Assignment 1 (Parallelizing the Mandelbrot Set Computation)

Anas Albaqeri

202004427

Introduction

Mandelbrot Set

The Mandelbrot set is a two-dimensional set with a relatively simple definition that exhibits great complexity, especially as it is magnified. It is popular for its aesthetic appeal and fractal structures. (Wiki)

The Mandelbrot set is represented using the following formula

$z(n+1) = z_n^2 + c$ with $z_0 = 0$

The Mandelbrot set bounds the magnitude of Zn to a certain limit and if that limit is exceeded for a number c, it is then said that c is not a member of the Mandelbrot Set

Mandelbrot and Parallelization

we can parallelize the computation of Mandelbrot set by dividing the task (in our case the image) into smaller pieces and allocating each to a processor or a thread to be worked on simultaneously. Each processor works individually on part of the image, preforming the same computation for each pixel within the allocated section determining whether each pixel is inside or outside the the set.

Setup

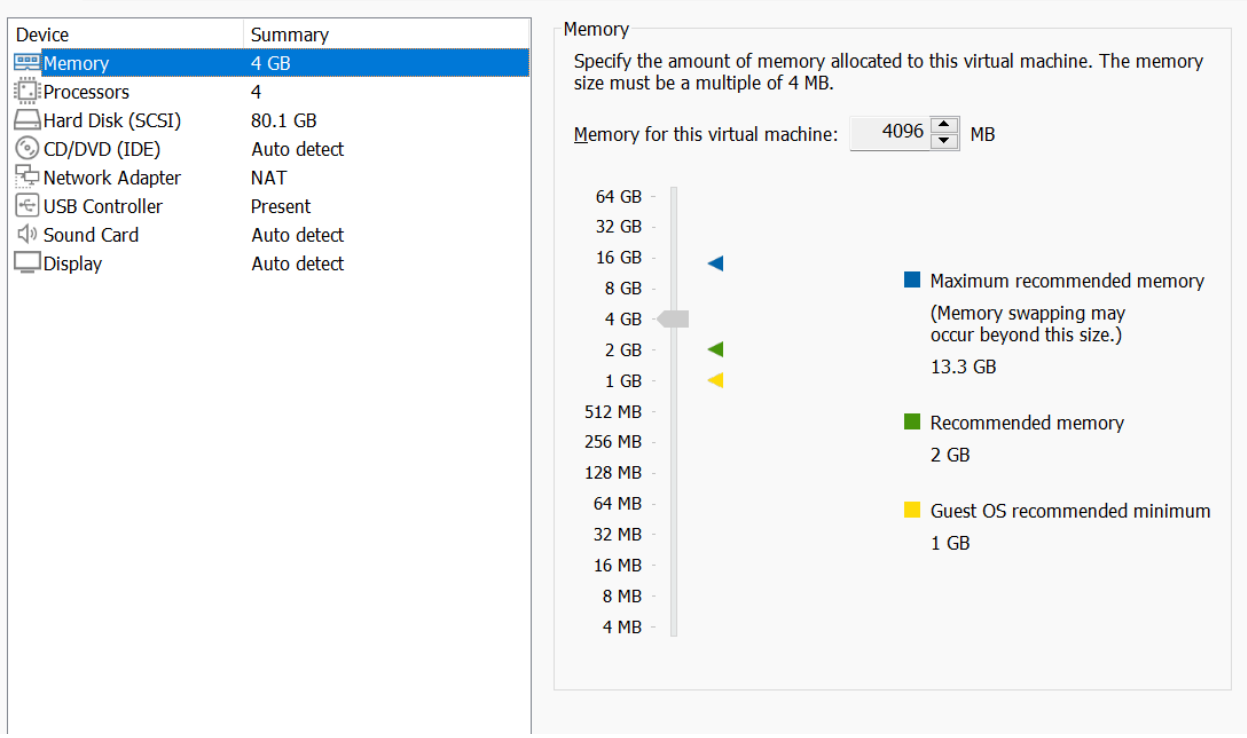
for this process, I used Kali Linux emulator to run the MPI program for Mandelbrot Set. The program was written and is copyrighted by Martin Ohman, 2015, and is free as stated by the author:

https://github.com/martinohmann/mpi-mandelbrot/blob/master/mandelbrot.h

MPI version used:

```
┌──(kali㉿kali)-[~]
└─$ mpirun --version
mpirun (Open MPI) 3.1.3

Report bugs to http://www.open-mpi.org/community/help/

┌──(kali㉿kali)-[~]
└─$
```

Kali Specifications:



for this task, I used 4 cores as 4 parallel processors working with 4 gb of RAM. This was the maximum my machine was able to handle for the Kali Emulator.

Sequential:

The sequential Mandelbrot solution is achieved by giving one processor the entire Image at once with all the processes happening in sequence.

Here is a break down of the code used with explanation of each step:

```c
#include <stdio.h>
#include <complex.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>

void compute_mandelbrot_subset(int *result, int iter_limit, int x_resolution, int y_resolution, double x_begin, double y_begin, double x_end, doub
le y_end);

int main(int argc, char *argv[])
{
    int width = 666;
    int height = 666;
    double x_min = -2.0;
    double x_max = 1.0;
    double y_min = -1.0;
    double y_max = 1.0;
    int iter_limit = atoi(argv[1]);
    int data[width * height];

    clock_t start_time = clock();

    clock_t end_time = clock();
    double computation_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;

    printf("Computation time: %f seconds\n", computation_time);
    printf("Iteration Number: %d\n", iter_limit);

    return 0;
}
```

```c
void compute_mandelbrot_subset(int *result, int iter_limit, int x_resolution, int y_resolution, double x_begin, double y_begin, double x_end, doub
le y_end)
{

    double x_step = (x_end - x_begin) / (x_resolution - 1);
    double y_step = (y_end - y_begin) / (y_resolution - 1);
    int i, j;

    for (i = 0; i < x_resolution * y_resolution; i++)
    {
        double complex c = x_begin + (i % x_resolution) * x_step + (y_begin + (i / x_resolution) * y_step) * I;
        double complex z = 0 + 0 * I;
        j = 0;
        while (cabs(z) <= 2 && j < iter_limit)
        {
            z = z * z + c;
            j++;
        }
        result[i] = j;
    }
}
```

