Distributed Memory Parallelism Optimisation

Abstract

The objective of this coursework was to incorporate parallelism to a serially optimised stencil code that implements a weighted 5-point stencil on a rectangular grid using Message Parsing Interface (MPI), a library interface for passing messages between processes. The following sections describe my approach to parallelism, comparison with the serial code and performance analysis.

Compiler Choice

Since optimising the serial code through parallelization, follows the Single Program Multiple Data model, I had to use MPI's version of an icc compiler wrapper `mpiicc` by Intel (version 2018). I initially used `-fast` as my compiler flag, however the `-fast` option forces a static linkage method for the Intel MPI Library. This results in the linker copying all library routines used in the program into the executable image. Hence, I decided to try the next best flags, `-Ofast` and `-O3`, with the addition of another flag `-xHOST` that Intel provides, which generates optimized codes for various instruction sets used in specific processors, in this case Intel Broadwell. I decided to proceed with `-O3`, as it is best for code that loops involve floating-point calculations, Furthermore, `-Ofast` enables some nonstandard compliant optimisations.

Domain Decomposition

The following sections describe my synchronous communication by splitting the image into sections, sending information and gathering all updated sections of the image using MPI functions. I had access to 56 cores available on 2 nodes of Blue Crystal Phase 4.

Initialization

Since the iteration order of the `stencil` function in the serial optimised code was in row-major, I decided to split an image in a row-wise manner. This is because the image is stored as a one-dimensional array and if the image were to be split row-wise, information can be stored contiguously. This would not require a send and receive buffer to pack all the cells to send between sections. The opposite would be said if the image is split column-wise. Furthermore, it is optimal for latency bound communication. Also, this was chosen over splitting the image into separate grids, as there are less neighbours to communicate with. The next step was to assign each section of an image based on the number of cores available and the height of the image. I used the function `calc_nrows_from_rank` to ensure the last ranked core is assigned with any remainder rows. Then, I allocated memory to each section and temporary section to hold information before writing onto the main image. I then initialized each section of an image with the values of that respective section of the input image. Finally, I assigned padding for each section (halo regions), this was to ensure information can be received from and sent to between neighbours.

Exchange of Information

Once the sections were initialized. The subsequent task was to ensure synchronous exchange between sections. I chose synchronous over asynchronous since, if messages sent asynchronously are buffered in a space managed by the OS, then a process may fill this space

by flooding the system with many messages. In addition, extra cores are necessary to handle the message parsing, which in our case the hardware existing is restricted. Furthermore, if a message cannot be delivered, the sender may never wait for delivery of the message, and thus never hear about the error, hence the use of synchronous messaging. To begin the Halo exchange of information, I had a loop over the number of iterations and used `MPI_Sendrecv` to send the last row of the current section to the section above and receive information from the last row of the lower section. I used to `MPI_Sendrecv` to ensure no deadlock occurs. Next, I ran the same command to receive row information from the top section and send row information to the lower section. The `stencil` function is then called to perform floating point operations each section and write onto the temporary section. `MPI_Sendrecv` is called again to work on the temporary sections and `stencil` will be called again to write back onto the section.

Gathering

After the row information has been successfully exchanged between sections, the final step was to gather all sections together by first checking if the current rank through `MPI_Comm_rank` is the first rank (master), this is because I wanted this rank to assign the information of a section to its corresponding section in the input image. If the rank was not the master, that respective rank will send all its information to the master rank using `MPI_Send` and the master rank will receive this information using `MPI_Recv` and allocates this information, to the corresponding section of the input image. Finally, I stopped communication between all threads using `MPI_Finalize`.

Comparison and Analysis

Once the synchronous communication was implemented, the next task was to calculate the run times with the parallelised code. The serially optimised code uses 1 core from 1 node with 0.10s, 2.95s and 11.28s as the runtimes for 1024x1024px, 4096x4096px and 8000x8000px respectively. Before calculating the run times, I wanted to see how well the parallel code was performing compared to the serial code, I decided to calculate the parallel efficiency for all cores, through strong scale efficiency, which uses the run time using one core to determine the parallel codes efficiency. While speedup is a metric to determine how much faster parallel execution is versus serial execution, efficiency indicates how well software utilizes the computational resources of the system. The Parallel Efficiency was calculated by dividing the serial run time of an image by the number of cores multiplied by the run time produced using those many cores. In addition to that comparison, however, it would be useful to compare the speedup relative to the upper limit of the potential speedup. This issue can be addressed using Amdahl's Law.

