



Titre: An Algorithm for the lattice Boltzmann method adapted to sparse geometries and distributed memory systems
Title:

Auteur: Morteza Namvar
Author:

Date: 2023

Type: Mémoire ou thèse / Dissertation or Thesis

Référence: Namvar, M. (2023). An Algorithm for the lattice Boltzmann method adapted to sparse geometries and distributed memory systems [Thèse de doctorat, Polytechnique Montréal]. PolyPublie. <https://publications.polymtl.ca/53370/>
Citation:

 **Document en libre accès dans PolyPublie**
Open Access document in PolyPublie

URL de PolyPublie: <https://publications.polymtl.ca/53370/>
PolyPublie URL:

Directeurs de recherche: Sébastien Leclaire
Advisors:

Programme: Génie mécanique
Program:

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

**An algorithm for the lattice Boltzmann method adapted to sparse geometries
and distributed memory systems**

MORTEZA NAMVAR

Département de génie mécanique

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*

Génie mécanique

Février 2023

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Cette thèse intitulée :

**An algorithm for the lattice Boltzmann method adapted to sparse geometries
and distributed memory systems**

présentée par **Morteza NAMVAR**

en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de:

François GUIBAULT, président

Sébastien LECLAIRE, membre et directeur de recherche

Jean-Yves TRÉPANIER, membre

Zahra K. MOTAMED, membre externe

RÉSUMÉ

Dans cette thèse, une modification de l'algorithme de la méthode de Boltzmann sur réseau est effectuée pour obtenir de meilleures performances et par conséquent moins de temps d'exécution. L'objectif principal du projet est d'améliorer les algorithmes de Boltzmann sur réseau actuels et d'en créer un qui fonctionnera efficacement sur des systèmes à mémoire distribuée ainsi qu'avec un seul processeur, et ce, pour simuler de grands systèmes avec des géométries irrégulières.

L'idée suivie dans la thèse est d'éviter d'emmagasiner les données des nœuds solides. L'autre amélioration de l'algorithme de Boltzmann sur réseau consiste à parcourir en une seule passe toutes les fonctions de distribution de probabilité afin d'effectuer les étapes de collision et de propagation en même temps. Ainsi, comme une des boucles principales est sur les fonctions de distribution de probabilité, il est possible d'utiliser un tableau de structures pour simplifier la mise en œuvre de l'étape de propagation. En considérant les difficultés d'implémentation des conditions aux limites avec la méthode de Boltzmann sur réseau, un nouvel environnement est développé pour simplifier la mise en œuvre des conditions limites. Une autre idée dans cette thèse est de concevoir un nouvel algorithme pour la communication de l'interface de passage de messages (*MPI* en anglais) de telle manière qu'au niveau de la programmation, il serait possible de cacher la couche de programmation parallèle des parties du code en série. Cependant, toute l'implémentation et la vérification sont faites pour des cas bidimensionnels.

En suivant les idées mentionnées pour atteindre les objectifs du projet, un nouvel algorithme de Boltzmann sur réseau a été introduit qui utilise une disposition de mémoire simple. Comme les étapes de collision et de propagation sont effectuées en même temps, la localité des données est augmentée. Par conséquent, un simple décalage des données est utilisé pour effectuer l'étape de propagation. En évitant d'emmagasiner les nœuds solides, l'algorithme proposé est adapté aux systèmes à faible taux de vide. De cette manière, outre la diminution de l'utilisation de la mémoire, comme aucun calcul n'est effectué pour les nœuds solides, les ressources de calcul sont également utilisées efficacement. Pour vérifier le code implémenté et basé sur l'algorithme mentionné, plusieurs cas tests académiques sont simulés, notamment l'écoulement de Poiseuille, l'écoulement dans une cavité et l'écoulement autour d'un cylindre. L'algorithme proposé utilise la même quantité de mémoire que l'algorithme à deux réseaux, mais plus que l'algorithme à deux étapes. Puisqu'il utilise un schéma d'adressage indirect, il est néanmoins plus efficace en termes d'utilisation de la mémoire, en particulier pour les géométries à faible taux de vide. Comme les données des nœuds solides ne sont pas

emmagasinées dans cette méthode, il n’y a pas de calcul pour ces types de nœuds, ce qui entraîne par conséquent de meilleures performances pour les géométries irrégulières avec de faibles taux de vide.

En tenant compte des difficultés de mise en œuvre des conditions limites, un nouvel environnement est introduit pour mettre en œuvre de nouvelles conditions limites. Dans cet environnement, une condition limite est mise en œuvre pour un seul côté/coin d’un domaine, puis appliquée aux autres côtés/coins de manière automatique. Ainsi, comme les formulations habituelles d’une condition limite dépendent de l’orientation des nœuds frontière, l’utilisation de cet environnement réduit le nombre d’implémentations, ce qui facilite l’implémentation des conditions limites typiques de la méthode de Boltzmann sur réseau. L’une des principales contributions de ce travail est le concept d’application de l’opération de décalage circulaire et de retournement de données sur un tableau de structure pour projeter les fonctions de distribution de probabilité de n’importe quel bord/coin vers le nœud frontière gauche/inférieur-gauche. Pour vérifier l’environnement, plusieurs des conditions limites communes de la méthode Boltzmann sur réseau ont été mises en œuvre, puis vérifiées à l’aide de références bien connues dans le domaine de la méthode de Boltzmann sur réseau.

Enfin, pour étudier la capacité de l’algorithme de Boltzmann sur réseau nouvellement introduit, la version *MPI* du code a été implémentée. Dans cette étude, un algorithme général de communication *MPI* pour la simulation de la méthode de Boltzmann sur réseau est proposé. L’un des critères de conception de cet algorithme est d’être indépendant de la structure de données du programme de méthode de Boltzmann sur réseau. Ainsi, cet algorithme peut être utilisé pour mettre en parallèle n’importe quel programme de Boltzmann sur réseau. De plus, comme il est indépendant du patron du réseau, il pourrait être utilisé dans n’importe quel programme de Boltzmann sur réseau bi- ou tridimensionnel. De plus, la communication des données est masquée sous les calculs principaux. De plus, les parties de programmation parallèle du programme mises en œuvre sont séparées des autres parties du programme de Boltzmann sur réseau. L’évolutivité (*scalability* en anglais) du code parallèle a été étudiée en exécutant le code sur un superordinateur et de bonnes performances ont été obtenues.

ABSTRACT

In this thesis, a modification of the Lattice Boltzmann Method (LBM) algorithm is done to achieve better performance and consequently less execution time. The main objective of the project is to improve upon the current LBM algorithms and create one that will work efficiently with distributed memory systems as well as with a single processor to simulate large systems with irregular geometries.

The idea followed in the thesis is to avoid storing the solid node data. The other improvement on the LBM algorithm is to traverse all the Probability Distribution Functions (PDF) to perform both the streaming and collision steps at the same time. So, as the loop is on the PDFs it is possible to use a structure-of-array memory layout to simplify the streaming step implementation. By considering the difficulties of the Boundary Condition (BC) implementation, a new framework is provided to simplify the BC implementation. Another idea that is followed in this thesis is designing a new algorithm for Message Passing Interface (MPI) communication in such a way that, at the programming level, it would be possible to hide the parallel programming layer from serial parts of the code. However, all the implementation and verification are done for 2D.

By following the mentioned ideas to reach the objectives of the project, a new LBM algorithm was introduced to use a simple memory layout. As the streaming and collision steps are done at the same time, the data locality is increased. Hence, a simple shift is used to do the streaming step. By avoiding storing the solid nodes the proposed algorithm is suitable for low void fraction systems. In this way, apart from the decrease in memory usage, as no computation is done for solid nodes the computational resources are used efficiently also. To verify the implemented code based on the mentioned algorithm several academic benchmarks, including Poiseuille flow, Cavity flow, and Cylinder flow, are simulated. Then the performance of the code is investigated by simulation of dense and sparse systems. This algorithm uses the same amount of memory as the two-lattice but more than the two-step algorithm. Since it uses an indirect addressing scheme, it is nevertheless more efficient in terms of memory usage, particularly for low void fraction geometries. As the data of the solid nodes are not stored in this method, there is no computation for these kinds of nodes, which consequently results in better performance for irregular geometries with low void fractions.

By considering the difficulties of BC implementation a framework is introduced to implement new BCs. In this framework, a BC is implemented for only one side/corner of a domain and then applied to the other sides/corners. So, as the usual formulations of a BC are dependent

on the orientation of the boundary nodes, using this framework reduces the number of implementations, making it easier to implement typical LBM BCs. One of the core contributions of this work is the concept of applying the circular shift and flip operation over an AoS data layout to map the PDFs of any edge/corner to the left/bottom-left boundary node. To verify the framework, several of the common BCs in the LBM were implemented, then verified using well-known benchmarks in the LBM field.

Finally, to investigate the capability of the newly introduced LBM algorithm the MPI version of the code was implemented. In this study, a general MPI communication algorithm for LBM simulation is proposed. One of the criteria in the design of this algorithm is to be independent of the LBM program data structure. So, this algorithm could be used to parallel any LBM program. In addition, as it is independent of the lattice stencil, it could be used in any 2D or 3D LBM program. Also, the communication of the LBM is hidden with the core computation. Moreover, the parallel programming parts of the implemented program are separated from the other parts of the LBM program. The scalability of the parallel code was investigated by executing the code on a supercomputer and good performances have been obtained..

TABLE OF CONTENTS

RÉSUMÉ	iii
ABSTRACT	v
TABLE OF CONTENTS	vii
LIST OF TABLES	xi
LIST OF FIGURES	xii
LIST OF ACRONYMS	xviii
CHAPTER 1 INTRODUCTION	1
1.1 Typical LBM formulation	2
1.1.1 Streaming	4
1.1.2 Collision	5
1.2 Boundary conditions	6
1.3 Using parallel computing in the LBM	7
1.4 Summary	8
CHAPTER 2 LITERATURE REVIEW	10
2.1 Data structure and algorithm for the LBM	10
2.1.1 Memory ordering	10
2.1.2 Memory addressing	11
2.1.3 Memory layout	13
2.1.4 LBM algorithm	14
2.2 Boundary conditions in LBMs	15
2.3 Parallel programming paradigms	17
2.4 Domain decomposition	19
2.4.1 Existing decomposition methods	20
2.4.2 Existing packages for domain decomposition	22
CHAPTER 3 RESEARCH GAP, OBJECTIVES AND OUTLINE	24
3.1 Research gap	24
3.1.1 About LBM algorithm	24

3.1.2	About difficulties of boundary condition implementation	24
3.1.3	About MPI communication algorithm	25
3.2	Objectives	25
3.3	Outline	25
CHAPTER 4 ARTICLE 1: SIMPLE LATTICE BOLTZMANN METHOD ALGO-		
	RITHM WITH LOW MEMORY USAGE	27
4.1	Abstract	27
4.2	Introduction	28
4.2.1	Summary	33
4.3	Lattice Boltzmann method	34
4.4	The proposed LBM algorithm	37
4.4.1	How to index nodes	38
4.4.2	How to index PDFs	39
4.4.3	The main loops	43
4.4.4	How to apply boundary condition	45
4.5	Results and discussion	45
4.5.1	Verification and validation	46
4.5.2	Memory usage and performance analysis	56
4.6	Conclusion	66
4.7	Acknowledgments	67
CHAPTER 5 ARTICLE 2: LABCOF: LATTICE BOLTZMANN BOUNDARY CON-		
	DITION FRAMEWORK	68
5.1	Abstract	68
5.2	Introduction	68
5.3	Lattice Boltzmann method	70
5.4	Proposed boundary condition framework	73
5.4.1	Concept of the provided solution	76
5.4.2	Class diagram of the implemented code	83
5.5	Results and discussion	85
5.5.1	Poiseuille flow	86
5.5.2	Cavity flow	90
5.5.3	Couette flow	96
5.5.4	Cylinder flow	99
5.6	Conclusion	106
5.7	Acknowledgments	107

5.8	Appendix: Common boundary conditions for the LBM	107
5.8.1	Zou and He pressure (edge)	107
5.8.2	Zou and He velocity (edge)	108
5.8.3	Moment-method	109
5.8.4	Periodic (edge)	109
5.8.5	Symmetry (edge)	110
5.8.6	Open boundary (edge)	111
5.8.7	Moment-method corner	112
5.8.8	Convex-corner	112
5.8.9	Full-way bounce-back	113
CHAPTER 6	ARTICLE 3: LABCOMA: LATTICE BOLTZMANN COMMUNICA- TION ALGORITHM	116
6.1	Abstract	116
6.2	Introduction	116
6.3	Proposed communication algorithm	119
6.3.1	Specifying send/receive PDFs	122
6.3.2	Communication algorithm	126
6.4	Class diagram of the implemented code	128
6.5	Results and discussion	130
6.5.1	Test cases used for the analysis	131
6.5.2	Effect of domain decomposition scheme on performance	132
6.5.3	Dense systems analysis	134
6.5.4	Sparse systems analysis	136
6.5.5	Large system simulation	139
6.6	Conclusion	140
6.7	Acknowledgments	140
6.8	Appendix: Case study: How to develop a parallel LBM program from a serial one	141
6.8.1	Serial LBM algorithm	142
6.8.2	Parallel LBM program	142
CHAPTER 7	GENERAL DISCUSSION	145
CHAPTER 8	CONCLUSION AND RECOMMENDATIONS	146
8.1	Conclusion	146
8.2	Limitations	146

8.3 Future Research and recommendations	146
REFERENCES	148

LIST OF TABLES

Table 4.1	Variables used in Algorithm 1 with short descriptions.	37
Table 4.2	Primary and bottom corner's vortex location compared with results from Ref. [1] for different Reynolds numbers and grid sizes.	52
Table 4.3	Comparison of the vortex location in the wake of the cylinder for different Reynolds numbers.	56
Table 4.4	Required arrays for the proposed, two-step, and two-lattice algorithm.	58
Table 5.1	The required degree rotations and flip operation for any boundary edge/corner to be mapped to the left-side edge/bottom-left corner. .	81
Table 5.2	Primary and bottom corner vortex location at Re=100 with grid size 129×129 compared to results from Ghia <i>et al.</i> [1].	93
Table 5.3	Density value at different locations for different directions of cavity flow simulation.	96
Table 5.4	Comparison of the location of vortex in the wake of the cylinder for different Reynolds numbers.	104
Table 5.5	Comparison of the lift and drag coefficient at Re=100. A grid of size 2000×320 was used for these simulations.	105
Table 5.6	Comparison of the lift and drag coefficient at Re=100.. . . .	106

LIST OF FIGURES

Figure 1.1	Typical two-dimensional lattice stencil of $D2Q9$ with the node in the center (with the color red) and PDFs indexation f_i with $i = 0, 1, \dots, 8$.	2
Figure 1.2	Illustration of the pull scheme for a $D2Q9$ model. For instance f_2 from the bottom-left node stream to the in-hand node.	5
Figure 1.3	The collision step is a relaxation process of the PDFs of a node toward a local equilibrium. Note that the dash-line ellipsoids are the same before and after the collision.	6
Figure 1.4	PDFs which should be specified in the process of applying boundary condition (only index $i = 1$ is shown for clarity purposes).	7
Figure 1.5	The geometry of two real problems. To simulate the full human vessels, around 1.6×10^9 nodes are required and for the Circle of Willis about 10.2×10^9 [2].	8
Figure 4.1	Data dependency in LBM streaming function.	31
Figure 4.2	Typical two-dimensional lattice stencil of $D2Q9$ with the node (in the center with color red) and PDFs indexation f_i with $i = 0, 1, \dots, 8$	35
Figure 4.3	A (3×3) regular grid with a lexicographical approach for node indexing.	38
Figure 4.4	A SoA memory layout is used to store PDFs in the array PDFv and the index of nodes in the array PDFn . In this approach all the f_i PDFs for index i are stored in the F_i structures and all the node indexes for index i are stored in equivalent N_i structures.	39
Figure 4.5	PDF indexing for a (3×3) grid.	42
Figure 4.6	The unknown PDFs, which should be specified in the process of applying boundary conditions. Just one component of PDFs, here f_1 , is depicted for clarity purposes.	45
Figure 4.7	The geometry of Poiseuille flow. Flow is driven by a pressure gradient from the left inlet to the right outlet.	47
Figure 4.8	Comparison of analytical and numerical velocity in x -direction for the Poiseuille flow test case, $Re=5$ and 50 , and grid size of (20×20) , (40×40) , (80×80) , and (160×160)	49
Figure 4.9	Logarithmic error for Poiseuille flow test case.	50
Figure 4.10	Geometry of the lid driven cavity.	51
Figure 4.11	Streamlines for square cavity flow at $Re=100$ and 400	52

Figure 4.12	Velocity profile of at the center line of the cavity for different Reynolds numbers and grid sizes.	53
Figure 4.13	Geometry (not to scale) and boundary condition of the cylinder flow test case.	54
Figure 4.14	Streamline patterns for steady flows past a circular cylinder for different Reynolds numbers and a grid size of 1000×160	55
Figure 4.15	Required arrays to perform a simulation. Size of PDFv and PDFn are $m \times nNode$ and size of R , U , V , sR , sU , and sV are $nNode$. . .	57
Figure 4.16	Comparison of memory consumption for the proposed and two-step algorithm for irregular geometries as function of the void fraction (VF). As an example, let's assume that the calculation is performed using double precision, i.e., a memory consumption of 8 bytes for each value. Thus, $12 \times nNode \times 8$ bytes of memory are used to run a simulation using the two-step D2Q9 algorithm. The number of "doubles per node" used for this algorithm is therefore 12.	59
Figure 4.17	Set of cylinders geometry to test the memory consumption and performance.	60
Figure 4.18	Comparison between the proposed and Palabos code for the required memory to simulate the "set of cylinders flow test case" with different VF.	61
Figure 4.19	Performance comparison between proposed and Palabos code to simulate the "set of cylinder flow test case" for different VF.	62
Figure 4.20	Set of channels geometry to study simulations with very low VF. By changing the width of the channels, it is possible to test different values of VF.	63
Figure 4.21	Required memory to simulate the set of channels flow test case with different VF.	64
Figure 4.22	Performance of the simulations for the "set of channels flow" test case for different VF.	65
Figure 5.1	Typical two-dimensional lattice stencil of <i>D2Q9</i> with the node (in the center with color red) and PDFs indexation f_i with $i = 0, 1, \dots, 8$	71
Figure 5.2	The boundary nodes emerging after domain discretization to generate a computational grid for LBM simulation.	73
Figure 5.3	The unknown PDFs that should be specified in the BC step (only f_1 is depicted for clarity purposes).	74

Figure 5.4	Three types of boundary nodes in 2D geometries. Note that the highlighted region is the fluid domain. The dotted lines are the boundaries and nodes represented by filled circles. The PDFs with a red dashed line are the unknown PDFs.	75
Figure 5.5	For corners with the same unknown PDFs, it is necessary to implement a BC twice as the combination of BCs for each axis changes. The PDFs with the dashed line are the unknown PDFs.	76
Figure 5.6	Classifying the boundary nodes into different regions based on their orientation. For example, all nodes belonging to the bottom side are stored in region 0, while the bottom-right corner is stored in region 5.	78
Figure 5.7	Storing the indexes of nodes of each region. The quantity $nReg$ represents the total number of regions. For example, the data illustrated in Region 0 of Fig. 5.6 is stored contiguously at the beginning of the array containing all the boundary nodes.	78
Figure 5.8	Creation of a local data structure for all the PDF indexes belonging to boundary nodes. These PDF indexes are copied from the base LBM code to a local data structure (iPDFbc) with an AoS memory layout.	79
Figure 5.9	Concept of the rotation operation. As the angle between PDF arrows is 45 degrees, to rotate PDFs by 45 degrees, it is necessary to apply a circular shift on the structure containing the PDF indexes by one unit. Similarly, for a 90 degree rotation, PDF indexes are shifted by two units. Anti-clockwise rotation requires a left shift and clockwise rotation requires right shift.	80
Figure 5.10	Flip operation for a typical lattice stencil.	81
Figure 5.11	All the vector quantities should be mapped in the same way as the PDF indexes are mapped. The quantities u and v represent the rotated velocities in the x - and y -direction, respectively.	83
Figure 5.12	Class diagram of the proposed boundary condition framework.	85
Figure 5.13	The geometry of Poiseuille flow. Flow is driven by a pressure gradient from the left inlet to the right outlet.	86
Figure 5.14	Different BCs used to simulate Poiseuille flow problem.	88
Figure 5.15	Comparison of analytical and numerical velocity in x -direction of Poiseuille flow of $Re=5$ and grid size (20×20) at $x = L/2$	89
Figure 5.16	Logarithmic error for Poiseuille flow test case, using boundary conditions depicted in Figs. 5.14a and 5.14b. In this analysis we used $Re=5$	90
Figure 5.17	Geometry of the lid driven cavity.	91

Figure 5.18	Different BCs for simulating the cavity flow problem: Left, right, and bottom sides are moment-method, top side is Zou and He velocity, bottom-left and bottom-right are moment-method corner, and top-left and top-right are convex-corner BC. The arrows indicate which node contributes to the extrapolation.	92
Figure 5.19	Streamlines for square cavity flow at $Re=100$ and grid size of 129×129 . The red colored points are the vortex locations from Ghia <i>et al.</i> [1]. .	93
Figure 5.20	Velocity profile at the center line of the cavity at $Re=100$ and grid size 129×129	94
Figure 5.21	Simulation of cavity flow for different moving sides. The position of streamlines are approximated.	95
Figure 5.22	Geometry and the flow field of Couette flow problem.	96
Figure 5.23	Various kinds of BCs for simulating the Couette flow problem: Zou and He velocity for top and bottom sides, periodic for left and right sides, and convex-corner for all the corners. The arrows indicate which node contributes by extrapolation.	97
Figure 5.24	Comparison of analytical and numerical velocity of Couette flow in <i>x-direction</i> for $Re=10$ and grid size of 20×20	98
Figure 5.25	Comparison of analytical and numerical velocity of Couette flow in <i>x-direction</i> for $Re=10$ and $Re=100$. A grid size of 20×20 and 100×100 was used for $Re=10$ and $Re=100$ respectively.	99
Figure 5.26	Geometry (not to scale) and BCs of the first test case for the cylinder flow. The arrows indicate which node contributes to the extrapolation. .	101
Figure 5.27	Geometry (not to scale) and BCs of the second test case for the cylinder flow. The arrows indicate which node contributes to extrapolation. .	102
Figure 5.28	Streamline patterns for steady flows past a circular cylinder for different Reynolds numbers and a grid size of 1000×160	103
Figure 5.29	Streamline patterns for unsteady flows past a circular cylinder for $Re=100$ and grid size 2000×320	104
Figure 5.30	Streamlines for unsteady flow past a circular cylinder at $Re=100$ and grid size 768×768 . The geometry and the BCs shown in Fig. 5.27 were used for this simulation.	105
Figure 5.31	The unknown PDFs after streaming: f_2, f_3, f_4 with dash lines are specified by the Zou and He BC.	108
Figure 5.32	Periodic boundary condition along the <i>x</i> -axis in the LBM.	110

Figure 5.33	The symmetry line for a vertical channel flow and how the unknown PDFs can be specified.	111
Figure 5.34	Typical extrapolation of PDFs with the open boundary BC. The boundary is on left side. Here, for simplicity, a zeroth-order extrapolation is illustrated using a direct copy of the PDFs from the adjacent node.	111
Figure 5.35	The unknown PDFs for the bottom-left corner for a typical $D2Q9$ lattice stencil.	112
Figure 5.36	The process of full-way bounce-back: the grey shaded domain is the solid (ghost) region and the line represents the wall boundary. (a) pre-collision PDF at t , (b) post-streaming PDF at $t + \Delta t$, (c) reversing the PDF as the full-way bounce-back BC at $t + \Delta t$, (d) post streaming PDF at $t + 2\Delta t$	113
Figure 5.37	The unknown PDFs for a convex node for a typical $D2Q9$ lattice stencil.	115
Figure 5.38	The unknown PDF for a concave node using a typical $D2Q9$ lattice stencil.	115
Figure 6.1	To perform the communication the send PDFs in Ω_2 should be sent to the P_1 , and the P_1 should receive them. Then the P_1 should write them to the proper location, <i>i.e.</i> receive PDF indexes, in the data structure of the base LBM program to perform the streaming step.	121
Figure 6.2	Specifying ghost nodes by labeling them based on a typical $D2Q9$ stencil. Diamond shape nodes are ghost nodes for Ω_i	122
Figure 6.3	Ghost PDFs of Ω_1 are those that belong to a ghost node and pointing to the Ω_1 . Here, just the ghost PDFs of Ω_1 are shown. P_1 and P_2 represent the process 1 and 2 respectively which responsible to do the computation of Ω_1 and Ω_2	123
Figure 6.4	Ghost PDF data structure (<i>GhostPDF</i>). The first chunk of this data structure stores the PDFs belonging to part 1, then those of part 2, and so on.	124
Figure 6.5	Receive PDF data structure (<i>RecvPDF</i>) is used to receive and write receive PDFs into the data structure of the LBM program.	125
Figure 6.6	Send PDFs data structure (<i>SendPDF</i>) is used to prepare the value of required PDFs to be sent to the other parts.	126
Figure 6.7	The algorithm of communicating PDFs in order to hide the communication behind the computation.	127
Figure 6.8	Class diagram of the implemented code associate to Section 4.	129
Figure 6.9	Class diagram of the proposed communication framework.	130

Figure 6.10	The geometry of the first and second test cases to investigate the performance for simulating the dense and sparse systems respectively. . .	132
Figure 6.11	Comparison of the speedup for using blocking and METIS as the partitioning method. In this study, the first test case, i.e. a lid-driven cavity, with a grid size of 20 and 200 million nodes is used.	133
Figure 6.12	Execution time of METIS library for different grid sizes.	134
Figure 6.13	Speedup and efficiency for a dense system. In this test, a grid of size 200 Million fluid nodes is used.	135
Figure 6.14	Speedup and efficiency of weak scalability with 200 processes for the dense system (lid-driven cavity test case).	136
Figure 6.15	Speedup and efficiency for a sparse system, <i>i.e.</i> the pseudo porous media geometry (a pseudo porous media). In this test, a grid of size 2 billion is used which consists of 200 million fluid nodes.	137
Figure 6.16	Speedup and efficiency for sparse system, <i>i.e.</i> pseudo porous media geometry. In this test 200 cores are used.	138
Figure 6.17	Execution time and <i>MDUPS</i> for sparse system, <i>i.e.</i> pseudo porous media geometry. In this test 400 cores are used.	140
Figure 6.18	Algorithm of the serial and parallel LBM code.	141
Figure 6.19	A typical element, forming from four nodes shown in diamond, is used to generate the input grid of METIS from a regular grid.	143
Figure 6.20	Grid decomposition steps. Here, the initial grid is partitioned into three parts, and each part is assigned to a process. The black dot represents fluid nodes and the grey dot show solid nodes.	144

LIST OF ACRONYMS

AMR	Adaptive Mesh Refinement
AoS	Array of Structures
BC	Boundary Condition
CPU	Central Processing Unit
CSoA	Collected Structure of Arrays
CAoSoA	Clustered Array of Structure of Array
CFD	Computational Fluid Dynamics
GPU	Graphics Processing Unit
GGP	Graph Growing Partitioning
LBM	Lattice Boltzmann Method
LGA	Lattice Gas Automata
MPI	Message Passing Interface
MDUPS	Millions of Distribution Update per second
NS	Navier-Stokes
PDF	Probability Distribution Functions
ORB	Orthogonal recursive bisection
SoA	Structure of Arrays
VF	Void Fraction

CHAPTER 1 INTRODUCTION

Computational Fluid Dynamics (CFD) is a virtual fluid dynamics simulator that uses numerical methods to simulate fluid motion and heat transfer. With the invention of computing devices and the interest in numerical simulation, attempts to develop efficient numerical methods have been one of the most interesting research fields. Conventional methods in CFD, i.e. the Finite Difference Method and Finite Volume Method, have matured over recent years. But, the difficulties in simulating complex fluid flows persuaded the researchers to consider the Lattice Boltzmann Method (LBM), which emerged in 1988 [3]. In recent years this method is aggressively followed by researchers.

The LBM is designed to solve macroscopic fluid dynamics problems, but it lives between the microscopic (molecular) and macroscopic (continuum) worlds. The Boltzmann equation describes the statistical behavior of the average number of microscopic flow particles moving between any neighboring positions, with a specific velocity. The statistical behavior is determined by the motion of the particles and the inter-particle interactions. Some variables associated with each position represent the probability of the presence of a particle with a given velocity. Particles could stream between two neighboring positions in discrete time steps, and also, they could collide with their neighboring particles.

The LBM is derived from the Lattice Gas Automata (LGA) [3]. As in the LBM, each step of an LGA consists of a collision and a streaming step. But, the LGA suffers from some major drawbacks such as intrinsic noise, non-Galilean invariance, an unphysical velocity-dependent pressure, and high numerical viscosity [4]. In 1986, a correct NS equation derivation starting from the LGA on a hexagonal lattice was proposed by Frisch *et al.* [5]. In 1991, a simplification has been done with a Bhatnagar-Gross-Krook collision operator, which assumes the discrete distribution functions are close to a local equilibrium and paved the way for further development in LBM [6–8].

In this chapter, a short review of the LBM formulation used in this study is done in Section 1.1. Also to go deep into the LBM algorithm later in the thesis and to introduce notation, the core functions of LBM, i.e. the streaming and collision steps are reviewed. Then in Section 1.2 the basic concept of boundary condition for the LBM, in terms of implementation, is reviewed as they are important and usually complicated in the LBM field compared to the classical flow solvers. Finally, in Section 1.3 a short explanation regarding the necessity for using parallel programming for the LBM simulation is presented.

1.1 Typical LBM formulation

In LBM, like other numerical methods, it is necessary to discretize the Cartesian space by introducing a set of nodes. These nodes could be distributed evenly in the computational domain. A general LBM model has n spatial dimensions and m probability distribution functions (PDFs), where each node is superimposed by a $DnQm$ discretization called "lattice stencil" [9]. For example, Fig. 5.1 depicts the PDFs associated with a node for a $D2Q9$ lattice stencil. The PDFs may be viewed as a representation of the probability of finding a group of fluid molecules at location $\vec{x}$ and at time t :

$$f_i(\vec{x}, t); \quad i = 0, \dots, m - 1 \quad (1.1)$$

where $m = 9$ is the total number of PDFs belonging to the lattice stencil in 2D.

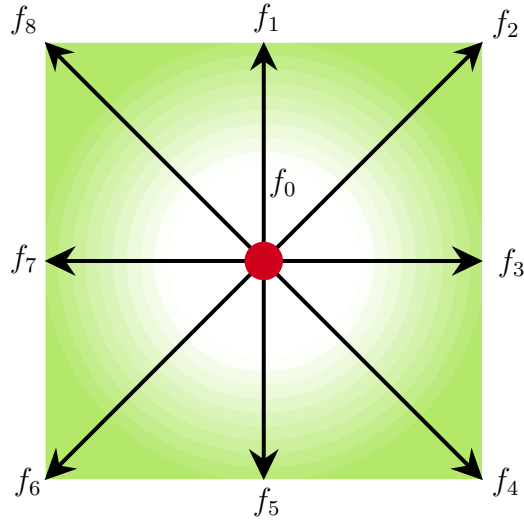


Figure 1.1 Typical two-dimensional lattice stencil of $D2Q9$ with the node in the center (with the color red) and PDFs indexation f_i with $i = 0, 1, \dots, 8$.

Each PDF has a discrete velocity $\vec{c}_i$. For example, the $D2Q9$ lattice stencil, depicted in Fig. 5.1, has the following discrete velocity [10]:

$$\vec{c}_i = (c_{i,x}, c_{i,y}) = \begin{cases} (0, 0), & i = 0 \\ (\cos \theta_i, \sin \theta_i) c, & i = 1, 3, 5, 7 \\ (\cos \theta_i, \sin \theta_i) \sqrt{2} c, & i = 2, 4, 6, 8 \end{cases} \quad (1.2)$$

where $\theta_i = \frac{\pi}{2} + \frac{\pi}{4}(1 - i)$, $c = \Delta x / \Delta t$ is the lattice velocity, Δx is the spatial step, and Δt is the time step. The most common form of the lattice Boltzmann evolution equation in the

literature is as follows [11]:

$$f_i(\vec{x} + \vec{c}_i \Delta t, t + \Delta t) = f_i(\vec{x}, t) - \frac{f_i(\vec{x}, t) - f_i^{(eq)}(\rho(\vec{x}, t), \vec{u}(\vec{x}, t))}{\tau} \quad (1.3)$$

Eq. 1.3 is usually solved in two steps, respectively named collision and streaming. The first collision model to recover the nearly incompressible NS equations was achieved by Qian *et al.* [12]. This operator is often called LBGK [12] collision:

$$f_i^*(\vec{x}, t) = f_i(\vec{x}, t) - \frac{f_i(\vec{x}, t) - f_i^{(eq)}(\rho(\vec{x}, t), \vec{u}(\vec{x}, t))}{\tau} \quad (1.4)$$

where f_i^* represents the distribution function after the collision, which is also named the post-collision PDF, and τ is the relaxation parameter which measures the average time for the PDFs to relax to its equilibrium via collisions [13]. The evolution Eq. 1.3 is completed with the streaming (or propagation) step:

$$f_i(\vec{x} + \vec{c}_i \Delta t, t + \Delta t) = f_i^*(\vec{x}, t) \quad (1.5)$$

with the left hand-side of Eq. 1.5 being the post-streaming PDF.

In Eq. 1.3, the $f_i^{(eq)}$ is the local equilibrium distribution function for isothermal fluid and is defined as:

$$f_i^{(eq)} = \rho w_i \left(1 + \frac{\vec{c}_i \cdot \vec{u}}{c_s^2} + \frac{(\vec{c}_i \cdot \vec{u})^2}{2c_s^4} - \frac{\vec{u} \cdot \vec{u}}{2c_s^2} \right) \quad (1.6)$$

where w_i is a weighting factor. For example, for the *D2Q9* lattice stencil, weighting factor is defined as:

$$w_i = \begin{cases} 4/9, & i = 0 \\ 1/9, & i = 1, 3, 5, 7 \\ 1/36, & i = 2, 4, 6, 8 \end{cases} \quad (1.7)$$

The parameter c_s is the LBM speed of sound [14]:

$$c_s = \frac{c}{\sqrt{3}} \quad (1.8)$$

The macroscopic quantities of interest are the fluid density ρ and velocity $\vec{u}$, defined as:

$$\rho(\vec{x}, t) = \sum_{i=0}^{m-1} f_i(\vec{x}, t) \quad (1.9)$$

$$\vec{u}(\vec{x}, t) = \frac{1}{\rho} \sum_{i=0}^{m-1} f_i(\vec{x}, t) \vec{c}_i \quad (1.10)$$

The kinematic viscosity of the fluid is determined by the following relation [9]:

$$\nu = c_s^2 \left(\tau - \frac{\Delta t}{2} \right) \quad (1.11)$$

and the pressure p is derived from an isothermal equation of state [9]:

$$p = c_s^2 \rho \quad (1.12)$$

Note that with the LBM it is common to use a system of units where the discrete Cartesian spatial and time steps are unity, i.e. $\Delta x = \Delta y = \Delta t = 1$, so that the lattice velocity $c = 1$. This is the system of units adopted for the rest of this work. Given an initial condition for the PDFs, solving Eq. 1.3 includes two main steps named *streaming* and *collision*.

1.1.1 Streaming

The streaming (or propagation) phase is one of the main computational steps in the LBM. There are two most well-known streaming schemes named pull and push. In this phase, PDFs are pulled/pushed from/to neighboring nodes according to their associated discrete velocity [15]. For example, the pull scheme is illustrated in Fig. 1.2. So, all f_i are pulled from the nearest neighbor nodes.

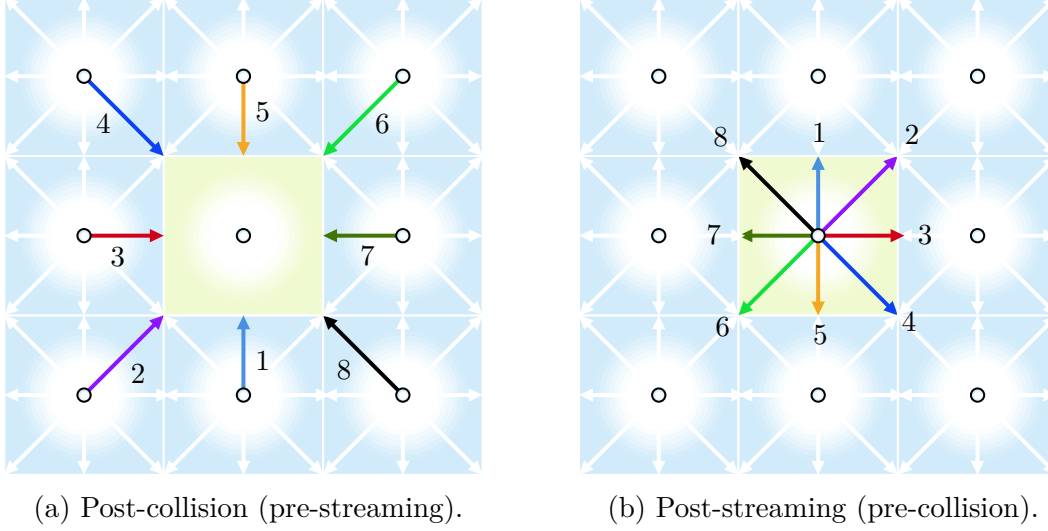


Figure 1.2 Illustration of the pull scheme for a $D2Q9$ model. For instance f_2 from the bottom-left node stream to the in-hand node.

1.1.2 Collision

The collision phase is the other most intensive computational step in the LBM. In this phase, the PDFs collide with other PDFs streaming into a node from different directions. In other words, Eq. 5.3 is implemented in this phase.

The collision phase could be interpreted as relaxation toward a local equilibrium. As illustrated in Fig. 1.3a, there are local post-collision PDFs at a node with the corresponding local equilibrium PDFs. After applying the collision, the PDFs are modified by the effect of the relaxation parameter as illustrated in Fig. 1.3b. Note that the equilibrium has not changed during the collision process, only the post-collision PDFs are closer to the equilibrium than the pre-collision PDFs.

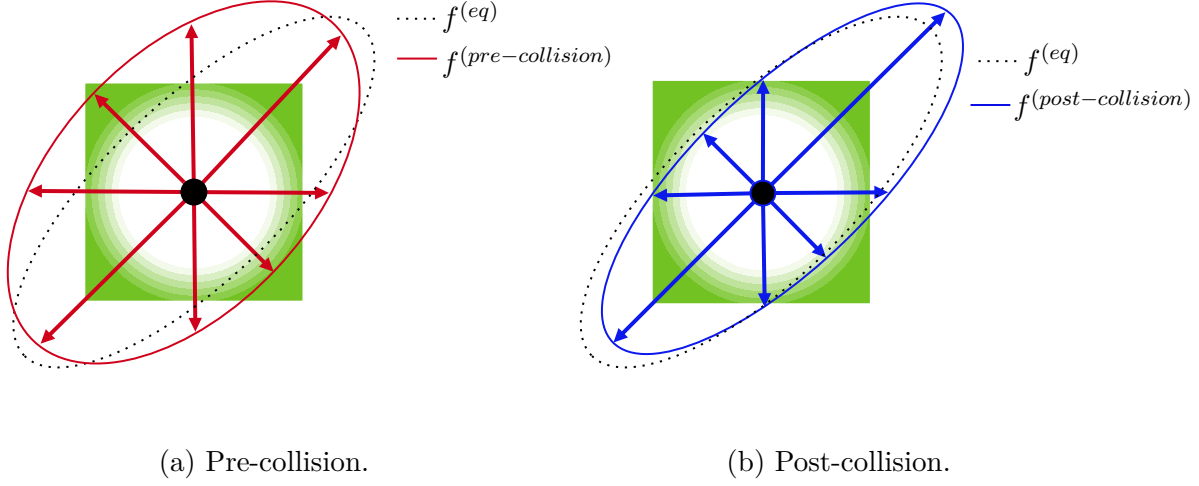


Figure 1.3 The collision step is a relaxation process of the PDFs of a node toward a local equilibrium. Note that the dash-line ellipsoids are the same before and after the collision.

1.2 Boundary conditions

From the analytical solution standpoint, boundary conditions (BCs) add additional equations to the governing equations to specify the state of the system. In numerical solutions, boundary conditions act dynamically to define the problem of interest. So, specifying the boundary conditions assumes critical importance in numerical methods. If BCs are not properly applied in the numerical scheme, severe problems may arise in the solution process. Hence, one of the most common reasons for the divergence of numerical solutions is an incorrect implementation of boundary conditions.

Now, which PDFs should be corrected (or specified), and generally, what happens in LBM that we need to apply boundary conditions? To answer this question, take a look at Fig. 4.6 and imagine the streaming step has been applied to all existing PDFs. In the streaming step, those PDFs belonging to the fluid nodes are streamed to neighboring nodes. After performing this process those PDFs pointing out of the domain have been streamed from the fluid nodes but, those which point inward the domain are unknown. So, these PDFs should be determined by a different set of rules named "boundary condition" in the LBM.

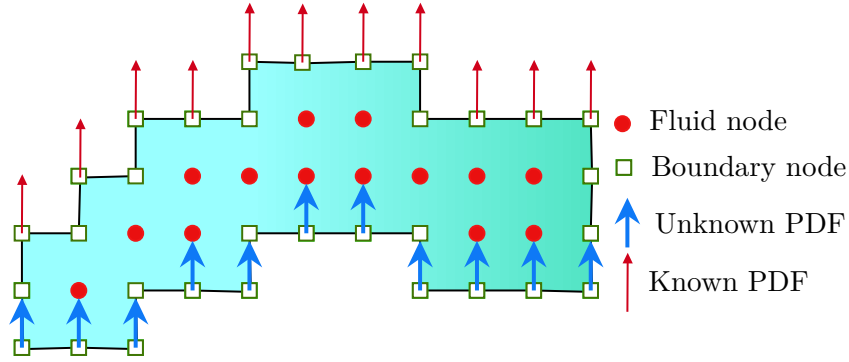


Figure 1.4 PDFs which should be specified in the process of applying boundary condition (only index $i = 1$ is shown for clarity purposes).

1.3 Using parallel computing in the LBM

Complex phenomena in engineering and science are increasingly being studied by using simulation. Even with the progress in computational power, the demand for high-resolution simulations is increasing. For example, to simulate the full human venous tree, as shown in Fig. 1.5a, approximately 1.6×10^9 nodes are used to do a simulation by the LBM. While the size of the problem to research the brain depicted in Fig. 1.5b requires even more resolution. As could be seen in Fig. 1.5b to resolve the circle of Willis arterial structure, 10.2×10^9 nodes are required [2]. So, to accomplish a simulation in real life, besides using a suitable physical model, the availability of substantial computational resources is necessary. But, the capability of exploiting the computational resources is more important from the viewpoint of numerical modeling. Hence, any numerical model should be evaluated in terms of its capability of using full computational power and memory resources.

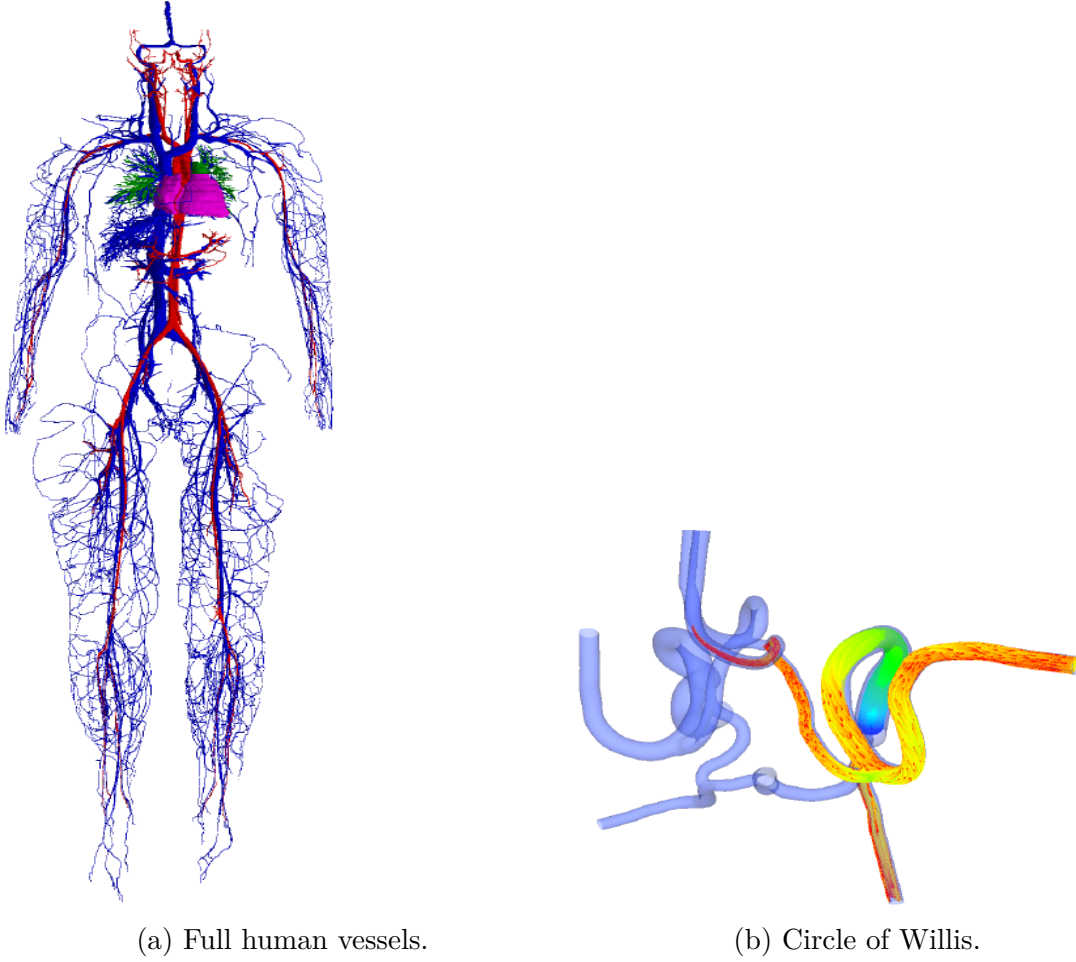


Figure 1.5 The geometry of two real problems. To simulate the full human vessels, around 1.6×10^9 nodes are required and for the Circle of Willis about 10.2×10^9 [2].

In the LBM, like any other CFD method, increasing the grid size will result in an increase in the simulation time. For some complex geometries, the number of grid nodes required for an accurate simulation is so large that the simulation time is unacceptable. So, to reduce the simulation time, and also to use the computational power efficiently, using parallel computing is crucial.

1.4 Summary

In this chapter, the governing equations of the LBM that will be used in the rest of this thesis have been presented. The approach of implementing the LBM algorithm needs a data structure to store the required data. In this thesis, the LBM is investigated in terms of the LBM algorithm and data structure. Also, the approach of applying boundary conditions to

the LBM will be investigated in this thesis. In addition, parallel programming in the field of LBM will be studied as well. It should be mentioned that the scope of this project is not to solve any applications in CFD like full human vessels or the circle of Willis presented in the previous section. In other words, the main objective is to improve the underlying approaches to solving flows using LBM.

CHAPTER 2 LITERATURE REVIEW

In this chapter, a literature review is done to show the importance of data structure and LBM algorithm on the efficiency of LBM application. To do that, the existing data structure of the LBM is investigated. After that, different kinds of LBM algorithm are studied. Then the boundary condition aspect of the LBM is reviewed to show the difficulties in this field of the LBM. After that, existing parallel programming approaches in the LBM are investigated. Finally, a literature review is done about the parallelization of the LBM and the existing methods for domain decomposition.

This literature review will then allow us to identify research gaps that will be discussed in the next chapter of this thesis. Note that for the sake of clarity, some elements that are discussed in the articles of this thesis, are repeated here.

2.1 Data structure and algorithm for the LBM

To simulate a flow field, based on a numerical method, a computational grid should be generated to discretize the computational domain. In the LBM just a set of nodes should be generated as the computational grid. Generally, for each node of the grid, three kinds of data should be stored in the LBM: geometry data, macroscopic variables, and PDFs. The geometry data represents the type of nodes (solid or fluid). The approach applied for storing and accessing the macroscopic variables and PDFs is the data structure. So, the primary issue in the data structure for the LBM are as follows:

1. How to order the nodes into a 1D memory space. This will be called memory ordering.
2. How to address and access data. This will be called as memory addressing.
3. How to store PDFs. This will be called memory layout.

While the first one will determine the memory consumption of the solver, the last two influence its performance.

2.1.1 Memory ordering

The way to assign a number (index) to any node is memory ordering. The most common method is lexicographical indexing [16]. Although lexicographical indexing is straightforward, flexibility is a criterion that should be taken into account particularly when it is desired to

simulate flow field around complex geometries. So, some works are using unstructured mesh as the grid for numerical simulation [17–20]. But, it should be noticed that generating an unstructured grid for complex geometries is one of the reasons for developing new ideas in the field of CFD and trying to get rid of mesh generation.

2.1.2 Memory addressing

The way of accessing and addressing data of each node such as geometry data, macroscopic variables, and PDFs is called memory addressing. In this regard, there are two main approaches named *direct* and *indirect* addressing. Besides, there are some other methods such as semi-direct and locally direct addressing.

