Lab 1

Mandelbrot set

By

Miguel Ibrahim

ID: 202005354

**Definition of Mandelbrot Set to understand how to program it:**

The Mandelbrot set is the set obtained from the quadratic recurrence equation:

Zn+1 = Z2n +C

with Zo = C, where points C in the complex plane for which the orbit of Zn does not tend to infinity are in the set. Setting Zo equal to any point *in the set* that is not a periodic point gives the same result. The Mandelbrot set was originally called a mui molecule by Mandelbrot. J. Hubbard and A. Douady proved that the Mandelbrot set is connected.

Extracted from: <https://mathworld.wolfram.com/MandelbrotSet.html>

**Simple Code Extracted from GitHub (Without MPI):**

<https://gist.github.com/andrejbauer/7919569>

It is quite hard to understand what is going on in this code, because it does not really talk about it that much however from the link: <https://rosettacode.org/wiki/Mandelbrot_set>

It’s a set of numbers, functions, symbols that somehow shape up the input into a sort of Mandelbrot set, such that they are composed of pixels of set height and width, and their exact positions relative to an image.

Such as:

..

.####

. # .##.

##\*###############.

#.##################

.######################.

######. #######################

##########.######################

##############################################

##########.######################

######. #######################

.######################.

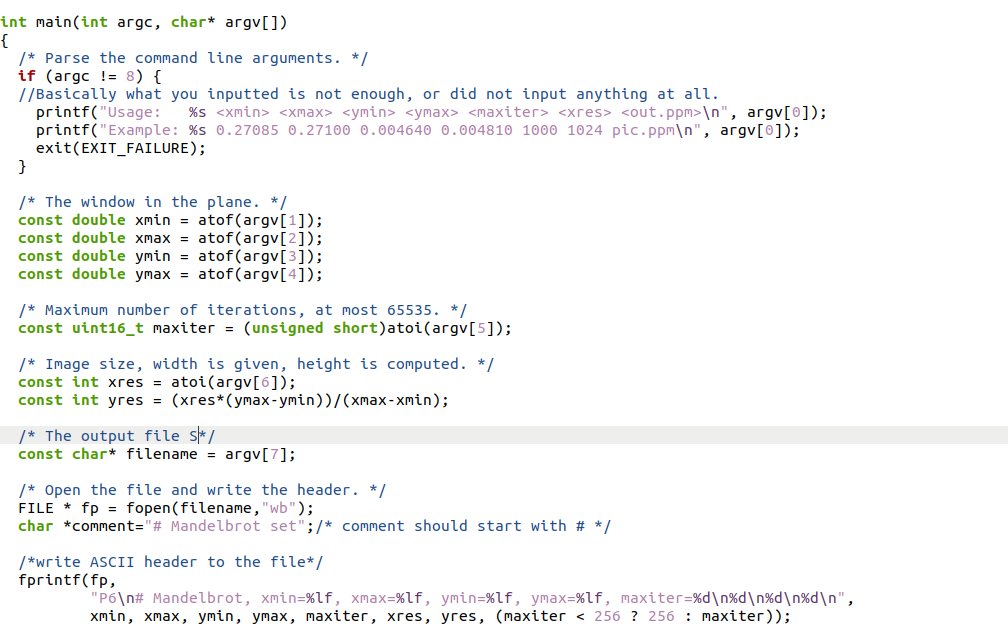
#.##################

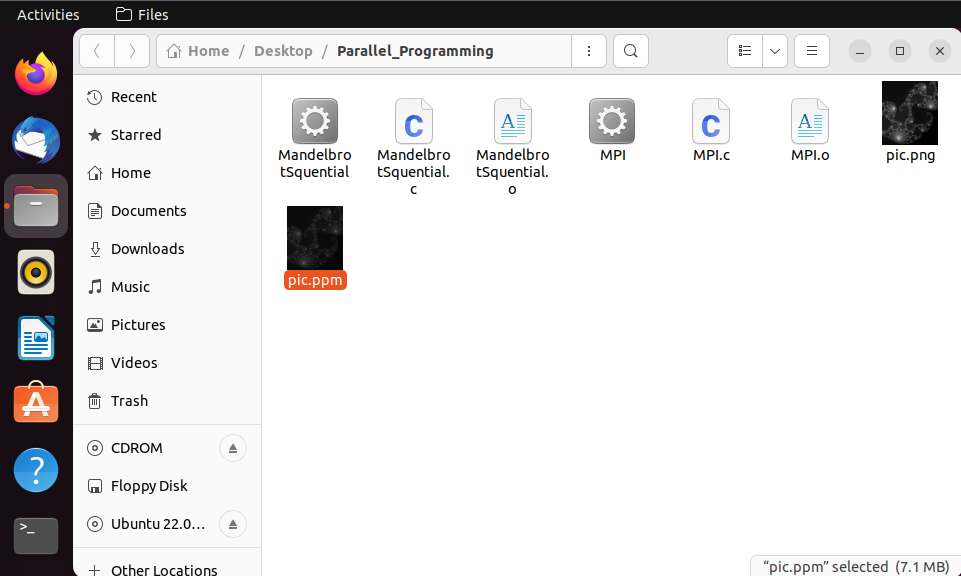
##\*###############.

. # .##.

.####

..

