GPU-based Multiscale Simulation to Model Active Matter Hydrodynamics in Fluid Medium

A Thesis

submitted to
Indian Institute of Science Education and Research Pune in partial fulfillment of the requirements for the BS-MS Dual Degree Programme

By
T N Suhal Siva Ratan

IISER PUNE

Indian Institute of Science Education and Research Pune
Dr. Homi Bhabha Road, Pashan, Pune 411008,
INDIA.

April, 2023

Supervisor: Apratim Chatterji

Certificate

This is to certify that this dissertation entitled **"GPU-based Multiscale Simulation to Model Active Matter Hydrodynamics in Fluid Medium"** towards the partial fulfillment of the BS-MS dual degree programme at the Indian Institute of Science Education and Research, Pune represents study/work carried out by T N Suhal Siva Ratan Indian Institute of Science Education and Research under the supervision of Apratim Chatterji, Associate Professor, Department of Physics, during the academic year 2022-2023.

Chatterji
11 May, 2023.

Apratim Chatterji

Committee:

Apratim Chatterji

Vijayakumar Chikkadi

Declaration

I hereby declare that the matter embodied in the report entitled **"GPU-based Multiscale Simulation to Model Active Matter Hydrodynamics in Fluid Medium"** are the results of the work carried out by me at the Department of Physics, Indian Institute of Science Education and Research, Pune, under the supervision of Apratim Chatterji and the same has not been submitted elsewhere for any other degree. The CPU version of the Multiscale simulation which was parallelized and optimized for the parallel processors as my MS thesis, was developed over the last 4 years by Nishant Baruah, Adrian Pinto and Tejal Agarwal.

[illegible]

T N Suhal Siva Ratan

This Thesis is dedicated to my parents

Acknowledgments

I am deeply grateful to Dr. Apratim Chatterji for supporting me as my supervisor during my Master's project. His exceptional guidance, unwavering support, and invaluable expertise shaped my research and academic journey. I am also thankful to Dr. Manish Modani and Hemant Giri from NVIDIA for their insightful assistance in helping me understand parallel programming tools, which was crucial in successfully completing my project. I would also like to thank Dr. Vijayakumar Chikkadi, whose valuable research provided me with an engaging research topic for my thesis.

I'm extremely grateful to my parents for their love, support, and encouragement throughout my journey. Their sacrifices and guidance have been indispensable in shaping me as a person. I would also like to thank my brother Chidhvilas, for being such an amazing brother who always believed in me and supported me, no matter what.

I would also like to thank IISER Pune and C-DAC for providing invaluable support and resources. In particular, I am thankful for the opportunity to access the high-performance PARAM-Brahma cluster, which helped me to complete my project.

I express my sincere gratitude to the Infosys Foundation Scholarship and DDN Scholarships for providing me with the financial support that has been crucial in financing my studies. Their generous contributions have made pursuing my academic aspirations possible.

I would also like to thank Shreerang Pande and Adrian Pinto for providing me with a deeper understanding of the physics underlying the problem. I would like to extend my sincere thanks to all my lab mates for their support, and encouragement throughout my research journey.

Finally, I express my gratitude to all of my friends for the valuable memories that we have shared throughout the years.

Contents

List of Figures

List of Tables

Abstract

Active matter systems are soft matter systems characterized by many interacting active particles, such as bacterial suspensions or Janus particles. While dry active matter systems have been studied extensively, hydrodynamic interactions in active matter systems remain an open area of investigation. This is particularly challenging because modeling these interactions requires simulating a large number of fluid particles in addition to the active colloidal particles. In this thesis, the main focus was placed on parallelizing this simulation suitable for a GPU. We benchmark and verify the different components of the model and also verify the results displayed by the complete model. The resulting parallelized simulation created in this project will be used to investigate the behavior of active matter systems under different conditions and parameter settings, with a focus on understanding the role of hydrodynamic interactions in emergent phenomena.

Chapter 1

Introduction

1.1 Background

1.1.1 What is Active Matter?

Active matter is a unique class of materials or systems consisting of self-propelled components that utilize stored or ambient energy for directed motion[1,2]. Representative examples of active matter include bacterial swarms, fish schools, and bird flocks.

In this thesis, the active matter is studied through the framework of statistical physics to comprehend how many interacting particles' collective behavior leads to emergent phenomena[3]. Due to this activity, active matter systems display a vast array of intriguing and often surprising behaviors, such as phase transitions[4], pattern formation, and long-range correlations.

Understanding active matter systems is essential, as numerous biological systems fall under this category. Such systems exhibit emergent active behaviors and complex self-assembly structures that cannot be explained by traditional equilibrium statistical mechanics since they are out of equilibrium[5]. However, simulating these systems through traditional CPU programming can be computationally expensive, especially for larger particle systems. Therefore, to overcome this challenge, we need to perform independent computations concurrently. Thus, developing active matter systems suitable for parallel processors can facilitate easier computations.

1.1.2 Hydrodynamics of Microswimmers(E.Coli)

Microswimmers refer to both natural or artificial systems that can move or swim in a fluid environment on a small scale. Their motion mechanisms may involve self-propulsion, fluid convection, or external forces. At such small-length scales, the viscous forces of the surrounding medium take precedence over the inertial forces. So, the physics governing the motion of microswimmers operates differently than what we usually observe/experience in our day-to-day life. Equation 1.1 shows that the Reynolds number for microswimmers will be way too small, where the viscous forces dominate the motion of the microswimmer.

$$Re = \frac{Inertial\ Forces}{Viscous\ Forces} = \frac{\rho u L}{\mu} \tag{1.1}$$

Where,

ρ = Density of the fluid

u = speed of the fluid flow

L = Characteristic Length

μ = kinetic viscosity of the fluid

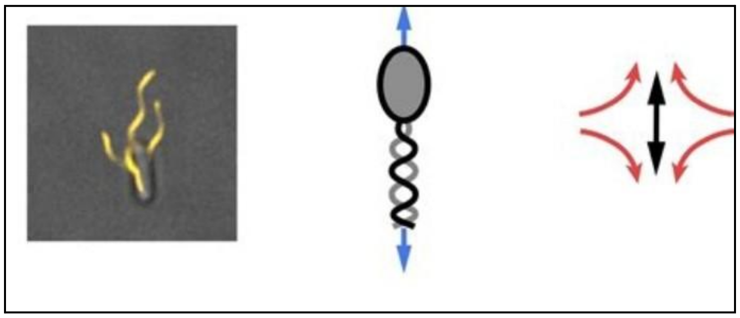


Figure 1.1: E.Coli acts as a microswimmer by pushing back the fluid with its helical flagella. Here the force by the E.coli is in black, and the flow field near the E.coli is shown in orange. This figure is cited from[35]

In our specific problem, we are utilizing the concept of modeling E.Coli as a microswimmer that propels itself through water using flagella[6]. By rotating these flagella, the E.Coli bacteria generates a propulsive force that propels it forward. As a result, the motion of the bacteria induces a flow field around its cell body. This flow field can be mathematically derived using the

Navier-Stokes equations, which take into account the physical properties of the fluid and the motion of the microswimmer[7]. Therefore, by analyzing the flow field generated by the E.Coli bacteria, we can gain insights into the mechanics of its propulsion and potentially inform the design of artificial microswimmers.

$$\rho \frac{du}{dt} = -\nabla p + \mu \nabla^2 u + F \tag{1.2}$$

$$\nabla u = 0 \tag{1.3}$$

By solving equations 1.2 and 1.3, we can obtain the flow pattern generated by E.Coli bacteria and analyze its impact on other microswimmers in the fluid. The hydrodynamic interactions between these microswimmers can give rise to a range of phenomena, including particle alignment in a particular direction and the development of intricate structures that surpass the size of individual organisms(self-assembly), all of which result from these indirect communications[8].

1.1.3 Modelling Microswimmers as Janus Particles

A Janus particle is a type of particle of micro/nanoscale that has two distinct regions or faces with different chemical or physical properties. These two regions can have different surface chemistry, charge, hydrophobicity, or optical properties, among other characteristics. Janus particles can be considered active matter because they have the ability to self-propel or move in response to their environment without the need for an external force. This movement is typically driven by chemical or physical gradients in their surrounding environment[9,10].

This motivated us to use Janus particles to model our microswimmer species, i.e., E.Coli, because of their self-propelling nature. When as a collective, they show swarming nature, phase separation, and cluster formation[11,12].

1.1.4 Passive Particles' Behavior in the Presence of Active Particles

Passive particles typically undergo Brownian motion as a result of collisions with fluid particles, as they lack an active force to propel them forward. However, when present in an active bath, passive particles have demonstrated the ability to aggregate and cluster, as mentioned in the literature[13,14] and experiments conducted at the Soft Matter and Active Matter Lab in IISER Pune. This phenomenon provides an alternative perspective for investigating the motility-induced self-assembly of colloids[15].

1.2 Physical System to be Modeled for the Parallel Processors

Our system was designed as described in Section 1.1, consisting of colloidal particles(both active and passive) in the fluid medium. Active particles in this system represent bacteria. We model for the following interactions to simulate the desired system

1. Interactions among the active particles
2. Interactions among the passive particles
3. Active-passive interactions among the colloidal particles
4. Self-propulsion of the active matter
5. Interactions between the fluid particles, which represent the hydrodynamics
6. Fluid-particle interactions through which the momentum transfer occurs.

To compute all these interactions until the system reaches equilibrium would be time-consuming. So, these computations can be done in parallel to make the simulation time-efficient.

1.3 Parallel Programming

1.3.1 Introduction

Parallel programming is the technique of separating a computational problem into smaller tasks that can be performed simultaneously on multiple cores to reduce the computation time drastically, involved in successive identical mathematical operations[16]. Parallel programming aims to utilize the available resources to optimize the application's performance and reduce the time taken to complete a task. However, parallel programming can also be challenging, as one needs to assess issues such as load balancing, data distribution, and data synchronization across the device(s).

To implement parallelization, one can either use multiple cores of CPU or use a Graphic Processing Unit (GPU) for computations. In this case, we have used GPU programming to improve the application performance.

1.3.2 Graphic Processing Unit

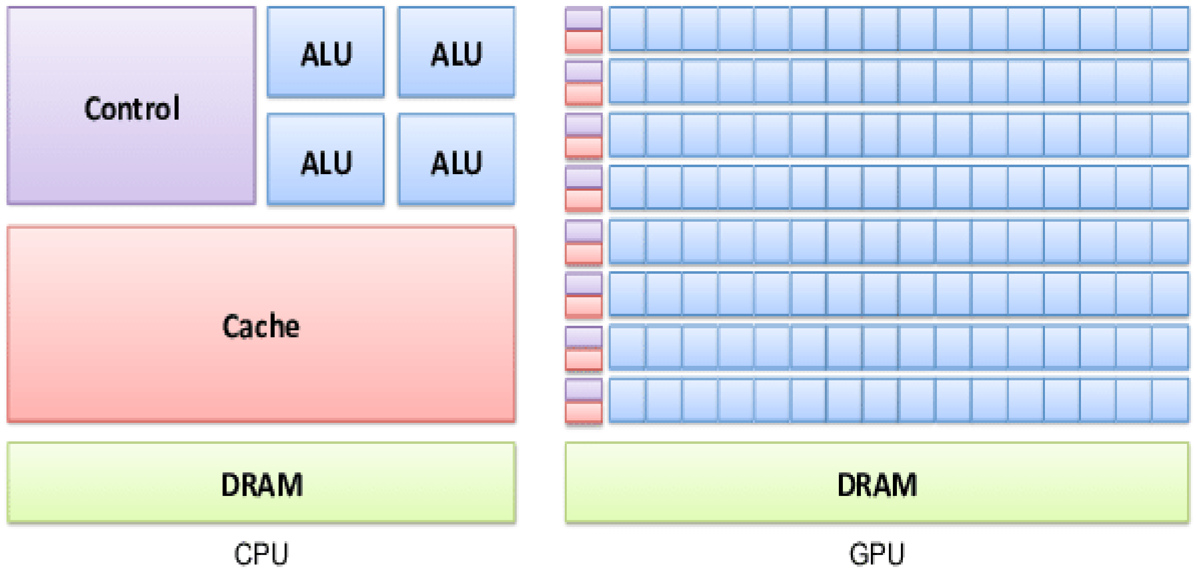


Figure 1.2: CPU and GPU architecture. Comparing the cores present in the CPU Vs GPU.[17]

GPUs are highly parallel processors with a large number of cores operating on a shared memory that can perform multiple calculations concurrently[17,18]. GPU can accelerate the performance of an application by reducing the load of some time-consuming tasks that can be run simultaneously on GPU cores. While the clock speed of each GPU core is slower than that of a CPU's core clock speed, a large number of cores on the GPU compensates for this and allows for high overall computational throughput. This makes them an ideal choice for computationally intensive scientific simulations, such as our Multiscale simulation, which combines both Molecular Dynamics (MD) and Multiparticle Collision Dynamics (MPCD).

A basic overview of the GPU's architecture and the procedure to implement the parallelization is explained in the next chapter.