Direct addressing Direct addressing is the most common approach for addressing [21] in which the computational domain is embedded in a bounding box that contains all nodes, including solid and fluid. So, a lexicographical approach is used for indexing the nodes. In this approach, using 3D/4D arrays is the most common method to store the macroscopic variables and PDF values for any 2D/3D geometry. Apart from simplicity at the implementation level, this approach is suitable for dense systems such as Cavity or Channel flow. But for small void fractions, and complex geometries this method would be inefficient. That is because a considerable amount of stored data may not be used which in turn results in low performance.

Indirect addressing Denying the nature of nodes (fluid/solid) may result in wasting memory resources and either not using all of the computational capabilities. Hence, in the indirect addressing [22,23] method, memory is allocated to fluid nodes and an additional array, called fluid index array, is used to determine the fluid node indexes. Therefore, the collision step is applied just to the fluid nodes array. Vidal *et al.* [24] stored the connectivity list for all fluid nodes by a vectorized scheme to avoid storing the solid nodes in order to improve the performance of large heterogeneous porous media simulations. Hasert *et al.* [25] introduced a new kind of indirect addressing, which is based on the octree mesh generation idea, to handle complex geometries and also multi-species flows. In this approach, an array containing the neighboring list for any node (connectivity matrix) needs to be used. That demands a considerable amount of memory resources. Also, during the streaming step, it is needed to load data of neighboring lattices from the mentioned neighboring list which includes the computational cost to load the indexes of neighboring nodes. In summary, indirect addressing would be efficient for small void fraction problems. However, using this approach may result in implementation complexity and also decrease the computational efficiency for dense

systems [16].

Semi-direct addressing Semi-direct addressing saves some memory resources by storing data at fluid nodes only while using indirect addressing. It is achieved by using an array of pointers directly addressing the fluid nodes [26]. There are some difficulties with direct addressing, and also semi-direct addressing to apply boundary conditions. Suppose it is desired to apply bounce-back boundary conditions for a porous media. To do that, all the nodes should be traversed to access the boundary nodes. So, Astorino *et al.* [27] improved the semi-direct addressing method and introduced a more simple approach to treat boundary conditions. Indeed, they store an indirect data structure just for boundary nodes. Although the total memory consumption is slightly increased as boundary node data are stored twice (one in semi-direct addressing and one in indirect addressing), it has considerable advantages in terms of implementation and performance.

Locally direct addressing In this approach, the computational domain is decomposed into some cuboids [16,28]. Fietz *et al.* [28] called this step as *shrinking*, which is done in the pre-processing step. In the shrinking step, those cubes which do not contain any fluid nodes will be removed. So, by using this technique, it is possible to decrease memory consumption considerably for complex geometries. A special data structure is used to determine the relationship between cuboids. However, direct addressing is used for nodes belonging to each cuboid. These sets of cuboids could be of the same size or even different sizes. They could also have different sizes and be placed arbitrarily. Although this approach is efficient for complex geometries, for porous media it is inefficient in terms of memory consumption. It is because inside each cube there are a lot of solid nodes and direct addressing is used to store both the fluid and solid nodes.

Adaptive mesh refinement In theory, a finer grid allows for greater accuracy but requires greater computing resources. Therefore, a non-uniform mesh would be a more efficient idea for higher accuracy and lower computational cost. The adaptive mesh refinement (AMR) [29,30] technique could meet this criterion. In this approach, the objective is to reduce the grid density in the smooth gradient area and increase the grid in the sharp gradient area of the computational domain. In multi-block AMR [31], the computational grid is divided into several blocks. A direct addressing scheme could then be used within each block, while a special treatment should be applied at the interface of each block. Another approach is to use an unstructured Cartesian grid as the node distribution for the LBM. This approach uses a tree-like data structure. One of the advantages of AMR is that, in the case of a

high void fraction and a smooth gradient region, it can significantly reduce the number of nodes. Despite this and other advantages, the tree-like data structure of this approach has its complexity that can complicate implementation and programming.

2.1.3 Memory layout

Data layout plays a major role in determining application performance on both the CPU and GPU. Since the streaming and collision step requires global and local access to data respectively, different data layouts should be employed to gain the best performance.

Array of Structures (AoS) In the AoS approach, PDFs at each node are stored in a contiguous chunk of memory according to the index of their associated nodes. In the collision step, the PDFs at each node need to be updated by applying a relaxation. So, all of the PDFs should be loaded from the main/GPU memory for each node. Consequently, it is required to store the PDFs belonging to each node as close as possible to each other. This requirement is satisfied by using an AoS data layout as studied in Refs. [32–34].

Structure of Arrays (SoA) In the SoA scheme, each PDF component with a certain velocity direction belonging to different nodes is stored contiguously. In the streaming step the PDFs need to be pulled/pushed from/to the adjacent node. So, in the streaming step, PDFs with the same index of different nodes should be stored contiguously. This means an SoA data layout is more compatible with the streaming step [16, 27].

Collected Structure of Arrays (CSoA) and Clustered Array of Structure of Array (CAoSoA) As mentioned before, AoS is compatible with collision, and SoA is compatible with streaming. So, in each time step, the LBM core functions need two different data layouts. Hence, there are two options to deal with data layout: the first option is to use different data layouts for each of the streaming and collision steps, and the second option is to have a unique data layout that satisfies both of the contradictory requirements of collision and streaming steps. Regarding the first option, Majeti *et al.* [35] demonstrated that using a single data layout is generally not optimal. But, for the second option, the CSoA data layout has been introduced [32]. In this approach, according to the size of the cache, a cluster of nodes is stored in an SoA format. So, the data layout of each cluster is suitable for the collision, while the entire data layout is compatible with the streaming step. However, still, some cache misses will happen during the execution of the streaming step. The cache misses are inevitable because it is impossible to store PDFs of all adjacent nodes in a cluster.

Because the PDFs belonging to a node are stored far from each other inside a cluster, some cache misses will occur during the collision step while loading the required data. So, to minimize these kinds of cache misses another modification on CSoA has been done and the CAoSoA was introduced [32].

2.1.4 LBM algorithm

As mentioned before, the streaming step is a global operator that propagates PDFs to adjacent nodes. Since the data dependency arises in the streaming step of the LBM, some solutions for this issue introduced over the years named as "LBM algorithm". In the following, some of them are investigated.

Two-Step algorithm The first algorithm to deal with this kind of data dependency is the two-step algorithm [22]. In this algorithm, it is needed to traverse all PDFs twice in each time step (once for streaming and once again for collision). That means reading and writing data overhead which consequently increases the number of cache misses. It could be said that this algorithm increases code modularity and is easy for maintaining at the cost of performance.

Two lattices algorithm As the name implies, in this algorithm two lattices (grid) are reserved for marching in time [22] (one for time t and another for time $t+1$). One of the advantages of this algorithm is its capability to fuse the streaming and collision step which results in improving performance. But, the main problem with this algorithm is memory consumption. In comparison to the two-step algorithm, here it consumes twice the amount of memory. As the main concern of LBM is usually memory consumption, extra memory usage is not desired particularly for large systems. So, some modifications have been done to this approach both for improving performance and memory consumption.

Grid merging The idea of grid merging improvement [36] is to increase spatial locality and consequently reduce cache miss and therefore increase the performance. While this modification reduces the execution time, it consumes the same amount of memory as the two-lattice algorithm.

Grid compression In the grid compression algorithm besides increasing the spatial data locality, compare to the two-lattice algorithm, nearly half of the memory will be saved. The key point in this approach is the fact that for updating PDFs of any nodes only data of 8

(for $D2Q9$) neighboring nodes are required. So, a shift procedure in predefined order will be done to perform the streaming step. For more detail refer to Ref. [36].

Circular shift algorithm In Morphy’s method, [37] some supplementary space should be allocated to do the shift without overwriting data. This space is called reserved memory. After several times of performing the shift operation, the reserved space will be finished. So, array elements should be copied back to their initial positions and the base pointer be reset intermittently. To avoid the memory overhead and either pointer reset operation in Morphy’s method, in Ref. [38] the circular shift has been introduced. In a circular shift algorithm, an SoA memory layout is used to store PDFs which is required for the shift operation to optimally perform the streaming step. But, during the collision operation as the value of PDFs is stored far away from each other the data locality criterion is not satisfied very well which results in several cache misses. So, to reduce the number of cache misses the values associated with a part of the grid can be copied into a small array called blocks that fit entirely in cache memory. Hence, the collision operation is applied to these blocks. Consequently, two copy operations will be added to the updating process. In other words, the data is copied from the initial array to the blocks, and just after performing the collision operation, they are copied back to the original array containing PDFs [38].

Swap algorithm The swap algorithm is a modification of the two-step algorithm. The principle of this idea is to fuse the streaming and collision at the implementation level by exploiting only a few temporal variables. So, unlike the shift and two-lattice algorithms, there is no need to allocate additional memory to achieve fused implementation. In this algorithm, the data dependency is removed by explicitly exchanging the PDFs of the in-hand node with those of neighboring nodes. In other words, PDFs of an in-hand node are streamed to its associated neighbor node. Just after that, the collision step is applied to the in-hand node. For more details about this algorithm refer to Ref. [39].

2.2 Boundary conditions in LBMs

Although researchers have developed a variety of Boundary Conditions (BC) that are used for Dirichlet and Neumann boundaries such as Zou and He [40], periodic [9], symmetry [41], open boundary [41], full-way, and half-way bounce-back, one of the extreme difficulties with LBM is applying the BC.

The Zou and He [40] is one of the most common BCs in the field of LBM. It has been generalized to $D3Q19$ lattice by Kutay *et al.* [42] with the limitation that the inflow should be

perpendicular to the boundary plane. Then, Hecht and Harting [43] dropped that restriction which allows specifying the velocity on a node exactly. Therefore, their proposed BC is of the local type that is appropriate for parallel programming. Agarwal [44] derived the formulation of Zou and He BC at the inflow and outflow for the $D3Q27$ lattice to simulate turbulent flows.

A pressure BC was suggested by Zhang and Kwok [45] for fully developed periodic flows. After that, Kim *et al.* [46] provided a generalized periodic BC and compared the results of their simulations to the body force approach. Leclaire *et al.* [47] developed a periodic BC which is globally mass conservative. That method uses a simple proportional-integral-derivative controller to control the pressure difference accurately.

Dubois *et al.* [48] proposed a general bounce-back BC with the Taylor expansion method to increase the accuracy of the original bounce-back. They presented the formulation, particularly for the bottom wall and a $D2Q9$ lattice. Latt *et al.* [49] have investigated and analyzed some of the velocity Dirichlet BCs by using a $D2Q9$ lattice stencil for implementation.

The most common and straightforward BC for curved boundaries is the bounce-back method, but it is inaccurate since a staircase representation is used to model the geometry. Although using the equilibrium PDFs at walls is easy for implementation, it is not suitable for high Reynolds numbers [50]. To solve this problem, Guo *et al.* [51] developed a so-called "extrapolation method" which is using a basic extrapolation scheme. Then Shyy *et al.* [52] improved the curved BC suggested by Filippova and Hänel [53] to develop a second-order BC. Benim *et al.* [50] did an investigation to compare the above methods and the conventional finite volume method to study both the accuracy and performance. They concluded if a sufficiently fine grid was used for the simulations, all the LBM results were similar to the finite volume. However, the curved boundary treatment with the extrapolation method has a comparably inferior computational performance [50].

There are some studies to evaluate and improve the existing curved BCs [51, 54–57]. A general bounce-back scheme proposed by Zhang *et al.* [58] to implement thermal BCs of the convection-diffusion equation in complex geometries like those in porous media. Based on kinetic theory, the Maxwellian Diffuse (MD) reflection BC was developed by Ansumali and Karlin [59]. Then a combination of the bounce-back and the MD schemes named diffusive bounce-back scheme for flat stationary walls proposed by Chai *et al.* [60] and Verhaeghe *et al.* [61]. After that, Chai *et al.* [62], argued the possibility of simulating fluid flows with complex geometries, but the geometric integrity can not be preserved, which may result in an inaccuracy of simulation. To solve this problem, Tao *et al.* [63] proposed a local kinetic BC to model the curved walls precisely in microscale flow simulation.

Due to the interpolation and extrapolation in the procedure of most of the curved BCs,

there is a mass loss or gain at each time step during the simulations, which is called mass leakage [64]. This problem may become severe in periodic flows and demands for choosing appropriate curved BCs for high-Reynolds-number flows. So, Sanjeevi *et al.* [64] did a comprehensive study to test the capability of mid-grid bounce-back, Bouzidi [65, 66], unified schemes [67], interpolation-free [68], and Yin-Zhang [69] schemes and finally provided some recommendations for choosing a proper BC scheme for a stable and accurate simulation.

As the process of implementation of a BC is tedious and prone to error, especially when the boundaries have different orientations, some papers tried to propose a more general BC for Dirichlet velocity. In this way, it is possible to get rid of multiple implementations for different sides of a computational domain. Ginzburg *et al.* [70] proposed a generic BC for LBM named the "multi-reflection" method for Dirichlet and Neumann time-dependent BCs. In this method, three points are used to calculate unknown parameters. Hecht *et al.* [43] extended the Zou and He velocity BC to the *D3Q19* and also generalized it to be implemented free of the orientation of the boundary by introducing some extra parameters in the formulations. Then Lee *et al.* [48] generalized it for multi-speed lattices, which were proposed for single-speed LBM. Latt *et al.* [71] proposed a general regularized BC for multi-speed LBM, which is applicable for any lattice stencil. They assumed the velocity is predefined on the boundary (Dirichlet BC). In addition, it is possible to apply this BC on both the straight edges and corners. However, their method depends on the orientation of the boundary.

2.3 Parallel programming paradigms

Apart from the advantages of parallel programming, as the programmer is responsible for both identifying and implementing parallelism, designing and developing parallel programs is a time-consuming, complex, error-prone, and iterative process. So, developing platforms that are less demand on code developer proficiency in parallel programming is a hot issue in recent years. In this way, various tools have been available to assist the programmer with converting serial programs into parallel programs. The most common type of tools used to automatically parallelize a serial program is a parallelizing compiler or pre-processor such as ComputCpp [72], Par4All [73], PIPS [74], PLUTO [75], AutoPar [76], OMP2MPI [77], and Julia [78]. In this approach, the compiler analyses the source code and identifies opportunities for parallelism.

For someone with the constraint on time and either someone with limited skills in parallel programming, maybe the first and best option is automatic parallelization. However, there are several important drawbacks to automatic parallelization. So, there are some platforms for code development on heterogeneous computing systems which facilitate the effort of code

developers. It should be noted that for massively parallel programming it is needed to execute the code on multi GPUs or in some cases on multi CPUs and either to use full of the capabilities of supercomputers may be different types of CPUs and GPUs. In this regard there are some platforms such as Charm++ [79], OpenACC [80], and Apache Hadoop [81].

Maybe the simplest option for the parallelization of a computationally intensive code is to use the OpenMP paradigm. This method of parallel programming is straightforward. However, the developer should be careful about some issues such as variable scoping, false sharing, and loop scheduling. Some works used openMP for parallelization of LBM code [82–84]. By considering the memory limitations problem associated with openMP, and by taking into account the high memory resources demand of LBM applications, using pure openMP is not an appropriate answer for accelerating LBM computation.

Unlike OpenMP, which uses shared memory and multiple threads to speed up the execution time of a program, in MPI, there are some independent processes with their memory. That means in the MPI approach, data communication is used to transfer data between different memory resources. So, because of the capability of using more than one memory in the MPI approach, it is suitable for massively parallel code since more memory resources are available in comparison to openMP. In this regard, there are some works for LBM code parallelization using the MPI approach [25, 85–87].

Besides using pure MPI or openMP, there is another option to employ a “mixed model” which uses OpenMP for parallelization inside the node and MPI for transferring data between nodes. So, as each CPU node has its memory, it is possible to use the openMP approach to parallel the code to do the computation of each block of lattice [88, 89].

Although GPU devices are a powerful means of computation, it requires to use of spatial programming languages such as the Compute Unified Device Architecture (CUDA) developed and supported by NVIDIA [90] and the Open Computing Language (OpenCL) [91]. For LBM implementation there are so many works in this field [27, 92–96]. In addition, some combine MPI as a means of transferring data and GPUs for parallelization to use all advantages of distributed memory systems and GPU capabilities [97, 98].

Among the existing open source projects for LBM simulation studied by Motamed *et al.* [99], some of them including OpenLB [100], TheLMA [101], Palabos [102], HemeLB [103], and waLBerla [104] are investigated briefly in terms of the parallel programming aspect.

OpenLB is using a hybrid parallelization concept, i.e. MPI+openMP [89]. The domain decomposition approach used in OpenLB is based on the model used in its LBM data structure. Consequently, the communication strategy of this framework is tightly coupled to the data

structure of the LBM algorithm.

TheLMA framework stands for Thermal LBM on Many-core Architectures which initially used the CUDA platform for multi-GPU parallelization [101, 105] but they mentioned for more flexibility in domain decomposition using MPI is required for communication. So, in Ref. [106] they extended the framework to use MPI for communicating data.

Palabos is a library that is written in C++ and is based on MPI for parallel executions. In this library, semi-direct addressing can be used which divides the domain into some blocks. According to this data structure, one or multiple ghost layers around each block component are used to handle the MPI communication [102].

HemeLB [103] is developed particularly to simulate fluid flow in complex geometries. So, it is using an indirect addressing scheme in the data structure. Although it is possible to use any decomposition method such as parMetis [107], it has its library for decomposition named ALL (A load-balancing Library) [108]. In the MPI part of the HemeLB, only the data of the nearest neighbor nodes are communicated during the streaming step. These nodes are labeled as "domain-edge" in the preprocessing step. Then in each time step, at first the collision is done for them. After that, they are communicated. Then the collision and streaming are done for the rest of the nodes. In this way, the communication is hidden with computation [2]. In Ref. [107] the existing pure MPI code is used to develop CUDA-based parallel code for multi GPUs.

The waLBerla framework uses an approach named "block forest" to manage the domain partitioning into blocks. In this framework, the communication module is organized in a layered pattern. These layers start from low-level MPI functions up to classes that implement communication strategies. Only the topmost layer is specific to the data structure of the waLBerla framework. In this approach finding the required data for communication is the developer's responsibility as this layer is dependent on the data structure of the LBM code [104]. A scalable multi-GPU of the code is presented in Ref. [109] which is using CUDA.

2.4 Domain decomposition

If the load balancing in the MPI approach is not satisfied, the processes must wait at the synchronization points for the slowest process, until the simulation can proceed. Consequently, the run-time of the whole simulation increases, and the hardware resources at hand are utilized less efficiently due to the idle times. So, for good scalability in massively parallel runs, communication overheads need to be minimized and the computational load should be balanced. So, the concept of domain decomposition should be taken into account. Domain

decomposition concept originally comes from graph hypothesis, but because of the nature of computational application in CFD, it has been used for partitioning the computational mesh.

2.4.1 Existing decomposition methods

In this section, the existing decomposition techniques will be reviewed. We start with the simplest method in terms of implementation, named block partitioning. Then, those methods that are widely used in commercial software will be reviewed.

Block partitioning This is the simplest method for domain decomposition. In this approach, any regular domain such as a rectangle or cube is divided into equal sizes or even non-equal sizes. If just one-dimensional partitioning is applied on the grid it is called slice partitioning which assigns each slice to different processes. The box and cube partitioning use two and three-dimensional partitioning [110]. Apart from the simplicity of this method, because of the good memory access and cache hit this strategy is widely used in many computational applications [110]. But, as in this method a regular domain will be divided into subdomains without considering the type of nodes (solid or fluid) lay in each subdomain, it doesn't guarantee load balancing. So, it is not suitable for complex geometries such as porous media with large void fractions.

Orthogonal recursive bisection (ORB) algorithm In this approach, the computational grid is decomposed into two partitions in a way that the workload, due to the difference in the computational cost of fluid and solid nodes in the LB method, is optimally balanced. The same process is then applied recursively to every part based on orthogonal bisection [111]. Although the implementation of this method is straightforward and ensures good load balancing, it does not result in optimum communication balancing [103].

Multi-level k-way partitioning scheme The multi-level k-way partitioning method [112] is suitable for irregular systems. Although the implementation of this method is complicated, fortunately, it could be avoided by using existing software such as the METIS library. This method consists of three phases; Coarsening, Partitioning, and Uncoarsening. This algorithm is widely used in most packages such as METIS [113], ParMETIS [114], PaGrid [115], JOSTLE [116], are investigated in section 2.4.2.

Space filling curves There are many modifications to the preliminary idea of space-filling curves such as one-constrain and two-constraints domain decomposition [117]. The detail of this algorithm could be followed in Ref. [118]. In this method, any computational mesh is linearized by the space-filling curves method. Then by storing the geometric data (coordinates of nodes in the LBM) in a one-dimensional array and then this array is simply cut into segments. The most interesting advantage of this method is preserving locality which means elements that are close in a 1D array are also close in the domain [119]. Although this method is used for approximately 40K particles in SPH simulation [118] the speed of this partitioning strategy is suboptimal since the computational cost usually increases as $O(N \log N)$ (N is the number of nodes) which makes them unsuitable for large systems.

Graph growing partitioning (GGP) In this algorithm, P partitions grow from P corresponding seeds (nodes in LBM) employing a breath-first expansion algorithm and using an iterative method to partition the domain [120]. As this algorithm does not scan the computational domain in a predefined manner, it needs to store global data for efficient execution which is one of the main limitations of most decomposition methods. However, the nature of this method is likely to yield good communication and computational load balancing [121]. Mazzeo *et al.* [103] improved on the original GGP algorithm which has some advantages such as execution time reduction by three orders of magnitude compared to the original GGP approach. However, the perfect load balancing that characterizes the original GGP approach is no longer guaranteed.

Cell-based decomposition method The most common domain decomposition methods are using an iterative approach to satisfy the load balancing. But, in the cell-based method, in the beginning, load balancing is performed by grouping the nodes in such a way that each subdomain contains almost the same number of fluid nodes. Then instead of balancing the load, a recovering interface step is done. Recovering interface means, finding those cells laying on the border of each subdomain to be used later in the communication step.

Compare to ORB, the Cell-based decomposition method has the advantage of exact load balancing. Furthermore, this method guarantees regular communication patterns, which means optimal communication between processes. That is because all the fluid nodes which need to be communicated are picked up once without a pointer jump. Moreover, this method doesn't need any search and resorting operations. Finally, it is simple for implementation and does not require extra CPU time to optimize the decomposition of the domain [112].

2.4.2 Existing packages for domain decomposition

In this section, some packages for domain decomposition will be reviewed and their advantages and drawbacks will be investigated.

METIS This package was developed at the University of Minnesota [113]. The last release was in 2016 but, the stable version is METIS 5.1.0 which has been released in 2013. The algorithm implemented in this package is based on the Multi-level k-way partitioning scheme and multi-constraints partitioning schemes. Although this is a sequential code, it can handle a mesh with millions of cells in just a few minutes. However, it is possible to use a heterogeneous mesh (a mesh consisting of different cell types such as triangular or quad cell) [113]. Also, this package could be used for partitioning a grid for a heterogeneous computing device.

hMETIS It is a version of METIS that is used for partitioning hyper-graphs [113]. A graph in which an edge can join any number of vertices is called a hyper-graph. The algorithm used in this package is based on a multi-level hypergraph partitioning scheme. One of the characteristics of this version is the quality of partitioning. However, it is slower than METIS. So, this package should be used in the case that the computation efficiency is not critical [113].

ParMETIS It is a parallel version of METIS that is suitable for large-scale numerical simulation [113]. It provides some useful functions for partitioning graphs and meshes and for matrix reordering. This package has been tested for the partitioning of a 7.5 million vertex graph in about 7(s) on 128 processors [114]. Although this software is suitable for a large-scale system, it should be noted that it is applicable just for homogeneous computing systems.

JOSTLE It has been released first in 1995 and developed at the University of Greenwich [116]. Recently this package has been commercialized and published as Networks. The algorithm of this software is mostly based on multi-level partitioning which is implemented in serial and parallel paradigms. It is also suitable for heterogeneous computing systems [116].

PART This is a package for partitioning graphs and computational mesh for the heterogeneous computing device [122]. PART uses simulated annealing as a search algorithm. So, this package is not fast for serial execution, but its parallel version has super-linear speedup in most cases [122].

PaGrid The algorithm of this software [115] uses the idea of directly minimizing execution time instead of indirectly minimizing communication costs. It is based on a multi-level graph partitioning method. Based on the characteristics of the computing system, i.e. computation power and communication speed, some load unbalancing is required for minimizing the execution time of an application. In PaGrid it is assumed that an estimation of communication to computation cost for a given application is known and based on that the partitioning will be performed [115]. Dethier used PaGrid for partitioning the grid for the LBM application to execute the code on a heterogeneous computing device [123].

CHAPTER 3 RESEARCH GAP, OBJECTIVES AND OUTLINE

In this chapter, based on the literature review which has been done in Chapter 2, several research gaps have been identified. Then, according to the research gaps the objectives of the thesis are derived as explained in the following. Then, an outline is presented at the end of this chapter describing briefly the three articles prepared.

3.1 Research gap

According to the literature review, three research gaps are found in the area of LBM algorithm, boundary conditions, and parallel programming for distributed memory systems.

3.1.1 About LBM algorithm

By considering the literature, it could be summarized that the following issues are still open to be solved in the LBM field:

- Simplifying the streaming step implementation;
- Increasing the data locality by using an appropriate data structure compatible with both the core functions, i.e. streaming and collision, of the LBM; and
- Avoiding the storing of the solid nodes.

If each of the above problems is considered individually, there is a solution for each of them as investigated in Chapter 2. But the problem is there is no solution that solves all the above at the same time.

3.1.2 About difficulties of boundary condition implementation

The boundary condition is one of the most challenging parts of the LBM simulations. One of the reasons which makes it hard to implement a new boundary condition is the dependency of the formulation on the orientation of the boundary nodes. So, any boundary condition formulation should be derived, implemented, and verified for any boundary orientations. In summary, the process of implementing a BC consists of (1) deriving its formulations, (2) programming the formulations, and (3) debugging and verifying the code. So, this process is tedious and prone to error, especially when the formulation of a BC should be derived

and implemented for all the boundaries with different orientations. In the LBM, this is an obstacle to developing new BCs.

3.1.3 About MPI communication algorithm

Because of the importance of parallel computing for an LBM simulation, evaluating any newly developed LBM algorithm in terms of capability to be executed on distributed memory system is crucial. Hence, any new proposed LBM algorithm should be extended and evaluated in terms of parallel computing. In the literature, there is not any general MPI framework for LBM that could be used individually. In the other words, in all the investigated open-source programs and works the MPI programming depends on the data structure of the base LBM program they have developed on top of them. So, extending and investigating any new LBM algorithm and data structure from the parallel computing viewpoint is one of the difficulties in the LBM.

3.2 Objectives

The main objective of the project is to improve upon the current LBM algorithms and create one that will work efficiently with distributed memory systems as well as with a single processor CPU in order to simulate large systems with irregular geometries. The following items are related sub-objectives:

1. Introducing a new LBM algorithm for further improvements, in terms of increasing data locality, using a simple data structure, and using memory resources efficiently.
2. Introducing a BC framework to simplify the BC implementation.
- 3a. Designing a new algorithm for MPI communication in such a way that, at the programming level, it would be possible to hide the parallel programming layer of the code.
- 3b. As a consequence, developing the parallel version of the introduced LBM algorithm in point 1 above.

3.3 Outline

The technical viewpoint of this thesis is presented in three articles as follows:

- Chapter 4: Present the first article published in *Journal of Computational Science*; In this paper, the newly introduced LBM algorithm is presented and it is evaluated in

terms of performance and memory usage for both dense systems and low void fraction systems.

- Chapter 5: Present the second article published in *Computer Physics Communications*; In this paper, the introduced BC framework is presented and evaluated to implement some popular BCs of the LBM.
- Chapter 6: Present the third article submitted in *Computer Physics Communications*; In this paper, a general algorithm for MPI communication is introduced and the new LBM algorithm, presented in paper 1, is extended for execution on distributed memory systems.

Following these three technical chapters, a general discussion of the thesis followed by a conclusion and recommendations for future work are presented in Chapters 7 and 8 respectively.

CHAPTER 4 ARTICLE 1: SIMPLE LATTICE BOLTZMANN METHOD ALGORITHM WITH LOW MEMORY USAGE

- M. Namvar and S. Leclaire, “Simple lattice Boltzmann method algorithm with low memory usage,” *Journal of Computational Science*, vol. 62, p. 101723, 28 May 2022. Available at: <https://doi.org/10.1016/j.jocs.2022.101723>

Unlike the existing paper which tries to modify the data structure to increase the data locality, in the proposed idea both the streaming and collision steps are done in one loop by traversing all the PDFs. Hence, a simple shift is used to do the streaming step. So, as the loop is on the PDFs it is possible to use a structure-of-array memory layout to simplify the streaming step implementation. In addition, by avoiding storing the solid nodes the proposed algorithm is suitable for low void fraction systems. In this way, apart from the decrease in memory usage, as no computation is done for solid nodes the computational resources are used efficiently also.

4.1 Abstract

As lattice Boltzmann simulation codes require updating several million floating point values thousands of times, optimized implementation and efficient data structure are crucial to minimize memory consumption and use all the available computational power. In this paper, a new lattice Boltzmann algorithm is proposed, suitable for the simulation of large-scale systems by avoiding storing the data of solid nodes. As a structure-of-array memory layout is used, the streaming step is performed by a simple shift operation. The advantages of this algorithm are the simplicity of the memory layout and streaming function implementation. Because the algorithm utilizes an indirect addressing scheme, particularly for low void fraction geometries the memory resources are used efficiently. As no computation is carried out for solid nodes, a good performance for irregular geometries with low void fractions is also achieved. To verify the implementation of the proposed algorithm several academic benchmarks are studied, including Poiseuille flow, Cavity flow, and Cylinder flow. A theoretical estimation of the memory consumption is carried out for different void fractions. Then the performance of the implemented code is investigated by simulating flow around two complex geometries and the memory usage and computational performance compared to the Palabos code.

4.2 Introduction

The Lattice Boltzmann Method (LBM) is one of the interesting approaches to simulate fluid flows in complex geometries. However, it requires high computational power and memory resources. This is because the LBM generally uses an explicit scheme with small time steps, and therefore requires thousands or even millions of time steps to update several million floating-point values up to the final time of interest. Optimization of LBM algorithms can therefore lead to lower energy consumption.

In addition, one bottleneck of the LBM is memory usage. Since available memory resources are always a limitation, the data structure of an LBM code should consume as little memory as possible. Also, the mathematical operations of the LBM are comparatively small in most cases, the load/write data from/to memory is a computational cost bottleneck. The first question regarding the data structure of any LBM code, then, is what approach to apply for storing and accessing both the macroscopic variables and Probability Distribution Functions (PDFs). Primary issues in data structure of the LBM are *memory ordering*, *memory addressing*, and *memory layout* [16]. Therefore an *algorithm* is used to modify the content of the data structures according to the requirement of the physics. At the implementation level, the LBM algorithm affects the memory layout and consequently memory usage and performance. These four topics are discussed in the following paragraphs.

Memory ordering The way to assign a number (index) to any node is memory ordering. The most common method is lexicographical indexing [16]. Although lexicographical indexing is straightforward, flexibility is a criterion that should be considered, particularly when the aim is to simulate flow fields around complex geometries. Some works use an unstructured grid as the LBM grid [17–20]. However, it should be noted that generating an unstructured grid for complex geometries is one of the reasons for developing new ideas in the field of CFD that try to get rid of this complex mesh generation process. By using an unstructured grid data structure, nodes can be indexed according to their type (solid or fluid). As in Refs. [22, 124] the types of nodes, *i.e.* solid or fluid, are stored as additional data to be used by the LBM algorithm. In the proposed method, besides avoiding storing data for solid nodes, fluid nodes can be numbered in any desired way. The drawback of this approach is consuming extra memory to store the neighboring node.

Memory addressing The way of accessing and addressing the data of each node, such as geometry data, macroscopic variables, and PDFs, is called memory addressing. There are two main approaches to this: *direct* and *indirect* addressing. There are also other methods

such as *semi-direct* and *locally direct addressing*.

Direct addressing is the most common approach for addressing data [125] where the computational domain is embedded in a bounding box that contains all nodes, including solid and fluid nodes. In addition to the simplicity at the implementation level, this approach is suitable for dense systems such as cavity or channel flow. However, for small void fractions and complex geometries, this method could be inefficient as it may result in wasting memory resources and computational power.

Indirect addressing was introduced by Mattila *et al.* [22]. In this method, memory is allocated to fluid nodes, then an additional array, called a fluid index array, is used to determine the fluid node indexes. It is possible to apply the collision step on just the fluid nodes. Vidal *et al.* [24] used this approach to improve the performance of large heterogeneous porous media simulations. Hasert *et al.* [25] then developed a new kind of indirect addressing based on octree mesh generation idea, to handle complex geometries and also multi-species flows. In this approach, an array containing the neighboring list (connectivity matrix) is required. It therefore demands a considerable amount of memory resources. Also, during the streaming step, the data of neighboring lattices needs to be loaded by referring to the neighboring list, which includes the computational cost to load the index of neighboring nodes. Geier and Schönherr [23] introduced the esoteric twist approach that is compatible with indirect addressing. In terms of memory usage this approach is optimized as it doesn't store solid nodes. But, to perform the streaming step, all PDFs are classified into two groups and push or pull schemes are applied in a sophisticated predefined order. In summary, indirect addressing would be efficient for small void fraction problems but increases the complexity of code while decreasing the computational efficiency for dense systems [16].

Semi-direct addressing saves some memory resources by storing data at fluid nodes while only using indirect addressing. This is achieved by using a directly addressed array of pointers to the fluid nodes [26]. There are some difficulties applying boundary conditions with direct addressing and also semi-direct addressing. Mattila *et al.* [39] improved semi-direct addressing and introduced a simpler way to treat boundary conditions: namely, by storing an indirect data structure just for boundary nodes. Although the total memory consumption increases slightly, as the boundary node data are stored twice (once in semi-direct addressing and once in indirect addressing), this approach has considerable advantages in terms of implementation and performance.

Locally direct addressing decomposes the computational domain into cuboids [16, 28, 103] and removing the ones that do not contain any fluid nodes. For complex geometries, with this technique it is possible to decrease memory consumption considerably. A special data structure is used to determine the relationship between the cuboids. The sets of cuboids may all be the same size or they may be different size as well as be placed arbitrarily. While this approach is efficient for complex geometries, it may not reduce memory consumption significantly in the case of porous media, as it uses direct addressing inside each cuboid.

Adaptive mesh refinement In theory, a finer grid allows for greater accuracy, but requires greater computing resources. Therefore, a non-uniform mesh would be a more efficient idea for higher accuracy and lower computational cost. The adaptive mesh refinement (AMR) [29, 30] technique could meet this criterion. In this approach, the objective is to reduce the grid density in the smooth gradient area and increase the grid in the sharp gradient area of the computational domain. In multi-block AMR [31], the computational grid is divided into several blocks. A direct addressing scheme could then be used within each block, while a special treatment should be employed at the interface of each block. Another approach is to use an unstructured Cartesian grid as the node distribution for the LBM. This approach uses a tree-like data structure. One of the advantages of AMR is that, in the case of a high void fraction and a smooth gradient region, it can significantly reduce the number of nodes. Despite this and other advantages, the tree-like data structure of this approach has its own complexity that can complicate implementation and programming.

Memory layout There are a variety of techniques available to store the data required for an LBM simulation. The particular scheme for storing the data affects both memory consumption and performance. Therefore, the most important aspect of an LBM code implementation is accessing the node data. While conventional Array of Structure (AoS) and Structure of Array (SoA) schemes [126] satisfy the requirements of an efficient data structure, there are some modifications that improve access to the data. Memory layout plays a major role in determining application performance on both the CPU and GPU. In this regard, there are two different approaches. In AoS the PDFs at each node are stored in a contiguous chunk of memory. They are also stored adjacently according to the node index. So there is an array containing a variety of structures where each structure contains the PDFs of a node [16, 127]. The SoA approach stores each PDF component contiguously with a specific velocity and direction belonging to each node. Hence, there is a structure containing arrays where each array contains specific PDFs for all nodes. Because the streaming and collision step requires both global and local access to data respectively, to gain the best performance

different memory layouts should be employed. Thus the LBM core functions ideally need two different memory layouts—one for each time step (i.e. collision is more compatible with AoS, while streaming is more compatible with SoA). There are two options to deal with memory layout: the first is to use a different memory layout for each of the steps; the second is to have a unique memory layout that satisfies the contradictory requirements of both the collision and streaming steps. Regarding the first option, Majeti *et al.* [35] demonstrated that using a single memory layout is generally not optimal. For the second option, Calore *et al.* [32, 128] introduced the Cluster Structure of Array (CSoA) memory layout. In this approach, a cluster of nodes is stored in a SoA format according to the size of the cache. The memory layout of each cluster is then suitable for the collision step while the entire memory layout is compatible with the streaming step. However, some cache misses will still occur during the execution of the streaming step, as reported in Ref. [32]. A modification of CSoA by Calore *et al.* [32] resulted in the Clustered Array of Structure of Array (CAoSoA), which has a memory layout more compatible for the collision step.

LBM Algorithm Given there are a variety of LBM algorithms that affect the implementation, what is the main issue that the various algorithms are trying to solve? The streaming step is a global operator that propagates PDFs to adjacent nodes. For instance, in Fig. 4.1 and using the $D2Q9$ stencil as an example, the f_3 of node 1 needs to be updated by pulling f_3 from node 3. Now for updating the f_3 of node 2 the f_3 of node 1 in the previous time step is needed; however it has already been updated. So, the f_3 of node 1 from the previous time step is not accessible. This is called "data dependency", for which there are some solutions.

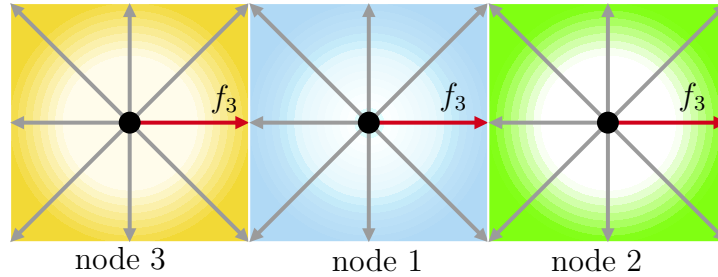


Figure 4.1 Data dependency in LBM streaming function.

Two-step algorithm The first algorithm to deal with this kind of data dependency is the two-step algorithm [22]. This algorithm is also called AA-pattern, which was originally introduced by Peter *et al.* [93]. As the name suggests, at each time step, the marching step is split into two: streaming and collision. This means, in the first step, the streaming is

completed for every node in the domain. Then the collision step is carried out. In this algorithm all the PDFs therefore need to be traversed twice in each time step (once for streaming and once again for the collision step), which creates reading and writing data overhead and consequently an increase in the number of cache misses. Hence, this algorithm increases the code modularity and maintainability at the cost of performance.

Two-lattice algorithm In this algorithm two lattices (grid) are reserved that march in time with each other [22] (one for time t and another for time $t+\Delta t$). Suppose a pull scheme is used for the streaming step: to update the PDFs in this step, the required PDFs are read from the adjacent nodes as the first step. Then the PDFs of the in-hand node are updated. Immediately following that, the collision step is completed. To deal with the data dependency, the new PDFs are written to the second array. Consequently, at the end of the streaming and collision step, for all of the nodes, the PDFs, which are already stored in the second array, are the $t+\Delta t$ data. For the next time step, the PDFs are read from the second array, and after streaming and collision, are stored in the first array. The main problem with this algorithm is memory consumption. Compared to the two-step algorithm, it consumes twice the amount of memory. As the main concern of LBM is usually memory consumption, extra memory usage is not desired, particularly for large systems. Some modifications to this approach have resulted in improving both performance and memory consumption.

Grid merging Pohl *et al.* [36] improved grid merging by modifying the two-lattice algorithm so as to increase spatial locality, thereby reducing cache misses and increase performance. At the implementation level and as mentioned above, there are two grids corresponding to time t and $t+\Delta t$. If the two grids are stored as close as possible to each other, the spatial locality will be increased. Therefore, when a chunk of memory loads to the cache, to update the PDFs of a node, both memory corresponding to time t and $t+\Delta t$ exist in the same cache. Thus, while this modification reduces the execution time, it consumes the same amount of memory as the two-lattice algorithm.

Circular shift algorithm In this method, a SoA memory layout is used to store PDFs which is required for the shift operation to optimally perform the streaming step. However, during the collision operation, as the PDFs values are stored far away from each other, the data locality criterion is not well satisfied, which results in some cache misses. To reduce the number of cache misses the values associated with a part of the grid can be copied into a small array called blocks that fit entirely in cache memory. The collision operation is then applied to these blocks. Consequently, two copy operations are added to the updating process. In

other words, the data is copied from the initial array to the blocks, and after performing the collision operation, are copied back to the original array containing PDFs [38].

Some algorithms, such as the swap algorithm, which is a modification of the two-step algorithm, fuse streaming and collision at the implementation level by exploiting only a few temporal variables. Unlike the two-lattice algorithm, there is no need to allocate additional memory to achieve fused implementation [39]. In the grid compression modification [36], besides increasing the spatial locality, nearly half the memory is saved compared to the two-lattice algorithm. This approach uses the fact that to update the PDFs of any node, the data of only eight (for $D2Q9$) neighboring nodes are required. Therefore, to perform the streaming step a shift procedure in predefined order is required.

4.2.1 Summary

In the above paragraphs, we reviewed the existing LBM data structures and algorithms. In the following paragraph, we briefly explain the concept of the proposed LBM algorithm. In addition, considering the most important aspects of the LBM, we state the difference between the proposed algorithm and existing LBMs. First of all, we note that to simulate a large-scale system, it is not acceptable to store solid node data, as storing solid node data results in additional memory consumption. Moreover, performing the unnecessary calculation of solid nodes means wasting computational power. As investigated above, although using an unstructured grid may be helpful to avoid storing solid nodes, generating this kind of grid is challenging in itself. Another possibility to avoid storing solid nodes is using a neighboring list, which is the path proposed in this work. However, unlike the data structure of a conventional unstructured grid, which stores the neighboring nodes, we store the associate node of any PDFs. This approach has at least two advantages: (1) it is possible to do the streaming step efficiently as a SoA data layout may be used to store the PDFs, and (2) it is possible to traverse PDFs to perform the collision step instead of traversing nodes, as usual in a lattice Boltzmann algorithm. Therefore, with the same memory layout (*i.e.* SoA), both the streaming and collision steps may be done efficiently. Although in our approach extra memory is required to store the associate node of PDFs, this is compensated by avoiding storing the solid node, particularly for low void fraction geometries. There are some modifications on AoS and SoA memory layout, such as CSoA and CAoSoA, which are more compatible with both the streaming and the collision functions, and that increases the data locality. However, these memory layouts increase the complexity of code at the implementation level. It appears that using pure AoS or SoA would be an ideal memory layout. Our solution is to use a SoA data structure while increasing data locality by changing the LBM algorithm, instead of

modifying the data structure. To increase the data locality, one option is to load/write data just once at each time step, which means using a fusing algorithm. The main problem of using a fusing algorithm, however, is the difficulties at implementation. We propose an algorithm that traverses PDFs to perform both the streaming and collision function, and as such, it would be easy to implement a fusing algorithm. In addition, the macroscopic calculation is carried out along with the streaming and collision step so this increase data locality.

Overall, considering the above requirements and problems related to existing LBM algorithms, a new LBM algorithm is introduced that satisfies the following requirements:

- Not storing data of solid nodes;
- High data locality; and
- Simple for implementation.

The rest of this paper is organized as follows. Section 4.3 outlines the formulation for the proposed LBM. Section 4.4 describes in detail the proposed algorithm. Section 4.5 presents the results obtained from academic test cases to verify the proposed algorithm. Then the memory usage and computational performance is investigated on a set of complex geometries, and the performance is compared to the Palabos [129] code. Finally, in the conclusion section, we discuss the advantages, limitations, and capabilities of the proposed algorithm.

4.3 Lattice Boltzmann method

The Lattice Boltzmann Method (LBM) [3] is derived from the method of Lattice Gas Automata (LGA). Each step of an LGA consists of a collision and streaming step. The LGA however, suffers from several major drawbacks, such as intrinsic noise, non-Galilean invariance, unphysical velocity-dependent pressure, and large numerical viscosity [4]. In 1986, a correct NS equation derivation starting from the lattice gas automata on a hexagonal lattice was proposed by Frisch *et al.* [5]. In 1991 a simplification with a lattice Bhatnagar-Gross-Krook collision operator, which assumes that PDFs are close to a local equilibrium, paved the way for further development of the method [6–8].

The LBM model has n spatial dimensions and m PDFs, where each node is superimposed by a $DnQm$ discretization called "lattice stencil". For example, Fig. 5.1 depicts the PDFs associated with a node for a $D2Q9$ lattice stencil velocity space discretization.

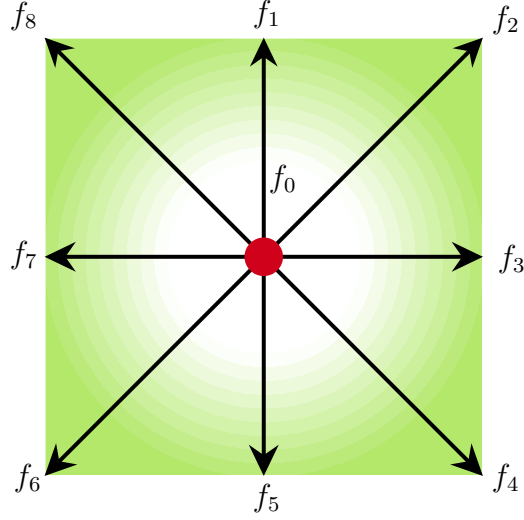


Figure 4.2 Typical two-dimensional lattice stencil of $D2Q9$ with the node (in the center with color red) and PDFs indexation f_i with $i = 0, 1, \dots, 8$.

In LBM, like other numerical methods, it is necessary to discretize the Cartesian space by introducing a set of nodes. It is also required to discretize the velocity space of the PDFs. For example, the $D2Q9$ lattice stencil [9] as depicted in Fig. 4.2 where $m = 9$ is the total number of PDFs belonging to the lattice stencil, and the PDFs are noted as:

$$f_i(\vec{x}, t); \quad i = 0, \dots, m - 1 \quad (4.1)$$

The PDFs may be viewed as a representation of the probability of finding a group of fluid molecules at location $\vec{x}$ and at time t with discrete velocity $\vec{c}_i$ [10]:

$$\vec{c}_i = (c_{i,x}, c_{i,y}) = \begin{cases} (0, 0), & i = 0 \\ (\cos \theta_i, \sin \theta_i) c, & i = 1, 3, 5, 7 \\ (\cos \theta_i, \sin \theta_i) \sqrt{2}c, & i = 2, 4, 6, 8 \end{cases} \quad (4.2)$$

where $\theta_i = \frac{\pi}{2} + \frac{\pi}{4}(1 - i)$, $c = \Delta x / \Delta t$ is the lattice velocity, Δx is the spatial step, and Δt is the time step. The most common form of the lattice Boltzmann evolution equation in the literature is as follows [11]:

$$f_i(\vec{x} + \vec{c}_i \Delta t, t + \Delta t) = f_i(\vec{x}, t) - \frac{f_i(\vec{x}, t) - f_i^{eq}(\rho(\vec{x}, t), \vec{u}(\vec{x}, t))}{\tau} \quad (4.3)$$

Eq. 5.1 is usually solved in two steps, respectively named collision and streaming. The first collision model which paved the way to recover the nearly incompressible NS equations was

achieved by Qian *et al.* [12]. This was illustrated by Chen *et al.* [130] through a Chapman–Enskog expansion. This operator is often called the LBGK [12] collision:

$$f_i^*(\vec{x}, t) = f_i(\vec{x}, t) - \frac{f_i(\vec{x}, t) - f_i^{(eq)}(\rho(\vec{x}, t), \vec{u}(\vec{x}, t))}{\tau} \quad (4.4)$$

where f_i^* represents the distribution function after the collision, which is also named the post-collision PDF, and τ is the relaxation parameter which measures the average time for the PDFs to relax to its equilibrium via collisions [13]. The evolution Eq. 4.3 is completed with the streaming (or propagation) step:

$$f_i(\vec{x} + \vec{c}_i \Delta t, t + \Delta t) = f_i^*(\vec{x}, t) \quad (4.5)$$

with the left hand-side of Eq. 4.5 being the post-streaming PDF.

In Eq. 4.4, the f_i^{eq} is the local equilibrium distribution function for isothermal fluid and is defined as:

$$f_i^{eq} = \rho w_i \left(1 + \frac{\vec{c}_i \cdot \vec{u}}{c_s^2} + \frac{(\vec{c}_i \cdot \vec{u})^2}{2c_s^4} - \frac{\vec{u} \cdot \vec{u}}{2c_s^2} \right) \quad (4.6)$$

where w_i is a weighting factor. For example, for the $D2Q9$ lattice stencil it is defined as:

$$w_i = \begin{cases} 4/9, & i = 0 \\ 1/9, & i = 1, 3, 5, 7 \\ 1/36, & i = 2, 4, 6, 8 \end{cases} \quad (4.7)$$

The parameter c_s is the LBM speed of sound [14]:

$$c_s = \frac{c}{\sqrt{3}} \quad (4.8)$$

The macroscopic quantities of interest are the fluid density ρ and velocity $\vec{u}$, defined as:

$$\rho(\vec{x}, t) = \sum_{i=0}^{m-1} f_i(\vec{x}, t) \quad (4.9)$$

$$\vec{u}(\vec{x}, t) = \frac{1}{\rho} \sum_{i=0}^{m-1} f_i(\vec{x}, t) \vec{c}_i \quad (4.10)$$

The kinematic viscosity of the fluid is determined by the following relation [9]:

$$\nu = c_s^2 \left(\tau - \frac{\Delta t}{2} \right) \quad (4.11)$$

and the pressure p is derived from an isothermal equation of state [9]:

$$p = c_s^2 \rho \quad (4.12)$$

Note that with the LBM it is common to use a system of units where the discrete Cartesian spatial and time steps are unity, i.e. $\Delta x = \Delta y = \Delta t = 1$, so that the lattice velocity $c = 1$. This is the system of units adopted for the rest of this work.

