Program Decomposition and Parallelisation

During matrix multiplication, where C = A * B, and A and B are matrices of size N * N, the element $C_{i,j}$ is simply the sum of each element in matrix A row i multiplied by each element in matrix B column j.

Due to the nature of this calculation, each element $C_{i,j}$ is calculated completely independently from one another, making it an excellent candidate for parallelisation.

If we take the two simple matrices, $A = \begin{bmatrix} 1 & 2 \\ 2 & 0 \end{bmatrix}$ and $B = \begin{bmatrix} 0 & 2 \\ 1 & 1 \end{bmatrix}$ we can see that:

$C_{0,0} = (1 * 0) + (2 * 1) = 2$, and

$C_{0,1} = (1 * 2) + (2 * 1) = 4$

We can continue this for all elements of C, and the calculation of each element will remain fully independent from all others, since we are only using the values in matrices A and B, which will not change during the calculation.

To implement an efficient parallelisation solution to this problem, there needs to be a significant amount of work done by each thread. If not, the overheads caused by initialising, starting, and joining the threads together will negate the efficiency improvements of running the calculations in parallel, and can make the calculation run much slower.

Therefore, instead of assigning threads to calculate individual elements of C, I will be assigning threads to calculate *rows* of C. In doing so, MAX_THREADS will be limited to N rather than N^2. The goal behind this design choice is to provide the thread with enough work to do, even when MAX_THREADS is used, to outweigh the overheads explained earlier.

To dynamically parallelise the solution, each thread will be assigned a portion of the total rows of the matrix, defined by N / NUM_THREADS, with the remainder, if any, being allocated to the final thread. In practise, this might look like the following example:

A = 10 * 10 matrix, B = 10 * 10 matrix, NUM_THREADS = 3

$portion = floor\left(\frac{10}{3}\right) = 3$
$Thread\ 1: iStart = 0, iEnd = 3$
$Thread\ 2: iStart = 3, iEnd = 6$
$Thread\ 3: iStart = 6, iEnd = 10$ (remainder rows added to the load of the final thread)

As you can see, the work is dynamically split up amongst all the available threads. Each thread then performs the calculation on the rows they have been assigned, as defined by iStart and iEnd, on a reference to both matrix A and B (to remove time taken to copy parameters), and stores the resultant elements in their correct positions in a reference to the output matrix C. Since elements of C are calculated independently as explained before,

there are no problems with race conditions when using a direct reference as the array elements are also accessed independently.

Finally, all the threads can be joined in sequence and program execution can continue.

To summarise:

- **Sub-tasks of Matrix Multiplication**
  - Initialise matrix A and matrix B **SEQUENCE**
  - Populate matrix A with random values **SEQUENCE**
  - Populate matrix B with random values **SEQUENCE**
  - Initialise matrix C **SEQUENCE**
  - Calculate C = A * B
    - $C_{0,0} = \sum_{n}^{k=0}(A_{0,k} * B_{k,0})$ **PARALLEL**
    - $C_{0,1} = \sum_{n}^{k=0}(A_{0,k} * B_{k,1})$ **PARALLEL**
    - …. **…**
    - $C_{n,m} = \sum_{n}^{k=0}(A_{n,k} * B_{k,m})$ **PARALLEL**

PERFORMANCE REVIEW - std::thread vs sequence

As expected, for smaller matrix sizes the calculations are already very quick (<5ms) using the triple-nested loop in sequence method, and the overhead from using threads quickly becomes problematic for efficiency. For a matrix of size 50 * 50 (2500 elements), using just 2 threads yields an average execution time of 1.127ms, while the sequence solution yields an average execution time of ~0.7s. Additionally, using the maximum amount of threads (50) yields very inefficient results at 13.267ms.

It is quite clear that the number of threads used for maximal efficiency is very dependent on both the amount of work that needs to be done to solve the problem and the amount of work done by each thread. Clearly, there are some instances where using threads is not an ideal solution, such as this first test case.

```
Average Exec Time (NUM_THREADS = 2, MATRIX_SIZE = 50 * 50, NUM_EXECUTIONS = 1000)
Single: 0.730000ms
Threaded: 1.127000ms
```

```
Average Exec Time (NUM_THREADS = 50, MATRIX_SIZE = 50 * 50, NUM_EXECUTIONS = 1000)
Single: 0.695000ms
Threaded: 13.267000ms
```

Using 75 * 75 matrices (5625 elements) is when we start seeing some performance results when using 2 threads, but already begin to see some deterioration from using just 5 threads, and finally a significant performance impact using MAX_THREADS.

```
Average Exec Time (NUM_THREADS = 2, MATRIX_SIZE = 75 * 75, NUM_EXECUTIONS = 1000)
Single: 2.429000ms
Threaded: 2.336000ms
```

```
Average Exec Time (NUM_THREADS = 5, MATRIX_SIZE = 75 * 75, NUM_EXECUTIONS = 1000)
Single: 2.647000ms
Threaded: 2.868000ms
```

```
Average Exec Time (NUM_THREADS = 75, MATRIX_SIZE = 75 * 75, NUM_EXECUTIONS = 1000)
Single: 2.576000ms
Threaded: 19.379999ms
```

Finally, we can see that using 225*225 sized matrices (50625 elements) yields some interesting results. We can see a significant performance increase (almost half the execution time of the sequence solution) with just 2 threads, but the performance continues to be improved even at 10 threads, down to 34.75ms compared with ~80ms for the sequence solution.

