Distributed Memory Parallelism With MPI

Asel Kitulagoda

ak17520

1 Introduction

The aim of this assignment is to parallelize the optimised serial code for stencil from the previous assignment with MPI for the three respective image sizes: 1024x1024,4096x4096 and 8000x8000.I first looked at compiler flags and then performed code optimisations on my implementation of stencil with MPI.All timings and speed-up values provided below were calculated as an average of three values for each input image specified.

2 Compiler Flags

I started by looking at compiler flags for optimizing run time for the stencil code with MPI. I made use of `mpiicc` since it provided Icc compiler functionality with a MPI wrapper which produced the fastest run times for Serial Optimisation assignment. I used the flags `-O3 -xHost` since we are unable to use `-fast` with mpiicc, I found out that these flags have the same effect as `-fast` and is a safer implementation.This produced timings that were very similar to my optimized serial code when running on 1 core.

3 Domain Decomposition

Note all timings in this section was made using all the cores across 2 compute nodes in BCP4(56 cores).
Key Notation: for example an image input size of 1024x1024 is represented by 1024

3.1 Splitting and Gathering the Image

I needed to split the input image across multiple cores to parallelize the the serialised stencil code while taking a Single Program Multiple Data approach. I decided to first decompose the image column-wise across all of the ranks within two node.This is because It does not make a difference to the run time whether I decompose the image row-wise or column-wise If I used contiguous access with both.This was because the input image was initialized in column major order so I decided to access the array contiguously since it makes use of pre-loaded data already in cache to avoid striding so you can achieve a higher rate or cache hits otherwise main memory is accessed more regularly.

I initially used for-loops to iterate through the input image and divide it among each of the ranks. If the overall number of ranks did not divide the image equally, I added the remainder of the image to the final rank and then gathered the local grids for each rank into a full grid inside the master rank.This was quite slow since I observed overall runtime for 0.37 seconds on average for a input size of 4096 so I decided to experiment with other methods. I tested splitting the image with `Memcpy`.This is because `Memcpy` is well implemented to leverage intrinsics so it is unlikely that `Memcpy` will ever be worse than a for-loop implementation. In the worse case `Memcpy` is not inlined and will only perform additional sanity checks which amounts to a small number of additional instructions when compared to a for-loop.This provided an overall speed of 1.85X when compared with the run-time of splitting the image using For-loops for an input image of 4096.

I made use of `MPI_Scatterv()` and `MPI_Gatherv()` to split and gather the image input.It creates the least load imbalance among processes since you can use it to send unequal numbers of elements to each rank.It allows all processes to use the same code.However after timing how long Scatter/Gather takes compared to Memcpy(), I found that Scatter and Gather takes up to 10X as much time to split and Gather the image which is even longer than timing the iterations of stencil.This could be because the root rank has to push a fixed amount of data to other ranks where the limiting factor is the memory bandwidth if all ranks are within the same compute node or the network bandwidth if multiple nodes are used.More setup time is also required for each rank that is scattered and gathered so the run time increases linearly with the number of ranks.

Scatter avoids the image from being initialised within every rank and It does not allow access to the input image for every rank.This produced an increase in speed by 1.25X for a version of stencil using Scatter and Gather for an input image of 1024 and 4096 when compared to a version using `Memcpy()`.

There is no significant increase in speed between Scatter and Gather since the results show that there is tolerance of +/- 0.08 seconds for results obtained when calculating the average for Scatter/Gather using an image size of 1024 and a tolerance of +/- 0.10 seconds for an image size of 4096.This means that random error has a large effect on the speed up value calculated so there is not much difference between the `Memcpy()` implementation and the Scatter and Gather approach.I chose to not to include Scatter and Gather within my final implementation.

3.2 Halo Regions and Stencil

Since I split my image column-wise, I made use of synchronous communication using `MPI_Send` and `MPI_Recv` to send the halo regions across to neighbouring ranks so computation can take place.Through this I am not relying on system buffering within my implementation which could make my program less robust and scalable since I make use of the Rendezvous Message Passing Protocol.

