Multiscale and Mesoscopic Modeling of Soft Matter and Biophysical Systems Using High Performance Computing and Machine Learning

by

Yu-Hang Tang

B.E., Zhejiang University; Hangzhou, Zhejiang, China, 2011

M.Sc., Brown University; Providence, Rhode Island, USA, 2015

A dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Division of Applied Mathematics at Brown University

PROVIDENCE, RHODE ISLAND

May 2018

This dissertation by Yu-Hang Tang is accepted in its present form
by the Division of Applied Mathematics as satisfying the
dissertation requirement for the degree of Doctor of Philosophy.

Date_____ _____

George Em Karniadakis, Ph.D., Advisor

Recommended to the Graduate Council

Date_____ _____

Martin Maxey, Ph.D., Reader

Date_____ _____

Nathan Baker, Ph.D., Reader

Approved by the Graduate Council

Date_____ _____

Andrew G. Campbell, Dean of the Graduate School

Vitae

Yu-Hang was born to Qun-Hui Tang and Fang Lei in June 1989 in the historical city of Xi'an, China. He graduated from Hangzhou No.2 High School in 2007 and began his undergraduate study at Zhejiang University in Hangzhou. In June 2011, he graduated as an outstanding graduate with a Bachelor of Engineering degree in Polymer Science. Afterwards, he came to the U.S. and briefly attended the University of Illinois at Urbana-Champaign before starting his study in the Division of Applied Mathematics at Brown University in Providence.

Yu-Hang obtained his very first exposure to scientific research at the age of ten by manually pasting microscopic photos of renal tumor cells into his father's Ph.D. dissertation. He learned to program the BASIC language on a Wenquxing PC-260 digital dictionary in middle school, and has since enjoyed programming throughout the subsequent years. Nonetheless, he chose polymer science as his college major, and spent one year in laboratory creating conductive polymer composites. He switched to molecular dynamics as his Bachelor's thesis topic under the advisory of Prof. Jun Ling and reaffirmed his passion in scientific computing.

At Brown University, he worked under the mentorship of Prof. George Karniadakis. He has broad interest spanning various aspects of scientific computing, and also devoted a significant amount of time in developing open-source scientific software.

Publications

1. **Y.-H. Tang**, D. Zhang, and G. E. Karniadakis. An atomistic fingerprint algorithm for learning *Ab Initio* molecular force fields. *manuscript in preparation*, 2017.

2. **Y.-H. Tang**, L. Lu, L. Grinberg, H. Li, V. Sachdeva, C. Evangelinos, and G. E. Karniadakis. OpenRBC: A fast simulator of red blood cells at protein resolution. *Biophysical Journal*, 112(10):2030–2037, 05 2017.

3. A. L. Blumers, **Y.-H. Tang**, Z. Li, X. Li, and G. E. Karniadakis. GPU-accelerated red blood cells simulations with transport dissipative particle dynamics. *Computer Physics Communications*, 217:171 – 179, 2017.

4. Z. Li, X. Bian, **Y.-H. Tang**, and G. E. Karniadakis. A dissipative particle dynamics method for arbitrarily complex geometries. *arXiv preprint arXiv:1612.08761*, 2016.

5. **Y.-H. Tang**, Z. Li, X. Li, M. Deng, and G. E. Karniadakis. Non-equilibrium dynamics of vesicles and micelles by self-assembly of block copolymers with double thermoresponsivity. *Macromolecules*, 49(7):2895–2903, 2016.

6. X. Bian, M. Deng, **Y.-H. Tang**, and G. E. Karniadakis. Analysis of hydrodynamic fluctuations in heterogeneous adjacent multidomains in shear flow. *Physical Review E*, 93(3):033312, 2016.

7. F. He, Y. Li, **Y.-H. Tang**, J. Ma, and H. Zhu. Identifying micro-inversions using high-throughput sequencing reads. *BMC genomics*, 17(1):141, 2016.

8. X. Li, E. Du, H. Lei, **Y.-H. Tang**, M. Dao, S. Suresh, and G. E. Karniadakis. Patient-specific blood rheology in sickle-cell anaemia. *Interface focus*, 6(1):20150065, 2016.

9. D. Rossinelli, **Y.-H. Tang**, K. Lykov, D. Alexeev, M. Bernaschi, P. Hadjidoukas, M. Bisson, W. Joubert, C. Conti, G. Karniadakis, et al. The in-silico lab-on-a-chip: petascale and high-throughput simulations of microfluidics at cell

resolution. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015.

10. **Y.-H. Tang**, S. Kudo, X. Bian, Z. Li, and G. E. Karniadakis. Multiscale Universal Interface: A concurrent framework for coupling heterogeneous solvers. *Journal of Computational Physics*, 297:13–31, 2015.

11. Z. Li, **Y.-H. Tang**, X. Li, and G. E. Karniadakis. Mesoscale modeling of phase transition dynamics of thermoresponsive polymers. *Chemical Communications*, 51(55):11038–11040, 2015.

12. **Y.-H. Tang** and G. E. Karniadakis. Accelerating dissipative particle dynamics simulations on GPUs: Algorithms, numerics and applications. *Computer Physics Communications*, 185(11):2809–2822, 2014.

13. X. Li, **Y.-H. Tang**, H. Liang, and G. E. Karniadakis. Large-scale dissipative particle dynamics simulations of self-assembled amphiphilic systems. *Chemical Communications*, 50(61):8306–8308, 2014.

14. Z. Li, **Y.-H. Tang**, H. Lei, B. Caswell, and G. E. Karniadakis. Energy-conserving dissipative particle dynamics with temperature-dependent properties. *Journal of Computational Physics*, 265:113–127, 2014.

Acknowledgments

First and foremost, I would like to thank my advisor, Professor George Em Karniadakis, for his support and guidance throughout the entire course of my Ph.D. study. He is a great mentor with vision and knowledge to guide me when I am in doubt, yet his role as an advisor extends far beyond teaching concepts and research skills. Through our day-to-day interactions, I am deeply motivated by his curiosity and passion to explore the unknown, as well as his courage to break the old and to establish the new. I am proud to join the cohort of his Ph.D. recipients, and I could never ask for a better teacher and mentor.

I would also like to extend my thanks to my dissertation committee for reading through my thesis. I have learned a lot about fluid mechanics from the course taught by Professor Martin Maxey, and deeply admires his approachable and elegant teaching style. I have worked collaboratively for a long time with Dr. Nathan Baker as well as his group members, and I am impressed by his wisdom which spans diverse disciplines.

Special thanks to many colleagues who have worked closely with me. Dr. Zhen Li and Dr. Xuejin Li have collaborated closely with me on many projects related to soft matter and blood flow. I have learned a great deal on programming from Shuhei Kudo while we worked together on the Multiscale Universal Interface. Ansel Blumers has devoted significant effort with me on GPU computing

for particle simulations, and has volunteered as an alpha tester for several software projects that I developed. The teamwork between Lu Lu, Dongkun Zhang, and me can be traced back to a parallel ray tracer, which initially was designed as a course project but later demonstrated huge practical value. Our later collaborations involved parallel particle solvers and machine learning algorithms. He Li contributed his whole-cell model for the development of OpenRBC. I am very grateful to Dr. Leopold Grinberg for the opportunity of working with many brilliant IBM researchers that he has brought about. Diego Rossinelli was a great team leader that eventually led us to the finals of the Gordon Bell competition. The discussion and support from Mingge Deng and Xin Bian helped me to solve many difficult problems. Huan Lei is the person who initiated me into the subject of Dissipative Particle Dynamics.

My thanks also go to all our group's current and former colleagues, Dmitry Aleexev, Hung-Yu Chang, Xuejuan Chen, Yixiang Deng, Anna Lischke, Seungjoon Lee, Guang Lin, Kirill Lykov, Changho Kim, Zhiping Mao, Xuhui Meng, Wenxiao Pan, Paris Perdikaris, Fangying Song, Nathaniel Task, Xiu Yang, Alireza Yazdani, Yue Yu, Fanhai Zeng, Lifei Zhao, Xuan Zhao, Mengdi Zheng, Xiaoning Zheng, for the wonderful time that we have enjoyed together.

Thank you to all my family for the encouragement and support throughout the years. I want to thank my parents for encouraging me and giving me the freedom to chase my dream. I know it is difficult for us to get separated at the opposite sides of the earth. My final acknowledgment is to my lifetime teammate, Yinjia Zhang, my wife. Your taught me the true meaning of responsibility and trust. Your company is what supported me through many difficult times. I truly value every day that we have and will have spent together!

Abstract of "Multiscale and Mesoscopic Modeling of Soft Matter and Biophysical Systems Using High Performance Computing and Machine Learning" by Yu-Hang Tang, Ph.D., Brown University, May 2018

This dissertation is composed around the subject of multiscale modeling of soft matter and biophysical systems with applications using large-scale computations. Specifically, it is expanded on three fronts:

1) Development of high-performance simulators and computational frameworks. On this front, I will discuss the design of three sets of software. The first one is an accelerated parallel particle simulator, which features many algorithmic innovations for harnessing the massively parallel threading architecture of general purpose graphics processing units. The second one is an ultrafast coarse-grained molecular dynamics simulator, which enables the simulation of an entire human red blood cell at protein resolution using a single computer workstation. This is realized by a novel algorithm that allows neighbor search in a sparse 3D space in linear time.The third one is a generic framework that utilizes the concept of meshless interpolation to faciliate the implementation of parallel concurrently coupled multiscale simulations.

2) Construction of mesoscopic models for amphiphilic and thermo-responsive polymers and their applications to large-scale mesoscopic simulations. This is manifested in a detailed study of the non-equilibrium dynamics of thermo-responsive polymers. One of the most interesting findings is that a thermo-responsive polymer membrane may invert its layered structure without actually rotating any of its composing molecules.

3) Data-driven algorithms for learning complex interatomic force fields. Here I have focus on a specific aspect of this field, *i.e.* the design of feature vectors that can efficiently and accurately quantify the similarity between atomistic configurations. To achieve this, a kernel minisum approach is proposed as a robust

and efficient replacement of the principal component analysis algorithm. A set of quadrature rules and parameters are also proposed for constructing a smoothed density field that allows either inner product- or norm-based comparison of structural similarity.

Contents

List of Tables

List of Figures

Chapter One

Introduction

1.1 Computational Science and Particle-based Methods

Computational science is a multidisciplinary research field that has seen rapid development since the advent of digital electronic computers. By exploiting the tremendous power of computers in carrying out arithmetic and logical operations, computational scientists strive to solve scientific problems that cannot be solved analytically or that involve computations traditionally deemed intractable. In fact, computational science is nowadays perceived as an integral and irreplaceable part of scientific methodology, complementing theory and experimentation. Today, the field has grown well beyond the very first topics, such as ballistic computation, Monte Carlo sampling, ordinary differential equations, and n-body simulations [42], as studied by first-generation computational scientists. It has expanded to encompass diverse areas of research including but not limited to aerodynamics, hydrodynamics, mechanical engineering, molecular biology, material science, process engineering, biomedical imaging, economical modeling and financial engineering.

Molecular Dynamics (MD) is among the very first set of algorithms implemented on digital electronic computers[42, 3, 121]. In this method, a collection of infinitesimal particles are used to model an atomistic system, whose evolution are governed by the Newton's law of motion. Each particle that represents an atom carries a certain number of degrees of freedom, *e.g.* position, velocity, mass, charge *etc*. The force that drives the movement of the atoms is calculated as the negative gradient of a Hamiltonian, which in turn is defined as a sum of pairwise and manybody potentials. One of the obvious advantages of MD is that it allows intuitive interpretation of the simulated process. Both the model specification

and simulation result can be directly connected to and easily interpreted within well-established models in traditional science areas such as chemistry and biology. Moreover, MD simulations allows observations of molecular systems at a spatial and temporal resolution exceeding any current experimental apparatus.

However, it is extremely difficult to simulate systems with atomistic molecular dynamics at, for example, beyond the micrometer length scale and millisecond time scale. Traditionally, the performance of MD simulation software was primarily constrained by the speed of pairwise force evaluations, where the interaction between each particle and its hundreds of neighbors had to be examined. This motivates the development of coarse-grained molecular dynamics (CGMD) models, where each particle represents a small group of tightly connected atoms. The benefit of coarse-graining is two-fold. First, it reduces the total number of particles that need to be computed. Second, it smooths out the potential energy landscape and thus reduces numerical stiffness of the system. However, the conventional CGMD approach of using an 'equivalent' conservative potential to replace the multi-atom interaction potential often resulted in a loss, at varying degrees, of the accuracy of the CGMD simulations, especially at reproducing the dynamic properties of the original system. For example, coarse-grained systems using only equivalent conservative potentials always tend to accelerate the diffusion process.

Meanwhile, thanks to the development of massively parallel processors and supercomputers, the performance hotspot of pairwise interaction calculation has been largely eliminated by the abundant arithmetic capabilities of modern multi-core and many-core processors. Therefore, simulations containing billions and even trillions of particles become feasible, at least at a time scale given by a few hundred thousand time steps. However, an obstacle still remains and prevents the realization of longer-time-scale simulations. It is the numerical stiffness of

particle systems, which severely constrains the largest usable time step for stable simulations.

The Dissipative Particle Dynamics (DPD) method has the potential to overcome this 'wall of stiffness'. In order to further extend the accessible time scale of particle-based simulations, DPD uses very soft conservative potentials to allow for aggressive coarse-graining and very large time steps. Meanwhile, it is imperative to correct for the effect of the lost degrees of freedom during this coarse graining to ensure correct dynamics of the coarse-grained system. More formally speaking, the coarse-graining process is essentially a process of model reduction, where a smaller set of variables are used in the hope that this reduced representation can still approximate the original full system with good accuracy. However, as have been pointed out by Mori [104] and Zwanzig [172], the lost degrees of freedom will eventually reemerge as noise and memory terms. The DPD framework allows for the explicit incorporation of the noise and memory terms, and has seen popularity in areas such as biofluids and soft condensed systems thanks to its ability to correctly capture both static and dynamic properties.[35].

MD, CGMD, and DPD all represent bottom-up particle methods in which the concept of particle is an intuitive generalization of atoms and molecules. In contrast, there is another category of particle methods, which are derived following a top down approach. In these methods, the identify of a particle is no longer endorsed by a certain amount of substance. Instead, the particles are merely used as grid points, *i.e.* mobile carriers of degrees of freedom, on which certain partial differential equations can be efficiently solved for. Typical examples of top-down methods includes Smoothed Particle Hydrodynamics (SPH) and Smoothed Dissipative Particle Dynamics (SDPD).

Particle methods used in this thesis primarily fall within the bottom-up category. Specifically, the formulation and implementation of the DPD method will be detailed in Chapter 2 Section 2.1 and Section 2.2. A coarse-grained molecular dynamics simulator will be introduced in Chapter 2 Section 2.3. I will make a brief use of a top-down particle method, *i.e.* the SPH method, in Chapter 3 Section 3.4.2. Chapter 4 presents a demonstration of the capability of DPD in studying real world scientific problems. An algorithm for accelerating ab-initio atomistic molecular dynamics simulations will be introduced in Chapter 5.

1.2 Mesoscale Science and Multiscale Modeling

Many research topics, whether computational or 'conventional', can be loosely classified as being either 'macroscopic' or 'microscopic', depending on the abstractions and equations that underpin the topic. Interestingly, there is often a direct correspondence between pairs of macroscopic and microscopic theories. For example, thermodynamics can be considered macroscopic because it primarily concerns the relationships between macroscopic observables such as pressure, energy, and temperature; its microscopic counterpart is statistical mechanics, which builds upon the concepts of molecules, Hamiltonians, and partition functions *etc*. Another example is hydrodynamics versus molecular dynamics, where the former is described by the phenomenological Navier-Stokes equation while the latter is governed by the Schrödinger's equation and Newton's law of motion.

Given a pair of macroscopic and microscopic theories, it is natural to ask how microscopic theories, which describes the non-linear and sometimes stochastic interactions between the individual building blocks of a system, can eventually

give rise to collective macroscopic behaviors that are deterministic, smooth, and predictable. In fact, The most challenging problem in understanding physical and biological systems is to correctly account for the mesoscale, the intermediate temporal and spatial scale which links microscopic dynamics to macroscopic functional behaviors. The challenge is huge, however, because the mesoscale often exhibits strong fluctuation and memory effects that spawn diverse phenomena such as anomalous transport, rare events, and heterogeneous interfaces. Hence, we must go beyond mean-field theory and tools to achieve an in-depth understanding of mesoscopic events.

Multiscale simulation, as an alternative approach to study mesoscopic systems, was proposed to circumvent the challenge in seeking a unifying theory that can treat the diverse mesoscale phenomena. Distinct from 'monolithic' methodologies, the theory and practice of multiscale and multiphysics modeling are dedicated to combining existing techniques. The potential of multiscale modeling lies in its ability in probing properties of hierarchical systems by capturing events that occur across a wide range of time and length scales that exceed the capability of any single solver and method [118, 102, 155]. Common multiscale modeling and simulation approaches includes parameter passing, domain decomposition (DD), heterogeneous multiscale method (HMM) [156].

Apart from theoretical developments, the implementation of concurrently coupled simulations can be inherently difficult. On one hand, hard-coding remains commonplace in projects that employ *ad hoc* coupling approaches. Such practice can quickly become an obstacle when further development is needed. On the other hand, despite the richness of available coupling schemes and tools [5, 78, 101, 30, 111, 15], adapting existing code to meet the programming interface specification of a coupling framework frequently leads to code refactoring

that consumes a substantial amount of labor [4]. Such frameworks could also be cumbersome, especially for theorists without an expertise in software engineering, when prototyping new coupling schemes.

In Chapter 3, I will discuss about our recent work in bringing in the idea of meshless interpolation for coupling numerical methods of different nature. This approach allows for the easy adoption and implementation of multiscale coupling, especially using existing solvers that were not initially design for coupling purposes. A library, *i.e.* the Multiscale Universal Interface, was implemented as a demonstration of the idea and concept. The library provides a concise set of interfaces that can be adapted to a wide range of simulation protocols and scenarios.

1.3 Towards Exascale Computing

The speed of computer has kept increasing exponentially following the Moore's law. The statement is true — at least if we measure the total amount of arithmetic operations that can be carried out per unit time, the capacity of the various levels of the storage hierarchy, and the bandwidth of high-speed interconnects. The statement, however, can fail should we start to examine the arithmetic throughput of a single processing core (sans vector instructions), the access latency of DRAM, or the round-trip latency of a fiber-optics interconnect. In fact, the performance of a scalar core has only been improving marginally in the last ten years, while the access latency between the CPU and the main memory has basically reached stagnation.

Not surprisingly, the performance improvement that we have been continuously enjoying when running large-scale scientific computations is the outcome of an exponentially growing amount of parallelism, both on a single die and across the thousands of nodes in a parallel cluster. For example, a NVIDIA Volta GPU may contain up to 64 streaming multiprocessors each capable of running 2048 simultaneous threads. A 2-socket AMD EPYC server consists of 8 NUMA nodes, 32 physical cores, and 64 simultaneously hardware threads. An IBM Power9 processor may contain up to 24 physical cores running a total of 96 simultaneous threads spread over 4 or 8 NUMA nodes. Obviously, this brute-force tiling of processing elements would not deliver any actual performance improvement without the synergistic effort of parallel algorithm design and programming library development. In fact, choreographing the execution and data flow between a massive number of potentially heterogeneous software threads is of vital importance to fully exploit modern computer hardware.

In Chapter 2, I will introduce my work on utilizing massively parallel processors as well as accelerators to implement high performance particle-based simulators. Specifically, Section 2.2 concerns the design of parallel algorithms for the CUDA GPGPU architecture, while Section 2.3 discusses optimal algorithm for harnessing hundreds of simultaneous CPU threads.

In addition, the abundance of computing power and storage capacity has given rise to the popularity of statistical and machine learning algorithms. As such, it is worthy to discover the opportunity for applying data science algorithms to accelerate conventional computational science methods and algorithms. I will explore this front in Chapter 5, where a fingerprint algorithm is proposed for learning high-dimensional non-linear potential energy surfaces of *ab initio* quantum mechanical systems.

Chapter Two

Particle Methods and HPC Particle Simulator

2.1 Formulation of Dissipative Particle Dynamics (DPD)

In a DPD simulation, the movement of particles are governed by the Newton's equation of motion as [54]

$$\frac{d\mathbf{r}}{d\mathbf{t}} = \mathbf{v}, \quad \frac{d\mathbf{v}}{d\mathbf{t}} = \mathbf{a} = \frac{\mathbf{f}}{m} \tag{2.1}$$

The force **f** acting on each particle consists of three pairwise additive parts: a conservative component, a dissipative component and a random component, *i.e.*

$$\mathbf{f}_i = \sum_{i \neq j} \mathbf{F}_{ij} = \sum_{i \neq j} (\mathbf{F}_{ij}^C + \mathbf{F}_{ij}^D + \mathbf{F}_{ij}^R) \tag{2.2}$$

given by

$$\mathbf{F}_{ij}^C = a_{ij} w_C(\mathbf{r}_{ij}) \mathbf{e}_{ij} \tag{2.3}$$

$$\mathbf{F}_{ij}^D = -\gamma_{ij} w_D(\mathbf{r}_{ij}) (\mathbf{e}_{ij} \cdot \mathbf{v}_{ij}) \mathbf{e}_{ij} \tag{2.4}$$

$$\mathbf{F}_{ij}^R = \sigma_{ij} w_R(\mathbf{r}_{ij}) \xi_{ij} \delta t^{-\frac{1}{2}} \mathbf{e}_{ij} \tag{2.5}$$

if $|\mathbf{r}_{ij}| \leq r_c$, where r_c is the cutoff distance, and

$$\mathbf{F}_{ij}^C = \mathbf{F}_{ij}^D = \mathbf{F}_{ij}^R = 0, \quad |\mathbf{r}_{ij}| > r_c. \tag{2.6}$$

One of the two weight functions w_D and w_R can be chosen arbitrarily, while the other is then fixed through the fluctuation-dissipation constraint [35]

$$w_D(\mathbf{r}_{ij}) = w_R^2(\mathbf{r}_{ij}) \tag{2.7}$$

In our implementation, w_R is chosen to be a power of w_C as

$$w_R(\mathbf{r}_{ij}) = w_C^s(\mathbf{r}_{ij}) = (1 - \frac{|\mathbf{r}_{ij}|}{r_c})^s \tag{2.8}$$

The exponent s can be chosen arbitrarily, making it a convenient tuning parameter for reproducing the viscosity of a wide range of fluids. The coefficients σ_{ij} and γ_{ij} are related to each other by

$$\sigma_{ij}^2 = 2\gamma_{ij}k_BT \tag{2.9}$$

as also dictated by the fluctuation-dissipation theorem.

2.1.1 Energy-Conserving Dissipative Particle Dynamics

The eDPD method [88] extends the original DPD framework [54] by explicitly modeling each particle's internal energy as a degree of freedom:

$$\frac{d\mathbf{r}_i}{dt} = \mathbf{v}_i \tag{2.10}$$

$$m\frac{d\mathbf{v}_i}{dt} = \mathbf{f}_i = \sum_{i\neq j}(\mathbf{F}_{ij}^C + \mathbf{F}_{ij}^D + \mathbf{F}_{ij}^R) \tag{2.11}$$

$$C_v\frac{dT_i}{dt} = q_i = \sum_{i\neq j}(Q_{ij}^C + Q_{ij}^V + Q_{ij}^R) \tag{2.12}$$

where t, $\mathbf{r}_i$, $\mathbf{v}_i$, $\mathbf{f}_i$, m_i, T_i, q_i and C_v denote time, position, velocity, force, mass, temperature, heat flux, and heat capacity of DPD particles.

The force $\mathbf{f}_i$ acting on each particle consists of three pairwise-additive components, *i.e.* a conservative one, a dissipative one, and a random one:

$$\mathbf{F}_{ij}^C = \quad a_{ij}(T_{ij})w_C(r_{ij})\mathbf{e}_{ij} \tag{2.13}$$

$$\mathbf{F}_{ij}^D = -\gamma_{ij}w_D(r_{ij})(\mathbf{e}_{ij}\cdot\mathbf{v}_{ij})\mathbf{e}_{ij} \tag{2.14}$$

$$\mathbf{F}_{ij}^R = \quad \sigma_{ij}w_R(r_{ij})\xi_{ij}\delta t^{-\frac{1}{2}}\mathbf{e}_{ij} \tag{2.15}$$

where $T_{ij} = (T_i + T_j)/2$ is the pairwise local temperature between particle i and j. A common choice for the pairwise repulsive force coefficient $a_{ij}(T_{ij})$ that reproduces the compressibility of water is

$$a_{ij}(T_{ij}) = \frac{75k_BT_{ij}}{\rho}. \tag{2.16}$$

The noise level σ_{ij} and the pairwise friction coefficient γ_{ij} are related to each other through the fluctuation-dissipation theorem [35]:

$$\sigma_{ij} = \sqrt{2\gamma_{ij}k_BT_{ij}}. \tag{2.17}$$

The pairwise heat flux also consists of three components:

$$Q_{ij}^C = k_{ij}w_{CT}(r_{ij})(\frac{1}{T_i} - \frac{1}{T_j}) \tag{2.18}$$

$$Q_{ij}^V = \frac{1}{2C_v}\left\{w_D(r_{ij})\left[\gamma_{ij}(\mathbf{e}_{ij}\cdot\mathbf{v}_{ij})^2 - \sigma_{ij}^2/m\right]\right.$$
$$\left.-\sigma_{ij}w_R(r_{ij})(\mathbf{e}_{ij}\cdot\mathbf{v}_{ij})\xi_{ij}\delta t^{-\frac{1}{2}}\right\} \tag{2.19}$$

$$Q_{ij}^R = \beta_{ij}w_{RT}(r_{ij})\zeta_{ij}\delta t^{-\frac{1}{2}} \tag{2.20}$$

while the contact heat flux coefficient k_{ij} and random heat flux level β_{ij} can be

determined as

$$k_{ij} = \frac{C_v^2 \kappa (T_i + T_j)^2}{4k_B} \tag{2.21}$$

$$\beta_{ij} = 2k_B k_{ij} \tag{2.22}$$

2.2 USERMESO: Scalable and Extensible GPU Computing for Mesoscopic Particle Methods

2.2.1 Introduction

Particle-based simulation has continuously benefited from the ever growing computing power provided by each new generation of hardware. Among the spectrum of parallel processors, the general purpose graphics processing unit (GPGPU) proves to be a particularly good fit for such simulation due to the massively parallel nature shared by inter-particle interaction evaluation and image rendering [6, 140, 98, 45]. Specifically, the Compute Unified Device Architecture (CUDA) has provided a parallel programming model that could harness the processing power of the GPGPUs, which is tens of times more than the contemporary CPUs [47, 107]. There is currently an array of molecular dynamics simulation applications that integrate the capability to use CUDA for part or all of the computation, such as AMBER, DL_POLY, HOOMD-BLUE, LAMMPS, NAMD and GROMACS [51, 136, 6, 116, 67, 119].

The combination of fast hardware and highly optimized software package enables large-scale simulations of atomistic systems. For example, a protein folding

simulation of 64 million particles using an all-atom resolution has been successfully carried out using NAMD for a total integration time of 100 ns [169]. Nevertheless, the classical all-atom molecular dynamics (AAMD) method is still prohibitively expensive for simulating atomistic systems at a length and time scale that is comparable to that in experiment. To bridge the gap between the microscopic time/length scale and the macroscopic ones, coarse-grained simulation techniques such as coarse-grained molecular dynamics (CGMD), dissipative particle dynamics (DPD) and smooth particle hydrodynamics (SPH) have been proposed in which a group of atoms in the AAMD model is represented by a single coarse-grained particle to reduce the computational complexity [62, 54, 35, 103, 129].

Among the various coarse-grained models, DPD has been our particular interest since it accurately reproduces the equilibrium and dynamic properties of systems in the area of biophysics, soft matter and fluid dynamics [87, 113, 39]. Unfortunately, little effort has been directed to the efficient implementation of DPD on the GPUs. Despite the intrinsic similarity in the governing principles, the DPD systems differ from their atomistic counterparts in that:

- A more complex functional form is used, *i.e.* a dissipative force, which involves the relative velocity in addition to the relative position of the interacting particles.
- A pairwise random force is essential to compensate for the reduced degrees of freedom in order to correctly reproduce the dynamics of the coarse-grained system.
- The system is sparse, which results in a much lower average neighbor count for the particles and makes it harder to achieve optimal performance on the GPU's SIMD architecture.

So far, Goga *et al.* have presented a GPU parallelization of the SD and DPD types in the GROMACS tools [50], with which they observed a performance boost of 70% on the GPU over a single processor. Phillips *et al.* proposed a hash-function based pseudo-random number generator for the evaluation of the stochastic term [114]. Wu *et al.* reported a GPU implementation of DPD with parallel neighbor list updating [162]. Wang *et al.* reported a multi-GPU implementation, which achieves a speedup of 90x on three GPUs over a single-threaded CPU implementation from Material Studio [152]. However, no scalable implementation that runs across nodes has been reported so far. In this section, I demonstrate a scalable implementation of the DPD simulation on the Kepler GPGPU architecture using a CUDA-MPI hybrid programming model targeting optimal performance within a GPU and also across nodes.

2.2.2 Algorithms and Implementation

We start with LAMMPS as the baseline code for the CPU portion of the program [116]. The GPU code is written in CUDA C due to the high level of availability and maturity. The basic control flow is shown in Algorithm 1 where the highlighted text indicates parts that are accelerated by the GPU. The code essentially runs in a time-stepping loop where tasks such as trajectory integration, communication, force evaluation, statistical analysis and data I/O are executed in order according to the Velocity Verlet algorithm [143]. Stray particles that run across domain boundaries are exchanged each time the neighbor list is rebuilt. For parallel execution over MPI, the default spatial decomposition scheme of LAMMPS is used, in which the entire simulation domain is divided into equally sized subdomains of the same shape. Each MPI rank handles a single GPU. Multiple MPI ranks can be launched

on one multi-GPU node and occupy the GPUs in a round-robin manner according to their local ranks.

Algorithm 1 Basic program control flow. The highlighted text indicates portions of the computation that involve the GPU.

```
1:  ▷ setup
2:  setup compute domain, determine particle ownership
3:  CPU --data--> GPU
4:  obtain ghost particles information from neighboring processors
5:  build neighbor list
6:  compute forces
7:  ▷ main loop
8:  for n steps do
9:      time integration: phase 1
10:     if time to rebuild neighbor list then
11:         GPU --data--> CPU
12:         exchange stray particles
13:         CPU --data--> GPU
14:         obtain ghost particles information from neighboring processors
15:         rebuild neighbor list
16:     else
17:         exchange ghost particle information
18:     end if
19:     compute forces
20:     time integration: phase 2
21:     calculate statistics
22:     if on demand then
23:         display runtime info, dump trajectory, write restart file, etc.
24:     end if
25: end for
```

LAMMPS uses an Array of Structures (AoS) layout to store vector-valued particle properties such as coordinate, velocity and force as illustrated in Figure 2.1. Such layout results in strided access on the GPU, *i.e.* threads with consecutive thread IDs will access memory locations that are separated by a stride larger than one, which reduces the effective memory bandwidth. On the other hand, too much

code modification is required if we were to change a data layout which is used essentially everywhere in the original program. As a compromise we used a pair of interleave/deinterleave kernels to convert data between the AoS and the SoA layouts at runtime. The kernels are carefully designed to achieve high memory bandwidth efficiency as summarized in Table 2.1.

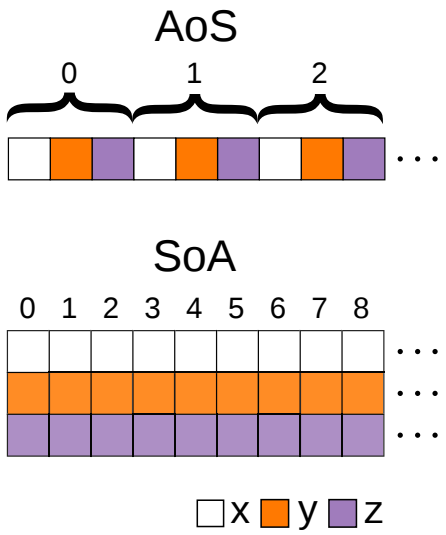


Figure 2.1: In the array of structure (AoS) layout, the coordinate vector for each particle is placed consecutively, whereas any specific component of the vector are separated by the other ones. In the structure of array (SoA) data layout, components are stored consecutively.

Table 2.1: Achieved bandwidth of the data marshaling and bandwidth-sensitive kernels. The benchmarks were done on a Kepler K20 GPU with a peak memory bandwidth of 208 GB/s.

Kernel	Read (GB/s)	Write (GB/s)	Aggregate (GB/s)
Interleave	79.59	79.62	159.21
Deinterleave	72.21	70.80	143.01
NeighborListJoin	62.66	57.83	120.49
NeighborListTranspose	78.03	79.06	157.09
MergeXVT	95.78	54.66	150.44

Sorting

The parallel sorting primitive plays a fundamental role in our code for reordering particles and building the cell list. Several existing algorithm packages such as Thrust and CUDA Data Parallel Primitives Library (CUDPP) provide high-level interfaces for this type of workload (CUDPP actually uses Thrust::sort as its back-end) [131, 60]. However, neither package supports the CUDA streams. In other words, they can only launch kernels on the default stream 0. In our code, streaming is essential for achieving high GPU computing efficiency as it helps to hide communication and kernel launching latency. Moreover, our benchmark reveals that these packages are better optimized for large arrays with millions or billions of key/value pairs, while in practical DPD simulations the particle number per GPU does not exceed a few millions. Therefore, we implemented our own radix sort that is optimized specifically for smaller arrays. As shown in Algorithm 2, each successive invocation of three kernels `gpuRadixHistogram`, `gpuRadixPrefixSum` and `gpuRadixPermute` completes one pass of the sorting algorithm, while 4 bit per pass was determined to be the optimal radix width through experimentation. The scan primitive by Sengupta *et al.* combined with the newly-introduced warp shuffle instructions is used to carry out the prefix summation [132]. The benchmark presented in Figure 2.2 compares the performance of our implementation versus Thrust over a wide range of array sizes.

Particle Reordering

Unlike grid/lattice-based methods, particle-based simulation deals with systems whose structures are largely irregular. As a result, the memory access pattern

Algorithm 2 Pseudo code for the radix sort template function.

```
1: function RadixSort<RADIX>(KeyArray, ValArray, BitLength, TargetStream)
2:     BufferIn ← {KeyArray, ValArray}
3:     BufferOut ← InternalBuffer
4:     for bit = 0 to BitLength by Log2(RADIX) do
5:         gpuRadixHistogram(BufferIn,BufferOut,PrefixBuffer,bit) on TargetStream
6:         gpuRadixPrefixSum(PrefixBuffer) on TargetStream
7:         gpuRadixPermute(BufferIn,BufferOut,PrefixBuffer,bit) on TargetStream
8:         SwapPointer(BufferIn, BufferOut)
9:     end for
10:    if BufferOut ≠ {KeyArray, ValArray} then
11:        {KeyArray, ValArray} ←cudaMemcpy— BufferOut
12:    end if
13: end function
```

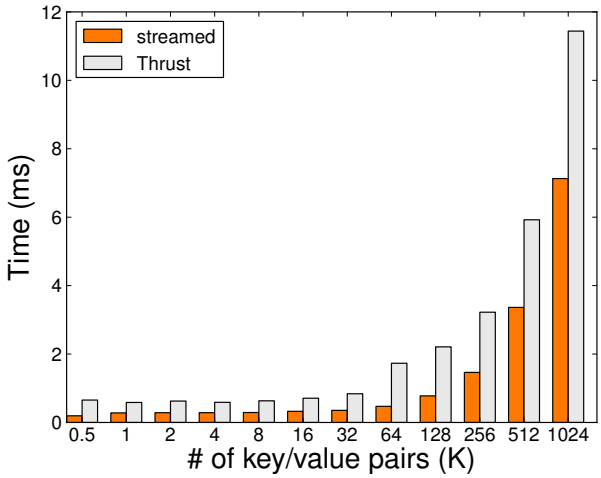


Figure 2.2: Sorting time: our streamed radix sort *vs.* Thrust.

of typical particle simulation codes exhibits poor locality. The situation turns out to be worse on GPU due to the SIMD nature of the GPU architecture and a lower per-core bandwidth. In fact, most particle-based simulation programs are memory-bound, while DPD suffers even more because both coordinate and velocity data are required for evaluating the pairwise interaction. Particle reordering along a space-filling curve has been proposed as an effective technique to mitigate the problem. For example, a space-filling curve pack (SFCPACK) algorithm is used in the HOOMD-blue simulation package to sort particles on a per-block basis [6]. Our code, on the other hand, uses a two-level Morton encoding (Z-curve) scheme to reorder particles as shown in Figure 2.3. In this scheme, cells are first sorted according to their Morton order, while at the same time particles within a cell are also sorted by their relative position along a local Morton curve in each individual cell. In addition, the boundary of the cells used for particle reordering was chosen to coincide with that used for building the cell list. The two-level Morton encoding achieves good locality while at the same time ensures particles belonging to the same cell have consecutive indices. This feature is later exploited by our neighbor list building algorithm. Note that at this point only local particles have been sorted and in order to proceed we need to obtain information of ghost particles from neighboring processors; this will be discussed in section 2.2.6. Figure 2.4 summarizes the performance gain brought about by this particle reordering scheme for DPD fluid simulations at four different sizes compared to those without particle reordering using the same set of parameters. The overall program performance is boosted by almost 100%, while individual GPU kernels benefit even more depending on their level of memory consumption. The effect of particle reordering is further characterized in Figure 2.5, which visualizes the interaction matrices for a system of 2048 particles with and without particle reordering. While diffusion tends to randomize the distribution of interactions

over the entire domain, reordering on the other hand strongly diagonalizes the matrix and aggregates off-diagonal components into concentrated blocks.

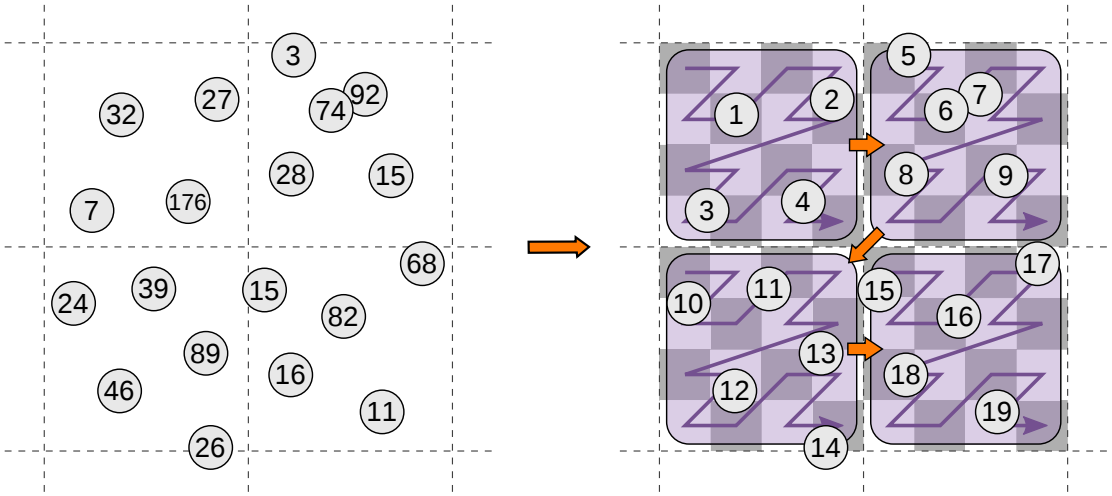


Figure 2.3: A 2-level Morton curve is obtained by ensuring that the number of bins for reordering is a multiple of the number of bins in the cell list by some integer power of 2.

