Parallelization of the Classic Lattice Boltzmann method to solve transient heat conduction equations

Rudra Panch
IIT Madras

Rachit Kumar
IIT Madras

Raul Om Deepak
IIT Madras

August 18, 2024

1 Abstract

The investigation detailed in this paper centers on the utilization of the Lattice Boltzmann method (LBM) to address transient heat conduction problems, The LBM method has been widely used in the engineering industry, especially in the study of heat transfer problems like steady and unsteady heat flows and Fourier equations. Given its computational intensity, there exists ample opportunity for enhancement through parallelization strategies. This study focuses primarily on solving Fourier equations, with an emphasis on using OpenMP and MPI for parallelization, demonstrated through proficient C programs.

At the heart of LBM lies the application of the Boltzmann equation and the Bhatnagar-Gross-Krook approximation. The objective is solving the 2D transient heat conduction problem under various boundary conditions, including Dirichlet, Neumann, or Robin. Each boundary condition will be thoroughly examined and efficient parallel solutions will be devised. Additionally, verification of the obtained solutions will be done by referring to existing solutions provided in multiple research papers on this topic

In this paper, we delve into the potential of parallel computing paradigms, specifically OpenMP and MPI, to accelerate LBM simulations. OpenMP facilitates shared-memory parallelism, enabling efficient utilization of multicore processors, while MPI enables distributed-memory parallelism, catering to high-performance computing clusters. By harnessing both approaches, known as hybrid parallelization, we aim to achieve optimal performance gains.

2 Introduction

The Lattice-Boltzmann method is widely used in solving problems related to fluid dynamics as an alternate for the Navier-Stokes equation. At the atomic scale, the Newton's laws of motion are used to characterize the position and velocity of fluid particles and at larger scales of order like millimeters and micrometers, the molecular properties of the fluid are discarded and the Navier-Stokes equation is used to model the fluid. The interesting region is thee one in between where we cannot discard all the molecular properties but we can leave out some irrelevant ones and take useful information only. In this scale, the LBM method is used. Let us see how the equations shape out using fluid dynamics as the basis and translate into heat conduction.

3 The Lattice Boltzmann Model

Let us define $f(\vec{v}, \vec{x}, t)$ as the number of particles at position $\vec{x}$ at time t moving with velocity $\vec{v}$. The particle distribution function graph is given by the well-known Maxwell-Boltzmann distribution. Typical Maxwell-Boltzmann distribution function plots have been shown below for reference. From here follows the derivation of the Boltzmann equation.

$$\frac{df(\vec{v}, \vec{x}, t)}{dt} = \left(\frac{\partial}{\partial t} + \frac{d\vec{x}}{dt} \frac{\partial}{\partial \vec{x}} + \frac{d\vec{v}}{dt} \frac{\partial}{\partial \vec{v}} \right) f(\vec{v}, \vec{x}, t)$$

According to Boltzmann, this is equal to a factor called the collision operator $\Omega(f)$, which is a pretty difficult value to estimate and complex expansions for this were found by Boltzmann and many others in research papers. To make it easier for us to solve, we simplify this using the Bhatnagar-Gross-Krook Approximation.

3.1 Bhatnagar-Gross-Krook Approximation

According to the BGK approximation, the fluid, if left alone, will reach equilibrium after some finite time and at equilibrium, there won't be any collisions which implies the collision operator reaches zero. Hence, at any point of time, the collision operator can be thought of as a scaled difference between the current state and equilibrium. With this, we can write a simplified version of the Boltzmann equation.

$$\Omega(f) = \left(\frac{\partial}{\partial t} + \vec{v}\frac{\partial}{\partial \vec{x}} + \frac{\vec{F}}{\rho}\frac{\partial}{\partial \vec{v}}\right) f(\vec{v}, \vec{x}, t) = -\frac{1}{\tau}\left(f - f^{eq}\right)$$

$$\tau = \frac{\alpha}{v^2} + \frac{\Delta t}{2} \tag{1}$$

$$\tau = \frac{3\alpha}{v^2} + \frac{\Delta t}{2} \tag{2}$$

where, formula for $\tau(1)$ is for 1D approximations and $\tau(2)$ for 2D. These results can be found in the Chapman-Enskog Analysis.