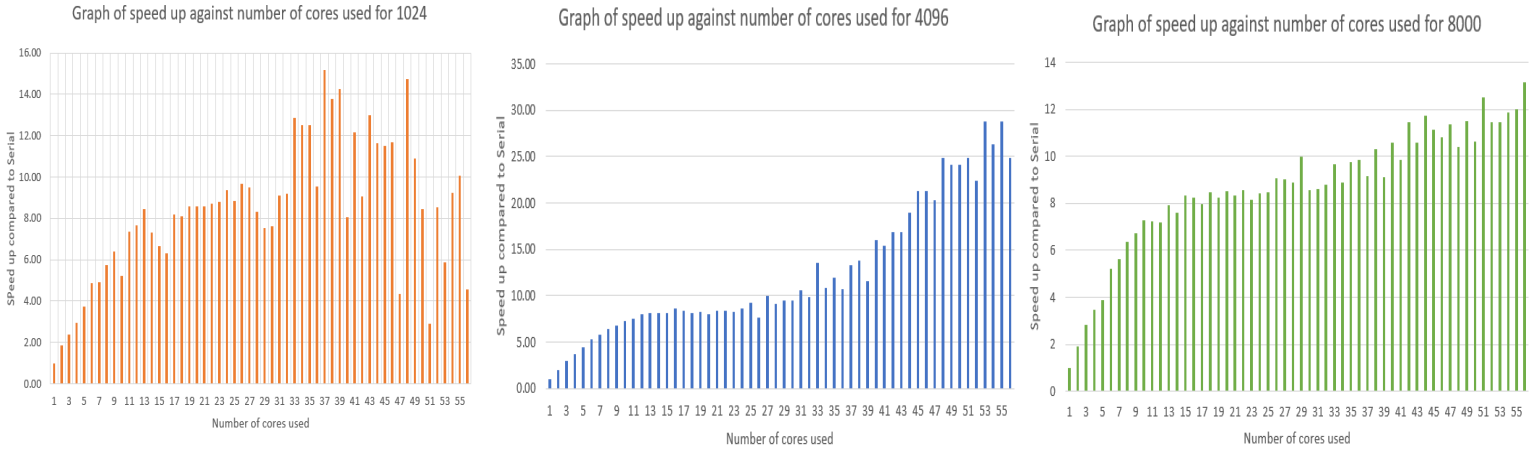
Initially, I included receive buffers within my Halo exchanges but after removing them I noticed a 1.4X speed up with all for run-times for image sizes of 8000 and 4096 so I omitted them by adding a pointer directly to my local image where either data is being received. However, there was no significant change in run-time after removing the send buffers within the implementation which is due to both implementations being accessed contiguously which takes up the bulk of the time within the timed region so the effect of including a send buffer with `Memcpy()` only manifests itself as noise.Therefore,I kept the send buffer within my latest implementation.I also made sure the master rank does not perform a halo exchange with the final rank.

I kept the implementation of Stencil almost exactly the same as the optimized serialized version.Since I found out Stencil is a memory bound problem in the previous assignment, the local 1D array which stores a section of my input image for each rank to perform work on which is accessed contiguously which resulted in fast memory access within the nested for loops inside the Stencil function.

I also experimented with asynchronous communication.This is because it reduces synchronization delays where the send process is waiting for acknowledgement from the receive process that it is okay to send a message such as with the halo exchange using `MPI_Send` and `MPI_Recv` within the iterations loop.By doing this, I had to capitalise on the time I gain from having a non-blocking `MPI_ISend` and `MPI_IRecv` by performing computation on the the inner part of the local grid without the halo regions and then call the stencil function on just the regions unaffected by the halo exchange and then call stencil on the halo regions separately.This allowed me to overlap communication with computation within my code.

However, Asynchronous communication did not improve my timings when using all cores in two compute nodes for any of the input image.This is because any background communication such as sending across the halo regions will happen in an additional thread but the main computations within stencil will use up all the available resources. This suggests that there will be a race condition by using asynchronous communication when using all the cores across two compute nodes so I decided to opt for synchronous communication with `MPI_Recv` and `MPI_Send`.