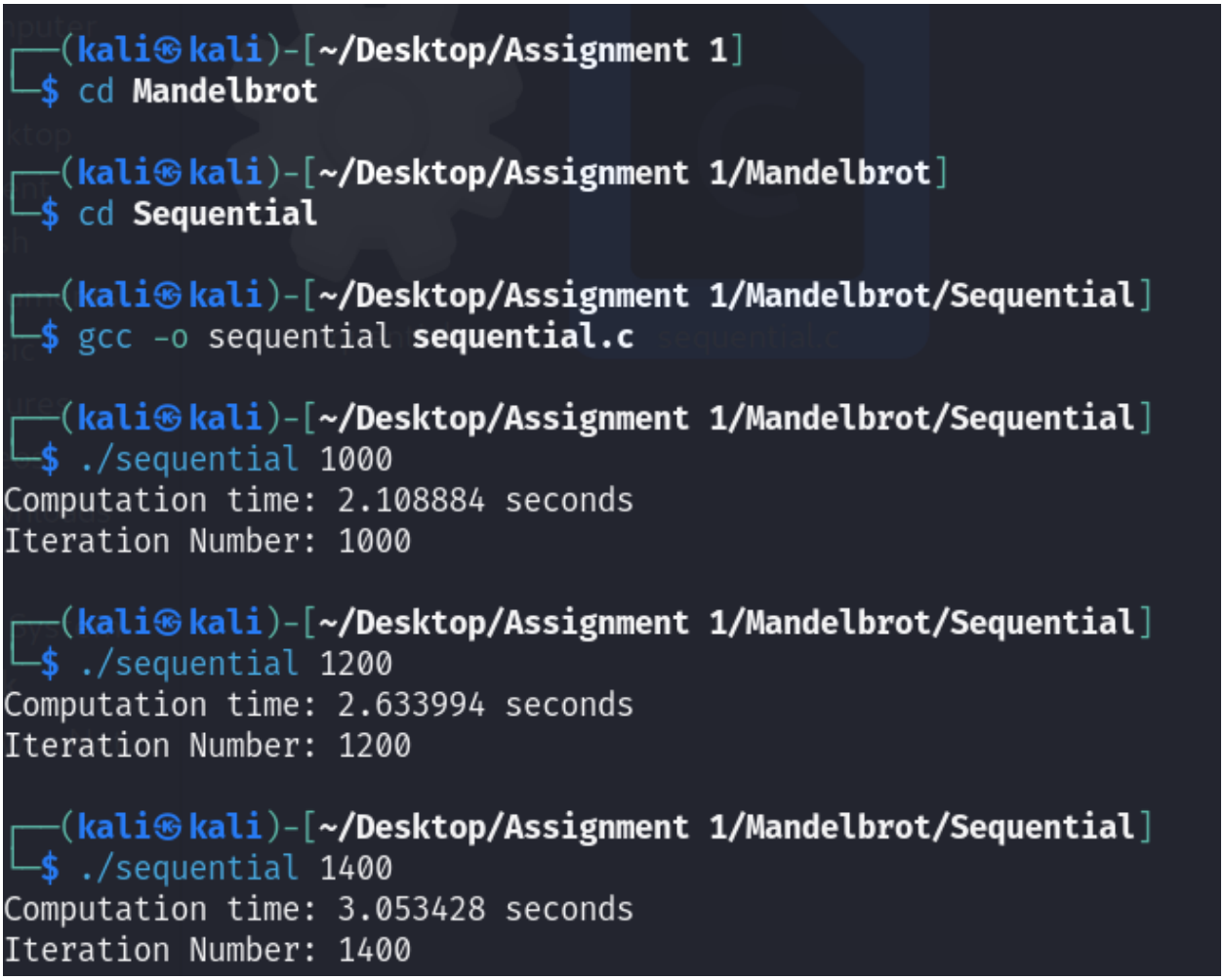
1. **Include Necessary Tools**: The code first makes sure it can use some helpful tools, like the ability to do complex math, measure time, and handle input/output.
2. **Declare Variables**: It sets up some important values:
    - `width` and `height` determine the size of the picture (666x666 pixels).
    - `x_min`, `x_max`, `y_min`, and `y_max` define the area of the picture to look at.
    - `iter_limit` comes from a command you give when running the program and controls how detailed the picture will be.
    - `data` is a big list that will hold the results for each pixel.
    - `start_time` is like a stopwatch; it remembers when we start.

3. **Calculate the Mandelbrot Set**: It runs a function called `compute_mandelbrot_subset` to do the real work. We'll explain this function more later.
4. **Stop the Stopwatch**: It notes the time again now that the work is done (`end_time`).
5. **Calculate Time Taken**: It figures out how much time was spent doing the calculations.
6. **Print Results**: It prints the time taken and the iteration limit you gave.
7. `compute_mandelbrot_subset` **Function**:
    - This part of the code calculates a piece of the Mandelbrot set.
    - It needs a few things to work:
        - `result` is where it will store the calculated values.
        - `iter_limit` tells it how many times to repeat a math process.
        - `x_resolution` and `y_resolution` say how many pixels wide and tall the piece is.
        - `x_begin`, `y_begin`, `x_end`, and `y_end` mark the boundaries of the piece.
8. **Calculate Pixel Steps**: It calculates how much to move from one pixel to the next in the x and y directions (`x_step` and `y_step`).
9. **Loop Over Pixels**: It starts looking at each pixel one by one:
    - For each pixel, it figures out a complex number `c` based on where the pixel is.
    - It sets another complex number `z` to zero.
10. **Calculate Mandelbrot Value**: For each pixel, it calculates how long it takes for `z` to get big (or if it ever does):

I ran the test fifteen times sequentially, increasing the iterations by 200 each time and these are the results obtained each time:

also since this is a sequential code, I ran the code using the gcc compiler as it is best fit for the the sequential code and would give more accurate speedup factors later on when measuring.

