José Hernández del Pino (jh3875@bristol.ac.uk)

Parallelising the Lattice Boltzman scheme

In this document we will present the various methods used to parallelise a reference C implementation of the lattice Boltzmann scheme.

The reference C implementation contains four functions: *accelerate flow, propagate, rebound* and *collision*, which are invoked one after the other for each time step of the simulation. At first glance these methods do not appear to easily lend themselves to parallelisation because they contain data dependencies that make it difficult to parallelise. In addition to this, the various functions also read continuously from shared read/write buffers. Inspection of the executable code under a profiler reveals that the program spends most of its time in the *collision*, *propagate, av_velocity and rebound*, functions, we will therefore concentrate our efforts to improve performance though optimisation and parallelisation in these areas.

%Time	Exclusive msec	Inclusive total msec	#Call	#Subrs	Inclusive usec/call	Name
100.0	45	2:29.862	1	20004	149862980	int main
89.1	98	2:13.530	10000	40000	13353	int timestep
60.1	1:30.110	1:30.110	10000	0	9011	int collision
27.5	41,146	41,146	10000	0	4115	int propagate
10.7	16,102	16,102	10001	0	1610	double av_velocity
1.3	1,918	1,918	10000	0	192	int rebound
0.2	256	256	10000	0	26	int accelerate_flow
0.1	157	157	1	0	157924	int write_values
0.0	25	25	1	0	25248	int initialise
0.0	0.004	1	1	1	1507	double calc_reynolds
0.0	0.684	0.684	1	0	684	int finalise

To obtain the greatest possible benefit from the parallelisation exercise, we must maximise the number of parallel regions in the code. In our case this amounts to merging some of the loops in the problem into one, a process known as loop fusion (Chapman, Jost, & van der Pas, 2007). We start by merging the rebound and collision functions, as they do not have data dependencies. The merged function distributes the workload more evenly amongst threads because the conditional statement within the inner loop now performs some computation when obstacles are both present and absent.

Before the propagate function can be merged with the previous two, its data dependencies must be removed. In its original form the propagate method pushes density values outwardly towards neighbouring cells and stores the intermediate result on a shared temporary buffer. Our version of the function will instead read values inwards from neighbouring cells, as shown in Figure 1, so that only the thread iterating over the current cell will ever write to the memory location belonging to that cell (Bella, Filippone, Rossi, & Ubertini, 2002).

Once the density propagation mechanism has been inverted, the *propagate* function can be easily merged with the *rebound* and *propagate* functions.

Next we turn our attention to memory optimisations noting that the combined *collision* function copies input data and results between source and temporary memory buffers unnecessarily. To avoid unnecessary movement of data in memory we can treat these two memory areas as simply source and destination, where source is a shared read-only area of memory containing all the input data and destination is the write-only shared area of memory where the end results are written to. E.g. instead of *source* →*temporary*→*source* we now simply have *source*→*destination*. Once a time step *k* has finished, all the program is then required to do is to swap the *source* and *destination* pointers before executing the next time step *k+1*.

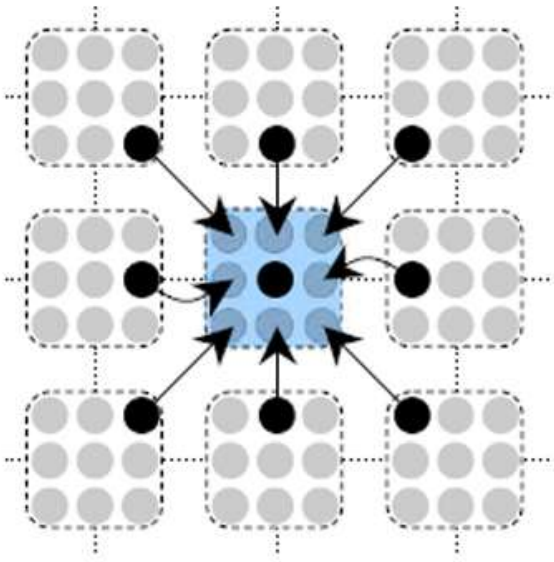


Figure 1. Inward density propagation.
Source:http://web.eecs.utk.edu/~dongarra/WEB-PAGES/cscads-libtune-09/talk11-williams.pdf

Temporary variables can also be eliminated from code within the inner loop of the collision function. I.e. when an obstacle is present, the new density values can be inverted and assigned directly to the target cell (propagate and rebound) without the need for copying to an intermediate temporary buffer.

