COMS30005: Report on Parallelism with MPI

Yi-Ching Chen - yc16011

I Introduction

In this coursework, the objective is to reduce the run-time of stencil more than the serial optimised stencil code by using Message Passing Interface to implement parallelism. To find a better method, I made different approaches at different stages and will all be discussed in the report. The code is compiled with mpicc, because mpiicc did not give me a better run-time, but is 90% higher than the results I got from mpicc, although it gives a more consistent run-time comparing to mpicc.

II MPI Implementations

1 Set-up

First of all, environment is set up for MPI to work by add `MPI_Init()` and `MPI_Finalize()`. This can be thought as a for loop where in each iteration the code between the two lines are ran by each individual core involved. Then I used `MPI_Initialized(&flag)` to double check that MPI has been initialized. For further computation and division of work, total number of cores and the current rank number are acquired and are stored in `rank` and `size`.

2 Design

There are different ways of distributing the workload for each core to process. Instead of having the master rank (rank 0) only acting as a boss who divides and distributes the workload to each rank, I also use the master rank to perform computation on one part of the image as well, so all 16 cores can be utilised.

There is a possibility where the image can be divided equally among all the ranks, by giving the last rank the extra work, this creates an unbalance in the workload and results in every other rank waiting for the last rank and cause longer time. Therefore I found a way to distribute it more equally. Instead of adding on a calculated size of array based on a fixed height `(nx/size)`, which will be rounded down, on to a starting row number `start` and leaving the last rank to have much more work, I use the next starting row number `nextStart` to minus the current one in order to get a more evenly distributed image size for each rank. The ending row number for current rank will be `end`.

```
int start = rank * nx/size;
int nextStart = (rank+1)*nx/size;
int end = nextStart-1;
```

(Code of defining starting and ending row number)

Attempt1

At first, I classified the ranks into three different groups: the top, bottom and middle strips. On top of the image array they have, the top and bottom will have one extra row at the bottom and top respectively, while the middle strips will have two extra rows on top and bottom, shown in Fig.1.

```
for (int i = 0; i < stripSize; ++i) {
 for (int j = 0; j < ny; ++j) {
   localImg[j+(i+1)*ny] = image[j+(start+i)*ny]; //for rank 0
   localImg[j+ny] = image[j+(start+i)*ny];       //for other ranks
 }
}
```

(Code of each rank getting its own parts of image)

In each rank, a part of the image will be stored into the array with specified size, like shown above, based on the rank number. Rank 0 will have its last row of `localImg` blank to store the neighbouring row, while last rank has to store the parts starting from the second row, leaving the top to store incoming data. Then, I created three

new functions for each cases: `topstencil()`, `bottomstencil()` and `midstencil()`. `topstencil()` computes the last row as middle row, and same for `bottomstencil()` on the first row, while `midstencil()` computes the image as it is the middle of an image. After each rank performing its own stencil function, the master rank will then collect all the processed parts from each rank. It puts the received parts according to their rank, excluding the extra rows attached, into a complete image, then output it.

Attempt2

First of all, `image` and `tmp_image` are initialised to have two extra rows and all elements set to zeroes, using `calloc(sizeof(float), (nx+2)*ny)` instead of `malloc`. In order to make both arrays have their starting indexes begin a row below, I shifted down by using `image += ny`, as shown Fig.2 . The purpose of having two extra rows on the top and bottom is so that all the ranks can be treated the same, as middle grids. Therefore we do not need to have extra if statements to check the position of the rank, and having to deal them separately with different stencil functions.
In this case, no extra variables are holding certain parts of the image in each rank like mentioned in attempt1. Instead, each rank will be processing on the full-sized image using one stencil function. The difference is that the stencil function will be performed on a specific area of the image based on the starting point and ending point given by the rank.

Change in stencil code
In order to accommodate the rank and its responsible parts of the image, changes are made to the stencil function. The way the middle pixels and pixels on the left and right edges are processed the same way as before. However, when it comes to the top and bottom edges, starting row and ending row should be specified by `start*ny` and `end*ny`. Also, since the top and bottom grids are treated like middle ones, I removed the corner cases, and instead, I treat them as part of the left and right edges.

