HPC Report

Throughout this report I aim to demonstrate my understanding of serial optimisation, vectorisation and parallelisation with OpenMP by increasing the performance of the Lattice Boltzmann code given to me.

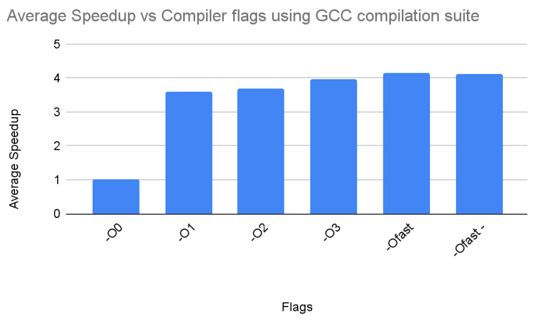
Serial Optimisation

At the start of this project I needed to understand both the code base and its performance. I used the profiler Gprof which informed me that ~70% of the total running time was being spent in the `collision()` function. So it was there I needed to devote a majority of my attention.

My immediate response to analysing the `collision()` function was to seek to help the compiler optimise some of the arithmetic statements in the function. I did this by finding mathematically equivalent equations that minimised the number of divides. However, the program is memory bandwidth bound so this was likely to lead to minimal improvement.

When optimising serial code it is important to pay attention to caching as accessing main memory can be 100 times slower than hitting L1 cache. Caches exploit both temporal and spatial locality. Caching assumes that data used once is likely to be used again, and that any nearby data is also likely to be used so it will bring a nearby block of data called a cache line into the cache. To help exploit this, I evaluated the decision between column and row major order. To ensure the program makes best use of our cache, it must move through memory in a contiguous fashion, so that the cache line contains the data it needs. For this to be the case, the innermost iterator in the double for loop must match the memory layout so that it moves through sequentially. This was already the case for the lattice boltzmann code.

The next stage was to fuse a number of loops in the code so that the program only moved through the grid once per timestep. This involved ensuring that there were no dependencies between one iteration of the loop and the other. I changed the code such that it reads values from `cells[]`, calculates new values and then writes them into the `tmp_cells[]`. Once the loop terminates it can then perform a pointer swap to ensure that there is no unnecessary copying between the two arrays and then continue on to the next timestep.



The above chart shows the average speed up of run time when varying the compilation flags for the GCC compiler suite version 9.3.0. An average of 3 runs was used to generate a run time using grid sizes 128x128, 128x256 and 256x256. 1024x1024 was ignored due to extremely long runtimes when using the -O0 flag. The relative speedups on the different grid sizes were consistent with each other.

I performed experiments comparing the performance of the Intel compiler suite vs the GCC compiler suite and also compared various versions of each of these suites. At this stage the best of these was GCC version 9.3.0.

The average serial run times are shown in the table below.

Grid Size	128x 128	256x 256	1024x1 024
Time(s)	24.14	194.04	826.92

Vectorisation

Single Instruction Multiple Data (SIMD) instructions can improve performance by executing a single instruction on a large amount of data. When code is able to make efficient use of SIMD instructions it is said to be vectorised. The advantage of vectorisation over parallelisation is that it reduces the total amount of work rather than just redistributing the work to get to the answer faster.

In order to get the compiler to produce vectorised code, I needed to make a number of changes. The first of these was to switch from an array of structures memory layout to a structure of arrays. This means that when a cache load is completed it brings with it the specific speed of the following cells instead of multiple speeds from a single cell. The former is more useful.

The next change was to ensure data alignment. Data alignment forces the compiler to generate objects in memory that are aligned to specific boundaries, which makes it easier for vectorisation to occur. This stage is two fold. Firstly, the code creates the data object using the `_mm_malloc()` function. It then informs the compiler of the alignment with `__assume_aligned()` before it enters the loop I want to vectorise. I chose an alignment of 64 bytes as this is the same as the cache line size. After running experiments with varying alignments, I found 64 bytes performed best.

