# DD2356 – Assignment 2

## Exercise 1

* This exercise is performed on Beskow, using GCC compiler. This flag ***-fopenmp*** needs to be enabled to compile the program.
* On Beskow, the flag -n is used to specify the number of nodes to be used.
* There are three ways to set the number of threads. The first method is to call the function **omp\_set\_num\_threads(X**), another way is to add **num\_threads(X)** to the pragma statement when creating parallel clauses. For example: **#pragma omp parallel num\_threads(4)**. The last method is to set the environment variable **OMP\_NUM\_THREADS** manually via **export** or **set**.

## Exercise 2

* The bandwidth was scaling proportional with the number of threads up to 8 threads, after that, the scaling flattened and ended up around 40-45 GB/s.

In the #pragma statement, the parameter **schedule(*policy*)** can be added to specify the scheduling of parallelization. The fastest schedule was *auto* , which might just assigned *static* to the scheduler.

## Exercise 3

N = 1 000 000

This exercise is performed on my home PC, with Ryzen 2600 (12 threads)

* Performance of the serial code: **average execution time = 0.82 ms, std = 0.03 ms**
* Performance of the parallelized version: **average execution time = 0.52ms, std = 0.03ms**

The output of this simple parallelized version was not correct. This is due to the race condition in the for-loop, where one thread is writing over the max value, but before it happens, other threads can also make it into the if clause and overwrite the value right after.

* Below is the graph of performance at different number of threads when using omp critical to protect the assignment of the max value. Omp atomic was not possible to implement because it does not support assignment of variables.

|  |  |  |
| --- | --- | --- |
| Version | Execution time (ms) | Standard deviation |
| Serial | 0.82 | 0.03 |
| Parallel without mutual exclusion | 1,24 | 0,49 |
| Parallel with mutual exclusion (12) | 0,90 | 0,27 |
| Parallel with mutual exclusion (8) | 0,64 | 0,20 |
| Parallel with mutual exclusion (4) | 0,46 | 0,05 |
| Parallel with mutual exclusion (2) | 0,51 | 0,05 |
| Parallel with mutual exclusion (1) | 1,77 | 0,54 |
|  |  |  |

The result is now correct, but the speed up is not ideal. Omp critical grants mutual exclusion of the protected section of code, it means only one thread at time can execute the code. There is increased overhead as the number of threads increases, and all the threads are basically waiting for each other, since there is nothing else to execute in the loop. 4 threads seem to be the sweet spot, where the benefit of parallelization is larger than the overhead. After that, the overhead of mutual exclusion has outgrown the benefit, hence the decreased performance.

* On the next page is the performance graph of the version where each thread finds its max value in its own dataset, then the results are combined at the end.

The performance did not increase as expected. It was almost the same as using omp critical. The reason could be false sharing, where the threads are updating different variables that are on the same cache-line. It causes cache conflicts and forces each thread to re-validate its variable, thus slowing down the execution.

* Below is the comparison of using padding to avoid false sharing with the version without padding.

In this exercise, the variable array is padded with 8 bytes. The speed up is more stable now.