CE8 - AVS
Vasiliki Ismiroglou

Mandelbrot

Process:

Loop implementation

The naive version is a simple nested loop that runs through all the real and imaginary components one by one, to create the complex number and start the iterative process. The Numba implementation was very simple as the @jit decorator works as is.

Vectorized implementation:

The vectorized version is using ‘masking’. At every iteration only the elements that have not reached the threshold have the operation applied. Getting @jit functionality in this version was not straightforward. Functionalities such as 2D indexing are not supported by numba and for that reason the grid was flattened and reshaped.

Multiprocessing:

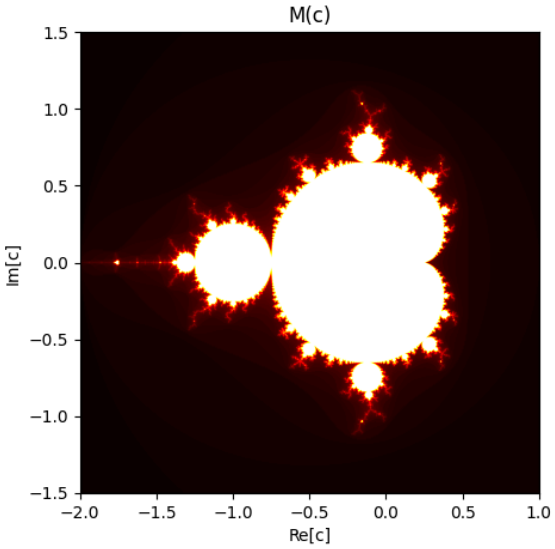
As the process is iterative, the calculation at each point in the complex plane could not be split between cores. Instead, the grid itself was split. The process was very similar for both loop and vectorized versions.

Configurations:

```
config = Namespace(name = None,
                   pre = 2000,
                   pim = 2000,
                   re_floor = -2,
                   re_ceiling = 1,
                   im_floor = -1.5,
                   im_ceiling = 1.5,
                   I = 100)
```

CE8 - AVS
Vasiliki Ismiroglou

Results



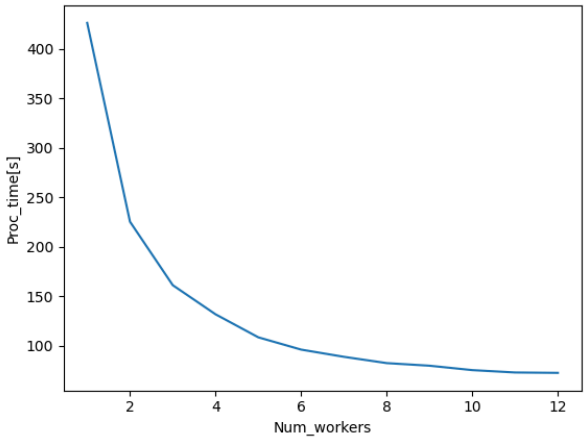


Figure 1

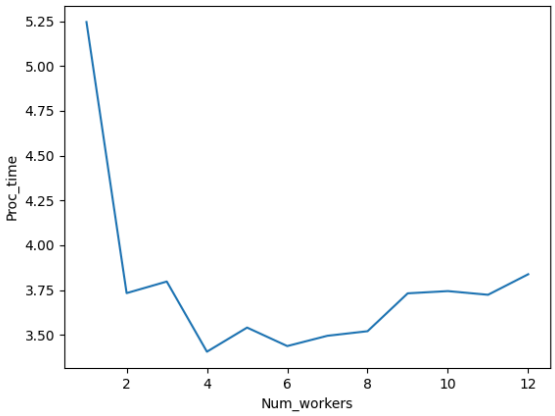


Figure 2

Model	Processing time[s]
Loop (naive)	439.70
Loop + Numba	2.20
Loop + MP	72.38
Vectorized	4.23

Vectorized + Numba	4.99
Vectorized + MP	3.41

Conclusions

The Just-In-Time compiler had the biggest impact by far when it came to performance improvements. That is to be expected as its purpose is double. For one, it compiles the main function, bypassing the slow version of the python interpreter. Additionally, though, it automatically parallelizes the operations.
The vectorized version was on its own very fast, however the jit compiler did not have as much of an impact.

Parallelization had the biggest impact in the naive implementation, reducing the processing time from 439.7s to 72.38s. From Figure 1, we can see that the speedup is much bigger for the first 6 workers compared to the remaining. That is because the program was run on a 6-core cpu with multi-threading. The actual physical cores are 6, but with the multi-threading functionality up to 12 different workers can run in parallel while sharing some physical resources. It is expected that increasing the number of workers further will slowly start introducing more and more overhead, increasing the processing time. Some impact from parallelization can also be observed with the vectorized implementation.

In general, it becomes apparent that vectorized operations are a very easy way of achieving speedup in code that supports it. Packages like numba are very effective but are also very restrictive on the tools one can use, and custom parallelization processes are mostly meaningful when the operation has been optimized in all other regards and the processing time is still much larger than the introduced overhead.