3.2 Equilibrium Distribution

In the context of fluids, the equilibrium distribution considered is usually the Maxwell-Boltzmann distribution which is given by the function(3) mentioned below. In the case of heat transfer, we shall be using a different equation(4) which is also mentioned below.

$$f^{eq}(\vec{v}, \vec{x}, t) = \rho\left(\frac{1}{2\pi RT}\right)^{3/2} e^{-|v^2|/2RT} \tag{3}$$

$$f_i^{eq}(\vec{x}, t) = w_i T(\vec{x}, t) \qquad s.t. \sum_{i=1}^{N} w_i = 1 \tag{4}$$

Further,

$$T(\vec{x}, t) = \sum_{i=1}^{N} f_i(\vec{x}, t)$$

Therefore,

$$\sum_{i=1}^{N} f_i^{eq}(\vec{x}, t) = \sum_{i=0}^{N} w_i T(\vec{x}, t) = T(\vec{x}, t)$$

$$\sum_{i=1}^{N} f_i^{eq}(\vec{x}, t) = \sum_{i=1}^{N} f_i(\vec{x}, t)$$

which implies $\sum_{i=1}^{N} \Omega = 0$ which shows that the Lattice Boltzmann obeys the conservation laws

3.3 Lattice Structure

We divide the whole space of interest into small uniform grids/boxes based on our convenience. Boltzmann came up with this ingenious method pf constraining velocity to only a few certain directions and solving for those instead of taking the whole space. This makes it much more easier to visualize and computationally cheaper. In 1D, we only have 2 directions, +x and -x. In 2D however, we can have 9 directions. Each direction has a w of it's own. We represent all these directions in a matrix c of 2*9 size with 0s and 1s indicating the path

The lattice is usually represented as DNQM where N is the number of dimensions considered and M is the number of velocity directions allowed. We shall mainly be analyzing the D2Q9 model in this paper

3.4 Lattice Boltzmann Equation

With this knowledge, we can derive the Lattice Boltzmann equation as

$$\begin{aligned}\frac{df(\vec{v},\vec{x},t)}{dt} &= \Omega(f)\\ \frac{f_i(\vec{x}+c_i\Delta t, t+\Delta t)-f_i(\vec{x},t)}{\Delta t} &= -\frac{1}{\tau}\left(f_i(\vec{x},t)-f_i^{eq}(\vec{x},t)\right)\\ f_i(\vec{x}+c_i\Delta t, t+\Delta t) &= f_i(\vec{x},t)-\frac{\Delta t}{\tau}\left(f_i(\vec{x},t)-f_i^{eq}(\vec{x},t)\right)\end{aligned}$$

3.5 Boundary Conditions

In this paper, we shall mainly look at 3 different boundary conditions, Dirichlet, Neumann and Robin. We shall take 3 boundaries to be Dirichlet and investigate on the fourth boundary. Let us first formally state the 3 BCs.

$$\begin{aligned}\text{Dirichlet:} \quad & T(\vec{x},t) = T_c\\ \text{Neumann:} \quad & q(\vec{x},t) = q_c\\ \text{Robin:} \quad & q(\vec{x},t) = \alpha\left[T(\vec{x},t)-T_a\right]\end{aligned}$$

In all cases we assume an initial condition of $T(\vec{x},0) = T_i$ for some initial temperature T_i. Here T_c refers to a constant temperature maintained. Similarly q_c is a constant heat flux. T_a is the ambient temperature. α here is the heat transfer coefficient.
A plate of dimensions L*L shall be taken as the basis. The edges x=0, y=0 and y=L will be assumed to be Dirichlet with $T = T_0$. If we assume the x=L boundary to be Dirichlet at $T = T_L$, the temperature of all points on the x=L boundary remains constant and there is no need to calculate for it. Calculating the value of f on the boundary is difficult as we do not the situation of points outside of the plate.
Taking Neumann, the heat flux at the boundary is a constant value q_L and using the basic heat transfer formulas of flux, we can find the temperature value at the boundary to be

