FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

COMPRESSIBLE LATTICE BOLTZMANN SOLVER
CPU BENCHMARK

MuCoSim WS 2023/24

PHASE: 1

Sudesh Rathnayake

Introduction

- Performance analysis of a OpenMP parallelize compressible lattice Boltzmann solver.
- Dwarfs → “Capture a pattern of computation and communication common to a class of important applications”[1]
- Seven Dwarfs
  1. Dense Linear Algebra.
  2. Sparse Linear Algebra.
  3. Spectral Methods.
  4. N-Body Methods.
  5. Structured Grids. → “Represented by a regular grid; points on grid are conceptually updated together. It has high spatial locality”’[1]
  6. Unstructured Grids.
  7. Monte Carlo.
- Lattice Boltzmann Method(LBM) → Commonly, LBM adopt a structured grid in practice.

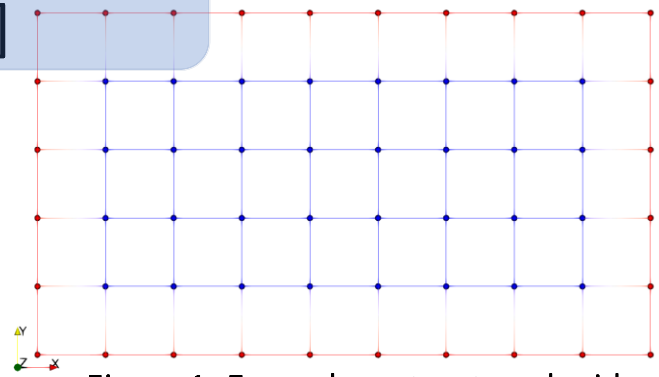


Figure 1: Exemplary structured grid

Lattice Boltzmann Method: Overview

- Two main approaches to simulate transport equations
  1. Continuum
  2. Discrete

- The numerical scheme for the Boltzmann equation ,
  - Quite simple, both to implement and to parallelize. (hyperbolic equation with force free formulation)

- Easy to apply.
  - For complex domains,
  - Easy to treat multi-phase and multi-component flows.

- Naturally adapted to parallel processes computing. [3]

- No need to solve Laplace equation at each time step as in FVM.

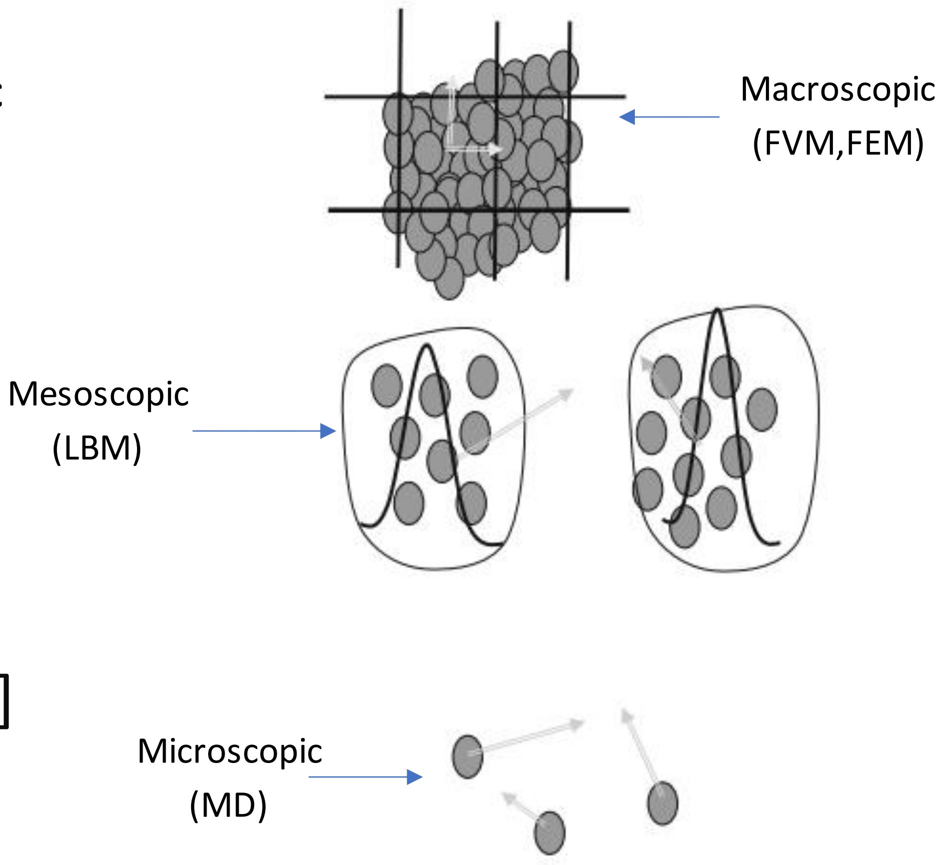


Figure02:Different scales in simulation.[2]

- The “Collide and Stream” algorithms.

$$f_i^*(x,t) = f_i(x,t) + \Omega_i(x,t) \qquad f_i(x + c_i\Delta t, t + \Delta t) = f_i^*(x,t)$$