Cell List

Our program adopts the conventional approach of using the neighbor list to accelerate the evaluation of non-bond pairwise interactions. In addition, a cell list is used to further facilitate the construction of the neighbor list. The cell list construction starts with *binning*, *i.e.* partitioning the simulation box into a number of equally sized cells and assigning particles into their corresponding cells. This can be done efficiently using parallel sorting with the particles' cell

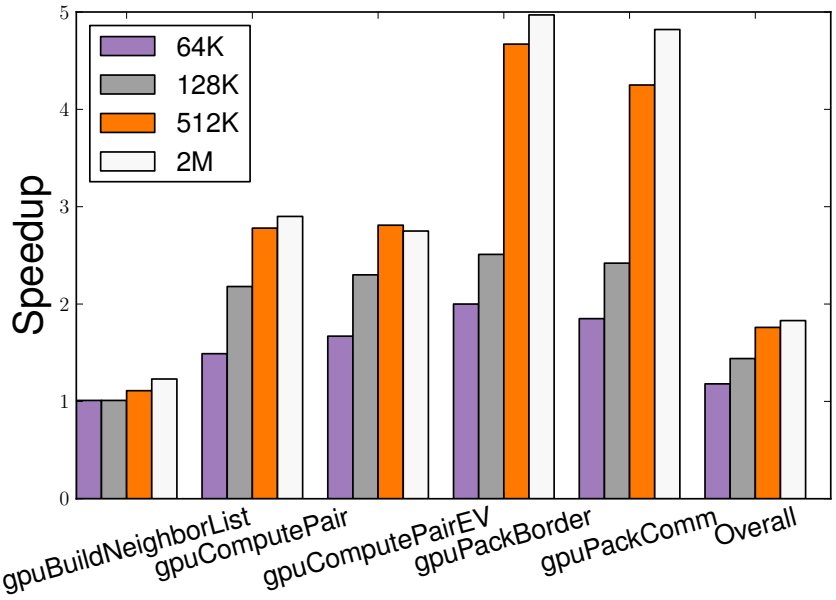


Figure 2.4: Performance gain due to particle reordering for the whole program and individual kernels. This metric also indirectly reflects the memory bandwidth consumption for each kernel.

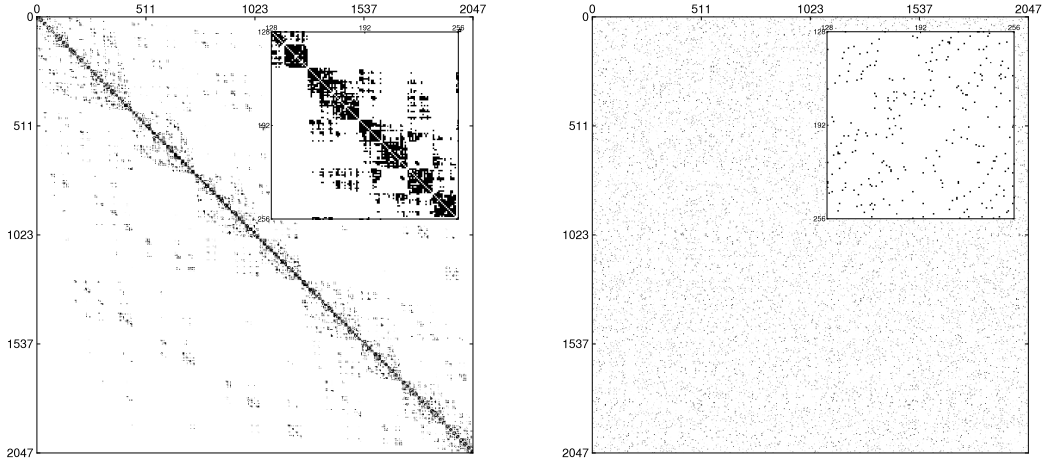


Figure 2.5: Interaction matrices obtained for 2048 particles with/without particle reordering. Reordering strongly diagonalizes the matrix and aggregates off-diagonal components into concentrated blocks. Particle diffusion results in randomly distributed interaction when reordering is off.

indices being the input key for sorting, while the determination of the cell index depends solely on the particle's own position and hence is embarrassingly parallel. This algorithm is employed by the GPU module of the LAMMPS package [22], which contrasts the approach in Wu's implementation where the atomic increment operation, a potential bottleneck at high parallelism, is used to build the cell list [162]. Alternatively, since particle indices are consecutive within each cell after the reordering process, the construction of the cell list can be performed by performing particle reordering followed by detecting the cell boundary in the one-dimensional particle array without the need for further sorting of cell indices. The boundary detection is done by launching one thread per particle, checking if its cell ID differs from the particle that precedes it. In our code we adopted the latter approach.

2.2.3 Neighbor List Construction

Stencil

As the first step, a coarse-grained stencil list is generated, which contains one stencil for each cell. Each stencil stores the indices of the neighboring cells surrounding its master cell. Within each stencil the cell indices are sorted according to the Morton order as illustrated in Figure 2.6. Such ordering of the stencil combined with the aforementioned particle reordering technique ensures that the particle indices in each stencil are monotonically increasing. Since the building of a coarse-grained stencil need only to be done once during the entire simulation, the sorting is simply implemented using bubble sort. The coarse-grained stencil list is then expanded into a fine-grained one, storing the actual indices of particles in each cell's neighboring cells. This is particularly useful for our DPD application

because the particle density in a DPD simulation is much lower than that of an AAMD simulation. For example, a cell of volume $(r_c + \Delta r)^3$ in a DPD system contains on average 10 particles, where Δr is the skin distance for building the neighbor list. The fine-grained stencil allows the neighboring particles from multiple cells to be loaded into our neighbor list builder in a coalesced manner without incurring branch divergence. Both the coarse-grained and the fine-grained stencils are stored in a row-major fashion.

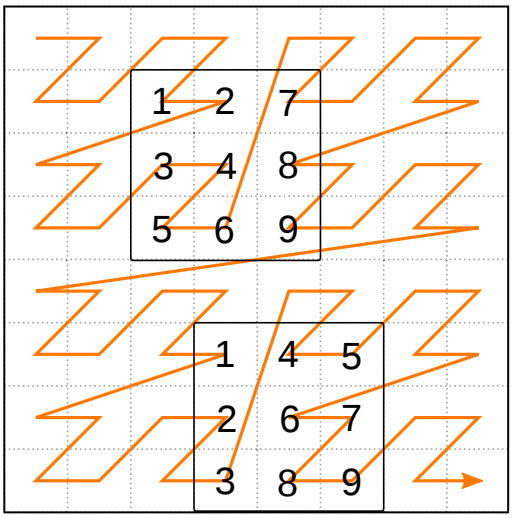


Figure 2.6: Expanding stencils along the same Morton curve for particle reordering ensures monotonicity of the particle indices within each stencil.

Neighbor List

For the actual job of building the neighbor list, we invented an atomics-free algorithm that is able to generate a fully ordered neighbor list in parallel. The algorithm is deterministic, meaning that the generated neighbor list will always be the same given the same initial configuration and runtime parameter. Such deterministic approach may benefit debugging and simulation cases that require reproducibility.

The neighbor list builder works on the fine-grained stencil. Instead of the common practice of assigning one block for each cell, a warp-centric programming model, in which each cell is taken care of by a single warp, is employed to optimally balance data reuse and inter-thread communication. A fixed number of shared memory slots are assigned to each warp for storing the information of the *i* particles in its working cell. Each slot consists of two integers and three fp32 values for the particle index, neighbor count and the x, y, z coordinate. In practice, 32 slots are allocated to each warp to coincide with the warp size, though in theory any number can be used as long as the shared memory capacity is not exceeded. The storage needed for each slot is $4 + 2 \times 2 + 4 \times 3 = 20$ bytes. On the Kepler architecture, an occupancy of 100% translates into 2048 concurrent threads per stream multiprocessor and hence a shared memory utilization of $20 * 2048/1024 = 40$KB, which fits well into the 48 KB on-chip shared memory.

As illustrated in Figure 2.7, each warp in our neighbor list builder works in a triple loop. The outer loop walks through the *i* particles in its working cell with a batch size of 32, with each thread loading one particle's index and coordinate into a slot in the shared memory. The middle loop scans over the cell's stencil with each thread loading one *j* particle into its private registers at a time. The inner loop performs the actual all-over-all distance checking with all threads looping through the shared *i* particles together. Such loop arrangement minimizes the *long tail* effect because branch divergence occurs only in the last iteration of the middle loop if the stencil length is not a multiple of the batch size. Typical stencils contain 200 particles each so the performance hit due to a branched last iteration is not significant.

Because multiple *j* particles may be found staying within the cutoff distance of one *i* particle at the same time, a commit of the *j* index into the correct position

of the neighbor list intuitively requires the use of atomic increment operation to resolve potential conflict. This can also be thought of as each thread needs the global hit/miss information from other threads to determine the insertion point for its own commit. However, realizing that in our algorithm at any given time there is only one warp evaluating the neighbors for a given particle, we can reduce the hard problem of global information gathering into a simpler one of collecting a Boolean value from a warp. The warp vote function, __ballot(), fits naturally for this job by enabling fast communication between the threads and eliminating the use of atomic instructions. The ballot intrinsics takes in a predicate value from each active threads of a warp and returns to each thread an unsigned integer whose N-th bit is set if the predicate evaluates to true for the N-th thread. In the inner loop of our neighbor list builder, each thread ballots its distance checking predicate to form a bit vector `nHit`. Shifting `nHit` by `warpSize - laneId` toward left, where `lane id` $\in [0..$`warpSize` $- 1]$ is the ordinal number of a thread within a warp, and then counting the number of remaining set bits in the bit vector tells a thread the number of hits made by threads with a lower `lane id` than itself. The insertion point is then simply determined by adding this number to the existing number of neighbors for the i particle from previous iterations. The number of hits from an iteration is then accumulated to the total neighbor count for the i particle by a single thread, in our case lane 0, after the cooperative neighbor list insertion. In contrast to the atomic-based approach, this warp vote-based approach for committing particle indices into the neighbor list is completely deterministic and only involves fast intra-warp communication. Recall that the indices in the stencil are monotonically increasing. This implies that the neighbor list generated in this way is also strictly increasing for each particle, yet no sorting was performed explicitly on the neighbor list. A careful benchmark shows that our atomics-free builder executes twice as fast on average compared to an equivalent

builder using atomic increment as shown in Table 2.2.

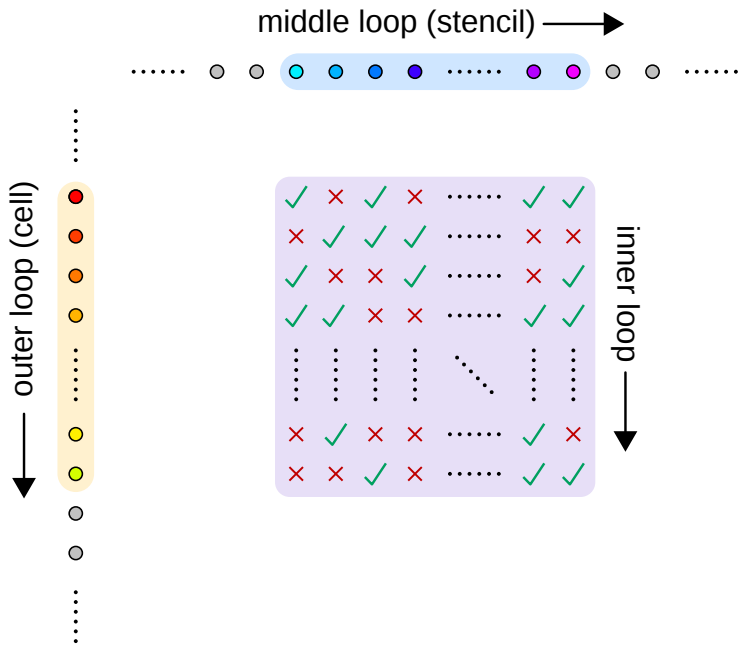


Figure 2.7: A graphical visualization of the triple loop in the atomics-free neighbor list builder.

Table 2.2: Performance comparison of the neighbor list builders using warp ballot or atomics increment.

system config	time(ms) ballot	time(ms) atomic	speedup
$\rho = 3, n = 64k$	0.76	1.55	2.02
$\rho = 3, n = 512k$	5.28	10.77	2.04
$\rho = 6, n = 64k$	1.31	2.92	2.24
$\rho = 6, n = 512k$	9.89	21.72	2.20
$\rho = 50, n = 64k$	14.58	30.15	2.07
$\rho = 50, n = 512k$	90.28	196.26	2.17

Another issue to be considered here is the layout of the neighbor table. A row-major layout assigns one consecutive line of memory for each particle, while a column-major layout, on the other hand, stores the n-th neighbors of all the particles consecutively in the n-th row. On the GPU we typically assign each thread a particle when evaluating the pairwise interaction, which implies that a column-major layout is more efficient because fetching such a list results in fully

Algorithm 3 Atomics-free neighbor list builder

```
1:  function gpuBuildNeighborList
2:      for each cell in parallel do
3:          shared slot[warpSize]
4:          l ← threadIdx mod warpSize
5:          n ← number of particles in the cell
6:          p ← 0
7:          ▷ load working cell
8:          while p < n do
9:              part ← Min( n - p, warpSize )
10:             if l < part then
11:                 slot[l].i ← cell[ p + l ]
12:                 slot[l].r ← Texture(coordinate, slot[l].i )
13:                 slot[l].nn ← 0
14:             end if
15:             ▷ load stencil
16:             for k = l to nStencil by warpSize in parallel do
17:                 j ← stencil[k]
18:                 r ← Texture(coordinate,j)
19:                 ▷ distance check
20:                 for i = 0 to part by 1 do
21:                     dr^2 ← ( r - slot[i].r )^2
22:                     hit ← dr^2 < r_c^2 ? true : false
23:                     nHit ← Ballot(hit)
24:                     nAhead ← ShiftLeft(nHit, warpSize - l)
25:                     pIns ← slot[i].nn + Popc( nAhead )
26:                     if hit then
27:                         neighborList[ slot[l].i ][ pIns ] ← j
28:                     end if
29:                     accumulate Popc(nHit) to slot[i].nn by lane 0
30:                 end for
31:             end for
32:             if l < part then
33:                 store slot[l].nn to global storage
34:             end if
35:             p ← p + warpSize
36:         end while
37:     end for
38: end function
```

coalesced memory loads. However, a column-major table is much less efficient when being written to because an entire warp works on the same *i* particle in the inner loop in our neighbor list builder. In this case, each insertion of the *j* indices from a thread generates a write request of 32 bytes, among which only 4 bytes are actually useful data, resulting in a memory efficiency merely 12.5%. To combine the advantages from both layouts, we adopt a write-transpose-read approach in which the neighbor table is built in a row-major fashion, and then transposed for maximum loading efficiency in the subsequent computation. In practice, we only *locally in-place* transpose each 32 by 32 tile of the neighbor table matrix. We made this compromise because a full transposition of such a large matrix is complicated with sub-optimal efficiency [142]. A schematic representation of the local transposition is presented in Figure 2.8.

Further, a double-insertion trick can be utilized by the neighbor list builder to help suppress branch divergence in the force kernel. This heuristic depends on the assumption that particles which were within the actual cutoff distance when building the neighbor list are more likely to be found within the cutoff distance again in subsequent time steps than those which initially lie between the cutoff and the skin distance. To separate the two types of neighbors in the neighbor list, the warps in our neighbor list builder can be modified to perform two commits per inner loop iteration, one for the *core* particles, and the other one for the *skin* particles. The *core* particle indices are stored in the original order, while the *skin* particle indices were stored at the back of each row in reversed order. The neighbor count integer used in the shared slots in the neighbor list builder is also split into two short integers correspondingly. In this way we still need only one table for storing the neighbor list. A separate kernel is used to join the two parts together, making the splitting trick completely transparent to the rest of the code. Note that

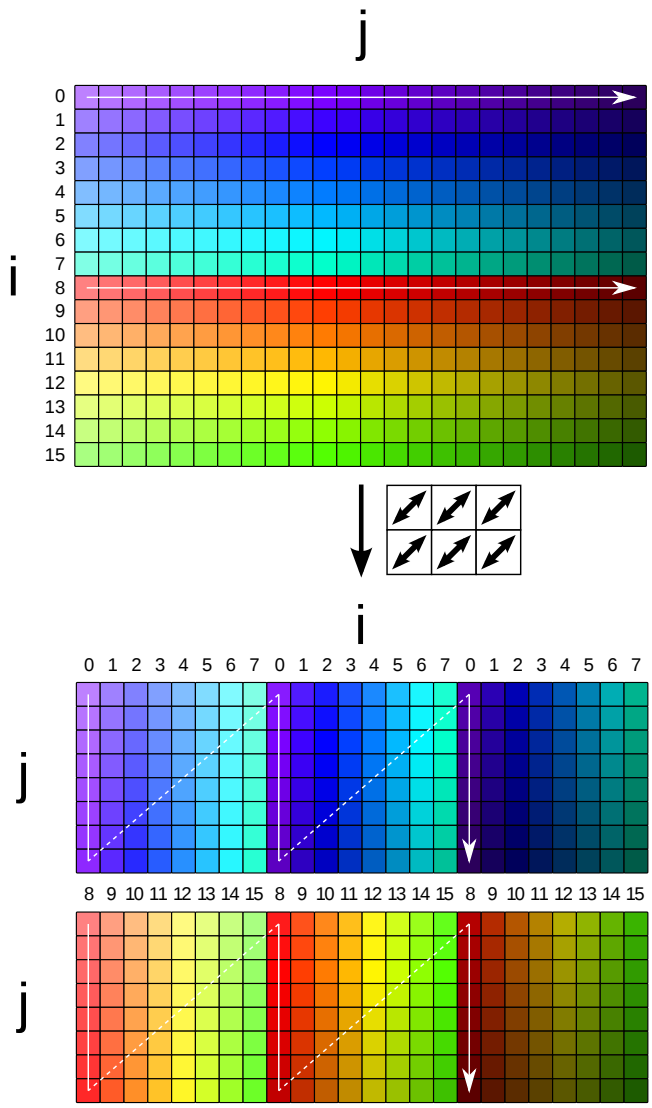


Figure 2.8: Local in-place transposition of the neighbor table. The memory footprint for a single thread when looping through its neighbors as indicated by the arrowed lines produces fully coalesced access.

the naive way of simply forcing every thread to enter the force evaluation branch regardless of the actual distance results in worse performance, since it issues more stress on the memory bandwidth and the texture units. It is even erroneous in DPD since the functional form does not accept $|\mathbf{r}_{ij}|$ greater than r_c.

2.2.4 Precision Model

We use a hybrid precision model for the neighbor list builder and force evaluation kernel to optimize GPU memory bandwidth usage. Force evaluation in DPD is expensive in terms of both memory bandwidth consumption and arithmetic complexity. The memory bandwidth consumption is high because both the coordinate and the velocity of the interacting particles are involved in the functional envelope. For each pair of interacting particles an input of 2 integer and 6 fp64 values, or a total of 56 bytes, is required to supply the required information. We used the standard approach of texture data mapping to alleviate the problem. Note that the coordinate and velocity are stored in double precision in our code everywhere else, but they are converted to single precision before being mapped as textures for force evaluation. Specifically, the single-precision coordinates are cast from the double-precision difference between the actual particle coordinates and the geometric center of the owning processor's compute domain. The double-precision velocity is cast to single precision directly. Our approach differs from that of Götz *et al* [52] in that the computation is still done in full double precision. The neighbor list builder works entirely in single precision because the distance comparison is not sensitive to precision. The possibility of using the newly-introduced non-coherent cache pathway to load such data was also explored. However, a microbenchmark revealed that on the current Kepler GPU the latency for a cache hit

in the non-coherent cache (140 cycles) is much longer than that of a texture fetch (108 cycles), though is still shorter than that of a regular global memory access (220 cycles) [159].

Table 2.3: Comparison between our custom double-precision math routines (prefixed with *fast*) and the CUDA native ones. This microbenchmark is done through a chain of dependent statements according to Ref [159].

Function	Instruction#	Conditional Branch	Latency(cycles)	Max Rel. Error
fastcospi	23	0	90	1.10×10^{-10}
cospi	54	3	420	1 ULP
fastlog	47	0	380	4.21×10^{-12}
log	89	5	603	1 ULP
fastpow	79	0	494	11 ULP
pow	258	18	1982	2 ULP

2.2.5 Numerical Optimization

The arithmetic complexity of the DPD pairwise interaction is high for two reasons. First, for each pair of the random force F_{ij}^R, one Gaussian random number has to be consumed. This translates into one square root, one natural logarithm, one trigonometric function and the generation of two uniform random numbers assuming the use of the Box-Muller algorithm [20]. Second, the power function is needed to evaluate the weight function $w_R(\mathbf{r}_{ij}) = w_C^s(\mathbf{r}_{ij}) = (1 - r/r_c)^s$ if the exponent s takes a value other than positive integers.

Using a separate RNG library to generate the random numbers beforehand and storing them in double precision requires a storage space that is twice the size of the neighbor list. Besides, it is inevitable for pairs of interacting particles to get separated onto different processors during domain partitioning. In this case, we either have to compute the pairwise interaction only once on one of the processors and then communicate the force back, or we have to reproduce the same

random number on different processors to save the communication. Generally, communication is expensive for the GPU so it is preferred if the random numbers can be reproduced. This dictates that we use lightweight, hash function-based generators and particle properties that are invariant across different nodes as the seed to generate the random numbers *in situ*. In particular, the tiny encryption algorithm (TEA) as shown in Algorithm 4 has been proposed as a suitable choice for DPD simulation with the input seeds chosen to be a mixture of the particle indices, current time step and a global seed [157, 166, 114].

The quality of the random numbers generated by TEA increases with the number of iterative rounds performed. Due to the limited sampling range of the input seeds in DPD (most simulations contain no more than a few million particles), at least 8 rounds of hashing are required to obtain sufficient randomness. We took a different approach of sampling from a wider source of entropy for the input seeds to guarantee randomness rather than simply increasing the number of iterative rounds. In particular, for each particle we take the bit-reversed particle identity tag as the first integer, and an interleaving of the first 11 bits of the mantissa of the three-dimensional velocity components as the second integer. Besides, we employ a pre-processing step to blend the two integers using 16 rounds of the TEA hashing. The two integer output from this preprocessing step are then combined using bitwise exclusive or into a single 32-bit binary signature for every particle. The overhead of such preprocessing is negligible since it is embarrassingly parallel with complexity $O(N)$. Only 4 rounds of TEA hashing are needed when the preprocessed binary signatures are used. In order to validate this approach, we dumped the random numbers generated during one simulation of 65,536 DPD particles and compared up to the 4-th moment of the numbers against that generated by MATLAB. The result matches perfectly as shown in Figure 2.9.

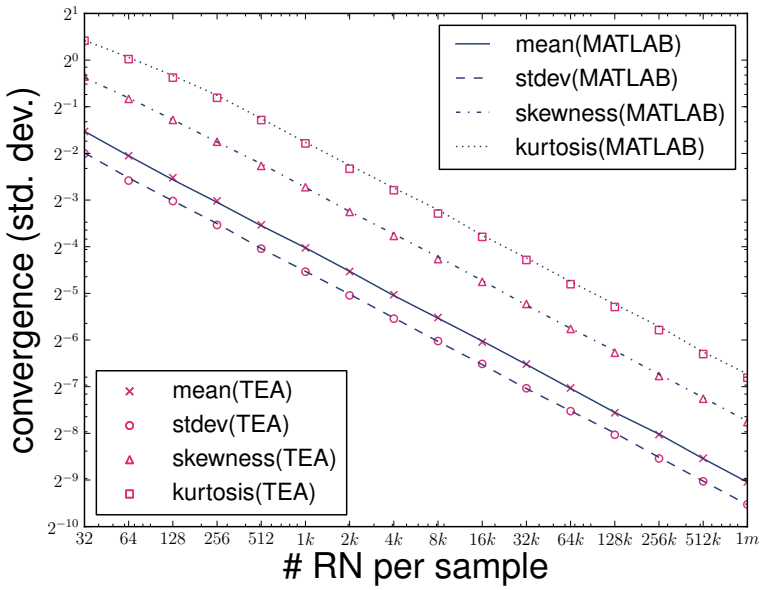


Figure 2.9: The convergence of the distribution of the random numbers generated by our signature-based TEA is compared to that generated by MATLAB using up to the 4-th moment.

Algorithm 4 The Tiny Encryption Algorithm [157]

and our signature generator.

function TEA(n, v_0, v_1)
    $sum \leftarrow 0$
    for n rounds **do**
        $sum \leftarrow sum + delta$
        $v_0 \leftarrow v_0 + \text{BitXor}(\text{ShiftLeft}(v_1, 4) + k0, \text{ShiftRight}(v_1, 5) + k1, v_1 + sum)$
        $v_1 \leftarrow v_1 + \text{BitXor}(\text{ShiftLeft}(v_0, 4) + k2, \text{ShiftRight}(v_0, 5) + k3, v_0 + sum)$
    end for
end function

function Preprocess(i)
    $v_0 \leftarrow \text{BitReverse}(\text{Tag}[i])$
    $b_x \leftarrow \text{GetMantissa}(\text{Velocity}[i].x, 11)$
    $b_y \leftarrow \text{GetMantissa}(\text{Velocity}[i].y, 11)$
    $b_z \leftarrow \text{GetMantissa}(\text{Velocity}[i].z, 11)$
    $v_1 \leftarrow \text{Interleave}(b_x, b_y, b_z)$
    TEA($16, v_0, v_1$)
    return BitXor(v_0, v_1)
end function

The uniform random numbers generated by the TEA algorithm above are converted to Gaussian using the Box-Muller equation. Profiling indicated that a plain implementation of the Box-Muller conversion alone consumes about 25% of the total time for pairwise force evaluation. This is actually not surprising since it involves the computation of transcendental functions as mentioned previously. However, there are facts that we can exploit to accelerate the math functions. First of all, we have perfect control over the range of input for the functions, *i.e.* the uniform random numbers are always between 0 and 1. In addition, even though we are producing double precision floating points as the final output, it does not necessarily mean that we need to compute in the full precision during the intermediate steps. By looking at the Box-Muller equation $z_1 = \sqrt{-2 \ln u_1} \cos(2\pi u_2)$, it is clear that the radial component of the number depends only on u_1, while the phase component depends only on u_2, both of which contain only 32 bits of entropy as specifies by the TEA algorithm. It is through the combination of the two parts that we obtain a result of full double precision.

Hence, we implement our own reduced-precision natural logarithm and cosine as follows:

a) Instead of converting the 32-bit unsigned integer v_1 into a double precision floating point number $u_1 \in [0, 1)$ and then performing the natural logarithm, we apply a binary logarithm directly to v_1. The evaluation can be split into the integral part and the fractional part as $\log_2 v_1 = \log_2 2^{I+F} = I + \log_2 2^F$, where I is an integer and F is a fractional number between 0 and 1. The integer part I can be obtained trivially using the relation $I = 31 - clz(v_1)$, where *clz* stands for *consecutive leading zeros*, which is a hardware-implemented PTX instruction on the Kepler GPU. The fractional part is approximated with an 8-th order Chebyshev polynomial as $\log_2 x = z \times P(z)$, where $z = \frac{x-1}{x+1}$ [57]. The natural logarithm is derived by

subtracting the binary logarithmic result by 32 and then multiplying by $\ln 2$. Our implementation compiles into 47 branch-free PTX instructions, which contrasts the CUDA native implementation that compiles into 89 PTX instructions with 5 conditional branches and one floating point reciprocal. The maximum relative error for our implementation is 4.21×10^{-12} over the entire range of unsigned integer, yielding a binary precision of 42.8 bits.

b) In the custom cosine function, the highest bit of v_2 is taken as a sign bit, while the low 31 bits are converted into a double precision floating point number $u_2 = v_2[0..30] \times 2^{-31} \in [0, 1)$. This simplifies the range reduction for the evaluation of cosine, enabling us to approximate the function with a 11-th order Chebyshev polynomial, which has only 6 terms thanks to the axial symmetry of the cosine function. The resulting function compiles into 23 branch-free PTX instructions using only cheap arithmetics such as bit shifting, floating point multiply and fused multiply-addition. The CUDA native implementation, on the other hand, compiles into 54 PTX instructions with 3 conditional branches and potential conflict in the constant cache. Our implementation gives a maximum relative error of 1.10×10^{-10} over the range [0,1], or a binary accuracy of 44.1 bits.

The double-precision power function used in evaluating the weight function $w_R(\mathbf{r}_{ij})$ is one of the most time-consuming functions in the CUDA math library. In order to conform to the IEEE floating point standard, the CUDA native implementation has to deal with the full range of inputs as well as possible exceptions. As a result, the native power function compiles into 258 PTX instructions with 18 conditional branches, 9 uniform branches, 3 reciprocal and 1 division with an average execution latency of 1982 clock cycles. Again in our implementation we seek to exploit the knowledge over the input range for optimization: the exception conditions that the base or the exponent being 0 can be precluded by the cutoff

testing prior to the function call; it is also unlikely that the base or the exponent would be NaN or Inf unless there are serious problems in the underlying physics of the model. Our custom double precision power function is based on the base-2 logarithm and exponential using the identity $a^b = 2^{b\log_2 a}$ as shown in Algorithm 5. The logarithm part is similar to that used in our Box-Muller implementation except that the order of the Chebyshev polynomial is increased to 14. The exponential part is approximated by a 11th order Chebyshev polynomial. Both routines give an maximum error of less than 1 unit in the last place (ULP). Special care has been taken to ensure the accuracy of the composite function when combining the results. The maximum error of our power function is 6 ULP given an input range of $a \in [10^{-102}, 10^{102}]$ and $b \in [0, 3]$, or 11 ULP given an input range of $a \in [10^{-51}, 10^{51}]$ and $b \in [0, 6]$. This should suffice for the purpose of DPD simulation, where typically $a \in [10^{-10}, 2]$ and $b \in [0.25, 3.0]$.

2.2.6 Communication

In our code, each processor maintains a *sendlist* for every neighboring processor recording the indices of local particles whose information is to be sent during the communication. The determination of the border particles can be implemented relatively straightforward on CPUs, whereas on GPUs it requires the cooperation of multiple kernels. Nevertheless, we still implemented it on GPU because the border determination must be done after particle reordering where particle indices are reassigned. A total of seven kernels are used as outlined in Algorithm 6 for border determination. Communication of ghost particle position and velocity between two border-determination steps involves only the last three steps of the algorithm. A CUDA-aware MPI implementation, *i.e.* the Cray MPI, was

Algorithm 5 Branchless power function with error $\leq$ 11 ULP. The `__hiloint2double`, `__double2hiloint` and `__fma` functions are double-precision floating point intrinsics from the CUDA math library.

```
// fast computing for integer power of 2
double power2( int n ) { return __hiloint2double( (1023+n)<<20, 0 ); }

// 11th order 2^x = P(x)
// error < 1 ULP for x in [0,1.0]
double exp2_frac( double x ) {...}

// fast converging 14-order log2(x) = z*P(z^2)
// error < 1ULP for x in [1,2]
double log2_frac( double x ) {...}

double fastpow(double a, double b)
{
  int hi, lo;
  __double2hiloint( a, hi, lo );
  // extract exponent
  double I  = ( hi >> 20 ) - 1023;
  // reset exponent, do fractional log
  hi = ( hi & 0X000FFFFF ) | 0X3FF00000;
  double F  = log2_frac( __hiloint2double(hi,lo) );
  // multiply by exponent, separate to 2 parts to mantissa precision loss
  double II = floor( b * ( I + F ) );
  return power2( II ) * exp2_frac( __fma( b, F, __fma( b, I, -II ) ) );
}
```

tested. However it was found that manually pipelined execution and data transfer between the host and device yields better performance. Peer-to-peer memory transfer is not utilized because each MPI rank handles only a single GPU.

Algorithm 6 Exchange of border information.

```
1: function BorderDetermination
2:     flag particles to be sent using a 1-bit boolean
3:     multi-block segmented prefix sum: down-sweep phase
4:     multi-block segmented prefix sum: blocks
5:     multi-block segmented prefix sum: up-sweep phase
6:     send list generation: parallel compaction using scan result
7:     pack information of particles in send list
8:     MPI communication...
9:     unpack list of incoming ghost particles
10: end function
```

2.2.7 Code Verification & Benchmark

Method

Benchmarks were carried out on the TITAN supercomputer located at the Oak Ridge National Laboratory [1]. The TITAN is a Cray XK-7 system with 18688 nodes. Each node is equipped with one AMD Opteron 6274 CPU and one NVIDIA Kepler K20X GPU. Each Opteron 6274 CPU has 16 integer cores and 8 floating point cores running at 2.2 GHz along with 16 MB of L3 cache. The Kepler K20X, with 2688 CUDA cores, has a theoretical double-precision performance of 1.31 TFLOPS and a peak memory bandwidth of 250 GB/s. Only the main loop is timed in the benchmark since initialization and clean up overheads are neglectable for long-time simulations. The CPU code for reference is a vanilla version of LAMMPS with our own DPD pair style implementing the aforementioned functional form. The polar form of the Box-Muller method is used for generating Gaussian random

numbers on the CPU because it has much better performance than the basic form when used on the CPU. All CPU code is compiled with the Intel compiler using -O3 optimization, while GPU code is compiled by the NVIDIA NVCC compiler.

Flow Simulation

To verify the correctness of our code, we aimed at reproducing the fluid viscosity measured from the double Poiseuille flow used by Backer *et al* [8]. The parameters are chosen as $\sigma = 4.5$, $\rho = 6.0$, $k_BT = 0.5$ and $\delta t = 0.001$. The conservative force is left out in the same way as in the original paper. A total of 4608 particles were simulated in a simulation box of dimension $12r_c \times 8r_c \times 8r_c$ in the x, y and z directions, respectively. A body force $g_z = 0.055$ was applied to drive the flow in the periodical space. The viscosity of the fluid was evaluated from 10 parallel simulations with different initial configurations and random seeds. We obtained a viscosity of 2.089 ± 0.009, which is in excellent agreement with the published value of 2.09 ± 0.02.

Another test case of *transient* double Poiseuille flow was employed to examine the accuracy of our code over long-time integration. The parameters are chosen as $\rho = 5.0$, $a_{ij} = 15.0$, $\sigma = 3.0$, $k_BT = 1.0$, $r_c = 1.0$ and $\delta t = 0.01$. The system consists of 262,144 particles in a simulation box of $59.4123 \times 7.42654 \times 118.825$ in the x, y and z directions. Velocity profiles along the z direction were collected for time T = 100, 200, 500, 1000, 2000 and 10000. For each time point the velocity profile is sampled from a time window of $[T - 0.5, T + 0.5]$. The simulation was repeated 10 times with different initial configurations and random seeds. The simulation gives good matching with the analytic solution given in Eq. 2.23 by Sigalotti *et al.*

[134].

$$u(z,t) = \frac{Fd^2}{8\upsilon}\left(1 - \left(\frac{2z}{d}\right)^2\right) - \sum_{n=0}^{\infty} \frac{4(-1)^n Fd^2}{\upsilon\pi^3(2n+1)^3} \cdot \cos\left[\frac{(2n+1)\,\pi z}{d}\right] \cdot \exp\left[-\frac{(2n+1)^2\pi^2\upsilon t}{d^2}\right] \tag{2.23}$$

A comparison of analytical and DPD simulation results is shown in Figure 2.10.

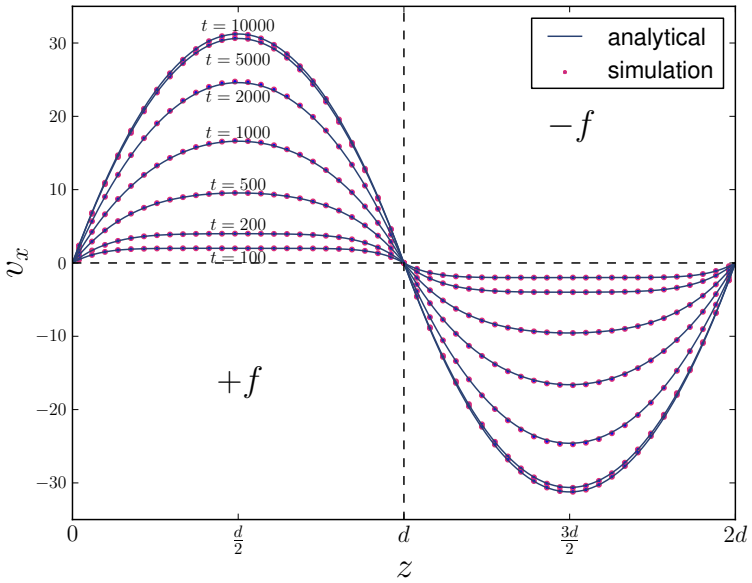


Figure 2.10: Time evolution of the velocity profile for the double Poiseuille flow.

Single Node Speedup

Three different particle densities, 3, 6, and 50, and two cutoff distances, 1.0 and 1.5, were used to fully characterize the performance of our code. Note that a particle density of 50 is only observed for atomistic systems, yet such comparison allows us to estimate the performance of our algorithm when applied to an atomistic system. A time step of 0.005, along with two neighbor list updating frequencies, 3 and 5, was used. Systems whose size ranged from 8192 to 2 million particles were simulated. To enable a fair competition between the GPU and CPU implementations, the speedup is measured for one Kepler GPU over all available CPU

cores on a single node. Speedup is measured as the ratio of wall time elapsed for carrying out a specific simulation. As shown in Figure 2.11, the GPU code generally performs better for larger systems, with the best speedup of over 30 measured with 2 million particles. This is not surprising since overhead, especially kernel launching latency, is independent of the system size. Another interesting fact is that the speedup increases rapidly once the system size grows over 200,000. We speculate that the CPU may be experiencing cache depletion above this point.

We also measured the speedup contributed by each of the optimizing techniques mentioned in Section 2.2.2 for systems containing $64k$, $128k$, $512k$, and $2m$ particles. The measurement was carried out by first substituting all aforementioned optimized kernels with their conventional counterparts, and then re-enabling them in the order as presented in the previous section. A more transferrable metric, million particles·steps per second, or *MPS/second*, is used because it faciliates performance characterization and comparison across systems of different particle numbers. The result as presented in Figure 2.12 shows that performance enhancement ranging from 80% to 130% was achieved when combining all five techniques. It is also shown that larger systems tend to benefit more from the optimization.

Weak and Strong Scaling

The same set of test cases from the single node benchmark were used to establish the weak scaling and strong scaling benchmarks. For weak scaling, particles per node was kept at 1 million for $\rho = 3$ and 5, and was kept at 128k for $\rho = 50$. A nearly linear speedup is observed for up to 1024 nodes as demonstrated by Figure 2.13.

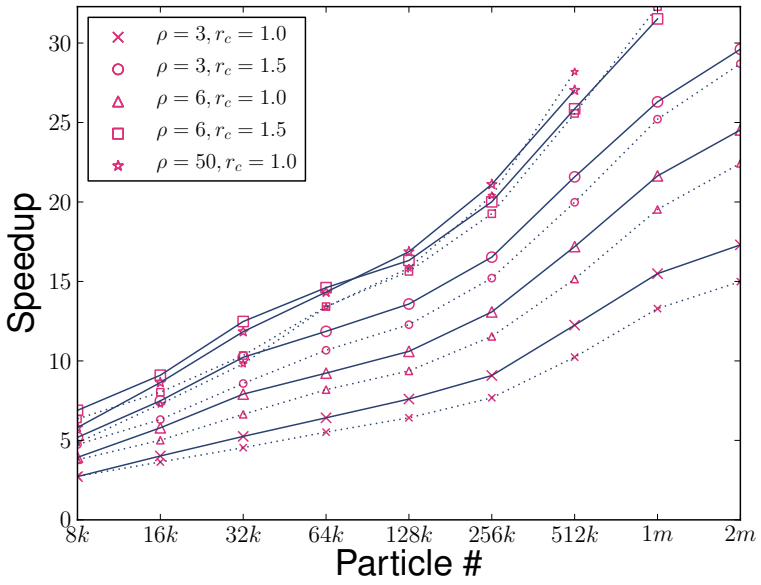


Figure 2.11: Speedup of our code on a single Kepler K20x GPU over 16 AMD Opteron 6274 CPU cores for various system size, cutoff distance and particle number density. Results obtained by using two neighbor list updating frequencies, *i.e.* 3 and 5, are represented by dotted and solid lines, respectively.

