Parallel Computing
Journal 2021
Dec 12, 2021

Parallelizing a Lattice Boltzmann simulation (flow over a cylinder) using OpenMP.

Saurabh Pargal

Department of Mechanical Engineering, Michigan State University,
East Lansing, Michigan, USA

Email: *pargalsa@msu.edu*

Abstract

Fluid over a cylinder is realized on a 2D grid using Lattice Boltzmann method on a D2Q9 lattice. Two cases are considered with grid nodes- (128X128) and (256X256) in x and y directions. The code i.e. the collision step, propagation step, initialization, and calculation of average velocity is parallelized using OpenMP. This paper will present the results on the code speed up with increase in number of threads, comment on the scalability and possible future improvements to improve performance.

NOMENCLATURE

β	Clauser parameter
δ	Boundary layer thickness
δ^*	Displacement thickness
ν	Kinematic viscosity
τ_w	Wall shear stress
θ	Momentum thickness
C_f	Skin friction
K	Acceleration parameter
R	Radius of curvature of the airfoil (suction side)

1 INTRODUCTION

Lattice Boltzmann method originated from lattice gas automata, is a type of CFD (computational fluid dynamics) method for flow simulations. Instead of solving for velocity and pressure, like in the case of other methods in solving navier stokes, equation, it computes particle distribution functions. Unlike other methods which solves macroscopic physics, it computes these distributive functions over a discrete lattice. It is sometimes also referred as discrete velocity Boltzmann method. This method essentially involves two steps, collision step and propagation step.

The collision step

$$f_i(\vec{x}, t+\delta_t) = f_i(\vec{x}, t) + \frac{f_i^{eq}(\vec{x}, t) - f_i(\vec{x}, t)}{\tau_f}$$

The streaming step

$$f_i(\vec{x} + \vec{e}_i, t+\delta_t) = f_i(\vec{x}, t)$$

Figure 1: Here 'f' is the probabibility distibution function of particles.

$$\begin{aligned}
\rho &= \sum_i f_i^{\mathrm{eq}}, \\
\rho\vec{u} &= \sum_i f_i^{\mathrm{eq}}\vec{e}_i, \\
0 &= \sum_i f_i^{(k)} \qquad \text{for } k = 1, 2, \\
0 &= \sum_i f_i^{(k)}\vec{e}_i.
\end{aligned}$$

Figure 2: Macroscopic variables calculated from distribution function

Both collision and streaming step are counted as 1 time step evolving in time. Here Bhatnagar Gross and Krook (BGK) model is used for relaxation to equilibrium via collisions between the molecules of a fluid. The model assumes that

the fluid locally relaxes to equilibrium over a characteristic timescale (τ_f). This timescale determines the kinematic viscosity, the larger it is, the larger is the kinematic viscosity. The macroscopic variables can be calculated from equilibrium and non-equilibrium distributions as shown in Figure 2.

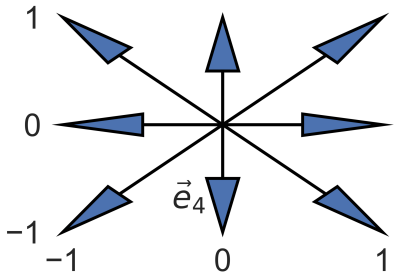

Figure 3: D2Q9 lattice for LBM method.

For case considered, flow is solved on a two dimensional grid, hence D2Q9 lattice is used for LBM method, as shown in Figure 3.

For the boundary conditions, we will have inlet, outlet and no slip boundary conditions at the wall. The wall boundary condition in LBM is simulated using 'bounce back' method.

Figure-4 shows the representation of fluid domain. There is generation of eddies following the obstacle (cylinder).

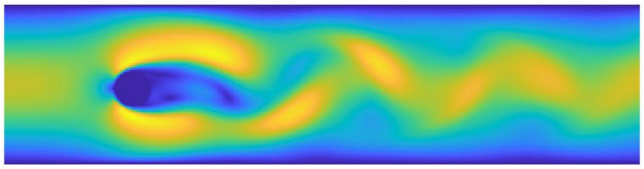

Figure 4: Flow over a cylinder (LBM solution from serial code)

Now for the given problem statement, objective is to parallelize the code using OpenMP. As indicated in previous literature, such as in Palabos benchmark (Figure 4), LBM is highly scalable.