- Collision operator: $\Omega_i \leftarrow f_i^{eq}$; Needs to be estimated

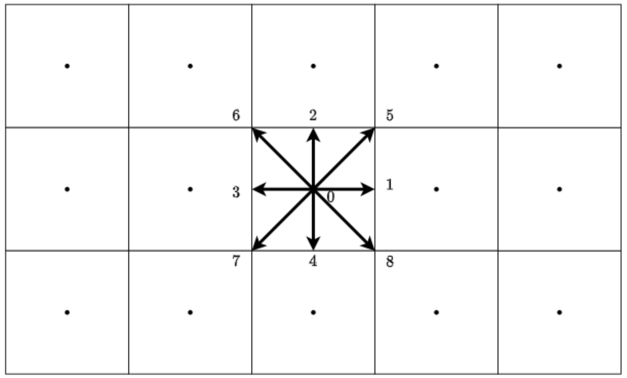


Figure 03:Lattice with D2Q9 stencil

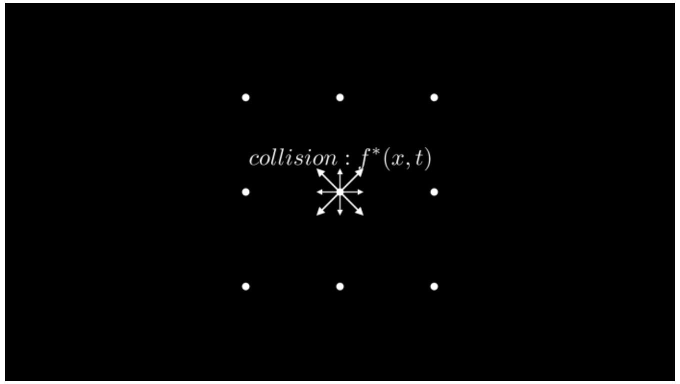


Figure 04: Collide and Stream

- Beauty, yet a weakness, of the LBM lies in its explicitness and uniform grid.
- Widely used for simulating incompressible fluid flows

- Compressible LBM $\rightarrow$ f_i^{eq} using discreate entropy function by formulating a minimization problem.

- 2 populations to integrate energy equation apart from the mass, and momentum equations.
- The present solver based on Entropic Lattice Boltzmann Method (ELBM)

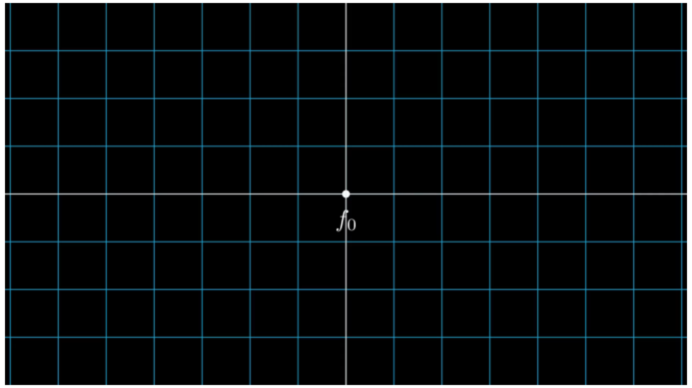


Figure 05: f and g populations in a lattice node

CLBM: Algorithm

**Algorithm : ** Compressible LBM algorithm for strong compressible fluid flows.

Data: Read computational grid and fluid properties;
Initialization: $\rho, \mathbf{u}, T$ and set temperature dependent weights;
Modification of c_i according to the shifted lattice velocity U;
Calculate: f^{eq};
Estimate Lagrangian multipliers $\leftarrow$ Newton-Raphson ;
Calculate: g^{eq} and Initialize g, f;
for $(int\ i = 0; i < Number\ of\ time\ steps; ++i)$
{
  Calculate $\rho, \mathbf{u}, T$ and set temperature dependent weights;
  Calculate time dependent relaxation times;
  Write .vtk files for post-processing;
  Calculate f^{eq};
  Estimate Lagrangian multipliers $\leftarrow$ Newton-Raphson;
  Calculate g^{eq};
  Calculate pressure tensor;
  Calculate quasi-equilibrium distribution g^*;
  Applying Knudsen number dependent stabilization;
  Collide;
  Reconstruction of f, g at grid nodes;
  Set boundary conditions;
  Stream;
  Swap f^{new} and f^{old};
}

Figure 07: CLBM algorithm

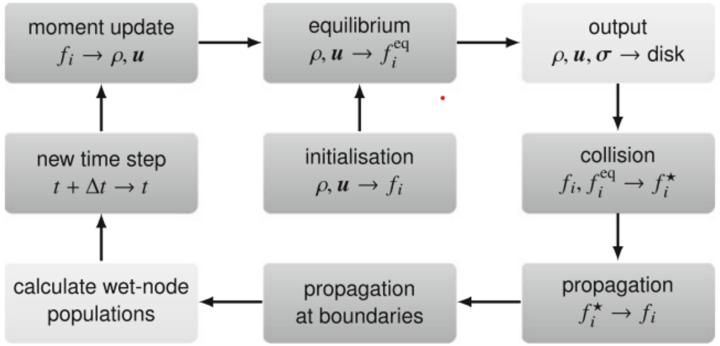


Figure 06: Standard LBM algorithm.[3]

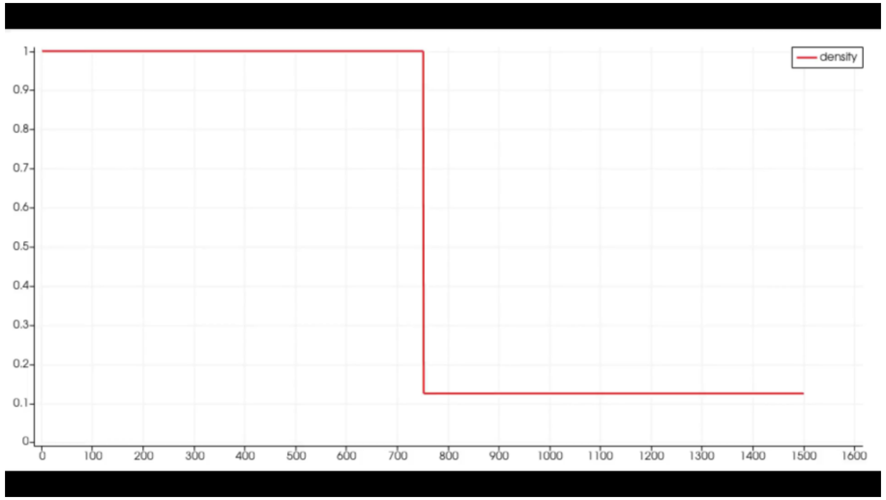


Figure 08: Results for density variation of air inside a 2D shock tube using CLBM solver

- Solver is based on C++ and OpenMP.
- Build using CMake.
- Input parameter file defines the simulation properties.
- For initial benchmarking and profiling, a 2D shock tube simulation is considered.

git@gitlab.cs.fau.de:yh54ojyn/mucosim.git

CLBM runtime profile

- A runtime profile in “Fritz”.
- Initially use gprof for the profiling.
- Vtune is preferred; much more insights.