$$q_L = -\lambda\left(\frac{T_{n,j}-T_{n-1,j}}{\Delta x}\right)$$

$$T_{n,j} = T_{n-1,j} - \left(\frac{q_L\Delta x}{\lambda}\right)$$

We can find the temperature values at the boundary using the interior points. So, it is imperative to do this step at the end of every iteration. Finally, the Robin condition looks like this,

$$\begin{aligned}q(L,t) = \alpha(T_{n,j}-T_a) &= -\lambda\left(\frac{T_{n,j}-T_{n-1,j}}{\Delta x}\right)\\ T_{n,j} &= \left(\frac{\alpha\Delta x}{\alpha\Delta x+\lambda}\right)T_a + \left(\frac{\lambda}{\alpha\Delta x+\lambda}\right)T_{n-1,j}\end{aligned}$$

4 Application

Consider a slab of some material of thickness 0.01m at an initial temperature of 0°C. The relaxation time τ is assumed to be 0.1, which is a reasonable assumption considering real-world materials. It is an uniform slab with no heat generation. We split the plane into a 50*50 grid with grid size 0.0002m in both directions. We take a time step of 0.1s and calculate for different times until convergence.
For all cases, 3D temperature contours over the whole slab have been shown. To understand the working better, I have also provided 2D plots of the function variation with x for a fixed y at different times to understand the development of the contour over time.

In the Dirichlet condition, we take one edge of the slab to be kept at 0°C and the opposite edge at 100°C at all times. The contours were observed to remain the same after approximately 650s.

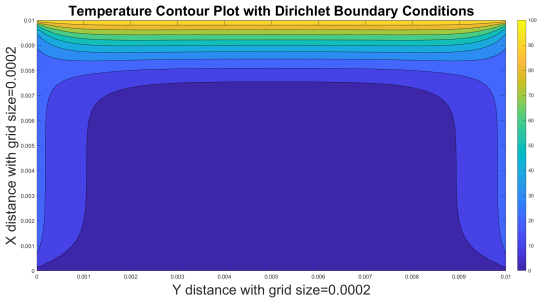

Figure 1: Temperature Contours at t=10s

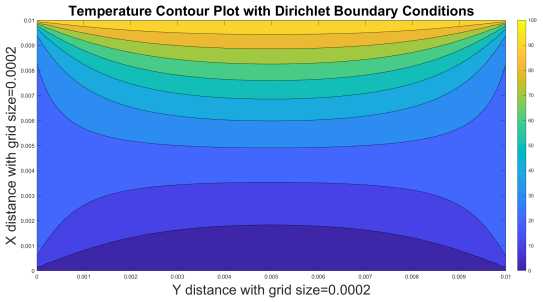

Figure 2: Temperature Contours at t=100s

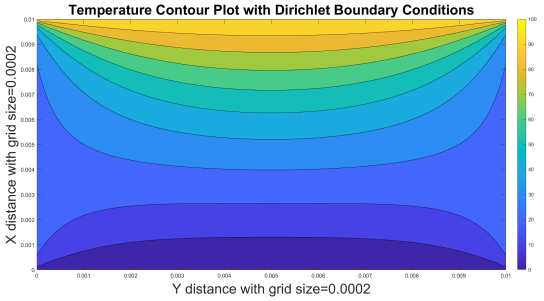

Figure 3: Temperature Contours at t=1000s

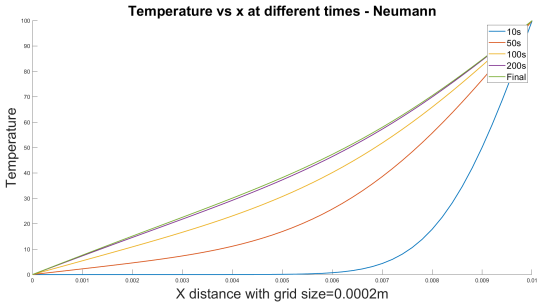

Figure 4: Temperature at various times

In the Neumann condition, we assume Dirichlet on one edge at 150°C and a constant flux $q = 1000W/m^2$ on the opposite edge. Here, the contours converged at around 1085s.