Similar to attempt1, the master rank will then collect the processed images, and because they all have the full sized image, only with different parts convoluted, when they combine together, it will integrate into a full processed image. No for loops will be needed to copy each parts to their original positions like in attempt 1.

3 Halo Exchange

There are several ways that the image can be decomposed into different cells: columns, rows, and tiles. The tiles would require more communication channels to neighbouring ranks. Each rank might have to perform at most 6 message passes, resulting more overheads comparing to other approaches. Also, it will require more complicated code to implement than the other two.
From the serialization optimisation last time, we discovered that when the image array is read in in the ny direction faster. That is why instead of jumping through the array, we optimised by accessing the elements consecutively. For the same reason, I decided to split the image into rows, like the image Fig.3. All ranks will be passing their own edges to neighbouring ranks, where rank 0 and last rank will be only sending to and receiving from one rank.

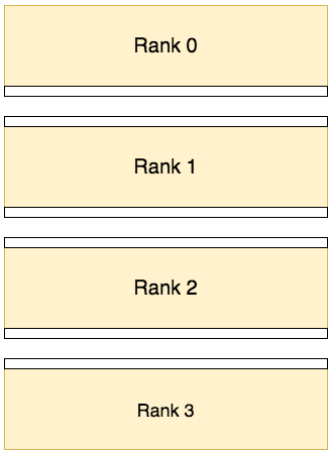

Fig. 1: Structure of Attempt1

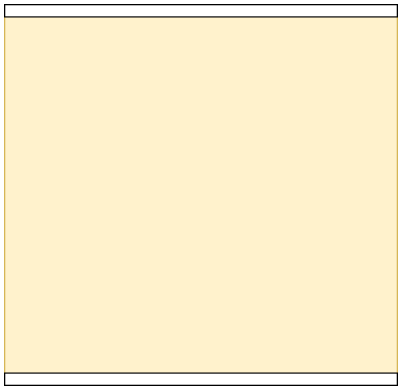

Fig. 2: Structure of Attempt2

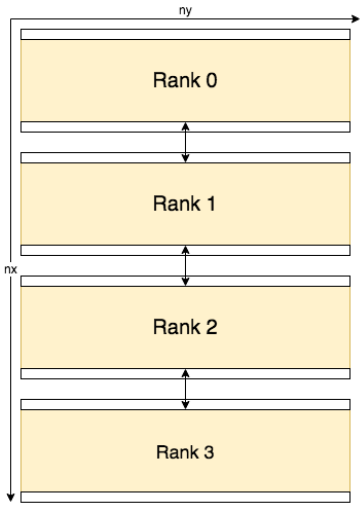

Fig. 3: Halo Exchange for Attempt2

Message Passing Methods

Due to the fact that each rank will only have access to the specified portion of the array, information exchange would be necessary so that the the computation can be perform correctly. A variety of message passing methods are available to choose from, such as `MPI_Send` and `MPI_Ssend`. To implement message passing in a safe and easier way, I avoided using `MPI_Send`, as it is unpredictable about its property of blocking when implementing on different machines. I started with `MPI_Ssend` and `MPI_Recv`. However, this means sychronisation, where the send and receive should match or else a deadlock would occur, and not allowing the program to continue. This was avoided when I instruct the odd ranks to receive first while the even ranks to send first, but if statements will have to be used, which is expensive. Also, the cores might be in idle, waiting for data, where this time can be put to a better use, such as further computation. This method also involves send and receive buffers. This indicates extra time will be sent in order to copy data into the buffer and out to another variable for further process. To copy the data, a lot of for loops had to be written. And for loops can be time consuming resulting to a longer run-time.

For `MPI_Sendrecv`, it needs fewer if statements, because all the data will be sent and received in the same direction, indicating we do not need to worry about whether the rank is odd or even. Its latency is less than two communication. Nevertheless, as there are buffers involved, a few for loops are still needed for data copying.