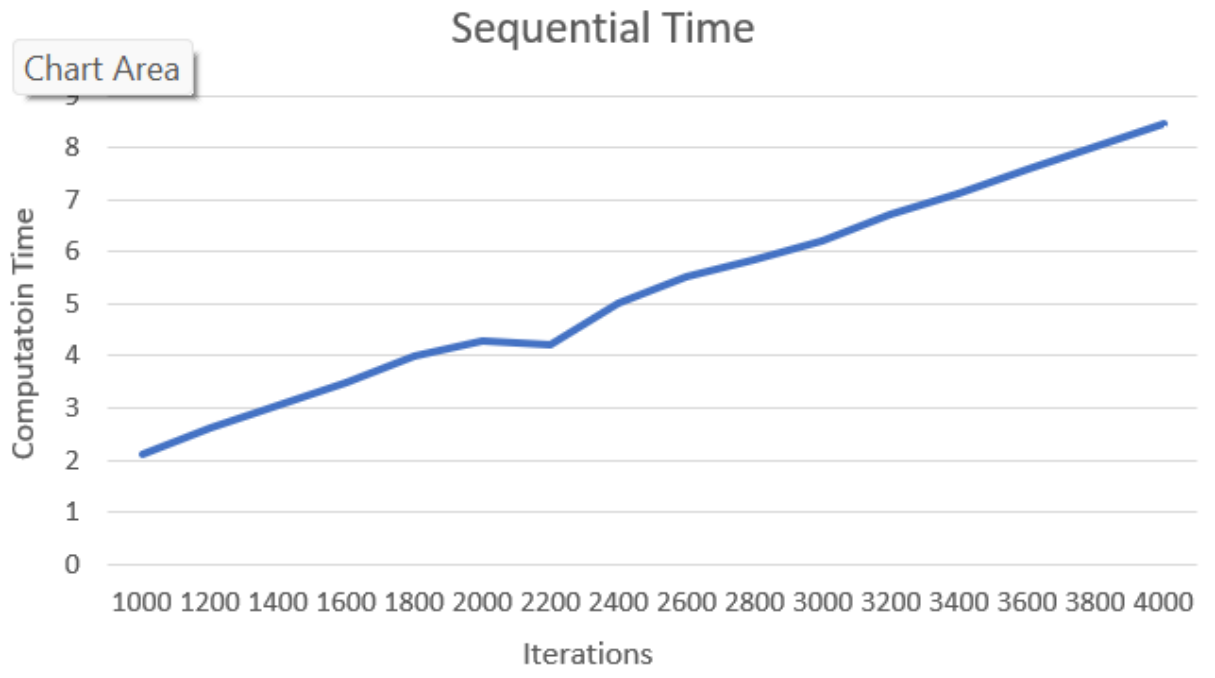
Output


┌──(kali㉿kali)-[~/Desktop/Assignment 1]
└─$ cd Mandelbrot
┌──(kali㉿kali)-[~/Desktop/Assignment 1/Mandelbrot]
└─$ cd Sequential
┌──(kali㉿kali)-[~/Desktop/Assignment 1/Mandelbrot/Sequential]
└─$ gcc -o sequential sequential.c
┌──(kali㉿kali)-[~/Desktop/Assignment 1/Mandelbrot/Sequential]
└─$ ./sequential 1000
Computation time: 2.108884 seconds
Iteration Number: 1000
┌──(kali㉿kali)-[~/Desktop/Assignment 1/Mandelbrot/Sequential]
└─$ ./sequential 1200
Computation time: 2.633994 seconds
Iteration Number: 1200
┌──(kali㉿kali)-[~/Desktop/Assignment 1/Mandelbrot/Sequential]
└─$ ./sequential 1400
Computation time: 3.053428 seconds
Iteration Number: 1400

```
┌──(kali㉿kali)-[~/Desktop/Assignment 1/Mandelbrot/Sequential]
└─$ ./sequential 1600
Computation time: 3.508554 seconds
Iteration Number: 1600

┌──(kali㉿kali)-[~/Desktop/Assignment 1/Mandelbrot/Sequential]
└─$ ./sequential 1800
Computation time: 3.982334 seconds
Iteration Number: 1800

┌──(kali㉿kali)-[~/Desktop/Assignment 1/Mandelbrot/Sequential]
└─$ ./sequential 2000
Computation time: 4.307324 seconds
Iteration Number: 2000

┌──(kali㉿kali)-[~/Desktop/Assignment 1/Mandelbrot/Sequential]
└─$ ./sequential 2200
Computation time: 4.214426 seconds
Iteration Number: 2200
```

```
┌──(kali㉿kali)-[~/Desktop/Assignment 1/Mandelbrot/Sequential]
└─$ ./sequential 2400
Computation time: 5.016324 seconds
Iteration Number: 2400

┌──(kali㉿kali)-[~/Desktop/Assignment 1/Mandelbrot/Sequential]
└─$ ./sequential 2600
Computation time: 5.522627 seconds
Iteration Number: 2600

┌──(kali㉿kali)-[~/Desktop/Assignment 1/Mandelbrot/Sequential]
└─$ ./sequential 2800
Computation time: 5.859339 seconds
Iteration Number: 2800

┌──(kali㉿kali)-[~/Desktop/Assignment 1/Mandelbrot/Sequential]
└─$ ./sequential 3000
Computation time: 6.202086 seconds
Iteration Number: 3000
```

```
┌──(kali㉿kali)-[~/Desktop/Assignment 1/Mandelbrot/Sequential]
└─$ ./sequential 3200
Computation time: 6.706728 seconds
Iteration Number: 3200

┌──(kali㉿kali)-[~/Desktop/Assignment 1/Mandelbrot/Sequential]
└─$ ./sequential 3400
Computation time: 7.122299 seconds
Iteration Number: 3400

┌──(kali㉿kali)-[~/Desktop/Assignment 1/Mandelbrot/Sequential]
└─$ ./sequential 3600
^[[17~^[[17~Computation time: 7.601885 seconds
Iteration Number: 3600

┌──(kali㉿kali)-[~/Desktop/Assignment 1/Mandelbrot/Sequential]
└─$ ./sequential 3800
Computation time: 8.015054 seconds
Iteration Number: 3800

┌──(kali㉿kali)-[~/Desktop/Assignment 1/Mandelbrot/Sequential]
└─$ ./sequential 4000
Computation time: 8.460889 seconds
Iteration Number: 4000
```



we notice that the computation time is increasing almost linearly with a bit of fluctuations as the number of iterations is increasing.