In Robin's case, again we assume Dirichlet on one end at 150°C and a temperature dependent flux $q = \alpha(T(\vec{x}, t) - T_a)$ on the opposite end. T_a here refers to ambient temperature and is assumed to 25°C constant throughout. α is taken to be $10W/m_2K$. The contours in this case, converged again around 1085s, almost the same as the Neumann case.

5 Parallelism Strategy

As we can see, our serial code is working perfectly well and we see that the Lattice-Boltzmann method efficiently solves these problems even in the smallest scale of millimeters. We shall now attempt to parallelize this code to improve it's time complexity. Also, we can increase the grid size and be able to obtain solutions in the same time frame. First let us look at the OpenMP implementation and then the MPI implementation

5.1 OpenMP Implementation

OpenMP provides a shared-memory parallelization technique. We need to analyze the code for loop carried dependencies before paralleling it.

- Looking at the formula, we can clearly see that the function value at any point in space at an instant of time depends on the function values of itself and all neighbouring points in the previous instant. This leads to a loop carried dependency over the time loop. Therefore, we cannot parallelize it.
- However, at an instant of time, to calculate the function value at any point, we require the neighbouring function values which leads to data dependency. But, this data having been calculated

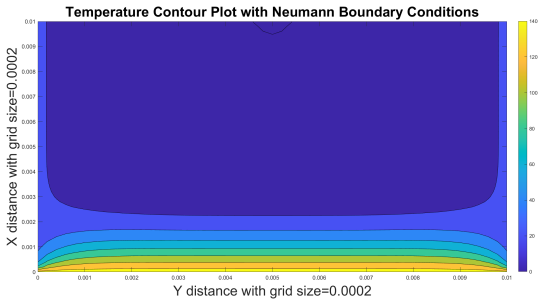

Figure 5: Temperature Contours at t=10s

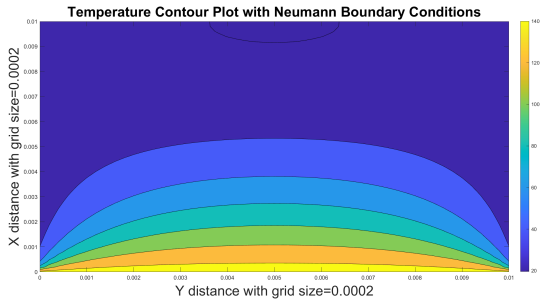

Figure 6: Temperature Contours at t=100s

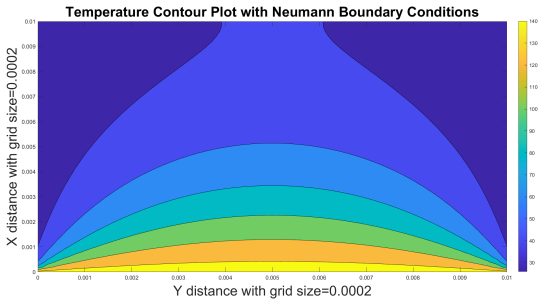

Figure 7: Temperature Contours at t=1100s

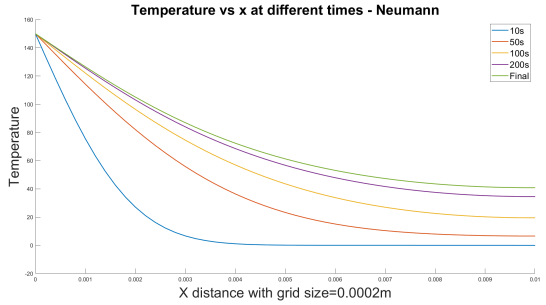

Figure 8: Temperature at various times

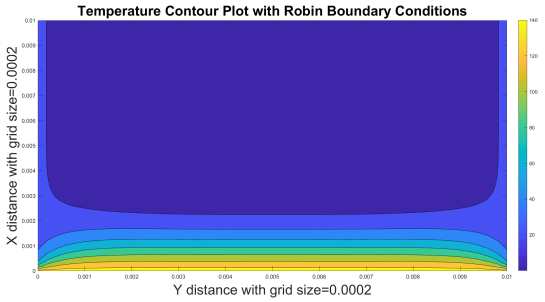

Figure 9: Temperature Contours at t=10s

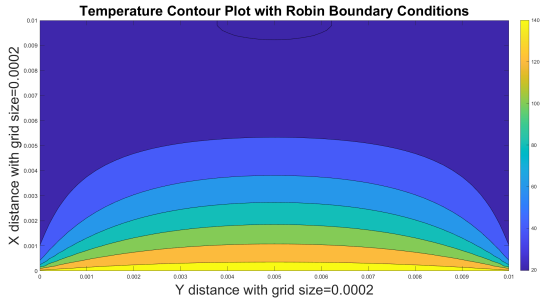

Figure 10: Temperature Contours at t=100s

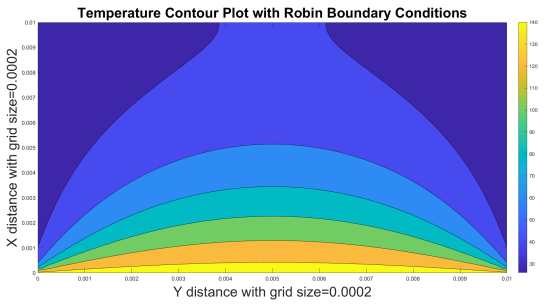

Figure 11: Temperature Contours at t=1100s

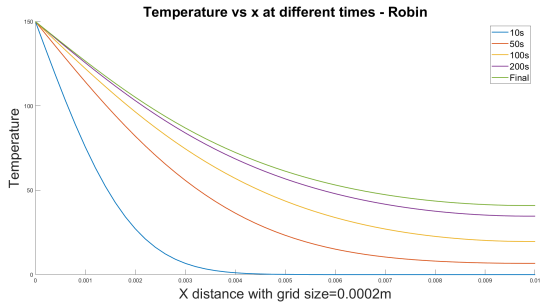

Figure 12: Temperature at various times

in the previous iteration is already available on the memory for all threads to access. Hence, the inner loop for calculating values at a certain time is parallelizable.

