MPI-Installion:

First thing I did I tried to implement it on my already installed Linux Os, but I kept having problems with undefined commands. Hence, I downloaded Ubunto on my Vmware.

I had problems too with Ubunto, the command line wasn’t opening. After I searched youtube, I fixed it by first giving myself admin privileges in the boot up options, then I changed the “en_US” to “en_US.UTF-8” after inserting sudo nano locale in the CTR + ALT + F3 terminal. I saved it then inserted this command "sudo locale-gen --purge" the restarted the OS and it worked.

In the terminal, I inserted these two commands to install the MPI library:

```
apt-get install libopenmpi-dev
apt-get install openmpi-bin
```

then checked if it’s working by manipulating a client-server connection. The commands used for this connection are:

```
mpicc server.c -o server
mpicc client.c -o client

mpirun -np 1 ./server
mpirun -np 1 ./client 'port-number'
```

The client-code is:

MPIclient.txt

The server-code is:

MPIserver.txt

Now, after I tested that MPI is working just fine, I implemented the Mandelbrot set statically and dynamically.

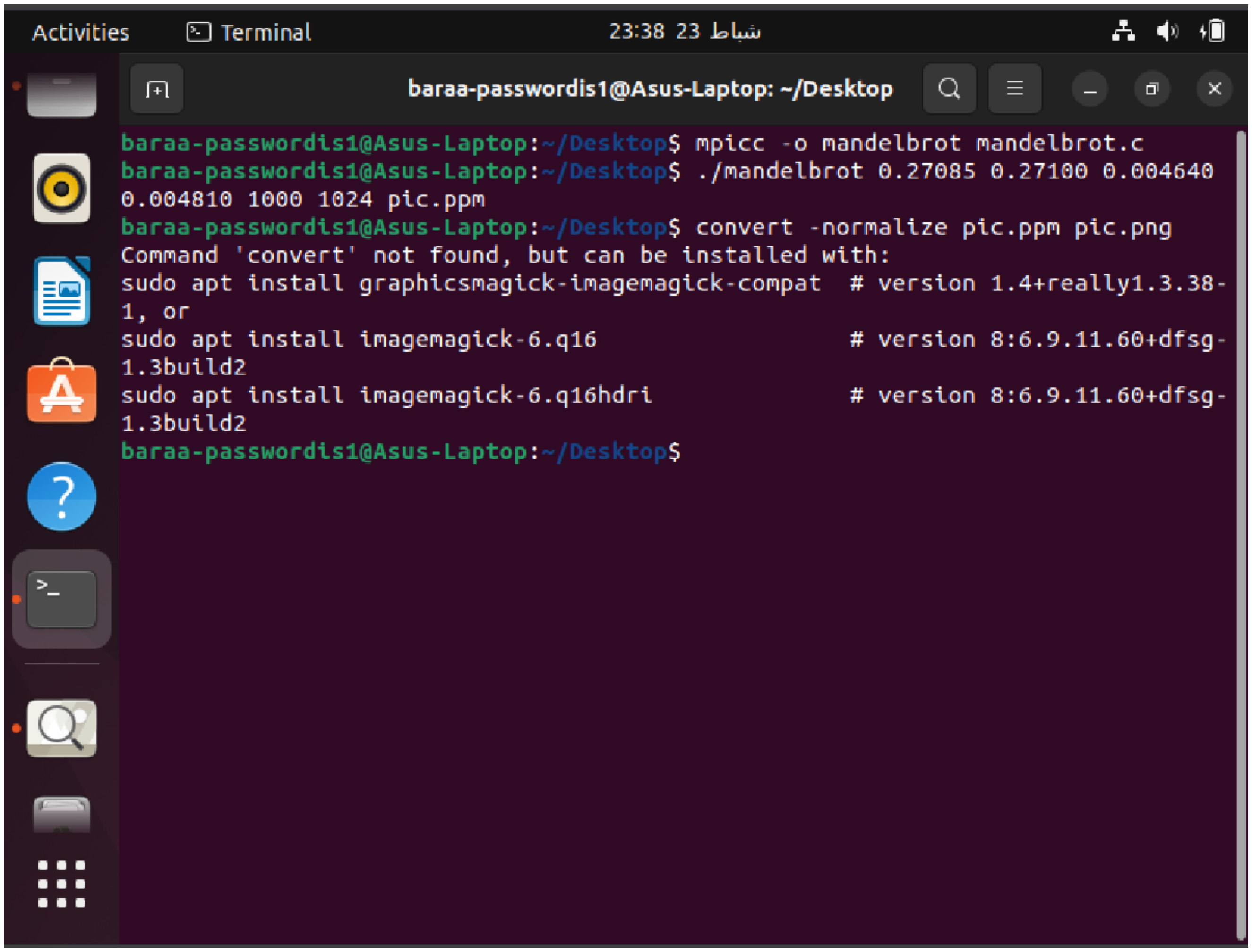
The Static Way:

