Graduate School Form 30
Updated 1/15/2015

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Prateek Nagar

Entitled

TOWARDS A HIGH PERFORMANCE PARALLEL LIBRARY TO COMPUTE FLUID AND FLEXIBLE STRUCTURES INTERACATIONS

For the degree of Master of Science

Is approved by the final examining committee:

Fengguang Song
Chair

Luoding Zhu

Snehasis Mukhopadhyay

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): Fengguang Song

Approved by: Shiaofen Fang 4/8/2015

Head of the Departmental Graduate Program Date

TOWARDS A HIGH PERFORMANCE PARALLEL LIBRARY TO COMPUTE FLUID AND FLEXIBLE STRUCTURES INTERACTIONS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Prateek Nagar

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

May 2015

Purdue University

Indianapolis, Indiana

"Enjoyment of life is only possible if we could get connected to the Spirit"

I humbly dedicate my work to H.H. Mataji Shri Nirmala Devi who helped millions in achieving their self-realization through Sahaja Yoga.

ACKNOWLEDGMENTS

This work would not have been possible without continued support of my advisor Dr. Fengguang Song, who helped me in designing and implementing the LBM-IB software package. Also, I would thank Dr. Luoding Zhu for trusting in me to carry out the work on his algorithm and helping me in understanding the overall algorithm and assuring the correctness of the design. I would also like to thank Dr. Snehasis Mukhopadhyay who accepted my request to be a part of advisory committee and enlightening me with his valuable feedback.

I am so much blessed to have such a supportive family including my parents and elder brother, who always showed their trust in me and helped me to overcome all challenges by assisting me in every way possible. A special mention of my loving nephews "Yugansh" & "Krutine" whose presence relieved me from the mental stress and complexities of the problem and playing with them was a great source of recreation.

TABLE OF CONTENTS

Page

LIST OF TABLES vi

LIST OF FIGURES vii

ABBREVIATIONS viii

ABSTRACT ix

1 INTRODUCTION 1
- 1.1 Thesis Statement 2
- 1.2 Contributions 2
- 1.3 The Fluid-Structure Interaction Problem 3
- 1.4 Organization 4

2 BACKGROUND 6
- 2.1 Computational Fluid Dynamics 6
- 2.2 LBM-IB Method 6
- 2.3 LBM-IB Underlying Math 7
- 2.4 OpenMP 11
- 2.5 Pthread APIs 13
- 2.6 Hybrid MPI/Pthread Programming 14

3 LBM-IB SERIAL VERSION 17
- 3.1 Algorithm: Implementation Point of View 17
  - 3.1.1 Initialization 17
  - 3.1.2 IB 22
  - 3.1.3 LBM 26
  - 3.1.4 Regeneration Functions For Next Time Step 31
- 3.2 Underlying Data Structure 34
- 3.3 Performance Analysis 37

4 LBM-IB SHARED MEMORY PARALLEL VERSIONS 41
- 4.1 OpenMP LBM-IB Version 43
- 4.2 Performance Evaluation: OpenMP LBM-IB version 47
- 4.3 Pthread Version: Block Distribution 51
- 4.4 Performance Evaluation:Cube Based Block Distribution 63

5 LBM-IB HYBRID MPI/PTHREAD DISTRIBUTED MEMORY VERSION 68
- 5.1 Process/Machine Distribution 68

Page
5.2 MPI Extensions for LBM-IB 71
6 RELATED WORK 82
7 CONCLUSION 90
7.1 Future Work 92
LIST OF REFERENCES 94

LIST OF TABLES

Table		Page
3.1	Gprof Profiling of Serial LBM-IB on BigredII	39
3.2	Gprof Profiling of Serial LBM-IB on Dragon	40
4.1	Dragon System	63
4.2	BigRedII	64
4.3	Thog System	65
4.4	Node Distance between 8 Different NUMA nodes: using "$numactl - hardware''$	66
5.1	Process Distribution for $HybridMPI/PthreadLBM - IB$	70

LIST OF FIGURES

Figure		Page
3.1	Fiber-Sheet	18
3.2	Fluid-Grid	19
3.3	Streaming	27
3.4	Channels of Fluid-Grid	32
3.5	Data Structure for Immersed Boundary.	34
3.6	Data Structure for 3-D Fluid grid Serial Version.	35
3.7	Data Structure for GV Serial Version.	35
3.8	Experimental Set up	37
4.1	Shared Memory Model	42
4.2	OpenMP LBM-IB Performance Evaluation on BigredII.	48
4.3	OpenMP LBM-IB Performance and Parallel Efficiency on Dragon.	49
4.4	Profiling Results for OpenMP LBM-IB.	50
4.5	BlockDistribution : Cube Based Pthread LBM-IB version	53
4.6	Thread Grid Mapping	54
4.7	Modified Fluid Grid Data structure for Block Distribution	56
4.8	Datastructure Changes for Cube based Pthread LBM-IB	56
4.9	Influenced Domain of Fluid Nodes around a fiber-node	59
4.10	Streaming Boundary conditions for Cube Based LBM-IB	60
4.11	Cube Based Pthread Version Performance Evaluation	67
5.1	Lateral Distribution for MPI version	70
5.2	Global shared Data structure for MPI version	73
5.3	Streaming in Hybrid MPI/Pthread version	78

ABBREVIATIONS

3-D	Three Dimensions
2-D	Two Dimensions
BGK	BhatnagarGrossKrook equation
CFD	Computational Fluid Dynamics
DF0	Equilibrium Distribution Function g^0
DF1	Equilibrium Distribution Function g for current Time step
DF2	Streamed Equilibrium Distribution Function g from DF1 to be used in next time step
FSI	Fluid-Structure Interactions
FFT	Fast Fourier Transform
GV	Global Variable
IB	Immersed Boundary
LB	Lattice Boltzmann
LBM	Lattice Boltzmann Method
LV	Local Variable
NS	Navier-Stokes Equations
PDEs	Partial Differential Equations
TLB	Translation Look-Aside Buffer

ABSTRACT

Nagar, Prateek. M.S., Purdue University, May 2015. Towards a High Performance Parallel Library to Compute Fluid and Flexible Structures Interactions. Major Professor: Dr. Fengguang Song.

LBM-IB method is useful and popular simulation technique that is adopted ubiquitously to solve Fluid-Structure interaction problems in computational fluid dynamics. These problems are known for utilizing computing resources intensively while solving mathematical equations involved in simulations. Problems involving such interactions are omnipresent, therefore, it is eminent that a faster and accurate algorithm exists for solving these equations, to reproduce a real-life model of such complex analytical problems in a shorter time period. LBM-IB being inherently parallel, proves to be an ideal candidate for developing a parallel software. This research focuses on developing a parallel software library, **LBM-IB** based on the algorithm proposed by [1] which is first of its kind that utilizes the high performance computing abilities of supercomputers procurable today. An initial sequential version of LBM-IB is developed that is used as a benchmark for correctness and performance evaluation of shared memory parallel versions. Two shared memory parallel versions of LBM-IB have been developed using OpenMP and Pthread library respectively. The OpenMP version is able to scale well enough, as good as 83% speedup on multicore machines for ≤ 8 cores. Based on the profiling and instrumentation done on this version, to improve the data-locality and increase the degree of parallelism, Pthread based data centric version is developed which is able to outperform the OpenMP version by 53% on manycore machines. A distributed version using the MPI interfaces on top of the cube based Pthread version has also been designed to be used by extreme scale distributed memory manycore systems.

1 INTRODUCTION

Computational Fluid Dynamics (CFD) is an important branch of physics that provides various numerical methods for simulations of real-world problems. Its importance is further amplified by the fact that various numerical methods used in this domain, provide an underlying foundation for simulating critical scientific, engineering and life-science applications. For instance, solving intricate geometry for aerodynamics, numerical calculations for forecasting weather to achieve realistic visualizations, studying the behavior of a capsule inside a human body to predict its side-effects or benefits in areas of health science research, etc [2–4]. Fluid-Structure Interactions (FSI) is a very active and an ongoing research area in the CFD domain. These interactions are a part of daily life problems as well as used extensively in industrial, engineering and medical science applications. The work done in this thesis is based on solving similar interaction problems, where a flexible fiber-sheet is immersed in a fluid boundary and the changes in the fiber-sheet in response to the changes in the fluid properties are computed.

With high super computing abilities available today, it is highly eminent that efficient and correct software package exists to model such cognate numerical methods in a fast and efficient manner. It will be helpful in simulating the problems involving FSI in a faster way and thus provide better insight to change the underlying physics with much ease. This thesis aims at providing a parallel library to model such complex behavior that is solved using Immersed Boundary (IB) method, which internally uses Lattice Boltzmann Method (LBM) to model fluid solution. The work done as a part of this thesis provides a software library called LBM-IB developed using C as base language for all the versions of LBM-IB and the simulation experiments are carried out on different multicore and manycore architectures (Refer tables 4.1, 4.2 & 4.3).

1.1 Thesis Statement

The objective of this thesis is to design and develop a parallel shared and distributed software version of IB-LBM method proposed by [1] and to evaluate its performance on manycore architectures. This thesis aims at developing an efficient parallel software which utilizes the high performance computing capabilities of supercomputers procurable today. There are four basic versions of this software

- LBM-IB Serial Version : Implementation of the Algorithm proposed by [1]
- LBM-IB Parallel Shared Memory Version: OpenMP version
- LBM-IB Parallel Shared Memory Version: Block Distribution based Cubed Pthread version
- LBM-IB Parallel Distributed Memory Version: Hybrid MPI/Pthread version

1.2 Contributions

Problems involving CFD are omnipresent, therefore, it is eminent that a faster and accurate algorithm be used in solving these equations so that a real-life model of any CFD problem is reproduced in a shorter time period. The main contributions of this work are enumerated as follows:

1. This research presents the parallelizations of the numerical LBM-IB method for the first time. Other existing parallel libraries [5–14] solves these simulations in a different manner or in isolation of LBM and IB.
2. Two parallel shared memory versions of the serial versions are developed using OpenMP and Pthread library interfaces. The OpenMP version of LBM-IB scales in a very efficient manner with a speedup of as good as 83% on multicore architectures (for ≤ 8 cores).

3. In order to improve the data locality and degree of parallelism, a new data centric Pthread parallel library of LBM-IB has been developed which exploits the resources of manycore architecture in a better way. For large input and higher number of cores, this version of LBM-IB is able to outrun the OpenMP version by 53%. The same principle can be used in parallelizing other CFD sub-problems.

4. To exploit extreme scale distributed processing capabilities available today and further improve the level of parallelism, distributed version of LBM-IB which uses MPI interfaces on top of the powerful Pthread libraries has been designed for the first time.

Consequently, a new LBM-IB software has been developed with four versions. The sequential version ($1^{st}version$) is in itself the first of its kind and the parallel versions of OpenMP($2^{nd}version$) and cube-based design using pthreads ($3^{rd}version$) foretells that a parallel version of the same is very necessary to utilize the available computing power in full extent. Also, this project embarks the Distributed Memory Version of LBM-IB($4^{th}version$) computation which has not been done so far.

1.3 The Fluid-Structure Interaction Problem

FSI problem can be seen as an interplay between a flexible structure inside a fluid medium. The macroscopic property of the fluid such as the pressure, velocity etc. are responsible for causing microscopic structural changes in the immersed structure in the form of bending or stretching. This in turn influences the fluid boundary and macroscopic attributes of the fluid. This again causes further structural deformation in the structure and this process of interaction progresses with time and changes the initial state of the computational domain (comprising the fluid and structure) [15].

In this thesis a flexible 2-D sheet is immersed in a 3-D fluid grid in order to study interaction between the two (based on the algorithm proposed by [1]). This arrangement of flexible structure (fiber-sheet) inside a viscous fluid medium (3-D fluid

grid) is an example of FSI problem and is used in designing, development and testing of LBM-IB software(all 4 versions). The algorithm is built to support 3-D IB method. LBM "(D3Q19 model)" has been used as an underlying simulator for studying the interactions between a flexible fiber-sheet submerged into a 3-D fluid structure. In this project, the NS equations for IB are solved using LBM method unlike traditional approaches of FFT, projection methods etc [1].The implementation differs from the original problem stipulated in [1] on following points:

1. The flexible fiber-sheet is not tethered from the middle point.

2. The software is able to compute the location of the flexible sheet structure for every time step, but [1] also talks about"Drag Scaling" which is not computed but can be easily known by recording the fiber-sheets's position at every time interval.

3. The changes in Drag Scaling with the change in the structure's flexibility has not been analyzed.

1.4 Organization

This thesis is organized in the following manner. Following introduction in this chapter, background on CFD, LBM-IB algorithm with its mathematical formalism, basics of OpenMP, Pthread and MPI programming are described in the 2^{nd} chapter. Then the 3^{rd} chapter describes the Algorithm, Data structure being used and performance analysis of LBM-IB serial version, followed by the two shared memory parallel versions on OpenMP and Pthread with their Experimental results in 4^{th} chapter. Then, the hybrid MPI/Pthread version of LBM-IB with related design changes in the form of algorithms is described in 5^{th} chapter. This chapter is followed by details on the existing related work in parallelizations of LBM and IB and other parallel algorithms in chapter 6. Then, in chapter 7^{th}, the overall summary of the LBM-IB

software, challenges in the design for each version and the scope of optimization as a part of future work is described in brief.

2 BACKGROUND

2.1 Computational Fluid Dynamics

CFD is a sub-branch of fluid mechanics that deals with fluid or gaseous flows and their interactions with different structures that affect their properties directly or indirectly. The basic approach lies in solving various PDEs (mostly NS equations) that helps in identifying different attributes like pressure, viscosity etc. These equations are used in modeling the real time simulation of any CFD problem. Traditionally and even today, a CFD problem is sub-divided into following problem steps [16].

- Recognizing the physical boundary and the behavior of the fluid.
- Decomposing a bigger CFD domain into solvable minuscule domain. This step requires efficient use of super-computing abilities at disposal.
- Analyzing the output which is ultimately used in developing multifarious applications.

In the past, engineers used to develop a live model of the CFD problem, which apart from being time-consuming was also not reusable for any changes required in the simulation or change in the design. With the advancement in the field of computer science, the basic steps enumerated above are configured in the form of flexible software libraries to save money, time and achieve better simulation results.

2.2 LBM-IB Method

LBM-IB has been assuring and the most commonly used approach for simulating fluid flows and flexible structure interactions. It's widespread use in various applications makes it an appropriate choice to develop an acceptable and functional parallel

software. Immersed Boundary method is one of the most popular methods used in CFD. It was originated by Peskin [17, 18] and has revolutionized the computation of flexible structure's interaction with a fluid body thenceforth. The crux behind any IB method is to obtain a solution for a "viscous in-compressible fluid" [1]. In this project, a 2-D flexible sheet is considered to be submerged in 3-D Fluid structure. The sheet is made up of cross-section of horizontal and vertical fibers parallel to each other. The intrinsic fluid properties are computed using LB approach, which prefers simulation of fluid flow from the "mesoscopic" properties such as equilibrium distribution function g($\boldsymbol{x}$, $\boldsymbol{\varepsilon}$, $\boldsymbol{t}$) , over "macroscopic" ones like pressure and velocity [1]. The fluid flow is simulated by a 3-D regular structure made of evenly spaced fluid nodes with a spacing of a unit between them. The fluid provides the boundary influence to the immersed flexible sheet. Under the influence of fluid's flow, the fiber structure exhibits an elastic force from stretching and bending of the fibers along the width and height of the fiber-sheet. These forces in turn affect the fluid's properties such as the velocity, fluid mass-density ρ, velocity distribution function g. LBM exploits "single particle distribution function g($\boldsymbol{x}$, $\boldsymbol{\varepsilon}$, $\boldsymbol{t}$)" [1]. These helps in decomposing a bigger domain into a smaller domain and hence make it an ideal candidate for parallelism. The entire LBM-IB algorithm with an emphasis on LB simulations are explained in detail in the subsequent section. The next sections describes the mathematical equations that are used in LBM-IB and the implementation of the same in the software developed.

2.3 LBM-IB Underlying Math

BGK equation [19] forms the basis for most of the CFD problems that deal with FSI, which is given as:

$$\frac{\partial g(\boldsymbol{x},\boldsymbol{\varepsilon},\boldsymbol{t})}{\partial t} + \varepsilon.\frac{\partial g(\boldsymbol{x},\boldsymbol{\varepsilon},\boldsymbol{t})}{\partial x} + f(\boldsymbol{x},\boldsymbol{t}).\frac{\partial g(\boldsymbol{x},\boldsymbol{\varepsilon},\boldsymbol{t})}{\partial \varepsilon} = -\frac{1}{\tau}\left(g(\boldsymbol{x},\boldsymbol{\varepsilon},\boldsymbol{t}) - g^0(\boldsymbol{x},\boldsymbol{\varepsilon},\boldsymbol{t})\right) \quad (2.1)$$

The right hand side of the equation describes BGK approximation which is used as a modeling factor in LBM such as D3Q19, D3Q27, D3Q15 etc. It is known as

"complex collision operator". As the name suggests, it is used to identify the interaction of the immersed structure with the fluid via "single particle velocity distribution function" of the fluid and the force imparted by the immersed structure on the fluid nodes annotated by f($\boldsymbol{x}$, $\boldsymbol{t}$) in above equation. In simple terms, this force is the summation of the stretching and bending forces of the horizontal and vertical fibers in the fiber-sheet which is ultimately spread to the non-moving fixed fluid nodes, which in turns decides the location of the fiber-sheet for next time step. Also, gravitational force can be included in this force. The "macroscopic properties" of the fluid such as "fluid mass density ρ" and "momentum " can be easily derived from the "mesoscopic" g($\boldsymbol{x}$, $\boldsymbol{\varepsilon}$, $\boldsymbol{t}$), where $\boldsymbol{x}$ is the "spatial coordinate", $\boldsymbol{\varepsilon}$ is the "particle velocity" and $\boldsymbol{t}$ is the time [1]. The location "$\boldsymbol{X}$" of the fiber-sheet structure at any time instant "$\boldsymbol{t}$" can be derived from the velocity "$\boldsymbol{U}$" of the structure. "$\boldsymbol{u}$", velocity of the fixed fluid nodes and position of the fluid nodes "$\boldsymbol{x}$" are both used to determine the formal i.e. the velocity "$\boldsymbol{U}$" of the structure [1].

As described above, the success of any CFD problem lies in the way how it is decomposed in a smaller domain. [1] stipulates a way to decompose the above problem in a smaller domain using the D3Q19 model [20, 21]. It decomposes the aforesaid BGK equation on a cubic fluid structure made up of evenly space fluid nodes unit distance apart. This uniformity is also required in the distance between any two fiber-sheet node along the width and height. There is a relation between these two distances, if the distance between two fluid nodes is given by ΔD, then the two fiber-sheet node should be approximately $\frac{\Delta D}{2}$. The model used by [1] has been more effective in terms of the correctness and performance of the fluid flow simulation of the BGK equation [22]. In this approach, the fluid node's distribution function g($\boldsymbol{x}$, $\boldsymbol{\varepsilon}$, $\boldsymbol{t}$) is streamed in the neighborhood of 18 different nodes, along with recording the distribution function for itself, making a total of 19 different values for ε in a given

time instant as shown in Figure 3.3 [1]. The following equation 2.2 [1] is used to assign and stream the values of g($\boldsymbol{x}$, $\boldsymbol{\varepsilon}$, $\boldsymbol{t}$) for these directions

$$\varepsilon_i = \begin{cases} (0,0,0), & \text{i=0} \\ (\pm1,0,0),(0,\pm1,0),(0,0,\pm1), & \text{i= 1,2...6} \\ (\pm1,\pm1,0),(\pm1,0,\pm1),(0,0,\pm1), & \text{i= 7,8...18} \end{cases} \tag{2.2}$$

$\boldsymbol{\varepsilon}$ represents the direction along which the g($\boldsymbol{x}$, $\boldsymbol{\varepsilon}$, $\boldsymbol{t}$) is distributed. This distribution function in the next time step is derived from equation 2.3 [1].

$$g_i(\boldsymbol{x}+\boldsymbol{\varepsilon}_i,\boldsymbol{t}+1) = g(\boldsymbol{x},\boldsymbol{t}) - \frac{1}{\tau}\left(g_i(\boldsymbol{x},\boldsymbol{t}) - g_i^0(\boldsymbol{x},\boldsymbol{t})\right) + \left(1-\frac{1}{\tau}\right).w_i.\left(\frac{\varepsilon_i - \boldsymbol{u}}{c_s^2} + \frac{\varepsilon_i.\boldsymbol{u}}{c_s^4}.\varepsilon_i\right).f \tag{2.3}$$

w_i in the above equation represents the "weight " which is given by equation 2.4 [1]

$$w_i = \begin{cases} \frac{1}{3}, & \text{i=0} \\ 1, & \text{i= 1,2...6} \\ \sqrt[2]{2}, & \text{i= 7,8...18} \end{cases} \tag{2.4}$$

"$c_s = \frac{c}{\sqrt[2]{3}}$ is the speed of the sound used in D3Q19 model and c is the lattice speed i.e representing the sound for the fluid structure with respect to D3Q19 " [1]. As [1] identifies, equations 2.5 & 2.6 are used to compute the "macroscopic properties" such as "density $\boldsymbol{\rho}(\boldsymbol{x},\boldsymbol{t})$" and "$\boldsymbol{\rho}.u$" for the 3-D fluid structure. These properties characterizes the individual discrete fluid nodes [1].

$$\boldsymbol{\rho}(\boldsymbol{x},\boldsymbol{t}) = \sum_i \boldsymbol{g}_i(\boldsymbol{x},\boldsymbol{t}) \tag{2.5}$$

$$\boldsymbol{\rho}.u(\boldsymbol{x},\boldsymbol{t}) = \sum_i \boldsymbol{\varepsilon}_i \boldsymbol{g}_i(\boldsymbol{x},\boldsymbol{t}) + \frac{f(\boldsymbol{x},\boldsymbol{t})}{2} \tag{2.6}$$

"Equilibrium Distribution function $\boldsymbol{g}^0$" is used in the calculation of $\boldsymbol{g}(\boldsymbol{x},\boldsymbol{\varepsilon},\boldsymbol{t})$ for the next time steps and is given by equation 2.7 as follows [1]

$$\boldsymbol{g}^0(\boldsymbol{x},t) = \boldsymbol{\rho}(\boldsymbol{x},t)w_i\left(1 + 3\varepsilon_i.\boldsymbol{u}(\boldsymbol{x},t) + \frac{9}{2}\Big(\boldsymbol{\varepsilon}_i.\boldsymbol{u}(\boldsymbol{x},t)\Big)^2 - \frac{3}{2}\Big(\boldsymbol{u}(\boldsymbol{x},t).u(\boldsymbol{x},t)\Big)\right) \tag{2.7}$$

For correctness in algorithm, it is important to consider the interaction at the boundary of fluid structure and the flexible fiber-sheet. Bounce back scheme proposed in [23] is used for the calculating the same in this algorithm. Notice that, since the computational domain is a regular cubic structure , there are **6** faces of the fluid cube and the boundary conditions are applied for front, rear, bottom and top surfaces to ensure "no- slip boundary" conditions as proposed in [23].

