# Scientific Computing using Python 2: High Performance Computing (2025)

Marcell Richard Fekete
Aalborg University
Department of Computer Science
`mrfe@cs.aau.dk`

July 2025

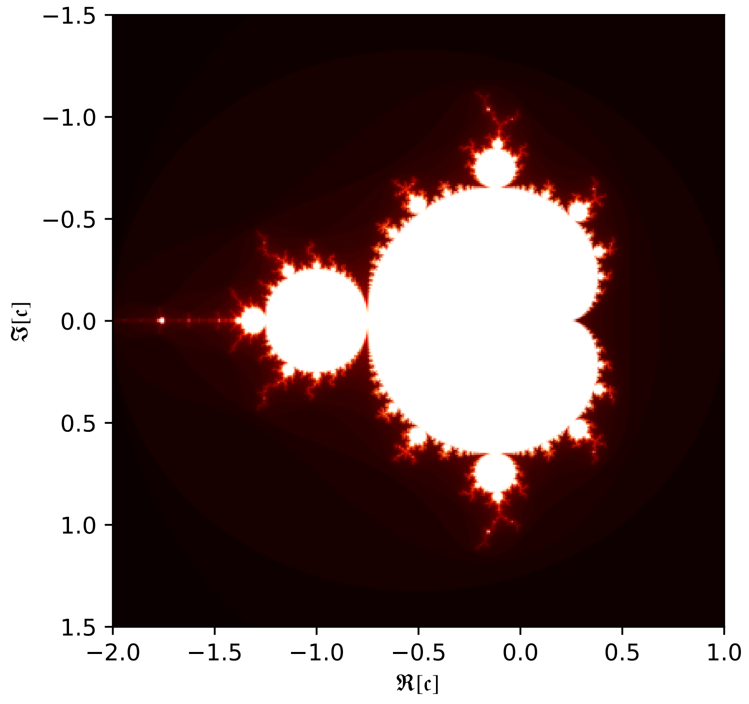

Figure 1: Mandelbrot set plotted using the `Dask` library with $p_{\text{re}} = 5000$, $p_{\text{im}} = 5000$, $I = 100$ and $T = 2$.

## 1 Introduction

The Mandelbrot set is a collection of $c$ points for which it is true that if the quadratic complex mapping in Equation 1 is applied for at least $I$ steps, the value of $z_I$ never reaches a set threshold $T$.

$$z_{i+1} = z_i^2 + c, \quad i = 0, 1, \ldots, I-1 \tag{1}$$

where $z_0 = 0$, and $c$ is a complex number. More specifically, the elements of a Mandelbrot set are mapped using a function $\mathscr{M}(c)$:

$$\mathscr{M}(c) = \frac{\imath(c)}{I}, \quad 0 < \mathscr{M}(c) \leq 1 \tag{2}$$

where $\imath(c)$ is defined as the first $i \in I$ at which the value of $z$ 'explodes', *i.e.*, reaches the threshold, normalised by the total number of $I$. For each $c$ that Equation 1 does not explode, the value of $\mathscr{M}(c)$ is defined as 1.0.

The Mandelbrot set is an interesting mathematical concept that can be used to generate two-dimensional fractal patterns, see Figure 1.

## 2 Algorithm

In this particular assignment, the goal is to calculate $\mathscr{M}(c)$ – the normalised escape value – of points of a $c$-mesh delimiting a complex matrix $\mathbf{C}$:

$$\mathbf{C} = \begin{bmatrix} -2.0 & \ldots & 1.0 \\ \vdots & & \vdots \\ -2.0 & \ldots & 1.0 \end{bmatrix} + j \cdot \begin{bmatrix} 1.5 & \ldots & 1.5 \\ \vdots & & \vdots \\ -1.5 & \ldots & -1.5 \end{bmatrix} \in \mathbb{C}^{p_{re} \times p_{im}} \tag{3}$$

See Algorithm 1 of how this is carried out.

**Algorithm 1** Calculating $\mathscr{M}(c)$ for all $c \in \mathbf{C}$

```
Initialise complex grid $\mathbf{C}$ of size $p_{\text{re}} \cdot p_{\text{im}}$
for $c \in \mathbf{C}$ do
    $z \leftarrow 0$
    for $i \leftarrow 1$ to $I-1$ do
        $z \leftarrow z^2 + c$
        if $|z| > T$ then
            $\mathscr{M}(c) \leftarrow \frac{i+1}{I}$
            break
        end if
    end for
    if $|z| \leq T$ then
        $\mathscr{M}c \leftarrow 1$
    end if
end for
```

## 3 Programming Solutions

For all solutions, I initialise **C** using the `NumPy` library:

```python
def init_meshgrid(
    real_val_lims: Tuple,
    imag_val_lims: Tuple,
    p_re: int,
    p_im: int
    ) -> np.ndarray:
    # initialise real and imaginary values
    real_vals = np.linspace(*real_val_lims, params.p_re)
    imag_vals = np.linspace(*imag_val_lims, params.p_im)

    # define the mesh grid
    Re, Im = np.meshgrid(real_vals, imag_vals)
    C = Re + 1j * Im

    return C
```

Additionally, I provided the same parameters to all of the three solutions described below: $p_{\text{re}} = 5000$, $p_{\text{im}} = 5000$, $I = 100$, $T = 2$, and the real and imaginary value limits of $[-2.0, 1.0]$ and $[1.5, -1.5]$, respectively. Parameters are passed via config files using the `TOML` format with certain arguments that can be overridden from the command line such as the number of workers and number of threads per workers. Passing parameters between functions involves the use of a custom `Parameters` dataclass except in the case of the generated meshgrid and the $\mathscr{M}(c)$ values of the Mandelbrot set.