4 Scalability of Stencil



(a) for 1024 x 1024 image (b) for 4096x 4096 image (c) for 8000x8000 image

Figure 1: Graph for speed up compared to optimized serial code for stencil against all numbers of cores across 2 compute nodes for all input images using -xHost -O3(all results taken as an average of 3.There was a tolerance of +/- 0.07 for image sizes of 4096,8000 and +/- 0.008 for an image size of 1024

To analyse the scalability of my code, I made a graph of performance against the number of cores used. Performance in this case is the speed-up in run time compared to the serial optimized code which is the first value shown in each of the graphs for one core. Although there are some outliers for the 1024x1024 image, it shows a sub-linear plateau.This could be due to parallel slowdown where parallelization of a parallel algorithm such as stencil beyond a certain point causes the program to run slower.Since it is clearly faster for values of cores between 33-45 according to Figure 1(a) but slows down as it gets closer to 56 cores.This is due to the communications bottleneck within the stencil code because as more cores are added , each core spends more time doing communication than useful computation.By around the value of core 50 in Figure 1(a), the communications overhead which is created by adding more cores overtakes the increased processing power that each new core provides so parallel slowdown comes into effect.Most of this overhead comes as a disadvantage of the distributed memory organization since there is a higher communication overhead due to message passing via networks to split work across more than one compute node.

However for larger image sizes such as 4096,there is an almost linear scaling for the speed-up between 19-43 cores.This suggests that it is very close to the ideal run time of the parallelized tasks if no overhead of any kind exists.This is because the processing power gained by every new core takes precedence over the increase in parallelization overhead due to communication overhead or idling overhead from imbalances of work between cores. However, towards the top of the Figure 1 (b) we can see that the performance is slowly reducing which could suggest that even the 4096 image plateaus towards the last cores 55-56 so if we tested across more compute nodes we might observe the same effect as with 1024.

For the 8000 image, I observed some linear scaling from cores 1-11 and then on there is a weak very linear scaling which follows the same trend as 4096 due to the processing power gained by each new core.But unlike 4096, I cannot observe the curve beginning to plateau which suggests that the program might need more cores until it begins to show signs of parallel slowdown since the amount of computations available is so big compared to a smaller image like 1024.This suggests that the run-times will continue to decrease up until a point If i keep on adding more processing elements to this image.

5 MPI Performance vs Optimized Serial Code

Table 1: shows final run-times for the stencil iterations loop for all inputs images using `icc -O3 -xHost` and `mpicc -O3 -xHost` for all cores within 2 compute nodes

Runtime(in seconds)	**1024**	**4096**	**8000**
MPI Optimized Code	0.022	0.12	0.84
Optimized Serial Code	0.10	2.93	11.25

Table 1 suggests that there is a 4.5X,24.3 and 13.3X increase in speed between the MPI_Optimized code and the Optimized serial code for each input image:1024,4096 and 8000.The greatest increase in speed is for the larger images since they are computationally heavier and require more memory accesses than the smaller images such as 1024 so it makes full use of the distributed memory system to access and transfer data between different processing elements to better parallelize the computation within the image.Whereas with 1024, the parallelization overhead takes precedence when compared to the computational power gained with each added processing element.

6 Improvements to Stencil

6.1 Convert to Cartesian Grid

Cartesian can be used to turn the 1D rank indexing in MPI_COMM_WORLD into an nD grid of processes.This allows me to pick a more optimal processes to subdomain mapping which is powerful since I have more control over how data is laid out.You can also avoid packing and unpacking Halo buffers using MPI derived types and halo exchange can be easily performed with one function `MPI_Neighbout_alltoallw` which could significantly improve the current timings where communication to complete halo exchange takes a significant fraction of the time.

6.2 OpenMP

OpenMP makes use of shared memory systems and could be ideal for Stencil since shared memory gives less overhead compared to distributed systems since multiple cores can share the same address space and still maintain a high bandwidth.However with OpenMP alone,we are limited to the maximum number of cores in one compute node so

is not as scalable as MPI.However, a combined approach of OpenMPI and Open MP might be better since you can still communicate between a distributed network of nodes and take advantage of the high bandwidth between cores.

