Introduction To High Performance Computing
MPI

Adam Sadiq - as16165

1 Introduction

In this paper, I will describe and analyse the methods and techniques utilised to improve the efficiency, and therefore runtime of the stencil.c code using message passing with MPI and shared memory with OpenMP. The advantage of using MPI is that with more processors, we have less idle time for potential processes to be running. Therefore, we can argue that the greater the processes, the lower the idle time and thus the faster the computation. Yet, the bottleneck with higher processes is communication between processes.

2 Structure

2.1 Row Decomposition

To increase computing performance, I decided to split up my image between a number of processes. However, how the stencil code is run, it needs to access memory that belongs to another process. To handle this, I used halo's regions to send the bordering pixels to the surrounding processes that required it. Therefore, to ensure the message passing was efficient, I split image with a row decomposition. Using a 1 Dimensional array, I was able to follow a contiguous access pattern and reduce the risk of cache thrashing, whereas with a column composition, after every memory-step we will have to do a halo exchange, as the whole column will not sit in the cache. Yet, it is also notable to mention that depending on the size of the image, a column decomposition may be preferred. Deciding between the two is an empirical problem as with an image with a much larger width than height, the cost of communications may outweigh the cost of cache thrashing: producing a large overhead when decomposing by rows.

2.2 Send & Receive

To distribute the image to the different processors I used the MPI Scatter Method. This worked by dividing the pixel values evenly across the number of processes I had. To utilise this I allocated memory to hold the address of a send buffer and used a fixed size for the number of elements to be sent to each process, with a receiving buffer. Through this implementation, I was able to almost trivially receive the image inside the stencil code. After computation using send and receives, I used MPI Gather to collect the image together, similar to Scatter. However, because I wanted to analyse speed up, rather than using 16 ranks, I used MPI Scatterv and Gatherv to be able to allocate a variable amount of data towards each rank. In the case where the #rows % #ranks != 0, I would allocate the remainder for the last rank. To send the halo regions across the image, I used the MPI_Send and MPI_Recv function. Because these calls are blocking, I had to ensure the functions were synchronised upon call. This led to my approach to make my even processes Send then Receive, and the odd processes Receive then Send. I used Send initially over Ssend because of it's latent potential to be asynchronous. Because of this it would be able to speed up runtimes with minimal hang times. Yet as Send uses a system buffer, it can carry overheads where Ssend would not. As Ssend was also a potentially safer option in terms of understanding the computational state of the program, i opted for Ssend over Send.

3 Performance Analysis

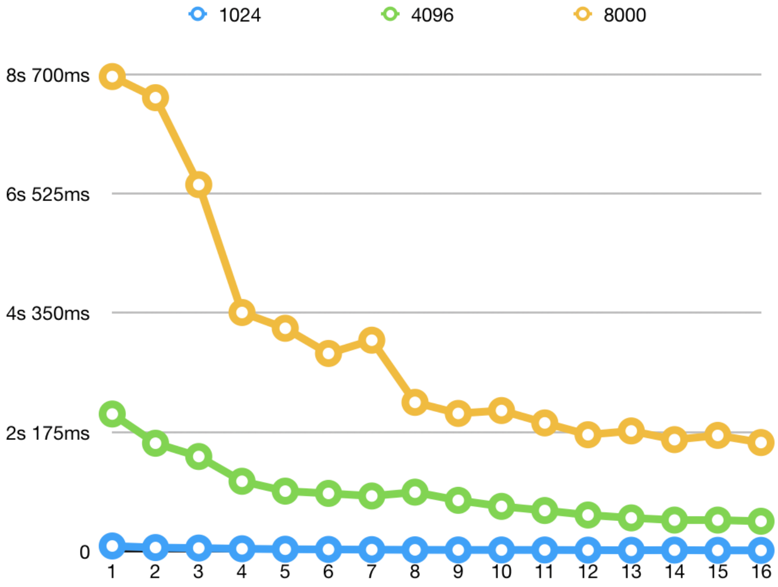

Figure 1: Graph showing the difference in runtimes per core

3.1 Bandwidth vs Latency

Observing the graph, we see that a possible sublinear declining occurring as cores increase. From 10 processes we see that for the 8000x8000 image, the time fluctuates slightly. This is most likely due to overheads in communication due to the number of processes communicating with each other. To calculate bandwidth, I used the number of calculations per pixel in the image, alongside the number of calls to stencil, dividing by the bottleneck runtime I got a bandwidth of 58GB's. Measuring this against STREAM as a simple memory bandwidth benchmark - I achieved a 88% of the peak DRAM bandwidth - yet also showing that my program is memory bounded by the L3 cache. The majority of the program receives a cache hit deep within the L3 cache.

3.2 GFlops

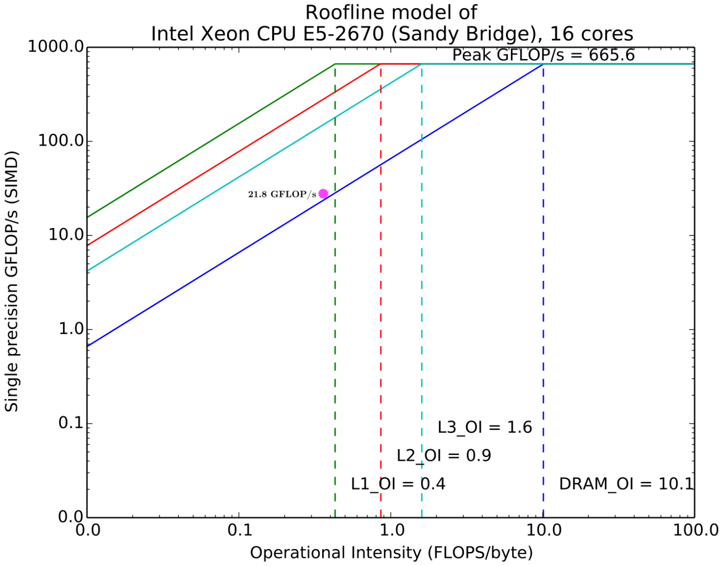

Figure 2: GLOP\s for stencil

After calculating my bandwidth, I wanted to calculate my attainable GLOP's. My first step was to calculate operational intensity. Defined as: operations per byte of memory movement - my program was $OIfp32 = 9/((5r + 1w) * 4) = 0.375$. To get my roofline value, I evaluated *Bandwidth * Operational Intensity* equalling *21.8GFLOP's.* [1] Now observing the graph we can see that my parallel stencil code is clearly memory bandwidth bound as well as network bound.

3.3 OpenMP

Implementing OpenMP fairly straightforward, showing its benefit over MPI. It was implemented by utilising a *#pragma omp parallel for*, enabling shared memory for the nested for loop in the stencil code. However it was significantly slower than MPI, perhaps with more optimisations it could even out.

MPI	OpenMP
19ms	482ms

4 Conclusion

In conclusion, I have learned how to create optimised programs, not only through algorithmic complexity, but also utilising shared memory & message passing, understanding how to distribute processes effectively, through Memory Hierarchical Techniques, avoiding Disk Thrashing, following convention, yet most of all, empirical experimentation!

5 Sources used

[1] McIntosh-Smith, S. (2018). Performance Analysis.