Andreas Georgiou - ag14774

High Performance Computing, Coursework #1

Introduction

The objective of this assignment was to optimise the given serial lattice Boltzmann code and then use OpenMP to parallelise it by using all 16 cores in a Blue Crystal phase 3 node to make it go as fast as possible. The approach used for both, the serial optimisations and OpenMP parts of the assignment to determine how to proceed to optimise the code was to first profile the code(using TAU) to determine the most time consuming parts, experiment by changing the code in an attempt to optimise it and finally profile again to determine whether the change resulted in a speedup and decide whether the change should be kept or discarded. In this report, I will present all the successful (and some of the not so successful) changes that resulted in the speedup of the serial lattice Boltzmann code running on 1 core from 855.450 seconds to the parallel version running on 16 cores at 7.30 seconds with double precision and 6.3 with single precision(average running time for the 256x256 input data). Throughout the report, the timing shown is the total wall-clock time calculated using the built-in function *gettimeofday()*.

Serial Optimisations

1)Simplifying expressions and removing divisions

After running the profiler on the initial code, it was clear that the most time consuming function is *collision()* taking up 76% of the time used. Knowing that divisions require a lot more CPU clock cycles than addition or multiplication, an attempt was made to simplify all the mathematical expressions as much as possible and to remove repeated divisions by multiplying with their reciprocal where the reciprocal has to be calculated only once(e.g division by *local_density* becomes multiplication with *1/local_density*). To aid in the simplification, the computational engine of WolframAlpha was used to find alternative forms of the expressions quickly. Moreover, the array *u[]* was completely removed and the variables u_x and u_y were directly used in the rest of the expressions. Clearly, as shown by the results below, this had a significant improvement in performance (approximately 2.56x faster than the initial code):

	BEFORE(GCC)	**AFTER(GCC)**
	Total elapsed time (s)	Total elapsed time (s)
128x128	105.491	41.395
128x256	213.252	83.241
256x256	855.450	334.035

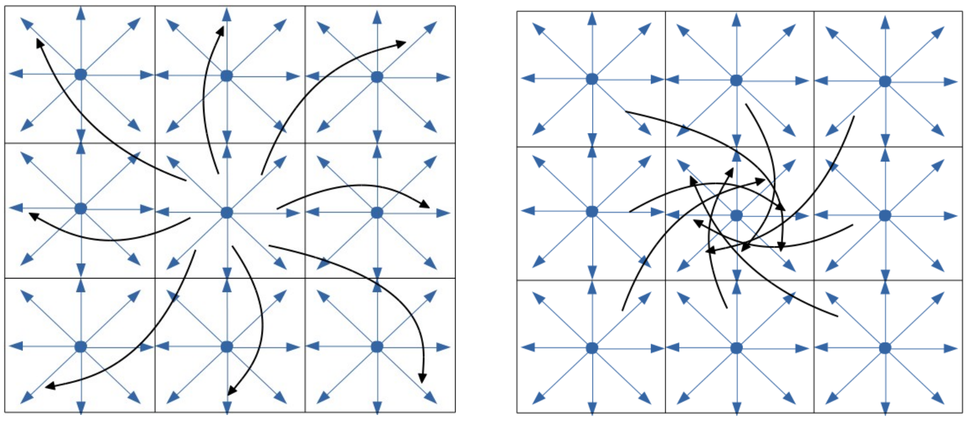
This speedup of course also comes at the expense of precision loss because multiplication with the reciprocal of the denominator cannot utilize the full IEEE division accuracy. However, the loss in precision is minimal (-1.2e-10 % difference in the final state).

2)Preprocessing obstacles (Failed optimisation)

At this point, *collision()* is still the most time consuming function. Since a check for an obstacle is made in every iteration of the loop, an attempt to preprocess the obstacles was made. During the initialization phase, the *obstacles* would be filled with negative and positive numbers representing the presence of consecutive blocked cells and unblocked cells respectively. For example, a cell containing -5 indicates that the next 5 cells, including the current cell, will be blocked while a +8 implies that the next 8 cells are unblocked. This meant that there was not need to constantly check for obstacles. However, after profiling the new presumably more efficient code, it was discovered that there was no performance gain at all. This is most probably due to the presence of a branch predictor circuit in the processor that effectively minimizes the impact of the if statement by predicting the outcome in advance.