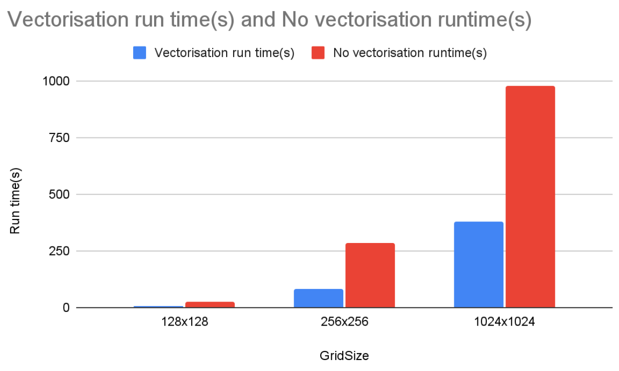
At this point the compiler was still unable to vectorise because I needed to provide aliasing information. To do this I marked all variables that could be constant as such. I used the `restrict` keyword to inform the compiler that the pointers will not overlap as this could introduce dependencies. This must be avoided when vectorising.

Loop unrolling is a technique the compiler may use to vectorise code. To aid the compiler in doing this I provided information about the loop counter using the `__assume()` function. With this I informed the compiler that the loop counter is divisible by 2 and powers of 2.

While completing the vectorisation, I found that the Intel Compiler was able to produce better vectorisation and thus better runtimes than GCC or other compilers. I found that best results were with Intel compiler version 19.1.3.304 and from here on I used this version for compilation. As a result of this vectorisation, faster run times were able to be achieved.The table below shows the average runtime of 3 runs at different grid sizes.

Grid Size	128x128	256x256	1024x1024
Time(s)	8.60	82.93	378.756

I performed an experiment using the structure of arrays code where all vectorisation was turned off using compiler flags and removing SIMD pragmas. The below graph shows the speed up that the vectorisation achieved.



Parallelisation

Parallelisation is the technique of using multiple threads to subdivide tasks and compute these tasks simultaneously. When programming with OpenMP, we use a shared memory which all the threads can access.

I began adding parallelisation to my code by using the `for` work sharing construct. After ensuring the data was either shared or private as appropriate, we saw an immediate decrease in runtime.

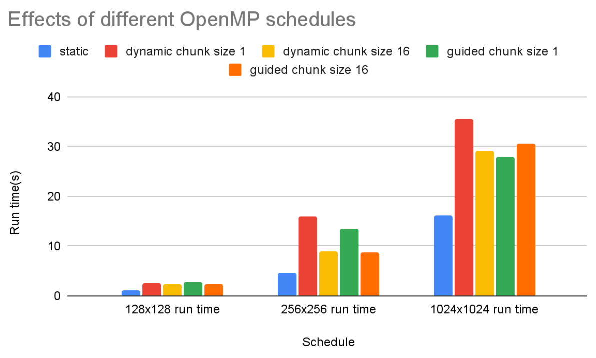
I then began to experiment with the scheduling clause. The scheduling clause dictates the distribution of work to threads. I tested performance using static, dynamic and guided schedules and additionally, tried varying the chunk size.

The static schedule divides the number of iterations as equally as possible across the number of threads.

When using the dynamic schedule, threads will request a chunk of the iteration space to work on until completed. It will then request more work until all work is completed. Dynamic scheduling works best with unbalanced loads. This is not the case for our program and using the dynamic schedule can add additional overhead which can degrade performance.

The guided schedule works in a similar fashion to the dynamic schedule except the size of the chunks decreases over time. This can reduce the amount overhead while still providing a method of tackling the unbalanced load problem.

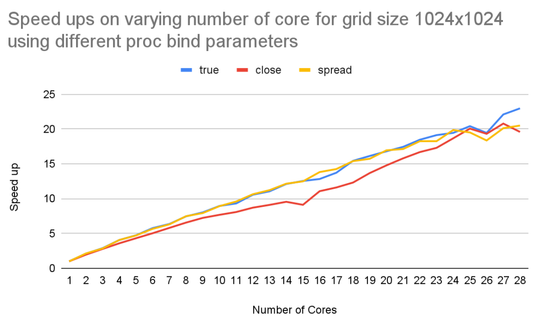
While running this experiment I used the runtime schedule which allowed me to change the schedule using environment variables. The results of these experiments were that the static schedule produced the best performance, as can be seen in the chart below.



In the main body of my program I have two shared variables `tot_u` and `tot_cells`. Access to these variables must be synchronised. The fastest method I found to complete this was by using the OpenMP reduction clause.

Blue Crystal Phase 4 has two sockets on each node and each of these sockets has an associated memory. Accessing memory from a different socket is slower than accessing its own as a socket-to-socket interconnect is used.