The above equations are relevant to the LB computation. It does not describes the intrinsic stretching and bending forces within a flexible fiber-sheet under the influence of the fluid flow. These force calculation are the IB part of the algorithm and the following equations illustrates it in a detailed manner [1] . Considering j= 1,2....n_f fiber nodes, Stretching force F_s and Bending force F_b of fiber node j is given by equations 2.8 & 2.9 as [1]

$$(F_s)_j = \frac{K_s}{\Delta\alpha_1^2} \sum_{k=1}^{n_f-1} \Big(|X_{k+1} - X_k| - \Delta\alpha_1 \Big) \frac{\boldsymbol{X}_{k+1} - X_k}{|\boldsymbol{X}_{k+1} - X_k|} \Big(\delta_{kj} - \delta_{k+1,j} \Big) \tag{2.8}$$

$$(F_b)_j = \frac{K_b}{\Delta\alpha_1^4} \sum_{k=2}^{n_f-1} (\boldsymbol{X}_{k+1} + X_{k-1} - 2\boldsymbol{X}_k | (2\delta_{kj} - \delta_{k+1,j} - \delta_{k-1,j}) \tag{2.9}$$

Here $\boldsymbol{X}$ denotes the location of the fiber-sheet node in x,y and z dimensions, α_1 represents the "Lagrangian coordinate " and δ_{kj} is "Kronecker Symbol" given as [1]

$$\delta_{kj} = \begin{cases} 1, & \text{if k = j} \\ 0, & \text{if k} \neq \text{j} \end{cases} \tag{2.10}$$

$\boldsymbol{F}_s$ and $\boldsymbol{F}_b$ together constitute the elastic force and is spread on the fixed fluid nodes, denoted by $\boldsymbol{f}$, this force is calculated as [1]

$$\boldsymbol{f} = \sum_{\alpha} \boldsymbol{F}(\alpha) \delta_l (\boldsymbol{x} - \boldsymbol{X}(\alpha)) \Delta\alpha \tag{2.11}$$

$\boldsymbol{f}$ is the elastic force of the fluid node and is used in calculating the velocity $\boldsymbol{u}$ of the fluid nodes. $\boldsymbol{U}$, which is the velocity of the fiber-sheet nodes and is interpolated from $\boldsymbol{u}$ in following way [1]

$$\boldsymbol{U}(\alpha) = \sum_{x} \boldsymbol{u}(x)\delta_l(\boldsymbol{x} - \boldsymbol{X}(\alpha))l^3 \tag{2.12}$$

Point worth-noting in equation 2.12 is that given any time-step 't', the position coordinate of the fiber-sheet node $\boldsymbol{X}$ from the previous time step i.e. 't-1' is used. Dirac δ_l function is calculated as following which is specific to IB method being dealt with [1]

$$\delta_l(x) = l^{-3}\psi\Big(\frac{x}{l}\Big)\psi\Big(\frac{y}{l}\Big)\psi\Big(\frac{z}{l}\Big) \tag{2.13}$$

where l is the spacing between two fluid nodes and ψ is given by [1]

$$\psi(z) = \begin{cases} \frac{1}{4}\Big(1 + \cos(\frac{\Pi z}{2})\Big), & \text{if } |z| \leq 2 \\ 0, & \text{otherwise} \end{cases}$$

Ultimately, the location coordinates of the fiber-sheet $\boldsymbol{X}$ is calculated as [1]

$$\text{“}\frac{\boldsymbol{X}^{n+1}(\alpha) - \boldsymbol{X}^{n}(\alpha)}{\Delta t} = \boldsymbol{U}^{n+1}(\alpha)\text{”} \tag{2.14}$$

The algorithm progresses sequentially from “n” time steps and the values used in one step serves as an input for the next time step [1]. The implementation of the aforesaid equations with respect to software design is described in the 3^{rd} chapter.

2.4 OpenMP

One of the shared memory versions of LBM-IB have been developed using OpenMP. OpenMP stands for “Open MultiProcessing” [24]. It is an API specification that supports parallelizations for C, C++ and Fortran. It is most commonly used in developing a multi-threaded shared memory program due to its high portability and scalability. It provides an interface for the programmers to utilize the underlying processing capabilities of a multi-core CPU's ranging from a simple desktop to that of a supercomputer. There are three main components provided by OpenMP to be used on top of the base program to make it multithreaded [24].

1. "Compiler directive": It indicates the underlying compiler to compile openmp constructs used in the base code. It is mostly specified in the form of flags during compilation and is language & compiler dependent. For LBM-IB, the underlying compiler used is ***gcc*** and ***-fopenmp*** flag is used as a compiler directive.

2. "Library Routines": The underlying implementation of spawning threads and distributing work to them is hidden from the programmer. By linking the library provided by OpenMP, the aforesaid behavior is guaranteed. LBM-IB includes ***"omp.h"*** header file to accomplish this.

3. "Environment Variables": These parameters controls the run time behavior of the program. For instance, how many threads should be forked, what should be the scheduling mechanism for loop iterations etc. LBM-IB makes use of ***"omp_set_num_threads"*** to share the work among different available resources inside a processor.

OpenMP is based on "fork-join" model in which when a control reaches a preprocessor directive specified in the program, a master thread spawns a number of slave threads and distribute work to those threads. It is responsible for thread creation, distributing work to threads and synchronizing them. The preprocessor directive being used in LBM-IB software is ***"#pragma omp parallel for"***.The main algorithmic design for OpenMP LBM-IB version is to identify the loop iterations that can be parallelized, to identify data dependencies in the form of public and private variables across threads and to identify implicit scoping mechanisms. Synchronization is achieved in the form of implicit barrier provided by OpenMP library. The shared memory version is scalable to manycore systems. The algorithm is discussed in detail in 4.1.

2.5 Pthread APIs

To effectively utilize the computing capabilities of the underlying hardware, it is necessary to change the software design in a way that makes the maximum use of resources at its disposal. Similar approach has been adopted my many existing parallel software libraries [5–8,10], to name a few. This project also provides a shared memory parallel version of LBM-IB method, in which the underlying data structure has been transformed from the serial and OpenMP versions and then light weight Pthread APIs are used to parallelize the simulations. Pthread can be considered as a collection of C programming types and APIs provided by ***"pthread.h"*** library. It is based on shared memory model as depicted in Fig 4.1. It is a low level programming when compared with OpenMP in which the programmer needs to take care of the thread creation and synchronizations unlike OpenMP, but as Pthread works on light weight threads, they are most suitable for situations when optimization cannot be traded with the programming comfort. Moreover, since both threads and the process lie in the same shared space, the memory restrictions for a Pthread program are not limited [25].

POSIX standards support different parallel programming model such as "Manager/Worker", "Pipeline" , "Peer " etc [25]. The pthread version of LBM-IB is based on ***"Peer"*** threaded model in which the underlying idea is analogous to master/slave model, but the master thread that has created the slave threads also participates in the work [25]. Every thread object in Pthreads is identified by a **pthread_** prefix. From the point of view of cube based pthread version of LBM-IB the pthread API's can be categorised in the following two groups [25]

1. **Thread Creation:** The interfaces that are responsible for creating the threads and managing them are discussed here. For instance, initializing the thread object via ***pthread_t*** data type & managing them via ***pthread_create*** and ***pthread_exit***. The initialized threads are actually made execution worthy by passing the type of **pthread_t** to **pthread_create** method which also carries

information about the routine on which this initialized threads should work. Once the execution completes, the allocated threads can be terminated via **pthread_exit** [25].

2. **Thread Synchronization:** Thread Synchronization in this context can be understood in two ways. i) To avoid data being incorrectly read or write by other threads in action when working on a routine and ii) To stop the main thread from exiting until the work is completed. Though, both represent the same idea of thread synchronizations, in i) the synchronization is within the spawned threads including the master or main thread(as it is a Peer model). whereas in the ii) the synchronization is between the master or main thread and the other threads that master has created. The pthread version of LBM-IB uses ***pthread_barrier_wait***, ***pthread_mutex_lock*** and ***pthread_mutex_unlock*** for i), whereas ***pthread_join*** for ii). To further elaborate on synchronization with barrier and mutexes, each of them take objects initialized with **pthread_barrier_t** data type for barrier **pthread_mutex_t** data type for mutex. The barrier routines helps in achieving synchronization between two functions within LBM-IB simulations i.e. to stop other threads from stepping into next steps of simulation until all threads have completed, whereas the lock and unlock feature of mutexes ensures that their is no overlapping of data writing between threads [25].

LBM-IB uses ***-lpthread*** flag to let the compiler know that the code is going to implement functionalities provided by pthread library. The changes in the data-structure and thread synchronizations in context of LBM-IB are discussed in 4.3.

2.6 Hybrid MPI/Pthread Programming

High performance computing via supercomputers allows distribution of work among different nodes. This allows a higher level of parallelization with the work now being distributed first to different nodes residing in the supercomputer and then

shared by the local resources of those nodes. This distribution of tasks and sharing the resources thenceforth is called Hybrid Programming. [26] has shown that hybrid programming has significant advantages over simple shared memory design in many cases where their is less communication overhead, less data dependency or memory utilization and high load imbalance. The underlying hardware design of such systems varies [26] and the programmer needs to take care of various node interconnects to make best use of the computing capabilities effectively. In Distributed computing, every node has its own private memory or address space [27] and before computation, a message passing mechanism is adopted to let the communicating nodes exchange data required for computation. MPI "Message Passing Interface" provides similar interfaces which helps programmers to pass messages from one node to other and achieve distribution. The main objective of MPI specification is to help programmers in building an efficient parallel message-passing program suitable for distributed computing. MPI provides library interfaces as a binding for programs written in C and Fortran and its supports inter-node communication by providing proper synchronizations at the node level or in MPI terms at "COMM_WORLD" level. "COMM_WORLD" identifies all available nodes participating in communication. MPI assigns an individual rank or id to each node, also referred as task or process, and makes them part of the COMM_WORLD [28]. In this thesis, an approach has been made for the first time to provide a distributed hybrid version of LBM-IB. Unlike other hybrid approaches in areas of distributed computing which primarily combines OpenMP with MPI, LBM-IB has been combined with powerful pthread programming model and MPI to utilize data locality along with powerful features of MPI interfaces. ***"cc"*** flag on BigRedII (cray compiler) and ***"mpicc"*** flag on Dragon (Refer Tables 4.1 & 4.2 for system details) is used as a compiler directive for the program. ***"mpi.h"*** is included to provide the underlying implementation of communication. Existing routines in the pthread version of LBM-IB has been tailored to provide point-to point communication between communicating nodes using ***"MPI_Send & MPI_Recv"***. Synchronization

between the processes is done using ***"MPI_Barrier"*** function provided by MPI library. The details of the algorithm is discussed in chapter 5.

3 LBM-IB SERIAL VERSION

3.1 Algorithm: Implementation Point of View

Current work involves creating a serial version of the aforesaid discretization and simulation of flexible structure's interaction with a 3-D fluid structure followed by a shared memory and a distributed version. The base language for all the versions is **C**. This section describes the serial version in detail in terms of different function implemented, entire algorithm of the C program and underlying Data structure. The software has been aptly named as LBM-IB.

3.1.1 Initialization

Before starting the LBM-IB simulations, a sequential step of initialization is performed in the LBM-IB software. The software is made highly flexible, which takes various input required for the simulation in the form of command line arguments from the user. User specifies the fiber-sheet (including number of horizontal and vertical fibers) dimensions, fluid grid dimensions (in terms of number of fluid-nodes in x, y and z directions), initial location of the fiber-sheet in the fluid grid, Number of threads (for parallel version) and number of machines/computer nodes(for Distributed version). Following points enumerates various initialization steps of the software that precedes the actual simulation:

1. **Generation Steps for rectangular Fiber shape and 3-D regular fluid Structure**:

    - Fiber-Shape structure: Based on the inputs from the user, first step is to generate the flexible fiber-sheet structure. It is made up of parallel strands of horizontal and vertical fibers (Refer Figure 3.1), the same

being named(in the software) and refereed as *fibers_row* and *fibers_clmn* thenceforth. In an attempt to make the software more user-friendly, many different fiber-sheet can be used in the system to make a composite fiber-shape. In the software, however, only one fiber-shape with a single rectangular fiber-sheet is simulated. The generation is carried out by the function ***gen_fiber_shape***. The input to this function are fiber-sheet's width, height, total number of fibers along row, total number of fibers along column and original location of the fiber-sheet in 3-D fluid world i.e. initial position of x, y and z coordinates of the fiber-sheet. As depicted in Figure 3.1, the fiber-shape structure is granulated to a fiber-node level with each microscopic fiber-node having a coordinate value in x, y and z directions. Since, it is a rectangular 2-D structure, the x-coordinate value is constant for all fiber-nodes before simulation. Memory allocation is done on heap using `malloc`. Fiber-shape being generated is the outcome of this routine.

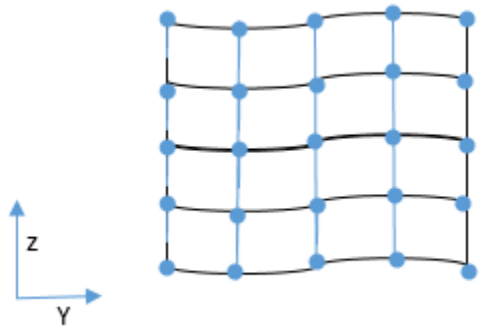


Figure 3.1. Immersed Fiber structure: with 5 parallel fiber strands along rows and columns, enclosing 25 fiber-nodes in total.

- Fluid-Grid Generation: After generating the fiber-sheet, the viscous 3-D fluid grid generation is done via function ***gen_fluid_grid***. As above, the input for this function are taken from the user and passed to the formal

parameters defined in the function definition. The fluid-grid is granulated to level of a fluid-node, with each fluid node having a dimensionless distribution function ρ as well as the velocity vector along x, y and z directions. User specifies the number of fluid nodes in each direction denoted by *fluidgrid_x*, *fluidgrid_y* and *fluidgrid_z*. The software has different fluid-grid generation routine for shared and distributed version, as the data-structure being used are different in different versions.

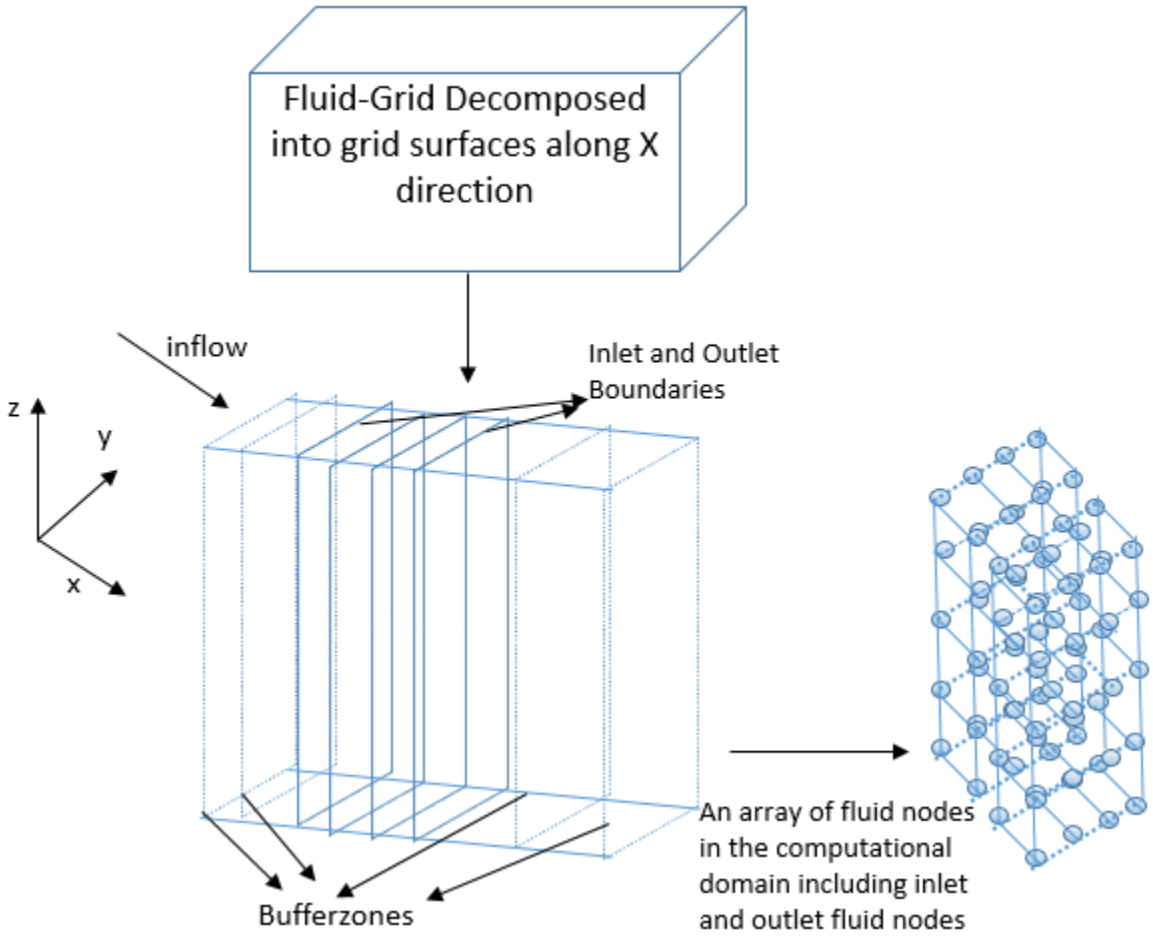


Figure 3.2. 3D fluid grid with 4 surfaces in computation domain, each surface comprising 2-D array of fluid nodes.

2. **Initializing GV**: Next step that precedes simulation is the initialization of various constants that are required to be assigned before simulation. This initialization is necessary to model the virtual configuration of LBM-IB method. Shared Memory Model is being used as the basic programming model for the parallel versions of LBM-IB and this "GV" object is used in those versions for sharing the data across different threads or different processes. GV (Global Variable) stores the data that is shared across all the routines and even in the serial version some parameters being stored in GV can be reused eventually. An object of GV denoted by 'gv' from now on, is used to collect the required information by accessing a pointer to it. ***init_gv*** carries out this initialization. Fibershape and fluid-grid shape are passed to this method. The basic initialization carried out in ***init_gv*** are summarized below-:

   - For Fiber-sheet: As illustrated in equations 2.8 & 2.9, both stretching and bending force of the fiber-sheet requires constants K_s and K_b respectively. The same are calculated and assigned to gv in this method

$$K_s = K_{shat} * \rho * u_l * L_l \tag{3.1}$$

$$K_b = K_{bhat} * \rho * u_l * u_l * L_l^4 \tag{3.2}$$

     K_{shat} is the "stretching compression coefficient" and is taken as "20", ρ is the "fluid mass density" of each fluid node, u_l is the initial velocity of the fluid and L_l is the dimensionless length which should be the smallest among width and height of the fluid. Apart from this, the fiber-sheet nodes are moved in the direction of the fluid velocity. For example, if the initial location of the fiber-sheet are 20, 20.5 and 11.5 in x, y and z direction, and if the fluid flows initially in x direction with $u_l = .001$ then, inside ***init_gv*** the coordinates are changed to 20.001, 20.5 and 11.5. Note that this is the origin of the fiber-sheet and if the width and height are taken as 20, the corner-most point will be at 40.001, 40.5 and 31.5.

- For Fluid-grid: One of the major discretization being done is sub-dividing the particle velocity 'ε' in 19 different directions based on the equation 2.2. This is characteristic to D3Q19 model of LBM and is assigned in this method for every 19 direction. Other than this, the inflow speed is assigned to the fixed fluid-grid lattice in each direction which is represented by *vel_x*, *vel_y* and *vel_z*. Based on [1], the values are taken to be .001, 0.0 and 0.0 respectively. This is same as u_l used in above calculations.
- Different constants such as the gravitational effect g_l, sound of the model c_s, τ used in calculating Distribution function are stored in gv to be accessed later. Also, the time of simulation denoted by *TIME_STOP* is initialized in this method, algorithm is repeated till the stipulated simulation steps are completed.
- The actual computation domain of the fluid grid is surrounded by buffer zones from top, bottom, front and rear side of the fluid grid. These buffer zone boundaries are also evaluated and assigned in this method. This is done to ensure that computation is correct under the buffer-zone and to create a virtual long fluid channel [1].

3. **Initializing DF0 & DF1**: Equilibrium Distribution function g^0 needs to be calculated before simulation and the function ***init_eqlbrmdistrfuncDF0*** is developed for the same purpose. It is stored as DF0 in LBM-IB software and is calculated using equation 2.7. Based on 19 different ε values, every fluid node is assigned a unique DF0 value. Once DF0 is assigned, every fluid node gets its unique distribution function value for that time step which is stored as DF1 in LBM-IB software. It is calculated using equation 2.3 and ***init_DF1*** initializes the DF1 value before the simulation starts. During the simulations, the same is done by ***compute_eqlbrmdistrfuncDF1*** function.

4. **Initializing inlet and outlet boundaries**: As depicted in Figure 3.2, the computational domain for a fluid grid is enclosed in the inlet and outlet bound-

aries. This inlet and outlet boundaries are themselves enclosed in the buffer zones. Inlet boundary is the face of the fluid-grid facing the direction of the inflow fluid velocity u_l and outlet is the face from where the virtual fluid flow exits. The distribution function value for each fluid node lying on these inlet and outlet boundaries are assigned by ***init_df_inout*** routine, which as the name suggests, copies the calculated DF0's for every fluid node to the inlet and outlet boundaries.

After Initialization, simulation is started for stipulated number of time steps. The entire algorithm can be sub-divided into IB and LBM part. IB method involves computing the elastic forces on the fiber-sheet, finding influential domain of a fiber-node and spreading those forces to the influenced fluid node. LBM part involves computing DF1 from the elastic forces being spread from fiber-sheet, streaming those force to the neighboring fluid nodes, apply bounce-back scheme for front, rear, top & bottom fluid surfaces [23], evaluate new ρ & velocities of the fluid-nodes and ultimately move the fiber-sheet under the influence of the changed mesoscopic properties of the fluid. As such their is no clear distinction between the IB and LBM method, as the NS equations from IB are solved using LBM, but from the implementation point of view the distinction is quite lucid. The IB method involves studying both the structure and fluid properties on moving "Lagrangian" grid points and fixed "Eulerian" plane respectively [1]. The following sections describe these methods and their implementation (in software) in detail.

