Introduction to High Performance Computing
Assignment 2: Distributed Memory Parallelism with MPI

Finn Hobson - fh16413

December 2018

1 Introduction

This report discusses the attempts I made to parallelise my optimised 5-point stencil code to make it run as fast as possible on multiple cores of BlueCrystal Phase 3.

For every runtime used in this report, I ran my code three times on BlueCrystal Phase 3 and took an average of the three times shown in the output files. This ensures that all of my timings are as accurate as possible and there are no anomalies.

2 MPI Parallelisation

2.1 Choice of Compiler and MPI Library

After optimising my serial stencil program during the first assignment, my code ran faster when compiled using Intel C++ Compiler version 16.0.2 (`icc`) than the GNU Compiler Collection version 7.1.0 (`gcc`). Because of this, I decided to continue using Intel's compiler along with the Intel MPI Library version 5.1.3 (`mpiicc`). This would allow all of my serial optimisations to continue to be effective in my parallelised version of the stencil code.

2.2 Domain Decomposition

Before paralellising my stencil code, I had to decide how the computational work should be divided among parallel processes. For the stencil code, the most effective way to decompose the problem is to divide the checkerboard into blocks and make each rank execute the `stencil` function on one of these blocks in parallel. The code applies a stencil to each pixel using the neighboring pixel values, so for the pixels on the edges of a block, the core will need access to pixels held by a different rank. Therefore, each rank will need to learn the pixel values adjacent to its block by communicating with neighboring ranks. This process is known as a halo exchange.

Functional decomposition would not be an effective method of parallelising the stencil code because as shown by GProf, approximately 99% of the computational work is done in the `stencil` function and this function cannot easily be divided into subtasks.

There are three possible options when dividing the checkerboard into blocks - tiles, rows and columns. Tiles would mean each rank could have up to four adjacent ranks to communicate with, whereas diving the checkerboard into rows or columns would mean each rank will only have a maximum of two adjacent ranks to communicate with. This means dividing the checkerboard into tiles could make the halo exchange more complex, potentially increasing the communication overhead of the stencil code. Also, it should be possible to execute my stencil code on any number of cores from 1 to 16 and when the stencil code is being executed using an odd number of cores, it would be very difficult to divide the checkerboard into tiles of approximately the same size. My checkerboard is arranged in a 2-dimensional array, where each row of the checkerboard is stored as a 1-dimensional array. If the checkerboard is divided vertically into columns, each rank will be working on a portion of every 1-dimensional array in the image array. If the checkerboard is divided horizontally into rows, each rank will be working on a number of complete 1-dimensional arrays where all of their pixels are adjacent to each other in memory. This sequential memory access means the correct pixel data is more likely to be stored in cache making each rank process their section of the board faster. If the number of rows in the checkerboard does not divide exactly among the cores, this can easily be solved by giving the remainder rows to the final rank. Therefore, dividing the checkerboard into rows is the most effective method of domain decomposition.

After dividing the checkerboard in this way and assigning a section to each rank, I ran my stencil program on 16 cores without any communication between the ranks. This produces an incorrect output image, but it shows the reduced computation time of my stencil code, without the communication overhead that will be added once the ranks are performing the halo exchange. The runtimes of my stencil code at this stage were 0.0206 seconds for the 1024x1024 pixel image, 0.507 seconds for the 4096x4096 pixel image and 1.90 seconds for the 8000x8000 pixel image.

2.3 Further Serial Optimisation

Even without communication between the 16 cores, my stencil code was not quite achieving the ballpark runtime for the 1024x1024 pixel checkerboard. I therefore decided to first see if there were any further serial optimisations I could make, in addition to the optimisations I made during the first assignment to increase the performance of my code.

One optimisation that I attempted during the first assignment was removing the `if` statements from the `for`-loops within the `stencil` function because branching within loops is a very computationally-expensive operation. However, the optimisation was not successful and unexpectedly, the runtime of my code increased. I tried this optimisation again, but this time instead of adding each weighted pixel value to the new pixel value in a separate statement using the `+=` operation, each new pixel value is calculated in a single statement. For each non-edge pixel calculation, this change has decreased the number of bytes loaded from 36 to 20, the number of bytes stored from 20 to 4 and the number of arithmetic operations from 9 to 6. Therefore, the operational intensity of my stencil code has increased from 0.161 to 0.250.