4.4 The proposed LBM algorithm

In this section, the new algorithm is presented in pseudo-code as illustrated in Algorithm 1. Also, a short description of the variables appearing in this pseudo-code is provided in Table 4.1. First, the pre-processing step of the proposed algorithm is explained in detail. Then, each core function of the algorithm is detailed. We also provide the information required about the arrays for implementation at each part. Once the data structure is explained, the three main steps of the LBM calculation, referred to as collision, streaming, and macroscopic steps, are explained in detail. Finally, we discuss how the boundary conditions are applied to the proposed algorithm.

Table 4.1 Variables used in Algorithm 1 with short descriptions.

Variable	Description
$iVel$	A counter to traverse PDF discrete velocities.
m	Number of the discrete velocities, e.g. $m = 9$ for $D2Q9$.
$firstPDF$	Index of first PDFs with discrete velocity $iVel$.
$lastPDF$	Index of last PDFs with discrete velocity $iVel$.
$iPDF$	A counter to traverse PDF.
$iNode$	Associate node index of a PDF.
$nNode$	Number of fluid nodes constructing the grid.
PDFn	An array to store the associate node index of PDFs.
PDFv	An array to store PDF values.
(R,U,V)	Arrays to store macroscopic quantities.
(sR,sU,sV)	Temporary arrays to store momentum quantities.

Algorithm 1: proposed LBM algorithm.

```

1 Pre-processing step
2 while Time marching condition do
3   for  $iVel = m-1$  to 0 do
4     Load discrete velocities and weights
5     Define  $firstPDF$  and  $lastPDF$ 
6     for  $iPDF = lastPDF$  to  $firstPDF$  do
7       Load  $iNode \leftarrow \mathbf{PDFn}[iPDF]$ 
8       Load  $f_i \leftarrow \mathbf{PDFv}[iPDF]$ 
9       Load  $(\rho, u, v) \leftarrow (\mathbf{R}[iNode], \mathbf{U}[iNode], \mathbf{V}[iNode])$ 
10      Collision Function
11      Streaming Function
12      Macroscopic Function
13    end
14  end
15  Apply Boundary Condition
16  Calculate Macroscopic Values
17 end
18 Post-processing step

```

4.4.1 How to index nodes

The first step for implementing the algorithm is node indexing. As mentioned above, node indexing is assigning a number to the nodes. A lexicographical indexing approach is applied. At this step, we use the concept of a regular grid, which means a bounding box containing the whole geometry in which any node has an (i, j) index. For instance, in Fig. 4.3, a 3×3 regular grid is depicted where all the nodes are indexed by lexicographical indexing.

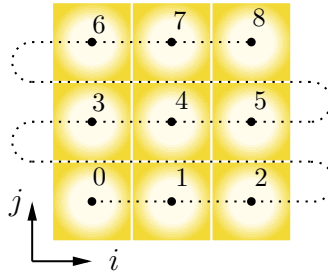


Figure 4.3 A (3×3) regular grid with a lexicographical approach for node indexing.

At implementation, it is enough to define how to obtain the node index when the (i, j) index is given. For a regular grid of size $(iMax \times jMax)$, the index of any node is obtained by the

following equation:

$$iNode = j \times iMax + i \quad (4.13)$$

However, the regular grid is needed for just the PDF indexing presented in section 4.4.2. Then, it is no longer needed as the solid node data are not stored in the proposed algorithm.

4.4.2 How to index PDFs

PDF indexing is the most important step for implementing the proposed algorithm. The streaming step streams each PDF to the neighboring one. Based on this fact, one key point for the proposed work is to store PDFs in such a way to perform the streaming step by just a shift operation. To do that a SoA memory layout is used to store PDF values as depicted in Fig. 4.4. The container array for storing the PDF values is named **PDFv**, and is used at line 8 of Algorithm 1. Then inside each sub-structure F_i , as illustrated in Fig. 4.4, the PDF values are stored based on their discrete velocity direction and the index of their associate nodes.

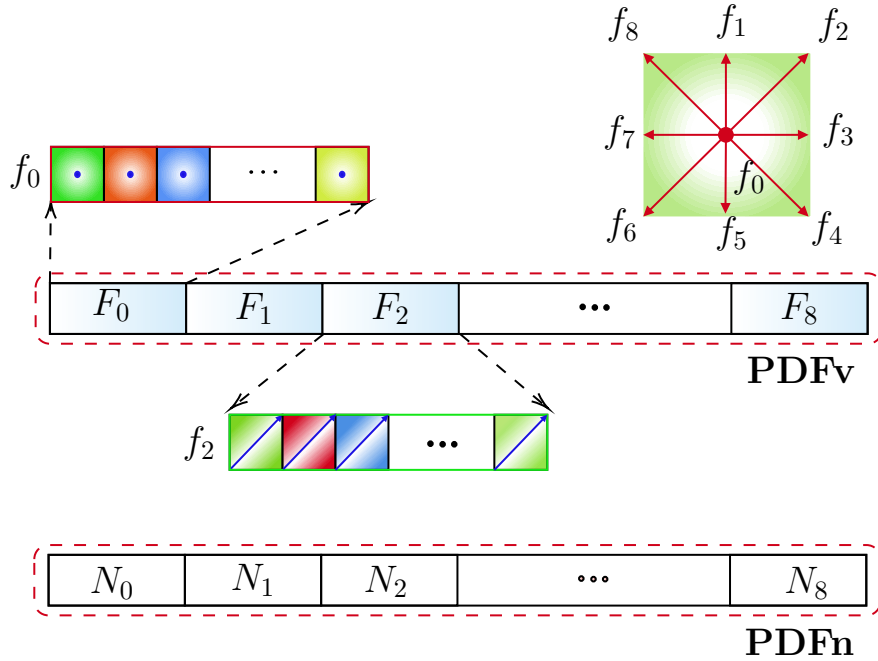


Figure 4.4 A SoA memory layout is used to store PDFs in the array **PDFv** and the index of nodes in the array **PDFn**. In this approach all the f_i PDFs for index i are stored in the F_i structures and all the node indexes for index i are stored in equivalent N_i structures.

All the PDFs are indexed as depicted in Fig. 4.5. We begin with the F_0 structure storing the f_0

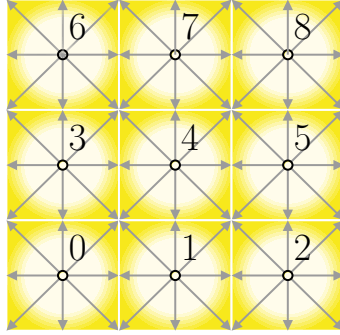
PDFs, *i.e.*, the PDFs with a zero velocity. These types of PDFs may be stored in an arbitrary order. However, in this paper, they are indexed based on the index of their associated node as depicted in Fig. 4.5a. There are nine nodes in Fig. 4.5a, so in this example, nine PDFs in total belong to the F_0 structure. To explain the concept of PDF indexing in the F_1 structure for f_1 PDFs, let us suppose there are some imaginary arrows, A0, A1, and A2, which lay on all the PDFs belonging to the nodes with the same $i = 1$ index, as shown in Fig. 4.5b. All the PDFs laying over these arrows are indexed starting from the first PDF on an arrow. As for f_1 , a similar procedure for the F_2 structure for f_2 PDFs is used. For the PDFs depicted in Fig. 4.5c there are B0, B1, ..., B4 imaginary arrows which show how to index the PDFs. For the rest of the PDFs, *i.e.* $f_3, f_4, \dots$, and f_8 , a similar procedure following imaginary arrows is applied to index all existing PDFs.

It is crucial to point out that PDFs are indexed by following their velocity direction (except for f_0 PDFs). This allows us to do the streaming step by a simple shift operation. The above simple procedure using imaginary arrows explains how to align PDFs inside each F_i structure of the array **PDFv**. It should also be noted that if a PDF belonging to a solid node is encountered while traversing the imaginary arrows, this in-hand PDF is simply skipped and not stored in **PDFv**. This completes the explanation of the array **PDFv**.

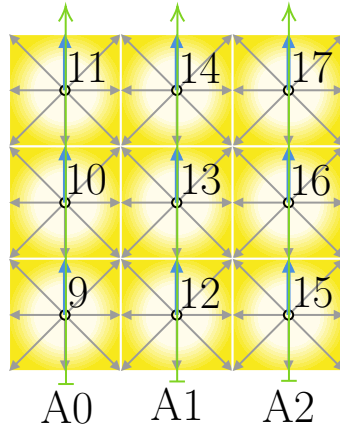
The construction of the array **PDFn** is given in detail below by providing an example also based on Fig. 4.5. First, suppose the regular grid is bounded by $iMax$ and $jMax$ with the type of nodes well defined, *i.e.* it is known if a node is solid or fluid. Also, suppose there are $nNode$ of fluid nodes in the grid, and a $DnQm$ lattice stencil is used with m discrete velocity associated with each node. There is, then, a total of $m \times nNode$ PDFs to be indexed. Therefore, in the beginning, an integer array of size $m \times nNode$ is allocated to store the index of the node that belongs to each PDF. This is the **PDFn** array, which points to the node index of the PDFs, and is used on line 7 of Algorithm 1 to load the node index. It should be noted, only data in **PDFn** of fluid nodes are stored at any time. Similar to the **PDFv** array, if a solid node is encountered while traversing the imaginary arrows, the node index of the in-hand PDF is again simply skipped and not stored in **PDFn**. As a consequence, and as depicted in Fig. 4.4, the **PDFn** arrays are addressed using the same underlying SoA memory layout. Hence the array **PDFn** also has equivalent N_i structures of the same length as the F_i structures of **PDFv**, as shown in Fig. 4.4.

For the N_0 structure in **PDFn**, nine node indexes are stored exactly as in Fig. 4.5a, *i.e.* $N_0 = [0, 1, 2, 3, 4, 5, 6, 7, 8]$ in the present example. For the N_1 structure in **PDFn**, the imaginary arrows A0, A1, and A2 are again followed, where $N_1 = [0, 3, 6, 1, 4, 7, 2, 5, 8]$. For the N_2 structure in **PDFn**, the imaginary arrows B0, B1, ..., and B4 are again followed, and

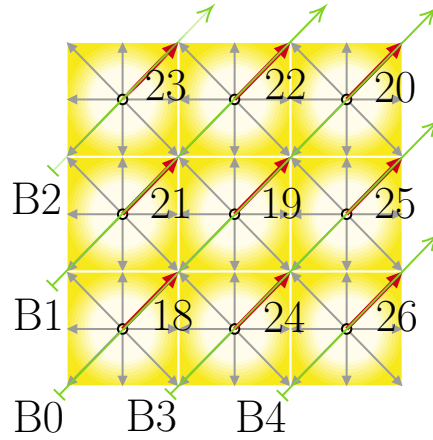
$N_2 = [0, 4, 8, 3, 7, 6, 1, 5, 2]$. Then the rest of the N_i structures are constructed, always following the same imaginary arrows chosen for the F_i structures. This completes the explanation for the array **PDFn** and the explanation of PDFs indexing.



(a) A lexicographical indexing method for indexing f_0 PDFs (same as for indexing nodes in Fig. 4.3).



(b) Indexing f_1 PDFs.



(c) Indexing f_2 PDFs.

Figure 4.5 PDF indexing for a (3×3) grid.

4.4.3 The main loops

At the implementation level, just one loop is required to traverse all PDFs to perform the streaming, collision, and macroscopic steps. However, to decrease the loading number of discrete velocities, that loop is broken down into two loops, as in Algorithm 1. The outer loop ($iVel=m-1$ to 0) is a loop over the number of discrete velocities and the inner loop ($iPDF=lastPDF$ to $firstPDF$) is over all the PDFs with same discrete velocity. For example, if $iVel=3$ the PDFs of f_3 category will be traversed for computation. Hence, in Algorithm 1, $firstPDF$ is always the first cell index of each structure containing PDFs, i.e $F_0, F_1, \dots, F_m$ in Fig. 4.4. Similarly, $lastPDF$ is the last cell index of each structure, which is calculated according to Eqs. 4.14 and 4.15:

$$firstPDF = iVel \times nNode; \quad (4.14)$$

$$lastPDF = (iVel + 1) \times nNode \quad (4.15)$$

How to access data

As mentioned above, one of the critical aspects of LBM code is how to access the data. In the loop over PDFs, *i.e.* lines 6 to 13 in Algorithm 1, we need to know the index of the node, *i.e.* $iNode$, associated with the PDF as the first step of the calculation. After that, the PDF values, f_i , and also the macroscopic values ρ , u , and v of the $iNode$, are necessary to do the calculation. Note that macroscopic values are stored in separate arrays $\mathbf{R}$, $\mathbf{U}$ and $\mathbf{V}$ which respectively contain density, x - and y -velocity components.

Collision Function

To perform the "Collision Function" at line 10 of Algorithm 1, each PDF needs to be modified by Eq. 4.4, as the loaded PDFs are either the initial (first time step) or the post-streaming PDF data (second and following time step). At the collision step, the macroscopic values of a node associated with the PDF are also needed. Hence, it is necessary to store the index of the node belonging to each PDF. This is done in the pre-processing step with the construction of the array $\mathbf{PDFn}$. Accordingly, at this step, the loaded data is used to carry out the collision step by applying Eqs. 4.4 and 5.5 thus obtaining the post-collision PDFs.

Streaming Function

As mentioned above, based on the approach used for indexing the PDFs, the streaming step is carried out by a simple shift operation. It is enough to just pull PDFs from the previous cell of the **PDFv** array. However, this shift operation does not apply to the f_0 PDFs. Rather, in the proposed algorithm, this is accomplished by writing the post-collision PDF into the next PDF cell in the **PDFv** array. In the *Streaming Function* at line 11 of Algorithm 1 it is carried out as in listing 1.

Listing 4.1 Implementation of "Streaming Function" in C++ (line 11 of Algorithm 1). In this function, f_i represents the in-hand post-collision PDF f^* .

```

1  if (iVel!=0) {
2      PDFv[iPDF+1] = fi;
3  }
4  else if (iVel==0) {
5      PDFv[iPDF] = fi;
6  }

```

As in the streaming step it is needed to pull a PDF from previous one, the mentioned loop with $iPDF$ counter should be executed in a reverse way from the *lastPDF* to the *firstPDF*, as done in line 6 of Algorithm 1.

Macroscopic values calculation

The macroscopic values are a kind of summation on all the PDFs belonging to any nodes as described by Eqs. 4.9 and 4.10. This macroscopic calculation requires two steps: first, the summation is obtained by traversing all the PDFs, then the macroscopic values are calculated using the summations.

Some temporary arrays, *i.e.* **sR**, **sU**, **sV**, are reserved to do the summations. As the post streaming PDF values are available after invoking "Streaming Function", these summations are performed in the "Macroscopic Function" at line 12 of Algorithm 1. Note that, as post streaming PDF f_i now belongs to the next cell of array **PDFv**, it is necessary to load data from $iNode$ of the next **PDFv** cell. All that is required in the "Macroscopic Function" is described in Listing 2.

Listing 4.2 Implementation of "Macroscopic Function" in C++ (line 12 of Algorithm 1). The "if" statement is because the streaming operation does not apply to f_0 PDFs. In this function, f_i represents the post-streaming PDF. The variables C_x and C_y are defined according to Eq. 5.2.

```

1  if (iVel!=0) iNode = PDFn[iPDF+1];
2  sR[iNode] += fi;
3  sU[iNode] += fi * Cx[iVel];
4  sV[iNode] += fi * Cy[iVel];

```

After calculating the summation of Listing 2, it is possible to calculate macroscopic values in a separate function at line 16 of Algorithm 1, which follows after traversing the PDFs. It is possible to calculate macroscopic values in a separate loop over the entire nodes as in Listing 3.

Listing 4.3 Implementation of "Calculate Macroscopic Values" in C++ (line 16 of Algorithm 1). In this function, `nNode` represents the number of entire fluid and boundary nodes.

```

1  for (int i=0; i<nNode; i++){
2      R[i] = sR[i];
3      U[i] = sU[i]/sR[i];
4      V[i] = sV[i]/sR[i];
5  }
```

4.4.4 How to apply boundary condition

After applying the streaming step, as illustrated in Fig. 4.6, the PDFs pointing to the domain but that belong to the boundary nodes are computed incorrectly. These unknown PDFs are specified in *Apply boundary condition* step at line 15 of Algorithm 1.

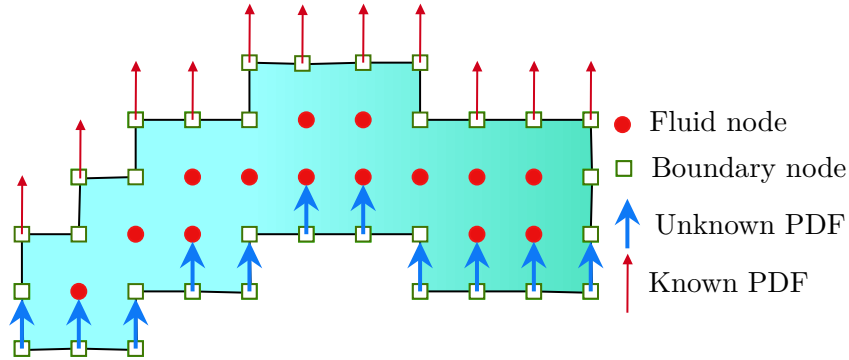


Figure 4.6 The unknown PDFs, which should be specified in the process of applying boundary conditions. Just one component of PDFs, here f_1 , is depicted for clarity purposes.

The summation from the *Macroscopic Function* at line 12 of Algorithm 1 is also incorrect for the boundary nodes, and should be recomputed. Before exiting the "Apply boundary condition" function, the sR , sU , and sV values of boundary nodes are therefore replaced by recomputing the density and velocities using the new PDF values imposed by the boundary condition.

4.5 Results and discussion

To evaluate the correctness of the implemented LBM solver described in this paper, a series of test cases for verification and validation were carried out, as described in section 4.5.1.

Several academic benchmarks were studied, including Poiseuille, Cavity, and Cylinder flow. In section 4.5.2, a second series of test cases and investigations to study the memory usage and performance of the proposed code are discussed. First, a theoretical estimation of the memory consumption for different void fractions is provided. Then, the performance of the implemented code is investigated by simulating flow around two complex geometries, and the memory usage and computational performance compared to the Palabos code.

4.5.1 Verification and validation

To verify the code, a Poiseuille flow test case was carried out, as there is an analytical solution for this kind of flow. A cavity flow was investigated using a higher Reynolds number for the second test case. Then, a cylinder flow problem was performed for a range of Reynolds numbers to study complex flow.

Poiseuille flow

The main purpose of this test case was to verify the developed code based on the proposed algorithm. In particular, for this test case, an order of accuracy evaluation and grid convergence study was performed.

For this test case, we considered the Poiseuille flow of a fluid in a rectangular channel $2W$ wide and L long, as depicted in Fig. 5.13. The flow is driven by a pressure gradient between the inlet and outlet sides of the channel. Only one component of velocity in x -direction is non-zero. The Zou and He pressure boundary condition [40] was applied on the left inlet and right outlet. For the no-slip boundary condition, a half-way bounce back was applied on the top and bottom sides. The analytical solution, u_a , for the horizontal velocity profile of the Poiseuille flow is given as [40]:

$$u_a(y) = \frac{W^2 \Delta P}{2\rho\nu L} \left(1 - \frac{y^2}{W^2}\right) \quad (4.16)$$

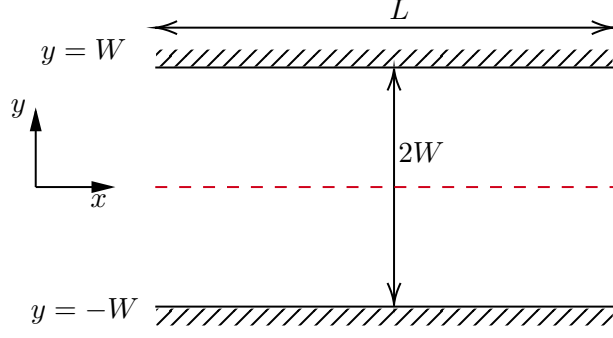


Figure 4.7 The geometry of Poiseuille flow. Flow is driven by a pressure gradient from the left inlet to the right outlet.

To define the pressure drop ΔP between the inlet and outlet, let us consider the following definition for the Reynolds number Re of this Poiseuille flow:

$$Re = \frac{UW}{\nu} \quad (4.17)$$

where the maximum velocity U occurs at the centerline of the channel. So, by considering Eq. 4.17 this maximum velocity is:

$$U = \frac{\Delta P W^2}{2\rho\nu L} \quad (4.18)$$

By combining Eq. 4.17 and 4.18 the pressure drop in the function of the Reynolds number and other parameters is calculated as:

$$\Delta P = \frac{2\rho\nu^2 L}{W^3} Re \quad (4.19)$$

The density at the inlet/outlet are given by $\rho_{in} = \bar{\rho} + \frac{1}{2} \frac{\Delta P}{c_s^2}$ and $\rho_{out} = \bar{\rho} - \frac{1}{2} \frac{\Delta P}{c_s^2}$. So that the average density in the channel is $\bar{\rho} = 1$. The initial PDFs are the equilibrium PDFs computed from Eq. 5.5 with $\rho = \bar{\rho} = 1$ and $\vec{u} = 0$. To do a grid study a set of grids ranging from (20×20) to (320×320) nodes, with $\nu = 1/6$, and Re number of 5 and 50 has been used in the simulations presented below.

In this test case, the velocity profile in x - direction is compared with the analytical solution. The agreement of numerical and the analytical solution for various grids are shown in Fig. 4.8. Both the numerical and analytical velocity is normalized by the maximum analytical velocity at the centerline. In particular, the results for Reynolds number five are illustrated in Fig. 4.8a, which has good agreement with the analytical solution even for a coarse grid.

However, for Reynolds number 50, a fine grid is required to have an acceptable result, as shown in Fig. 4.8b. It should be noted that it is possible to obtain very accurate results with only a few points when periodic boundary condition is combined with the bounce-back and the Two-Relaxation-Time (TRT) collision model [9]. But, to keep the test case as simple as possible, instead of using periodic boundary condition, which is a non-local boundary condition, Zou and He pressure boundary condition was used.

In any case, the results improved as grid size is increased. It is expected to have second order accuracy when a half-way bounce back boundary condition is used in this test case [9]. The L^2 norm, $||.||_{L^2}$, normalized by the centerline velocity U was used to determine the error and the degree of accuracy of the model, according to the error computed as:

$$\text{L2error} \equiv \frac{1}{U} ||u - u_a||_{L^2} = \frac{1}{U} \sqrt{\frac{1}{iMax \times jMax} \sum_{i=0}^{iMax-1} \sum_{j=0}^{jMax-1} \left(u(i, j) - u_a(i, j) \right)^2} \quad (4.20)$$

Fig. 4.9 illustrates the error evolution as a function of the domain size ($iMax \times jMax$) for $Re=5$ and $Re=50$. The slopes of the linear curves are approximately -2, indicating a second-order accuracy of the nominal scheme, as expected [41].

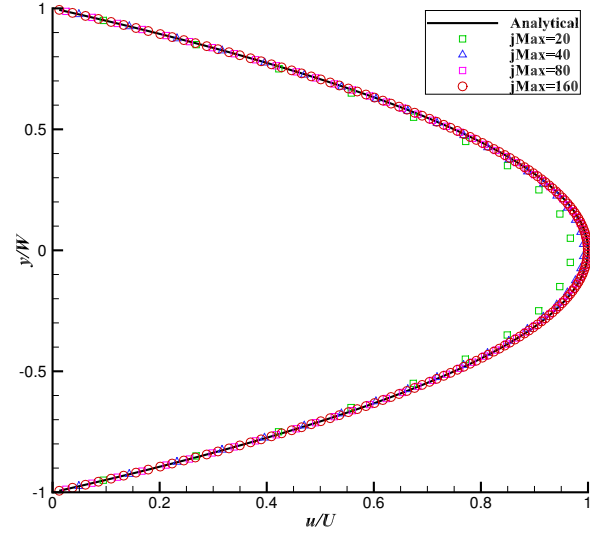
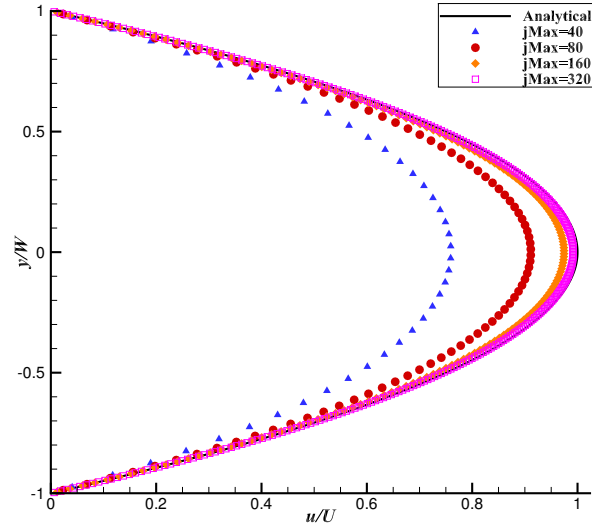
(a) $Re = 5$.(b) $Re = 50$.

Figure 4.8 Comparison of analytical and numerical velocity in x -direction for the Poiseuille flow test case, $Re=5$ and 50 , and grid size of (20×20) , (40×40) , (80×80) , and (160×160) .

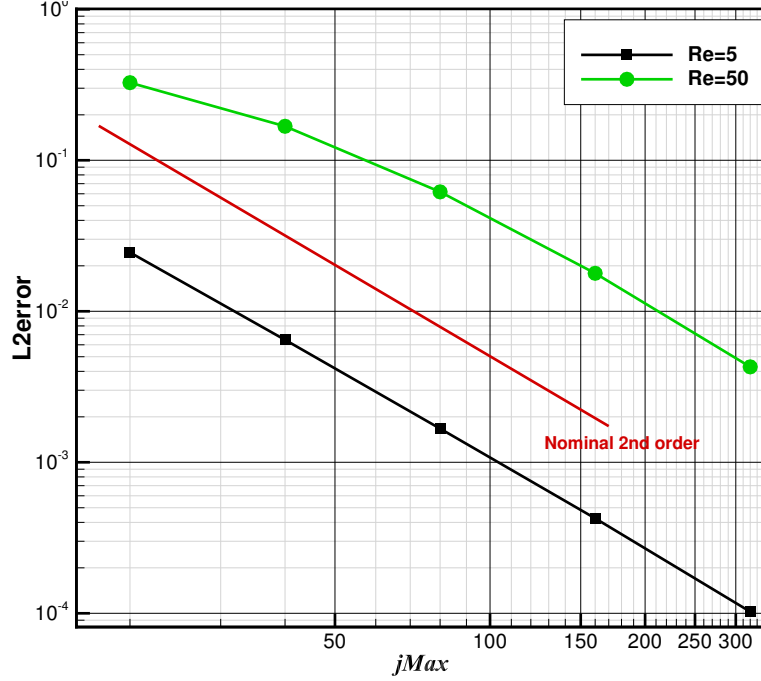


Figure 4.9 Logarithmic error for Poiseuille flow test case.

Although more investigations are done in following sections, it seems at this point the proposed algorithm and its implementation are working well. Indeed, the expected second order of accuracy is achieved. In addition, the boundary conditions used in this test case, *i.e.* Zou and He pressure and half way bounce back, are well implemented and coupled to the LBM algorithm. The full-way bounce back boundary condition was tested. Compared to the half-way bounceback, the results in this case are the same to machine accuracy since the flow has a steady state solution.

Cavity flow

The lid-driven cavity is one of the most tested and studied cases in computational fluid dynamics. As illustrated in Fig. 5.17, the cavity is square, and the flow is driven by the top lid, which moves with uniform speed U . For this type of geometry, the Reynolds number is based on the length of the moving lid L , its speed U , and the fluid's kinematic viscosity ν according to the Eq. 5.22:

$$\text{Re} = \frac{UL}{\nu} \quad (4.21)$$

The Zou and He velocity boundary conditions [40] were applied on each side of the geometry. For the top side, a uniform velocity of U was applied, and for the remaining sides, a zero velocity for both x - and y - direction.

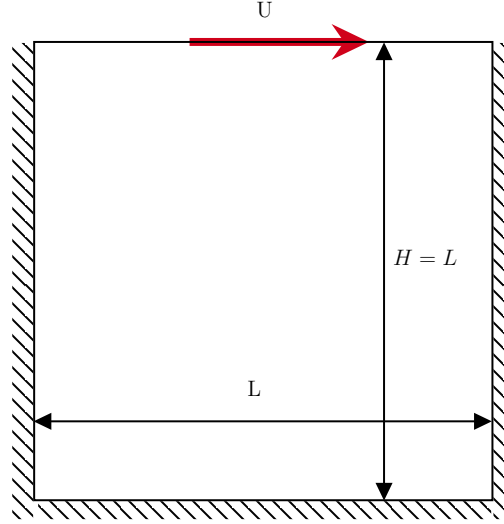


Figure 4.10 Geometry of the lid driven cavity.

In this flow problem, the presence of a central vortex and the recirculation areas at the bottom corners were considered. The Reynolds number of the flow affects the size and location of these vortices considerably. All the numerical data obtained from the proposed code, including velocity profiles and vortex locations, were compared to the results of Ghia et al. [1]. The quantities used for this problem are the velocity in both x - and y - direction at the midline of the cavity and also the location of the primary and bottom corner vortices. In this study, $Re=100$ and 400 , and grid sizes of 129×129 and 257×257 were used respectively for the validation of the code. The goal of this test case is to compare the results of the proposed algorithm to the numerical results from Gia [1]. So, in this study, no grid sensitivity analysis is done.

In the results illustrated in Fig. 4.11, the vortex location is simulated correctly for $Re=100$ and 400 . For $Re = 100$, the lid velocity creates a vortex in the upper region of the cavity. When the Reynolds number increases, that vortex moves to a more centered position, as depicted by the streamlines in Figs. 4.11a and 4.11b. Also, locations of the vortices from Ref. [1] and the proposed work are provided in Table 4.2.

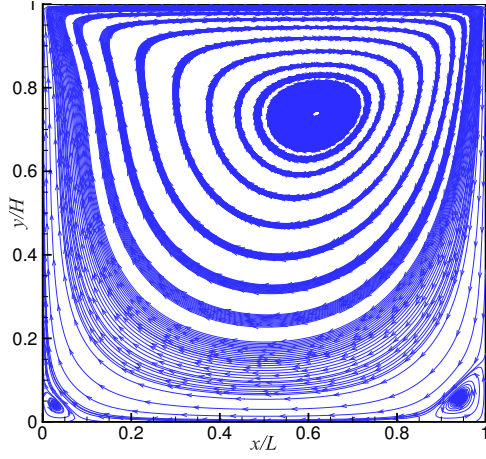
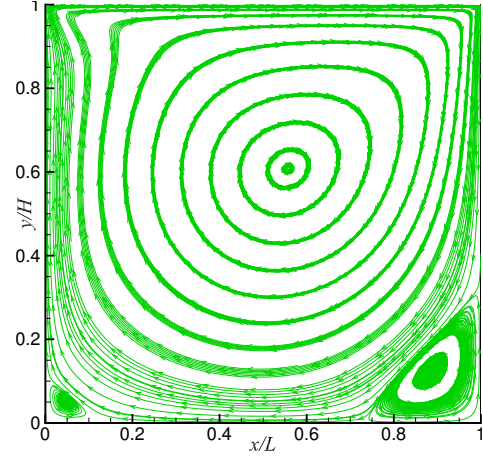
(a) Re=100 and grid size of 129×129 .(b) Re=400 and grid size of 257×257 .

Figure 4.11 Streamlines for square cavity flow at Re=100 and 400.

Table 4.2 Primary and bottom corner's vortex location compared with results from Ref. [1] for different Reynolds numbers and grid sizes.

	Re	Grid size	Primary Vortex (x,y)	Lower Left (x,y)	Lower Right (x,y)
Ghia [1]	100	(129×129)	(0.6172,0.7344)	(0.0313,0.0391)	(0.9453,0.0625)
Proposed algorithm	100	(129×129)	(0.6187,0.7401)	(0.0290,0.0394)	(0.9406,0.0556)
Ghia [1]	400	(257×257)	(0.5547,0.6055)	(0.0508,0.0469)	(0.8906,0.125)
Proposed algorithm	400	(257×257)	(0.5593,0.6082)	(0.0466,0.0476)	(0.8841,0.1224)

In Fig. 4.12, the velocity profiles obtained via the LBM simulations are compared to the results of Ref. [1] for Reynolds numbers of 100 and 400. In particular, the velocity profiles presented are for the vertical and horizontal centerlines of the cavity. On the vertical line, the velocity profile in x - direction is provided, while the velocity in y - direction is reported on the horizontal line. There is an acceptable agreement with the velocity profile reported in Ref. [131] and the result obtained from the proposed algorithm.

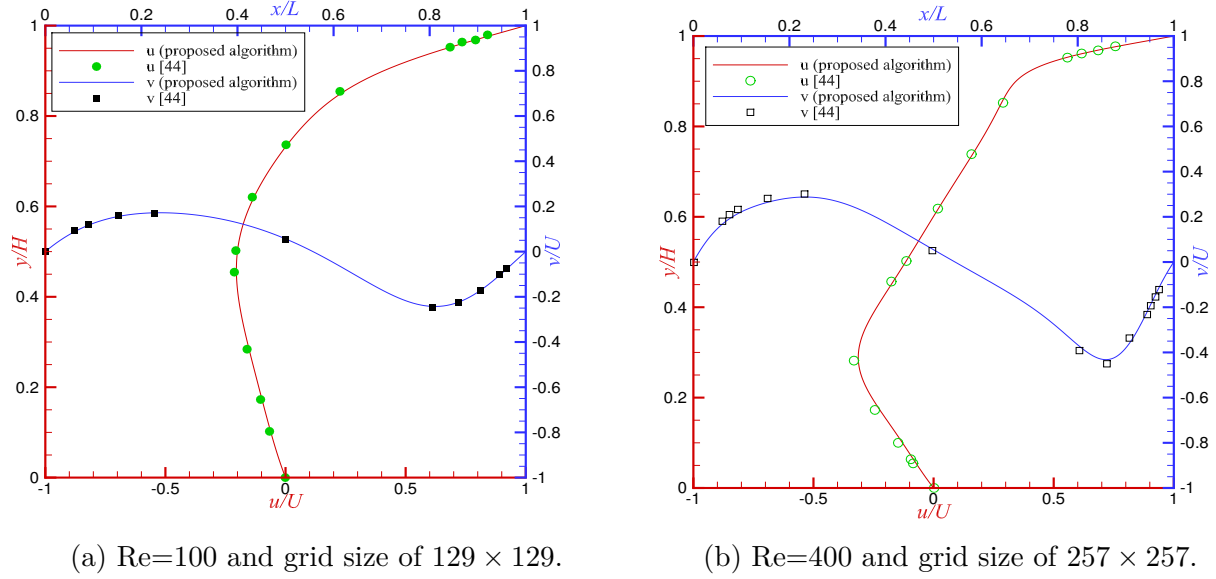


Figure 4.12 Velocity profile of at the center line of the cavity for different Reynolds numbers and grid sizes.

In this cavity flow test case, a higher Reynolds number was tested using a more complex two dimensional flow compared to the Poiseuille flow test case. The cavity flow test case results have improved our confidence in the correctness of the proposed algorithm.

Cylinder flow

Another interesting test case is to put an obstacle inside a domain to show the flow field with a more complicated flow behavior. In this test case, a cylindrical obstacle is placed inside a rectangular domain, as illustrated in Fig. 4.13. Based on the investigation in Ref. [132], the schematic diagram of this problem is presented in Fig. 5.26. The circular cross-section of the cylinder of diameter D is mounted centrally inside a planar channel of height H . The channel length, set as $L = 50D$, was chosen to reduce the influence of the outflow boundary conditions on accuracy. The upstream length, $l = L/4$ and $H = 0.16L$, was set up as in Ref. [132]. To produce a flow around the cylinder a uniform velocity profile was applied. The Zou and He velocity [40] was used as the boundary condition at the left inlet of the domain, the top and bottom side boundary conditions were set symmetrically, and half-way bounce back applied to the cylinder. For the right outlet of the domain, an "open" boundary condition was applied [41].

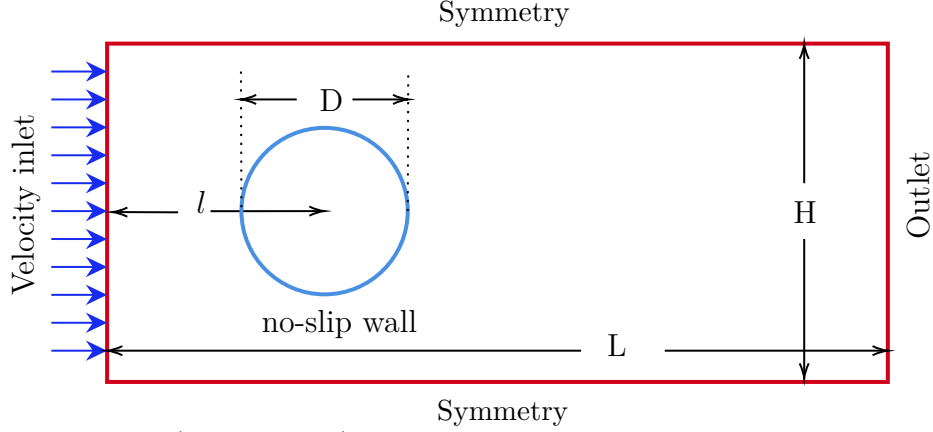


Figure 4.13 Geometry (not to scale) and boundary condition of the cylinder flow test case.

The Reynolds number is based on the cylinder's diameter D , the inflow velocity U_{in} , and the fluid's kinematic viscosity ν as in Eq. 5.25:

$$\text{Re} = \frac{U_{in} D}{\nu} \quad (4.22)$$

To perform the simulations for the steady flow regime, the Reynolds numbers chosen were $\text{Re}=10, 20, 30$, and 40 . As can be seen in Fig. 4.14a for $\text{Re}=10$, the flow is separated, and there are two vertices behind the cylinder. For $\text{Re}=40$, it is clear there is an increase in the size of these vertices, as depicted in Fig. 4.14b. For a quantitative comparison, the location of vortices behind the cylinder was used, as presented in Table 3. These results show an acceptable agreement to the results published in Ref. [132].

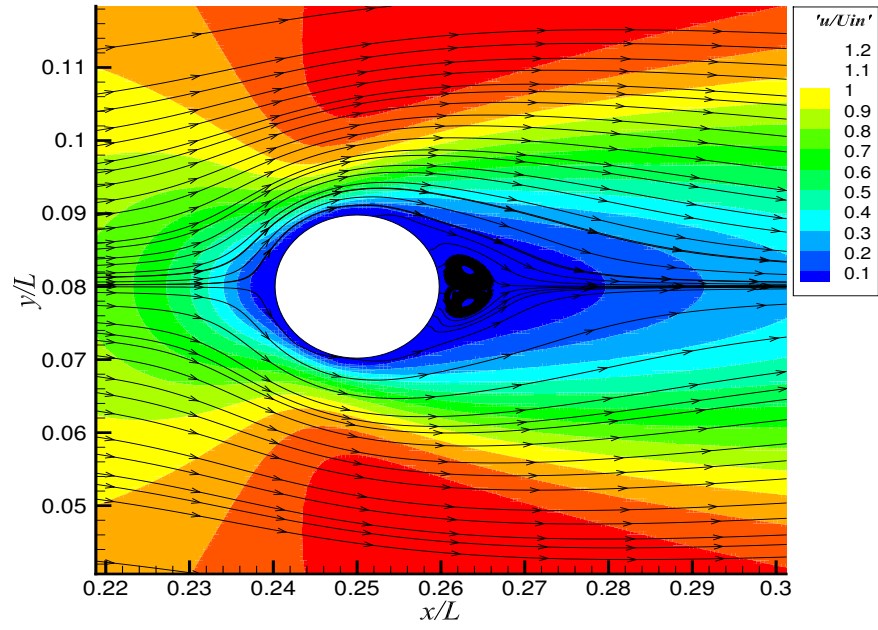
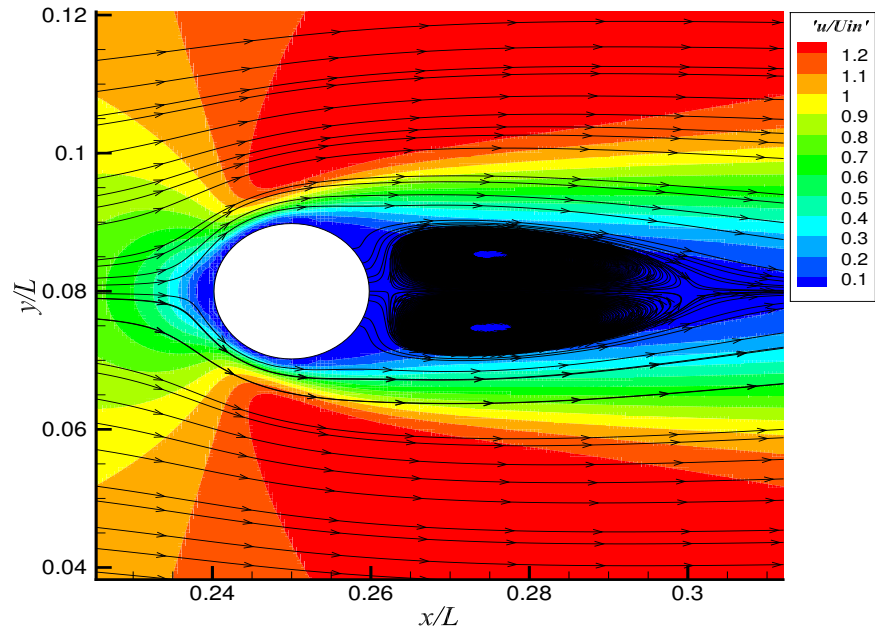
(a) $Re=10$.(b) $Re=40$.

Figure 4.14 Streamline patterns for steady flows past a circular cylinder for different Reynolds numbers and a grid size of 1000×160 .

Table 4.3 Comparison of the vortex location in the wake of the cylinder for different Reynolds numbers.

Re	Grid size	Proposed algorithm	Ref. [132]
10	1000×160	0.263	0.264
20	1000×160	0.266	0.266
30	1000×160	0.274	0.274
40	1000×160	0.275	0.276

In this study, the complex flow around a cylinder for a range of Reynolds numbers was performed. In addition, by applying half-way bounce back on the cylinder, the implementation of this boundary condition for curved boundaries was tested using a staircase scheme. In this test case, the eddies locations are compared to the values reported in Ref. [132] for a quantitative evaluation of the results obtained from the proposed algorithm. So, in this study, no grid analysis is done.

4.5.2 Memory usage and performance analysis

After code verification and validation, the next step was to assess the performance and memory usage of the implemented code. To analyze the memory consumption of the proposed algorithm, a theoretical estimation of memory usage of the implemented code for different void fractions was carried out. Subsequently, a porous media like geometry was used to test the code, in terms of memory usage and performance, for different void fractions (VF). Using this test case, the performance of the algorithm was compared with Palabos [129]. For this set of cylinder geometry, it is impossible to decrease the void fraction values to anything less than 0.15. Hence, another irregular simple test geometry was studied that contains a set of channels.

All of the simulations were executed with double-precision floating-point numbers. The computer used for the simulations was equipped with an Intel(R) Core(TM)i9-9900 @3.10GHz processor and 64GB of main memory. Compiler options "Maximum Optimization (Favor Speed) (/O2)" were used in MS Visual Studio.

Performance of the implementation was measured in Millions of Distribution (f_i) Update per second ($MDUPS$) according to Eq. 6.1:

$$MDUPS = \frac{nNode \times m \times nTimeStep}{t_{tot} \times 10^6} \quad (4.23)$$

where m is the number of PDF (velocity components) belonging to each node, $nTimeStep$ is the total number of simulation time steps, and t_{tot} the time in seconds required to perform the simulation on the computing device. In all the simulations for performance measuring, the quantity $nTimeStep$ is chosen large enough to avoid any fluctuation in performance measuring. So, the performance metric in this study is the mean of repetition.

Theoretical memory usage

To investigate the memory usage, the first step was to study the required arrays to implement the code. As mentioned above, a SoA array is needed to store the PDF values (array **PDFv**). As described previously, in the collision step, the index of nodes associated with each PDF (array **PDFn**) needs to be stored. In addition, three arrays are required to store all the macroscopic values for the 2D cases (**R**, **U**, **V**). Finally, based on the macroscopic calculation method explanation, three more temporary arrays, *i.e.* **sR**, **sU**, and **sV**, are needed to calculate the summation required for macroscopic value calculations. All these arrays are illustrated in Fig. 4.15.

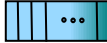
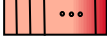
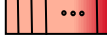
F_0	F_1	F_2	$\dots$	F_m	PDFv
N_0	N_1	N_2	$\dots$	N_m	PDFn
	$\mathbf{R}$		$\mathbf{sR}$		
	$\mathbf{U}$		$\mathbf{sU}$		
	$\mathbf{V}$		$\mathbf{sV}$		

Figure 4.15 Required arrays to perform a simulation. Size of **PDFv** and **PDFn** are $m \times nNode$ and size of **R**, **U**, **V**, **sR**, **sU**, and **sV** are $nNode$.

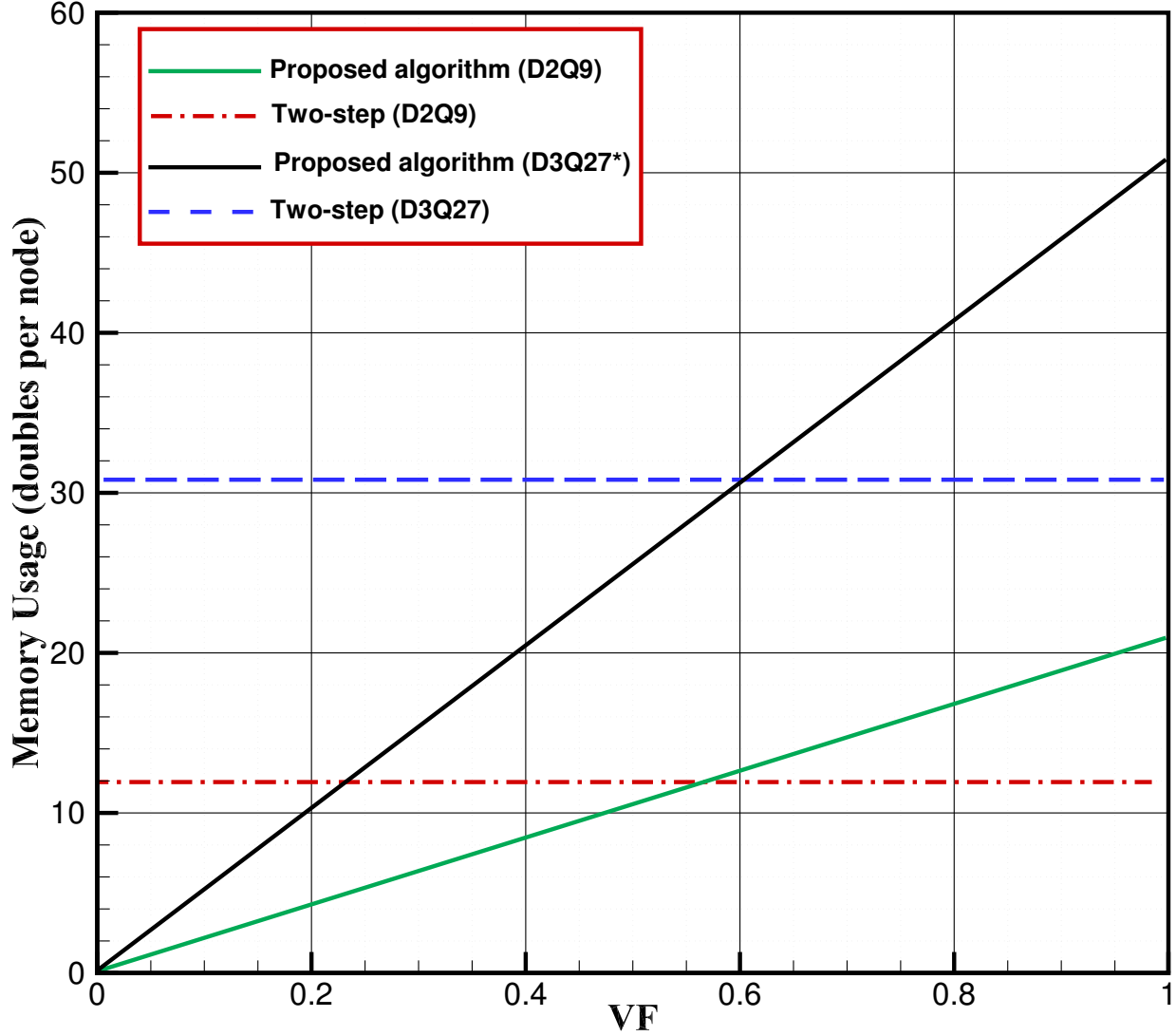
It is now possible to compare an optimum implementation of the two-lattice and two-step LBM algorithm for the proposed work, as depicted in Table 4.4. As seen from this table, if the grid size is $nNode$ and a $D2Q9$ lattice stencil is used, to perform a simulation with the two-step and the two-lattice algorithm, the total size of the arrays vary from $12 \times nNode$ and $21 \times nNode$ respectively. The algorithm uses approximately the same amount of memory resources, *i.e.* $20 \times nNode$, as the two-step algorithm, using a fusing approach, which improves code performance. It should be noted, in the 3D cases, the amount of memory used for the proposed algorithm is less than the two-lattice algorithm.