Memory consumption can be reduced by changing the way the obstacle's buffer is allocated in memory. Originally defined to be "width * height * sizeof(pointer to signed integer)" in size, it can instead be defined to be "width * height * sizeof(unsigned char)". This effectively reduces the size of the obstacles buffer to an eighth of its original size. Please note that further compression can be achieved by using one bit per obstacle instead of the eight bits used by an unsigned char.

The final step of the serial optimisation process before adding *OpenMP* support, is to extract common elements repeated calculations from the collision step and perform the common calculation only once, storing the result in an intermediate variable to be used by the reworked expressions. At the same time we also replace division with multiplication in those expressions in order to speed up execution. From the algorithmic perspective, the extraction of common sub-expressions in the collision function can be interpreted as an optimisation devised to exploit the symmetrical properties of the directional velocity component calculations. E.g. the east and west directional velocity components, as do the other three pairs, both carry the same value but have opposite signs.

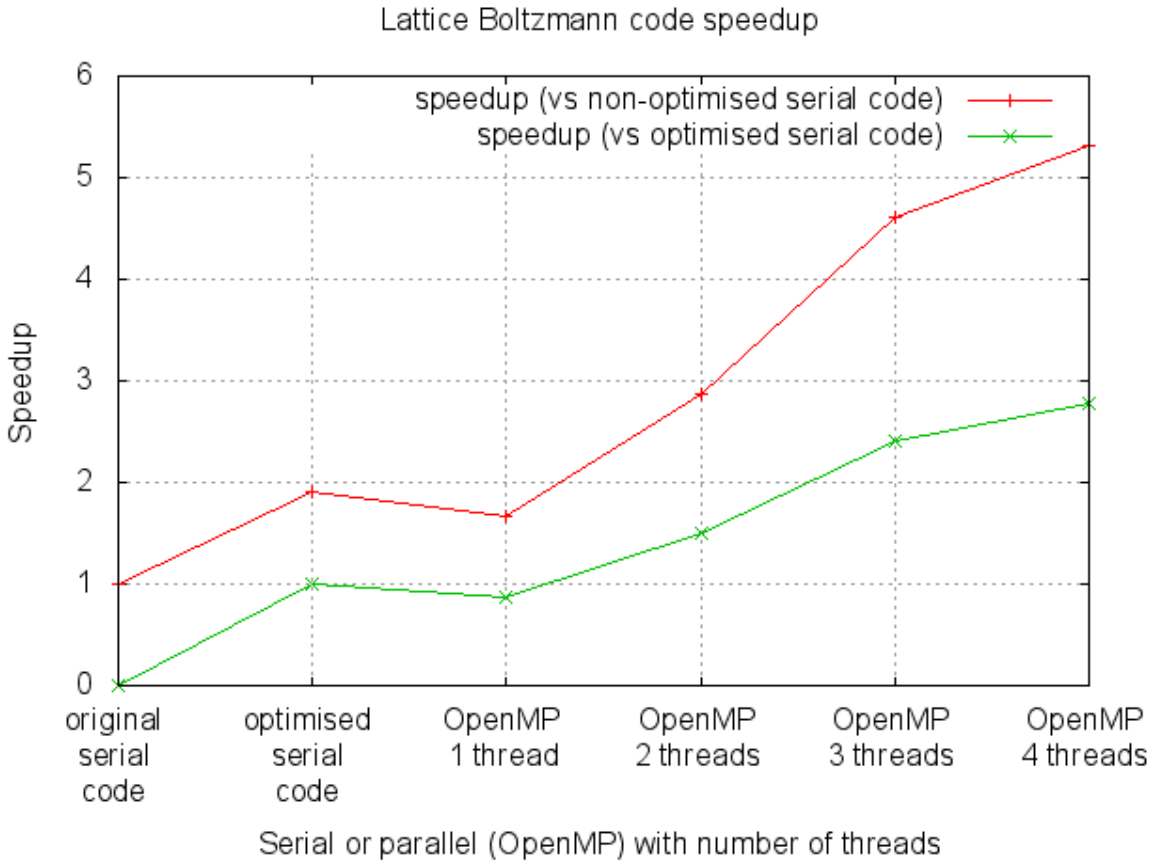
Once all the changes described above are in place, we add *OpenMP* pragma directives to a number of *for* loops in the program. Most notably, the combined *collision* function and the *av_velocity* function.