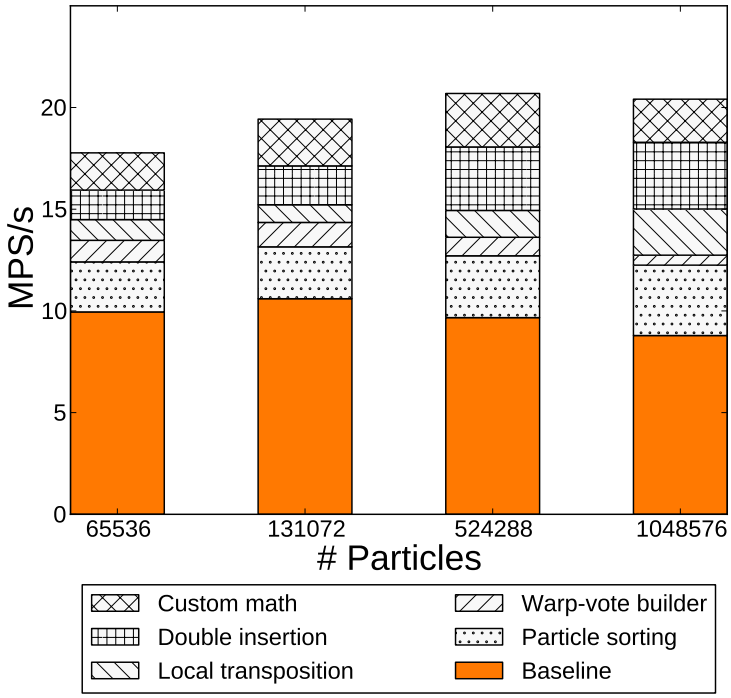


Figure 2.12: Speedup contributed by each of the optimizing techniques mentioned in Section 2.2.2 for systems containing 64k, 128k, 512k, and 2m particles.

Systems containing 2 million particles at different particle densities were simulated for the strong scaling benchmark. The result is shown in Figure 2.14. Parallel efficiency is satisfactory for up to 64 nodes, but then it deteriorates gradually. The efficiency declination is attributed primarily to the overhead of copying data between the main memory and the GPUs when preparing the MPI communication. An attempt to optimize such latency using the newly-introduced GPUDirect RDMA pathway combined with the CUDA-aware Cray MPI was made. However, no speedup was observed because RDMA is only efficient for smaller message sizes due to its limited bandwidth despite lower communication latency. The strong scaling limit for our code is around 512 to 1024 nodes, which corresponds to the point when communication overhead dominates the entire simulation. Increasing node count beyond this point results in worse performance.

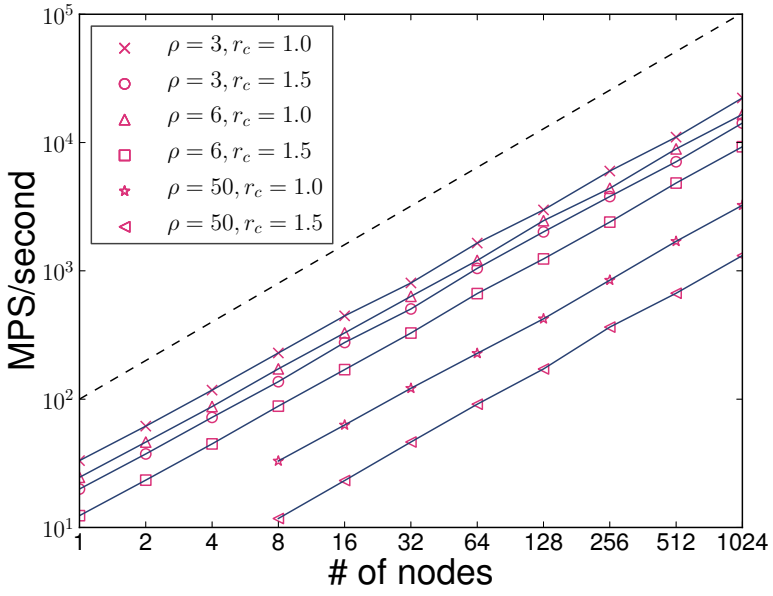


Figure 2.13: Weak scaling performance of our code for various cutoff distance and particle number density. The system size is kept at 1 million particles per node.

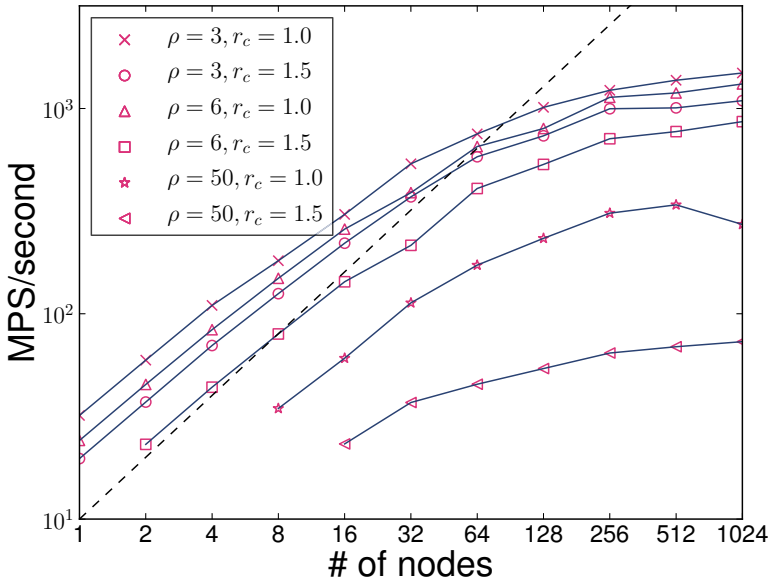


Figure 2.14: Strong scaling performance of our code for various cutoff distance and particle number density. The system size is fixed at 2 million particles regardless of the node number.

Amphiphilic polymer self-assembly

A simulation of spontaneous vesicle formation using 134,217,728 particles is performed to further demonstrate the capability of our new DPD program to study complex fluids. An aqueous surfactant solution system consisting of water and an amphiphilic triblock copolymer were modeled. The particles representing water, hydrophilic block and hydrophobic block are denoted as S, A, B, respectively. The triblock copolymer has a chain configuration of BBBAABBB. The system was initialized to be a random distribution of water particles and polymer molecules. Bounce-forward boundary condition was applied instead of the periodic boundary condition. Beads within a molecule are tied together using a harmonic potential $F = K(r - r_0)$, where the coefficients were chosen as $r_0 = 0.38$ and $K = 80$. The DPD parameters were chosen as $\rho = 5.0$, $\sigma = 3.0$, $\gamma = 4.5$, $k_BT = 1.0$, and $\delta t = 0.01$. The pairwise repulsive force magnitudes a_{ij} were adapted from Ref. [75] and scaled

inversely proportionally with regard to the particle density:

$$\begin{pmatrix} & A & B & S \\ A & 15 & 120 & 15 \\ B & 120 & 15 & 120 \\ S & 15 & 120 & 15 \end{pmatrix}$$

The simulation took 54 hours to complete on 1024 nodes of TITAN. A total of 12,400,000 time steps, or a time span of 124,000 in reduced DPD uints, were sampled. The time evolution of the system as illustrated in Figure 2.15 proves that a mixture of membranes, simple vesicles and multi-compartment vesicles with a remarkable morphological variation were grown from the initially random distribution. The structure of the vesicles as visualized by a transmission electron microscopy-style rendering technique exhibits many similarities to those of the onion-like multilamellar multicompartment vesicles observed by Bangham *et al* [9] as shown in Figure 2.16. Figure 2.17 demonstrates that the single-compartment vesicle in Figure 2.16a is actually formed through the fusion and spontaneous wrapping of membrane-like structures.

2.2.8 Sectional Conclusion

We presented a complete design of the DPD simulation code on the CUDA GPUs. The program is derived from LAMMPS and it achieves substantial speedup over the original CPU version. Novel algorithms are used in all parts of the program including force evaluation, data layout and numerics. An atomics-free, deterministic algorithm for constructing the neighbor list is proposed. We have

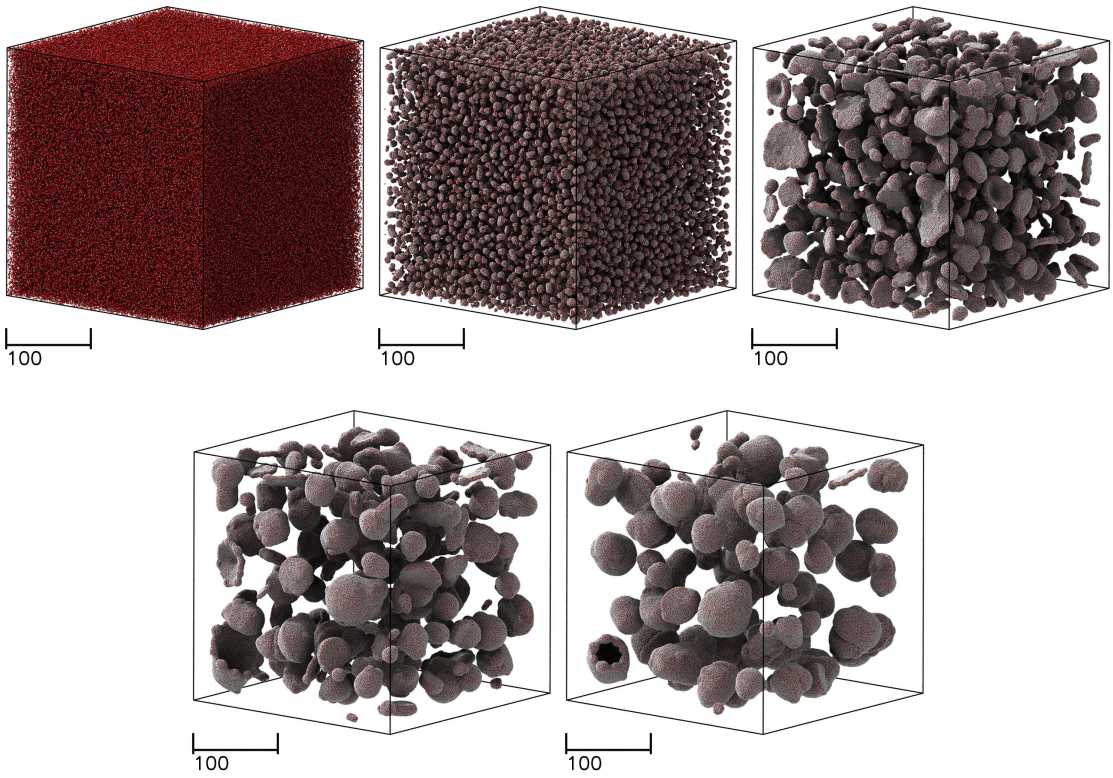


Figure 2.15: A variety of vesicles and membrane-like structures were grown from the 128 million particle simulation. The BBBAABBB triblock copolymer was initially randomly distributed in a cubic system of size $299.4 \times 299.4 \times 299.4$. The concentration of the surfactant is 10%. Scale bars are in reduced DPD units.

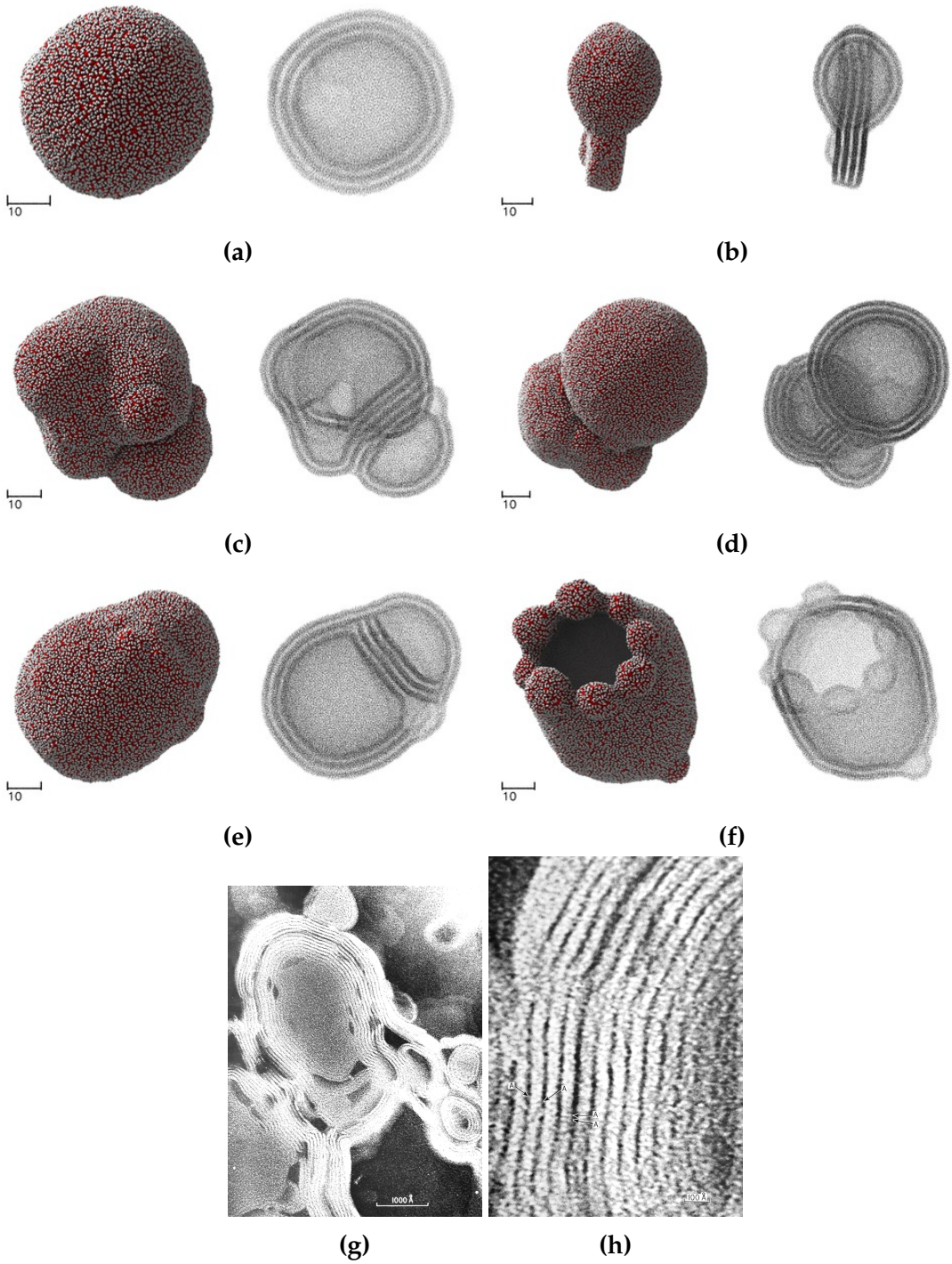


Figure 2.16: The multi-walled vesicle shown in a-f exhibit notable structural similarity to that observed by Bandham *et al* [9] in g and h (reprinted with permission from Elsevier). The hydrophilic and hydrophobic particles are rendered in white and red, respectively, in the colored images; and in white and black, respectively, in the transmission microscopy-style images. Scale bars are in reduced DPD units.

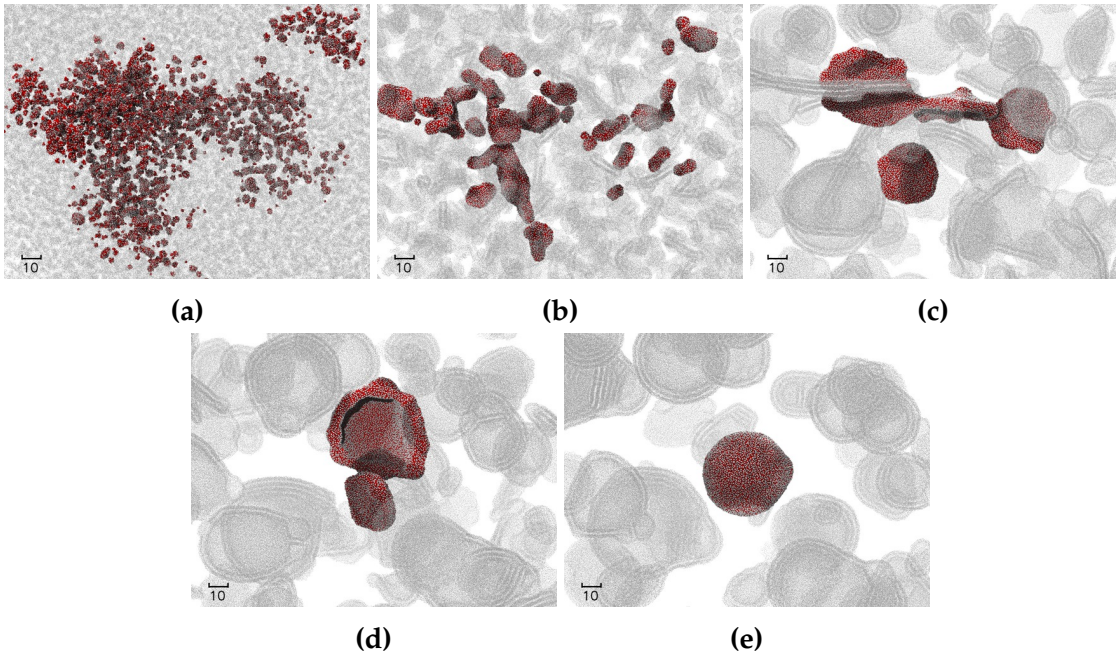


Figure 2.17: The vesicle shown in Figure 2.16 is formed through the fusion and spontaneous wrapping of membrane-like structures. Polymer molecules that eventually became part of the vesicle are highlighted, while irrelevant ones are drawn as shadows. Scale bars are in reduced DPD units.

also been able to address locality and neighbor list building in a tightly-coupled framework. Numerical optimization targeting random number generation and functional form evaluation were proposed to fully utilize the computing power of the GPU. The code executes on large-scale supercomputers with near-optimal weak scaling efficiency, while strong-scaling efficiency is compromised by the GPU-CPU data transfer overhead. A large-scale simulation of spontaneous vesicle formation demonstrated the practicality of our code. Future work includes optimizing communication latency through overlapping computation with communication as well as load balancing for non-trivial domain geometry. The code will be contributed as a LAMMPS user module that is freely available to the general public.

2.3 OpenRBC: Whole Cell Simulators at Protein Resolution

OpenRBC[1] is a coarse-grained molecular dynamics code capable of performing an unprecedented *in silico* experiment — simulating an entire mammal red blood cell lipid bilayer and cytoskeleton as modeled by multiple millions of mesoscopic particles — using a single shared memory commodity workstation. To achieve this, we invented an adaptive spatial-searching algorithm to accelerate the computation of short-range pairwise interactions in an extremely sparse 3D space. The algorithm is based on a Voronoi partitioning of the point cloud of coarse-grained particles, and is continuously updated over the course of the simulation. The algorithm enables the construction of the key spatial searching data structure in our code, *i.e.* a lattice-free cell list, with a time and space cost linearly proportional to the number of particles in the system. The position and shape of the cells also adapt automatically to the local density and curvature. The code implements OpenMP parallelization and scales to hundreds of hardware threads. It outperforms a legacy simulator by almost an order of magnitude in time-to-solution and more than 40 times in problem size, thus providing a new platform for probing the biomechanics of red blood cells.

2.3.1 Background

The red blood cell (RBC) is one of the simplest, yet most important cells in the circulatory system due to their indispensable role in oxygen transport. An average RBC assumes a biconcave shape with a diameter of 8 μm and a thickness of

[1]The source code is available at https://github.com/yhtang/OpenRBC .

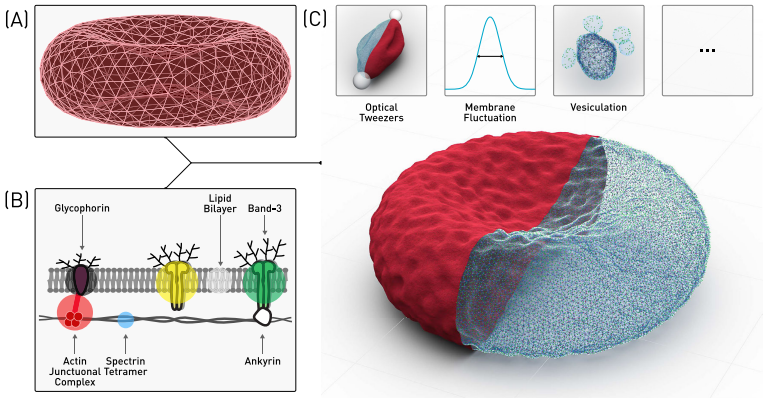


Figure 2.18: (A) A canonical hexagonal triangular mesh of a biconcave surface representing the cytoskeleton network is used together with **(B)** the two-component CGMD RBC membrane model to reconstruct **(C)** a full-scale virtual RBC which allows for a wide range of computational experiments.

2 μm. Without any intracellular organelles, it is supported by a cytoskeleton of a triangular spectrin network anchored by junctions on the inner side of the membrane. Therefore, the mechanical properties of an RBC can be strongly influenced by molecular level structural details that alter the cytoskeleton and lipid bilayer properties.

Both continuum models [36, 41, 117, 59] and particle-based models [40, 130, 146, 148] have been developed with the aim to help uncover the correlation between RBC membrane structure and property. Continuum models are computationally efficient, but require *a priori* knowledge of cellular mechanical properties such as bending and shear modulus. Particle models are useful for extracting RBC properties from low-level descriptions of the membrane structure and defects. However, it is computational demanding, if not prohibitive, to simulate the large number of particles required for modeling the membrane of an entire RBC. To the best of our knowledge, a bottom-up simulation of the RBC membrane at the cellular scale using particle methods remains absent.

Recently, a two-component coarse-grained molecular dynamics (CGMD) RBC membrane model which explicitly accounts for both the cytoskeleton and the lipid bilayer was proposed [82]. The model could potentially be used for particle-based whole-cell RBC modeling because its coarse-grained nature can drastically

reduce computational workload while still preserving necessary details from the molecular level. However, due to the orders of magnitude difference in the length scale between a cell and a single protein, a total of 4 million particles are still needed to represent an entire RBC. In addition, the implicit treatment of the plasma in this model eliminates the overhead for tracking the solvent particles, but also exposes a notable spatial density heterogeneity because all CG particles are exclusively located on the surface of a biconcave shell. The space inside and outside of the RBC membrane remains empty. This density imbalance imposes a serious challenge on the efficient evaluation of the pairwise force using conventional algorithms and data structures, such as the cell list and the Verlet list, which typically assumes a uniform spatial density and a bounded rectilinear simulation box.

In this paper we present OpenRBC, a new software tailored for simulations of an entire RBC using the two-component CGMD model on multicore CPUs. As illustrated in Figure 2.18, the simulator can take as input a triangular mesh of the cytoskeleton of a RBC and reconstruct a CGMD model at protein resolution with explicit representations of both the cytoskeleton and the lipid bilayer. This type of whole cell simulation of RBCs can thus realize an array of *in silico* measurements and explorations of:

- RBC shear and bending modulus,
- membrane loss through vesiculation in spherocytosis and elliptocytosis [83],
- anomalous diffusion of membrane proteins [84],
- interaction between sickle hemoglobin fibers and RBC membrane in sickle cell disease [79, 85],
- uncoupling between the lipid bilayer and cytoskeleton [113],
- adenosine triphosphate (ATP) release due to deformation [138],
- nitric oxide (NO) modulated mechanical property change [160],

· cellular uptake of elastic nanoparticles [170].

2.3.2 Software Overview

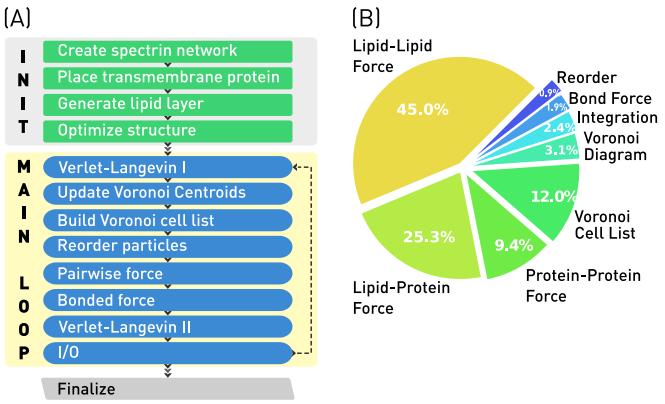


Figure 2.19: The A) flow chart and B) typical wall time distribution of OpenRBC.

OpenRBC is written in C++ using features from the C++11 standard. To maximize portability and allow easy integration into other software systems [145], the project is organized as a header-only library with no external dependencies. The software implements SIMD vectorization [2] and OpenMP shared memory parallelization, and was specifically optimized toward making efficient use of large numbers of simultaneous hardware threads.

As shown in Figure 2.19A, the main body of the simulator is a time stepping loop, where the force and torque acting on each particle is solved for and used for the iterative updating of the position and orientation according to a Newtonian equation of motion. The time distribution of each task in a typical simulation is given in Figure 2.19B. The majority of time is spent in force evaluation, which is compute-bound. This makes the code highly efficient in utilizing the high thread count of modern CPUs with the shared memory programming paradigm.

To maximize customizability and reusability, we strive to decompose the source code into functionally independent modules. The structure of OpenRBC in terms of the source file organization is given in Table 2.4.

Table 2.4: List of source files relevant to customization

File	Functionality
openrbc.cpp	Main program loop
orbc-util.cpp	Utility program for file format conversion *etc.*
config_static.h	Compile-time options mostly relevant to performance optimization
forcefield_*.h	CGMD particles definition and interaction parameters
container.h	CGMD particle container
init_*.h	Initial structure generation algorithm
compute_*.h	Force evaluation driver (outer loop among adaptive cells)
pairwise_kernel_*.h	Pairwise force evaluation (inner loop between particles)
integrate_*.h	Time integrators, substeps, etc.
kdtree.h	k-d tree algorithm
math_vector_*.h	Vector operation support, SIMD wrapper
reorder_*.h	Sorting algorithm to improve data locality
runtime_parameter.h	Options controlling program behavior at run time
voronoi.h	Voronoi cell list
trajectory.h, topology.h	File I/O
util_*.h, service.h, timer.h	Miscellaneous utilities

2.3.3 Initial structure generation

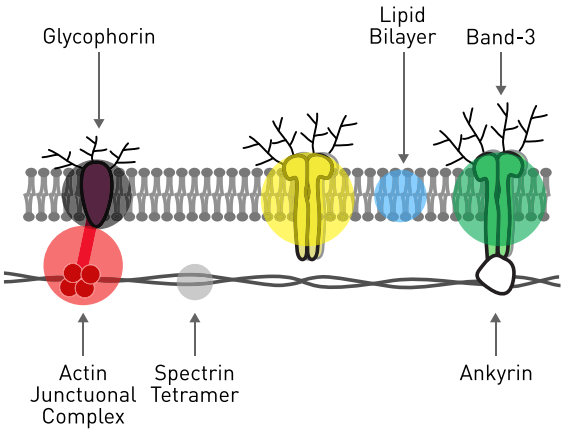


Figure 2.20: A schematic illustration of the CGMD model. Blue: lipid; red: actin junctional complex, silver: spectrin, black: glycophorin, yellow: mobile band-3, green: immobile band-3.

We first briefly introduce the two-component CGMD RBC membrane model. For more detailed description see Reference [81, 82]. As illustrated in Figure 2.20, the model describes the RBC as a two-component system, comprised of a cytoskeleton and a lipid bilayer. The cytoskeleton consists of spectrin filaments connected at actin junctional complexes forming a hexagonal network. The actin junctional complexes, as represented by the red particles, have a diameter of approximately 15 nm and are connected to the lipid bilayer via glycophorin. Spectrin is a protein tetramer formed by head-to-head association of two identical heterodimers. Each spectrin filament is represented by 19 spectrin particles connected by unbreakable springs. Spectrin chains are linked to band-3 particles via a spring potential. The two ends of the spectrin chains are also connected to the actin junctional complexes via the spring potential. Spectrin particles that are not connected by the spring potential interact with each other via a Lennard-Jones potential.

The CG particles, which form the lipid bilayer and transmembrane proteins, carry both translational and rotational degrees of freedom (x_i, n_i), where x_i and n_i are the position and the orientation (direction vector) of particle i, respectively. The rotational degrees of freedom obey the normality condition $|n_i| = 1$.

The lipid particles interact with each other through a pairwise additive poten-

tial:

$$u_{ij}(n_i, n_j, x_{ij}) = u_R(r_{ij}) + A(\alpha, a(n_i, n_j, x_{ij}))u_A(r_{ij}) \tag{2.24}$$

$$u_R(r) = \begin{cases} \epsilon(\frac{r_c - r}{r_c - r_{eq}})^8, & r < r_c \\ 0, & \text{otherwise} \end{cases} \tag{2.25}$$

$$u_A(r) = \begin{cases} -2\epsilon(\frac{r_c - r}{r_c - r_{eq}})^4, & r < r_c \\ 0, & \text{otherwise} \end{cases} \tag{2.26}$$

$$A(\alpha, a(n_i, n_j, x_{ij})) = 1 + \alpha(a(n_i, n_j, \hat{x}_{ij}) - 1) \tag{2.27}$$

$$a(n_i, n_j, \hat{x}_{ij}) = (n_i \times \hat{x}_{ij}) \cdot (n_j \times \hat{x}_{ij}) = n_i \cdot n_j - (n_i \cdot \hat{x}_{ij})(n_j \cdot \hat{x}_{ij}) \tag{2.28}$$

where $x_{ij} = x_j - x_i$ is the distance vector between particles i and j, $r_{ij} = \|x_{ij}\|$ is the distance between i and j, $u_R(r_{ij})$ and $u_A(r_{ij})$ are the repulsive and attractive components of the pair potential, respectively. α is a tunable linear amplification factor. The function $A(\alpha, a(n_i, n_j, x_{ij})) = 1+\alpha(a(n_i, n_j, x_{ij})-1)$ tunes the energy well of the potential, through which the fluid-like behavior of the membrane is regulated.

The translational motion of the particles is governed by the Newtonian equation of motion (EOM) of force, velocity, and position

$$\ddot{x} = \dot{v} = -\nabla U/m \tag{2.29}$$

while the rotational motion for the CG particles forming the lipid bilayer and proteins in the lipid bilayer is governed by

$$\tilde{m}_i \ddot{\mathbf{n}}_i = -\frac{\partial(\sum_{j=1}^{N} u_{\mathrm{mem},ij})}{\partial \mathbf{n}_i} + (\frac{\partial(\sum_{j=1}^{N} u_{\mathrm{mem},ij})}{\partial \mathbf{n}_i} \cdot \mathbf{n}_i)\mathbf{n}_i - \tilde{m}_i(\dot{\mathbf{n}}_i \cdot \dot{\mathbf{n}}_i)\mathbf{n}_i \tag{2.30}$$

where m is the mass of each particle, $\tilde{m}$ is a pseudo-mass with dimension of energy $\cdot$ time^2, and the right-hand side of Eq. 2.30 obeys the normality constraint $|n_i| = 1$. The Verlet algorithm combined with a Langevin thermostat is used to update the particle's position and orientation according to the EOMs.

As shown in Figure 2.18, a two-component CGMD RBC system can be generated from a triangular mesh which resembles the biconcave shape of a RBC at equilibrium. Note that the geometry may be alternatively sourced from experimental data using techniques such as optical image reconstruction because the algorithm itself is general enough to adapt to an arbitrary triangular mesh. This feature can be useful for simulating RBCs with morphological anomalies. Actin and glycophorin protein particles are placed on the vertices of the mesh, while spectrin and immobile band-3 particles are generated along the edges. The band-3–spectrin connections and actin–spectrin connections can be modified to simulate RBCs with structural defects. Lipid and mobile band-3 particles are randomly placed on each triangular face by uniformly sampling each triangle defined by the three vertices [109]. A minimum inter-particle distance is enforced to prevent clutter between protein and lipid particles. The system is then optimized using a velocity quenching algorithm to remove collision between the particles.

2.3.4 Spatial Searching Algorithm

As typical in molecular dynamics simulations, pairwise force evaluation accounts for more than 70% of the computation time in OpenRBC as well as other molecular dynamics software[144, 126]. A brute-force all-versus-all computation over a system of N particles incurs $O(N^2)$ work and is prohibitively expensive even for just a few thousand particles. The approach to accelerate neighbor searching in the legacy solver, as well as many in many other existing MD software, is to use a Verlet list, which is essentially a table storing the indices of particles within a given distance $r_v \geq r_c$ for each particle in the system. The Verlet list can be constructed efficiently with the help of a cell list, which makes use of a uniform lattice to partition the system into many nearly-cubic cells and stores the indices of the particles within each cell. The cells are numbered consecutively along the axes, allowing the index of cell that each particle belongs to be determined by simply dividing the particle's coordinate by the length of the cells and then flooring to the nearest integer.

Given a system of N particles occupying a volume of $L_x \times L_y \times L_z$, a Verlet list can be constructed from a cell list using $O(N)$ time and storage by looping over each particle, first finding the cell that the particle belongs to, and then comparing against other particles in this cell as well as particles in all 26 immediate neighboring cells. The cell list itself takes $O(L_x L_y L_z + N)$ storage and $O(N)$ time using the algorithm as shown in Algorithm 7.

Alternatively, it is possible to directly use the cell list for pairwise force computation by looping over all pairs of neighboring cells. At first sight, the approach appears suboptimal because on average only 16% of the interactions will be within

Algorithm 7 The conventional rectilinear cell list algorithm.

```
Method RectilinearCelllist( Np: integer,
                            ncell: integer[3],
                            cell_size: integer[3],
                            coord: real[N][3] )
  Ncell     = ncell[0] * ncell[1] * ncell[2]
  bin_size  = zeros[Ncell] # O(L^3) spacce
  local_seq = integer[N]   # O(N)   space
  cid       = integer[N]   # O(N)   space

  # Get cell id and cell-local index for each particle
  # Count cell size
  for i = 0:N
    # operator ./: element-wise divison
    cell_xyz = floor( coord[i] ./ cell_size )
    cid[i] = cell_xyz[0] + cell_xyz[1] * ncell[0] + cell_xyz[2] * ncell[1] *
    ncell[2]
    local_seq[i] = bin_size[ cid[i] ]
    ++bin_size[ cid[i] ]

  # O(N) prefix sum for the starting index of each cell
  bin_start = zeros[Ncell] # O(L^3) space
  for i = 1:Ncell
    bin_start[i] = bin_start[i-1] + bin_size[i-1]

  # Scatter particle indices into corresponding cell
  cell_list = integer[N]  # O(N) space
  for i = 0:N
    cell_list[ bin_start[ cid[i] ] + local_seq[i] ] = i

  return cell_list, bin_start
```

r_c, comparing to 46% when using the Verlet list with $r_v = 1.3r_c$. However, as has been demonstrated previously [126], this saves the work and storage to construct the Verlet list, as well as the memory bandwidth to load the huge list during force evaluation. On modern computer systems that are typically rich in computing power but poor in memory bandwidth, the cell list approach usually results in better performance. Nevertheless, the cell list algorithm still suffers from one inefficiency that is particularly pronounced when the CG particles are concentrated on the surface of a biconcave geometry: the $O(L_xL_yL_z)$ storage term, i.e. number of particles in each cell, dominates when handling systems of large spatial density heterogeneity as illustrated in Figure 2.21. With a cutoff distance of 5 nm, to simulate a system of $8\mu m \times 8\mu m \times 8\mu m$, the $O(L_xL_yL_z)$ term translates into a storage requirement for 3.8×10^9 cells which occupies ~16 GB memory. Streaming the list into the processor every time step can waste a large fraction of the CPU-memory bandwidth.

To efficiently simulate the reconstructed RBC model, we invented a lattice-free spatial partitioning algorithm that is inspired by the concept of Voronoi diagram. The algorithm, at the high level, can be described as

```
1. Group particles into a number of 'adaptive' clusters
2. Compute interactions between neighboring clusters
3. update cluster composition after particle movement
4. repeat from step 2
```

As illustrated in Figure 2.21, the algorithm adaptively partitions a particle system into a number of Voronoi cells that are approximately equally populated. In contrast, a lattice-based cell list leaves many cells vacant due to the density heterogeneity. Thus, the algorithm can provide very good performance in partitioning the system, maintaining data locality and searching for pairwise neighbors in a

sparse 3D space. It is implemented in our software using a *k*-means clustering algorithm, which is, in turn, enabled by a highly optimized implementation of the *k*-d tree searching algorithm, as explained below.

A Voronoi tessellation [33] is a partitioning of a n-dimensional space into regions based on distance to a set of points called the centroids. Each point in the space is attributed to the closest centroid (usually in the L2 norm sense). An example of a Voronoi Diagram generated by 12 centroids on a 2D rectangle is given in Figure 2.22A.

The *k*-means clustering [58] is a method of data partitioning that aims to divide a given set of n vectors into k clusters in which each vector belongs to the cluster whose center is closest to it. The result is a partition of the vector space into a Voronoi tessellation generated by the cluster centers as shown in Figure 2.22B. Searching for the optimal clustering which minimizes the within-cluster sum of square distance is NP-hard, but efficient iterative heuristics based on *e.g.* the expectation-maximization algorithm [32] can be used to quickly find a local minimum.

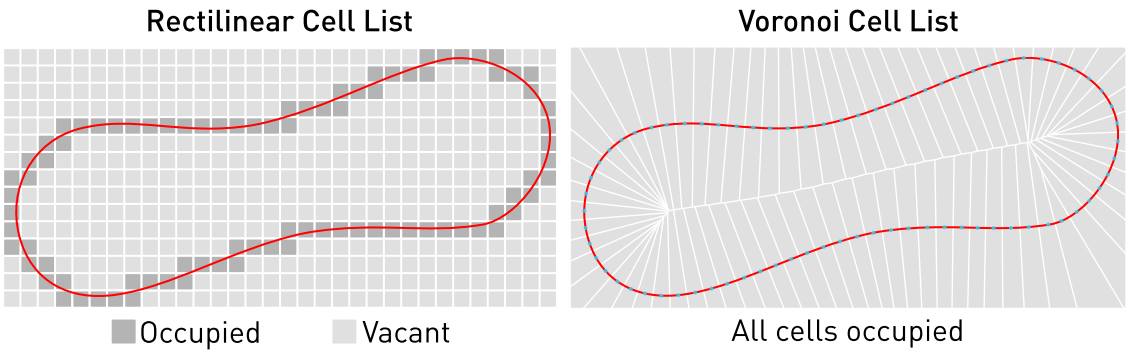


Figure 2.21: Left: Only cells in dark gray are populated by CG particles in a cell list on a rectilinear lattice. This results in a waste of storage and memory bandwidth. **Right:** All cells are evenly populated by CG particles in a cell list based on the Voronoi diagram generated from centroids located on the RBC membrane.

A *k*-d tree is a spatial partitioning data structure for organizing points in a

k-dimensional space [14]. It is essentially a binary tree that recursively bisects the points owned by each node into two disjoint sets as separated by an axial-parallel hyperplane. It can be used for the efficient searching of the nearest neighbors of a given query point in $O(\log N)$ time, where N is the total number of points, by pruning out a large portion of the search space using cheap overlap checking between bounding boxes.

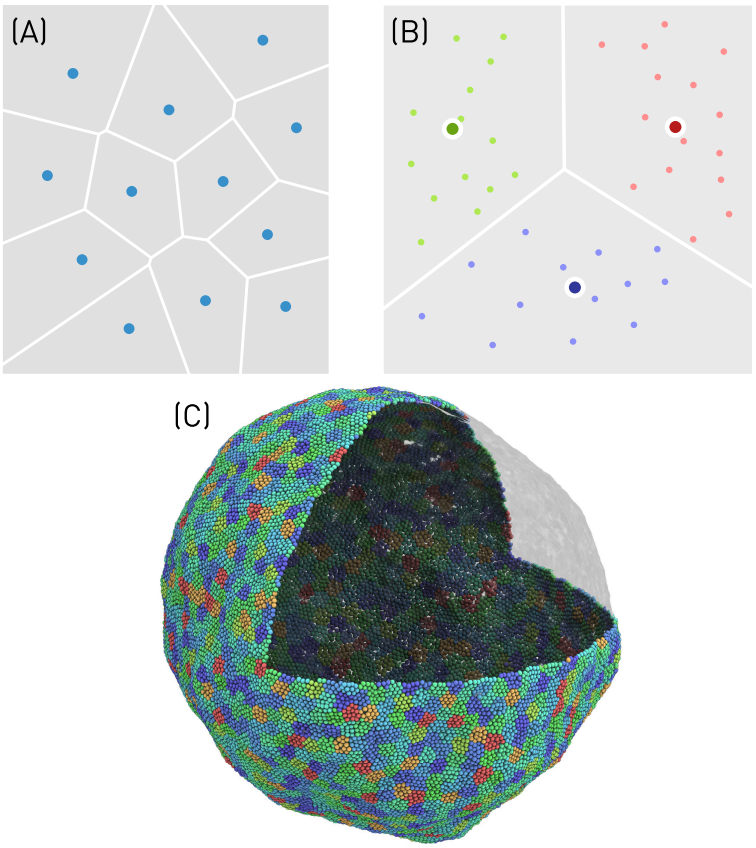


Figure 2.22: (A) A Voronoi partitioning of a square as generated by centroids marked by the blue dots. (B) A k-means (k=3) clustering of a number of points on a 2D plane. (C) A vesicle of 32,673 CG particles partitioned into 2000 Voronoi cells.

