gaining write access to the remote system. The issue was that, despite using the same user name on both systems, the user id was set differently. My user id was "1000" on `mini` but 1001 on `xeon`. The user ids are set in `/etc/passwd`. I edited the line corresponding to my user name to contain the correct number, and similarly updated `/etc/group`,

```
user@xeon:/nfs$ more /etc/passwd
...
user:x:1000:1000:User,,,:/home/user:/bin/bash

user@xeon:/nfs$ more /etc/group
...
user:x:1000:
```

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

```
user@mini:/nfs$ mpirun -np 3 -host xeon:2 -host mini:1 ./mpi_test
I am 2 of 3 on mini
I am 1 of 3 on xeon
I am 0 of 3 on xeon
```

Who-hoo!

Step 9. Activate firewall (optional)

Nowadays, firewall on Ubuntu systems is controlled via the `ufw` command. By default, firewall is turned off. We can activate it using

```
user@mini:/nfs$ sudo ufw enable
Firewall is active and enabled on system startup
```

But now, trying to run the parallel code leads to no output and MPI timing out

```
user@mini:/nfs$ mpirun -np 3 -host xeon:2 -host mini:1 ./mpi_test
 ------------------------------------------------------------
A process or daemon was unable to complete a TCP connection
to another process:
  Local host:    xeon
  Remote host:   192.168.1.4
This is usually caused by a firewall on the remote host. Please
```

```
check that any firewall (e.g., iptables) has been disabled and
try again.
------------------------------------------------------------
```

We can open a hole in the firewall to allow connection from any system in the `10.0.xxx.xxx` subnet using

```
user@mini:/nfs$ sudo ufw allow from 10.0.0.0/16
user@mini:/nfs$ mpirun -np 3 -host xeon:2 -host mini:1 ./mpi_test
I am 2 of 3 on mini
I am 0 of 3 on xeon
I am 1 of 3 on xeon
```

STEP 10. GET COMPUTING!

And that's it! Congratulations, you now have an MPI cluster. Stay tuned for an upcoming article on parallelization techniques for plasma particle simulation codes. In the meantime, check out my code parallelization course and my book on plasma simulations. The recent article on computing PI using different architectures also discusses different parallelization strategies.

Related Articles:
- Brief Intro to GPU PIC with CUDA
- The Electrostatic Particle In Cell (ES-PIC) Method
- Current Density Limit

Subscribe to the newsletter. Send us an email if you have any questions.