1.4 Goal of the Project

The goal of this thesis is to parallelize an existing multiscale simulation system that combines molecular dynamics (MD) and multiparticle collision dynamics (MPCD) simulations for multiscale modeling of hydrodynamic interactions, utilizing GPU acceleration. The objective is to optimize the performance and scalability of the system while verifying the hydrodynamic behavior of complex fluids and their interactions with the active matter particles that are present in the fluid medium. By implementing the simulation on a GPU, this work seeks to achieve faster computation times and enable larger simulations, thus providing new insights into the dynamics of these systems.

1.5 Thesis Outline

The proposed structure of this thesis is as follows: Chapter 2 will focus on describing the methods and algorithms used for modeling the various interactions in the system, as previously outlined. This will include detailing the parallelization techniques employed for simulating the entire system. In Chapter 3, we will begin by discussing the benchmarking and verification of the simulation techniques outlined in Chapter 2. This verification will be performed through comparison with pre-existing numerical simulations method developed by Soft and Living Matter Group, IISER-Pune.

Chapter 2

Methodology

In this chapter, before explaining the numerical simulation techniques, I will first provide an overview of GPU architecture and GPU programming tools such as OpenACC programming construct. Since GPU programming is a relatively new technology compared to traditional CPU/sequential programming, this introduction will provide readers with a fundamental understanding of parallelization and assist them in replicating the simulation accurately.

2.1 Overview of GPU and Parallel Programming Tools

2.1.1 Understanding GPU Architecture

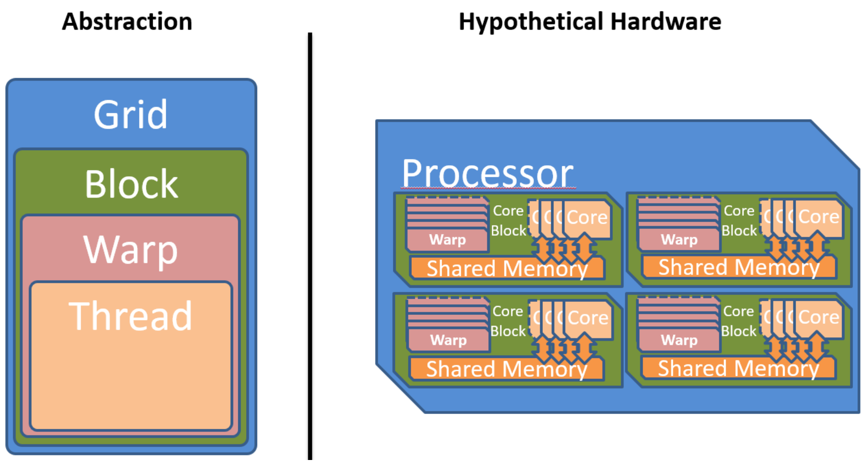


Figure 2.1: This figure provides a schematic of the hierarchy of execution and memory that happens in GPUs[19].

A GPU is made of a large number of highly efficient processing cores that work together to perform complex calculations on large amounts of data. This allows us to perform complex scientific simulations on GPU and make them time efficient. The following terms describe the organization of computations on the GPU and how they are mapped to the hardware[18,20,21].

- Threads: A thread is the smallest unit of work on a GPU. The computations are executed in parallel for threads. So, threads are used to perform independent calculations.
- Warps: A warp is basically a small collection of threads that are executed together in lockstep.
- Blocks: A block can be defined as a group of threads communicating with one other, operating with the help of shared memory. So threads in a block can synchronize and share data if interthread communications are required.
- Grids: A grid is a grand assemblage of blocks that run in parallel. Based on the complexity/requirements of the computation involved in the program, we can use one, two, or three-dimensional grids accordingly.

I have not focused on the memory hierarchy of the GPU as we will be using a programming construct called OpenACC which does not require in-depth knowledge of the GPU.

2.1.2 Programming Construct Vs. Programming Language

A programming language is a set of rules and instructions that define how a programming application is written and executed. It is a formal language with a standard set of syntax and semantics that allow programmers to write computer programs that can be executed on the hardware used.

On the other hand, A programming construct is a characteristic or tool within a programming language that provides a specific functionality or behavior of the programming application[22]. It is a way of expressing a specific concept or idea within the context of a programming language. They are basically keywords that are part of a programming language.

2.1.3 Open Accelerator (OpenACC)

In this project, we used a parallel programming construct named OpenACC for the Fortran programming language. With the help of this construct, the user can use a set of directives, equivalent to pragmas(compiler directives), that can be added to existing code written in C, C++, and Fortran to specify regions of code that can be offloaded from the CPU to an accelerator, such as a GPU[23,24]. These directives can be used to determine loops, data regions, and other code sections that can be parallelized without requiring a complete rewrite of the codebase.

2.1.4 Levels of Parallelism in OpenACC

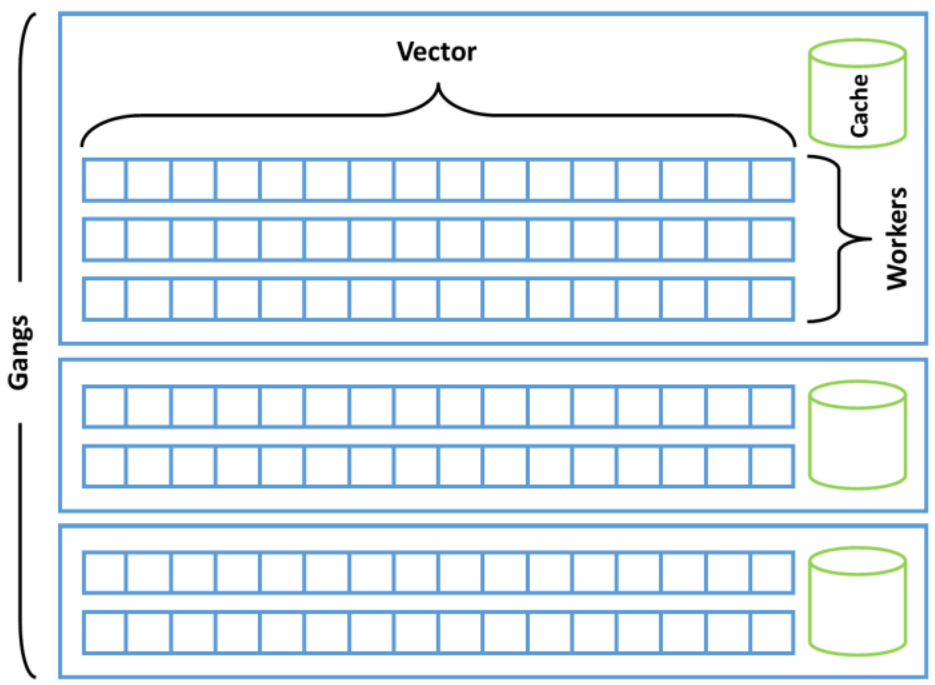


Figure 2.2: Levels of Parallelism in OpenACC. Each gang will have a cache memory that can be shared by all the workers and vectors of that particular gang[24].

OpenACC provides three levels of parallelism known as gang, worker, and vector. These levels of parallelism can be described as follows:

- Gangs: A gang is a group of one or more workers that execute the same block of code concurrently on the device. This level of parallelism is utilized to parallelize the outermost loop of a nested loop structure.
- Workers: Workers are individual threads that execute the same code block concurrently within a gang. The worker level of parallelism lies between gangs and vectors.
- Vectors: Vectors are a set of elements in memory that can be processed simultaneously by a single instruction. In OpenACC, vectors are employed to parallelize the inner loop of a nested loop structure.

In short, we can say that for a given task, a gang employs the required number of workers, where each worker performs the job on a particular length which we call a vector. Based on the complexity of the loop, we can employ the required level of parallelism accordingly. In section 2.7, we have discussed how a CPU/sequential application can be parallelized using OpenACC.

2.2 Molecular Dynamics (MD)

This section outlines the techniques employed in the implementation of Molecular Dynamics suitable for a parallel processor. Molecular dynamics is a computational method used to simulate the behavior of particles(atoms and molecules) over time. This numerical method is widely used in materials science, chemistry, and biochemistry, among other fields, to learn about the systems that are complex to solve analytically. However, it would be computationally expensive to model interactions through traditional/sequential programming as updates are computed one particle after the other. Therefore, employing parallelization techniques can significantly improve the efficiency of this simulation.

During molecular dynamics simulations, the particles' positions and velocities are regularly adjusted by computing the forces acting on them using a potential energy function that characterizes the particles' interactions. As the simulation progresses, these updates are repeated at each time step to generate particle trajectories, allowing for a comprehensive analysis of system dynamics and thermodynamics. This process involves calculating the positions, velocities, and forces of all particles with each time step, providing a detailed understanding of their behavior.

2.2.1 MD Algorithm

(a) Initialization of positions and velocities

In Molecular Dynamics simulations, initialization is one of the crucial moments that need to be executed carefully. While initializing the positions of the particles, one must not initialize particles with any overlaps with respect to one another. This can be done by initializing all the positions of these particles on a lattice or at random, where two particles won't coincide with one another.

While initializing velocities, we must keep in mind that the velocities initialized should correspond to the given fixed temperature. We know that at thermal equilibrium, mean-square velocity along any component can be written as

$$< V_i^2 > \; = k_B T/m \qquad (2.1)$$

where i = x,y,z

For this, we can call velocities randomly from -0.5 to 0.5 from the uniform distribution. Now V_i~Uniform(-0.5,0.5) has a variance of 1/12. So, to obey equation 2.1, the obtained velocities are multiplied with the velocity_scalefactor shown below.

$$velocity\_scalefactor = \sqrt{12 k_B T/m} \qquad (2.2)$$

As the Centre of Mass(COM) of the system does not move, we need to shift COM motion to zero. For this, we can calculate the average velocity of each component from the velocities obtained above and shift velocities such that the mean is zero.

(b) Velocity-Verlet Algorithm

To implement the molecular dynamic simulations to our desired problem, we used the Velocity-Verlet Algorithm. With the help of the Velocity-Verlet Algorithm, one can compute the positions and velocities of the particles in the next time step based on their positions and velocities of the current time step. The following equations are used to update the positions and velocities of the particles based on the Velocity-Verlet Algorithm:

$$\bar{x}(t+\Delta t) = \bar{x}(t)\Delta t + \frac{1}{2}\bar{a}(t)\Delta t^2 \tag{2.3}$$

$$\bar{v}(t+\Delta t) = \bar{v}(t) + \frac{\bar{a}(t)+\bar{a}(t+\Delta t)}{2}\Delta t \tag{2.4}$$

(or)

$$\bar{x}(t+\Delta t) = 2\bar{x}(t)\Delta t + \bar{a}(t)\Delta t^2 - \bar{x}(t-\Delta t) \tag{2.5}$$

$$2\bar{v}(t)\Delta t = \bar{x}(t+\Delta t) - \bar{x}(t-\Delta t) \tag{2.6}$$

Here,

$\vec{x}$ = Position vector of the particles

$\vec{v}$ = Velocity vector of the particles and

$\vec{a}$ = Acceleration vector of the particles

So, if the positions and velocities of all particles are known from the previous time-step, the potential energy can be calculated based on particle positions, which can then be utilized to determine the forces acting between particles. There are two options available for updating the positions and velocities of the particles: equations 2.3 and 2.4 or equations 2.5 and 2.6, depending on the researcher's preference.

(c) Interparticle Forces and the Potentials

The Lennard-Jones(LJ) potential was utilized to simulate the interactions between particles. By using this potential, the pairwise potential between all particles in the system can be determined, ultimately providing the system's total potential energy. To calculate the force acting on a particle with respect to another particle, one can differentiate the interaction potential between these two in the system.

$$F_{ij} = -\nabla U_{ij} \qquad (2.7)$$

Then, the total force acting on the particle would be the sum of all the forces between that particle and all the remaining particles in the system.

$$F_i = \Sigma F_{ij} \qquad (2.8)$$

However, this would be computationally more expensive because, for a system of n particles, we need to perform n x n calculations for a single time step. Though we implement parallelization for this system, if given a large number of particles, even the hardware won't be able to help. So, in order to reduce the computation time, we should neglect potentials/forces that are far away from the particle (i.e., set the potentials and forces due to the farther particles as zero). Because of this cutoff distance, we can modify the Lennard-Jones potential as the truncated Lennard-Jones potential, which can be written as,

$$V_{\mathrm{LJT}}(r) = \begin{cases} 4\varepsilon[(\frac{\sigma}{r})^{12} - \frac{\sigma}{r})^{6}] & r \leq r_{\mathrm{c}} \\ 0 & r > r_{\mathrm{c}} \end{cases} \qquad (2.9)$$

From the above equation, we can calculate the truncated Lennard-Jones force (F_{LJT}) by differentiating equation 2.8 as shown in equation 2.7 and get,

$$F_{\mathrm{LJT}}(r) = \begin{cases} 4\varepsilon[12\frac{\sigma^{12}}{r^{13}} - 6\frac{\sigma^6}{r^7}] & r \leq r_{\mathrm{c}} \\ 0 & r > r_{\mathrm{c}} \end{cases} \tag{2.10}$$

Where,

σ = Diameter of the particle

r = Distance between the ith and jth particle

r_c = Cutoff-distance beyond which the potentials and forces are neglected.