 Or such as this in a sequential code in C, this algorithm ouputs a picture of the Mandelbrot Set using the inputs provided in the terminal, where the output is the picture in ppm format



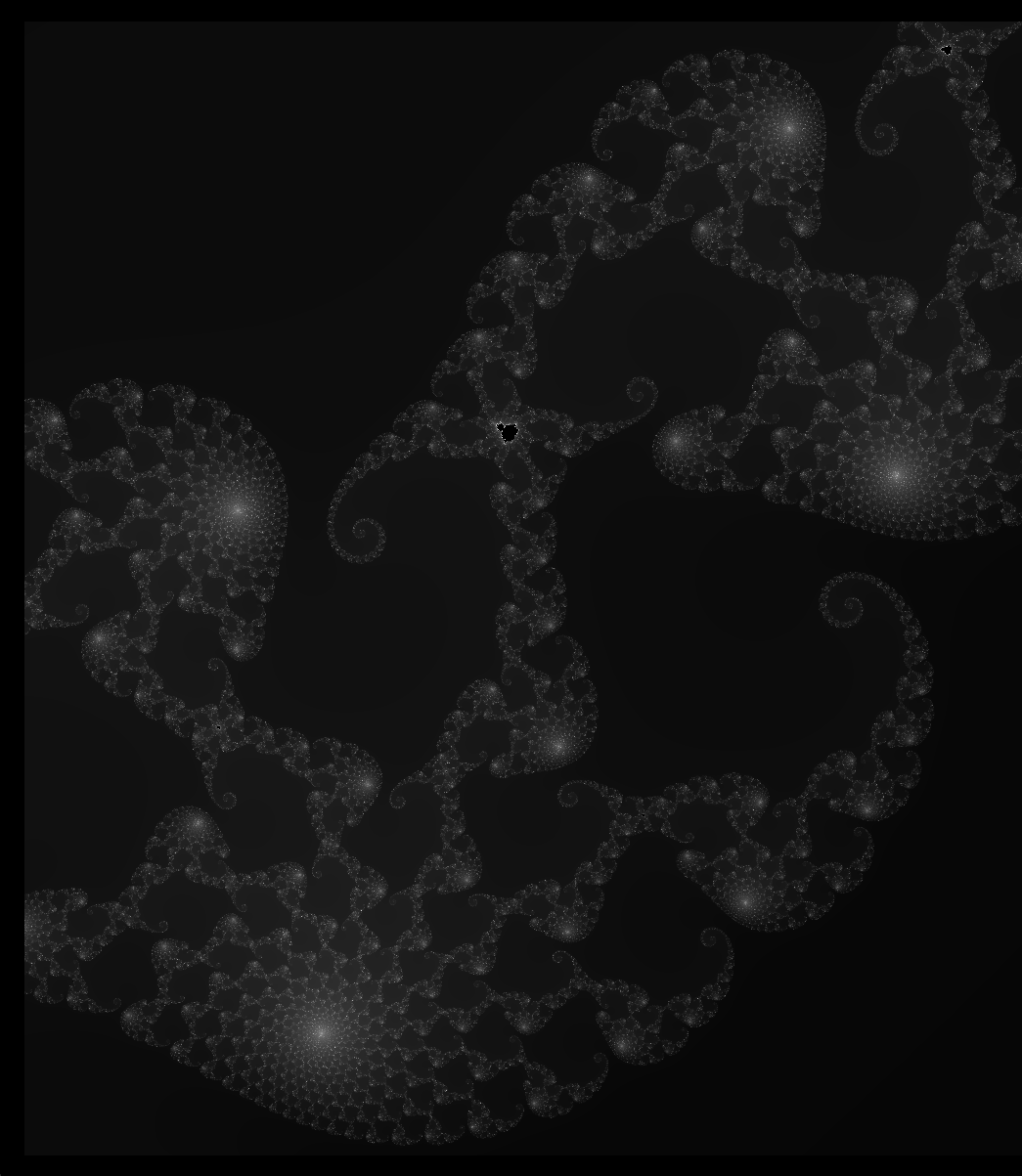
Note: The sequential code is completely copied from github repository to understand the code and how to understand the output:

<https://gist.github.com/andrejbauer/7919569>

Step 1: Navigate to folder and execute using gcc -c since it’s not mpi.

Step2: Execute the compiled file using: ./ with variables: 0.27085 0.27100 0.004640 0.004810 1000 1024 pic.ppm, pic.ppm is the output file.