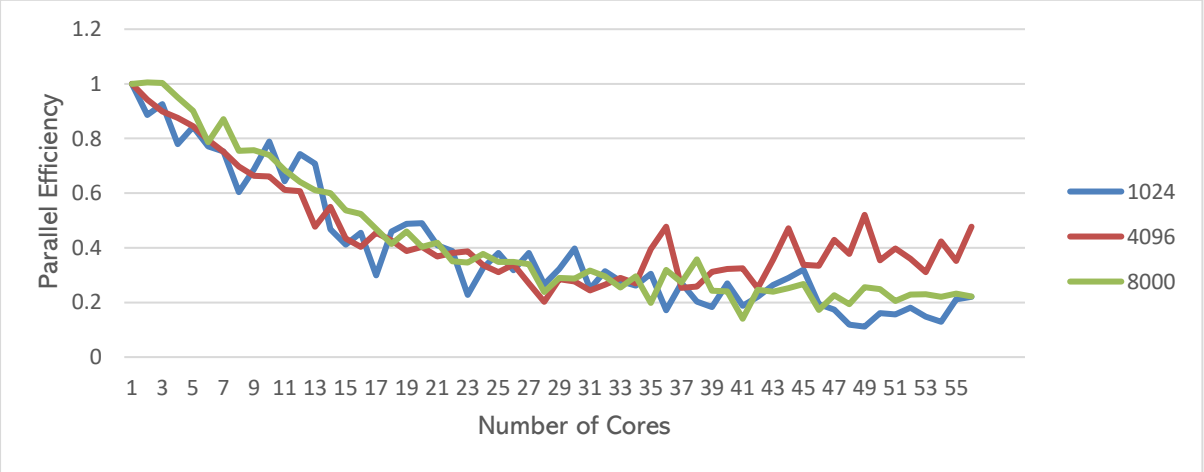

Figure 1: Parallel Efficiency for 1024x1024px, 4096x4096px and 8000x8000px

In *Figure 1*, we can see that as the number of Cores increases, the parallel efficiency decreases for all image sizes. Because, when the number of cores increases, the effectiveness of data transfer decreases. Since, the results imply that, on average, over the course of the execution, the amount of time each of the cores are left idle increases, when the number of cores increases. This is the result of a communications bottleneck. As more cores are added, each core spends progressively more time doing communication than effective processing. At some point, the communications overhead created by adding another core surpasses the increased processing power that core provides, and parallel slowdown occurs. In addition, we can infer from *Figure 1*, by stating that for large input sizes. The parallel efficiency is better compared to smaller inputs, provided the number of cores does not exceed an optimal number of cores. Finally, the results from *Figure 1* suggest this system is not scalable with the given images, as it cannot maintain efficiency by increasing the number of processors and the problem size simultaneously. Also, a scalable parallel system can always be made cost-optimal by adjusting the number of processors and the input size. In conclusion, a larger input image would improve the parallel efficiency, as it makes better use of the cores.
The next phase of analysis was to measure the run times as the number of cores increases. For each number of cores, I took the average runtime from 5 readings. This was to draw a relationship between the number of cores and average runtime using those cores. The average results using 56 cores from 2 nodes resulted in 0.01s with a variation of ± 0.04, 0.10s with a variation of ± 0.03 and 0.94s with a variation of ± 0.05, for 1024x1024px, 4096x4096px and 8000x8000px respectively. The section below describes my analysis of the data.