Still, we must confirm the continuity of force at the mentioned cutoff distance. To correct this, we can subtract the force at r_c from the LJ force calculated for the particles when $r \leq r_c$. This can be anointed as the truncated and shifted Lennard-Jones force(F_{LJTS}) and can be written as

$$F_{\mathrm{LJTS}}(r) = \begin{cases} F_{\mathrm{LJ}}(r) - F_{\mathrm{LJ}}(r_{\mathrm{c}}) & r \leq r_{\mathrm{c}} \\ 0 & r > r_{\mathrm{c}} \end{cases} \tag{2.11}$$

Where,

$$F_{\mathrm{LJ}}(r) = 4\varepsilon[12\frac{\sigma^{12}}{r^{13}} - 6\frac{\sigma^6}{r^7}]$$

This corresponds to the following form of the potential, called the truncated and shifted Lennard-Jones Potential(V_{LJTS}) or Weeks-Chandler-Andersen potential (V_{WCA}),

$$V_{\mathrm{LJTS}}(r) = \begin{cases} V_{\mathrm{LJ}}(r) - (r - r_{\mathrm{c}})F_{\mathrm{LJ}}(r_{\mathrm{c}}) - V_{\mathrm{LJ}}(r_{\mathrm{c}}) & r \leq r_{\mathrm{c}} \\ 0 & r > r_{\mathrm{c}} \end{cases} \tag{2.12}$$

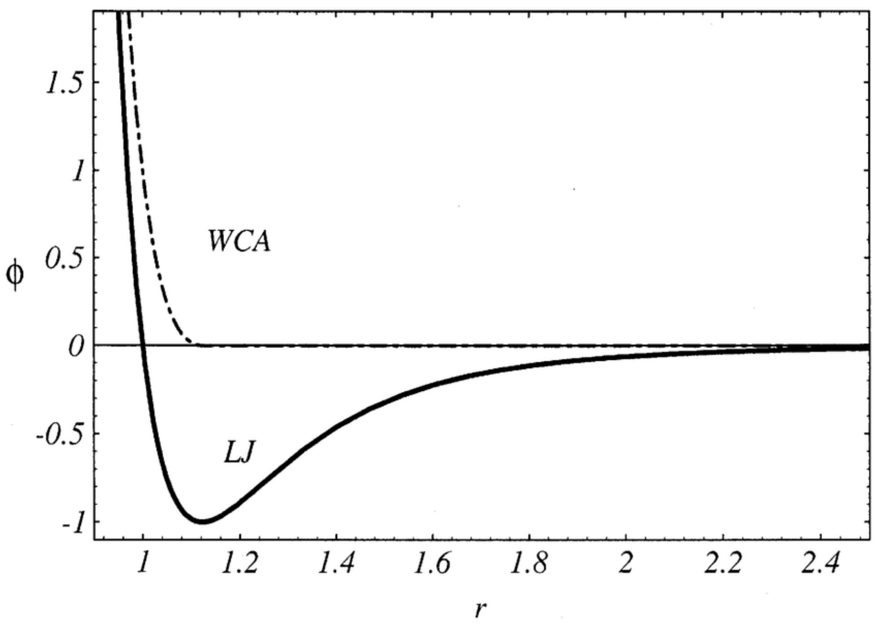


Figure 2.3: Lennard-Jones Potential Vs. Weeks-Chandler-Andersen potential (V_{WCA})[25].

(d) Neighbour Lists

Now, the particles that are inside the cutoff distance need to be accessed easily in order to reduce the computation cost and also to reduce the number of threads considering the particles only where these calculations of these LJ potentials and forces between the particles lead to a non zero value thus utilizing the parallel processor effectively under its limitations. To implement this, we created a list such that for a given particle, we dropped the particles that were slightly greater than the cutoff distance r_c and named the list as the Verlet neighbour lists.

$$r_m - r_c \geq v_{max}\Delta t \qquad (2.13)$$

(e) Thermostat

As the system needs to be simulated to mirror the real world, it is crucial to maintain the temperature of the system to be constant for a given number of particles and the size of the simulation box throughout the simulation time, which describes a canonical ensemble. A thermostat was introduced to balance the system's temperature. After a small number of iterations (let us assume some number g), the velocities of all the particles in the system were rescaled. This process was repeated for every g iterations. Rescaling of these particles was done according to the equipartition theorem.

$$\langle \frac{1}{2} mV^2 \rangle = \frac{3}{2} k_B T \qquad (2.14)$$

From the above equation 2.13, we can find the temperature of the system from the average velocity of the particles and estimate the rescaling factor with respect to our desired(fixed) temperature. The obtained rescaling factor is multiplied by all the particle velocities in the system.

2.2.2 Implementation of MD using OpenACC

1. The positions of all the particles were initialized such that no two particles overlapped with each other. This initialization step is carried out in the CPU as this step won't be repeated again.
2. The velocities were initialized, as explained in section 2.2.1 (a). This process was also initialized on the CPU(sequentially).
3. The generated arrays of positions and velocities by the CPU were copied to the GPU memory, and the other arrays to carry out the MD simulation(force, neighbour lists) were created in the GPU memory.
4. Based on the positions of the particles, neighbour lists were generated. (In sequential programming, this process takes $^{N}C_2$ steps for N particles. Now, these $^{N}C_2$ computations are assigned to $^{N}C_2$ threads where this operation takes place concurrently.)
5. From the obtained neighbour lists, inter-colloidal forces between these particles were calculated parallelly using OpenACC.

6. From the obtained forces, positions and velocities of these particles were updated based on the Velocity-Verlet Algorithm. As particle updates are independent of one another, N computations can be run at the same time on a parallel processor, thus reducing the time significantly by updating all of them parallelly.
7. Step 5 and Step 6 were iterated over and over until we reached the desired number of iterations.
8. Neighbour lists were updated for every 40 MD iterations
9. The thermostat was implemented for every 100 iterations to rescale the particle velocities
10. To validate the Molecular Dynamics simulation on the parallel processor. Data were collected for the following quantities
    a. Potential Energy, Kinetic Energy, and Total Energy of the system for every 10 iterations.
    b. Pair correlation of the system particles for every 100 iterations.
    c. Velocities of all particles for every 100 iterations to check whether it follows Maxwell-Boltzmann distribution.

2.3 Multiparticle Collision Dynamics(MPCD)

Multiparticle Collision Dynamics(MPCD) is a simulation technique that can be used to model the behavior of a fluid. Fluid can be represented as point-like particles with some mass and velocity. This particle-based approach can be used to simulate the dynamics of the fluid through the stochastic collisions that occur among these particles[26]. MPCD is highly beneficial for examining non-equilibrium systems, as it can manage complex geometries and boundary conditions. Furthermore, it has been shown to model hydrodynamics accurately with thermal fluctuations[26,27]. It can be employed in a wide range of research fields, including materials science, chemical engineering, and biophysics.

Coming to our problem, the bacteria(E.Coli) interact with one another in a fluid medium through hydrodynamic interactions. Since we have used MD to model our bacteria as active particles, we need to confirm whether MD can be integrated with MPCD. The developers of the MPCD technique, Malavenets, and Kapral, demonstrated in their research that MPCD has the potential to function as a mesoscopic-level model for solvent dynamics, providing a comprehensive depiction of the system[28]. Prior work of the Soft and Living Matter Group, IISER Pune focused on simulating active matter particles through MD and coupled with fluid dynamics simulated through MPCD to develop a numerical method for hydrodynamic interactions with active particles. This work has mainly focused on redesigning this coupling to be appropriate for parallel processors and to make this simulation time-efficient by reducing the runtime.

2.3.1 MPCD Algorithm

A fluid modeled through MPCD mainly has two steps:

- Stochastic Collision step
- Deterministic Streaming Step

(a) Stochastic Collision step

The collision rule involves splitting the simulation box into small cubic cells called collision boxes. All fluid particles are assigned to these collision boxes such that the average number of particles per cell is p. During each collision step, we shift to the center of the mass frame for each cell and find the average velocity of the cell by

$$\vec{u}_p = \sum_{i=1}^{p} \frac{v_i}{p} \qquad (2.15)$$

With respect to $\vec{u}_p$ relative velocity for each fluid particle of that cell were calculated, and were rotated randomly along a random vector with some angle α(to achieve this a rotation tensor is calculated for the cell). This process was repeated for all other cells, and each cell direction was randomized. Then the velocity of the particles for a given collision box was calculated as

$$\vec{v} = \vec{u}_p + rot * (\vec{v} - \vec{u}_p) \qquad (2.16)$$

Where

$\vec{v}$ = velocity vector of the fluid particle

$\vec{u}_p$ = velocity of the collision box

rot = rotation tensor calculated for the given collision box

It was shown by Ihle and Kroll that the Galilean invariance is lost because of partitioning the system into the collision cells[29] during the collision step. To correct this, the grid is randomly shifted before performing the collision step.
After this, we move to the next step, i.e., the ‘Deterministic Streaming Step’.

(b) Deterministic Streaming Step

In this step, particles can move freely inside the simulation box. The positions of the fluid particles are updated from time t to Δt by following

$$\vec{r}_i(t + h) = \vec{r}_i(t) + \Delta t\, \vec{v}_i(t) \qquad (2.17)$$

Where,

$\vec{r}_i$ = position of the ith fluid particle

$\vec{v}_i$ = velocity of the ith fluid particle

(c) Thermostat in MPCD

In MPCD, a comparable thermostat to MD was employed, which is a global thermostat that rescales velocity at the level of collisional cells. During each collision step, the velocities of all fluid particles were rescaled based on the average cell velocity($\vec{u}_p$) that had already been rescaled.

(d) Angular Momentum Conservation

The MPCD simulation implementation does not ensure the conservation of angular momentum, unlike energy and linear momentum conservation. This lack of conservation could result in unrealistic behavior in systems with rotating fluid velocity fields. To rectify this issue, an extra step can be introduced to conserve angular momentum in MPCD. However, this correction is not expected to cause any significant deviation from the results obtained using the non-angular momentum conserving version, known as MPC-SRD-a (in contrast to the angular momentum conserving version, MPC-SRD+a), which was used in previous work. As they had successfully obtained accurate results using the MPC-SRD-a version, the same technique was employed even in the parallelization of this simulation.

(e) Periodic Boundary Conditions(PBC)

Periodic boundary conditions (PBCs) are the most common technique used in the simulations, which helps us to mimic the behavior of an infinite system. In PBC, the simulation box is copied infinitely in all the dimensions provided by the user/researcher. So, when the particle leaves the simulation box on one side when positions are updated, it will re-enter from the other side of the box. In this work, PBC was implemented in both MC and MPCD simulations.

By implementing periodic boundary conditions, we can,

- Mimic bulk behavior: As the simulation box is infinitely repeated throughout the space, it allows us to study properties of materials, thermodynamic properties, and phase behaviour depending on the bulk behaviour of the system.
- Increase simulation efficiency: Through PBC, we can simulate a bulk/infinite system with a small number of particles. Thus, reducing the computations required in our simulation.
- Reduces boundary effects: In a simulation box, particles experience different forces when compared to the particles that are in the center of the box. This might lead to artificial results that won't explain the system we desire. Implementing PBC will neglect the boundary effects of the simulation.

2.3.2 Implementation of MPCD (without MD)

1. Positions and velocities of fluid particles were initialized in the simulation box (only the initialization procedure was run sequentially on the CPU).
2. After initialization, the particles were allowed to move based on the streaming step, as explained above.
3. A grid of collision boxes was created, and particles were classified into these collision boxes based on their position.
4. Then, the collision step was performed on the system.
5. Thermostat was implemented for every collision step.
6. Then again, the same process is repeated from the step 2.

2.4 Fluid-Particle Interactions

In this section, we discuss the interactions that occur between the colloidal particles and the fluid particles when MD and MPCD are coupled together. This can be described by explaining the boundary conditions between the different types of particles(active, passive, and fluid) that were implemented in this system.

2.4.1 Boundary conditions for the fluid flow

The fluid flow is affected by the presence of various interfaces. When a solid object obstructs the fluid's path, the fluid's velocity perpendicular to the solid surface is zero, while the velocity parallel to the surface depends on the type of interaction occurring at the solid-fluid interface. This difference in velocity between the fluid and solid surface is known as slip, which is governed by the Navier-Maxwell linear boundary condition,

$$\vec{u}_{||} = \lambda \vec{n} [\nabla \vec{u} + (\nabla \vec{u})^T]. (1 - \vec{n}\vec{n}) \qquad (2.18)$$

Where,

$\vec{u}$ = velocity vector of the fluid particle at the point of contact

$\vec{n}$ = Normal vector of the surface at the point of contact

λ = Approx. distance where $\vec{u}_{||}$ goes to zero

So, based on the Navier-Maxwell equation 2.18, the following boundary conditions are defined,

1. Stick (or no-slip) boundary condition: Here, the fluid's tangential velocity at the surface is zero relative to that of the boundary, resulting in the two surfaces sticking together.
2. Partial slip boundary conditions: In this case, the relative tangential velocity is reduced, but not to zero, at the fluid-solid interface.
3. Slip(or perfect slip) boundary conditions, on the other hand, dictate that the tangential velocity is not affected by the interface between the fluid and the surface.

In this system, stick boundary conditions were implemented between the colloid and the fluid.

