Intro to HPC Assignment 2: Parallel Optimisation

Kheeran Naidu (kn16063)

***Abstract*—Optimisation is a problem that is faced in all aspects of life. In terms of computing, we try to minimise the use of resources (particularly time and memory) without corrupting the output. For many years a significant optimisation was achieved with serial optimisation as Moore's Law predicted a doubling in processing speed every year, this has however since been decaying [4]. Therefore, we introduce parallel optimisation which allows us to simultaneously take advantage of the processing speed of multiple processors at once.**

I. Serial Optimisation

Using the basic optimisations from the previous coursework - including restricting the images in the function stencil, changing double to floating point precision, removing division operations [1] and using Intel's C compiler with the Ofast and xAVX flags [2] - I also removed the if statements from the stencil function by padding the images with 0s to allow for optimal scalability when applying parallel optimisations. (Discussed later)

II. Parallel Optimisation

Parallel optimisation is the usage of multiple processors to speed up the execution of a programme (or part thereof). For security reasons and performance slow downs [5], the processors do not share the same memory and so we use MPI (message passing interface) in order to communicate between cores.

My approach to parallel optimisation, in regards to stencil, was to split up the work into N processors, where N=1,2,...,16. This was done by taking the $nx \times ny$ image and partitioning its nx dimension into N portions. Since nx isn't always divisible by N, I determined the partition of the first N-1 processors, $partX = \lceil \frac{nx}{N} \rceil$, and found the partition of the last, $partX\_end = nx \ mod \ partX$. This caters for images of all sizes such $N \leq nx$.

To reduce the amount of space allocated in each processor, I altered my stencil function to still take in the full image and tmp_image, but to only work on the portion determined by which rank it was in[1]. Using MPI, after each function call of stencil the necessary halos [6] on each processor are updated. This is done because stencil required access to data that is being updated by another processor.

Figure 1 shows how I implemented MPI in the case of 3 cores. To prevent deadlocking [7], I performed the communication by sending halos to the next processor and receiving from the previous processor, followed by sending to the previous ad receiving from the next. This was done to provide a good flow for the communication whereby the wait time of any request was minimal.

[1]refer to stencil.c in file store from the safe submission.

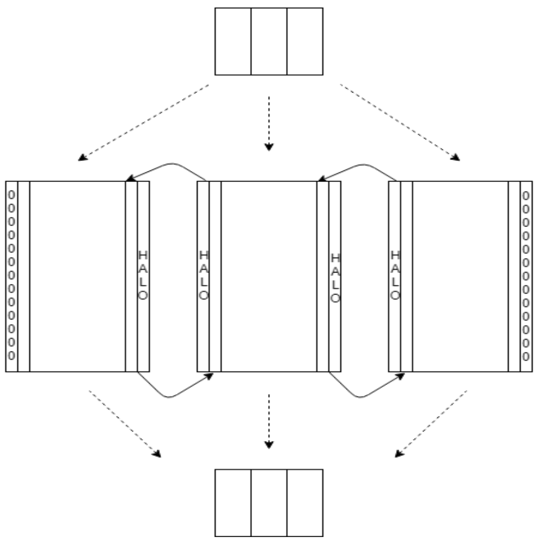

Fig. 1. Implementation of MPI where N=3.

In the final step of the parallel optimisation, I sent only the portion that was worked on by each processor (not inclusive of the halos) back to the MASTER processor (rank = 0) directly into their corresponding places in the full image, which was then outputted by the standard function. This final process could've been optimised using MPI I/O whereby each processor writes directly to the output file to reduce the amount of data sent between processors [8], however this was not necessary for the scope of the assignment as this part wasn't timed.

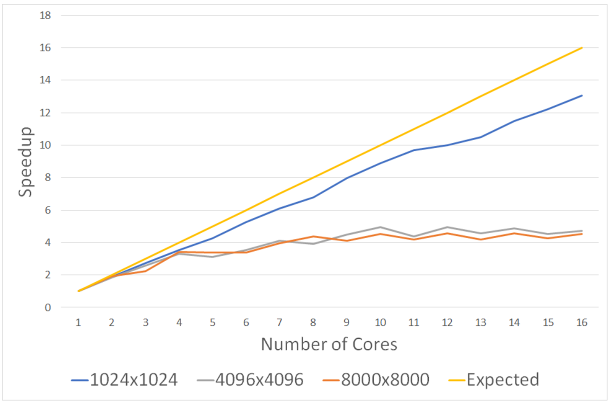

Fig. 2. Figure comparing speedup with extra cores w.r.t. 1 core for images of 3 different sizes.