The k-means/Voronoi partitioning of a point cloud adapts automatically to the local density and curvature of the points. As such, we exploit this property to create a generalization of the cell list algorithm using the Voronoi diagram. The algorithm can be described as a two-step procedure: 1) clustering all the particles in the system using k-means, followed by an online Expectation-Maximization

algorithm that continuously updates the system's Voronoi cells centroid location and particle ownership; 2) sorting the centroids and particles with a two-level data reordering scheme , where we first order the Voronoi centroids along a space filling curve (a Morton curve, specifically) and then reorder the particles according to the Voronoi cell that they belong to. The pseudocode for the algorithm can be found in Algorithm 8. The reordering step in updating the Voronoi cells ensures that neighboring particles in the physical space are also statistically close to each other in the program memory space. This locality can speed up the k-d tree nearest-neighbor search by allowing us to use the closest centroid of the last particle as the initial guess for the next particle. This heuristic helps to further prune out most of the k-d tree search space and essentially reduces the complexity of a nearest-neighbor query from $O(\log N)$ to $O(1)$. In practice, this brings about 100 times acceleration when searching through 200,000 centroids. As shown by Figure 2.22C, the Voronoi cells generated from a k-means clustering of the CG particles are uniformly distributed on the surface of the lipid membrane. The Voronoi cells can be used directly for efficient pairwise force computation between lipids with a quad loop that ranges over all Voronoi cell v_i, all neighboring cells v_j of v_i, all particles in v_i, and all particles in v_j as shown in Algorithm 9. The cell-wise neighbors for each cell v is determined by the criterion:

$$d_{ij} \leq r_i + r_j + r_c$$

where d_{ij} is the centroid-to-centroid distance between the Voronoi cell v_i and v_j, r_i is $\max_{k \in v_i} \|x_k - c_i\|$, and r_j is $\max_{k \in v_j} \|x_k - c_j\|$.

Algorithm 8 The Voronoi cell list construction algorithm.

```
Class Voronoi_Celllist
  Ncell = SIMULATION CASE-SPECIFIC VALUE
  centroids = EMPTY
  bin_start = EMPTY
  cell_list = EMPTY
  tree      = KDTree()

  # find k centroids to minimize the within-cluster sum of squares
  Method KMeans( k: integer,
                 N: integer,
                 points: real[N][3] )
    ...


  Method UpdateCentroids( coord: real[Np][3] ):
    for i = 0:Ncell
      com = (0, 0, 0)
      n = bin_start[i+1] - bin_start[i]
      for j = 0:n
        com = com + coord[ bin_start[i] + j ]
      centroids[i] = com / n

  Method BuildCelllist( Np: integer, coord: real[Np][3] ):
    if centroids = EMPTY
      centroids = KMeans( Ncell, Np, coord )
    else
      Update_Centroids( coord )

    local_seq = integer[N]
    cid       = integer[N]
    # Calculate the Voronoi cell that particles fall in
    # Compute cell size and local indices for particles
    tree.rebuild( centroids )
    previous = ( id = 0, dist = Infinity )
    for i = 0:N
      nearest = tree.find_nearest( coord[i], previous )
      cid[i] = nearest.id
      local_seq[i] = bin_size[ nearest.id ]
      ++bin_size[ nearest.id ]
      previous = nearest

    # O(N) prefix sum for the starting index of each cell
    bin_start = zeros[Ncell]
    for i = 1:Ncell
      bin_start[i] = bin_start[i-1] + bin_size[i-1]

    # Scatter particle indices into corresponding cell
    cell_list = integer[N]
    for i = 0:N
      cell_list[ bin_start[cid[i]] + local_seq[i] ] = i

    destroy cid, local_seq, bin_size
    return cell_list, bin_start
```

Algorithm 9 Algorithm for pairwise force evaluation using the Voronoi cell list.

```
Method ComputePairwise( x : real[N][3], # coordinate
                        f : real[N][3], # force
                        o : real[N][3], # orientation
                        t : real[N][3], # torque
                        tree : KDTree,
                        voronoi_cell )
  for i = 0 : voronoi_cell.n_cells
    for each j in tree.find_around( i )
      for p1 in voronoi_cell[i]
        for p2 in voronoi_cell[j]
            if dist( x[p1], x[p2] ) < cutoff
              f, tau = pairwise_force( x[p1], x[p2],
                                       o[p1], o[p2] )
              f[p1] += f
              f[p2] -= f
              t[p1] -= tau
              t[p2] -= tau
```

2.3.5 Force Evaluation

Lipid particles accounts for 80% of the population in the whole-cell CGMD system. The Voronoi cells can be used directly for efficient pairwise force computation between lipids with a quad loop that ranges over all Voronoi cell v_i, all neighboring cells v_j of v_i, all particles in v_i, and all particles in v_j as shown in SI Algorithm 9. Since the cytoskeleton of a healthy RBC is always attached to the lipid bilayer, its protein particles are also distributed following the local curvature of the lipid particles. This means that we can reuse the Voronoi cells of the lipid particles, but with a wider searching cutoff, to compute both the lipid-protein and protein-protein pairwise interactions. For diseased RBCs with a fully or partially detached cytoskeleton, a separate set of Voronoi cells can be set up for the cytoskeleton proteins to compute the force. A list of bonds between proteins is maintained and used for computing the forces between proteins that are physically linked to each other.

A commonly used technique in serial programs to speed up the force compu-

tation is to take advantage of the Newton's 3rd law of action and reaction. Thus, the force between each pair of interacting particles is only computed once and added to both particles. However, this generates a race condition in a parallel context because two threads may end up simultaneously computing the force on a particle shared by two or more pairwise interactions.

Our solution takes advantage of the strong spatial locality of the particles as maintained by the two-level reordering algorithm, and decomposes the workload both spatially and linearly-in-memory into patches by splitting the linear range of cells indices among OpenMP threads. Each thread will be calculating the forces acting on the particles within its own patch. As shown in Figure 2.23, force accumulation without triggering racing condition can be realized by only exploiting the Newton's 3rd law on pairwise interactions where both Voronoi cells belong to a thread's own patch. Interactions involving a pair of particles from different patches are calculated twice (once for each particle) by each thread. The strong particle locality minimizes the shared contour length between two patches and hence also minimizes the amount of inter-patch interactions.

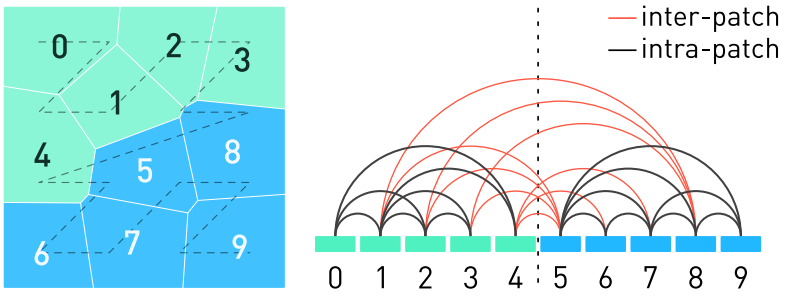


Figure 2.23: Thanks to the spatial locality ensured by reordering particles along the Morton curve (dashed line), we can simply divide the cells between two threads by their index into two patches each containing five consective Voronoi cells. The force between cells from the same patch is computed only once using Newton's 3rd law, while the force between cells from different patches is computed twice on each side.

2.3.6 Particle Storage

The layout of particle storage has an effect on the performance of our OpenRBC simulator. The data of the lipid and protein particles are stored separately in two containers for the following reasons. First, the protein-protein and protein-lipid potentials are much more complicated than the lipid-lipid potential. Thus, frequently choosing between the corresponding force kernels by particle type will incur a lot of branching instructions which may hurt the processor's front end performance. This can also be solved by working separately on the two classes of particles. Second, protein particles carry more information, *e.g.* type, tag, and bonds, than lipid particles. Hence a separation between the two classes of particles can save 2 arrays of size $O(N)$, which can be significant in terms of cache performance when millions of particles are present in the system.

2.3.7 Time Stepping

The Verlet integration algorithm coupled with the Langevin thermostat consists of embarrassingly parallel loops that iterates over particles to update their position, velocity, orientation and angular momentum. The implementation of the algorithm is divided into 2 stages, one before force evaluation and one after. Each of the stages consists of 3 passes that perform different tasks such as position updating, bounce back, orientation renormalization, force and torque reset, temperature calculation and temperature adjustment. Naively, each of the passes can be trivially parallelized with a single line of OpenMP `parallel for` directive. However, this will invoke a total of 12 parallel regions per time step to process both the lipid and protein container. Due to the low computation/transfer ratio of

the arithmetics within each pass, the entire workload is largely memory bound. As a consequence, the software initially displayed a performance degradation when going from 4 to 8 hardware threads on Power8 CPUs.

We implemented a fused version of the time stepping algorithm by extracting the core algorithm inside each pass as functors that resemble GPU kernels. A fusion can then be performed to the greatly simplified kernels. A C++11 variadic driver function is then used to start a single parallel region, within which an arbitrary number of containers can be processes by the kernels. This effectively reduced the total number of parallel regions encountered per time step to 2, and maximizes cache line reuse without compromising program readability. The new time stepping scheme can benefit from using all the hardware threads available on the physical cores.

2.3.8 Memory Access

Non-uniform memory access (NUMA) / Non-uniform cache access (NUCA) are the prevalent memory system design in current processor architectures, where the latency of memory access depends on the link topology between the memory location relative to the processing elements. To maximize local memory access, the code will pin OpenMP threads to hardware threads in a depth-first manner such that consecutive threads reside on the same physical core. The scheduling of most OpenMP loops that operate on array-like objects are done with a central work scheduler which controls the range processed by each thread. This ensures consistency in the memory access footprint to each array object across different functions. The two most frequently accessed and performance-critical data structures in the program are the particle container and the Voronoi cell list. Thanks to

the data locality as provided by the particle reordering algorithm, the partitions of the two structures can be aligned naturally with a simple linear split. Overall, in most of the parallel regions each thread will only need to work on its own partition without the need to touch data owned by other hardware cores. The non-local, pairwise nature of the particle interaction makes it inevitable for threads in OpenRBC to access part of the particle array which may be far away from its physical location. However, most of the non-local access is read-only and hence does not incur as much penalty as that in a read-write scenario.

2.3.9 Validation and Benchmark

In this section we present validation of our software by comparing simulation and experimental data. We also compare the program performance against that of a legacy CGMD RBC simulator used in Ref [82]. The legacy simulator, which performs reasonably well for a small number of particles in a periodic rectangular box, was written in C and parallelized with the message passing interface (MPI) using a rectilinear domain decomposition scheme and a distributed memory model. Three computer systems were used in the benchmark each equipped with a different mainstream CPU microarchitecture, *i.e.* the Intel Haswell, the AMD Piledriver, and the IBM Power8 [139]. The specification of the machines are given in Table 2.5.

To compare performance between OpenRBC and the legacy simulator, the membrane vesiculation process of a miniaturized RBC-like sphere with a surface area of of 2.8 μm^2 was simulated. The evolution of the dynamic process is visualized from the simulation trajectory and shown in Figure 2.24A. OpenRBC achieves almost an order of magnitude speedup over the legacy solver in this case on all

Table 2.5: A summary of capability and design highlights of OpenRBC and the specifications of the computer systems used in benchmark.

Capability & Design				Performance - time steps / day				
	OpenRBC	**Legacy**	**Improvement**	**Cores**	**Particles**	**OpenRBC**	**Legacy**	**Speedup**
Max system size (#particles)	$> 8 \times 10^6$	2×10^5	> 40 times	20	8.34×10^6	3.90×10^5	-	-
Line of code	4,677	7,424	37% less	4	1.88×10^5	3.86×10^6	0.42×10^6	9.2

CPU	**Architecture**	**Instruction Set**	**Freq. (GHz)**	**Physical Cores**	**Hardware Threads**	**Total threads**	**Last Level Cache (MB)**	**GFLOPS (SP)**	**Achieved FLOPS by OpenRBC**	**Memory Bandwidth**
IBM POWER 8 'Minsky'	Power8	Power	3.5	10×2	8	160	80×2	560.0	8.7%	230 GB/s
Intel Xeon E5-2695 v3	Haswell	x86-64	2.3	14×2	2	56	35×2	1030.4	4.6%	136 GB/s
AMD Opteron 6378	Piledriver	x86-64	2.4	16×4	1	64	16×4	614.4	4.5%	204 GB/s

three computer systems as shown in Table 2.5.

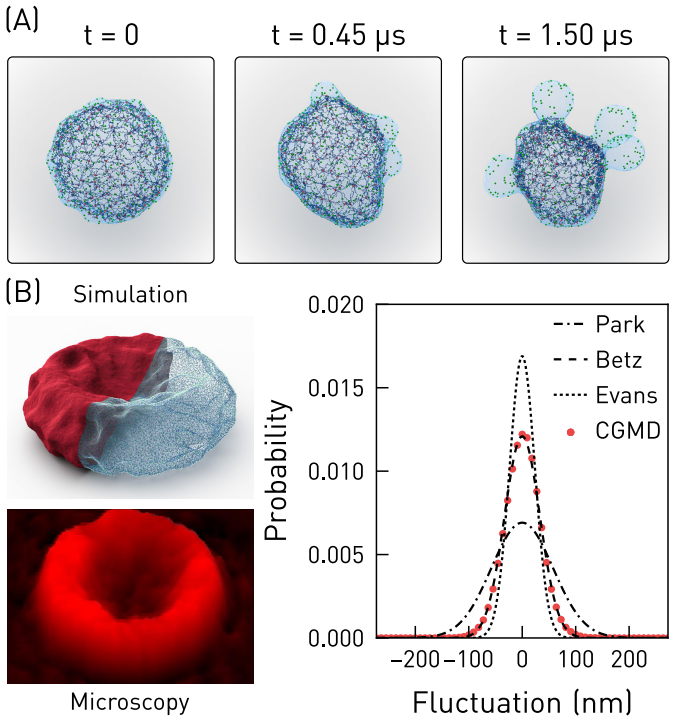


Figure 2.24: (A) The vesiculation procedure of a miniature RBC. (B) The instantaneous fluctuation of a full-size RBC in OpenRBC compares to that from experiments [37, 110, 16]. Microscopy image reprinted with permission from Ref. [110].

Furthermore, OpenRBC can efficiently simulate an entire RBC modeled by 3.2 million particles and correctly reproduce the fluctuation and stiffness of the membrane as shown in Figure 2.24B. The legacy solver was not able to launch the simulation due to memory constraint. The simulation was carried out by implementing the experimental protocol of Ref. [110] that measures the instantaneous vertical fluctuation $\Delta h(x, y)$ along the upper rim of a fixed RBC. In addition, a harmonic volume constraint is applied to maintain the correct surface-to-volume ratio of the RBC. We measured a membrane root-mean-square displacement of 33.5 nm, while previous experimental observations and simulation results range between 23.6 nm to 58.8 nm [37, 110, 16, 38].

Scaling benchmark for the whole cell simulation on the three computer sys-

tems is given in Figure 2.25. It can be seen that compute-bound tasks such as pairwise force evaluation can scale linearly across physical cores. Memory-bound tasks benefit less from hardware threading as expected, however thanks to thread pinning and a consistent workload decomposition between threads there is no performance degradation due to side effects such as cache and bandwidth contention.

It is also worth noting that Fu *et al.* recently published an implementation of a related RBC model in LAMMPS [46], which can simulate 1.15×10^6 particles for 10^5 time steps on 864 CPU cores in 2761 seconds. However, the use of explicit solvent particles in their model generates difficulty in establishing a direct performance comparison between their implementation and OpenRBC. Nevertheless, as a rough estimate and assuming perfect scaling, their timing result can be translated into simulating 8.34×10^6 particles for 0.41×10^5 time steps per day on 864 cores. OpenRBC can perform roughly the same amount of time steps on 20 CPU cores. We do recognized that the explicit solvent model carries more computational workload, and that implementing non-rectilinear partitioning schemes may not be straightforward within the current software framework of LAMMPS. Nonetheless, this comparison does serve to demonstrate the potential of shared-memory programming paradigm on *fat* compute nodes with large amounts of strong cores and memory.

2.3.10 Sectional Summary

We presented a from-scratch development of a coarse-grained molecular dynamics software, OpenRBC, which exhibits exceptional efficiency when simulating systems of large density discrepancy. This capability is supported by an innovative

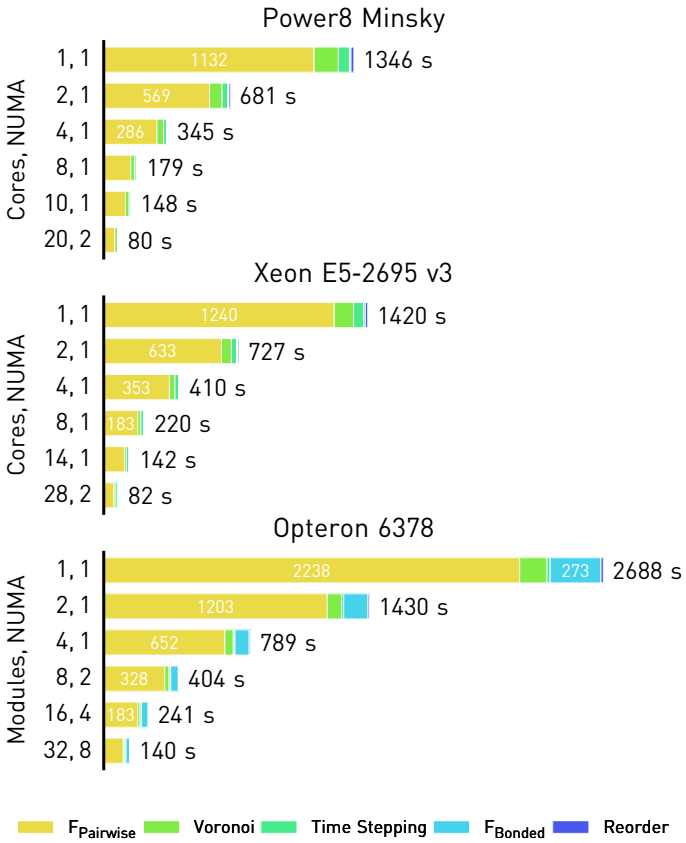


Figure 2.25: Scaling of OpenRBC across physical cores and NUMA domains when simulating an RBC of 3.2 million particles.

algorithm that computes an adaptive partitioning of the particles using a Voronoi diagram. The program is parallelized with OpenMP and SIMD vector instructions, and implements threading affinity control, consistency loop partitioning, kernel fusion, and atomics-free pairwise force evaluation to increase the utilization of simultaneous hardware threads and to maximize memory performance across multiple NUMA domains. The software achieves an order of magnitude speedup in terms of time-to-solution over a legacy simulator, and can handle systems that are almost two orders of magnitude larger in particle count. The software enables, for the first time ever, simulations of an entire RBC with a resolution down to single proteins, and opens up the possibility for conducting many *in silico* experiments concerning the RBC cytomechanics and related blood disorders [86].

Chapter Three

Multiscale Coupling Framework

3.1 Introduction

The potential of multiscale modeling lies in its ability of probing properties of hierarchical systems by capturing events that occur across a wide range of time and length scales that exceed the capability of any single solver and method [118, 102, 155]. More recently, one specific branch, *i.e.* domain decomposition-based concurrent coupling, has seen rapid development since it allows on-the-fly information exchange and interaction between multiple simulation subdomains handled by different solvers.

Multiscale concurrent coupling using domain decomposition dates back to the classical Schwartz alternating method [94, 164], where solutions of a partial differential equation (PDE) on two subdomains can be pursued iteratively. In this method, the calculation for subdomain A is first performed with an enforced pseudo-boundary which extends into the other subdomain B, while the solutions on the pseudo-boundary are dictated to be the known values of B at corresponding locations. A similar calculation is then performed for subdomain B. This procedure is repeated iteratively until convergence of solution in the hybrid region or global domain is achieved. In general, there are two practical strategies to enforce the pseudo-boundary between two subdomains. The first is state-variable, *e.g.* density, velocity, etc., based coupling, where the constraints are placed on the state variables on the two pseudo-boundaries alternately [150, 25]. The other is flux based coupling, where the flux, *e.g.* mass flux, momentum flux, etc., flowing into/out of one subdomain is compensated by the other subdomain so that the laws of conservation are respected [31, 112].

Despite existing theoretical developments, implementing concurrently cou-

pled simulations remains difficult. On one hand, hard-coding remains commonplace in projects that employ *ad hoc* coupling approaches. Such practice can quickly become an obstacle when further development is needed. On the other hand, despite the richness of available coupling schemes and tools [5, 78, 101, 30, 111, 15], adapting existing code to meet the programming interface specification of a coupling framework frequently leads to code refactoring that consumes a substantial amount of man-hours [4]. Such frameworks could also be cumbersome, especially for theorists without an expertise in software engineering, when prototyping new coupling schemes.

As far as we know, a general and non-expert-friendly software library that assists the concurrent coupling of independently developed solvers remains unavailable. The Multiscale Universal Interface (MUI) project aims to fill in this gap by creating a light weight plugin library that can glue together essentially all numerical methods including, but not limited to, Finite Difference, Finite Volume, Finite Element, Spectral Method, Spectral Element Method, Lattice Boltzmann Method, Molecular Dynamics, Dissipative Particle Dynamics and Smoothed Particle Hydrodynamics. Hence, it can deal with Lagrangian or Eulerian descriptions or a mixture of both. It is expected to be able to accommodate a wide range of coupling schemes regardless of the quantities being exchanged, the equations being solved, the time stepping pattern and/or the degree of spatial and temporal separation.

In order to achieve such a high level of universality, MUI is designed to avoid defining the math behind the coupling procedure, *i.e.* it does not specify *which* and *how* quantities are coupled. Instead, it provides services to facilitate the effort of constructing arbitrary coupling schemes by enabling the communication and interpretation of arbitrary physical quantities using arbitrary data types as

demanded by each participating solver.

MUI is simple to use, in the sense that existing solvers do not have to be refactored before using it. MUI provides a very small set of programming interfaces instead of dictating any from the solver. The entire library is coded in a header-only fashion with the Message Passing Interface (MPI) being the only external dependency. Hence, it can be used in exactly the same way as the C++ standard library without pre-compilation. It does not interfere with existing intra-solver communications for solvers using MPI.

MUI is also fast, in the sense that using it only consumes a small amount of CPU time as compared to that used by the solver itself. To achieve this goal we heavily employ the C++ generic programming/template metaprogramming feature to eliminate the abstraction overhead that may otherwise arise when maintaining the high-level flexibility of the framework.

As visualized in Figure 3.1, MUI assumes a **`push-fetch`** workflow and serves as the data exchange and interpretation layer between solvers. While more concrete examples on the usage of MUI are given in Section 2.2.7, the list below outlines the typical steps for incorporating MUI into an existing solver:

1. Substituting the MPI global communicator (Section 3.3.3);
2. Allocating MUI objects;
3. Identifying code regions that supply information to peer solvers and pushing the data as points using MUI (Section 3.2.1);
4. Identifying code regions that require information from peer solvers and fetching through MUI's sampling interface (Section 3.2.2);

5. Configuring inter-solver synchronization (Section 3.2.3);

6. Optimizing performance by managing memory allocation, simplifying communication topology and tuning low-level traits *etc.* (Section 3.2.3, Section 3.3.4, and Section 3.3.2). This step is not mandatory for obtaining correct results but may have an impact on simulation efficiency.

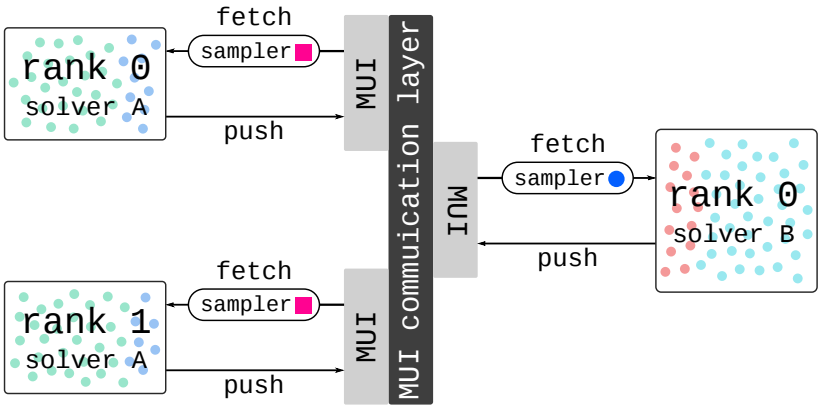


Figure 3.1: MUI facilitates the exchange of information across solvers by letting solvers push and fetch data points. Flexibility of interpolation is achieved by allowing the expression of interpolation algorithms as samplers as well as by accepting data points of arbitrary types.

3.2 Generalized Interpolation

3.2.1 Data Points

A universal coupling framework entails a generalized data representation framework. By observing the fact that discretization is the first step toward any numerical approximation, we realize that essentially every simulation system can be treated as a cloud of *data points* each carrying three attributes, *i.e.* position, type,

and value, as shown in Fig. 3.2. The points might be arranged on a regular grid or connected by a certain topology in some of the methods, but for the sake of generality it is useful to ignore this information temporarily.

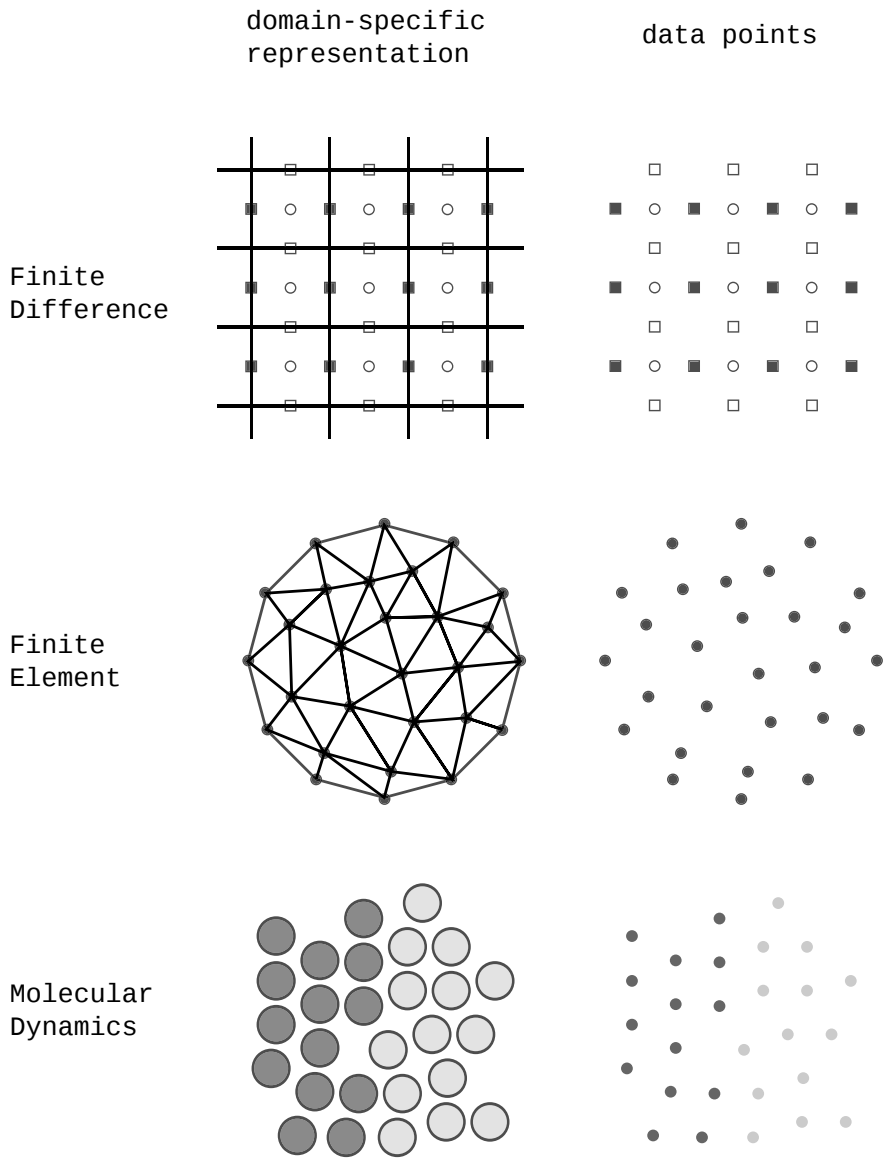


Figure 3.2: Any discrete systems can be generalized as a cloud of data points by ignoring the domain-specific knowledge such as meshes.

MUI defines a generic **push** method for solvers to exchange points carrying different types of data in a unified fashion. The method assumes the signature:

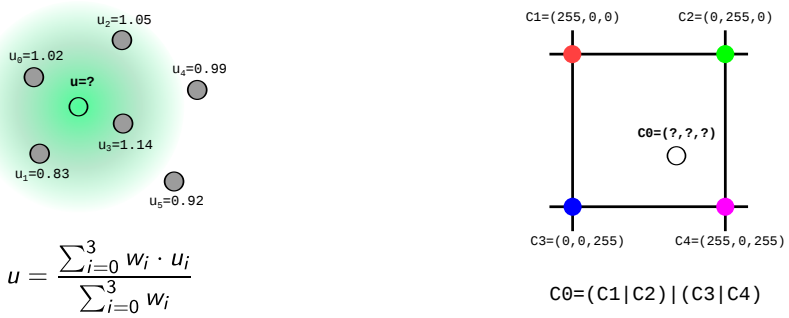
```
template<typename TYPE> inline
bool push( std::string name, mui::point location, TYPE value );
```

The **push** method can accept data points of arbitrary type because it takes the type of the value as a template argument. Points belonging to the same physical variable, *i.e.* points pushed under the same name, are accumulated in a continuous container and sent out to receivers collectively to avoid fragmented communication. Note that the solver takes the responsibility of determining which points get pushed in, because the determination of the interface region uses mostly *prior* knowledge and hence it does not necessarily require direct aid from the coupling library.

3.2.2 Data Sampler

A generic **fetch** method for universal data interpretation on top of the data point representation is less trivial, however. The challenge of achieving universality here lies in the fact that solvers may be agnostic of the math and method used by their peers. Thus, a finite difference code might find itself in need of the value of pressure at grid point (x_0, y_0), yet none of the vertices supplied by its peer finite element solver lies exactly on that point. In this case, the MUI interface is not supposed to simply throw out an exception. Possible solutions could be to use the pressure value defined at the nearest point, or to perform some sort of weighted interpolation using nearby points as shown in Figure 3.3a. The decision for the best algorithm requires knowledge beyond the reach of MUI, but we implemented a flexible data interpretation engine so that users can choose to plug in an appropriate one with minimal effort.

Such engine is enabled by the *data sampler* construct, which is derived from the concept of texture sampling in computer graphics [63]. A texture is essentially a rasterized image of discrete pixels being mapped onto some 3D surface. As a result of 3D projection and transformation, there is no one-to-one correspondence between the pixels of the surface as shown on the screen and the pixels on the texture, and the color of the texture at a fractional coordinate could be displayed. In this case, as shown in Fig. 3.3b, the graphics hardware performs an interpolation (usually bilinear) of the pixel values adjacent to the requested fractional coordinate, and return the interpolated value as the color at the requested point.



(a) Weighted average **(b)** Bilinear color interpolation

Figure 3.3: Weighted kernel sampling and Texture sampling.

The MUI data sampler works in a similar, but enhanced, way: each physical quantity is treated as a *samplable object*, while data samplers are used to interpolate values from the cloud of discrete data points contained in it. A MUI sampler is a class implementing the interfaces **`filter`** and **`support`** as shown by the example of the MUI built-in Gaussian kernel sampler in Listing 3.1. A line-by-line explanation of the C++ code is given below:

- line 1: class template argument declaration. By making the input and output type of samplers template arguments, it is possible to reuse the same interpo-

lating algorithm without duplicating the code merely for the different data types used in different solvers.

- line 4-6: The internal basic data types of MUI are globally parameterized in a configuration class as detailed in Section 3.3.2.
- line 8-11: Constructor that sets up the shape parameters of the Gaussian kernel.
- line 13-27: The **`filter`** method performs data interpolation/interpretation using data points fed by MUI. The MUI virtual container object maps to the subset of the data points that falls within the sampler's support while its usage pattern resembles that of **`std::vector`**.
- line 29-31: MUI uses geometry information provided by the **`support`** method as the extent of the sampler's support to efficiently screened off outlying particles with an automatically tuned spatial searching algorithm.
- line 33-36: storage for sampler parameters, etc.

Listing 3.1: The implementation of the MUI built-in Gaussian kernel sampler

```
1  template<typename OTYPE, typename ITYPE=OTYPE, typename CONFIG=default_config>
2  class sampler_gauss {
3  public:
4    using REAL       = typename CONFIG::REAL;
5    using INT        = typename CONFIG::INT;
6    using point_type = typename CONFIG::point_type;
7
8    sampler_gauss( REAL r_, REAL h_ ) :
9      r(r_),
10     h(h_),
11     nh(std::pow(2*PI*h,-0.5*CONFIG::D)) {}
```

```
12
13   template<template<typename,typename> class CONTAINER>
14   OTYPE filter( point_type focus, const CONTAINER<ITYPE,CONFIG> &data_points )
       const {
15     REAL  wsum = 0;
16     OTYPE vsum = 0;
17     for(INT i = 0 ; i < data_points.size() ; i++) {
18       auto d = (focus-data_points[i].first).normsq();
19       if ( d < r*r ) {
20         REAL w = nh * std::exp( (-0.5/h) * d );
21         vsum += data_points[i].second * w;
22         wsum += w;
23       }
24     }
25     if ( wsum ) return vsum / wsum;
26     else return OTYPE(0);
27   }
28
29   geometry::any_shape<CONFIG> support( point_type focus ) const {
30     return geometry::sphere<CONFIG>( focus, r );
31   }
32
33 protected:
34   REAL r;
35   REAL h;
36   REAL nh;
37 };
```

The sampling procedure works as:

1. Solver invokes the **`fetch`** method of MUI with a **`point of interest`** and a sampler;

2. MUI collects all points that lies within the sampler's support around the point of interest into a virtual container;

3. MUI feeds the sampler with the collected points and lets the sampler perform its own interpolation;

4. The sampler returns the interpolation result back to the user/solver through MUI.

MUI achieves generality in interpolation by allowing users to easily create new samplers to express custom approximation algorithm that can leverage domain-specific knowledge of the system. The value at an arbitrary desired location can be obtained by using samplers that interpolate values from nearby points. In addition, a single piece of sampler code can be used for different data types, *e.g.* **`float`**, **`double`** or **`int`**, because the **`filter`** and **`fetch`** method take the type of the data points as a template argument.

The sampling framework makes it possible for users to fully focus on the design of algorithms while delegating the data management job to MUI. To further simplify the usage, MUI includes several predefined samplers such as a Gaussian kernel sampler, a nearest neighbor sampler, a moving average sampler, an exact point sampler, *etc*.

3.2.3 Storage and Time coherence

Regardless of the actual simulation algorithm, the main body of a solver is essentially a time marching loop in which the quantity of interest is being iteratively solved. Hence, points of the same quantity may be sent to a MUI interface repeat-

edly during a simulation. However, it is inevitable that one solver may run faster than its peer due to factors such as intrinsic performance disparity, load imbalance and transient interruption. In such situations, data points from a later time step may override previous ones belonging to the same quantity before the receiver could ever get a change to sample them.

To address this problem, MUI stores the collection of numerical results generated during each time step as a **frame**. Frames are indexed by their timestamps so different frames do not override each other. Technically, all data points being pushed in for a single physical quantity within a single time step are collectively stored in an instance of MUI's dynamically typed data container. A frame is a collection of mappings from quantity names to actual MUI data container objects, while the frames themselves are again organized in a mapping where time stamps are used as indexing keys. This sparse storage structure, as illustrated in Figure 3.4, allows efficient allocation of memory regardless of whether the time frames are equally distributed or not. It also allows each physical quantity to be selectively committed in a subset of all time frames.

A set of *time samplers* are also predefined in MUI. Time samplers work in essentially the same way as the data samplers, except for that they are one-dimensional along the time axis and use the output from a spatial sampler as the input. In Figure 3.5 we demonstrate the concept of a simple averaging sampler. It is also straightforward to implement more sophisticated time samplers with features such as filtering or prediction.

The memory allocation for frames is managed transparently by a buffering scheme. The deallocation, however, must be set up by user because it is impossible to predict whether a frame will be reused in the future. Utility methods are

provided for users to either explicitly request the disposal of time frames or to let MUI automatically discard frames that are older than a certain age in the simulation units. The default memory length is infinity so no frames will be freed automatically. In situations where batches of data points have to be moved between components of MUI, we use the C++11 move semantic to avoid duplicate memory allocation or copying.

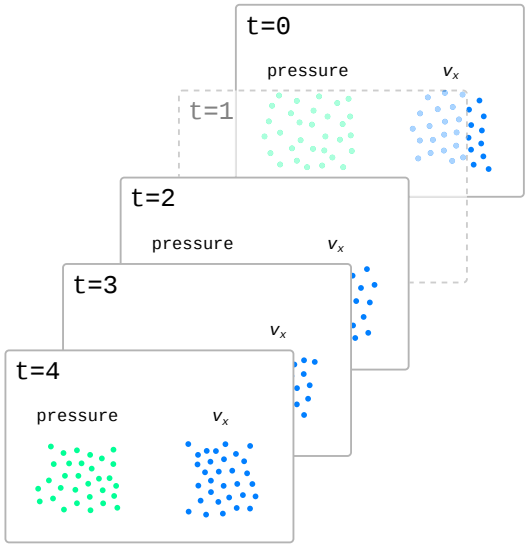


Figure 3.4: Data points committed from different time steps are organized in time frames. Time frames can be non-uniformly distributed. Individual quantities can appear in a select subset, instead of all, of the time frames.

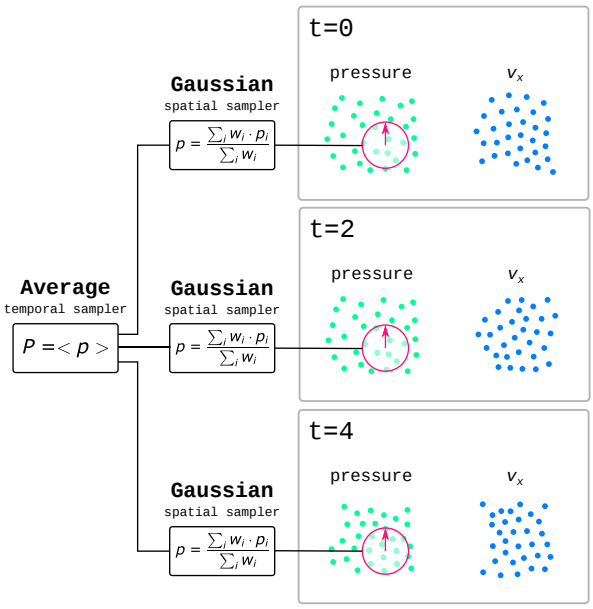


Figure 3.5: Time samplers can interpolate or extrapolate the output of a spatial sampler over a range of time frames.

3.3 Computer Implementation Using the Generic Programming Paradigm and the Message Passing Interface

3.3.1 Typing system

MUI implements a hybrid dynamic/static typing system to combine the performance of static typing with the flexibility of dynamic typing. The dynamic typing behavior of MUI is performed at the level of physical quantities. The value of the first data point received by the **push** method for each physical quantity determines the type of the quantity, while the type of subsequently pushed data points will be examined against the type of the existing storage object. The entire storage object is type-dispatched only once on the receiver's side for each sampling

request. Hence, MUI does not have to perform the expensive type-dispatching for each data points. This coarse-grained dynamic-typing technique is especially important for sampling where data points are being frequently accessed.

