Distributed Computing - Hackathon Report

When I first started the hackathon, my main objective was to create a solution for all the tasks in a simplistic form using only threads and vectors. When it came to the default template matrix size of 10000, my laptop couldn’t handle that. It would freeze and take forever to complete the process. To temporarily remedy this I reduced the size to 100 for easier testing purposes. To address this issue near the end of the hackathon I added multithreading methods to distribute computational tasks across multiple CPU cores making the process quicker than in it’s original state.

Trade-Offs and optimisations

The choice to use a thread pool introduced a trade-off between resource allocation and performance optimisation. Although multithreading offered better performance enhancements by distributing tasks in parallel, it introduced challenges associated with synchronisation.

Operation 1 – Matrix Transposition

When I began working on the “Matrix Transposition” task I first created the solution in rather simplistic form using only threads. This worked perfectly, however, I wanted to add more methods to this to improve the process and quality of my code. I first added “Barrier.h” to make it so a certain number of threads finish their processes before continuing and printing the final result matrix. I later changed this and decided to use mutex and thread pooling to make the process more efficient and effective.

Operation 2 – Zone Sum

For the "Zone Sum" task, I distributed the computation of neighbouring cell sums across multiple threads. This approach allowed for efficient utilisation of available resources while also reducing irrelevant computations. However, the granularity of task division and the idea of thread management required careful consideration to achieve optimal performance.

Operation 3 – Matrix Multiplication

Similar to the other operations, “Matrix Multiplication” utilised multithreading to distribute the computation across multiple CPU cores. However, due to the sequential nature of matrix multiplication, achieving significant performance improvements through this was challenging. After optimizing task scheduling and resource allocation within the thread pool, I managed to get an improvement in speed.