3.1.2 IB

Under the influence of fluid's initial velocity u_l initialized in ***init_gv***, stretching and bending forces, collectively termed elastic forces starts developing on the microscopic fiber-sheet nodes. Computation of stretching forces exerted by the fiber-nodes is implemented in ***compute_stretchingforce*** and bending forces in ***compute_bendingforce***. Then, the two forces are summed up together in ***com-***

pute_elasticforce which is used for spreading . These forces are characteristic attribute of an individual fiber node and are stored in the data structure allocated for Fibershape (Refer Figure 3.5). Following the calculation of forces, for every fiber-node an influence domain of 4x4x4 fluid-nodes is calculated, which identifies the "Eulerian" or fixed fluid-nodes. The elastic forces from the fiber-nodes are then spread to the influenced fluid-nodes. These two-fold work of identifying the influential domain and spreading the forces is implemented in ***find_ifd_and_SpreadForce***. The order of calculating bending forces and stretching forces is not important, but once calculated, they are summed up for every fiber-sheet node and then eventually spread. The calculation is done for both horizontal and vertical parallel strands of fiber in succession. Also, there is a separate treatment for the boundary fiber-sheet nodes while calculating the forces which will be described in detail in the following subsections. The following sections describes the rearrangement of the mathematical equations in the code and their use.

Calculation of Bending Forces:

Bending Force for a fiber-node is calculated from the position or the location coordinate of the fiber-node and its neighboring fiber-nodes. To elucidate further, for a given fiber-node, it's bending force is dependent on it's own location as well as the location of its immediate two neighbors lying to its left, right, top and bottom. The following equation simplifies the mathematics behind the equation 2.9 and illustrates it from the implementation point of view. This simplified equation is implemented in ***compute_bendingforce***. Here FN_i indicates the location of fiber-node at i^{th} location and BF_i denotes the Bending Force of the fiber at i^{th} location. The sub scripted values in terms of i denotes the location of the fibers in the neighborhood

$$BF_i = bending_{const}\Big(- FN_{i+2} + 4 * (FN_{i+1}) - 6 * (FN_i) + 4 * (FN_{i-1})\Big) \qquad (3.3)$$

The above equation is applicable only for the fiber nodes lying in the middle of the sheet. Kronecker symbol defined previously is used to derive the formulas for the

corner most nodes, second fiber-node and pen-ultimate nodes as follows. For the first fiber-node following formula is used.

$$BF_i = bending_{const}\Big(- FN_{i+2} + 2 * (FN_{i+1}) - FN_i\Big) \tag{3.4}$$

While, for the second most fiber-node the formula becomes

$$BF_i = bending_{const}\Big(- FN_{i+2} + 4 * (FN_{i+1}) - 5 * (FN_i) + 2 * (FN_{i-1})\Big) \tag{3.5}$$

For the penultimate fiber-node the formula becomes

$$BF_i = bending_{const}\Big(2 * (FN_{i+1}) - 5 * (FN_i) + 4 * (FN_{i-1})\Big) \tag{3.6}$$

Whereas, the following is used for the last fiber-node

$$BF_i = bending_{const}\Big(- FN_i + 2 * (FN_{i-1} - FN_{i-2})\Big) \tag{3.7}$$

In all the above equations $bending_{const}$ is calculated as $\frac{K_b}{\Delta\alpha_1^4}$ described in the previous chapter. Once the Bending force for a given fiber-node is calculated in one direction vertically or horizontally, the same is summed with the other direction consequently. Though, the fiber-sheet is 2-D, the calculation is carried out for all x, y and z directions to know the position of the fiber in the 3-D fluid grid.

Calculation of Stretching Forces:

Stretching force for a given fiber-node is also calculated on similar lines as that of bending forces. It is calculated based on the distance between its left, right, top and bottom fiber-nodes. The distance between immediate fiber-node in the right is calculated as follows

$$dist_{right} = \sqrt{(FN_{i+1} - FN_i)_x^2 + (FN_{i+1} - FN_i)_y^2 + (FN_{i+1} - FN_i)_z^2}$$

$$dist_{left} = \sqrt{(FN_{i-1} - FN_i)_x^2 + (FN_{i-1} - FN_i)_y^2 + (FN_{i-1} - FN_i)_z^2}$$

Then, the stretching force for a fiber-node at i^{th} location is given by

$$SF_i = stretch_{const}\Big((dist_{right} - ds_1) * (\frac{FN_{i+1} - FN_i}{dist_{right}}) + (dist_{left} - ds_1) * (\frac{FN_{i-1} - FN_i}{dist_{left}})\Big) \tag{3.8}$$

The above generalization changes for the first and last point as follows. For the first point it is given as

$$SF_i = stretch_{const}\Big((dist_{right} - ds_1) * (\frac{FN_{i+1} - FN_i}{dist_{right}})\Big) \tag{3.9}$$

Whereas for the last point it is calculated as

$$SF_i = stretch_{const}\Big((dist_{left} - ds_1) * (\frac{FN_{i-1} - FN_i}{dist_{left}})\Big) \tag{3.10}$$

The same formalization is carried out for the top and bottom neighbors and from the implementation point of view it is calculated for the fiber-nodes along the columns. ds_1 is the distance between two adjacent fiber nodes. For our experiments, the distance are kept uniform for horizontal and vertical fibers as the width and height of the fiber-sheet is same. $streatch_{const}$ is given by $\frac{K_s}{ds_1^2}$. Unlike, Bending force calculation, where it is required to take care of even the penultimate and second fiber-node as boundary cases, here the formulation changes only for the first and the last fiber-node. Once, the above calculation is carried out in one direction say horizontally, the same is summed following a similar calculation in the vertical direction. So, at a given instant a fiber-node has the force in relation to its neighbors on left, right, top as well as bottom. ***compute_elasticforce*** is a trivial function which just sums both the bending and stretching forces calculated and stores in the form of elastic force of a fiber-node.

Finding Influential Domain & Spreading Forces:

After completing the force calculation for a given fiber-node, its interaction with the fluid structure starts. The first step in this interaction is to find the fixed fluid nodes arranged on a Eulerian lattice which will be influenced for a given fiber-node. For every fiber-node, a 4x4x4 space around that fluid node is identified and using the *floor* operation 64 points are evaluated. Then, before spreading the forces directly on

those 64 fluid-nodes, distance between the fluid node and fiber-node is calculated as follows

$$temp_{dist} = \frac{1}{64} * \left(\left(1 + \cos(\frac{\pi}{FluidNode_{i0:64} - FiberNode}) \right)_{x,y,z} \right)$$

This $temp_{dist}$ is multiplied for x, y and z direction in the above equation. For every influenced fluid node having a different distance, elastic fiber for that fluid node is spread as follows

$$ElasticForce_{Fluidnode} = ElasticForce_{Fibernode} * temp_{dist}$$

This calculation is repeated for all the influenced fluid node (in this case 64) for a given fiber-node. Similar to the bending and stretching forces, the elastic forces for all the fluid-nodes are also summed together for all the fluid-nodes lying in the influenced region.

3.1.3 LBM

Following IB simulation, once the forces are computed for all the influenced fluid nodes, LBM starts to solve the NS equations for IB. The first step is to simulate the mesoscopic fluid attribute, equilibrium Distribution function DF1 followed by streaming of these values in the neighborhood. After streaming, a bounce back scheme is applied to all the fluid nodes lying closer to the rigid surfaces which are top, bottom, front and rear faces of the fluid-grid.The macroscopic property of the fluid:ρ and velocities are computed and then the fiber-sheet is moved under the influence of newly computed velocities. Following sections explains in detail about the functions implemented to achieve the same.

Particle Collision Factor or DF1:

DF1 is the naming convention being used in the software for simple understanding. It represents the equilibrium distribution function 'g($\boldsymbol{x}$, $\boldsymbol{\varepsilon}$, $\boldsymbol{t}$)' in a given time. Since,

it is the first value of 'g($\boldsymbol{x}$, $\boldsymbol{\varepsilon}$, $\boldsymbol{t}$)' for the fluid node, it is named as DF1 and when the same value is used for streaming it is stored as DF2 buffer. In simple terms, DF1 can be understood as computing the collision factor in the neighborhood of 19 fluid nodes, including the fluid node itself. It is calculated using equations 2.3 and 2.7 and in the code is implemented in ***compute_eqlbrmdistrfuncDF1*** function. This is one of the costliest function in terms of time spent during one iteration of the entire LBM-IB algorithm. As described in the aforesaid equations 2.3 and 2.7, DF1 value is computed for 19 different values of particle discrete velocity $\boldsymbol{\varepsilon}$ and its associated weight. Therefore, for a given a fluid node, it becomes a very compute intensive routine and hence is an ideal candidate for change in the computation strategy used currently. It can be optimized using loop unrolling.

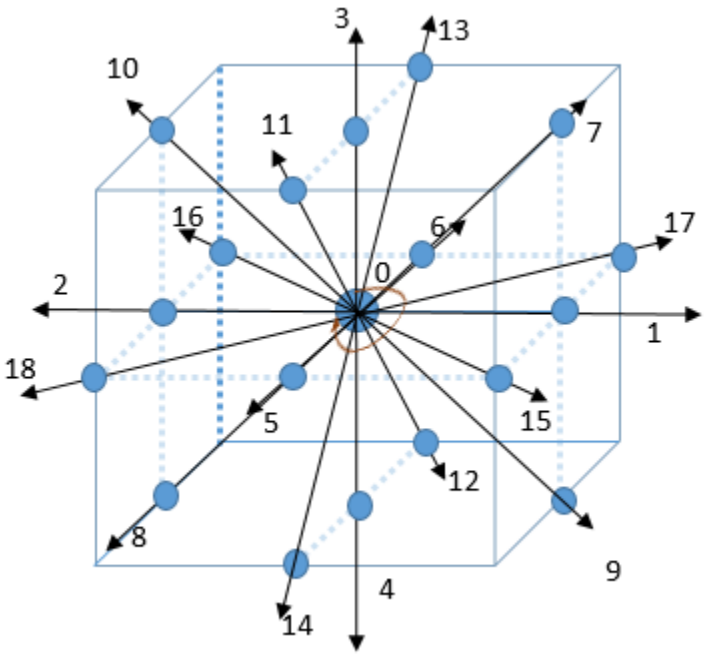


Figure 3.3. LBM D3Q19 Model: Distribution function is streamed in 19 different directions including the node itself [1].

Streaming:

This LBM portion is specific to the model being used in the computation, which is D3Q19 in this case. As the figure 3.3 illustrates, every influenced fluid node spreads its distribution function value computed in ***compute_eqlbrmdistrfuncDF1*** to its neighborhood. It is nothing but a new DF1 value being stored as DF2 for the neighboring fluid nodes. It is being termed as DF2 because the distribution function value stored as DF2 for a fluid-node in time-step 'n' is used in time-step 'n+1' for computing macroscopic properties of that fluid-node such as 'ρ' and 'velocity'. ***stream_distrfunc*** implements the aforesaid functionality. It is pretty straightforward for the serial version but needs to be addressed carefully for shared versions as the change in data structure for the former introduces new boundary cases for both shared and distributed versions, both discussed in relevant chapters in detail.

Handling Rigid-walls:

The 3-D fluid grid arrangement of the Eulerian fluid lattice encloses the fluid-nodes from top, bottom, front and rear. These surfaces can be considered as rigid walls, from which the fluid nodes rebounds in the opposite direction. Worth-noting is the fact that the fluid-grid has to be open ended from left to right to support a viscous fluid flow in that direction. Therefore, before updating ρ and velocities from DF1, it is necessary to evaluate the bounce back conditions for the fluid-nodes on these surfaces to update the distribution function of the fluid-nodes on the stipulated boundary surfaces [23]. This is implemented in ***bounceback_rigidwalls*** function. The underlying idea is to copy the DF1 value of the fluid node on the boundary value and store it as DF2 for the opposite direction. This ensures that the fluid-nodes on these rigid surfaces acts as if they are being bounced back and exhibit the same 'g($\boldsymbol{x}$, $\boldsymbol{\varepsilon}$, $\boldsymbol{t}$)' value form previous time step. For instance, for handling the boundary conditions following switching is performed in this function.

For Bottom surface:

$$DF2_{\varepsilon=3} <= DF1_{\varepsilon=4}; \quad DF2_{\varepsilon=7} <= DF1_{\varepsilon=8};$$

$$DF2_{\varepsilon=10} <= DF1_{\varepsilon=9}; \quad DF2_{\varepsilon=11} <= DF1_{\varepsilon=12}; \quad DF2_{\varepsilon=13} <= DF1_{\varepsilon=14};$$

For top surface:

$$DF2_{\varepsilon=4} <= DF1_{\varepsilon=3}; \quad DF2_{\varepsilon=8} <= DF1_{\varepsilon=7};$$

$$DF2_{\varepsilon=9} <= DF1_{\varepsilon=10}; \quad DF2_{\varepsilon=12} <= DF1_{\varepsilon=11}; \quad DF2_{\varepsilon=14} <= DF1_{\varepsilon=13};$$

For front surface:

$$DF2_{\varepsilon=6} <= DF1_{\varepsilon=5}; \quad DF2_{\varepsilon=12} <= DF1_{\varepsilon=11};$$

$$DF2_{\varepsilon=13} <= DF1_{\varepsilon=14}; \quad DF2_{\varepsilon=16} <= DF1_{\varepsilon=15}; \quad DF2_{\varepsilon=17} <= DF1_{\varepsilon=18};$$

For Rear surface:

$$DF2_{\varepsilon=5} <= DF1_{\varepsilon=6}; \quad DF2_{\varepsilon=11} <= DF1_{\varepsilon=12};$$

$$DF2_{\varepsilon=14} <= DF1_{\varepsilon=13}; \quad DF2_{\varepsilon=15} <= DF1_{\varepsilon=16}; \quad DF2_{\varepsilon=18} <= DF1_{\varepsilon=17};$$

Updating Fluid's Macroscopic properties:

Next major steps involved in LBM simulation is to derive the fluid's 'fluid mass density ρ' and then compute the fluid-nodes velocities from ρ using equations 2.5 and 2.6 respectively. 'ρ' is directly derived from the summation of DF2 values streamed from the neighbors. Then, the speed of the sound in the model is used to calculate the velocity of fluid-node. ***compute_rho_and_u*** routine is implemented for the same purpose. Here 'u' specifies the velocity of the fluid node in each direction, which is calculated as shown in equations 3.11a & 3.11b.

$$vel_{x,y} = \frac{\sum_{\varepsilon=0}^{18} c_\varepsilon * DF2_\varepsilon + 0.5 * t * ElasticForce_{x,y}}{\rho}, \tag{3.11a}$$

$$vel_z = \frac{\sum_{\varepsilon=0}^{18} c_\varepsilon * DF2_\varepsilon + 0.5 * t * (ElasticForce_y + g_l * \rho)}{\rho} \tag{3.11b}$$

Here 'g_l' denotes other external forces such as gravitational forces, which are not considered in the calculation and 't' is the current time-step value.

Updating Fiber-sheet's position

The next step in the LBM part is to move the fiber-sheet under the influence of the fluid-nodes velocities calculated above. This is another important step in LBM simulation as it completes the mutual interaction between fiber-node and fluid-node or in other words, fluid & flexible structure interaction. As per the implementation, first the influential domain of a fiber-node is evaluated as done in the function ***find_ifd_and_SpreadForce***, then the velocities of the fluid-nodes residing in the influenced domain is used to update the x,y and z coordinates of that fiber-node. All influenced fluid-nodes contribute in moving a single fiber-node. This is achieved by summing the velocities of all the fluid-nodes in the influenced region as follows

$$Pos_X = t * \sum_{0}^{64} Vel_X * (1 + \cos(\frac{\Pi}{2} * r_x)) * (1 + \cos(\frac{\Pi}{2} * r_y)) * (1 + \cos(\frac{\Pi}{2} * r_z)) \tag{3.12}$$

The left hand side of the equation denotes the fiber-nodes coordinates values in x,y and Z direction and in the right hand side, the vel attribute is the velocity of the fluid node in those direction respectively. r_x, r_y and r_z is the distance between the influenced fluid-node and the fiber-node whose position is updated in x, y and z direction. Since, there is a small influenced region of 4x4x4, the summation is carried for all 64 influenced fluid nodes. Aforesaid functionality is implemented in ***moveFiberSheet*** function.

3.1.4 Regeneration Functions For Next Time Step

3.1.3 ends the LBM-IB computation, but in order to continue the simulation for next time-steps following functions are implemented:

- ***copy_buffer's_DF***: As mentioned before in 3.1.1, the actual computational domain is surrounded by buffer zones on the inlet and outlet boundaries. It is therefore necessary to restore the buffer zone's conditions in n^{th} time-step to that in $(n+1^{th})$ time step. This is done by replacing the streamed Distribution function value of all the fluid nodes lying on these inlet and outlet boundaries i.e. DF2, by the distribution function value initialized in 3.1.1.

- ***copy_DistributionFunction***: As illustrated in equation 2.3, distribution function of a fluid-node in a given time step, is derived from the distribution function computed in previous time-step. Therefore, the streamed distribution function DF2 of all fluid nodes are copied back to the DF1 buffer so that it can be reused in computing distribution function for the next time step.

- ***PeriodicBC***: As shown in fig 3.4 the 3-D fluid arrangement is supposed to be a long cylindrical hollow tube with the computation domain for a given time instant being a regular cube. Therefore, in order to achieve simulation for entire cylindrical fluid grid, distribution function of fluid-nodes on the extreme inlet and outlet boundaries are swapped with each other as shown in figure 3.4.

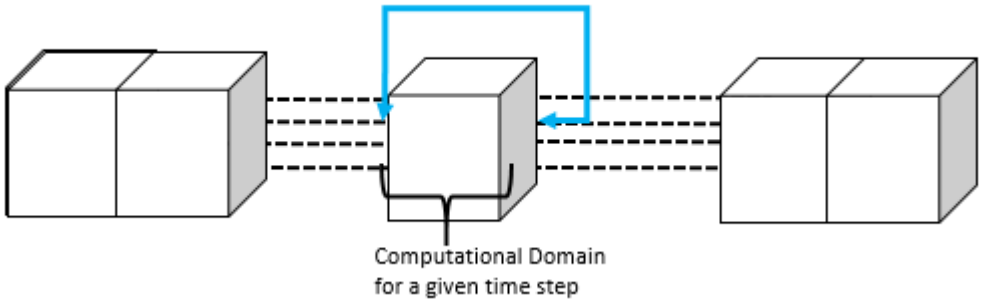


Figure 3.4. Distribution function being swapped at the inlet and outlet boundaries to accommodate elongated channel flow.

The entire LBM-IB simulations have been illustrated in Algorithm 1. Here, fsh_w, fsh_h, tf_r,tf_c,fs_{x0}, fs_{y0} and fs_{z0} are the fiber-sheets parameter which represents fiber-sheet's width, height, total fibers along horizontal direction or row, total fibers along vertical direction or column, starting x, y and z coordinate for the fiber-sheet respectively. Whereas, fl_x, fl_y, fl_z are the number of elements or fluid nodes in x, y and z direction for fluid-grid.

Algorithm 1 LBM-IB Sequential Version : Input:(fsh_w, fsh_h, tf_r, tf_c, fl_x, fl_y, fl_z, fs_{x0}, fs_{y0}, fs_{z0})

/*Refer 3.1.1 for the following initializations*/

fiber_shape = gen_fiber_shape(fsh_w,fsh_h, tf_r, tf_c, fs_{x0}, fs_{y0}, fs_{z0});

fluid_grid = gen_fluid_grid(fl_x, fl_y, fl_z);

init_gv; /*initialized value including fiber_shape and fluid_grid stored in gv object*/

init_eqlbrmdistrfuncDF0(gv);

init_DF1(gv);

init_df_inout(gv);

/*Initialization ends*/

$time \leftarrow 0$

while $time <= TIME\_STOP$ **do** ▷ TIME_STOP initialized in GV

1)$compute\_bendingforce(gv)$; ▷ /*IB Simulation starts Refer 3.1.2*/

2)$compute\_stretchingforce(gv)$;

3)$compute\_elasticforce(gv)$;

4)$find\_ifd\_and\_SpreadForce(gv)$; ▷ ifd:influential domain

/*IB Simulation ends*/

5)$compute\_eqlbrmdistrfuncDF1(gv)$;▷ /*LBM simulation starts Refer 3.1.3*/

6)$stream\_distrfunc(gv)$;

/*LBM Simulation ends*/

7)$bounceback\_rigidwalls(gv)$;

8)$compute\_rho\_and\_u(gv)$; ▷ u refers to fluid-nodes's velocity

9)$moveFibersheet(gv)$;

10)$copy\_buffer's\_DF(gv)$; ▷ regeneration functions starts Refer 3.1.4

11)$copy\_DistributionFunction(gv)$;

12)$PeriodicBC(gv)$;

/*regeneration functions ends*/

end while

3.2 Underlying Data Structure

This section describes in brief important data structures being used in LBM-IB serial version.

```
/* a set of fiber sheets compose a general IB structure*/
typedef struct fiber_shape_t {
  Fibersheet *sheets;
  int       num_sheets;
} Fibershape;
```

(a) Fibershape

```
/* one fiber sheet */
typedef struct fiber_sheet_t {
 Fiber*    fibers;        // i.e., an array of fibers
 double    width,   height
 int      num_cols, num_rows;
 /* bottom left corner: located at <min_y, min_z> */
 double    x_orig;        // starting point for fibersheet x0 inside fluid_grid
 double    y_orig;        // starting point for fibersheet y0 inside fluid_grid
 double    z_orig;        // starting point for fibersheet z0 inside fluid_grid
} Fibersheet; /* initial configuration of the sheet is given by users!!!*/
```

(b) Fibersheets inside Fibershape

```
/* a single fiber consisting of a number of fiber nodes */
typedef struct fiber_t {
 Fibernode* nodes;/*pointing to an array of fiber nodes*/
 int    num_nodes; /*how many fiber nodes on a fiber*/
} Fiber;
```

(c) Strands of fibers inside a fibersheet.

```
typedef struct fiber_node_t {
  double  x, y, z;
  double  bend_force_x, bend_force_y, bend_force_z;
  double  stretch_force_x, stretch_force_y, stretch_force_z;
  double  elastic_force_x, elastic_force_y, elastic_force_z
} Fibernode;
```

(d) Microscopic Fibernode

Figure 3.5. Data Structure for Immersed Boundary.

```
typedef struct fluid_grid_t {
  int         x_dim;  //equivalent to number of surfaces
  int         y_dim;    //Surface dimension along Y direction (or #columns)
  int         z_dim;    //Surface dimension along Z direction (or #rows)
  Fluidsurface*  surfaces;  //pointing to an array of fluid surfaces
  Fluidsurface*  inlet;    //with constant values*/
  Fluidsurface*  outlet;   //with constant values*/
} Fluidgrid;
```

(a) Fluid Grid

```
typedef struct fluid_surface_t {
  Fluidnode* nodes;
} Fluidsurface;

typedef struct fluid_node_t {
double vel_x, vel_y, vel_z;
double rho;
double df1[19], df2[19] dfeq[19];
double df_inout[2][19];
double elastic_force_x, elastic_force_y, elastic_force_z;
} Fluidnode;
```

(b) FluidGrid Surface and microscopic Fluid nodes.

