Joshua Caudill

CS6068

Assignment #2

Table of Contents

Table of Figures

Introduction

The goal of this assignment was to demonstrate the speedup of a Python script that generated the Mandelbrot Set for an image. The Ohio Supercomputer Center (OSC) was used to create an environment for demonstrating the speedup achieved by using Numba. Numba is an open-source, just-in-time compiler used to speed up Python scripts. Using the @jit decorator, Numba translates Python code into bytecode optimized for the given environment. The Python script used in this assignment was a good candidate for using Numba since it looped through NumPy arrays. Numba is especially good at generating efficient compiled code for scripts that use NumPy.

Tools

- generate_mandelbrot_set.py
- time, numpy, numba.jit, pylab.imshow, pylab.show
- Pitzer Desktop (1 GPU, 48 Cores, 1 Visualization Node)
- Python 3.6.6 :: Anaconda Custom (64-bit)

Code

```
"""
Starter code for the fast generation of the Mandelbrot Set.
"""


import time


import numpy as np
from numba import jit
from pylab import imshow, show


@jit(nopython=True)
def mandel(x, y, max_iters):
    """
    Computes the behavior of '0' under max_iters iterations for the value c.
    """
    c = complex(
        x, y
    )  # Generate complex number, c, given real and imaginary components.
    z = 0.0j
    for i in range(max_iters):
        z = z * z + c  # Compute the function.
        if (
```

```
            z.real * z.real + z.imag * z.imag
        ) >= 4:  # Return iteration value if z becomes larger than 4.
            return i

    return max_iters  # Return max_iters otherwise.

@jit(nopython=True)
def create_fractal(min_x, max_x, min_y, max_y, img, iters):
    """
    The Mandelbrot Set is a fractal. Create the fractal.
    """
    height = img.shape[0]
    width = img.shape[1]

    # Calculate pixel sizes.
    pixel_size_x = (max_x - min_x) / width
    pixel_size_y = (max_y - min_y) / height

    for x in range(width):
        real = min_x + x * pixel_size_x
        for y in range(height):
            imag = min_y + y * pixel_size_y
            color = mandel(real, imag, iters)
            img[y, x] = color  # Assign a color to the image.

if __name__ == "__main__":
    image = np.zeros((1024, 2024), dtype=np.uint8)  # Generate the image.
    start = time.time()  # Get the start time.
    create_fractal(-2.0, -1.7, -0.1, 0.1, image, 20)
    end = time.time()  # Get the end time.
    print(f"Elapsed = {(end - start)}")  # Print the elapsed time.
    imshow(image)
    show()  # Show the Mandelbrot Set.
```

Results

The Python script, generate_mandelbrot_set.py, was executed in both sequential and parallel fashion. The Python script was executed in a sequential fashion with the @jit decorators commented, and it was executed in a parallel fashion with the @jit decorators uncommented. The following image shapes were tested with 20 iterations: (24, 1024), (1024, 2024), and (2024, 3024). The following number of iterations were tested with an image shape of (1024, 2024): 10 iterations, 20 iterations, and 40 iterations. The speedups shown below were calculated. The screenshots shown below were taken from OSC.

- $\text{Speedup } [(24, 1024), 20 \text{ Iterations}] = 0.05488 \text{ sec} / 0.2825 \text{ sec} = 0.1944$
  - Not an error. Strange behavior for low number of rows.
- $\text{Speedup } [(1024, 2024), 10 \text{ Iterations}] = 4.026 \text{ sec} / 0.2703 \text{ sec} = 14.89$
- $\text{Speedup } [(1024, 2024), 20 \text{ Iterations}] = 4.113 \text{ sec} / 0.3284 \text{ sec} = 12.52$
- $\text{Speedup } [(1024, 2024), 40 \text{ Iterations}] = 4.211 \text{ sec} / 0.2677 \text{ sec} = 15.73$
- $\text{Speedup } [(2024, 3024), 20 \text{ Iterations}] = 11.97 \text{ sec} / 0.3789 \text{ sec} = 31.59$