The system uses a type list to enumerate all possible types that may be handled by MUI and to automatically generate type-dispatching code. A type list is essentially an instantiation of a variadic class template with the template arguments being the list members. Using recursive templates we can either query the type of a list member using an index (a compile-time constant) or check the index of a type in a given list. A default type list containing frequently used C++ built-in data types is predefined in MUI's default configuration.

Support for new types can be trivially added into MUI by **1)** adding the type into the predefined type list; and **2)** defining the insertion and extraction operator of the type with regard to the data serialization classes in MUI which share the same usage pattern with the C++ standard input and output streams. Since MUI predefines the insertion and extraction operators for all C++ primitive types, an overloaded operator for any composite type can be implemented easily in terms of the primitive ones as illustrated in Listing 3.2.

Listing 3.2: The insertion and extraction operator for type **bond** can be overloaded in a straightforward way as shown below. **mui::istream** and **mui::ostream** are the built-in data serialization classes in MUI.

```
1 // bond is a composite data type
2 struct bond {
3   // double is a primitive data type
4   double k, r0;
5 };
6
7 // overload serialization operator
```

```
8  mui::ostream& operator <<( mui::ostream& ost, const bond &v ) {
9    // enumerate over primitive members
10   return ost << v.k << v.r0;
11 }
12
13 // overload deserialization operator
14 mui::istream& operator >>( mui::istream& ist, bond &v ) {
15   return ist >> v.k >> v.r0;
16 }
```

3.3.2 Customizability

MUI provides a vast customization space regarding communication content, spatial and temporal interpolation algorithm and communication pattern. In addition, MUI allows a number of its low-level traits to be parameterized at compile-time using a configuration class. A default configuration class **`crunch`** is shown in Listing 3.3. The class serves as the last template argument of all MUI component classes and samplers, and is passed along during inheritance and member definition so users only need to specify it once when instantiating the MUI top-level object. It allows the tweaking of:

- dimensionality of the physical space;
- precision of floating point numbers;
- integer width;
- time stamp type;
- type list (as mentioned in Section 3.3.1);

- debugging switch;
- exception handling behavior.

Such static configuration mechanism can eliminate unnecessary runtime polymorphic overhead and also allows MUI to receive better performance optimization during the compilation phase.

Listing 3.3: The default configuration class for MUI

```
struct crunch {
  static const int D = 3;

  using REAL       = double;
  using INT        = int64_t;
  using point_type = point<REAL,D>;
  using time_type  = REAL;

  static const bool DEBUG = false;

  template<typename... Args> struct type_list_t {};
  using type_list = type_list_t<int,double,float>;

  using EXCEPTION = exception_segv;
};
```

3.3.3 MPI Multiple-Program-Multiple-Data Setup

MUI uses MPI as the primary communication mechanism due to its portability, ubiquity, efficiency and compatibility with existing codes. The multiple-program-multiple-data (MPMD) mode of MPI is a natural fit for the purpose of concurrent

coupling. In this mode, each solver is compiled and linked separately as independent executables but invoked simultaneously as a single MPI job using the MPMD syntax. An example of the MPMD launch syntax is given in Listing 3.4. Theoretically, users can launch an arbitrary amount of ranks for each of the solvers irrespective of the number of ranks spawned for its peer solvers.

Listing 3.4: MPI MPMD launch syntax

```
1 mpirun -np n1 solver1 solver1_arguments : -np n2 solver2 solver2_arguments
```

The MUI inter-solver communication topology is established dynamically from the MPI runtime job configuration. To ensure that this MPMD topology is only visible to MUI itself and hidden from the solver code, it is mandated by MUI that solvers should make no direct reference to the MPI predefined world communicator `MPI_COMM_WORLD` for any of its own communications. Instead, a globally accessible MPI communicator, *i.e.* a variable of type **`MPI_Comm`**, should be defined to hold a solver-specific global communicator, which can be obtained from a MUI helper function call that effectively splits the MPI predefined world communicator into subdomains using MPI application numbers. This is in fact one of the few modifications to the solvers that is ever dictated by MUI.

In order to identify and connect MUI instances belonging to different domains, each solver instance needs to initialize MUI using a uniform resource identifier (URI) domain descriptor with the format **`protocol://domain/interface`**. The **`protocol`** field must always be **`mpi`** in the current MUI implementation. It indicates that the inter-solver communication manager sends and receives messages through MPI. Internally, the communication managers are C++ objects allocated through an object factory mechanism, which would allow straightforward addition of new communication managers using alternative protocols such as TCP/IP

or UNIX pipe. In our current communication scheme, the hash value of the **domain** sub-string is used as the *color* for splitting the MPI built-in world communicators into smaller ones each containing a single physical domain. The **interface** sub-string is also hashed to generate a unique integer value for identifying different interface regions in the case of simulating multi-interface systems. The actual algorithm, as given in Algorithm 10, makes use of the MPI **MPI_COMM_SPLIT** and **MPI_INTERCOMM_CREATE** method. Figure 3.6 demonstrates the setup process in a system consisting of two subdomains and an interface between the subdomains.

Algorithm 10 MPI MPMD setup.

```
function InitMpiMpmd(URI)
   ▷ Duplicate world communicator to avoid interfere with solver
   World ← MPI_COMM_DUP(MPI_COMM_WORLD)
   GlobalSize ← MPI_COMM_SIZE(World)

   ▷ Parse URI string
   DomainString, InterfaceString ← Parse(URI)
   DomHash ← std::Hash(DomainString)
   IfsHash ← std::Hash(InterfaceString)

   ▷ Create local and inter-communicators for solver
   LocalDomain ← MPI_COMM_SPLIT(World,DomHash)
   AllDomHash[0..GlobalSize-1] ← MPI_ALLGATHER(DomHash)
   AllIfsHash[0..GlobalSize-1] ← MPI_ALLGATHER(IfsHash)
   root ← min_i | AllDomHash[i] ≠ DomHash & AllIfsHash[i] ≡ IfsHash
   RemoteDomain ← MPI_INTERCOMM_CREATE(LocalDomain,0,World,root,IfsHash)

   return LocalDomain,RemoteDomain
end function
```

3.3.4 Asynchronous I/O and smart sending

MUI assumes an asynchronous communication model because it can be difficult to find synchronization points between multiple heterogeneous solvers. Specifically, MPI collective methods are not used. Instead, MUI makes use of point-to-point non-blocking send and blocking receive methods. The send buffers are stored in a queue alongside with their corresponding MPI requests, and are freed upon completion of the communication. Whenever MUI finds itself in need of data (e.g., due to a fetch request), it continuously accepts incoming MPI messages while also

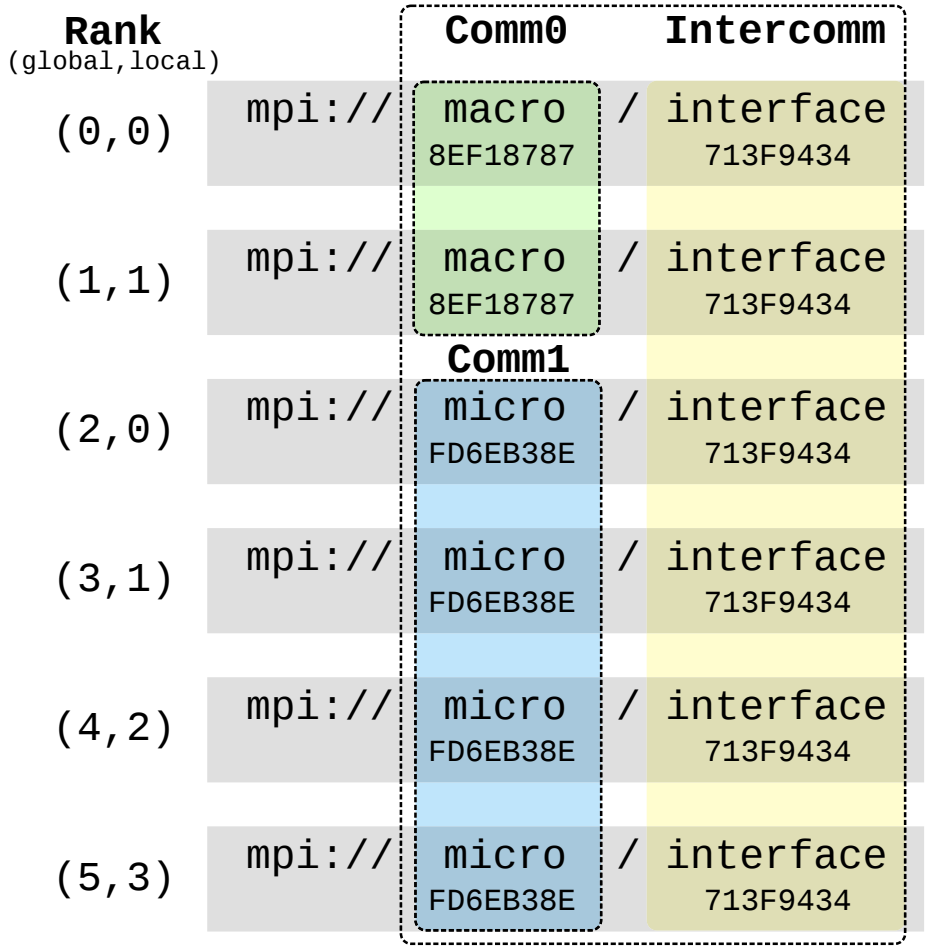


Figure 3.6: In this example we demonstrate how an MPI job consisting of two macroscopic solver ranks and four microscopic solver ranks are partitioned according to the URI domain descriptor. The C++ **`std::hash`** functor can generate unique integer-valued hashes from the domain sub-string. MUI uses the hashes as the colors for splitting the MPI communicator. The hash value of the interface sub-string is then used to establish the inter-communicators that encompass both intra-communicators.

testing for the completion of pending sends until the arrival of the needed data. This asynchronicity is encapsulated within MUI and is completely transparent to the solver. A non-blocking test method is also provided for advanced users to query the availability of data.

An optional *smart sending* feature is also introduced to optimize the amount of MPI messages. It is a selective communication mechanism based on spatial overlap detection. Each solver instance can define two *regions of interest*, *i.e.* a fetch region and a push region, through a Boolean combination of geometric primitives such as spheres, cuboids and points. As illustrated in Figure 3.7, the regions are broadcast among all the processes so that the communication between a sender and a receiver whose regions of push/fetch have no overlap can be safely eliminated. In this way, the communication made by each MUI instance can be localized to a few peers who are truly in need of the data. To accommodate the case of moving boundaries, each region of interest is associated with a validity period. The smart sending feature can be safely ignored for convenience because both regions would default to the entire $\mathbb{R}^d$ with a validity period of infinity as a safety fall-back.

3.4 Demonstration and Application

3.4.1 Couette flow: SPH-SPH coupling

To give a concrete example of the usage of MUI, we present a minimal-working-example (MWE) benchmark of concurrently coupled Smoothed Particle Hydrodynamics (SPH) simulation.

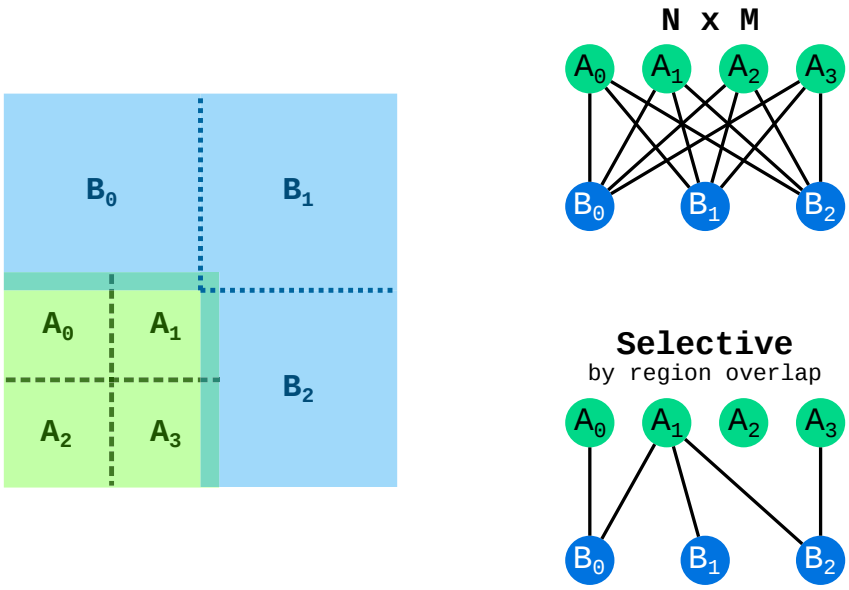


Figure 3.7: An example illustrating the effect of smart sending. The green domain is handled by solver A and spatially decomposed between 4 MPI ranks, while the blue domain is handled by solver B using 3 MPI ranks. By checking for the overlap between the interface regions owned by the ranks, MUI can eliminate unnecessary data transfer between ranks such as $A_0 - B_1$, $A_2 - B_2$ and $A_3 - B_0$ etc.

The algorithm is based on a velocity coupling scheme described in Algorithm 11. As illustrated in Figure 3.8, the system was simulated using two overlapping SPH domains, *i.e.* a lower one and an upper one, using either same or different resolutions. During each time step, the velocity of the SPH particles lying within the receiving part of the overlapped region is set as the average velocity of nearby particles from the other domain as interpolated using a SPH quintic interpolation sampler. We used LAMMPS [115] as the baseline solver, and inserted only about 70 lines of code to implement the algorithm using MUI as given in Listing B.1.

Algorithm 11 SPH-SPH coupling scheme. The C++ code for the **`Quintic`** and **`ExactTime`** samplers are given in Listing B.2 and Listing B.3 in SI, respectively.

```
for t = 0:δt:T_total do
   ▷ Push
   for each particle i do
      if WithinSendRegion(i) then
         MUI::Push("v_x",coord[i],vel_x[i])
      end if
   end for
   MUI::Commit(t)
   ▷ Fetch
   for each particle i do
      if WithinReceiveRegion(i) then
         S_spatial ← Quintic(r,h)
         S_temporal ← ExactTime(ε)
         vel_x[i] ← MUI::Fetch("v_x",coord[i],t,S_spatial,S_temporal)
      end if
   end for
   MUI::Forget(t)
end for
```

We then used the MUI-equipped LAMMPS to model a Couette flow by solving the Navier-Stokes equation. A system of a unit cube was simulated. The volumetric number densities of SPH particles were 20^3 and 40^3 for the lower and upper domains, respectively. As shown in Figure 3.9 the velocity profile obtained from the coupled simulation is consistent with the analytic solution.

The performance and strong scalability of the MUI-equipped SPH solver were further characterized using a similar simulation setup. A system of size $6 \times 1 \times 6$ was simulated as a lower domain spanning from 0 to 0.6125 and an upper

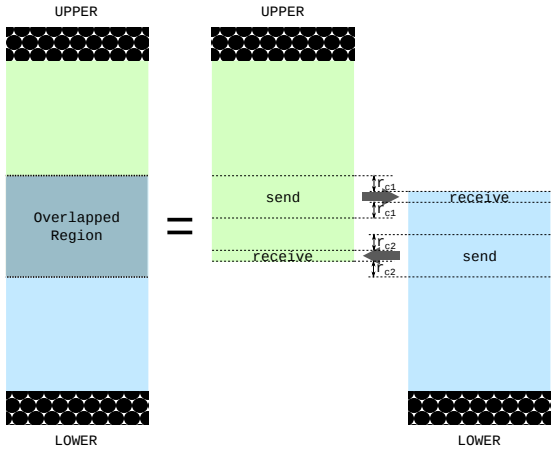


Figure 3.8: The flow was simulated by two overlapping SPH domains. Each domain contains a send region and a receive region that are not overlapping with each other.

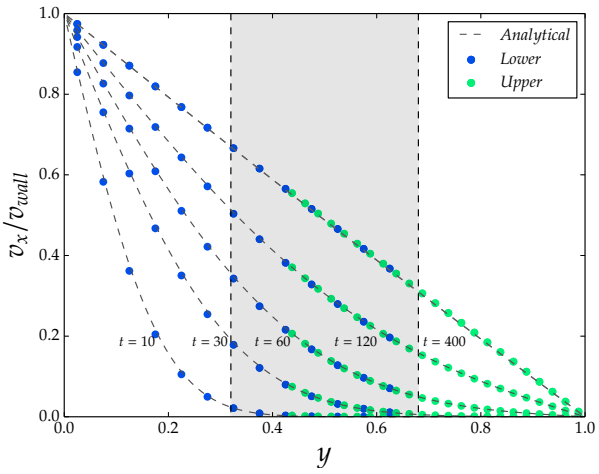
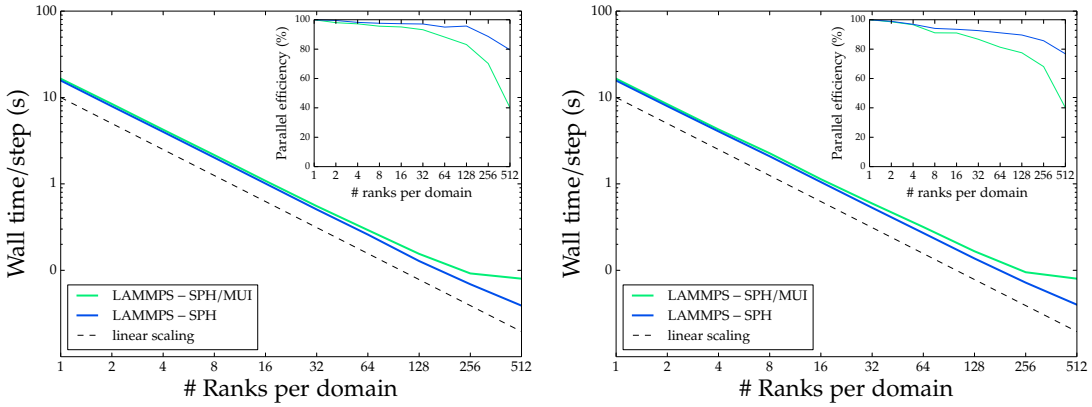


Figure 3.9: Transient velocity profile of a Couette flow. The system was modeled by two SPH domains of different resolutions. Numerical solutions obtained from the lower and upper domains are plotted in blue and green, respectively. The analytic solutions are shown by dashed lines, while the interface region is indicated by the shaded box.

domain spanning from 0.3875 to 1. A number density of 40^3 was used for both the lower and upper subdomains for the ease of performance evaluation. Each domain thus contained 1,382,400 fluid particles and 172,800 wall particles. The computation was performed on the Eos supercomputer using up to 512 ranks for each subdomain on a total of 128 nodes each containing 2 Intel Xeon E5-2670 CPUs at 2.6 GHz. The asynchronous progress engine feature of Cray MPI was enabled to better accommodate the communication pattern of MUI. Two different rank placement strategies, *i.e.* breadth-first and depth-first, were used when increasing the amount of MPI ranks. With the breadth-first strategy, one MPI rank was spawned on each node until a maximum of 128 nodes were used. After that, the number of ranks per node was increased to further fill up the nodes until a total 1,024 ranks were spawned. With the depth-first strategy we first increased the number of ranks per node up to 8 before adding in more nodes. Baseline performance metrics were obtained by simulating the lower and upper domains independently using the original LAMMPS solver.

Figure 3.10 visualizes the measured scalability and parallel efficiency of the MUI-equipped LAMMPS versus the baseline solver as obtained from the benchmark. Figure 3.11 presents a percentage breakdown of the CPU time spent in various parts of the solver at different levels of parallelism. In all cases, MUI consumes no more than 7% of the total CPU time. Note that this includes both the sampling time which is actually part of the useful work and hence should not be counted as overhead.



(a) Breadth-first **(b)** Depth-first

Figure 3.10: Strong scaling performance comparison between the MUI-equipped and the original LAMMPS code.

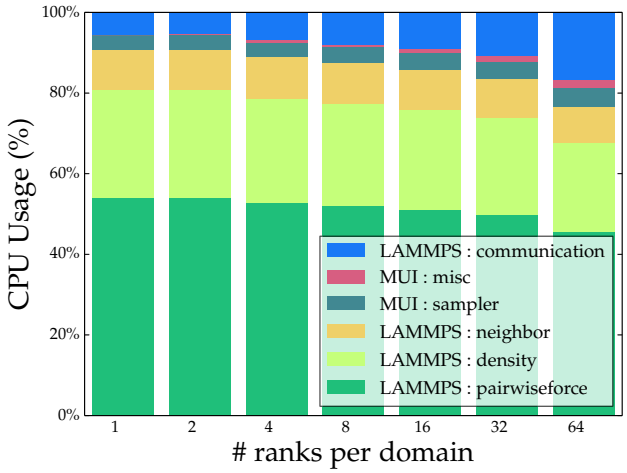


Figure 3.11: Couette flow: a breakdown of the CPU time usage.

3.4.2 Soft Matter: SPH-DPD coupling

Next we demonstrate a concurrently coupled deterministic/stochastic simulation using a similar coupling scheme. As illustrated in Figure 3.12, the flow between two parallel infinitely-large plates driven by a uniform body force was simulated. The upper plate corresponds to a simple no-slip boundary, while the lower plate is grafted by a hydrophobic fourth order binary dendrimers. We used a coupled simulation to investigate the effect of coating on the hydrodynamics of the system. All quantities/parameters mentioned in this simulation are in reduced DPD units.

A system of size $40 \times 110 \times 40$ was constructed using an upper domain and a lower domain. The parameters for setting up the simulation are listed in Table 3.1. A gravity of 0.001 in the x direction was imposed for both domains.

The flow field in the upper domain with a simple no-slip boundary condition was simulated using the Smoothed Particle Hydrodynamics (SPH) method solving the Navier-Stokes equation. The SPH domain spanned from $y = 20$ to $y = 110$ and included a stationary upper wall lying between $y = 100$ to 110 which serves to enforce the no-slip boundary condition.

The flow field in the lower domain was simulated using Dissipative Particle Dynamics (DPD) solving Newton's equation of motion in stochastic form. The DPD domain spanned from $y = -1$ to $y = 28$, and included the solvent and a stationary wall lying between $y = -1$ to $y = 0$ with its upper surface grafted by 160 fourth order binary dendrimers. The surface coverage of the dendrimers was 70%. Interaction parameters between DPD particles are given in Table 3.2. The viscosity of the DPD solvent was measured as 3.72.

The MUI-based coupling scheme is similar to that use in the previous SPH-SPH simulation as described in Section 3.4.1. However, the SPH solver assumes a time step size which is 50 times that of the DPD solver. Accordingly, as demonstrated in Algorithm 12, the SPH domain samples the average velocity of the DPD domain over the last 50 frames to smooth out the randomness, while the DPD domain always samples the latest time frame sent from the SPH domain. The velocity profile converged to that of a Poiseuille flow after 10000 DPD units. As shown in Figure 3.13 the effective channel width is reduced by the hydrophobic coating by about 0.5 DPD unit.

The simulation was done on a workstation with two quad-core Intel Xeon E5-2643 CPUs running at 3.30GHz as well as four nVidia GeForce GTX TITAN GPUs each with 2688 cores. The DPD simulation was done on the four GPUs using the USERMESO package [144], while the SPH code ran on a single CPU core using the same LAMMPS SPH solver as mentioned in the previous example. This processor resource allocation was based on the observation that simulating the DPD domain is orders of magnitude more expensive than simulating the SPH domain.

Table 3.1: Soft matter: parameters for the SPH-DPD simulation

DPD			SPH		
arg	val	description	arg	val	description
N_p	84,375	number of particles	N_p	4,000	number of particles
ρ	5	particle number density	ρ_N	0.064	particle number density
γ	4.5	noise level	ρ_m	5	mass density
σ	3.0	dissipation	η	3.72	viscosity
δt	0.01	time step	δt	0.5	time step
r_c	1.0	cutoff distance	r_c	10	cutoff distance
k_BT	1.0	temperature level	c_s	1.5	speed of sound
g	0.001	body force	g	0.001	body force
$w_D = w_C^{0.5}$		weight function			

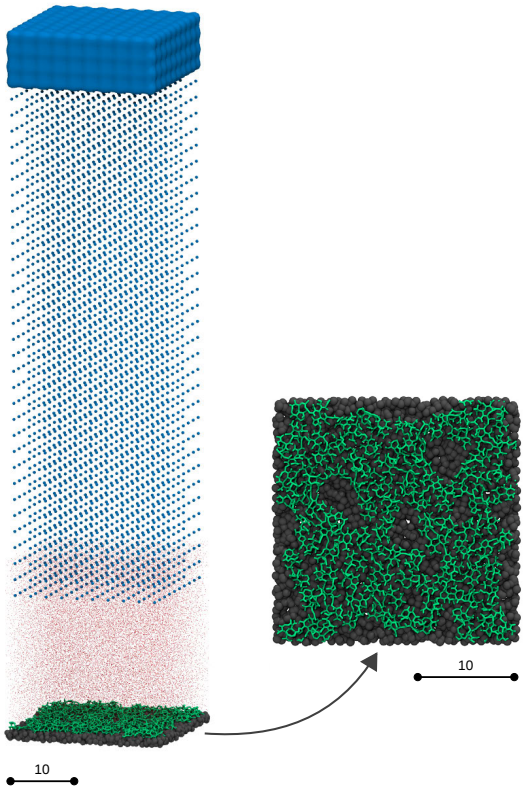


Figure 3.12: A hybrid system consisting of an SPH upper domain and a DPD lower domain was modeled to study the hydrodynamical properties of dendrimer grafted surface. DPD wall particles, dendrimers, DPD solvent particles and SPH fluid particles are rendered ins black, green, red and blue, respectively.

Table 3.2: Repulsive force constants a_{ij} for DPD

	wall	dendrimer	solvent
wall	15	15	15
dendrimer	15	15	75
solvent	15	75	15

Algorithm 12 SPH-DPD coupling scheme. The C++ code for the **SumOver** sampler is given in Listing B.4 in SI.

```
▷ SPH domain

for t = 0:50δt:T_total do
   ▷ Push
   for each particle i do
      if WithinSendRegion(i) then
         MUI::Push("v_x",coord[i],vel_x[i])
      end if
   end for
   MUI::Commit(t)
   ▷ Fetch
   for each particle i do
      if WithinReceiveRegion(i) then
         S_spatial ← Quintic(r_DPD,h_DPD)
         S_temporal ← SumOver(50δt)
         vel_x[i] ← MUI::Fetch("v_x",coord[i],t,S_spatial,S_temporal)
      end if
   end for
   MUI::Forget(t)
end for

▷ DPD domain
for t = 0:δt:T_total do
   ▷ Push
   for each particle i do
      if WithinSendRegion(i) then
         MUI::Push("v_x",coord[i],vel_x[i])
      end if
   end for
   MUI::Commit(t)
   ▷ Fetch
   t_SPH ← Floor(t,50δt)
   for each particle i do
      if WithinReceiveRegion(i) then
         S_spatial ← Quintic(r_SPH,h_SPH)
         S_temporal ← ExactTime(ε)
         vel_x[i] ← MUI::Fetch("v_x",coord[i], t_SPH ,S_spatial,S_temporal)
      end if
   end for
   if Mod(t,50δt) = 0 then
      MUI::Forget(t − 50δt)
   end if
end for
```

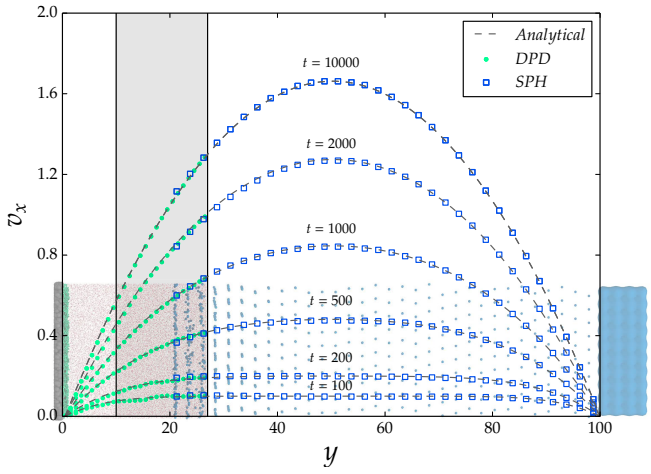


Figure 3.13: The velocity profile derived from the coupled SPH/DPD simulation converges to that of a Poiseuille flow. SPH results, DPD results and the analytic solutions are plotted in blue dots, green dots and dashed line, respectively. The background picture indicates the system composition at the corresponding y position.

3.4.3 Conjugate Heat Transfer

We further demonstrate a coupled Eulerian/Lagrangian simulation of the cooling process of a heating cylinder immersed in a channel flow using the energy-conserving Dissipative Particle Dynamics (eDPD) method [88] and the finite element method (FEM). The eDPD model is an extension to the classical DPD model [62, 35] with explicit temperature and heat transferring terms. The FEM solver can solve the time-dependent heat equation

$$\frac{\partial T}{\partial t} = -\alpha \Delta T + f$$

where α is the thermal diffusivity and $f = f_0 + f_{\text{interface}}$ the heat source.

The coupling scheme is shown in Algorithm 13. The FEM domain pushes the temperature of boundary vertices, with which the eDPD solver calculates the heat flux generated by each particles surrounding the cylinder. The eDPD solver then pushes the flux data back into the FEM solver. The FEM solver averages the

fluxes computed by eDPD and assigns the result to boundary vertices based on a Voronoi diagram of the vertices. The heat flux value is used as the Neumann boundary condition required in solving the time-dependent Poisson equation. The accuracy of the scheme was validated by solving for the temperature profile in a quartz-water-quartz system whose left and right boundaries were fixed at $270K$ and $360K$ as shown in Figure 3.14. The thermal diffusivities of water and quartz were assumed to be constant at $0.143 \times 10^6 m^2/s$ [17] and $1.4 \times 10^6 m^2/s$ [49] over the temperate range, respectively, while the interfacial thermal diffusivity was chosen as the arithmetic mean between the two values.

Algorithm 13 eDPD-FEM coupling scheme. The C++ code for the **`VoronoiMean`** and **`Linear`** samplers are given in Listing B.5 and Listing B.6 in SI, respectively.

```
▷ eDPD domain
for t = 0:δt:T_total do
   ▷ Push
   t_FEM ← Floor(t, 10δt)
   for each particle i do
      if WithinCutoffOfCylinder(i) then
         S_spatial ← Linear(h_max)
         S_temporal ← ExactTime(ε)
         T_wall ← MUI::Fetch("T", coord[i], t_FEM, S_spatial, S_temporal)
         q ← PerParticleHeatFlux(T[i], T_wall)
         MUI::Push("q", coord[i], −q/C_v)
      end if
   end for
   MUI::Commit(t)
   if Mod(t, 10δt) = 0 then
      MUI::Forget(t − 10δt)
   end if
end for

▷ FEM domain

for t = 0:10δt:T_total do
   ▷ Push
   for each boundary vertex i do
      MUI::Push("T", coord[i], T[i])
   end for
   MUI::Commit(t)
   ▷ Fetch
   for each boundary vertex i do
      S_spatial ← VoronoiMean(Vertices)
      S_temporal ← MeanOver(10δt)
      f_interface[i] ← MUI::Fetch("q", coord[i], t, S_spatial, S_temporal)
   end for
   MUI::Forget(t)
   SolveForNextStep
end for
```

A system composed of a 3D fluid domain filled with water, a 2D solid domain

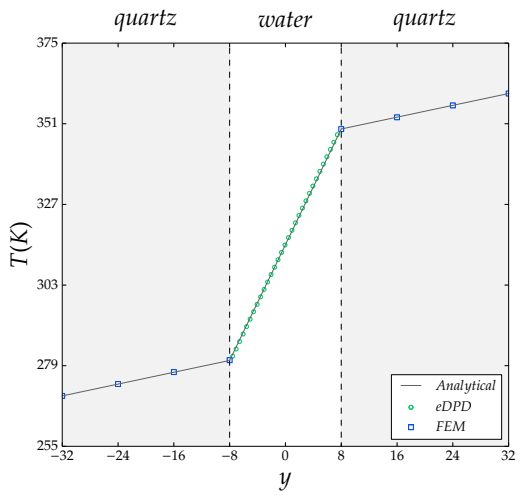


Figure 3.14: Heat conduction: temperature profile obtain for a quartz-water-quartz tri-layer system with the FEM solver handling the solid domain and the eDPD solver handling the fluid domain.

and a fluid-solid interface was then simulated using the validated scheme. The entire domain is periodic in x and z direction, but is bounded by a pair of no-slip infinite walls of constant temperature in the y direction. The fluid domain and the walls are simulated using eDPD, while the solid domain is solved using FEM. Parameters and scaling factors used to set up the eDPD and FEM calculations are given in Table 3.3. The simulation result is shown in Figure 3.15. The Reynolds number is defined by $Re = (v_{max}D)/\nu = 1.97$ where $v_{max} = 0.65L_0/\tau$ is the maximum inlet velocity, $\nu = 6.62$ the kinematic viscosity and $D = 20.0L_0$ the diameter of the cylinder.

We obtained a smooth temperature transition in the hybrid domain using MUI and the aforementioned coupling scheme. The example demonstrates the capability of MUI in coupling two different fields, *e.g.* a flow field and a thermal field. The method can facilitate the study of inhomogeneous coolants, *e.g.* colloidal suspensions, thanks to the flexibility brought about by the particle method.

The simulation was performed on a workstation with two hexa-core Intel Xeon E5-2630L CPUs running at 2.0GHz. The eDPD simulation occupied 11 CPU cores

using our customized LAMMPS code. The FEM solver ran on a single CPU core. This computational resource configuration was based the relative computational cost of the two codes.

Table 3.3: Conjugate heat transfer: parameters for the eDPD-FEM simulation of immersed heating cylinder are taken from Ref [88].

length scale	$L_0 = 11nm$
time scale	$\tau = 0.935ns$
temperature scale	$T_0 = 300K$
mass scale	$m_0 = 3.32 \times 10^{-22}kg$

eDPD		FEM	
arg	val	arg	val
α	$1.43 \times 10^{-7}m^2/s$	α	$1.43 \times 10^{-7}m^2/s$
δt	0.0125τ	δt	0.125τ
ν	$8.57 \times 10^{-7}m^2 \cdot s^{-1}$	f_0	$0.004T_0/\tau$
		space	P_1

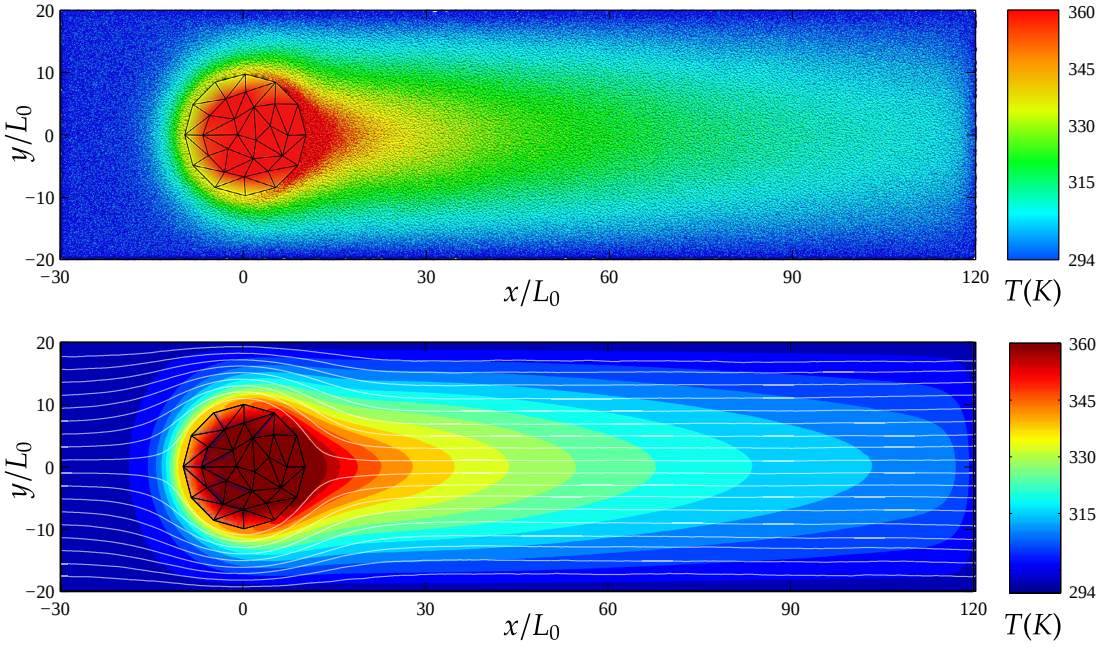


Figure 3.15: Conjugate heat transfer: a snapshot of the hybrid particle/mesh structure of the domain at steady state is shown in the upper plot; streamlines and the temperature field in the system at steady state are visualized in the lower plot by white lines and color-mapped contours, respectively.

Chapter Four

Non-Equilibrium Dynamics of Thermo-Responsive Polymers

4.1 Thermo-Responsive Polymers as Smart Materials

Thermoresponsive polymers constitute a unique class of smart materials [65, 127, 34] that have received increasing attention due to their unique potential applications in fields such as drug delivery [100, 133, 68, 73, 147, 168] and nanotechnology [72, 153, 125, 149]. Above a certain critical temperature, a thermoresponsive polymer exhibits a sharp transition of solubility, manifested on the phase diagram either as an upper critical solution temperature (UCST) or a lower critical solution temperature (LCST). Simply put, a UCST polymer dissolves above its critical temperature, whereas an LCST polymer precipitates.

Akin to conventional amphiphilic copolymers [167, 66], copolymers with multiple thermoresponsive blocks [171, 99, 105, 71, 106] can form highly tunable self-assemblies which can be engineered for various applications [92, 95]. While a quantitative understanding of the rich dynamic behavior of doubly thermoresponsive self-assemblies is vital for the efficient fabrication and utilization of these materials, experimental characterization can be laborious considering the vast parameter space generated by the combination of UCST/LCST behavior and transition temperature of each constituting block as well as environmental factors. In this paper, we present a study of the behavior of doubly thermoresponsive micelles and vesicles through computer simulations using a new energy-preserving Dissipative Particle Dynamics (eDPD) model [88, 144, 89], an extension of the classical Dissipative Particle Dynamics (DPD) method.

Among various molecular modeling techniques, DPD has been proven to be a powerful tool for mesoscopic modeling of soft matter due to its capability in reproducing both static and dynamic properties of the system [90, 48]. The clas-

sical DPD formulation integrates a built-in pairwise thermostat that effectively maintains the system temperature at a target value. Being a valuable feature for most simulation scenarios that require isothermal conditions, this thermostat turns out to be an obstacle for studying temperature-related phenomena. The eDPD formulation allows us to overcome this limitation by modeling (at a coarse-grained molecular level) internal energy as a degree of freedom of the particles and therefore to calculate temperature-dependent pairwise interaction parameters on a per-particle basis. This capability allows us to model the *transient* and *local* behavior of thermoresponsive block copolymers under different heating rates and in flow fields with uneven temperature distribution. The eDPD formulation, together with auxiliary functionalities, was implemented into the GPU-accelerated LAMMPS package `USER-MESO` [144]. The source code is open under the GPLv3 license at `http://www.cfm.brown.edu/repo/release/USER-MESO/`.