2.4.2 Stochastic Boundary Conditions

Padding et al.[30] introduced a method for simulating stick boundaries called Stochastic Boundary conditions, which we have incorporated into our simulations.

$$P(\vartheta_n) \propto \vartheta_n\, exp(-\ \beta\vartheta_n^2) \tag{2.19}$$

$$P(\vartheta_t) \propto\ exp(-\ \beta\vartheta_t^2) \tag{2.20}$$

Implementation

- If there are any overlaps of MPC particles with the boundary, the point of contact was found by going back the half-time step.
- Tangential velocity of the particles (v_t) was chosen randomly from the distribution shown in equation 2.19
- Normal velocity of the velocities (v_n) was chosen randomly from the distribution shown in equation 2.20
- The obtained new velocities are updated for different fluid particles that are in contact with the colloidal particles and a sum of momentum all these particles was reversed and delivered to the colloid(see figure 2.4).

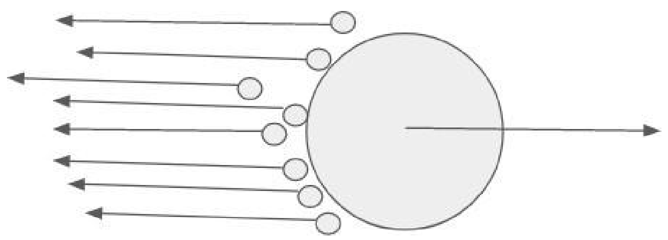


Figure 2.4: The fluid particles represented by the small spheres are all pushed back by the same momentum the active particle(large sphere) is being propelled forward. This mimics the pusher motion of E.Coli to move forward in the fluid medium.[36]

2.5 Active Particle Modeling

We know active particles move in a fluid due to their own activity. In order to simulate the bacteria as an active matter, first, we know how the bacteria acts as a microswimmer. The E. coli bacterium exhibits a run-and-tumble motion[4,31], which is a result of the rotation of motor proteins situated in the cell membrane. The flagella bundle up and move the bacterium forward when the motors rotate in the same direction, causing a run. However, these runs are stochastically interrupted by brief tumble events, where the motor rotation reverses, leading to flagella disassociation and rotation in the body(In our model, rotation happens with respect to the momentum transfer through fluid-colloid collisons). Following this, the motor rotation resumes in the earlier direction, causing the flagella to bundle up again and propel the bacterium forward once more.

2.5.1 Hydrodynamics and momentum transfer during the run phase

In this part, we explain the momentum transfer between the active particle and fluid and describe the active nature of the active particles. To understand the flow fields created by the different microswimmers(active matter)[32], we need to solve the Navier-Stokes equation for the time-independent flow,

$$\eta \nabla \vec{u} - \nabla p + \vec{f} = 0 \tag{2.21}$$

Solution for the above equation can be written as,

$$\vec{u} = -\underset{\substack{\Downarrow \\ \text{I}}}{\frac{A_{st}}{r}(\vec{I} + \hat{r}\hat{r}).\hat{y}} - \underset{\substack{\Downarrow \\ \text{II}}}{\frac{A_{str}}{r^2}(1-3(y/r)^2)\,\hat{r}} - \underset{\substack{\Downarrow \\ \text{III}}}{\frac{A_{sd}}{r^3}(\frac{\vec{I}}{3} - \hat{r}\hat{r}).\hat{y}} + \dots\dots\dots\dots \tag{2.22}$$

I. **Stokeslet**: It is a point force that describes flow generated by a small particle(in this case bacteria) in the fluid.

II. **Stresslet**: It is generated by a force dipole produced by the microswimmer in the fluid.

III. **Source Dipole**: In this case, the trajectory of the fluid flow field is in the closed loop.

The Stresslet fluid profile is prevalent among swimming micro-organisms such as E.coli bacteria. While the multipolar approach overlooks the flow field's nuances near the flagella, it has a distinct benefit in that higher-order multipoles diminish more rapidly as the bacteria moves away.

To mimic the stresslet fluid profile(i.e., to create force dipoles) in the system, forward momentum was given to the active particles, and fluid particles behind the active particle (a shell around a unit thickness) were given the same momentum in the opposite direction to that of the active particle. This activity facilitates the action particle to move forward and collide with other particles in front, and momentum is transferred to them.

2.6 Algorithm to implement the Multiscale simulation

The Multiscale(MD+MPCD) simulation was implemented such that for each MPCD step, 10 MD cycles take place. The procedure is explained as follows:

1. Positions and velocities of colloids(both active and passive) were initialized in the simulation box
2. Positions and velocities of fluid particles were initialized as point-size particles in simulation box
3. Fluid particles were classified into the newly created collision boxes based on their positions.
4. Neighbour lists were created for the colloids, and force calculations were made for these particles.

5. Initial direction of active particles in the system was evaluated.
6. MPCD step was initialized (in each MPCD step, the collision step happens once)
7. The inner loop for MD steps was initialized. This loop contains the streaming step of the fluid particles. As said above, 10 MD steps occur in one MPCD step. So, 10 MPCD streaming steps happen in one MPCD collision step.
8. In this inner loop MD procedure that was explained in subsection 2.2.1(updates of positions & velocities, neighbour lists, and force calculations) was executed along with the MPCD streaming step.
9. Fluid colloid collisions and momentum transfer due to collisions that happen with respect to the stochastic boundary conditions were also executed in this inner loop.
10. As active particles' orientation changes after the collision, updating the activity direction for each active particle was also implemented here.
11. Neighbour lists for MPCD was implemented for every 20 MD steps
12. Neighbour lists for MD were implemented for every 40 MD steps.
13. After exiting the inner loop after 10 MD/streaming steps, the run step was initialized, where active particles gain momentum in the active direction, and the fluid particles behind it gain opposite momentum.
14. Thermostat was implemented for every collision step
15. The loops are repeated until the system reaches equilibrium.

2.7 Code Porting to OpenACC

Code porting is the process of making changes to the original code in order to make it compatible new platform/environment. In this case, we are porting the CPU code to OpenACC, to make the code suitable for running on parallel processors. To parallelize our simulation, first, we need to see what are the intensive mathematical operations that are being performed on arrays and how they can be run concurrently. In this section, we discuss the various operations on large arrays which can be parallelized and provide a basic methodology for porting applications on parallel processors using OpenACC.

2.7.1 Parallelization Strategies for Diverse Computational Scenarios

The following operations were extensively in parallelizing the multiscale simulation.

Increment/Update:

Consider an array that needs to be incremented by a scalar s or a vector v, as shown in code snippets below

do i = 1, n	**do** i= 1, n
a(i) = a(i) + s	a(i)=a(i)+v(i)
end do	**end do**

Table 2.1: Code snippet of a CPU code where array elements are incremented by using a scalar (on the left) and by using a vector (on the right).

In these two cases, we can observe that a(i) is independent of the other elements of a(i.e.; incrementing one array element will not affect the other elements of the array). So each element of the array can be passed to a different thread of the GPU, and this kind of operation can be performed simultaneously. To parallelize this, we use the parallel construct of OpenACC as follows

!$acc parallel loop	***!$acc parallel loop***
do i = 1, n	**do** i= 1, n
a(i) = a(i) + s	a(i)=a(i)+v(i)
end do	**end do**

Table 2.2: Using parallel loop directive to the CPU code from Table 2.1 where array elements are incremented by using a scalar (on the left) and by using a vector (on the right).

Using the parallel loop directive, the compiler is informed that the desired loop is safe to parallelize. However, it is important for the programmer to ensure that the loop can indeed be parallelized safely and effectively. This requires a thorough understanding of the loop's dependencies, memory usage, and potential conflicts.

Note: Above, I have explained the incrementing of a given array using the addition operator. The same explanation is valid for other arithmetic(-, *, /) and logical operations(and, or etc…)

Vector Operations

Instead of incrementing a given array mentioned in the above case. Here the values are written based on the arithmetic operations employed between two vectors. For our convenience, I am using the addition operator to explain the same.

```
do i= 1, n
        a(i) = b(i) + c(i)
end do
```

Table 2.3: Code snippet of a CPU code representing the addition of vectors

In the above code snippet, a is the sum of two vectors, b, and c. In this case, too, we don’t see the interdependence between the array of elements of one variable while updating the values. So this can be written as in Table 2.4.

```
!$acc parallel loop
do i= 1, n
        a(i) = b(i) + c(i)
end do
```

Table 2.4: Implementing parallelization for vector addition

A combination of Vector Operations and Update methods are widely used in parallelization as in most of the computations, array elements get updated without affecting the other elements.

We have extensively employed these computations in various parts of the code. But, these methods can be better observed in the position and velocity updates of the particles. Looking at equations 2.5, 2.6, and 2.17, we can notice that there is no interdependence between the array of elements for these computations.

However, just by implementing a parallel loop, we can't perform operations such as dot product, matrix multiplication, etc. This is because this requires the elements of the array need to be reduced to some value. Implementing only a parallel loop won't provide us with the correct results in these situations. Hence we employ reduction operation as explained below.

Reduction

Now, consider the case where we must perform the sum of elements of the given array. To perform this on a CPU, we write as,

```
sum=0
do i= 1, n
        sum = sum + a(i)
end do
```

Table 2.5: Code snippet of a CPU code to find the sum of the elements for a given array

If we pass the above loop directly to a parallel processor, we won't get the actual sum of the array, as computations that occur in a thread are independent of other threads. But in this case, the variable sum depends on all array elements.

To perform this, we employ the reduction clause of OpenACC. Reduction clause is used to reduce all the private copies of the variable to one final result of the desired parallel region. In this case, it reduces all the copies of the variable sum present in the different threads and adds them up to give the sum of the elements of the array. This can be implemented as follows

```
sum=0
!$acc parallel loop reduction(+: sum)
do i= 1, n
        sum = sum + a(i)
end do
```

Table 2.6: Implementing reduction method in OpenACC to find the sum of the elements of an array

However, the reduction clause can be applied only to scalar variables and for common operations, such as +, *, min, max, and logical operators. The general syntax to execute a reduction operation can be written as

$$\textit{Syntax: reduction(operator: variable)} \tag{2.23}$$

Using reduction operation on a parallelized loop, we can perform dot product, matrix multiplication, summing elements in a row or a column for 2D matrix, and other operations where the computations need to reduce the array elements into one single result. For a better understanding of the reduction operation implementation of dot product on OpenACC is shown in Table 2.7.

We define dot product between two vectors as,

$$dot(a, b) = \sum_{i=1}^{n} a_i * b_i \tag{2.24}$$

Here, the values of $a_i * b_i$ should be reduced(summed up) over all the array elements to give to the dot product as the final result.

```
dot=0
!$acc parallel loop reduction(+: dot)
do i= 1, n
        dot= dot+ a(i)*b(i)
end do
```

Table 2.7: Implementing reduction method to find the dot product of two vectors

Atomic Operations:

The reduction method that was explained earlier does not guarantee the order in which loop iterations will occur. In the case of the summation example provided above, the order in which we add elements of the array does not matter; as long as all elements are considered, the final result will be the same. However, in cases where one loop iteration modifies the value of a variable, and another iteration attempts to read from the same variable in parallel, different outcomes may occur depending on which iteration occurs first.

For more complex situations that cannot be resolved using the reduction operation, the use of atomics is helpful as they ensure that no two threads try to perform the same computation at the same time.

We have used mostly the **update** and **capture** clauses of the atomic directive in our parallelized simulation

i) Atomic Update: This operation ensures that no two threads will read and write simultaneously for the given parallel region. The above explanation of summation can be written using atomic directive as shown below

```
!$acc parallel loop
do i= 1, n
        !$acc atomic update
        sum=sum+a(i)
        !$acc end atomic
end do
```

Table 2.8: Performing sum of the elements of an array using atomic update directive

Note: The sum of array elements obtained using reduction or atomic operation will give you the correct result. However, using the atomic directive makes sure that computation doesn't happen for two different threads at the same time, and this can also be used on vectors which is not the case for reduction

Either the reduction method or atomic update directive was used to compute particle interactions that are mentioned in the previous sections of this chapter. For instance, consider that we have to estimate the total force for a given particle in the MD system. To calculate the total force acting on a given particle, all forces acting upon that particle from all other particles in the system are summed up.

Similarly, using reduction operation or atomic update directive to estimate other particle interactions mentioned in section 1.2 give us the right results.

ii) Atomic Capture: Sometimes, in certain scenarios, such as creating lists of neighboring particles within a simulation box, it's necessary to first update the count of neighbors for a particular particle before storing the positional index of the neighboring particle. To accomplish this type of a task, it's important to store the calculated value in a thread so that it can be utilized in subsequent code following the update.

```
!$acc parallel loop
do i=1,no_of_fluid
      box_no = 1+int(pos_fl(3*i-2))+lx*int(pos_fl(3*i-1))+lxly*int(pos_fl(3*i))
      !$acc atomic capture
      fluid_no(box_no) = fluid_no(box_no) + 1
       j = fluid_no(box_no)
      !$acc end atomic
       box_part(j,box_no) = i
end do
```

Table 2.9: Passing fluid particle to their respective collision boxes using atomic capture operation

The above code is used to arrange the fluid particles based on their positions. Initially, the box number is evaluated based on the position of the fluid particle and stored in the box_no variable. Later count of the fluid particle for a given box is evaluated sequentially in the atomic region. The capture operation ensures the variable j is different for each thread and allows it to be used in the next part of the program.