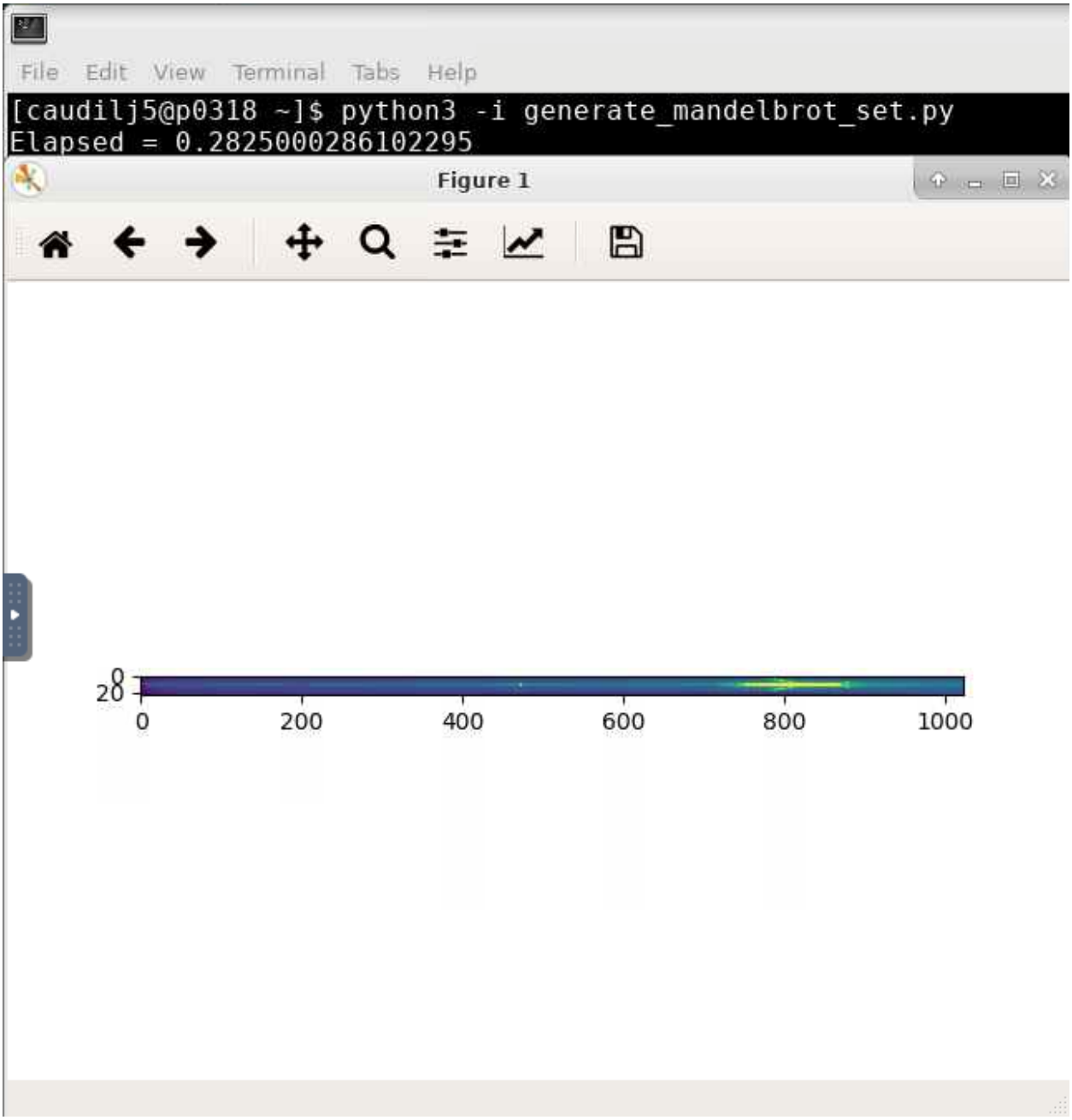

Figure 1: Parallel, (24, 1024), 20 Iterations

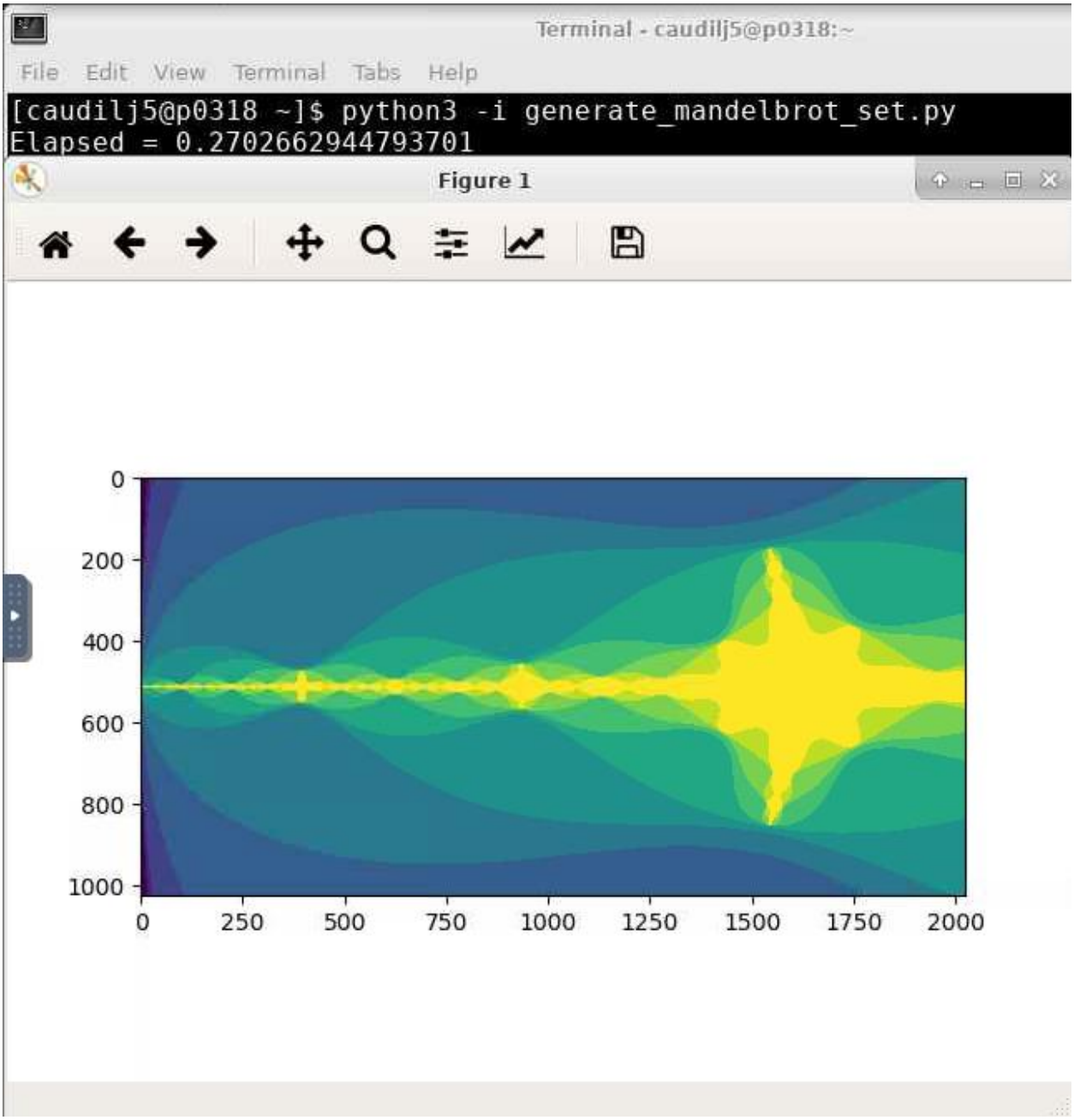

Figure 2: Parallel, (1024, 2024), 10 Iterations