Table 4.4 Required arrays for the proposed, two-step, and two-lattice algorithm.

Array name	Proposed algorithm	Two-step	Two-lattice
PDFn	$m \times nNode(int)$	—	—
PDFv	$m \times nNode$	$m \times nNode$	$2 \times m \times nNode$
R, U, V	$(D+1) \times nNode$	$(D+1) \times nNode$	$(D+1) \times nNode$
sR, sU, sV	$(D+1) \times nNode$	—	—
Overall	$2(D+1+3m/4) \times nNode$	$(D+1+m) \times nNode$	$(D+1+2 \times m) \times nNode$
<i>D2Q9</i>	<i>20 nNode</i>	<i>12 nNode</i>	<i>21 nNode</i>
<i>D3Q27*</i>	<i>50 nNode</i>	<i>31 nNode</i>	<i>58 nNode</i>

* *D3Q27* stencil is a projected information, but not simulated in this work.

The comparison in Table 4 is for a dense grid. For irregular and low void fraction geometries, the algorithm has much better efficiency in terms of memory usage, since here, just the fluid nodes are stored. Based on the diagram in Fig. 4.16 and considering Table 4.4, if all kinds of nodes (solid and fluid) are stored in the data structure of the LBM code, the proposed algorithm consumes more memory compared to the two-step algorithm. However, the intersection of the curves for both the *D2Q9* and the *D3Q27* lattice stencils are close to a void fraction (VF) of 0.6. This means that for $VF < 0.6$ the proposed algorithm outperforms both the two-step algorithm and two-lattice in terms of memory usage.



* $D3Q27$ stencil is a projected information, but not simulated in this work.

Figure 4.16 Comparison of memory consumption for the proposed and two-step algorithm for irregular geometries as function of the void fraction (VF). As an example, let's assume that the calculation is performed using double precision, i.e., a memory consumption of 8 bytes for each value. Thus, $12 \cdot n_{\text{Node}} \cdot 8$ bytes of memory are used to run a simulation using the two-step D2Q9 algorithm. The number of "doubles per node" used for this algorithm is therefore 12.

Set of cylinders flow

In this test case, a symmetry boundary condition was applied on both the top and bottom and a no-slip wall used on the cylinders. For the left and right sides, the Zou and He boundary condition was used to apply a pressure gradient. As a first step, the simulation

was carried out with a small $VF=0.31$. The velocity field and each geometry of this test case are illustrated in Fig. 6.10b. For larger VF it is possible to decrease the radius of the circles.

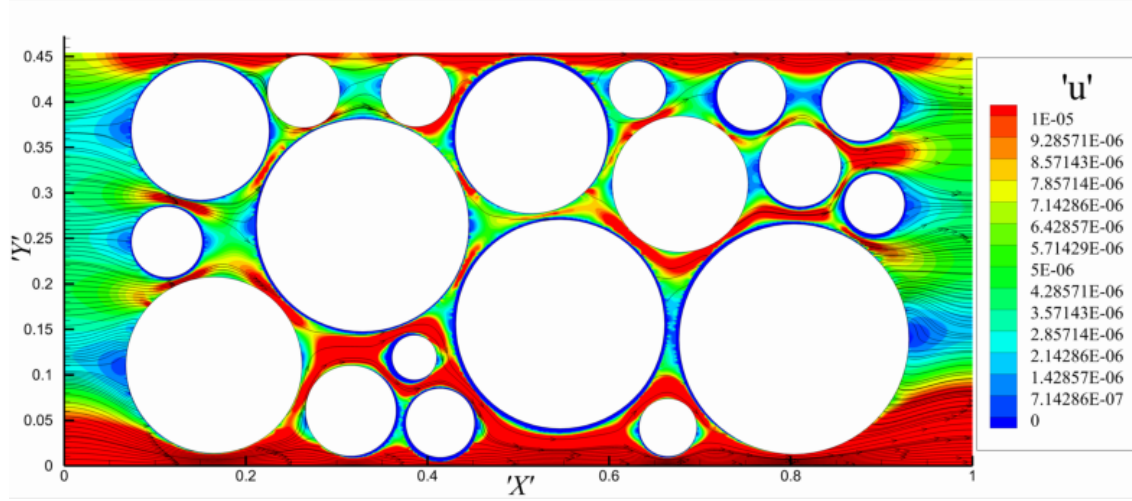


Figure 4.17 Set of cylinders geometry to test the memory consumption and performance.

In addition to the study of memory usage for an ideal implementation, a comparison between the proposed code and the open-source Palabos code was performed [129]. Palabos code has sparse block management and can be configured to use only local cuboids near fluid nodes, which eliminates the solid nodes data from the geometry. Although this approach has many advantages, it may not be possible to reduce memory usage in some cases: For instance, in chaotic and more random geometries, where the local cuboids are large relative to the intricacy of the fluid region, e. g. porous media such as rock with very low void fraction. In addition, the size of the local cuboids is chosen manually by the user, which affects performance; optimal performance is therefore achieved through a trial-and-error process. Moreover, the size of the local cuboids is constant and may not be suitable for geometries with different spatial scales. The proposed algorithm has built-in a wrapping of the fluid region completely, and therefore automatically does not store any solid nodes. This means both the Palabos and the proposed algorithm are advantageous depending on the intended applications.

Following the above discussion, comparison of this work with Palabos is not intended to privilege the proposed code in any respect but rather to demonstrate a baseline, and assist the reader to understand where the memory usage and performance of the proposed algorithm resides. It is therefore worth mentioning that the execution of the Palabos code was performed in fusing algorithm mode without sparse block management, which means that Palabos

efficiently uses memory resources as a two-step algorithm.

As is seen in Fig. 4.18, the proposed algorithm uses about 1.5 times more memory compared to Palabos for $VF=1$. However, as VF decreases, the amount of required memory to simulate a fixed resolution decreases with the proposed code while it is constant for Palabos. In the following figures, the equivalent regular grid size means that all solid and fluid nodes are counted.

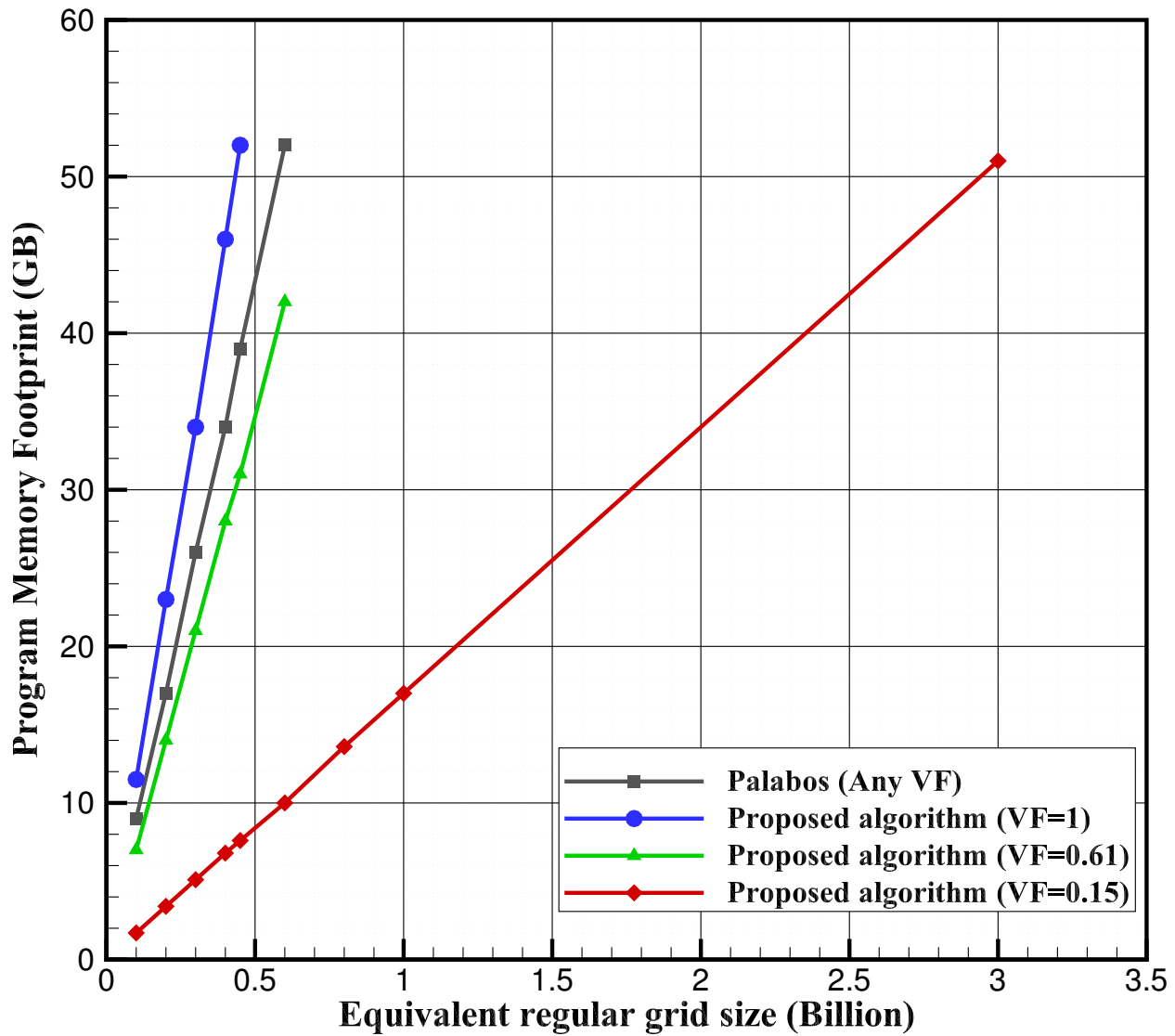


Figure 4.18 Comparison between the proposed and Palabos code for the required memory to simulate the "set of cylinders flow test case" with different VF .

A further interesting result of the proposed model is its performance simulating large-scale systems. As seen in Fig. 4.19 Palabos performance is better than the proposed code for $VF=1$

but, as the VF decreases, the performance of this code is higher. It should be mentioned in Palabos results presented in Fig. 4.19 the LBM computation is done for all the solid and fluid nodes, but in the proposed algorithm only the fluid nodes participate in the LBM computation. So, for simulating a system with the same grid size, the performance of Palabos is less than the proposed algorithm for sparse geometries. As the L3 cache size of the machine used to perform the simulations was 16 MB, from the point that memory usage exceeds the L3 size, the cache misses on L3 start, and the performance degrades for the proposed algorithm.

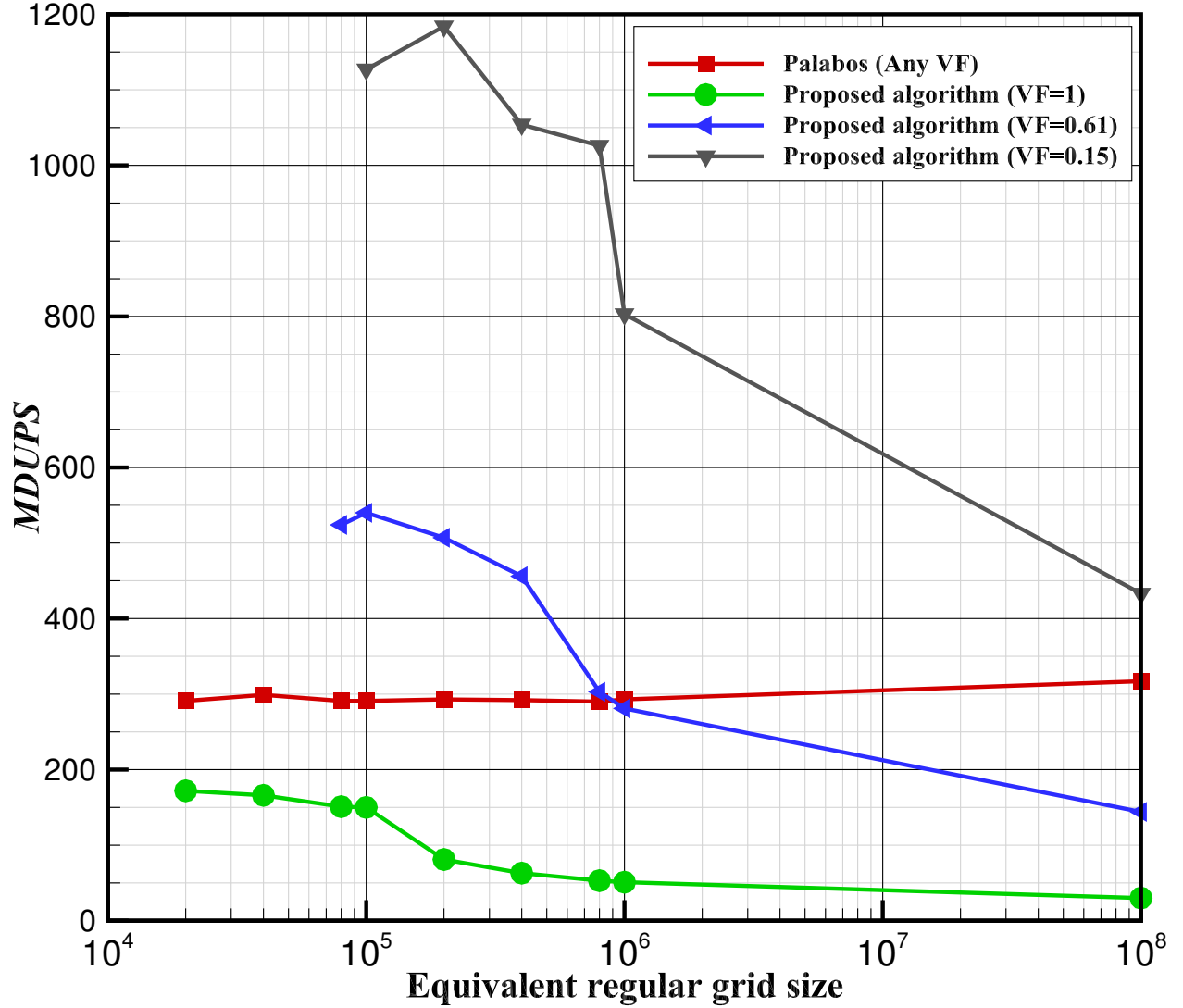


Figure 4.19 Performance comparison between proposed and Palabos code to simulate the "set of cylinder flow test case" for different VF.

Set of channels flow

As depicted in Fig. 4.20, this geometry consists of two horizontal and one vertical channel. For the simulation, a Zou and He pressure boundary condition was applied on the inlet and outlet of the channels, with half way bounce back on the walls as a no-slip boundary condition. To simulate this geometry by a direct addressing scheme, all the nodes need to be stored, including both solid and fluid nodes. However, in the proposed algorithm, only the fluid nodes are stored, which results in efficient memory usage and performance for low VF geometries. The goal for this test case was to study low VF simulations. Hence, five different VF (i.e. 0.05, 0.1, 0.2, 0.4, 0.57) were studied.

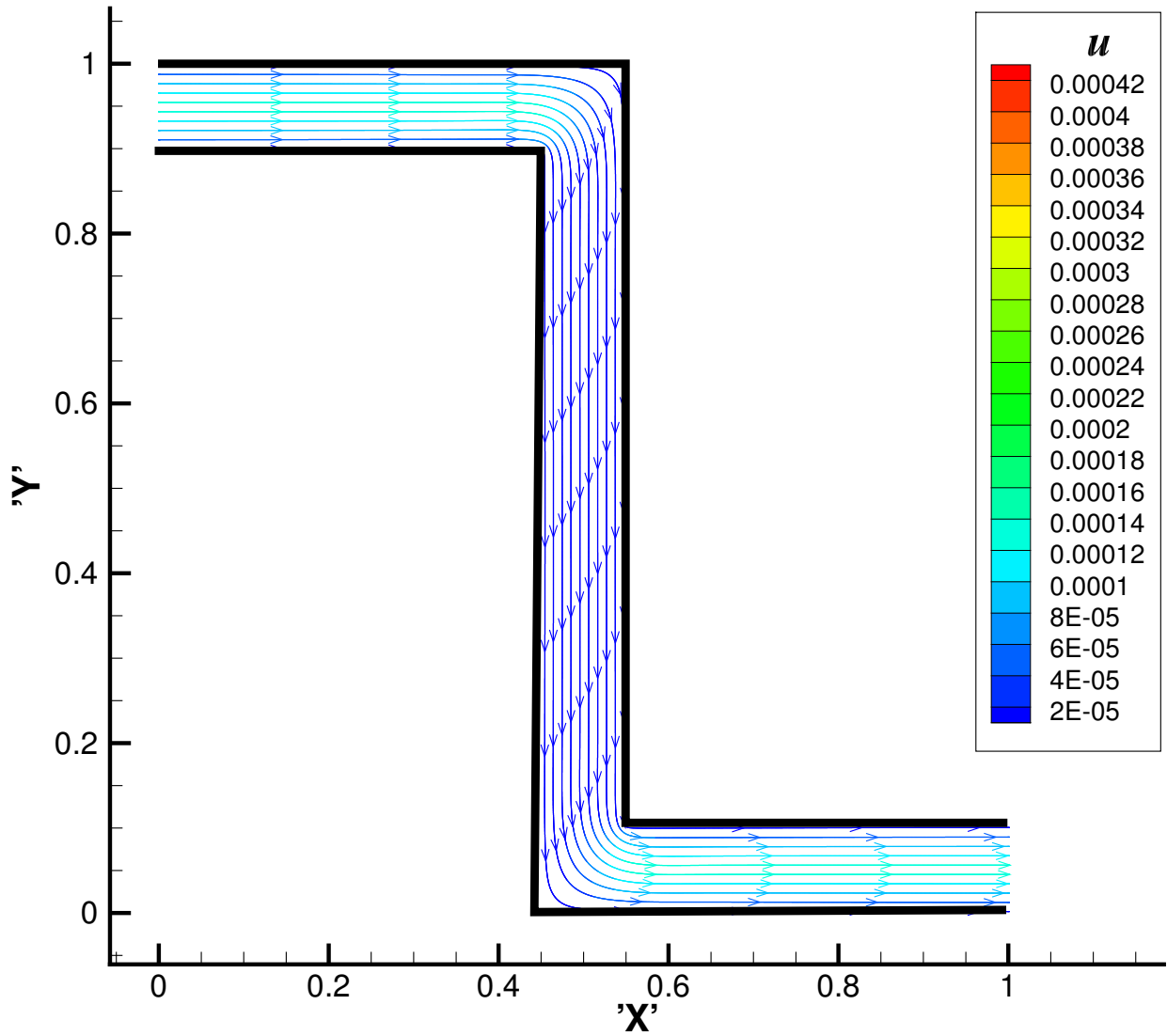


Figure 4.20 Set of channels geometry to study simulations with very low VF. By changing the width of the channels, it is possible to test different values of VF.

To investigate the memory usage of the code for different VFs and grid sizes, a series of simulations were performed, as depicted in Fig. 4.21. As seen from this figure, the memory usage for $VF=0.05$ is less than the other VFs. However, if a direct addressing scheme is used, the memory usage is the same as $VF=1$ for all VFs. For example, to simulate a grid consisting of three billion nodes for $VF=0.05$, the proposed code uses about 35 GB of memory.

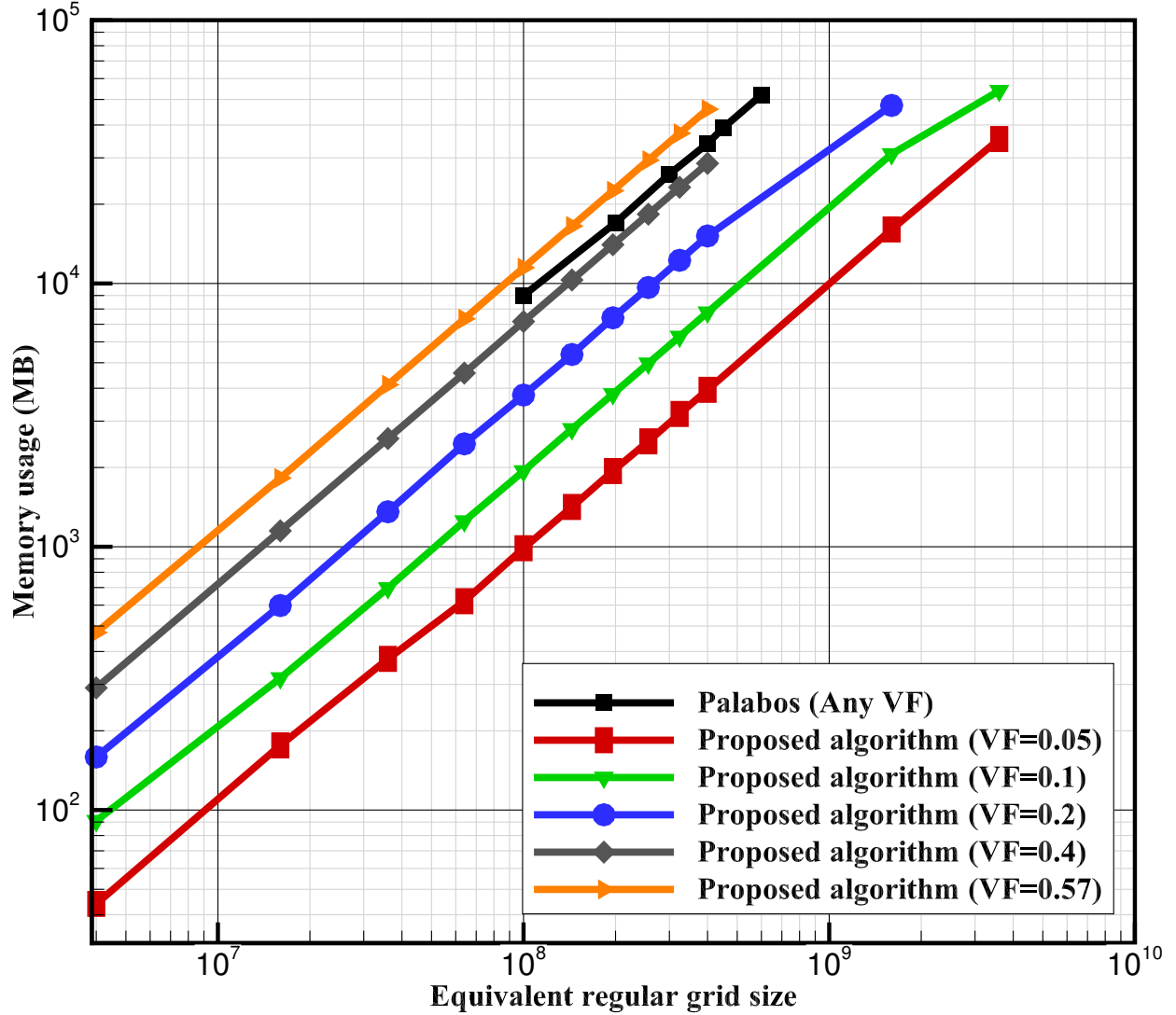


Figure 4.21 Required memory to simulate the set of channels flow test case with different VF.

The performance study was carried out as shown in Fig. 4.22. According to this figure, for an equivalent regular grid size of 100 million and $VF=0.05$, the *MDUPS* is about 1000. This value for the same grid size but $VF=0.57$ is around 200. Based on the trend of the *MDUPS*

as a function of VF, it is concluded that for the same grid size and $VF=1.0$, the *MDUPS* is much less. If we use direct addressing where the VF is always 1.0, we expect to have fewer values for *MDUPS*. Accordingly, we may conclude, for sparse grids, the proposed algorithm will result in higher performance. As can be seen from Fig. 4.22, the performance degrades as the size of the grid increases. One of the reasons for this behavior is the number of cache misses in the simulations. It is one of the drawbacks of the algorithm, however, it does not prevent a gain in performance with VF reduction. This may be viewed as the “cost” of this algorithm in order to reduce memory usage with decreasing VF.

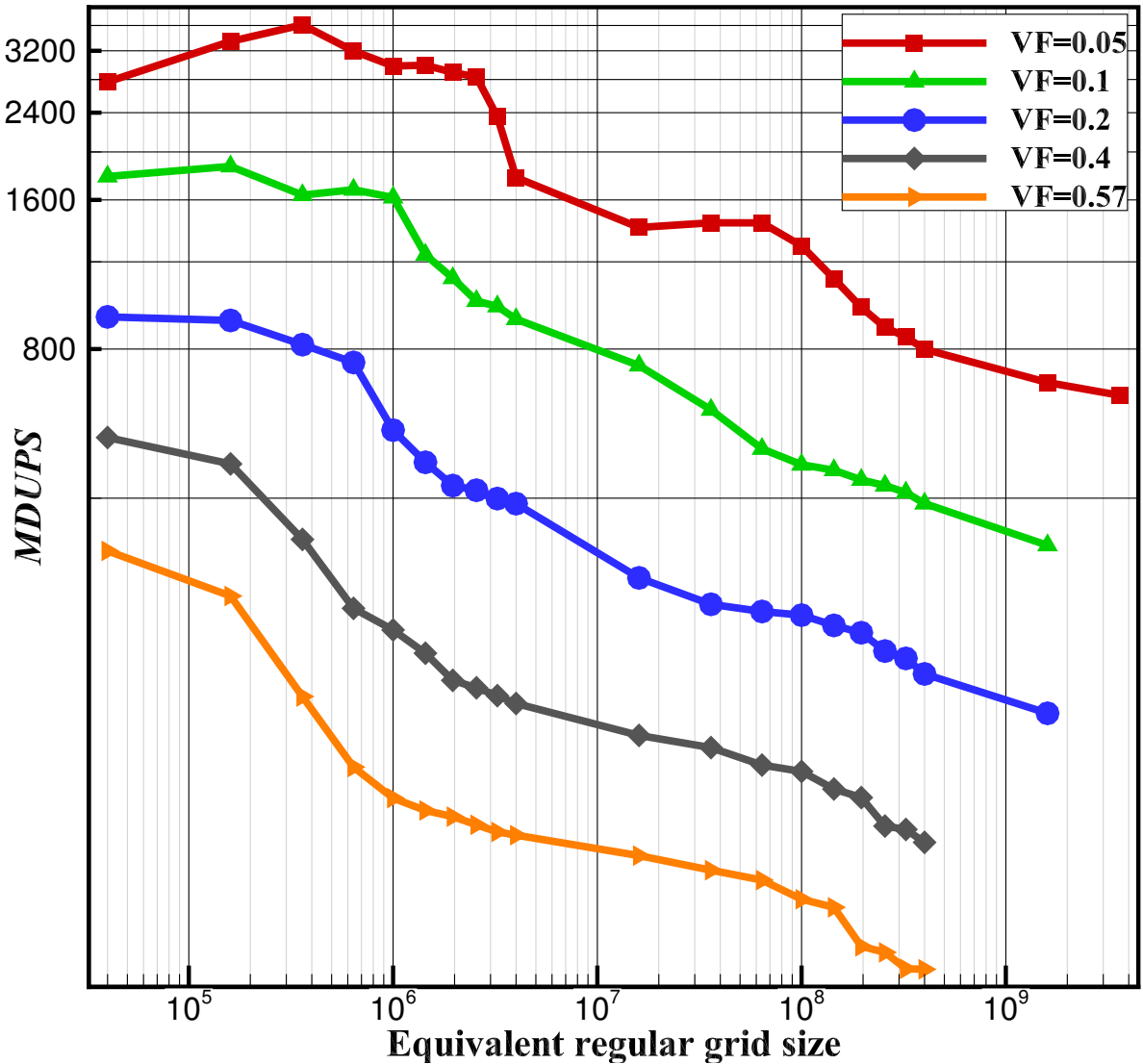


Figure 4.22 Performance of the simulations for the "set of channels flow" test case for different VF.

4.6 Conclusion

What was done: In this paper, a new LBM algorithm is proposed to simulate large-scale systems by avoiding storing the solid node data.

Novelty: In the proposed algorithm instead of modifying the data structure, the LBM algorithm is changed to keep the data structure as simple as possible. To increase the data locality all the PDFs are traversed once to perform both the streaming and collision steps. The strategy of performing the streaming step by a simple shift operation resulted in using the structure of array memory layout. So, the simplicity of the memory layout and streaming function implementation is one of the advantages of this algorithm.

Achievements: This algorithm uses the same amount of memory as the two-lattice but more than the two-step algorithm. Since it uses an indirect addressing scheme, it is nevertheless more efficient in terms of memory usage, particularly for low void fraction geometries. As the data of the solid nodes are not stored in this method, there is no computation for these kinds of nodes, which consequently results in better performance for irregular geometries with low void fractions.

Capabilities and limitations: The expected application of this scheme could be flow simulation in irregular geometries, porous media, blood capillary, pipe systems, and generally low void fraction geometries. As the initial goal of the proposed algorithm is to simulate stationary boundaries, simulation of flows past time-evolving objects, such as reactive porous media that changed shape over time were not investigated in this work.

Future works: Overall, this work paves the way for a new type of simple lattice Boltzmann algorithm with low memory usage for simulating flows with low void fraction geometries. In addition, as the SoA data layout of the proposed algorithm is generally viewed as a highly compatible data layout with GPUs, so, the proposed algorithm could potentially be more efficient for parallelization on GPUs. So, interesting future work could be as follows:

- Parallelization on GPUs and heterogeneous CPU-GPU architectures;
- Investigation to reduce cache misses for further improvement;
- Investigation about optimizing the code in term of memory bandwidth;
- Extension and study of the performance of this algorithm on distributed memory systems; and
- Extension to three-dimensional flow simulations.

4.7 Acknowledgments

We acknowledge the support of the Natural Sciences and Engineering Council of Canada (NSERC), [funding reference number *RGPIN-2020-04512*].

CHAPTER 5 ARTICLE 2: LABCOF: LATTICE BOLTZMANN BOUNDARY CONDITION FRAMEWORK

- M. Namvar, S. Leclaire, LaBCof: Lattice boltzmann boundary condition framework, *Computer Physics Communications*, vol. 285, p. 108647, 27 December 2022. [Online]. DOI = <https://doi.org/10.1016/j.cpc.2022.108647>

This article introduces a new algorithm for implementing boundary conditions of the LBM. In this approach, instead of deriving a BC formulation for each side of the domain, all the edge/corner boundaries are mapped to one edge/corner to use just one formulation for that BC. In addition, the concept of applying the circular shift and flip operation was introduced to map the PDFs of any edge/corner to the left/bottom-left boundary node.

5.1 Abstract

The boundary condition is one of the most challenging aspects of lattice Boltzmann simulations. One of the reasons that make it difficult to implement a new boundary condition is the dependency of formulation on the orientation of the boundary nodes. So, any boundary condition formulation should be derived, implemented, and verified in terms of the boundary orientation. In this study a new framework for the implementation of boundary conditions in the lattice Boltzmann method has been developed. The main goal of this framework is to implement a boundary condition for one side/corner of a domain which can then be applied to any other side/corner. Since implementing a boundary condition depends on the orientation of the boundary nodes, using this framework makes it easier to implement any set of boundary conditions. The framework is developed in C++ using object-oriented programming. It is then verified by implementing some common boundary conditions in LBM. This framework can be used for any kind of data structure of lattice Boltzmann code.

5.2 Introduction

Due to the invention of new computing devices and the growing interest in numerical simulation, attempts to develop efficient numerical methods have been among the most in-demand research fields. In Computational Fluid Dynamics (CFD), the most popular methods are finite difference method and finite volume, both of which have matured over recent years. However, the difficulties in simulating complex fluid flow have led researchers

to consider the lattice Boltzmann method (LBM), which emerged in 1988 [8]. In recent years this method has been aggressively pursued and has found different applications in CFD [24, 102, 103, 131, 133, 134]. Although researchers have developed a variety of boundary conditions (BCs) for Dirichlet and Neumann boundaries, one of the extreme difficulties with LBM is the BC.

Some BCs, such as symmetry, [41] are not so complicated, while others require different implementations as the numerical scheme or physical condition changes. For example, the periodic BC [9, 45] has a modification by Leclaire *et al.* [47] to be globally mass conservative and makes use of a simple proportional-integral-derivative controller to accurately control the pressure difference. Another example is open boundary, [41] which can be first order or second order.

One of the most common Dirichlet BCs was developed by Zou and He [40] to apply pressure or velocity at a straight boundary. Other researchers [42–44]. Latt *et al.* [46] have investigated and analyzed five different velocity Dirichlet BCs including Inamura *et al.* [135], Zou and He [40], regularized method [136], finite-difference velocity gradient [49], and nonlinear finite-difference [49]. They used a $D2Q9$ lattice for implementation and presented the formulations resulting from all the above methods for the upper side boundary.

One of the most useful BCs in the field of CFD is the wall boundary. For LBM, bounce-back is the most common BC. Dubois *et al.* [48] proposed a general bounce-back BC with the Taylor expansion method to increase the accuracy of the original bounce-back. But now there are many studies that model the curve boundaries [50–69] in order to make them more accurate.

After reviewing the above studies, it appears that the implementation of these BCs depends on the lattice topology and orientation of the boundaries to which they should be applied. This issue is explained in detail in Section (5.4). Notably, some papers in the LBM field have attempted to propose more general BCs.

Ginzburg [70] proposed a generic BC for LBM he called the "multi-reflection" method for Dirichlet and Neumann time-dependent BCs. In this method, three points are used to calculate unknown parameters, which means it depends on the boundary orientation. Hecht *et al.* [43] extended the Zou and He velocity BC to $D3Q19$ and also generalized it by introducing extra parameters so as to be implemented free of the boundary orientation. Lee *et al.* [48] then generalized it for multi-speed lattices which he applied to single-speed LBM. Latt *et al.* [71] proposed a general regularized BC for multi-speed LBM, which is applicable for any lattice stencil. They assumed the velocity is predefined on the boundary (Dirichlet BC). In addition, it is possible to apply this BC on both straight edges and corners. How-

ever, their method depends on boundary orientation. Kim *et al.* [46] provided a generalized periodic BC and compared the results of their simulations to the body force approach.

In summary, the process of implementing a BC consists of (1) deriving its formulations, (2) programming the formulations, and (3) debugging and verifying the code. The process is tedious and prone to error, especially when the boundaries have different orientations. In LBM, this is an obstacle to developing new ideas about BCs. The aim of this work is therefore to propose a general framework to avoid multiple implementations of a BC. Overall, this framework will facilitate the implementation and testing of new LBM BCs. The research field will therefore be stimulated in this regards. In this framework, a BC for just one of the boundary types, e.g. edge or corner types is implemented and then the framework can apply it on other boundary with a different orientation. Note that, without further generalization, this framework can only be used in 2D.

The rest of this paper is organized as follows. In Section 5.3, we review the lattice Boltzmann model and the governing equations. In Section 5.4, the problem with BC implementation in LBM is demonstrated and the proposed solution described in detail. Finally, to verify the proposed framework, the results of some benchmarks, including Poiseuille, Cavity, Couette, and cylinder flows are presented. The formulation of the implemented BCs in the proposed framework is described in 5.8.

5.3 Lattice Boltzmann method

The most common form of the lattice Boltzmann equation in the literature is as follows [11]:

$$f_i(\vec{x} + \vec{c}_i \Delta t, t + \Delta t) = f_i(\vec{x}, t) - \frac{f_i(\vec{x}, t) - f_i^{eq}(\rho(\vec{x}, t), \vec{u}(\vec{x}, t))}{\tau} \quad (5.1)$$

where f_i represents probability distribution functions (PDFs) which represent the probability of finding a group of fluid molecules at location $\vec{x}$ and at time t with discrete velocity $\vec{c}_i$ [10] for a $D2Q9$ lattice stencil:

$$\vec{c}_i = (c_{i,x}, c_{i,y}) = \begin{cases} (0, 0), & i = 0 \\ (\cos \theta_i, \sin \theta_i) c, & i = 1, 3, 5, 7 \\ (\cos \theta_i, \sin \theta_i) \sqrt{2} c, & i = 2, 4, 6, 8 \end{cases} \quad (5.2)$$

where $\theta_i = \frac{\pi}{2} + \frac{\pi}{4}(1 - i)$, $c = \Delta x / \Delta t$ is the lattice velocity, Δx is the spatial step, and Δt is the time step. A typical $D2Q9$ stencil, with PDF indexes shown in Fig. 5.1.

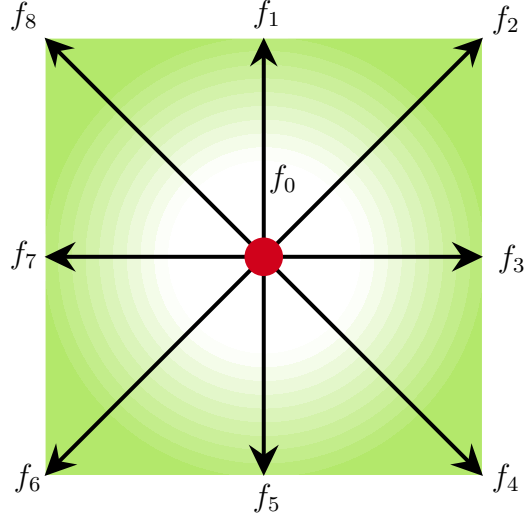


Figure 5.1 Typical two-dimensional lattice stencil of $D2Q9$ with the node (in the center with color red) and PDFs indexation f_i with $i = 0, 1, \dots, 8$.

Usually, Eq. 5.1 is solved in two steps, called collision and streaming. The collision operator is often called the LBGK [12] collision:

$$f_i^*(\vec{x}, t) = f_i(\vec{x}, t) - \frac{f_i(\vec{x}, t) - f_i^{(eq)}(\rho(\vec{x}, t), \vec{u}(\vec{x}, t))}{\tau} \quad (5.3)$$

where f_i^* is called the post-collision PDF, which represents the distribution function after collision, and τ , frequently named as a relaxation parameter, measures the average time for the PDFs to relax to their equilibrium via collisions [13]. The evolution Eq. 5.1 is completed with the streaming (or propagation) step:

$$f_i(\vec{x} + \vec{c}_i \Delta t, t + \Delta t) = f_i^*(\vec{x}, t) \quad (5.4)$$

where the left hand-side of Eq. 5.4 is the post-streaming PDF.

In Eq. 5.3, the f_i^{eq} is the local equilibrium distribution function defined as:

$$f_i^{eq} = \rho w_i \left(1 + \frac{\vec{c}_i \cdot \vec{u}}{c_s^2} + \frac{(\vec{c}_i \cdot \vec{u})^2}{2c_s^4} - \frac{\vec{u} \cdot \vec{u}}{2c_s^2} \right) \quad (5.5)$$

where w_i is a weighting factor. For example, the $D2Q9$ lattice stencil is defined as:

$$w_i = \begin{cases} 4/9, & i = 0 \\ 1/9, & i = 1, 3, 5, 7 \\ 1/36, & i = 2, 4, 6, 8 \end{cases} \quad (5.6)$$

The parameter c_s is the LBM speed of sound [14]:

$$c_s = \frac{c}{\sqrt{3}} \quad (5.7)$$

The macroscopic quantities of interest are the fluid density ρ and velocity $\vec{u}$, defined as:

$$\rho(\vec{x}, t) = \sum_{i=0}^{m-1} f_i(\vec{x}, t) \quad (5.8)$$

$$\vec{u}(\vec{x}, t) = \frac{1}{\rho} \sum_{i=0}^{m-1} f_i(\vec{x}, t) \vec{c}_i \quad (5.9)$$

where m is the total number of PDFs belonging to the lattice stencil. The kinematic viscosity of the fluid is determined by the following relation [9]:

$$\nu = c_s^2 \left(\tau - \frac{\Delta t}{2} \right) \quad (5.10)$$

and the pressure p is derived from an isothermal equation of state [9]:

$$p = c_s^2 \rho \quad (5.11)$$

Using a Chapman-Enskog expansion [137], the LBM presented above is equivalent to the following macroscopic system of equations:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \quad (5.12)$$

$$\frac{\partial (\rho \mathbf{u})}{\partial t} + \nabla \cdot (\rho \mathbf{u} \mathbf{u}) = -\nabla p + \nu \nabla \cdot (2\rho \mathbb{S}) \quad (5.13)$$

$$\mathbb{S} = \frac{1}{2} [(\nabla \mathbf{u}) + (\nabla \mathbf{u})^T] \quad (5.14)$$

where $\mathbb{S}$ is the usual macroscopic strain-rate tensor. One may note that with these equivalent LBM macroscopic equations, the kinematic viscosity ν is outside the divergence while the density ρ is inside. Consequently, this LBM system of equations is equivalent to the isothermal Navier-Stokes fluid flow equations in the incompressible limit only [137].

Note that with the LBM it is common to use a system of units where the discrete Cartesian spatial and time steps are unity, *i.e.* $\Delta x = \Delta y = \Delta t = 1$, so that the lattice velocity $c = 1$. It is also the system of units adopted for the rest of this work.

5.4 Proposed boundary condition framework

Suppose the objective is to execute a simulation on the geometry depicted in Fig. 5.2a. The first step is to generate a grid to cover the geometric entity. As illustrated in Fig. 5.2b any node may be classified as *Fluid*, *Solid*, or *Boundary* nodes. By definition, fluid nodes participate in the LBM calculation, solid nodes do not have any fluid node neighbors, and finally, boundary nodes have at least one solid and one fluid node neighbor including diagonals. In the LBM, all the nodes are superimposed by a set of so-called PDFs which are updated at each time step of the LBM simulation.

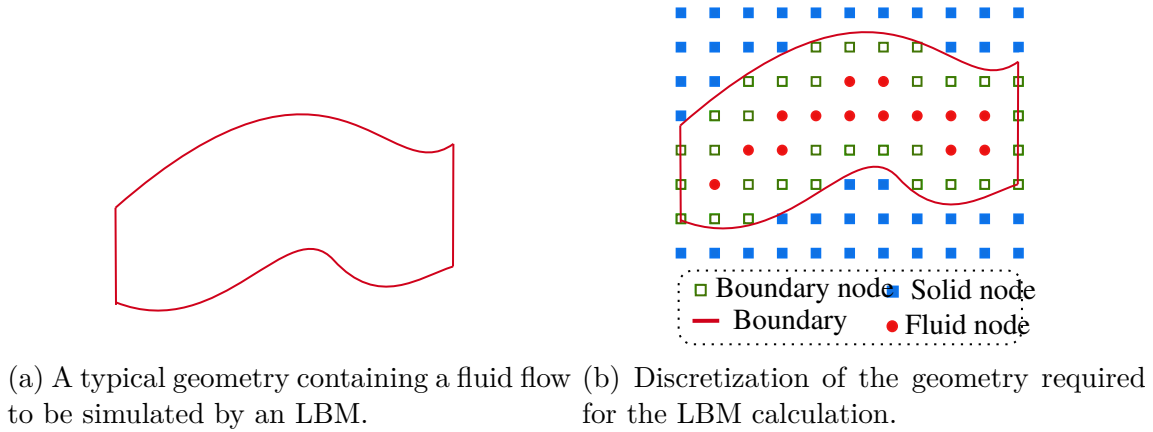


Figure 5.2 The boundary nodes emerging after domain discretization to generate a computational grid for LBM simulation.

Generally, what happens in the LBM algorithm that is needed to apply a BC? And which PDFs should be specified at the BC step? To answer these questions, examine Fig. 5.3 and imagine the streaming step has been applied to all existing PDFs. In the streaming step,

PDFs belonging to the fluid nodes are streamed to the neighboring nodes. After performing this process, the PDFs pointing out of the domain have been streamed from a fluid node's PDFs, which are known. But, those which belong to the boundary nodes and point inward the domain are unknown. So, these PDFs need to be specified by a different set of rules.

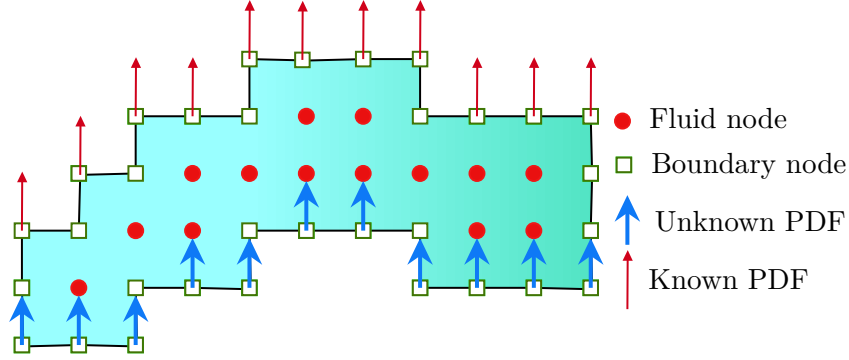
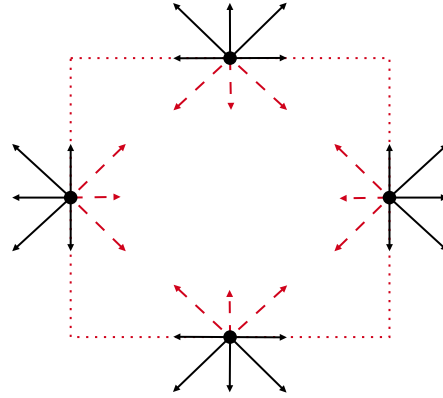


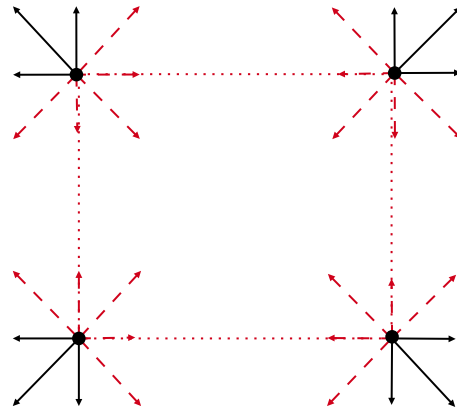
Figure 5.3 The unknown PDFs that should be specified in the BC step (only f_1 is depicted for clarity purposes).

As depicted in Fig. 5.4, there are three types of boundary nodes for 2D geometries:

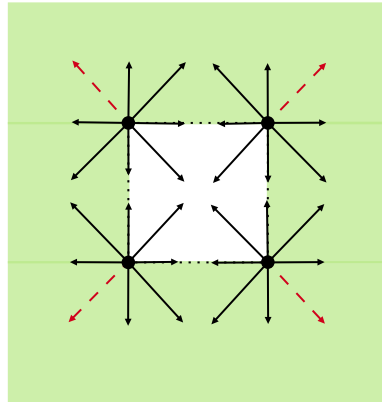
- *Edge boundaries* (as depicted in Fig. 5.4a): These lie on the edges. For 2D geometries, there are four types of these nodes, left, right, top, and bottom boundaries;
- *Convex corners* (depicted in Fig. 5.4b): These appear when the number of unknown PDFs is more than known PDFs. Four types of these corners exist; and
- *Concave corners* (as shown in Fig. 5.4c): These appear when the number of unknown PDFs is fewer than known PDFs. Four types of these corners exist.



(a) Left, right, top, and bottom edge boundaries.



(b) Convex corners: bottom-left, bottom-right, top-left, and top-right.



(c) Concave corners. For example, all the corners of a square obstacle in a channel flow.

Figure 5.4 Three types of boundary nodes in 2D geometries. Note that the highlighted region is the fluid domain. The dotted lines are the boundaries and nodes represented by filled circles. The PDFs with a red dashed line are the unknown PDFs.

What problem is frequently encountered when implementing a new BC? To answer this

question, it is helpful to consider the process. Commonly, it is necessary to derive the formulation of a BC for each type of boundary node depicted in Fig. 5.4. For example, for an edge boundary, formulations should be derived for all four sides. After programming, the code is then debugged and verified, which is time-consuming and prone to error.

The problem becomes more complicated for those corners which have to satisfy two different BCs. For instance, in Fig. 5.5a we wish to apply symmetry BC on x -axis and Zou and He on the y -axis, but as shown in Fig. 5.5b, doing so will change the system of equations. Hence, for these kinds of corners, two different formulations need to be implemented. As the number of combinations is increased, the time for debugging and verifying them is increased as well.

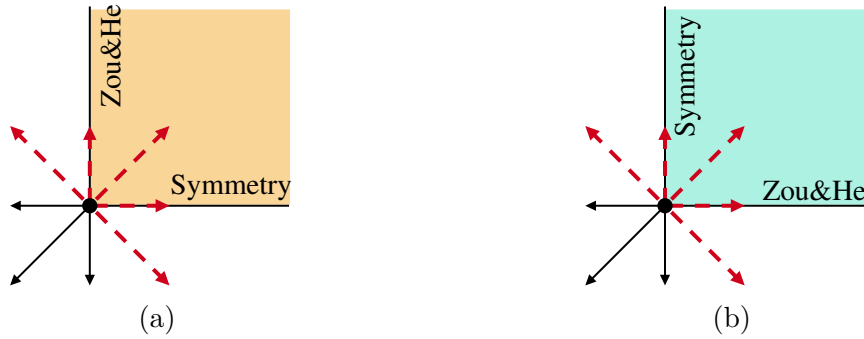


Figure 5.5 For corners with the same unknown PDFs, it is necessary to implement a BC twice as the combination of BCs for each axis changes. The PDFs with the dashed line are the unknown PDFs.