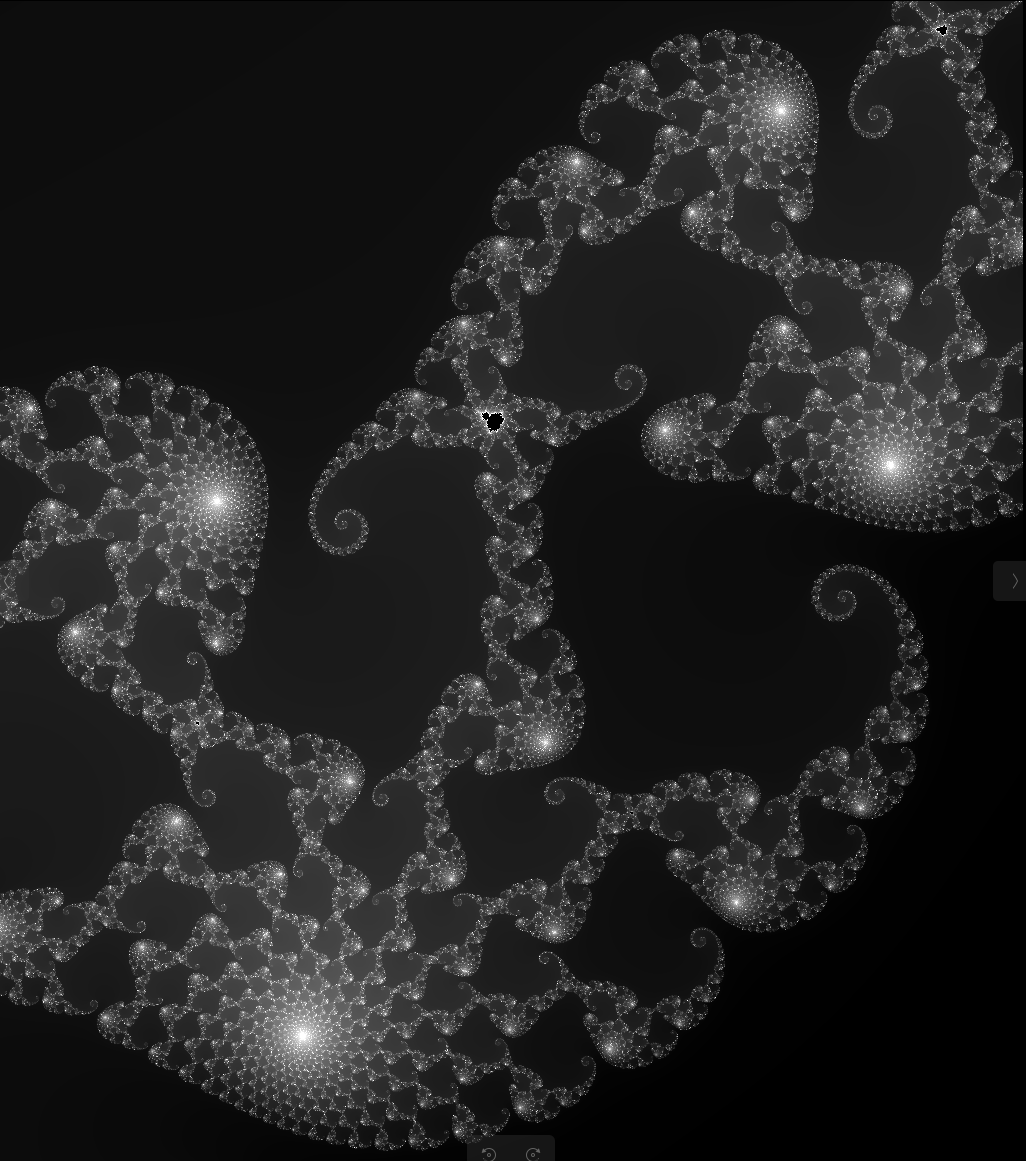
The output file image:



However, the picture is not clear enough and must be converted into png using:

convert -normalize pic.ppm pic.png

Resulting image:



Which is much better.

1. How to parallelize this computation?

The code sequentially works on the image one by one and sums it up into one image at the end.

It would be possible to distribute the task to create the image using multiple slaves and one master that gathers the results of the slaves into one image

Referring to: <https://www.cs.uaf.edu/2009/fall/cs441/lecture/11_19_mandelbrot_compare.html>

The best way to do this is via a reduce function and a round robin distribution of tasks (parts of an image) to the slaves:

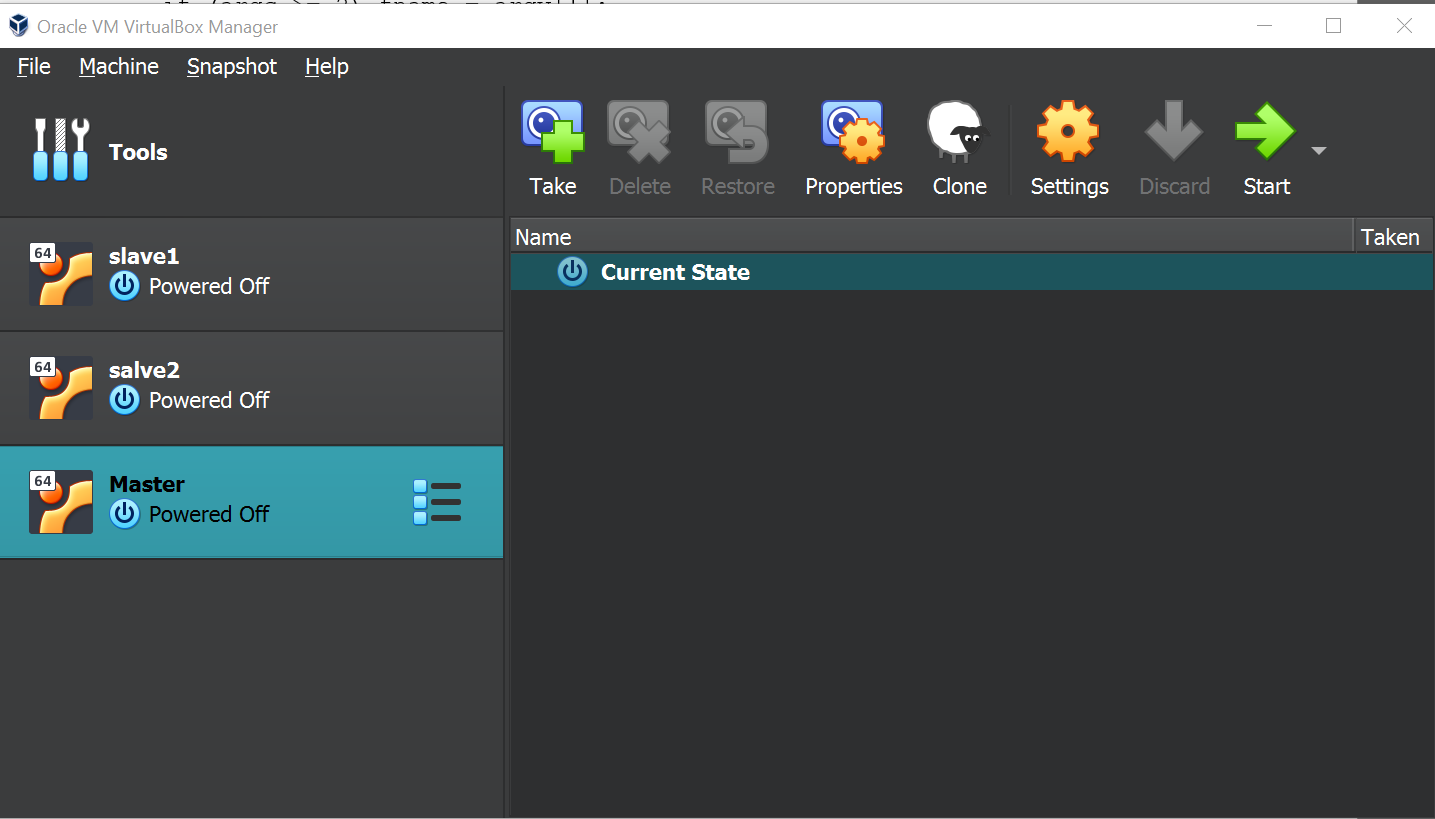
MPI\_Reduce(output,totoutput,sizeof(pixel)\*wid\*ht,MPI\_BYTE,MPI\_BOR,0,c);

