Parallel Programming

Exercise 2, Data Parallelism – Parallel Mandelbrot

Team Members

Aljoscha Alquati - se21m019
Michael Goll - se21003

Approach

The programming language chosen was C/C++. The production of the Mandelbrot image was implemented using two for-loops looping over the x- and y-axes of the image. Parallelism was implemented on the loop processing the y-axis. A test involving the parallelization of both loops with `# pragma omp for collapse(2)` did not yield any improvements and was therefore discarded. The chosen scheduler was “Static” (default) since the amount of work is pre-determined and predictable for all threads involved.

The realization emerged that - due to the simple application of the OpenMP capabilities by prepending one line of code to a loop – the parallelization of the method of calculating the Mandelbrot image involved less effort than producing and displaying the image itself. Also, having to re-learn C/C++ syntax was more difficult than implementing the actual parallelization.

Special thought must be given to the variables (resources) shared and made private when implementing parallelism with OpenMP. Missing variables led to disappointing performance results.

Performance

The benchmark was done on an AMD Ryzen 7 4800H CPU with 8/16 Cores and a base clock of 2900 Mhz and a max boost of 4300 Mhz.

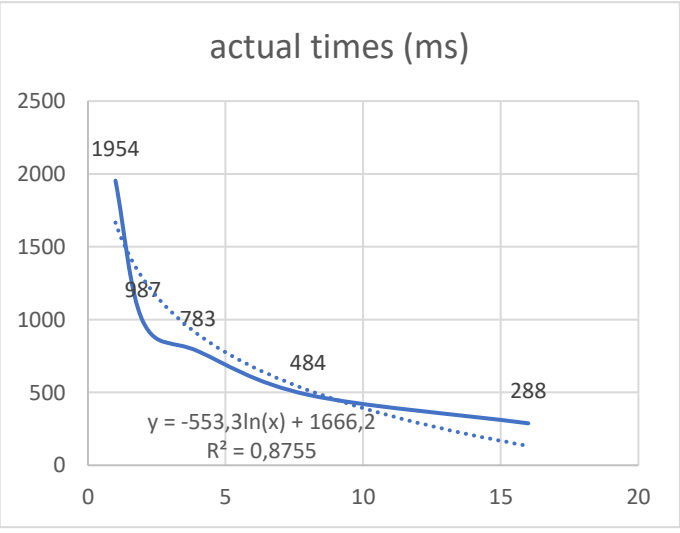
The following table sums up the speed increases (average from 100 repetitions):

number of threads	**1**	**2**	**4**	**8**	**16**
actual times (ms)	1954	987	783	484	288
speedup quotient	1,000	1,980	2,496	4,037	6,785

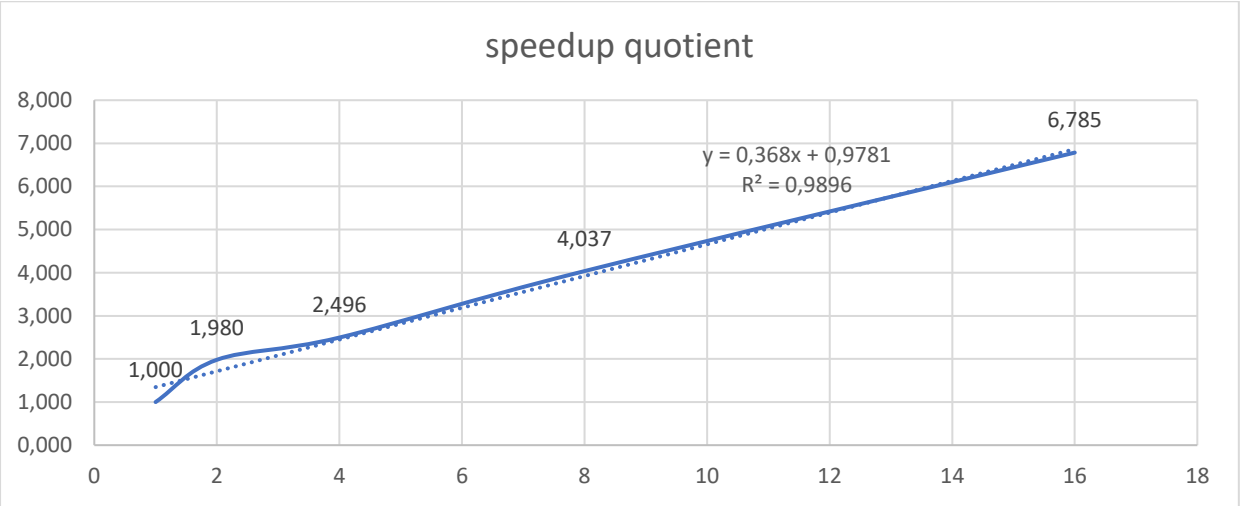
The “actual times (ms)” where measured excluding the time necessary to write the image to disc but including the time necessary to store each pixel to memory.

These results show a logarithmical decline in execution times (dashed trendline is logarithmic).

When visualizing the speedup quotient, the increase in speed is close to linear at least up to 16 threads. The exception is the first increase in



threads (to 2). This increases the speedup by a factor of nearly 2 (1,98).



The fact that doubling the number of threads does not double the speedup quotient each time might be explained by the added overhead incurred when having to manage multiple threads.