Similarly, the atomic capture directive is employed in generating the neighbour lists of MD and MPCD particles.

In section 2.7.1, I have mentioned the most frequently used operations, which programmers might have misconceptions about in parallelizing an application. For more details that are concerned with the OpenACC programming, please refer to these citations[20,21,23].

2.7.2 Method to port the code to the GPU with OpenACC

1. Run the existing sequential code on the CPU to establish a baseline for parallelization.
2. Identify the computationally expensive parts of the code by identifying loops containing large-sized arrays where similar operations are performed for one array element at a time. Different kinds of operations that are used to parallelize the simulation are explained in section 2.7.1
3. Pass one such heavy computation region to run on the GPU using OpenACC.
4. Verify the results after parallelizing the desired region and compare the execution time with the baseline.
5. Repeat steps 2 to 4 for such heavy computation regions in the code.
6. After parallelizing all required regions, optimize the code by minimizing the data transfers between the host(CPU) and the device(GPU).

Following these steps will help you to effectively parallelize the code using OpenACC and achieve improved performance.

Chapter 3

Results and Verification of parallelized simulation

In this chapter, we go through the benchmarking simulation methods that were discussed in the previous chapter and also verify the system's behaviour to ensure the compatibility of the simulation for the parallel processors. In complicated simulations, it is important for us to verify every step, as the simulations use approximations to mimic a physical system. It becomes complicated when parallel computing is used because one needs to ensure that the data mapping on the hardware is happening correctly. So, the simulation was verified for every step of the system.

The multiscale simulation was performed for three different sets of parameter values shown in Table 3.1

Case	**Simulation Box Size**	**No. of Fluid particles**	**Size ratio (Passive/Active)**
1	50 x 50 x 50	1.25×10^6	1
2	70 x 70 x 70	3.43×10^6	1
3	68 x 68 x 68	3.14×10^6	2

Table 3.1: Parameter settings used to implement the multiscale simulation for different cases

Initially the multiscale simulation was implemented for Case 1 in Table 3.1. After the successful verification of the simulation on Case 1, the other cases mentioned in Table 3.1 were performed for this simulation. It is shown to be robust for the other cases as well.

3.1 Benchmarking the Hardware

All these simulations were implemented on the PARAM Brahma Supercomputing facility at IISERPune. Hardware specifications of PARAM Brahma of CPU and GPU have mentioned below[33]

OS: Linux – CentOS 7.6

Specifications	**CPU**	**GPU**
Name	Intel Xeon Platinum 8268	Nvidia Tesla V100
Cores	48	5120
Base Clock Speed	2.9 GHz	1245 MHz[34]
Memory	192 GB	16 GB

Table 3.2: Hardware specifications of PARAM Brahma

Note: All the following simulations were run only on one CPU and one GPU. Performance benchmarks of these simulations are shown in their respective sections.

3.2 Verification of Molecular Dynamics Simulation

Molecular Dynamics simulation was verified by plotting

- Energy conservation of the system
- Pair Correlation of the system
- Velocity Distribution of the MD particles

3.2.1 Simulation Units

The simulations performed were not written in the real-world units. Employing simulation units in place of real-world units can improve the efficiency and standardization of the simulation. For this simulation, the following terms are scaled,

$$k_B T = 1,\ m = 1,\ \sigma = 1,\ \tau = 0.05$$

Where,

$k_B T$	= Energy unit of the system
m	= Mass of the particle
σ	= Diameter of the particle
τ	= Simulation time step

3.2.2 System Parameters

The following table has the list of the parameters used to verify the MD simulation

Parameter	**Value**
Number of particles	3×10^3
Size of the simulation box	20x20x20
Number of the iterations	4×10^4
Critical distance(rc)	3.0

Table 3.3: Parameters used for the MD simulation

3.2.3 Energy plots of the MD simulation

(a) Thermostat absent

From figure 3.1, we can observe that the total energy of the system looks like a perfectly straight line, by which one can say that the total energy is conserved when the thermostat is not implemented.

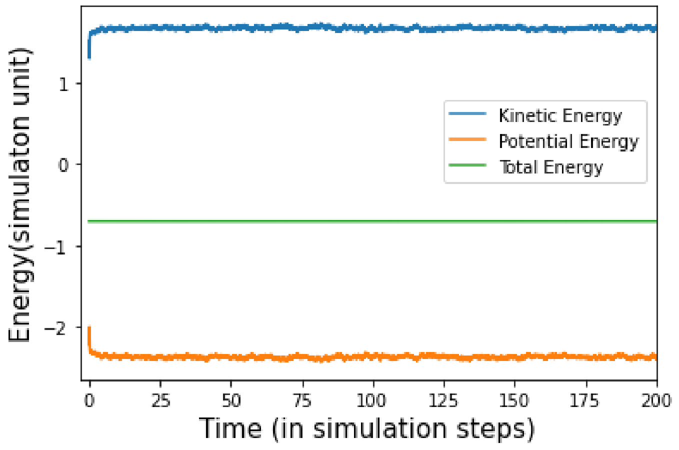


Figure 3.1 Kinetic, Potential, and Total energies per particle over 4 x 10^4 iterations with a simulation time step of 0.005 on GPU when the thermostat was not implemented in the system. Energies are in the units of k_BT, and time is in the units of the simulation time step(τ). Simulation units are shown in subsection 3.2.1

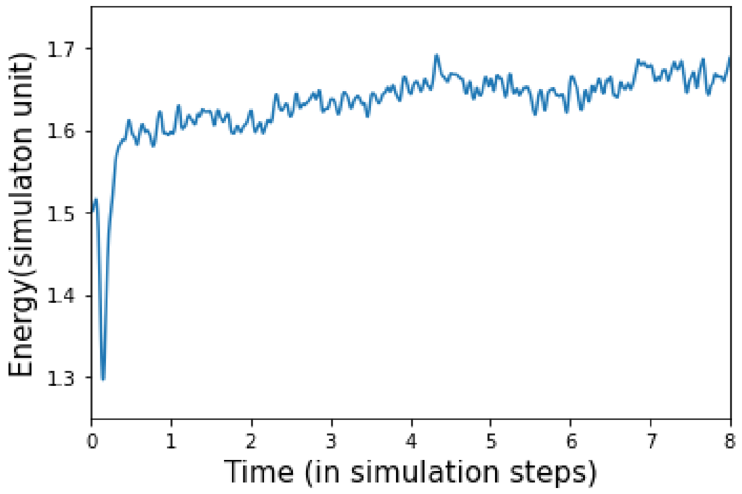


Figure 3.2: Zoomed-in plot of the kinetic energy per particle when no thermostat was used. We can observe the increasing trends of the kinetic energy per particle with iterations, and it has deviated from $1.5k_BT$. Simulation units are shown in subsection 3.2.1

However, if we take a closer look at the kinetic energy, which was shown in Figure 3.2, we can see the rise in the value of kinetic energy per particle from $1.5k_BT$. Based on equation 2.1, an increase in the average velocity of these particles leads to an increase in the average temperature of the system.

(b) Thermostat present

To correct this thermostat was implemented. In this case, the total energy per particle is not a straight line like in the previous case. This happens because the thermostat rescales the velocities of the particles to maintain the temperature T. In Figure 3.4(a), we can see the total energy has a brief equilibration period where it looks like a straight line and a sudden step change in its value occurs when the thermostat is called. However, if we look at the kinetic energy per particle plot(from figure 3.4(b)), though there were large fluctuations in the first, the system was equilibrated after some time, and these values fluctuate over $1.5k_BT$, which implies that our thermostat is working correctly.

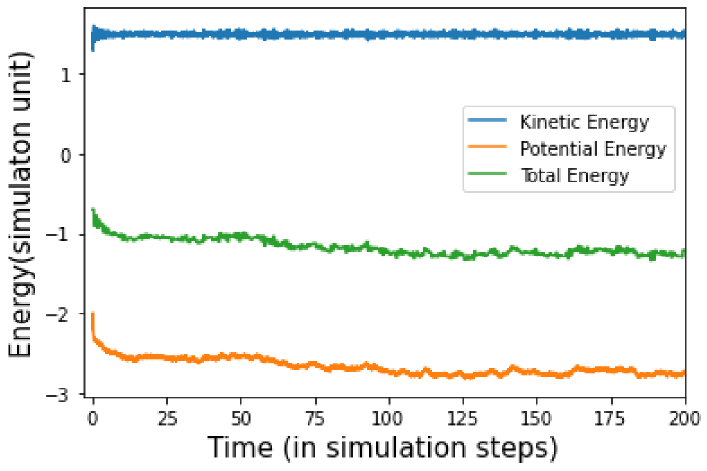
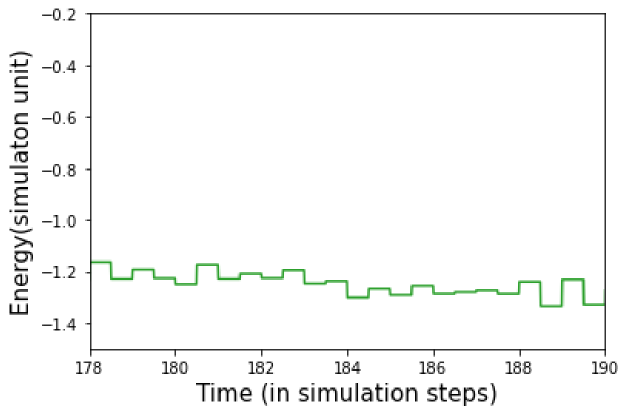
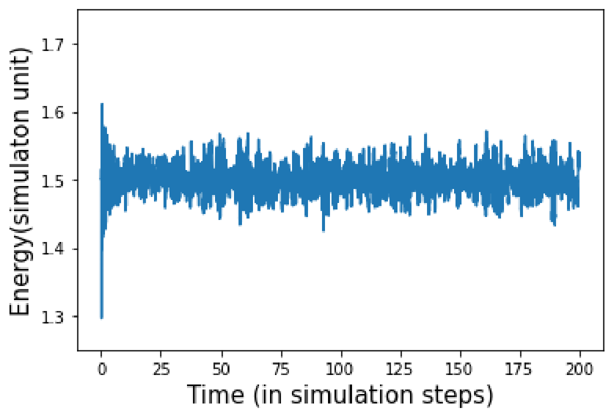


Figure 3.3: Kinetic, Potential, and Total energies per particle over 4 x 10^4 iterations with a simulation time step of 0.005 when the thermostat was implemented in the system. Energies are in the units of k_BT, and time is in the units of the simulation time step(τ). Simulation units are shown in subsection 3.2.1



(a)



(b)

Figure 3.4: (a) Zoomed-in plot of the total energy per particle with time: step changes in the values of total energy occur when the thermostat step is executed in the system. (b) Kinetic Energy per particle is observed to fluctuate around $1.5k_BT$ throughout the time.

3.2.4 Pair correlation Plots

Pair Correlation function, also known as the Radial Distribution function, helps us to understand the variation in the density of particles along the distance from the center of the given particle. This quantity helps us in studying particle clustering in research. It can be defined as the probability of finding a particle at a radial distance r away from the chosen particle.

From figure 3.5, we observe that g(r),

- r<1: In this case, g(r) is zero because we modeled hard spheres using WCA potential. As there will be no particle center present inside a hard spherical particle, g(r) will be zero in this region.
- r=1: Peak is observed at this point because this is the minimum distance where two particles can be nearest to each other without any overlapping.
- r=1.5: A local minimum is observed at this point. This is because if a particle is at r=1.5, then there shouldn't be any particle at r=1, or they would overlap with each other.

- r>>1: In this region, the chosen particle won't affect these particles as they are much farther from it. So, the g(r) value here is normalized to 1 as the density is independent of the particles at longer distances.

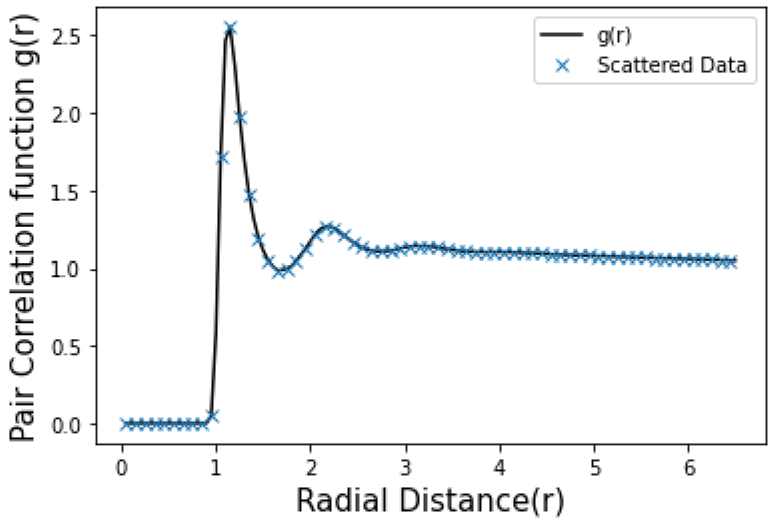


Figure 3.5: Pair Correlation(g(r)) of 3000 MD particles in a 30x30x30 box. The calculation of g(r) was performed every 100 iterations for a total of 4 x 10^4 iterations. r is in the units of σ.