The reason behind using a MPI\_Reduce function is because reassembling the image is very slow just by using GATHER function.

1. Setup, and implementation

My setup:

* intel i7-11800H: 2 cores on VBox – Master, intel i7-11800H: 1 core on VBox – Slave1, intel i7-11800H: 1 core on VBox – Slave2, intel i7-11800H: 1 core on VBox – Slave3
* 4GB of RAM at speed: 3200MHz for each (This reduced the overall speed of my system)
* 128GB NVME SSD for Master, 32GB for each slave.
* Laptop Nvidia 3070, 8GB VRAM



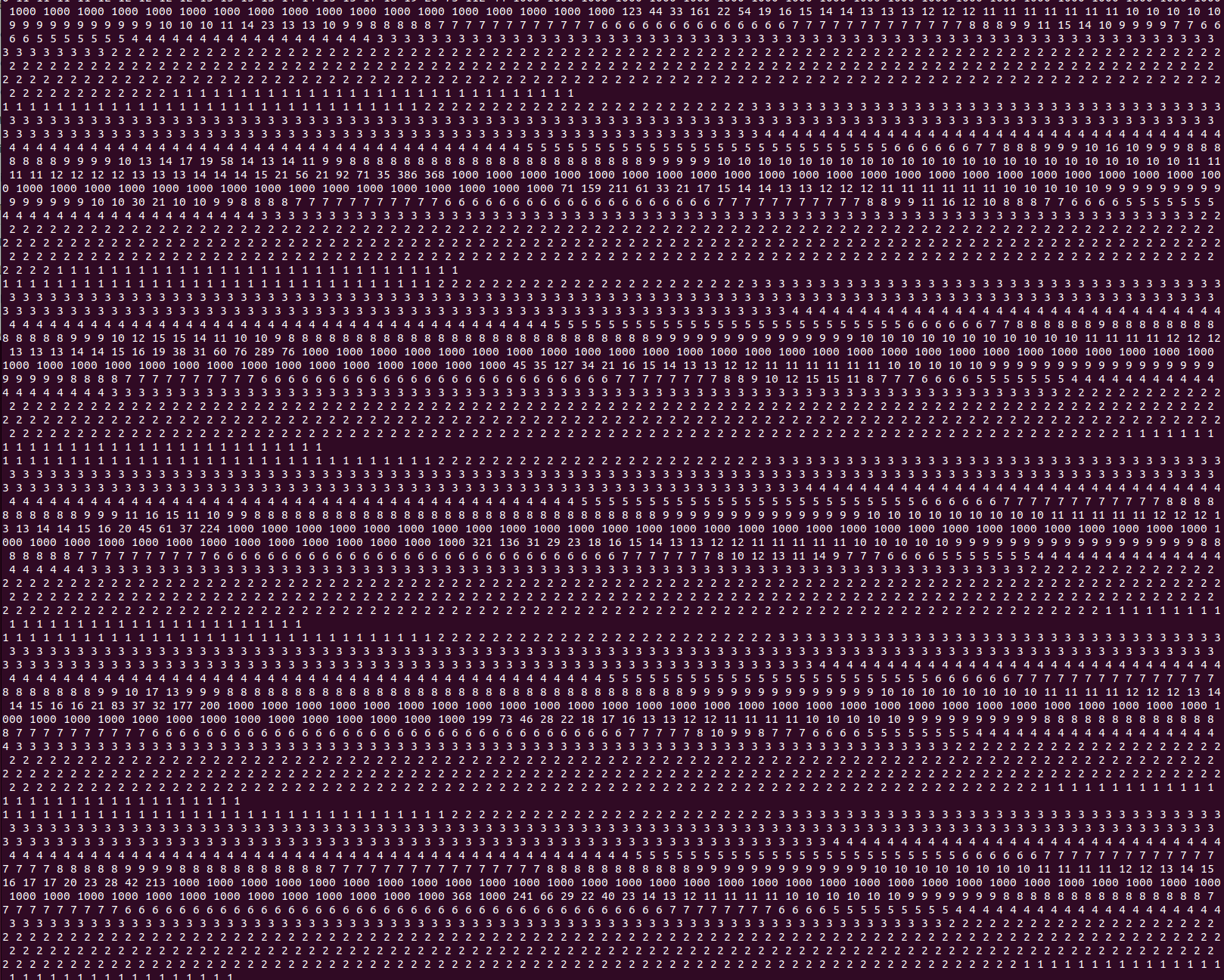
Implementing Parallelized Version of Mendelbrot Set:

We shall distribute the functionalities as discussed above to different “slaves” in network. In this case however, I have one machine that can be virtualized, while others are of different operating systems (MAC).\In this case, I will use multicore clusters on the same machine as if they’re different computers, reducing latency, and idling states.