- Inside the time loop, we have 3 more loops. One on the x direction, one on the y direction and one for the 9 different directions of the LBM model. As, we need the previous values for calculation, we cannot update the new values into the same array, we rather make use of a temporary 3D matrix to store these values and then replace it after the end of all 3 loops.

- This looks like the Jacobi iterative method of solving the Lattice-Boltzmann equation. The Jacobi is known to be very efficient once parallelized and also has a good scope for parallelization.

- There is no need to parallelize the loop on 9 directions as it's range is small. We can parallelize at least one out of the 2 distance loops. Applying parallelism on any one of the loops leads to almost same level of efficiency. Using nested parallel by splitting a thread into many more threads, we can parallelize both loops. Depending on the grid size, this may or may not be useful. In our case, we shall be testing on a maximum of n=100, beyond which memory issues and space complexity issues shall start ruining the code execution. For this small grid number, it is not necessary to parallelize both the loops.

5.2 MPI Implementation

Let us now analyze the code and see if it can be parallelized using MPI, which is a distributed memory interface.

- We divide the whole grid into blocks, row-wise and each processor works on one of these blocks. This is sometimes also called row-block decomposition. We can also use other techniques like checkerboard decomposition and row-cyclic decomposition. The checkerboard is a complicated method and the row-cyclic is used when the work is not evenly distributed among rows. Our case does not require the use of these methods.

- As MPI is distributed memory, all processes do not have access to the updated values after every iteration. For the points lying on the boundary, this will cause a problem as they values which are being updated by a different processor. So, we need to have data transfer after every every time step

- We shall maintain values for these points in an array of points called ghost points. After every iteration, we shall receive updated values of these from the neighbouring processor. Also, the boundary lines of one process contains the ghost points for the next process and each process needs to send it's boundary lines for updating.