Figure 3.6. Data Structure for 3-D Fluid grid Serial Version.

```
typedef struct gv_t {
  /* The immersed structure */
  Fibershape* fiber_shape;
  /* The fluid grid */
  Fluidgrid* fluid_grid;
  /* Constant parameters used in LBM-IB */
  double    tau, nu_l, u_l, rho_l, L_l, g_l, Ks_l, Kb_l;
  double    Re, cs_l, Kshat, Kbhat, Fr;
  int       dt, time, TIME_STOP;
  double    c[19][3];              //stores 19 different velocities for ksi
  int       ib, ie, je, jb, ke, kb;    //For Fluid Grid's actual computation part
}* GV;
```

Figure 3.7. Data Structure for GV Serial Version.

- **Immersed Boundary or Fiber-sheet**: To have flexibility in the software reuse, immersed structure is defined as collection of fiber-sheets. Currently, the results have been obtained for only one sheet. Every Fiber-shape has a pointer to store sheets with each sheet identifying the microscopic fiber-node via a pointer to fiber-node structure as shown in the figure 3.5.

- **FluidGrid**: As shown in fig 3.4, fluid grid represents computational domain inside a long micro channel of fluid grids. As shown in Figure 3.2, 3-D fluid grid is decomposed into grid surfaces along one direction (X in this case). Every grid surface can be considered as a 2-D array of fluid nodes along the remaining two directions. The fluid grid has three buffers for 2 dimensional surfaces. Two of them being for inlet and outlet boundaries and the remaining for two dimensional stack of fluid surfaces inside those boundaries pointed by surfaces (Refer 3.6(a)). The dimensions x_dim, y_dim and z_dim in 3.6(a) are actually number of fluid nodes along those directions. FluidSurface structure has access to all the fluid nodes on that surface via pointer to the structure Fluidnode. As depicted in Fig 3.6(b) every microscopic fluid node carries the required mesosocopic property of the fluid node used in the algorithm. The attribute df_inout[2][19] represents the distribution function of the fluid nodes on inlet and outlet: where df_inout[0][19] and df_inout[1][19] are the buffers used for inlet and outlet respectively.

- **Global Variable GV**: It represents the global data that is available to entire LBM-IB software. As shown in figure 3.7, besides from pointers to the Fiber-shape and Fluid grid, it stores various constants which are initialized before simulations in ***init_gv*** function. For instance, **c[19][3]** stores the fluid-nodes discrete velocities given by 2.2, **tau** represents the "relaxation time" used in equation 2.3 etc. As mentioned before regarding buffer-zones surrounding the actual computational domain, these zones are identified by **ib, ie, jb, je, kb,**

ke in x, y and z dimensions. Here ib and ie identifies beginning index and ending index in x direction and like wise for jb-je & kb-ke for y and z dimensions.

3.3 Performance Analysis

The experiment was conducted for a 124x64x64 fluid grid in which a 20x20 fiber-sheet is immersed as shown in Fig 3.8. The fiber-sheet comprises of 52x52 strands of fibers parallel to each other along its width and height. Also, the initial position of the fiber and the number of time-steps for simulations should be selected carefully. For instance, if the fiber-sheet is placed at the extreme boundary and the experiment is carried out for a larger time step, then the fiber-sheet will go out of the fluid-grid and will not complete the simulation. The initial position of the fiber-sheet has been kept at 20, 20.5 and 11.5 in x, y and z dimensions with respect to the fluid grid coordinates. Before parallelizing the aforementioned Algorithm 1, GNU profiler

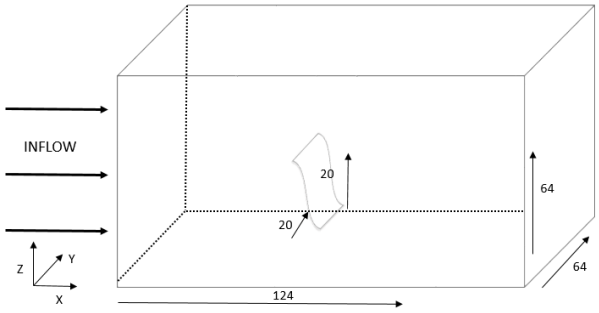


Figure 3.8. A 2-D flexible fiber-sheet of 20x20 dimension is immersed in a 3-D viscous in-compressible fluid of 124x64x64 dimension.

gprof [29] was used to carry out a simple flat profile on the serial version and experiments were conducted on BigRedII and Dragon (Refer Table 4.1 & 4.2 for System details). Performance of the software is carried out for the LBM-IB simulations and the regeneration steps i.e. the functions being called inside while loop as shown in Algorithm 1. GNU profiling helps to identify the time spent by each function or

kernel in a very efficient manner. It helps in identifying the bottleneck of the entire algorithm and gives a chance to optimize those kernels.

Table 3.1 shows the profiling for sequential LBM-IB on BigredII which is Linux machine with two AMD Opteron 16-core CPUs and 64GB of memory [30] and Table 3.2 shows the same on Dragon which is also a Linux machine but with two Intel 12-core CPUs at 2.80GHz and 50 GB of memory. The table lists the functions stated in Algorithm 1, with the first column being their execution order of function index specified in Algorithm 1, second column identifying the function name and the third column denoting the percentage of total time taken by the function during the entire simulation. As evident from the table, it can be observed that for both Dragon and BigredII, the first four kernels take up almost 97% of the total execution time. All these functions are related to the fluid-node computations, in which a fluid-node is being visited in four levels of iterations: first the fluid grid surfaces along X axis followed by the nodes lying in either direction of Y and Z axis as elucidated in Fig 3.2 and then in 18 different directions for a fluid-node corresponding the ε values as shown in Fig 3.3.

The performance results are in tune with the input to the algorithm. As the size of the fluid grid is much larger than that compared with the fiber-sheet the memory consumption and the resource utilization for computations involving those fluid-nodes takes up almost the entire memory and processing capabilities provided by the processor. An interesting observation is that functions at positions 3^{rd} and 4^{th} in the Table namely **stream_distrfunc** and **copy_DistributionFunction**, in which one data buffer is copied to other data buffer, with no extra computations also contribute towards 13% of the total time. Another striking observation is that the two different machines do not have identical kernel rankings at low level. For instance, for AMD processor, **stream_distrfunc** is faster than **copy_DistributionFunction** whereas it is just the opposite on an Intel processor. This initial profiling of the serial code with same input on two different machines helped in analyzing the different time bounds and restrictions involved when LBM and IB are combined together which

will be very helpful in optimizing the LBM-IB approach in general and which will eventually help in creating an efficient LBM-IB base to be used for parallelizations. Though, the project does not aim to optimize the existing algorithm but it gives an idea on how to effectively look out for routines taking more time and modify them as a part of future work.

Table 3.1
Gprof Profiling of Serial LBM-IB on BigredII

Function Index	**Function Name**	**Percentage of Total Time**
5)	compute_eqlbrmdistrfuncDF1	73.21%
8)	compute_rho_and_u	12.58%
11)	copy_DistributionFunction	5.93%
6)	stream_distrfunc	5.35%
4)	find_ifd_and_SpreadForce	1.36%
9)	moveFiberSheet	0.74%
12)	periodicBC	0.29%
7)	bounceback_rigidwalls	0.22%
10)	copy_buffer's_DF	0.17%
1)	compute_bendingforce	0.03%
2)	compute_stretchingforce	0.02%
3)	compute_elasticforce	0.00%

Table 3.2
Gprof Profiling of Serial LBM-IB on Dragon

Function Index	**Function Name**	**Percentage of Total Time**
5)	compute_eqlbrmdistrfuncDF1	72.48%
8)	compute_rho_and_u	10.76%
6)	stream_distrfunc	7.17%
11)	copy_DistributionFunction	5.71%
4)	find_ifd_and_SpreadForce	2.86%
10)	copy_buffer's_DF	0.32%
12)	periodicBC	0.25%
9)	moveFiberSheet	0.20%
7)	bounceback_rigidwalls	0.10%
1)	compute_bendingforce	0.03%
2)	compute_stretchingforce	0.02%
3)	compute_elasticforce	0.02%

4 LBM-IB SHARED MEMORY PARALLEL VERSIONS

In an attempt to develop a parallel library for the Algorithm 1 stipulated in 3.1, two shared memory versions of the LBM-IB method are developed. The first version is developed using OpenMP interface which uses the same data structure as that in the serial version, whereas, the second version is built using Pthread library with a modified data structure and changes in LMB-IB implementation to address those changes.

The first step in developing the shared memory versions (both OpenMP and Pthread) was to identify suitable candidates which can be parallelized. On a broader level, LBM-IB inherently can be parallelized on two levels, first for the fluid-grid components or fluid nodes and the other on the fiber-sheet level or for the fiber-nodes. As illustrated in Fig 3.2, every fluid grid is being discretized by a fluid surface which is vertical to x axis and lying on a y-z plane. To perform simulations on an individual fluid node, first every grid surface is visited and then 2-D stack of fluid nodes is visited in the other two dimensions using nested for loops. Similarly, the fiber-nodes inside the fiber-sheet as shown in Fig 3.1, can be considered as a 2-D matrix, wherein every fiber-node along row has stipulated number of fiber-nodes along columns. IB operations of force calculation is done for all the fiber-nodes in one direction for instance along row, the elastic forces computed in this direction represents partial force for that direction only. Then these forces are added to the same fiber nodes but in opposite direction to get the total force (stretching or bending) for a given fiber-node. The loop iterations in all these cases are being performed using nested for loops as in the case of fluid-nodes.

The performance analysis of the serial version reveals that the LBM part for computing Distribution Function DF1 of a fluid node takes the maximum time. This is due to the fact that every fluid node is being visited in a given iteration which

happens to be large input of 124x64x64 fluid elements. But in a given iteration, only few fluid nodes which lie in the periphery of 18 directions as shown in fig 3.3 are influenced. LBM-IB software intends to utilize the memory capacity of a system and store those values in different buffers (refer Figure 3.6). This has made possible to share the data across different resources within a node. Moreover, the initial computation requires them to access some global data which is made available to every thread via "*gv*" object. The same is applicable for the fiber-sheet, where the fiber-nodes calculate the elastic forces based on the position of fiber-nodes which is known through fiber_shape instance in *gv*. Following OpenMP version, a cube-based data centric version has also been developed to achieve a better performance than OpenMP version. As elucidated in fig 4.1, the use of *gv* object makes it possible to share the data across all the available resources of a system.

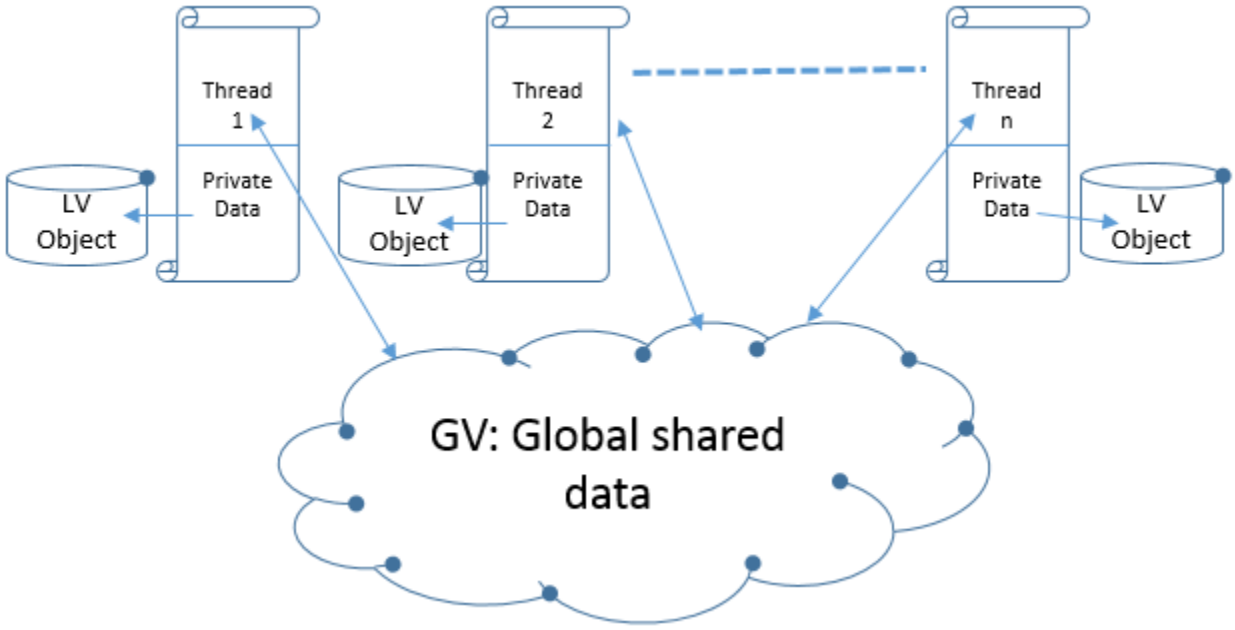


Figure 4.1. Every thread has an access to its private data via LV and global data via GV.

4.1 OpenMP LBM-IB Version

The programming model of OpenMP is based on "fork-join" model [24], where a master thread is responsible for spawning stipulated amount of threads on encountering a pre-processing directive which is ***"pragma omp parallel"*** for a C binding. After the loop ends, the master thread joins the remaining threads in a synchronized manner [24]. OpenMP offers programmers a rich set of constructs which are useful in designing a complete multithreaded shared memory programming software. To elaborate, these constructs are helpful in [24]

- sharing the data among different threads,
- allocating tasks to individual threads via "Work-sharing"
- distinguishing between private and public data among threads,
- scheduling the flow of iteration for example "static","dynamic" or "guided",
- synchronizing the threads after completion of tasks and
- managing run time environment variables like finding a thread id, or setting up number of threads.

Barring "Work-Sharing",OpenMP LBM-IB utilizes the aforesaid constructs for both fluid and fiber-nodes. The underlying Data structure used for serial version has been intentionally designed in a manner so that it can be used for parallelizations for later versions of LBM-IB. OpenMP LBM-IB uses the same set of Data-structure specified in Fig 3.5, 3.6 and 3.7. The loop iterations for both fiber-sheet and fluid-grid are identified and then data dependencies inside those loops are analyzed to use appropriate "data sharing attribute clauses" [24] provided by OpenMP. The changes done in OpenMP LBM-IB from serial version are enumerated as follows:

- **Input Changes**: Along with the fiber-sheet's and fluid-grid's parameter from the user as specified in Algorithm 1 another additional input is taken in the

form of command line argument, which specifies the number of threads that should be launched once the **pragma** directive is detected. This user provided number of threads is then set as run time environment variable using ***"omp_set_num_threads"***. The user should selectively choose this parameter relative to the system configuration. It would not be wise to launch more number of threads than supported by the underlying hardware configuration of the system. For instance, on Dragon (Refer Table 4.1), which supports 12 threads per cpu core, the maximum parallelization level that can be achieved is for 12 cores and hence the range for this parameter should lie between 2-12 for this multicore machine.

- **Fiber-Node Parallelization**: It can also be called as IB parallelization, since it deals with the immersed boundary or in other words fiber-sheet's computations done in serial version. The same are carried out in ***compute_bendingforce***, ***compute_stretchingforce***, ***compute_elasticforce***, ***find_ifd_and_SpreadForce*** and ***moveFibersheet***. As mentioned before for force calculation, every fiber node is visited twice along either directions. "Parallel Construct" ***"#pragma omp parallel for"*** is used to alert the underlying system to spawn specified number of threads for this loop. The fiber-node array(FN) in a particular row or column and it's corresponding row (i) and column index (j) are kept private to each thread to avoid race condition. "Static" scheduling mechanism is adopted which allows allocations of iterations to all threads before the actual computation in the loop starts [24]. The same is elucidated in the following algorithm in which tf_c and tf_r indicates the total number of fiber elements along column and row.

Algorithm 2 OpenMP LBM-IB Fiber-sheet Parallelization: Input(tf_r, tf_c)

/* Along fiber row*/

#pragma omp parallel for default (shared) private(FN,i,j)

for $i \leftarrow 0$ to tf_r **do**

Handle special boundary conditions for fiber-nodes

for $j \leftarrow 0$ to tf_c **do** $FN_{i,j}$.force $\leftarrow$ $FN_{i,j^{th}neighbours}$.force;

end for

end for

/* Along fiber column*/

#pragma omp parallel for default (shared) private(FN,i,j)

for $j \leftarrow 0$ to tf_c **do**

Handle special boundary conditions for fiber-nodes

for $i \leftarrow 0$ to tf_r **do** $FN_{j,i}$.force $\leftarrow$ $FN_{j,i^{th}neighbours}$.force;

end for

end for

The local pool of private data accessible to each thread is very less to those being parallelized for ***find_ifd_and_SpreadForce*** and ***moveFibersheet*** functions. In these functions, apart from the fiber-node structure (FN), the influenced region along three directions(IFD_x, IFD_y, IFD_z), distance between the fiber-node and all fluid nodes in the influenced region(temp_dist), the corresponding index for a specific fluid node in the influenced region (idx_fluidnode) are kept private to each thread. Thus, the **pragma** construct in the above algorithm changes to ***#pragma omp parallel for default (shared) private(FN, IFD_x, IFD_y, IFD_z, temp_dist, idx_fluidnode))*** and the calculation is carried for an individual fiber node in both the directions together.

- **Fluid-Grid Parallelization**: The kernels or functions involved in these parallelizations are ***compute_eqlbrmdistrfuncDF1***, ***stream_distrfunc***,

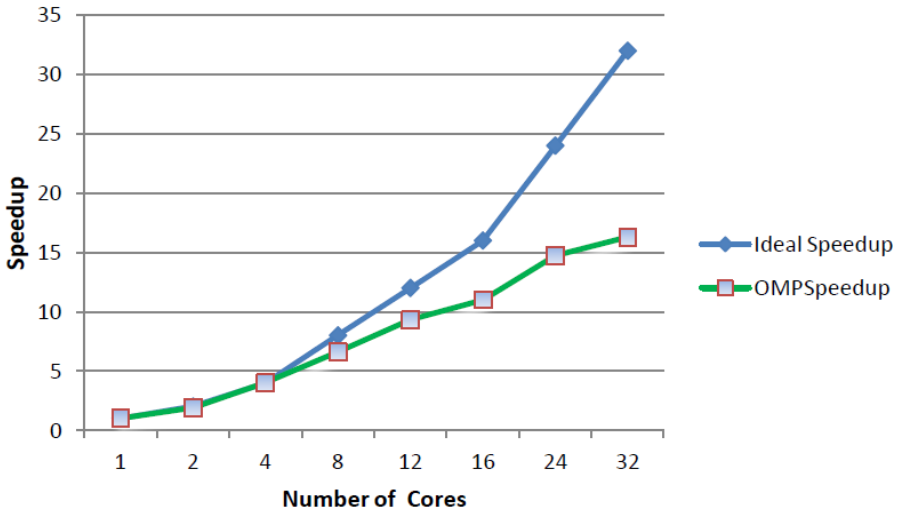
bounceback_rigidwalls, ***compute_rho_and_u***, ***copy_buffer's_DF***, ***copy_DistributionFunction*** and ***PeriodicBC***. As discussed before, every fluid nodes is visited by first visiting the grid surface and then the other two dimensions. From parallelization point of view, the data sharing has to be taken care as done above for fiber-sheet. The functions stipulated above exhibit a similar behavior in terms of iterations and therefore, share almost similar parallel pragma construct. If $elem_x$, $elem_y$ and $elem_z$ represents the number of fluid-nodes in x,y and z dimensions then the parallel construct being used is ***# pragma omp parallel for default(shared) private(***$elem_x$***,*** $elem_y$***,*** $elem_z$ ***)***. It changes slightly while computing the particle collision in ***compute_eqlbrmdistrfuncDF1*** and updating the fluid's properties of ρ and velocities where the thread has their own local copy of the current value of ε representing the direction and partial sums from different directions for the two functions respectively. The overall algorithm for this parallelization is given by following algorithm in which $elem_x$ also represents the total number of fluid surfaces as well as the total fluid-nodes along x direction (Refer Figure 3.2).

Algorithm 3 OpenMP LBM-IB FluidGrid Parallelization Input($elem_x$, $elem_y$, $elem_z$)

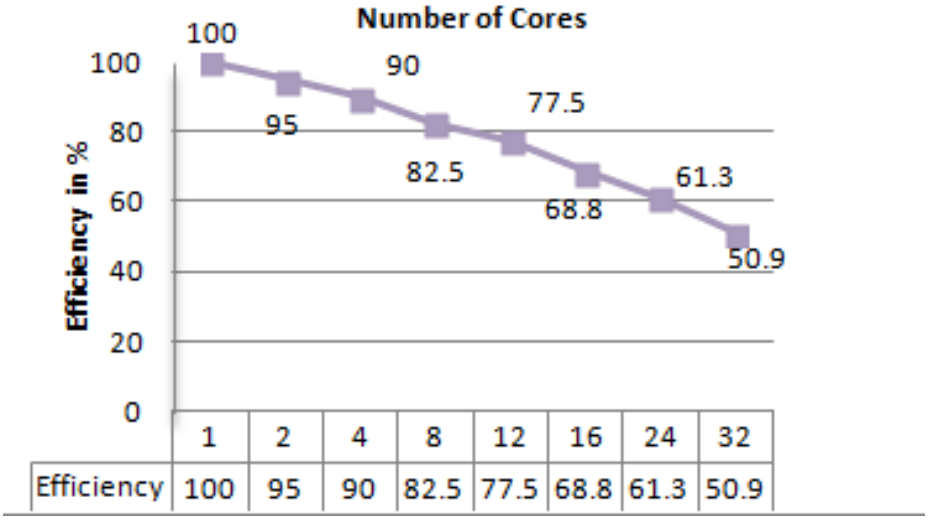
#pragma omp parallel for default (shared) private($elem_x$**,** $elem_y$**,** $elem_z$**)**
for $i \leftarrow 0$ to $elem_x$ **do**
  for $j \leftarrow 0$ to $elem_y$ **do**
    for $k \leftarrow 0$ to $elem_z$ **do**
      for $\varepsilon \leftarrow 0$ to 18 **do**
        $Fluid\_Node_{ijk}.[\varepsilon] \leftarrow$ function($Fluid\_Node_{i,j,k}.[\varepsilon]$);
      end for
    end for
  end for
end for

4.2 Performance Evaluation: OpenMP LBM-IB version

BigredII, which supports high performance super-computing at Indian university was used to evaluate the performance of OpenMP LBM-IB version. Big Red II is a Cray XE6/XK7 supercomputer with two AMD Opteron 16-core Abu Dhabi 2.9 GHz CPUs and memory of 64 GB. [30] (Refer Table 4.2). The experiments were conducted for fixed input size of 124x64x64 fluid grid and 20x20 fiber-sheet made up of 52 fiber-elements in either directions. The experiment was done for 1000 time-steps for both serial version and OpenMP LBM-IB version. As shown in fig 4.2, the speedup is fairly good till the number of cores are 8 but it drops as the number of cores are increased further. A similar experiment was conducted on another Linux Machine (Dragon) belonging to Math Department at IUPUI. It is an Intel(R) Xeon(R) CPU model family supporting 12 cores 2.80GHz (Refer Table 4.1). The results were quite similar as shown in fig 4.3. Here also the parallel efficiency drops as the number of cpu cores increases.