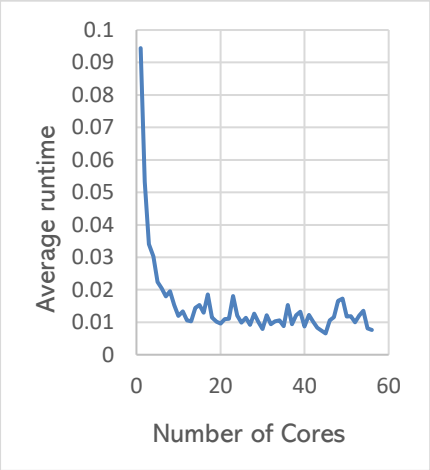

Figure 2: Runtimes for 1024x1024

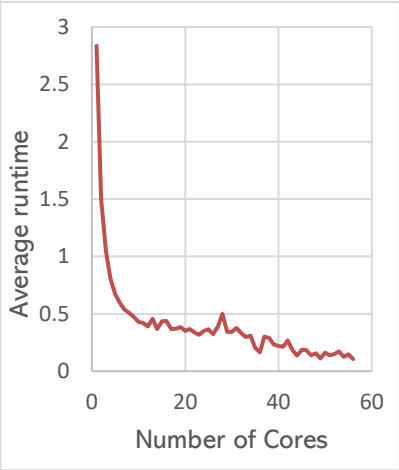

Figure 3: Runtimes for 4096x4096

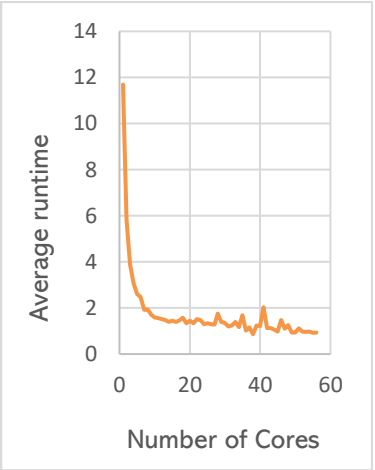

Figure 4: Runtimes for 8000x8000

As seen in *Figures 2, 3 and 4*, as the number of cores increases, the run time decreases in a logarithmic manner. This is beacuse, the cost is for the parallel system is $\Theta(n * \log * n)$ and for the sequential system is $\Theta(n)$. Since, the parallel efficiency initially is very high, as seen in *Figure 1*, there is a good amount of utilization of every core, leading to a linear scaling in performance initially. However as more cores are added, it decreases reaching a plateauing state, hence the flatness shown in *Figures 2, 3 and 4*, this results in a sublinear scaling in performance. One reason this occurs is due to an increase in data locality to minimize communication. Furthermore, for larger images the access in cache memory is less efficient with the use of larger number of cores. Moreover, numerous cores results in multiple splits leading to sections being too small, hence the run time will reduce every so slightly due to low parallel effiiciency. Finally, we can conclude this analysis by saying that as the number of halo

exchanges increases due to the increase in image size and cores used, there is a high proportion of communication over computation of floating point operations and a higher parallel efficiency leads to a bigger drop in runtime.

Performance Analysis

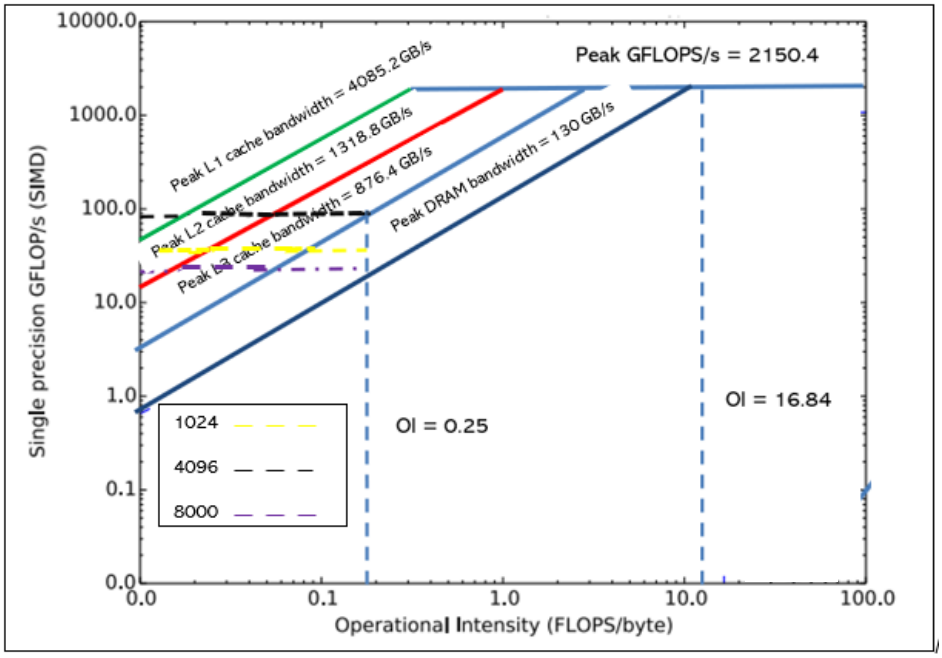

Figure 5: Roofline model of Intel Xeon CPU E5-2680 v4 (Broadwell) on 1 node (28 cores)

The final part of this experiment was to analyse the optimized code's performance, through a roofline model. This is to ensure if our updated code is within the memory bandwidth bound. As shown in *Figure 5*, I updated the peak cache bandwidth for L1, L2 and L3 caches by multiplying it's peak cache bandwidth for 1 core with 28, since I wanted to test the optimised code's performance for all cores available in a node. The same was done to calculate the peak theoretical GFLOPS/s and DRAM bandwidth for the hardware by multiplying its peak GFLOPS/s (1075.2) and DRAM bandwidth (65) for 14 cores (1 socket), with 2. We use Operation Intensity (OI) to measure the operations per byte of DRAM traffic. This is because we want to measure traffic between the caches and DRAM. The total bytes accessed are those that go to the main memory after they have been filtered by the cache hierarchy. That is, we calculate traffic between the caches and memory rather than between the processor and the caches. Since the OI will remain the same, the aim was to produce significantly better GFLOPs/sec compared to the serially optimised code. Calculating the GFLOPS/sec for 28 cores produced 99.3 GFLOPS/sec for 1024px, 53.3 for 4096px and 49.1 for 8000px. The GFLOPS/sec in the serially optimised code produced 12.71 GFLOPS/sec for 1024px, 6.83 for 4096px and 6.81 for 8000px. As you can see, there has been is significant increase for all image sizes. Furthermore, it shows that the program is memory bound. The Roofline sets an upper bound on performance of a kernel depending on its operational intensity. One improvement could be to improve instruction level parallelism (ILP). For superscalar architectures, the highest performance comes when fetching, executing, and committing the maximum number of instructions per clock cycle. In addition, you can ensure memory affinity, where you allocate data and threads tasked to specific data to the same memory-processor pair, so that the processors rarely have to access the memory attached to other chips.