The Code:

#include <stdio.h>  
#include <stdlib.h>  
#include <complex.h>  
#include <math.h>  
#include <mpi.h>  
  
#define WIDTH 1000  
#define HEIGHT 1000  
#define MAX\_ITER 1000

int mandelbrot(double complex c) {  
    double complex z = 0;  
    int iterations = 0;  
    while (cabs(z) <= 2 && iterations < MAX\_ITER) {  
        z = z \* z + c;  
        iterations++;  
    }  
    return iterations;  
}  
  
int main(int argc, char\* argv[]) {  
    MPI\_Init(&argc, &argv);  
  
    int rank, size;  
    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  
    MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);  
  
    int chunk\_size = HEIGHT / size;  
    int start = rank \* chunk\_size;  
    int end = start + chunk\_size;  
  
    int result[chunk\_size][WIDTH];  
  
    for (int i = start; i < end; i++) {  
        for (int j = 0; j < WIDTH; j++) {  
            double real = (j - WIDTH/2) \* 4.0/WIDTH;  
            double imag = (i - HEIGHT/2) \* 4.0/WIDTH;  
            double complex c = real + imag \* I;  
            int iterations = mandelbrot(c);  
            result[i-start][j] = iterations;  
        }  
    }  
  
    int \*recv\_result;  
    if (rank == 0) {  
        recv\_result = (int\*) malloc(HEIGHT \* WIDTH \* sizeof(int));  
    }  
  
    MPI\_Gather(result, chunk\_size \* WIDTH, MPI\_INT, recv\_result, chunk\_size \* WIDTH, MPI\_INT, 0, MPI\_COMM\_WORLD);  
  
    if (rank == 0) {  
        for (int i = 0; i < HEIGHT; i++) {  
            for (int j = 0; j < WIDTH; j++) {  
                printf("%d ", recv\_result[i \* WIDTH + j]);  
            }  
            printf("\n");  
        }  
    }  
  
    MPI\_Finalize();  
    return 0;  
}

Output after using: mpicc MPI.c -o MPI -lmto compile and ./MPI 4 for cores (Nodes), 

This can be translated into colors at specific positions using an output analyser that can b downloaded using ImageMagick.

1. Gihub Link:

Personal Github rep: <https://github.com/MiguelIbrahimE/Parallel-Programming>

Github reps used:

Sequential Mandelbrot Set Dynamic: <https://github.com/martinohmann/mpi-mandelbrot>

1. Working implementation:

#include <complex.h>

#include <stdio.h>

#include <stdlib.h>

#include "mpi.h"

const int MASTER = 0, TAG = 1;

int STOP = -1;

unsigned int bailout(float complex z, unsigned int max\_iters) {

unsigned int i = 0;

float complex zi = 0.0 + 0.0 \* I;

while (creal(zi)\*creal(zi)+cimag(zi)\*cimag(zi) < 4 && i++ < max\_iters) {

zi = zi\*zi + z;

}

if (i != max\_iters) {

return i;

} else {

return max\_iters;

}

}

int main(int argc, char \*\*argv) {

int err, num\_ranks, rank;

MPI\_Status status;

err = MPI\_Init(&argc, &argv);

if (err != MPI\_SUCCESS) {

fprintf(stderr, "Couldn't start MPI.\n");

MPI\_Abort(MPI\_COMM\_WORLD, err);

}

MPI\_Comm\_size(MPI\_COMM\_WORLD, &num\_ranks);

if (num\_ranks < 1) {

fprintf(stderr, "Need at least one rank besides master.\n");

MPI\_Finalize();

exit(EXIT\_FAILURE);

}

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

float x0 = -2.0, y0 = -1.0, xn = 1.0, yn = 1.0;

unsigned int w = 800, h = 600, max\_iters = 20;

if (argc >= 3) w = (unsigned)atoi(argv[2]);

if (argc >= 4) h = (unsigned)atoi(argv[3]);

if (argc >= 5) max\_iters = (unsigned)atoi(argv[4]);

if (argc >= 6) x0 = atof(argv[5]);

if (argc >= 7) y0 = atof(argv[6]);

if (argc >= 8) xn = atof(argv[7]);

if (argc >= 9) yn = atof(argv[8]);

float scale\_x = (xn - x0) / (float)w;

float scale\_y = (yn - y0) / (float)h;

unsigned int \*\*img = NULL; /\* Used by master \*/

if (rank == MASTER) {

img = (unsigned int\*\*)malloc(h\*sizeof(unsigned int\*));

for (size\_t i = 0; i < h; ++i) {

img[i] = (unsigned int\*)malloc(w\*sizeof(unsigned int));

}

int i = 0;

while (i < h) {

int k = 1;

do {

err = MPI\_Send(&i, 1, MPI\_INT, k, TAG, MPI\_COMM\_WORLD);

if (err != MPI\_SUCCESS) MPI\_Abort(MPI\_COMM\_WORLD, err);

err = MPI\_Recv(img[i], w, MPI\_UNSIGNED, k, TAG, MPI\_COMM\_WORLD,

&status);

if (err != MPI\_SUCCESS) MPI\_Abort(MPI\_COMM\_WORLD, err);