Following the above discussion, section 5.4.1 explains in detail a solution to the problem of developing BCs in the LBM. It is then followed by Section 5.4.2, where the implemented framework is described using a UML diagram.

5.4.1 Concept of the provided solution

To eliminate tedious programming work, one idea to simplify BC development is to rotate the boundary nodes' lattice stencils in such a way that they present the same unknown PDFs. For example, in Fig. 5.4a, all the boundary nodes belonging to the right side can be rotated to have the same unknown PDFs as the left boundary nodes. Similarly for the bottom and top nodes. Hence, the rotation operation is based on the orientation of the boundary nodes.

Developing a BC framework independent of the base LBM code increases the modularity of the code. This can be done by using a local data structure for the BC framework, which is independent of the data structure of the base LBM code. In this way, after the unknown PDFs

inside the BC framework are computed, they are transferred back to the original location in the data structure of the base LBM code. However, to connect this framework to an LBM code, the programmer needs to transfer all the required data from the base LBM code. The proposed framework is shown in Algorithm 2.

Algorithm 2: Boundary condition framework.

```

1 Specifying the boundary nodes
2 Specifying the PDFs that belong to the boundary nodes
3 Specifying the orientation of boundary nodes
4 Mapping PDF pointers
5 Writing the original PDF values into the local data structure
6 for All the regions do
7   for All the nodes of the region do
8     Mapping quantity vectors (if required)
9     Invoking BC function
10  end
11 end
12 Writing the local PDF values into the base LBM code data structure
```

Note that only non-moving boundaries are studied in this work; consequently, lines 1-4 are carried out in the pre-processing step, and lines 5-12 in the time marching loop of the base LBM code. Each line of the above framework is explained in more detail in the following section.

Specifying the boundary nodes

First we note that the appropriate data structure for implementation of the proposed BC framework is an Array of Structure (AoS) [126] memory layout, which means all the PDFs associated with a boundary node are stored contiguously in the memory. Regardless of the data structure of the base LBM code, a local data structure for BC implementation is created that includes two steps: specifying the boundary nodes and specifying the PDFs belonging to them. The first step is explained in this section and the second step in Section 12.

To specify the boundary nodes, each boundary node can be classified based on two criteria: (1) the orientation of the boundary a node belongs to, and (2) the type of BC that should be applied. For example, in Fig. 5.6 all of the boundary nodes can be categorized into eight regions based on their orientation. At the implementation level, one may declare an array to store the number of nodes belonging to each region and another array, **iNodeBC**, to

store the indexes of each boundary node. However, nodes belonging to each region should be stored contiguously, as depicted in Fig. 5.7.

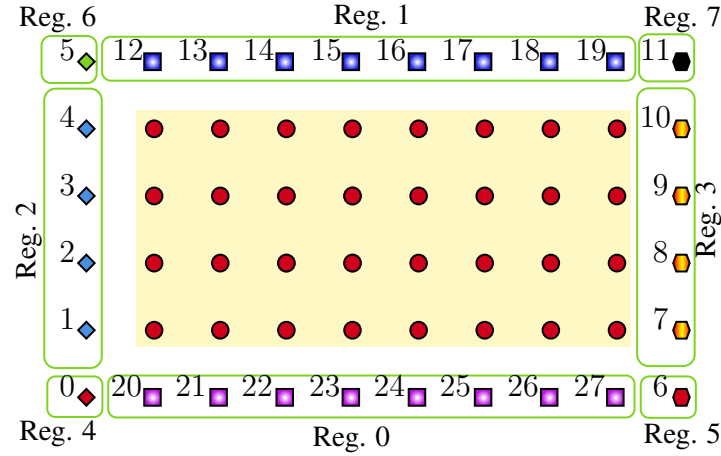


Figure 5.6 Classifying the boundary nodes into different regions based on their orientation. For example, all nodes belonging to the bottom side are stored in region 0, while the bottom-right corner is stored in region 5.

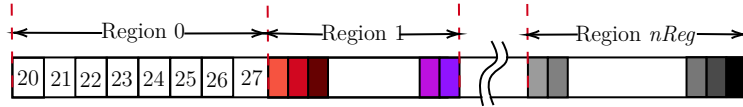
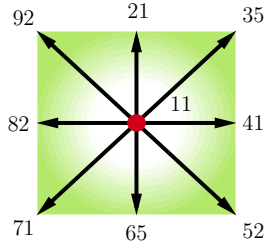


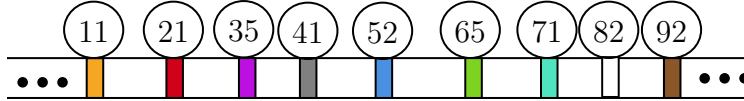
Figure 5.7 Storing the indexes of nodes of each region. The quantity $nReg$ represents the total number of regions. For example, the data illustrated in Region 0 of Fig. 5.6 is stored contiguously at the beginning of the array containing all the boundary nodes.

Specifying the PDFs belonging to the boundary nodes

For BC implementation all the PDF indexes belonging to the boundary nodes are stored in the second step of creating a local data structure. As mentioned previously, this data structure uses an AoS data layout to store the PDFs. For a typical node, as illustrated in Fig. 5.8a, the indexes of PDFs belonging to this node are (11, 21, 35, 41, 52, 65, 71, 82, 92). An array like the one illustrated in Fig. 5.8b contains the PDFs belonging to the data structure of the base LBM code. As illustrated in Fig. 5.8c, the method is to create a local array **iPDFbc** containing the indexes of PDFs belonging to boundary nodes. The indexes of these PDFs are stored in **iPDFbc** using an AoS memory layout.



(a) Boundary node PDF indexes that need to be stored in a local data structure.



(b) Address of the PDF in the base LBM code data structure. These PDFs could be anywhere in the code.



(c) Storing the PDF indexes of each boundary node in a local AoS (**iPDFbc**). Here each structure contains all the PDFs of a node. For instance, PDFs of nodes in Fig. 5.7 expand to form array **iPDFbc**.

Figure 5.8 Creation of a local data structure for all the PDF indexes belonging to boundary nodes. These PDF indexes are copied from the base LBM code to a local data structure (**iPDFbc**) with an AoS memory layout.

Specifying the orientation of boundary nodes

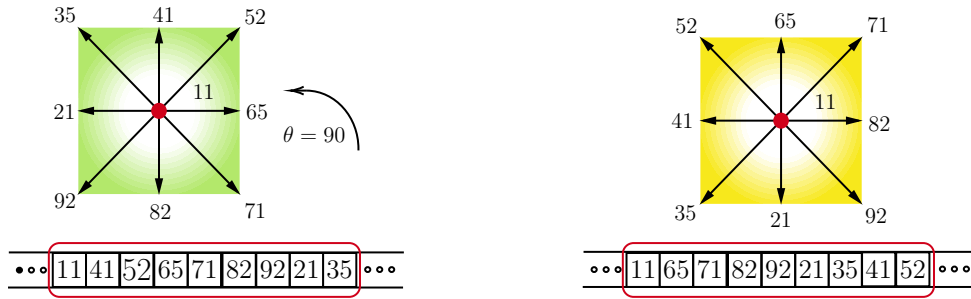
To perform the mapping operation, which is described later in Section 12, it is first required to specify the orientation of the boundary nodes. Here, an index base table is used to store the orientation. For example, indexes 0 to 3 represent the bottom, top, left, and right side boundary. As in line 3 of Algorithm 1, this step is performed in the pre-processing step for stationary geometries.

Mapping PDF pointers

As seen in Fig. 5.4, any boundary node belonging to an edge may be rotated in such a way that it has the same unknowns as the left side boundary. Convex and concave corners

depicted in Fig. 5.4 can be set to have the same unknown PDFs as the bottom-left corner. But first, let's clarify two required operations called *rotation* and *flip*.

Rotation operation means applying a circular shift on the part of the array **iPDFbc** that contains all the node's PDF indexes. For example, suppose it is required to rotate a typical stencil illustrated in Fig. 5.9a by 90 degrees. As depicted in Fig. 5.9b the indexes of PDFs after rotation are indicated on the lattice stencil. At the implementation level, it is enough to shift the PDF indexes in the array containing local boundary PDFs. Note that f_0 does not participate in the rotation operation.



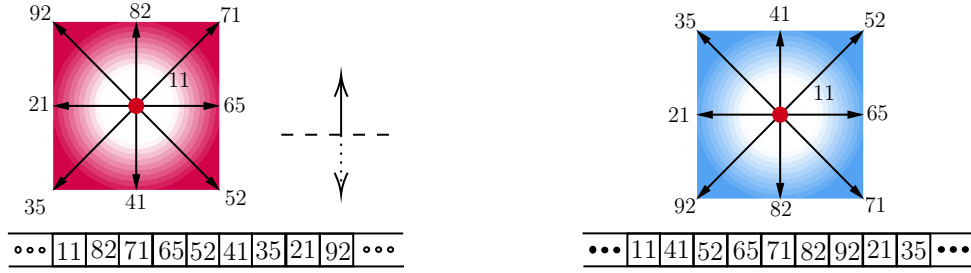
(a) A typical stencil and associated part of (b) Indexes of PDFs and the **iPDFbc** array before rotation. after rotation.

Figure 5.9 Concept of the rotation operation. As the angle between PDF arrows is 45 degrees, to rotate PDFs by 45 degrees, it is necessary to apply a circular shift on the structure containing the PDF indexes by one unit. Similarly, for a 90 degree rotation, PDF indexes are shifted by two units. Anti-clockwise rotation requires a left shift and clockwise rotation requires right shift.

Flip operation means to mirror the PDF indexes about x -axis (note that there is no flip around y -axis). Suppose we want to flip the lattice stencil illustrated in Fig. 5.10a. As seen in Fig. 5.10b, all we need to do is to swap the indexes of the PDFs as follows:

$$\begin{aligned} f_2 &\Leftrightarrow f_4 \\ f_1 &\Leftrightarrow f_5 \\ f_8 &\Leftrightarrow f_6 \end{aligned} \tag{5.15}$$

while the other PDFs remain unchanged.



(a) A typical lattice stencil and its associated PDF indexes. (b) Rearranging PDF indexes of the stencil after a flip operation.

Figure 5.10 Flip operation for a typical lattice stencil.

To map the PDFs of a boundary node to the left side boundary, all the required degree rotations, as well as the flip operation, are provided in Table 1. As can be seen in Table 1, to map the edge boundaries to the left side boundary, the rotation operation alone is sufficient.

To map all the corners to the bottom-left corner, a combination of rotation and flip operation is required. Although it is possible to do the mapping only by rotation, it is critical to note that after mapping a corner, those parts of the corner lying on the x -axis should be mapped in such a way that they lie exactly on the x -axis after mapping. This is because when two BCs from a corner are mapped, the resulting combination should be the same as the original. For example, suppose it is desired to apply symmetry BC on x -axis and a velocity profile on the y -axis at the bottom-right corner. After mapping the symmetry and velocity should be applied on the x - and y -axis. In summary, all the required operations to map all the corners to the bottom-left corner are provided in Table 1.

Table 5.1 The required degree rotations and flip operation for any boundary edge/corner to be mapped to the left-side edge/bottom-left corner.

Boundary orientation	Flip	Rotation (degree)
Bottom side	No	-90
Top side	No	90
Left side	No	0
Right side	No	180
Bottom-Left corner	No	0
Bottom-Right corner	Yes	180
Top-Left corner	Yes	0
Top-Right corner	No	180

Writing the original PDF values into the local data structure

At this point, there is an array called **iPDFbc** which contains all the boundary PDF indexes. Note that this array is organized as an AoS data layout. Consequently, to access any PDF index inside **iPDFbc**, the following formula is used:

$$G(l, i) = l \times m + i \quad (5.16)$$

where the quantity l denotes the l th ordered node in array **iPDFbc**, m is the number of PDF belonging to a node (for example in *D2Q9*, this is $m=9$), the index i refers to the desired PDF that has velocity $\vec{c}_i$, while G is the location of the PDF index in memory.

In this framework, an array called **vPDFbc** is also used to store the boundary PDF values. As mentioned previously, the indexes of boundary PDFs are known within the framework as they are stored in **iPDFbc**. Hence, it is also possible to access the PDF value in the base LBM code. At this step, a copy of the boundary PDF values is stored in the **vPDFbc** array using the same AoS memory layout as the **iPDFbc** array.

How to traverse boundary nodes

As illustrated in Algorithm 2 lines 6 to 11, to compute the unknown PDFs it is necessary to traverse all the boundary nodes. Based on the data structure shown in Fig. 5.7, a two-nested loop is used to access a boundary node, as shown in Listing 1. As described in Section 12, the indexes of the boundary nodes are stored in array **iNodeBC**, and then used in the inner loop of Listing 1.

Listing 5.1 How to access boundary nodes. The quantity $nReg$ is the number of boundary regions and $nNodeReg$ the number of nodes belonging to each region. The outer loop traverses all the regions while the inner one traverses the nodes belonging to each region.

```

1  int sum = 0;
2  for (int iReg = 0; iReg < nReg; iReg++) {
3      for (int i = sum; i < sum + nNodeReg[iReg]; i++) {
4          iBoundaryNode = iNodeBC[i];
5          //Computing the unknown PDFs (i.e. line 8 and 9
6          //of Algorithm 1.)
7      }
8      sum += nNodeReg[iReg];
9  }
10
11
```

Mapping quantity vectors

Since the PDFs belonging to a boundary node need to be mapped, the vector quantities associated with that node should be mapped accordingly. So, all the operations, *i.e.* rotation

and flip, applied to the PDF indexes should also apply to the vector quantities such as the velocity components. For example, as depicted in Fig. 5.11a, there is an edge boundary located on the top side with a given velocity. As explained previously, to map a node belonging to this boundary to the left side boundary, its PDF indexes should be rotated by 90 degrees. Consequently, the velocity vector should be rotated the same way, as illustrated in Fig. 5.11b.

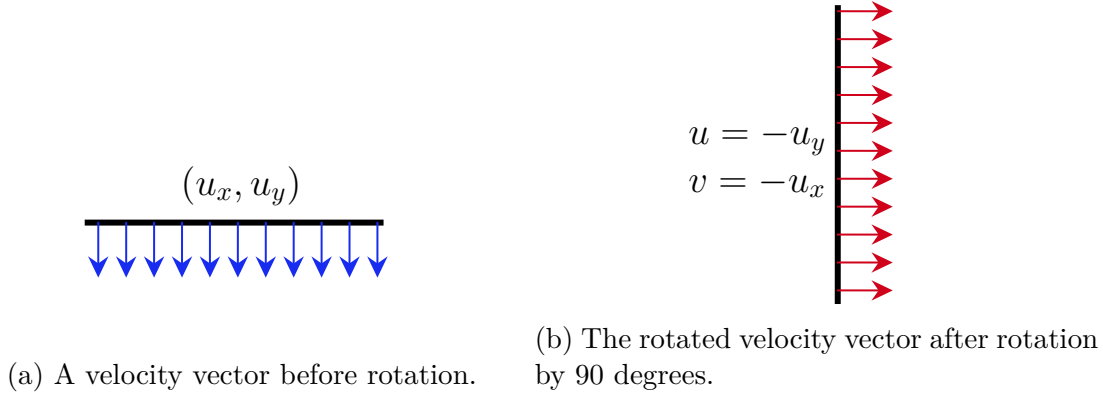


Figure 5.11 All the vector quantities should be mapped in the same way as the PDF indexes are mapped. The quantities u and v represent the rotated velocities in the x - and y -direction, respectively.

Invoking BC function

Based on Algorithm 2, any BC is implemented only for the left side/bottom-left corner. So, in the nested loops described in Listing 1, the appropriate BC function is invoked to compute the unknown PDF values. Following that, all the computed PDFs are written in the local data structure of the framework, *i.e.* in the **vPDFbc** array.

Writing back the local PDF values into the base LBM code's data structure

After computing the unknown PDFs, just after the two-nested loops, they are written back into the original array containing all the PDFs of the base LBM code. Since the PDF indexes are mapped, there is no need to rotate/flip the results back to the original position.

5.4.2 Class diagram of the implemented code

In this section, we provide a brief description and instruction for using the proposed framework. Note that C++ object-oriented programming was used for implementation. The source

code of the framework is available on **GitHub** [138].

As illustrated in Fig. 5.12, to transfer data from/to base LBM code, we designed an abstract class called **DataStBC**. This class already contains some methods as well as the local data structure of the framework. Hence, this abstract class is implemented to link any LBM code to the framework.

The main class of this framework called **MainBC** and is composed of all the implemented BC classes. To use the framework in any LBM code, an object must be created from this class. After the construction of the **MainBC** object, the only public API the user needs to use is the *bcPDFvCalculation()* method. This method is responsible for specifying the unknown boundary PDFs. Note that an object of an implemented class inherited from **DataStBC** is passed to the constructor of **MainBC** class.

There is also an abstract class called **D2Q9BC**, which is implemented to develop a new BC class. This class has two methods: *execut()* and *scheme()*. In the execute method, lines 6-11 of Algorithm 2 are implemented. However, the *scheme()* method is a virtual method that needs to be implemented in the derived classes. This method is responsible for updating the unknown PDFs as it is invoked in the *execut()* method of the **D2Q9BC** class.

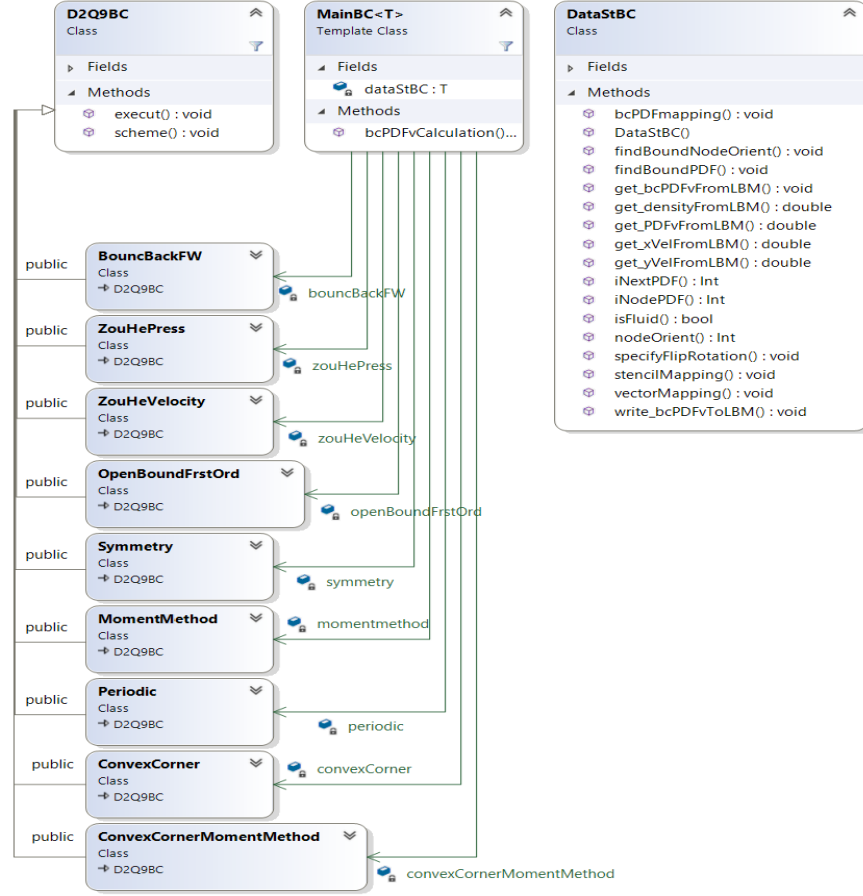


Figure 5.12 Class diagram of the proposed boundary condition framework.

5.5 Results and discussion

As a goal to test the capabilities of the proposed framework for implementing different BCs, some of the most common BCs in the LBM are implemented. The results obtained are compared and checked against the analytical solution or known numerical results. These included Zou and He pressure, Zou and He velocity, open boundary, symmetry, full-way bounce-back, periodic, moment-method (for zero velocity), convex corner, and moment-method for corners (for zero velocity). These common BCs are explained in more detail in 5.8. In addition, in this section, Poiseuille, cavity, Couette, and cylinder flows are used to study BCs. The BCs were also tested for all types of boundaries and different orientations to verify the accuracy of the implementation. In particular, the flows were simulated for all directions, *e.g.* right to left, left to right, top to bottom, and bottom to top. However, as the results were consistently identical, note that only the cavity flow results in all directions are presented in Section 5.5.2. Overall, the results section focuses on the verification aspect of code [139].

5.5.1 Poiseuille flow

In particular, this test case aims to verify the implementation of full-way bounce-back, moment-method, Zou and He pressure, and convex-corner BCs, as described in 5.8. To accomplish this the velocity profile in x -direction, obtained from simulation, are compared to the analytical solution.

As depicted in Fig. 5.13, we consider the Poiseuille flow of a fluid in a rectangular channel $2W$ wide and L in length. The flow is driven by a pressure gradient between the inlet and outlet sides of the channel. So, only one component of velocity in x -direction is non-zero.

The analytical solution, u_a , for the horizontal velocity profile of the Poiseuille flow is given as [40]:

$$u_a(y) = \frac{W^2 \Delta P}{2\rho\nu L} \left(1 - \frac{y^2}{W^2}\right) \quad (5.17)$$

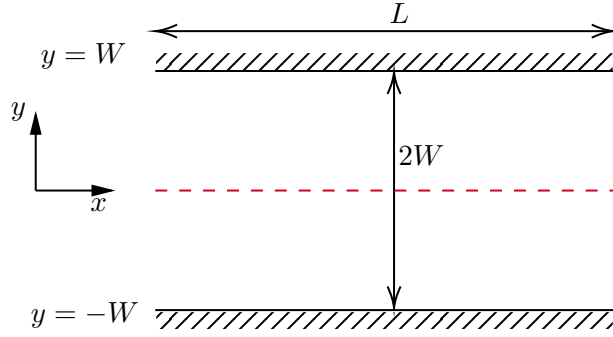


Figure 5.13 The geometry of Poiseuille flow. Flow is driven by a pressure gradient from the left inlet to the right outlet.

To define the pressure drop ΔP between the inlet and outlet, we consider the following definition for the Reynolds number Re of this Poiseuille flow:

$$Re = \frac{UW}{\nu} \quad (5.18)$$

where the maximum velocity U occurs at the centerline of the channel. So, the maximum velocity as given by Eq. 5.18 is:

$$U = \frac{\Delta P W^2}{2\rho\nu L} \quad (5.19)$$

By combining Eq. 5.18 and 5.19 the pressure drop in the function of the Reynolds number

and other parameters is calculated as:

$$\Delta P = \frac{2\rho\nu^2 L}{W^3} Re \quad (5.20)$$

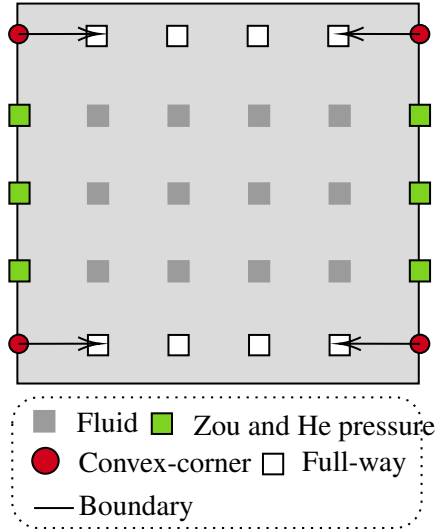
The density at the inlet/outlet are given by $\rho_{in} = \bar{\rho} + \frac{1}{2} \frac{\Delta P}{c_s^2}$ and $\rho_{out} = \bar{\rho} - \frac{1}{2} \frac{\Delta P}{c_s^2}$. So that the average density in the channel is $\bar{\rho} = 1$. The initial PDFs are the equilibrium PDFs computed from Eq. 5.5 with $\rho = \bar{\rho} = 1$ and $\vec{u} = 0$. To do a grid study a set of grid sizes, with $\nu = 1/6$, and Re number of five has been used in the simulations presented below. The L^2 norm, $||\cdot||_{L^2}$, normalized by the centerline velocity U is used to determine the error and the degree of accuracy of the model, and this is done according to the error computed as:

$$\begin{aligned} \text{L2error} \equiv \frac{1}{U} ||u - u_a||_{L^2} = \\ \frac{1}{U} \sqrt{\frac{1}{iMax \times jMax} \sum_{i=0}^{iMax-1} \sum_{j=0}^{jMax-1} \left(u(i, j) - u_a(i, j) \right)^2} \end{aligned} \quad (5.21)$$

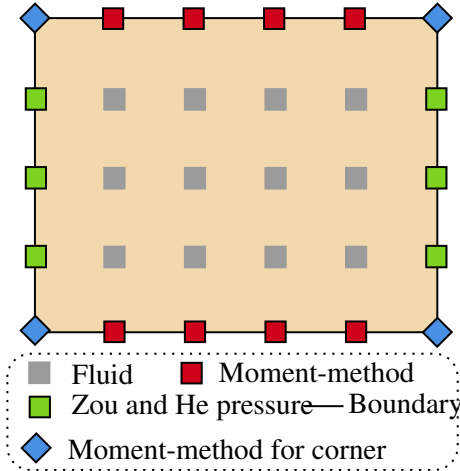
where $iMax$ and $jMax$ are the grid size in the x and y directions respectively.

Based on the description of this test case, fixed values must be applied for pressure (or density) on the left and right sides of the geometry, which requires the Zou and He pressure BC to be applied on the left inlet and right outlet. In this problem, one of the objectives was to verify full-way bounce-back and moment-method to simulate the no-slip wall, applied on both the top and bottom side nodes in each simulation. As depicted in Fig. 5.14a to apply full-way bounce back the wall is located midway from the boundary node. For this case, to apply the convex-corner BC, the velocity must be extrapolated. The nearest neighbor extrapolation was used, which means the velocity at the corner nodes is equal to the node lying on the bottom/top boundary as depicted in Fig. 5.14a.

In addition, to apply the moment-method, wall boundaries are lying at the boundary nodes as shown in Fig. 5.14b. In this case, an extrapolation is not needed as the boundary nodes are lying on the physical no-slip wall.



(a) Left and right sides are Zou and He pressure, top and bottom are full-way bounce-back. All the corners are convex-corner BC. The arrows indicate the nodes contributed to the extrapolation. Here, collision is applied to the full-way nodes, as they are not so-called ghost nodes. For more detail refer to 5.8.9.



(b) Left and right sides are Zou and He pressure, top and bottom are moment-method. All the corners are moment-method for corner.

Figure 5.14 Different BCs used to simulate Poiseuille flow problem.

In this test case, the velocity profile in x - direction is compared with the analytical solution. Both the numerical and analytical velocities are normalized by the maximum analytical velocity at the centerline. The velocity profile depicted in Fig. 5.15 shows good agreement with the analytical solution, which verifies the implementation of the BC using the proposed framework.

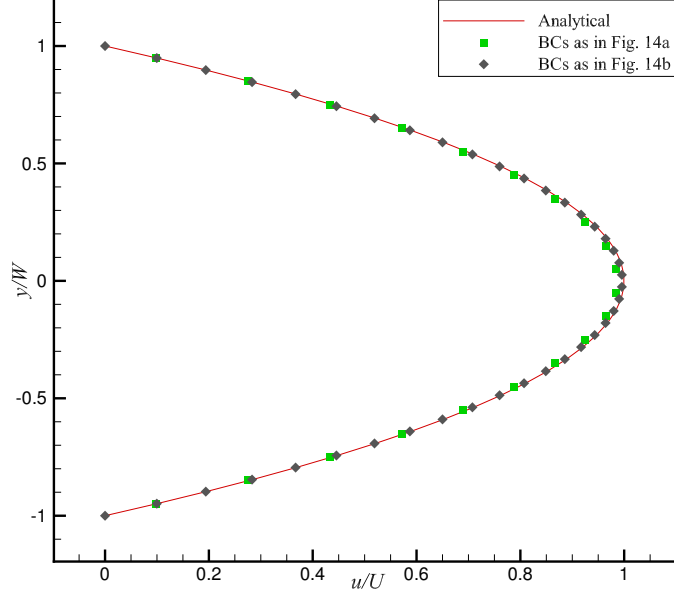


Figure 5.15 Comparison of analytical and numerical velocity in x -direction of Poiseuille flow of $Re=5$ and grid size (20×20) at $x = L/2$.

Fig. 5.16 illustrates the error evolution as a function of the domain size in y -direction for $Re=5$. For the BCs in Fig. 5.14a, the slope of the linear curve is approximately -2, indicating second-order accuracy, as expected [41]. Note that in this case a domain of $L = 2W$ based on the notation of Fig. 5.13 is simulated. To preserve this aspect ratio a grid size of (12×22) , (24×46) , (48×94) , (96×190) , and (192×382) was selected. For the BCs in Fig. 5.14b, the results indicate second-order accuracy. In this test case, a grid size of (12×24) , (24×48) , (48×96) , (96×192) , and (192×384) was selected to preserve the aspect ratio.

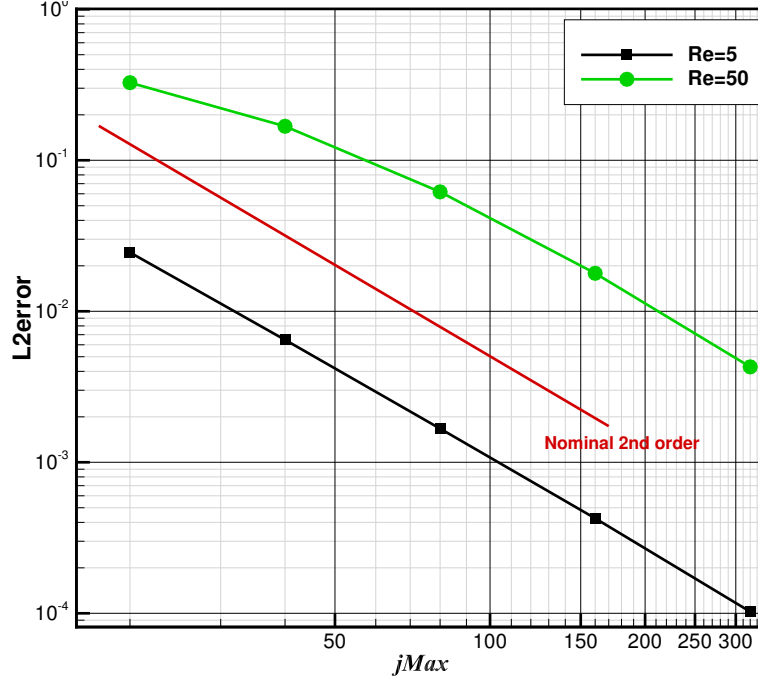


Figure 5.16 Logarithmic error for Poiseuille flow test case, using boundary conditions depicted in Figs. 5.14a and 5.14b. In this analysis we used $Re=5$.

In summary, by simulating the Poiseuille flow problem and applying different BCs we conclude that the proposed framework to implement any particular BC for just one side and then apply it to any desired side is working well. In addition, the convex-corner BC was implemented for just the bottom-left corner, and in this problem, used for all the corners. Moreover, these test cases were carried out for four orientations (i.e. a flow from left to right, right to left, top to bottom, and bottom to top) with identical results. All tests used a unique implementation for each kind of BC.

5.5.2 Cavity flow

The initial purpose of this test case was to verify the implementation of the Zou and He velocity BC. Also, to again test the full-way bounce-back, moment-method, Zou and He pressure, and convex-corner BCs. However, simulating a more complex flow field was necessary to have confidence in the correct implementation of any newly developed CFD code.

One of the most tested and studied test cases in the CFD field is the lid-driven cavity flow. As depicted in Fig. 5.17, the cavity is square, and the flow is driven by the top lid, which moves with uniform speed U . In this test case, the Reynolds number is based on the length

of the moving lid L , its speed U , and the fluid's kinematic viscosity ν , according to Eq. 5.22:

$$\text{Re} = \frac{UL}{\nu} \quad (5.22)$$

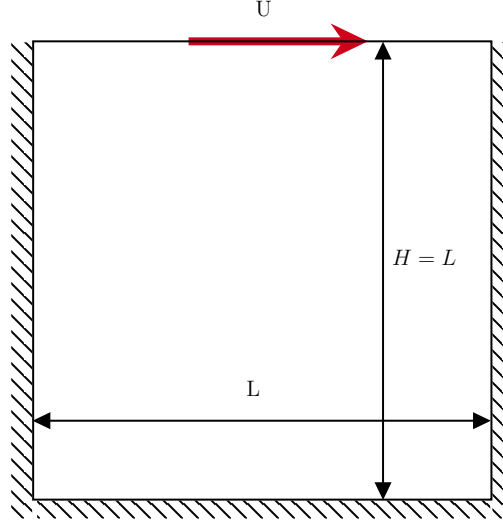


Figure 5.17 Geometry of the lid driven cavity.

The Zou and He velocity BC was applied on the top side by imposing a uniform velocity of U . For the remaining sides, the moment-method BC was used as a no-slip wall. As the bottom-right and bottom-left corners are the intersections of two no-slip walls, it is possible to use a moment-method corner BC. However, since the velocity is non-zero at the top-right and top-left corners, for these corners the convex-corner BC was applied as illustrated in Fig. 5.18.

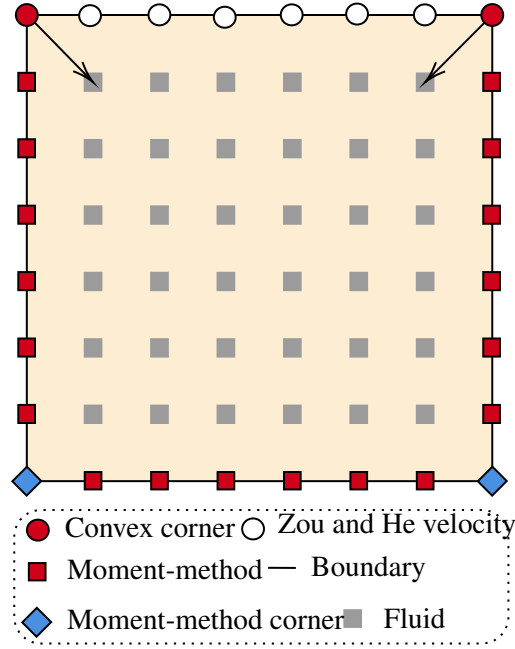


Figure 5.18 Different BCs for simulating the cavity flow problem: Left, right, and bottom sides are moment-method, top side is Zou and He velocity, bottom-left and bottom-right are moment-method corner, and top-left and top-right are convex-corner BC. The arrows indicate which node contributes to the extrapolation.

In this problem, the presence of a central vortex and the recirculation areas at the bottom corners were considered. All the numerical data obtained from the proposed code, including velocity profiles and vortex locations, were compared to the results of Ghia *et al.* [1]. In this problem, the velocity in x - and y -direction at the midline of the cavity, and also the location of the primary and bottom corner vortices, were used for verification.

As the results in Fig. 5.19 show, the vortex location is properly simulated. In addition, for quantitative comparison, Table 5.2 shows the position of the vortices from Ref. [1] and the proposed work.

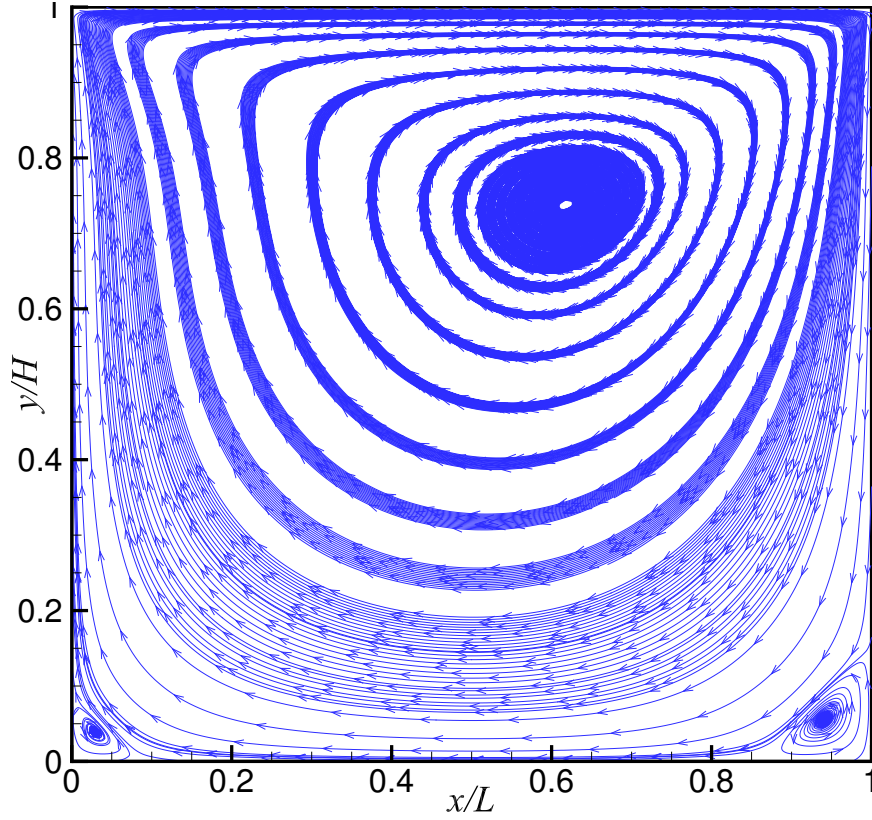


Figure 5.19 Streamlines for square cavity flow at $Re=100$ and grid size of 129×129 . The red colored points are the vortex locations from Ghia *et al.* [1].

Table 5.2 Primary and bottom corner vortex location at $Re=100$ with grid size 129×129 compared to results from Ghia *et al.* [1].

	Primary Vortex (x , y)	Lower Left (x , y)	Lower Right (x , y)
Ghia <i>et al.</i> [1]	(0.6172 , 0.7344)	(0.0313 , 0.0391)	(0.9453 , 0.0625)
Proposed work	(0.6187 , 0.7401)	(0.0290 , 0.0394)	(0.9406 , 0.0556)

In Fig. 5.20, the velocity profiles obtained via the LBM simulations are compared to the results of Ghia *et al.* [1]. In particular, the velocity profiles presented are for the vertical and horizontal centerlines of the Cavity. On the vertical line is the velocity profile in x -direction, while the horizontal line, shows the velocity in y -direction. There is also acceptable agreement with results reported in Ref. [1].

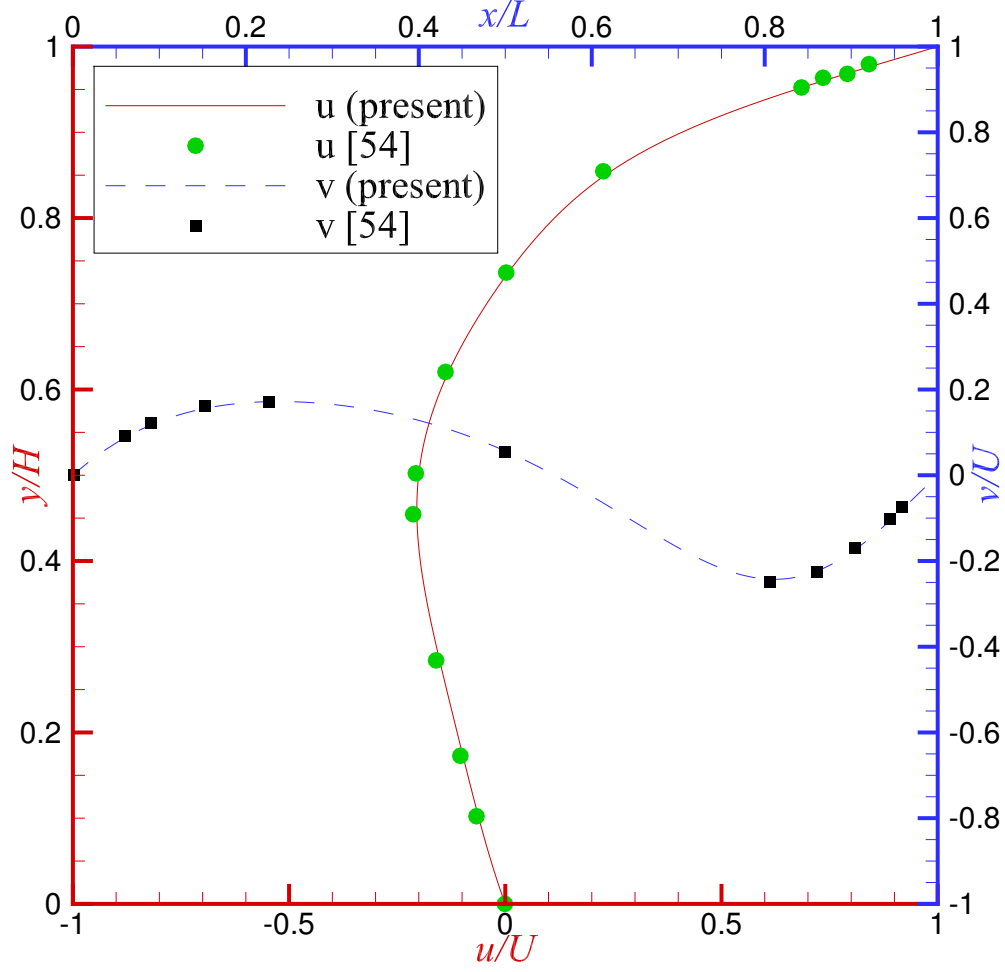


Figure 5.20 Velocity profile at the center line of the cavity at $Re=100$ and grid size 129×129 .

As mentioned previously, the main goal of this framework is to eliminate multiple implementations of BC for different orientations. Hence, for the cavity flow problem, the moving side, *i.e.* lid-driven, was applied on different sides. The streamlines and the vertex positions are shown in Fig. 5.21 which verify the implementation of the framework. In addition, for quantitative comparison, the density values at points 1, 2, 3, and 4 in Figs. 5.21b to 5.21d are compared to Fig. 5.21a. The density at each point (*i.e.* 1, 2, 3, and 4) was expected to be the same in different simulations. Comparison of density at these points are identical to 16 digits, as shown in Table 5.3

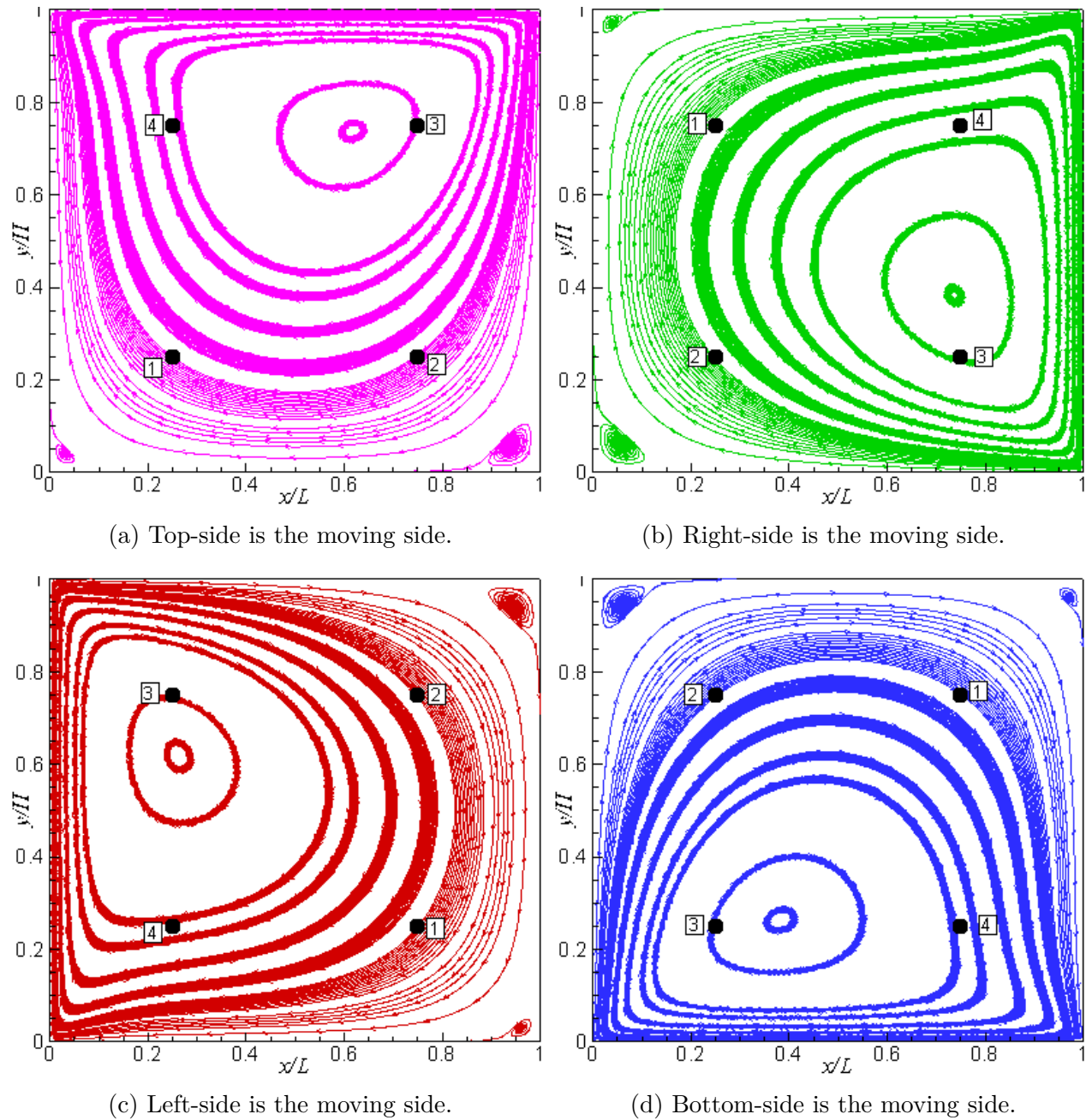


Figure 5.21 Simulation of cavity flow for different moving sides. The position of streamlines are approximated.

Table 5.3 Density value at different locations for different directions of cavity flow simulation.

Moving side	Point 1	Point 2	Point 3	Point 4
Top (Fig. 5.21a)	0.9976270759507726	1.0018862779944189	0.9999158213869475	0.9965243244188207
Right (Fig. 5.21b)	0.9976270759507724	1.0018862779944187	0.9999158213869475	0.9965243244188208
Left (Fig. 5.21c)	0.9976270759507724	1.0018862779944187	0.9999158213869475	0.9965243244188208
Bottom (Fig. 5.21d)	0.9976270759507722	1.0018862779944187	0.9999158213869472	0.9965243244188206

In this test case, the Zou and He velocity was verified. So, to this point, it is clear the proposed framework can be used for the implementation of local and non-local BCs for both edges and corners. All orientations have been tested and results are identical to machine accuracy. Again, for this test case, only one edge/corner of each BC was implemented.

5.5.3 Couette flow

The Couette flow is a specific flow driven by the motion of a plane over another as depicted in Fig. 5.22. To implement a non-local BC another interesting test case is the Couette flow, which is suitable to verify the periodic BC. Here again the Zou and He velocity and convex-corner BCs were verified. In addition, this test case was used to verify an unsteady flow as there is an analytical solution for unsteady Couette flow.

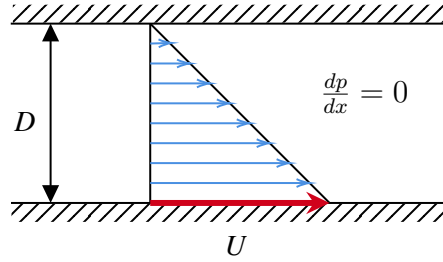


Figure 5.22 Geometry and the flow field of Couette flow problem.

Since there is no pressure gradient in the direction of the flow, only the moving bottom plate creates the fluid motion. The transient solution [140] of this flow is given by Eq. 5.23:

$$u/U = \left(1 - \frac{y}{D}\right) - \frac{2}{\pi} \sum_{n=1}^{\infty} \frac{\sin(\lambda_n y)}{n} e^{-\lambda_n^2 \nu t} \quad (5.23)$$

which $\lambda_n = n\pi/D$. However, by substituting the t to infinity we obtain the steady solution:

$$u/U = \left(1 - \frac{y}{D}\right) \quad (5.24)$$

Since the study is focused on no-pressure gradient Couette flow, a periodic BC was applied

on the left and right sides of the domain. The Zou and He velocity BC was applied to the bottom and top sides. However, the velocity is zero at the bottom side. The convex-corner BC was used for the corner nodes. A summary of the BCs is illustrated in Fig. 5.23.

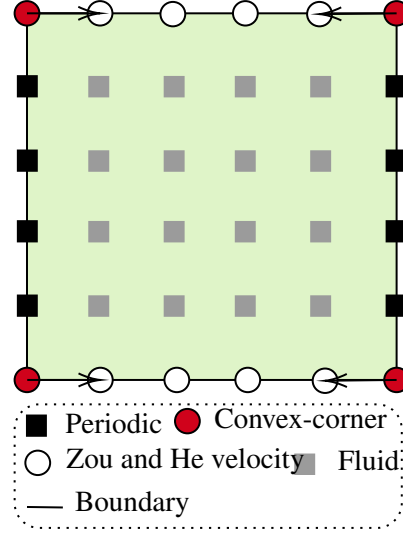


Figure 5.23 Various kinds of BCs for simulating the Couette flow problem: Zou and He velocity for top and bottom sides, periodic for left and right sides, and convex-corner for all the corners. The arrows indicate which node contributes by extrapolation.

