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). 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.

Image size / pixels	Average Time / s
1024 x 1024	0.0124
4096 x 4096	0.1484
8000 x 8000	0.9178

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

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 stages: 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 work 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 it 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 columns in the image adjacent to the section it is working on. The process itself does not work on the halo regions but requires its data to properly calculate the adjoining columns. I have achieved successful, non-deadlocking message passing by using the *MPI_Sendrecv* call.
First, each process 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 sending data to an incorrect rank.

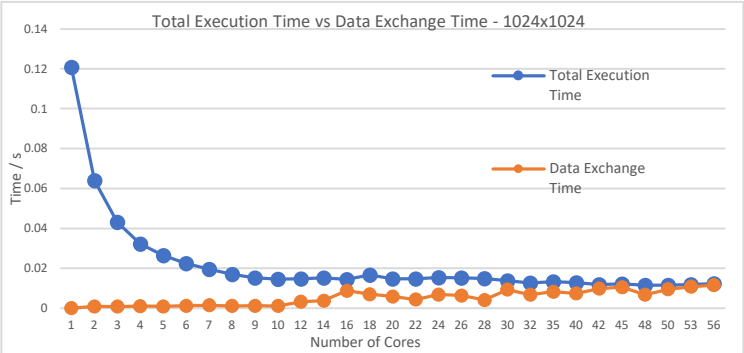
The final stage of parallel processing is collecting the data back in one place afterwards. I carried this out in an iterative process. Rank 0 would make (**n-2**) *MPI_Recv* calls, where **n** is the number of processes currently being used, and every other rank would make an *MPI_Send* call once the 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 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 when increasing core count lets us see how much, if at all, it is a limiting factor in the execution speed of the program.

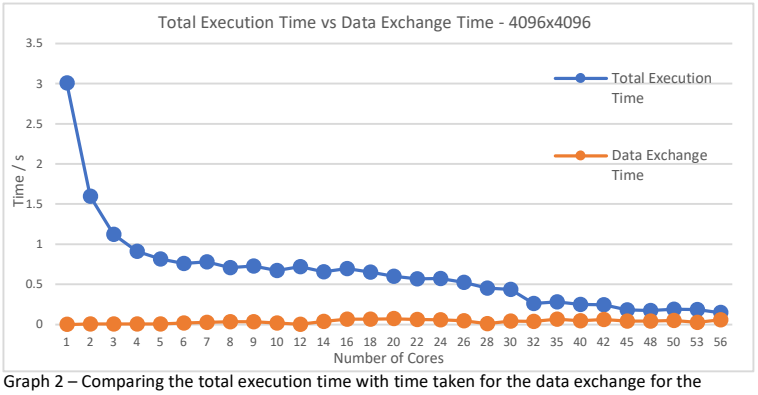
From Graph 1, we can see that for the smallest image size as the core count increases, the data exchange between ranks nearly matches total execution time towards the upper core counts. This tells us that the number of data exchanges being made is likely to be the sole limiting factor in the parallel 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.



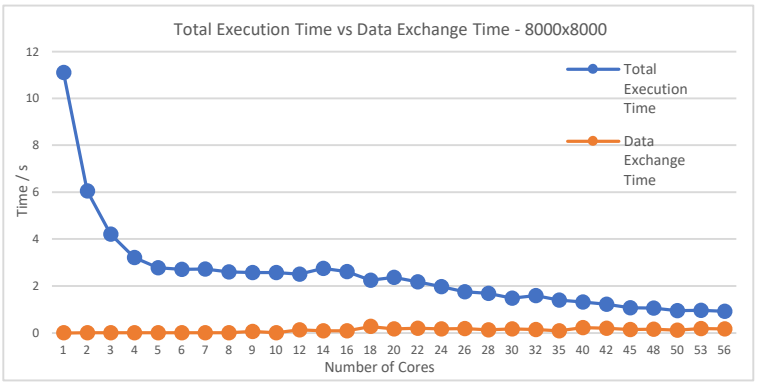
Graph 1 – Comparing the total execution time with time taken for the data exchange for the image size 1024x1024 pixels

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 continues its trend. 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 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

