# Lattice Boltzmann method: from a serial optimization to a parallel version using OpenMP

Carlo Alessi
Student ID: ca16296
Student No: 1661349

## 1) Introduction

The arise of multiprocessors in the Computer industry gave a reason to develop sophisticated methods for parallel programming. The older generation of programmers wrote their parallel code using the pthread library. Nowadays, OpenMP, discussed in this report, provides high level constructs that help the programmers to focus on what matters.

The Lattice Boltzmann method is often used in fluids flow simulation. So, it is important to compute it fast.

The outline of this report is as follows. In section 2 I discuss the methods used for both serial and parallel optimization. In section 3 I report a small range of experiments showing the difference between serial and parallel performance. Section 4 concludes with a short discussion of the main results.

## 2) Method

The Methods described in this section are presented in a top-down manner. Here I refer to “main loop” as the for loop that dictates the number of time steps.

### 2.1) Serial Optimization

#### 2.1.1) Functions

I removed all the function calls and I put them in the main loop[1]. That operation saved the time spent for the *push* and *pop* operations of the stack frames relative to each function call. This led to a better reuse of variables as discussed later in section 2.1.4.

#### 2.1.2) Restructure of the code

I observed that the main functions of the time step went through the matrix several times when it was not necessary. I merged *accelerate_flow()* with *propagate()* in a for loop and merged *collision()*, *rebound()* and *av_velocity()* in another loop. To achieve this I had to add some '*if*' statement.

I found that adding branching was better than revisit memory more frequently than I needed.

#### 2.1.3) Loop unrolling

In *collision()* and *av_velocity(),* I unrolled the inner for loops needed to computing the *local_density* because I realized that the time spent computing *kk++* and *kk < NSPEEDS* was non-trivial compared to the amount of work required by the operations inside the body of the loops. The same considerations are applied to the *relaxation step* in *collision().*

#### *2.1.3) Access of the grid*

For each cell of the matrix, I stored the address of the first position of the *speeds* array in a pointer. The nature of the data structure led to a faster way to access the memory locations in two steps:

1) double* s = cells[ii].speeds

2) s[kk]

where the first step is done just at the beginning

1 This is not a good practice, because it makes the code hard to read. I would not do that if it was not a matter of performance.

of the iteration, and the second step is done each time that the speed values are needed.

#### 2.1.4) Constant values

The constant values, such as the square of the speed of the sound and the weighting factors were defined as macros as well as the values derived arithmetically:

```
#define C_SQ        (1.0 / 3.0)
#define W0          (4.0 / 9.0)
#define W1          (1.0 / 9.0)
#define W2          (1.0 / 36.0)
#define CSQ_2       (2.0 * C_SQ)
#define CSQ_2_CSQ   (CSQ_2 * C_SQ)
```

Other values decided at runtime but don't change throughout the program lifetime, such as the number of rows and columns of the grid, the omega parameter, were stored in global variables or placed outside the main loop:

```
int NX        = params.nx;
int NY        = params.ny;
int secondRow = (NY - 2) * NX;
double omega  = params.omega;
```

The values based on the *params.density* and *params.accel* are initialized once and for all in the *initialise() function and are defined as global variables.*

#### 2.1.5) Algebra and Logical operators

In the *rebound()* function, I computed the neighboring cells, taking into account that the grid was wrapped. The mathematical way to do it was with the modulo operator, but its implementation through division made it too expensive. I tried two different ways, and both were faster than the original code:
The first method was with a branching and the second with a bitwise AND:

```
1) (ii == NY - 1) ? 0 : (ii + 1)
2) (ii+1) & (NY-1)
```

The formula with the bitwise operation is valid only if the divisor is a power of two, so for the sake of generality, I used the first method.

#### 2.1.6) Compilers

I tried different compilers such as gcc version 5.0.0 and icc version 16.0.0. I discovered that my code ran slightly slower when I used gcc.

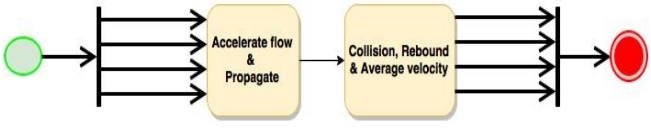
As the code developed throughout my experiments, I achieved better performances with one compiler than with another. I tried different compiler flags such as -Ofast, -O3, -funroll-loops, -march=native for the GNU compiler and -fast, -xHOST for the Intel compiler. Overall, the best performance was achieved with the Intel compiler with the -fast flag enabled.

### 2.2) Parallel optimization

For each iteration of the main loop, I spawned a team of threads and shared the work in *accelerate_flow()* and *propagate()*. I then shared the work in *collision()*, *rebound()* and *av_velocity()*, and then destroyed them.

The parallel region began just at the beginning of each iteration of the main loop and ended just before the update of the array *av_vels.*

We can represent the typical iteration with the following Activity Diagram:




Even though creating and destroying a team of threads in each iteration may be considered a waste of time, it is easy to implement and works quite well in practice, as shown in the results in

the next section.

I found a small improvement with the parallel performance when I started experimenting with environment variables. I exported *OMP_PROC_BIND=true* to bind each thread with a processor, so that they would not move in different cores as the program executed.