Steady state solution

As expected from the theoretical solution of Couette flow the velocity profile is linear in the steady state. Also, the lid condition on the bottom wall was well simulated, as the velocity of the bottom wall and the layer of fluid in contact is identical as depicted in Fig. 5.24.

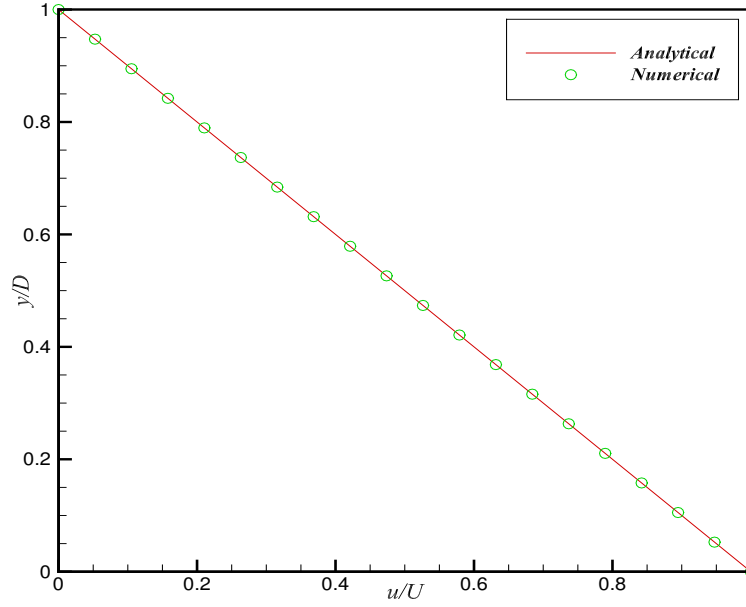


Figure 5.24 Comparison of analytical and numerical velocity of Couette flow in x -direction for $Re=10$ and grid size of 20×20 .

Unsteady solution

For the unsteady simulation the Couette flow problem was carried out for $Re=10$ and 100 . The results of these simulations are depicted in Fig. 5.25. For $Re=10$ a grid size of 20×20 was used and the numerical and analytical velocity profile for time step 100 and 300 compared in Fig. 5.25a. The same investigation was carried out for $Re=100$ but the grid size was increased to 100×100 . In this test case the velocity profile was taken at time step 3000 and 5000. As seen in Fig. 5.25b there is good agreement between the numerical and the analytical solution.

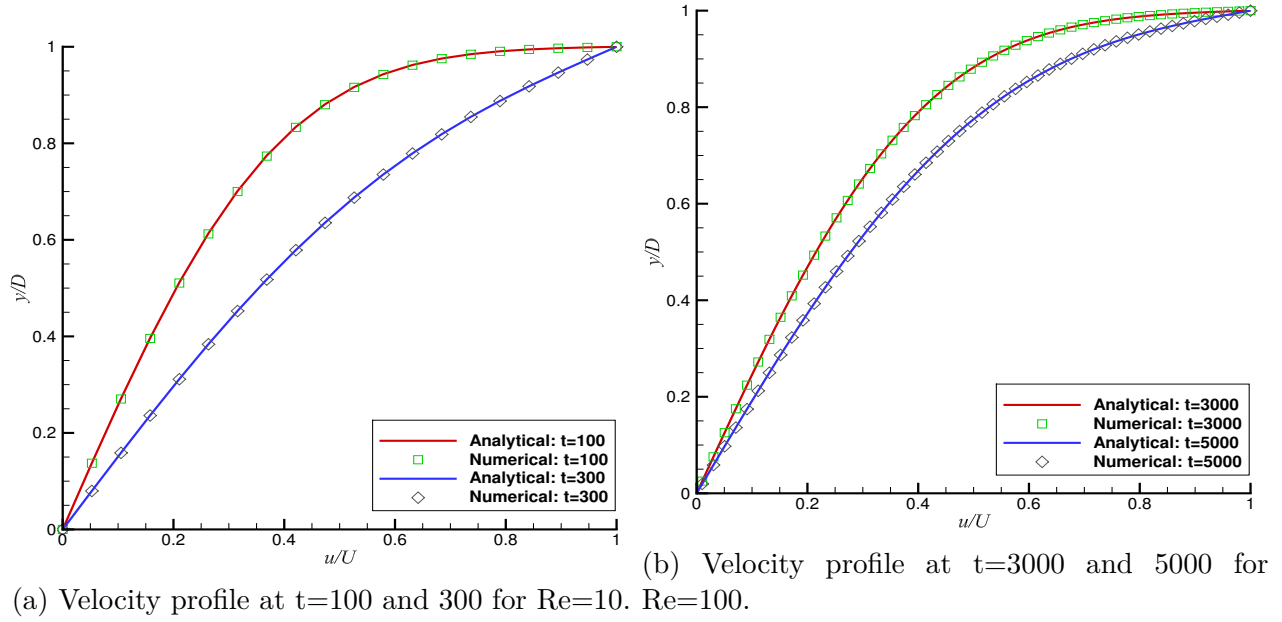


Figure 5.25 Comparison of analytical and numerical velocity of Couette flow in x -direction for $Re=10$ and $Re=100$. A grid size of 20×20 and 100×100 was used for $Re=10$ and $Re=100$ respectively.

In this test case, the implementation of periodic BC as a non-local BC was investigated, and results indicate the implementation is correct. In this framework, the user just needs to determine the two periodic boundaries and the periodicity is applied automatically (for more detail refer to 5.8.4). Again, for this test case, periodic BC was implemented only for the left side then applied automatically on both the left and right sides. For this test case, all orientations were tested and the results were identical to machine accuracy.

In this test case the flow is linear and the LBM is second order. So, it can capture exactly the velocity profile. It can therefore capture the velocity profile with the machine's accuracy. A grid convergence analysis is therefore not necessary.

5.5.4 Cylinder flow

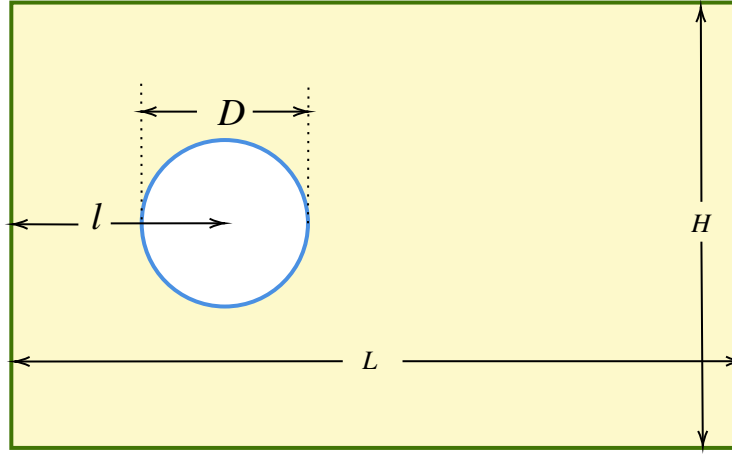
The goal of this test case was to verify the implementation of symmetry and open boundary BC. Simulating an unsteady flow using this BC framework was also investigated. Since the open boundary is not a local BC, like periodic BC, it depends not only on the PDFs of the node to which the BC is applied. Implementation of full-way bounce-back for a non-straight wall was also studied in this test case. In this situation, the orientation of the boundary nodes are different as they do not lie on a straight line. To accomplish these goals two different

geometries and set of BCs were investigated.

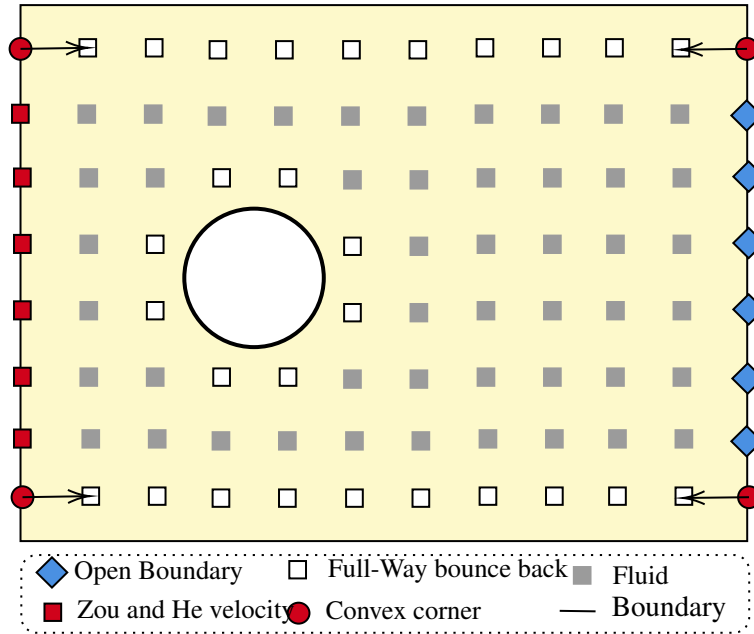
In the first test case for the cylinder flow, a cylindrical obstacle is placed in a rectangular domain as depicted in Fig. 5.26. This schematic diagram corresponds to the numerical study of the test case studied in Ref. [132]. The circular cross-section of the cylinder of diameter D is mounted centrally inside a planar channel of height H . The channel length is set at $L = 50D$ to reduce the influence of the outflow BCs on accuracy. The upstream length $l = L/4$ and $H = 0.16L$ is set as in Ref. [132]. The Reynolds number is based on the cylinder diameter D , the inflow velocity U_{in} , and the fluid kinematic viscosity ν as in Eq. 5.25:

$$\text{Re} = \frac{U_{in}D}{\nu} \quad (5.25)$$

As depicted in Fig. 5.26b a uniform velocity profile was applied on the left side of the domain to produce a flow around the cylinder using the Zou and He velocity BC. The top and bottom sides as well as the cylinder were set as full-way bounce back. The open boundary BC was applied to the right outlet of the domain. The convex-corner BC was applied on all four corners. For the two corner nodes at the inlet side the y -direction velocity was set to zero and the x -direction velocity set as the inlet velocity. Also, the density was extrapolated from a node on the bottom/top side. For the two corner nodes on the outlet the y -direction velocity was set to zero but the x -direction velocity and the density were extrapolated from a node on the bottom/top side.



(a) Geometry.



(b) Boundary conditions.

Figure 5.26 Geometry (not to scale) and BCs of the first test case for the cylinder flow. The arrows indicate which node contributes to the extrapolation.

In the second test case for cylinder flow, a cylindrical obstacle was put in a square domain as depicted in Fig. 5.27a. The cylinder of diameter D was mounted centrally in a square domain of length $L=32D$. The Reynolds number was based on Eq. 5.25. All the applied BC for this test case are depicted in Fig. 5.27b. By using the Zou and He velocity BC a uniform velocity was applied on the left side of the domain to produce a flow around the cylinder. The BC top and bottom sides were set as symmetry and a full-way bounce-back

applied on the cylinder. For the right outlet of the domain, the Zou and He pressure BC was applied using $\rho_0 = 1$ and $u_x = 0$. The convex-corner BC was applied on all four corners. For the two corner nodes at the inlet side the y -direction velocity was set to zero and the x -direction velocity and density extrapolated from a node on the bottom/top side. For the two corner nodes on the outlet the y -direction velocity was set to zero and the density set to the inlet density $\rho_0 = 1$, but the x -direction velocity was extrapolated from a node on the bottom/top side.

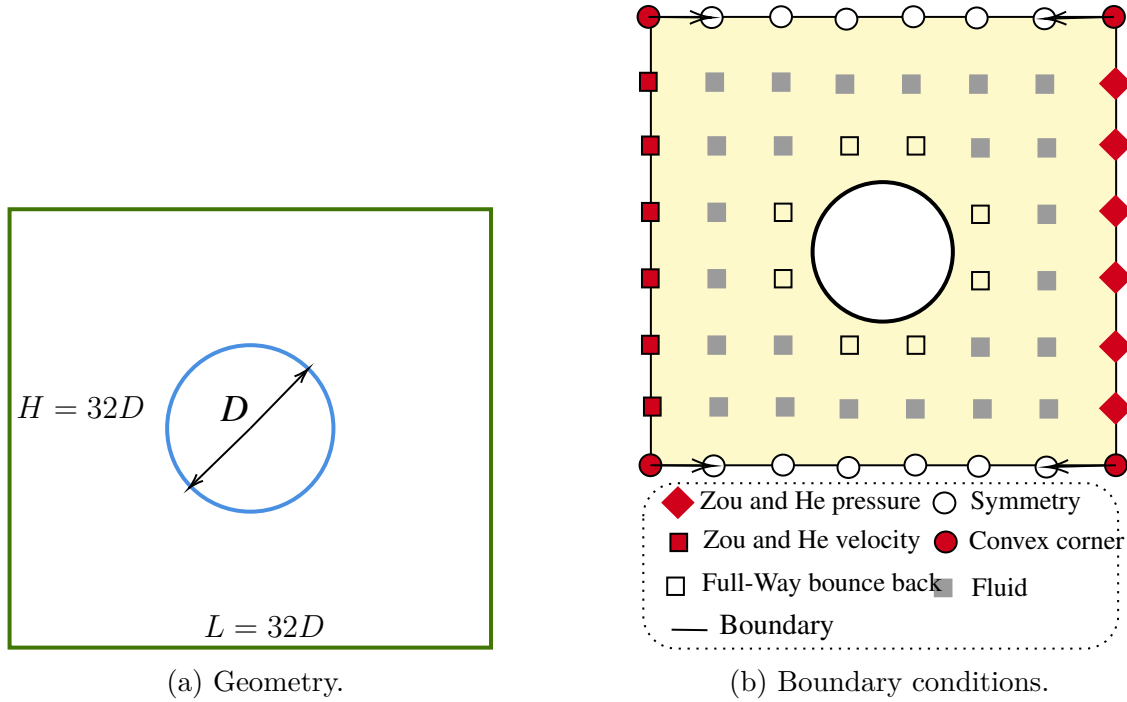


Figure 5.27 Geometry (not to scale) and BCs of the second test case for the cylinder flow. The arrows indicate which node contributes to extrapolation.

Steady state flow

To study the steady state flow around the cylinder only the first geometry and BCs depicted in Fig. 5.26 were used. In this test case, Reynolds numbers $Re=10, 20, 30$, and 40 were chosen to perform the simulations in the steady flow regime. As can be seen in Fig. 5.28a for $Re=10$ the flow is separated and there are two vertices behind the cylinder. However, for $Re=40$ there is an increase in the size of these vertices as illustrated in Fig. 5.28b. The location of vortices behind the cylinder was used as a quantitative comparison as presented in Table 5.4. The results show an acceptable agreement to the results published in Ref. [132].

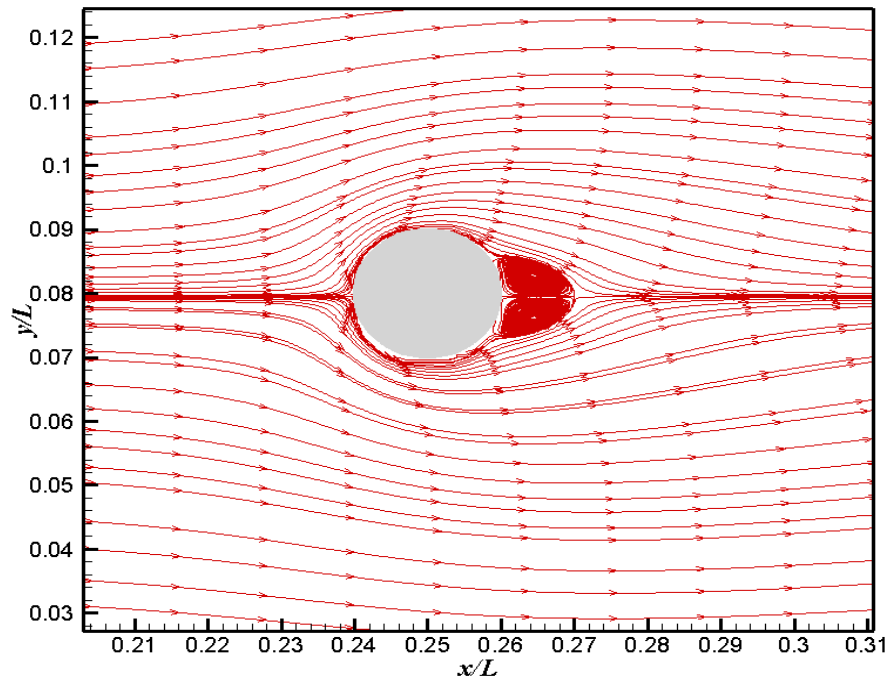
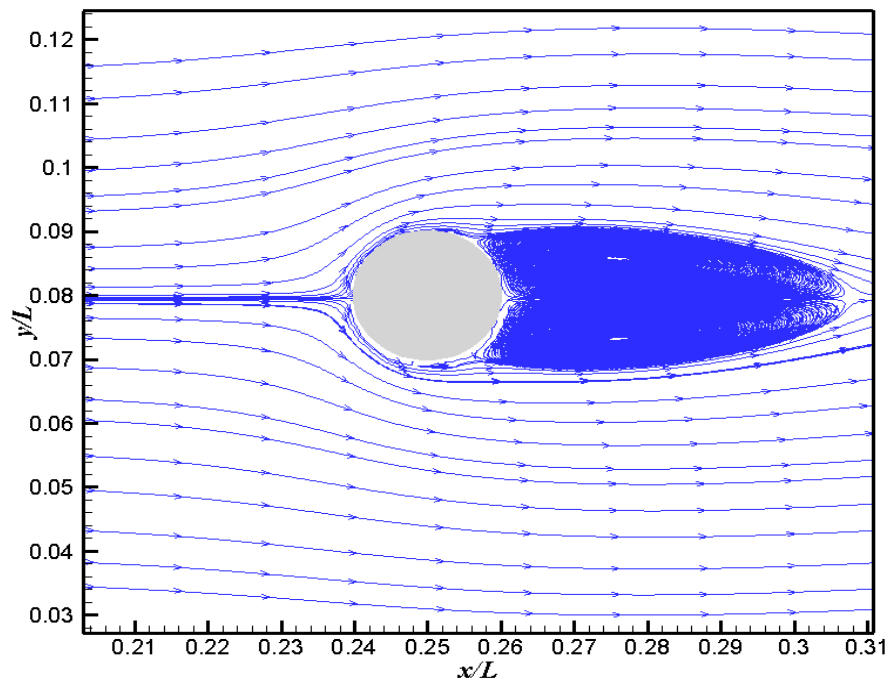
(a) $Re=10$.(b) $Re=40$.

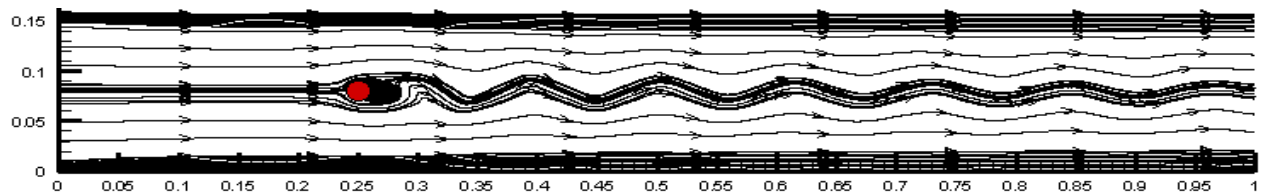
Figure 5.28 Streamline patterns for steady flows past a circular cylinder for different Reynolds numbers and a grid size of 1000×160 .

Table 5.4 Comparison of the location of vortex in the wake of the cylinder for different Reynolds numbers.

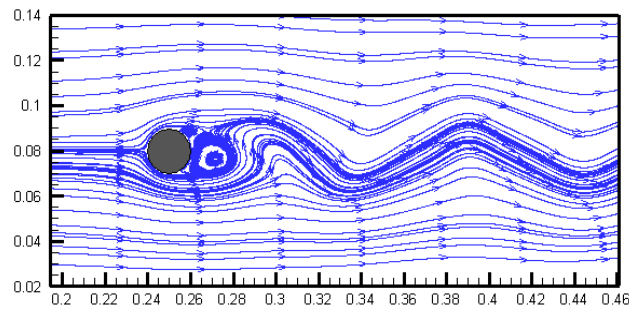
Re	grid size	proposed work	Ref. [132]
10	1000×160	0.263	0.264
20	1000×160	0.266	0.266
30	1000×160	0.274	0.274
40	1000×160	0.275	0.276

Unsteady state flow

The first geometry and BCs illustrated in Fig. 5.26 were used to study an unsteady state flow field around the cylinder using a grid of size 2000×320 . As seen in Fig. 5.29 the vortex shedding was behind the cylinder and the flow unsteady. As the top and bottom sides of the rectangular grid are no-slip wall, a boundary layer forms in these regions. Also, the stream lines at the inlet of the domain show a uniform velocity field. In addition, the lift and drag coefficients were measured and presented in Table 5.5. As the aerodynamic forces are oscillatory for unsteady cylinder flow the mean value of the drag coefficient is compared to Ref. [132] while the amplitude was used for the lift coefficient.



(a) Far view.



(b) Close view.

Figure 5.29 Streamline patterns for unsteady flows past a circular cylinder for $Re=100$ and grid size 2000×320 .

Table 5.5 Comparison of the lift and drag coefficient at $Re=100$. A grid of size 2000×320 was used for these simulations.

Authors	$C_d(\text{Mean})$	C_l
Perumal et al. [132]	1.290	± 0.289
Proposed work	1.28	± 0.284

The second geometry and BCs illustrated in Fig. 5.27 were used in the second part of the unsteady state flow investigation around the cylinder. The main difference from the first test is that in this second test a density is imposed on the outlet, and symmetry BCs are imposed on the top and bottom. As seen in Fig. 5.30 vortex shedding is behind the cylinder and the flow is unsteady. Also the stream lines at the inlet of the domain were uniform. In addition, the lift and drag coefficients were measured and are presented in Table 5.6. Here the mean value and drag amplitude coefficient are compared to Ref. [57]. Only the amplitude of the lift coefficient is compared as the mean value of lift is zero. Compared to Nicolas et al. [57] the result is less accurate as first order staircase bounce back was used instead of the second order BC that Nicolas et al. [57] used.

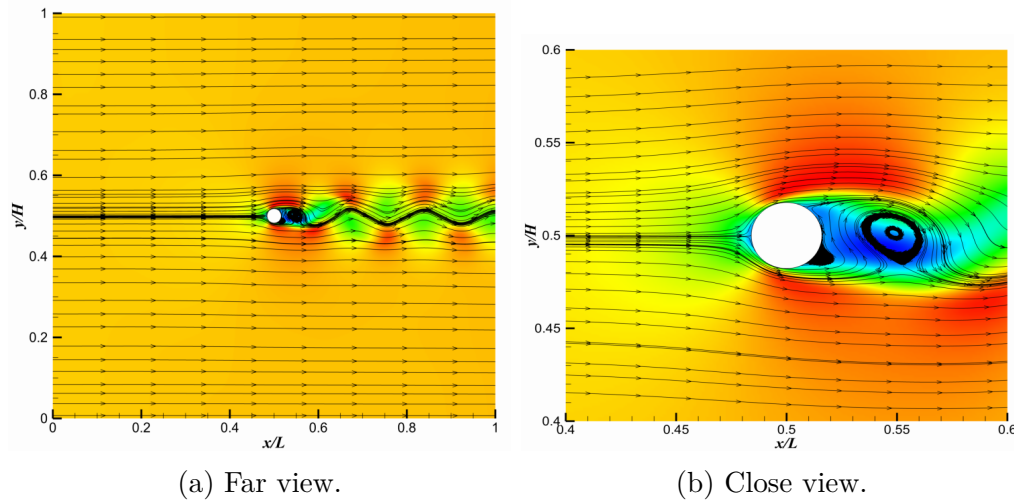


Figure 5.30 Streamlines for unsteady flow past a circular cylinder at $Re=100$ and grid size 768×768 . The geometry and the BCs shown in Fig. 5.27 were used for this simulation.

Table 5.6 Comparison of the lift and drag coefficient at $Re=100$.

Authors	$C_d(\text{Mean})$	$C_d(\text{Amplitude})$	C_l
Nicolas et al. (768×768) [57]	1.330	0.010	± 0.324
Proposed work (768×768)	1.302	0.012	± 0.304
Proposed work (1536×1536)	1.345	0.013	± 0.310

In this section, a complex flow around a cylinder was carried out for a range of Reynolds numbers. By applying full-way bounce-back on the cylinder, the implementation of this boundary condition for curved boundaries was tested using a staircase scheme. As the staircase boundary includes edge boundaries, and convex and concave corners, the type of nodes were detected automatically and appropriate BC applied (for more details about implementation refer to Section 5.8.9). In addition, indicate that the implementation of symmetry and open boundary BC are correct. It should mention like other BCs, symmetry, Zou and He pressure, and open boundary BCs are implemented just for the left side, but they are applied on different boundaries.

5.6 Conclusion

In this paper, a new framework for LBM boundary condition implementation was developed and all aspects explained in detail. In this framework, a BC is implemented for only one side/corner of a domain and then applied to the other sides/corners. So, as the usual formulations of a BC is dependent on the orientation of the boundary nodes, using this framework reduces the number of implementations, making it easier to implement typical LBM BCs. One of the core contributions of this work is the concept of applying the circular shift and flip operation over an AoS data layout to map the PDFs of any edge/corner to the left/bottom-left boundary node. To verify the framework, several of the common BCs in the LBM were implemented, then verified using well-known benchmarks in the LBM field.

Extension of this framework to three-dimensional flows would be interesting and valuable future work. For example, to develop a 3D framework the first step is to classify the boundary nodes and develop boundary conditions for each type. There are five types of boundary nodes including on-face nodes, convex edges, concave edges, convex corners, and concave corners. A formulation for on-site velocity boundary conditions are described in Ref. [43]. However, special treatments are required for three-dimensional flow because singularity can occur for some boundaries. Also, it is possible to use this framework for any other lattice stencil in 2D, e.g. *D2Q5*, but the appropriate formulation of BCs should be implemented.

5.7 Acknowledgments

We acknowledge the support of the Natural Sciences and Engineering Council of Canada (NSERC), [funding reference number *RGPIN-2020-04512*].

5.8 Appendix: Common boundary conditions for the LBM

In this appendix, we present the formulation of the implemented boundary conditions using the proposed framework. The aim of this section is not to investigate the boundary conditions but review their drawbacks and advantages. Here we derive the formulations for implementation in the framework. Hence, for edge boundaries, the formulation for the left boundary is derived, while for the corners, only the bottom-left corner is considered. A considerable amount of valuable time is saved as there no need to derive the formula for the other sides/corners.

5.8.1 Zou and He pressure (edge)

This BC is applied when the pressure (or density) is defined at a boundary [40]. It is based on the bounce-back rule of the non-equilibrium part of the PDFs. In the formulations explained in this paper, it was assumed the boundary is aligned with the coordinate axis, *i.e.* $u_{||} = 0$.

Based on Fig. 5.31, after streaming, besides the normal velocity, there are three unknown PDFs. There are mass and momentum equations in two directions (*i.e.* Eq. 5.8 and 5.9 respectively), which makes three equations. However, there are four unknowns (f_2, f_3, f_4, u_x). So, to close the system of equations, we use the bounce-back rule of the non-equilibrium part of the PDFs (*e.g.* for the left boundary and *D2Q9* lattice stencil Eq. 5.26):

$$f_7 - f_7^{eq} = f_3 - f_3^{eq} \quad (5.26)$$

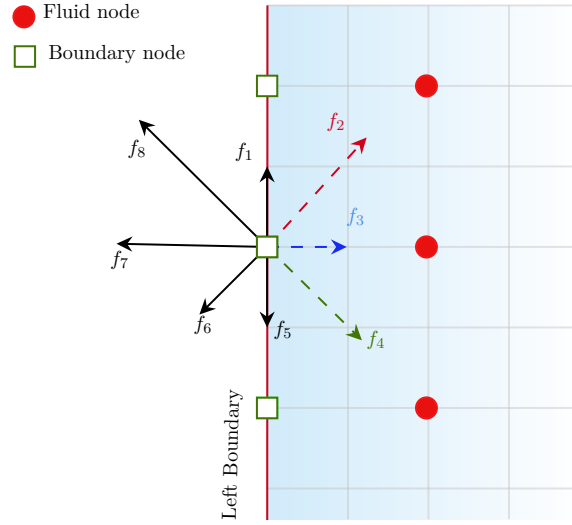


Figure 5.31 The unknown PDFs after streaming: f_2, f_3, f_4 with dash lines are specified by the Zou and He BC.

After some algebraic operations the final form of the unknowns are as follows:

$$\begin{aligned}
 u_x &= 1 - \frac{f_0 + f_1 + f_5 + 2(f_6 + f_7 + f_8)}{\rho} \\
 f_3 &= f_7 + \frac{2\rho u_x}{3} \\
 f_2 &= f_6 + \frac{f_5 - f_1}{2} + \frac{\rho u_x}{6} \\
 f_4 &= f_8 - \frac{f_5 - f_1}{2} + \frac{\rho u_x}{6}
 \end{aligned} \tag{5.27}$$

5.8.2 Zou and He velocity (edge)

This boundary condition is the same as Zou and He pressure but, instead of a predefined pressure, the velocity components at the boundary [40] are known. However, the density and the incoming PDFs to the domain are unknown but can be solved similarly to the procedure for the Zou and He pressure. So, the final form of unknowns for the left side of a rectangular

domain are as follows:

$$\begin{aligned}
 \rho &= \frac{f_0 + f_1 + f_5 + 2(f_6 + f_7 + f_8)}{1 - u_x} \\
 f_3 &= f_7 + \frac{2}{3}\rho u_x \\
 f_2 &= f_6 + \frac{f_5 - f_1}{2} + \frac{1}{2}\rho u_y + \frac{1}{6}\rho u_x \\
 f_4 &= f_8 - \frac{f_5 - f_1}{2} - \frac{1}{2}\rho u_y + \frac{1}{6}\rho u_x
 \end{aligned} \tag{5.28}$$

5.8.3 Moment-method

The moment-method BC introduced by A. Hantsch *et al.* is an extension of the work by Noble *et al.* [30] and Bennett [3]. T. Reis *et al.* [141] then studied the accuracy of moment-method boundary conditions for no-slip walls using the dipole-wall collision benchmark. This method is a general methodology for imposing macroscopic boundary conditions in the LBM. At a flat boundary aligned with grid nodes, the hydrodynamic moments of density, momentum, and momentum flux can be grouped according to the combinations of the incoming unknown PDFs for any boundary. The unknown PDFs can then be specified by solving a system of equations. The proposed framework is ideal for this type of BC.

As the example in Ref. [141] shows, for a stationary and no-slip wall, by choosing three independent equations from the set of equations, unknown PDFs for left side boundary can be specified as in Eq. 5.29:

$$\begin{cases} \rho = f_0 + f_1 + f_5 + 2(f_6 + f_7 + f_8) \\ f_3 = f_5 + f_1 + f_7 + 2(f_6 + f_8) - \frac{1}{3}\rho \\ f_4 = \frac{1}{6}\rho - f_5 - f_6 \\ f_2 = \frac{1}{6}\rho - f_1 - f_8 \end{cases} \tag{5.29}$$

5.8.4 Periodic (edge)

Periodic BC means the fluid leaving the domain on one side will, instantaneously, re-enter at the opposite side. For these kinds of flow it is possible to apply periodic boundary conditions [9]. For the NS equation periodic flow conditions along a single dimension are applied as:

$$\rho(\vec{x}, t) = \rho(\vec{x} + \vec{L}, t) \tag{5.30}$$

$$\rho \vec{u}(\vec{x}, t) = \rho \vec{u}(\vec{x} + \vec{L}, t) \quad (5.31)$$

where the vector $\vec{L}$ describes the periodicity direction and length of the flow pattern.

Implementation of a periodic boundary condition is straightforward [9, 41] as the unknown PDFs of one side are given by those leaving the domain at the opposite side as depicted in Fig. 5.32. For this particular implementation, the boundary is passing directly over the nodes.

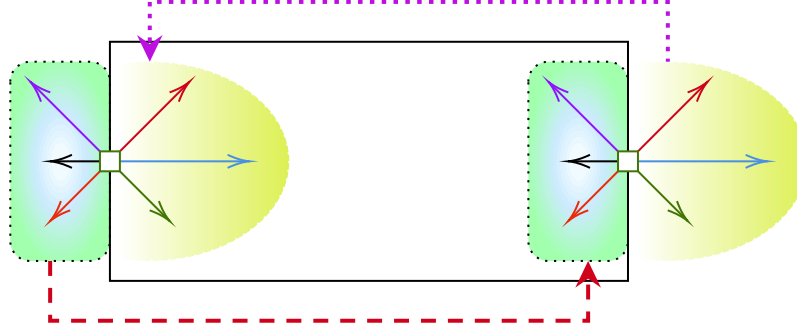


Figure 5.32 Periodic boundary condition along the x -axis in the LBM.

The user needs to specify the two boundaries with a periodic BC. At the implementation level, they are detected by the framework, then the index of the node that the unknown PDFs are taken from is stored at each node belonging to these boundaries. All these procedures are carried out in the pre-processing step. In the process of applying BC by traversing any of the regions with periodic BC, the first step is to retrieve the index of its corresponding node and then acquire the unknown PDFs from the known PDFs of that node.

5.8.5 Symmetry (edge)

A flow field may have a symmetry line or plane. In this case, the symmetry BC can be used to reduce the size of the computational domain. As illustrated in Fig. 5.33, we see how it is possible to reduce the number of required nodes to simulate the channel flow. So, as the flow on one side of the symmetry boundary is the mirror of the other, the unknown PDFs can be replaced by known ones [41]. For example in Fig. 5.33 the unknown PDFs are specified as follows:

$$\begin{aligned} f_2 &\leftarrow f_8 \\ f_3 &\leftarrow f_7 \\ f_4 &\leftarrow f_6 \end{aligned} \quad (5.32)$$

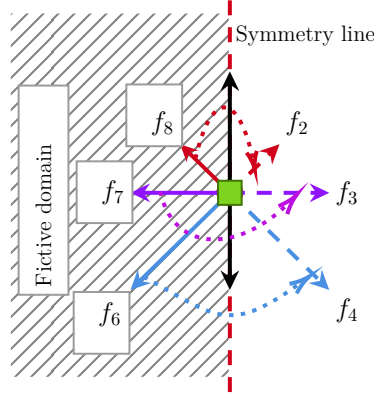


Figure 5.33 The symmetry line for a vertical channel flow and how the unknown PDFs can be specified.

5.8.6 Open boundary (edge)

In some cases, the pressure (or density) and/or velocity at a boundary are unknown. In this case, open BC may be used [41]. In this BC it is possible to use extrapolation from the interior nodes. As the outlet boundary is at the left side of the domain, as depicted in Fig. 5.34, the PDFs at the boundary can be obtained by using a second-order or first-order extrapolation. However, the first-order extrapolation is more stable than the second-order [41]. Here, however, a zeroth-order extrapolation is used with a "direct copy".

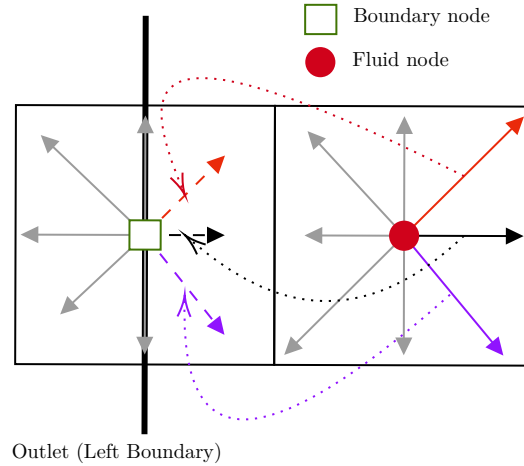


Figure 5.34 Typical extrapolation of PDFs with the open boundary BC. The boundary is on left side. Here, for simplicity, a zeroth-order extrapolation is illustrated using a direct copy of the PDFs from the adjacent node.

5.8.7 Moment-method corner

As explained in Ref. [141], to apply the moment-method to convex corners and to simulate a no-slip wall, five equations are needed as there are five unknowns. The unknown PDFs for a typical $D2Q9$ lattice stencil are depicted in Fig. 5.35. So, in this method, five equations from the existing general system of equations are chosen. Since in this paper all the BCs for corners are derived for bottom-left, this BC formulation is derived as follows:

$$\begin{cases} \rho = f_0 + 2(f_5 + f_7) + 4f_6 \\ f_1 = \frac{2}{3}\rho - f_0 - f_5 \\ f_2 = -\frac{2}{3}\rho - f_0 - f_7 + f_5 + f_6 \\ f_3 = \frac{2}{3}\rho - f_0 - f_7 \\ f_4 = -\frac{1}{6}\rho - f_5 - f_6 \\ f_8 = \frac{1}{6}\rho - f_7 - f_6 \end{cases} \quad (5.33)$$

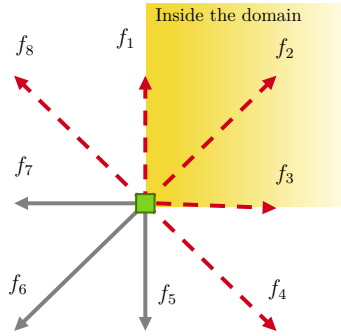


Figure 5.35 The unknown PDFs for the bottom-left corner for a typical $D2Q9$ lattice stencil.

For those interested, the details of the derivation of this BC can be found in Ref. [141].

5.8.8 Convex-corner

As described in Section 5.8.7 for a convex corner, there are five unknowns. Hence there should be five equations for specifying the unknowns to close the system of equations. To develop a general form of BC for convex corners, it is possible to use three equations of mass and momentum in combination with the two equations of bounce-back rule in x and y direction. To satisfy the mass and momentum equations, it is assumed the macroscopic values, *i.e.* the density and velocities, are known at the corner node. In the case where they are not given, an extrapolation from a fluid node can be used to specify them. In addition, it is assumed the bounce-back rule is still correct for the non-equilibrium part of the PDFs normal to the

boundary. In summary, by solving the system of equations, the unknown PDFs are specified as follows:

$$\begin{cases} f_1 = f_5 + \frac{2}{3}\rho v \\ f_2 = f_6 + \frac{1}{6}\rho(u + v) \\ f_3 = f_7 + \frac{2}{3}\rho u \\ f_8 = -\frac{1}{2}(\rho(u - 1) + \frac{2}{3}\rho v + f_0 + 2f_5 + 2f_6 + 2f_7) \\ f_4 = -\frac{1}{2}(\rho(v - 1) + \frac{2}{3}\rho u + f_0 + 2f_5 + 2f_6 + 2f_7) \end{cases} \quad (5.34)$$

5.8.9 Full-way bounce-back

The full-way bounce-back is one of the most interesting BC in the field of LBM. As depicted in Fig. 5.36 it is assumed the wall is located midway between the solid and boundary nodes. For this BC, we assume the LBM algorithm is such that first, the collision step is applied, and then the streaming step. Based on Fig. 5.36a, it assumed PDFs are at time t . So, if the collision and streaming step are applied on these PDFs the product will be at time $t + \Delta t$ as shown in Fig. 5.36b. At this time step, the BC are applied to the PDFs. In the process of applying boundary conditions, the post-streaming PDFs will be reversed as depicted in Fig. 5.36c. Finally, in the next time step, *i.e.* $t + \Delta t$, the PDFs belonging to the solid nodes stream to the boundary nodes 5.36d. Notice that the collision step should not be applied to the PDFs of solid (ghost) nodes. Hence, the magnitude of PDF depicted in Fig. 5.36d are not changed.

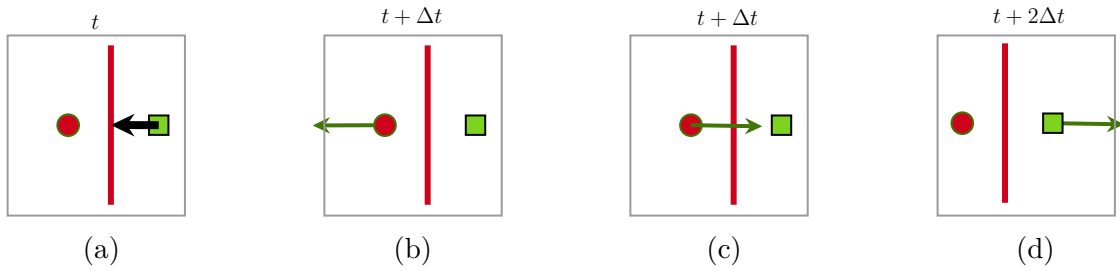


Figure 5.36 The process of full-way bounce-back: the grey shaded domain is the solid (ghost) region and the line represents the wall boundary. (a) pre-collision PDF at t , (b) post-streaming PDF at $t + \Delta t$, (c) reversing the PDF as the full-way bounce-back BC at $t + \Delta t$, (d) post streaming PDF at $t + 2\Delta t$.

As explained in Fig. 5.36, the most common implementation in the literature is a set of solid nodes which are used as ghost nodes. In this scheme, the collision step should not be applied

to these nodes. The problem of this approach is the lose of generality of the collision step as it cannot be applied on all the existing nodes. So, in the proposed implementation a kind of buffer is used rather than a set of ghost nodes. So, as there are no additional nodes in the domain the collision step remains general. Hence, there is no need to use a condition statement to not apply collision on some parts of nodes.

In the proposed framework, the implementation of full-way bounce-back is for the most general form (*i.e.* curve boundary). Therefore, a staircase boundary can be used as a curve boundary. However, a staircase is a combination of edge boundaries, convex and concave corners.

As mentioned previously, only three kinds of boundary nodes (*edge, concave corner, and convex corner nodes*) exist in the proposed implementation. So, in the pre-processing step, the kind of boundary nodes are specified and their orientations stored in an array. Then based on this data an appropriate BC *i.e.* full-way bounce back for edge or concave and convex corners is applied.

The most important issue in this implementation is how a buffer layer may be stored. As in Fig. 5.36, it is possible to store the PDFs depicted in Fig. 5.36b in a buffer and use them in the next time step to apply full-way bounce-back BC, as shown in Fig. 5.36d. So, at each time step, after performing collision and streaming, the PDFs belonging to the boundary nodes are copied in an array indicated by f^{m-1} and used in the next time step. Then to treat each node, based on their orientation, one of the formulations below are applied:

Edge: For those nodes belonging to a straight line the unknown PDFs are specified as follows:

$$\begin{cases} f_2 = f_6^{m-1} \\ f_3 = f_7^{m-1} \\ f_4 = f_8^{m-1} \end{cases} \quad (5.35)$$

Convex node: According to Fig. 5.37, for a convex node, there are five unknowns which can be determined by applying the full-way bounce-back as follows:

$$\begin{cases} f_1 = f_5^{m-1} \\ f_2 = f_6^{m-1} \\ f_3 = f_7^{m-1} \\ f_4 = f_8^{m-1} \\ f_8 = f_4^{m-1} \end{cases} \quad (5.36)$$

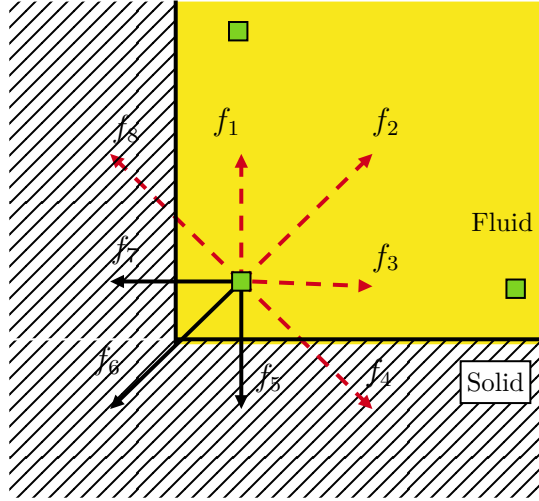


Figure 5.37 The unknown PDFs for a convex node for a typical $D2Q9$ lattice stencil.

Concave node: As shown in Fig. 5.37, for a concave node there is just one unknown which can be determined as follows:

$$f_6 = f_2^{m-1} \quad (5.37)$$

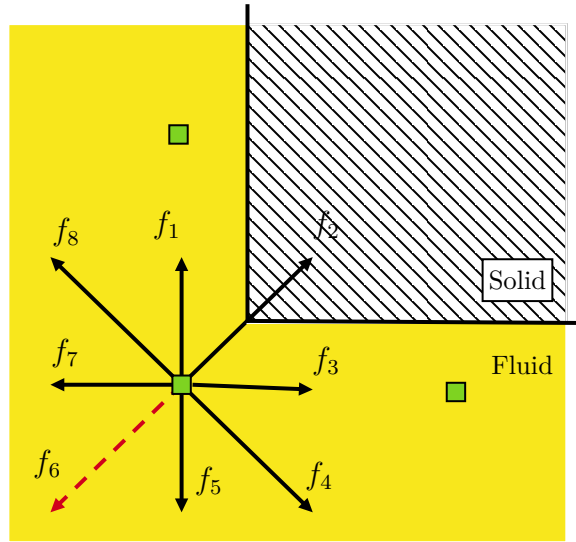


Figure 5.38 The unknown PDF for a concave node using a typical $D2Q9$ lattice stencil.

CHAPTER 6 ARTICLE 3: LABCOMA: LATTICE BOLTZMANN COMMUNICATION ALGORITHM

- M. Namvar, S. Leclaire, "LaBComA: Lattice Boltzmann Communication Algorithm", *Computer Physics Communications*, January 4, 2023 (submitted)

which introduces a new algorithm for MPI communication for the LBM computation on distributed memory systems. This new communication algorithm is independent of the data structure of the base LBM code. As it is independent of the lattice stencil it could be used in 2D and 3D LBM code. This algorithm is implemented by using C++ object-oriented programming as a framework. To extend the newly developed LBM algorithm to parallel, presented in article 1, this framework was used.

6.1 Abstract

In this study, a general MPI communication algorithm for lattice Boltzmann method (LBM) simulation is proposed. One of the criteria in the design of this algorithm is to be independent of the LBM program data structure. So, the proposed algorithm could be used to parallel any LBM program.

Also, the communication of the LBM could be hidden with the computation by using non-blocking communication. For example, the speedup is super-linear for a small number of processes. Moreover, the parallel programming parts of the implemented program could be separated from the other parts of the LBM program which results in easier code development. To verify the proposed algorithm it is implemented by using object-oriented programming in C++ and MPI library. A speedup analysis was done by using two kinds of dense and sparse systems. To use the published code in any LBM program it is required just to provide some functions to implement the abstract classes. However, the authors just tested the algorithm for a two-dimensional LBM program, but it is expected to also work for three-dimensional LBM program as the communication algorithm is independent of the lattice stencil.

6.2 Introduction

Recently, LBM has been popular as an alternative to the classical method of computational fluid dynamics [21, 24, 103, 131, 133, 134, 142]. Since the LBM demands high memory resources

[143,144] without using parallel programming, large-scale system simulations are impossible due to the lack of memory resources and long processing times. Fortunately, based on the fact that the LBM needs only nearest neighbor data, the algorithm is ideal for parallel computing.

Despite the progress in GPU computing, using CPUs as computing devices is still very popular as more memory resources are available compared to GPU memory. There are a lot of works in the LBM that uses MPI to execute a program on distributed computing systems [24, 87, 145–147]. Wang *et. al.* [85] used up to 100 processes and get a maximum speedup of around 60. Shang *et. al.* [148] used 1024 processes and achieved a speedup of about 800 using a large-scale grid including 0.864 billion nodes. The coupling of an MPI parallel LBM solver with suspended particles has been done by Puurtinen *et. al.* [149]. Xu *et. al.* [150] proposed a parallel algorithm for D3Q19. They reported an efficiency of 86% on 3072 cores. David *et. al.* [24] used a vectorized data structure and based on that data structure they simply distributed the fluid nodes evenly between processes.

To utilize heterogeneous CPU–GPU clusters, hiding the communication is critical as using GPUs could result in a significant speedup. Although in this case, GPU is the computing device, communication between CPUs is necessary. So, some combine MPI, as a means of transferring data, and GPUs for parallelization to use all advantages of distributed memory and GPUs capabilities [97, 97, 98, 151].

There are some open source projects for LBM simulation such as OpenLB [100], TheLMA [101], Palabos [102], HemeLB [103], and waLBerla [104]. These projects are investigated briefly in terms of the parallel programming aspect.

OpenLB is using a hybrid parallelization concept, *i.e.* MPI+openMP [89]. The domain decomposition approach used in OpenLB is based on the model used in its LBM data structure. Consequently, the communication strategy of this framework is tightly coupled to the data structure of the LBM algorithm.

TheLMA framework stands for Thermal LBM on Many-core Architectures which initially used the CUDA platform for multi-GPU parallelization [101, 105], but they mentioned that for more flexibility in domain decomposition using MPI is required for communication. So, Obrecht *et al.* [106] extended the framework to use MPI for communicating data.

Palabos is an LBM library based on MPI for parallel executions. In Palabos, a kind of data structure is used which divides the computational domain into some blocks, named semi-direct addressing. According to this data structure, one or multiple ghost layers around each block component are used to handle the MPI communication [102].

