Name: Mike Cooper, User: mc14641

Serial optimisations and OpenMP Report

The report will discuss serial and parallel optimisation methods used to speed up a template Lattice Boltzman code. Serial optimisations were applied and maximised before attempting to run the program in parallel using OpenMp. The code has been designed to run on a single node on Blue Crystal Phase 3 (2 x Intel E5-2670) with Intel's ICC compiler, version 15. Note that all timings and tests will refer to the 128x256 input file to eliminated the possibility that an optimisation would only work on an n by n input type.

Serial Optimisations

1.1 Flags

Without any optimisations, using only the GCC compiler with '-std=c99' flag, the run time was 493s. After adding the '-O3' optimisation flag, the time decreased to 213s. Comparing this to the ICC compiler with the '-xHOST' flag, it ran faster again at 209s. I believe the difference in time between the two compilers is most likely because GCC is designed to produce extremely portable code that runs on a multiple x86 architectures. Whereas the ICC compiler is specifically designed for intel CPUs and therefore will have more specific optimisations for Intel architectures. The optimisation flags on both compilers clearly make a very significant difference, producing around 2.3x speedup on each. Without any optimisation flags, I have learnt that the compilers aim is to reduce the cost of compilation and to make debugging produce the expected results; therefore, turning on these flags hinders these objectives. They produce optimisations such as loop unrolling, common-sub expression elimination, constant propagation, vectorisation, function inclining. I realised that understanding how the compiler optimises code is important in order to prevent unnecessary changes and to appropriately structure the program, specially the mathematical functions (discussed later).

1.2 Temporal Locality

The functions *propagate(), rebound(), collision()* and *av_velocity()* are called on each iteration in *timestep()* and they all repeat the same nested for loop, operating on the same data structures. As both *rebound()* and *collision()* read from *tmp_cells* and write to *cells* at the same indexes, they are only sequentially dependent. I therefore I was able to move *rebound()* into *collision()* making sure the code from *rebound()* precedes that in *collision();* this resulted in a speed increase to 206s. The same relationship occurs between collision() and av_velocity(), they are only sequentially dependent. I combined the two, resulting in a speed up to 202s. I believe the reason for the speed improvement is because the repeated for loops have been combined and repeated conditional statements have been removed, therefore reducing overhead.

Both the *collision()* and *av_velocity()* function calculate the velocity magnitude using values u_x, u_y and local density, however *collision()* calculates these values before the 'relaxation step' and av_velocity() afterwards. I found that if you only use the values calculated in *collision()*, ignoring av_velocity(), the correct values are still produced. By making this change, we effectively include the velocity magnitude for the 1st iteration before the relaxation step (a value which was not used in the original implementation) but consequently miss the velocity magnitude on the last iteration. This could account for the slight increase in 'Total av_vels difference' which increased by 0.003*e^-9. However, given we have significantly reduced the amount of computation necessary (now only calculating u_x, u_y, and local density once) and the error is negligible, I think this is acceptable.

1.3 NUMA

Propagate() assigns a new speed to each surrounding cell and uses two data structures, *cells* and *tmp*_cells. The propagation method can be seen in the diagram *Figure 1* or illustrated by code: *tmp_cells[index + shift].speed(j) = cells[index].speed(j)* .
This method means results in write operations occurring in non-localised memory. It's likely that writing to cells at multiple locations, which could exist outside of L1 cache or on another core, is less efficient that only writing to one location. It is my understanding that the reason this is less efficient is because if we write to a memory located on another core, we would need to flush the current cache line, load the data, write to that location and write back to memory. Whereas, if we only read from another core, we flush the current cache line, load the data and store the required value; we do not need to write back to memory as we have not invalidated cache line.

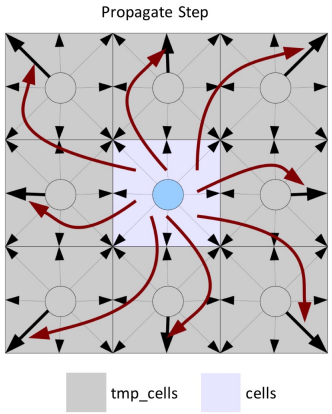

Figure 1

To resolve this inefficiency, I needed to alter the data aggregation method such that the propagate function only writes to the current index (ii * params.nx + jj) of tmp_cells. I reversed the existing procedure as shown in the 'Reverse Propagate Step' *Figure 2*. Instead of *tmp_cells[index + shift].speed(j) = cells[index].speed(j),* we now have *tmp_cells[index].speed(j) = cells[index + shift].speed(j)*. This means that writing back to the memory can now occur as a batch process, which is much more efficient and resulted in a speed increase to147s.