CPU name: Intel Xeon Platinum 8360Y CPU @ 2.40GHz
CPU type : Intel Icelake SP processor
Number of lattice nodes: 2000 x 2000= 4000000 Lattice nodes.
Compiler: g++
Compiler Directives: -O3 –pg –march=native –mavx –ftree-vectorize

Required modules:
gsl (GNU scientific library)
Cmake
vtune

time	seconds	self seconds	calls	self (ns/call)	self(ns/call)	name
49.99	735.5	735.5				frame_dummy
20.57	1038.1	302.6				j33(node&)
4.83	1109.16	71.06				calc_parameters_from_PDF(grid&, double const&, int)
4.7	1178.29	69.13	55652400	1.24	1.24	j23(node&)
4.03	1237.63	59.35	55666385	1.07	1.07	j13(node&)
2.56	1275.27	37.64	27848884	1.35	1.35	j11(node&)
2.44	1311.15	35.87	27812635	1.29	1.29	func1(node&, double)
2.36	1345.86	34.71	27845283	1.25	1.25	j22(node&)
2.34	1380.24	34.38	27845516	1.23	1.23	func3(node const&, double)
2.05	1410.41	30.18	27852987	1.08	1.08	func2(node&, double)
2.04	1440.47	30.06	27855020	1.08	1.08	j12(node&)
1.97	1469.45	28.98	27833950	1.04	1.04	j21(node&)
0.13	1471.36	1.91				grid::grid(int, int)
0.02	1471.6	0.24				j32(node&)
0.01	1471.68	0.08				createFlags(grid&)
0.01	1471.76	0.08				swap(grid&)
0	1471.76	0	1	0	0	_GLOBAL__sub_I__Z7readingRNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEER4grid

Figure09: gprof hotspot results.

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

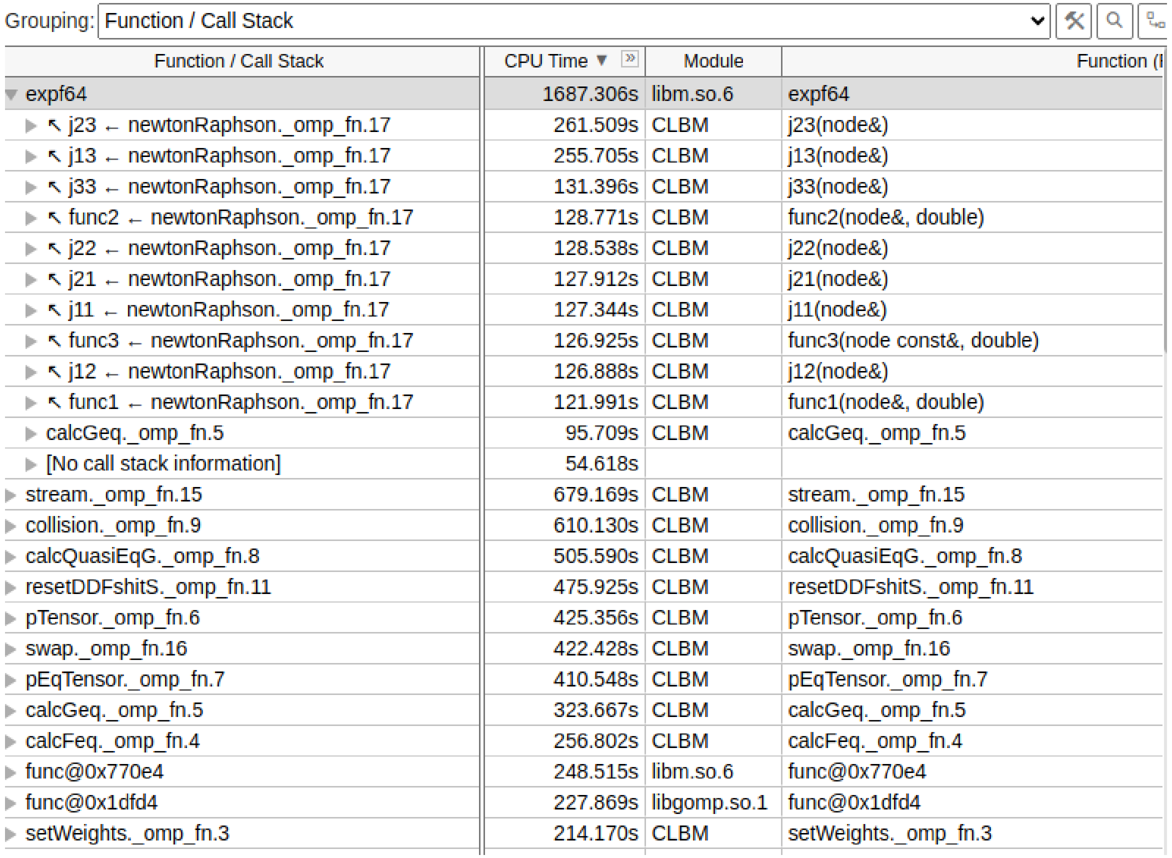
Function	Module	CPU Time	% of CPU Time
expf64	libm.so.6	1687.306s	23.2%
stream._omp_fn.15	CLBM	679.169s	9.3%
collision._omp_fn.9	CLBM	610.130s	8.4%
calcQuasiEqG._omp_fn.8	CLBM	505.590s	7.0%
resetDDFshitS._omp_fn.11	CLBM	475.925s	6.5%
[Others]	N/A*	3311.410s	45.6%

*N/A is applied to non-summable metrics.