HemeLB [103] is developed particularly to simulate fluid flow in complex geometries. So, it is

using an indirect addressing scheme in the data structure. Although it is possible to use any domain decomposition method such as parMetis [107], it has its library for decomposition named ALL (A load-balancing Library) [108]. In the MPI part of the HemeLB, only the data of the nearest neighbor nodes are communicated during the streaming step. These nodes are labeled as "domain-edge" in the preprocessing step. Then in each time step, first the collision is done for the domain-edge nodes. After that, the data of these nodes are communicated. Then the collision and streaming are done for the rest of the nodes. In this way, the communication [2] is hidden with computation. In Ref. [107] the existing pure MPI code is used to develop CUDA-based parallel code for multi GPUs.

The waLBerla framework uses an approach named "block forest" to manage the domain partitioning into blocks. In this framework, the communication module is organized in a layered pattern. These layers start from low-level MPI functions up to classes that implement communication strategies. Only the topmost layer is specific to the data structure of the waLBerla framework. In this approach finding the required data for communication is the developer's responsibility as this layer is dependent on the data structure of the LBM program [104]. Risso *et al.* [109] presented a scalable multi GPU of the code which is using CUDA.

As mentioned before, because of the importance of parallel programming for the LBM simulations, evaluating any new algorithm in terms of capability to be executed on distributed memory system is critical. Hence, any new proposed LBM algorithm should be developed and evaluated in terms of parallel computing. In all the above-investigated works and open-source projects, there is not any general MPI framework that could be used individually. In other words, the MPI programming part of them depends on the data structure of the LBM program they have developed. So, the primary objective of this study is to develop an MPI communication algorithm for LBM that would be independent of the base LBM program.

The authors recently developed a new LBM algorithm [152] which should be developed and evaluated in terms of parallel computing. Hence the secondary objective of this work is to use the proposed communication algorithm to parallel the algorithm in Ref. [152].

The rest of this paper is organized as follows. First, the proposed algorithm for MPI communication is explained in detail. Then, the implemented framework is described to show how to use it in any LBM program. After that, a case study is done by parallelization of a serial LBM algorithm [152] using this framework, and the results are presented for two test cases of dense and sparse geometries. Finally, a conclusion is provided to sum up this work. In addition, the procedure of using the presented communication algorithm to parallel the mentioned new LBM algorithm [152] is also available in Appendix B.

6.3 Proposed communication algorithm

In this section, at first, the requirements which are used to develop an algorithm for MPI communication are explained. Then, to find an answer to each of these requirements, the authors try to find a solution in the literature or propose a solution for them. After that, the main idea of the algorithm is presented. Then the proposed algorithm is explained in detail.

To have a short description of the LBM simulation on distributed memory systems, suppose there is an "initial grid" which a set of processes should perform the computation of this initial grid. So, as the first step, it is required to decompose this initial grid into some partitions. To do the decomposition, at the implementation level, an index is assigned to all the nodes belonging to each partition. These indexes are called "part index" indicating a node belongs to which partition of the decomposed grid. Then each process takes a partition of the decomposed grid to do the simulation. So, each process has a local LBM data structure. In the LBM simulation to do the streaming step the PDFs belonging to the neighboring nodes are required. So, for those nodes belonging to the interfaces of the decomposed grid, some of the PDFs should be communicated.

The following requirements are used to develop an efficient communication algorithm to extend an LBM serial program to parallel:

1. Exploiting an efficient domain decomposition method for balancing the computational load among the processes and also minimizing the amount of data that should be communicated.
2. Applying an efficient communication pattern to minimize the number of Probability Distribution Functions (PDFs) which are needed to be communicated in order to perform the LBM simulation.
3. Hiding the parallel parts of the LBM algorithm from the serial parts, to keep the algorithm's implementation as simple as possible. That means the serial version of any algorithm should remain unchanged when they are extended to parallel.
4. The communication algorithm should be independent of the LBM data structure. That is because, at the implementation level, parallel programming is non-trivial and time-consuming, so it requires a reusable and maintainable software design concept.

The objective of this study is to find a solution to satisfy all the mentioned criteria as following:

Req. 1 To satisfy requirement 1, using the external libraries for domain decomposition is the strategy that is followed in this study. Hence, the domain decomposition procedure is separated from the communication algorithm. So, to give flexibility to the end user, the design of the algorithm is in such a way that it is possible to use any desired grid decomposition approach.

Req. 2 Assume a computational domain is decomposed and each subdomain is assigned to a process. In order to do an LBM simulation on this decomposed domain by using a distributed memory system, exchanging PDFs data between processes are the only communication overhead. In general, there are two approaches regarding the exchange of PDFs data between processes. In the first approach, the whole PDFs belonging to the nodes at the interfaces of subdomains are communicated. This method is the simplest approach, but its communication overhead is not efficient as some unnecessary data participate in the communication. In the second approach, just those PDFs which are required to perform the streaming step are communicated. In this study, the second approach is used. Another issue regarding PDF communication is the type of PDFs to be communicated between processes. Based on the algorithm of the LBM, there are two types of PDFs during the execution of the LBM algorithm called post-collision (pre-streaming) and post-streaming (pre-collision) PDFs. The question is whether the post-collision PDFs should be communicated or the post-streaming PDFs. Suppose the post-streaming PDFs are communicated. In this case, the macroscopic values of the associated nodes belonging to these PDFs should be communicated to calculate the post-collision PDFs at the destination process to perform the collision. So, in this case, in addition to the PDFs, the macroscopic values should be communicated as well which means more communication overhead. On the other side, communicating the post-collision PDFs would be more efficient in terms of communication overhead, which is followed in this study.

Req. 3 There is a dominant approach to use one or multiple ghost layers around each subdomain to handle the communication and computation of those PDFs that should be communicated across processes. This approach is followed in many studies [153–156]. Another idea is to use the data structure of the LBM algorithm and extend it to handle the communicating PDFs. In the proposed communication algorithm, a local data structure is used to separate the communication algorithm from the LBM algorithm. So, at the implementation level, it is possible to hide the parallel programming from the serial parts to satisfy requirement 3.

Req. 4 To satisfy requirement 4, the design of the communication algorithm is in such a way that its data structure is independent of the LBM algorithm data structure. And also, the communication algorithm is independent of the lattice stencil, e.g. $D2Q9$ or $D3Q19$, used in many LBM algorithms.

To explain the main idea of the proposed communication algorithm, suppose an initial grid is decomposed into two parts Ω_1 and Ω_2 . To perform an LBM simulation by using this decomposed grid, as illustrated in Fig. 6.1, Ω_1 and Ω_2 are assigned to process 1 (P_1) and process 2 (P_2). So, to do the LBM computation by P_1 it is required that P_1 get the "receive PDFs" from P_2 . On the other side, P_2 should send some PDFs indicated as "send PDFs" to P_1 . As one of the most confusing parts of the parallel programming for the LBM simulation is specifying these PDF indexes. In this study, a general algorithm is proposed for this issue. Then in each time marching loop of the LBM simulation, the "send PDFs" are communicated to perform the LBM computation. It should be noted that the "PDF index" means the number of a cell containing a PDF value in a data structure. For example, a PDF with an index equal to 10 means the value stored in the 10th of an array storing the PDF values.

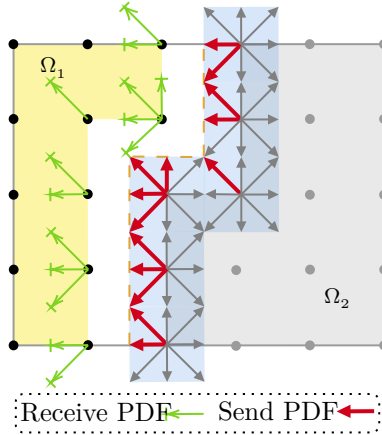


Figure 6.1 To perform the communication the send PDFs in Ω_2 should be sent to the P_1 , and the P_1 should receive them. Then the P_1 should write them to the proper location, *i.e.* receive PDF indexes, in the data structure of the base LBM program to perform the streaming step.

In the following sections, the proposed algorithm which is composed of two steps is presented in detail. In the first step, the index of PDFs participating in communication is specified, as shown in Fig. 6.1,. That step is done in the pre-processing step which is explained in detail in Section 6.3.1. Then the mentioned PDFs are used to communicate the associated PDF values as explained in section 6.3.2.

6.3.1 Specifying send/receive PDFs

Based on Algorithm 3, the first step to specifying the send/receive PDFs is finding the ghost nodes. The ghost nodes are used to find the index of ghost PDFs in the next step. Then in line 3 of Algorithm 3, a data structure for receiving PDFs is created by specifying receive PDF indexes. Finally, a data structure is formed to store the sending PDFs. Also, in the rest of this work, any word in *italics* and **bold** represents a data structure and any word in *italic* is an array.

Algorithm 3: How to specify send and receive PDFs.

- 1 Finding ghost nodes
 - 2 Finding ghost PDFs
 - 3 Specify receive PDF indexes
 - 4 Specify send PDF indexes
-

Finding ghost nodes

As shown in Fig. 6.2, if a $D2Q9$ lattice stencil is used, the PDFs data of 8 nearest neighbors are needed to do the LBM computation of the node depicted by a square. Among these 8 nodes, those which belong to other parts (shown as a diamond shape in Fig. 6.2) are labeled as ghost nodes for Ω_i . So, in this step, the global indexes of all the ghost nodes for each part (partition) are specified and stored in an array. Global index means the index of nodes in the initial grid.

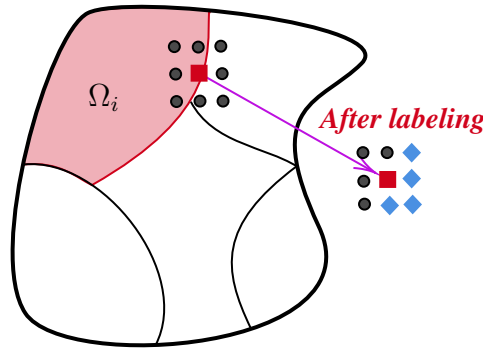


Figure 6.2 Specifying ghost nodes by labeling them based on a typical $D2Q9$ stencil. Diamond shape nodes are ghost nodes for Ω_i .

As mentioned before, the global index of the ghost nodes should be stored. Later these indexes are used to retrieve the part index of a node. Consequently, it is required to store

the part index of any node of the initial grid, which means a big array is needed to store these data. Hence, to avoid allocating any array to store the attribute of the initial grid, just the part index of the ghost nodes is stored. In addition, the part index of the neighboring nodes of any ghost nodes is stored as well. Note that each part of the decomposed grid has its ghost nodes array.

Finding ghost PDFs

By definition, "ghost PDFs" of a part are those that belong to ghost nodes and, according to the lattice stencil, point to one of the nodes of the part. For instance, in Fig. 6.3 those PDFs, belonging to Ω_2 , that pointing to one of the nodes of Ω_1 are considered as ghost PDFs for Ω_1 .

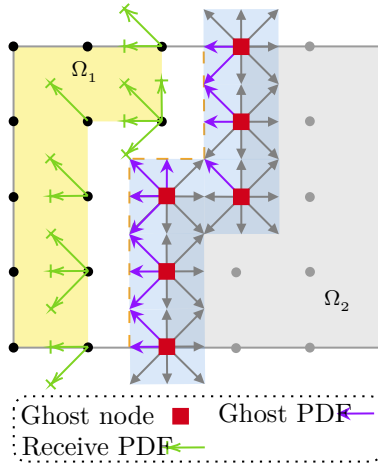


Figure 6.3 Ghost PDFs of Ω_1 are those that belong to a ghost node and pointing to the Ω_1 . Here, just the ghost PDFs of Ω_1 are shown. P_1 and P_2 represent the process 1 and 2 respectively which responsible to do the computation of Ω_1 and Ω_2 .

It should be noted that the ghost PDFs belong to other parts and they should be requested from other parts to send them. For instance, in Fig. 6.3, P_1 , which is responsible to do the LBM computation of Ω_1 , doesn't know the index of ghost PDFs as they belong to the data structure of P_2 . So, a data structure is created to store the following data for each ghost PDF:

- *iVel*: velocity index of PDF according to the lattice stencil.
- *iNode*: global index of the node that the PDF belongs to.
- *iPart*: part index of the *iNode*.

It is required to sort ghost PDFs based on the indexes of the part, *i.e.* $iPart$, that these PDFs belong to. This should be done to have efficient communication since by putting data contiguously it would be possible to send/receive a chunk of data which consequently results in fewer MPI send/receive function invokings [157]. So, after sorting them, the first chunk of the data structure shown in Fig. 6.4, which contains the data of ghost PDFs of a part, stores the data of PDFs belonging to part 1 then part 2, and so on. Note that each process that is responsible to do the computation of a part has its own ghost PDF data structure.

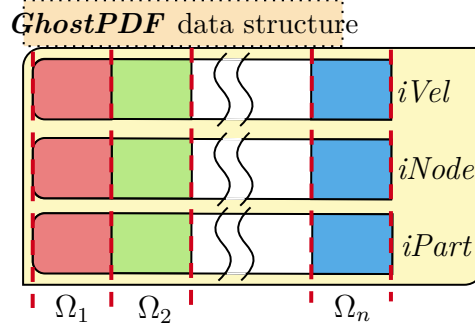


Figure 6.4 Ghost PDF data structure (***GhostPDF***). The first chunk of this data structure stores the PDFs belonging to part 1, then those of part 2, and so on.

Specifying Receive PDF indexes

According to Fig. 6.1, to perform the streaming step of Ω_1 , the ghost PDFs should be received from P_1 . Then the value of these PDFs should be written to the LBM data structure of Ω_1 . Note that, according to Fig. 6.3, the next neighbor PDF to a "ghost PDF" is a "receive PDF". In this section, the indexes of receive PDFs are specified.

Based on the data structure of ghost PDFs shown in Fig. 6.4, it is possible to specify the index of the said next PDF, *i.e.* the receive PDFs, in the data structure of each part of the decomposed grid. It should be considered that the ghost PDFs data structure shown in Fig. 6.4 are global *i.e.* they belong to the initial grid. But, the receive PDF indexes should be local which means they belong to a part of the decomposed grid. So, by using the ghost PDF data structure, *i.e.* ***GhostPDF***, the indexes of receive PDFs are specified according to the data structure of the LBM. In summary, according to Fig. 6.5 two arrays are reserved to constitute the data structure of receive PDFs, *i.e.* ***RecvPDF***:

- $nRecv$: indicates the number of PDFs that should be received from each of the other processes. The size of this array is equal to the number of processes.

- *iRecv*: stores the index of receive PDFs according to the data structure of the LBM program.

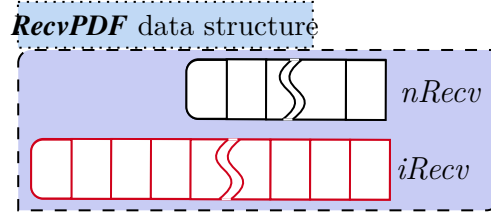


Figure 6.5 Receive PDF data structure (***RecvPDF***) is used to receive and write receive PDFs into the data structure of the LBM program.

As mentioned before, the ghost PDFs are sorted in the ***GhostPDF*** data structure. So, when the receive PDFs, *i.e.* ***RecvPDF***, are extracted from ghost PDFs they are sorted accordingly. Hence, based on Fig. 6.5, the number of receive PDFs stored in the first cell of *nRecv* array indicates the number of PDFs which should be received from part 1, then part 2, and so on. In the same way, those PDFs which should be received from part 1 are stored in *iRecv* at first, then those from part 2, and so on. It should be mentioned that *iRecv* indexes are used to write the values of receive PDFs into the data structure of the LBM program.

Specifying Send PDF indexes

As mentioned before, all the required data to specify the index of a PDF are stored in the data structure of ghost PDFs *i.e.* ***GhostPDF*** as depicted in Fig. 6.4. These data are sent to the other processes. Then the local indexes of these PDFs are specified in the destination processes. Finally, the index of these PDFs is stored in a data structure named ***SendPDF***. As depicted in Fig. 6.6 there are two arrays constituting the send PDFs data structure:

- *nSend*: indicates the number of PDFs needed to be sent to other parts. The size of this array is equal to the number of parts.
- *iSend*: stores the indexes of the send PDFs according to the data structure of the LBM program.

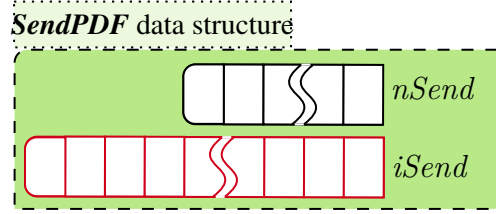


Figure 6.6 Send PDFs data structure (*SendPDF*) is used to prepare the value of required PDFs to be sent to the other parts.

In summary, at the end of the pre-processing step, each process has two data structures named *RecvPDF* and *SendPDF* shown in Figs. 6.5 and 6.6 respectively. These data structures are used for communicating the PDF data. It should be noted that the data stored for ghost nodes and ghost PDFs, explained in Section 4 and 4 respectively, are temporary and are deleted after the preprocessing step.

6.3.2 Communication algorithm

To have an efficient parallel program the communication overhead should be hidden with the computation. To do that, the communication is done behind all of the LBM computations. In this algorithm, in each time step, the post-collision PDFs are sent to their destination processes. So, it is assumed the post-streaming PDFs are available at the beginning of each time marching loop. The communication algorithm is done according to the numbered itemization within Fig. 6.7 which are explained with details in the following numbered itemization steps:

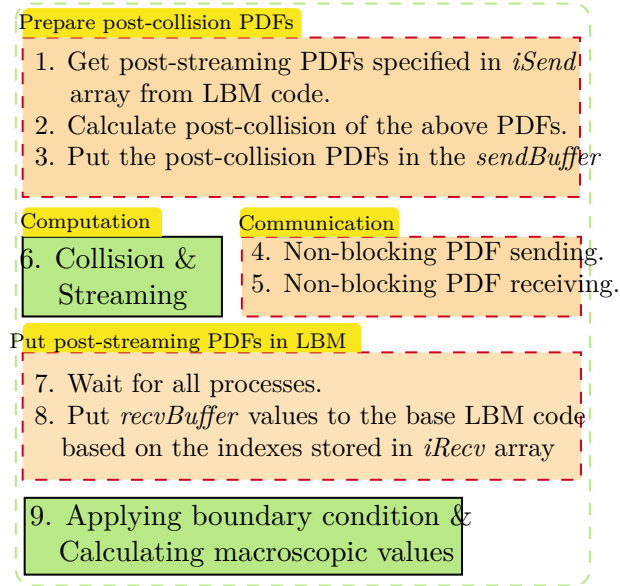


Figure 6.7 The algorithm of communicating PDFs in order to hide the communication behind the computation.

1. According to Fig. 6.6, the indexes of those PDFs which should be sent to the other processes are stored in the ***SendPDF*** data structure, *i.e.* in the *iSend* array. In this step, the post-streaming values of these PDFs are obtained from the LBM data structure.
2. Since in this communication algorithm the post-collision PDFs are sent to the other processes, the collision function is applied to the PDFs selected in step 1.
3. Since non-blocking communication is used in this algorithm, these PDFs should be stored in a buffer array named *sendBuffer*.
4. Then, each chunk of *sendBuffer* is sent to associated destinations by using a non-blocking send function of the MPI library.
5. Just after that, a non-blocking receive function is invoked to receive all the PDFs which are sent from other processes. These, PDFs are stored in another buffer array, named *recvBuffer*, to be written in the LBM program data structure later.
6. Meanwhile, after invoking the receive function of the MPI library, all the required computations *i.e.* collision and streaming steps are started without waiting to perform the communication.
7. Wait for all the processes to complete the communication.

8. After the computation, those PDFs which have been received from other processes, already stored in *recvBuffer*, should be written in the LBM program data structure. As mentioned before, based on Fig. 6.5, the indexes of PDFs which should be received from the other processes are stored in the ***RecvPDF*** data structure, *i.e.* in the *iRecv* array. So, the values of the receive PDFs are written in the data structure of the LBM program according to *iRecv* array. It should be noted that the streaming step for the PDFs stored in *iRecv* array is done implicitly as explained in Fig. 6.1.
9. Finally, the boundary condition is performed and macroscopic values are calculated.

By considering the algorithm explained in Fig. 6.7 the computation of the LBM is done in step 6 while the communication is performed in steps 4 and 5. In this way, the communication is hidden by the computation.

6.4 Class diagram of the implemented code

In this section, a brief description and instructions for using the presented algorithm are provided. It should mention for implementation, C++ object-oriented programming has been used. For those who are not familiar with UML diagrams it is recommended to find an explanation regarding these kinds of diagrams in Ref. [158].

To provide the ghost nodes data, explained in Section 4, the **GhostNode** class is designed, as shown in Fig. 6.8. This class has only one public method named *writeGhostNodesData()* which is responsible to write the ghost nodes data. At the implementation level, a C++ interface is used to link the base LBM program to this class. So, an abstract class named **LBM2GhostNode** was designed which should be implemented by the developers according to their own base LBM program.

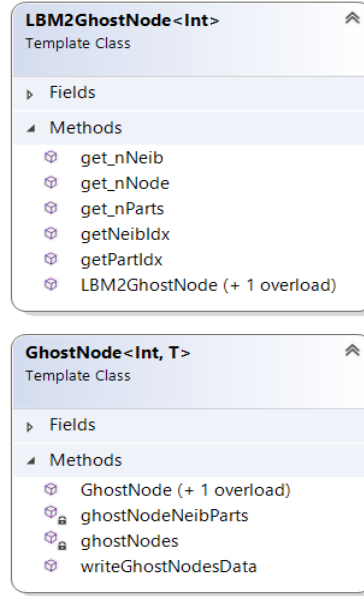


Figure 6.8 Class diagram of the implemented code associate to Section 4.

To implement the **LBM2GhostNode** abstract class, two methods should be implemented, *i.e.* *getNeibIdx()* and *getPartIdx()*. The first one returns the index of neighboring nodes in the initial grid and the second one returns the part index of a node. As these methods need the initial grid data, this part of the program should be executed just by the master process. Otherwise, each process would have its own initial grid which consumes a lot of memory.

The implementation of data structures explained in Sections 4, 4, and 4 are done in classes **GhostPDF**, **RecvPDF**, and **SendPDF** respectively, as shown in Fig. 6.9. Also, the implementation of the algorithm explained in Section 6.3.2 is done in class **ComunicatPDF** which is composed of some classes shown in Fig. 6.9. To link the base LBM program to this class a C++ interface is used. So, an abstract class named **LBM2MPI** was designed which should be implemented by the developer by considering the base LBM program. Then that implemented class should be passed to the **ComunicatPDF** class to make an object. It should be mentioned that this part of the code is executed by each of the processes. Based on this figure, **ComunicatPDF** has two public methods named *send()* and *recv()* which are doing the required computation and communication explained in Section 6.3.2.

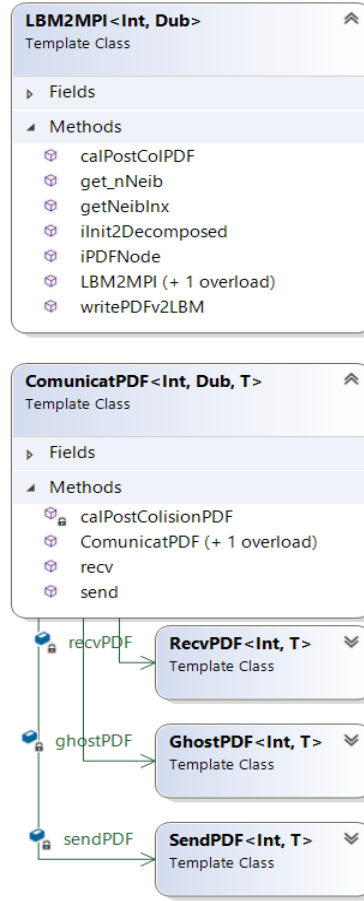


Figure 6.9 Class diagram of the proposed communication framework.

As could be seen in the class diagrams, depicted in Figs. 6.8 and 6.9, it is possible to hide all the communication algorithms from the other parts of the program. That is done by implementing **LBM2GhostNode** and **LBM2MPI** interfaces.

6.5 Results and discussion

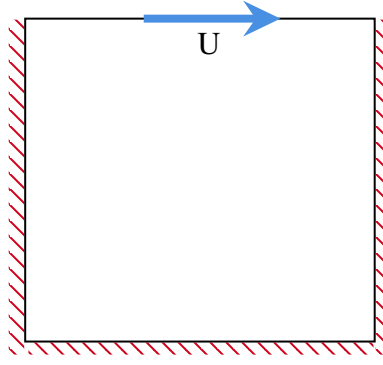
To evaluate the proposed communication algorithm, it is implemented and applied to a serial LBM program recently developed by the authors [152]. Since this work is an extension of the previous work, the serial program was found to work well. The code was verified internally by comparing the results of the parallel and serial programs. Since the results were found to be identical, they are not presented in this work. As mentioned earlier, it is possible to use the proposed communication algorithm in any LBM program without modifying said LBM program. Thus, for those who wish to know more about the use of the proposed algorithm, Appendix 6.8 explains how to parallel the serial LBM program mentioned [152].

Then, the two test cases used in this study are explained in Section 6.5.1. After that, in Section 6.5.2 the effect of grid decomposition on efficiency is studied by using block partitioning and the multi-level k-way method. Finally, the scalability of the program is evaluated using a dense and sparse system.

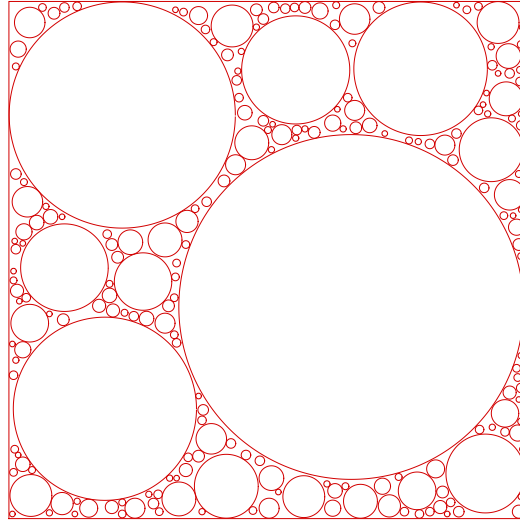
6.5.1 Test cases used for the analysis

To evaluate the proposed communication algorithm two sets of geometry have been used. In the first test case, to study a dense system a lid-driven cavity is used, as shown in Fig. 6.10a. The Zou and He velocity boundary conditions [40] were applied on each side of the geometry. For the top side, a uniform velocity was applied, and for the remaining sides, a zero velocity for both x - and y - direction. For all the corners, a convex corner BC is used [159].

In the second test case, a pseudo porous media (a set of cylinders geometry), Fig. 6.10b, is used as the sparse system. To generate this geometry a series of cylinders with random positions and radius is used to satisfy the void fraction equal to 0.1. In this test, a symmetry boundary condition was applied on both the top and bottom. A no-slip wall, with full-way bounce back, was used on the cylinders. For the left and right sides, the Zou and He boundary condition was used to apply a pressure gradient.



(a) First test case, *i.e.* the lid-driven Cavity.



(b) Second test case, *i.e.* pseudo porous media.

Figure 6.10 The geometry of the first and second test cases to investigate the performance for simulating the dense and sparse systems respectively.

All simulations are done on the Digital Research Alliance of Canada / Niagara cluster which is a large cluster of 2,024 Lenovo SD530 servers each with 40 Intel "Skylake" cores at 2.4 GHz or 40 Intel "CascadeLake" cores at 2.5 GHz. Each node of the cluster has 188 GB of RAM per node. More detailed hardware characteristics of the Niagara supercomputer can be found in Ref. [160]. However, for all the simulations in this study "Skylake" cores are used.

6.5.2 Effect of domain decomposition scheme on performance

To evaluate the effect of domain decomposition on the performance of distributed memory systems the results for two decomposition approaches, *i.e.* the blocking and multi-level k-way method, by using METIS library [113], are compared in. To do that, the decomposition

methods are applied to the lid-driven cavity. In this study as it is desired to investigate any number of partitions, the grid is partitioned only in the x -direction when the blocking decomposition is used. For example, it is impossible to decompose the grid into 5 partitions by splitting the grid in both the x - and y -direction.

As can be seen in Fig. 6.11, using METIS decomposition for a grid of size 20 million results in more speedup compared to the blocking partitioning. For larger grid sizes the difference between the gained speedup is less. As shown in Fig. 6.11, for a grid size of 200 million the speedup is almost the same for up to 150 processes. But, after that, with an increase in the number of processes, the speedup for blocking partitioning degraded, unlike the METIS partitioning.

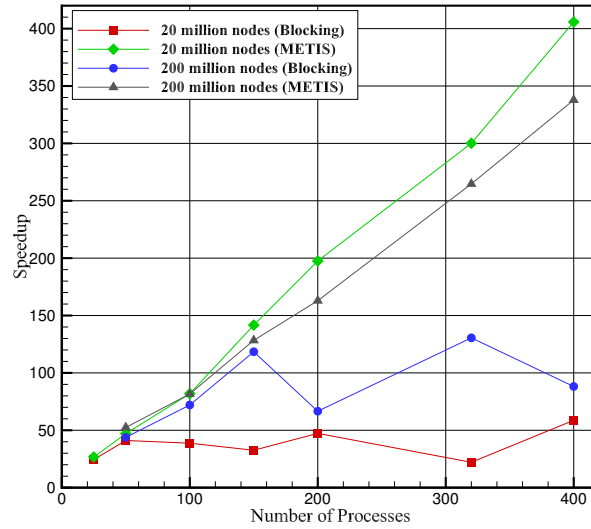


Figure 6.11 Comparison of the speedup for using blocking and METIS as the partitioning method. In this study, the first test case, i.e. a lid-driven cavity, with a grid size of 20 and 200 million nodes is used.

Maybe a challenge regarding the partitioning scheme is the execution time for grid decomposition. So, the execution time of the METIS library to decompose a series of grid sizes is presented in Fig 6.12. As can be seen, the execution time for one billion nodes is about 270s which is negligible compared to the overall execution time of the simulation. So, only the METIS partitioning will be used for the rest of this work.

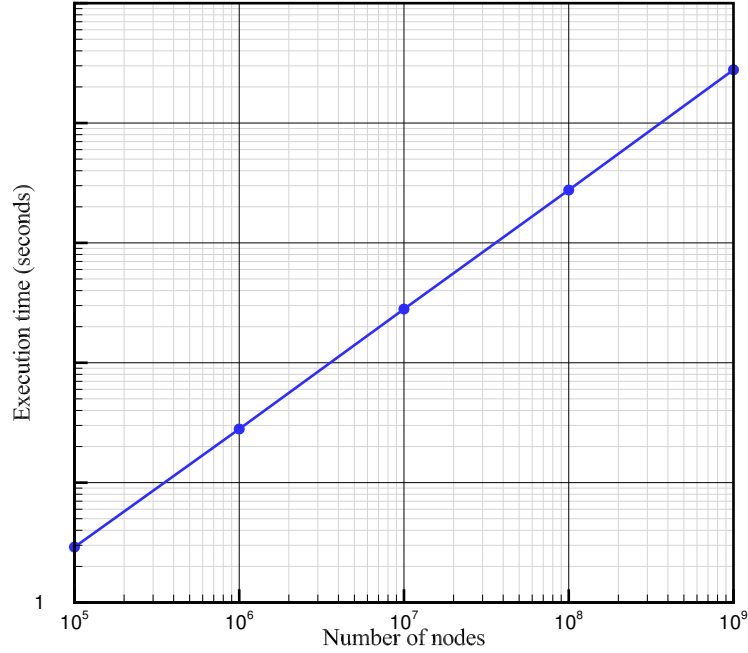


Figure 6.12 Execution time of METIS library for different grid sizes.

It could be concluded, that the decomposition method has a considerable effect on the performance of any distributed memory system. For example, in the presented investigation although a unique communication strategy is used, the performance of the program was much better when the METIS partitioning is used.

6.5.3 Dense systems analysis

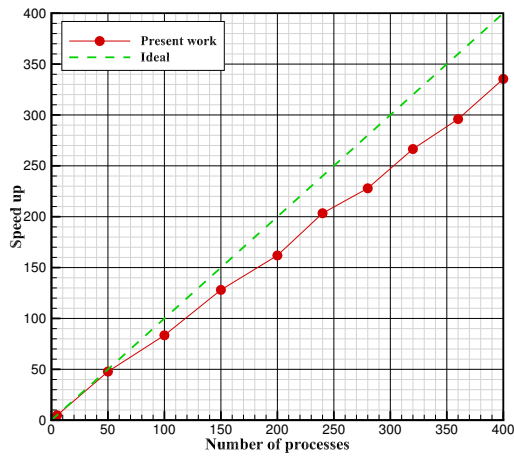
To evaluate the performance of the proposed communication algorithm, a dense system, *i. e.* the Cavity flow, is used in this section. All the parallel simulations are done for 10000 time steps. As the execution of the serial simulation is more time-consuming, only a few time steps are done and the computational time is extrapolated to 10000 time steps. In the following, the strong and weak scalability are studied.

Strong scalability

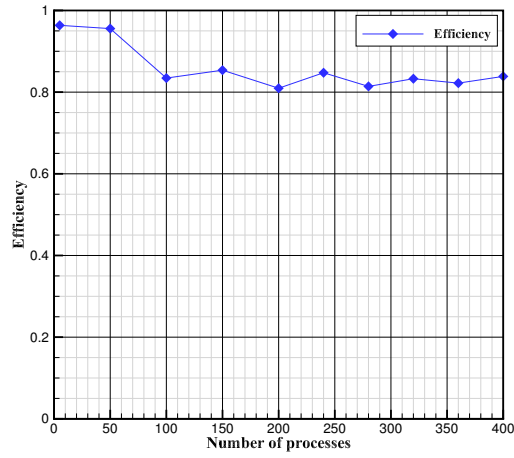
Fig. 6.13a illustrates the speedup, which indicates a scalable program. In this simulation, the speedup is around 340 for 400 processes. As could be seen from Fig. 6.13a there is not a considerable degradation in the speedup as the number of processes increases.

Based on Fig. 6.13b for less than 50 cores the efficiency is more than one which is due to the super-linear speedup. Although, the optimal result is linear, the speedup is super linear up

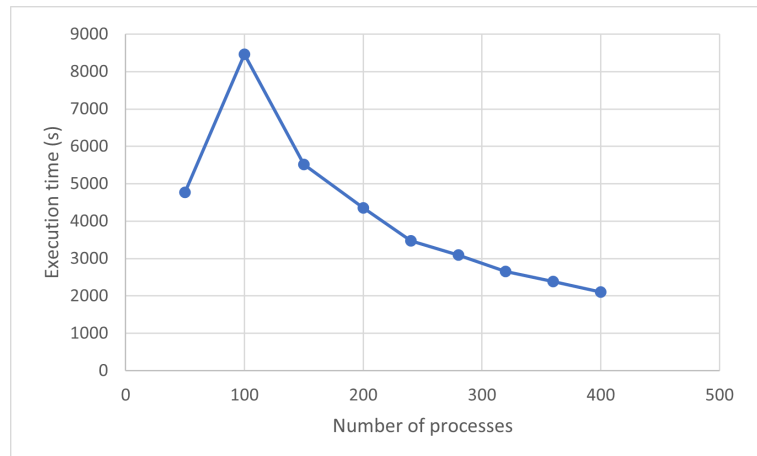
to 50 processes. This behavior is because there are some cache misses in the serial code [152] as the grid size is comparatively large, but as the grid is broken down into smaller sub-grids, in the parallel version, the number of cache misses decreases. However, the interesting thing about this graph is that the efficiency doesn't decrease sharply as the number of processes increases. This behavior shows the high scalability of the present program. In addition, the execution time of the simulations are depicted in Fig. 6.13c.



(a) Speedup.



(b) Efficiency.



(c) Execution time in seconds.

Figure 6.13 Speedup and efficiency for a dense system. In this test, a grid of size 200 Million fluid nodes is used.

Weak scalability

To do the weak scalability analysis a series of tests were performed by using 100, 200, and 400 processes. In these tests, the grid size gradually increased from 20 million to 500 million

nodes. Hence, for an analysis done on 400 processes, each process gets 50,000 to 1,250,000 nodes for simulation. As could be seen in Fig. 6.14a the speedup is decreased as the grid size increases up to 200 million nodes. After that, there is an increase in the speedup as the number of nodes increases.

Fig. 6.14b shows the efficiency for the weak scalability. As could be seen from this figure, the efficiency decreased slightly to about 0.8 by increasing the grid size. Then by increases in the grid size from 200 million nodes the efficiency increase to be more than 1 for 500 million nodes.

By considering the graphs shown in Fig. 6.14, the speedup for 500 million nodes is about 600 when 400 cores are used. Consequently, the efficiency is more than 1 which indicates a super linear speedup. So, it could be concluded that this communication algorithm is suitable for large-scale simulation.

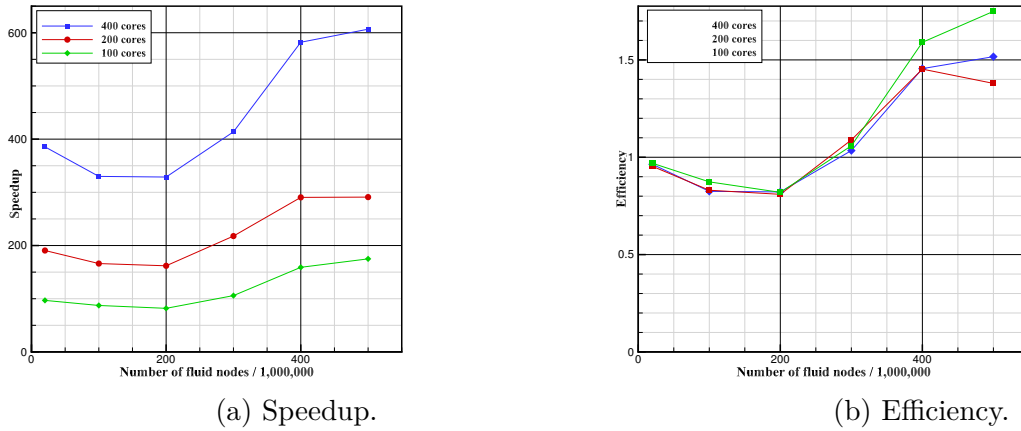


Figure 6.14 Speedup and efficiency of weak scalability with 200 processes for the dense system (lid-driven cavity test case).

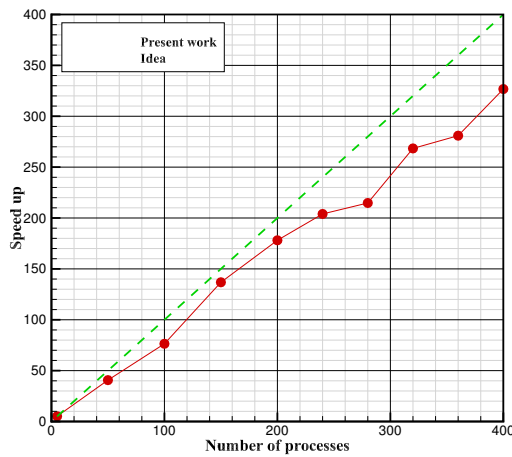
6.5.4 Sparse systems analysis

After the evaluation of the presented communication algorithm for dense systems, in this section, a sparse system is used to evaluate the communication algorithm for these kinds of geometries. To do that the second test case, *i.e.* the pseudo porous media, illustrated in Fig. 6.10b is used. The strong and weak scalability are investigated.

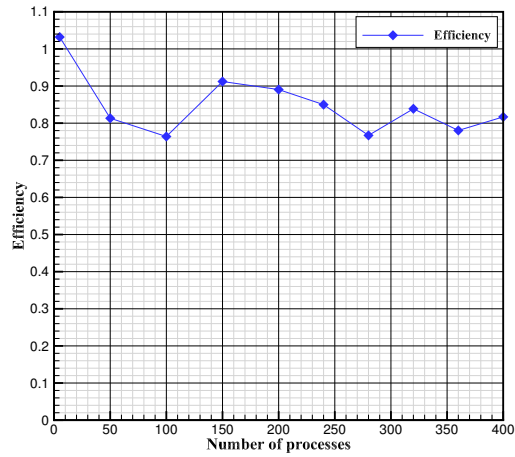
Strong scalability

In this test case, a regular grid of size 2 billion nodes is used and approximately 200 million of them are fluid nodes. Also, up to 400 processes are used to do the analysis. As could be seen in Fig. 6.15a by increasing the number of processes the speedup increases. For example, when using 50 processes the speedup is 40.6 and for 400 processes the speedup is 326.7.

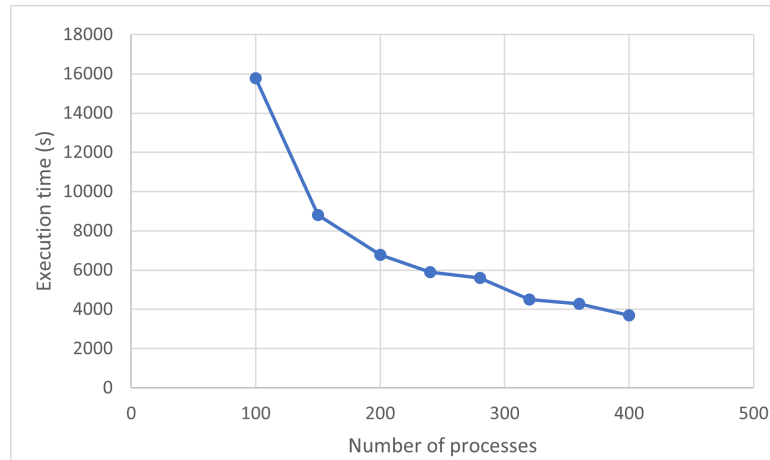
By considering both of the graphics in Fig. 6.15, by increasing the number of processes there is not a sharp decrease in the efficiency and the speedup. The minimum amount of efficiency is about 0.75 which is reasonable for a parallel system. In addition, the execution time of the simulations are illustrated in Fig. 6.15c.



(a) Speedup.



(b) Efficiency.



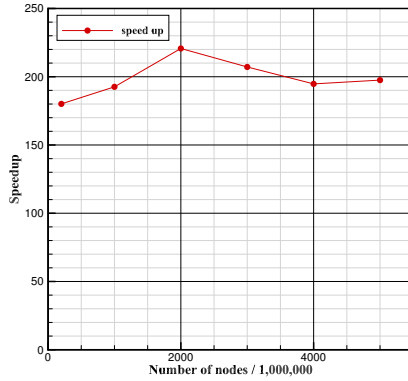
(c) Execution time in seconds.

Figure 6.15 Speedup and efficiency for a sparse system, *i.e.* the pseudo porous media geometry (a pseudo porous media). In this test, a grid of size 2 billion is used which consists of 200 million fluid nodes.

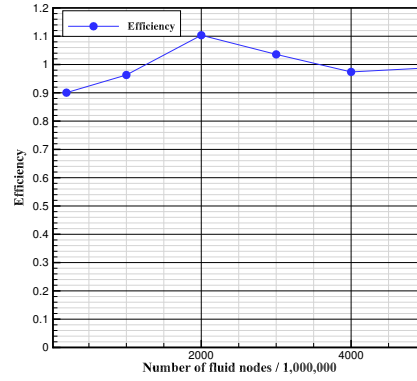
Weak scalability

To do the weak scalability analysis 200 processes are used and to increase the size of the problem, initially, a regular grid of size 20 million is used. Then the grid size is increased gradually up to 5 billion.

As could be seen in Fig. 6.16 the speedup is increased as the grid size increases up to 2 billion but after that, there is a decrease in the speedup. However, by considering Fig. 6.16b, the efficiency for grid sizes larger than 2 billion is about one which indicates a nearly linear speedup.



(a) Speedup.



(b) Efficiency.

Figure 6.16 Speedup and efficiency for sparse system, *i.e.* pseudo porous media geometry. In this test 200 cores are used.

As shown by the analysis of sparse systems, the speedup doesn't degrade for these kinds of geometries. One of the reasons is that only the required PDF components are exchanged at the interfaces of the decomposed grid. Suppose there is a dense and a sparse grid of the same size. For the sparse systems, if all of the PDFs at the interfaces of decomposed grid participate in the communication, the fraction of computation/communication decreases compared to the dense system. This is because, in this case, the amount of communication remains unchanged, but the computation decreases as there is no computation for solid nodes. So, in this case, the communication might not hide by the computation which could result in a decrease in the performance. To have a fair comparison in the above investigation, the number of fluid nodes for the sparse and dense systems is almost the same. So, by considering Fig. 6.13b and Fig. 6.15b the efficiency of these systems is almost the same which shows the performance of the program doesn't degrade for the sparse systems.

6.5.5 Large system simulation

In this section, the capability of the proposed algorithm to handle a large grid size is evaluated. To do that the second test case shown in Fig. 6.10b is used to do a weak scalability analysis. Here, the performance is measured in Millions of Distribution (f_i) Update per second (*MDUPS*) according to Eq. 6.1:

$$MDUPS = \frac{nNode \times m \times nTimeStep}{t_{tot} \times 10^6} \quad (6.1)$$

where $nNode$ is the total number of nodes, m is the number of PDF (velocity components) belonging to each node, $nTimeStep$ is the total number of simulation time steps, and t_{tot} the time in seconds required to perform the simulation on the computing device. In all the simulations, the quantity $nTimeStep$ is chosen as 5000 to avoid any fluctuation in performance measuring. So, the performance metric in this study is the mean of repetition. This analysis is done on a series of grid sizes ranging from 10 to 25 billion nodes. As the void fraction is 0.9 therefore the number of fluid nodes is approximately ranging from 1 to 2.5 billion fluid nodes. To do the simulation 400 processes are used and the METIS library is used to decompose the grid.

In Fig. 6.17, the execution time for 5000 time steps and the *MDUPS* is shown. These quantities are illustrated as a function of equivalent regular grid size and also the number of fluid nodes assigned to each process. As could be seen in the figure, although the *MDUPS* decreased by increasing the grid size, the amount of decrease in *MDUPS* is not considerable. Compare to *MDUPS* of about 66000 for 10 billion nodes, it is 57000 *MDUPS* for 20 billion nodes which shows approximately 13% degradation in performance. The analysis shows a sub-linear increase in the execution time by increasing the grid size which is a good result.

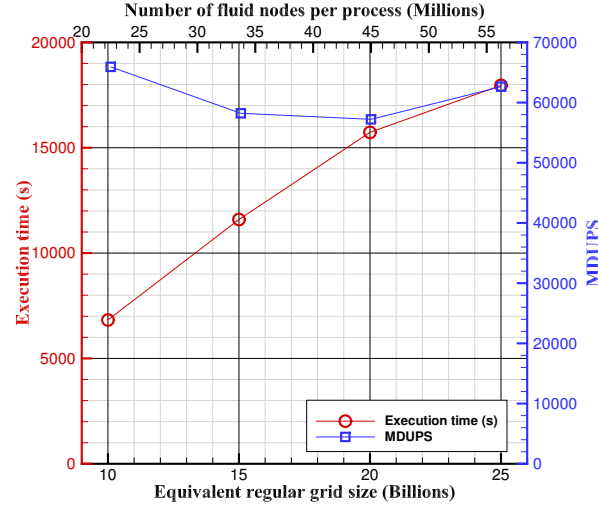


Figure 6.17 Execution time and *MDUPS* for sparse system, *i.e.* pseudo porous media geometry. In this test 400 cores are used.

6.6 Conclusion

In this study, a general algorithm for MPI communication for LBM simulation is proposed which is independent of the LBM program data structure and also from the lattice stencil. Also, by using non-blocking communication, the communication could be hidden with the computation. In addition, to have flexibility regarding the decomposition method, it is possible to use any partitioning approach. The proposed algorithm is in such a way that it is possible to hide the parallel programming parts from the base LBM program by using object-oriented programming. As explained in the case study within 6.8, there is no need to do a major modification in the serial code to develop it to parallel. The performance evaluation shows good scalability both for dense and sparse systems. However, using an efficient domain decomposition is still critical to minimize communication and also load balancing. One interesting future work is to test the proposed algorithm on a three-dimensional LBM program and also extend it for multi-GPU execution.

6.7 Acknowledgments

We acknowledge the support of the Natural Sciences and Engineering Council of Canada (NSERC), [funding reference number *RGPIN-2020-04512*].

6.8 Appendix: Case study: How to develop a parallel LBM program from a serial one

Developing a parallel program based on a serial LBM program [152] is done to evaluate the proposed communication algorithm. According to Fig. 6.18, for the parallel version of the LBM program just some more steps, shown with dashed line blocks in Fig. 6.18b, are added to the serial program. In Section 6.8.2, these steps will be explained in details. But first, in Section 6.8.1, a short explanation about the serial program is provided.