This serial optimisation reduced the runtime of my code on 16 cores (without communication between the ranks) for the 1024x1024 checkerboard from 0.0206s to 0.00758s (2.7 times faster). However, interestingly, there was no noticeable decrease in the runtime of my code for the other two image sizes. This shows that for the 4096x4096 pixel image and the 8000x8000 pixel image, the code is memory bandwidth bound because an increase in the speed of computation has not increased performance.

2.4 Communication Between Processors

In order to produce a correct output image, every time new pixel values are computed by the `stencil` function, each rank must send their edge rows to the adjacent ranks. MPI offers a number of options when performing point-to-point message passing between parallel processes so I decided to explore which of these would give my stencil program the smallest communication overhead.

I started by using blocking communication functions `MPI_Send` and `MPI_Recv`. After executing the `stencil` function, each rank prepares to receive a 1-dimensional array of pixel values from the previous rank into a buffer by calling the `MPI_Recv` function. Once received, the rank unpacks the pixel values from the buffer into the correct row of the image array. Then the rank packs the final row of its section into a buffer for sending and sends it using the `MPI_Send` function. In order to prevent deadlock, rank 0 sends to rank 1 first then the cores communicate in a chain by receiving and then sending. This process is then repeated, but in the opposite direction so each processor sends their top row to the previous rank. This implementation now produces a correct output image. Table 1 shows the runtimes of my code at this stage along with an approximate communication overhead that has been calculated using the runtimes of the code without any communication between cores. For the 1024x1024 pixel image, the communication overhead accounts for approximately 63% of the total runtime, meaning this method of communication between cores is significantly slowing down the code.

Image size (pixels)	Runtime with communication (seconds)	Runtime without communication (seconds)	Communication overhead (seconds)
1024x1024	0.0205	0.00758	0.0129
4096x4096	0.593	0.507	0.0860
8000x8000	2.09	1.90	0.190

Table 1: Runtimes of my stencil code with and without communication between 16 parallel processes, and the approximate communication overhead based on these values.

As the ranks are sending and receiving complete 1-dimensional arrays from the 2-dimensional image arrays, I realised that the rows could be sent and received directly to and from the image arrays rather than using buffers. This change eliminated all of the time that each rank spent packing and unpacking the buffers, which reduced the communication overhead. This reduced the runtime of my stencil code for the 1024x1024 pixel image to 0.0161 seconds, for the 4096x4096 pixel image to 0.558 seconds and for the 8000x8000 pixel image to 2.01 seconds.

As mentioned previously, I have prevented deadlocking by sending pixel values from rank 0 to rank 1 first, then each rank receives and then sends in a chain until the bottom of the image array is reached. However, this means that each rank is waiting until the rank before has received and sent. This waiting time increases the communication overhead of my stencil code. To improve this, I changed the communication pattern so that processes with an even rank number send first and then receive, and processes with an odd rank number receive first and then send. This means that half of the ranks are able to send their row of pixels immediately so the time spent waiting is reduced. This reduced the runtime of my stencil code for the 1024x1024 pixel image to 0.00972 seconds, for the 4096x4096 pixel image to 0.525 seconds and for the 8000x8000 pixel image to 1.97 seconds.

As `MPI_Send` and `MPI_Recv` are blocking functions, they do not return until a matching send or receive has been posted and data transmission has completed. This means that processors are spending time waiting to receive before they send a row of pixels themselves. In order to further reduce the time that each rank spends waiting, I decided to instead use non-blocking communication functions `MPI_Isend` and `MPI_Irecv`. These functions return immediately after requesting communication so the core can continue with computation. However, using these functions could make the image arrays unsafe for re-use as the updated pixel values may not have been received when a rank proceeds to execute the `stencil` function. To prevent this from happening, I have used the `MPI_Wait` function to check that the receive requests have been completed before allowing a rank to proceed to execute the `stencil` function. This change reduced the runtime of my stencil code for the 1024x1024 pixel image to 0.00879 seconds, for the 4096x4096 pixel image to 0.510 seconds and for the 8000x8000 pixel image to 1.91 seconds.