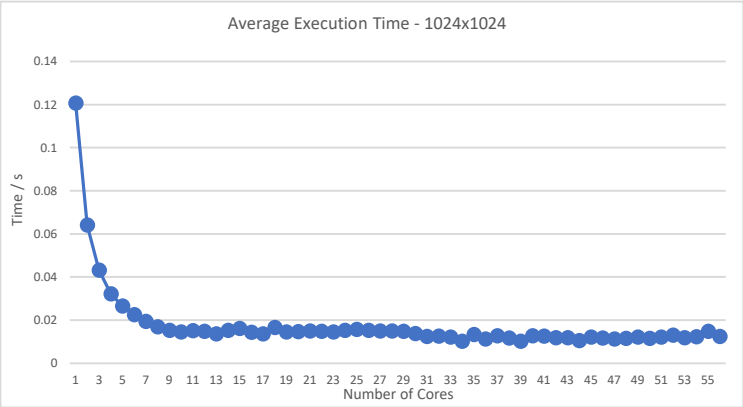
Scalability of the Implemented MPI

Analysing the scalability of the parallelisation that was implemented by using strong scaling allows us to infer how successful the implementation was, while also seeing how the size of the image was affected by 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.

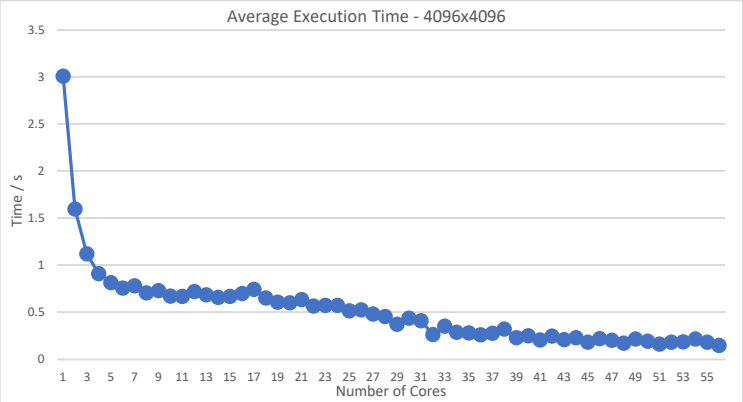
Image Size / Pixels	Average Time on 1 Core / seconds	Average Time on 56 Cores / seconds	Total Speedup
1024 x 1024	0.1207	0.0124	9.73X
4096 x 4096	3.0083	0.1484	20.27X
8000 x 8000	11.1026	0.9178	12.10X

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

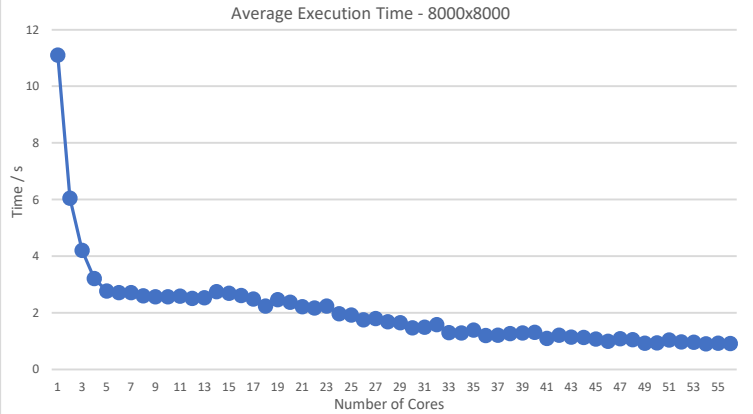
As seen from Table 2 above, although none of the image sizes worked on had perfect strong scaling, the middle image size had the greatest speedup over the entire parallelisation process. Better understanding of the speedups shown in Table 2 can be gained by looking at graphs 4, 5, and 6.