Figure10: Vtune hotspot results.

CLBM runtime profile

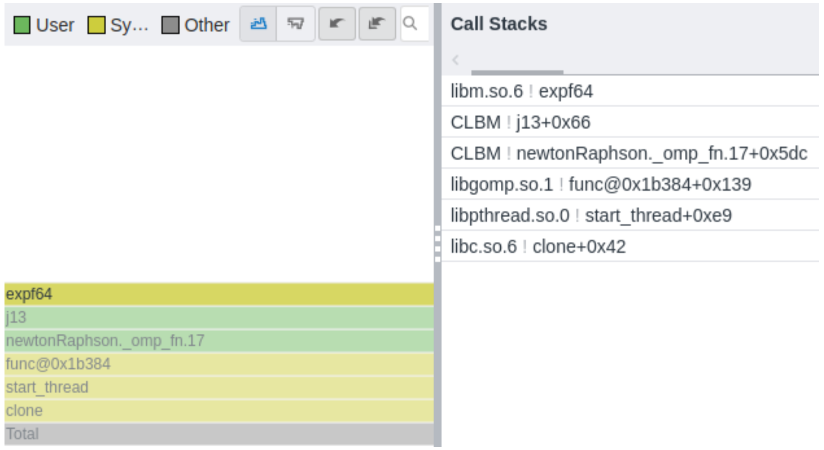
- High OpenMP overhead is observed for small data set.
- The newtonRaphson function is the most critical one in CLBM function.
- On top of that the calculation of exponential function using math.h library is critical.



Grouping: Function / Call Stack

Function / Call Stack	CPU Time ▼	Module	Function (I
expf64	1687.306s	libm.so.6	expf64
↖ j23 ← newtonRaphson._omp_fn.17	261.509s	CLBM	j23(node&)
↖ j13 ← newtonRaphson._omp_fn.17	255.705s	CLBM	j13(node&)
↖ j33 ← newtonRaphson._omp_fn.17	131.396s	CLBM	j33(node&)
↖ func2 ← newtonRaphson._omp_fn.17	128.771s	CLBM	func2(node&, double)
↖ j22 ← newtonRaphson._omp_fn.17	128.538s	CLBM	j22(node&)
↖ j21 ← newtonRaphson._omp_fn.17	127.912s	CLBM	j21(node&)
↖ j11 ← newtonRaphson._omp_fn.17	127.344s	CLBM	j11(node&)
↖ func3 ← newtonRaphson._omp_fn.17	126.925s	CLBM	func3(node const&, double)
↖ j12 ← newtonRaphson._omp_fn.17	126.888s	CLBM	j12(node&)
↖ func1 ← newtonRaphson._omp_fn.17	121.991s	CLBM	func1(node&, double)
calcGeq._omp_fn.5	95.709s	CLBM	calcGeq._omp_fn.5
[No call stack information]	54.618s		
stream._omp_fn.15	679.169s	CLBM	stream._omp_fn.15
collision._omp_fn.9	610.130s	CLBM	collision._omp_fn.9
calcQuasiEqG._omp_fn.8	505.590s	CLBM	calcQuasiEqG._omp_fn.8
resetDDFshitS._omp_fn.11	475.925s	CLBM	resetDDFshitS._omp_fn.11
pTensor._omp_fn.6	425.356s	CLBM	pTensor._omp_fn.6
swap._omp_fn.16	422.428s	CLBM	swap._omp_fn.16
pEqTensor._omp_fn.7	410.548s	CLBM	pEqTensor._omp_fn.7
calcGeq._omp_fn.5	323.667s	CLBM	calcGeq._omp_fn.5
calcFeq._omp_fn.4	256.802s	CLBM	calcFeq._omp_fn.4
func@0x770e4	248.515s	libm.so.6	func@0x770e4
func@0x1dfd4	227.869s	libgomp.so.1	func@0x1dfd4
setWeights._omp_fn.3	214.170s	CLBM	setWeights._omp_fn.3

Figure10: Vtune hotspot results.



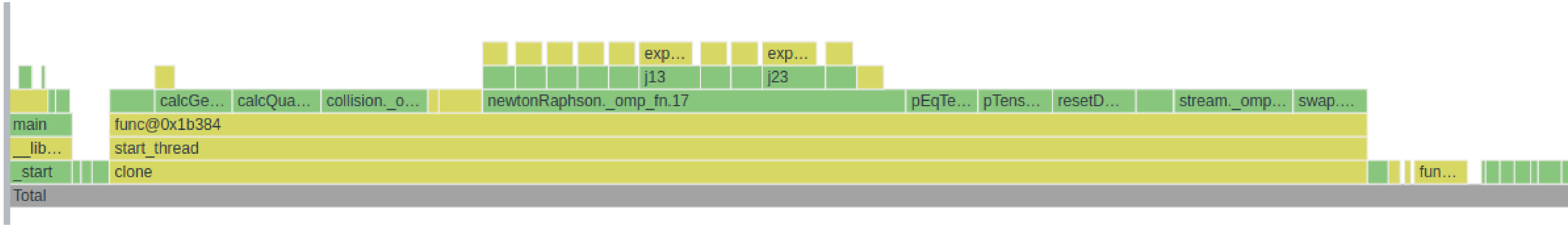


Figure09: Vtune hotspot flame graph.