- This is done using a simple MPI Send/Receive statement. Each process shall first receive the ghost array from the previous process and send its boundary as well. Then it shall send the other boundary line to the successive processor and receive the corresponding ghost points from that processor. The exceptions are the first and last process which only send and receive from one side. The ordering of the Send/Receive calls have to be kept in mind in order to avoid hanging of the program due to infinite blocking

5.3 Hybrid OpenMP and MPI implementation

- Let us now study the scope of hybrid parallelization i.e, using both OpenMP and MPI simultaneously to solve the problem at hand. The performance of the hybrid version against the pure MPI/pure OpenMP has not been studied extensively and this would be a good chance to compare the three methods.

- As we know, OpenMP allows for incremental parallelization, which implies we can have sequential parallel and serial blocks of code. This is not possible in MPI though. This feature allows to run the code on MPI entirely while splitting into a few threads at some instants to solve particular blocks of code, wherever parallelization is necessary.

- MPI is a distributed memory interface and each processor gets it's own memory. Now, this personal memory is used to split into threads and all threads work on the same small memory using OpenMP, which is a shared memory construct.
- However, the Send/Receive commands which works on the Interconnect between the distributed memories should be done on the master thread only. This is where the Master directive of OpenMP comes in handy. #pragma omp master allows only the master thread to perform the enclosed operations.
- The communication of ghost points between processors is done in this manner. After all calculations have been completed, we use the master directive to perform the MPI Send/Recv between processors and update the ghost values

Listing 1: Hybrid OpenMP and MPI code

```
#include <stdio.h>
#include <mpi.h>
#ifdef _OPENMP
#include <omp.h>
#endif

int main(int argc, char** argv) {
    ...serial code
    MPI_Init(&argc,&argv);
    {
        // Initialize variables
        ...code run on distributed memory

        // Split into threads
        #pragma omp parallel for num_threads(thread_count)
            for(int i=0; i<n; i++){
            ....
            }

        // Send and Recieve ghost points on the master thread only
        #pragma omp master
        {
            MPI_Send(...)
            MPI_Recv(...)
        }
    }
    MPI_Finalize();
    ...serial code
    return 0;
}
```

6 Results

Both the implementations were worked out by coding them and checking the accuracy of the solutions obtained with the serial one and also their time of executions. The respective speed ups were calculated and plotted. The results of both the OpenMP and the MPI solutions are shown below

6.1 OpenMP Solution

The OpenMP code was run for a grid size of 0.0002m and 0.0001m as well. The final solutions obtained have been plotted. The Temperature contours and also the temperature variation on the midline have been plotted to show its similarity with the solution obtained from the serial code. I have concentrated more on the time of execution and speed ups obtained

6.2 MPI Solution

The MPI code as well was run for grid sizes of 0.0002m and 0.0001m. The final solutions which have been plotted again, are almost the same as those obtained from the serial code. Again, focus was more on the speed ups achieved.

6.3 Hybrid OpenMP and MPI Solution

The hybrid code as well was run for grid sizes of 0.0002m and 0.0001m. The final solutions which have been plotted again, are almost the same as those obtained from the serial code. We can conclude that all 3 codes are running perfectly and producing very efficient results. It is time to look at the speed ups they achieve.

