NUMERICAL SCIENTIFIC COMPUTING: Mini project 2

Lars Depuydt - CE8-NDS

The assignment

The miniproject is about computing the Mandelbrot set. This can be done in several ways, one faster than the other. Computing this requires a loop to run over each value of a complex matrix, which makes the problem O(x^3).
For each point in the matrix, we must check whether it is stable using the Mandelbrot quadratic complex mapping. This is done for a number of iterations I and with a threshold T. The set is calculated within a predefined area and density.
This time we try to optimize the runtime by optimizing the datatype used and using Dask distributed computing. Furthermore, some code improvements are made to break out of the loop early and not waste computing time.

Runtime

Multiple implementations of the problem were made and their runtime is compared against each other in the following table. The code tries to optimize the naive implementation of the algorithm. This by trying some optimizations and looking at the execution time. All the implementations were tested with the following parameters:

x_min, x_max, y_min, y_max = -2.0, 1.0, -1.5, 1.5
p_im, p_re = 5000, 5000
I = 30
T = 2

As default the matrix C is ‘complex128’ and the result matrix ‘float64’

	Run 1 (s)	**Run 2 (s)**	**Run 3 (s)**	**Average (s)**
Naive (loops), Input = **complex128**, result matrix = **float64**	34,12	34,12	40,25	**36,16**
Naive (loops), Input = **complex128**, result matrix = **float32**	37,26	36,56	34,49	**36,10**
Naive (loops), Input = **complex128**, result matrix = **float16**	32,15	32,14	32,68	**32,32**
Naive (loops), Input = **complex64**, result matrix = **float64**	79,90	84,02	87,14	**83,69**
Naive (loops), Input = **complex64**, result matrix = **float32**	78,45	77,20	76,75	**77,47**
Naive (loops), Input = **complex64**,	69,80	72,03	73,35	**71,73**

result matrix = **float16**				

For some reason, the ‘complex128’ type performs better than the ‘complex64’ type. Furthermore, we see that, as we would expect, preparing the result matrix of type float16 yields the best result.

Numpy vs Dask

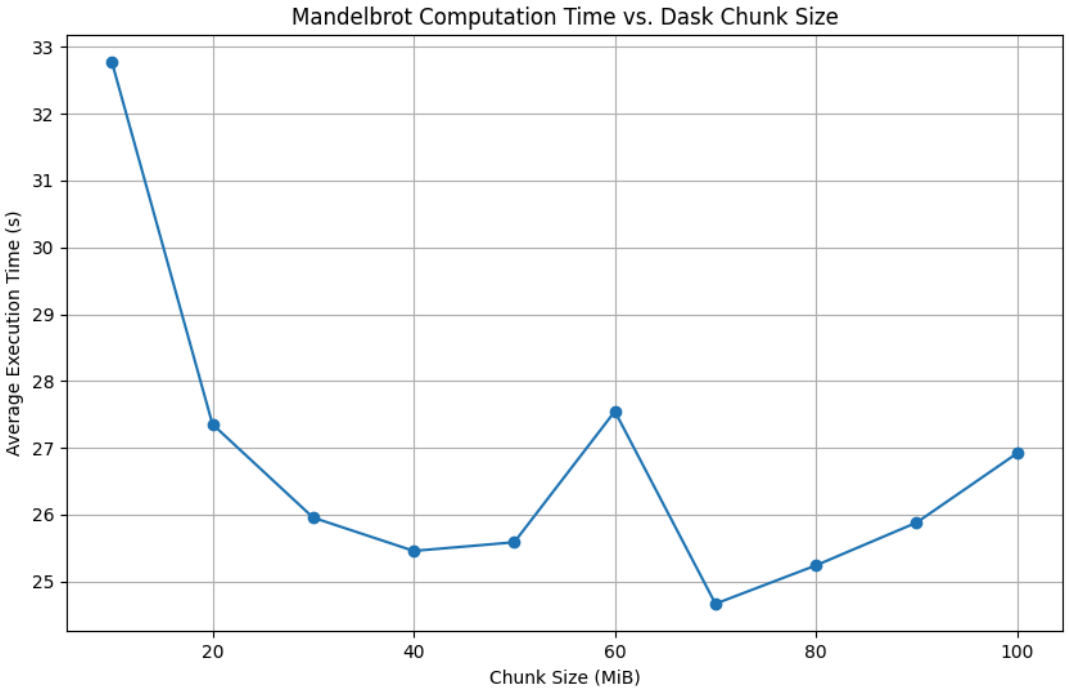
We compare our Numpy implementation against the Dask one. From this point on in the report, we use the following parameter change: p_im, p_re = 8000, 8000.

	Run 1 (s)	**Run 2 (s)**	**Run 3 (s)**	**Average (s)**
Vectorized Numpy (single core)	25,97	28,89	38,01	**30,96**
Vectorized Numpy multiprocessing	8,84	9,92	13,01	**10,59**
Dask local multiprocessing	5,43	5,89	5,68	**5,66**

We can see that even on a local computer, Dask outperforms our manual multiprocessing with Numpy.

Dask local multiprocessing

The Dask implementation of the program was run for a number of different chunk sizes. The graph shows the result, where the mean of 3 runtimes was taken. We can see that the program performs best for a chunk size of 70MiB, with an average runtime of just below 25 seconds.



Dask cluster multiprocessing

Next, the implementation was run on a cluster with one scheduler and 3 worker nodes. We varied the chunk size again. It can be seen that a value of ‘70MiB’ is again a good value, but also ‘160MiB’ gives a good result.