III. Parallel vs Serial

From Figure 2 we see that increasing the number of processors greatly increases the performance. This increase is linear with the smallest image 1024×1024. With the other 2 larger images, 4096×4096 and 8000×8000, we see that the speed up loses its linearity at N=4 cores and plateau after N=7 cores at about a 4x speedup. Parallel will always faster or as fast as serial since each processor in parallel will apply the serial optimisations.

$$Speedup = \frac{Runtime\ on\ 1\ core}{Runtime\ on\ N\ cores}$$

IV. Optimisation Analysis

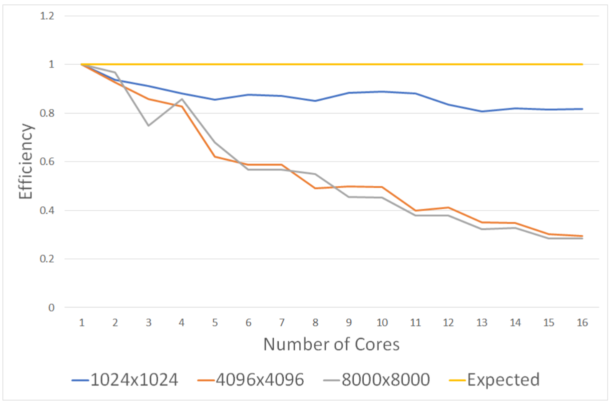

Fig. 3. Efficiency of parallel optimisation on N cores

I calculated the efficiency using Eq 1. The efficiency gives a comparison of how well the results are compared to the expected. By Amdhal's Law [3] and since we are only timing the parallel optimisation part of the code, the speed up is expected to increase by the same amount as the number of cores w.r.t. 1 core. In the case of the smallest image, 1024×1024, the efficiency is above 80%. This is very good and shows that it has been very well optimised, serially and parallel-wise. We can never get 100% efficiency because there will always be hardware limitations which prevent the ideal speed up. In this case the speed up is bounded mainly by the time taken for MPI. This is because sending and receiving messages from other processors isn't an instantaneous occurrence. We can conclude this because the decrease in efficiency is linear with a constant gradient.

$$Expected\ speedup = \frac{N}{1 + P(N-1)}$$

where P = The proportion of serial optimised only code. [3]

In the case of the larger images, 4096×4096 and 8000×8000, we see that the decreases in efficiency is also linear, but which a much steeper gradient. This is because the size of the image arrays are much larger and so the size of the arrays being transferred, which is the size of a row, between processors after each function call is significantly larger. This brings the efficiency down to 28% for the largest image, 8000×8000, with 16 cores.

$$Efficiency = \frac{Actual\ speedup}{Expected\ speedup} \quad (1)$$

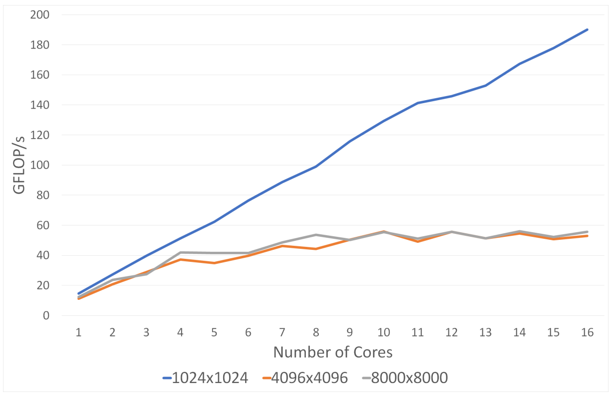

Fig. 4. Floating point operations per second for each N with 3 images.

