Nuo Chen
nuoc@kth.se

DD2356 – Assignment 2

Exercise 1

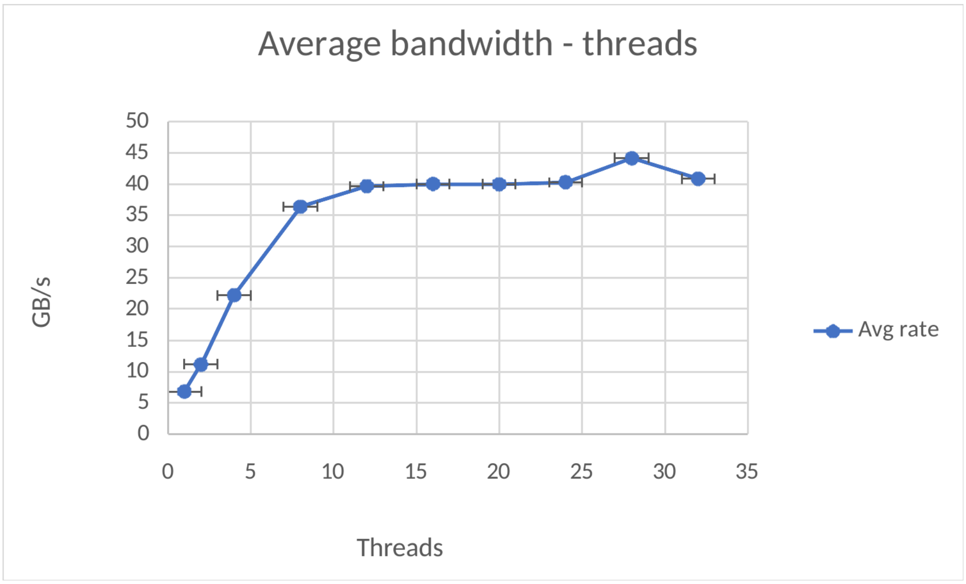
1.1 This exercise is performed on Beskow, using GCC compiler. This flag ***-fopenmp*** needs to be enabled to compile the program.

1.2 On Beskow, the flag -n is used to specify the number of nodes to be used. For example "srun -n 1 ./a.out"

1.3 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**.

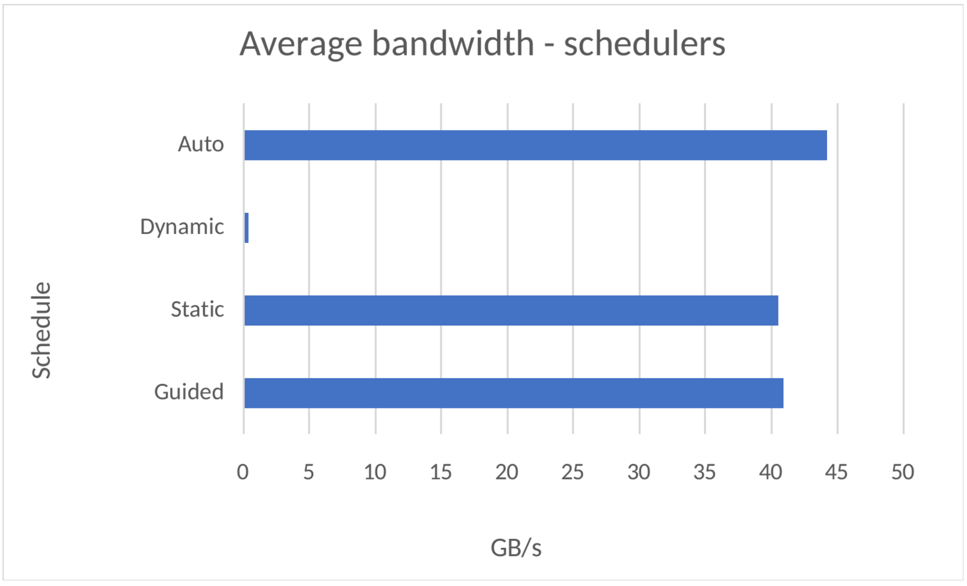
Exercise 2

2.1



2.2 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.