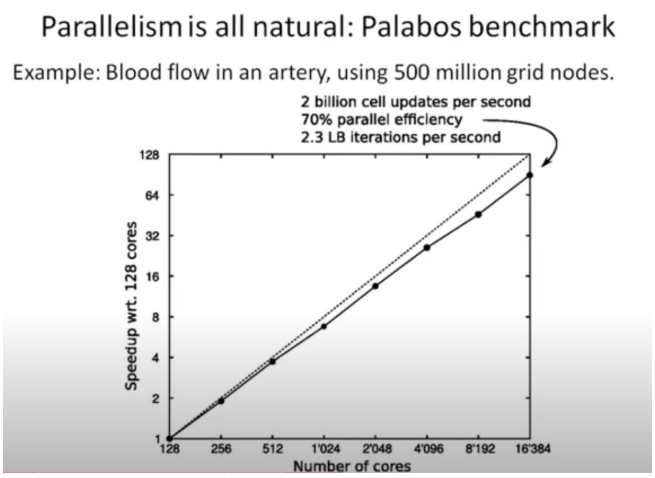

Figure 5: Palabos benchmark: Strong scaling in LBM method (Ref:Introduction to Lattice Boltzmann Method @ Nasa Glenn 2013)

Parallelization is important for such flow simulation, because with increasing grid nodes, we have lower errors and higher accuracy but more computational time. And since, several steps in LBM are local, which makes it highly parallelizable in comparison to other CFD codes. So these grid nodes, or physically speaking, the flow domain will divided across cores. So the objective is to parallelize the code using OpenMP, and check the speed up with increasing number of threads i.e. how strongly scaled.

Figure-6 shows the parallelizing the codes i.e. the simulation steps: Collision, propagation, rebound and initialization, which divides the lattices across cores.

Figure 6: Parallelized the simulation steps using '#pragma omp parallel' directive for the loops

Now, in the next section, the performance is compared for

two grid sizes- (128 X 128) and (256 X 256) with increasing number of threads from 1 to 32 threads.

2 RESULTS

For the results, two cases are considered Grid A- (128 X 128) nodes and Grid B- (256 X 256) nodes. Results are shown with and without 'O3' optimizer flag for compiling.

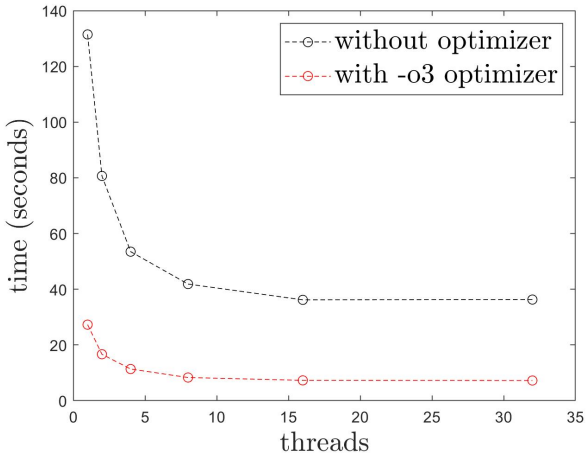

Figure 7: Grid A: Time vs threads

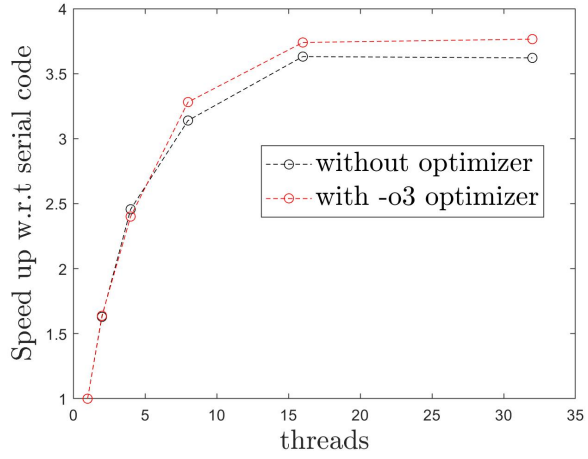

Figure 8: Grid A: Speed up vs threads

2.1 Grid A- 128 X 128 nodes

With 128 X 128 lattice nodes, the code was compiled with and without optimizer. With -03 optimizer, for the serial code, the time taken was almost reduced to 1/5th, going from 131.45 seconds to 27.3 seconds. Next for both the cases, number of threads are increased from 1 to 32, and saw a speed of approximately 3.6 times w.r.t to serial code, going from 27 seconds to 7 seconds approximately. But for both the cases, it asymptotically stabilized to that value with further increase in threads.

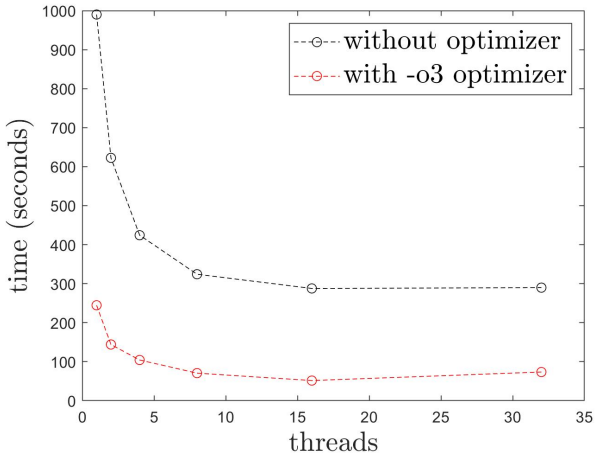

Figure 9: Grid B: Time vs threads

2.2 Grid B- 256 X 256 nodes

For grid-B, i.e. 256 X 256 nodes, the speed up with '-O3' optimizer was approximately 4 times, and the maximum speed was improved to 4.7 times with 16 threads, going from 244.7 seconds to 51.62 seconds. But with further increase in threads, the speed up actually reduced a bit to 3.5 times.