3)Merging *propagate()*, *rebound()*, *collision()* and *av_velocity()*

The next step is to merge the most time-consuming functions into a single function. The reason is that those functions each require a full pass over the whole grid, which means that memory is being revisited and large amounts of data has to be reloaded to the cache. Merging *collision()* and

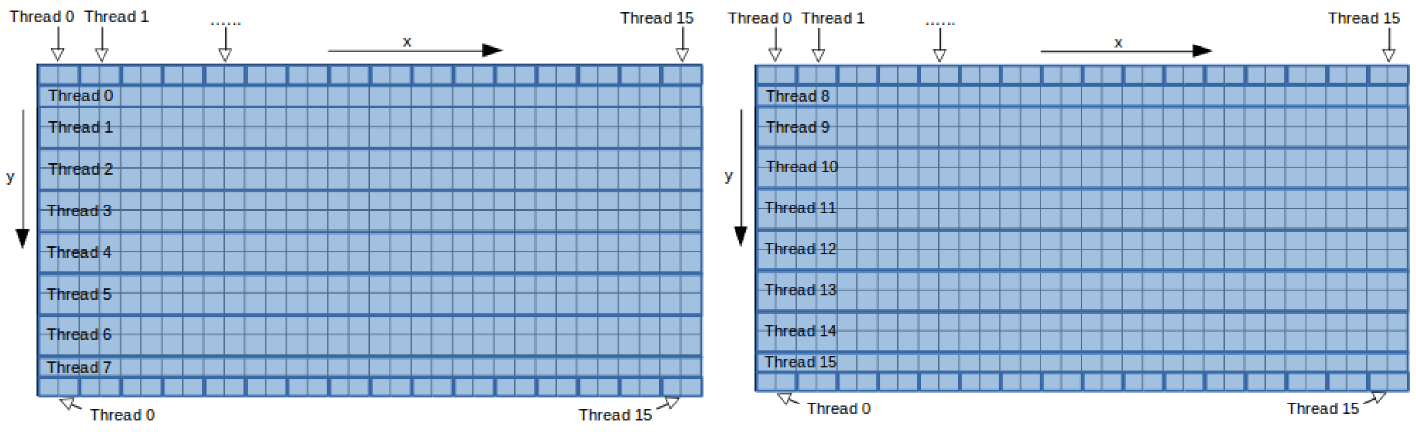

(**figure 1**) Left: Original propagate(). Speeds are copied **to** surrounding cells. Right: Speeds are copied **from** surrounding cells

rebound() is straightforward, since they are both executed after *propagate()*, with the first being executed when a cell is not blocked and the latter when it is blocked, so combining them into a single loop with an if-else statement instead of two loops should give a significant performance boost. In addition, *av_velocity()* can also be combined in the same loop since u^2 is already calculated as part of *collision()* and thus total speed can easily be computed. Lastly, merging *propagate()* is to a certain degree trickier. When a single iteration is finished, all the speeds of the current cells are being copied to the surrounding cells, which means that the current cell's speeds have not been updated yet and hence, it is not possible to proceed on using that cell to execute either *collision()* or *propagate()*. To solve this problem, we have to observe the dependencies between the cells. It is clear that for a cell to be fully updated, the loop body of *propagate()* has to be executed for the surrounding cells first. To circumvent that restriction, the loop body was changed so that each iteration now updates the current cell by copying the speeds of the surrounding cells(**figure 1**). This change now causes further problems with our newly merged *collision/rebound* function. Since those functions copy back from our scratch space to our main array, it is clear that the original speeds of the current cell in the current iteration would be overwritten before its surrounding cells had the chance to use them. This can be easily solved by two small changes: 1)change the function *collision/rebound* to make any changes in place(i.e in our scratch space). This leaves our main array with all the original speeds that are required by the rest of the cells. 2)the pointers *cells* and *tmp_cells* must now be swapped in every timestep because after a single iteration our updated state of the game resides in *tmp_cells*. Hence *tmp_cells* should be used as the main array and *cells* as the scratch space for the next iteration. All 4 functions are now merged to a single function called *timestep()* which lead to a further speedup of 2.30x as shown in the table below:

	BEFORE(GCC)	**AFTER(GCC)**
	Total elapsed time (s)	Total elapsed time (s)
128x128	41.395	18.090
128x256	83.241	36.324
256x256	334.035	145.178