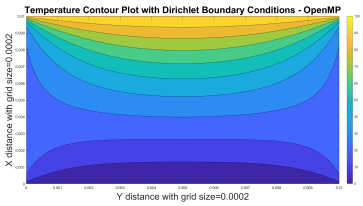

Figure 13: OpenMP - Dirichlet

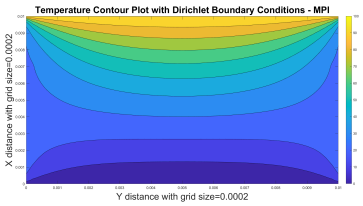

Figure 14: MPI - Dirichlet

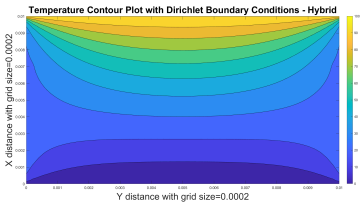

Figure 15: Hybrid - Dirichlet

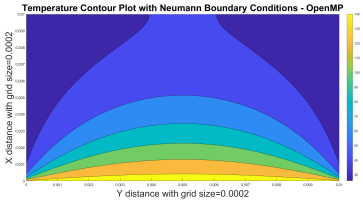

Figure 16: OpenMP-Neumann

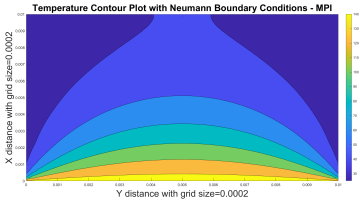

Figure 17: MPI - Neumann

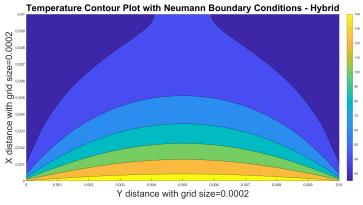

Figure 18: Hybrid - Neumann

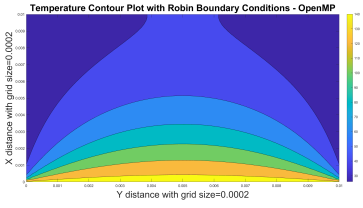

Figure 19: OpenMP - Robin

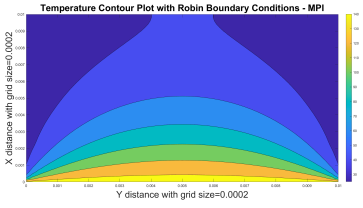

Figure 20: MPI - Robin

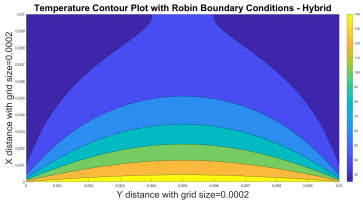

Figure 21: Hybrid - Robin

6.4 Speed Up And Execution times

The time of execution and the speed up achieved by all the different implementations on different boundary conditions and different grid sizes in tabulated below.

- The OpenMP code was run on 2 and 4 threads, the MPI one on 2 and 4 processes and the hybrid on 2 processes each splitting into 2 more threads and then on 4x4 distribution.
- As observed, for the grid size of 50, all of them perform pretty much the same. The speed ups obtained were around the 2.3 average mark with the hyrid code performing slightly better.
- When the hybrid code was run on 4x4 for gird size 50, it's performance dropped significantly. This is due to excessive parallelizations increasing the overhead on the code.
- In the case of a 100*100 matrix, the MPI code's performance went down. This can be due to the excessive MPI Send/Recv commands increasing the parallel overhead.

- The OpenMP code performed the best on the 100 sized square matrix with 4 threads, but with 2 threads, it did the worst. A possible conclusion can be that the time of forking the threads is not very high at small number of threads but the execution times itself increased due to the code being half as parallelised as previous.
- In the hybrid code, even though we split into multiple threads inside the distributed memory, only one of the threads(master thread) is assigned as the communicating thread. This leads to the other non-communicating threads being idle when the master thread is performing the MPI Send/Recv commands to transfer data on the interconnect.
- The 4x4 hybrid code is excessively parallelized and most of the threads being idle during a major part of code leads to this distribution not performing well when compared to the pure MPI or OpenMP codes.
- Generally, the code for Dirichlet condition was faster compared to the other two. This can be attributed to complex boundary point calculations in the Neumann and Robin techniques.