Dynamic Code

The code used runs MPI to Parallelize the computation across 4 processors. Master process here coordinates the computation and collect results from the “slave” processes while each perform part of the computation. Here is a break down of the dynamic aspects of the code:

1. **Dynamic Workload Allocation**:
    - In the master process (`master_proc` function), rows of pixels are assigned to slave processes dynamically within a loop.
    - The loop sends rows to slave processes for computation using `MPI_Send`.
    - The master process keeps track of the number of running tasks and sends more work as needed.
    - This dynamic distribution of rows ensures that each slave process receives work continuously without waiting for all others to finish.

```
cCopy code
while (running_tasks > 0) {
    // ...
    // Dynamically assign rows to slave processes.
    // ...
}
```

2. **Adaptive Load Balancing**:
    - Adaptive load balancing is achieved by assigning rows to slave processes as soon as they become available.
    - The code ensures that slave processes don't remain idle by assigning more rows as they request additional work.
    - Load balancing is maintained throughout the computation.
3. **Progressive Row Processing**:
    - Rows of pixels are processed incrementally, and as each slave process completes its assigned work, it requests the next block of rows.

- The master process responds to these requests by assigning more rows dynamically, allowing for continuous progress.

```
cCopy code
while ((MPI_Recv(rows, opts->blocksize, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &sta
tus) == MPI_SUCCESS) && status.MPI_TAG == MO_CALC) {
    // ...
    // Process rows and request more work dynamically.
    // ...
}
```

4. **Communication Overheads**:
    - Communication overhead is introduced by the use of MPI communication functions (`MPI_Send` and `MPI_Recv`) to exchange data and coordinate work distribution.
    - The code measures communication time (`comm_time`) to evaluate the impact of these overheads.
5. **Progress Tracking**:
    - The code includes an option to print a progress bar to track the progress of the computation.
    - The progress bar dynamically updates as rows are processed, providing feedback on the ongoing computation.

```
cCopy code
if (opts->show_progress) {
    // ...
    // Print a dynamic progress bar.
    // ...
}
```

These dynamic aspects collectively enable efficient parallelization of the Mandelbrot set computation by ensuring that work is distributed dynamically among processes, load is balanced, and progress is continuously tracked.

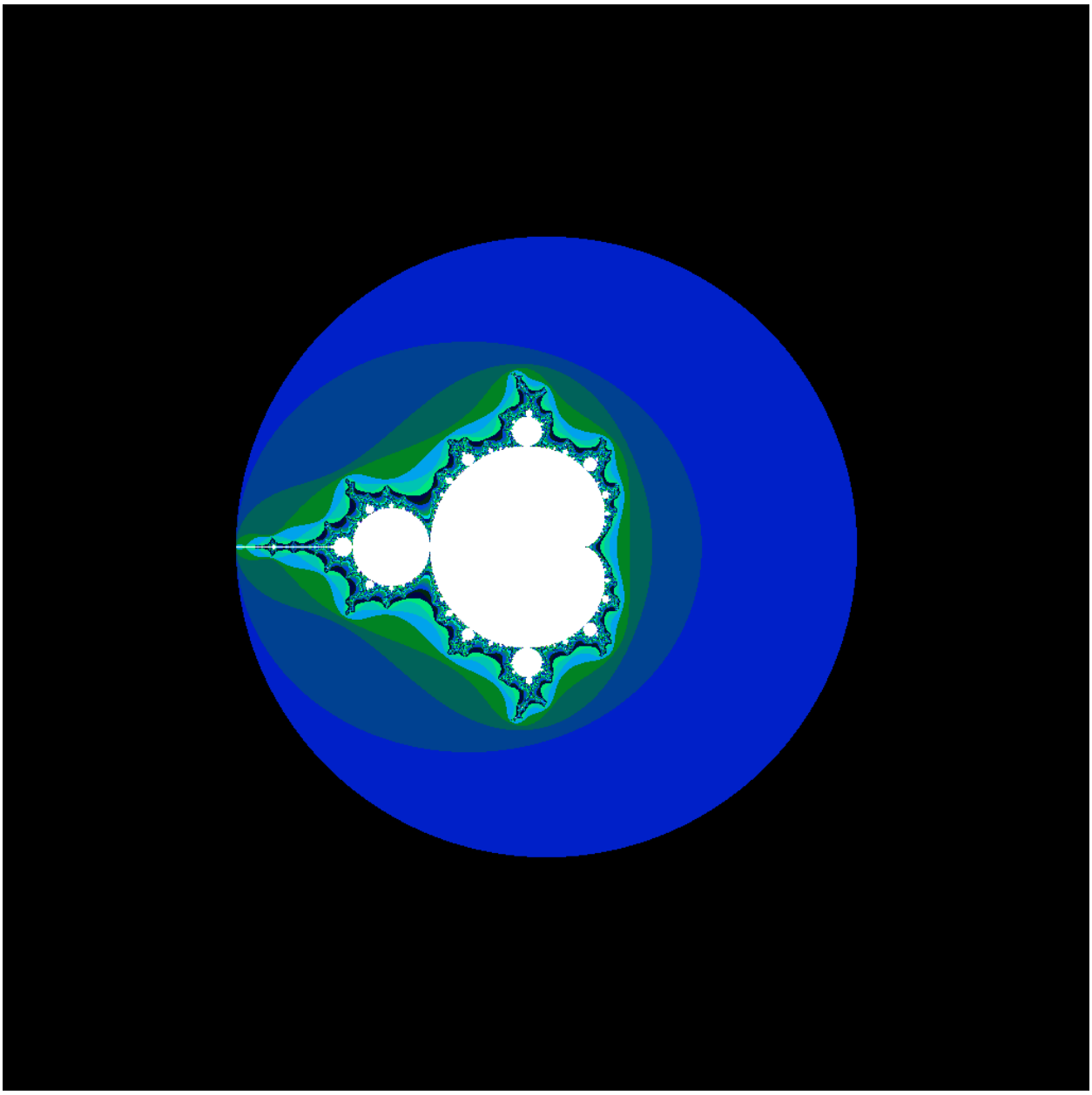
Output

Same 15 tests with iterations ranging from (1000 to 4000) were run using 2, 3 and 4 processors as follows:

Using 4 Parallel processors

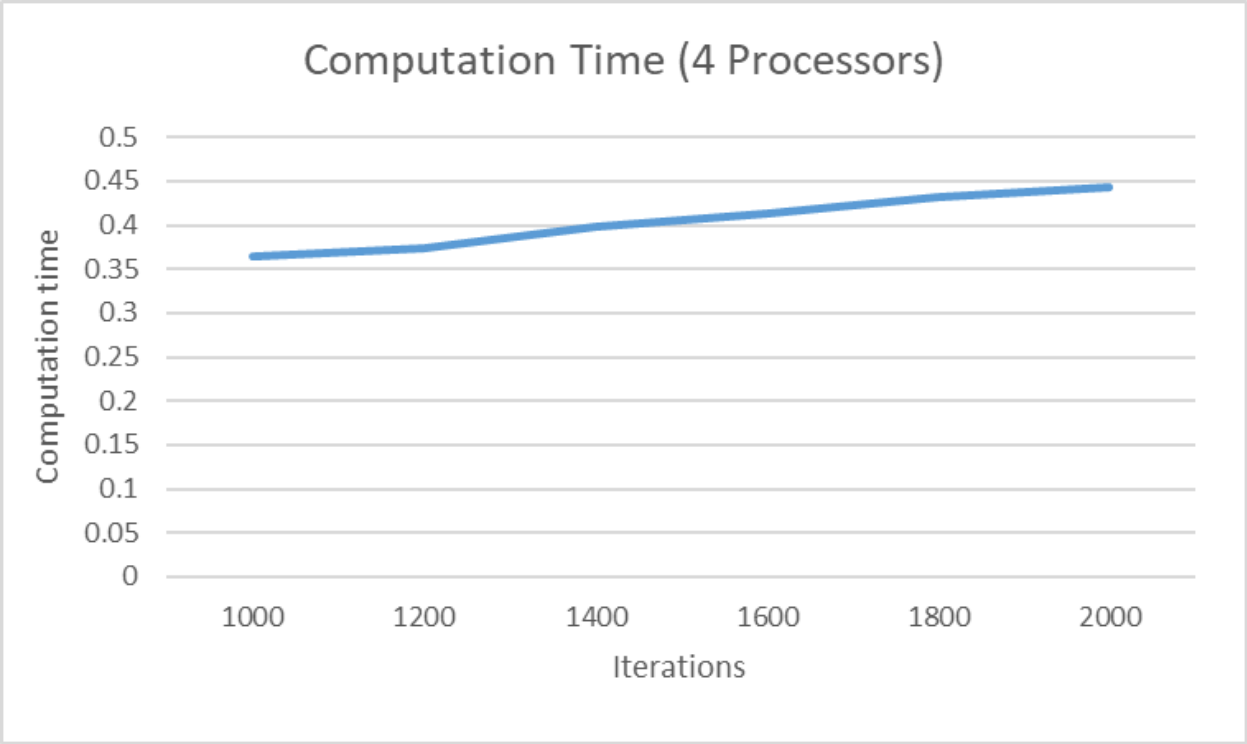
```
File  Actions  Edit  View  Help
┌──(kali㉿kali)-[~/…/Assignment 1/Mandelbrot/Parallel/dynamic]
└─$ mpicc mandelbrot.c -o mandel

┌──(kali㉿kali)-[~/…/Assignment 1/Mandelbrot/Parallel/dynamic]
└─$ mpirun -np 4 mandel 1000  -a 3.5
Computation parameters:
    output file              ./mandelbrot.bmp
    maximum iterations       2000
    blocksize                1
    image width              1024
    image height             1024
    minimum color            0×000000
    maximum color            0×ffffff
    color mask               0×ffffff
    x-offset                 0
    y-offset                 0
    axis length              3.5
    coordinate system range  [-3.5, 3.5]

Computation started.
Finished. Computation finished in 0.379133 sec.
Finished. Communication finished in 0.346119 sec.

Creating bitmap image.
Finished. Image stored in './mandelbrot.bmp'.
```

Image Generated at 1000 iterations



We repeat the test 14 times but with a cap of 2000 iterations as recommended by the code author:

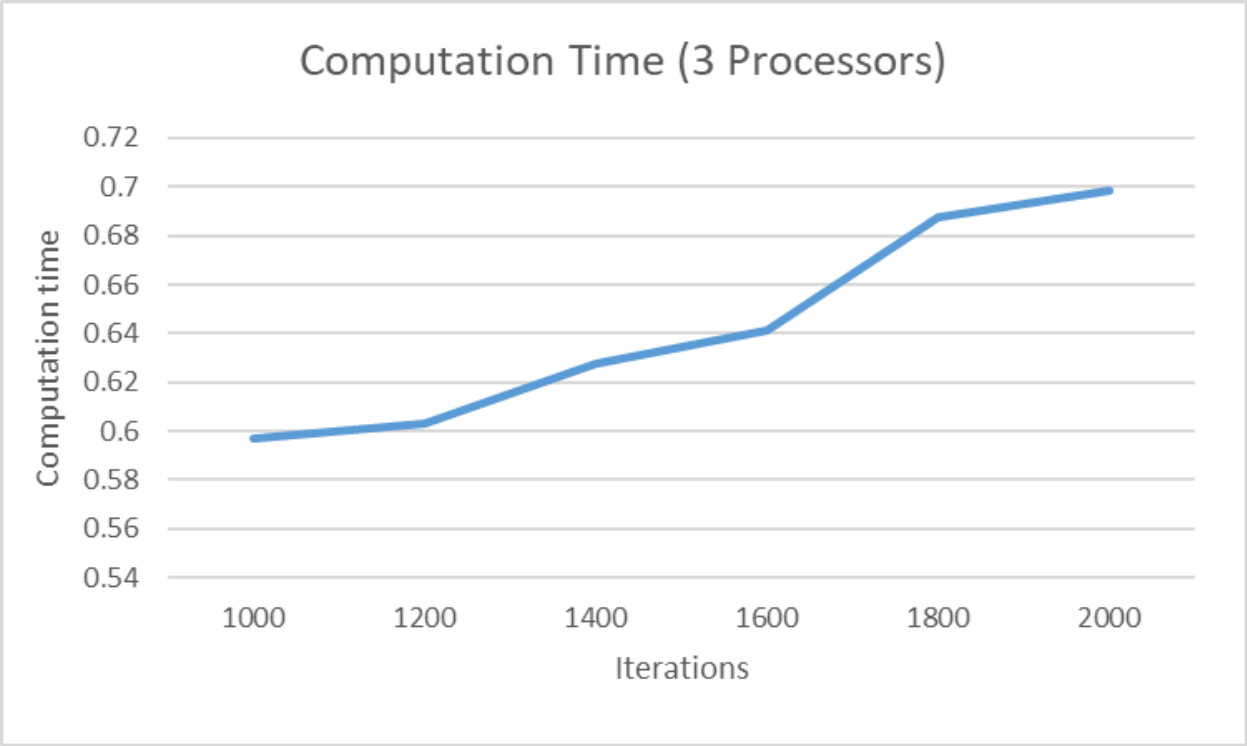
and we get the following results:



Now repeating the same test on 3 parallelized processors:

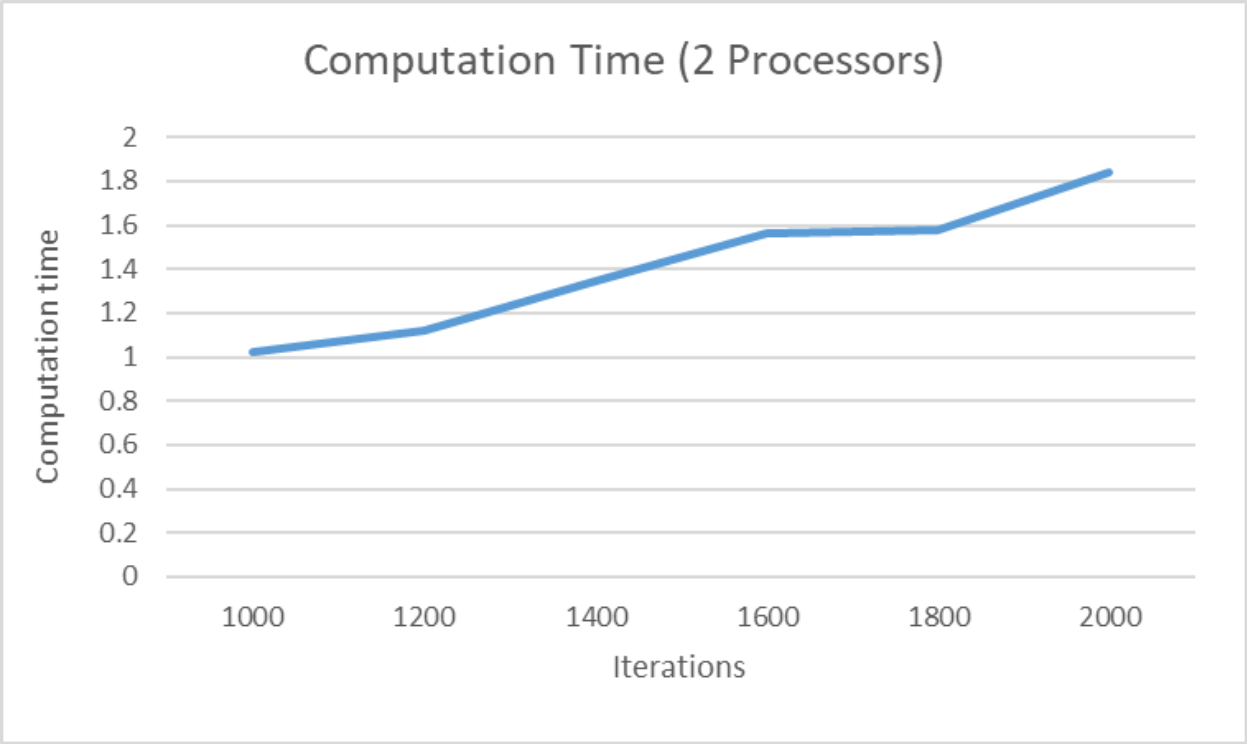
```
┌──(kali㉿kali)-[~/…/Assignment 1/Mandelbrot/Parallel/dynamic]
└─$ mpirun -np 3 mandel 1000 -a 3.5
Computation parameters:
    output file              ./mandelbrot.bmp
    maximum iterations       2000
    blocksize                1
    image width              1024
    image height             1024
    minimum color            0×000000
    maximum color            0×ffffff
    color mask               0×ffffff
    x-offset                 0
    y-offset                 0
    axis length              3.5
    coordinate system range  [-3.5, 3.5]

Computation started.
Finished. Computation finished in 0.597193 sec.
Finished. Communication finished in 0.568795 sec.
```