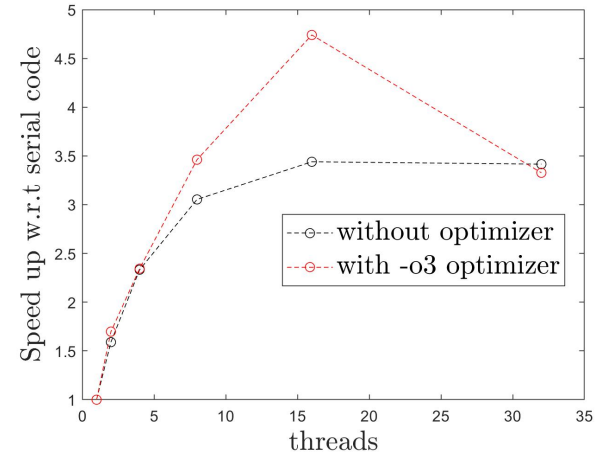

Figure 10: Grid B: Speed up vs threads

Cases	Grid A	Threads	Time (seconds)	Speed up	Time (seconds) without optimizer (-03)	Speed up
	1128 X 128	1	131.45	1	27.3	1
	2128 X 128	2	80.67	1.629478121	16.67	1.637672
	3128 X 128	4	53.5	2.457009346	11.37	2.401055
	4128 X 128	8	41.87	3.139479341	8.32	3.28125
	5128 X 128	16	36.2	3.631215547	7.3	3.739726
	6128 X 128	32	36.3	3.621212121	7.25	3.765517
	Grid B					
	7256 X 256	1	990	1	244.7	1
	8256 X 256	2	622.55	1.590233716	144.1	1.698126
	9256 X 256	4	424.3	2.333254773	104.4	2.34387
	10256 X 256	8	324.2	3.053670574	70.69	3.461593
	11256 X 256	16	287.8	3.439888812	51.62	4.740411
	12256 X 256	32	289.85	3.415559772	73.55	3.326988

Figure 11: Tabulated results for Grid A and Grid B

3 Discussion

The results are tabulated in Figure 11 Their is considerable speed up with respect to serial code to about 3.5 times for Grid A and 4.7 times for Grid B at maximum. But the scalability is not good. Some of the reasons or future improvements are discussed in this section:

3.1 Data level parallelism: Using #pragma omp simd

SIMD processing exploits data-level parallelism i.e. a set of vector elements are transformed at the same time i.e. single instruction can be applied to multiple data elements in parallel. Since data is copied from cells to temporary cells and vice versa for each time iteration, simd instruction can provide good speed up, by vectorizing the loop.

3.2 Grid copying

The whole grid is copied from cells to temporary cells, and vice-versa for each time step. These copy operations can be reduced, and data can be saved at larger intervals of time.

3.3 Reducing number of functions

For each function, the code has to iterate over grid again, i.e. each time for collision, propagation, acceleration, initialization steps. All of these functions can be combined into a single function, which will eventually reduce the number of times the code has to iterative over the grid.

3.4 Parallelizing write operation using reduction directive

The write operation can be parallelized as well, with #pragma omp parallel reduction(), in which total density is summed over, and calculating the velocities at each grid node. At the moment, this is leading to segmentation fault i.e. probably race condition. This can be a part of future work for further improvement.

3.5 Using hybrid parallelism: MPI and OpenMP

Palabos library for LBM, who are able to have strong scalability for their flow simulation, use a hybrid approach which includes using OpenMP with MPI. This can be beneficial for propogation step which is not necessarily local. Literature in Palabos library suggests that hybrid parallelization is needed i.e. using MPI with OpenMP, which will improve the parallelization capabilites in time and in propagation step. This can be the part of future work.

4 Compiling and running the code

The code can be compiled by - "gcc LBM.c -lm -fopenmp -O3". And then running the executable by ./a.out input.params cylinder.dat. Files-input.params and cylinder.dat includes the parameter, domain, and grid, which should be in the same directory as the executable.

5 References

References

[1] Prof. Bill Punch CMSE 822: Parallel computing notes CMSE 822: Parallel computing notes **F**all semester - 2021

[2] Palabos library- LBM https://palabos.unige.ch/ (University de Geneve) **2**020

[3] Introduction to Lattice Boltzmann Method @ Nasa Glenn 2013

https://www.youtube.com/watch?v=I82uCa7SHSQ **2**013

[4] ESPResSo Simulation Package https://www.youtube.com/watch?v=jfk4feD7rFQ **2**021

[5] Shakeel, Yusuf 521: Lattice Boltzmann Method https://www.youtube.com/watch?v=2YkVc3h8SHM **2**020