Optimizing the Computation of the Mandelbrot Set in Python

High school internship project

University of Basel
Faculty of Science
Department of Mathematics and Computer Science
High Performance Computing Group

Author: Oleh Chekhovych
Email: oleh.chekhovych@unibas.ch

February 27, 2025

Abstract

The Python programming language is easy to use and has a rich community. Python is an interpreted language, which slows its execution compared to compiled languages such as C. This report explores ways to overcome this slowness using the example of generating the Mandelbrot set. Tools such as Numba and PyOMP allow Python to close this speed gap with other programming languages. The results show that the computation can be speeded up by a factor of 146, with Numba giving the largest boost of 20, while parallelizing with a specific schedule with 8 threads gives a speedup of about 7. Some limitations of the tools were also found, including: support for only one scheduling type for PyOMP, and Numba's inability to apply the Numba compiler to methods of user-defined classes.

Contents

Chapter 1

Introduction

1.1 Motivation

The Mandelbrot set is one of the most famous fractals, and rightly so. It has a lot of beautiful shapes in it and can be zoomed in to create fascinating videos. Another advantage of this set is that it is easy to create using any programming language. To generate the set, you need to decide on two parameters: its resolution, i.e. the number of points, and max iterations, a parameter that determines the colouring. And when it comes to programming languages to program the set, what better choice than one of the most famous - Python. Mandelbrot sets can be generated quite easily with Python, that is, if the resolution and maximum iterations are limited. Generating high-resolution Mandelbrot sets, or zooming in on the most interesting parts of the set, is quite computationally intensive. Python will simply take too long compared to uninterpreted languages like C. However, Python has its advantages, such as simple syntax, easy installation and a diverse community. So the goal is to optimise the computation while staying in Python. To achieve this, there are some frameworks that optimise Python and put it on the same level as other languages. These frameworks and their comparison are the focus of this thesis.

1.2 Outline

The remainder of this report is organised as follows. The necessary background information is given in Chapter 2, how to generate the set in Python and how to improve its performance is explained in Chapter 3. Chapter 4 presents the evaluation results and Chapter 5 concludes the thesis and gives an outlook for future work.

Chapter 2

Background and Related Work

2.1 The Mandelbrot set

The Mandelbrot set is defined as the points C in the complex plane for which the recursively defined series $z_{n+1} = z_n^2 + C$ with $z_0 = c$ stays in circle $|z| <= 2$ under a maximum number of iterations n_{max}. The visualization of the Mandelbrot set is based on the coloring of pixels in the complex plane, where the x-axis represents the real number and the y-axis represents the imaginary unit of the complex number C. The colors symbolize how fast equation diverges. The representation of the set can be seen at Figure 2.1

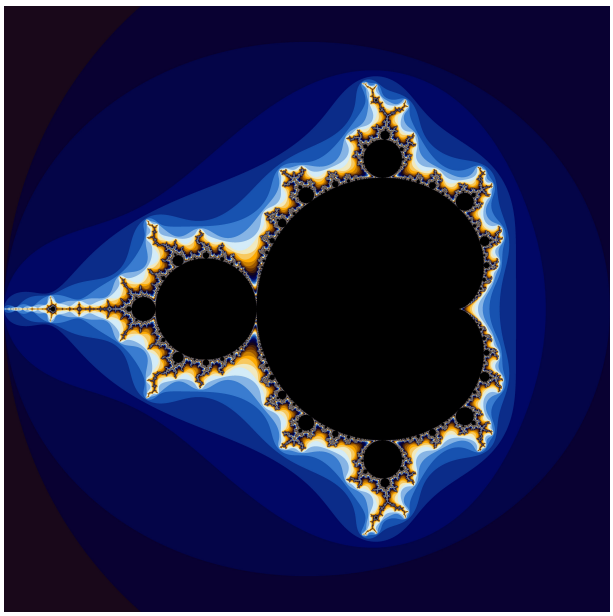

Figure 2.1: Mandelbrot set generated with 5000x5000 points resolution, 1000 maximum iteration. Custom colormaping was used

2.2 The Reasons for Python's Slowness

To understand how to speed up Python, it is necessary to understand why Python is slow and how it differs from other languages. The reasons are as follows:

- Python is an interpreted language, unlike fast languages like C. Interpreted languages require more steps to be executed, which makes them slow. Python must be compiled into byte code, which is then interpreted and executed line by line by a virtual machine. C, on the other hand, is compiled and executed directly by its compiler; there is no interpretation step.

- Another difference from C, and a reason that makes it slow, is that Python is dynamically typed. Although this makes the development process more convenient and faster, it comes at the cost of needing another type checking step.
- To handle threads, Python uses the Global Interpreter Lock (GIL), which locks program execution to only one thread. This behavior, while allowing concurrency, makes parallelism impossible.

2.3 Parallelization and OpenMP

One way of speeding up a program is to split the program's tasks and run them in parallel on multiple machines or on multiple cores of the CPU. These small tasks are stored and executed on threads. A running programme/process can have an unlimited number of threads, but their effectiveness depends on the number of CPU cores on the machine. This report focuses on optimising computations on a single machine, with a single CPU. Threads in a CPU usually share their memory, so we will focus on the shared memory model of parallelisation.
A popular framework called OpenMP (Open Multi-Processing) is based on this model. It works by using a set of constructs that tell how, where and when to parallelize code. Unfortunately, Python cannot use OpenMP directly, but using methods listed in this work, the ease of using OpenMP can be extended to Python.

2.4 PyOMP

PyOMP (1) is Numba based framework that allows to use OpenMP constructs easily. It however suffers from some installations constrains, it can only be installed on specific OS systems, namely linux-64 (x86 64), osx-arm64 (mac), linux-arm64, and linux-ppc64le, through conda forge, as stated in PyOMP github Installation guide (2). This framework will be the focus of this report.

2.5 Omp4py

Omp4py(3) is pure python implementation of OpenMP as their creators describe it. It allows to use OpenMP constructs with simple import statement and right python version, namely Python >= 3.13, where it is possible to disable GIL. This framework was not explored in this work due to some installation issues.

2.6 Numba

Since PyOMP requires Numba (4) to function, the influence of Numba on the optimization of the computation must be analyzed. Numba is a just-in-time compiler for Python that uses decorators (wrappers) to mark which functions to compile. When a Numba-decorated function is called, it is compiled into machine code for execution, and the marked code can then run at native machine code speed.
Numba reads the Python bytecode for a decorated function and combines it with information about the types of the function's input arguments. It analyses and optimises code, and finally uses the LLVM compiler library to generate a machine code version of function that is tailored to the CPU's capabilities. This compiled version is then used each time the function is called.
Not all functions can be compiled this way, there are several restrictions on the function. Some of them are:

- All data types must be supported and identified by the PyOMP compiler, although Numba can infer the type of most standard data types by analyzing the operations performed on them.
- Later it was discovered, and it also follows from the previous rule, that njitted functions cannot be methods of a user-defined class, they can only be functions.

Chapter 3

Methods

3.1 Computing Mandelbrot set with python

My implementation of computing the Mandelbrot set uses two functions: compute_points and mandelbrot_pixel, and parameter generating code. The description of these follows:

- The first function (refer to Code 3.1) takes as argument real and imaginary axes (here xDomain, yDomain); maximum iteration; 2d array filled with zeros and returns array filled with iteration, when it escaped the bound for each pixel. It has two loops, one nested inside the other. The outer loop iterates row by row, and starts the inner loop to iterate through each pixel. This inner loop calls the Mandelbrot pixel function to get the escaping iteration. The function ends when all iterations are done and the array is filled.

Code 3.1: Implementation of compute_points function in Python

```
1    def compute_points(xDomain, yDomain, max_iterations, iterationArray):
2        for y in yDomain:
3            for x in xDomain:
4                c = complex(x, y)
5                iterationArray[y, x] = mandelbrot_pixel(c, max_iterations)
6        return iterationArray
```

- Next function (refer to Code 3.2)takes a point in the complex space and the maximum count, and returns the iteration at which that point diverges (gets out of circle $|z| = 2$), or the maximum iteration if it does not diverge. This is done by simple loop and two return statements.

Code 3.2: Implementation of mandelbrot_pixel function in Python

```
1    def mandelbrot_pixel(c, max_iterations):
2        z = 0
3        for iterationNumber in range(max_iterations):
4            if abs(z) >= 2:
5                return iterationNumber
6            z = z**2 + c
7        return max_iterations
```

It is possible to combine functions into one big function with three nested loops, but to modulate the code and for future ease, I chose to have two functions.

3.2 Improving Performance with Numba

It is quite easy to make previous mentioned functions numba compiled. You just need to make sure that all your loops are iterating over `range()` and not something else, and add decorator @njit to the function definition. This implementation can be seen at Code 3.3, and Code 3.4

Code 3.3: Implementation of compute_points enhanced with numba

```
@njit
def compute_points(xDomain, yDomain, max_iterations, iterationArray):
    for y_i in range(len(yDomain)):
        for x_i in range(len(xDomain)):
            c = complex(xDomain[x_i], yDomain[y_i])
            iterationArray[y_i, x_i] = mandelbrot_pixel(c, max_iterations)
    return iterationArray
```

Code 3.4: Implementation of mandelbrot_pixel enhanced with numba

```
@njit
def mandelbrot_pixel(c, max_iterations):
    z = 0
    for iterationNumber in range(max_iterations):
        if abs(z) >= 2:
            return iterationNumber
        z = z**2 + c
    return max_iterations
```

3.3 Improving Performance with PyOMP

To parallelize the functions with PyOMP, the context manager 'with' is used. This statement allows you to use the constructs from OpenMP in a similar way as they are used in OpenMP. My implementation parallelizes the outer loop of compute_points and divides the work among a number of threads, which is passed as a new argument to the function. This implementation can be seen at Code 3.5

Code 3.5: Implementation of compute_points parallelized with OpenMP

```
@njit
def compute_points(xDomain, yDomain, max_iterations, iterationArray, use_omp, num_threads):
    omp_set_num_threads(num_threads)
    with omp('parallel'):
        with omp('for'):
            for y_i in range(len(yDomain)):
                for x_i in range(len(xDomain)):
                    c = complex(xDomain[x_i], yDomain[y_i])
                    z = mandelbrot_pixel(c, max_iterations)
                    iterationArray[y_i, x_i] = z
    return iterationArray
```

3.4 Improving Performance with better scheduling

The previous setup uses a parallel for loop with no scheduling specified, so it uses the default. The default scheduling of PyOMP and OpenMP most probably is the same, which is static scheduling with a chunk size of $number\_iterations/number\_threads$. PyOMP unfortunately does not support different types of scheduling, it only supports static scheduling, but the chunk size can vary, which leaves room for further optimization. The scheduling type and chunk size can be specified within the OpenMP construct. This can be seen in Code 3.6

Code 3.6: Implementation of compute_points specifying chunk size

```
    @njit
    def compute_points(xDomain, yDomain, max_iterations, iterationArray, use_omp, num_threads):
        omp_set_num_threads(num_threads)
        with omp('parallel'):
            with omp('for schedule(static, 1)'):
                for y_i in range(len(yDomain)):
                    for x_i in range(len(xDomain)):
                        c = complex(xDomain[x_i], yDomain[y_i])
                        z = mandelbrot_pixel(c, max_iterations)
                        iterationArray[y_i, x_i] = z
        return iterationArray
```

Chapter 4

Experimental Results and Discussion

Table 4.1: Design of factorial experiments to evaluate the performance of Mandelbrot set computation.

Factor	**Value**	**Properties**
Experiment Set	Baseline: Sequential Python	Number of Points: 100, 500, 2'000, 5'000; Maximum iterations: 100, 500, 1000;
	Compiling with Numba	Number of Points: 100, 500, 2'000, 5'000; Maximum iterations: 100, 500, 1000;
	Parallelizing with PyOMP	Number of Points: 100, 500, 2'000, 5'000; Maximum iterations: 100, 500, 1000; Number of threads 1-8; Chunk size: default, 1;
Computing system	MacBook Pro	Chip: Apple M2 Max, Total Cores; 12 (8 performance, 4 efficiency); 32 GB RAM; MacOS Sequoia version 15.3.1
Experiment	Repetitions	5
Metric	Parallel execution time	Average time per experiment across all repetitions (seconds)

In total there was $5 * ((4 * 3 * 2 + 4 * 3 * 8 * 2) = 1080$ Experiments

4.1 Results with normal Python

As expected, normal Python runs consistently very slowly. Runtime for most intensive configuration valued at 304.58 seconds.

4.2 Results with Numba

Numba makes Python code extremely fast, in our configuration Numba speeds up Python the most. However, Numba always takes a little time to compile the function. This time is negligible and only counts for the first call of the function. Runtime for most intensive configuration valued at 14,86 seconds, so approximately 20 times faster as normal Python.

4.3 Results with PyOMP

The runtime of code with PyOMP with default scheduling depends on the number of threads used, as shown in the Figure 4.1.

We see that as the number of threads increases, the runtime decreases at a steady rate. However, there is a strange relationship where odd numbers of threads are less effective at speeding up than even

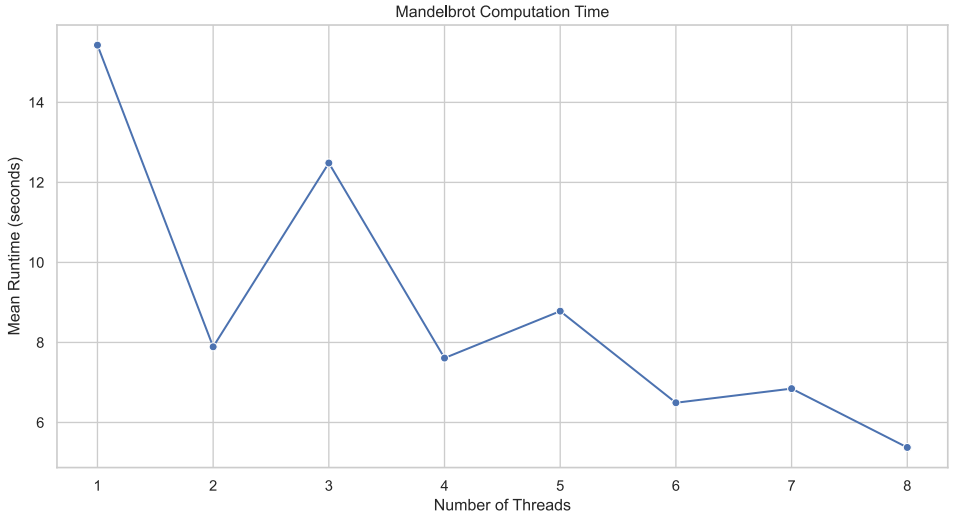

Figure 4.1: Runtime vs Threads, with parameters = 5000 number of points, 1000 maximum iterations.

numbers, and the overall speedup does not match the theoretical one. Theoretically, the runtime should decrease inversely proportionaly with the number of threads, but the result shows an irregular decrease. To explain this strange result, the work distribution of each thread was examined next. To measure the distribution of work among threads, each thread added the number of iterations it received from mandelbrot_calculate. At the end, there was an array with the total number of iterations performed by thread for all threads. The plots representing the results are shown in Figure 4.2

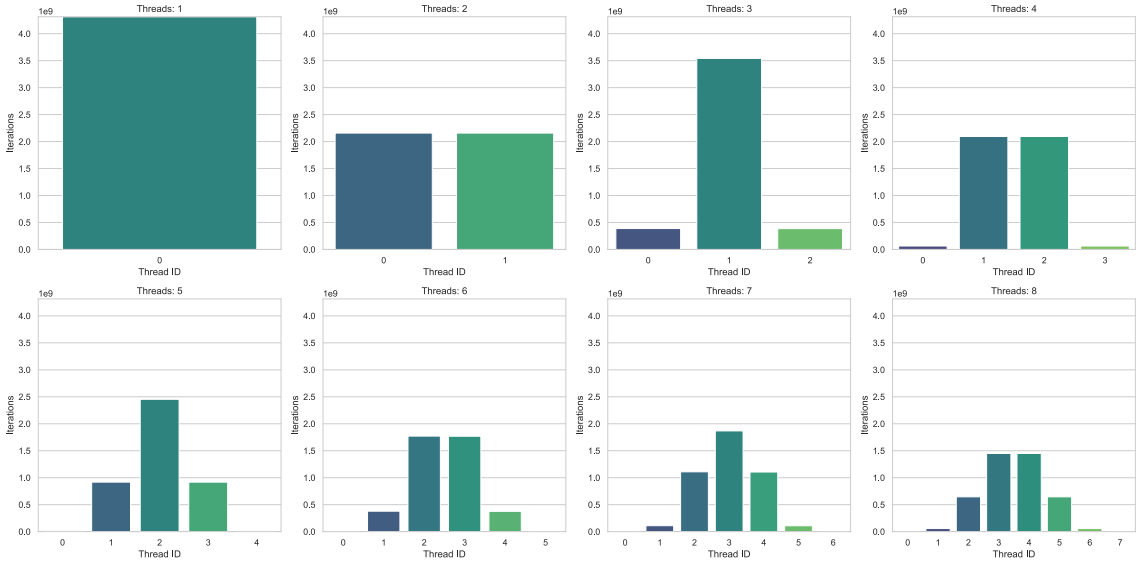

Figure 4.2: Iterations distribution plots for each threads range, with parameters = 5000 number of points, 1000 maximum iterations.

Looking at the figure, we can conclude that the work was not evenly distributed, some threads got regions that are less work intensive. For example, if we look at the third plot, which computed the Mandelbrot set with 3 threads, the 2nd thread most likely got the horizontal region in the middle of the set, where most pixels do not escape the set within 1000 iterations. The 1st and 3rd threads got the regions where the points escape much faster.

4.4 Varying chunk size

As we saw in the previous section, the work was not evenly distributed. To fix this, it was tried to vary the chunk size, as this is the only thing that can currently be changed in PyOMP's scheduling. Due to PyOMP's limitation of not being able to change the OpenMP construct at runtime and time constraints, only the chunk size of 1 and the default were tested. We can see the new runtime and iterations distributions plots, at Figure 4.3 and Figure 4.4

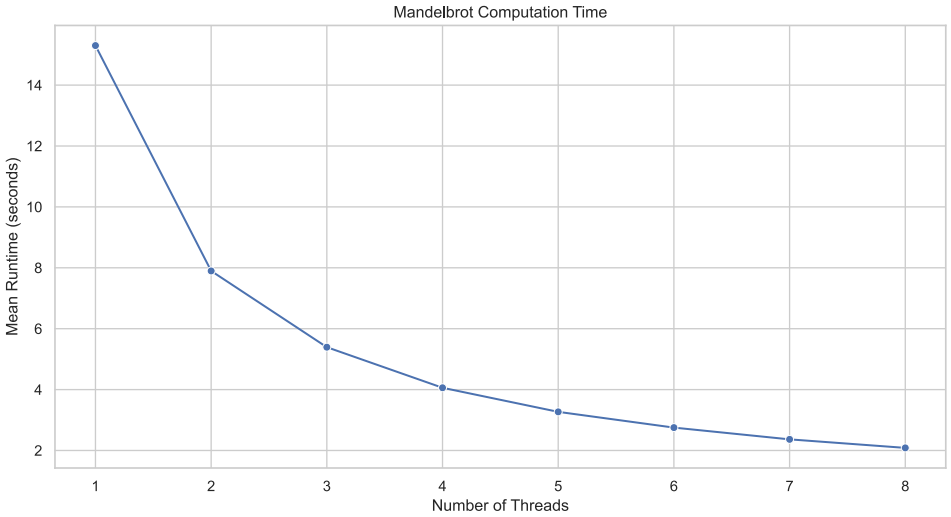

Figure 4.3: Runtime vs Threads, with parameters = 5000 number of points, 1000 maximum iterations.

We see that each thread now affects the runtime the way it should, namely the runtime is now inversely proportional to the number of threads. Two threads reduce the runtime by a factor of 2, three threads by a factor of 3, etc.

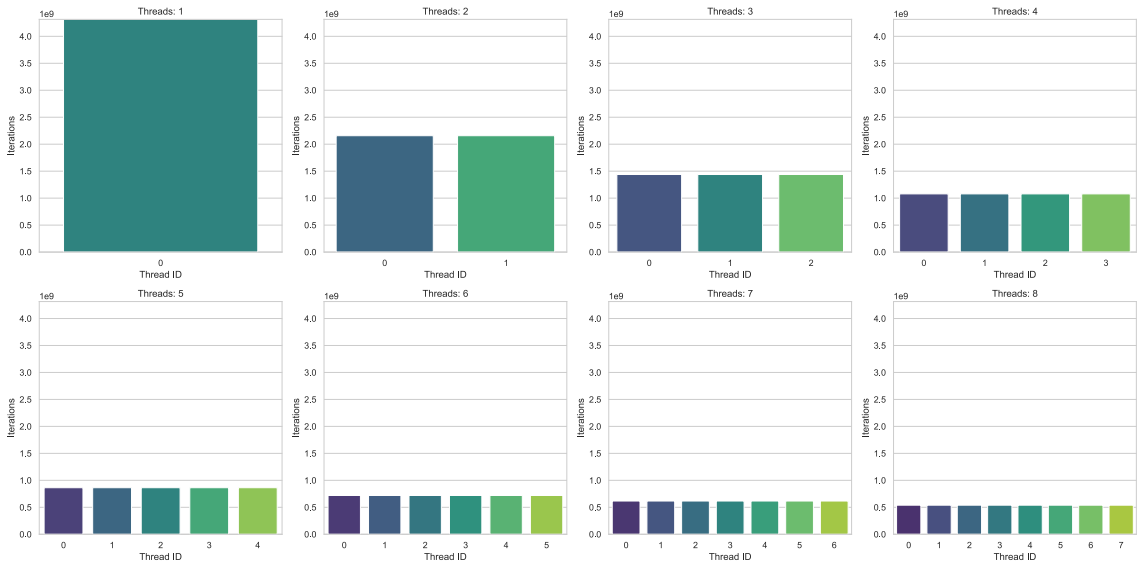

Figure 4.4: Iterations distribution plots for each threads range, with parameters = 5000 number of points, 1000 maximum iterations.

Looking at the Figure 4.4, we see that the work is now almost evenly distributed across all threads, which makes the computation faster.

4.5 Overall Results

Overall, we can document results in a Table 4.2. Relative speedup says how much faster the current implementation is than the previous one, absolute speedup says how much faster the current implementation is than Python's. Numba compilation time was not included into runtime.

Table 4.2: Table representing speedup from all experiments.
Parameters are as follows: num_points=5000, max_iterations=1000

Method	Runtime (s)	Relative speedup	Absolute speedup
Normal Python	304.58	1	1
Numba compiled	14.86	20.49	20.49
PyOMP (8 threads, default chunk size)	5.38	2.76	56.66
PyOMP (8 threads, chunk size 1)	2.08	2.58	146.04

Chapter 5

Conclusion

In conclusion, it is clear that Python can be massively accelerated by using just a few tools. This report was able to speed up the computation by a factor of 146, reducing the total runtime from hundreds of seconds to just a few seconds. In particular, Numba offers the possibility of a large speedup of a factor of 20. We also see that the Mandelbrot set can be significantly accelerated by parallelizing with PyOMP. 8 threads were used and with proper scheduling the code was speeded up by a factor of 8 in our configuration, but this is not the limit of what parallelization can achieve. With more threads, parallelization can speed up the computation even more.

5.1 Future Work

However, there is still room for improvement and further work. PyOMP does not yet support different scheduling types and varying chunk sizes dynamically at runtime, which may change to allow for finding and selecting the best scheduling. Also, this work has not explored other methods and tools that provide alternatives to PyOMP. These include OMP4Py mentioned in the Methods section, the Multiprocessing Python framework, and the PyPy framework.

Bibliography

[1] T. G. Mattson, T. A. Anderson, and G. Georgakoudis, "Pyomp: Multithreaded parallel programming in python," *Computing in Science Engineering*, vol. 23, no. 6, pp. 77–80, 2021.

[2] "Pyomp installation reference," Accessed 2025.

[3] C. Piñeiro and J. C. Pichel, "Omp4py: a pure python implementation of openmp," 2024.

[4] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: a llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, (New York, NY, USA), Association for Computing Machinery, 2015.