# Assignment 2: Programming with OpenMP

**Exercise 1 - OpenMP Hello World, get familiar with OpenMP Environment**

**1. Write an OpenMP C code with each thread printing Hello World from Thread X! where X is the thread ID.**

**2.How do you compile the code in question 1? Which compiler and flags have you used?**

`cc -openmp  hello_world.c`(gray)

OpenMP flag:`-openmp`

**3.How do you run the OpenMP code on Dardel? What flags did you set?**

```
srun -n 1 ./hello_world.out
```

```
export OMP_NUM_THREADS=4
```

**4.How many different ways can the number of threads in OpenMP be changed? Which are they?**

Two ways.

`export OMP_NUM_THREADS=<number of threads to use>`(bash shell)

`setenv OMP_NUM_THREADS <number of threads to use>`(csh or tcsh shell)

**Exercise 2 - STREAM benchmark and the importance of threads**

**Exercise 3 - Parallel Sum**

**Measure the performance of the serial code (average + standard deviation).**

Out of 5 measurements with 10^7 elements we obtained:

- Average: 0.032559 s
- Standard deviation: 3.74e-5 s

**Implement a parallel version of the serial_sum called omp_sum and use the omp parallel for construct to parallelize the program. Run the code with 32 threads and measure the execution time (average + standard deviation). Is and should the code be working correctly? If not, why not?**

Using `omp parallel for` and 32 threads we get:

- Average: 0.052364 s
- Standard deviation: 0.00201

The code doesn't run correctly as multiple threads try to access and modify the sum variable concurrently, making it susceptible to race conditions.

**Implement a new version called omp_critical_sum and use the omp critical to protect the code region that might be updated by multiple threads concurrently. Measure the execution time for the code in questions 2 and 3 by varying the number of threads: 1, 2, 4, 8, 16, 20, 24, 28, and 32. How does the performance compare to the program in questions 1 and 2? What is the reason for the performance gain/loss?**

| | 1 | 2 | 4 | 8 | 16 | 20 | 24 | 28 | 32 |
|---|---|---|---|---|---|---|---|---|---|
| omp_sum | 0.033876 | 0.078455 | 0.056979 | 0.054617 | 0.053033 | 0.049611 | 0.054719 | 0.050377 | 0.051933 |
| omp_critical_sum | 0.069273 | 0.161543 | 0.173142 | 0.193622 | 0.197640 | 0.195231 | 0.191220 | 0.193787 | 0.193857 |

Performance is lost when using critical to protect the code as the code is essentially serialised as threads only access the sum one by one (nonetheless giving a correct result).

**Try to avoid the use of a critical section. Implement a new version called omp_local_sum. Let each thread find the local sum in its own data, then combine their local result to get the final result. For instance, we can use temporary arrays indexed by their thread number to hold the values found by each thread, like the code below.**

- `double local_sum[MAX_THREADS];`
- Measure the performance of the new implementation, varying the number of threads to 1,32,64, and 128 threads. Does the performance increase as expected? If not, why not?

As expected, this method gives the correct sum. We get the following performance depending on the amount of threads:

| | 1 | 32 | 64 | 128 |
|---|---|---|---|---|
| omp_local_sum | 0.044071 | 0.027366 | 0.015175 | 0.014685 |

The performance increases with number of threads, and is better than the serial sum.

**Write a new version of the code in question 4 called opt_local_sum using a technique to remove false sharing with padding. Measure the performance of the code by varying the number of threads to 1, 32, 64, and 128.** Using a 256 byte struct (double + 248 char padding) the following results are obtained

| | 1 | 32 | 64 | 128 |
|---|---|---|---|---|
| omp_local_sum | 0.052609 | 0.041504 | 0.027234 | 0.029865 |

There is an improvement in performance over the serial sum, but omp local sum seems to perform better overall despite not accounting for false sharing.

**Exercise 4 - DFTW, The Fastest DFT in the West**