4)Vectorising *timestep()*

The last serial optimisation is to vectorise certain parts of the timestep() function. Since Blue Crystal supports the Advanced Vector Extensions(AVX) of the x86 instruction set, it is possible to process 4 cells at a time using 256-bit registers(4 doubles). To achieve that, each row is split into tiles of 4 cells. Each time, the current 4-cell tile is copied to a local array and then multiple for loops process the tile 1 cell at a time. Variables such as u_x and u_y are now arrays of size 4 and because the inner for-loops have a constant size of 4, the compiler can now very easily vectorise them and allow for processing multiple pieces of data in a single step. For this, the Intel Compiler (icc 16.0) was used. The first attempt at this was unsuccessful with the compiler complaining about unaligned access which resulted in a significant slowdown. Fortunately, the Intel compiler provides all the tools



(**figure 2**) shows how work is divided among the threads. The first half of the array allocated closer to the first processor is shown on the left while the second half is shown on the right. The grid used here is of size only 32x32 to make illustration easier. Thread 0 has to give up processing of row 0 because it is a special row that requires access to both arrays(same for thread 7, 8 and 15 for rows 15,16 and 31 respectively). Special rows are processes separately by all the threads.

required to easily fix this problem by allowing us to align statically declared arrays using "*__attribute__((aligned(32)))*" (32-byte alignment is recommended by Intel for 256-bit access[1]) and by reassuring the compiler that alignment was taken care of using "*pragma vector aligned*" before each loop that needs to be vectorised. In addition, all pointers were declared using the keyword "restrict" so that the compiler knows that there is no aliasing or overlap between pointers and that writes to a pointer will not affect the values read through any other pointer. This resulted in a significant speedup of 1.53x as shown in the table below:

	BEFORE(GCC)	**AFTER(ICC)**
	Total elapsed time (s)	Total elapsed time (s)
128x128	18.090	11.972
128x256	36.324	23.915
256x256	145.178	94.688

Moreover, because of the vectorised loops, switching from double precision to single, allows the CPU to process more cells at the same time because now 8 values can fit into its 256-bit registers. Even though this resulted in a speedup of 11.2 seconds bringing the serial code to 83.480, the change was reverted in favor of double precision, due to the fact that using single precision arithmetic resulted in a considerable error of 0.11% in both the final state of the grid and the resulting average speeds.

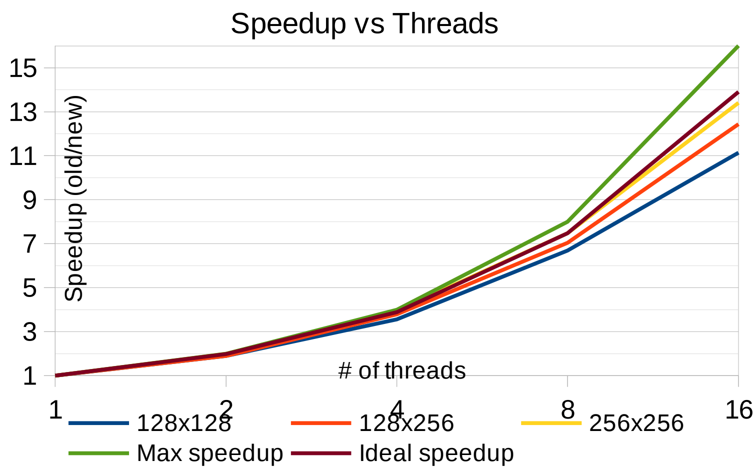
5)Blocking loops (Failed optimisations)

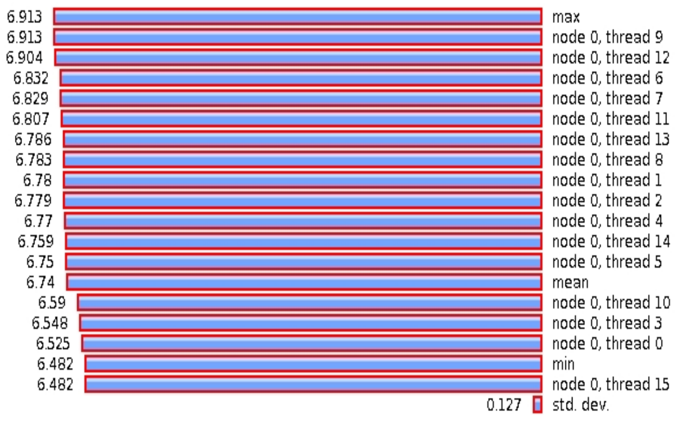
Trying to get the most out of the cache and reduce cache thrashing, blocked loops were used, where cells where processed in blocks of 4x4. However, this was not successful and instead caused a slowdown of approximately 4.02 seconds for the 256x256 case. This is most probably due to the fact that the Intel compiler already does this for us, and therefore the introduction of manually blocked loops prevents the compiler from performing its usual optimisations.