We can begin to see the overheads coming into effect after this at 20 threads where we have a slower execution time, but still markedly faster than the sequence solution. Interestingly, even at MAX_THREADS, the threaded solution is **still faster** than the sequence solution, indicating that the amount of work involved to calculate a single row of the 225 * 255 matrix (225 elements) is enough to outweigh the performance hit of initialising, starting, and joining 225 threads when compared to calculating C in sequence.

```
Average Exec Time (NUM_THREADS = 2, MATRIX_SIZE = 225 * 225, NUM_EXECUTIONS = 100)
Single: 78.849998ms
Threaded: 45.169998ms
```

```
Average Exec Time (NUM_THREADS = 5, MATRIX_SIZE = 225 * 225, NUM_EXECUTIONS = 100)
Single: 80.889999ms
Threaded: 41.169998ms
```

```
Average Exec Time (NUM_THREADS = 10, MATRIX_SIZE = 225 * 225, NUM_EXECUTIONS = 100)
Single: 79.209999ms
Threaded: 34.750000ms
```

```
Average Exec Time (NUM_THREADS = 20, MATRIX_SIZE = 225 * 225, NUM_EXECUTIONS = 100)
Single: 78.370003ms
Threaded: 40.959999ms
```

```
Average Exec Time (NUM_THREADS = 225, MATRIX_SIZE = 225 * 225, NUM_EXECUTIONS = 100)
Single: 81.949997ms
Threaded: 77.360001ms
```

PERFORMANCE REVIEW – OpenMP vs std::thread vs sequence

For the test cases below, I used the most ideal NUM_THREADS value in order to make more accurate comparisons to the std::thread average execution time. Additionally, I used the directive "#pragma omp parallel for" on the outermost for loop of the matrix multiplication for the OpenMP example.

Right from the beginning, we can see that the OpenMP implementation has significant improvements over the std::thread implementation when the workload is smaller, presumably due to thread implementation differences that reduce overheads, or perhaps even better compiler optimisations. However, it is clear that the sequence implementation is still favoured when the workload is small, up until we get to the 50 * 50 matrices.

At this stage, the std::thread implementation is still over 100% slower than the sequence implementation, yet OpenMP runs ~0.15ms faster than the sequence solution.

As we increase the matrix size and the number of elements to calculate increases, we see that OpenMP and std::thread implementations get closer and closer together as the overheads from std::thread become more and more negligible due to the increased workload within the threads. This is best seen in the matrix size of 225 * 255, where the std::thread implementation runs just 22% slower than OpenMP, compared to 806% slower at N = 10, 249% slower at N = 50, and 157% slower at N = 100.

```
Average Exec Time (NUM_THREADS = 2, MATRIX_SIZE = 10 * 10, NUM_EXECUTIONS = 1000)
Single: 0.009000ms
Threaded: 1.153000ms
OpenMP: 0.143000ms
```

```
Average Exec Time (NUM_THREADS = 2, MATRIX_SIZE = 25 * 25, NUM_EXECUTIONS = 1000)
Single: 0.106000ms
Threaded: 0.845000ms
OpenMP: 0.134000ms
```

```
Average Exec Time (NUM_THREADS = 2, MATRIX_SIZE = 50 * 50, NUM_EXECUTIONS = 1000)
Single: 0.722000ms
Threaded: 1.467000ms
OpenMP: 0.588000ms
```

```
Average Exec Time (NUM_THREADS = 2, MATRIX_SIZE = 75 * 75, NUM_EXECUTIONS = 1000)
Single: 2.500000ms
Threaded: 2.320000ms
OpenMP: 1.212000ms
```

```
Average Exec Time (NUM_THREADS = 4, MATRIX_SIZE = 100 * 100, NUM_EXECUTIONS = 1000)
Single: 6.351000ms
Threaded: 5.030000ms
OpenMP: 3.205000ms
```

```
Average Exec Time (NUM_THREADS = 10, MATRIX_SIZE = 225 * 225, NUM_EXECUTIONS = 1000)
Single: 74.456001ms
Threaded: 32.426998ms
OpenMP: 26.549000ms
```

From these findings it is clear that the OpenMP implementation is favoured over the std::thread implementation for this scenario, but still doesn't compare to the sequence implementation for small workloads. It may be possible to further optimise the std::thread solution by converting A and B into 1-dimensional arrays but for this task I feel as though the comparisons are much more accurate when using the exact same logical implementations for matrix multiplication.

It may be possible for the std::thread solution to overtake the OpenMP implementation in terms of performance, but my program could not handle above N = 225, due to stack overflows.