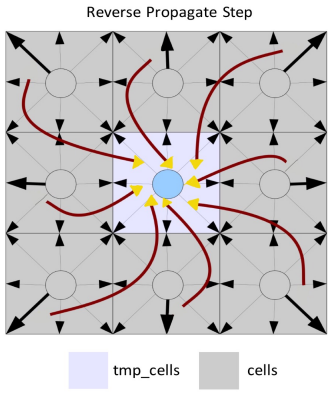

Figure 2

1.4 Data Dependencies

The change to the propagate function also solved a data dependence problem between *collision()* and *propagate()*. As both functions only write to tmp_cells and now write to the same index of the tmp_cells, I moved *propagate()* into collision(). This resulted in a speed increase to 139s; at this point all calculations were contained within one nested for loop, with the exception of accelerate_flow().

After combining the two functions, I observed a relationship between *tmp_cells* and *cells*. On each iteration, the code from *propagate()* assigns data from surrounding indexes (from *cells*) to the current index of tmp_cells (as previously explained). Following this, the same index of tmp_cells is used to calculate u_x, u_y and the relaxation step. This means that the assignment of cells to tmp_cells in the propagate step can effectively be bypassed. Therefore I replaced every occurrence of *tmp_cells[index].speed(i*) with *cells[index + shift].speed(i)*. On each iteration, *collision()* is now only ever reading from *cells* and writing to tmp_cells. However, only reading from the *cells* array means it will never be updated. To resolve this, I swap the pointers of cells and tmp_cells on each iteration of *timestep()*. This significantly reduced the number of times we read and write to memory and resulted in a speed increase to 109s.

Interestingly trying to combine accelerate_flow() and collision() resulted in a slower time. As accelerate_flow() only iterates over the top row of the grid, the number of iterations in comparison to collision() is much smaller and not computationally expensive. When combined with collision(), the conditional statement necessary inside the nested for loop will most likely create more overhead than putting the computation in a separate function that precedes collision().

1.5 Mathematic Optimisations

The calculations for axis and diagonal speeds, contained in the d_equ array in collision() appeared to be very repetitive. After substituting in u_x and u_y from the 'directional velocity components', *u[NSPEEDS]* and 'velocity squared, u_*sq*, it was clear that the equations could be reduced. I found that the best method was to substitute *u_x* and *u_y* using a find and replace tool in my editor and then reduce them by hand. Here is an example reduction:

Before: (1.0 + u_y / c_sq + (u_y * u_y) / (2.0 * c_sq * c_sq) - u_x * u_x + u_y * u_y / (2.0 * u_x * u_x + u_y * u_y))

After: c*(4 + u_x * 12 + (u_x * u_x) * 648 * d1- (216 * d1 * (u_x * u_x + u_y * u_y)))

The first attempt at minimising the equations worked well, the speed reduced to 79.3s. The large reduction in speed is most likely down to the reduced number of calculations/operations the program has to compute. As these calculations are located in the inner most for loop, on a 128x256 input they would be calculated a large number of times, any reduction will have a significant effect on run speed. Given this, I found that the key to speeding up this area of the code was to create repeated calculations. It's my understanding that this enables the compiler to only calculate a subexpression once and then reference the value on all subsequent occurrences. Therefore, the calculations were re-written with as many repeated expression as possible whilst still remaining reduced. In 8 / 9 calculations (*u_x * u_x)*, (*u_y * u_y)*, and in 4/9 *(u_x + u_y), (u_x - u_y)* were repeated. This resulted in a speed increase to 70.2s.

Computational expense is approximately ordered in the following way, where the integers represent relative weights:

(1 - Addition/Subtraction) < (4 – Multiplication) < (10 - Division/Modulo) < (50 – sqrt, pow)

Therefore, it's obvious that reducing the number of expensive computation will help speed up the program. To reduce the number of divisions, I added a 'static const' that was initialised at the start of *collision()*. Using 'static' means the value is maintained between functions calls, this is useful as *collision()* is called many times and dynamically allocating a constant value on every iteration would seem inefficient. Following this, I replaced all divisions with multiples of the declared 'static const' and factorised expressions such that a maximum of one division occurs per calculation. I also replaced any small multiplications for the equivalent addition or subtraction; in total this resulted in a speed increase to 59.4s.