I used the SIMD (single instruction multiple data) clause to vectorize both loops. The compiler showed a warning stating that it was not able to vectorize, even though the code ran faster. The speedup might have happened because the compiler was still able to do some clever optimizations.

#### The first shared loop

In the first shared loop I achieved a small improvement using the collapse() clause, to parallelize the nested loops and create a single loop of length NxM, where N and M are respectively the number of rows and columns of the grid.
I found that it was better to use *#pragma omp simd* instead of collapsing the loop.

In both the shared loops, I put the *schedule()* clause. Experiments showed that the program ran faster with the *static* schedule, which assigned the iterations in a round-robin manner among all threads, at compile time. Other parameters, such as *dynamic*, *auto* and *guided* were significantly slower for this type of code, because they distributed the iterations at runtime.

I did not find a chunk-size of iterations that performed better than the default.

#### The second shared loop

In the second shared loop, the key part is the computation of the average velocity, I handled that problem using the OpenMP *reduction(Op:varList)* clause. Thus I put into a reduction the accumulators *tot_u* and *tot_cells*. That operation eliminated the problem of the synchronization of threads when they enter in critical sections and where they acquire and release locks. Put simply, with the reduction, each thread has its own copy of the variables listed in *varList*. Since, based on Op, their values are combined two by two after the work-sharing construct and the result is stored in the global variable, it means we are computing the average velocity in O(log n) time, where n is the number of threads.

I also put a *nowait* clause to avoid the implicit barrier at the end of the for loop, the correctness of the program is preserved because there is another implicit barrier at the end of the parallel region. Thus before update the *av_vels* array, the environment is consistent.

## 3) Results

In the sections below, first I give a quantitative measure of the optimization that I carried out, and then I investigate how the parallel program scales as more threads are added.

### 3.1) Speedup

Table 1 gives a description of the speed-ups achieved on different grids and with different compiler flags enabled, according to the formula below (Amdahl's law) :

$$Speedup = \frac{T1}{Tp}$$

where T1 is the execution time on a single processor and Tp is the execution time on multiple cores.

| Size | -fast | T1 | Tp | **Speedup** |
|---|---|---|---|---|
| 128x128 | enabled | 31.863 | 2.758 | 11.55 |
| | disabled | 96.388 | 7.324 | 13.16 |
| 128x256 | enabled | 64.016 | 4.901 | 13.06 |
| | disabled | 194.585 | 14.074 | 13.83 |
| 256x256 | enabled | 256.61 | 19.514 | 13.15 |
| | disabled | 781.37 | 56.47 | 13.84 |

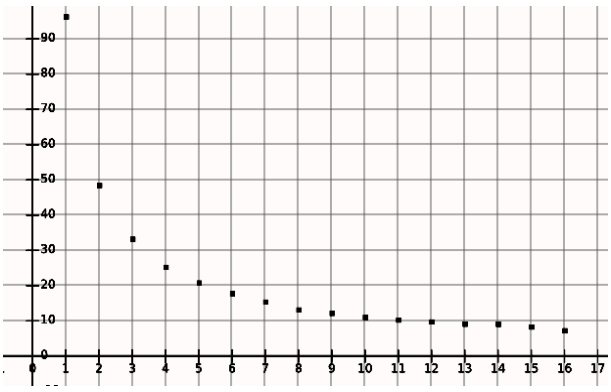
Table1: Description of the speed-ups. T1 and Tp are measured in seconds. The second column indicates whether the -fast flag was enabled or not.

Here we can see that, when the size of the grid grows, the parallel program scales well and we achieve a greater speedup.

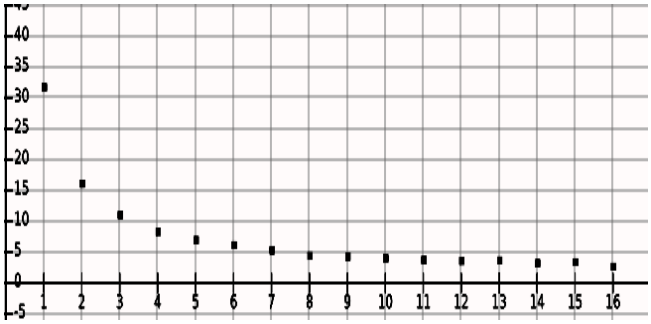
### 3.2) Experiments

I evaluated the effect of introducing more threads in the process from a single sequential flow of execution, to keep all the cores busy.

Figure 1 shows how the program behaves when we introduce more and more threads[2]. The horizontal axis represents the number of threads used and the vertical axis represents the elapsed time, expressed in seconds.




(a) -fast not enabled




(b) -fast enabled

Figure 1: Timing for different number of threads. The x-axis represents the number of threads, the y-axis represents seconds.

This is our ideal scaling line, where for twice the cores we go twice as fast. The log-scale behavior is repeated in both figures (a) and (b), although the overall time is different.

### 4) Concluding remarks

Several executions of the program gave an average timing of 18.323 seconds. If I use 256.61 seconds, from Table 1 (c), as a reference value for the serial time, I achieved a Speedup of 14. Also, never trust compilers.

2 I gathered this dataset when the elapsed time was an average of 19.46 seconds on the 256x256 grid with the -fast flag enabled. Data refer to the 128x128 grid.