Advanced High Performance Computing: Optimising Lattice-Boltzmann with OpenMP

Michael Rollins (mr16338)

1 Introduction

In this assignment we were asked to optimise an implementation of the Lattice-Boltzmann algorithm for simulating fluids according to the Boltzmann equation. First I was required to perform serial optimisations to minimise the run time on a single processor core, then to implement OpenMP to allow the code to be run on multiple threads on the same node. The first thing I did was select the compiler and compiler flags that I would use to produce the lowest run times. I decided on using ICC 16.0.2 with the flags -Ofast and -xAVX, these flags tell the compiler to maximise the optimisations it performs on the code and to generate SIMD instructions based on the CPU architecture respectively. Using these compiler options on the originally provided code produces run times on square grids of size 128x128, 256x256 and 1024x1024 of around 25.2s, 202.8s, and 714.7s respectively.

2 Serial Optimisations

2.1 Loop Fusion

The first serial optimisation I performed was to perform a loop fusion by combining the propagate(), rebound() and collision() functions into one function. This means that we reduce the number of times the program needs to iterate over the grid, thus reducing the number of arithmetic and memory operations performed and resulting in a speed up. I also realised that it was possible to merge av_velocity() into the same function as the local_density, u_x and u_y calculations were repeated from the collision() function. These changes meant that I could have the main function return the calculation that av_velocity() previously returned, which I then had the timestep() function return which assigned it to av_vels[tt].

2.2 Pointer Swapping

In the original code, the entire grid is copied from the cells grid, into the tmp_cells grid and then back to the cells grid for each iteration, it quickly became clear that this was unnecessary and that only one copy operation was required per iteration, which would cut down on the number of memory operations performed and thus increase performance. To do this all nine speeds for each cell (the loop acts on one cell per iteration) are transferred into temporary variables, which are then used for the rest of the calculations, so they only need to be copied into tmp_cells once during the relaxation step. At the end of the loop this leaves the current grid arrangement in the tmp_cells grid, so I had to edit the timestep loop so that tt incremented by 2 each time so that I could have cells and tmp_cells swap between each iteration of the main loop.

2.3 Vectorising

To help the compiler perform vectorising on the code, I went through and set as many variables as possible to const so the compiler knows they will not change, I also set the function parameters of cells and tmp_cells to restrict so that the compiler knows there is no overlap between them. A significant change I made in the code was to change the data layout from an Array of Structures configuration to a Structure of Arrays configuration. Applying this change initially slows the code down by around 6 seconds on the 128x128 grid, however once the compiler starts vectorising the code efficiently the speed gain is far greater than this. To help the compiler further, I used memory techniques including swapping malloc() function calls for _mm_malloc() calls instead, which returns the data aligned to a requested boundary, in my case 64 bytes to match the cache size. Also at the start of the main calculation function I added _assume_aligned() statements which tells the compiler the pointer it is describing is

aligned to a certain number of bytes, in this case 64. Finally, I added #pragma omp simd outside the start of each loop which explicitly tells the compiler to vectorise these loops if possible. Once the compiler started vectorising the code there was a significant increase in performance with runtimes of 7.67s, 62.43s and 272.39s for square grids of size 128, 256 and 1024 cells.

3 OpenMP Optimisations

Having successfully performed serial optimisations, I moved onto to using the OpenMP interface to allow the code to run on multiple threads. The first thing I did was to use an #pragma omp parallel for statement around the main loop, this informed the compiler to use worksharing to fork a team of threads to run on multiple cores, in this case to a maximum of 16. By using the default schedule the iterations of the loop are divided equally across the number of threads, this appeared to be the most efficient way of dividing the work. Next I noticed that for each iteration of the loop two reductions were taking place, for every cell that is not an obstacle 1 is being added to a variable counting the total number of cells and the normal of the x and y velocity components are also being totalled. This lead me to adding reduction(+:tot_cells, tot_u) after my parallel for statement so that the reduction would be completed in parallel. Having implemented OpenMP which I set to run on 16 threads I managed to achieve times of 1.10s, 5.67s, and 48.82s for grids of size 128x128, 256x256 and 1024x1024 respectively. These are speeds ups of 7 times, 11 times and 5.6 times.

3.1 NUMA-aware Code

The final improvement I made to my code was to parallelise the initialisation of the cells and obstacle variables in the same way in which the main loop is parallelised. This takes advantage of the operating system's first touch policy which means that memory will be allocated near to the first thread that touches it (initialisation). By parallelising the initialisation in the same way as the main loop it minimises the amount of socket-to-socket interconnect transfers necessary, these take longer for a thread to access than memory that is stored nearby to it, which reduces the speed of the code. To prevent the operating system from moving threads between cores, which would mess up NUMA-aware code, I used the environment variable OMP_PROC_BIND=true to prevent threads from being moved. With these changes I achieved my fastest times, which on 128x128, 256x256 and 1024x1024 grids were; 0.93s, 5.50s and 41.22s. However I did find that running the largest grid on just 14 cores actually increased the performance and the run time was only 33.44s.

4 Conclusion

There is similar behaviour across all three sizes of grid, as more threads are used, there is an increased speed up. The rate of increase in speed it up is quite close to linear, where the speed up is just less than the number of threads being used. This makes sense as for 2 threads, the work of each thread is approximately halved and of course there will be some overhead.