Graph 4 – Shows average execution times for every possible core count for 1024x1024 image size



Graph 5 – Shows average execution times for every possible core count for 4096x4096 image size



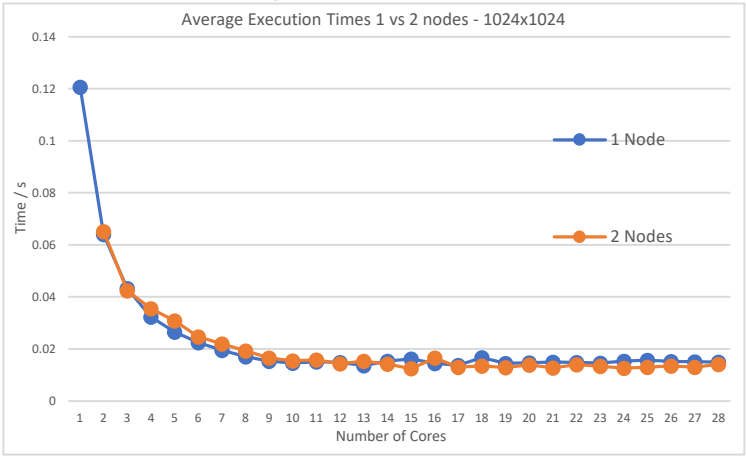
Graph 6 – Shows average execution times for every possible core count for 8000x8000 image size

Within graph 4 we can observe the quick drop off of the speedup when using 7 cores. From here onwards, we see a drastic sublinear speedup, caused by the data fitting into the core's cache early in the parallelisation process. This sublinear speedup has therefore affected the overall possible speedup that we could obtain, telling us that the smallest image size does not scale particularly well with more than about 8 cores.
This observation in the smallest image size's scalability can also be seen by calculating the parallel efficiency at both 10 and 56 cores. With 8 cores we get a strong scaling efficiency percentage of 89.3% (0.1207/(0.0169*8)), whilst this drops to 17.4% (0.1207/(0.0124*56)) for 56 cores.

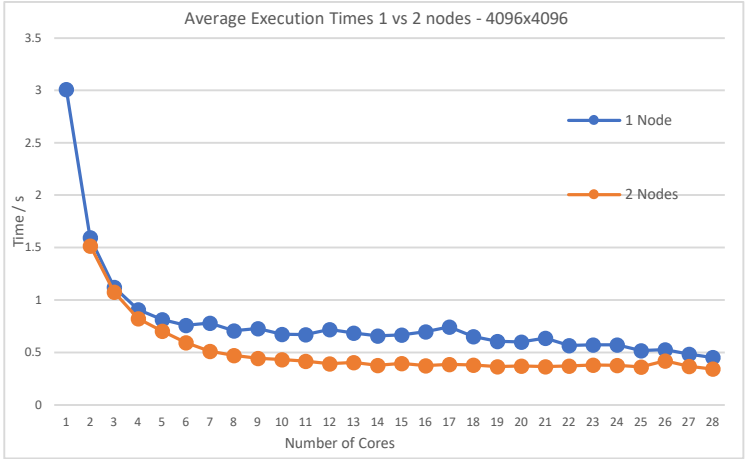
Looking at graphs 5 and 6, which represent the two larger image sizes, whilst initially we see the same similar sudden speedup within the first 4 cores, there are clear negative gradients to both graphs. Different to graph 4, graphs 5 and 6 still have noticeable decreases in execution time throughout the scaling up of the core count, with both beginning to level off as we approach 56 cores.

Although the latter section of Graph 6 looks like it is plateauing, the scale of the graph somewhat deceives the actual continued regression. Graph 5 on the other hand is portrayed more accurately and is correctly seen to plateau fully at around 40 cores. The later plateau of 4096x4096 compared to 1024x1024 explains the respectively higher speedup.

