MPI Report

Jack Zhao

ID: fz17963

Introduction

MPI is used to parallelise and improve the runtime of the 5-point stencil code in this report. I used perf to check the cache.

Before starting the further optimisation and implement the parallelism of the serial code, I need to record the running time of current code as a reference, which is shown as follow:

Size of image	Average Run time(s)
1024*1024	0.09
4096*4096	2.86
8000*8000	10.70

Table 1. Initial run time with different size.

To apply MPI_Neighbor_alltoall function in the topology graph, Cartesian communicators is required to use, for the reason of using MPI_Neighbor_alltoall, I will discuss this later in this report.

Cartesian communicators

Every process of my structure of the computation is not necessary to communicate with every other process. I used Cartesian 2D grid to make processes act as if they are 2D orders and then communicate with their neighbours. Each point in Cartesian communicator has a coordinate and a rank, which are queried with MPI_Cart_coords and MPI_Cart_rank respectively. Which means I can reorder the rank of the processes. Before building the communicators, I reset all the input image, then I set up cartesian sub topologies from CART_COMM_WORLD. Now I have the problem dimensions of the local image.

Decomposition

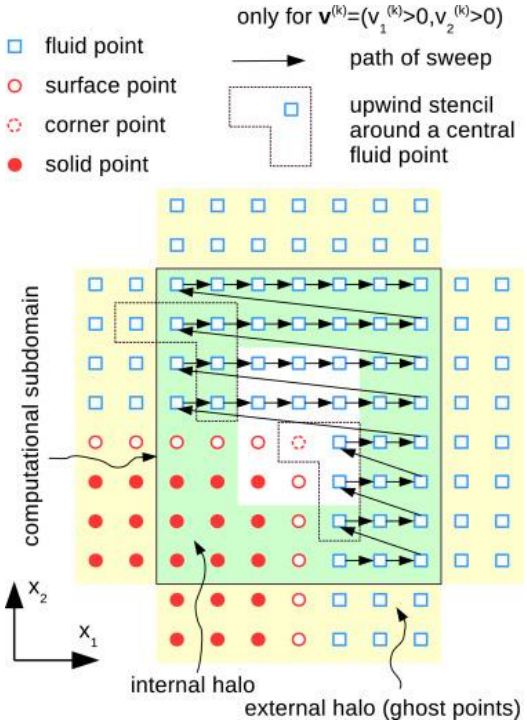

Figure 1. Structure of the halo space

Figure 1 shows the structure of the halo space, apparently, we can then go through the halo space based on this graph.

To implement the halo exchange among neighbouring cores in the image, which involved fewer sending message or data when communicating by using a master, after building the Cartesian communicators, I already have the positions of each pixel in the image, the size of halos, and obtain the layout of the local image by using functions MPI_Dims_create and MPI_Cart_create. Then I did scatter to take data from the image to distribute it amongst other sub-image, and after a series of processing, after a few times of experiment, I decided to use MPI_Scatterv, which allows me to send different amounts of data in any order to each process, which avoids manipulating the same amount of data distributed all the time. For the same reason, after a series of operations to collect all the information I need from sub-images, instead of using MPI_Sendv and MPI_Recv, memcpy function is implemented to copy the information from the local image to the buffer, then applying MPI_Neighbor_alltoall is to make different data items can be sent to

each neighbour, for instance, the k-th block in send buffer is sent to the k-th neighboring process and the l-th block in the receive buffer is received from the l-th buffer. Then it copies the information from receiver buffer to the local image by doing this again. During this process, I realized that not all the information is required due to the mismatched size of the local image. Which is caused by the padding edge of the image, which is different from the one I set in the beginning of the program, which is shown as Figure 2, the blue rows and green columns shows the new padding edge exactly, after manipulating the buffer and local image, I used MPI_Gatherv to assemble all the data to one process. To make sure that the output image is the same as the initial one, I print out the image at the beginning and after gathering. Thereafter, I used MPI_Neighbor_alltoall to exchange the halos.