Strong scaling

CPU name: Intel Xeon Platinum 8360Y CPU @ 2.40GHz
CPU type : Intel Icelake SP processor
Compiler: g++
Compiler Directives: -O3 –march=native –mavx –ftree-vectorize

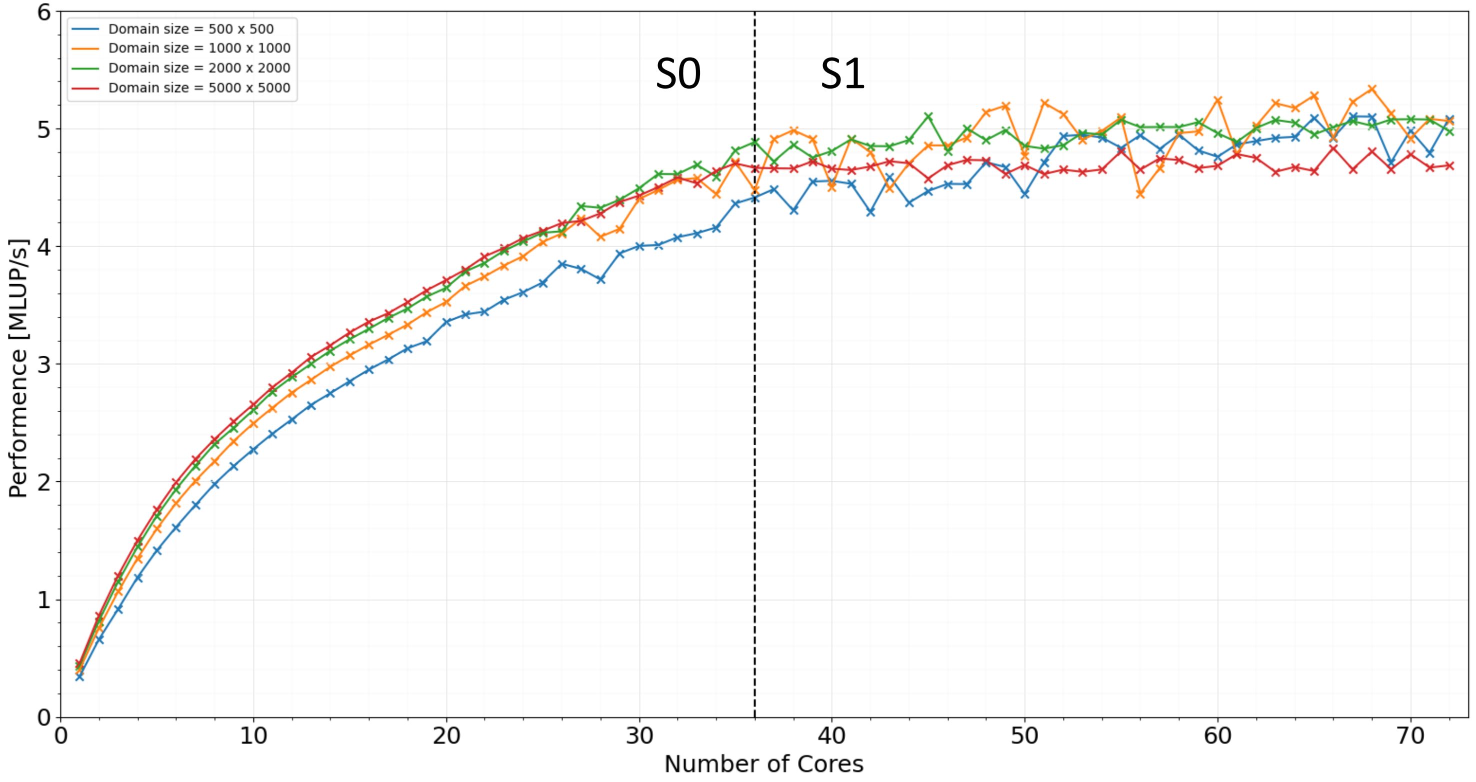


Figure10: Performance saturation in one node.

- 500 x 500 domain size does not have performance saturation.
- According to single core performance it fits to L3 cache.

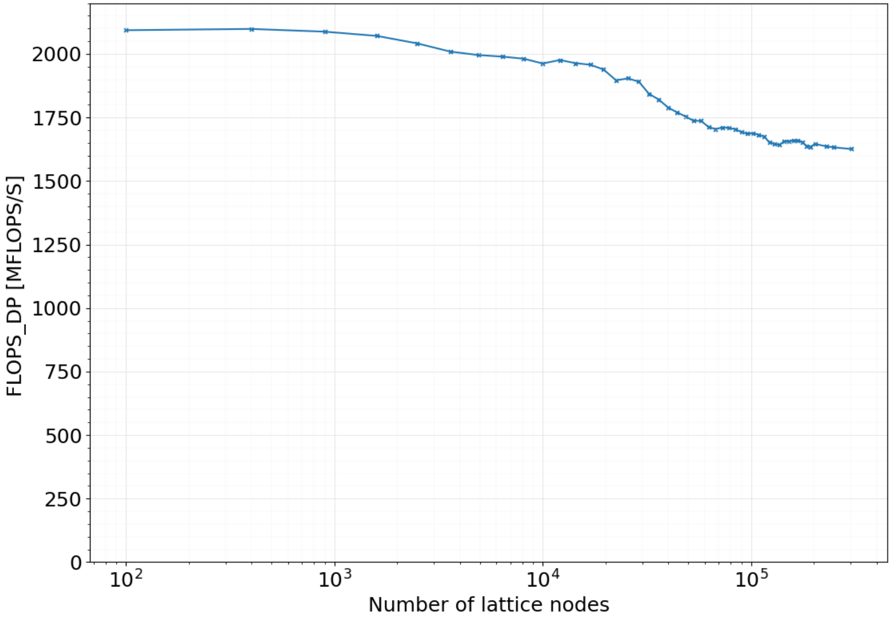


Figure 11: single core performance for fixed 2.4GHz.