4.2 eDPD Model for Thermo-Responsive Polymers

The model is based the eDPD formulation as detailed in Section 2.1.1. The following weight functions were used in this study for forces and heat fluxes:

$$w_R(r_{ij}) = w_C(r_{ij}) = w_{RT}(r_{ij}) = (1 - \frac{|r_{ij}|}{r_c}) \tag{4.1}$$

$$w_D(r_{ij}) = w_{CT}(r_{ij}) = w_R^2(r_{ij}) \tag{4.2}$$

The affinity between different types of DPD particles is controlled by the conservative force coefficients a_{ij}. A value bigger than the reference value $a_{ij}^* \doteq 75 k_B T/\rho$ indicates that the particles are immiscible, while a smaller value indicates good

compatibility. In order to reproduce the temperature-dependent behavior of LCST and UCST polymers, we set the excess repulsion $\delta a_{ij} \doteq a_{ij} - a_{ij}^*$ to be a sigmoidal function of T [89]:

$$\delta a_{ij} = \frac{\delta a_{ij}^0}{1 + \exp[\omega(T_{ij} - T_0)]} \tag{4.3}$$

As visualized in Fig. 4.1, δa_{ij}^0 defines the maximum excess repulsion, while negative and positive values of ω give rise to LCST and UCST behaviors, respectively. The magnitude of ω determines the sharpness of the transition.The complete form of the temperature-dependent conservative force coefficients therefore becomes:

$$a_{ij}(T_{ij}) = \frac{75 k_B T_{ij}}{\rho} + \frac{\delta a_{ij}^0}{1 + \exp[\omega(T_{ij} - T_0)]} \tag{4.4}$$

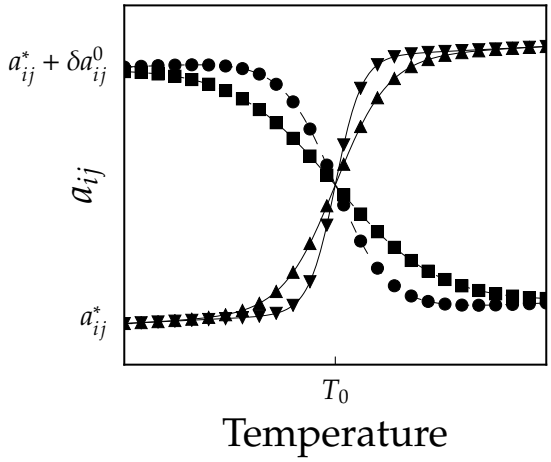


Figure 4.1: Pairwise repulsive coefficient a_{ij} as a function of temperature as determined by Eq. (4.4).

We carried out the simulations in reduced units. The dimensionless temperature T, length L and mass m are scaled from their physical counterparts T^*, L^* and

m^* with a reference scale $T_0 = 300$ K, $L_0 = 2.99$ nm and $m_0 = 4000$ Da [88]:

$$T \doteq \frac{T^*}{T_0} = \frac{T^*}{300\ \mathrm{K}},\ L \doteq \frac{L^*}{L_0} = \frac{L^*}{2.99\ \mathrm{nm}},\ m \doteq \frac{m^*}{m_0} = \frac{m^*}{4000\ \mathrm{Da}} \tag{4.5}$$

where m_0 is determined using the density of water and a DPD particle number density $\rho_n = 4$.

The measured diffusivity D^P of the solvent beads, each representing 222 water molecules ($N_\mathrm{w} = 222$), is 0.305 in the dimensionless units. By connecting this to the experimental self diffusivity of water $D^W = 2.43 \times 10^{-9} \mathrm{m}^2/\mathrm{s}$, the time scale can be determined as [53]

$$\tau = \frac{N_\mathrm{w} * D^P * L_0^2}{D^W} = \frac{222 \times 0.305 \times (2.99 \times 10^{-9}\ \mathrm{m})^2}{2.43 \times 10^{-9}\ \mathrm{m}^2 \cdot \mathrm{s}^{-1}} \approx 250\ \mathrm{ns} \tag{4.6}$$

A time step size of $\delta t = 0.01\tau$ was used throughout the study.

To verify the ability of our proposed eDPD model in reproducing the temperature-induced conformational variation of thermoresponsive polymers near the θ temperature, we plot the time-averaged per-monomer radius of gyration $\langle \frac{R_g^2}{N} \rangle$ of a single polymer chain versus temperature and contour length as obtained from ensemble-averaged equilibrium eDPD simulations. With $a_{ij}^* = 10.0$, $\delta a_{ij}^0 = 27.5$, $\omega = 100$, we observe the lines for different chain length converge at the θ temperature as shown in Fig. 4.2A. This is in qualitative agreement with Ref. [161, 122] as in Fig. 4.2B.

block in the direction perpendicular to the membrane. The modes are ca
by the percentage of molecules that belonged to the mode as classified using a
nearest-neighbor algorithm under cosine similarity (see Supplementary I
tion for details of the POD analysis). The two dominant modes, combined with
examples as given in Fig. 4.3C, can be interpreted as such: 1) in the first m
trajectories of LCST and UCST blocks cross once, thus indicating that the n

flips during the process; 2) in the second mode, the two trajectories move from above zero to below in parallel, indicating the molecule *slips* from one leaflet of the membrane into the other one without rotation. The other modes represent higher frequency oscillations due to the stochastic nature of the molecular movement. Surprisingly, 21.5% of the molecules in fact assumed the *slip* mode and hence did not invert orientation although the membrane inverted its overall composition. This exchange rate is significantly higher than the spontaneous flip-flop rate between leaflets of lipid membranes [28] and may be used as a means of cross-membrane signaling for thermoresponsive polymersomes.

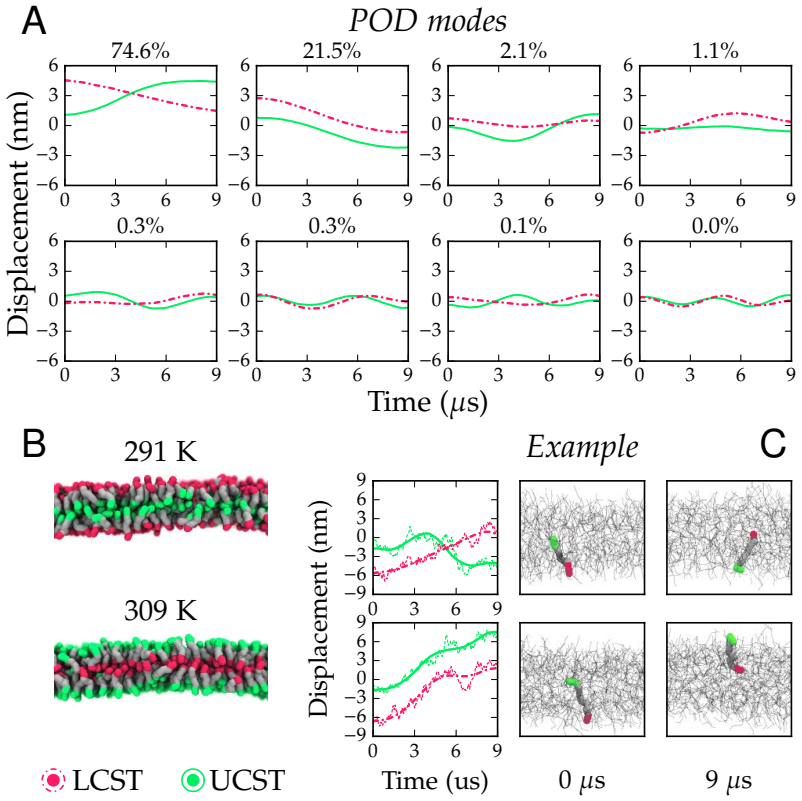


Figure 4.3: POD analysis reveals two dominant molecular movement modes, *i.e.* flip and slip, during membrane inversion. ***(A)*** POD modes of UCST and LCST block trajectories during membrane inversion; ***(B)*** bilayer membrane formed by the $L_2N_5U_2$ thermoresponsive block copolymer; ***(C)*** examples of molecules following the *flip* and *slip* modes (see also Supplementary Video 1 and 2); surrounding molecules are rendered as gray lines. Dashed lines represent exact trajectories, while solid lines are smoothed versions of the trajectories reconstructed from the 8 most energetic POD modes. LCST, UCST, and non-responsive blocks are colored in red dashes, green solids, and gray, respectively.

4.4 Vesicle dynamics

Under certain conditions, a unilamellar vesicle (ULV) formed by a $L_2N_5U_2$ triblock copolymer may invert by switching its surface and internal thermoresponsive blocks reversibly as shown in Fig. 4.4A. The L and U blocks are LCST and UCST, respectively, while the N blocks are non-responsive hydrophobic blocks. The evolution of the radial density distribution $\rho(r)$ for each type of polymer blocks during one inversion is given in Fig. 4.4B. An interchange of the shapes of LCST and UCST density distribution is observed. The peaks of the curves raised or lowered vertically over time instead of shifting horizontally, while the density distribution of the non-responsive blocks only displayed minor change. This indicates that the molecules did not invert collectively in a lock-step fashion but rather by diffusing through the wall formed by the non-responsive blocks.

The collapse and subsequent release of the containing fluid of a vesicle is a stochastic event due to the metastable nature of ULVs [108], and is critical in designing vesicle-based drug carriers. To quantify this behavior, a thermal loading test was performed in which 576 ensembles each containing a single ULV were subjected to repeated heating-cooling cycles of frequencies ranging between 5 μs/cycle and 1.25 ms/cycle. As shown in Fig. 4.4C, both very high and low thermal loading frequencies eventually led to the collapse of the vesicles in all ensembles. For these cases the collapse probability distribution parameters, mean survival time, and half-life were estimated and given in Table 4.1. The disruption of vesicle structure under the 5 μs/cycle and 12.5 μs/cycle loading frequencies can be attributed to the constant changing of hydrophilicity of the thermoresponsive blocks at a pace faster than that of a complete inversion. The collapse of vesicles at very low frequencies is likely a consequence of the system staying too long near the

Figure 4.4: Thermoresponsive vesicles invert by diffusion and respond differently t
thermal loading frequencies. ***(A)*** a vesicle formed by $L_2N_5U_2$ thermoresponsive block c
can invert repeatedly when subjected to thermal loading cycles and may collapse irreversibly;
(B) radial density distributions of polymer blocks during one inversion. ***(C)*** at both high
frequencies vesicles collapse completely, but at intermediate frequencies many vesicles ca
for a long time. LCST, UCST and non-responsive blocks are in red dashes, green solids,
respectively.

θ temperature, where both thermoresponsive blocks were weakly hydrophobic. At intermediate frequencies, however, up to 50-70% of the vesicles can survive for a long time after a short knockout phase which destroyed the less stable ones.

Table 4.1: Survival rates and estimated collapse probability of vesicles under different thermal loading frequencies. An asterisk indicates a field that cannot be estimated reliably due to the long tail of the failure probability.

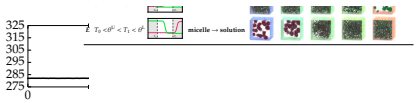
Period	Survived	$p(t;a,b) = \frac{t^{a-1}}{\Gamma(a)b^a}e^{-\frac{t}{b}}$		Mean lifetime	Half-life
		a	b	(ms)	(ms)
5 μs	0.0%	1.170	0.239	0.28	0.21
12.5 μs	0.0%	0.394	23.10	9.11	3.25
25 μs	47.2%	*	*	*	*
50 μs	58.3%	*	*	*	*
125 μs	58.3%	*	*	*	*
0.25 ms	54.2%	*	*	*	*
0.50 ms	70.8%	*	*	*	*
1.25 ms	4.2%	0.461	29.07	20.36	8.59

4.5 Micelle dynamics at still and in flow

Consider the behavior of micelles formed by a LCST-UCST diblock copolymer upon heating and denote by θ^{L} and θ^{U} the critical temperature of the LCST and UCST blocks, respectively. Given a temperature range of $[T_0, T_1]$ that the system may experience, the combination of LCST and UCST behavior and the relative difference between θ^{L}, θ^{U}, T_0 and T_1 gives rise to five possible cases of thermally induced assembly as revealed by our eDPD simulations and summarized in Table 4.2. In case ***A***, the LCST block and the UCST block share the same critical temperature. In this case, an *inversion* of the micellar core-shell structure was observed upon heating from below the critical temperature to above due to an interchange of solvent affinity of the polymer blocks. In case ***B***, the UCST block has a critical temperature above the highest temperature simulated and hence

was always hydrophobic during the simulation. In this case, the polymers formed micelles at low temperature but Janus-like globular aggregates at high temperature as the LCST block lost hydrophilicity upon heating. In case ***C***, the LCST block has a critical temperature below the lowest temperature accessible by the simulation and therefore was always hydrophobic. In this case, the polymers formed hydrophobic globular aggregates at low temperature but micelles at high temperature as the UCST block gained hydrophilicity upon heating. Cases ***B*** and ***C*** can be effectively viewed as the first and second half of the process in Case ***A***. In case ***D***, the UCST critical temperature is low enough to render the UCST blocks constantly hydrophilic. The polymers assembled into micelles from a uniform dispersion as the LCST block lost hydrophilicity upon heating. Case ***E*** is simply the inverse of ***D***. ***D*** and ***E*** can be regarded as the process of drug loading and unloading into/from the hydrophobic core of the micelles, respectively.

Table 4.2: Five thermally induced assembly scenarios leading to fundamentally different structures. A tag is assigned to each scenario for later discussion. Snapshots of the systems over the heating process are given in the last column. LCST and UCST blocks are in red dashed and green solid lines, respectively. Background colors of the simulation boxes indicate temperature ranging from 288 K (blue) to 312 K (red).

Tag		
θ	$A_{\text{polymer/solvent}}$ $T_0 \xrightarrow{\Delta} T_1$	

Given the non-equilibrium nature of the inversion of thermoresponsive micelles, it is natural to expect that the process can be controlled by adjusting the rate at which solvent affinity changes [56]. To probe this, we simulated case ***A*** again with two replicas from exactly the same initial configuration. The first replica underwent a smooth heating procedure, while the second one experienced a sharp temperature jump. In the first replica, large aggregates formed via the fusion of smaller ones at around $T = 300$ K, *i.e.* the θ temperature of the blocks. These aggregates eventually formed large spherical micelles as visualized in Fig. 4.5A. In the second replica, fast heating quickly generated large numbers of small micelles with a narrower size distribution as illustrated by Fig. 4.5B. Therefore, it is possible to use the *heating rate* to tune the size distribution of thermoresponsive micelles [135].

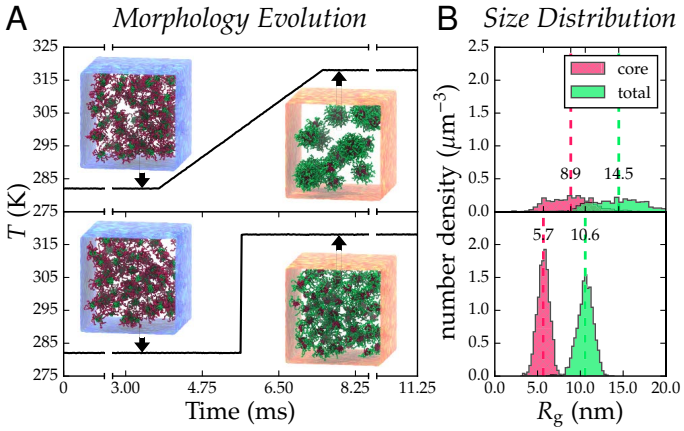


Figure 4.5: Fast heating gives rise to smaller micelles with a sharper size distribution. ***(A)*** morphological change of micelles in fast and slow heating; ***(B)*** size distribution of the micelles after inversion.

The critical temperature, as an intrinsic property of the polymers, also significantly affects the inversion dynamics. The difference $\Delta\theta \doteq \theta^{\mathrm{U}} - \theta^{\mathrm{L}}$ between the critical temperatures of the blocks in a copolymer is more informative than the absolute values because the latter can be always tuned with respect to a target temperature[154, 70]. Therefore, we simulated UCST-LCST diblock copolymers of different $\Delta\theta$ subjected to a gradual heating process and characterized the process

in Fig. 4.6 using direct visualization and the average radius of gyration R_g of the polymers.

Large positive $\Delta\theta$, as shown in Fig. 4.6A, yielded V-shaped R_g curves with a deep valley followed by a high plateau. A max-min difference of up to ~ 0.83 nm indicates that the polymer chains deformed significantly at intermediate temperatures. The reason is that the LCST blocks lost hydrophilicity quite early at a low temperature, while the UCST blocks only became hydrophilic at a relatively high temperature. As a result neither block was soluble at intermediate temperatures. This led to large chain deformations during the consequent aggregate formation and fusion process. An overall increase of ~ 0.35 nm in R_g was observed at the end of the process. This is mostly contributed by the LCST blocks whose R_g increased by ~ 0.4 nm as shown in Fig. 4.6B. Meanwhile the R_g of the UCST blocks slightly decreased by ~ 0.05 nm.

If $\Delta\theta$ is negative or close to zero, the R_g curves are W-shaped with two shallow minima as shown in Fig. 4.6A. The two minima correspond to the solvation of the UCST blocks and the precipitation of the LCST blocks. The solvation events happened before the precipitation, resulting in an intermediate state of fully dispersed polymer solution. This may result in a thorough release of the payload due to the complete disassembly of thermoresponsive micelles. The change in average R_g after the inversion is negligible because the increase of the LCST blocks is canceled by the decrease of the UCST blocks as shown by Fig. 4.6B. The average size of the obtained micelles is small and similar to those formed in the fast heating scenario, because no aggregate formation or fusion process occurred.

We observed that the final average R_g of the polymers is highly sensitive to $\Delta\theta$ as indicated in Fig. 4.6C. A bisection search revealed the existence of a

critical temperature $\Delta\theta_c$ = 4.08 K, around which a slight difference in $\Delta\theta$ could result in a sharp change in R_g. This can be considered together with a direct visualization of the micellar morphology evolution for $\Delta\theta$ = 3 K and 6 K as given in Fig. 4.6D. A disintegration-reintegration process occurred when $\Delta\theta$ = 3 K, while an aggregation-fission process occurred instead when $\Delta\theta$ = 6 K, where toroidal micellar intermediates formed and eventually broke down into spherical micelles. It is the alteration between the disintegration-reintegration and the aggregation-fission pathway that gives rise to this sensitive response of R_g with respect to $\Delta\theta$. We also note that a sharp temperature jump always direct the system into the disintegration-aggregation pathway irrespective of $\Delta\theta$.

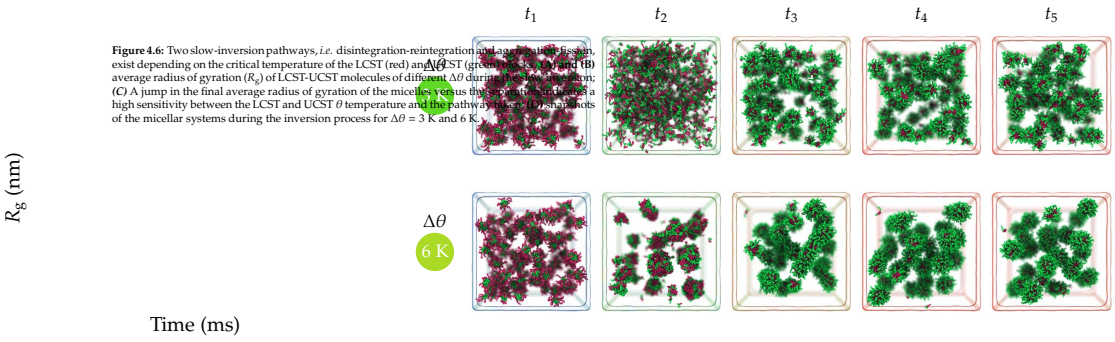


Figure 4.6: Two slow-inversion pathways, *i.e.* disintegration-reintegration and aggregation-fission, exist depending on the critical temperature of the LCST (red) and UCST (green) [B] and **(B)** average radius of gyration (R_g) of LCST-UCST molecules of different $\Delta\theta$ during the slow inversion; **(C)** A jump in the final average radius of gyration of the micelles versus the $\Delta\theta$ and thus a high sensitivity between the LCST and UCST θ temperature and the pathway taken. **(D)** Snapshots of the micellar systems during the inversion process for $\Delta\theta$ = 3 K and 6 K.

4.6 Related Work

Self-assemblies formed by thermoresponsive copolymers have been observed experimentally to switch their inside and outside blocks following a change in ambient temperature [7]. However, there is a lack of details on how the inversion proceeds in terms of the dynamics of individual polymer chains. In absence of direct data on the inversion dynamics of *thermoresponsive* polymers, we consider experimental work reporting the morphology and inversion dynamics of *PH-sensitive* polymer aggregates as a close analog for comparing against our simulation results. Wang *et al.* [151] examined the inversion dynamics of micelles formed by PVBA-PMEMA diblock copolymers using a stopped-flow apparatus with light scattering detector. They found that the inversion from VBA-core to MEMA-core follows a fusion-disintegration-reintegration pathway, while the inversion from MEMA-core to VBA-core proceeds by micelle splitting. Similar events can be found in Fig. 4.6 but in a different ordering of the events. Liu and Eisenberg [96] studied the inversion dynamics of vesicles formed by PAA-PS-P4VP triblock copolymers and postulated that the inversion involves the diffusion of polymer chains through a softened wall but did not provide evidence. We provide a systematic and quantitative evidence for this as shown in Fig. 4.4, which characterizes the evolution of block density distribution, as well as the molecular movement modes analysis as detailed in Section 4.3.

The classical DPD method has been used for the study of various thermoresponsive polymer systems such as micelles [61], self-regenerating gels [165], multilayered polymersomes formed by asymmetric thermoresponsive brushes [23], and model liposomes [163]. However, in these studies the interaction between the thermoresponsive polymer and solvent is either defined as a function of the

global system temperature T or simply adjusted manually at a given point during the simulation to mimic the thermoresponsive effect. This is an artificial consequence of the pairwise thermostat of the classical DPD method which constraints the system in an isothermal state. As a result only equilibrium properties or dynamic processes incurred by an infinitely sharp temperature jump can be reliably studied. In our eDPD model, however, the pairwise repulsion only depends on the local temperature T_i and T_j of the two interacting particles and hence allows the study of non-isothermal processes and systems.

4.7 Detailed simulation setup

4.7.1 Thermoresponsive Micelles

DPD systems of size 179.4 nm $\times$ 179.4 nm $\times$ 179.4 nm, or $60r_c \times 60r_c \times 60r_c$, were initialized by generating linear $A_{20}B_{20}$ polymers using a random walk algorithm. Each system contains 43,200 polymer beads and 820,800 solvent particles, corresponding to a polymer volume fraction of 5%. A boundary condition that emulates the effect of a semipermeable membrane was applied in all three dimensions: the boundaries are periodic for solvent particles but impenetrable for polymers. An integral conservative force as described in Ref [88] is used to prevent polymers from approaching the boundary while introducing minimal perturbation to particle density near the boundaries.

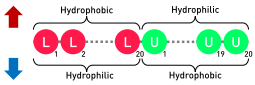
The parameters in Table 4.3 were used for simulating different scenarios of thermally induced self assembly. The simulations were started by first keeping the system temperature at 282K for $15\,000\tau$, or 3.75 ms. The temperature was then

Table 4.3: Summary of the conservative force parameters used throughout the study. In all equations $T_{ij} = (T_i + T_j)/2$ and $T_i \doteq T_i^*/T_0$ is the reduced temperature of particle i.

Radius of Gyration Validation

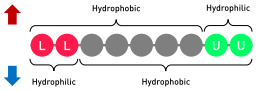
	LCST	**Solvent**
LCST	$18.75\ T_{ij}$	$10.0\ T_{ij} + \frac{27.5}{1+\exp[-150(T_{ij}-1.00)]}$
Solvent	*	$18.75\ T_{ij}$

Thermoresponsive Micelles



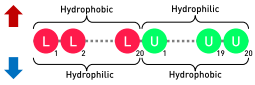
	LCST	**UCST**	**Solvent**	**Tag**
LCST	$18.75\ T_{ij}$	*	*	
UCST	$26.25\ T_{ij}$	$18.75\ T_{ij}$	*	
Solvent	$10.0\ T_{ij} + \frac{27.5}{1+\exp[-150(T_{ij}-0.99)]}$	$10.0\ T_{ij} + \frac{27.5}{1+\exp[150(T_{ij}-1.01)]}$	$18.75\ T_{ij}$	***A***
Solvent	$37.50\ T_{ij}$	$10.0\ T_{ij} + \frac{27.5}{1+\exp[150(T_{ij}-1.00)]}$	$18.75\ T_{ij}$	***B***
Solvent	$10.0\ T_{ij} + \frac{27.5}{1+\exp[-150(T_{ij}-1.00)]}$	$18.75\ T_{ij}$	$18.75\ T_{ij}$	***C***
Solvent	$10.0\ T_{ij} + \frac{27.5}{1+\exp[-150(T_{ij}-1.00)]}$	$37.50\ T_{ij}$	$18.75\ T_{ij}$	***D***
Solvent	$18.75\ T_{ij}$	$10.0\ T_{ij} + \frac{27.5}{1+\exp[150(T_{ij}-1.00)]}$	$18.75\ T_{ij}$	***E***

Thermoresponsive Vesicles



	UCST	**LCST**	**Hydrophobic**	**Solvent**
UCST	$18.75\ T_{ij}$	$26.25\ T_{ij}$	$26.25\ T_{ij}$	$10.0\ T_{ij} + \frac{65.0}{1+\exp[+225(T_{ij}-1.01)]}$
LCST	*	$18.75\ T_{ij}$	$26.25\ T_{ij}$	$10.0\ T_{ij} + \frac{65.0}{1+\exp[-225(T_{ij}-0.99)]}$
Hydrophobic	*	*	$18.75\ T_{ij}$	$37.50\ T_{ij}$
Solvent	*	*	*	$18.75\ T_{ij}$

Carriers in Flow
all inter-polymer $a_{ij} = 26.25\ T_{ij}$



	UCST	**LCST**	**Solvent**	**Wall**
UCST	$18.75\ T_{ij}$	$26.25\ T_{ij}$	$10.0\ T_{ij} + \frac{50.0}{1+\exp[+150(T_{ij}-1.01)]}$	$75.00\ T_{ij}$
LCST	*	$18.75\ T_{ij}$	$10.0\ T_{ij} + \frac{50.0}{1+\exp[-150(T_{ij}-0.99)]}$	$75.00\ T_{ij}$
Solvent	*	*	$18.75\ T_{ij}$	$18.75\ T_{ij}$
Wall	*	*	*	$18.75\ T_{ij}$
	Hydrophilic	**Hydrophobic**	**Solvent**	**Wall**
Hydrophilic	$18.75\ T_{ij}$	$26.25\ T_{ij}$	$10.00\ T_{ij}$	$75.00\ T_{ij}$
Hydrophobic	*	$18.75\ T_{ij}$	$75.00\ T_{ij}$	$75.00\ T_{ij}$
Solvent	*	*	$18.75\ T_{ij}$	$18.75\ T_{ij}$
Wall	*	*	*	$18.75\ T_{ij}$

linearly elevated to 318K in 3.75 ms and then further maintained for another 3.75 ms.

To heat up or cool down the eDPD system, each eDPD particle is coupled with a thermal background of desired temperature $T^B(t)$. This induces a heat flux

$$Q_i^B(t) = \lambda\, C_{\mathrm{v}}\, [T^B(t) - T_i(t)],$$

where λ is a relaxation factor chosen to be 5×10^{-5} in all our simulations. It is shown in Fig. 4.7 that the method can effectively control the system temperature while still preserving a natural (Gaussian) particle temperature distribution.

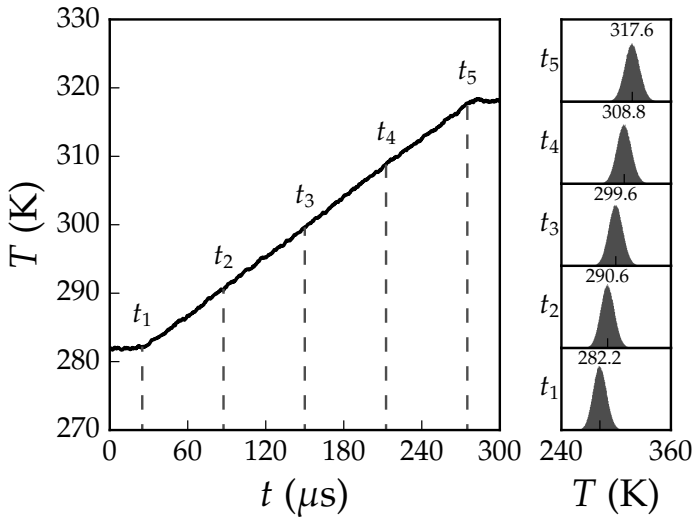


Figure 4.7: Temperature evolution and distribution during a heating process as controlled by a coupled thermal background.

The effect of heating rate on the micelle inversion mechanism was examined by repeating the simulation for case ***A***, but eliminating the linear heating phase of 3.75 ms. Instead, the system temperature was raised almost instantly after t_1.

We further studied the effect of $\Delta\theta \doteq \theta^{\mathrm{U}} - \theta^{\mathrm{L}}$ on the micellar inversion pathway. This is done by simulating case ***A*** from the same initial configuration but varying

θ^{U} and θ^{L} of the blocks while keeping $(\theta^{U} + \theta^{L})/2 \equiv 300K$.

4.7.2 Thermoresponsive Vesicles

To simulate thermoresponsive vesicles and membranes we used an $LCST_2N_5UCST_2$ triblock copolymer, whose conservative force constants within itself and between water are given in Table 4.3. The polymer was chosen based on two criteria: 1) linear symmetry is necessary to ensure that the inverted membrane is structurally similar to the one before inversion; 2) shorter chains are preferred in order to contain an entire vesicle within the finite spatial and temporal extent accessible by our simulation.

Unilamellar vesicles (ULVs) were constructed by randomly inserting 6511 polymers into a spherical shell of outer diameter 112 nm and thickness 7.48 nm, The spherical shell was placed at the center of a simulation box of 179.4 nm$\times$179.4 nm$\times$ 179.4 nm, or $60r_c \times 60r_c \times 60r_c$. Solvent particles were then inserted into the box to obtain an overall number density of 4. Each system was then equilibrated by an isothermal pre-run of 10000 τ, or 2.5 ms.

To quantify the collapse and subsequent release of the containing fluid of vesicles, a thermal loading test was performed where ULVs were subjected to repeated heating-cooling cycles. Let ϕ be the duration of each cycle, $f \doteq 1/\phi$ the loading frequency, and T_{hi}, T_{lo} the highest and lowest temperature, the temperature

of the system is controlled as a function of time t:

$$T(t) = \begin{cases} \mathrm{L}(T_{lo}, T_{hi}, t, 0), & 0 \le t < \frac{1}{4}\phi \\ T_{hi}, & \frac{1}{4}\phi \le t < \frac{1}{2}\phi \\ \mathrm{L}(T_{hi}, T_{lo}, t, \frac{1}{2}\phi), & \frac{1}{2}\phi \le t < \frac{3}{4}\phi \\ T_{lo}, & \frac{3}{4}\phi \le t < \phi \end{cases} \tag{4.7}$$

where $L(T_0, T_1, t, t_0) \doteq T_0 + (T_1 - T_0) \cdot (t - t_0)/\frac{\phi}{4}$ is essentially a linear interpolation from (t_0, T_0) to $(t_0 + \frac{\phi}{4}, T_1)$.

We tested 8 frequencies corresponding to ϕ ranging between 5 μs and 1.25 ms and simulated 72 ensembles for each of the frequency, giving rise to a total of 576 DPD systems each containing 864000 particles simulated for $\sim$ 22 M time steps, or $\sim$ 55 ms in physical time. The computation was done on the Titan supercomputer at Oak Ridge Leadership Computing Facility using 18432 nodes for 24 hours. The time of observed collapse events of the vesicles was used to obtain a maximum likelihood estimate of a right-truncated gamma distribution:

$$p(t; a, b) = \frac{t^{a-1}}{\Gamma(a) b^a} e^{-\frac{t}{b}}$$

with which half-life of the vesicles is then calculated from. However, for intermediate frequencies many vesicles can survive for a long time and the available information appeared insufficient for getting a reliable estimate of a long-tail distribution. In this case only the final survival percentage is given.

4.7.3 Molecular Mechanism of Inversion

To extract patterns from thousands of noisy molecular trajectories, the Proper Orthogonal Decomposition (POD) method was employed [24]. We assembled a N-by-M matrix A, where each row of A represents one molecular trajectory over time. A singular value decomposition of A gives

$$A_{N\times M} = U_{N\times N}\Sigma_{N\times M}V^T_{M\times M} \tag{4.8}$$

where columns of V represent the orthogonal modes, arranged in decreasing magnitude of importance, decomposed from the inversion dynamics. The diagonal elements of Σ, i.e. the singular values of A, quantify the importance of each mode.

Micelle

To reveal the molecular movement patterns of thermoresponsive micelles, we construct periodic simulation boxes of size 59.8 nm $\times$ 59.8 nm $\times$ 59.8 nm, or $20r_c \times 20r_c \times 20r_c$, each containing ~ 100 $LCST_8UCST_8$ polymers placed near the center of the box. Solvent particles were then filled to achieve an overall number density of 4. The system was equilibrated at $T = 291K$ for 100 τ before an instant temperature jump to $309K$ was applied to trigger inversion. The radial component of the center of mass (COM) trajectories of the LCST and UCST block of each individual molecule was collected for POD analysis. We further aggregate data obtained from 100 independent ensembles to improve result accuracy and smoothness.

The matrix A in this case is 10436-by-720:

$$A = \begin{pmatrix} l_0^1 & l_{10}^1 & \cdots & l_{3600}^1 & u_0^1 & \cdots & u_{3600}^1 \\ l_0^2 & l_{10}^2 & \cdots & l_{3600}^2 & u_0^2 & \cdots & u_{3600}^2 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ l_0^{10436} & l_{10}^{10436} & \cdots & l_{3600}^{10436} & u_0^{10436} & \cdots & u_{3600}^{10436} \end{pmatrix} \tag{4.9}$$

where l_j^i and u_j^i are the radial component of the center of mass (COM) of the LCST and UCST blocks of molecule i with respect to the center of mass of the entire micelle at time $j \cdot \delta t \cdot \tau$ after the onset of inversion.

Fig. 4.8A illustrated the morphology of the micelle before and after the inversion. A single dominant mode, which corresponds to local inversion, can be seen in Fig. 4.8B.

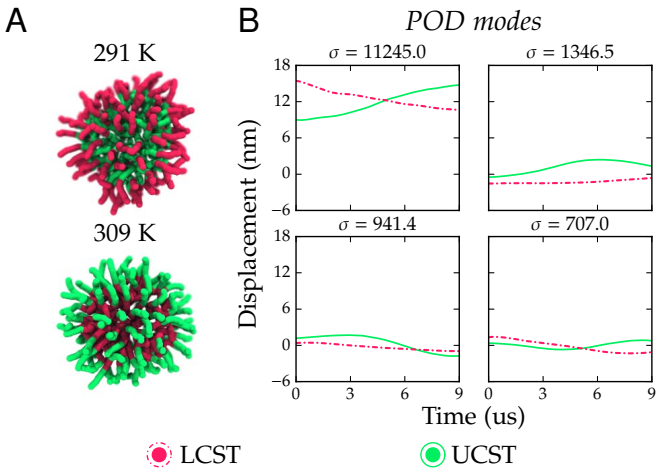


Figure 4.8: Micelles formed by a L_8U_8 thermoresponsive block copolymer invert upon instantaneous heating though local inversion of individual molecules. ***(A)*** an example of the micelle before and after inversion; ***(B)*** POD modes of UCST and LCST block trajectory during inversion; LCST, UCST blocks are colored in red dashed and green solid lines, respectively.

Membrane

To identify movement patterns of individual molecules during the inversion process, a two-dimensional periodic bilayer membranes was used as a zoomed-in model of the vesicle surface. The membrane was initialized by randomly placing 6784 polymer molecules within a xz-parallel slab measuring 11.96 nm thick in a periodic box of 179.4 nm × 179.4 nm × 179.4 nm, or $60r_c \times 60r_c \times 60r_c$, while the rest of the system is filled with solvent particles for an overall number density of 4. The system was first equilibrated at $291K$ for 2.5 ms, or 10000 τ. After that, the system temperature was elevated to $309K$ almost instantaneously to trigger the inversion. The matrix A in this case is 6784-by-720:

$$A = \begin{pmatrix} l_0^1 & l_{10}^1 & \cdots & l_{3600}^1 & u_0^1 & \cdots & u_{3600}^1 \\ l_0^2 & l_{10}^2 & \cdots & l_{3600}^2 & u_0^2 & \cdots & u_{3600}^2 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ l_0^{6874} & l_{10}^{6874} & \cdots & l_{3600}^{6874} & u_0^{6874} & \cdots & u_{3600}^{6874} \end{pmatrix} \tag{4.10}$$

where l_j^i and u_j^i are the y component of the center of mass (COM) of the LCST and UCST blocks of molecule i at time $j \cdot \delta t \cdot \tau$ after the onset of inversion. In other words, the trajectory of the LCST and UCST blocks of each molecule was sampled at a frequency of 4 ps^{-1} and used as one *record*, while POD was used to discover the patterns across the records.

4.7.4 Thermoresponsive Carrier in Flow

The periodic cylindrical tube measures 800 nm ($267.56r_c$) long with a radius of 59.8 nm ($20r_c$). The temperature at the first and second half of the channel wall

along the axial direction was kept at 288 K and 312 K, respectively. The no-slip boundary condition and a Dirichlet boundary condition for temperature were applied using an integral method as described in Ref. [88]. An aggregate, either micelle or vesicle, is centered on the axis, 120 nm from the left end of the tube. In the case of micelles, a sphere of radius 21 nm was randomly filled with ~ 210 $A_{20}B_{20}$ polymers. Three scenarios were simulated, which correspond to micelles formed by a LCST-hydrophilic copolymer ($\text{LCST}_{20}\text{Hydrophilic}_{20}$), a LCST-UCST copolymer ($\text{LCST}_{20}\text{UCST}_{20}$), and a non-responsive amphiphilic copolymer ($\text{Hydrophilic}_{20}\text{Hydrophobic}_{20}$), respectively, using parameters given in Table 4.3. In the case of vesicles, a hollow spherical shell of outer radius 35.88 nm and thickness 11.96 nm was randomly filled with ~ 2700 $LCST_2N_5UCST_2$ polymers. Solvent particles were then inserted to achieve an overall particle number density of 4. A uniform pressure gradient was applied in the positive x direction to drive the flow for a terminal velocity of 2 mm/s. The simulation was performed for 50000 τ, or 12.5 ms for micellar systems and 40000 τ, or 10 ms for vesicle systems. Ensemble averaging was used to obtain smooth statistics of mean temperature, horizontal velocity, and interface interaction energy along the flow direction. The disintegration process of the LCST-hydrophilic copolymer micelle is shown in Fig. 4.9, while more discussion can be found in the main text.

4.8 Sectional Summary

We have performed systematic mesoscopic simulations to elucidate the molecular mechanisms and quantified the dynamics of self-assemblies formed by doubly thermoresponsive block copolymers. We simulated several scenarios of micelle formation and destruction, and examined factors that affect the inversion process

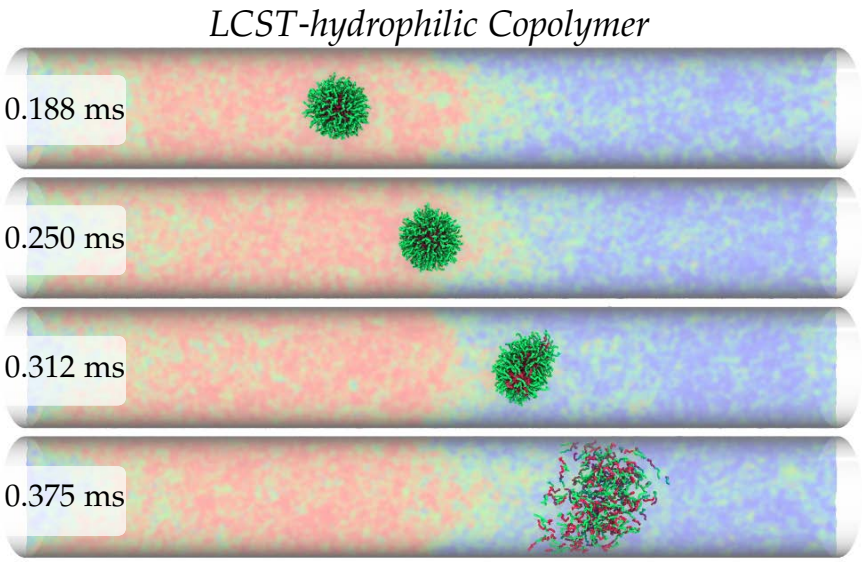


Figure 4.9: A LCST-hydrophilic copolymer micelle disintegrated after crossing the thermal interface and entering a high-temperature zone where the LCST blocks became hydrophilic.

of micelles of dual thermoresponsivity. We discovered two different micelle inversion pathways during slow temperature change depending on the solubility of the polymers at intermediate temperatures. We identified a frequency regime where thermoresponsive unilamellar vesicles can invert reversibly in repeated thermal loading cycles and quantified the failure probability of the vesicles under frequencies that cause instability. We found that thermoresponsive bilayer membranes exhibit a very high ratio of molecular migration between the two leaflets during thermally-induced inversion. Simulations of micelles in a small vessel revealed the unique behavior of thermoresponsive micelles in flow fields with uneven temperature distribution. We discovered some surprising mesoscopic molecular behavior and provided new evidence for theoretical hypotheses. Our work demonstrated the capability and potential of mesoscopic computer simulation in assisting and accelerating the design and optimization of complex self-assembly structures consisting of thermoresponsive polymers.