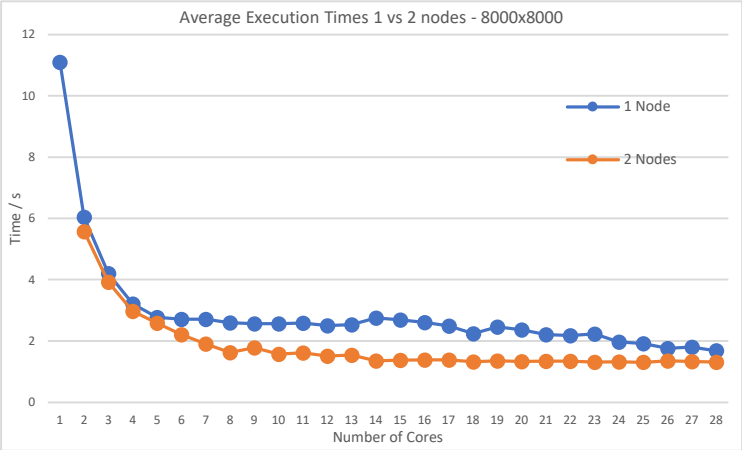
On the other hand, the substantial gain in speedup that 4096x4096 has on 8000x8000 is likely due to the data for the larger image not being able to fit within the core's cache. Evidence to support this can be seen from graphs 7, 8, and 9, where execution times were measured when scaling from 1 to 28 cores using both 1 and 2 nodes.



Graph 7 – Average execution times for all cores from 1 to 28, using both 1 and 2 nodes for 1024x1024 image size



Graph 8 – Average execution times for all cores from 1 to 28, using both 1 and 2 nodes for 4096x4096 image size



Graph 9 – Average execution times for all cores from 1 to 28, using both 1 and 2 nodes for 8000x8000 image size

In graphs 8 and 9 we can clearly see a significant speedup when using 2 nodes instead of 1 on core counts exceeding 4. Due to there being twice as much cache available with the addition of a second node, the processes are able to fit more data into cache much earlier in the scaling process. Additionally, we can also see from graph 7 that the addition of a second node makes little difference to execution time. Again, this supports what was said previously regarding fitting data within the cache on the smaller task size.

Looking at the latter two of the graphs, we see the times for single node and double node converge. For Graph 8, this makes perfect sense, as we see the timings plateau at about 40 cores from Graph 5, whereas this plateau occurs much earlier with the extra cache available from the second node.

Similarly, in Graph 9 whilst the two lines are converging, it is to a lesser extent than that of Graph 8. Rather than solely a cache issue, this is likely also due to each process having to compute on roughly double the data for the larger image, and so will converge at a slower pace.

Both of the larger images sizes have a fairly low strong scaling efficiency at 56 core usage, with 4096x4096 being 36.2% (3.0083/(0.1484*56)) and 8000x8000 being 21.6% (11.1026/(0.9178*56)), however both are better than the smallest image size. From these, and the data presented above we can say that *stencil.c* scaled best for the mid-sized image and was excessive for the smallest image size.

Regarding the largest image, there is enough evidence to suggest that scaling higher, with an additional node, would provide an increased speedup value, as well as an increased strong scaling efficiency. Not only would there be more cache available, but due to the amount of data that each rank must process, reducing this would also provide more speedup. Supporting this is Graph 3, where we can clearly see that the data exchange is not the limiting time factor, but instead is the computation itself.

Possible Improvements

Whilst my implementation of MPI parallelisation substantially improved the execution speed of *stencil.c*, some improvements could have been made to increase computation speed or make better use of STREAM memory bandwidth.

One way to do this would be 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_UNSIGNED_CHAR,* then I could reduce the amount of data transmitted by 2x. This is possible as the data we are dealing with has a value from 0 to 255, and so only 8bits are required 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, by reducing how much of a limiting factor the data exchange is.

Another way to improve use of STREAM memory bandwidth would be to share memory between processes that are running on the same CPU. This would eliminate the need for most processes needing to communicate at all, therefore decreasing data exchange time dramatically. Whilst there could be some accuracy issues regarding the data, this could be overcome with the use of MPI_Barrier to ensure all data was up to date before being read by another process.

A final improvement to speed up computation time would be to split left-over columns over a number of workers, instead of giving them all to the *master* rank. In certain instances, the *master* rank could have almost double the columns to work on compared to other ranks.in a case such as this, the increased core count would have been no use time wise, as the time output is that of the slowest.

Conclusion

Implementing MPI has proves successful with *stencil.c*. We have seen that the middle-sized task scales the most efficiently with the resources available and that with more resources, the larger task size could have provided better speedup. Although the smallest task size was executed quickly, it did not scale efficiently with the amount of cores present. Instead, it was limited by communication speeds rather than computation or memory restrictions.