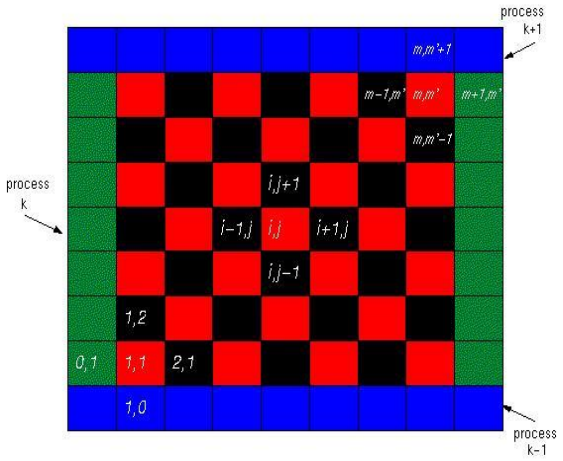

Figure 2. Example of local image after scattering

Besides this, I tried using row decomposition as well, in fact, this method is slower than the previous one. On the one hand, I did more loops in this method to fill up a buffer by going through each row, I only can send the buffer until all the loops are finished, on the other hand, in this case, each process will send the bottom row below and the top row above(receive a row from above and below), there is no sending halos which are in contiguous memory addresses, this one kept at an average of 4.5s run time. However, using row decomposition for the 1024 image is not slower, then I went back to the parameters of the MPI_Neighbor_alltoall function, I found that I had to fill the buffer with all of the halos, this indicates that I always need to wait to send and receive data, the data couldn't be sent until the buffer is ready. The other reason which make this happen I think of is the row decomposition in my program can be relatively easily finished in the stencil function without extra calling, which means that the row-accessing and processing is already finished while there is still lots of room to improve in the tile decomposition. The result of all size image of row decomposition is shown as Figure 3.

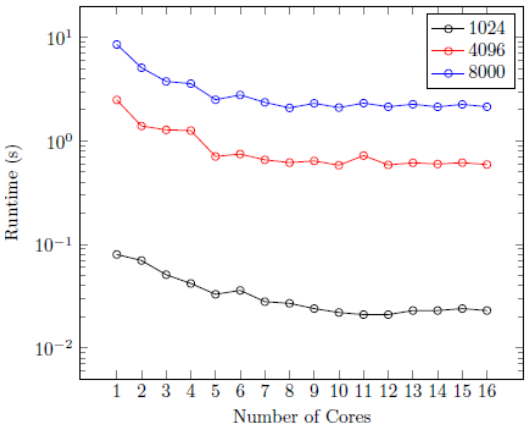

Figure 3. Run time of row decomposition with different cores

Sending Image

As mentioned before, I tried different ways to send the image data on the column decomposition implementation, to find whether there is a way to speed it up, firstly, I implemented MPI_Send and MPI_Recv for 4 ranks data transmission. But when I tried sending data to the right, it caused deadlock due to the inappropriate order. Then I changed from MPI_Send to MPI_Ssend, it worked but it was slow, because MPI_Send uses a buffer to put the message in while MPI_Ssend uses blocking synchronous send, which means nothing returns until the receiver is available and receives the message from the process, it does wait for a while,

which is 2 times slower approximately. Until now, there is no one of these methods which is faster than using memcpy to simply copy the information between the buffer and the local image before sending it, and by using MPI_Neighor_alltoall to process. The size of data/message sent also gives different results of run time.

Run Time Analysis

Size of image	Average Run time(s)
1024*1024	0.017
4096*4096	0.16
8000*8000	1.02

Table 2. Run time of MPI with MPI_neighbor_alltoall.

The running is based on the previous row decomposition with 2 nodes and 28 tasks per node instead of using MPI_Neighbor_alltoall based on Cartesian communicators, because the program itself doesn't work properly.

Not surprisingly, based on the data from Table 3. It shows that for all size of images, row-major accessing and processing performs worse, which is 0.045s, 0.279s, 1.514 corresponding to 1k, 4k, 8k image approximately.

Size of image	Average Run time(s)
1024*1024	0.045
4096*4096	0.279
8000*8000	1.514

Table 3. Run time of MPI with row-decomposition.

Therefore, I can safety decide to use the first technique instead of using row accessing.

To check the performance of the code, using GFLOP for the different image sizes for the MPI, the memory bandwidth is 100.4GB/s for 8000*8000 image, by using operational intensity 0.375 FLOPS/byte, the detail is shown on table 4.

Size of image	MPI
1024*1024	98.7
4096*4096	167.7
8000*8000	100.4

Table 4. GFLOP/S for performance

To further obtain the relationship between the run time and the number of cores used, I did many attempts by changing the stencil.job file. The result is shown as follow:

Running time			
Cores	1024*1024	4096*4096	8000*8000
1	0.065	1.523	3.6251
4	0.035	0.871	1.924
8	0.026	0.303	1.651
12	0.021	0.191	1.551
16	0.015	0.180	1.031

Table 5. Running time with different cores

Summary and Conclusion

Size of image	Serial Run time(s)	MPI Run time(s)
1024*1024	0.09	0.017
4096*4096	2.86	0.16
8000*8000	10.70	1.02

Table 6. Final average run time

Table 6. shows the comparison of run time result between serial and MPI optimisation, it's clearly show that MPI optimisation improved a lot on program performance computing, and there are a few things I haven't tried in this report, implementation of OpenMP is not used, and some other MPI functions. And more decomposition strategies can be adopted for this assignment.

In conclusion, the performance achieved in the MPI library with reasonable number of cores makes the program much faster than only one core used. But there are also some limitations over here, such as the latency and delays among different processing communications. The performance could be better if have fewer latency.