Trying to acheive shorter run-time, I then picked `MPI_Isend` and `MPI_Isend` for message passing. This is an asychronised way of communication and does not use a buffer, meaning the communication can happen at any time, resulting no occurrence of deadlocks. To ensure that the message is being passed in a correct direction, `MPI_Request` is used to identifies the operation. However, to try to avoid arguments not being safe to use right away, I put the computations that do not require received data first. To prevent any chance of corrupting the message, `MPI_Wait` is used to wait for completion of an operation identified with `MPI_Request` just before the received data is needed for process.

In order for the master rank to gather the processed data, I first used collective communication `MPI_Gather`, but then realised it only takes in data of the same size. This would not work if the height of the image cannot be divided equally by the number of cores available. That is why `MPI_Gatherv` is used, it can deal with different sizes of data. Both works really similarly, but takes in slightly different parameters. It acts like the master rank is performing one send to itself and (size-1) number of receives. They do not use buffers, indicating no extra time is spent on copying data over. `MPI_Gatherv` takes in an array of initial position of where each data set sits in the memory, as well as an array specifying the size of each dataset. All the data would be put into the same variable in their correct position, resulting a fully operated image.

III Comparison of MPI and Optimised Serial Code

The difference between the MPI code and optimised serial code is that parallel processing is implemented in MPI. By taking advantage of multiple cored in one node, each core can operate on similar amount of work at the same time, so it is faster than only one core doing all the work. From the assignment, the fastest run-times I got for 1024, 4096 and 8000 were 0.117s, 2.66s and 9,20s. According to the table given below, we can tell that the run-time has decreased significantly by approximately 90%.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1024	0.0934	0.0515	0.0353	0.0272	0.0226	0.0221	0.0211	0.0186	0.0173	0.0148	0.0145	0.0142	0.0131	0.0138	0.0146	0.0139
4096	2.41	2.03	1.87	1.42	1.19	1.17	1.06	0.987	0.912	0.878	0.831	0.772	0.789	0.793	0.857	0.5104
8000	8.52	6.21	3.94	3.26	2.97	3.31	3.35	3.43	3.36	3.11	2.89	2.44	2.31	2.12	2.09	1.99

Fig. 4: Table of runtime for all 3 images and with all number of cores

IV Scalability

From the line graphs shown above, we can see that overall the time is decreasing with the number of cores used. The speed up when there are fewer cores is more significant comparing towards when there are more number of cores. The difference is less obvious in the end is probably because the increase in overheads from more communication among the cores kind of cancel out the increase in the speedup.

For 1024, the run-time almost halved every time we increase the number of cores, from 1 to 4, then the change is minimal as the number of cores increase from here. The overall speed up from 1 core to 16 cores is around 90%.

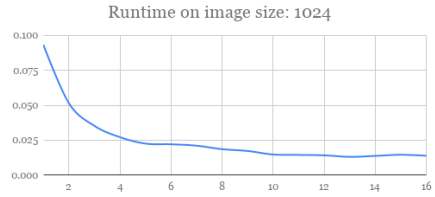

Fig. 5: Runtime for 1024

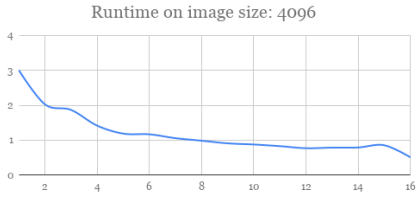

Fig. 6: Runtime for 4096

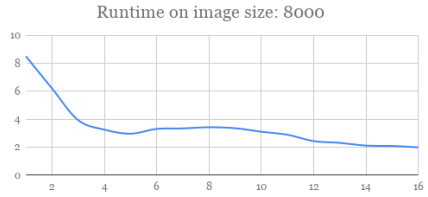

Fig. 7: Runtime for 8000

For 4096, the improvement in performance is not as obvious, but it reduced by around a third when increasing the number of cores by 2 in the beginning, then as the more cores are used, the run-time stayed below 1.0s and reached to less than 0.6s with full 16 cores, with an improvement of around 80%.

As for 8000, the performance also improve obviously from 1 core to 4 cores. It eventually got to an average of 2 seconds with full number of cores, with a speed up of 75%.

So looking at the chart, it can be deduced that by increasing the number of cores even more would not really contribute to any speed up in the time, however, I think it might perform worse due to larger amount of message passing. The The final run-times using all 16 cores on the 3 images, has each reached to an average of 0.0139s, 0.510s and 1.99s.