Repeating the test up to 2000 iterations we get the following results:



finally with two parallelized processors:

```
┌──(kali㉿kali)-[~/…/Assignment 1/Mandelbrot/Parallel/dynamic]
└─$ mpirun -np 2  mandel 1000 -a 3.5
Computation parameters:
    output file              ./mandelbrot.bmp
    maximum iterations       2000
    blocksize                1
    image width              1024
    image height             1024
    minimum color            0×000000
    maximum color            0×ffffff
    color mask               0×ffffff
    x-offset                 0
    y-offset                 0
    axis length              3.5
    coordinate system range  [-3.5, 3.5]

Computation started.
Finished. Computation finished in 1.02635 sec.
Finished. Communication finished in 1.00068 sec.

Creating bitmap image.
Finished. Image stored in './mandelbrot.bmp'.
```



Static

This code demonstrates a static parallelization approach for computing the Mandelbrot set, where each MPI process is responsible for a fixed portion of the grid, and the workload is evenly divided among processes. The results are gathered at the end for image generation. Timing measurements provide insights into the performance of the code

Here is a detailed break down of the used static code:

1. **Maximum Iterations and Grid Size**:
    - `#define MAXITER 1000`: This macro defines the maximum number of iterations for the Mandelbrot set computation.
    - `#define N 666`: This macro defines the size of the grid, which determines the resolution of the Mandelbrot set image.
2. **Variable Declarations**:

- `int i, j, k, loop, groupSize, rank, ncpu, pos;`: These variables are declared to store various indices and values used in the computation.
- `float *x, *y;`: These arrays are declared to store computed values (`x`) and the final Mandelbrot set image data (`y`).
- `float complex z, kappa;`: These complex numbers are used in the Mandelbrot set computation.
- `FILE *fp; int green, blue;`: These variables are used for file handling and color calculations.

3. **MPI Initialization**:
   - `MPI_Init(&argc, &argv);`: MPI (Message Passing Interface) is initialized to enable parallel computing.
   - `MPI_Comm_rank(MPI_COMM_WORLD, &rank); MPI_Comm_size(MPI_COMM_WORLD, &ncpu);`: The rank and size of the MPI communicator are determined to identify each process and the total number of processes.
4. **Workload Division**:
   - `groupSize = N * N / ncpu;`: The total workload is divided equally among MPI processes based on the grid size (`N`) and the number of CPU cores (`ncpu`).
5. **Memory Allocation**:
   - `x = (float *)malloc(groupSize * sizeof(float));`: Memory is allocated dynamically for the `x` array, which stores computed values.
   - `if (rank == 0) { y = (float *)malloc(N * N * sizeof(float)); }`: Memory is allocated for the `y` array to store the final Mandelbrot set image data. This allocation is done only by the rank 0 process.
6. **Mandelbrot Set Computation**:
   - The Mandelbrot set computation is performed within a loop, and each MPI process computes a portion of the Mandelbrot set based on its rank and `groupSize`.
   - The results are stored in the `x` array.
7. **Timing Measurements**:

- `MPI_Wtime()` is used to measure the time taken for various parts of the computation, including the Mandelbrot set calculation, waiting time, and communication time.
- These time measurements are used to profile the performance of the code.

8. **Data Gathering**:
   - `MPI_Gather(x, groupSize, MPI_FLOAT, y, groupSize, MPI_FLOAT, 0, MPI_COMM_WORLD);`: The computed values (`x`) from all MPI processes are gathered to the rank 0 process (`0`) and stored in the `y` array.
9. **Printing Timings and Finalization**:
   - Timings for calculation, waiting, and communication are printed for each MPI process.
   - The rank 0 process writes the Mandelbrot set image data to a file named "mandelbrot1.ppm" based on the computed values in the `y` array.
   - Memory allocated for arrays is released, and MPI is finalized.

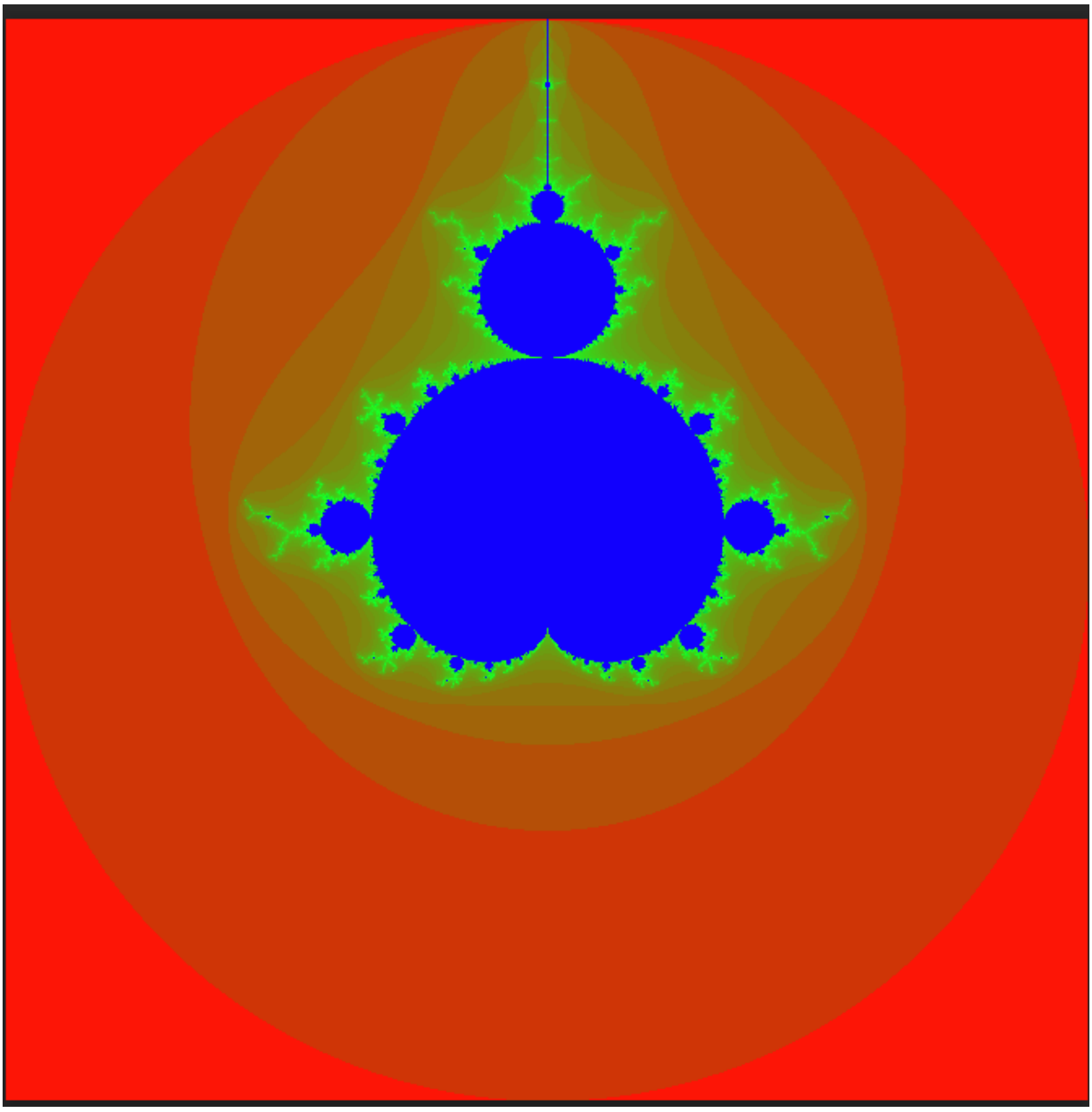
Now for our testing, we manually redefine the max iterations possible for run:

```
File  Edit  Search  View  Document  Help
 1 #define MAXITER 1000
 2 #define N 666
 3
 4 #include <mpi.h>
 5 #include <stdio.h>
 6 #include <stdlib.h>
 7 #include <math.h>
 8 #include <complex.h>
 9
10 /* ------------------------------------------------------------
11
12 int main(int argc, char*argv[]) {
13     int    i, j, k, loop, groupSize, rank, ncpu, pos;
14     float  *x, *y;
15     float complex   z, kappa;
16
17
18     FILE   *fp;
```