} while (++i < h && ++k < num\_ranks);

}

for (int k = 1; k < num\_ranks; ++k) {

MPI\_Send(&STOP, 1, MPI\_INT, k, TAG, MPI\_COMM\_WORLD);

}

} else {

int i = !STOP;

while (i != STOP) {

err = MPI\_Recv(&i, 1, MPI\_INT, MASTER, TAG, MPI\_COMM\_WORLD, &status);

if (err != MPI\_SUCCESS) MPI\_Abort(MPI\_COMM\_WORLD, err);

if (i == STOP) break;

unsigned int \*row = (unsigned int\*)malloc(w\*sizeof(unsigned int));

float im = y0 + i \* scale\_y;

for (unsigned int j = 0; j < w; ++j) {

float re = x0 + j \* scale\_x;

float complex z = re + im \* I;

row[j] = bailout(z, max\_iters);

}

err = MPI\_Send(row, w, MPI\_UNSIGNED, MASTER, TAG, MPI\_COMM\_WORLD);

free(row);

if (err != MPI\_SUCCESS) MPI\_Abort(MPI\_COMM\_WORLD, err);

}

}

/\* All done, print to file if requested and free. \*/

if (rank == MASTER) {

char const \*fname = NULL;

if (argc >= 2) fname = argv[1];

FILE \*f = fopen(fname, "w");

for (size\_t i = 0; i < h; ++i) {

if (f) {

for (size\_t j = 0; j < w; ++j) {

fprintf(f, "%zu,", img[i][j]);

}

fprintf(f, "\n");

}

free(img[i]);

}

free(img);

if (f) {

fclose(f);

}

}

MPI\_Finalize();

return EXIT\_SUCCESS;

}

After using a similar input to the one used with the non parallel-programming version.

­­compiling instructions:

\* mpicc mandelbrot.c -std=c99 -o output\_file

\*

\* Running instructions:

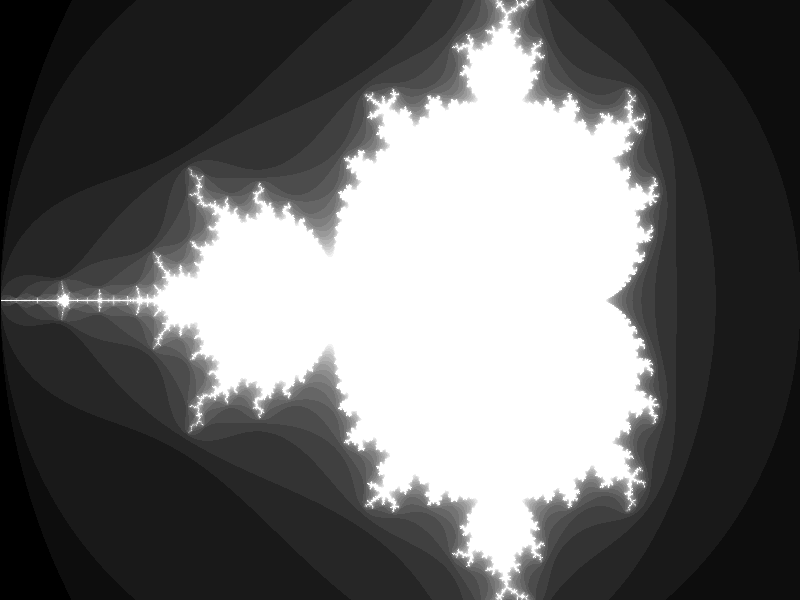
\* mpirun -np NUM\_RANKS output\_file [FNAME] [W] [H] [MAX\_ITER] [X0] [Y0] [XN]

\* [YN]

Note: This is a static implementation

Much help from: <https://codereview.stackexchange.com/questions/78124/mandelbrot-fractal-with-mpi>

Output:



1. For the sake of calculating the time taken, we shall use the include: #include <time.h>

   t = clock();

    fun(); //Function

    t = clock() - t;

    double time\_taken = ((double)t)/CLOCKS\_PER\_SEC; // in seconds

    printf("fun() took %f seconds to execute \n", time\_taken);

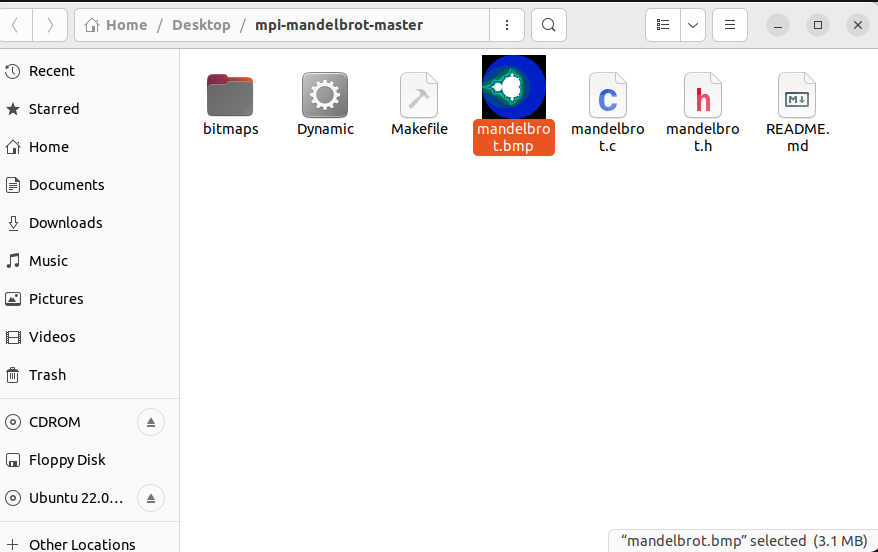
Average compilation time for the non-parallelized program after running 10 tests:

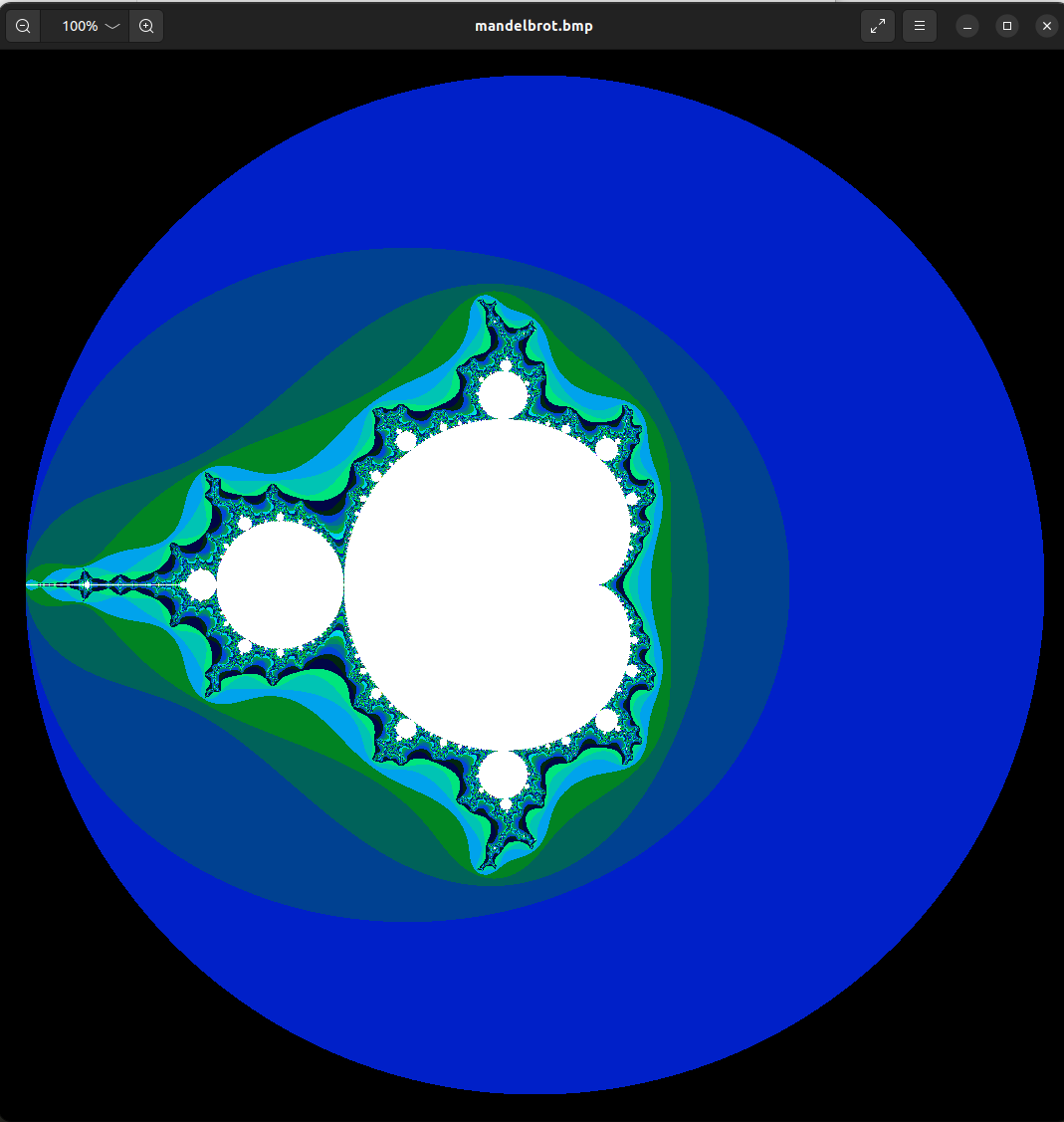
2.9 Seconds using the setup: 6 cores on VMware core i7: 11800H base clock: 2.3GHz, 8GB RAM, 128 GB SSD.

Average compilation time for the parallelized program (dynamic) from: <https://github.com/martinohmann/mpi-mandelbrot>

For: 1 Master, 3 Slaves using Virtual Box (On the Same machine) Note: This made the system extremely slow and nothing was capable of running, average running time: 10.5 seconds with the fans running at 63Dbs.

Output of the Dynamic C file:





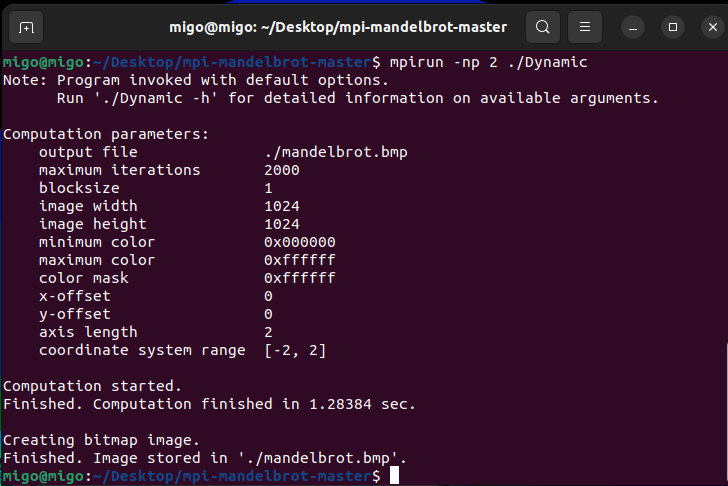
On the same machine, on the same Virtual Machine (VMWare)