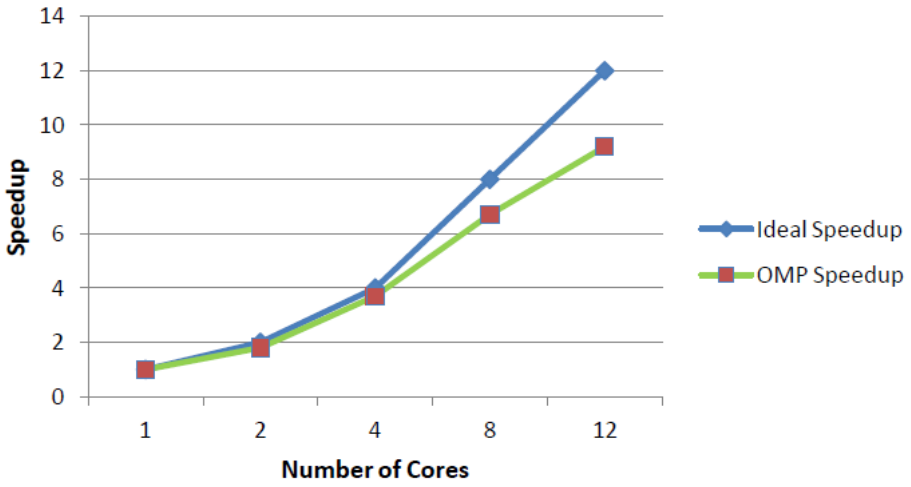


(a) OpenMP Speedup compared with ideal speedup which is equivalent to execution time of one core.

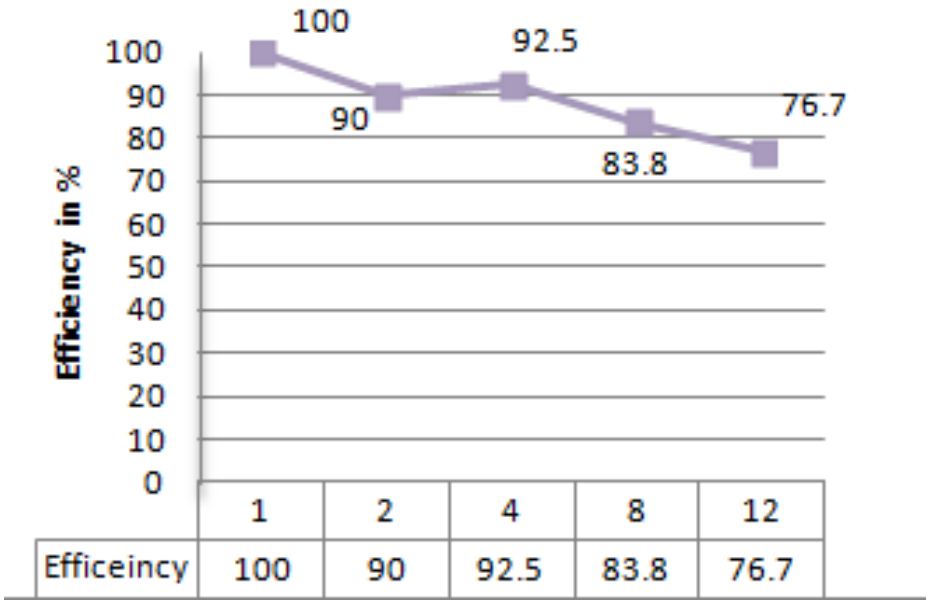


(b) Parallel Efficiency relative to ideal speedup.

Figure 4.2. OpenMP LBM-IB Performance Evaluation on BigredII.



(a) OpenMP Speedup compared with ideal speedup which is equivalent to execution time of one core.

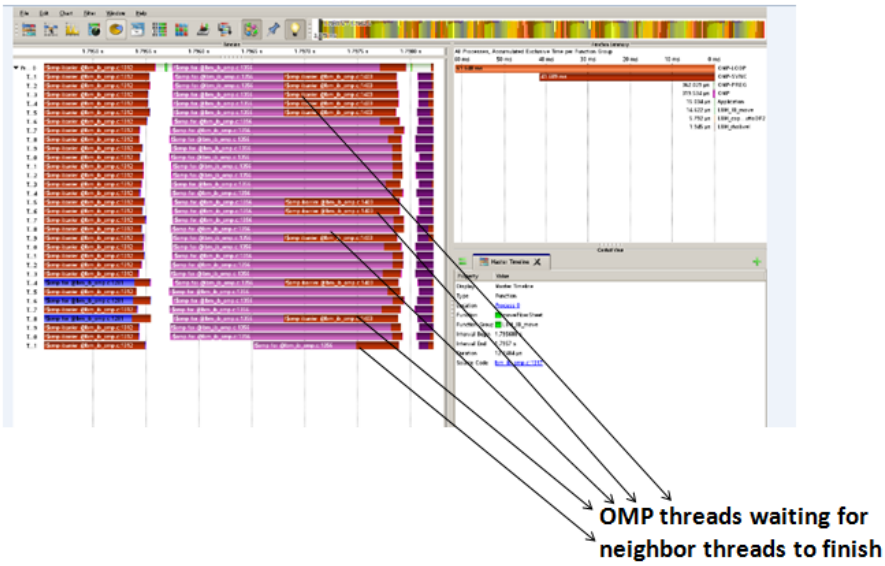


(b) Parallel Efficiency relative to ideal speedup.

Figure 4.3. OpenMP LBM-IB Performance and Parallel Efficiency on Dragon.

In order to analyze the performance degradation relative to the cpu cores; Vampir instrumentation [31], OpenMP profiling [32] and PAPI interface [33] were used. From

vampir results it was evident that there is a load imbalance for certain functions in which certain omp threads were waiting for other threads to finish their tasks as shown in 4.4(a). Also, it was found that though L1 data cache miss rate is considerably low, OpenMP LBM-IB exhibits a high L2 miss rate as shown in 4.4(b). It also shows high load imbalance when the number of cores increases beyond 8.



(a) Vampir Instrumentation on BigredII: Depicts load imbalance for a function in OpenMP LBM-IB.

Cores	L1 miss rate	L2 miss rate	Load Imbalance
1	1.76%	26.10%	0%
2	1.75%	26.10%	1.80%
4	1.75%	26.10%	1.40%
8	1.75%	26.20%	5.10%
16	1.74%	27.10%	11%
32	1.76%	27.60%	13%

(b) Performance Metric Data for OpenMP LBM-IB.

Figure 4.4. Profiling Results for OpenMP LBM-IB.

To overcome the aforesaid limitations of the parallel version, a new modified data centric algorithm based on Pthreads is developed. The same is discussed in the following section.

4.3 Pthread Version: Block Distribution

OpenMP LBM-IB version has shown good results for less number of cpu cores but as the cores are increased parallel efficiency drops considerably. From the profiling and instrumentation done, it was found that OpenMP LBM-IB suffers from load imbalance and less data locality, as many threads are idle in a given parallel omp construct. Therefore, to better utilize the availability of idle resources, a data centric block distribution based LBM-IB method is designed in which parallelization is achieved by pthreads. It differs significantly from OpenMP version as now it is required to manually distribute the threads and synchronize them to achieve correct results in a faster way unlike OpenMP version. The most important changes for this version is the change in treatment of the data-structure and user-defined distribution function to address those changes. It is called as Block distribution because the threads are now being limited to a sub-portion of the fluid-grid also called **"cube"**, is described in more detail in the following section. This block distribution version of LBM-IB can also be addressed as cube based pthread LBM-IB.

Block Distribution: Cube Based Pthread version

The gnu profiling on serial LBM-IB and vampir instrumentation on OpenMP LBM-IB revealed that the particle collision operation or the ***compute_eqlbrmdistrfuncDF1*** is the most expensive kernel taking maximum time in a given time step. This happens to be logically correct as well, since the number of fluid-nodes are very large as compared to the immersed structure. To address the problem statement for the LBM-IB algorithm, this is a basic requirement which makes fluid grid sufficiently larger than the immersed structure [1]. At any instant of time,

the major computation surrounds the immersed structure and the fluid-nodes lying in its influential domain. So if we divide the fluid-grid in small cubes and allocate those cubes to specific threads, the data-locality in a thread neighborhood will increase and load-imbalance will be reduced. To achieve this distribution, 3D fluid grid is first divided into 3D stack of regular sub-grids also called cubes. If the fluid-grid is made up of $elem_x$, $elem_y$ and $elem_z$ fluid-nodes, then the entire fluid-grid is decomposed into $\frac{elem_x}{k}$ x $\frac{elem_y}{k}$ x $\frac{elem_z}{k}$ cubes. 'k' is the dimension of the cubic sub-grid, thus, every individual cube has k x k x k fluid nodes."k" is a user provided parameter and is very crucial in evaluating boundary conditions for some functions. These fluid nodes are stored in contiguous memory block and the same is implemented in ***gen_fluid_grid*** function. Along with "k", user also specifies the number of threads to be used in the simulation. This parameter is taken in the form of P, Q and R variable which describes the dimensions of the thread grid (as shown in Fig 4.6) and the total number of threads such that total threads equals P x Q x R. To have a non-overlapping distribution of threads for a given cube, a restriction is imposed on P, Q and R such that P and $elem_x$ should be divisible by each other, likewise for Q & $elem_y$ and R & $elem_z$.

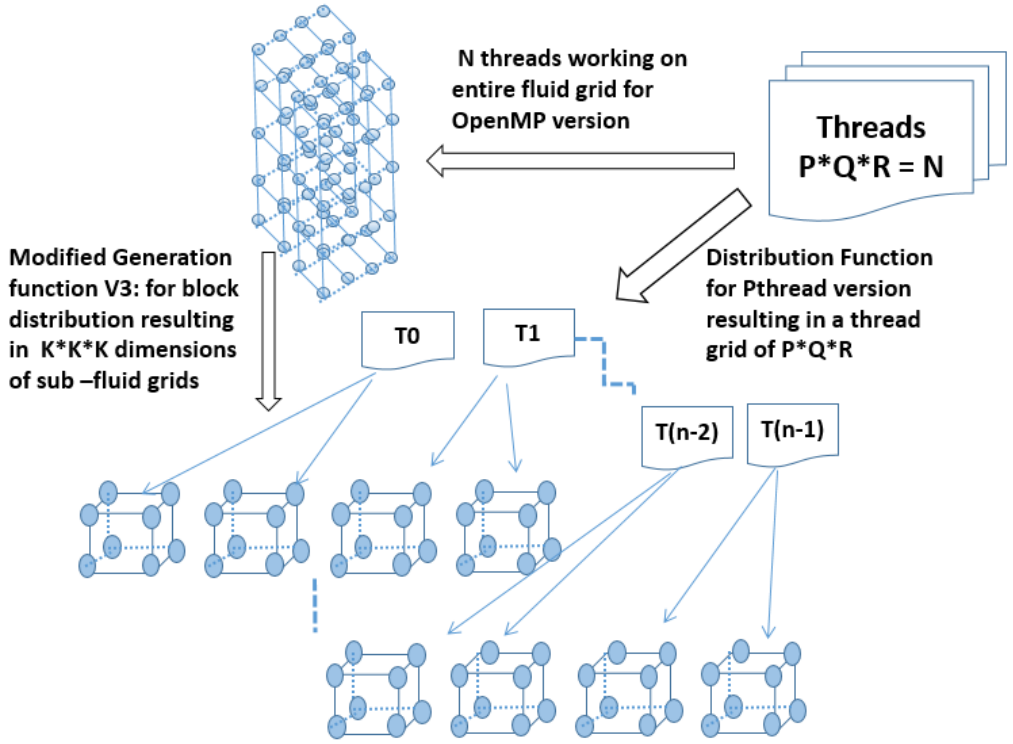


Figure 4.5. Comparison of thread distribution in OpenMP and in Pthread version: In cube based Pthread version threads are local to sub-fluid grid of K x K x K dimension.

The distribution of threads to individual cube is done via ***cube2thread*** function which returns the thread id allocated for individual cubes. Every cube is assigned an index in three dimensions of x,y and z based on the total number of cubes and the number of fluid elements in that direction. Let this be denoted by $cidx_x$, $cidx_y$ and $cidx_z$. Then the distribution function returns the thread id given by the following equation 4.1 as:

$$Thread_{id} = \frac{cidx_x * P}{elem_x} * Q * R + \frac{cidx_y * Q}{elem_y} * R + \frac{cidx_z * R}{elem_z} \tag{4.1}$$

A similar distribution is achieved for fiber-sheet as well in which an array of fiber-nodes for instance all fiber-nodes lying on a particular range of rows are mapped to

a thread. This is implemented via function ***fiber2thread***. For fiber-nodes lying on i^{th} row, thread id is calculated in following way:

$$Thread_{id} = \frac{fiber_i * P * Q * R}{total\_fibers_{row}} \tag{4.2}$$

The above distribution can be better understood from the fig 4.6 in which the fluid grid is mapped to individual cubes via the aforesaid distribution functions to a P x Q x R thread grid where P, Q and R are 3.

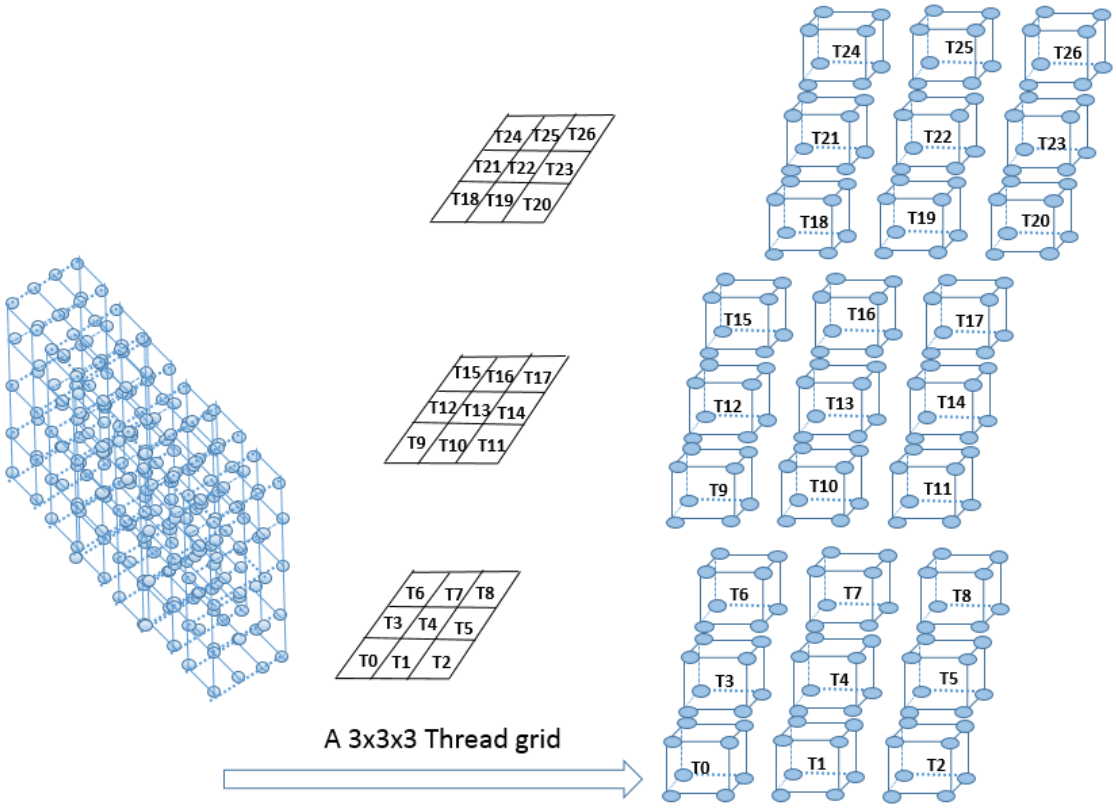


Figure 4.6. A fluid grid is mapped to individual cubes via 3x3x3 thread grid. Every thread owns an individual cube after mapping via distribution functions. Here P = Q = R = 3.

Pthread Parallelization & Algorithm:Cube Based LBM-IB

Pthread programming model is based on the shared memory model shown in fig 4.1. Every thread has an access to the shared data besides it's local data which is private to the thread. [25]. Unlike OpenMP version, where the parallel construct took care of the thread creation and synchronization, in this version Pthread APIs are used. Threads are created using ***pthread_create*** and synchronization is ensured via ***pthread_join***, ***pthread_barrier_wait pthread_mutex_lock/unlock*** interfaces. Also, distribution function discussed in equations 4.1 and 4.2 ensures that their is no overlapping in thread allocation. Apart from the data-structure used in LBM-IB serial version, a new data structure (LV) for the local object specific to individual thread is defined as shown in fig 4.8 and GV also has additional attribute to ensure thread safety and synchronization. Apart from storing thread related attributes of pthread object, mutex lock (lock_fluid and barrier object), gv also stores total number of threads provided by the user, cube_size representing the dimension of the individual cubes (shown as 'K' in fig 4.5) and the total number of cubes in x, y and z direction represented by num_cubes_x,num_cubes_y and num_cubes_z respectively. The variation in GV data structure from the LBM-IB serial version are shown in bold in Figure 4.8.

The important change to support Block Distribution is the change in the Fluidgrid Data structure as shown in Fig 4.7. A new data structure called sub fluid grid is created which represents the innermost cube of the fluid grid, which contains the microscopic Fluid node unlike the serial version in which every surface had the array of fluid-nodes. The same is accessed via a pointer in Fluid-grid. A fluid grid can be visualized as uniform integration of these smaller sub-fluid grids or cubes.

```
typedef struct fluid_grid_t {
 int         x_dim;
 int         y_dim;
 int         z_dim
 Fluidsurface*  inlet;   //with constant values*/
 Fluidsurface*  outlet;  //with constant values*/
 Sub_Fluidgrid* sub_fluid_grid;
} Fluidgrid;
```

```
typedef struct sub_fluid_grid_t {
Fluidnode*    nodes;
 int         grid_dim_x;//elem_x in inner cube =k
  int        grid_dim_y; //elem_y in inner cube =k
  int        grid_dim_z; //elem_z in inner cube =k
} Sub_Fluidgrid;
```

Figure 4.7. Modified Fluidgrid data structure to support block distribution.

```
typedef struct gv_t {
  /* The immersed structure */
  Fibershape* fiber_shape;
  /* The fluid grid */
  Fluidgrid* fluid_grid;
  /* Constant parameters used in LBM-IB */
  double    tau, nu_l, u_l, rho_l, L_l, g_l, Ks_l, Kb_l;
  double    Re, cs_l, Kshat, Kbhat, Fr;
  int       dt, time, TIME_STOP;
  double    c[19][3];              //stores 19 different velocities for ksi
  int       ib, ie, je, jb, ke, kb;    //For Fluid Grid's actual computation part
/*Added for Cube Based Pthread version */
pthread_t *threads;
pthread_mutex_t *lock_Fluid;
int total_threads, cube_size, P,Q,R; //cube_size is K
int num_cubes_x,  num_cubes_y, num_cubes_z;
pthread_barrier_t barr;
}* GV;
```

```
typedef struct lv_t{
  int tid;
  GV  gv;
}* LV;
```

Figure 4.8. GV and LV Data structure to accommodate Pthread constructs and support block distribution.

A unique thread object, an attribute object, name of the function to be threaded and arguments to that function which is threaded are passed as an argument to **pthread_create** interface [25]. Steps 1-12 described In Algorithm 1 are now passed to a new function called ***do_thread*** which is called from main thread as shown in Algorithm 4. The input to this algorithm are same as that described earlier with addition of cube dimension K, thread governing parameters P, Q and R. ***do_thread*** routine can be considered as a Thread entry function where stipulated amount of threads start the LBM-IB simulations as shown in algorithm 5. It takes the local thread specific object wrapped as v and starts the simulations in a parallel synchronized manner.

Algorithm 4 Main Function: Input:(fsh_w, fsh_h, tf_r, tf_c, fl_x, fl_y, fl_z, fs_{x0}, fs_{y0}, fs_{z0}, K, P, Q, R)

fiber_shape = gen_fiber_shape(fsh_w, fsh_h, tf_r, tf_c, fs_{x0}, fs_{y0}, fs_{z0});

fluid_grid = gen_fluid_grid(fl_x, fl_y, fl_z, K);/*Cube size dimension K being passed*/

total_threads ← P*Q*R;

gv← total_threads, gv← $lock_{obj}$;

for $i \leftarrow 0$ to total_threads **do** pthread_mutex_init(&$lock_{obj}$[i], NULL);
end for

pthread_barrier_init(&$barr_{obj}$, NULL, total_threads); gv← $barr_{obj}$;

/*Do rest of the initializations as in Algorithm 1*/

pthread_t *threads; ▷ Creating a Pthread object

gv ← threads;

for $i \leftarrow 0$ to total_threads **do**

lv_{tid} ←i;

$lv_i \leftarrow$ gv;

$pthread\_create(threads + i, NULL, \textbf{do\_thread}, lv + i)$;
end for

for $i \leftarrow 0$ to total_threads **do** $pthread\_join(threads[i], void*)$;
end for

$pthread\_exit(NULL)$;

A thread has an access to the shared global object *gv* and is assigned a unique thread id as depicted in Algorithms 4 and 5. Once, a thread starts it carries out the simulations of LBM-IB in an organized fashion. Based on the thread id returned by the distribution functions all threads starts the fluid- flexible structure interaction for a given time step restricted by TIME_STOP. Unlike serial version, now every routine works on a distributed set of fiber-nodes or fluid-subgrids or cubes and *lv* object specific to a thread is passed to those routine. These parallelizations can be enumerated as below:

1. IB parallelization is achieved first in which the elastic forces of fibers are computed. Equation 4.2 governs the thread allocation for an array of fiber-nodes. So, a group of threads start computing the bending and stretching forces in ***compute_bendingforce*** and ***compute_stretchingforce*** routines. In both force calculation schemes, a barrier is required between the computations of vertical and horizontal directions for fiber-nodes, which was guaranteed by a **omp parallel for** construct in OpenMP version. Then this forces are added together in ***compute_elasticforce*** which can be considered embarrassingly parallel routine. ***fiber2thread*** distribution ensures that only a range of fiber-nodes are distributed to a thread in a synchronized manner. Then another barrier after elastic force computations ensures that all distributed threads finish their computations before spreading the force to the fluid-nodes. While spreading the forces to the influenced fluid-nodes in ***find_ifd_and_SpreadForce***, mutex objects provided by pthread library are used. The lock is required because more than two threads may try to spread the forces on same fluid-node, since the influenced region for a fiber-node is of 4 x 4 x 4 size as shown in the figure 4.9. Every thread has a lock to protect its cubes in influenced region. If other threads want to access those cubes, then they will try to acquire the same lock unique to every thread before spreading.

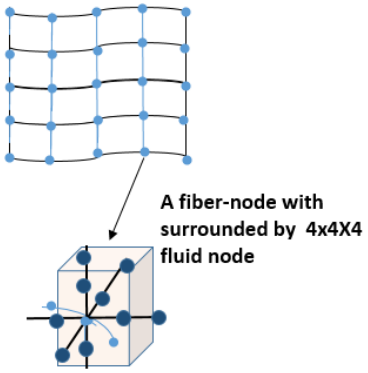


Figure 4.9. A fiber-node is surrounded by a 4x4x4 cubic fluid grid which forms the influential region for that fiber node.