6.4.1 Serial Code

Grid Size	Boundary Condition	Iterations	Execution Time
50	Dirichlet	6732	6.077
	Neumann	10844	10.234
	Robin	10839	10.591
100	Dirichlet	25145	100.095
	Neumann	40084	160.611
	Robin	40069	145.630

6.4.2 OpenMP Code

Grid Size	Boundary Condition	Iterations	Run Time p=4	Speed Up	Run Time p=2	Speed Up
50	Dirichlet	6732	2.633	2.30	3.690	1.64
	Neumann	10844	4.474	2.28	5.891	1.73
	Robin	10839	4.177	2.53	6.022	1.75
100	Dirichlet	25145	38.487	2.60	64.658	1.54
	Neumann	40084	58.070	2.76	116.534	1.37
	Robin	40069	63.972	2.27	111.665	1.30

6.4.3 MPI Code

Grid Size	Boundary Condition	Iterations	Run Time p=4	Speed Up	Run Time p=2	Speed Up
50	Dirichlet	6730	2.655	2.28	3.477	1.74
	Neumann	10852	4.250	2.41	5.610	1.82
	Robin	10847	4.509	2.35	6.273	1.69
100	Dirichlet	25140	41.945	2.38	55.225	1.81
	Neumann	40099	80.608	1.99	89.668	1.79
	Robin	40084	77.571	1.87	89.081	1.63

6.4.4 Hybrid Code

Grid Size	Boundary Condition	Iterations	Run Time p=2x2	Speed Up	Run Time p=4x4	Speed Up
50	Dirichlet	6730	2.543	2.39	4.354	1.40
	Neumann	10852	3.992	2.56	8.188	1.25
	Robin	10847	4.107	2.58	7.298	1.45
100	Dirichlet	25140	52.386	1.91	52.166	1.92
	Neumann	40099	92.894	1.73	90.523	1.77
	Robin	40084	89.230	1.63	81.655	1.78

7 Conclusion

In this paper, we have parallelized the Lattice-Boltzmann method of solving transient heat conduction problems. The specific problem considered was that of Temperature variations and distributions in a 2D grid under different conditions, namely Dirichlet, Neumann and Robin boundary conditions. For the conditions taken, numerical values were assumed wherever necessary and a simple code was written to plot the contours over the grid at different time instances. The solutions obtained were consistent with the solutions provided in the research papers followed.

Once the solutions were obtained, we went ahead with parallelizing the code for better performance. We used 3 different implementations, the pure OpenMP code, pure MPI code and the hybrid OpenMp+MPI code. All three codes are working and solutions obtained were similar to that of the serial case. The time of execution of these codes were found to be much faster than the serial code. We also observed the problem of parallel overhead slowing down the code by significant amounts in the MPI code. The hybrid code, which has been studied very less was found to give accurate solutions but quite inefficient in this particular problem. Even with 4 processors splitting into 4 more threads, the speed up was comparable to that of just 4 threads or 4 processors.

All codes have been verified with plots and graphs, all run time values have been reported. Anaysis has done been done for variation with grid size and with number of processors/threads and also with different boundary conditions. The LBM is seen to be a very efficient method with a very high scope for parallelization and many possible applications in various fields, mainly in fluid dynamics and heat transfer problems.

8 References

1. Grażyna Kałuża, The numerical solution of the transient heat conduction problem using the lattice Boltzmann method, Scientific Research of the Institute of Mathematics and Computer Science, 2012, Volume 11, Issue 1, pages 23-30.

2. Lattice Boltzmann method applied to the solution of the energy equations of the transient conduction and radiation problems on non-uniform lattices Bittagopal Mondal, Subhash C. Mishra, International Journal of Heat and Mass Transfer 51 (2008) 68–82

3. Mixed Boundary Conditions for Two-Dimensional Transient Heat Transfer Conduction under Lattice Boltzmann Simulations. R. Chaabane†, F. Askri and S.B. Nasralla, Journal of Applied Fluid Mechanics, Vol. 4, No. 2, Special Issue, pp. 89-98, 2011.

4. Using the Lattice Boltzmann Method for the numerical study of non-fourier conduction with variable thermal conductivity. Ahmad Reza Rahmati *, Ali Gheibi, Journal of Heat and Mass Transfer Research 5 (2018) 1-9