These changes to the communication between the parallel processes have significantly reduced the communication overhead of my stencil code to just a number milliseconds for each image size.

3 Analysis of my Code

3.1 Scalability

Once my stencil code was running under the ballpark runtimes while using 16 cores, I decided to test how it performs scaling from 1 core to 16 cores. Table 2 shows the runtimes achieved for each number of cores on each image size.

For the 1024x1024 pixel image, increasing the number of cores increases performance almost linearly. For example, increasing the number of cores from 1 to 2 reduces the runtime from 0.119 seconds to 0.0641 (1.9 times faster) and increasing the number of cores from 4 to 16 reduces the runtime from 0.0315 to 0.00871 (3.6 times faster). The increase in communication overhead as the number of cores increases means that performance does not scale perfectly linearly.

However, the performance increase for the other two board sizes is sublinear and the runtimes plateau. For the 4096x4096 pixel image, increasing the number of cores from 1 to 8 reduces the runtime from 3.13 seconds to 0.514 seconds (6.1 times faster) and there is almost no improvement in the runtime after this as 16 cores achieve a runtime of 0.510 seconds. Similarly, for the 8000x8000 pixel image, increasing the number of cores from 1 to 8 reduces the runtime from 9.01 seconds to 1.93 seconds (only 4.7 times faster) and there is almost no improvement in performance after this as 16 cores achieve a runtime of 1.91 seconds.

Number of Processes	1024x1024 Checkerboard Runtime (seconds)	4096x4096 Checkerboard Runtime (seconds)	8000x8000 Checkerboard Runtime (seconds)
1	0.119	3.13	9.01
2	0.0641	1.85	4.98
3	0.0434	1.39	3.43
4	0.0315	1.17	4.33
5	0.0270	1.01	3.77
6	0.0246	0.679	3.16
7	0.0200	0.583	2.20
8	0.0159	0.514	1.93
9	0.0146	0.564	2.11
10	0.0131	0.513	1.92
11	0.120	0.553	2.06
12	0.0119	0.516	**1.91**
13	0.0110	0.549	2.08
14	0.0107	0.515	1.92
15	0.00969	0.550	2.06
16	**0.00871**	**0.510**	**1.91**

Table 2: Final runtimes of my parallelised stencil code scaling from 1 core to 16 cores on 3 different checkerboard sizes.

The reason for this sublinear plateau is that the larger board sizes are memory bandwidth bound because they have so many pixel values to access from memory. This means that the increase in the speed of computation that happens when increasing the number of cores does not increase performance because memory cannot be accessed any faster by the cores. The larger boards also have a larger communication overhead as more pixels must be sent between the cores.

3.2 Roofline Model

I decided to investigate what fraction of STREAM memory bandwidth my stencil code is achieving for each image size while using 16 cores.

For the 1024x1024 pixel image, my stencil code performs approximately 5.05GB of memory transfers (load or store) in 0.00871 seconds, giving a memory transfer rate of 580GB/s. This means performance using 16 cores on this image size is bound by L2 cache memory bandwidth which has a peak of 780GB/s on BlueCrystal Phase 3. This performance achieves approximately 74% of STREAM L2 memory bandwidth.

For the 4096x4096 pixel image, my stencil code performs approximately 80.6GB of memory transfers in 0.510 seconds, giving a memory transfer rate of 158GB/s. For the 8000x8000 pixel image, my stencil code performs approximately 307GB of memory transfers in 1.91 seconds, giving a very similar memory transfer rate of 160GB/s. This means performance using 16 cores for both of these images sizes is bound by L3 cache memory bandwidth which has a peak of 418GB/s on BlueCrystal Phase 3. This performance achieves approximately 38% of STREAM L3 memory bandwidth.