Setup:

* 6 Cores at 2.3GHz Base Clock Speed
* 8GBs Ram at 3200MHz

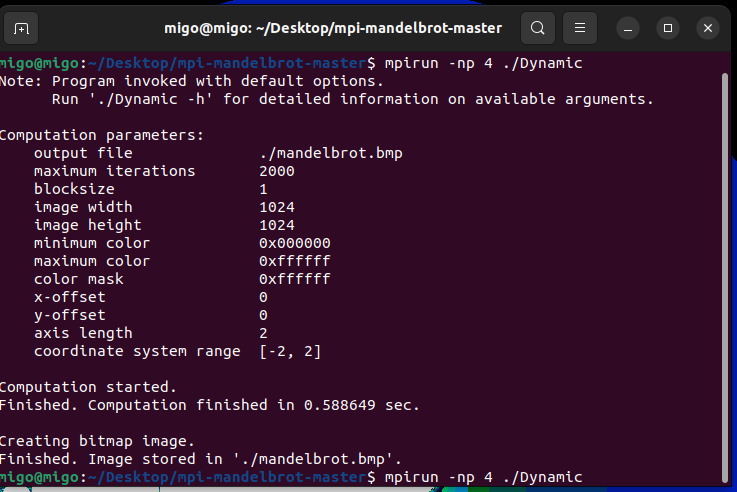
Output:

With 2 Cores:

 Finished in 1.28384 seconds

Average time: 1.2738 (8 Runs)

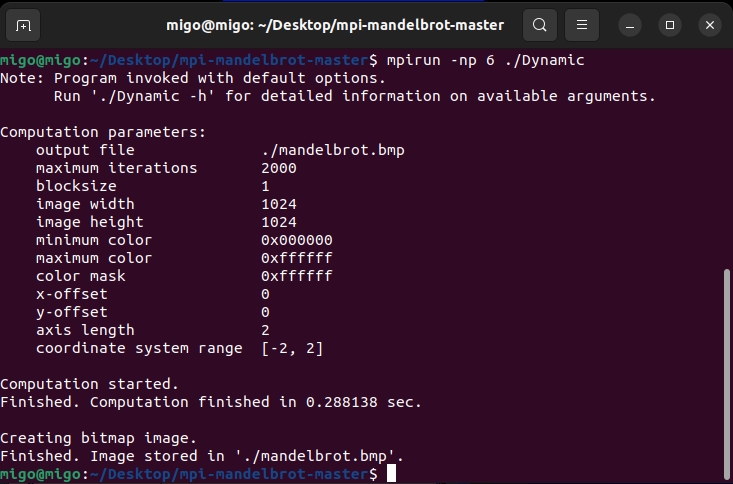
With 4 Cores:

 Finished in: 0.58849 seconds

Average time:

0.4514 seconds (8 Runs)

Output after using 6 Cores:

 Finished in 0.288138

Average 0.26901 (8 Runs)

Dynamic

Time to discuss Speedup factor Sequential vs Dynamic MPI.

* Speedup:

Time on one processor: 2.9 Seconds Average

Time on 6 processors: 0.26901 Seconds Average

2.9/0.26901= 10.78025839150961

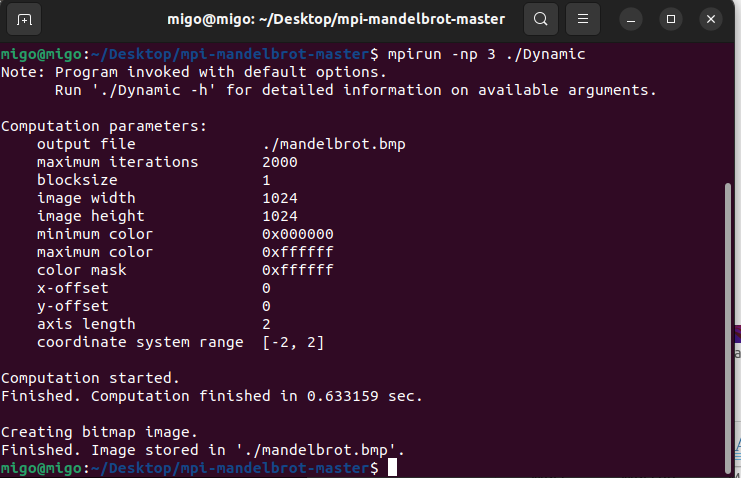
- Efficiency:

2.9/0.26901\*6= 2.49=> 249%

- Computation to communication:

On the system that are 3 separate VMs it took 10.5 Seconds (3 cores)

On the same VM however, after using 3 cores:



It averaged around 0.57 (Which is weird because 3 vs 4 cores are almost the same)

10 seconds vs 0.57 is a great difference hence an increase in speed that could be important, lost in latency, or an overload on a single system maybe.

1. Discussion

After seeing the program in Sequential, Static, and Dynamic

The Program performed best using the Dynamic variation With the best variation on my specs being the 6 cores variation on the same Virtual Machine. The next best is Static allocation of recourses with 6 cores, going back to dynamic with 4 cores, etc.

The worst performing one is not even the Sequential one which is to be expected, the worst performing one after all these tests is the Sequential Using 3 Different Virtual Machines and it makes sense that it is the worst considering the problems it causes the main system while running, but as well as the communication between the different VMs that are causing it to wait for instructions and sending them back, it takes too long!