The results obtained shows a clear example of load imbalance between the processors as processor 1 is taking almost 4 times the time processor 2 is taking:

```
Process 0
        Calc: 5.116614 s
        Wait: 0.000102 s
        Comm: 0.002921 s
Writing mandelbrot1.ppm
Process 1
        Calc: 1.983047 s
        Wait: 3.133779 s
        Comm: 0.002915 s
```

The image generated with 2 parallelized processors :



3 parallel processors - static

```
┌──(kali㉿kali)-[~/…/Assignment 1
└─$ mpirun -np 3 mandel
Process 1
        Calc: 2.903823 s
        Wait: 0.000088 s
        Comm: 0.000454 s
Process 0
        Calc: 0.577561 s
        Wait: 2.326339 s
        Comm: 0.001480 s
Writing mandelbrot1.ppm
Process 2
        Calc: 0.011592 s
        Wait: 2.892351 s
        Comm: 0.001453 s
```

4 parallel processors - static

```
└─$ mpirun -np 4  mandel
Process 1
        Calc: 2.491282 s
        Wait: 0.000073 s
        Comm: 0.000518 s
Process 2
        Calc: 1.014096 s
        Wait: 1.477320 s
        Comm: 0.000909 s
Process 0
        Calc: 0.273706 s
        Wait: 2.217662 s
        Comm: 0.001842 s
Writing mandelbrot1.ppm
Process 3
        Calc: 0.006471 s
        Wait: 2.484894 s
        Comm: 0.001850 s
```

we notice from the previous tests that there is an apparent improvement in the computation time. However, we still have a huge overhead in communication in oppose to the dynamic output.

Speed up Factor:

In order to get the most reliable speed up factor in our case; we fix the number of iterations at 1000 for all tests and compare computation time with best sequential time acquired.

for that purpose: we have 6 different speedup factors for the test cases used:

1- Using 2 processors dynamically

Speedup Factor

$$S(p) = \frac{\text{Execution time using one processor (best sequential algorithm)}}{\text{Execution time using a multiprocessor with } p \text{ processors}} = \frac{t_s}{t_p}$$

referring back to the formula we get the following results

$S(p) = 2.108884/1.02635$ = 2.05474

2- Using 3 processors dynamically

$S(p) = 2.108884/0.597193$ = 3.531327393

3- Using 4 processor dynamically

S(p) = 2.108884/ 0.363935 = 5.794672

4- using 2 processors statically

$S(p) = 2.108884/5.116615$ = 0.41126

5- Using 3 processors statically

$S(p) = 2.108884/2.903823$ = 0.72624

6- Using 4 processors statically

$S(p) = 2.108884/2.491282$ = 0.81

Discussion

As we saw earlier, the computation of Mandelbrot Set using dynamic and static approaches gave varying degrees of speedup depending on the method used and number of parallel processors. Dynamically, utilizing 3 and 4 processors in parallel gave excellent speedup factors of around 3.53 and 3.8 respectively, which suggest efficient parallelism and almost linear scaling. However, this result might indicate that sequential processing suffered lots of overhead and underutilization of resources, especially that this was run on a virtual machine with limited resources. On the other hand, the static approach with 2,3,4 processors resulted in worse times than the sequential one. This indicate issues such as load imbalance and communication overhead. It is needed to mention in this case that this issue might have caused mostly due to the virtual machine being scheduled by the CPU as it tries to execute simultaneous tasks. Notably, the dynamic approach demonstrated better scalability and more promising speedup, highlighting the significance of load balancing techniques and data distribution optimizations in achieving efficient parallel performance.

References

Please note that I have used these following tools to set up conduct the experiment, analysis and troubleshooting issues for this Assignment

1. ChatGPT.
2. Wikipedia.
3. Vinod Kumar Rajasekaran. "Impact of Virtual Machine Overhead on TCP Performance." Submitted to the graduate faculty of the Department of Computer Science, Clemson University, June 17, 2004.
4. Martin Ohmann. "Dynamic MPI Mandelbrot Algorithm." Copyright (C) 2015. This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation.