# Numerical Scientific Computing
# Mini Project
# Mandelbrot set

Holger Bovbjerg, Peter Fisker - SPA Group 870

03/06/2021

## 1 The Mandelbrot set

The Mandelbrot set is the set of numbers for which the function $f_c(z) = z^2 + c$ does not diverge when evaluated recursively starting with $z = 0$, i.e. $f_c(0), f_c(f_c(0))$ etc. has bounded absolute value. The Mandelbrot set can be approximated by testing a number of point in the complex plane for divergence. If the number does not diverge it is part of the Mandelbrot set and if it "explodes" it is not a part of the Mandelbrot set. To do this a threshold value and max iteration number is set. In this mini project a threshold value of 2 and a max iteration number of 100 is used. This means that if a point goes beyond 2 in less than 100 iterations it is not a part of the Mandelbrot set, and if the point stays within the threshold of 2, then it is included in the Mandelbrot set.

Computation of the Mandelbrot set belongs to a category of computational problems which are described as conveniently parallel [1]. This is due to the fact that the evaluation of each point in the complex plane is tested using the same function. In addition, the evaluation of each point is independent of other points.

In this mini project, a number of Python algorithms for computation of the Mandelbrot set is presented. The set of points that is evaluated is limited to the complex matrix given by (1) and the number of points is set by the resolution $p_{\text{re}}$ and $p_{\text{im}}$.

$$\mathbb{C} = \begin{bmatrix} -2 & \dots & 1 \\ \vdots & \ddots & \vdots \\ -2 & \dots & 1 \end{bmatrix} + j \begin{bmatrix} 1.5 & \dots & 1.5 \\ \vdots & \ddots & \vdots \\ -1.5 & \dots & -1.5 \end{bmatrix} \tag{1}$$

The function `create_mesh(x,y)` seen in Listing 1 generates a mesh inside the specified region $\mathbb{C}$. In this project the is set as $p_{\text{re}} = 2^12$ and $p_{\text{im}} = 2^12$.

```python
def create_mesh(real_points: int, imag_points: int):
    '''
    Function that generates a mesh of complex points from the complex
    plane, in the region: -2 < Re < 1  and -1.5 < Im < 1.5
    The resolution of the mesh is determined by the input values.
    :param real_points: Number of points on the real axis
    :param imag_points: Number of points on the imaginary axis
    :return: 2D ndarray of complex values.
    '''
    Re = np.array([np.linspace(-2, 1, real_points), ] * real_points)
    Im = np.array([np.linspace(-1.5, 1.5, imag_points), ] * imag_points).transpose()
    return Re + Im * 1j
```

Listing 1: Create C-mesh function.

### 1.1 Mandelbrot image

Points that only needs a few iterations to exceed the threshold can be seen as very reactive points, whereas points that have many iterations before they exceed the threshold are less reactive. Using this information, and associating the complex points with 2D image pixels using the iteration number as a colour intensity value, a graphical illustration of the Mandelbrot set can be made. A naive algorithm for computing Mandelbrot intensity values is seen in Algorithm 1.

**Algorithm 1:** Naive Mandelbrot algorithm.

```
Input: $\mathbb{C}$ (input mesh), $T$ (threshold), $I$ (max iterations)
Output: Mandelbrot heatmap
for $i = 1$ to $p_{\text{re}}$ do
    for $j = 1$ to $p_{\text{i}}$ do
        $z := 0$;
        $n := 0$ while $|z| \leq T$ and $n < I$ do
            $z := z^2 + \mathbb{C}[i, j]$
        end
        heatmap$[i, j] = n/I$
    end
end
```

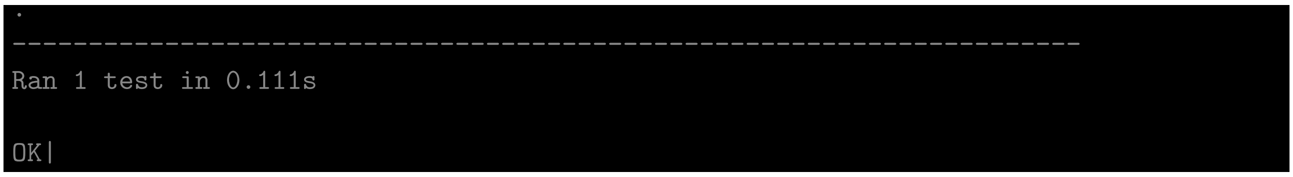
## 2 Test strategy

In order to test the correctness of the different implementations, some test strategy is needed. In this mini-project the naive implementation is used as a baseline. The other implementations are then tested against this function by checking whether they produce results that are equal (or close to equal). This is done using the `numpy` function `allclose(a,b)`. The Python package `unittest` is used to automate these tests. An example from a test of the vectorized implementation is seen in Listing 2. Only a small subset of the complex plane is used, as the amount of points chosen should only impact the computation time and not the accuracy of the results.

```python
class TestMandelbrotMethods(unittest.TestCase):
    def test_vector(self):
        c = mf.create_mesh(50, 50)
        T = 2
        I = 100
        self.assertTrue(
            np.allclose(
                mf.mandelbrot_naive(c, T, I),
                mf.mandelbrot_vector([c, T, I])
            )
        )
if __name__ == '__main__':
    unittest.main()
```

Listing 2: Python unittest example.

Running the test succesfully will yield the following output



```
.
----------------------------------------------------------------------
Ran 1 test in 0.111s

OK|
```

## 3 Software design

The developed software is divided into a number of files which can be found on `https://github.com/HolgerBovbjerg/Numerical-Scientific-Computing-mandelbrot-mini-project`. The code is divided into a main runfile containing the main program, a function file containing all functions used in the project and a test file which contains the test of each Mandelbrot function. Some additional files related to the specific implementations are also found such as a GPU kernel and a Cython file. All files are named according to their purpose or functionality. Timing of the various implementations is done using `time.time()` from Python's `time` library [2]. To make the execution time estimations more robust a number of iterations is run from which the mean execution time can be calculated. The results are saved using the Python package `h5py` [3]. The Mandelbrot figures are plotted and saved using `matplotlib.pyplot` [4].

## 4 Naive Mandelbrot implementation

The naive implementation of the Mandelbrot function is based on Algorithm 1. Therefore, the Python implementation is essentially just a mapping of algorithmic description to Python syntax. This implementation is not very efficient as each point is processed sequentially.

```python
def mandelbrot_naive(c: np.ndarray, T: int, I: int):
    """
    Function that calculates all M(c) values in the c-mesh given.
    Implemented the naive python way with nested for-loops that calculates
    each M(c) sequentially using the mandelbrot definition.

    :param c: c-mesh containing segment of the complex plane
    :param T: Threshold value used to determine if point is in Mandelbrot set
    :param I: Maximum number of iterations used to determine if point is in Mandelbrot set.
    :return: np.ndarray with M(c) values for each point in C
    """
    n = np.zeros_like(c,dtype=int)
    dim = c.shape
    for i in range(dim[0]):
        for j in range(dim[1]):
            z = 0
            while abs(z) <= T and n[i,j] < I:
                    z = z*z + c
                    n[i,j] += 1
    return n/I
```

Listing 3: Naive Mandelbrot implementation.

The resulting Mandelbrot image can be seen in Figure 1. As is seen the naive implementation takes 412 s which roughly equates to seven minutes. Hopefully, it is possible to find some speedup by utilising methods for optimising numerical computations.

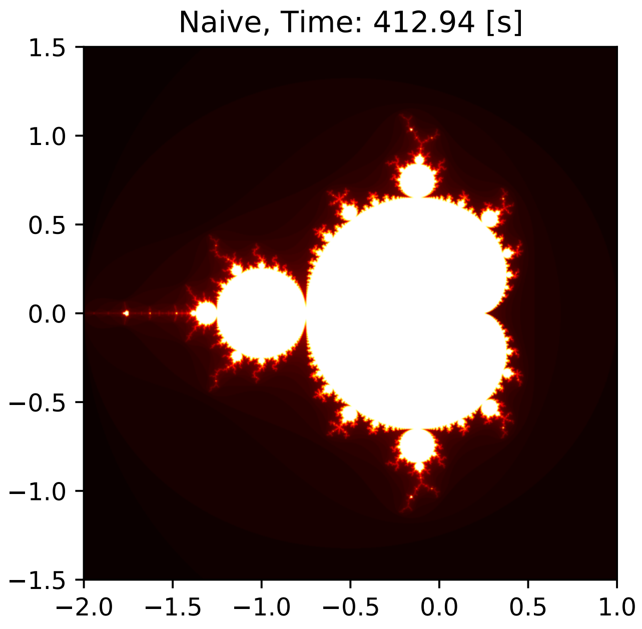


Figure 1: Naive Mandelbrot heatmap

## 5 Vectorized Mandelbrot implementation

In the Mandelbrot algorithm given by Algorithm 1 the processing of each input point is independent of the processing of the other points. Therefore, the code can be made more efficient by using vectorized code. This can be done using packages such as `numpy` [5]. The `numpy` library makes use of low-level linear algebra algorithms such as those specified in BLAS [6]. In Listing 4 a vectorized version of the Mandelbrot function is found. Here the for loops of Listing 3 are replaced by array operations using `numpy` commands.

```python
import numpy as np

def mandelbrot_vector(data: list):
    """
    Function that calculates the M(c) values in the c-mesh given,
    implemented in a vectorised way using numpy.
    Here each point in the mesh is updated "at once" at each iteration.
    In order to use this function for the multiprocessing and distributed functions
    the input has been packed into a list.

    :param data: Data is a list containing:
        :param c: c-mesh containing segment of the complex plane
        :param T: Threshold value used to determine if point is in Mandelbrot set
        :param I: Maximum number of iterations used to determine if point is in Mandelbrot set.
    :return: np.ndarray with M(c) values for each point in c.
    """
    z = np.zeros_like(c)
    n = np.zeros_like(c,dtype=int)
    ind = np.full_like(c,True,dtype=bool)
    while np.any(np.abs(z) <= T)  and np.all(n < I ):
        z[ind] = np.add(np.multiply(z[ind],z[ind]),c[ind])
        ind[np.abs(z) > T] = False
        n[ind] += 1
    return n/I
```

Listing 4: Numpy vectorized implementation.

In Figure 2 the Mandelbrot image from the vectorized implementation is seen. With an execution time of 52.9 s, it is a significant speedup from the naive implementation. This is a result of two things. First, `numpy` makes use of available vector instructions in the processor instruction set. Secondly, the use of memory-optimised linear algebra algorithms reduces data transfer.

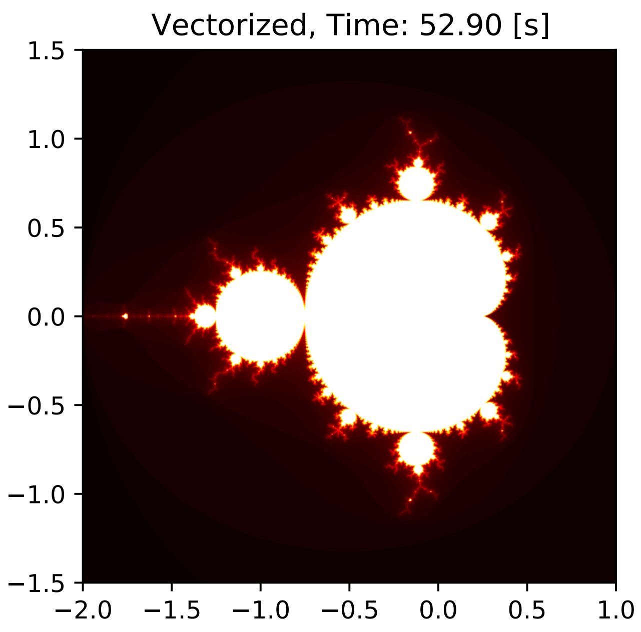


Figure 2: Vectorized Mandelbrot heatmap.

## 6 Numba accelerated implementation

As Python is an interpreted language, the code is not compiled into machine instructions before being run. Numba is a Python package which translates Python code to optimised machine code during runtime [7]. It uses the LLVM compiler library and can produce code for Python implementations of numerical algorithms which are near the same speed as C implementations. The use of numba is rather straight forward as the process mainly consist of adding a decorator `@jit(nopython=True)` to the function definition as well as ensuring that the numba package supports the operations used in the function. In general numba supports standard Python code as well as `numpy` code. In Listing 5 an example of the numba optimised naive implementation is seen. When running the numba optimised function the first time, the function will not be executed as fast as expected. This is due to the fact the numba optimisation has to be done during the first function call. The following function calls will then use the numba optimised code.

```python
import numpy as np
import numba
from numba import jit

@jit(nopython=True)
def mandelbrot_numba(c: np.ndarray, T: int, I: int):
    """
    Function that calculates the M(c) values in the c-mesh given,
    implemented using the numba library on the naive implementation.
    Numba analyzes and optimizes the code before compiling it to a
    machine code version tailored to the CPU.
    For more info on Numba, visit https://numba.pydata.org/

    :param c: c-mesh containing segment of the complex plane
    :param T: Threshold value used to determine if point is in Mandelbrot set
    :param I: Maximum number of iterations used to determine if point is in Mandelbrot set.
    :return: np.ndarray with M(c) values for each point in c.
    """
    n = np.zeros_like(c, dtype=numba.int64)
    dim = c.shape
    for i in range(dim[0]):
        for j in range(dim[1]):
            z = 0
            while abs(z) <= T and n[i, j] < I:
                z = z * z + c[i, j]
                n[i, j] += 1
    return n / I
```

Listing 5: Numba implementation.

The result which is depicted in Figure 3 shows an execution time of only 2.53 s. Where the vectorized implementation yielded a speedup of 7.8 the numba implementation yields a speedup of 163. This rather significant difference between the two can be explained by the fact, the numba optimiser sees all the code, and can thus optimise on the whole code structure, whereas the numpy version has each numpy function call as an isolated call to an optimised numpy implementation.

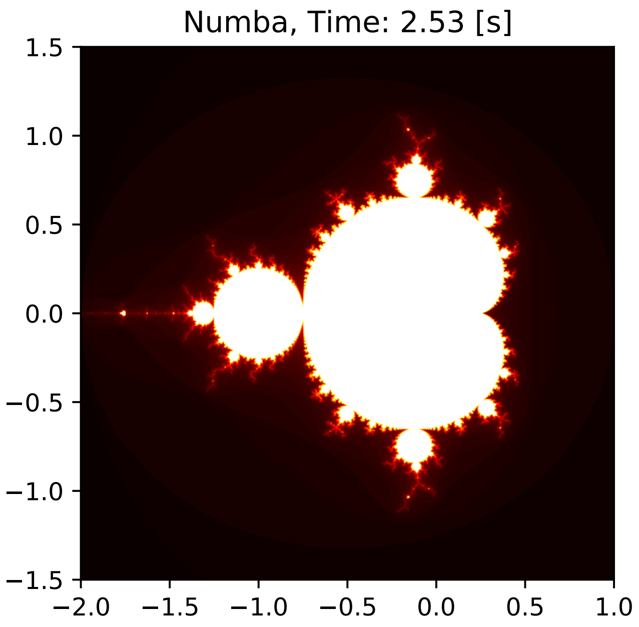


Figure 3: Numba Mandelbrot heatmap

## 7 Cython accelerated implementation

Many algorithm optimisations make use of machine-level optimisations, therefore, low-level programming languages such as C often achieve better performance than high-level languages like Python although with the caveat of more complicated code and slower development. Cython is an optimising static compiler for Python and the Cython language [8]. Cython works by compiling Python or Cython code into C code which can be run from a Python environment. This makes it possible to add C-extensions to Python programs in order to optimise certain parts of code. The speedup is mainly found from static type declarations of both C and Python variables, as the Cython compiler can then map parts of the code to C semantics, which can be translated into efficient C-code. A Cython implementation of the naive Mandelbrot function is seen in Listing 6. The code includes a number of static type assignments as well as two compiler directives `@cython.boundscheck(False)` and `@cython.wraparound(False)`. These compiler directives tell the Cython compiler that it does not need to worry about checking for iteration beyond the bound of an array, and that the code does not make use of Python-like wraparound (negative indexing).

```
import cython
import numpy as np
cimport numpy as np

ctypedef np.complex128_t cpl_t
cpl = np.complex128

@cython.boundscheck(False) # compiler directive
@cython.wraparound(False) # compiler directive
def mandelbrot_naive_cython(np.ndarray[cpl_t,ndim=2] c, int T, int I):
    dim = c.shape
    cdef int x,y
    x = dim[0]
    y = dim[1]
    cdef np.ndarray[int, ndim=2] n = np.zeros((x,y), dtype=int)
    for i in range(x):
        for j in range(y):
            z = 0 + 0j
            while abs(z) <= T and n[i, j] < I:
                z = z * z + c[i, j]
                n[i,j] += 1
    return n / I
```

Listing 6: Cython implementation.

The result of the Cython implementation is seen in Figure 4. The execution time of 49.7 s is a bit faster than the numpy implementation. Keeping in mind that the algorithm structure is simply the most naive implementation, this is a quite impressive speedup from simply declaring static data types. In Figure 5 the result of running the Cython compiler on the numpy version is seen. As might have been expected, there is no speedup in comparison to the numpy implementation. This is probably due to the fact that the Cython compiler most likely does not change the numpy function calls.

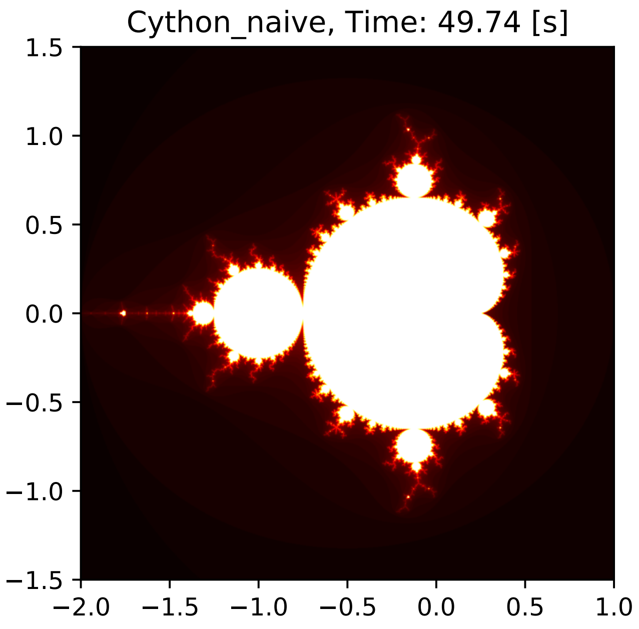


Figure 4: Cython Naive Mandelbrot heatmap

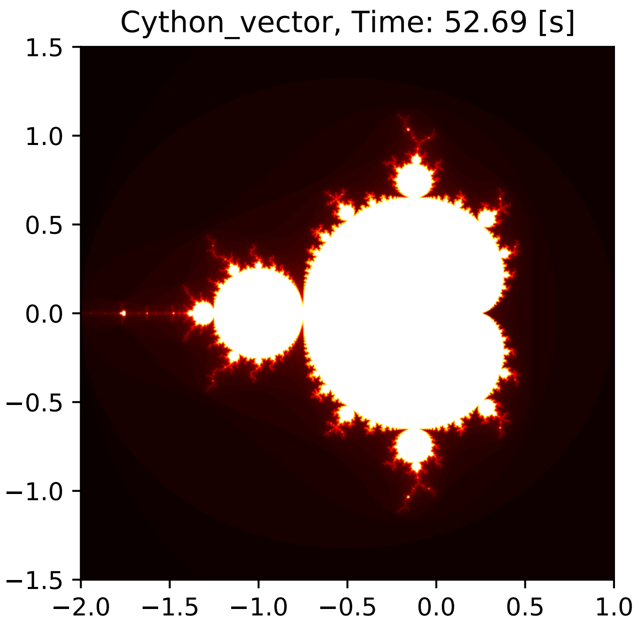


Figure 5: Cython Vector Mandelbrot heatmap

## 8 Multi-core parallel implementation using multiprocessing

The former implementations have all been focused on single-processor execution, however, most modern processor platforms have a number of cores which can be all work in parallel. As the Mandelbrot problem belongs to the category of conveniently parallel problems, parallel execution should be able to reduce the execution time significantly. Both numpy and numba support parallel execution of code, however, for the sake of studying parallel speedup, more control can be gained by using Python `multiprocessing` library. In Python the `multiprocessing` library makes it possible to create a parallel task pool and assign multiple tasks to the pool which are then executed in parallel [9].

Ideally the parallel execution time is equivalent to dividing the scalar execution time by the number of processors utilised [1]. However, as the data has to be distributed and the results gathered some communication overhead is added to the computation time. In addition, the tasks may not finish at the same time, or they may not distribute evenly on the number of processors which can lead to some processors being idle, also known as load unbalance.

In Listing 7 an implementation of the vectorized Mandelbrot function utilising the functionality of the multiprocessing package is found. First a processing pool is created with a number of associated processors. The input data is then divided into a number of blocks which can be divided out on the processors. Next the `map_async(fun,iterable)` function is used to map each `mandelbrot_vector(C,T,I)` function call to the task, where the number of function calls is equivalent to the number of blocks to pool. The `pool.close()` command closes the pool when all workers are finished and the `pool.join()` waits for all workers to finish. The result can then be retrieved by calling `results.get()`. The execution time will vary depending on the block division of the $\mathbb{C}$-matrix and the number of processors utilised.

```python
def mandelbrot_parallel_vector(c: np.ndarray, T: int, I: int, processors: int,
                                blockno: int, blocksize: int):
    """
    Function that calculates the M(c) values in the c-mesh given,
    using multiprocessing to do calculate in parallel.
    The functions uses the python multiprocessing library and assigns work
    with the asynchronous map function.
    The c-mesh is divided into equal size blocks, and each block is sent
    to the vectorized mandelbrot function.
    A block consists of one or more whole rows of c-mesh.

    :param c: c-mesh containing segment of the complex plane
    :param T: Threshold value used to determine if point is in Mandelbrot set
    :param I: Maximum number of iterations used to determine if point is in Mandelbrot set.
    :param processors: Number of processors to divide the workload amongst.
    :param blockno: Number of blocks to divide the c-mesh into.
    :param blocksize: The amount of rows of c-mesh in a single block.
    :return: np.ndarray with M(c) values for each point in c-mesh.
    """
    pool = mp.Pool(processes=processors)
    data = [[c[blocksize * block:blocksize * block + blocksize], T, I] for block in range(blockno)]
    results = pool.map_async(mandelbrot_vector, data)

    pool.close()
    pool.join()
    out_matrix = np.vstack([row for row in results.get()])
    return out_matrix
```

Listing 7: Multiprocessing implementation

Figure 6 shows the Mandelbrot image and execution time from the multiprocessing implementation using 12 processors. As is seen in Figure 7 the speedup from adding processors is not even close to ideal however, a speedup of up to 2.75 is achieved. As is seen the speedup actually decreases when using more than 8 processors which can be explained by the communication cost becoming more significant than the parallel speedup.

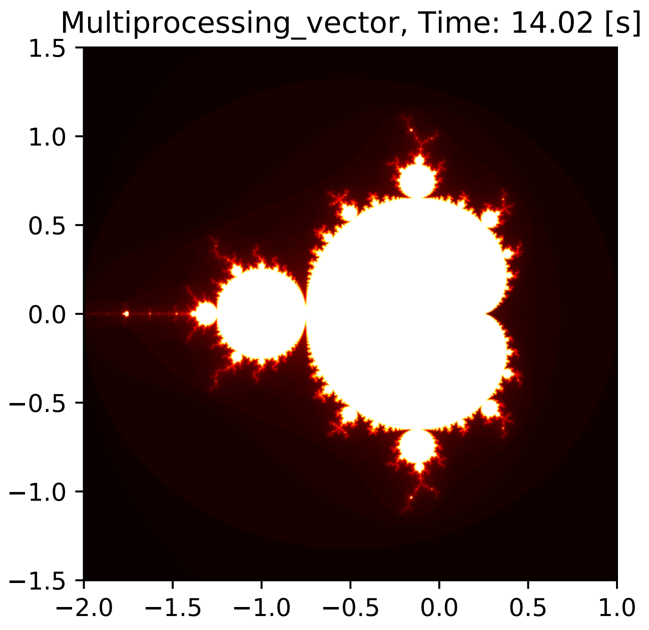


Figure 6: Multiprocessing Mandelbrot heatmap

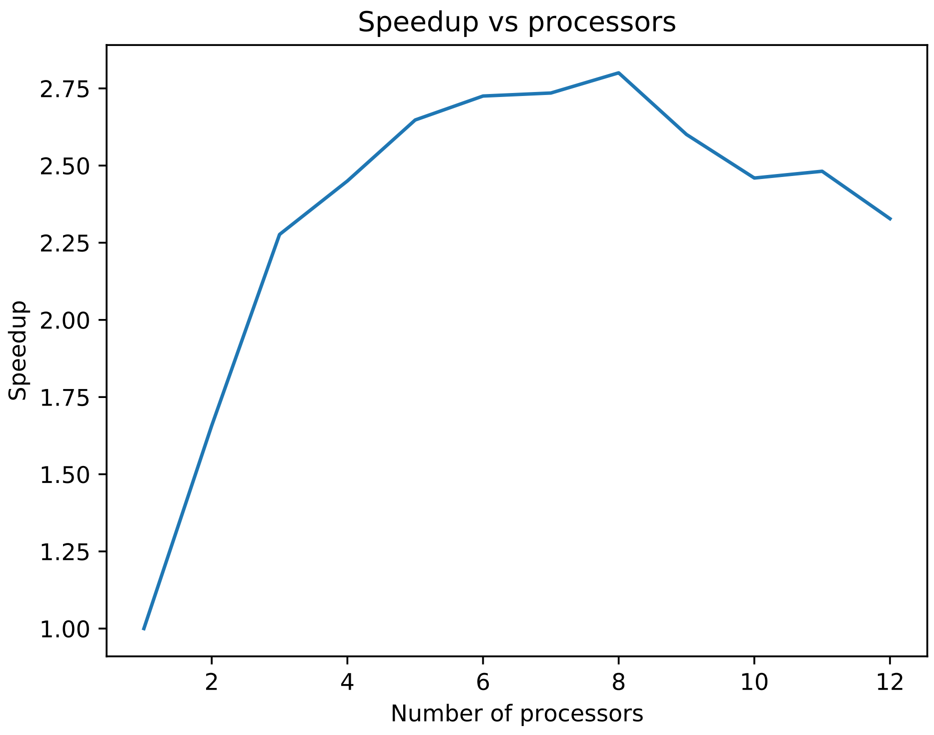


Figure 7: Multiprocessing speedup varying number of processors.

## 9 Distributed parallel implementation using DASK

While the utilisation of multiple cores can significantly increase performance, there is a limit on the number of available cores and memory on a single computer. To mitigate this limitation, the computations might be distributed to multiple machines. DASK is a python package which makes it easy to write scalable code for parallel execution [10]. DASK makes it possible to handle large datasets by automatically dividing data into blocks (called chunks in DASK). This is useful when the data is too large to fit into memory. In addition to this DASK provides a scalable framework for parallel computing which is easily scalable from a single laptop computer to a large multi-node cluster. It provides the `delayed()` decorator to specify function calls which can be scheduled "lazily", i.e. as late as possible. This makes it possible to generate a parallel execution schedule using independence relationships between tasks. The syntax of DASK is very alike that of the multiprocessing package. To initiate a parallel processing environment `client=Client(n_workers=processes)` is called. The data is then submitted to client using `client.scatter(data)`. The function is then applied to the data using `client.submit(fun,scattered_data)`. Lastly the results are retrieved using `client.gather(results)`.

```python
def mandelbrot_distributed_vector(c: np.ndarray, T: int, I: int, processes: int, blockno: int, blocksize: int):
    """
    Calculates the M(c) values in the c-mesh given, by dividing the mesh
    into blocks and then sending each block out to a node in distributed
    network.
    The vectorized implementation of the Mandelbrot calculation is used for
    the calculations on the nodes.
    (NOTE: This function distributes the blocks to nodes on the local machine
    similarly to the multiprocessing implementation. )

    :param c: c-mesh containing segment of the complex plane
    :param T: Threshold value used to determine if point is in Mandelbrot set
    :param I: Maximum number of iterations used to determine if point is in Mandelbrot set.
    :param processes: Number of workers to divide the workload amongst.
    :param blockno: Number of blocks to divide the c-mesh into.
    :param blocksize: The amount of rows of c-mesh in a single block.
    :return: np.ndarray with M(c) values for each point in c-mesh.
    """
    client = Client(n_workers=processes)
    results = []

    for block in range(blockno):
        # Send the data to the cluster as this is best practice for large data.
        data = [c[blocksize * block:blocksize * block + blocksize], T, I]
        big_future = client.scatter(data)
        results.append(
            client.submit(mandelbrot_vector, big_future)
        )

    client.gather(results)
    out_matrix = np.vstack([result.result() for result in results])
    wait(out_matrix)
    client.close()

    return out_matrix
```

Listing 8: DASK implementation.

Although DASK is meant for distributed computing, the DASK implementation has been tested on a single machine using a so-called “LocalCluster” which simulates a distributed cluster. As is seen in Figure 8, the execution time of the DASK implementation is slower than the multiprocessing implementation. This can be explained by the fact that the `submit` function has some overhead in addition to the communication overhead of the DASK scheduler. As DASK is meant for use on problems that cannot fit on a single machine, this is expected. If a multi node cluster was used and the problem was scaled up to a larger $\mathcal{C}$-grid a bigger speedup effect is expected, as the DASK overhead will then be marginalised.

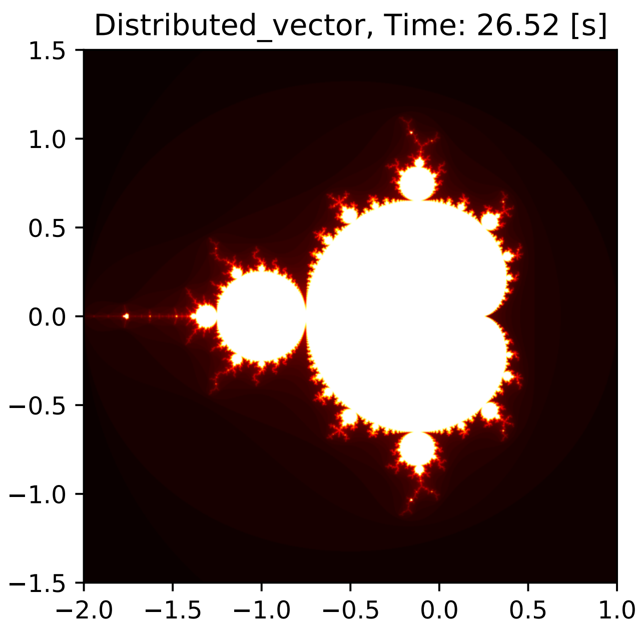


Figure 8: DASK Mandelbrot heatmap

## 10 GPU implementation using PyOpenCL

GPUs are processors which are highly specialised in SIMD type instruction, i.e. performing the same operations on different data. Although they originate for the need of specialised processors for graphics computations, they have become a popular tool for accelerating scientific numerical computing [1]. Multiple APIs exist for programming GPUs, one of them being OpenCL which is supported by all big graphics card vendors (NVIDIA, Intel, AMD).

PyOpenCL is a Python package which makes it possible to use OpenCL kernels for GPU computing in Python [11]. PyOpenCL works by first creating a context for interacting with the GPU. A command queue is then formed to which computational tasks can be assigned. A number of buffers and some memory flags also has to be defined in order to communicate data back and forth with the GPU. When this is done the actual program which is to be run on the GPU is build. This program is called a "kernel" and is written in the C-like language OpenCL. The OpenCL code is written in such a way that the GPU can take advantage of its many processing elements. When writing the kernel it is also important to be aware of the memory structure of the GPU as its memory is divided into private (work item), local(work group), global(GPU) and constant memory(GPU). The kernel written for computing the Mandelbrot function is seen in Listing 9. The python function using this kernel is seen in Listing 10.

```c
#define PYOPENCL_DEFINE_CDOUBLE
#include <pyopencl-complex.h>

__kernel void mandelbrot(
    __global const cdouble_t *c_gpu,
    __global double *result_g,
    const int MAX_ITER,
    const int THRESHOLD
    )
{
    int gidx = get_global_id(0);
    int gidy = get_global_id(1);
    int width = get_global_size(0);

    cdouble_t z = cdouble_new(0, 0);
    cdouble_t c = c_gpu[gidx * width + gidy];
    double n = 0;

    while (cdouble_abs(z) <= THRESHOLD && n < MAX_ITER){
        z = cdouble_add(cdouble_mul(z, z), c);
        n = n + 1 ;
    }
    result_g[gidx * width + gidy] = n/MAX_ITER;
}
```

Listing 9: OpenCL kernel for PyOpenCl implementation

In the kernel, care has been taken to operate on private memory as often as possible, because transferring data to and from the global memory takes more time. So the input c and the variables z and n are saved to private memory, as they are accessed multiple times in the while loop. After the while loop, the result are saved back to global memory.

```python
import pyopencl as cl

def mandelbrot_GPU(c: np.ndarray, T: int, I: int):
    result_matrix = np.empty(c.shape).astype(np.float64)
    # Set up the GPU stuff
    # Create a context and choose device interactively
    # ctx = cl.create_some_context(interactive=True)

    # If interactive mode does not work, use this snippet to choose manually instead.
    platform = cl.get_platforms()[0] # select platform, e.g. Intel or NVIDIA
    my_device = platform.get_devices()[0] # select device
    ctx = cl.Context([my_device]) # create context

    queue = cl.CommandQueue(ctx) # create command queue

    mf = cl.mem_flags
    c_gpu = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=c)
    result_g = cl.Buffer(ctx, mf.WRITE_ONLY, result_matrix.nbytes)

    kernelsource = open("mandelbrot_kernel.cl").read()
    prg = cl.Program(ctx, kernelsource).build()

    # Execute the "mandelbrot" kernel in the program.
    mandelbrot = prg.mandelbrot
    mandelbrot.set_scalar_arg_dtypes([None, None, np.int32, np.int32])
    mandelbrot(
        queue,  # Command queue
        c.shape,  # Global grid size
        None,  # Work group size
        c_gpu,  # param 0
        result_g,  # param 1
        I,  # param 2
        T,  # param 3
    )
    cl.enqueue_copy(queue, result_matrix, result_g)

    return result_matrix
```

Listing 10: PyOpenCL implementation

Figure 9 shows the Mandelbrot image and execution time of the GPU implementation is seen. The execution time of 0.96 s shows that the GPU implementation yields an impressive speedup, as it is approximately 412 times faster than the naive implementation. The main factor is the fact that the mandelbrot algorithm can be posed as a Single Program Multiple Data (SPMD) program. This means that the program consists of a number of function calls to the same function on multiple data. These type of programs are highly suitable for GPU computation.

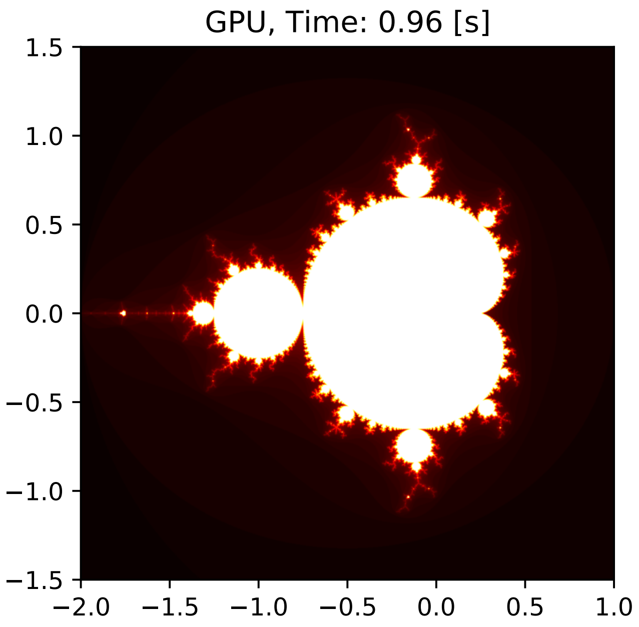


Figure 9: GPU Mandelbrot heatmap

## 11 Execution time of implementations

A summary of execution times for all implementations is found in Figure 10. Here two implementations stand out, namely the GPU and numba implementation. The GPU implementation takes advantage of the high number of GPU threads to achieve a speedup of approximately 412, whereas the numba implementation takes advantage of compiling the Mandelbrot function using a number of low-level optimisations. An additional test has been done where the size of the input is varied.

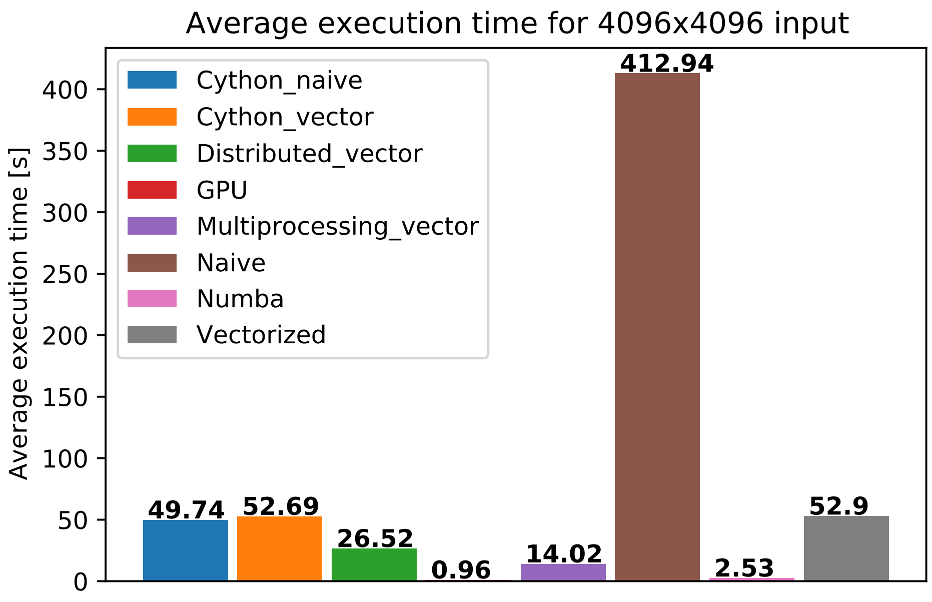


Figure 10: Execution time comparison for all implementations.

Figure 11 shows a comparison of the naive and vectorised implementation. Here it is seen that the execution time of the naive implementation grows rapidly with input size, whereas the execution time of the vectorised implementation grows less significantly.

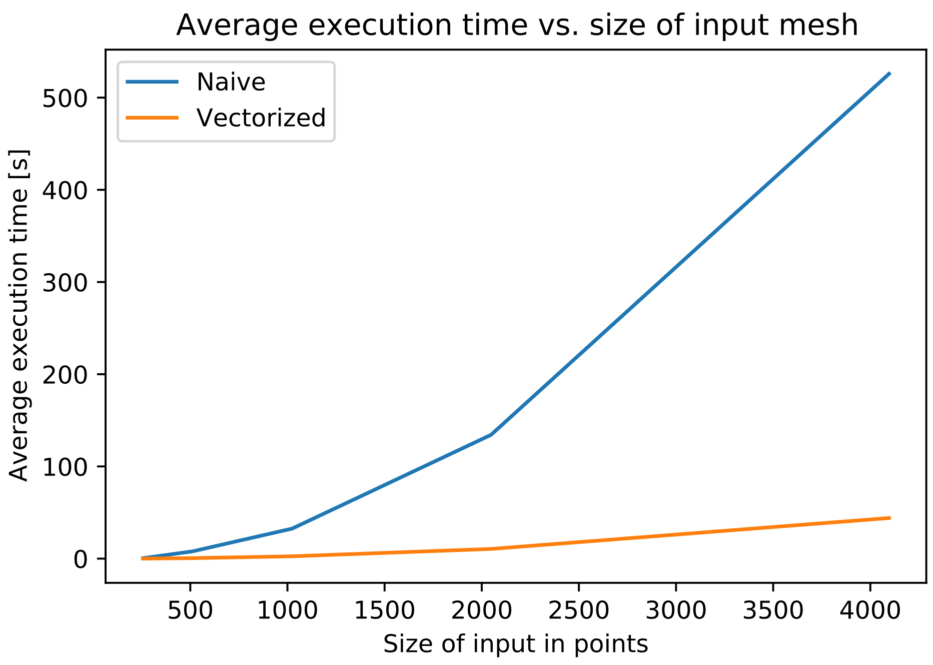


Figure 11: Execution time comparison varying input size naive vs. vectorized.

The results from the different optimisation of the naive implementation are illustrated in Figure 12. Here it is seen that the vectorised and Cython implementations take significantly longer time to execute as the input size is increased. The other implementations do not grow significantly in execution time and the GPU and numba version are close to constant in execution time.

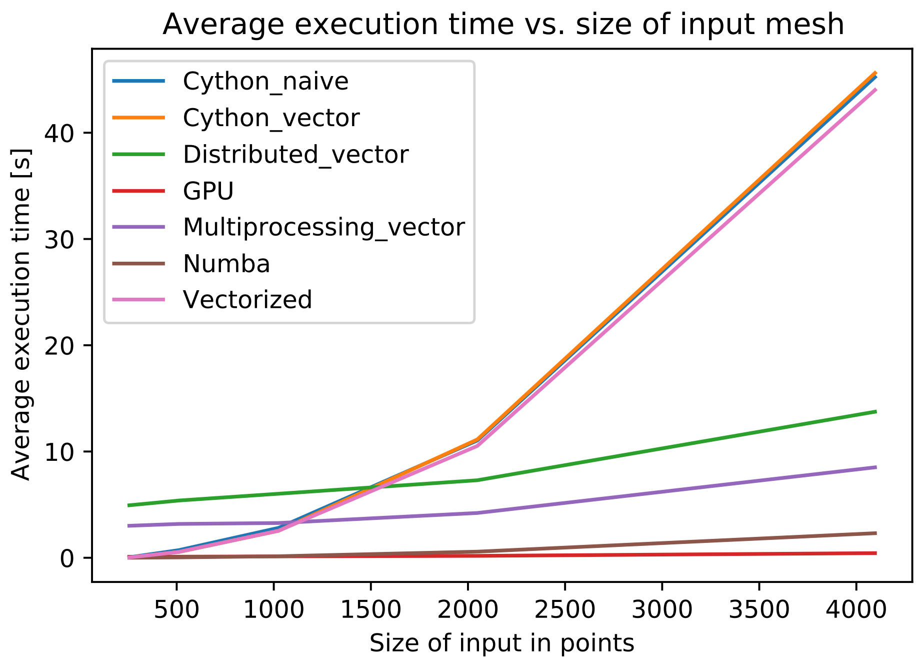


Figure 12: Execution time comparison varying input size.

## References

[1] V. Eijkhout, E. Chow, and R. van de Geijn, *Introduction To High Performance Computing*, 3rd ed. lulu.com, 2020, ISBN: 978-1-257-99254-6.

[2] Python. (2021). "Time access and conversions," [Online]. Available: `https://docs.python.org/3/library/time.html` (visited on 05/31/2021).

[3] h5py. (2021). "H5py quick start guide," [Online]. Available: `https://docs.h5py.org/en/stable/quick.html` (visited on 05/31/2021).

[4] matplotlib. (2021). "Matplotlib: Visualization with python," [Online]. Available: `https://matplotlib.org/stable/index.html` (visited on 05/31/2021).

[5] NumPy. (2021). "Numpy documentation," [Online]. Available: `https://numpy.org/doc/stable/` (visited on 05/31/2021).

[6] netlib. (2021). "Blas (basic linear algebra subprograms)," [Online]. Available: `https://www.netlib.org/blas/` (visited on 05/31/2021).

[7] Numba. (2021). "Numba makes python code fast," [Online]. Available: `https://numba.pydata.org/` (visited on 05/31/2021).

[8] Cython. (2021). "Cython documentation," [Online]. Available: `https://cython.readthedocs.io/en/latest/` (visited on 05/31/2021).

[9] Python. (2021). "Process-based parallelism," [Online]. Available: `https://docs.python.org/3/library/multiprocessing.html` (visited on 05/31/2021).

[10] DASK. (2021). "Dask documentation," [Online]. Available: `https://docs.dask.org/en/latest/` (visited on 05/31/2021).

[11] PyOpenCL. (2021). "Pyopencl documentation," [Online]. Available: `https://documen.tician.de/pyopencl/` (visited on 05/31/2021).