3.2.5 Maxwell-Boltzmann Distribution

As the system is in thermal equilibrium, the velocities of MD particles follow the Maxwell-Boltzmann distribution, which is of the form

$$dN = N\,4\pi\left(\frac{m}{2\pi k_B T}\right)^{3/2} v^2\, e^{-\frac{mv^2}{2k_B T}}\, dv \qquad (3.1)$$

Here,

m = mass of the particle =1

dv = width of the velocity bin = 0.05

N= Total number of particles = 3 x 000

dN= Number of particles in the range v to v+dv

For the velocity distribution that follows the Maxwell Boltzmann distribution the most probable velocity would be,

$$V_{mp} = \sqrt{\frac{2k_B T}{m}} \tag{3.2}$$

Substituting the values of m=1 and $k_B T$ =1 we get,

$$V_{mp} = \sqrt{2}$$

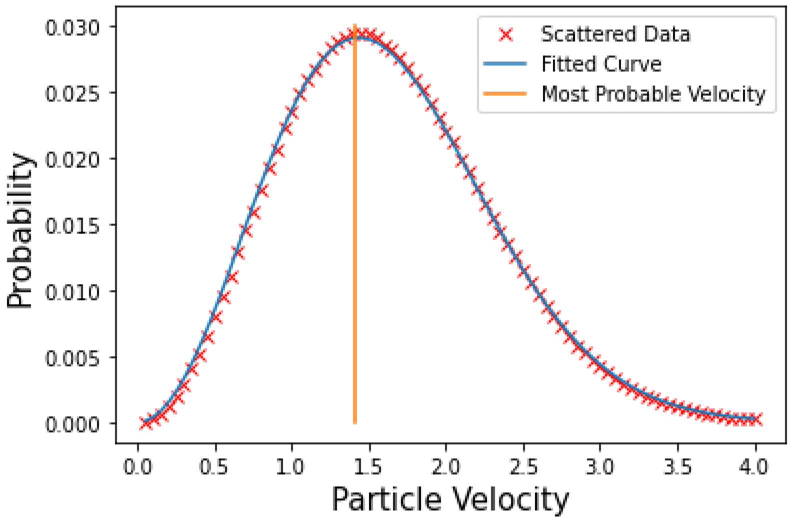


Figure 3.6: The above plot shows the probability distribution of velocities of the MD particles for the parameters shown in Table 3.3. This distribution is plotted from MD simulation, and it was fitted against the Maxwell-Boltzmann Distribution.

From Figure 3.6, it is evident that the straight line, which represents the most probable velocity, V_{mp} cuts the velocity distribution curve at its peak. So, we can conclude that our MD system works on parallel processors perfectly.

3.2.6 Performance benchmarking of MD simulation

Sl. No.	CPU run time (seconds)	GPU run time(seconds)
1	5.641×10^2	3.117×10
2	5.639×10^2	3.157×10
3	5.647×10^2	3.360×10
4	5.636×10^2	$2. 966 \times 10$
5	5.654×10^2	3.078×10
Average run time	**5.643×10^2**	**3.136×10**

Table 3.4: CPU Vs GPU run times of the MD Simulation over 10^4 simulation time steps

As all the tests verify the parallelized MD simulation we need to know how much faster this parallelized application works. For that purpose, using the parameters in Table 3.3, five independent runs over 10^3 iterations were given for the MD simulation on both CPU and GPU, and run time was noted for each run and tabulated in Table 3.4. The average run times of the program on both CPU and GPU performance can be used to estimate the speed up of an application as,

$$speed\ up = \frac{Average\ run\ time\ in\ CPU}{Average\ run\ time\ in\ GPU} \qquad (3.3)$$

Substituting the values we get,

$$speed\ up = \frac{5.643\ X\ 10^2}{3.136\ X\ 10}$$

$$= 17.99 \approx 18$$

The parallelized MD simulation is 18 times faster than the MD simulation executed on a CPU.

3.3 Verification of Multi-Particle Collision Dynamics Simulation

3.3.1 Parameters of the MPCD system

Parameter	Value
Dimensions of the simulation box	50x50x50
Number of the fluid particles	lx*ly*lz*10= 1.25×10^6
Dimensions of the collision box	1x1x1
Number of iterations	5×10^5

Table 3.5: Parameters used for the MPCD simulation

Simulation parameters for the MPCD system are the same as the MD system mentioned in subsection 3.1.1

3.3.2 Velocity distribution of MPCD particles

As explained in the previous section, 3.1.5, from equation 3.2, the Most probable velocity, in this case, will be

$$V_{mp} = \sqrt{2} \quad \text{(as m=1 and } k_B T = 1\text{)}$$

From Figure 3.7, the velocity distribution of the MPCD system is verified as the Most probable velocity curve cuts through the peak of the velocity distribution curve.

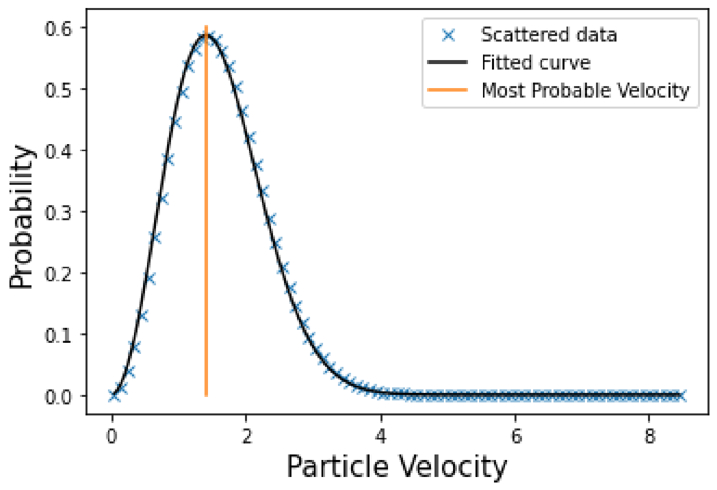


Figure 3.7 The above plot shows the probability distribution of velocities of the fluid particles for the parameters shown in Table 3.5. The data for the plot above is obtained through MPCD simulation, and it was fitted against the Maxwell-Boltzmann Distribution.

3.3.3 Performance benchmarking of MPCD simulation

To determine the speed up of the parallelized MPCD simulation, it was run on CPU five times(five independent runs were given), and the same process was done on GPU for parallelized MPCD simulation. The simulation was run for 5 x 10^4 iterations each time, and the results are shown in Table 3.6.

Sl. No.	**CPU run time (seconds)**	**GPU run time(seconds)**
1	3.607 x 10^4	8.212 x 10^2
2	3.613 x 10^4	8.154 x 10^2
3	3.602 x 10^4	8.155 x 10^2
4	3.636 x 10^4	8.321 x 10^2
5	3.624 x 10^4	8.113 x 10^2
Average run time	**3.617 x 10^4**	**8.191 x 10^2**

Table 3.6: CPU Vs GPU run times of the MPCD Simulation over 5 x 10^4 iterations

Substituting the average run time values in equation 3.3 we get,

$$speed\ up\ = \frac{3.617\ X\ 10^{4}}{8.191\ X\ 10^{2}}$$

$$= 44.15 \approx 44$$

The parallelized MPCD simulation is 44 times faster than the MPCD simulation executed on a CPU. We can see a significant difference in the speed up when compared to the sequential execution of this application. This is because if we look at the parameters from Table 3.5, there are 1.25 x 10^5 fluid particles, and these are updated one by one in the CPU. Using GPU to parallelize this simulation greatly benefited us as it significantly reduced the time involved in computing these large arrays.

3.4 Verification of Fluid-Colloid Interactions

Now, by coupling both MD and MPCD methods, Fluid-Colloid Interactions are simulated, and the verification tests performed are discussed in this section.

3.4.1 Simulation Units and Parameters

Simulation units are used as follows.

- For energy, $k_BT = 1$
- MD time-step = t
- Collision step for MPCD, τ = 10*t

Parameter	Value
Dimensions of the simulation box	50x50x50
Dimensions of the collision box	1x1x1
Average density of fluid particle(ρ)	10
Number of the fluid particles	50x50x50xρ
Diameter of the colloid for colloid-colloid collisions($\sigma_{colloid}$)	5
Mass of the Colloid, $m_{colloid}$	654.1
Cutoff distance for LJ interactions, (r_{cutoff})	$2 \times 2^{1/6} \times \sigma_{colloid}$
Cutoff distance for neighbour lists	$3 \times \sigma_{colloid}$
Iterations	5×10^5

Table 3.7 Parameters for the verification of the fluid flow profile

3.4.2 Velocity flow Profile in the presence of an Active particle

Flow profile created in the presence of moving particles makes it very difficult to analyze the particle interaction with the fluid particles because we need to average over a large number of infinitesimal time-steps. Because of moving particles, random noise will also be a part of the flow profile, so we have to keep the particles stationary in order to study the flow profile. This can be done by allowing the active particle to interact with the other particles and allowing the momentum transfers between them, but the position of the active particle is not updated.

To observe the velocity profile more clearly, we have placed one active particle in the fluid. We obtained the plot shown in Figure 3.8 by simulating the flow profile under these conditions. The observed flow profile matches our expectations based on the discussions of the simulated active forces. The active particle's forward motion is driven by pushing away the fluid particles behind it. Meanwhile, the fluid particles from the top and bottom move in to fill the space vacated by the ones pushed away from the front and back of the active particle. This mechanism produces the observed flow profile, which is depicted here.

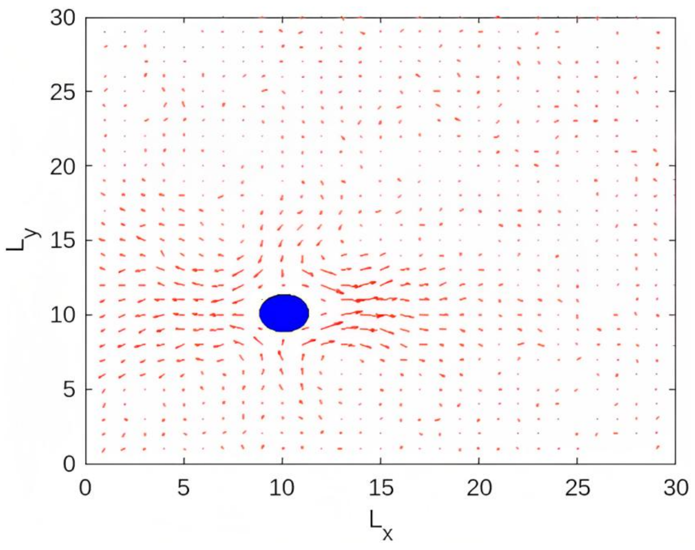


Figure 3.8: Velocity profile of the system in the presence of an active particle(blue). The box dimensions are 50x50x50. However, the plot limits from 0 to 30 in both the x and y directions to observe the flow profile better. The active particle is placed at (10,10,10).

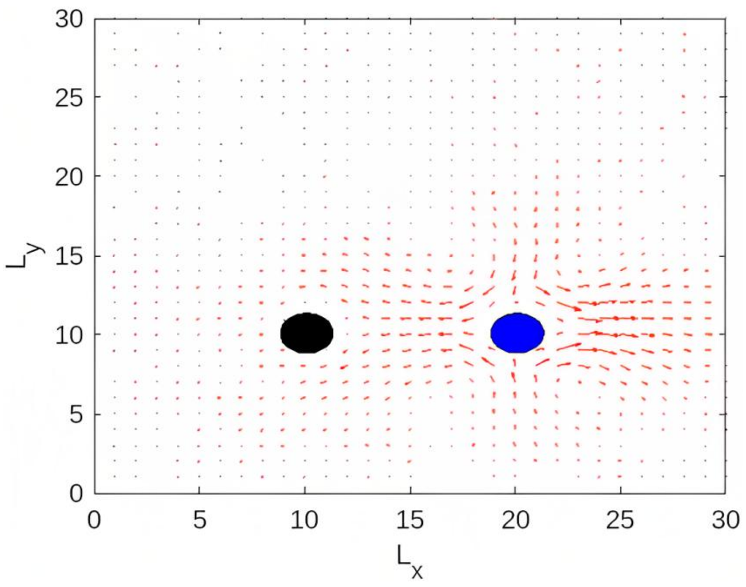


Figure 3.9: Velocity profile of the system in the presence of one active particle(blue) and one passive particle(black). The box dimensions are 50x50x50. However, the plot limits from 0 to 30 in both the x and y directions to observe the flow profile better. The active particle is placed at (20,10,10), and the passive particle is placed at (10,10,10).

Based on these observations, we can say that the modeled fluid-colloid interactions are verified.

3.5 Verifying the Multiscale Simulation

Simulation Parameters and units of this system are the same as the parameters described in subsection 3.3.1. But instead of one colloid, we have placed 200 colloidal particles in this simulation box, where 40 of them are passive particles, and the remaining 160 particles are active colloids.

3.5.1 Correlation plots among colloids

To verify this system, it was run for 1 million iterations, and the pair correlations among different colloidal particles(passive-passive, active-active, and passive-active) were plotted and compared with the results of the previous work.

