# Introduction

This report aims to serve as an analysis and documentation for the process of parallelisation of the maxwell source code provided by the assignment. Three versions of parallelisation are needed for this assignment, and these are done following three programming models: with the help of OpenMP, The Message Passing Interface (MPI) and CUDA. This document contains information about the attempts made by the developer to parallelise with the given programming models, how the validation of each programming model was done, and if it is valid, the testing environment alongside with the results produced and an evaluation of the performance each version created produces, to the degree of completion each version it is.

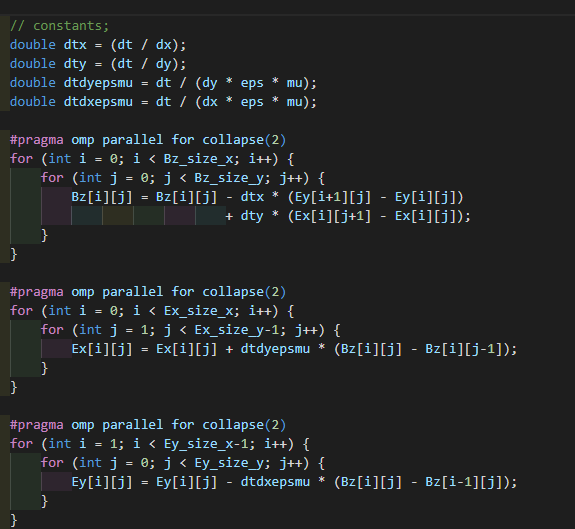
# Parallelisation Approach

Parallel computing/programming is a computer programming technique that enables parallel execution of operations. It uses multiple processors in parallel to solve problems more quickly than with a single processor. If you cannot increase the clock, do more operations by one clock. But, we cannot build an infinite processor due to temperature, cooling problems, interconnect bottleneck, etc. Performance gained by multicore processor strongly dependent on the software algorithms and implementation. [1]

## Approach taken for the parallelisation of programming model OpenMP

OpenMP (Open Multi-Processing) is an API that supports shared memory multiprocessing in C, C++ and Fortran. It provides an implementation of the Fork-Join model alongside some other parallel execution schemes and was first released for Fortran in 1997.

For this programming model, the approach was to do small and simple optimisations and then parallelise one of the most important functions of the maxwell program. Upon different tries, the decision to parallelise this program was by performing a parallelisation of the for loops from the update\_fields function and an example of a simple optimisation can be seen in this function as well, and this was done by adding this part inside the code and getting the constants out of the loop:



## Approach taken for the parallelisation of programming model MPI

The Message Passing Interface is a standard for passing data and other messages between running processes which may or may not be on a single computer. It is commonly used on computer clusters as a means by which a set of related processes can work together in parallel on one or more tasks. Unlike the OpenMP approaches to parallelism, the parallel strands of execution in a MPI environment do not share any memory: these strands (processes) must therefore communicate data and other information by passing messages between each other.

For this programming model, the approach was to do a halo exchange on the maxwell equation. First there was the initialization of the arguments and then setting up the number of processes and the ranking variable;

As we are doing a halo exchange, we distribute it equally to the number of processes, for this, the X variable would have to be divided into equal parts for each process into the size in the setup.

Next step, we need the ghost columns which are setup when we are allocating the arrays; Additional arrays that appeared the global\_E, and global\_B needed to gather all the visualisation, as now the X value is divided equally for the total number of processes. When setting up the guassian curve around the centre, we need to take into consideration the new value of X.

Now there is the creation of the MPIDatatypes as seen in the laboratory example; Now when updating the fields we need to send and receive the values. When setting up the boundaries it depends on the ranks, when resolving the Ex, Ey and Bz fields we need to consider the ghost columns as well. After each call of this function we put the barrier to get all the values of the E and B magnitude field performing a reduction to store all the values. And at the end we free the MPI datatypes created.

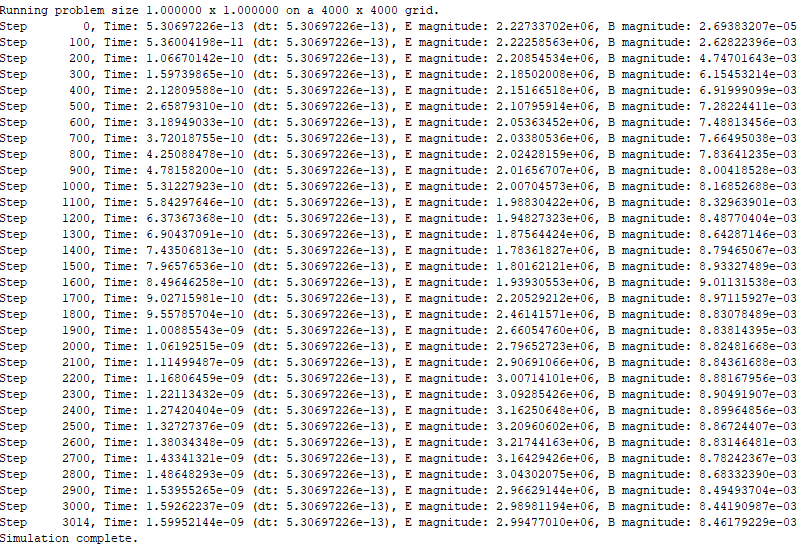
## Approach taken for the parallelisation of the CUDA programming model

# Validation

The validity of each programming model was done by comparing the log files produced by each version against the log file produced by the original code; The main validation method was done by checking the results of the log file for the default setup of 4000x4000 grid and from 100 to 100 steps values.

The most important values that were compared where those of the variables called: E magnitude and B magnitude from each 100 steps until the end of the execution.

The values that are given from the original maxwell program are:



## Validity of OpenMP variant

## Validity of MPI variant

## Validity of the CUDA variant

# Experimental Setup

A summary of the systems used for performance evaluation and an account of the process of collecting results.

The experiments were conducted on different setups.

The setups are as follows:

The university teaching laboratory which its hardware consists of:

The second experimental setup on which all the programming versions were tested on is the university’s supercomputer Viking.

And the machines CSE066 for CUDA testing, which consists of:

# Performance Evaluation

Appropriate data demonstrating the performance and scaling behaviour of your applications.

The performance was tested by doing a comparison between the execution time of each version of the program;

In theory, if all the parallelisation approaches were completely functional, the rankings of the performance should be around:

OpenMP vs Serial CPU: 7.1x faster

CudaSlow vs Serial CPU: 10.5x faster

CudaFast vs Serial CPU: 82.8x faster

CudaSlow vs OpenMP: 1.5x faster

CudaFast vs OpenMP: 11.7x faster

# Comparative Analysis – Conclusion for it

A comparative analysis of your three applications. And the conclusion.

# References

[1] K. S. Mohamed, ‘Parallel Computing: OpenMP, MPI, and CUDA’, in *Neuromorphic Computing and Beyond: Parallel, Approximation, Near Memory, and Quantum*, K. S. Mohamed, Ed. Cham: Springer International Publishing, 2020, pp. 63–93. doi: 10.1007/978-3-030-37224-8\_3.