Chapter Five

Fingerprints for Learning *Ab Initio* Force Fields

5.1 Introduction

Molecular Dynamics (MD) simulations have been widely used for studying atomistic systems, *e.g.* proteins and catalysts, due to its ability to precisely capture transient events and to predict macroscopic properties from microscopic details [169, 93]. In its most prevalent implementation, the trajectory of an atomistic system is integrated in time according to the Newton's law of motion using forces calculated as the negative gradient of a Hamiltonian, whose functional form and parameters are collectively referred to as a force field [123, 29, 158]. Traditionally, the pairwise and many-body terms that comprise a force field are derived by fitting to quantum mechanical calculations and experimental data.

Three properties directly relate to the applicability of a force field: accuracy, transferrability, and complexity [44, 76]. Over the years, a large number of force fields have been developed, each carrying a particular emphasis over these three properties. However, the combinatorial complexity of atomistic systems can easily outpace force field development efforts, the difficulty of which explodes following the curse of dimensionality [26]. A deceptively simple system that can demonstrate the situation is water, a triatomic molecule with a well-characterized molecular structure. In fact, all common water models, such as SPC-E, TIP3P, and TIP4P, have only succeeded in reproducing a small number of structural and dynamical properties of water due to the difficulty in modeling strong intermolecular many-body effects such as hydrogen bonding and polarization [21, 18].

In lieu of a force field, quantum mechanical (QM) calculations can be employed straightforwardly to drive molecular dynamics simulations. The method achieves significantly better accuracy and transferrability by solving for the electronic struc-

ture of the system. However, the computational complexity of QM methods is at least cubic in the number of electrons, and consequently the time and length scales accessible by QM-driven molecular dynamics are severely constrained.

Assuming that there is smoothness in the potential energy surface of the atomistic system, one possible strategy to accelerate QM-driven molecular dynamics is to use QM calculations on only a subset of the time steps, and to interpolate for similar atomic configurations [13, 10, 91, 69]. A schematic overview of the process is given in Figure 5.1, which is enabled by the recent development of high-dimensional nonlinear statistical learning and regression techniques such as Gaussian process regression (GPR) [124] and artificial neural networks [137].

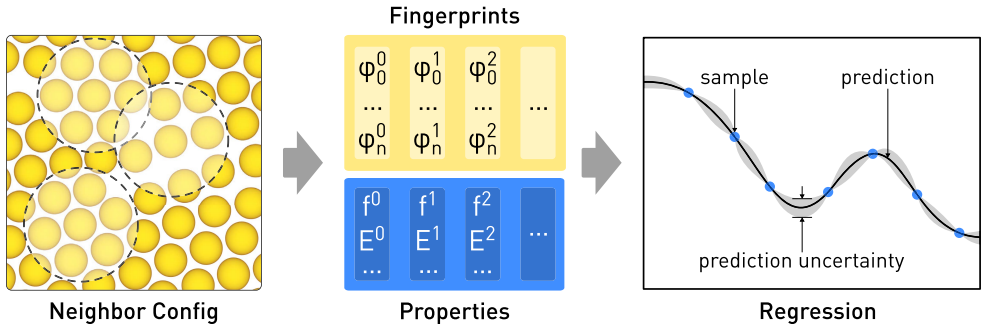


Figure 5.1: In the pipeline of machine learning-driven molecular computations, atomistic neighborhood configurations are transformed into feature vectors, called fingerprint, and used to train non-linear regression models.

This chapter focuses on a particular aspect of the machine-learning-driven molecular computation pipeline, *i.e.* fingerprint algorithms, whose importance arises naturally from the aforementioned regression protocol. A fingerprint is an encoding of an atomistic configuration that can facilitate regression tasks such as similarity comparison across structures consisting of variable numbers of atoms and elements. As has been pointed out previously [10], a good fingerprint should possess the following properties:

1. It can be encoded as a fixed-length vector so as to facilitate regression (particularly for artificial neural networks).

2. It is *complete*, i.e. different atomistic neighborhood configurations lead to different fingerprints and vice versa, and the difference between the fingerprints should be proportional to the intrinsic difference between the atomistic neighborhood configurations.

3. It is *continuous* with regard to atomistic coordinates, and the change in fingerprint should be approximately proportional to the structural variation as characterized by, for example, some internal coordinates.

4. It is *invariant* under permutation, rotation, and translation.

5. It is computationally feasible and straightforward to implement.

Before we proceed to the details of our work, we will first briefly review several fingerprints that are closely related to our work, *i.e.* the SOAP kernel [10], the Coulomb matrix [128], and the GRAPE kernel [43].

Smooth Overlap of Atomic Positions The SOAP kernel is built on the idea of representing atomistic neighborhoods as smoothed density fields using Gaussian kernels each centered at a neighbor atom. Similarity is measured as the inner product between density fields, while rotational invariance is achieved by integrating over all possible 3D rotations, which can be performed analytically using the power spectrum of the density field. In fact, our fingerprint algorithm is inspired by this idea of treating atoms as smoothed density fields. However, we take a different approach to endorse the fingerprint with rotational invariance, and use the Euclidean distance instead of inner product as a distance metric.

Coulomb Matrix The practice of using graphs to represent atomistic neighbor configurations was first implied by the Coulomb matrix, and later further formulated in the GRAPE kernel, where the *diffusion distance* was proposed as a similarity measure between different local chemical environments [141]. The idea is to construct an undirected, unlabeled graph $G = (V, E)$ with atoms acting as the vertices and pairwise interactions setting the weights of the edges. For example, the Coulomb matrix can be treated as a physically-inspired Laplacian matrix [27]

$$\mathbf{M} = \mathbf{D} - \mathbf{A} \tag{5.1}$$

$$\mathbf{D}_{IJ} = \begin{cases} 0.5\, Z_I^{2.4} & \text{if } I = J \\ 0 & \text{if } I \neq J \end{cases} \tag{5.2}$$

$$\mathbf{A}_{IJ} = \begin{cases} 0 & \text{if } I = J \\ \frac{Z_I Z_J}{\|\mathbf{R}_I - \mathbf{R}_I\|} & \text{if } I \neq J \end{cases} \tag{5.3}$$

where the degree matrix $\mathbf{D}$ encodes a polynomial fit of atomic energies to the nuclear charge, while the adjacency matrix $\mathbf{A}$ corresponds to the Coulombic interactions between all pairs of atoms. Due to the use of only relative positions between atoms in the adjacency matrix, the Coulomb matrix is automatically invariant under translation and rotation. However, the matrix itself is not invariant under permutation, as swapping the order of two atoms will result in an exchange of the corresponding columns and the rows. To address this, the sorted list of eigenvalues of the Coulomb matrix can be used instead as a feature vector, while an L^p norm can be used as a distance metric. In practice, due to the fact that the number of neighbor atoms may change, the shorter list is padded with zero in a distance computation.

Graph Approximated Energy The GRAPE kernel evaluates the simultaneous random walks on the direct product of the two graphs representing two atomistic neighborhood configurations. Permutational invariance is achieved by choosing a uniform starting and stopping distribution across nodes of both graphs. However, the cost of distance computation between two graphs scales as $O(N^2)$ with a one-time per-graph diagonalization cost of $O(N^3)$.

In the sections below, I present a new fingerprint algorithm, namely the Density-Encoded Canonically Aligned Fingerprint (DECAF).

5.2 Localized Canonical Coordinate Frame for Rotationally Invariant Description of Atomistic Neighborhood

5.2.1 Kernel Minisum Approach

To improve model generalization while minimizing data redundancy, a fingerprint algorithm should be able to recognize atomistic structures that differ only by a permutation of atoms within the same element or a rigid-body transformation, and extract feature vectors invariant under these transformations. As summarized in Table 5.1, a variety of strategies have been successfully employed by common fingerprint algorithms to achieve rotational invariance.

However, these approaches do not provide a means for the acquisition of vector-valued quantities in a rotational invariant form. One approach is to only

acquire and interpolate the potential energy, a scalar quantity, and then take the derivative of the energy interpolation. This approach, however, further triggers the need for methods to decompose the total energy among the atoms because it is a property of the entire system rather than individual atoms [11].

Another approach proposed by Li *et al.* [91, 19] is to project vector quantities onto a potentially overcomplete set of non-orthogonal basis vectors obtained from a weighted sum of the atomic coordinate vectors:

$$\mathbf{V}_k = \sum_i \mathbf{x}_i \, \exp\left[-\left(\frac{||\mathbf{x}_i||}{R_\mathrm{c}}\right)^{p_k}\right]. \tag{5.4}$$

However, the approach may suffer from robustness issues. For example, all of the $\mathbf{V}_k$ generated with different p_k will coincide if the radial distance of the atoms are all equal. As a second example, the configuration with 4 atoms at $(r\cos\varepsilon, r\sin\varepsilon), (0, r), (-r, 0), (0, -r)$ yields

$$\mathbf{V}_k = c \cdot [(r\cos\varepsilon, r\sin\varepsilon) + (0, r) + (-r, 0) + (0, -r)] \tag{5.5}$$

$$= c \cdot r \cdot (1 - \cos\varepsilon, \sin\varepsilon). \tag{5.6}$$

Thus, if ε gets close to zero, V_k will always point toward either $(0, 1)$ or $(0, -1)$, even if the vector quantity of interest may point in other directions.

Table 5.1: Comparison of strategies used by fingerprint algorithms to obtain feature vectors which are invariant under translation, permutation, and rotation.

Fingerprint	**Invariance**		
	Translation	**Permutation**	**Rotation**
Coulomb	relative distance	sorting eigenvalues	all vs. all graph
Behler	relative distance	summation	ignoring angular information
SOAP[10]	relative distance	summation	integrating over all rotations
GRAPE	relative distance	uniform distribution	uniform distribution

Here, we present a robust kernel PCA-inspired algorithm for the explicit determination of a canonical coordinate frame, within which the projection of the atomistic neighborhood is invariant under rigid-body rotation. Furthermore, the canonical coordinate frame can be directly used to capture vector-valued quantities in a rotational-invariant form. Given N atoms with position $\mathbf{x}_1, \ldots, \mathbf{x}_N \in \mathbb{R}^d$, we first formulate the L_p PCA algorithm as an optimization problem where we seek a unit vector $\mathbf{w}^*$ that maximizes the sum of projection of the data:

$$\mathbf{w}^* = \text{argmax}_{\|\mathbf{w}\|=1} \sum_{i=1}^{N} \left|\mathbf{w}^\mathsf{T}\mathbf{x}_i\right|^p . \tag{5.7}$$

$$= \text{argmax}_{\|\mathbf{w}\|=1} \sum_{i=1}^{N} |r_i|^p \left|\mathbf{w}^\mathsf{T}\mathbf{e}_i\right|^p . \tag{5.8}$$

where $r_i = \|\mathbf{x}_i\|$ is the distance from the origin to atom i, $\mathbf{e}_i = \mathbf{x}_i/r_i$ is the unit vector pointing toward atom i, respectively. The optimization process can only uniquely determine the orientation of a projection vector up to a line, because $\left|\mathbf{w}^\mathsf{T}\mathbf{e}\right| \equiv \left|-\mathbf{w}^\mathsf{T}\mathbf{e}\right|$. As a consequence, further heuristics are needed to identify a specific direction for the PCA vectors.

To overcome this difficulty, we generalize the $|r_i|^p$ term into a weight function $g(r_i)$, and the $\left|\mathbf{w}^\mathsf{T}\mathbf{e}_i\right|^p$ term into a bivariate kernel function $\kappa(\mathbf{w}, \mathbf{e}_i)$. We then attempt to seek a unit vector $\mathbf{w}^*$ that minimizes the kernel summation:

$$\mathbf{w}^* = \text{argmin}_{\|\mathbf{w}\|=1} \sum_{i=1}^{N} g(r_i)\, \kappa(\mathbf{w}, \mathbf{e}_i). \tag{5.9}$$

In particular, we have found a square angle (SA) kernel and an exponentiated

cosine (EC) kernel as two robust options:

$$\kappa_{\mathrm{SA}}(\mathbf{w}, \mathbf{e}) \doteq \frac{1}{2} \arccos^2(\mathbf{w}^{\mathsf{T}}\mathbf{e}), \tag{5.10}$$

$$\kappa_{\mathrm{EC}}(\mathbf{w}, \mathbf{e}) \doteq \exp\left(-\mathbf{w}^{\mathsf{T}}\mathbf{e}\right). \tag{5.11}$$

As shown in Figure 5.2, both kernels are minimal when $\mathbf{w}$ and $\mathbf{e}$ are parallel, and monotonically reach maximum when the two vectors are antiparallel. Intuitively, optimizing the minisum objective function generated by the SA kernel will yield a vector that, loosely speaking, bisects the sector between the atoms. The EC kernel exhibits very similar behavior but leads to a smoother objective function. As shown in Figure 5.2, this allows for the determination of a projection vector without ambiguity, even if the atom configuration contains perfect symmetry.

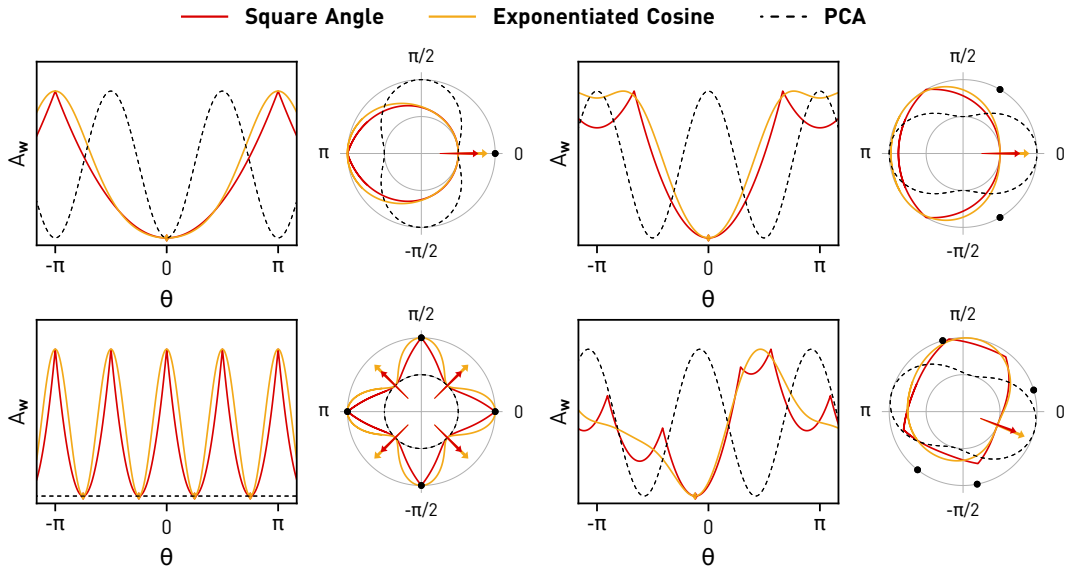


Figure 5.2: Shown here is an illustration of the minisum algorithm that determines projection vectors for rotational invariant description of atomistic neighbor configurations. Black dots represent atoms which all carry equal importance. The vectors that point from the origin to the atoms are used as the input to bivariate kernels to compute the minisum objective function, which are drawn in solid lines. For reference, the negated values of the PCA objective function are drawn in dashed lines. Projection vectors are obtained by finding the unit vector $\mathbf{w}^*$ that minimizes the objective function.

5.2.2 Solving the Kernel Minisum Optimization Problems

The optimization problem can be solved very efficiently using a gradient descent algorithm as detailed below.

Square Angle The objective function of the minisum problem using the square angle (SA) kernel is

$$A_{\mathrm{SA}}(\mathbf{w}) \doteq \frac{1}{2}\sum_{i=1}^{N} g(r_i)\,\arccos^2(\mathbf{w}^\mathsf{T}\mathbf{e}_i). \tag{5.12}$$

The gradient of A_{SA} with respect to $\mathbf{w}$ is

$$\nabla_{\mathbf{w}} A_{\mathrm{SA}} = \sum_{i=1}^{N} -g(r_i)\,\frac{\arccos(\mathbf{w}^\mathsf{T}\mathbf{e}_i)}{\sqrt{1-(\mathbf{w}^\mathsf{T}\mathbf{e}_i)^2}}\,\mathbf{e}_i. \tag{5.13}$$

Note that $\frac{\arccos(\mathbf{w}^\mathsf{T}\mathbf{e}_i)}{\sqrt{1-(\mathbf{w}^\mathsf{T}\mathbf{e}_i)^2}}$ is singular when $\mathbf{w} \parallel \mathbf{e}_i$. This can be treated numerically by replacing the removable singularities at $\mathbf{w}^\mathsf{T}\mathbf{e_i} = 1$ with the left-limit $\lim_{\mathbf{w}^\mathsf{T}\mathbf{e}_i \to 1^-} \frac{\arccos(\mathbf{w}^\mathsf{T}\mathbf{e}_i)}{\sqrt{1-(\mathbf{w}^\mathsf{T}\mathbf{e}_i)^2}} = 1$, while truncating the gradient near the poles at $\mathbf{w}^\mathsf{T}\mathbf{e_i} = -1$.

A local minimum can be iteratively searched for with gradient descent while renormalizing $\mathbf{w}$ after each iteration. Moreover, due to the locally quadratic nature of the objective function, we have found that the Barzilai-Borwein algorithm [12] can significantly accelerate the convergence at a minimal cost. The algorithm is presented in Alg. 14.

Algorithm 14 Gradient descent for solving the square angle minisum problem.

function MinSquareAngle($\mathbf{E} = [\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_N]$, $\mathbf{G} = [g_1, g_2, \ldots, g_N]$, $\mathbf{w}_0$)
  $\mathbf{w} \leftarrow \mathbf{w}_0$
  repeat
    Compute gradient $\nabla_{\mathbf{w}}$ using Eq. 5.13.
    Obtain tangential component of gradient $\nabla^{\perp}_{\mathbf{w}} \leftarrow (\mathbf{I} - \mathbf{w}\mathbf{w}^{\mathsf{T}})\nabla_{\mathbf{w}}$
    if at step 0 **then**
      $\alpha \leftarrow 0.01$ ▷ **Small initial step size for bootstrapping**
    else
      $\alpha \leftarrow \frac{(\mathbf{w}-\mathbf{w}_{-1})^{\mathsf{T}}(\nabla^{\perp}_{\mathbf{w}}-\nabla^{\perp}_{\mathbf{w}_{-1}})}{\|\nabla^{\perp}_{\mathbf{w}}-\nabla^{\perp}_{\mathbf{w}_{-1}}\|^2}$ ▷ **Adaptive subsequent steps by Barzilai-Borwein**
    end if
    Save $\mathbf{w}$ as $\mathbf{w}_{-1}$, $\nabla^{\perp}_{\mathbf{w}}$ as $\nabla^{\perp}_{\mathbf{w}_{-1}}$
    Update $\mathbf{w} \leftarrow \mathbf{w} - \alpha\,\nabla^{\perp}_{\mathbf{w}}$ and normalize to unit length
  until $\|\mathbf{w} - \mathbf{w}_{-1}\| < \varepsilon$
  return $\mathbf{w}$
end function

Exponentiated Cosine The objective function of the minisum problem using the exponentiated cosine (EC) kernel is:

$$A_{\mathrm{EC}}(\mathbf{w}) \doteq \sum_{i=1}^{N} g(r_i)\, \exp\left(-\mathbf{w}^{\mathsf{T}}\mathbf{e}_i\right). \tag{5.14}$$

The gradient of A_{EC} with respect to $\mathbf{w}$ is

$$\nabla_{\mathbf{w}} A_{\mathrm{EC}} = \sum_{i=1}^{N} -g(r_i)\, \exp\left(-\mathbf{w}^{\mathsf{T}}\mathbf{e}_i\right)\mathbf{e}_i. \tag{5.15}$$

The gradient contains no singularity. However, it is not always locally quadratic. This can cause the Barzilai-Borwein algorithm to generate negative step sizes and consequently divert the search towards a local maximum. Luckily, this can be easily overcome by using the absolute value of the step size as calculated by the Barzilai-Borwein algorithm to prevent the minimization algorithm from going uphill. The complete algorithm is given in Alg. 15.

As shown in Table 5.2, both Alg. 14 and Alg. 15 converge quickly and con-

Algorithm 15 Gradient descent for solving the exponentiated cosine minisum problem.

function MinExpCosine($\mathbf{E} = [\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_N]$, $\mathbf{G} = [g_1, g_2, \ldots, g_N]$, $\mathbf{w}_0$)
  $\mathbf{w} \leftarrow \mathbf{w}_0$
  repeat
    Compute gradient $\nabla_{\mathbf{w}}$ using Eq. 5.15.
    Obtain tangential component of gradient $\nabla^{\perp}_{\mathbf{w}} \leftarrow (\mathbf{I} - \mathbf{w}\mathbf{w}^{\mathsf{T}})\nabla_{\mathbf{w}}$
    if at step 0 **then**
      $\alpha \leftarrow 0.01$ ▷ **Small initial step size for bootstrapping**
    else
      $\alpha \leftarrow \left| \frac{(\mathbf{w}-\mathbf{w}_{-1})^{\mathsf{T}}(\nabla^{\perp}_{\mathbf{w}}-\nabla^{\perp}_{\mathbf{w}_{-1}})}{\|\nabla^{\perp}_{\mathbf{w}}-\nabla^{\perp}_{\mathbf{w}_{-1}}\|^2} \right|$ ▷ **Adaptive subsequent steps by Barzilai-Borwein**
    end if
    Save $\mathbf{w}$ as $\mathbf{w}_{-1}$, $\nabla^{\perp}_{\mathbf{w}}$ as $\nabla^{\perp}_{\mathbf{w}_{-1}}$
    Update $\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla^{\perp}_{\mathbf{w}}$ and normalize to unit length
  until $\|\mathbf{w} - \mathbf{w}_{-1}\| < \varepsilon$
  return $\mathbf{w}$
end function

sistently handling a wide range of representative point configurations commonly found in molecular systems. However, the gradient descent method can only find local optima. Thus, multiple trials should be performed using different initial guesses to ensure that a global minimum can be located.

5.2.3 Complete Set of Orthogonal Projection Vectors as A Canonical Coordinate Frame

In 3D, a complete set of orthogonal bases can be found greedily using the protocol as described in Alg. 16. Specifically, we use the globally optimal solution of the minisum optimization problem as the first basis $\mathbf{b}_{\alpha}$, and the constrained optimal solution in a plane orthogonal to $\mathbf{b}_{\alpha}$ as the second basis $\mathbf{b}_{\beta}$. Special care must be taken for determining the third basis $\mathbf{b}_{\gamma}$, as the only degree of freedom now is its sign due to the orthogonality constraint. The straightforward approach of

Table 5.2: List here is the number of iterations and initial guesses used by the gradient descent algorithm to find an local optimum solution of the kernel minisum problems. The numbers are averaged over 500 repetitions, and the convergence criterion is 10^{-14}. In cases where the iterative algorithm does not converge within 64 iterations, the optimization will be restarted with a new guess.

Kernel	**Square Angle**		**Exponentiated Cosine**	
Iterations	Itrs./Guess	Guesses	Itrs./Guess	Guesses
Single point	8.8	1	7.9	1
Two point, angle $< \pi/2$	7.1	1	7.7	1
Two point, angle $\geq \pi/2$	6.1	1	6.3	1
Two point, angle $= \pi$	6.0	1	5.6	1
Planar C_3	6.2	1	6.3	1
Planar C_4	5.6	1	6.9	1
Tetrahedra	10.6	1	7.7	1
Octahedra	11.7	1	9.4	1
Improper S_4	17.9	1	23.1	1
Improper S_6	14.7	1	18.0	1
2D random 10 points	7.2	1.1	8.9	1
3D random 50 points	16.9	1.2	14.4	1

choosing the direction that gives the smaller objective function value may fail, for example, when the system contains improper rotational symmetry. In that case, $\mathbf{b}_\alpha$ and $\mathbf{b}_\beta$ are interchangeable and both perpendicular to the rotation-reflection axis. As a result, the two candidates of $\mathbf{b}_\gamma$ will both align with the rotation-reflection axis and are thus indistinguishable by kernel minisum. However, the projection of the atoms into the two seemingly equivalent coordinate frames are not identical, but rather mirror images of each other. Fortunately, this can be addressed by choosing the direction of the half-space, as created by the plane $\mathbf{b}_\alpha$-$\mathbf{b}_\alpha$, that yields the smaller kernel objective function between the *bisector* of $\mathbf{b}_\alpha$ and $\mathbf{b}_\beta$ versus the points lies in that half-space. This rule can also handle general situations with/without symmetry.

Algorithm 16 The procedure for determining a canonical coordinate frame using kernel minisum.

function GETCANONICALPROJECTION3D($\mathbf{E} = [\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_N]$, $\mathbf{G} = [g_1, g_2, \ldots, g_N]$)
  $\mathbf{b}_\alpha \leftarrow$ global minimum of MINISUM($\mathbf{E}$, $\mathbf{G}$) using multiple runs of Alg. 14 or Alg. 15
  if $\mathbf{E}$ contains only 1 point **then**
    construct arbitrary $\mathbf{b}_\beta \perp \mathbf{b}_\alpha$
    $\mathbf{b}_\gamma \leftarrow \mathbf{b}_\alpha \times \mathbf{b}_\beta$
  else
    $\mathbf{b}_\beta \leftarrow$ global minimum of MINISUM($\mathbf{E}$, $\mathbf{G}$) using multiple runs of Alg. 14 or Alg. 15, subjecting to the constraint that the probe vector and the gradient are all $\perp \mathbf{b}_\alpha$
    $\mathbf{d} \leftarrow (\mathbf{b}_\alpha + \mathbf{b}_\beta)/\sqrt{2}$
    if $\sum_{\forall \mathbf{e}_i;\ \mathbf{e}_i^\mathsf{T}(\mathbf{b}_\alpha \times \mathbf{b}_\beta) \geq 0} g_i\, \kappa(\mathbf{d}, \mathbf{e}_i) \leq \sum_{\forall \mathbf{e}_i;\ \mathbf{e}_i^\mathsf{T}(\mathbf{b}_\alpha \times \mathbf{b}_\beta) < 0} g_i\, \kappa(\mathbf{d}, \mathbf{e}_i)$ **then**
      $\mathbf{b}_\gamma \leftarrow \mathbf{b}_\alpha \times \mathbf{b}_\beta$
    else
      $\mathbf{b}_\gamma \leftarrow -\mathbf{b}_\alpha \times \mathbf{b}_\beta$
    end if
  end if
  return $\mathbf{b}_\alpha$, $\mathbf{b}_\beta$, $\mathbf{b}_\gamma$
end function

5.3 Density-Encoded Canonically Aligned Fingerprint

5.3.1 Density Field and Approximation of Volume Integral

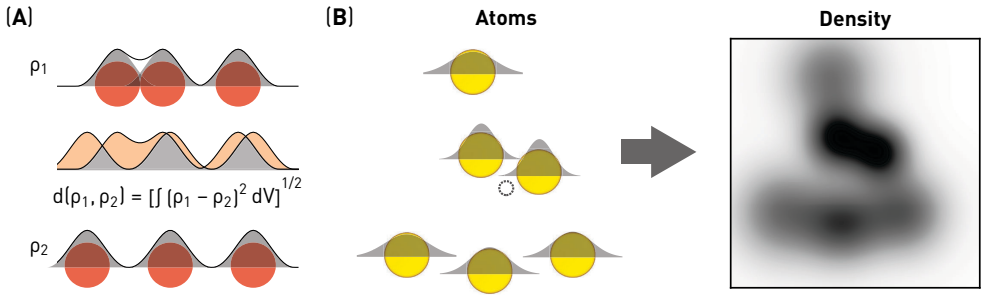


Figure 5.3: **(A)** Two 1D density profiles, ρ_1 and ρ_2, are generated from two different atomistic configurations using atom-centered smoothing kernel functions. The 'distance' between them is measured as the L_2 norm of their difference, which corresponds to the highlighted area in the middle plot. **(B)** Shown here is a 2D density field using smoothing kernels whose widths depend on the distances of the atoms from the origin.

The local density field $\rho_\mathbf{s}(\mathbf{r})$ around a point $\mathbf{s}$ is formulated as a superimposition

of smoothing kernel functions each centered at a neighbor atom $i = 1, 2, \ldots, N$ with relative displacement $\mathbf{x}_i$ with regard to $\mathbf{s}$ and within a cutoff distance R_c:

$$\rho_{\mathbf{s}}(\mathbf{r}) = \sum_{i, \|\mathbf{x}_i - \mathbf{s}\| < R_\mathrm{c}}^{N} W_c(\mathbf{x}_i - \mathbf{s})\, W_d(\mathbf{x}_i, \mathbf{r}) \tag{5.16}$$

This density field, as has been pointed out previously [10], may be used as a fingerprint of the local atomistic environment. To achieve rotational invariance, we transform the atom coordinates into the canonical coordinate frame $\mathbf{R} \doteq [\mathbf{b}_\alpha, \mathbf{b}_\beta, \mathbf{b}_\gamma]$ as determined by the kernel minisum algorithm:

$$\boxed{\rho_{\mathbf{s}}(\mathbf{r}) = \sum_{i, \|\mathbf{x}_i - \mathbf{s}\| < R_\mathrm{c}}^{N} W_c(\mathbf{R}^\mathsf{T}\mathbf{x}_i - \mathbf{s})\, W_d(\mathbf{R}^\mathsf{T}\mathbf{x}_i, \mathbf{r}).} \tag{5.17}$$

Depending on the specific application, $\mathbf{s}$ may not necessarily overlap with any of the $\mathbf{x}_i$. Here, we assume that the smoothing kernel W_d takes the form of a stationary Gaussian $\sigma^{-1} \exp\left[-\|\mathbf{r}\|^2/2\sigma^2\right]$, but will later discuss about other possibilities in Section 5.3.2. The density scaling function W_c ensures the continuity of the density field when atoms enter or exit the cutoff distance, and will also be further discussed in a later section. Scalar properties can be acquired directly from the target atom, while vector-valued properties, such as force, can be acquired and interpolated in the local orthogonal coordinates as $\tilde{\mathbf{y}} = \mathbf{R}^\mathsf{T}\mathbf{y}$.

We define the distance between two density fields ρ_i and ρ_j by a weighted L_2 norm of their difference:

$$\Delta\rho_{\mathbf{s}_1\mathbf{s}_2}(\mathbf{r}) \doteq \rho_{\mathbf{s}_1}(\mathbf{r}) - \rho_{\mathbf{s}_2}(\mathbf{r}), \tag{5.18}$$

$$\mathrm{d}(\rho_{\mathbf{s}_1}, \rho_{\mathbf{s}_2}) \doteq \left(\int_{\mathbb{R}^3} w(\mathbf{r}) \left| \Delta\rho_{\mathbf{s}_1\mathbf{s}_2}(\mathbf{r}) \right|^2 \mathrm{d}V(\mathbf{r}) \right)^{1/2}. \tag{5.19}$$

The weight function $w(\mathbf{r})$ provides additional flexibility for emphasizing particular regions of the atomistic neighborhood, and can strongly facilitate the task of fitting fast varying properties such as the repulsive part of the Lennard-Jones potential.

To efficiently compute the integral, we would like to evaluate the density field using an optimal quadrature rule. To determine the locations and weights of the quadrature nodes, we decompose the volume integral in Eq. 5.19 into a surface integral over spherical shells and a 1D integral along the radial direction:

$$\int_{\mathbb{R}^3} w(\mathbf{r}) \left|\Delta\rho_{\mathbf{s}_1\mathbf{s}_2}(\mathbf{r})\right|^2 \mathrm{d}V(\mathbf{r}) = \int_{r=0}^{\infty} \left(\int_{\varphi=0}^{2\pi} \int_{\theta=0}^{\pi} w(r,\theta,\varphi) \left|\Delta\rho_{\mathbf{s}_1\mathbf{s}_2}(r,\theta,\varphi)\right|^2 \sin\theta \,\mathrm{d}\theta \,\mathrm{d}\varphi \right) r^2 \,\mathrm{d}r \tag{5.20}$$

The surface integral can be optimally approximated using the Lebedev quadrature rule [77]:

$$\Delta P_{\mathbf{s}_1\mathbf{s}_2}(r) \doteq \int_{\varphi=0}^{2\pi} \int_{\theta=0}^{\pi} w(r,\theta,\varphi) \left|\Delta\rho_{\mathbf{s}_1\mathbf{s}_2}(r,\theta,\varphi)\right|^2 \sin\theta \,\mathrm{d}\theta \,\mathrm{d}\varphi \approx 4\pi \sum_{m=1}^{N_\mathrm{a}(r)} w(r,\mathbf{q}_m)\, \beta_m \left|\Delta\rho_{\mathbf{s}_1\mathbf{s}_2}(r\cdot\mathbf{q}_m)\right|^2, \tag{5.21}$$

where β_m, $\mathbf{q}_m \doteq (x_m, y_m, z_m)$, and N_a are the weights, positional unit vectors, and number of the Lebedev nodes, respectively. The radial integral fits well into the generalized Laguerre-Gauss quadrature formula with weight function $r^2\, e^{-r}$ [120]:

$$\int_0^{\infty} \Delta P_{\mathbf{s}_1\mathbf{s}_2}(r)\, r^2 \,\mathrm{d}r \approx \sum_{n=1}^{N_\mathrm{r}} \alpha_n\, e^{r_n}\, \Delta P_{\mathbf{s}_1\mathbf{s}_2}(r_n), \tag{5.22}$$

where α_n, r_n, and N_r are the weights, coordinates, and number of the Laguerre nodes, respectively. Combining Eq. 5.20–5.22, a composite quadrature rule can be generated consisting of several spherical layers of nodes. As shown in Figure 5.4, the radial coordinates of the quadrature nodes are determined by the Laguerre

quadrature nodes, while the angular positions are determined by the Lebedev quadrature nodes, respectively:

$$\int_{\mathbb{R}^3} w(\mathbf{r}) \left|\Delta\rho_{\mathbf{s}_1\mathbf{s}_2}(\mathbf{r})\right|^2 \mathrm{d}V(\mathbf{r}) \approx 4\pi \sum_{n=1}^{N_\mathrm{r}} \sum_{m=1}^{N_\mathrm{a}(n)} \alpha_n \beta_m w_{nm} e^{r_n} \left|\Delta\rho_{\mathbf{s}_1\mathbf{s}_2}(r_n \cdot \mathbf{q}_m)\right|^2$$
$$= \sum_{k=1}^{N} w_k \left|\rho_i(\mathbf{r}_k) - \rho_j(\mathbf{r}_k)\right|^2 . \tag{5.23}$$

Here $k = (n, m); 1 \leq n \leq N_\mathrm{r}, 1 \leq m \leq N_\mathrm{a}(n)$ is the linearized index of the k-th quadrature node located at $\mathbf{r}_k = r_n \cdot \mathbf{q}_m$ with weight $w_k = 4\pi\, \alpha_n\, \beta_m\, w(r_n \cdot \mathbf{q}_m)\, e^{r_n}$. A discretized version of Eq. 5.19 that computes the distance between the fingerprints is given by:

$$\mathrm{d}(\rho_{\mathbf{s}_1}, \rho_{\mathbf{s}_2}) \approx \left[\sum_{k=1}^{N} w_k \left|\rho_{\mathbf{s}_1}(\mathbf{r}_k) - \rho_{\mathbf{s}_2}(\mathbf{r}_k)\right|^2\right]^{1/2} . \tag{5.24}$$

In addition, the quadrature nodes could be radially scaled such that the outer most nodes lie at a radius R^* within the cutoff distance R_c. This allows us to pair a Laguerre quadrature of any order with an arbitrary cutoff distance. The scaled quadrature rule is given by:

$$\tau = R^* \,/\, \max_n (r_\mathrm{n}) , \tag{5.25}$$

$$\mathrm{d}^*(\rho_{\mathbf{s}_1}, \rho_{\mathbf{s}_2}) \approx \boxed{\left[\sum_{k=1}^{N} \tau^3 w_k \left|\rho_{\mathbf{s}_1}(\tau\, \mathbf{r}_k) - \rho_{\mathbf{s}_2}(\tau\, \mathbf{r}_k)\right|^2\right]^{1/2}} . \tag{5.26}$$

Since the scaling is constant among all nodes, it can be safely omitted in many regression tasks where only the relative difference between the fingerprints are of significance. A list of Laguerre and Lebedev quadrature nodes and weights are given in the Appendix.

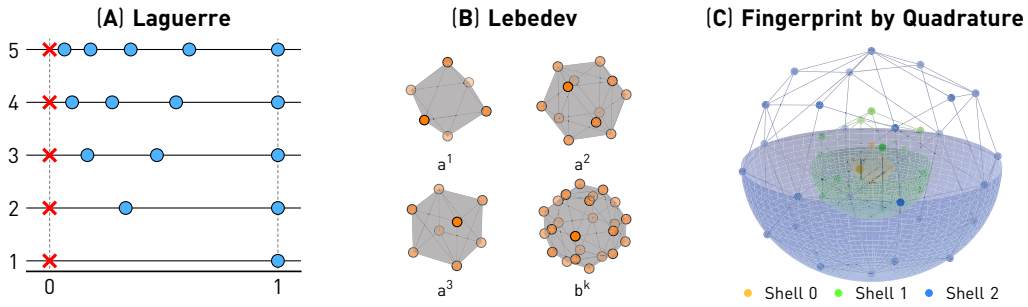


Figure 5.4: (A) Laguerre quadrature nodes with order 1 to 5, normalized by the reciprocal of the largest node onto the unit interval. **(B)** The a^1, a^2, a^3, and b^k class of Lebedev grid points on a unit sphere. **(C)** The DECAF molecular fingerprint essentially comprises of a grid of quadrature nodes that optimally samples the density field induced by the neighbor atoms. Shown here is an example of one such composite quadrature grid which combines a 3-node Laguerre quadrature rule with three layers of Lebedev quadrature nodes of 3rd, 5th, and 7th order, respectively.

5.3.2 Radial Weight Functions

In this section, we discuss about two types of weight functions that can be used to fine-tune the density field fingerprint: the density scaling function W_c as in Eq. 5.17 and the weight of integral $w(r)$ as in Eq. 5.19.

Driven by the interest of reducing computational cost, we would like to use a cutoff distance to select atoms involved in constructing the density field. However, it is important to ensure that atoms will enter and exit the neighborhood smoothly. This naturally requests that the contribution of an atom to be zero outside of the cutoff, and to increase continuously and smoothly when the atom approaching entrance. Correspondingly, W_d should: 1) become unity at the origin; 2) smoothly approach zero at the cutoff distance; and 3) be twice-differentiable to ensure the continuity of the fingerprint-based regression. Candidates satisfying the above conditions include, for example, a tent-like kernel

$$W_d(\mathbf{r}) = (1 - \|r\|/R_c)^t, \; t > 2 \tag{5.27}$$

and a bell-shaped polynomial kernel with compact support

$$W_d(\mathbf{r}) = \frac{-b\,(1 - \|r\|/R_\mathrm{c})^a + a\,(1 - \|r\|/R_\mathrm{c})^b}{a - b},\ a > b > 2 \tag{5.28}$$

as detailed in Appendix.