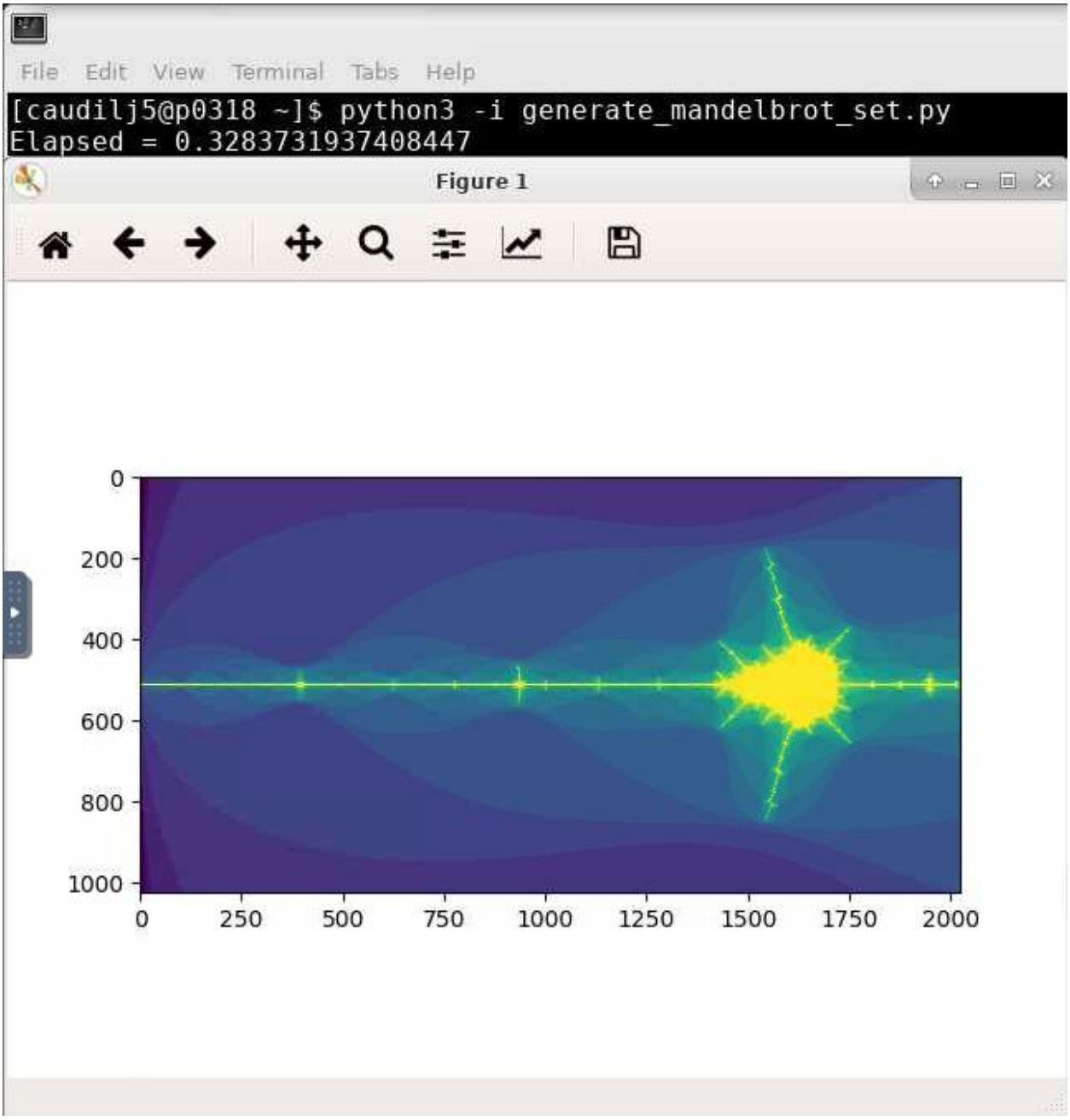

Figure 3: Parallel, (1024, 2024), 20 Iterations

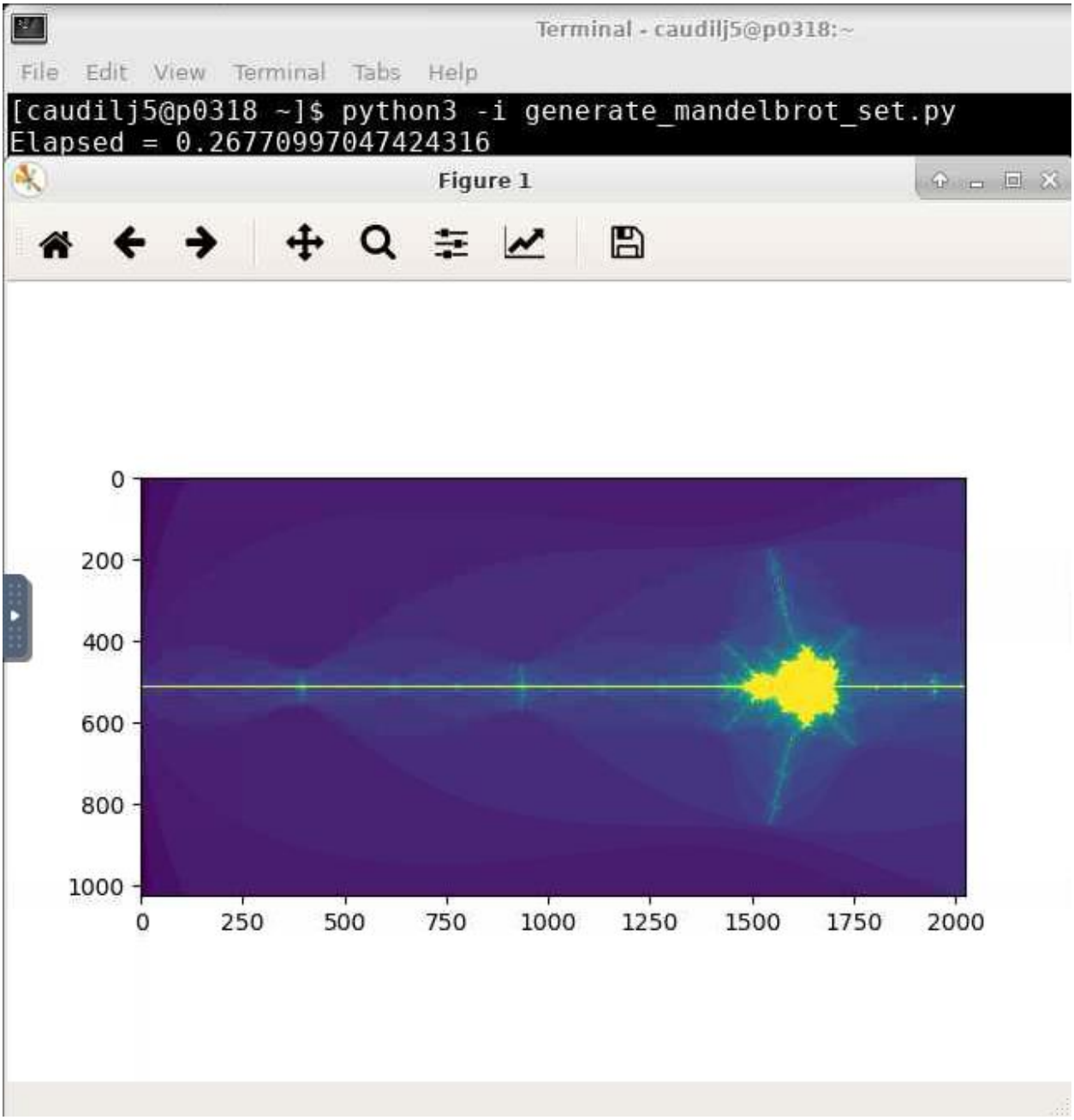