Note: Sizes and masses of both passive and active particles are the same.

Remember our discussion on the pair correlation function from section 3.1.4, Figure 3.10 gives us an idea of how the density of pairs of active particles changes as a function of the distance. $g_{aa}(r)$ plot(Figure 3.10) is comparable to that of the MD system. As the diameter of the colloid($\sigma_{colloid}$) was chosen to be 5(see Table 3.7), we can see the first peak at 5 and the second peak at 10, and so on. So, we can say active particles can form clustering through hydrodynamic interactions.

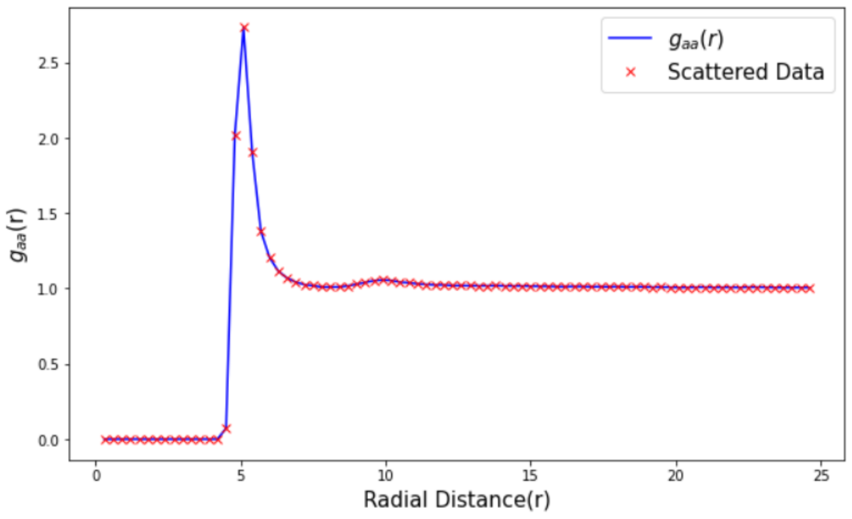


Figure 3.10: Pair Correlation of A-A interactions for 160 active particles in a 50x50x50 simulation box. The size ratio between the passive and active particles is 1. The calculation of g(r) was performed every 5 x 10^4 iterations for a total of 1 Million iterations. Here A-A interactions denote Active-Active interactions.

In Figure 3.11, the pair correlation of passive particles in an active bath is displayed. As previously discussed in 1.1.4, passive particles in an active bath tend to cluster together, which is evident in Figure 3.11. The peaks at integral values of sigma (r=5, 10, 15) demonstrate that clustering also occurs among passive particles in the presence of active particles.

These obtained results verified the parallelized simulation successfully.

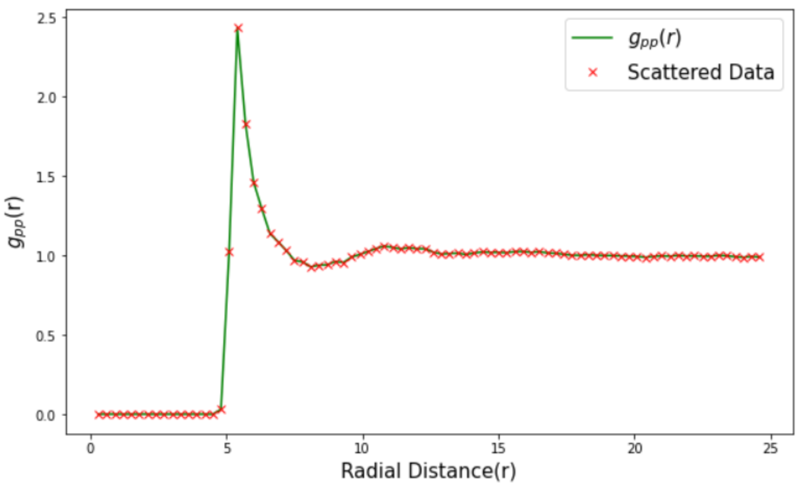


Figure 3.11: Pair Correlation of P-P interactions for 40 passive particles in a 50x50x50 simulation box. The size ratio between the passive and active particles is 1. The calculation of g(r) was performed every 5 x 10^4 iterations for a total of 1 Million iterations. Here P-P interactions denote Passive-Passive interactions.

3.5.2 Performance benchmarking of the Multiscale-simulation

Sl. No.	CPU run time (seconds)	GPU run time(seconds)
1	1.972 x 10^5	1.647 x 10^4
2	1.979 x 10^5	1.625 x 10^4
3	1.973 x 10^5	1.659 x 10^4
4	1.979 x 10^5	1.615 x 10^4
5	1.979 x 10^5	1.641 x 10^4
Average run time	**1.976 x 10^5**	**1.638 x 10^4**

Table 3.8: CPU Vs GPU run times of the Multiscale Simulation over 2 x 10^5 iterations

Performance speed up of the multiscale simulation can be obtained by substituting average run times from Table 3.8 in equation 3.3,

$$speed\ up \;=\; \frac{1.976\ X\ 10^{5}}{1.638\ \ X\ 10^{4}}$$

$$=\ 12.06 \approx 12$$

Parallelized Multiscale simulation is 12 times faster than that of CPU code for the given parameters discussed in section 3.4. However, if we compare it with the accelerated performance of MD and MPCD simulations, MPCD speed up is not as fast as the others. So, one might be wondering why this is happening. It is because after coupling both MD and MPCD simulations, various kinds of interactions take place among all kinds of particles(active, passive, and fluid), which were briefly mentioned in section 1.2. As these interactions are not independent of one another and more complex to compute, some parts of the code were executed serially on the GPU. As the GPU has a lower clock speed than the CPU, these serial operations would consume more time. Still achieving a 12 times speed up is significantly great because the run time for this system was reduced from 2.5 days(approx) to 4.5 hours. This allows us to carry out further research to investigate the hydrodynamics of active matter systems in the fluid medium under different conditions and parameters.

3.6 Correlation plots for different parameter settings

As the accelerated model for the multiscale simulation has been verified by performing various tests described in previous sections of this chapter, we now implemented this model for the different simulation parameters. By plotting the correlation between particles under various parameter settings, one can confirm the stability of the parallelized simulation by examining whether the model remains stable even when the parameters are modified.

3.6.1 Pair Correlation plots for 70 x 70 x70 box

(a) Simulation Units and Parameters

Simulation units are used as follows.

- For energy, $k_BT = 1$
- MD time-step = t
- Collision step for MPCD, τ = 10*t

Parameter	Value
Dimensions of the simulation box	70x70x70
Dimensions of the collision box	1x1x1
Number of the fluid particles (ρ) per collision box	10
Number of the fluid particles	3.43×10^4
Diameter of the colloid for colloid-colloid collisions($\sigma_{colloid}$)	5
Mass of the Colloid, $m_{colloid}$	654.1
Number of Active Particles	420
Number of Passive Particles	105
Size ratio = Passive particle size/ Active particle size	1
Cutoff distance for LJ interactions, (r_{cutoff})	$2^{1/6} \times \sigma_{colloid}$
Cutoff distance for neighbour lists	$3 \times \sigma_{colloid}$
Iterations	1 million (10^6)

Table 3.9: Parameters for the multiscale simulation for 70x70x70 box and for size ratio 1

(b) Pair Correlation plots for 70 x 70 x70

$g_{aa}(r)$ plot(Figure 3.12) shows how the density of the active particles varies with the radial distance. As the diameter of the colloid($\sigma_{colloid}$) was chosen to be 5(see Table 3.9), we can see the first peak at 5 and some smaller subsequent peaks. Based on this, we can say that the active particles can form clustering through hydrodynamic interactions for the parameters mentioned in Table 3.9.

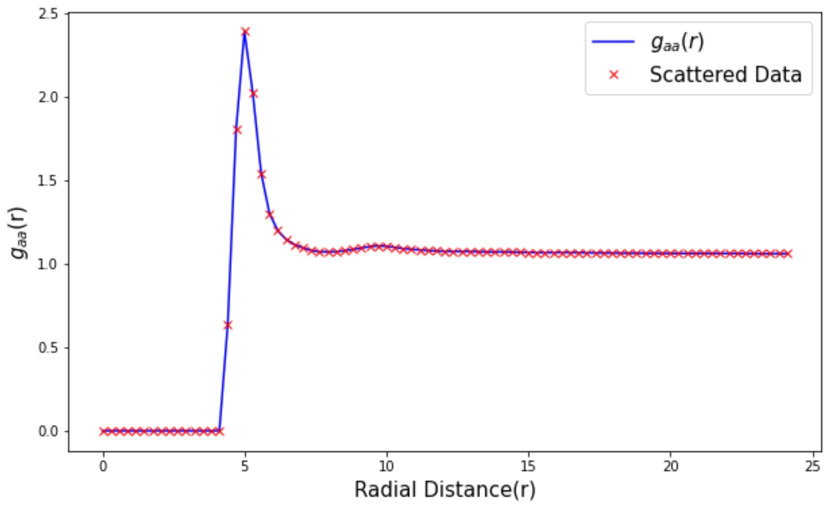


Figure 3.12: Pair Correlation of A-A interactions among 420 active particles in the 70x70x70 simulation box. The size ratio between the passive and active particles is 1. The calculation of g(r) was performed every 5 x 10^4 iterations for a total of 1 Million iterations. Here A-A interactions denote Active-Active interactions.

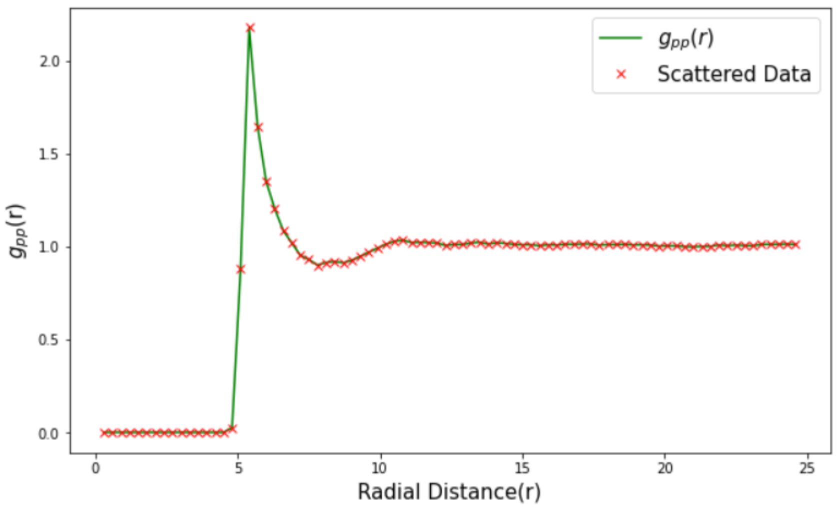


Figure 3.13: Pair Correlation of P-P interactions among 105 passive particles. The size ratio between the passive and active particles is 1. The calculation of g(r) was performed every 5 x 10^4 iterations for a total of 1 Million iterations. Here P-P interactions denote Passive-Passive interactions.

Similarly, from Figure 3.13, we can see that the clustering also happens in the passive particles even if the size of the simulation box is changed.

(c) Performance Benchmarking for the new parameters

With the accelerated model, it took 1.384 x 10^5 seconds(1.5 days approx.)to complete 1 million iterations. To compare it with the CPU time, the non-accelerated version of this simulation was run on the CPU for 100 iterations.

On the CPU for the given parameters from Table 3.9, it took 349.5 seconds for 100 iterations. So, for 1 million iterations, it takes approximately 3.495 x 10^6 seconds(40.5 days). By comparing the rescaled times of both CPU and GPU, the speed up was found to be 26.9(27 approximately) times faster in GPU.

3.6.2 Pair Correlation plots for size ratio =2

(a) Simulation Units and Parameters

Parameter	**Value**
Dimensions of the simulation box	68x68x68
Dimensions of the collision box	1x1x1
Average density of fluid particle(ρ) per collision box	10
Number of the fluid particles	3.14×10^6 (approx.)
Diameter of the passive colloid($\sigma_{passive}$)	10
Diameter of the active colloid (σ_{active})	5
Mass of the Active Colloid, m_{active}	654.1
Mass of the Passive Colloid, $m_{passive}$	5236
Number of Active Particles	160
Number of Passive Particles	40
Cutoff distance for LJ interactions between active-active	$2^{1/6} \times \sigma_{active}$
Cutoff distance for LJ interaction between passive-passive	$2^{1/6} \times \sigma_{passive}$
Cutoff distance for LJ interaction between active-passive	$2^{1/6} (0.5\sigma_{active} + 0.5\sigma_{passive})$
Cutoff distance for neighbour lists	$3 \times \sigma_{passive}$
Iterations	1 million

Table 3.10: Simulation Parameters for size ratio= 2 between the passive and active colloids in a 68x68x68 simulation box.

Simulation units for the system are used as follows.