V Conclusion

By taking advantage of multiple cores, the performance was significantly improved from the charts and runtime shown below. But the techniques used for implementing the parallelism and message passing should be carefully designed, so that the cores can reach its optimal performance, and not wasting too much time due to overheads. The run-time for all ranks can vary when using mpicc compiler, as the Fig.8 shown, the minimum speed can be 0.008 seconds, while the maximum can reach 0.016s.

Looking at the roofline analysis, it can be deduced that the it is memory bandwidth bound. The GFLOPS/s for 1024, 4096 and 800 are 145, 60 and 57.6. This illustrates the fact that when the code is operated on bigger image, the performance gets worse. To improve that, GFLOP/s should be increased. This can be done by even making the run-time even shorter. Ways of doing that can be implementing processing using multiple threads, which maybe allow even faster processing since the memory are shared within a core. Or maybe another cell division Cartesian grids can be used instead along with the message passing MPI_Alltoall.

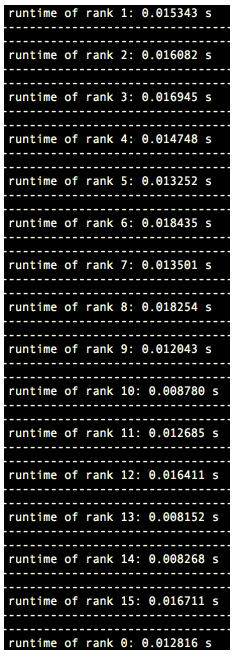

Fig. 8: Runtime for 1024

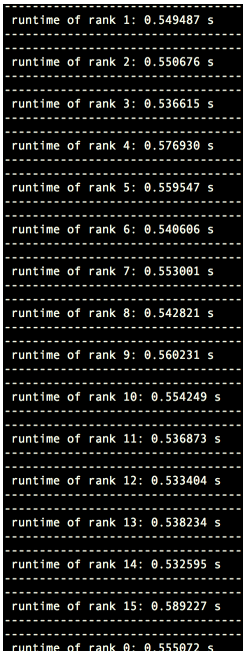

Fig. 9: Runtime for 4096

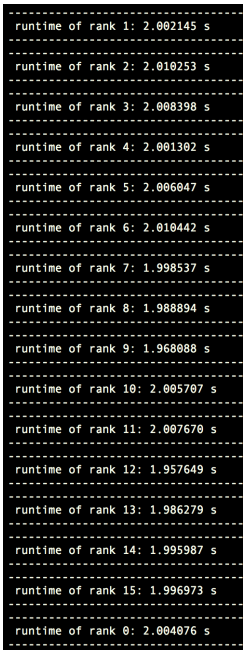

Fig. 10: Runtime for 8000

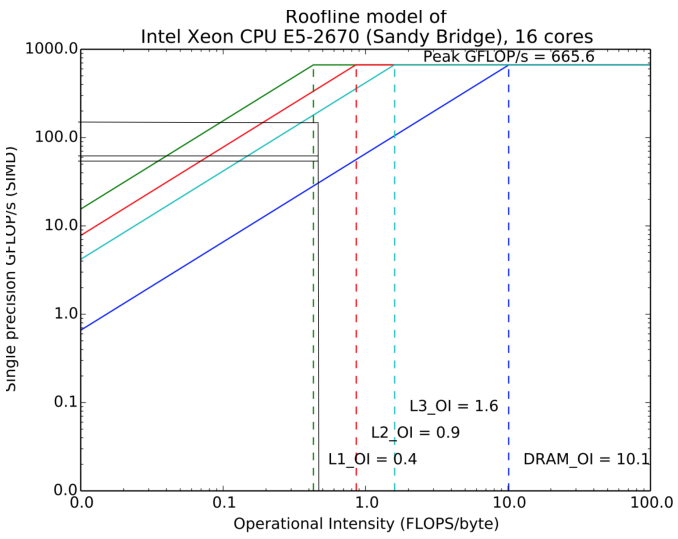

Fig. 11: Roofline model