Figure 4: Parallel, (1024, 2024), 40 Iterations

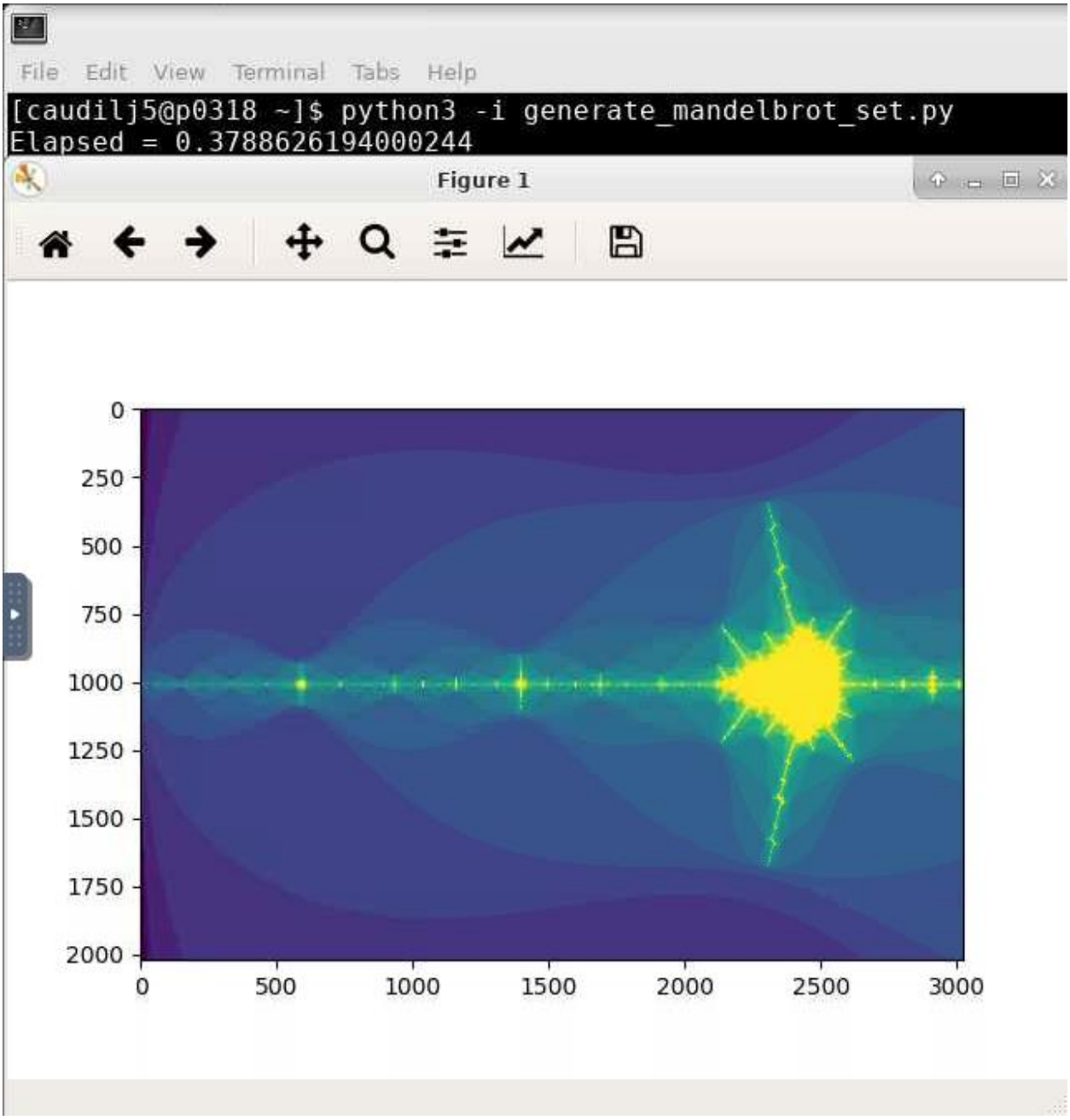

Figure 5: Parallel, (2024, 3024), 20 Iterations

Figure 6: Sequential, (24, 1024), 20 Iterations

