Assignment 4

and

1 PORTING THE CODE TO C

When porting the code, we noticed that some of the loops had bad spacial locality. When porting the code, we made sure to change this without changing the functionality. We also changed how the offset is done in the loop in `apply_stencil`, in order to allow us to collapse and parallelize the nested loop as well. All of this improved the wall time tremendously.

2 PARALLELIZATION

To parallelize the code, we simply made sure to parallelize all loops and change to code to allow for as much of the loops to be parallelized as possible. We made use of the library `OpenMP`. When the parallel code is running, there should be no processes fighting over access to data, because of the checkerboard pattern update method. However, when calculating the update for a tile, retrieving the data from the adjacent tiles isn't simple for all four. Since our array is row-major, the left and right tiles and right next to the tile we want to update. This means they are close in memory and have a very high chance of being in the same block or be retrieved in the same burst when moving it to the cache. Thus accessing these should be fast. However, the upper and lower tiles and both in different rows, and as such will be 2·width spaces from each other in the array. This means, not only are they far from each other in memory, but also will be in different blocks as well as have a very low chance of being retrieved in the same burst. This gives a high chance of conflict misses when the board is too large to be fit in the cache, as we will the have to have several bursts, which will override data we will want to retrieve again later.

3 SPEEDUP, AMDAHL'S LAW, AND THROUGHPUT

All of these calculations are based on the example, where we run the simulations on a 1000×1000 grid for 196 iterations.

Speedup. When comparing our C program with itself, running single-threaded with a speed of 4.500 second (usind the Unix `time` function) we see a good amount of speed-up. We calculate the speed-up for `OMP_NUM_THREADS=2`:

$$S_{N=2} = \frac{4.500}{2.509} = 1.7935$$

and `OMP_NUM_THREADS=4`:

$$S_{N=4} = \frac{4.500}{1.429} = 3.1491$$

where $S_{N=a}$ denotes the speed-up for a threads.

As we can see, the speedup is greater, when we use 4 threads on a Intel(R) Core(TM) i7-8565U CPU at 1.8GHz, which has 4 cores. When we increase the number of threads to 8, we see a little decrease in speed. It runs around 0.1-0.01 seconds slower than with 4 cores on average. When we increase to 16 threads, we see an even bigger decrease in speed. Here, it ran about the same speed as with 2 threads. This is due to the fact that the different threads having to wait for each other.

When comparing with the original python program, where the program runs for a short 8 minutes and 7 seconds - we do not know if the handed out code had any reason to be *this* slow - we see the following speed-up for

Authors' address:

`OMP_NUM_THREADS=1`:

$$S_{N=2} = \frac{487}{4.500} = 108.222$$

for `OMP_NUM_THREADS=2`:

$$S_{N=2} = \frac{487}{2.509} = 194.1012$$

and for `OMP_NUM_THREADS=4`:

$$S_{N=4} = \frac{487}{1.429} = 340.7978$$

As we can see, the speed-up is ridiculously large. Again, we do not know if the handed oyt python code had any reason to run at such a slow pace. If we set `use_clib = True` we see a run time below 2 seconds.

Amdahl's law. We are to report what fraction of the code can be parallelized. Amdahl's law is a theoretical law, but we can still use this i practice. If we have a concrete speed-up value $S(N)$ then we can isolate p in Amdahl's law:

$$S(N) = \frac{1}{(1-p) + \frac{p}{N}} \iff p = \frac{N \cdot (S(N) - 1)}{S(N) \cdot (N - 1)}$$

First we compare with the single-threaded C program. For $N = 2$ we had a $S(2) = 1.7935$ and therefore we get the following p:

$$p = \frac{2 \cdot (1.7935 - 1 =}{1.7935 \cdot (2 - 1)} = 0.8849$$

meaning that 88.49% of our program could be parallelized in this specific example.
For $N = 4$ we had a $S(4) = 3.1491$ and therefore we get the following p:

$$p = \frac{4 \cdot (3.1491 - 1)}{3.1491 \cdot (4 - 1)} = 0.9099$$

meaning that 90.99% of our program could be parallelized in this specific example.

We do not really see a point in doing this with the python program, as the numbers are so skewed.

Throughput. To calculate througput we first need to decide how we want to measure workload and time. For this we decided to go with amount of cells update as workload and the real time passed in milliseconds as time. Thus our troughput will show how many cells we calculate pr. millisecond.

Nr. of processors	Workload	Time	Troughput
1	$196 \cdot 10^6$ cells	4500 ms	43.555,55 cells/ms
2	$196 \cdot 10^6$ cells	2509 ms	78.118.77 cells/ms
4	$196 \cdot 10^6$ cells	1429 ms	137.158,85 cells/ms

4 FLOPS/BYTE

In `apply_stencil` we have 5 loads, 1 store and 5 float operations. This gives us $\frac{5}{6 \cdot 4} \frac{\text{FLOP}}{\text{Byte}}$. In `compute_delta` we have 2 loads, 1 store and 2 float operations. This gives us $\frac{2}{3 \cdot 4} \frac{\text{FLOP}}{\text{Byte}}$.

5 REFLECTIONS

From our calculations with Amdahl's law as well as visual inspection of the resulting bmp file, it is obvious that parallelization works as expected. We see from running the program on MODI as compared to our own machine, that

hardware makes a big difference. Trying to run more threads than the machine has cores will in some cases result in worse results than just optimizing the code to run fast sequentially.

We've gained an insight into what things you should keep in mind when constructing your code, if you want to later parallelize it. What seems like irrelevant decisions, when working with high abstraction programming languages, suddenly become important design choices. An example of this, is how you choose to build an array or iterate through it. While complex data structures might make abstraction and modelling of a problem easier in programming languages like Python, choosing to structure your data correctly can mean the world with regards to speed in C.