2. After spreading the forces, every cube is visited to calculate the particle collision or Distribution function value DF1 for fluid nodes lying in the cube implemented in ***compute_eqlbrmdistrfuncDF1***. The distribution via ***cube2thread*** ensures that the cubes belonging to intended threads are only visited. After computing DF1, a barrier call ensures that all threads have completed their collision calculation and are now ready to stream those values in neighborhood of 18 fluid-nodes as shown in Fig 3.3. This is done in ***stream_distrfunc***, which becomes little tricky to compute as the local indices within one cube which represent the actual fluid-nodes imparts a boundary for other cubes. To elaborate, if every cube is 4 x 4 x 4 dimension, then the local indices for those cubes will range from 0-3 in x, y and z direction. Now, for the fluid-nodes lying on the boundary i.e at 0^{th} and 3^{rd} position, we need to pass appropriate values to the neighboring cubes. For instance, if streaming is done for $\varepsilon = 1$, then 'I' index of the cube changes to I+1 and local index value changes to the beginning index i.e. 0 for $(I+1)^{th}$ cube as shown in Figure 4.10. Similar boundary condition checks are applied for all different directions of 'ε'.

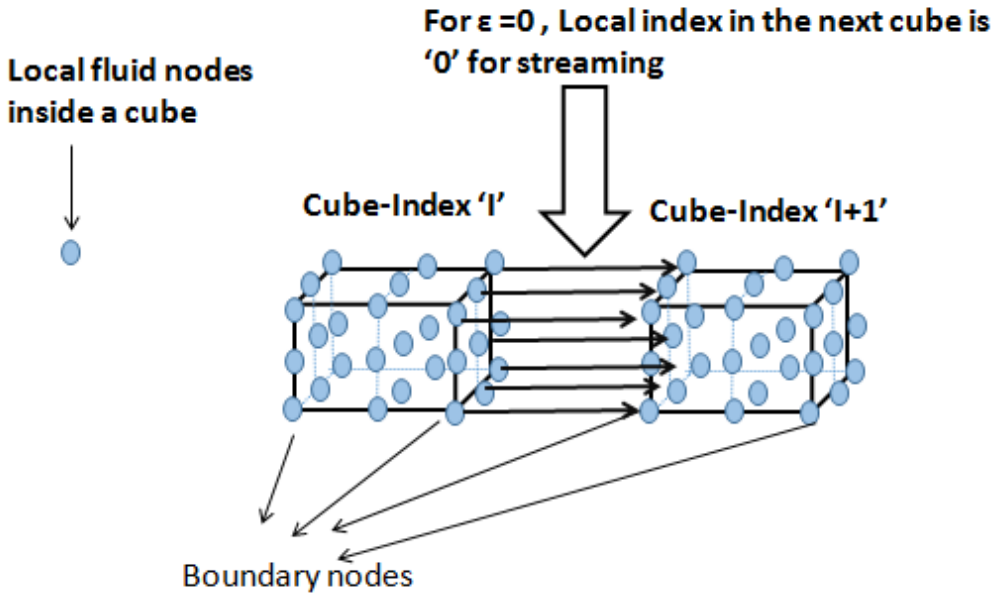


Figure 4.10. A simple case of streaming in which the DF values are streamed to neighboring nodes in the next cube for 'ε' =0.

3. Next simulation step is to ensure that the boundary fluid nodes on the rigid walls are treated properly as in serial version of LBM-IB implemented in ***bounce-back_rigidwalls***. In this function very few cubes lying on the periphery of the original fluid-grid participate. The threads belonging to a specific cube distributed via ***cube2thread*** will work on copying the buffers from DF1 to DF2 as described in 3.1. Another barrier after this function ensures that now all fluid-nodes in all the sub-fluid grids or cubes can have an updated velocity and ρ value.

4. Then, the threads work on their local cubes assigned by ***cube2thread*** to update the velocity and ρ values for the fluid-nodes belonging to their cubes. This is implemented in ***compute_rho_and_u***. Then, before moving the fiber-sheet a barrier is introduced to have the mesoscopic properties updated for all fluid-nodes.

5. This is the final step of fluid-structure interaction in which the fiber-nodes extracts the updated velocities of the fluid-nodes in the influenced region. Note that here, we do not need to lock the fluid-nodes as the threads are going to read the data from the fluid-nodes and hence at a time a fluid-node can be accessed by any thread. This is implemented in ***moveFibersheet***.

6. Then the regeneration functions namely ***copy_buffer's_DF***, ***copy_DistributionFunction*** and ***PeriodicBC*** explained in 3.1.4 ensures the continuity of LBM-IB simulations as for serial version. These routines work only on the fluid nodes and hence use only ***cube2thread*** for distribution.

The entire process is outlined in Algorithm 5. Several barriers enumerated in above steps are not shown in the algorithm for simplicity.

Algorithm 5 do_thread: Input:(void* v)

/*Every local thread gets access to gv and is identified by unique thread id*/

lv ← v; lv_{gv} ← gv; tid ← lv_{tid};

while $time <= TIME\_STOP$ **do**

for every $fiber_i$ **do**

if fiber2thread($fiber_i$)==tid **then**

1)$compute\_bendingforce(lv)$;

2)$compute\_stretchingforce(lv)$;

3)$compute\_elasticforce(lv)$;

4)$find\_ifd\_and\_SpreadForce(lv)$; ▷ mutex lock used for fluid inside

end for $pthread\_barrier\_wait()$;

for every $Fluid - Cube_{I,J,K}$ **do**

if cube2thread($Fluid - Cube_{I,J,K}$)==tid **then**

5)$compute\_eqlbrmdistrfuncDF1(lv)$;

6)$stream\_distrfunc(lv)$;

end for $pthread\_barrier\_wait()$;

for every $Fluid - Cube_{I,J,K}$ **do**

if cube2thread($Fluid - Cube_{I,J,K}$)==tid **then**

7)$bounceback\_rigidwalls(lv)$;

end for

$pthread\_barrier\_wait()$;

for every $Fluid - Cube_{I,J,K}$ **do**

if cube2thread($Fluid - Cube_{I,J,K}$)==tid **then**

8)$compute\_rho\_and\_u(lv)$;

end for $pthread\_barrier\_wait()$;

for every $fiber_i$ **do**

if fiber2thread($fiber_i$)==tid **then**

9)$moveFibersheet(lv)$;

end for $pthread\_barrier\_wait()$;

for every $Fluid - Cube_{I,J,K}$ **do**

if cube2thread($Fluid - Cube_{I,J,K}$)==tid **then**

/*Call regeneration functions */ ▷ Refer Algorithm 1

end for $pthread\_barrier\_wait()$;

end while

4.4 Performance Evaluation:Cube Based Block Distribution

In order to evaluate the performance of the cube-based distributed algorithm, an initial set of experiments were conducted on Dragon and BigredII (Refer Tables 4.1 and 4.2). The input for both the versions in the experiments were same and even with a change in 'K' i.e the cube-size of sub-fluid grid ,both OpenMP version and the Pthread version were comparable in performance for less number of cores. To better analyze the performance of the new algorithm, a series of experiments were conducted on Thog System (Refer 4.3), which is a 64 core AMD system located at University of Tennessee, Knoxville.

Table 4.1

Dragon System

System details for Dragon: A linux machine owned by Math department at IUPUI	
Parameter	Description
Processor Type	Intel(R) Xeon(R) X5660 2.80GHz
Number of Processors	2
Number of Cores	24
Sockets	2
L1d Cache	32 K
L1i Cache	32 K
L2 Cache	256K
L3 Cache	12288K
Number of NUMA nodes	2
Cores per NUMA node	12 x 2 each shared by a processor
OS	Linux 2.6.32
Compilers	gcc 64 bit 4.4.3

Table 4.2

BigRedII

System details for BigRedII: A HPC supercomputer owned by Indiana University	
Parameter	Description
Processor Type	AMD Opteron(TM) Processor 2.5GHz
Number of Processors	2
Cores per Processor	16 x 2 , shared by each processor
Number of Cores	32
Sockets	2
L1d Cache	16 K
L1i Cache	64 K
L2 Cache	2048K
L3 Cache	6144K
Number of NUMA nodes	4
Cores per NUMA node	32
OS	Linux 2.6.32
Compilers	gcc 64 bit 4.3.4

Thog is a **manycore** system supporting 16 cores distributed across four AMD processors. As shown in Table 4.3, on every processor L2 cache is being shared by 2 cores whereas 8 cores share L3 cache. the overall memory of the entire system is 256GB. As depicted in Table 4.3, manycore system provide more NUMA nodes when compared to other systems being used earlier (Dragon and BigRedII). NUMA stands for "Non uniform Memory access" and system designed with high NUMA nodes tend to utilize data locality feature of the software in a better way as the time required to access memory locations that are shared by other NUMA nodes or local to other NUMA nodes is more than the nodes residing in the same memory or local to a NUMA node [34]. As shown in table 4.4, this access time can be at most 2.2 times

Table 4.3

Thog System

Experimental system for Cube Based Pthread and OpenMP LBM-IB versions comparison	
Parameter	Description
Processor Type	AMD Opetron 6380 2.5 GHz
Cores per Processor	16
L1 Cache	16 KB per core
L2 unified Cache	8 x 2 MB, each shared by two cores
L3 unified Cache	2 x 12 MB, each shared by eight cores
Number of Processors	4
Number of NUMA nodes	8
Cores per NUMA node	8
Memory per NUMA node	32 GB
OS	Linux 3.9.0
Compilers	gcc 64bit 4.6.3

longer than accessing the local node's memory. This helped to correctly evaluate the improvement in data locality offered by the new design.

Another important factor in performance improvement is the consideration of weak scalability. The new algorithm works well if the respective cores have enough data to compute. i.e. if we increase the number of cores and increase the input size; which happens to be the fluid grid elements, then cube based Pthread version showed around 53% improvement over OpenMP as shown in Figure 4.11. For every increase in the number of cores in the experiment, the number of fluid nodes are increased accordingly for both OpenMP and Pthread version.

The fiber-sheet elements are kept to be uniform of 104x104 for all the experiments, but the fluid grid size is increased with the increase in cpu cores. For example if the

Table 4.4
Node Distance between 8 Different NUMA nodes: using "*numactl − hardware*"

nodeid	0	1	2	3	4	5	6	7
0	10	16	16	22	16	22	16	22
1	16	10	22	16	22	16	22	16
2	16	22	10	16	16	22	16	22
3	22	16	16	10	22	16	22	16
4	16	22	16	22	10	16	16	22
5	22	16	22	16	16	10	22	16
6	16	22	16	22	16	22	10	16
7	22	16	22	16	22	16	16	10

single core took 128 x 128 x 128 fluid node, then for dual core the input was changed to 256x128x128, which changes to 256x256x256 for eight core experiment and so on.

Figure 4.11 illustrates the relative change in execution time of the two versions (cubed LBM-IB) and (OpenMP LBM-IB) with the change in number of cores on *Thog*. As evident, for same workload, the new block Distribution based algorithm shows an improvement over OpenMP as the number of cores increases. The execution time for OpenMP increases more exponentially than cube based Pthread version. For OpenMP, it increases by 25% from dual to quad core, by 36% from 4 to 8 cores , by 22% from 8 to 32 cores and from 32 to 64 cores, it increases at very high rate of 42%. Whereas, the execution time for Pthread version increases linearly at 13% from 2 to 32 cores and for 32 to 64 cores it increases only by 18%. For 64 cores, the cube based block distribution wins over the OpenMP version which lacks data locality feature. As shown, cube based Pthread version is able to outrun OpenMP by 53% on 64 cores.

In an ideal case, with increase in number of cores and input, the execution time for both versions should not vary much. But for OpenMP version the rate at which

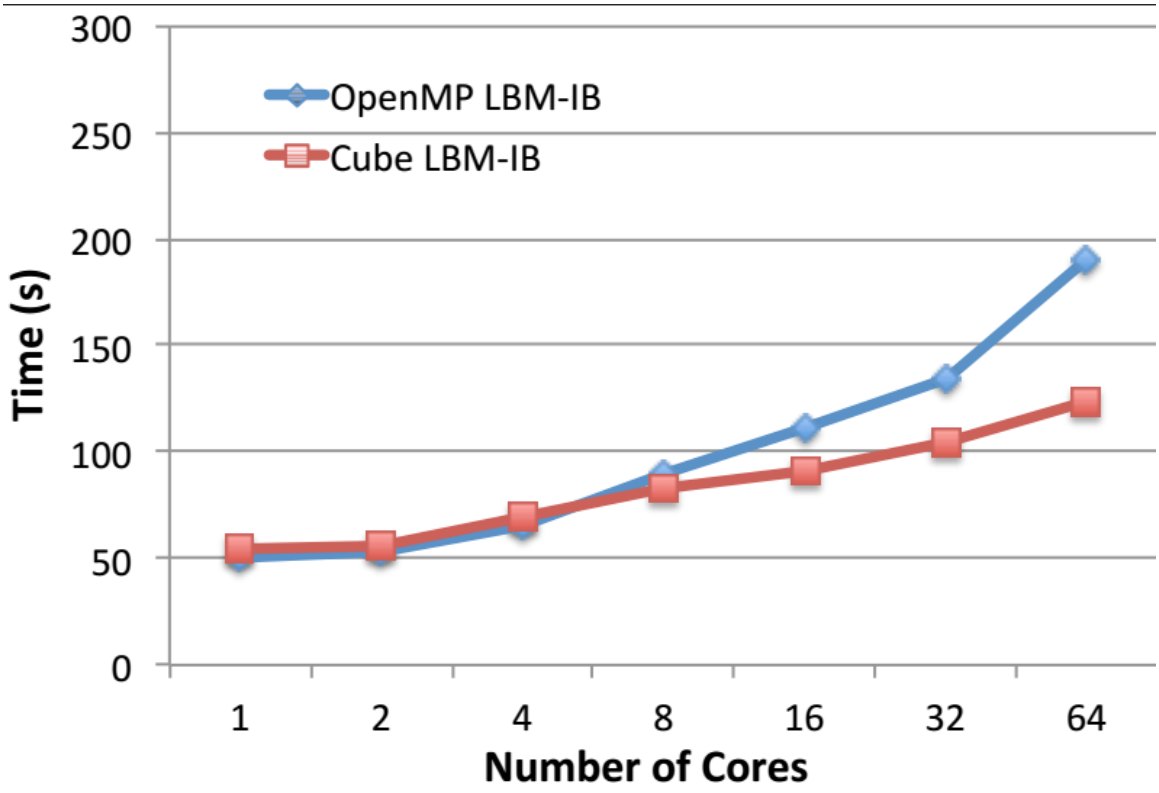


Figure 4.11. Cubed Algorithm is 53% faster than OpenMP considering Weak scalability.

execution time increases is more than that by the cube based Pthread version. The execution time increases with an increase in number of cores because the thread synchronizations constricts memory bandwidth provided by hardware even though every core works on a constant workload. Therefore, the data centric feature of the new algorithm which makes better use of the available resources in an optimal manner, does not over-exhaust the memory bandwidth and surpasses OpenMP for higher number of cores.

5 LBM-IB HYBRID MPI/PTHREAD DISTRIBUTED MEMORY VERSION

Hybrid Programming has become an inspirational parallel programming paradigm for libraries developed for High performance Computing. As discussed before, many software libraries built on Hybrid approach outperform their shared memory version in certain situations [26]. Load Imbalance being one of them, this Hybrid version of LBM-IB aims at eliminating earlier limitations of OpenMP and Pthread versions and also provide for the first time a library for extreme scale distributed memory manycore systems in areas of LBM-IB. Most of the existing hybrid approaches first distribute the work to different nodes or a multicore system and then share the resources available on that multicore or manycore system. This hybrid version of LBM-IB is also built on similar approach. Though not an optimal solution, this version of LBM-IB is the first approach in combining LBM-IB together for the first time. It provides MPI interfaces on top of the existing cube based Block distribution which uses pthread library for parallelizations in shared memory. This chapter first introduces the machine Distribution logic, followed by the MPI code extensions and then demonstrating those changes in the form of algorithms specific to routines involved in message passing. Throughout this chapter node and machine are used interchangeably which identifies different computing unit located in the same network.

5.1 Process/Machine Distribution

MPI interfaces works on a pool of processes that reside in **MPI_COMM_WORLD** which are initialized through **MPI_Comm_size** and are identified by rank allocated to them through **MPI_Comm_rank**. This processes are termed as machines in rest of the thesis for simple understanding. Another input parameter is taken from the user which tells the number of machines on which the

LBM-IB simulations needs to be distributed. Hybrid MPI/Pthread LBM-IB version distributes these processes laterally. As shown in fig 5.1, fluid grids are distributed laterally to "N-1" machines out of "N" machines and one machine is reserved for fiber-sheet. Current distribution logic for fluid-grid assumes distribution along 'X' axis, which can be changed in a user-defined function as required along 'Y' or 'Z' axis. This distribution is implemented in ***cube2thread_andmachine*** function. The underlying logic in this function is very simple and the machine ranks are assigned as illustrated in equation 5.1. Here, $CubeIndex_I$ is the cube index along 'X' direction depending on the total number of cubes along that direction which is represented as $Totalcubes_x$ in 5.1. For example, if the number of fluid-nodes along 'X' axis are 128 and the dimension of the smallest cube 'K' based on block distribution discussed in 4.3 is '4' then the value of $Totalcubes_x$ will be 32 and $CubeIndex_I$ will range from 0 to 31 and accordingly machine ranks will be assigned by the equation 5.1a . Fiber machine rank is simply the rank of last machine used in distribution as shown in fig 5.1 and illustrated in table 4.4. This distribution of machines/processes can be considered as the first level of work distribution in the hybrid version. Every machine will spawn local threads to carry out the simulations in their local memory thenceforth. Apart from handling the distribution for different nodes, ***cube2thread_andmachine*** is also responsible for identifying threads local to a specific cube residing inside that particular machine. This thread mapping is similar to that done in cube based Pthread version of LBM-IB and the thread id is calculated in a similar fashion as done in equation 4.1.

$$FluidMachine_{rank} = \frac{CubeIndex_I * K}{Totalcubes_x} \tag{5.1a}$$

$$FiberMachine_{rank} = N - 1 \tag{5.1b}$$

Table 5.1
Process Distribution for $HybridMPI/PthreadLBM - IB$

Cube-Index Range	Machine Rank
0-7	0
8-15	1
16-23	2
24-31	3
Fiber-Machine	4

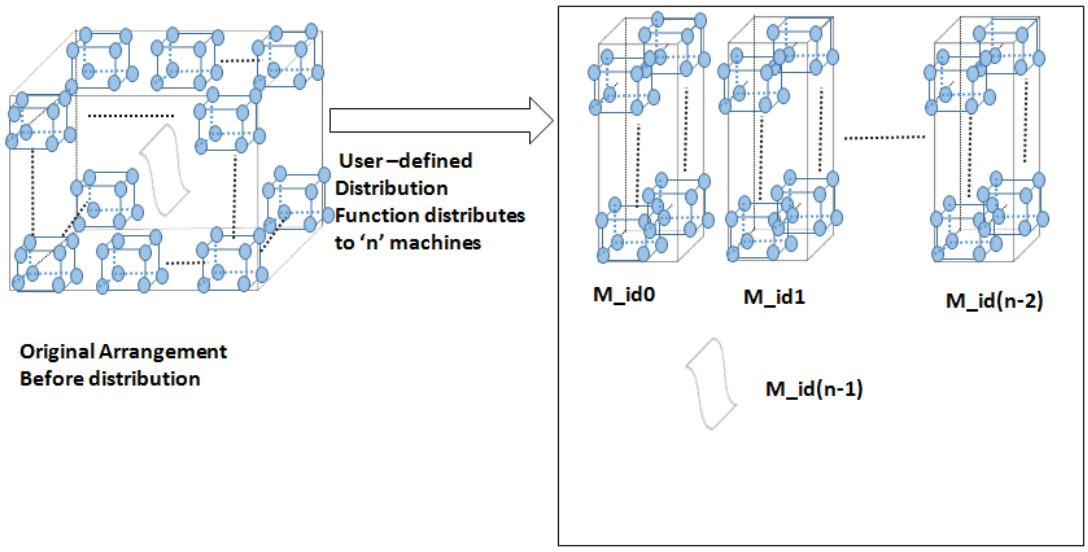


Figure 5.1. Fluid grid and fiber-sheet resides in different machines, Fluid grid is distributed laterally along with its block distribution to n-1 machines.

5.2 MPI Extensions for LBM-IB

Initialization & LBM-IB simulation changes:

This section describes the relevant changes in cube based pthread version of LBM-IB to address distributed computation of LBM-IB. Firstly, MPI initialization is done in the main function which involves initializing the MPI environment via ***MPI_INIT***, allocating the size of the communicator world (provided by **MPI_COMM_WORLD** from MPI library) via ***MPI_Comm_size*** and allocating the ranks to the pool of machines in the communicator via ***MPI_Comm_rank*** [28]. The size of communicator world is decided by the input parameter from the user (**N**), taken as the number of machines participating in distribution as discussed above. The underlying data structure used in Pthread version is used in this version as well with a little modification for Global shared values to accommodate the aforesaid changes. Now, every gv object also carries the information of the number of machines and the machine rank assigned by the MPI library in ***num_macs*** and ***my_rank*** variable respectively as shown in figure 5.2. Note that the simulation steps carried out in **do_thread** function as shown in Algorithm 7, are iterated for the number of time- steps as done for cube based version in Algorithm 5. Also, there are more MPI and Pthread barriers to accommodate data dependencies, but are not shown for simplicity. The generation routines for the fluid-grid namely ***gen_fluid_grid*** is also changed to address the distribution change depicted in fig 5.1. Now, every individual machine allocates memory only for the fluid-nodes residing in those machine and similarly, the fiber-machine allocates fiber related properties only in the machine reserved for it. Then, the initializations as described in 3.1.1 are carried out for both fluid and fiber machines, following which every machine starts simulations implemented in ***do_thread*** function as for cube based Pthread version. Here, every machines share its resources based on the number of threads provided by the user (P*Q*R) as done in Pthread version and starts LBM-IB simulations. The distribution function ***fiber2thread*** assigns a thread id for fiber machine, whereas for the

fluid machines, same functionality is carried out by ***cube2thread_andmachine*** for fluid machines(tid in Algorithms 8, 9 & 10).