1.6 Vectorisation

After looking at the ICC compiler vectorisation report it stated that there were vector dependences between the two data structures *cells* and *tmp_cells*, specifically FLOW and ANTI dependence, and so the loop could not be vectorised. However, in my implementation of the collision, this isn't true. Data is only written to *tmp_cells* but never read, and only read from *cells* but never written. I attempted to manually indicate to the compiler that it should ignore these dependencies, however the attempts did not work. After changing the ICC compiler version from 16 to 15 and using the flag '-xHOST', the compiler vectorised the loop without dependency problems. The speed then decreases to 46.34s. It is my understanding that reason for the speed up is down to the utilisation of the SIMD architecture that enables the processing of multiple data with a single instruction. Vectorisation decreases the number of computations performed by the CPU by performing the same operation on multiple data points simultaneously. After reading the documentation I learnt that the '-xHOST' flag tells the compiler to use the highest SIMD instruction available on your host machine (as long as it's supported). The Intel E5-2670 chip set has a AVX unit with vector registers of 256 bit. This effectively means four double-precision floating-point values can be operated on in parallel which significantly reduces the number of operations.

1.7 Redundant Optimisations

I implement a method that used only a single array instead of a current array and a scratch space array. I changed the size of the array such instead of n*m it was (n+1)*(m+1). This meant that after the values were initialised there was an unused row and column. The extra space allowed me implement an algorithm such that the 'Reverse Propagation Step' could be used to arrogate the values for the current cell but instead of storing them in the current cell, you store the values in the cell diagonally left (downwards), as can be seen in figure 3. This means you're not overwriting data that will be used later in the iteration but at the same time can maintain the structure of the n*m grid. Storing the new values in this manner effectively means the grid alternates between shifting diagonally right (upwards) and diagonally left (downwards) on each iteration shown in figure 3 & 4. This resulted in a speed up to 32.34s however, when running in parallel this method was invalid because each iteration is dependent on the previous iteration. This approach is perhaps more suitable for an MPI implementation.

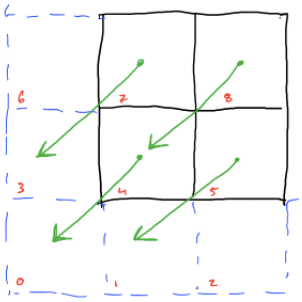

Figure 3 – ith iteration

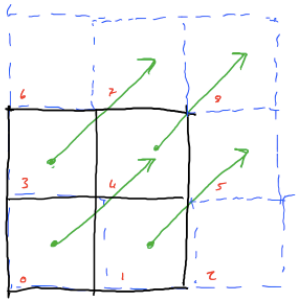

Figure 4 – ith + 1 iteration

On the each core, the L1 cache is 20KB and the size of the t_speed data structure is 72B, therefore L1 cache can hold up to 444 t_speed elements at any one time. If our aim is to operate on only data stored in L1 cache then the current method of iterating line by line is somewhat inefficient, as each index requires data from the all surrounding cells not just adjacent cells. I believe a more efficient method would involve loop blocking/tiling where the majority of computation is completed with as few cache misses as possible (temporal locality) and afterwards compute the edge cases. I attempted to implement this using tile block size of 16 by 16. The reason for this is because if L1 cache can hold 444 elements, sqrt(444) = 21.07, 16 is the closest 2^x value, where x is an integer. Unfortunately, I could not get the tiled implementation to run faster than the non-tiled version.

Parallel Optimisations

2.1 Threads

To select an appropriate location to start a parallel section, I used VTune to profile my code. I was not surprised to see that over 99% of computation is completed in my collision function. It seemed like a sensible choice to put a "#pragma omp parallel for" around the outer for loop in *collision()*. However, this ran at 105s, about 2.4x slower than the serial equivalent. It's my understanding that, 'parallel for' creates a new team of threads, and assigns each team to handle different portions of the loop. However, I then realised that each thread had access to two shared variables, *tot_cells* and *tot_u*, which could be creating a critical region. As such, each thread was having to acquire and release all the locks created by this region (mutex). My solution was to use a reduction clause, and this reduced the time to 4.213s. I believe this creates a private copy of each variable in each thread. At the end of the reduction, the reduction variable sums all private copies of the shared variable, and the final result is written to the original globally shared

variable. I also saw another improvement by specifying the *num_threads* clause. As 16 is the number of cores available on a node and loop_iterations%16 = 0 is always true on the given inputs, I believe using the *num_threads(16)* should optimise the workload distribution.