All computations are carried out with the same MacBook Pro 14-inch laptop from 2021 with an Apple M1 Pro chip and 16GB of RAM.

## 3.1 Naive Solution

The naive computational solution follows Algorithm 1 in its exact implementation. Looping through each point in $\mathbf{C}$, it repeats the iteration of $z$ until either reaching a value over $T$ or reaching the last $I$. The code is simple but I made no particular effort to make it efficient or parallelised, hence using the general parameters, the computation time is 71-72 seconds.

```
1  def mandelbrot_eq(z: int|float, c: complex) -> complex:
2      return z**2 + c
3  
4  def iteration(c: complex, I: int, T: int|float) -> float:
5      z = 0
6      for i in range(1, I + 1):
7          z = mandelbrot_eq(z, c)
8          if abs(z) > T:
9              return i / I
10     return 1.0
11 
12 def compute_mandelbrot_set_naive(C: np.ndarray, params: Parameters):
13     output_array = np.zeros(C.shape[0] * C.shape[1])
14     for idx, c in enumerate(C.ravel()):
15         output_array[idx] = iteration(c, params.I, params.T)
16     return output_array.reshape(C.shape[0], C.shape[1])
```

Results of profiling show that the most costly computations in terms of time include looping through all $c \in \mathbf{C}$ (line 14), and carrying out all iterations of $z$ for $c$ (line 15).

## 3.2 Vectorised Solution

It is evident that the naive solution can be improved from sensible steps such as just-in-time compilation and especially vectorisation. Using a 'unfunc-like' kernel defined via the `vectorize` decorator of the `Numba` library lead to considerable performance improvements since it enables parallel calculation on each items of the $c$-mesh. Additionally, comparing the value of $z^2$ to $T^2$ is also more efficient than recomputing the square root of $z$ to carry out the same step. These improvements amount to decreasing the total runtime of the function to approximately 0.5 seconds.

```
@vectorize(
    ['float64(complex128, int64, float64)'],
```

```python
    target="parallel"
)
def compute_mandelbrot_set_vectorized(c: complex, I: int, T: int) -> float:
    z = 0j
    threshold = T * T  # speeding up processing not to use the square root
    for i in range(I + 1):
        z = z*z + c
        if (z.real*z.real + z.imag*z.imag) > threshold:
            return i / I
    return 1.0
```

Profiling shows that the program is quite efficient in a total time cost that is lower than either of the time-consuming steps of the naive solution.

## 3.3 Distributed Solution

The final implementation enables distributed computing of the Mandelbrot set via the Dask library. This may prove especially useful when initialising a meshgrid $\mathbb{C}$ with a considerably larger dimensionality than what our current $p_{\mathrm{re}}$ and $p_{\mathrm{im}}$ values define. Such a grid may not fit into a single computer and dividing $\mathbb{C}$ and computing $\mathscr{M}(c)$ parallel over computational units of a larger cluster may prove the only way to compute the Mandelbrot set.

```python
def compute_mandelbrot_set_distributed(C: np.ndarray,
                                       params: Parameters) -> np.ndarray:
    # define client for computations
    client = Client(threads_per_worker=params.threads_per_worker,
                    n_workers=params.n_workers)
    # define dask array chunks
    chunksize = C.shape[0] // params.n_workers
    dask_C = da.from_array(C, chunks=chunksize)
    # carry out computation
    output_array = da.map_blocks(
                        compute_mandelbrot_set_vectorized_wrapper,
                        dask_C, params, dtype=C.dtype)
    return output_array.compute(scheduler="processes")
```

However, as current implementation relies on the vectorised solution and the grid is not too large, we do not benefit from the potential performance improvements over simply applying Numba. See computation times in Table 1 with various worker and thread counts, confirming that starting too many workers or not having enough threads per worker harms execution time.

| Workers | Threads per worker | Time (in seconds) | Speed up |
|---:|---:|---:|---:|
| 1 | 16 | 5.07 | 14.34 |
| 2 | 8 | 6.04 | 12.04 |
| 4 | 2 | 6.12 | 11.88 |
| 4 | 4 | 5.34 | 13.62 |
| 4 | 8 | 5.22 | 13.92 |
| 4 | 16 | 5.53 | 13.13 |
| 8 | 2 | 6.74 | 10.78 |
| 16 | 2 | 8.46 | 8.59 |

Table 1: Execution time (in seconds) and speed up compared to the naive version when using varying number of `Dask` workers and threads per worker.

The profiling shows that the most costly operations include starting up the client (line 4), creating the `Dask` array from the meshgrid and chunking it (line 8), and calling the computation to finish (line 13).

## 4 Validation

Testing and validation is carried out to ensure both that all solutions work, as well as that they arrive to the same results given the same inputs.

**Visual Validation** The plots of the naive, the vectorised, and the distributed solutions are identical, confirming the similarity of the solutions.

**Validation of the Behaviour of Points** Constants $c$ such as $\{0,1\}$ belong the Mandelbrot set and never escape, thus all three solutions should return 1 for them. On the other hand, we expect $\{2, 1+1j\}$ to escape quickly, thus expecting $\mathscr{M}(c)$ values between 0 and 1. Finally, $c$ such as $\{-0.75, 0.001+0.001j\}$ are near the edge of the Mandelbrot set. Instead of expecting specific values, we only need the three solutions to converge in the value of $\mathscr{M}(c)$ with respect to these edge cases.

**Validation of Properties of the Mandelbrot Set** We know that the Mandelbrot set is symmetrical with respect to the real axis. This means that for all three implementations, we expect that $z$ for a value $c$ and its conjugate $c'$ is identical.

All the tests above confirm the required behaviours from all three solutions.