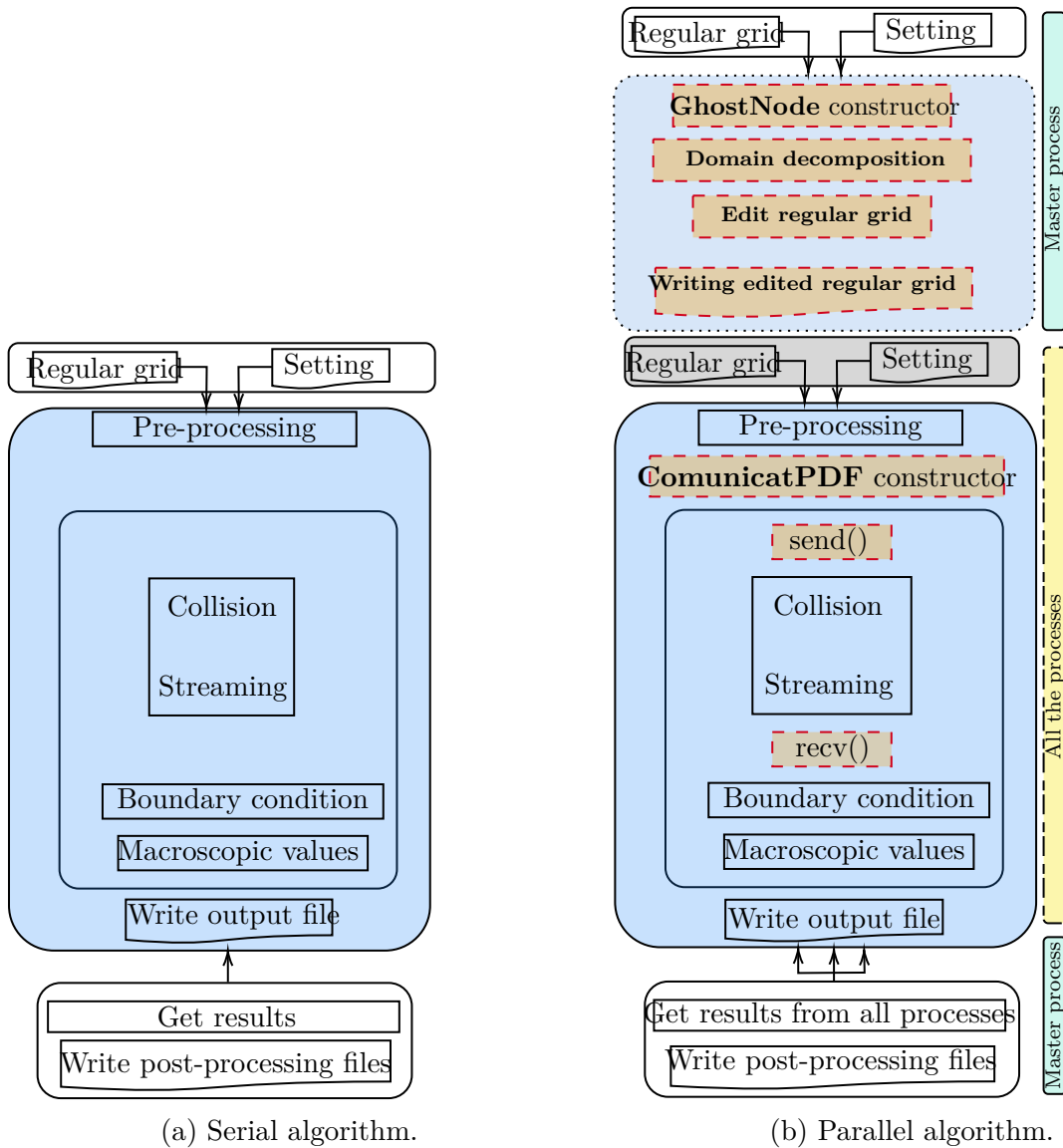


Figure 6.18 Algorithm of the serial and parallel LBM code.

6.8.1 Serial LBM algorithm

As depicted in Fig. 6.18a, all the required data are provided as input files. As a short description, the setting file contains all of the parameters required for the execution of the code. Also, the regular grid file represents the geometry of the computational domain.

At the implementation level, these input files are read and then some pre-processing tasks are done. After that, the LBM calculation is performed in a time marching loop by applying the collision, streaming, boundary condition, and macroscopic calculation functions. At the end of the simulation, all the output quantities are written in the output files to be used as input data for the post-processing module. For, more details about these steps refer to Ref. [152].

6.8.2 Parallel LBM program

In this section, each steps added to the serial program to extend it to the parallel program, shown in Fig. 6.18b, are explained in detail. It should be noted that in the parallel program each process executes the same code.

GhostNode constructor

As explained in Section 6.4, an object of the **GhostNode** class is created and then all the data of the ghost nodes are written by invoking the *writeGhostNodesData()* method in a temporary file which will be used by the communication algorithm framework later.

Domain decomposition

What is done in the decomposition step is assigning each node of the regular grid to a process. As the objective of this work is not to introduce a new decomposition method, the popular methods of decomposition are used. Since in this work, the blocking and multilevel k-way approaches are used for decomposition, in this section these methods are explained briefly.

Implementation of slicing or blocking is straightforward. In this method, the regular grid is divided into some blocks. Then a number indicating the partition index is assigned to any nodes of each of the blocks. This method is applied on the regular grid without considering the type of nodes, *i.e.* fluid or solid node. Hence for sparse geometries, this can not result in an efficient decomposition.

It is possible to use other decomposition approaches such as the multilevel k-way approach implemented in METIS [107] or parMETIS [107] to have a more efficient load balancing and

communication overhead simultaneously. This approach is efficient specifically for sparse systems such as porous media. Generally, METIS is to decompose grid and graph, but in the LBM there is not any grid like those used in classical CFD methods such as finite volume or finite element. Hence, to prepare the input data for METIS in the present study, a connectivity matrix is generated by using the regular grid. For, detail about the input data for METIS one can use Ref. [113]. As the regular grid nodes are distributed evenly throughout the computational domain a stencil depicted in Fig. 6.19 is used to generate an element for the METIS library. However, solid nodes are not used in the process of generating the input grid of METIS.

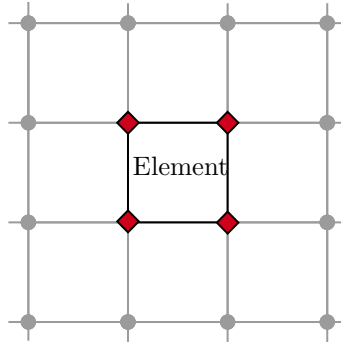
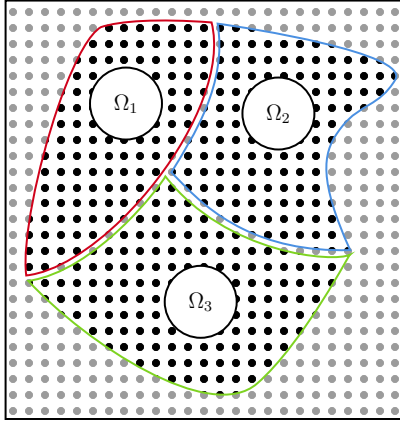


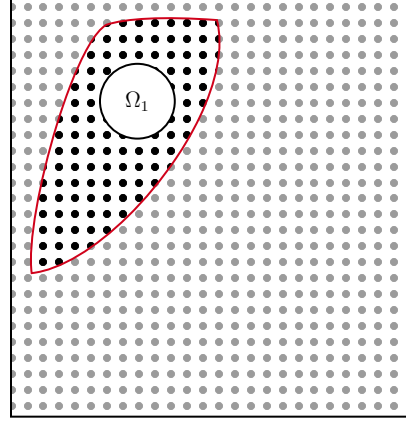
Figure 6.19 A typical element, forming from four nodes shown in diamond, is used to generate the input grid of METIS from a regular grid.

Edit regular grid

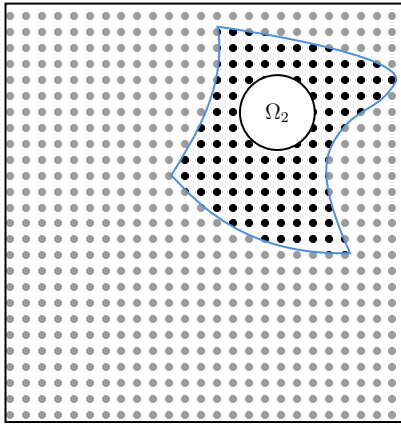
As depicted in Fig. 6.20a suppose there is a regular grid that is decomposed into three parts. In this figure, the black dots represent the fluid nodes and the grey dots are solid nodes. As illustrated in Fig. 6.20b, process 1 is responsible to do the computation of Ω_1 . In the present approach of parallel programming, just those nodes that belong to Ω_1 are considered fluid nodes, and all the other nodes are treated as solid nodes. For other parts of the grid which are shown in Fig. 6.20c and Fig. 6.20d, just Ω_2 and Ω_3 are considered as fluid nodes. As in the LBM algorithm presented in Ref. [152] the solid nodes are not stored, they are not participating in the decomposition step. Hence, only the fluid nodes are assigned to the existing processes.



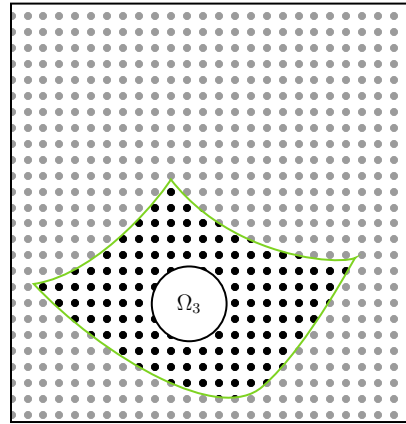
(a) Decomposed grid.



(b) Part 1 after decomposition.



(c) Part 2 after decomposition.



(d) Part 3 after decomposition.

Figure 6.20 Grid decomposition steps. Here, the initial grid is partitioned into three parts, and each part is assigned to a process. The black dot represents fluid nodes and the grey dot show solid nodes.

Writing edited regular grid

Based on the explanation of the serial code, each process should read a regular grid file. So, the master process should write the edited regular grid in a file to be read by each process.

CommunicatPDF constructor

An object of the **CommunicatPDF** class is created before the time marching loop, as explained in Section 6.4. Then in the time marching loop before the core functions of the LBM program the *send()* method is invoked. After that, at the end of computation of the LBM program the *recv()* method is invoked.

CHAPTER 7 GENERAL DISCUSSION

Based on the literature review of the thesis, it is concluded that in order to develop a suitable LBM algorithm for simulating the flow field for irregular geometries it is desired to avoid storing solid nodes. In addition, to increase the data locality and respect the simplification of implementation, using an appropriate data structure compatible with streaming and collision, of the LBM is necessary.

The idea followed in the thesis was to propose a new LBM algorithm to traverse all the PDFs to perform both the streaming and collision steps at the same time. So, as the loop is on the PDFs it is possible to use a structure-of-array memory layout to simplify the streaming step implementation. But, the compatible data structure with the BC is an array-of-structure. In this way, it is possible to have local BCs which is compatible with distributed memory system parallelization. Besides, one of the reasons which makes it hard to implement a new boundary condition is the dependency of the formulation on the orientation of the boundary nodes. This may enforce the developer of the LBM program to develop any new BC for each side of the computational domain. Which is a tedious process and prone to error. So, a new approach is proposed for BC implementation in this thesis. In this approach, instead of deriving a BC formulation for each side of the domain, all the edge/corner boundaries are mapped to one edge/corner to use just one formulation for that BC. Also, a local data structure is used which is compatible with the BC implementation. Because of the importance of parallel computing for an LBM simulation, evaluating any newly developed LBM algorithm in terms of capability to be executed on distributed memory system is crucial. In addition, in the literature there is not any general MPI framework for LBM that could be used individually, to parallel the proposed serial LBM algorithm. So, a new communication algorithm is proposed which is independent of the data structure of the base LBM code. As it is independent of the lattice stencil it could be used in 2D and 3D LBM code. However, in this thesis just the 2D program has been evaluated. This algorithm is implemented by using C++ object-oriented programming as a framework. To extend the newly developed LBM algorithm to parallel, presented in article 1, this framework was used.

CHAPTER 8 CONCLUSION AND RECOMMENDATIONS

8.1 Conclusion

In this thesis, a new LBM algorithm was proposed in which by avoiding storing the solid nodes the proposed algorithm is suitable for low void fraction systems. To verify the implemented code based on the mentioned algorithm some benchmarks are simulated. Then the performance of the code is investigated and compared to the Palabos open source project by simulation dense and sparse systems. The proposed algorithm uses the same amount of memory as the two-lattice but more than the two-step algorithm. As it uses an indirect addressing scheme, it is more efficient in terms of memory usage, particularly for low void fraction geometries. The performance of the implemented program, based on the proposed algorithm, for irregular geometries with low void fractions, is better compared to directly addressing LBM algorithm. This is because the data of the solid nodes are not stored in this method. so, there is no computation for these kinds of nodes. By considering the difficulties of BC implementation a framework is introduced to implement new BCs. To prove the capabilities of the implemented BC framework, some popular BCs are implemented and verified. To investigate the capability of the newly introduced LBM algorithm the MPI version of the code was implemented. The scalability of the parallel code was investigated by executing the code on the Niagara cluster. The performance evaluation shows good scalability both for dense and sparse systems.

8.2 Limitations

Besides the advantages and innovation of this thesis, there are some limitations. The new LBM algorithm, the BC framework, and the communication framework are implemented only for 2D simulations. The parallel version of the code is done only for distributed memory systems.

8.3 Future Research and recommendations

The following steps could be done in order to extend the present project to 3D and also the GPU parallel version of the code:

1. Extending boundary condition framework in GPU-based LBM solver: As the idea in developing the BC framework was to make it easy for the implementation of BC,

another capability could be added to the framework to be integrated into those LBM solvers that use GPU for parallelization. So, to implement a new BC the developer should not consider the GPU parallel programming but it should be executed on GPU.

2. Three-dimensional boundary condition framework: As the introduced BC framework is independent of the LBM data structure it could be used in other LBM solvers. So, by following the idea of that framework it could be extended to 3D.
3. Verifying the communication framework in 3D: The development of the communication framework is not dependent on the LBM solver and also on the dimension of the simulation. So, just by doing the domain decomposition and providing necessary data via some coding, it is possible to verify the framework for 3D simulations.
4. Unified 2D and 3D: It is possible to have a unified 2D/3D code for LBM. So, the main objective is to develop the 3D version of the proposed LBM algorithm. But, in the end, only one version should be used for 2D and 3D simulation.
5. Single GPU: The proposed LBM algorithm should be extended to use the GPU to accelerate the computations. In developing the 2D code already done in the study, one of the concerns was to separate the computationally intensive part of the code in order to be compatible with parallel programming. So, to develop the GPU-based parallel version of the code just those parts need to be extended.
6. Multi GPUs: At this point, everything is almost ready to verify the developed code for multi GPUs. Since the MPI communication and single GPU program are done, the next step is to merge them in order to have a multi-GPU program.

REFERENCES

- [1] U. Ghia, K. N. Ghia, and C. Shin, “High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method,” Journal of computational physics, vol. 48, no. 3, pp. 387–411, 1982.
- [2] I. Zacharoudiou, J. McCullough, and P. Coveney, “Development and performance of a hemelb gpu code for human-scale blood flow simulation,” arXiv preprint arXiv:2202.11770, 2022.
- [3] G. R. McNamara and G. Zanetti, “Use of the boltzmann equation to simulate lattice-gas automata,” Phys. Rev. Lett., vol. 61, pp. 2332–2335, Nov 1988.
- [4] D. A. Perumal and A. K. Dass, “A review on the development of lattice Boltzmann computation of macro fluid flows and heat transfer,” Alexandria Engineering Journal, vol. 54, no. 4, pp. 955–971, 2015.
- [5] U. Frisch, B. Hasslacher, and Y. Pomeau, “Lattice-gas automata for the Navier-Stokes equation,” Physical review letters, vol. 56, no. 14, p. 1505, 1986.
- [6] H. Chen, S. Chen, and W. H. Matthaeus, “Recovery of the Navier-Stokes equations using a lattice-gas Boltzmann method,” Physical review A, vol. 45, no. 8, p. R5339, 1992.
- [7] F. J. Higuera and J. Jiménez, “Boltzmann approach to lattice gas simulations,” EPL (Europhysics Letters), vol. 9, no. 7, p. 663, 1989.
- [8] G. R. McNamara and G. Zanetti, “Use of the Boltzmann equation to simulate lattice-gas automata,” Physical review letters, vol. 61, no. 20, p. 2332, 1988.
- [9] T. Krüger, H. Kusumaatmaja, A. Kuzmin, O. Shardt, G. Silva, and E. M. Viggen, The Lattice Boltzmann Method: Principles and Practice. Springer, 2016.
- [10] S. Leclaire, M. Reggio, and J.-Y. Trépanier, “Numerical evaluation of two recoloring operators for an immiscible two-phase flow lattice Boltzmann model,” Applied Mathematical Modelling, vol. 36, no. 5, pp. 2237–2252, 2012.
- [11] X. He, S. Chen, and G. D. Doolen, “A novel thermal model for the lattice Boltzmann method in incompressible limit,” Journal of computational physics, vol. 146, no. 1, pp. 282–300, 1998.

- [12] Y. H. Qian, D. D'Humières, and P. Lallemand, "Lattice BGK models for navier-stokes equation," Europhysics Letters (EPL), vol. 17, no. 6, pp. 479–484, 1992.
- [13] Y. Li, "An improved volumetric LBM boundary approach and its extension for sliding mesh simulation," Ph.D. dissertation, Iowa State University, 2012.
- [14] R. R. Nourgaliev, T.-N. Dinh, T. G. Theofanous, and D. Joseph, "The lattice Boltzmann equation method: theoretical interpretation, numerics and implications," International Journal of Multiphase Flow, vol. 29, no. 1, pp. 117–169, 2003.
- [15] M. C. Sukop and D. T. Thorne, Lattice Boltzmann Modeling. Springer, 2006.
- [16] G. Herschlag, S. Lee, J. S. Vetter, and A. Randles, "GPU data access on complex geometries for D3Q19 lattice Boltzmann method," in 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2018, pp. 825–834.
- [17] M. K. Misztal, A. Hernandez-Garcia, R. Martin, H. O. Sørensen, and J. Mathiesen, "Detailed analysis of the lattice Boltzmann method on unstructured grids," Journal of Computational Physics, vol. 297, pp. 316–339, 2015.
- [18] G. Peng, H. Xi, and S.-H. Chou, "On boundary conditions in the finite volume lattice Boltzmann method on unstructured meshes," International journal of modern physics C, vol. 10, no. 06, pp. 1003–1016, 1999.
- [19] Z. Fan, Y. Zhao, A. Kaufman, and Y. He, "Adapted unstructured LBM for flow simulation on curved surfaces," in Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation, 2005, pp. 245–254.
- [20] N. Pellerin, S. Leclaire, and M. Reggio, "Solving incompressible fluid flows on unstructured meshes with the lattice Boltzmann flux solver," Engineering Applications of Computational Fluid Mechanics, vol. 11, no. 1, pp. 310–327, 2017.
- [21] S. Leclaire, N. Pellerin, M. Reggio, and J.-Y. Trépanier, "An approach to control the spurious currents in a multiphase lattice Boltzmann method and to improve the implementation of initial condition," International Journal for Numerical Methods in Fluids, vol. 77, no. 12, pp. 732–746, 2015.
- [22] K. Mattila, J. Hyväluoma, J. Timonen, and T. Rossi, "Comparison of implementations of the lattice-Boltzmann method," Computers & Mathematics with Applications, vol. 55, no. 7, pp. 1514–1524, 2008.

- [23] M. Geier and M. Schönherr, “Esoteric twist: An efficient in-place streaming algorithmus for the lattice boltzmann method on massively parallel hardware,” Computation, vol. 5, no. 2, 2017.
- [24] D. Vidal, R. Roy, and F. Bertrand, “On improving the performance of large parallel lattice Boltzmann flow simulations in heterogeneous porous media,” Computers & Fluids, vol. 39, no. 2, pp. 324–337, 2010.
- [25] M. Hasert, K. Masilamani, S. Zimny, H. Klimach, J. Qi, J. Bernsdorf, and S. Roller, “Complex fluid simulations with the parallel tree-based lattice Boltzmann solver Musubi,” Journal of Computational Science, vol. 5, no. 5, pp. 784–794, 2014.
- [26] N. S. Martys and J. G. Hagedorn, “Multiscale modeling of fluid transport in heterogeneous materials using discrete Boltzmann methods,” Materials and structures, vol. 35, no. 10, pp. 650–658, 2002.
- [27] M. Astorino, J. B. Sagredo, and A. Quarteroni, “A modular lattice Boltzmann solver for GPU computing processors,” SeMA journal, vol. 59, no. 1, pp. 53–78, 2012.
- [28] J. Fietz, M. J. Krause, C. Schulz, P. Sanders, and V. Heuveline, “Optimized hybrid parallel lattice Boltzmann fluid flow simulations on complex geometries,” in European Conference on Parallel Processing. Springer, 2012, pp. 818–829.
- [29] S. M. Guzik, T. H. Weisgraber, P. Colella, and B. J. Alder, “Interpolation methods and the accuracy of lattice-Boltzmann mesh refinement,” Journal of Computational Physics, vol. 259, pp. 461–487, 2014.
- [30] Y. Ma, Y. Wang, and M. Xie, “Multiblock adaptive mesh refinement for the SN transport equation based on lattice Boltzmann method,” Nuclear Science and Engineering, vol. 193, no. 11, pp. 1219–1237, 2019.
- [31] F. Schornbaum and U. Rude, “Extreme-scale block-structured adaptive mesh refinement,” SIAM Journal on Scientific Computing, vol. 40, no. 3, pp. C358–C387, 2018.
- [32] E. Calore, A. Gabbana, S. Schifano, and R. Tripiccione, “Optimization of lattice Boltzmann simulations on heterogeneous computers,” The International Journal of High Performance Computing Applications, vol. 33, no. 1, pp. 124–139, 2019.
- [33] A. G. Shet, K. Siddharth, S. H. Sorathiya, A. M. Deshpande, S. D. Sherlekar, B. Kaul, and S. Ansumali, “On vectorization for lattice based simulations,” International Journal of Modern Physics C, vol. 24, no. 12, p. 1340011, 2013.

- [34] E. Calore, A. Gabbana, S. F. Schifano, and R. Tripiccione, “Design and optimizations of lattice Boltzmann methods for massively parallel gpu-based clusters,” in Analysis and Applications of Lattice Boltzmann Simulations. IGI Global, 2018, pp. 54–114.
- [35] D. Majeti, K. S. Meel, R. Barik, and V. Sarkar, “Automatic data layout generation and kernel mapping for CPU+GPU architectures,” in Proceedings of the 25th International Conference on Compiler Construction, 2016, pp. 240–250.
- [36] T. Pohl, M. Kowarschik, J. Wilke, K. Iglberger, and U. Rde, “Optimization and profiling of the cache performance of parallel lattice Boltzmann codes,” Parallel Processing Letters, vol. 13, no. 04, pp. 549–560, 2003.
- [37] S. Murphy, “Performance of lattice Boltzmann kernels,” Ph.D. dissertation, 2005.
- [38] G. Dethier, P.-A. de Marneffe, and P. Marchot, “Lattice Boltzmann simulation code optimization based on constant-time circular array shifting,” Procedia Computer Science, vol. 4, pp. 1004–1013, 2011.
- [39] K. Mattila, J. Hyvluoma, T. Rossi, M. Aspns, and J. Westerholm, “An efficient swap algorithm for the lattice boltzmann method,” Computer Physics Communications, vol. 176, no. 3, pp. 200–210, 2007.
- [40] Q. Zou and X. He, “On pressure and velocity boundary conditions for the lattice Boltzmann BGK model,” Physics of fluids, vol. 9, no. 6, pp. 1591–1598, 1997.
- [41] S. Girimaji, Lattice Boltzmann Method: Fundamentals and Engineering Applications With Computer Codes. American Institute of Aeronautics and Astronautics, 2013.
- [42] M. E. Kutay, A. H. Aydilek, and E. Masad, “Laboratory validation of lattice boltzmann method for modeling pore-scale flow in granular materials,” Computers and Geotechnics, vol. 33, no. 8, pp. 381–395, 2006.
- [43] M. Hecht and J. Harting, “Implementation of on-site velocity boundary conditions for d3q19 lattice Boltzmann simulations,” Journal of Statistical Mechanics: Theory and Experiment, vol. 2010, no. 01, p. P01018, 2010.
- [44] A. Agarwal, “Modification of On-Site Velocity Boundary Condition in LBM for D3Q27 Lattice Structure,” Transactions of the Indian National Academy of Engineering, vol. 5, 02 2020.
- [45] J. Zhang and D. Y. Kwok, “Pressure boundary condition of the lattice Boltzmann method for fully developed periodic flows,” Phys. Rev. E, vol. 73, p. 047702, Apr 2006.

- [46] S. H. Kim and H. Pitsch, “A generalized periodic boundary condition for lattice Boltzmann method simulation of a pressure driven flow in a periodic geometry,” Physics of Fluids, vol. 19, no. 10, p. 108101, 2007.
- [47] S. Leclaire, J. Latt, D. Vidal, and F. Bertrand, “Multiphase periodic pressure difference boundary condition enhanced by a proportional-integral-derivative controller for the lattice Boltzmann method,” International Journal for Numerical Methods in Fluids, vol. 88, no. 9, pp. 434–446, 2018.
- [48] F. Dubois, P. Lallemand, and M. M. Tekitek, “Generalized bounce back boundary condition for the nine velocities two-dimensional lattice Boltzmann scheme,” Computers & Fluids, vol. 193, p. 103534, 2019.
- [49] J. Latt, B. Chopard, O. Malaspinas, M. Deville, and A. Michler, “Straight velocity boundaries in the lattice boltzmann method,” Phys. Rev. E, vol. 77, p. 056703, May 2008.
- [50] E. Aslan, I. Taymaz, and A. C. Benim, “Investigation of LBM curved boundary treatments for unsteady flows,” European Journal of Mechanics - B/Fluids, vol. 51, pp. 68–74, 2015.
- [51] Z. Guo, C. Zheng, and B. Shi, “An extrapolation method for boundary conditions in lattice Boltzmann method,” Physics of fluids, vol. 14, no. 6, pp. 2007–2010, 2002.
- [52] R. Mei, L.-S. Luo, and W. Shyy, “An accurate curved boundary treatment in the lattice Boltzmann method,” Journal of Computational Physics, vol. 155, no. 2, pp. 307–330, 1999.
- [53] O. Filippova and D. Hänel, “Grid refinement for lattice-bgk models,” Journal of Computational Physics, vol. 147, no. 1, pp. 219–228, 1998.
- [54] J. Huang, Z. Hu, and W.-A. Yong, “Second-order curved boundary treatments of the lattice boltzmann method for convection–diffusion equations,” Journal of Computational Physics, vol. 310, pp. 26–44, 2016.
- [55] L. Wang, S. Tao, J. Hu, K. Zhang, and G. Lu, “Discrete effects on boundary conditions of the lattice boltzmann method for convection-diffusion equations with curved geometries,” International Communications in Heat and Mass Transfer, vol. 122, p. 105130, 2021.

- [56] J.-M. Tucny, D. Vidal, S. Leclaire, and F. Bertrand, “Comparison of existing and extended boundary conditions for the simulation of rarefied gas flows using the lattice Boltzmann method,” International Journal of Modern Physics C, vol. 31, no. 05, p. 2050070, 2020.
- [57] N. Pellerin, S. Leclaire, and M. Reggio, “Equilibrium distributions for straight, curved, and immersed boundary conditions in the lattice Boltzmann method,” Computers & Fluids, vol. 101, pp. 126–135, 2014.
- [58] T. Zhang, B. Shi, Z. Guo, Z. Chai, and J. Lu, “General bounce-back scheme for concentration boundary condition in the lattice-Boltzmann method,” Phys. Rev. E, vol. 85, p. 016701, 2012.
- [59] S. Ansumali and I. V. Karlin, “Kinetic boundary conditions in the lattice Boltzmann method,” Physical Review E, vol. 66, no. 2, p. 026311, 2002.
- [60] Z. Chai, Z. Guo, L. Zheng, and B. Shi, “Lattice Boltzmann simulation of surface roughness effect on gaseous flow in a microchannel,” Journal of Applied Physics, vol. 104, no. 1, p. 014902, 2008.
- [61] F. Verhaeghe, L.-S. Luo, and B. Blanpain, “Lattice Boltzmann modeling of microchannel flow in slip flow regime,” Journal of Computational Physics, vol. 228, no. 1, pp. 147–157, 2009.
- [62] Z. Chai, B. Shi, Z. Guo, and J. Lu, “Gas flow through square arrays of circular cylinders with klinkenberg effect: a lattice Boltzmann study,” Communications in Computational Physics, vol. 8, no. 5, p. 1052, 2010.
- [63] S. Tao and Z. Guo, “Boundary condition for lattice Boltzmann modeling of microscale gas flows with curved walls in the slip regime,” Phys. Rev. E, vol. 91, p. 043305, Apr 2015.
- [64] S. K. P. Sanjeevi, A. Zarghami, and J. T. Padding, “Choice of no-slip curved boundary condition for lattice Boltzmann simulations of high-Reynolds-number flows,” Phys. Rev. E, vol. 97, p. 043305, Apr 2018.
- [65] M. Bouzidi, M. Firdaouss, and P. Lallemand, “Momentum transfer of a boltzmann-lattice fluid with boundaries,” Physics of fluids, vol. 13, no. 11, pp. 3452–3459, 2001.
- [66] P. Lallemand and L.-S. Luo, “Lattice boltzmann method for moving boundaries,” Journal of Computational Physics, vol. 184, no. 2, pp. 406–421, 2003.

- [67] D. Yu, R. Mei, and W. Shyy, “A unified boundary treatment in lattice Boltzmann method,” in 41st aerospace sciences meeting and exhibit, 2003, p. 953.
- [68] P.-H. Kao and R.-J. Yang, “An investigation into curved and moving boundary treatments in the lattice boltzmann method,” Journal of Computational Physics, vol. 227, no. 11, pp. 5671–5690, 2008.
- [69] X. Yin and J. Zhang, “An improved bounce-back scheme for complex boundary conditions in lattice boltzmann method,” Journal of Computational Physics, vol. 231, no. 11, pp. 4295–4303, 2012.
- [70] I. Ginzburg, “Generic boundary conditions for lattice Boltzmann models and their application to advection and anisotropic dispersion equations,” Advances in Water Resources, vol. 28, no. 11, pp. 1196–1216, 2005.
- [71] O. Malaspinas, B. Chopard, and J. Latt, “General regularized boundary condition for multi-speed lattice Boltzmann models,” Computers & Fluids, vol. 49, no. 1, pp. 29–35, 2011.
- [72] “ComputCpp,” 2022. [Online]. Available: <https://www.codeplay.com/products/computesuite/computecpp>
- [73] “Par4All,” 2022. [Online]. Available: <http://par4all.github.io/>
- [74] “PIPS,” 2022. [Online]. Available: <http://pips4u.org/>
- [75] “PLUTO,” 2022. [Online]. Available: <http://pluto-compiler.sourceforge.net/>
- [76] “Autopar,” 2022. [Online]. Available: https://en.wikibooks.org/wiki/ROSE_Compiler_Framework/autoPar
- [77] “OMP2MPI,” 2022. [Online]. Available: <https://arxiv.org/abs/1502.02921>
- [78] “Julia,” 2022. [Online]. Available: <https://julialang.org/>
- [79] “Charm++,” 2022. [Online]. Available: <http://charm.cs.illinois.edu/index>
- [80] “OpenACC,” 2022. [Online]. Available: <https://www.openacc.org/about>
- [81] “Apache hadoop,” 2022. [Online]. Available: <http://hadoop.apache.org/>
- [82] M. Basha and N. A. C. Sidik, “Numerical predictions of laminar and turbulent forced convection: Lattice Boltzmann simulations using parallel libraries,” International Journal of Heat and Mass Transfer, vol. 116, pp. 715–724, 2018.

- [83] K. Sabri, H. Rabbani, and P. H. Gunawan, “Openmp performance for benchmark 2d shallow water equations using lbm,” in Journal of Physics: Conference Series, vol. 971, no. 1. IOP Publishing, 2018, p. 012033.
- [84] M. Schönherr, K. Kucher, M. Geier, M. Stiebler, S. Freudiger, and M. Krafczyk, “Multi-thread implementations of the lattice Boltzmann method on non-uniform grids for cpus and gpus,” Computers & Mathematics with Applications, vol. 61, no. 12, pp. 3730–3743, 2011.
- [85] J. Wang, X. Zhang, A. G. Bengough, and J. W. Crawford, “Domain-decomposition method for parallel lattice Boltzmann simulation of incompressible flow in porous media,” Physical Review E, vol. 72, no. 1, p. 016706, 2005.
- [86] Y. Fu, F. Li, F. Song, and L. Zhu, “Designing a parallel memory-aware lattice Boltzmann algorithm on manycore systems,” in 2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). IEEE, 2018, pp. 97–106.
- [87] C. Schepke and N. Maillard, “Performance improvement of the parallel lattice Boltzmann method through blocked data distributions,” in 19th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD’07). IEEE, 2007, pp. 71–78.
- [88] N. Thürey, T. Pohl, and U. Rüde, “Hybrid parallelization techniques for lattice boltzmann free surface flows,” in Parallel Computational Fluid Dynamics 2007. Springer, 2009, pp. 179–186.
- [89] V. Heuveline, M. J. Krause, and J. Latt, “Towards a hybrid parallelization of lattice boltzmann methods,” Computers & Mathematics with Applications, vol. 58, no. 5, pp. 1071–1080, 2009.
- [90] J. Tölke, “NVIDIA CUDA. Programming Guide,” 2009 International Conference on Parallel Processing, no. Version 4.0, 2010.
- [91] “opencl,” 2022. [Online]. Available: <https://www.khronos.org/opencl/>
- [92] J. Tölke, “Implementation of a lattice Boltzmann kernel using the compute unified device architecture developed by nvidia,” Computing and Visualization in Science, vol. 13, no. 1, pp. 29–39, 2010.

- [93] P. Bailey, J. Myre, S. D. Walsh, D. J. Lilja, and M. O. Saar, “Accelerating lattice boltzmann fluid flow simulations using graphics processors,” in 2009 international conference on parallel processing. IEEE, 2009, pp. 550–557.
- [94] F. Kuznik, C. Obrecht, G. Rusaouen, and J.-J. Roux, “Lbm based flow simulation using gpu computing processor,” Computers & Mathematics with Applications, vol. 59, no. 7, pp. 2380–2392, 2010.
- [95] M. Bernaschi, E. Agostini, and D. Rossetti, “Benchmarking multi-GPU applications on modern multi-GPU integrated systems,” Concurrency and Computation: Practice and Experience, vol. 33, no. 14, p. e5470, 2021.
- [96] S. Potluri, H. Wang, D. Bureddy, A. K. Singh, C. Rosales, and D. K. Panda, “Optimizing mpi communication on multi-GPU systems using CUDA inter-process communication,” in 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum. IEEE, 2012, pp. 1848–1857.
- [97] C. Zhu, J. Liu, L. Feng, and X. Deng, “Research on the simulation of PF-LBM model based on MPI+CUDA mixed granularity parallel,” AIP Advances, vol. 8, no. 6, p. 65017, 2018.
- [98] Y. Ye, K. Li, Y. Wang, and T. Deng, “Parallel computation of entropic lattice Boltzmann method on hybrid CPU-GPU accelerated system,” Computers & Fluids, vol. 110, pp. 114–121, 2015.
- [99] P. Michalski, “State of the practice for lattice boltzmann method software,” Master’s thesis, McMaster University, 2022.
- [100] M. J. Krause, A. Kummerländer, S. J. Avis, H. Kusumaatmaja, D. Dapelo, F. Klemens, M. Gaedtke, N. Hafen, A. Mink, R. Trunk et al., “Openlb—open source lattice boltzmann code,” Computers & Mathematics with Applications, vol. 81, pp. 258–288, 2021.
- [101] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux, “The thelma project: Multi-gpu implementation of the lattice boltzmann method,” The International Journal of High Performance Computing Applications, vol. 25, no. 3, pp. 295–303, 2011.
- [102] J. Latt, O. Malaspinas, D. Kontaxakis, A. Parmigiani, D. Lagrava, F. Brogi, M. B. Belgacem, Y. Thorimbert, S. Leclaire, S. Li, F. Marson, J. Lemus, C. Kotsalos, R. Conradin, C. Coreixas, R. Petkantchin, F. Raynaud, J. Beny, and B. Chopard, “Palabos: Parallel lattice Boltzmann solver,” Computers & Mathematics with Applications,

- vol. 81, pp. 334–350, 2021, development and Application of Open-source Software for Problems with Numerical PDEs.
- [103] M. D. Mazzeo and P. V. Coveney, “Hemelb: A high performance parallel lattice-Boltzmann code for large scale fluid flow in complex geometries,” Computer Physics Communications, vol. 178, no. 12, pp. 894–914, 2008.
 - [104] M. Bauer, S. Eibl, C. Godenschwager, N. Kohl, M. Kuron, C. Rettinger, F. Schornbaum, C. Schwarzmeier, D. Thönnies, H. Köstler et al., “walberla: A block-structured high-performance framework for multiphysics simulations,” Computers & Mathematics with Applications, vol. 81, pp. 478–501, 2021.
 - [105] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux, “Multi-gpu implementation of the lattice boltzmann method,” Computers & Mathematics with Applications, vol. 65, no. 2, pp. 252–261, 2013.
 - [106] C. Obrecht and Kuznik, “Scalable lattice Boltzmann solvers for CUDA GPU clusters,” Parallel Computing, vol. 39, no. 6-7, pp. 259–270, 2013.
 - [107] B. T. Shealy, M. Yousefi, A. T. Srinath, M. C. Smith, and U. D. Schiller, “Gpu acceleration of the hemelb code for lattice boltzmann simulations in sparse complex geometries,” IEEE Access, vol. 9, pp. 61 224–61 236, 2021.
 - [108] A. Patronis, J. McCullough, R. Richardson, R. Halver, R. Marshall, M. Ruefenacht, B. Wylie, T. Odaker, M. Wiedemann, B. Lloyd et al., “Developments for the efficient self-coupling of hemelb,” 2019.
 - [109] J. V. T. Risso, M. Bauer, P. R. de Carvalho, U. Rüde, and D. Weingaertner, “Scalable gpu communication with code generation on stencil applications,” in 2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). IEEE, 2019, pp. 88–95.
 - [110] C. Schepke, N. Maillard, and P. O. Navaux, “Parallel lattice Boltzmann method with blocked partitioning,” International Journal of Parallel Programming, vol. 37, no. 6, pp. 593–611, 2009.
 - [111] A. Hoekstra and P. Sloot, “Distributed particle simulation of flow in complex geometries,” WIT Transactions on Information and Communication Technologies, vol. 23, 2000.

- [112] G. Karypis and V. Kumar, “Parallel multilevel series k-way partitioning scheme for irregular graphs,” Siam Review, vol. 41, no. 2, pp. 278–300, 1999.
- [113] K. V. Karypis George, “A fast and high quality multilevel scheme for partitioning irregular graphs,” SIAM Journal on scientific Computing, vol. 20, no. 1, pp. 359–392, 1998.
- [114] K. Schloegel, G. Karypis, and V. Kumar, “Parallel static and dynamic multi-constraint graph partitioning,” Concurrency and Computation: Practice and Experience, vol. 14, no. 3, pp. 219–240, 2002.
- [115] S. Huang, E. Aubanel, and V. C. Bhavsar, “PaGrid: A mesh partitioner for computational grids,” Journal of Grid Computing, vol. 4, no. 1, pp. 71–88, 2006.
- [116] C. Walshaw and M. Cross, “JOSTLE–Multilevel Graph Partitioning Software: An Overview,” Mesh Partitioning Techniques and Domain Decomposition Techniques, Kap, vol. 2, pp. 27–58, 2008.
- [117] S. Aluru and F. E. Sevilgen, “Parallel domain decomposition and load balancing using space-filling curves,” in Proceedings fourth international conference on high-performance computing. IEEE, 1997, pp. 230–235.
- [118] S. Tsuzuki and T. Aoki, “Effective dynamic load balance using space-filling curves for large-scale sph simulations on gpu-rich supercomputers,” in 2016 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (Scala). IEEE, 2016, pp. 1–8.
- [119] J. Qi, “Efficient Lattice Boltzmann Simulations on Large Scale High Performance Computing Systems,” Ph.D. dissertation, Siegen, 2017.
- [120] T. Goehring and Y. Saad, “Heuristic algorithms for automatic graph partitioning,” Citeseer, Tech. Rep., 1994.
- [121] M. D. Mazzeo, “Lattice-boltzmann simulations of cerebral blood flow,” Ph.D. dissertation, UCL (University College London), 2010.
- [122] J. Chen and V. E. Taylor, “Mesh partitioning for efficient use of distributed systems,” IEEE Transactions on Parallel and Distributed Systems, vol. 13, no. 1, pp. 67–79, 2002.
- [123] G. Dethier, “Design and implementation of a distributed lattice Boltzmann-based fluid flow simulation tool,” Ph.D. dissertation, Université de Liège, Liège, Belgique, 2011.

- [124] T. Tomczak and R. G. Szafran, “A new GPU implementation for lattice-Boltzmann simulations on sparse geometries,” Computer Physics Communications, vol. 235, pp. 258–278, 2019.
- [125] M. Mohrhard, G. Thäter, J. Bludau, B. Horvat, and M. J. Krause, “Auto-vectorization friendly parallel lattice boltzmann streaming scheme for direct addressing,” Computers & Fluids, vol. 181, pp. 1–7, 2019.
- [126] S. Jubertie, I. Masliah, and J. Falcou, “Data layout and simd abstraction layers: decoupling interfaces from implementations,” in 2018 International Conference on High Performance Computing & Simulation (HPCS). IEEE, 2018, pp. 531–538.
- [127] M. J. Mawson and A. J. Revell, “Memory transfer optimization for a lattice Boltzmann solver on Kepler architecture nVidia GPUs,” Computer Physics Communications, vol. 185, no. 10, pp. 2566–2574, 2014.
- [128] E. Calore, F. Mantovani, and D. Ruiz, “Advanced Performance Analysis of HPC Workloads on Cavium ThunderX,” in 2018 International Conference on High Performance Computing Simulation (HPCS), 2018, pp. 375–382.
- [129] J. Latt, O. Malaspinas, D. Kontaxakis, A. Parmigiani, D. Lagrava, F. Brogi, M. B. Belgacem, Y. Thorimbert, S. Leclaire, S. Li, F. Marson, J. Lemus, C. Kotsalos, R. Conradin, C. Coreixas, R. Petkantchin, F. Raynaud, J. Beny, and B. Chopard, “Palabos: Parallel lattice Boltzmann solver,” Computers & Mathematics with Applications, vol. 81, pp. 334–350, 2021, development and Application of Open-source Software for Problems with Numerical PDEs.
- [130] S. Chen and G. D. Doolen, “Lattice Boltzmann method for fluid flows,” Annual review of fluid mechanics, vol. 30, no. 1, pp. 329–364, 1998.
- [131] D. A. Perumal and A. K. Dass, “A review on the development of lattice Boltzmann computation of macro fluid flows and heat transfer,” Alexandria Engineering Journal, vol. 54, no. 4, pp. 955–971, 2015.
- [132] P. D. Arumuga, G. V. Kumar, and A. K. Dass, “Lattice Boltzmann simulation of flow over a circular cylinder at moderate reynolds numbers,” Thermal Science, vol. 18, no. 4, pp. 1235–1246, 2014.
- [133] J. Fietz, M. J. Krause, C. Schulz, P. Sanders, and V. Heuveline, “Optimized hybrid parallel lattice Boltzmann fluid flow simulations on complex geometries,” in European Conference on Parallel Processing. Springer, 2012, pp. 818–829.

- [134] S. Leclaire, A. Parmigiani, B. Chopard, and J. Latt, “Three-dimensional lattice Boltzmann method benchmarks between color-gradient and pseudo-potential immiscible multi-component models,” International Journal of Modern Physics C, vol. 28, no. 07, p. 1750085, 2017.
- [135] T. Inamuro, M. Yoshino, and F. Ogino, “A non-slip boundary condition for lattice Boltzmann simulations,” Physics of fluids, vol. 7, no. 12, pp. 2928–2930, 1995.
- [136] J. Latt, “Hydrodynamic limit of lattice Boltzmann equations,” Ph.D. dissertation, University of Geneva, 2007.
- [137] S. Hou, Q. Zou, S. Chen, G. Doolen, and A. C. Cogley, “Simulation of cavity flow by the lattice boltzmann method,” Journal of Computational Physics, vol. 118, no. 2, pp. 329–347, 1995. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0021999185711035>
- [138] “Source code of LaBCof,” 2022. [Online]. Available: <https://github.com/LaBCof/LaBCof>
- [139] W. L. Oberkampf and T. G. Trucano, “Verification and validation in computational fluid dynamics,” Progress in Aerospace Sciences, vol. 38, no. 3, pp. 209–272, apr 2002. [Online]. Available: <http://linkinghub.elsevier.com/retrieve/pii/S0376042102000052>
- [140] E. Kreyszig, Advanced Engineering Mathematics 9th Edition with Wiley Plus Set. John Wiley & Sons, 2007.
- [141] S. Mohammed, D. Graham, and T. Reis, “Assessing moment-based boundary conditions for the lattice Boltzmann equation: A study of dipole-wall collisions,” Computers & Fluids, vol. 176, pp. 79–96, 2018.
- [142] S. Leclaire, J. Latt, D. Vidal, and F. Bertrand, “Multiphase periodic pressure difference boundary condition enhanced by a proportional-integral-derivative controller for the lattice Boltzmann method,” International Journal for Numerical Methods in Fluids, vol. 88, no. 9, pp. 434–446, 2018.
- [143] M. Goodarzi, M. R. Safaei, A. Karimipour, K. Hooman, M. Dahari, S. N. Kazi, and E. Sadeghinezhad, “Comparison of the finite volume and lattice Boltzmann methods for solving natural convection heat transfer problems inside cavities and enclosures,” Abstract and Applied Analysis, vol. 2014, 2014.

- [144] O. Shardt, “Comparison of finite volume and lattice Boltzmann methods for multicomponent flow simulations,” Canadian Journal of Chemical Engineering, vol. 98, no. 1, pp. 44–53, 2020.
- [145] X. Tang, C. Xue, T. Wu, and T. Wang, “Optimization of parallel program based on lattice BGK method,” in Proceedings of the ACM Turing Celebration Conference-China, 2019, pp. 1–7.
- [146] P. Tang, A. Song, Z. Liu, and W. Zhang, “An implementation and optimization of lattice boltzmann method based on the multi-node cpu+ mic heterogeneous architecture,” in 2016 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC). IEEE, 2016, pp. 315–320.
- [147] L. Axner, J. Bernsdorf, T. Zeiser, P. Lammers, J. Linxweiler, and A. Hoekstra, “Performance evaluation of a parallel sparse lattice boltzmann solver,” Journal of Computational Physics, vol. 227, no. 10, pp. 4895–4911, 2008. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S002199910800051X>
- [148] Z. Shang, M. Cheng, and J. Lou, “Parallelization of lattice Boltzmann method using mpi domain decomposition technology for a drop impact on a wetted solid wall,” International Journal of Modeling, Simulation, and Scientific Computing, vol. 5, no. 02, p. 1350024, 2014.
- [149] T. Puurtinen, J. I. Toivanen, K. Mattila, J. Hyväluoma, R. W. Nash, P. V. Coveney, and J. Timonen, “Coupling of lattice-Boltzmann solvers with suspended particles using the mpi intercommunication framework,” Advances in Engineering Software, vol. 111, pp. 52–57, 2017.
- [150] L. Xu, A. Song, Z. Liu, and W. Zhang, “Scalable parallel algorithm of finite difference lattice boltzmann method.” IAENG International Journal of Computer Science, vol. 44, no. 4, 2017.
- [151] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. K. Panda, “Gpu-aware mpi on rdma-enabled clusters: Design, implementation and evaluation,” IEEE Transactions on Parallel and Distributed Systems, vol. 25, no. 10, pp. 2595–2605, 2013.
- [152] M. Namvar and S. Leclaire, “Simple lattice boltzmann method algorithm with low memory usage,” Journal of Computational Science, vol. 62, p. 101723, 2022. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1877750322001168>

- [153] C. Zhu, J. Liu, L. Feng, and X. Deng, “Research on the simulation of pf-lbm model based on mpi+ cuda mixed granularity parallel,” Aip Advances, vol. 8, no. 6, p. 065017, 2018.
- [154] C. Riesinger, A. Bakhtiari, M. Schreiber, P. Neumann, and H.-J. Bungartz, “A holistic scalable implementation approach of the lattice Boltzmann method for CPU/GPU heterogeneous clusters,” Computation, vol. 5, no. 4, p. 48, 2017.
- [155] W. Xue, C. W. Jackson, and C. J. Roy, “Multi-CPU/GPU parallelization, optimization and machine learning based autotuning of structured grid CFD codes,” p. 0362, 2018.
- [156] D. Li, C. Xu, Y. Wang, Z. Song, M. Xiong, X. Gao, and X. Deng, “Parallelizing and optimizing large-scale 3d multi-phase flow simulations on the tianhe-2 supercomputer,” Concurrency and Computation: Practice and Experience, vol. 28, no. 5, pp. 1678–1692, 2016.
- [157] B. Mohr, “Introduction to parallel computing,” Multiscale Simulation Methods in Molecular Sciences Lecture Notes, p. 535, 2006.
- [158] B. McLaughlin, G. Pollice, and D. West, Head First Object-Oriented Analysis and Design: A Brain Friendly Guide to OOA&D, ser. Head First. O’Reilly Media, 2006. [Online]. Available: <https://books.google.ca/books?id=vWI5Jx5VRgEC>
- [159] M. Namvar and S. Leclaire, “LaBCof: Lattice Boltzmann boundary condition framework,” Computer Physics Communications, vol. 285, p. 108647, 2023. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0010465522003666>
- [160] “Specifications of niagara cluster,” 2022. [Online]. Available: <https://docs.alliancecan.ca/wiki/Niagara>