# Parallel Computing HW0325

電機碩一 陳昕佑 61375017H

```
(base) rueii@rueii-Alienware-Aurora-R6:~/Graduation/First_Year/Parallel-Computing/WEEK6$ ./HW0325-example
Parallel elapsedTime: 213.401722 ms
Sequential elapsedTime: 4.060941 ms
Test pass!
(base) rueii@rueii-Alienware-Aurora-R6:~/Graduation/First_Year/Parallel-Computing/WEEK6$ ./Matrix_multiplication
Sequential elapsedTime: 15.387607 ms
Parallel elapsedTime: 13.049749 ms
Test pass!!!
(base) rueii@rueii-Alienware-Aurora-R6:~/Graduation/First_Year/Parallel-Computing/WEEK6$ ./HW0325_61375017H
Sequential (no transpose) elapsed time: 16.957482 ms
Sequential (with transpose) elapsed time: 12.189894 ms
Pthread (no transpose) elapsed time: 2.831054 ms
Pthread (with transpose) elapsed time: 2.189223 ms
(base) rueii@rueii-Alienware-Aurora-R6:~/Graduation/First_Year/Parallel-Computing/WEEK6$
```

The performance of the example code went against common sense; the sequential method is much faster than the parallel method, and this isn't uncommon. The code creates one thread per matrix element, which leads to launch $M * N = 100 * 100 = 10000$ threads, one per output element. This overwhelms the CPU's thread scheduler and creates huge thread creation + context switching overhead, far outweighing the benefit of parallelism.

The parallel elapsed time is much faster using OpenMP-based parallel matrix multiplication, and that's because OpenMP splits the outer loop over $i$ (rows) among all available threads. Each thread works on its chunk of rows in parallel, then all threads work simultaneously, leading to significant speedup. To accelerate matrix multiplication using Pthread:

1. Work Division by Row: Each thread is assigned a range of rows $(start = id * N/num\_threads)$ to process. This avoids thread creation overhead and makes each thread do meaningful work.
2. Parallelism Using pthread_create: Threads are created based on the number of logical cores using $sysconf(\_SC\_NPROCESSORS\_ONLN)$, a great way to scale across machines.
3. Transposed Matrix for Better Cache Locality: Multiplying $A * B^t$ allows the inner loop to access memory in a more cache-friendly manner, improving performance.

About the comparison between the original matrix and the transposed matrix, using a transposed matrix speeds up matrix multiplication mainly due to better memory access patterns, which improves CPU cache utilization. Modern CPUs use cache lines (typically 64 bytes). When the CPU loads memory, it grabs a whole line. Sequential access (like row-wise) means the next data is probably already in cache. But column access skips across memory, causing many cache misses, and each miss costs hundreds of CPU cycles.