- For energy, $k_BT = 1$
- MD time-step = t
- Collision step for MPCD, τ = 10*t

(b) Pair Correlation plots for size ratio =2

In all previous simulations, the size ratio between the active and passive colloids was maintained as 1. In this case, the size of the passive particles is set to be twice that of the active particles. However, the density is the same for both active and passive particles and was set to 10. Then the mass of these particles was evaluated by,

$$Mass = Density \ x \ Volume \tag{3.4}$$

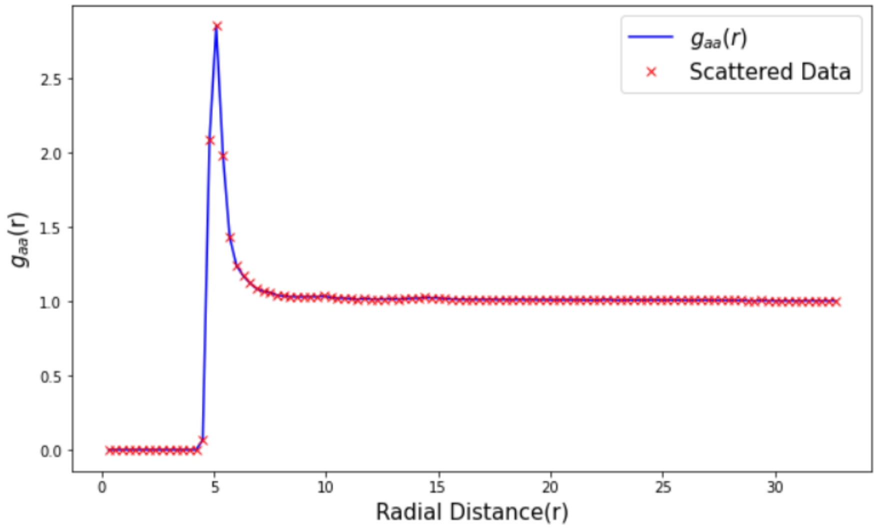


Figure 3.14: Pair Correlation of A-A interactions for size ratio = 2 between the passive and active particles. This simulation was performed in the 68x68x68 simulation box. The calculation of g(r) was performed every 10^5 iterations for a total of 1 Million iterations. Here A-A interactions denote Active-Active interactions.

By examining the Pair Correlation plots in Figures 3.14 and 3.15, it is apparent that:

- For $\sigma_{active} = 5$, the first peak in the pair correlation plot is observed at r = 5 in Figure 3.14, indicating the potential for clustering among active particles under the system parameters outlined in Table 3.10.

- Likewise, for $\sigma_{passive} = 10$, the first peak in the pair correlation plot is seen at r = 10 in Figure 3.14, suggesting that clustering may occur among passive particles in the presence of an active bath.

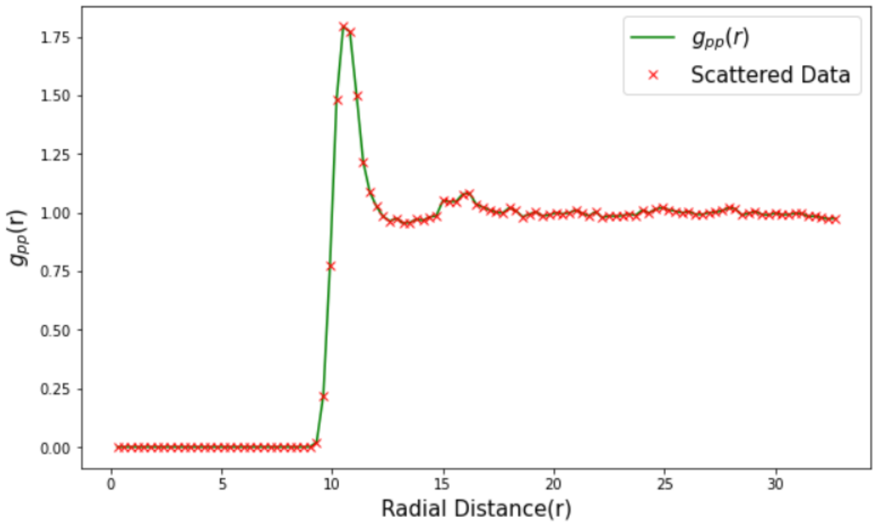


Figure 3.15: Pair Correlation of P-P interactions for size ratio=2 between the passive and active particles. This simulation was performed for the 68x68x68 simulation box. The calculation of g(r) was performed every 10^5 iterations for a total of 1 Million iterations. Here P-P interactions denote Passive-Passive interactions.

(c) Performance Benchmarking for size ratio = 2

With the accelerated model, it took 46 hours(nearly 2 days) to complete 1 million iterations. From Adrian's work, it was estimated that to run 1.2 x 10^5 iterations, it takes approximately 4 days(96 hours) to complete on a CPU, so 1 million iterations will be completed after approximately 33.33 days. If we scale the performance based on this information, we can say that the accelerated model is 17 times faster than the CPU code.

3.7 Summary of Performance Benchmarking for Parallelized Applications

In sections 3.5.2, 3.6.1(c), and 3.6.2(c), we have estimated the speed-up factors of the parallelized model compared with the sequential execution of the multiscale simulation under different cases. If one wants to compare the performance of parallelized simulation over parameter settings, we can determine the time taken to perform a particular number of computations. In this case, we estimated the time taken to perform 10^2 iterations, and the results are tabulated in Table 3.11.

Case	**Simulation Box Size (lx *ly *lz)**	**No. of Fluid particles**	**Size ratio**	**Speed-Up Factor (CPU Vs GPU)**	**Time per 100 iterations on GPU (seconds)**
1	50 x 50 x 50	1.25×10^6	1	12	8
2	70 x 70 x 70	3.43×10^6	1	27	13
3	68 x 68 x 68	3.14×10^6	2	17	17

Table 3.11: Acceleration achieved for multiscale simulation under different parameter settings

But for this accelerated model this can’t be done because I have not optimized the kernel performance in this simulation. So, the OpenACC compiler assigns gangs and workers to the loop automatically based on the complexity of the loop. If there are unused vectors, the OpenACC compiler assigns them zero values and performs computations on them as if they were active. This can be avoided by specifying more information, such as the number of gangs, workers, and vectors required for a loop[21].

I have not optimized the loop performance because mapping these numbers varies with the accelerator used. The current accelerated model has more flexibility allowing the users to execute the simulation on the desired hardware.

Note: For a particular set of hardware, after performing data optimization, kernel optimization, and other required operations, parallel programmers try to run the accelerated application for different parameters to find out the optimum parameter size that yields the best speed-up factor. This is done by observing the trends in the speed-up factor with respect to the parameter size. In most cases, we may find that the speed-up initially increases as the parameter size increases, but after a while, it levels off and decreases after a point due to hardware limitations. So, after changing the parameters, compare the speed-up factor from previous data and modify the parallelization techniques by optimizing the kernel's performance, implementing different parallel algorithms, or using a different set of hardware.

Chapter 4

Conclusion and Future Work

We have parallelized the Multiscale simulation successfully, coupling the Molecular Dynamics(MD) system and the Multiparticle Collision Dynamics(MPCD) simulation. By placing both active and passive particles in the fluid medium, we have observed clustering for both active and passive particles in the presence of hydrodynamics. This parallelized model has been verified successfully against the simulation done previously. Based on the successful verification of the parallelized multiscale simulation, the simulation was run under various parameter settings. The results obtained from these simulations demonstrate the effectiveness and reliability of the parallelized multiscale simulation in producing accurate and efficient results

In conclusion, our simulation has demonstrated its usefulness as a tool for further research. Its ability to parallelize the complex system involving hydrodynamics allows the research to be at an accelerated pace. In the future, this model will be used by the soft matter research groups of IISER Pune, IISER Bhopal, and IIT Bombay to investigate the role of hydrodynamic interactions in emergent phenomena.

Bibliography

[1] Marchetti M C, Joanny J F, Ramaswamy S, Liverpool T B, Prost J, Rao M and Simha R A 2013 Hydrodynamics of soft active matter *Rev. Mod. Phys.* **85** 1143–89

[2] Vicsek T and Zafeiris A 2012 Collective motion *Phys. Rep.* **517** 71–140

[3] Ramaswamy S 2010 The Mechanics and Statistics of Active Matter *Annu. Rev. Condens. Matter Phys.* **1** 323–45

[4] Schwarz-Linek J, Valeriani C, Cacciuto A, Cates M E, Marenduzzo D, Morozov A N and Poon W C K 2012 Phase separation and rotor self-assembly in active particle suspensions *Proc. Natl. Acad. Sci.* **109** 4052–7

[5] Das M, Schmidt C F and Murrell M 2020 Introduction to Active Matter *Soft Matter* **16** 7185–90

[6] Kumar M S and Philominathan P 2009 The physics of flagellar motion of E. coli during chemotaxis *Biophys. Rev.* **2** 13–20

[7] Purcell E M 1977 Life at low Reynolds number *Am. J. Phys.* **45** 3–11

[8] Aditi Simha R and Ramaswamy S 2002 Hydrodynamic Fluctuations and Instabilities in Ordered Suspensions of Self-Propelled Particles *Phys. Rev. Lett.* **89** 058101

[9] Howse J R, Jones R A L, Ryan A J, Gough T, Vafabakhsh R and Golestanian R 2007 Self-motile colloidal particles: from directed propulsion to random walk *Phys. Rev. Lett.* **99** 048102

[10] Jiang H-R, Yoshinaga N and Sano M 2010 Active Motion of a Janus Particle by Self-Thermophoresis in a Defocused Laser Beam *Phys. Rev. Lett.* **105** 268302

[11] Thutupalli S, Seemann R and Herminghaus S 2011 Swarming behavior of simple model squirmers *New J. Phys.* **13** 073021

[12] Zöttl A and Stark H 2014 Hydrodynamics Determines Collective Motion and Phase Behavior of Active Colloids in Quasi-Two-Dimensional Confinement *Phys. Rev. Lett.* **112** 118101

[13] Valeriani C, Li M, Novosel J, Arlt J and Marenduzzo D 2011 Colloids in a bacterial bath: simulations and experiments *Soft Matter* **7** 5228–38

[14] Pushkin D O and Yeomans J M 2014 Stirring by swimmers in confined microenvironments

J. Stat. Mech. Theory Exp. **2014** P04030

[15] Gonnella G, Marenduzzo D, Suma A and Tiribocchi A 2015 Motility-induced phase separation and coarsening in active matter *Comptes Rendus Phys.* **16** 316–31

[16] Kirk D and Hwu W W 2013 *Programming massively parallel processors: a hands-on approach* (Amsterdam: Elsevier, Morgan Kaufmann)

[17] Fig. 1. Difference Between GPU and CPU Architecture GPU architecture... *ResearchGate*

[18] Suryowinoto A 2016 *Cuda by Example An Introduction To General Purpose GPU Programming*

[19] Warp — alpaka 0.5.0 documentation

[20] Ruetsch G and Fatica M 2014 *CUDA Fortran for scientists and engineers: best practices for efficient CUDA Fortran programming* (Amsterdam : Boston: Morgan Kaufmann, an imprint of Elsevier)

[21] Storti D and Yurtoglu M 2016 *CUDA for engineers: an introduction to high-performance parallel computing* (New York: Addison-Wesley)

[22] 2018 Programming Constructs for Beginners *DEV Community*

[23] Larkin J 2017 Chapter 19 - Parallel programming with OpenACC *Programming Massively Parallel Processors (Third Edition)* ed D B Kirk and W W Hwu (Morgan Kaufmann) pp 413–41

[24] OpenACC Programming and Best Practices Guide

[25] Anon Fig. 1. The Lennard-Jones (LJ, thick curve) and the WCA (dashed curve)... *ResearchGate*

[26] Kapral R 2008 Multiparticle collision dynamics: Simulation of complex systems on mesoscales *Adv. Chem. Phys.* **140** 89

[27] Ihle T 2009 Chapman–Enskog expansion for multi-particle collision models *Phys. Chem. Chem. Phys.* **11** 9667–76

[28] Malevanets A and Kapral R 1999 Mesoscopic model for solvent dynamics *J. Chem. Phys.* **110** 8605–13

[29] Ihle T and Kroll D M 2001 Stochastic rotation dynamics: A Galilean-invariant mesoscopic model for fluid flow *Phys. Rev. E* **63** 020201

[30] Padding J T, Wysocki A, Löwen H and Louis A A 2005 Stick boundary conditions and rotational velocity auto-correlation functions for colloidal particles in a coarse-grained

representation of the solvent *J. Phys. Condens. Matter* **17** S3393

[31] Berg H C 2004 Cell Populations *E. coli in Motion* Biological and Medical Physics, Biomedical Engineering (New York, NY: Springer) pp 19–30

[32] Kos Ž and Ravnik M 2018 Elementary Flow Field Profiles of Micro-Swimmers in Weakly Anisotropic Nematic Fluids: Stokeslet, Stresslet, Rotlet and Source Flows *Fluids* **3** 15

[33] Anon PARAM Brahma User's Manual

[34] Anon NVIDIA Tesla V100 16GB GPU

[35] Chen Xuanyi, "Starting from Birds and Bacteria: Disorder, Order, Fluctuation, Stability and Instability", Physics Bimonthly Taiwan, (2020-06-02)

[36] " Non-equilibrium self-assembly of colloidal particles in active liquids", Nishant Barua(MS thesis), IISER-Pune