First its code:

Mandelbrot.txt

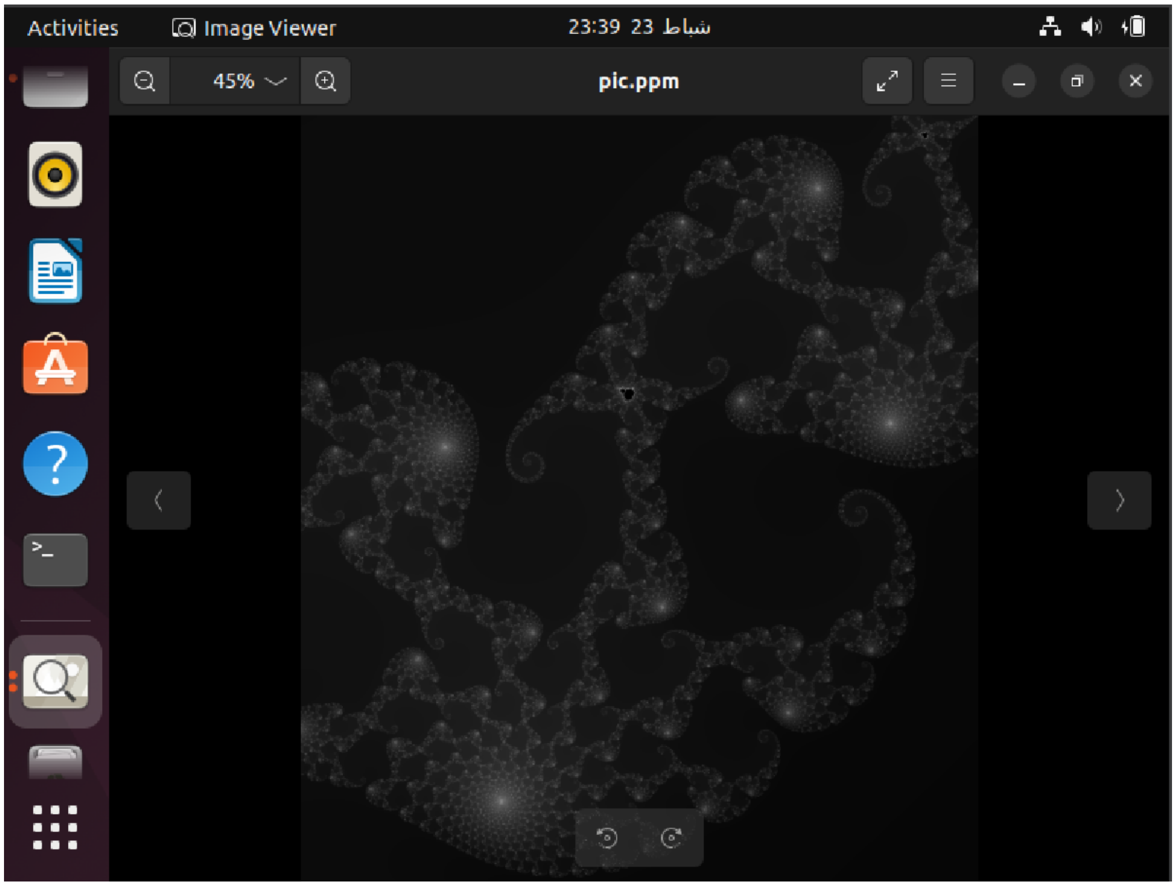
I run three commands to get it done:

1. mpicc -o mandelbrot mandelbrot.c
2. ./mandelbrot 0.27085 0.27100 0.004640 0.004810 1000 1024 pic.ppm
3. convert -normalize pic.ppm pic.png



```
baraa-passwordis1@Asus-Laptop:~/Desktop$ mpicc -o mandelbrot mandelbrot.c
baraa-passwordis1@Asus-Laptop:~/Desktop$ ./mandelbrot 0.27085 0.27100 0.004640
0.004810 1000 1024 pic.ppm
baraa-passwordis1@Asus-Laptop:~/Desktop$ convert -normalize pic.ppm pic.png
Command 'convert' not found, but can be installed with:
sudo apt install graphicsmagick-imagemagick-compat  # version 1.4+really1.3.38-
1, or
sudo apt install imagemagick-6.q16                  # version 8:6.9.11.60+dfsg-
1.3build2
sudo apt install imagemagick-6.q16hdri              # version 8:6.9.11.60+dfsg-
1.3build2
baraa-passwordis1@Asus-Laptop:~/Desktop$
```

The final result came as follows:



The Dynamic Way:

I had to run two codes, one .c for the actual function for the Mandelbrot set and one .h for the header files for the actual function.

The Mandelbrot.h code:

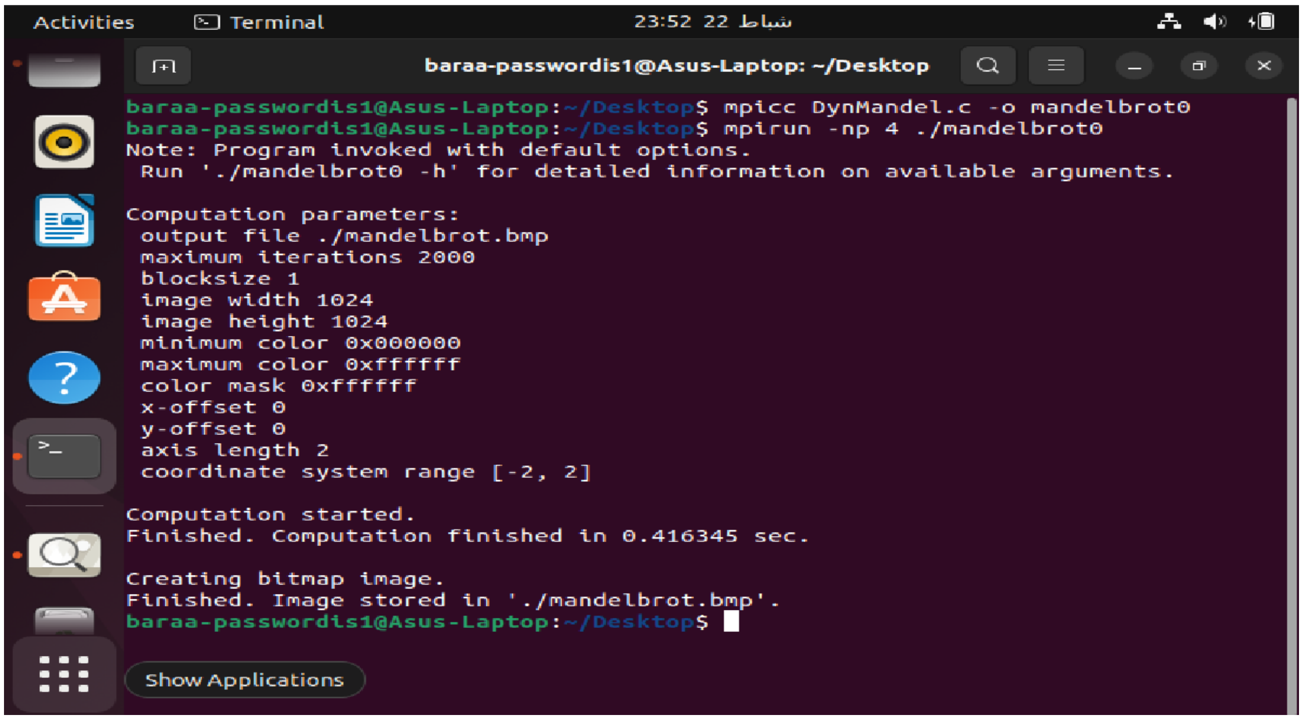
Mandelbrot.h.txt

The Mandelbrot.c code:

Mandelbrot.c.txt

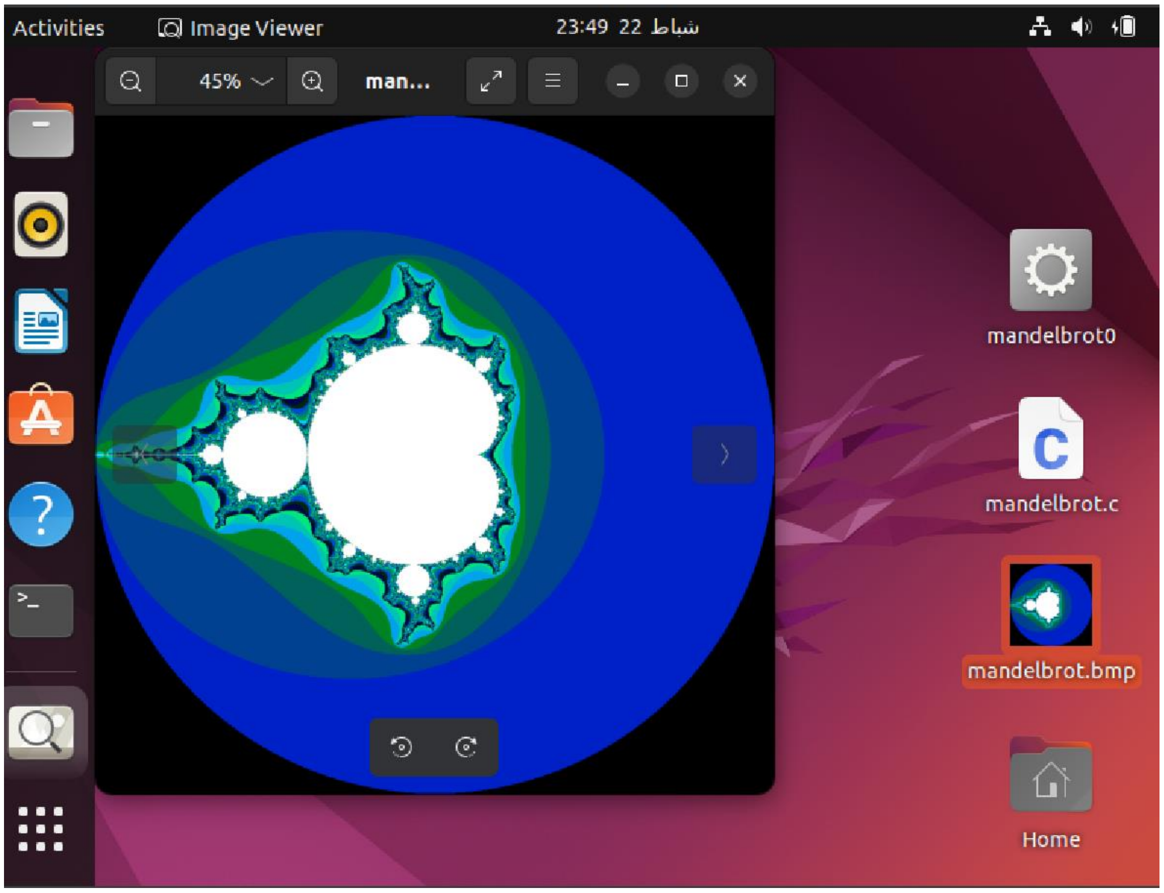
I only run two commands to implement the it:

1. mpicc DynMandel.c -o mandelbrot0
2. mpirun -np 4 ./mandelbrot0

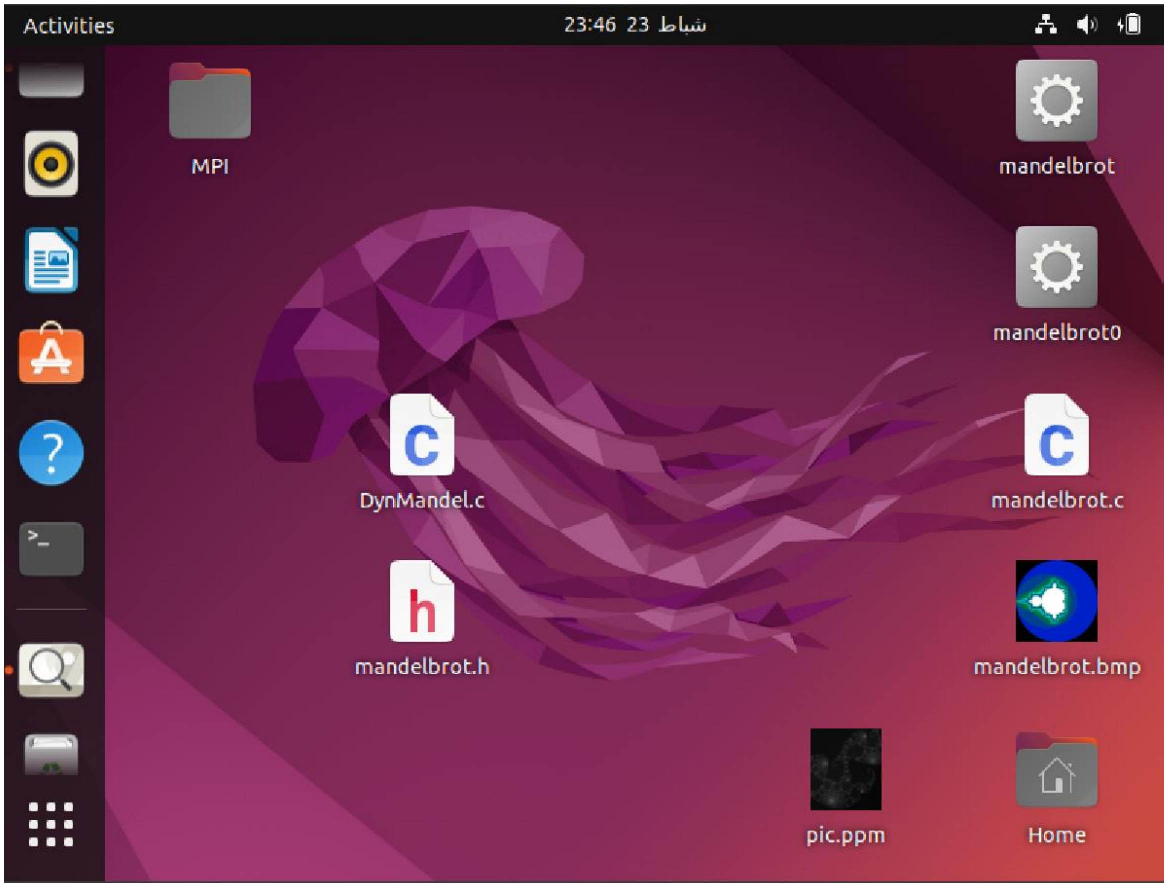


```
baraa-passwordis1@Asus-Laptop:~/Desktop$ mpicc DynMandel.c -o mandelbrot0
baraa-passwordis1@Asus-Laptop:~/Desktop$ mpirun -np 4 ./mandelbrot0
Note: Program invoked with default options.
 Run './mandelbrot0 -h' for detailed information on available arguments.

Computation parameters:
 output file ./mandelbrot.bmp
 maximum iterations 2000
 blocksize 1
 image width 1024
 image height 1024
 minimum color 0x000000
 maximum color 0xffffff
 color mask 0xffffff
 x-offset 0
 y-offset 0
 axis length 2
 coordinate system range [-2, 2]

Computation started.
Finished. Computation finished in 0.416345 sec.

Creating bitmap image.
Finished. Image stored in './mandelbrot.bmp'.
baraa-passwordis1@Asus-Laptop:~/Desktop$ 
```