2.3



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

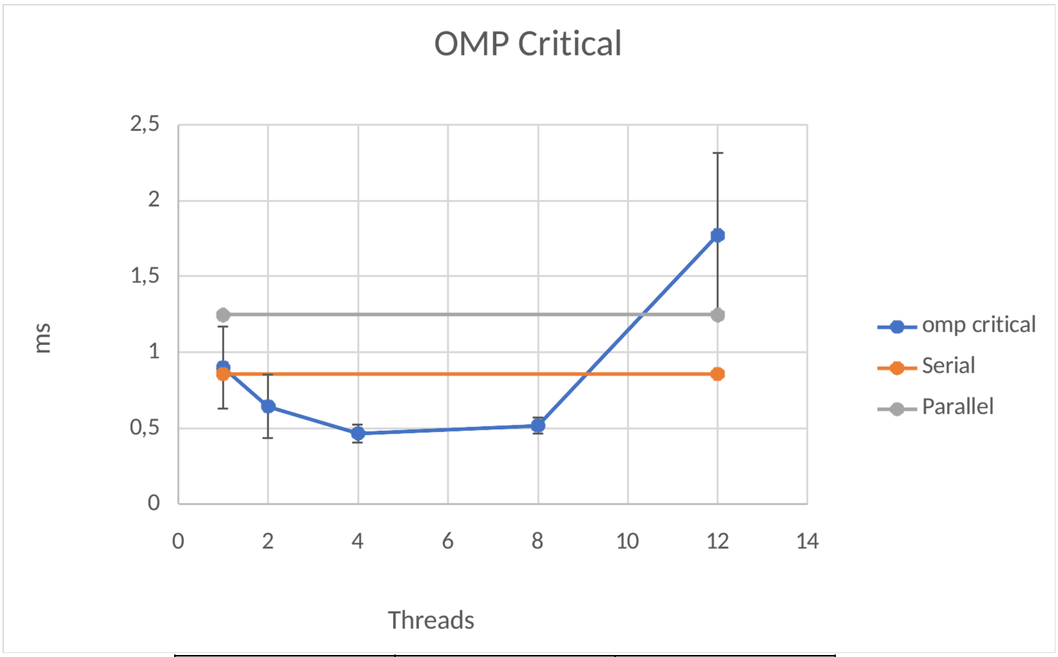
This exercise is performed on my home PC, with Ryzen 2600 (12 threads)

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

3.2 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.

3.3 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.

Nuo Chen
nuoc@kth.se

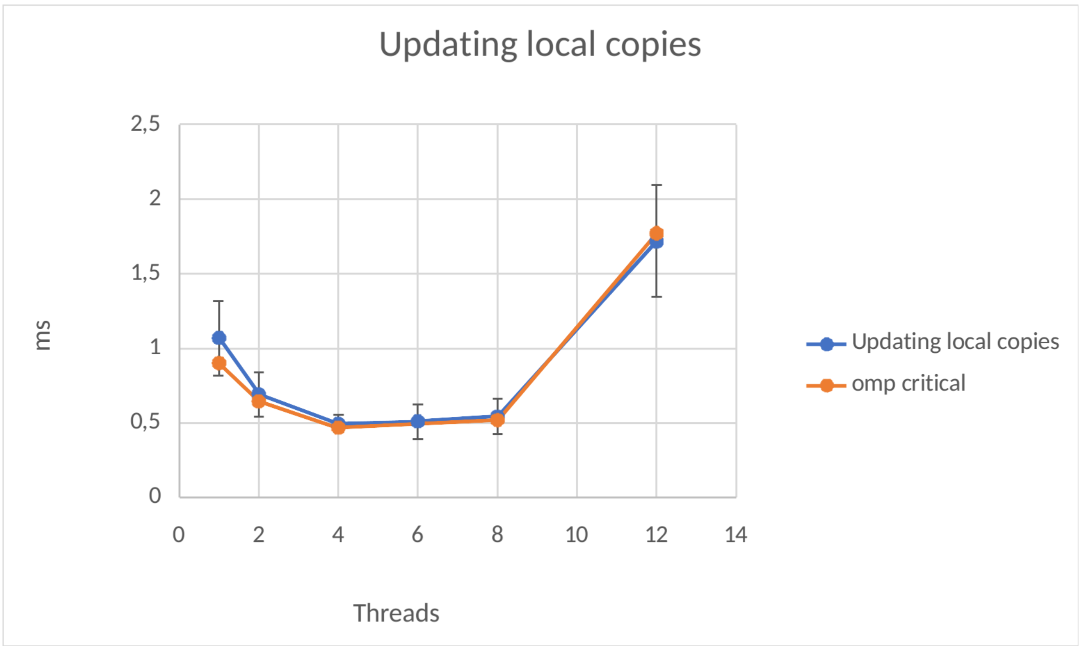


Version	Execution time (ms)	Standard deviation
Serial	0.82	0.03
Parallel without mutual exclusion	1,24	0,49
Parallel with mutual exclusion (12)	0,90	0,27
Parallel with mutual exclusion (8)	0,64	0,20
Parallel with mutual exclusion (4)	0,46	0,05
Parallel with mutual exclusion (2)	0,51	0,05
Parallel with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

