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

Introduction

|  |  |
| --- | --- |
| **Image size / pixels** | **Average Time / s** |
| 1024 x 1024 | 0.0123696 |
| 4096 x 4096 | 0.1484404 |
| 8000 x 8000 | 0.9179906  Table 1 – Average timings for each image size utilising the full 56 cores available |

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). To ensure accurate readings, *MPI\_Barrier* was used to sync all processes to the same point before recording the time, ensuring that the slowest processing time was recorded.

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, and 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 in order to populate the halo regions within each process. Halo regions are section of the image adjacent to the section it is working on, which the process itself does not work on, but requires the information from. I have achieved successful, non-deadlocking message passing by using the *MPI\_Sendrecv* call.   
First, each process sends packs the appropriate data into the buffer and sends it to the previous rank, whilst subsequently receiving data from the rank succeeding it. Next, each process does the opposite; sends to the next rank and receives data from the previous rank. Executing the message passing in this way ensures that a process is not trying to send and receive data from the same rank at the same time, or two processes are both awaiting data from one another.  
For ranks 0 and (n-1), where **n** is the number of processes currently being used, they obviously do not have a predecessor or successor respectively. In these edge cases, when it comes to sending or receiving data from these non-existing ranks, the source/destination rank is set to *MPI\_PROC\_NULL*. This ensures that the *MPI\_Sendrecv* call is not waiting in deadlock to receive data, or for data to be received.

The final stage of parallel processing is collecting the data back in one place afterwards. I carried this out in an iterative process. Rank 0 would make **n** *MPI\_Recv* calls, where **n** is the number of processes currently being used, and every other rank would make an *MPI\_Send* call once the appropriate columns were packed into the send buffer. On receipt of the data, the master rank would calculate which columns it had been sent using the message’s source rank. Whilst not the most time efficient method, it works without deadlock and without error. An alternative method would have been to use *MPI\_Gather*, however this MPI call writes data in row major, and so this was not a usable option for my implementation.

Through implementing the stages above, I have accomplished a successful parallelised version of the *stencil.c* code, which meets the exceeds the ballpark times requested.

Performance of Data Exchange

Crucial to the success of MPI parallelism is the exchange of data between processes. By recording the average time of the data exchanges at increasing core count lets us see how much, if at all, it is a limiting factor in the execution speed of the program as a whole.

From Graph 1, we can see that for the smallest image size as the core count increases, the data exchange between ranks nearly matches total execution time towards the upper core counts. This tells us that the number of data exchanges being made is likely to be the sole limiting factor in the parallel performance of the code. As the number of cores utilised exceeds roughly 4, it is likely that the amount of data each rank must work on starts to fit into local cache, and here the program starts to become limited by the communication overhead. This is clearly seen by the sublinear speedup present on Graph 1. Additionally, as the number of cores used exceeds 28, a second node must be used due to Blue Crystal 4’s nodes having a total of 28 cores each. Transmission of data between nodes will take longer than intra processor on intra node communication due to physical distance between the nodes. This spike in communication times can be clearly seen on Graph 1 at 30 core count.

Looking at graphs 2 and 3, we can see that the data exchange effects two the larger image sizes to a much lesser degree. Whilst the two lines do tend towards each other over the course of scaling the number of processes used, it is to a much lesser effect than that found in Graph 1. On Graph 2 at the 32 core count, we see evidence of super linear speedup, which could be down to the data each image has to work on suddenly fitting into the local cache. After this point, we see a drop off in the gradient of the execution timeline, whilst the data exchange time also continues to trend upwards. This is clear evidence of the communication overhead starting to become the limiting factor in the performance for the 4096x4096 image size.  
Communication overhead doesn’t seem to become the limiting factor with the amount of processes that we are able to allocate in regards to the largest image size. Whilst the two lines do converge over the scaling up of cores used, there is still a significant difference in communication time and total execution time. If we were able to allocate more nodes, and subsequently more cores to the program, then we would likely see a super linear speedup, followed by a continued sublinear speed up causing the data exchange to then become the limiting factor with this larger image size.

Graph 1 – Comparing the total execution time with time taken for the data exchange for the image size 1024x1024 pixels

Graph 2 – Comparing the total execution time with time taken for the data exchange for the image size 4096x4096 pixels

Graph 3 – Comparing the total execution time with time taken for the data exchange for the image size 8000x8000 pixels

Scalability of the Implemented MPI

Analysing the scalability of the parallelisation that was implemented will allow us to infer how successful the implementation was, while also seeing how the size of the image effected the super and sub linear speedups that are present. Table 2 shows the average times for each image size when using 1 and 56 cores, along with the final speedup of the code.

Table 2 – showing the average times for each image size using 1 and 56 cores, as well as the calculated speedup as a result of parallelisation

|  |  |  |  |
| --- | --- | --- | --- |
| **Image Size / Pixels** | **Average Time on 1 Core / seconds** | **Average Time on 56 Cores / seconds** | **Total Speedup** |
| 1024x1024 | 0.1207 | 0.0124 | 9.73X |
| 4096x4096 | 3.0083 | 0.1484 | 20.27X |
| 8000x8000 | 11.1026 | 0.9178 | 12.10X |

As seen from Table 2 above, the middle image size had the greatest speedup over the entire parallelisation process. Better understanding of the speedups shown in Table 2 can be gained by looking ats graphs 4, 5, and 6.

Conclusion

L 0.5 page MAX

One way to reduce how much of a limiting factor the data exchange is to pack the data into a smaller data type. Currently in *stencil.c* I am sending the data as its default type *MPI\_FLOAT* which is 16bits. If I were to instead pack the data into an *MPI\_CHAR* then I could reduce the amount of data transmitted by 4x. Whilst there would be some computational overhead regarding the packing and unpacking of the data, it could still be a viable option to reduce the execution time, for smaller sized images especially, even further.