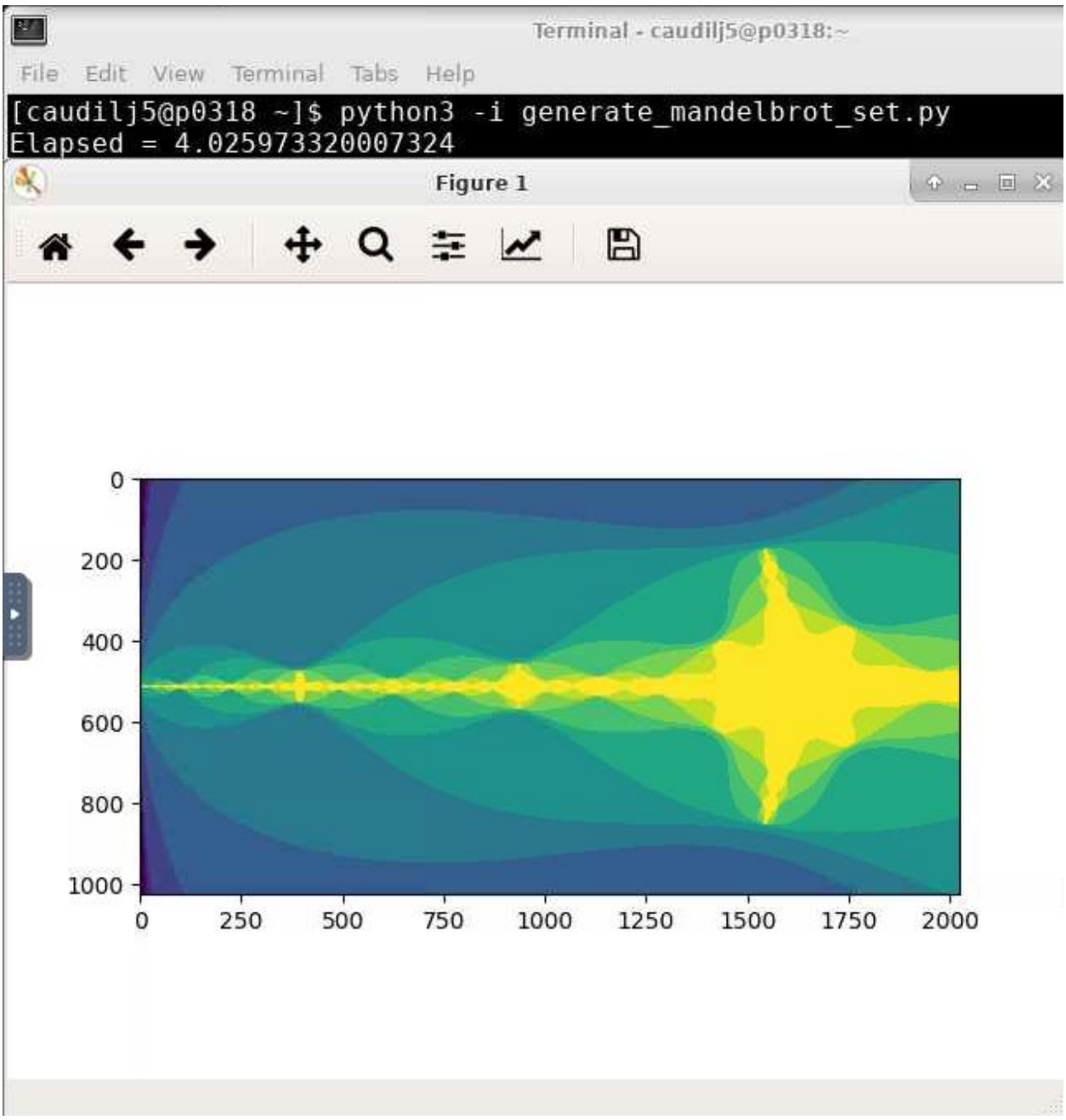

Figure 7: Sequential, (1024, 2024), 10 Iterations

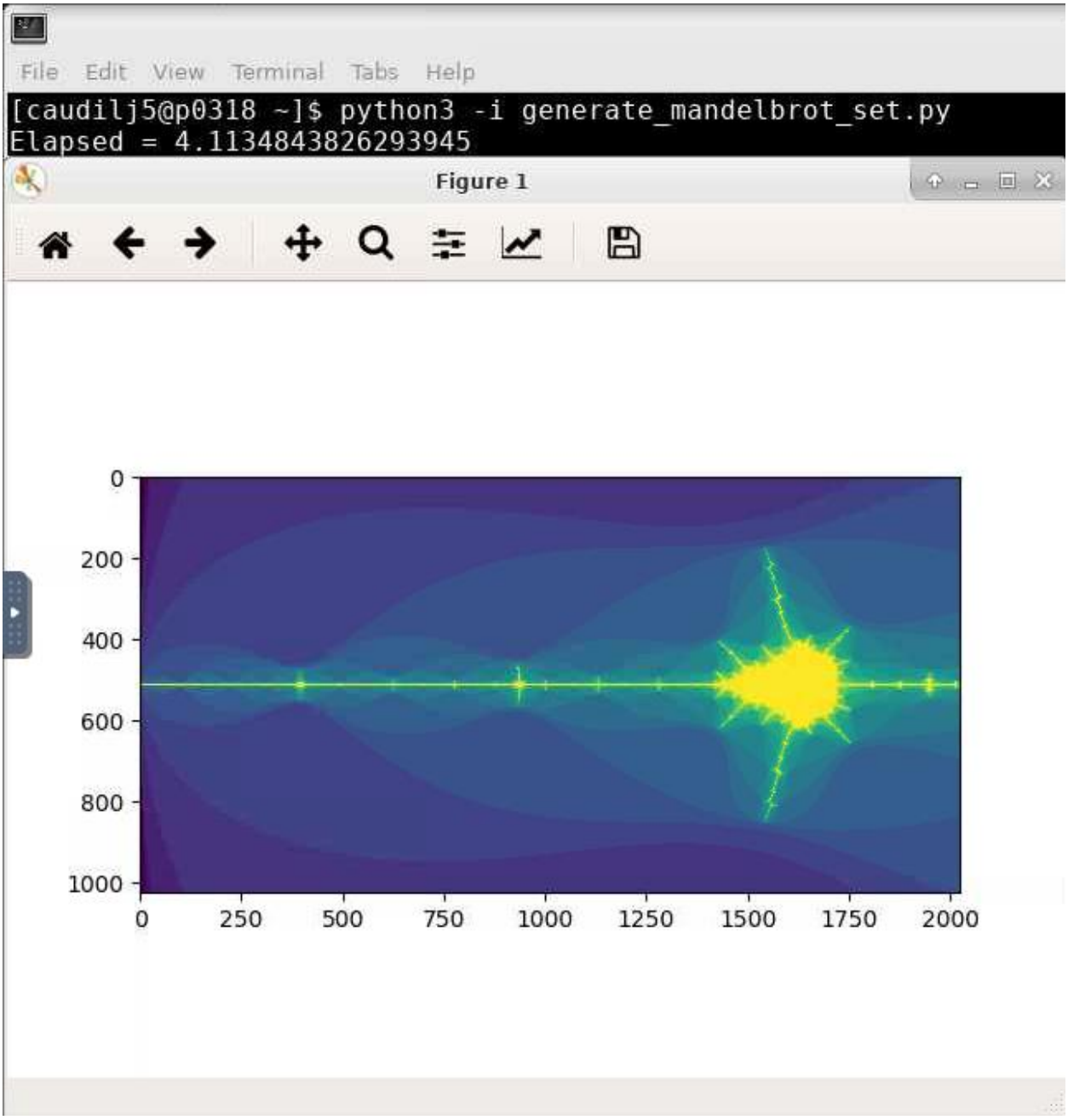

Figure 8: Sequential, (1024, 2024), 20 Iterations

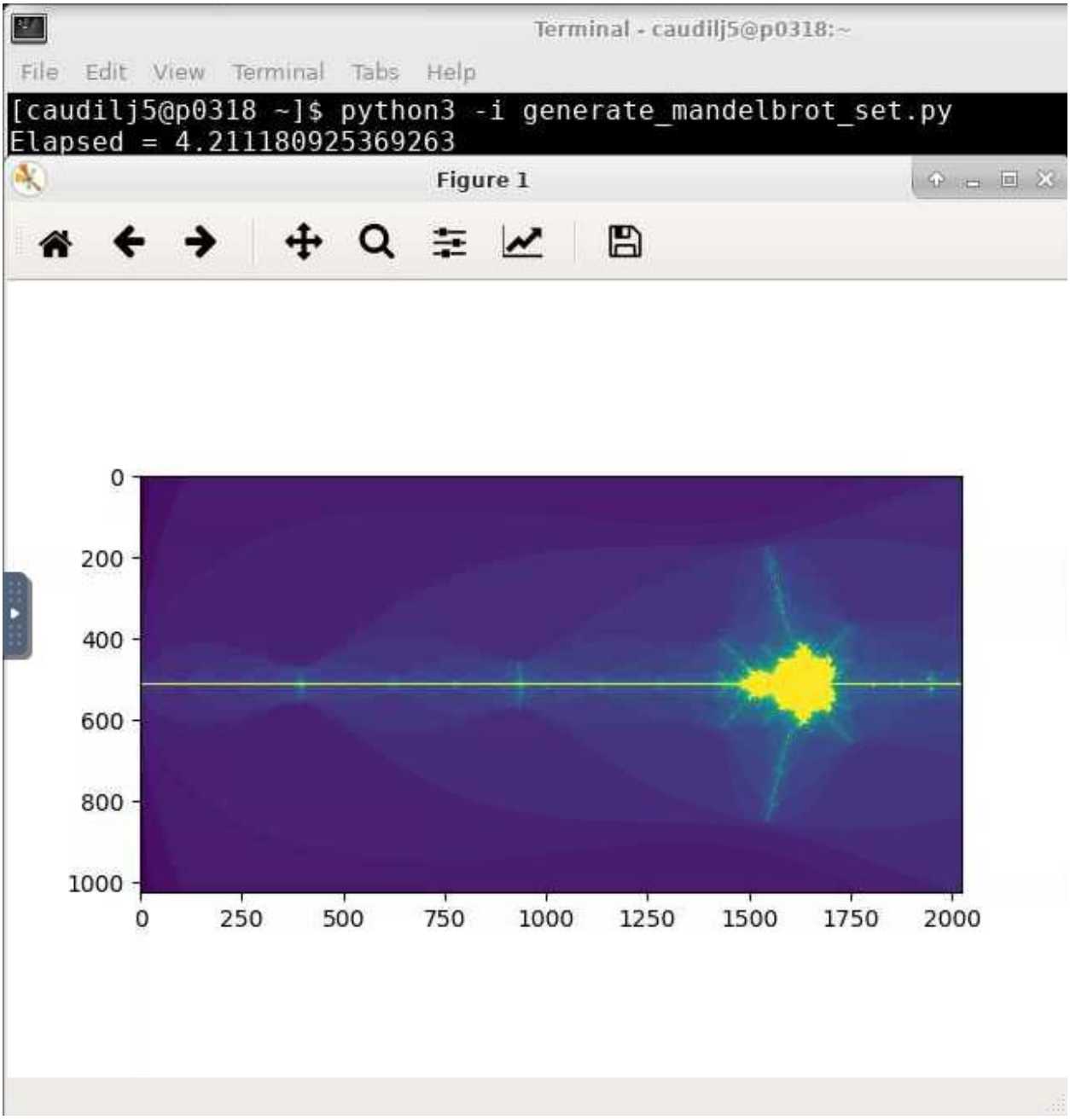

Figure 9: Sequential, (1024, 2024), 40 Iterations

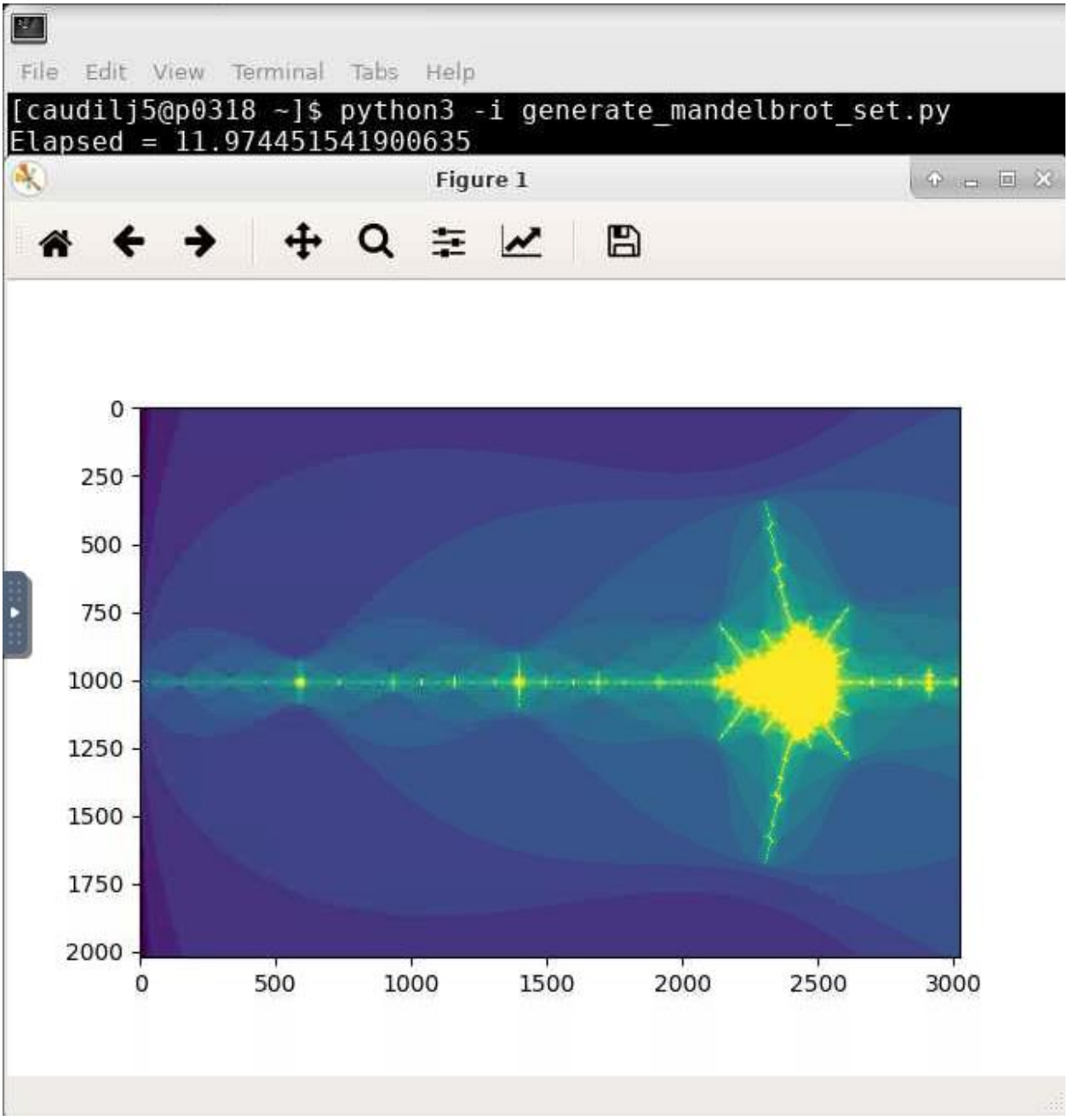

Figure 10: Sequential, (2024, 3024), 20 Iterations

Conclusion

Numba was used to speed up execution of generate_mandelbrot_set.py, which generated the Mandelbrot Set for an image. Using the @jit decorator resulted in a speedup in four out of five tests. In general, the speedup was larger for tests that performed more work (e.g., larger image size).