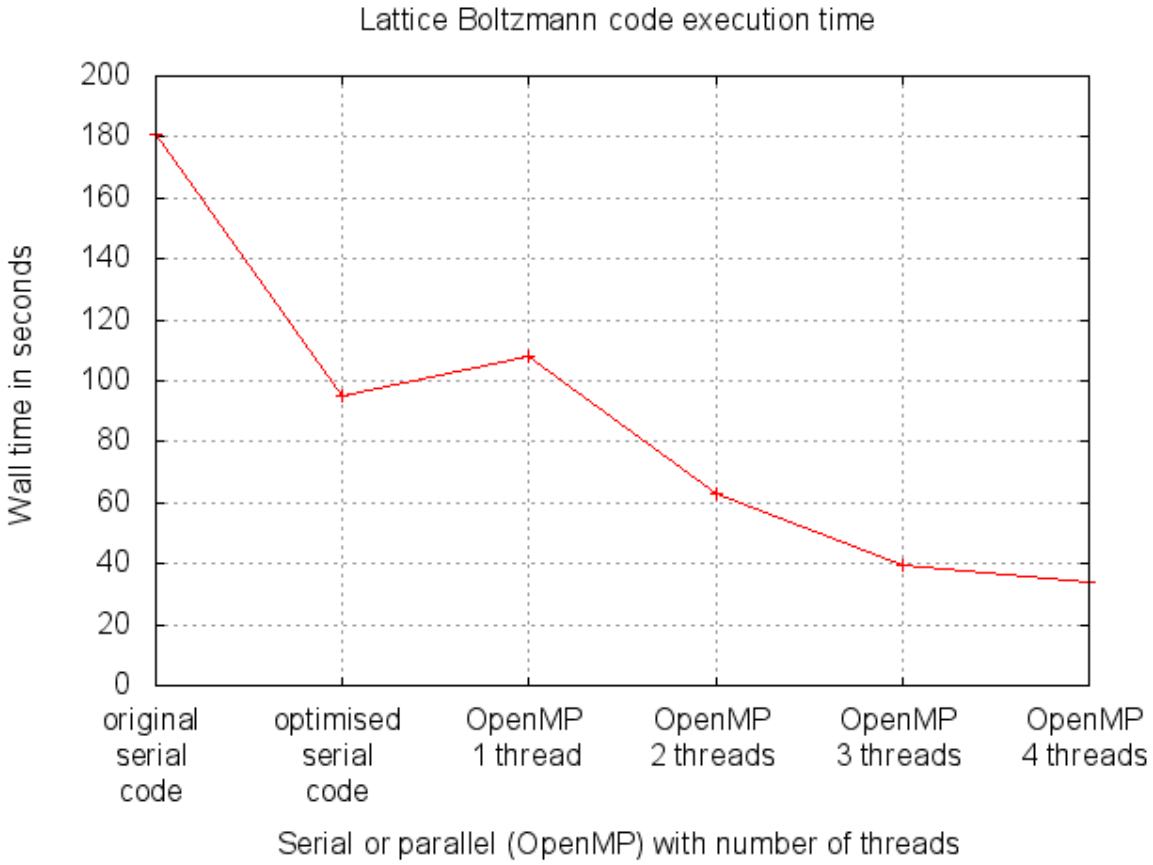
The following results were obtained by running the optimised/parallel software on BlueCrystal Phase 1 using one, two, three and four cores respectively. The results of running the original serial and the optimised serial version of the program have also been included so as to provide a basis for comparison and allow us to calculate the speedup factor.

Serial or parallel and number of threads (all using the -O3 flag)	Wall time execution in seconds averaged over 30 runs	Speedup against non-optimised serial code	Speedup against optimised serial code
original serial code	180.988		
optimised serial code	94.788	1.909	
parallel code 1 thread	108.077	1.675	0.877
parallel code 2 thread	63.156	2.866	1.501
parallel code 3 thread	39.292	4.606	2.412
parallel code 4 thread	34.071	5.312	2.782

Table 1. Execution time and speedup results

Our results showed a maximum discrepancy of 5^{-10} when compared to the reference results obtained by running the original unmodified program. The reader should note that different compilers (or even the same compiler using different flags) will produce different numerical results on the same machine.





Bibliography

Bella, G., Filippone, S., Rossi, N., & Ubertini, S. (2002). Using openMP on a hydrodynamic lattice-Boltzmann code. *Proceedings of the Fourth European Workshop on OpenMP*. Roma.

Chapman, B., Jost, G., & van der Pas, R. (2007). *Using OpenMP Portable Shared Memory Parallel Programming*. Massachusetts: The MIT Press.

The OpenMP® API specification for parallel programming. (n.d.). Retrieved February 17, 2012, from OpenMP: http://www.openmp.org