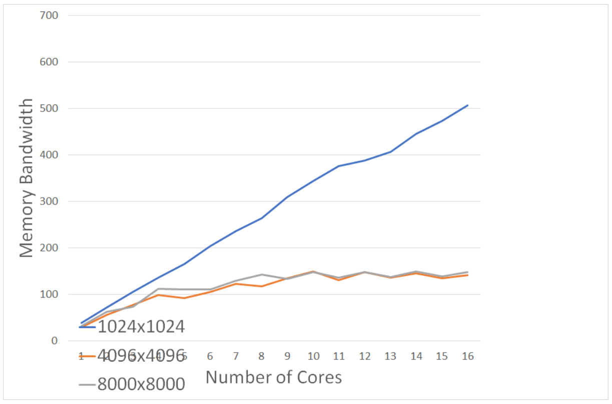

Fig. 5. Memory Bandwidth per second for each N with 3 images.

Next, I calculated the number of floating point operations per second (FLOP/s), seen in Figure 4, the memory bandwidth (bytes/s) 5, and compared them with the roof line model given. We first see that the smallest image is stored in L3 cache which has capacity 20MB. The smallest images has size 1MB but wont fit in the L2 cache of size 256kb. We also see that the largest image is of size 64MB and so exceeds the capacity of L3 cache and so is in the DRAM. This explains why the memory bandwidth is significantly lower from the smallest to the largest. Even though the mid image, 4096×4096, should be in L3 cache, it seems to be located in the DRAM. We expect that as the number of processors increases, and the size of the portions stored on each cache to decrease, that the array in each processor will eventually be small enough to be in the L1 cache (L2, L3 and DRAM are unified), which would significantly increase the memory bandwidth. This is the case in that of the smallest image however the other 2 aren't split enough on just 16 cores.

$$OP\ intensity = \frac{GFLOP/s}{Memory\ Bandwidth(GB/s)} \quad (2)$$

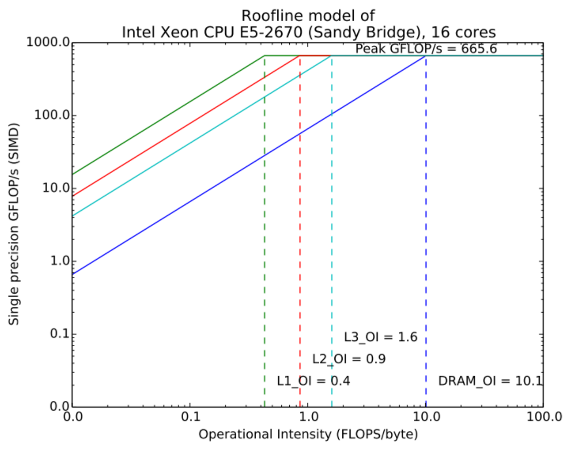

Fig. 6. Given Roofline model on 16 cores

I then calculated the operational (OP) intensity to compare with the roof line model given 6. For all 3 images the op intensity is 3. This shows that the programme is memory bandwidth bound.

V. Conclusion

This shows us that even though some code can be parallalised, the amount of resources used, 16 cores compared to 4 - where for the 2 larger images, 4096×4096 and 8000×8000, the efficiency was 82% and 85% respectively - may not be worth the speed up, as it would cost more in terms of resources used (parallel processing on a super computer is not cheap).

References

[1] Nicolas LIMARE - Integer and Floating-Point Arithmetic Speed vs Precision http://nicolas.limare.net/pro/notes/2014/12/12_arit_speed/

[2] Boston University Info Services - Intel Compiler Flags http://www.bu.edu/tech/support/research/software-and-programming/programming/compilers/intel-compiler-flags/

[3] Parallel Code Optimisation, Chris Brady, 2008. https://warwick.ac.uk/fac/sci/csc/people/computingstaff/chris_brady/optim.pdf

[4] Examining the limits of Moores law - Possible influence of technological convergence on redefining the curriculum in ICT institutions, Markku Arimo Kinnunen (Advised by Sakari Lukkarinen), 2015

[5] Introduction to Computer Architecture in 2017/18, Professor. David May.

[6] COMS30005 - Lecture Week 7 - MPI 3: Advanced MPI, Gethin Williams, 2018.

[7] COMS30005 - Lecture Week 5 - MPI 1: Intro, Gethin Williams, 2018.

[8] Lecture32: Introduction to MPI I/O -William Gropp - University of Illinois. http://wgropp.cs.illinois.edu/courses/cs598-s16/lectures/lecture32.pdf