- 2000 x 2000 elements seems to be good starting point for measurements.

Memory bandwidth

CPU name: Intel Xeon Platinum 8360Y CPU @ 2.40GHz
CPU type : Intel Icelake SP processor
Compiler: g++
Compiler Directives: -O3 –march=native –mavx –ftree-vectorize

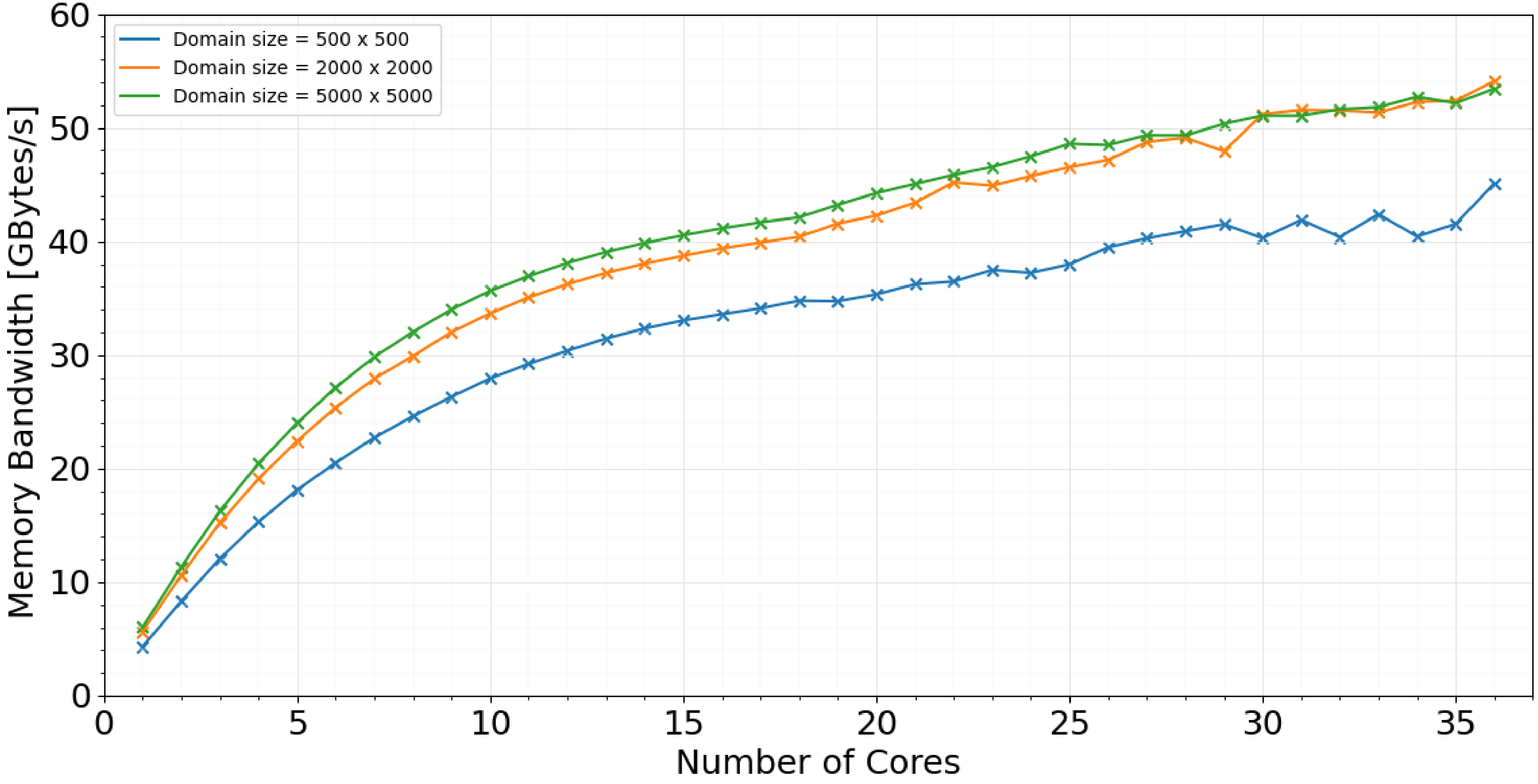


Figure12: Memory bandwidth saturation in first socket.

- Total memory bandwidth has a saturating trend in 1st NUMA domain.
- Generally, LBM solvers are known to be memory bound.[3]

Energy and power

CPU name: Intel Xeon Platinum 8360Y CPU @ 2.40GHz
CPU type : Intel Icelake SP processor
Compiler: g++
Compiler Directives: -O3 –march=native –mavx –ftree-vectorize