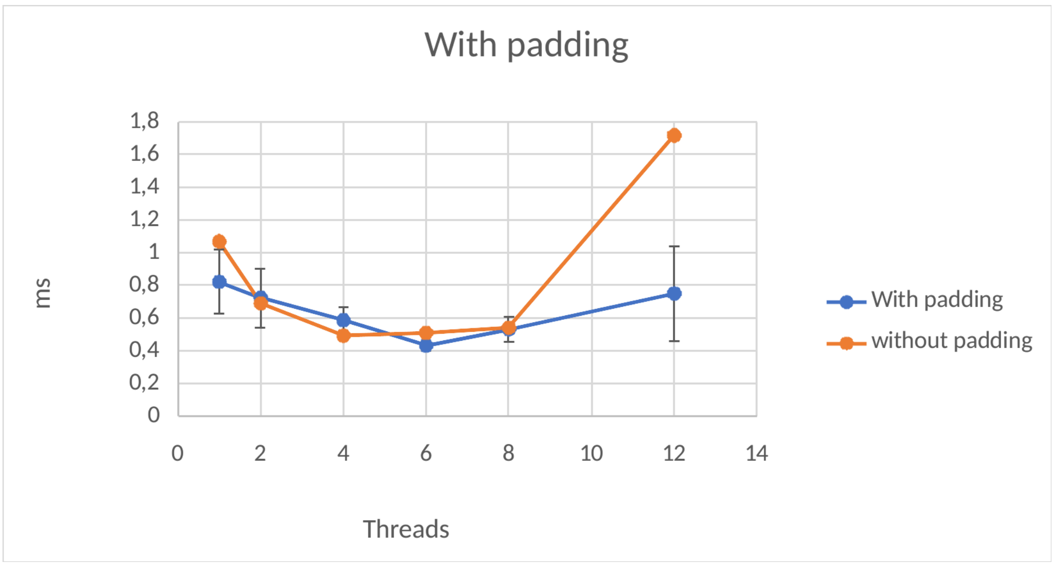
Nuo Chen
nuoc@kth.se

3.4 Below 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.

3.5 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 and a bit better now.

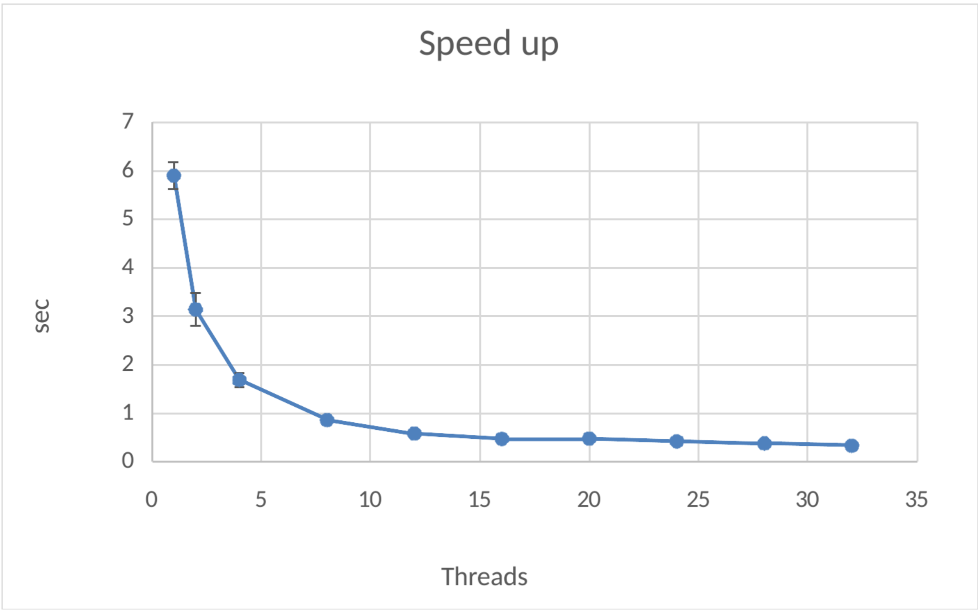
Nuo Chen
nuoc@kth.se

Exercise 4

4.1 The DFTW algorithm is parallelized by simply wrapping the outer for-loops with #pragma omp parallel for, because every thread is updating different elements in Xr_o and Xi_o.

4.2 The measured average execution time on Beskow machine is 0.33s with standard deviation is 0.008.

4.3 Below is the speed up graph on Beskow machine.



The speed up is proportional to the number of threads up to 8 threads, after that it becomes marginal.

4.5 Another performance optimization that is suitable for DFT is vectorization. It should be possible to represent the inner for-loop as vector operation. For example, in Python with Numpy I could write it as the following:

```
    for k in range(N):
        Xr_o[k] = xr * cos(np.arange(N)*k*PI2 / N) +
        idft*xi*sin(np.arange(N)*k*PI2 / N)
        Xr_i[k] = -idft * xr * sin(np.arange(N)*k*PI2 / N) +
        xi*cos(np.arange(N)*k*PI2 / N)
```

Nuo Chen
nuoc@kth.se

Exercise 5

5.1.1 Performance of serial simple and reduced algorithms using 100 cycles and delta t=0.05:

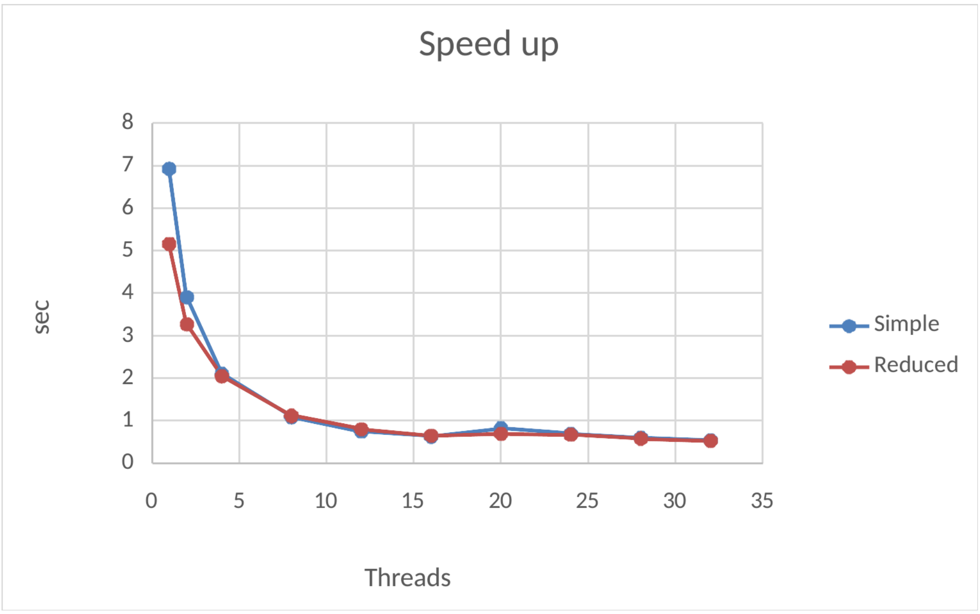
	500 particles	1000 particles	2000 particles
Simple	0.198 s	0.688 s	2.685 s
Reduced	0.159 s	0.593 s	2.289 s

5.1.2 The reduced version has better cache reuse, it is confirmed in perf log that it has lower cache miss rate than the simple version. It is probably due to better utilization of local spatiality. When loading and updating forces[k], all it neighboring elements will also be loaded into the cache, thu saving time on the subsequent iterations.

5.2.1 There is a risk of race condition in both versions, where some threads start to update particles' position and velocity while there are some others threads that have not finished updating the forces. It will cause inaccurate simulation results. One way to avoid this situation is to have two parallel sections, one section for updating the forces and another for updating the positions and velocities. Another way is to add a barrier between the two for-loops. Another potential problem is the local variables. They must be declared as private using the OMP construct, if there are defined outside the parallel section. Otherwise, these variables will be shared between all threads, making the result unpredictable.

It is also imortant not to parallelize the inner for-loop in both versions, otherwise there will be multiple threads updating the same force[q] at the same time, causing inaccurate results.

5.2.2 Below is the speed up graph on Beskow machine.



At low number of threads, the reduced version is significantly faster than the simple version, but this advantage is lost as the number of threads increases.

Nuo Chen
nuoc@kth.se

5.2.3 Below is the results of differnt scheduling policy with chunk size 32 on Beskow machine.

	Dynamic	Static	Guided
Simple	0,504	0,534	0,531
Reduced	0,475	0,526	0,556

The best scheduling policy is the dynamic (cyclic) scheduling.