Numerical Scientific Computing

Jakob Lund Jensen
jjens19@student.aau.dk

November 29, 2024

Mini Project Description

1 Title

Mini-project step 2: naive, numpy, numba and multi-core

2 Moodle description

For this hand-in, you are expected to deliver your new work on the Mandelbrot mini-project, including the following:

Code as runnable Python files:

1. one or more updated versions of your previous code (naive, numpy or multiprocessing) where you investigate the achievable gain from using other data types, e.g., numpy.float32 instead of numpy.float64, etc. Please include execution time results for the original and optimized code.
2. Dask-version of the mandelbrot algorithm with execution time results for:
   (a) multi-core execution on a single computer. Compare the results to your normal numpy vectorized implementation.
   (b) (optional) cluster execution, either on strato cluster or another type of job-server/workstation.

Worksheet (either as supplementary PDF or as executable Jupyter Notebook) that includes a comparison of performance (in terms of execution time) for each of the algorithms.

3 Testing different data types

Table 1 shows the different data types that I experimented with, where regular is without using any specified data types.

Interestingly enough, I was able to use data types that are not complex, however, doing so produces the following warning: *ComplexWarning: Casting complex values to real discards the imaginary part int8 = np.int8(complex_ matrix(-2, 1, -1.5, 1.5, pxd))*, respectively for each non-complex data type that I used. Albeit this warning, I am still able to get results, and some of the tests even happen to be quicker than using the complex data types, although it has to automatically cast the values to become a complex value.

Regular	46.0
bool8	22.7
int8	18.3
intp	58.9
uint8	14.3
float16	50.6
c64	35.0
c128	46.0

Table 1: Table of data types with results rounded to one decimal

To make sure that the computations actually produced the correct results and didn't just give me nonsense, I made comparisons of the final matrix of all the different data types along with the regular version as a control. Furthermore, I printed the plots of all the results to make a visual comparison. In both cases all the results are identical.

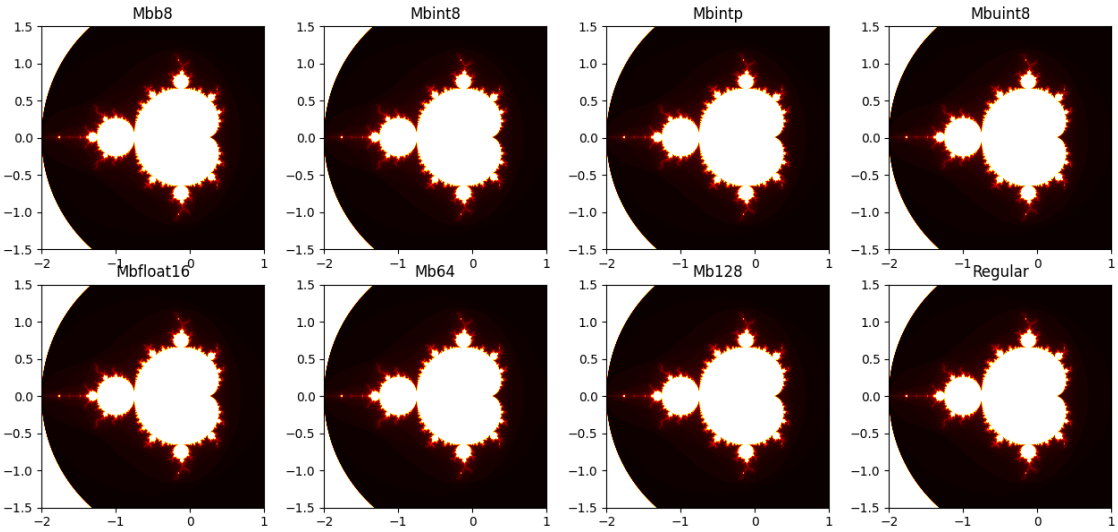

Figure 1: Plots of the Mandelbrot set using different data types

4 Dask implementation

Whenever I tried to run the code with chunk sizes smaller than 565x656, the code would produce a warning saying the chunk size is too small and would then automatically upscale the chunk size.

Chunk size	Execution time	Chunk size	Execution time
500	13,1	2800	24,8
600	14,5	2900	25,4
700	14,3	3000	25,8
800	17,0	3100	26,1
900	17,1	3200	26,7
1000	18,2	3300	27,4
1100	18,5	3400	29,6
1200	18,7	3500	29,3
1300	18,7	3600	35,1
1400	19,0	3700	34,3
1500	19,7	3800	31,0
1600	20,3	3900	31,9
1700	20,5	4000	33,2
1800	19,9	4100	33,7
1900	19,2	4200	34,4
2000	19,0	4300	35,2
2100	19,7	4400	36,2
2200	20,3	4500	37,0
2300	20,9	4600	37,8
2400	21,8	4700	38,6
2500	22,3	4800	39,8
2600	22,8	4900	40,9
2700	23,4	5000	41,1

Table 2: Table of tested chunk sizes and execution speed

From the table it is easy to see that smaller chunk sizes produces faster execution speed. **My normal Numpy solution finishes execution in 34.5s, compared to the fastest dask execution of 13.1s.**

By using the dask.visualize function I produced the following visualization of the parallelization performed by dask. Figure 2 is for chink sizes 1250x1250

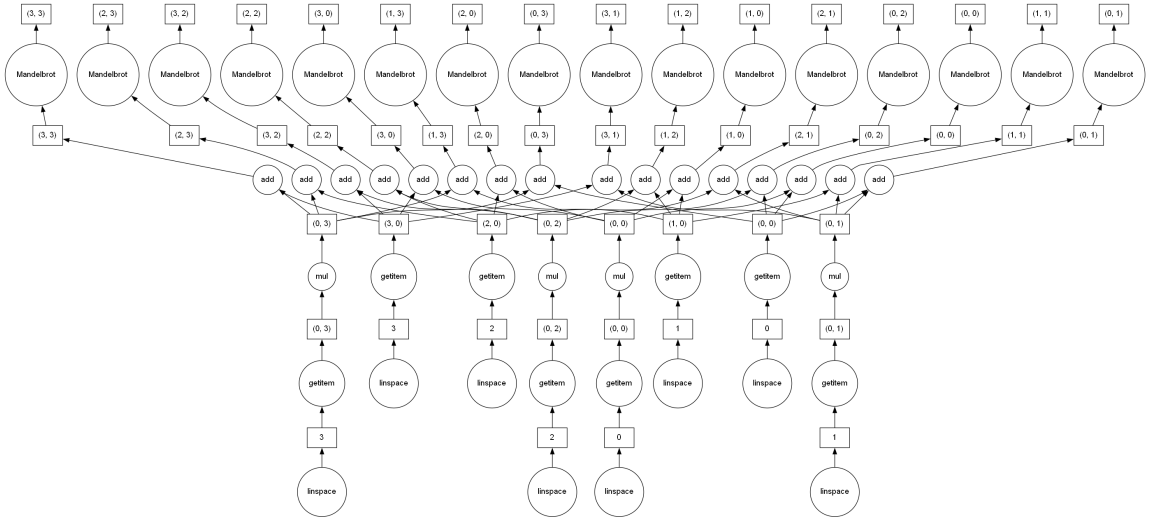

Figure 2: Dask parallelization