Algorithm 6 Main Function:Input:(fsh_w, fsh_h, tf_r,tf_c, fl_x, fl_y,fl_z, fs_x0, fs_y0, fs_z0, K, P, Q, R, num_macs)

int my_rank;

$MPI\_Init(\&argc, \&argv)$;

$MPI\_Comm\_size(MPI\_COMM\_WORLD, \&num\_macs)$;

$MPI\_Comm\_rank(MPI\_COMM\_WORLD, \&my\_rank)$;

gv ← &my_rank;

gv ← num_macs;

fiber_shape = gen_fiber_shape(fsh_w,fsh_h, tf_r,tf_c, fs_x0, fs_y0, fs_z0);

fluid_grid = gen_fluid_grid(fl_x, fl_y,fl_z, K);/*Cube size dimension K being passed*/

total_threads ← P*Q*R;

gv← total_threads, gv← $lock_{obj}$;

for $i \leftarrow 0$ to total_threads **do** pthread_mutex_init(&$lock_{obj}$[i], NULL);
end for

pthread_barrier_init(&$barr_{obj}$, NULL, total_threads); gv← $barr_{obj}$;

/*Do rest of the initializations as in Algorithm 1*/

pthread_t *threads; ▷ Creating a Pthread object

gv ← threads;

for $i \leftarrow 0$ to total_threads **do**

lv_{tid} ←i;

$lv_i \leftarrow$ gv;

$pthread\_create(threads + i, NULL, \textbf{do\_thread}, lv + i)$;
end for

for $i \leftarrow 0$ to total_threads **do** $pthread\_join(threads[i], void*)$;
end for

$MPI\_Barrier()$;

$MPI\_Finalize()$;

```
typedef struct gv_t {
 /* The immersed structure */
 Fibershape* fiber_shape;
 /* The fluid grid */
 Fluidgrid* fluid_grid;
 /* Constant parameters used in LBM-IB */
 double    tau, nu_l, u_l, rho_l, L_l, g_l, Ks_l, Kb_l;
 double    Re, cs_l, Kshat, Kbhat, Fr;
 int       dt, time, TIME_STOP;
 double    c[19][3];              //stores 19 different velocities for ksi
 int       ib, ie, je, jb, ke, kb;    //For Fluid Grid's actual computation part
/*Added for Cube Based Pthread version*/
pthread_t *threads;
pthread_mutex_t *lock_Fluid;
int total_threads, cube_size, P,Q,R; //cube_size is K
int num_cubes_x,  num_cubes_y, num_cubes_z;
pthread_barrier_t barr;
/*Added for MPI version*/
int my_rank;//rank assigned by COMM_WORLS to diff machines
int num_macs;// Total number of processes/machines
}* GV;
```

Figure 5.2. Data structure changes for GV object (shown in bold).

The initializations done above can be referred as first phase of Hybrid programming in which the MPI part is initialized first and then shared with resources on individual machines via ***pthread_create*** interface call. Hitherto, no inter-machine communication takes place, which starts after segregating the fiber-machine computation from the fluid-grid. The fiber-machine is responsible to carry out elastic force computations implemented in ***compute_bendingforce***, ***compute_stretchingforce*** and ***compute_elasticforce*** without the interference from fluid-machines. Similarly, fluid-machines also works independently without communicating with fiber-machine for calculating particle collision factor or distribution function in ***compute_eqlbrmdistrfuncDF1***, update distribution function for fluid particles at the rigid walls in ***bounceback_rigidwalls***, updating the velocity& ρ value via ***compute_rho_and_u*** and working on regeneration steps in ***copy_buffer's_DF***, ***copy_DistributionFunction*** & ***PeriodicBC*** routines. For these routines, there is no message passing and hence a simple check as shown in Algorithm 7 is sufficient.

As shown in Algorithm 6, main thread after initializing the MPI constructs and LBM-IB initializations, allocates thread object and distributes the LBM-IB simulations across total number of threads(total_threads). Every thread in a process calls

the entire simulation steps involving fluid-structure interactions via ***do_thread*** routine(Algorithm 7).

Algorithm 7 do_thread

if $my\_rank == fibermachine$ **then**

  for every $fiber_i$ **do**

    compute_bendingforce($fiber_i$)

    compute_stretchingforce($fiber_i$)

    compute_elasticforce($fiber_i$)

  end for

$pthread\_barrier\_wait()$; $MPI\_Barrier()$;

$find\_ifd\_and\_SpreadForce(fiber_i)$

$pthread\_barrier\_wait()$; $MPI\_Barrier()$;

for every Fluid-Cube$_{I,J,K}$ **do**

  $FluidMachine_{rank} \leftarrow cube2thread\_andmachine(I, J, K)$

  if $my\_rank == FluidMachine_{rank}$ **then**

    $compute\_eqlbrmdistrfuncDF1$(Fluid-Cube$_{I,J,K}$)

    $stream\_distrfunc$(Fluid-Cube$_{I,J,K}$)

    $bounceback\_rigidwalls$(Fluid-Cube$_{I,J,K}$)

    $compute\_rho\_and\_u$(Fluid-Cube$_{I,J,K}$)

end for

$pthread\_barrier\_wait()$; $MPI\_Barrier()$;

$moveFiberSheet(fiber_i)$

for every Fluid-Cube$_{I,J,K}$ **do**

  if $my\_rank == FluidMachine_{rank}$ **then**

    /*Call regeneration fucntions as in Algorithm 1*/

end for

$pthread\_barrier\_wait()$; $MPI\_Barrier()$;

Algorithms for MPI

One of the important aspect of this version is to identify routines that will be involved in message passing. Based on the current distribution there exists two situations in which message passing is required

1. Since, fluid and fiber machine lie on different machines, every fluid- fiber structure interaction requires message passing. The first case is for function ***find_ifd_and_SpreadForce*** in which a fiber machine identifies the influenced region of fluid nodes and spreads its elastic force on those influenced region. Second situation is for the function ***moveFiberSheet***, in which apart from identifying the influenced region from a fiber-node, the velocities of the fluid-nodes lying in the influenced region update fiber's position. Note that the former is one way message passing in which the fiber-machine is the sender and one or more fluid machines are the receivers, whereas, the later is two way message passing in which both fiber and fluid-machines acts as senders and receivers.

2. In ***stream_distrfunc***, where a fluid-node streams it's distribution function to its neighbors, a one way communication between different fluid-machines may be required. In this function, as described before in 3.1.3, DF1 buffer is copied to DF2 buffer of a fluid node lying in its vicinity, a message passing is required if the neighborhood of the fluid-node is not in the current machine. As the current distribution logic assigns different machine ranks laterally as shown in Fig 5.1, some of the fluid nodes in the streamed region might lie in different machines altogether as shown in Fig 5.3 and will need information of the distribution function before updating their DF2 buffer.

The algorithmic changes for the aforesaid changes involving message passing are described below:

1. ***find_ifd_and_SpreadForce:*** Once the elastic forces of fiber-nodes are calculated in fiber-machine, they are spread to fluid-nodes residing in the influenced

region for that fiber-node. Hence, this function requires one-sided communication from fiber-machine to fluid-machine. The fiber-machine acts as sender and the fluid machines acts as receivers. Point worth noting is that there can be more than one receivers, as the influenced region of a fiber-node can involve fluid-nodes which may be part of more than one fluid machine. In fiber-machine first the influenced fluid-nodes and their respective cube indices are known which are passed to ***cube2thread_andmachine*** to know the rank of the receiving fluid machine. Then for all the fiber-nodes the information related to its influenced region and its elastic forces is packed in a buffer using ***MPI_Pack*** and sent to the intended fluid machine using ***MPI_Send***. Apart from the relevant information, a stop flag is also sent in the buffer to let the receiver know when to stop receiving messages. This stop flag is an indication that all fiber-nodes in the fiber-sheet have evaluated their influential domain and once this is completed flag is changed. Every fluid machine is sent stop signal to stop receiving messages further to avoid deadlock. From the receivers perspective, the intended receivers receives the messages using ***MPI_Recv*** and unpacks the buffer using ***MPI_Unpack***. From the buffer information, the $owner_{machine}$ rank is calculated via ***cube2thread_andmachine*** which identifies the fluid machine in which the influenced fluid-nodes reside via its rank. Also, the thread id ($owner_{threadid}$) returned from this function is used to lock and unlock the mutex where spreading of force takes place (Refer Algorithm 8). The receivers, which comprises of all the fluid machines, keep receiving the messages in an infinite while loop and they break out of the loop depending on the stop flag. Here, only the owner machine rank as indicated in the algorithm carries out the actual spreading part and other machines just wait to receive the stop signal. Then these updated forces on fluid-nodes are used to calculate collision factor which is carried out by fluid machines alone.

Algorithm 8 Find Influential Domain and spread Force to Fluid:**find_ifd_and_SpreadForce**

$stop_{flag} \leftarrow$0; $Fluidmachine_{rank}$; $buffer$ ▷ Initialize variables
if $my\_rank == fibermachine$ **then**
  for every $fiber_i$ **do**
    if fiber2thread($fiber_i$)==tid **then**
      Calculate Influential domain from $fiber_i's$ position
      Find Fluid-Cube$_{I,J,K}$ from Influential domain
      $Fluidmachine_{rank} \leftarrow cube2thread\_andmachine$(Fluid-Cube$_{I,J,K}$)
      if $fiber_i == Lastfibernode$ **then** $stop_{flag} \leftarrow$1;
        for every $FluidMachine$ **do**
          $MPI\_Send(stop_{flag})$
      else if $stop_{flag} == 0$ **then**
      $buffer \leftarrow MPI\_Pack(fiber_i'sForce$,Fluid-Cube$_{I,J,K}$&$stop_{flag})$
      $MPI\_Send(buffer)$ ▷ Sent to $Fluidmachine_{rank}$
  end for
else
  while true **do**
    $MPI\_Recv(buffer)$; $MPI\_Unpack(buffer)$
    $FluidNode \leftarrow$ Fluid-Cube$_{I,J,K}$
    $owner_{threadid}$,$owner_{machine} \leftarrow cube2thread\_andmachine$(Fluid-Cube$_{I,J,K}$)
    $Pthread\_Mutex\_lock(owner_{threadid})$ ▷ to prevent duplicate writing
    if $my\_rank == owner_{machine}$ **then**
      $FluidNode_{ElasticForce}$ calculated from $fiber_i'sForce$ ▷ Spreading forces
    $Pthread\_Mutex\_unlock(owner_{threadid})$
    if $stop_{flag} == 1$ **then**
      $break$;
  end while

2. ***stream_distrfunc:*** This function streams the values of distribution function from a fluid-node in the neighborhood of 18 neighbors as shown in fig 3.3 and hence require message passing if the fluid-nodes that are part of this neighborhood belong to different fluid machine. As for cube based Pthread version this calculation was complex, it becomes more complex for this version, since 18 different values of ε changes the computation region and the neighbor fluid node may belong to different machine as shown in fig 5.3. Therefore a message is passed from the current machine identified by **my_rank** attribute to the intended machine using MPI interfaces. The figure does not include the boundary evaluation along Y and Z axis for simplicity, but the software supports any change in the distribution logic similarly in either directions.

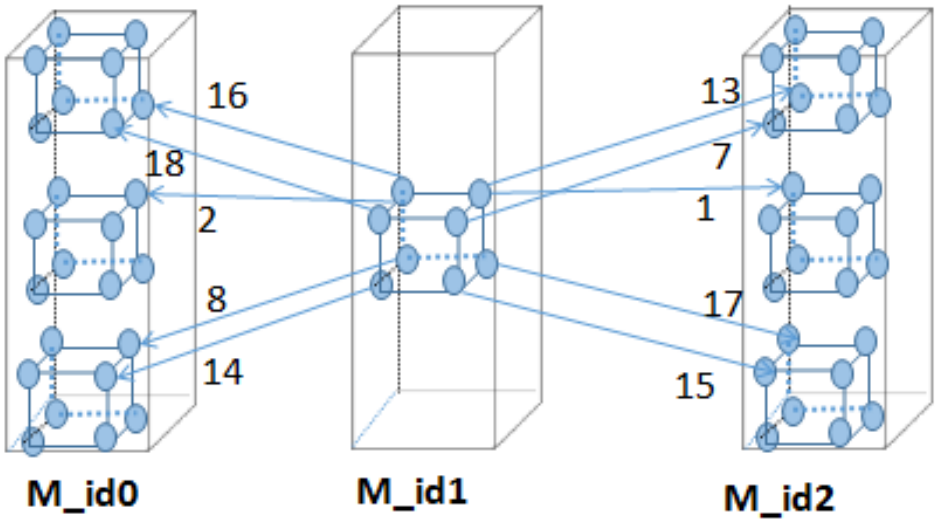


Figure 5.3. Showing boundary cases requiring MPI message passing for lateral distribution of machines along X axis considering three fluid machines. Here M_id1 is the sender and M_id0 and M_id2 are receivers.

Algorithm 9 Streaming:**stream_distrfunc**

for every Fluid-Cube$_{I,J,K}$ **do**

$FluidMachine_{sender} \leftarrow cube2thread\_andmachine$(Fluid-Cube$_{I,J,K}$);

if $cube2thread\_andmachine$(Fluid-Cube$_{I,J,K}$) == tid **then**

Calculate local indices of cube from Fluid-Cube$_{I,J,K}$

$I', J', K' \leftarrow$ New positions of local indices if Streaming is applied for that ε

if Fluid-Node$_{I',J',K'}$ outside 'K' range **then**

$FluidMachine_{recvr} \leftarrow cube2thread\_andmachine$(Fluid-Cube$_{I',J',K'}$);

if $FluidMachine_{recvr} \neq FluidMachine_{sender}$ **then**

if $my\_rank == FluidMachine_{sender}$ **then**

$buffer \leftarrow MPI\_Pack(I', J', K'$,Fluid-Cube$_{I,J,K}.DF1_{\varepsilon}$);

$MPI\_Send(buffer)$;

else if $my\_rank == FluidMachine_{recvr}$ **then**

$MPI\_Recv(buffer)$;

$MPI\_UnPack(buffer)$;

/*Streaming done here if diff machine*/

Fluid-Cube$_{I',J',K'}.DF2_{\varepsilon} \leftarrow$ Fluid-Cube$_{I,J,K}.DF1_{\varepsilon}$;

else/*Streaming done here for same sender and receiver machine*/

Fluid-Cube$_{I',J',K'}.DF2_{\varepsilon} \leftarrow$ Fluid-Cube$_{I,J,K}.DF1_{\varepsilon}$;

end for

3. ***moveFiberSheet:*** This is the last routine which requires message passing in either directions from fiber machine to fluid machine, and then again from fluid machine to fiber machine. As outlined before, first the influenced region of the fibers are known and the fluid machine is identified as in algorithm 8. The fiber machine then sends the relevant information which is required to calculate the velocities of the fluid nodes in a buffer. Then, in the fluid machine, based on equations 2.5 and 2.6, partial sums are stored in buffers and sent back to the fiber machine. These partial sums are the sum of velocities in 'x', 'y' and 'z' directions for the influenced fluid nodes. Fiber machine receives messages

from all fluid machines and update the new position of the fiber-nodes based on the velocity of the influenced fluid nodes. The fluid machines which are not part of the influenced region sends '0' as partial sum to ensure correctness in computation.

Algorithm 10 Updating Fiber-sheets new Position:**moveFiberSheet**

if $my\_rank == fibermachine$ **then** ▷ Fibermachine sending
  for every $fiber_i$ **do**
    if fiber2thread($fiber_i$)==tid **then**
      Find the influential domain of the fiber-node described in Algorithm 8
      $dist \leftarrow$ Distance between the fiber-node and Influenced Fluid-node
      $buffer \leftarrow MPI\_Pack$(Fluid-Cube$_{I,J,K}$, $stop_{flag}$ & $dist$);
      $MPI\_Send(buffer)$;
  end for
else
  while true **do**
    $MPI\_Recv(buffer)$; $MPI\_Unpack(buffer)$;
    FluidNode $\leftarrow$ $Fluid\text{-}Cube_{I,J,K}$;
    FluidNode$_{vel} \leftarrow dist$; ▷ Updating fluid velocity
    if $stop_{flag} == 1$ **then**
      $break$;
  end while
  $MPI\_Send(FluidNode_{vel};)$ ▷ Sending updated velocities to Fiber machine
/*Fluid machine calculation ends*/
if $my\_rank == fibermachine$ **then** ▷ Fibermachine receiving
  MPI_Recv(FluidNode$_{vel}$)
  for every $fiber_i$ **do**
    if fiber2thread($fiber_i$)==tid **then**
      $fiber'_isNewPos \leftarrow$ time* FluiDNode$_{vel}$ ▷ time:current Time step value
  end for

Once the fiber sheet is moved for a given time step, then a set of regeneration functions are called for the fluid machines in ***do_thread*** routine. Hybrid MPI/Pthread LBM-IB version presents very simplistic design of LBM-IB method which can be optimized in many ways as a part of the future work. One of the immediate optimizations that can be done is to store the influenced region in the fiber-sheets data structure and share it across in both ***find_ifd_and_SpreadForce*** and ***moveFiberSheet***. Also, based on the current implementation, message passing takes place for a single fiber-node which can be greatly improved if the message buffers for influenced nodes around a fiber-node can be regrouped and a group based message passing is performed.

6 RELATED WORK

This chapter talks about different existing software libraries that solves fluid structure interactions using LBM or IB. [1] identifies many IB methods being developed based on Peskin's approach [17, 18] to solve analogous CFD problems. These methods have been tailored to match the requirements of CFD applications in question. For instance, the "vortex-method" approach [35],"Volume-Conservation" approach [36, 37], "Adaptive-Mesh" approach [38], "Second -Order Versions" [39, 40], "Multigrid version" [41] , "Penalty-Version" [42]. Apart from these there are other "Implicit-versions" and "stochastic-versions" being developed [1]. As outlined, there is a history behind IB simulations in the areas of physics and new methodologies and techniques have evolved out of it ever since it's inception, one of them being LB method for solving NS equations. LBM has proven to be a powerful tool in the inventories of CFD as it is user-friendly, simple to discretize and very flexible to accommodate additional physics in the existing problem [20, 21, 43–50]. The hybrid LBM-IB methodology to solve flow solutions can be considered in it's nascent stages. The first works on combining the two were done very latterly by Feng and Michaelides [51, 52]. In it's budding stages, LBM-IB was still in the two dimensional world where the solution was provided for 2-D fluid structure. Following these developments, [1] lists various other improvements over this approach but still in 2-D. For instance, "Modified Momentum Exchange Method" [53] , "Multi-BLock version" [54] based on [55,56]. To summarize, there are very few works on hybrid LBM-IB approach that deals with 3-D barring those by [52, 57, 58].

The underlying idea behind the method proposed by [1] is that the calculation of forces; the one exerted by the flexible structure on the fixed fluid nodes as well as the boundary forces, are analyzed within LBM. This approach of IB force calculation has been derived from [59] and is quite similar to the contributions of [57]. The

hybrid LBM-IB approach proposed by [1] differs from other comparable schemes in the following ways-:

1. The IB-LBM hybridization is applicable for flexible structures, and hence Newton's Second Law for force calculation is not accountable for the motion of particles in the fiber-sheet. "$\frac{\partial X}{\partial t} = U$" is used for calculating the motion of the particles which together constitute the submerged flexible structure. Here X is the location of the fiber-sheet structure in 2 dimensions at any given instant of time t and U is the velocity of the fiber-sheet. But, the force that is spread on the fluid nodes from the fiber-sheet is calculated using Newton's Second law "$F = ma$" [1].

2. Based on [59] formalism, the way to compute foreign forces inside IB makes it analogous to a "in-compressible viscous fluid flow" CFD problem without any limitations. Whereas, other methods replicate the same behavior with restrictive conditions. [1]

LBM-IB, being a relatively new simulation algorithm does not have equivalent number of counterpart in the computer world. The parallel flavors of the two exists in isolation. This project offers the integrated LBM-IB software to be used as an application for the first time. However, there exists some libraries and interfaces that have utilized the potential of IB and LBM individually. This chapter describes in brief some of the existing parallel algorithms and software libraries that solves IB and LBM. Also, some of the parallel algorithms that are similar to the cube based implementation are described as state-of-the-art Parallel Algorithms.

1. **IB:** In the areas of IB simulations different parallel libraries and their implementation exists. The main idea behind all IB simulations is an interplay between fixed Lagrangian fluid-nodes and Eulerian structure or fiber-nodes in motion. The underlying technique for solving NS equations varies from one implementation to the other. The parallel implementation of IB by Givelberg and Yelick [8] are worth notifying. They have implemented a distributed parallel version of

IB method which uses 3-D FFT (Fast Fourier Transform) for solving the fluid flow. They call their software implementation as "IB package" that is built on Titanium programming language. Titanium is a high performance computing language which is based on JAVA and is thus object-oriented. Their work identifies different problems that arises when implementing a parallel algorithm. They have also identified load imbalance as a major challenge in developing an efficient and scalable distributed software. These findings are similar to the results being obtained from the OpenMP version of LBM-IB in this thesis. [8] further demonstrated that an efficient selection of data structure to utilize the computing capabilities of a processor's cache improves the scalability of the software. The cube based implementation of LBM-IB also is based on this principle to alleviate the bottlenecks of load imbalance. [8] first divides the entire Fluid grid and the fiber-structure and then distribute it to available processors. FFT approach to solve flow simulations deals with all the fluid nodes present in the fluid grid, but in the case of LBM, to calculate the distribution function for a fluid node for next time step, very few fluid nodes ("18 in case of D3Q19 model of LBM") are visited. This makes LBM easier to parallelize than its counterpart [1]. Also, if their is a change (in the form of modification or addition) in the underlying physics of the problem, LBM is able to adapt to those changes with ease when compared to FFT based simulations [1].

Another important contribution in the areas of IB is that by Griffith [5], in which the NS equations are solved by FFT or projection methods. [5] identifies that dividing the grid and the structure first and then distributing them together increases the inter-processor communication for situation in which structure and grid share boundaries with different processors. Therefore, [5] presents a different approach in which the grid is divided and distributed first followed by structure's division and distribution. This process ensures less inter-processor communication as now there is a more uniform distribution. The software implementation of the aforesaid method uses SAMRAI [60, 61] and PETSc [62]

libraries to build a more scalable distributed version of IB. They call this package as "IBAMR" [63] and it includes the entire specification details for this software.

The above related works and their results are co-related with the existing LBM-IB software developed in this thesis. For instance, the cube-based block distribution also stresses on uniform distribution of data as that done by uniform division of the grid and structure in [5]. Similarly, as pointed out earlier [8] gives more importance to change in the data-structure to efficiently use the available resources, which can be visualized from the cube based Pthread version of LBM-IB. But in both the approaches NS equations are not solved by LBM but rather by FFT or projection methods whereas, in this project, it has been solved entirely by LBM.

2. **LBM:** Computation of mesoscopic properties of the fluid under the influence of a moving structure is the crux behind every LBM simulation. Since, LBM by virtue of its formalization offers parallelization, many parallel algorithms exists for LBM.