OpenMP and Parallel Design

The best and most elegant way to parallelise this code would be with a single "#pragma omp parallel for reduction(+:tot_u)". This keeps the serial design unchanged while providing a reasonable speedup of 8.07x with the 256x256 grid running at 11.727 seconds. However, in an effort to achieve the best possible speed, a more complex parallel design was implemented. Having in mind the effects of Non-Uniform Memory Access(NUMA) and the first touch policy used by the memory allocator, the initialization procedure was parallelised by allocating half of the space required to Memory Socket 0(i.e. closer to the first Xeon processor), and the other half to the second(i.e. closer to the second

1 https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions





(**figure 3**)Left: Scaling graph of ideal and max speedups. Right: Load-balancing of *timestep()*

Xeon processor) which involved having two pointers for each array(e.g. tmp_cells0 and tmp_cells1). Moreover, to make sure that the threads bind to the correct processors(i.e. threads 0-7 to cores 0-7 and threads 8-15 to cores 8-15) and they are not moved, OMP_PROC_BIND environment variable was enabled. When processing the cells close to the edges, access to both arrays is still required. To avoid the overhead of having to check each time in which array a cell resides, the special cases are treated separately in a function *timestep_row()*, which given the row number, it will advance that row by a step by basically doing all the operations timestep() would do with some additional checks to determine the arrays that are required to be accessed. In addition, to avoid the destruction and recreation of all the threads in each iteration, the threads are created once before the execution of the main loop. The function timestep() now is parallelised by having each thread calculate the range of rows it is going to process based on its thread id, while the function t*imestep_row()* is parallelised by having the threads calculate the range of cells to execute using the same approach(**figure 2**). All threads calculate the total speed of their respective parts and accumulate it at the end of the timestep to calculate the average speed. This is done atomically to prevent race conditions. Moreover, barriers are placed before and after the execution of *accelerate_flow()*, again to prevent race conditions. In order to minimize the effect of the barriers, the wait policy of OpenMP was changed to ACTIVE. This stops the threads from going to sleep and instead forces them to busy wait for the rest of the threads.

Parallel Performance Analysis vs. Serial Performance

The final times for the parallel system are: 1.072, 1.923 and 7.308 for the 128x128, 128x256 and 256x256 cases respectively. To better understand the effectiveness of the parallel system, it is important to compare it with the optimised serial code and to determine how well it scales as the processor cores increase. To demonstrate this, a graph(**figure 3 Left**) is being used where the maximum and ideal speedups are shown as well as the speedups of each of the 3 input data set when run with different number of threads. The ideal speedup is calculated using Amdahl's law assuming that 99% of the program can be parallelised(i.e. $1/\left(\frac{0.99}{N}+0.01\right)$where N is the number of threads used). In the case of 16 threads, the ideal speedup is 13.91x which is very close to actual speedup achieved of 13.41x. Moreover, it is clear that the larger the problem size is, the effect of the 1% serial code diminishes and it approaches the ideal speedup. This is also due to the fact that the parallelisation overhead is much smaller compared to the rest of the problem. Another important factor in a parallel system is load-balancing. As it can be seen in (**figure 3 Right**) all threads are kept busy for the same amount time during the most time consuming function *timestep()*. This ensures that all threads arrive at barriers at similar times and therefore the wait is minimised. Concluding, it is important to note, that there is always room for improvement. For example, potential speedup could be achieved by compressing data (e.g.the obstacle array could be represented by a bitmask) or by using ICC's profile guided optimisation flag. Finally, an important observation is that significant speedup can be achieved without necessarily breaking the serial code and that is the approach that should be followed in most cases.