7 Performance Analysis

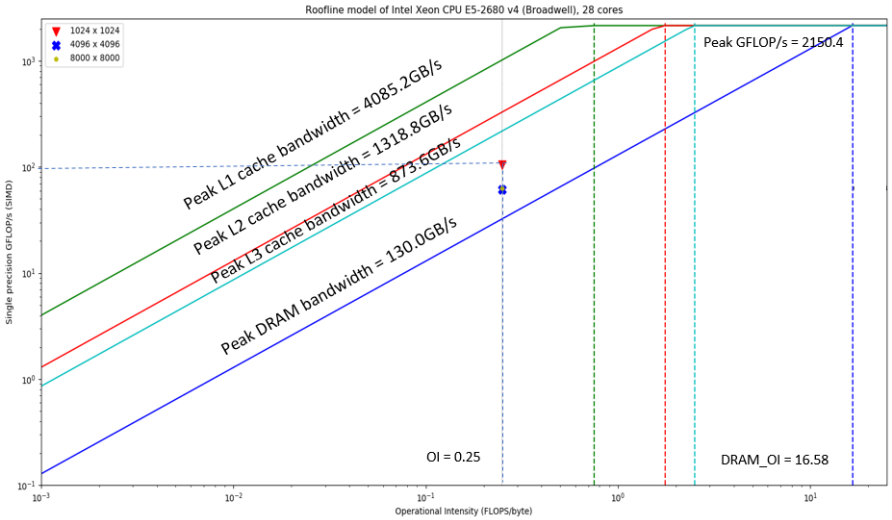


Figure 2: Roofline model of Intel Xeon CPU E5-2680 v4(Broadwell) 28 Cores with values of Operational Intensity in FLOPS/Byte and Performance in GFLOPS/s with points showing performance and OI for each input image size using MPI.

I calculated the roofline model for 28 cores since I wanted to compare the performance of all the cores within one compute node against my serial implementation for Stencil. Since there was no roof line model available for 28 cores, I made use of the roof-line provided for 14-cores and the roofline-provided for 1-core to produce Figure 2. Since the values of Peak cache bandwidth scale with each core, I multiplied the value of peak bandwidth for one core by 28 to get the values shown in Figure 2. The DRAM bandwidth and the peak performance scale with the number of sockets so I decided to multiply the value of DRAM bandwidth and peak performance from the 14-core roofline model by 2 since there are 14 cores in one socket. The operational intensity once again stays the same as the original optimized serial code since there are only 6 FLOPS inside the nested stencil loop where each memory load is 4 bytes and I calculated the operational intensity as $\frac{FLOPS}{bytesloaded}$. I then proceeded to calculate the values of performance for each of the input of the input sizes respectively by using $Performance(GFLOPS/s) = \frac{nx \times ny \times 100 \times 2 \times 6}{runtime}$.I make use of constants 100,2 and 6 which corrrespond to the number of iterations stencil is run, the number of times stencil is run within the iterations loop and the number of FLOPS. The run-time here is for the respective input image where nx and ny is the image height and image width respectively.

For the final MPI implementation for Stencil, I calculated values of performance of 104.86 GFLOPS/s, 61.00 GFLOPS/s and 61.94 GFLOPS/s for each respective input image size. For the optimized serial implementation of stencil, I calculated performance values of 12.6 GFLOPS/s,6.9 GFLOPS/s for 4096 and 6.8 GFLOPS/s for each respective input image.However, we cannot compare performance against the serialized version of stencil since I am using 28 cores with the parallelized implementation and one core with the serialized version of Stencil. Since the program is memory bound, I can compare the peak streaming bandwidth of the CPU with 28 cores which against the memory bandwidth of the one node within Intel Broad well CPU. One socket within Intel Broadwell has a max memory bandwidth of 76.8 GB/s so since Memory bandwidth scales with the number of sockets , there is a peak memory bandwidth of 153.6 GB/s.

8 Conclusion

I managed to achieve the ball park timings provided as the benchmark for this assignment with the final timings for each input image size being 0.022s for 1024,0.12s for 4096 and 0.84s for 8000.However, there is still room of improvement in regards to making my code more computationally efficient and experimenting with other methods of decomposition to achieve even faster run times.