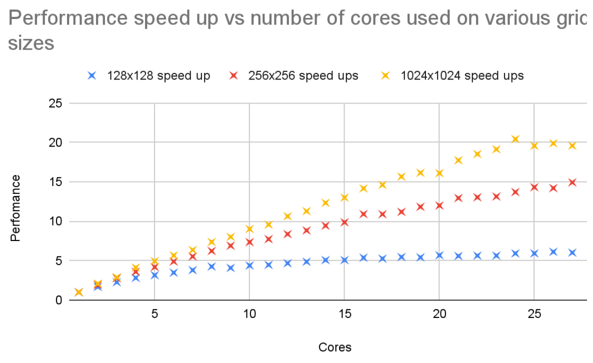
To reduce socket-to-socket interconnect transfers I wanted the threads to use data in memory close to the CPU the thread is running on. To achieve this, I parallelised the initialisation so that when the memory was first touched it was assigned to the memory close to the accessing threads. However, the OS can move threads to different cores or sockets, which can cause socket-to-socket interconnect transfers again. This can be prevented by pinning the threads using the OpenMP environment variables OMP_PLACES and OMP_PROC_BIND. OMP_PLACES controls how threads are assigned to places and OMP_PROC_BIND controls where/if the threads are pinned. Doing all this reduced the variability in runtimes, particularly in the larger grid sizes.



The above graph shows the speedups as the number of cores increases when using the 1024x1024 size grid. On Blue Crystal phase 4 there are 14 cores per socket.The close parameter ensures that threads are kept close to the primary thread which is why there is a sublinear plateau around the 14th core.

The runtimes across the different grid sizes of the parallel code are in the table below.

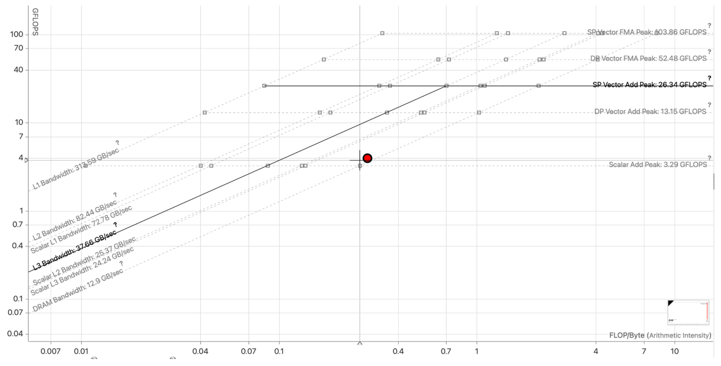
Grid Size	128x 128	256x 256	1024x1 024
Time(s)	1.12	4.68	14.69



The above graph shows how the performance improved as the number of cores increased on the different grid sizes I experimented with. In the case of the 128x128 grid there was evidence of sublinear plateau, whereas in the 256x256 and 1024x1024 grids there is a much more linear trend with decline happening only towards the higher number of cores. When working on a larger area, the benefits of parallelisation outway the additional overhead of more threads which is this behaviour can be seen.

Roofline graph

A roofline graph has axes of operational intensity (the number of operations being performed in the loop that dominates runtime divided by the number of bytes of data moved to do that) on the x axis vs performance specific to the data type that is being used, in this case single precision floats. A roofline graph contains a plot representing the peak theoretical performance of the hardware and also, a plot of the peak memory bandwidth multiplied by the operational intensity. The minimum of these two lines is taken and together they form the namesake roofline.



This roofline graph was generated earlier in the project with Intel Advisor after running the 128x128 grid on 1 core with only ~8% of the time being spent in auto vectorised loops. At this grid size, all of the data should be able to fit into the L3 cache. The Vector add peak and the L3 Bandwidth are indicated with darker lines. The result is to the left of the L3 cache ridge point so the program is memory bandwidth bound. The code is using 55% of the L3 memory bandwidth which is good.

Conclusion

The final times for serial optimization, serial optimization plus vectorisation and parallelisation are included in the table below demonstrating that using a combination of serial optimisation, vectorisation and parallelisation produced the shortest run times.

Grid Size	128x 128	256x 256	1024x 1024
Serial optimisation times(s)	24.14	194.04	826.92
Vectorisation time(s)	8.60	82.93	378.75 6
Parallelisation time(s)	1.12	4.68	14.69