High Performance Computing - MPI Assignment

Michael Rollins - mr16338

1 Introduction

In the first half of the High Performance Computing assignment we were tasked with optimising a piece of code which applied a weighted 5-point stencil across a grid. During this assignment I employed serial optimisation techniques in order to decrease the run time of the code, these included experimenting with different compilers and their optimisation flags, alternating from column-major to row-major grid access, removing conditionals and changing datatypes. From these optimisations I improved the run times on grid sizes of 1024px, 4096px, and 8000px down to 0.091s, 2.435s, and 8.480s.

In this assignment I was tasked with moving my code from a simple piece of serial code, to one that could make use of multiple processes using the C Message Passing Interface (MPI) in order to utilise parallel programming techniques to run the code across multiple processing cores, up to 16. This would involve segmenting the grid in a way of my choosing to spread the operational intensity across multiple processing units, which could run concurrently therefore reducing the run time of the code. Throughout this assignment I used the mpicc compiler with flags -Ofast and -march=native.

2 Parallel Implementation

The first step to this assignment was to get the stencil code I had written running on multiple processors. To do this I first had to decide how to divide and distribute the grid, I chose to divide horizontally, as my serial code already used row-major accessing and processing it this way made the most sense as it resulted in the simplest code. Distribution works by having all processes initialise the full sized chequerboard, but using the size of the board and number of processes, they each calculate the first and last rows which each process is acting on, and then having it only run the stencil code on those rows. After this the next steps is to run the stencil algorithm over the sections of the board, the calculations of the top and bottom rows of each sections requires receiving information from the processes calculating the sections of the board directly above and below it. This is done using halo exchange of the rows needed, so each process will send the bottom row below, and top row above - and receive a row from above and below. Finally, as each process has it's own copy of the board with its sectioned blurred, these images need to be 'stitched' together

into the final output image, this was done by having all ranks send their board to the master process which then added the section each process had operated on into its master image. Below I show I table of the run times I achieved with my first implementation using the MPI library:

Run times			
Cores	1024x1024	4096x4096	8000x8000
1	0.0953	2.620	8.597
4	0.0300	0.695	2.495
8	0.0140	0.551	1.966
12	0.00973	0.521	1.950
16	0.00790	0.538	1.953

3 Attempted Optimisations

The first change I made to optimise was to experiment with different kinds of message passing provided by the MPI library. In my first implementation I used the basic MPI_Send() and MPI_Recv() functions, these happened consecutively so the first change I tried was to implement the MPI_Sendrecv() function however as this is functionally the same as doing a MPI_Send() followed by and MPI_Recv() there was little to no difference in the run times it generated. MPI_Send() can be either asynchronous or synchronous depending on whether or not there is a system buffer, I know that when ran on Blue Crystal phase 3 it acts asynchronously as I have all of my process running MPI_Send() at the same time, which if it was acting synchronously would deadlock as none of them would be receiving what is being sent, so the function would never return. As my first two attempts were with asynchronous communication, it seemed logical to attempt to use the MPI_Ssend() function which acts synchronously, meaning it will not return until the process it is passing a message to receives it. To make use of this function I edited my code so that first the processes with even rank numbers sent their messages whilst the odd ranks received them, and then swapped. The run times produced however were actually slower than my first run times, for example on 16 processes running on square grids of size 1024px, 4096px, and 8000px respectively the run times were 0.00843s, 0.550s, and 1.974s. This is because all processes have to wait until all processes are either sending or receiving at the same time, where as in an asynchronous implementation a process can continue execution as soon as the process it is receiving from has placed its row in its receiving buffer.

In my initial implementation to send a row, it first was copied into a send buffer which is then passed into the message sending function, this is then copied into a receiving buffer in the next process. This is fairly memory intensive as it involves copying large data arrays. As my code is row major, it means within the array a row is a contiguous piece of data, which means I could pass a pointer to the first cell of the row to the the MPI_Send() function and then it would send the number of elements specified in the function parameters which would equate

to the row I intended to send. This did make a noticeable, if slight, improvement to some of the run times. This is because it removed the operation of copying the row into a sending buffer, which when there are 8000 rows is a costly operation, below I have shown the run times for the same number of cores as in the first table:

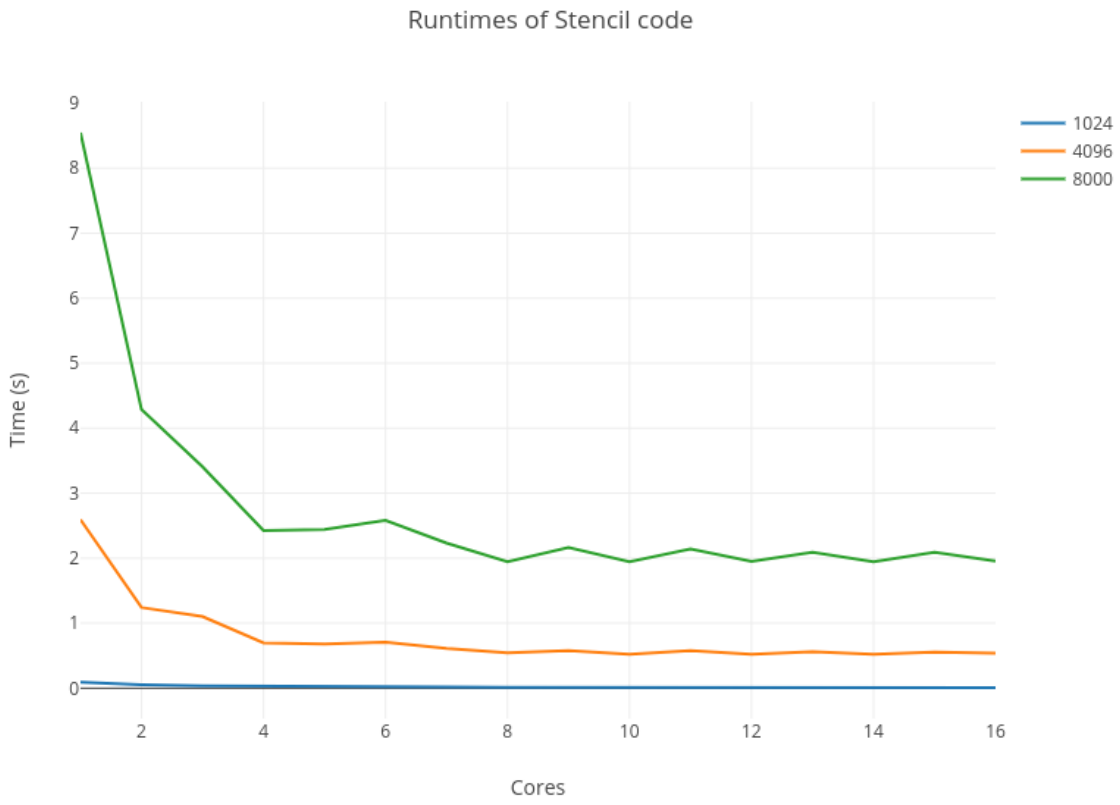
Run times			
Cores	1024x1024	4096x4096	8000x8000
1	0.0912	2.593	8.547
4	0.0285	0.698	2.425
8	0.0130	0.547	1.948
12	0.00931	0.520	1.948
16	0.00754	0.537	1.955

4 Performance Analysis

Comparing the run times achieved by the code which makes use of the MPI library to the run times of the serial code shows large decreases in their run times, on a grid of size 1024x1024 the MPI code running across 16 cores was 12.1 times faster, on the grid of size 4096x4096 it was 4.53 times faster and on the 8000x8000 grid it was 4.33 times faster. This shows that by distributing the graph across multiple cores makes a significant improvement in the run times possible. The reason the smallest size of grid has a much larger relative increase in speed than the two larger grids is because this computation is memory bandwidth bound, which these figures show. As you increase the number of processes running in parallel you reduce the amount of computation each rank needs to perform, however the communication between processes has a significant overhead so as you increase the number of process the amount of computing time decreases but the time for communication increases limiting the extent to which this program can be sped up.

The graph below shows how the three different sized grids performed when operated on my increasing numbers of cores. The graph shows the how the run times decrease quite dramatically at first and then plateau starting at around 4 cores, this is when the overhead of the communication in between cores will start to become too great for the improvements made by reducing the amount of computation each process is doing to continue to improve the run times. The shape of the curves on this graphs indicate that his code scales with a sub-linear plateau, meaning as you add more processes there is a less than linear increase in performance. It may be interesting to test the code with even more processes to see if the run times will eventually start to increase again as the time for communication starts to out weight the speed up gained by reducing the amount of computation needed. As you can also see, most obviously with the 8000x8000 grid, the code performed most quickly on an even number of cores because this most evenly distributed the rows amongst the processes. In my code any remaining rows after the total number of rows were floor divided

by the number of processes were given to the process with highest rank, so all the other processes had to wait for it to finish its extra computation before they could proceed with the next iteration.



5 Conclusions

In conclusion, the times achieved using the MPI library across 16 cores were significantly faster than those that were produced when running the code on a single core, however this was limited by the latency of communication between different process. I did manage to reduce this in a small way by removing the need to copy a row into a buffer before sending it, by instead sending the address of the first element in the row and the number of elements succeeding it which were in the row. If this was an operational piece of code that had operating constraints of both time and resource usage, it would most optimally be run across around 8 cores as this achieved a very similar level of performance without the need to take up as much of Blue Crystal's resources.