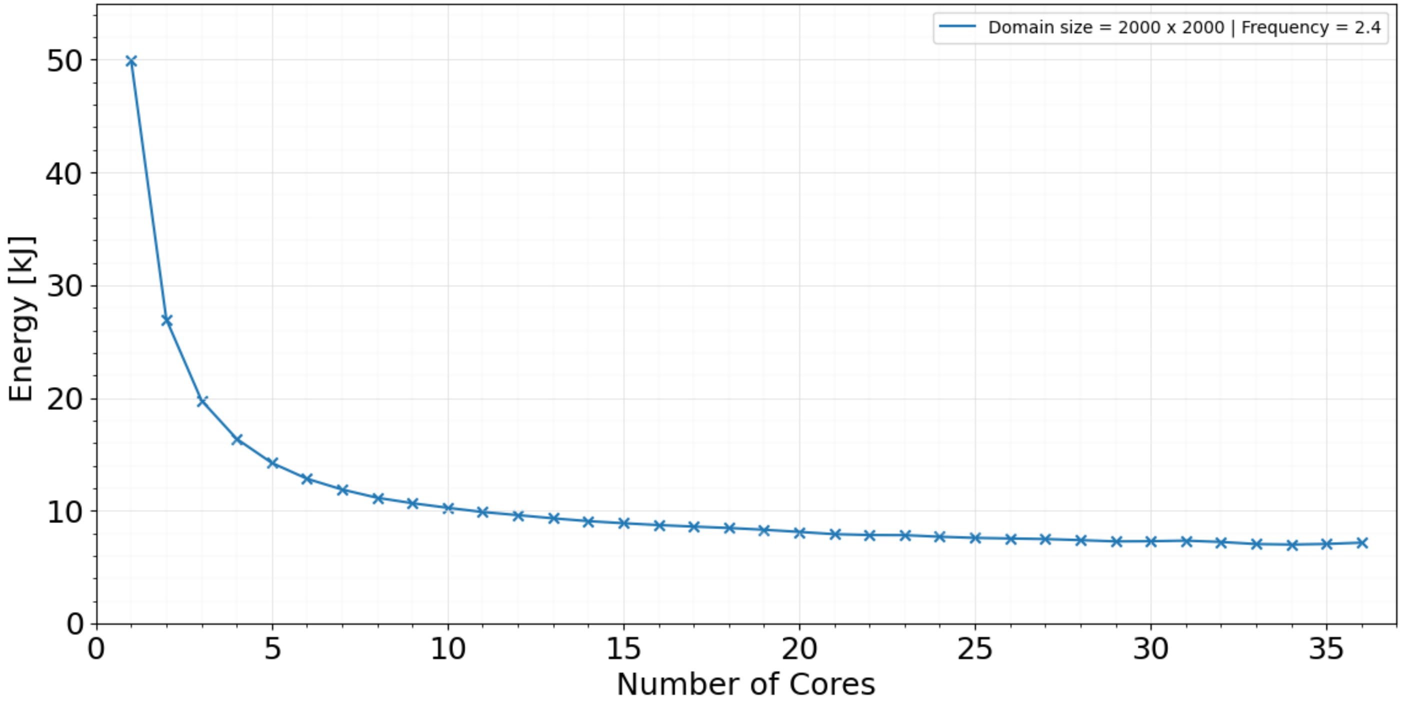


Figure13: Energy variation in first socket.

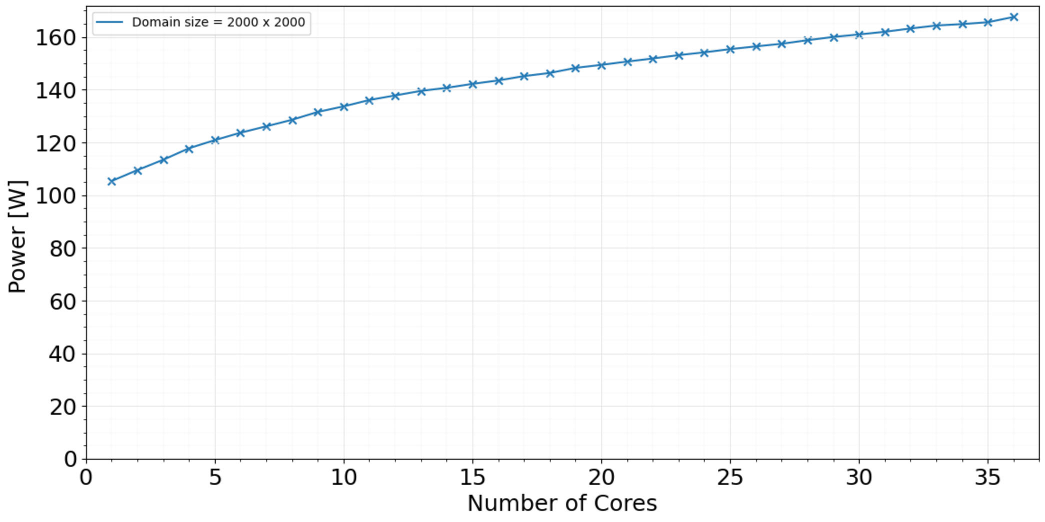


Figure14: Power variation in first socket.

- Total energy decreases with increasing core count.
- Power increase with increasing core count.

More on energy

- Even with the increasing frequencies total energy curves tend to meet at 36 core count.
- newtonRaphson function also has the same trend as the total energy.