2.2 Scheduling

Memory and thread distribution is key to parallel programming and in OpenMP this is somewhat controlled using the scheduling clause. After experimenting with the dynamic sechduller, there was no speed up. The fastest time I could reach was 5.06s using a chunk size of 4. I believe this is because dynamic scheduling is more suited to uneven workloads based on the fact that chunks are handled on a first-come first-serve basis, however in my implementation almost every thread will have the same amount of computation.

The same test was repeated but with this time with the static scheduling clause. The best time was found using a chunk size of 16, giving a time of 3.9s. Static scheduling divides the iteration space up into the chunk size specified and at most one chunk is distributed to each thread in a round robin fashion. Clearly this is more suited to even workloads and thus should work better with this program. As the program is running on the 128x256 input file, the outer loop completes 256 iterations. It makes sense that 16 is optimum as to find the most efficient chunk size (assuming even workload) we can divide the #max_iters / #threads = Chunk size, 256/16 = 16 (I would expect 8 to be more efficient on the 128x128 input file). However, despite this, the fastest time 3.85s, came from not specifying the chunk size, which meant the iteration space was broken up into roughly equal size by the program at run time.

Interestingly, when the chunk size was small, for both the dynamic and static scheduling, the times were much higher. I think this is because as the chunk size decreases, the number of times a thread needs to fetch work from the work queue increases. Thereby increasing overhead and reducing performance.

Results

The incremental improvements have been documented throughout the report but a summary of the optimum timings and x fold speed increase, for parallel and serial, can be seen in figure 5.

	128x128	128x256	256x256
Serial	22.98s	46.34s	185.41s
Parallel	1.88s	3.38s	12.62s
xSpeedup	12.2x	13.7x	14.7x

Figure 3

The graph in figure 6 shows the comparison between the 128x128 (orange line) and 256x256 (blue line) grid inputs as the the number of cores/threads varies, with the dotted line representing perfect scaling. Both the blue and orange line scale linearly, which shows that the overheads in the OpenMP implementation also scales at the relatively linear rate. This could be a good and a bad feature. On the one hand, it shows that the code is very portable and could be used on nodes with a different number of cores (if cache sizes are similar). However, it might also suggest there is potential improvement, whereby the code runs more optimally when using more cores but has more overheads when that number reduces. It also shows that cache thrashing and workload distribution are well maintained even when varying the number of cores. Both lines are very close to perfect scaling with the blue line achieving 14.7x speed up when running on 16 cores with the 256x256 input. Interestingly, the orange line is below the blue line at almost every point. As the 128 grid has much less computation per iteration, it is more likely to diverge away from perfect scaling as it must create and destroy threads more frequently.

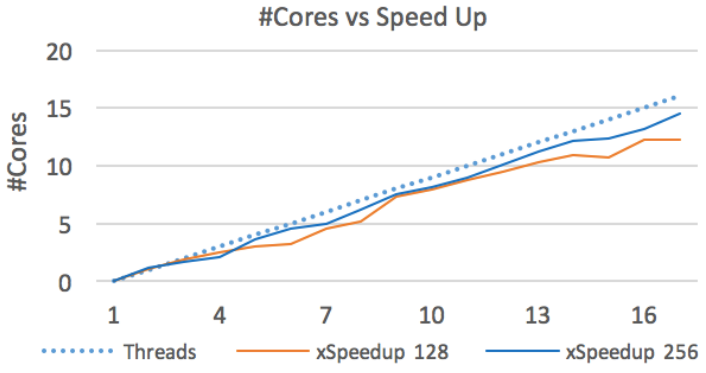

Figure 4

To conclude, the final time on the 256x256 input file was 12.62s. Overall compared to the original template code on the 256x256 file, my implementation achieved 67.75x speed up. In terms of optimising the code for parallel programming, according to Amdahl's law (assuming 99.9% of the computation was performed in parallel regions) the theoretical speedup is 1/((1-0.999)+0.999/16) = 15.8x. If I comparing this to my 14.7x, I have achieved 93% of the optimal parallel speed up.