   Williams et al. [9] identifies that LBM implementation in the past has shown relatively poor scalability due to complexity in designing the data structures and relative tight coupling between different sub kernels of LBM. They have developed an auto tuned "LBMHD" application which can be compared with the cube based Pthread LBM-IB version. The 2-D decomposition of the fluid nodes in [9] is analogous to the data structure used in LBM-IB serial version. They have created a Perl based generator which is responsible for carrying out the LBM simulations of particle collision and streaming. After thread based optimization, the TLB locality has been addressed as next level of optimization in "LBMHD", followed by relative code changes in the loop unrolling of LBM simulations to be used by specific multicore architecture performing those simulations. A noteworthy contribution by their work is the dynamic code optimization and experimentation for streaming and particle collision on different

multiprocessor chips. Their implementation differs from LBM simulations done in this project on the choice of lattice model, they have used "D3Q27", whereas in this project "D3Q19" model has been used. Also, the underlying physics in their work is centered around "magnetohydrodynamics (MHD)", whereas, this project aims at addressing pure fluid mechanics problem of Fluid Structure interactions. Gotz et al. developed a parallel algorithm for simulation of particle laden flows which find usage in multifarious applications such as sedimentation, fluidization etc [7]. The work demonstrates parallelization of LBM to solve NS equations for a moving rigid body which is simulated by a physical engine, which is a software simulator that simulates the motion and attributes of the rigid body [7]. The main contribution of their work is to optimize and parallelize LBM fluid flow solution using a "patch" data structure and MPI interfaces. The patch data structure combines both the fluid related attributes as well as the rigid body's attributes [7]. Their work can also be clubbed under IB parallelization as the rigid body parallelization is also supported. Their work differs significantly from this project as LBM-IB deals with flexible structure and not rigid bodies.

A numerous optimization approaches and problem nature for solid-fluid interaction have been proposed by Valero-Lara [11]. Code optimization being done in [11] is architecture specific such as a multicore or a GPU architecture. The results are very promising and supports the fact that the software behaves differently on different architectures and one of the many consideration in software design for High Performance Computing should also include the parallelization strength provided by the underlying hardware. This observation is well supported by the profiling variation of LBM-IB serial version on Intel and AMD chipsets(Refer table 3.1and 3.2).

There are some other parallel libraries that focuses LBM parallelization on GPU like accelerators, such as by Li et al. [12], implementation of pLBM library by Peng et al [13] and 2-D LBM implementation by Tölke on CUDA kernels [14]

3. **state-of-the-art Parallel Algorithms:** There are many different parallel algorithms that solve applications other than LBM-IB and provide different solutions to overcome the problems in designing a parallel software. The underlying idea behind cube based implementation of LBM-IB is to divide the data set from a large fluid-grid to smaller cubic sub fluid grids and hence achieve more data locality. This decomposition of data helps in effective utilization of the memory bandwidth available in the form of different multilevel caches, as the small working sets now make use of the idle cache memory and shows better performance results on a manycore system.

   This approach is analogous to that of a block/tile algorithm and software blocking by [6], in which block data decomposition to improve cache performance has been outlined. Apart from evaluating the "Translation Look-Aside Buffer"(TLB) and cache performance of tiling with different data decomposition methods, [6] has presented an algorithm that determines the parameter for selecting the smallest size of the block used in the block distribution in conjunction with tiling. The ideas presented by [6] has been used in solving matrix problems and simulations for CFD domains. The major contribution of this work is that for larger data sets, they have considered TLB misses as well and designed an algorithm that determines the best parameter to be used in the block data layout along with tiling to reduce TLB as well as the cache miss ratio. In LBM-IB library, the tuning parameter 'K' which is the cube size of the smaller sub-grids, can be visualized analogously to the auto tuning parameter described in [6].

   Another noteworthy contribution in designing parallel algorithms based on tiling is that by Dongarra et al., in which they present a classic set of tile algorithms to solve linear algebra problems on multicore architectures [64,65]. They have presented tile based solution to parallelize "Cholesky, LU and QR factorization", in which the computation domain is decomposed into smaller sub domains represented in the form of "block data layout". The algorithms presented provides

a prototype for parallel software suitable for multicore architectures. [64, 65] have outlined that improving data locality via tiling and restricting the thread barriers in existing parallel libraries to solve linear algebra problems limits the scalability of the software. It has been identified that to further utilize the computing capabilities offered by multicore architectures, the existing software libraries needs to be redesigned. As a solution to this problem, [64, 65] have proposed dynamic scheduling of the synchronization tasks through a "graph based model" that limits the data transfer from one local memory of a core to another and improves performance as well as scalability of the software. Apart from presenting the thread based parallelism of the problem, [65] also suggests advancing the loosely coupled tasks involved in the computation to distributed systems. This is quite similar to the LBM-IB library, in which Pthread based shared memory version with intrinsic data dependencies and block distribution is developed first, that can be compared with the "block data layout" proposed in [64, 65], followed by a distributed memory Hybrid MPI/Pthread LBM-IB.

On a similar formalism of designing efficient parallel software for mullticore architectures, sparse cache blocking technique by Williams et al. which utilizes the cache blocking for memory bound matrix vector multiplications [10] is notable. In their work apart from improving the low level optimization including changes in the existing code for matrix multiplications (data structure changes) that deals primarily with single core, optimization strategy for multicore architectures have been provided. However, unlike LBM-IB library where the entire simulation is under one library, [10] have used Perl based generators to generate the low level matrix multiplication routines. [10] presents an adaptable auto tuning framework, which based on the underlying architecture of the multicore machine, uses the best suitable kernel for that system, generated dynamically.

Similar parallelization exists for non-uniform structures as well. For instance, the tiling algorithm demonstrated by Giles et al. [66], identifies that major performance overhead for parallel applications is the frequent data transfers

between cache and the main memory. In order to restrict these data movement, [66] proposes intra-cache communication, in which the data is being reused between L1 and L2 cache. They have focused their work around "Hydra: a large scale CFD code" used in the industry to solve the flow solutions for turbo-machinery design. Both shared memory and distributed memory designs for Hydra has been proposed by [66] based on tiling and the same promises to have reduction in data movement by a factor of four.

Many of the existing software libraries provide parallelizations on either LBM or IB but not both. In this thesis, a new LBM-IB software has been developed with four versions. The sequential version is in itself the first of its kind and the parallel versions of OpenMP and cube-based design foretells that a parallel version of the same is very necessary to utilize the available computing power in full extent. Also, this project embarks the Distributed Memory Version of LBM-IB computation which has not been done so far.

7 CONCLUSION

The serial version, OpenMP version, Block Distribution based Pthread version and MPI/Pthread based Hybrid Distributed version of LBM-IB together constitute a powerful tool for LBM-IB based simulations. The complexities of the mathematical calculations involved in simulating IB methods via LBM has been provided by the serial version of LBM-IB. This version acts as a benchmark for correctness and performance. Before diving into the parallelization domain, gprof profiling [29] helped in analyzing the current state of the simulations. To elaborate, it helped in identifying the rankings of the kernels based on the computation and memory costs. It also showed that the underlying hardware of the machine (Dragon being Intel and BigredII being AMD) also contributes in the run time behavior of the code. As shown in the profiling tables for the two systems in 3.1 and 3.2, some of the kernels/routines are faster in one machine than in the other.

Following serial version, a shared memory version with OpenMP and Pthread library interfaces has been developed. One of the challenges in designing the OpenMP version was to identify the private variables for each pragma construct. This helped in analyzing the underlying simulation process in a better way and develop a correct OpenMP version for LBM-IB simulations. This version showed very good performance speedup, as good as 83% for less number of cores($\leq$ 8). Initially all the scheduling of threads launched by **pragma omp parallel for** was **static** and performance was also evaluated by changing it to **dynamic** , but it showed no performance improvement.

To understand the poor speedup for cores $>$ 8 for OpenMP version, an instrumentation using vampir [31] tool and profiling via OpenMP profiler OmpP [32] & PAPI [33] was done, which helped in identifying that the OpenMP version suffers from load imbalance. As shown in Fig 4.4(a), many OMP threads wait for other

threads before proceeding to the next job. This indicated that idle threads have not been utilized completely in parallelization. Therefore, to achieve more data locality a new data centric block distribution based shared version using light weight Pthreads has been developed. The basic motivation behind using Pthreads was that it will remove the performance overhead for OpenMP threads as they internally use Pthreads. Designing the Data structure and distribution of threads to the new data structure was another challenge, since for this version, the creation and thread management is entirely in the hands of the programmer. With much efforts, a new data structure for the fluid grid was designed and the implementation was verified for correctness against the serial version. The new data structure change added another complexity in design for streaming in which now there were new boundary conditions between cubes to stream the distribution function to the adjacent fluid-nodes. A lot of debugging time was spent on designing a correct streaming function. Initial set of experiments on BigredII and Dragon yielded comparable speedup between OpenMP and Pthread version for less number of cores. Therefore, to better analyze the performance of pthread version, experiments were conducted on a **manycore** machine *Thog* supporting high NUMA depth. It was able to outperform the OpenMP version by about 53% on manycore architecture.

Following the shared memory versions of Pthread and OpenMP, to better utilize the computation power of supercomputers available today, a new distributed hybrid version of LBM-IB has been designed. It is a hybrid version of MPI and Pthreads. This version uses the same data structure as that of cube based version and provides added distribution of node/machines over shared memory. The most challenging and difficult part was designing the routines supporting message passing. For instance, in finding the influential domain around a fiber-node. Also, for streaming as for the pthread version, a new level of boundary case is introduced. First level checks the node or machine responsible for computation and the second level checks the cubes within those machines to be used internally by threads. Though, very basic and

simplistic design, this version will be helpful in understanding the performance of distributed LBM-IB in future.

IB method developed using LBM [1] provides a good foundation for developing a parallel library to solve FSI problems as the shared memory version of LBM-IB shows an impressive speedup of about 83% (for cores ≤ 8). The design and implementation of LBM-IB has helped in understanding that in order to make best use of the available resources, the changes in software design is necessary. For example, data locality feature provided by cube based LBM-IB scales in a much better way on manycore machine Thog than on BigredII and Dragon. Also it was observed that the speedup improved when the input was increased even on multicore architectures. As the number of cores are increased, to fully utilize the available resources, it is very important to develop a data-centric algorithm for High performance computing applications to avoid load imbalance and improve the degree of parallelism. In an ideal case, with increase in number of cores and input, the execution time for both versions should not vary much. But it increases with increase in number of cores, as thread synchronizations constricts the memory bandwidth provided by hardware even though every core works on the constant workload. Therefore, the data centric feature of the new algorithm which makes better use of the available resources in an optimal manner, does not over-exhaust the memory bandwidth and surpasses OpenMP for higher number of cores. Though, this software packages focuses mainly on LBM-IB simulations, the same idea can be used in designing other parallel algorithms that relies on data-locality.

7.1 Future Work

The current library of LBM-IB has a lot of potential to be optimized. For instance, memory optimization can be done on the usage of inlet and outlet buffers as the values lying on this buffers are not used in computation. Also, the influential domain of the fluid-node around a fiber-node can be stored in memory to save re-computation in

a given time-step. For the Hybrid MPI/Pthread version, the distribution is very simple with fluid-nodes and fiber-nodes on different machines. This helps in easier implementation of the message passing interfaces, but the message passing can be improved greatly if instead of sending the message for every fiber-node, a buffer is stacked up for all the fiber-nodes and then a single message is sent. Also, apart from individual function optimization as described, other optimization will include overlapping different time step. This will require a change in the LBM-IB algorithm but will introduce new level of concurrency between different time steps. The global synchronizations using GV object are heavily loaded now which can be improved by using dynamic task scheduling.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Luoding Zhu, Guowei He, Shizhao Wang, Laura Miller, Xing Zhang, Qian You, and Shiaofen Fang. An immersed boundary method based on the lattice boltzmann approach in three dimensions, with application. *Computers & Mathematics with Applications*, 61(12):3506 – 3518, 2011. Mesoscopic Methods for Engineering and Science Proceedings of ICMMES-09 Mesoscopic Methods for Engineering and Science.

[2] R Steijl and G Barakos. Sliding mesh algorithm for cfd analysis of helicopter rotor–fuselage aerodynamics. *International journal for numerical methods in fluids*, 58(5):527–549, 2008.

[3] Yiannis G Perivolaris, Anna N Vougiouka, Vasilis V Alafouzos, Dimitis G Mourikis, Vaggelis P Zagorakis, Kostas G Rados, Dimitra S Barkouta, Arthouros Zervos, and Quin Wang. Coupling of a mesoscale atmospheric prediction system with a cfd microclimatic model for production forecasting of wind farms in complex terrain: Test case in the island of evia. In *Proceedings of the European wind energy conference, Athens, Greece*, 2006.

[4] Shigang Wang, Handan Liu, and Wei Xu. Hydrodynamic modelling and cfd simulation of ferrofluids flow in magnetic targeting drug delivery. *International Journal of Computational Fluid Dynamics*, 22(10):659–667, 2008.

[5] Boyce E Griffith, Richard D Hornung, David M McQueen, and Charles S Peskin. Parallel and adaptive simulation of cardiac fluid dynamics. *Advanced computational infrastructures for parallel and distributed adaptive applications*, page 105, 2010.

[6] Neungsoo Park, Bo Hong, and Viktor K Prasanna. Tiling, block data layout, and memory hierarchy performance. *Parallel and Distributed Systems, IEEE Transactions on*, 14(7):640–654, 2003.

[7] Jan Götz, Klaus Iglberger, Christian Feichtinger, Stefan Donath, and Ulrich Rüde. Coupling multibody dynamics and computational fluid dynamics on 8192 processor cores. *Parallel Computing*, 36(2):142–151, 2010.

[8] Edward Givelberg and K Yelick. Distributed immersed boundary simulation in titanium. *SIAM Journal on Scientific Computing*, 28(4):1361–1378, 2006.

[9] Samuel Williams, Jonathan Carter, Leonid Oliker, John Shalf, and Katherine Yelick. Lattice boltzmann simulation optimization on leading multicore platforms. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–14. IEEE, 2008.

[10] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix–vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178–194, 2009.

[11] Pedro Valero-Lara. Accelerating solid–fluid interaction based on the immersed boundary method on multicore and gpu architectures. *The Journal of Supercomputing*, 70(2):799–815, 2014.

[12] Wei Li, Xiaoming Wei, and Arie Kaufman. Implementing lattice boltzmann computation on graphics hardware. *The Visual Computer*, 19(7-8):444–456, 2003.

[13] Liu Peng, Ken-ichi Nomura, Takehiro Oyakawa, Rajiv K Kalia, Aiichiro Nakano, and Priya Vashishta. Parallel lattice boltzmann flow simulation on emerging multi-core platforms. In *Euro-Par 2008–Parallel Processing*, pages 763–777. Springer, 2008.

[14] Jonas Tölke. Implementation of a lattice boltzmann kernel using the compute unified device architecture developed by nvidia. *Computing and Visualization in Science*, 13(1):29–39, 2010.

[15] Jong Chull Jo. Fluid-structure interactions. *Korea Institute of Nuclear Safety, Republic of Korea*, 2004.

[16] http://en.wikipedia.org/wiki/computational_fluid_dynamics, Date Accessed: 3/2/2014.

[17] Charles S Peskin. Numerical analysis of blood flow in the heart. *Journal of Computational Physics*, 25(3):220 – 252, 1977.

[18] Charles S. Peskin. The immersed boundary method. *Acta Numerica*, 11:479–517, 1 2002.

[19] Prabhu Lal Bhatnagar, Eugene P Gross, and Max Krook. A model for collision processes in gases. i. small amplitude processes in charged and neutral one-component systems. *Physical review*, 94(3):511, 1954.

[20] Yue Hong Qian. Lattice gas and lattice kinetic theory applied to the navier-stokes equations. *PhD thesisEcole Normale Superieure and University of Paris*, 6, 1990.

[21] Shiyi Chen, Hudong Chen, Daniel Martnez, and William Matthaeus. Lattice boltzmann model for simulation of magnetohydrodynamics. *Physical Review Letters*, 67(27):3776, 1991.

[22] Renwei Mei, Wei Shyy, Dazhi Yu, and Li-Shi Luo. Lattice boltzmann method for 3-d flows with curved boundary. *Journal of Computational Physics*, 161(2):680–699, 2000.

[23] Shiyi Chen, Daniel Martinez, and Renwei Mei. On boundary conditions in lattice boltzmann methods. *Physics of Fluids (1994-present)*, 8(9):2527–2536, 1996.

[24] OpenMP Specifications: http://openmp.org/, Date Accessed: 3/2/2015.

[25] Blaise Barney: Lawrence Livermore National Laboratory. POSIX Threads Programming, https://computing.llnl.gov/tutorials/pthreads, Date Accessed: 3/2/2015.

[26] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 427–436. IEEE, 2009.

[27] Distributed Computing, http://en.wikipedia.org/wiki/distributed_computing, Date Accessed: 11/2/2014.

[28] MPI Specifications: http://www.mpi-forum.org, Date Accessed: 3/20/2015.

[29] GNU profiler: gprof, https://sourceware.org/binutils/docs/gprof, Date Accessed: 12/20/2014.

[30] BigRedII at Indiana University, https://kb.iu.edu/d/bcqt, Date Accessed: 1/20/2014.

[31] Vampir: Center for Information Services and High Performance Computing (ZIH), https://www.vampir.eu, Date Accessed: 11/2/2014.

[32] OpenMP Profiler, http://www.ompp-tool.com, Date Accessed: 11/20/2014.

[33] PAPI Project, http://icl.cs.utk.edu/papi, Date Accessed: 11/24/2014.

[34] NUMA, http://en.wikipedia.org/wiki/non-uniform_memory_access, Date Accessed: 3/2/2015.

[35] M.F. McCracken and C.S. Peskin. A vortex method for blood flow through heart valves. *Journal of Computational Physics*, 35(2):183 – 205, 1980.

[36] Charles S. Peskin and Beth Feller Printz. Improved volume conservation in the computation of flows with immersed elastic boundaries. *Journal of Computational Physics*, 105(1):33 – 46, 1993.

[37] ME Rosar and Charles S Peskin. Fluid flow in collapsible elastic tubes: a three-dimensional numerical model. *New York J. Math*, 7:281–302, 2001.

[38] Alexandre M Roma, Charles S Peskin, and Marsha J Berger. An adaptive version of the immersed boundary method. *Journal of Computational Physics*, 153(2):509 – 534, 1999.

[39] Boyce E. Griffith and Charles S. Peskin. On the order of accuracy of the immersed boundary method: Higher order convergence rates for sufficiently smooth problems. *Journal of Computational Physics*, 208(1):75 – 105, 2005.

[40] Ming-Chih Lai and Charles S. Peskin. An immersed boundary method with formal second-order accuracy and reduced numerical viscosity. *Journal of Computational Physics*, 160(2):705 – 719, 2000.

[41] Luoding Zhu and Charles S. Peskin. Simulation of a flapping flexible filament in a flowing soap film by the immersed boundary method. *Journal of Computational Physics*, 179(2):452 – 468, 2002.

[42] Yongsam Kim and Charles S. Peskin. Penalty immersed boundary method for an elastic boundary with mass. *Physics of Fluids (1994-present)*, 19(5):–, 2007.

[43] Shuling Hou. Lattice boltzmann method for incompressible, viscous flow. 1995.

[44] Xiaoyi He and Li-Shi Luo. Theory of the lattice boltzmann method: From the boltzmann equation to the lattice boltzmann equation. *Physical Review E*, 56(6):6811, 1997.

[45] Xiaoyi He and Li-Shi Luo. A priori derivation of the lattice boltzmann equation. *Phys. Rev. E*, 55:R6333–R6336, Jun 1997.

[46] Li-Shi Luo. Unified theory of lattice boltzmann models for nonideal gases. *Phys. Rev. Lett.*, 81:1618–1621, Aug 1998.

[47] Xiaoyi He, Shiyi Chen, and Raoyang Zhang. A lattice boltzmann scheme for incompressible multiphase flow and its application in simulation of rayleigh–taylor instability. *Journal of Computational Physics*, 152(2):642–663, 1999.

[48] Dieter A Wolf-Gladrow. *Lattice-gas cellular automata and lattice Boltzmann models: An Introduction*. Number 1725. Springer Science & Business Media, 2000.

[49] Li-Shi Luo. Theory of the lattice boltzmann method: Lattice boltzmann models for nonideal gases. *Phys. Rev. E*, 62:4982–4996, Oct 2000.

[50] Sauro Succi. *The Lattice-Boltzmann Equation*. Oxford university press, Oxford, 2001.

[51] Zhi-Gang Feng and Efstathios E Michaelides. The immersed boundary-lattice boltzmann method for solving fluidparticles interaction problems. *Journal of Computational Physics*, 195(2):602 – 628, 2004.

[52] Zhi-Gang Feng and Efstathios E. Michaelides. Proteus: a direct forcing method in the simulations of particulate flows. *Journal of Computational Physics*, 202(1):20 – 51, 2005.

[53] X.D. Niu, C. Shu, Y.T. Chew, and Y. Peng. A momentum exchange-based immersed boundary-lattice boltzmann method for simulating incompressible viscous flows. *Physics Letters A*, 354(3):173 – 182, 2006.

[54] Yi Sui, Yong-Tian Chew, Partha Roy, and Hong-Tong Low. A hybrid immersed-boundary and multi-block lattice boltzmann method for simulating fluid and moving-boundaries interactions. *International Journal for Numerical Methods in Fluids*, 53(11):1727–1754, 2007.

[55] Olga Filippova, Sauro Succi, Francesco Mazzocco, Cinzio Arrighetti, Gino Bella, and Dieter Hänel. Multiscale lattice boltzmann schemes with turbulence modeling. *Journal of Computational Physics*, 170(2):812–829, 2001.

[56] Huidan Yu, Sharath S Girimaji, and Li-Shi Luo. Dns and les of decaying isotropic turbulence with and without frame rotation using lattice boltzmann method. *Journal of Computational Physics*, 209(2):599–616, 2005.

[57] Y Sui, YT Chew, P Roy, YP Cheng, and HT Low. Dynamic motion of red blood cells in simple shear flow. *Physics of Fluids (1994-present)*, 20(11):112106, 2008.

[58] Zhi-Gang Feng and Efstathios E Michaelides. Robust treatment of no-slip boundary condition and velocity updating for the lattice-boltzmann simulation of particulate flows. *Computers & Fluids*, 38(2):370–381, 2009.

[59] Zhaoli Guo, Chuguang Zheng, and Baochang Shi. Discrete lattice effects on the forcing term in the lattice boltzmann method. *Physical Review E*, 65(4):046308, 2002.

[60] Richard D Hornung and Scott R Kohn. Managing application complexity in the samrai object-oriented framework. *Concurrency and Computation: Practice and Experience*, 14(5):347–368, 2002.

[61] Richard D Hornung, Andrew M Wissink, and Scott R Kohn. Managing complex data and geometry in parallel structured amr applications. *Engineering with Computers*, 22(3-4):181–195, 2006.

[62] Satish Balay, WD Gropp, and BF Smith. Modern software tools in scientific computing. *Efficient management of parallelism in object oriented numerical software libraries. Birkhäuser Press, Boston*, pages 163–202, 1997.

[63] IBAMR, https://github.com/ibamr/ibamr, Date Accessed: 3/2/2015.

[64] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.

[65] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. Parallel tiled qr factorization for multicore architectures. *Concurrency and Computation: Practice and Experience*, 20(13):1573–1590, 2008.

[66] Mike B Giles, Gihan R Mudalige, Carlo Bertolli, Paul HJ Kelly, E Laszlo, and I Reguly. An analytical study of loop tiling for a large-scale unstructured mesh application. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 477–482. IEEE, 2012.