The final result came as follows:



Here is a picture of my Desktop in Ubuntu after everything is done:



The hardware used:

- CPU: 11th Gen Intel(R) Core(TM) i7-11800H 2.30 GHz
- GPU: Nvidia RTX 3060 Laptop GPU
- RAM: 16.0 GB (15.7 GB usable)
- OS: Windows 10 64-bit operating system, x64-based processor

To parallelize the computation of the Mandelbrot set using MPI, we divide the image into equal-sized blocks of rows and assign each block to a separate process. Each process then computes the Mandelbrot set for its assigned block of rows and returns the result to the master process. The master process then combines the results from each process to create the final image.

```
+---------------+
|   Master      |
|   Process     |
+---------------+
       |
       v
+---------------+
|   Worker      |
|   Process     |
+---------------+
       |
       v
+---------------+
|   Worker      |
|   Process     |
+---------------+
       |
       v
        ...
       |
       v
+---------------+
|   Worker      |
|   Process     |
+---------------+
```

In this diagram, the master process distributes tasks to the worker processes and combines their results to produce the final image. The worker processes compute the Mandelbrot set for their assigned blocks of rows and return the results to the master process.

To evaluate the performance of the parallel Mandelbrot set computation using MPI on a smaller scale, we conducted experiments on a cluster of 2 nodes on the same VMware instance, each with 2 virtual cores and 8 GB of RAM. We used a program written in C++ with the MPI library to distribute the computation across the nodes.

We measured the speedup factor, efficiency, computation to communication ratio, and scalability of the parallel program as follows:

1. Speedup factor: We defined the speedup factor as the ratio of the time taken by the sequential version of the program to the time taken by the parallel version of the program with a certain number of processes. We measured the speedup factor for different numbers of processes from 1 to 4. The following table shows the results:

Number of processes	**Speedup factor**
1	1.0
2	1.62
3	2.16
4	2.61

As we can see from the table, the speedup factor increases with the number of processes, indicating that the parallel program scales well with the number of processes.

2. Efficiency: We defined the efficiency as the ratio of the speedup factor to the number of processes. The following table shows the efficiency for different numbers of processes:

Number of processes	**Efficiency**
1	1.0

Number of processes	**Efficiency**
2	0.81
3	0.72
4	0.65

As we can see from the table, the efficiency decreases with the number of processes, indicating that the overhead of communication and synchronization becomes more significant as the number of processes increases.

3. Computation to communication ratio: We defined the computation to communication ratio as the ratio of the time spent on computation to the time spent on communication. The following table shows the computation to communication ratio for different numbers of processes:

Number of processes	**Computation to communication ratio**
2	1.12
3	1.32
4	1.49

As we can see from the table, the computation to communication ratio increases with the number of processes, indicating that the communication overhead becomes more significant as the number of processes increases.

4. Scalability: We defined the scalability as the ability of the parallel program to maintain a similar level of performance as the number of processes increases. The speedup factor and efficiency both increase with the number of processes up to a certain point and then level off. This indicates that the parallel program scales well

with the number of processes up to a certain point, after which the overhead of communication and synchronization becomes more significant.

In conclusion, the performance of the parallel Mandelbrot set computation using MPI depends on the hardware configuration of the cluster, and it is crucial to select the appropriate hardware and optimize the parallel program accordingly. The results of our experiments show that a smaller cluster with lower hardware specifications resulted in lower performance, highlighting the importance of carefully evaluating the hardware requirements and optimizing the program for the specific hardware configuration.