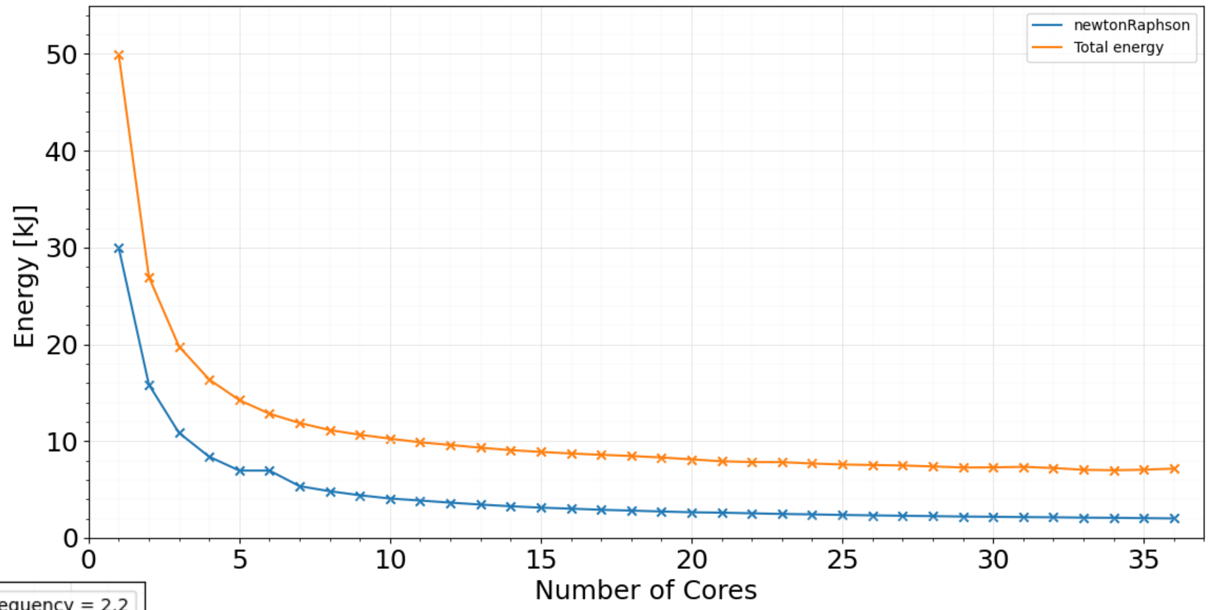


Figure15: Total energy for 2.4 GHz vs energy taken for the newtonRaphson function.

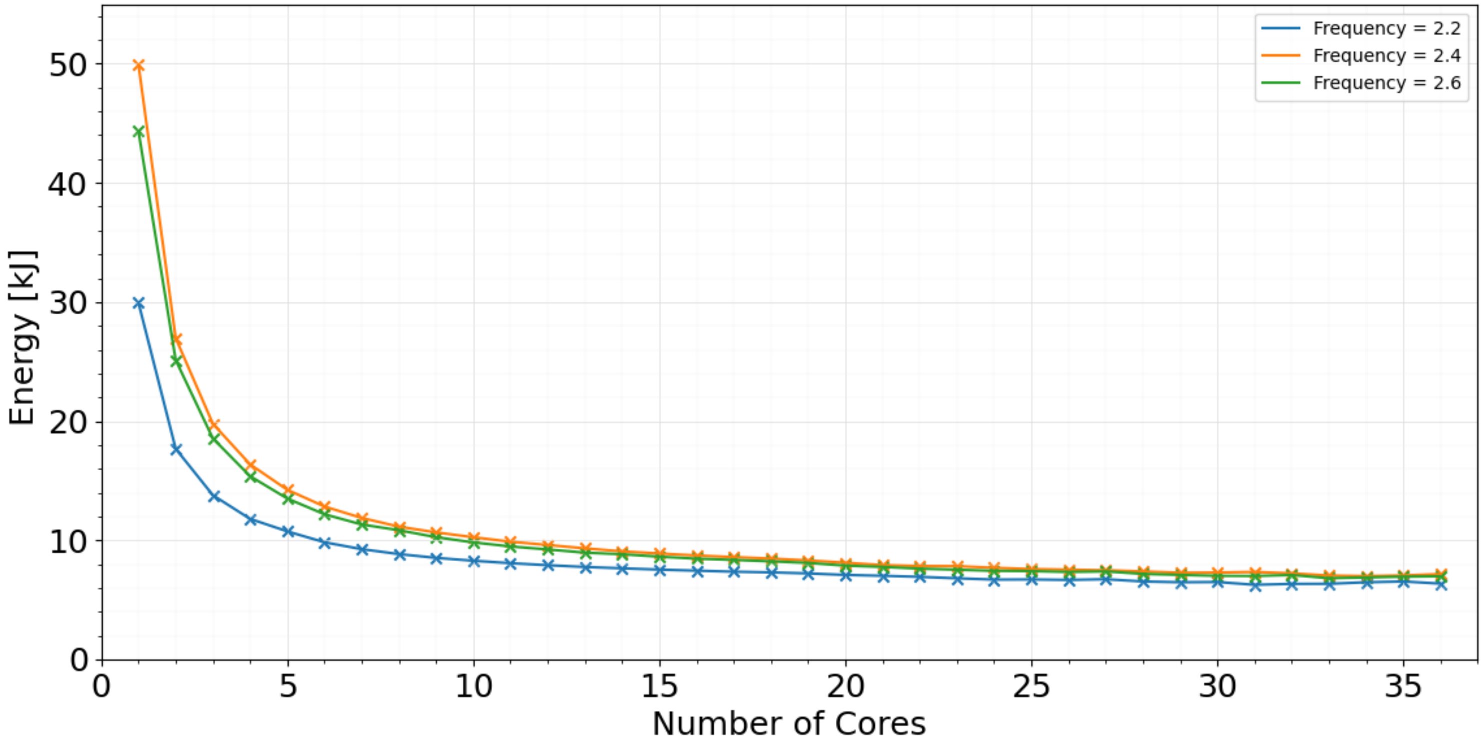


Figure16: Total energy variation for different frequencies for domain size of 2000 x 2000

Remarks and TODOs

- High OpenMP overhead can be observed according to the profiling results.
- Newton Raphson function is the most critical function in CLBM solver.
- According to the physics number of newton Raphson iteration not unique in each domain sizes.
- Solver is memory bound.
- Need to focus on parallelization.
  - Experiments with OpenMP scheduling
  - Barrier effect.
- Investigation of memory bandwidth in entire node.
- Investigation of the effect of different compilers.
- Benchmarking of 3D application.
- Investigation of optimal energy point for the present implementation.

Reference

1. **The Landscape of Parallel Computing Research A View from Berkeley,** 2006
   Electrical Engineering and Computer Sciences, University of California at Berkeley, *Electrical Engineering and Computer Sciences, University of California at Berkeley,*
2. A. A. Mohamad. **Lattice Boltzmann Method Fundamentals and Engineering Applications with Computer Codes**. Springer, 2019
3. Timm Krüger, Halim Kusumaatmaja, Alexandr Kuzmin, Orest Shardt, Goncalo Silva, and Erlend Magnus Viggen. **The Lattice Boltzmann Method**. Springer International Publishing, 2017.