**Intro to High Performance Computing – Distributed Memory Parallelism with MPI**

Introduction

For the assignment I set out to vastly improve the runtime performance of the *stencil.c* program using parallelism and MPI techniques. I have successfully achieved this aim by dynamically distributing the workload between a number of processes, making them communicate via message passing. Seen in Table 1 below are the final timing results for all three image sizes using 56 cores (2 nodes with 28 cores each).

Table 1 – Average timings for each image size utilising the full 56 cores available

|  |  |
| --- | --- |
| **Image size / pixels** | **Average Time / s** |
| 1024 x 1024 | 0.0123696 |
| 4096 x 4096 | 0.1484404 |
| 8000 x 8000 | 0.9179906 |

In this report I will be discussing how I implemented message passing and parallelising the program, along with analysing how the program scales when increasing the number of cores used. Additionally, I will be looking at the performance trends shown by the data exchange that takes place and how this limits execution regarding each image size.

MPI Implementation

When implementing Message Passing Interface(MPI) in order to parallelise a program, there are three main objectives: Initialising the data for each process, exchanging data between processes, collecting data once all computation has been completed.

Due to MPI implementing Single Program, Multiple Data (SPMD) from Flynn’s Taxonomy, each process will execute the same code. This leaves two ways in which to initialise data. The first way is to dynamically initialise the appropriate section of the image in each process. The second way (which I ended up implementing) is to initialise the whole image in each process, then only working on a select segment of the image. Despite the first method using less memory, it is much more difficult to implement correctly and provides no speed advantage in accordance to this assignment.   
After the image had been initialised, each rank dynamically calculated how many and which specific columns is will work on, based on the number of processes that had been allocated and the width of the image. The decision to decompose the image by columns rather than rows is due to the image being stored in column major.

When a case arises such that the number of processes doesn’t exactly divide the width of the image, the remaining columns (calculated using modulus) are assigned to rank 0, the *master* rank. Whilst this method is simple to implement, it is not the most efficient. This would be to distribute the remaining columns over several the processes, which I did not implement due to time constraints.

The *master* rank was introduced into my code in order to dictate certain functionalities of the parallel processing. In addition to processing the extra columns when needed, it is also where all the data is sent once processing has been completed and is the rank which outputs the resulting image to file.

Data exchange is required due to