The approximation of the radial integral using a Laguerre quadrature requires that the integrand, *i.e.* the difference between the atomistic density fields, decays sufficiently fast beyond the outermost quadrature nodes in order to achieve acceptable convergence. In addition, the steeply repulsive yet flat attractive interatomic short-range interactions prompt that the sensitivity of fingerprint be adjusted correspondingly in order to avoid numerical difficulties in training a regression model. The weight of the integral, $w(r)$, provides a convenient means for achieving the purpose. Different from W_d, $w(r)$ should instead satisfy the following conditions: 1) is normalized such that $\int w(r)\mathrm{d}V(\mathbf{r}) = 1$; 2) decays sufficiently fast, but not necessarily to 0, beyond the outer most quadrature node; and 3) be sufficiently smooth beyond the outermost quadrature node. Candidates for $w(r)$ includes the tent-like kernel and the bell-shaped kernel for W_d, albeit with a different normalization factor. The Laplacian kernel

$$w(r) = \exp(-|r|/l)/(8\pi l^3),\ l > 0 \tag{5.29}$$

with a sufficiently large length scale l also appears to be a candidate due to its similarity with e^{-r} part of the weight function of the Laguerre quadrature. Note that the constant kernel

$$w(r) = 3/(4\pi R_\mathrm{c}^3) \tag{5.30}$$

may also be a choice as long as the density field already decays fast enough due to the density scaling function W_d.

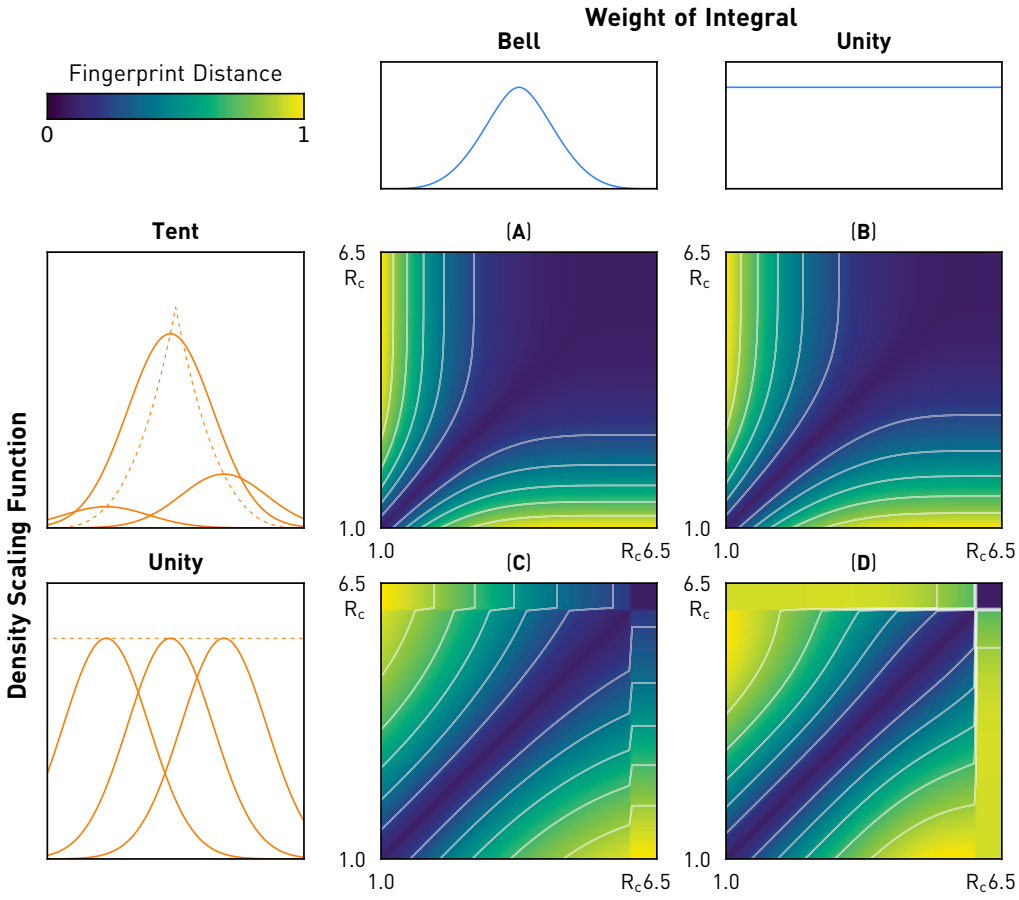


Figure 5.5: Shown here are examples of the distance matrices between fingerprints sampled from a biatomic system. As manifested by the difference between **(A)** against **(B)**, a bell-shaped weight of integral helps to emphasize the near field. Meanwhile, the obvious discontinuities in the second row of matrices demonstrate the importance of the density scaling function when the fingerprint algorithm uses only atoms within a finite cutoff distance.

In Figure 5.5, we demonstrate the effect of the density scaling function and the weight of integral on the distance matrices between fingerprint obtained from a pair of atoms. The comparison between panel A and B shows that a bell-shaped integration weight allows the distance between fingerprints to change more rapidly when the atoms are closer but more slowly when the atoms are farther apart. The visible discontinuity in the second row clearly demonstrates the importance of the damping function when only atoms within a finite cutoff

distance are used to compute the fingerprint.

We further examine the impact of the weight functions on the performance of Gaussian process regression (GPR) using the fingerprint of the interatomic force of a minimal system containing two nitrogen atoms. Despite the simplicity of the system, this case is of fundamental importance because of its ubiquity, and because the fast-growing repulsive regime of the Lennard-Jones potential could cause difficulty as a small change in the system configuration can trigger large changes in the regression target function. In Figure 5.6, we compare the performance among the combination of four weights of the integral and two density scaling functions. The initial training set consists of two samples collected at $r_{\text{N-N}} = 1.0$ and 6.0. The regression is then refined using a greedy strategy that consecutively learns the point with the largest posterior variance until the largest uncertainty, defined as twice the posterior standard deviation, is less than 0.1 eV/Å. Thanks to the active learning scheme, all combinations of the weight functions are able to deliver a good fit of the target function after refinement. However, the number of points required and the achieved accuracy vary. Therefore, it is important to evaluate and choose the weight functions in the context of specific application scenarios.

5.3.3 Quadrature Resolution

Despite the formal convergence of the composite quadrature in DECAF, a cost of $O(NM)$ distance calculations and kernel evaluations are needed to sample a density field generated by N atoms using M quadrature nodes. [1] Thus, in practice

[1] A L^2 distance calculation using the extracted feature vectors comes at a cost of $O(M)$ floating point operations.

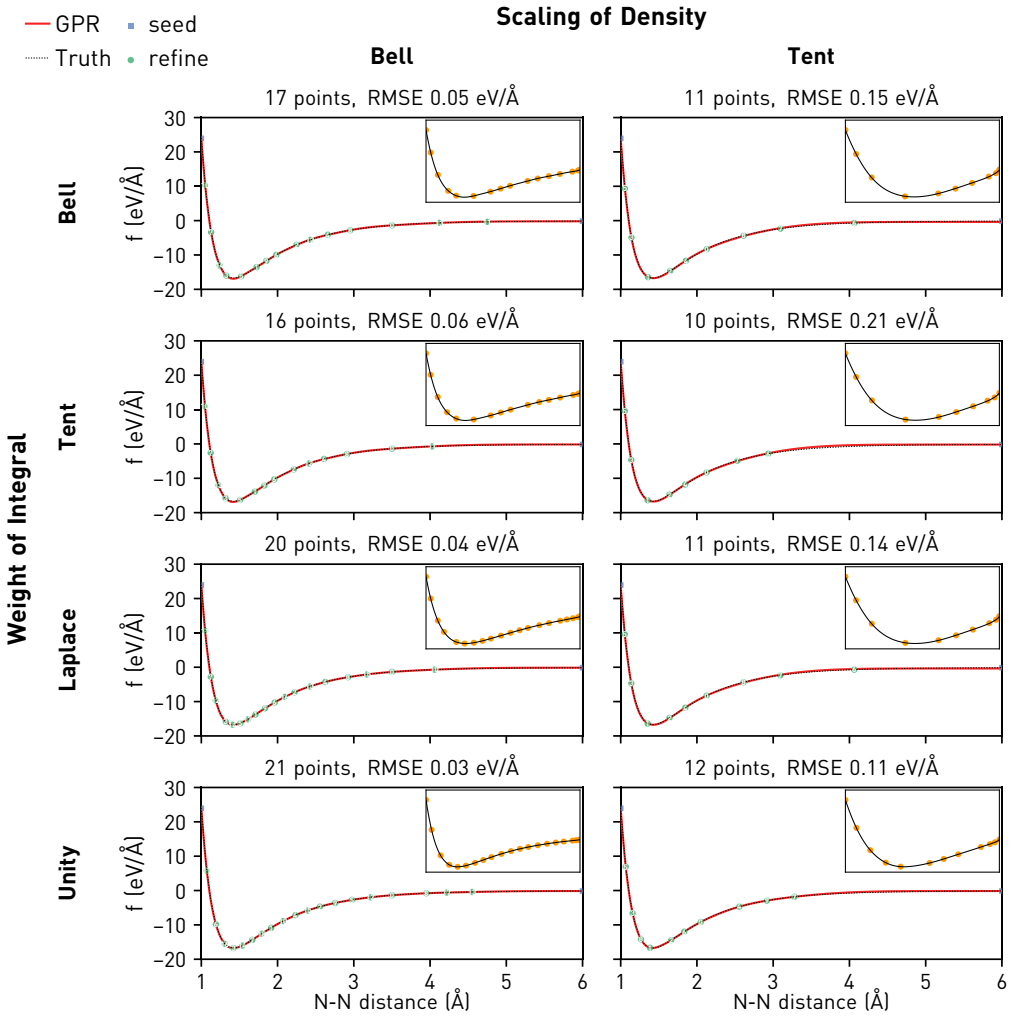


Figure 5.6: Gaussian process regression of the force between two nitrogen atoms as a function of interatomic distance using different combinations of radial weight functions. Inset figures are plots of the regression function using distances from the feature space.

it is often desirable to use as few as possible nodes to capture only information of the density field within a certain band limit[80]. Accordingly, the integral cutoff R_c, the number of quadrature nodes, and the width of the density kernel need to be tuned to obtain an optimal balance between resolution and computational speed.

When designing the composite quadrature rule, we chose the Laguerre quadrature for the radial direction because its nodes are denser near the origin but sparser farther away. This is consistent with our physical intuition that the near field generally has a stronger influence than the far field in an atomistic neighborhood. For example, the Van de Waals potential grows rapidly when atoms are in direct contact, but flattens out of the first coordinate shell. Accordingly, it may be possible for us to use sparser outer-layer grids to reduce the total number of quadrature nodes, while still keeping enough nodes in the inner layers to maintain the sensitivity of the quadrature toward close neighbors. Cooperatively, we can also use non-stationary Gaussian density kernels whose width dependent on the distance from the atom to the origin. In this way, the sparser nodes should still sufficiently sample the smoother far field. Wider kernels at remote atoms also reduces the possible difference between the far fields of two fingerprint. Thus, the contribution of the far field in the integral can be effectively tuned even though the weights on the quadrature nodes remain the same.

In Figure 5.7, we demonstrate how a variable-resolution quadrature can be combined with a widening smoothing density kernel to simultaneously reduce the computational complexity and preserve the quality of the fingerprint. In column A, a dense grid is used to sample density fields generated by a wide smoothing length. By examining the distance matrices of fingerprints sampled during bond stretching and angular stretching movements, we note that the radial similarity

decreases monotonically while the angular similarity changes nearly constantly. In column B, the number of quadrature nodes is kept the same, but the smoothing length is reduced as an attempt to increase fingerprint sensitivity. Better response in the near field of the radial direction is obtained, but the linearity in the far field in the angular direction is compromised. In column C, the fingerprint performs even worse due to the combination of a sparser quadrature grid and a small smoothing length. In column D, the performance recovered because we let the smoothing length depend on the radial distance and adjust the quadrature nodes according to this pattern.

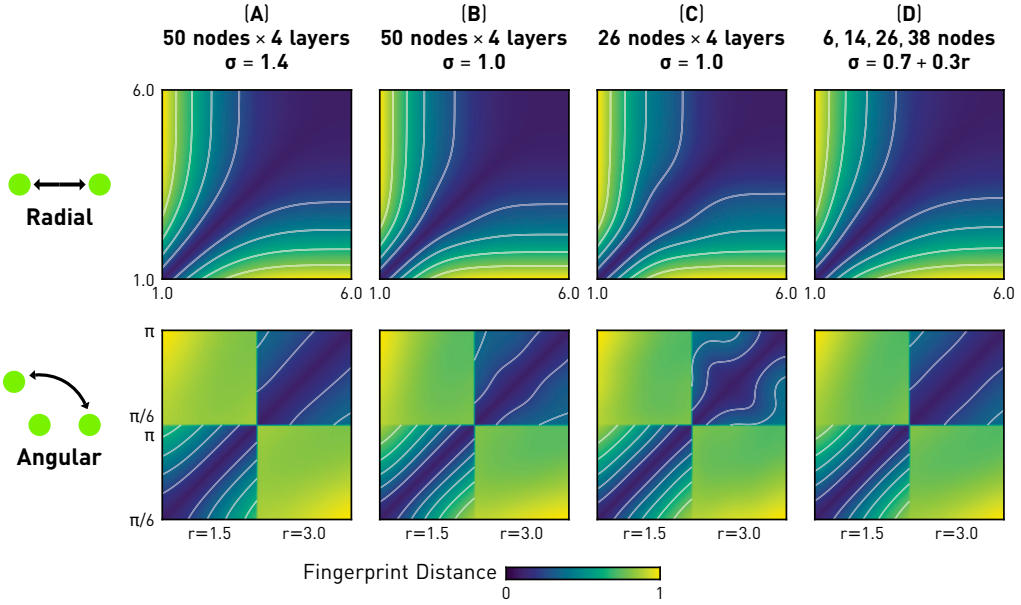


Figure 5.7: A comparison of fingerprint distance matrices corresponding to bond stretching and angular stretching movements. **(A)** Dense grid + large smoothing length: radial similarity decreases monotonically while angular similarity changes nearly constantly. **(B)** Dense grid + smaller smoothing length: better fingerprint sensitivity in the near field for bond stretching, compromised linearity in the far field for angular stretching. **(C)** Sparser grid + smaller smoothing length: compromised far-field performance for both bond and angular movements. **(D)** Variable-resolution grid + radially dependent smoothing length: good resolution and linearity in both near and far fields.

5.4 Demonstration

5.4.1 Potential Energy Surface

First, we attempt to fit the potential energy surface of a protonated water dimer system, in a head-to-head configuration, as a function of the oxygen-oxygen distance $r_{\text{O-O}}$ and the dihedral angle φ between the two planes each formed by a water molecule. As shown in Figure 5.8A, the system contains an improperly rotational symmetry, which we wish to capture with the kernel minisum algorithm. A GPR model was seeded with 8 training points corresponding to the combinations of $r_{\text{O-O}} = 2.2, 2.4, 2.6, 2.8$ and $\varphi = 0, \pi/2$. Subsequently, an active learning protocol was used to greedily absorb points with the highest uncertainty into the training set. Despite that we restricted all training data to be within the subdomain $\varphi <= \pi/2$, as shown by Figure 5.8B and 5.8C, we are able to accurately reproduce the target function over the entire parameter space after a few active learning steps.

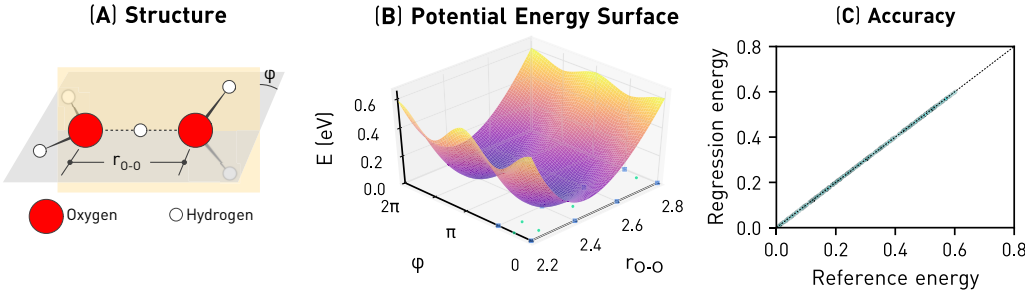


Figure 5.8: Gaussian process regression is carried out to fit the potential energy surface of a protonated water dimer as a function of two internal variables, *i.e.* the oxygen-oxygen distance $r_{\text{O-O}}$ and the dihedral angle φ between the plane determined by the two water molecules. This system contains an improperly rotational symmetry, which can be correctly recognized by the kernel minisum-based algorithm given in Alg. 16. Only a quarter of the domain was used to train the GP, yet the model can accurately predict energy of the entire parameter space thanks to symmetry detection.

Table 5.3: Geometry optimization and vibrational analysis of a single water molecule using GPR and our proposed fingerprint algorithm. 256 independent trials were performed using coordinates of water perturbed from equilibrium by a Gaussian noise $\mathcal{N}(\mathbf{0}, 0.15\mathbf{I})$ followed by a randomly chosen rigid-body rotation.

	GPR	DFT
Zero-point energy	0.591 ± 0.003 eV	0.583 eV
Static dipole	2.1580 ± 0.0001 D	2.159 D
Residual Force	0.0016 ± 0.0005 eV/Å	0.0003 eV/Å

Mode	Frequency (cm^-1)		Intensity (D/A)^2 amu^-1	
	GPR	DFT	GPR	DFT
0	1576.5 ± 1.4	1602.4	1.5726 ± 0.0005	1.5767
1	3819.3 ± 0.9	3817.5	0.2516 ± 0.0005	0.2159
2	3916.7 ± 1.6	3922.6	1.3349 ± 0.0028	1.3401

5.4.2 Geometry Optimization and Vibrational Analysis

Next, we demonstrate the usability of fingerprint for fitting vector-valued quantities by performing geometry optimization and vibrational analysis on a single water molecule. The process involves the simultaneous regression of: 1) energy, a molecular scalar quantity; 2) force, a per-atom vector quantity; and 3) dipole, a molecular vector quantity. Correspondingly, we performed GPR of energy and dipole using fingerprints extracted from the center of mass of the molecule, and GPR of force using fingerprints extracted from each atom. The training set consists of 45 configurations uniformly covering the range $r_{\text{O-H}} = 0.93, 0.95, 0.97, 0.99, 1.05$Å and $\theta_{\text{H-O-H}} = 101°, 105.5°, 111°$. As shown in Table 5.3, the GPR model can successful drive calculations of the infrared spectrum of the molecule from randomly perturbed initial structures in arbitrary orientation.

5.5 Discussion

5.5.1 Canonical Coordinate Frame

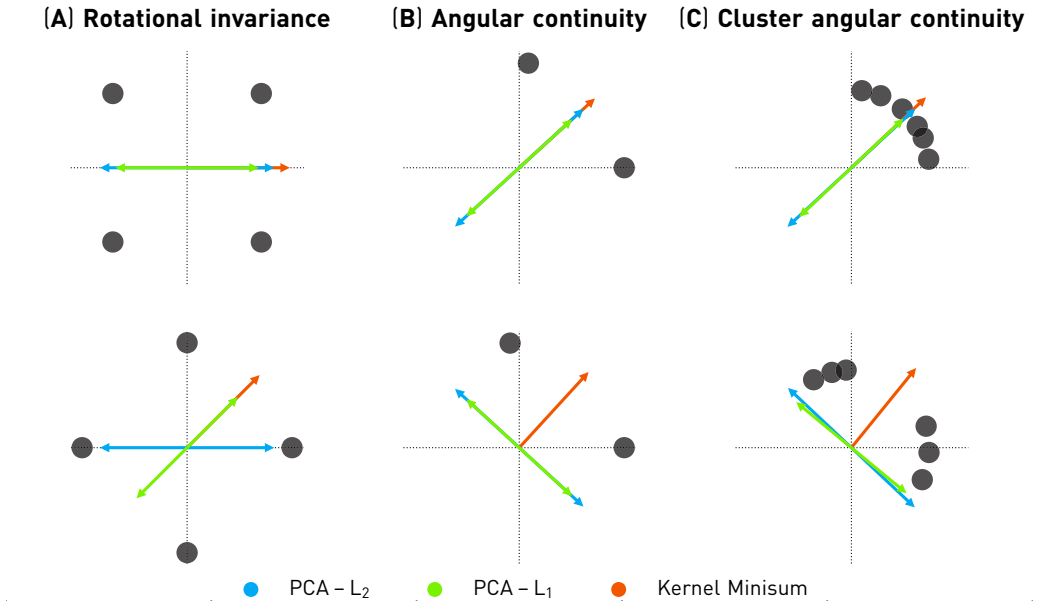


Figure 5.9: A comparison of the orthogonal bases obtained using principal components analysis (PCA) and kernel minisum with the square-angle (MSA) kernel. **(A)** The PCA algorithm, used in conjunction with the L_2 norm, fails to extract a principal axis that rotates with a system that exhibits planar C_4 symmetry. Both MSA and PCA with the L_1 norm can accommodate this scenario. **(B)** Both L_1 and L_2 principal axes changes orientation abruptly when the system undergoes a slight angular motion. In contrast, the MSA output is continuous with regard to this movement as it always bisects the angle formed by the atoms and the origin. **(C)** Loosely speaking, the MSA axis points at the majority direction of the atoms if only a single cluster is present within the cutoff distance, but bisects the angle between two atom clusters. This is different from PCA-based results and should deliver robust rotational invariance as well as continuity. Arrows are drawn at different lengths to improve visual clarity in situations of overlapping. They are to be understood as unit vectors. De-trending is not performed because the radial distances of the atoms carry physically significant information.

A major advantage of our kernel minisum approach as compared to PCA using the L^p norm, lies in its 1) robustness in the presence of structural symmetry; and 2) continuity of the resulting principal axes with respect to angular movement of the input data. As shown in Figure 5.9, kernel minisum is particularly suitable for atomistic systems where strong symmetries are common and the continuity against angular movement is desired. The minisum framework can also be used

with other customized kernels to suit for the characteristics of specific application scenarios.

5.5.2 Connection to Other Fingerprint Algorithms

In Figure 5.10, we compare the ability to distinguish atomistic configurations of our fingerprint as well as SOAP and the Coulomb matrix. Our work is inspired by the SOAP descriptor [10], which proposes the use of smoothed densities to represent atomistic neighborhoods. However, instead of converting the density field into the frequency domain using spherical harmonics, we perform density field sampling and comparison directly in the real space. This is enabled thanks to the available of canonical coordinate frame as computed through the kernel minisum optimization. We have mainly used the L_2 norm to compute the distance between atomistic neighborhoods. However, our fingerprint exhibits very similar behavior to SOAP when used together with an inner product formula

$$\mathrm{d}(\rho_{\mathbf{s}_1}, \rho_{\mathbf{s}_2}) \approx \frac{\sum_{k=1}^{N} w_k \, \rho_{\mathbf{s}_1}(\mathbf{r}_k) \, \rho_{\mathbf{s}_2}(\mathbf{r}_k)}{\sqrt{\sum_{k=1}^{N} w_k \, \rho_{\mathbf{s}_1}(\mathbf{r}_k) \, \rho_{\mathbf{s}_1}(\mathbf{r}_k)} \sqrt{\sum_{k=1}^{N} w_k \, \rho_{\mathbf{s}_2}(\mathbf{r}_k) \, \rho_{\mathbf{s}_2}(\mathbf{r}_k)}} \tag{5.31}$$

as demonstrated in Figure 5.10A. Thus, our fingerprint could be used in conjunction with a wide variety of covariance functions based on either the Euclidean distance or the inner product similarity.

At first sight, DECAF is very different from the Coulomb matrix fingerprint and GRAPE, which are both graph-based algorithms [128, 43]. However, instead of trying to capture the overall density field, if we measure the contribution from each individual atom on the quadrature nodes at $\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_M$ as a row vector, and

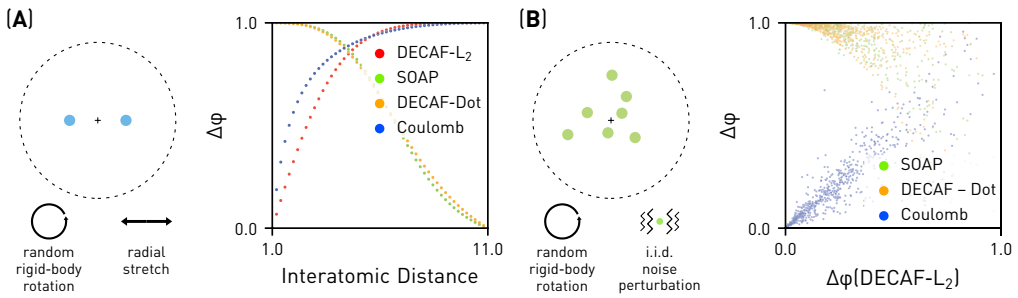


Figure 5.10: A comparison between the distance measure used DECAF, SOAP, and the Coulomb matrix. Fingerprint distances $\Delta\varphi$ shown in the plots are measured against $\mathbf{r}_{ij} = 1.0$ in **(A)** and an randomly chosen initial state in **(B)**.

stacked up the results to yield the matrix

$$\mathbf{E}_{ij}^{N\times M} = \mathrm{k}(\mathbf{x}_i, \mathbf{z}_j) \tag{5.32}$$

Then $\mathbf{E}$ can be regarded as an incidence matrix [55] between atoms and the quadrature nodes. This is similar to the graph-based abstraction as seen in the Coulomb matrix and the GRAPE kernel. However, in both cases the vertices in the graph represent atoms while the edges represent pairwise interatomic interactions. Here, the density-based incidence matrix adopts the opposite pattern and constructs a graph with the quadrature nodes being vertices and atoms being edges. The adjacency matrix in this case is computed as the inner product $\mathbf{E}^\mathsf{T}\mathbf{E}$:

$$\mathbf{A}_{ij}^{M\times M} = (\mathbf{E}^\mathsf{T}\mathbf{E})_{ij} = \sum_{k=1}^{N} \mathrm{k}(\mathbf{x}_k, \mathbf{z}_i)\,\mathrm{k}(\mathbf{x}_k, \mathbf{z}_j). \tag{5.33}$$

The weight on the edges, as represented by the elements of the adjacency matrix A, can be interpreted as the total flux as contributed by all paths each bridged by an atom k. We have numerically found that the smallest N eigenvalues (except for the 0 eigenvalue) of the symmetric normalized Laplacian

$$\mathbf{L} = \mathbf{I} - \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}, \text{ where } \mathbf{D}_{ii} = \delta_{ij}\sum_j \mathbf{A}_{ij} \tag{5.34}$$

is invariant under rotation up to a certain noise level, even if the quadrature nodes do not rotate with the atoms. Nonetheless, this detour appears to represent a pure theoretical interest rather than any practical value.

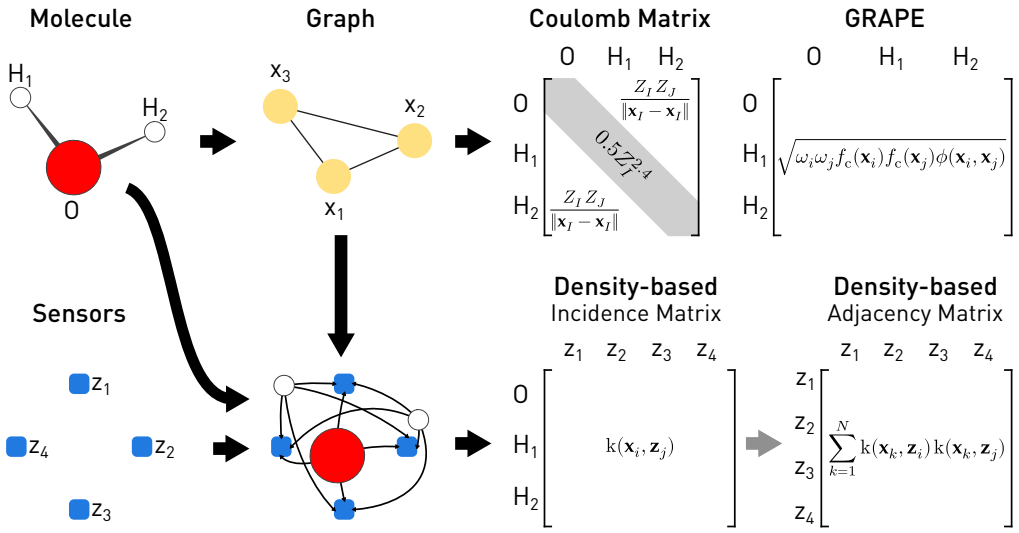


Figure 5.11: A comparison between graph-based molecular fingerprints. The Coulomb Matrix and the GRAPE kernel construct graphs where nodes corresponds to atoms while the weights on the edges are determined by some pairwise inter-atomic interactions. In contrast, in the density-based incidence matrix we construct a graph on a set of quadrature nodes whose connectivity is weighted by a sum of contributions from individual atoms.

5.6 Auxiliary Information

5.6.1 Polynomial Smoothing Functions with Compact Support

As candidates for the weight of integral and density scaling functions (Section 5.3.2), a class of compact polynomials that satisfy the criteria [97]:

1. is compactly supported,

2. is strictly positive within some cutoff distance r_c,

3. decreases monotonically,

4. is at least twice continuously differentiable

with minimal number of non-zero terms are:

$$W^{a,b}(s) = \frac{-b\,s^a + a\,s^b}{\sigma},\ a > b > 2, \tag{5.35}$$

where $s = 1 - r/h$ is the normalized complementary coordinate within the span h of the kernel, and

$$\sigma = 8\pi h^3 \left(\frac{a}{b^3 + 6b^2 + 11b + 6} - \frac{b}{a^3 + 6a^2 + 11a + 6} \right) \tag{5.36}$$

is an optional normalization factor to ensure that the integral of the kernel in a 3D ball of radius h is unity. The parameters a and b are free parameters that can be used to adjust the smoothness and width of the kernel, and can take any real numbers satisfying the condition $a > b > 2$. Note that the kernel $W^{4,3}$ is equivalent to the Lucy kernel commonly used in Smoothed Particle Hydrodynamics simulations [74]. The kernel can be evaluated very efficiently using only multiplication and addition when both a and b are integers.

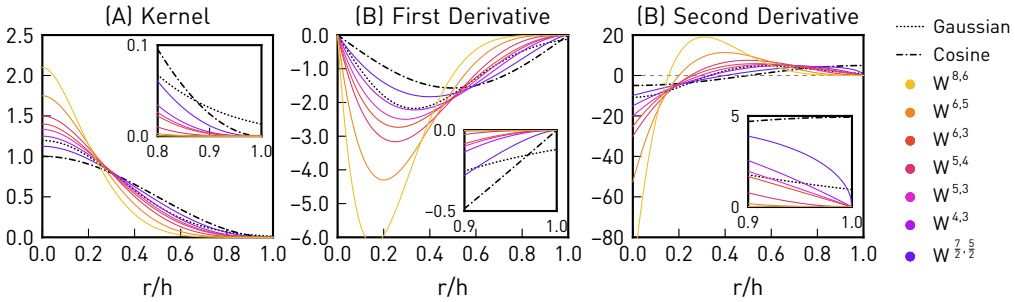


Figure 5.12: Visualization of the polynomial kernels as given in Eq. 5.35 with a unit support radius. The kernels are bell-shaped with a derivative of 0 at the origin. Both the first and second derivatives of the kernels transition smoothly to 0 at its support radius. In contrast, the Gaussian kernel and its derivatives does not decay to zero at any finite distance, while the second derivative of the Cosine kernel as mentioned in previous work [13, 10] is not zero at the cutoff distance.

5.6.2 Table of Quadrature Nodes and Weights

In the table below, we list the nodes and weights of the Laguerre quadrature rules up to $N_r = 6$, using notations from Eq. 5.22.

		n = 0	n = 1	n = 2	n = 3	n = 4	n = 5
Nr = 2	r_n	2.0000	6.0000				
	a_n	1.5000	0.5000				
Nr = 3	r_n	1.5174	4.3116	9.1710			
	a_n	1.0375	0.9058	0.0568			
Nr = 4	r_n	1.2268	3.4125	6.9027	12.4580		
	a_n	0.7255	1.0634	0.2067	0.0044		
Nr = 5	r_n	1.0311	2.8372	5.6203	9.6829	15.8285	
	a_n	0.5209	1.0667	0.3835	0.0286	0.0003	
Nr = 6	r_n	0.8899	2.4331	4.7662	8.0483	12.6004	19.2620
	a_n	0.3844	0.9971	0.5361	0.0795	0.0029	0.0000

In the table below, we list the nodes and weights of the Lebedev quadrature rules up to $N_r = 6$, using notations from Eq. 5.21.

Na = 6	x_m	y_m	z_m	b_m
m = 0,1	+-1.00000	0.00000	0.00000	0.16667
m = 2,3	0.00000	+-1.00000	0.00000	0.16667
m = 4,5	0.00000	0.00000	+-1.00000	0.16667

Na = 14	x_m	y_m	z_m	b_m
m = 0,1	+-1.00000	0.00000	0.00000	0.06667
m = 2,3	0.00000	+-1.00000	0.00000	0.06667
m = 4,5	0.00000	0.00000	+-1.00000	0.06667
m = 6-13	+-0.57735	+-0.57735	+-0.57735	0.07500

Na = 26	x_m	y_m	z_m	b_m
m = 0,1	+-1.00000	0.00000	0.00000	0.04762
m = 2,3	0.00000	+-1.00000	0.00000	0.04762
m = 4,5	0.00000	0.00000	+-1.00000	0.04762
m = 6-9	0.00000	+-0.70711	+-0.70711	0.03810
m = 10-13	+-0.70711	0.00000	+-0.70711	0.03810
m = 14-17	+-0.70711	+-0.70711	0.00000	0.03810
m = 18-25	+-0.57735	+-0.57735	+-0.57735	0.03214

Na = 38	x_m	y_m	z_m	b_m
m = 0,1	+-1.00000	0.00000	0.00000	0.00952
m = 2,3	0.00000	+-1.00000	0.00000	0.00952
m = 4,5	0.00000	0.00000	+-1.00000	0.00952
m = 6-13	+-0.57735	+-0.57735	+-0.57735	0.03214
m = 14-17	+-0.45970	+-0.88807	0.00000	0.02857
m = 18-21	+-0.88807	+-0.45970	0.00000	0.02857
m = 22-25	+-0.45970	0.00000	+-0.88807	0.02857
m = 26-29	+-0.88807	0.00000	+-0.45970	0.02857
m = 30-33	0.00000	+-0.45970	+-0.88807	0.02857
m = 34-37	0.00000	+-0.88807	+-0.45970	0.02857

Na = 50	x_m	y_m	z_m	b_m
m = 0,1	+-1.00000	0.00000	0.00000	0.01270
m = 2,3	0.00000	+-1.00000	0.00000	0.01270
m = 4,5	0.00000	0.00000	+-1.00000	0.01270
m = 6-9	0.00000	+-0.70711	+-0.70711	0.02257
m = 10-13	+-0.70711	0.00000	+-0.70711	0.02257

```
 m  = 14-17     +-0.70711         +-0.70711           0.00000          0.02257
 m  = 18-25     +-0.57735         +-0.57735         +-0.57735          0.02109
 m  = 26-33     +-0.30151         +-0.30151         +-0.90453          0.02017
 m  = 34-41     +-0.30151         +-0.90453         +-0.30151          0.02017
 m  = 42-49     +-0.90453         +-0.30151         +-0.30151          0.02017
 ================================================================================
```

The Laguerre and Lebedev quadrature nodes can be combined using Eq. 5.23-5.26 into composite grids for sampling the atomistic density field.

5.7 Summary

In this chapter, I presented the Density-Encoded Canonically Aligned Fingerprint (DECAF) algorithm, which explores the idea of using smoothed density fields to represent and compare atomistic neighborhoods. One of the key enabling technique in DECAF is a kernel minisum algorithm, which allows the unambiguous identification of a canonically aligned coordinate frame that can be used for rotationally invariant projection of the density field as well as any associated vector quantities. We have performed detailed analysis to study the behavior of the fingerprint by changing various parameter, such as resolution, smoothing length, and the choice of weight functions. We demonstrate that the fingerprint algorithm can be used to implement highly accurate regressions of both scalar and vector properties of atomistic systems including energy, force and dipole moment, and could be a useful building block for constructing data-driven next generation force fields for various applications such as molecular dynamics simulations and molecular docking.

Chapter Six

Conclusion

In summary, I have presented algorithms, models, and application of mesoscopic and multiscale scientific computation, as well as a suite of open-source high performance scientific computing software that provides the simulation capability:

Multiscale Universal Interface: The MUI project aims to address the difficulty in implementing concurrently coupled multiscale simulations by providing a very simple, yet powerful, set of interfaces for inter-solver communications. By introducing the concept of data points and data sampler, MUI allows information to be extracted by giving the user the freedom to choose their own interpolation or interpretation scheme. The library incurs negligible overhead when running on hundreds of processors.

***USER*MESO:** The USERMESO package is a GPU-accelerated extension of the LAMMPS simulator for Dissipative Particle Dynamics and other mesoscopic particle methods [144]. Many algorithmic innovations were incorporated, *e.g.* an atomics-free neighbor list constructor and specialized transcendental functions. It scales over thousands of nodes and is more than 20 times faster on one GPU over tens of CPU cores.

OpenRBC: OPENRBC is a coarse-grained molecular dynamics code which allows for the first time the simulation of an entire human red blood cell at protein resolution. An adaptive spatial searching algorithm was invented to accelerate the computation of short-range pairwise interactions in an extremely sparse 3D space based on a Voronoi partitioning of the coarse-grained particles. The code outperforms a legacy solver by 10 times in time-to-solution, thus providing a new platform for probing the biomechanics of red blood cells.

My future work will emphasize the establishment of a comprehensive ecosystem around the existing developments, and on further promoting the adoption

and evolution of mesoscopic research methodologies. However, methods and software is not the goal, but rather tools for reaching the goal. As such, I believe methodology development should always be guided by real world applications and problems. From another perspective, the ability to solve an realistic problem is also an arguably best way to demonstrate the applicability and the value of a method. As such, I am actively seeking opportunities to apply the algorithms and tools to address scientific problems, including and beyond the dynamics of enzyme catalysis processes and cytomechanics of healthy and diseased blood cells.

Appendix A

The Ermine Utility Code

A.1 ERMINE: A Generic Particle Placement Library

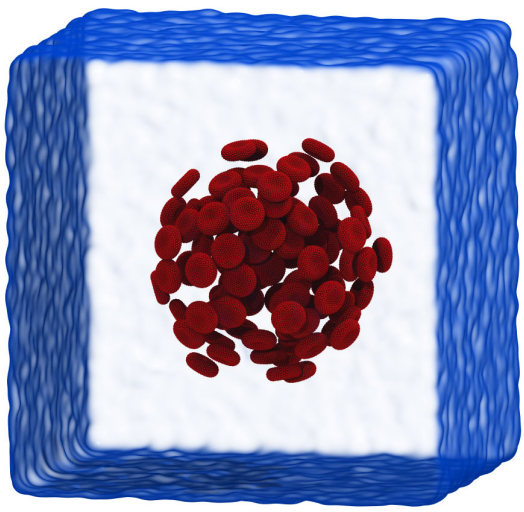
ERMINE is a programming library for building initial configurations for particle-based simulations. Instead of providing a fixed set of functionalities, ermine provides generic C++ templates and intrinsics that allow direct and intuitive manipulation of particle-based models. Users can use **ERMINE** to efficiently develop compact and high-performance C++ applets for expressive construction of diverse systems. **ERMINE** can be loosely, and perhaps inappropriately, thought of as the 3D particle-based version of Matplotlib [64].

A.2 Quick Start Example

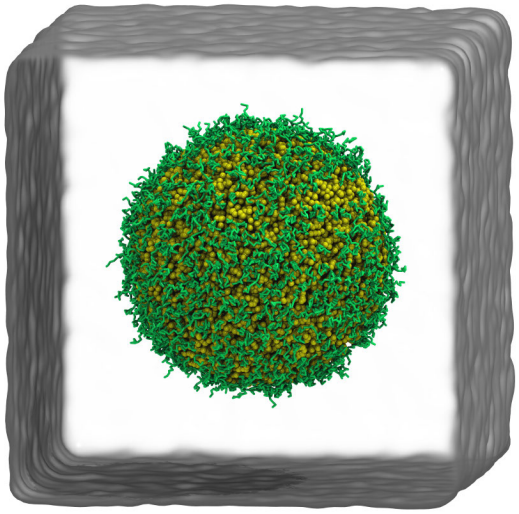
To download Ermine, go to `http://www.cosx-isinx.org/`

Below is a code that generates start polymer melt by duplicating a template star polymer at random locations and orientations.

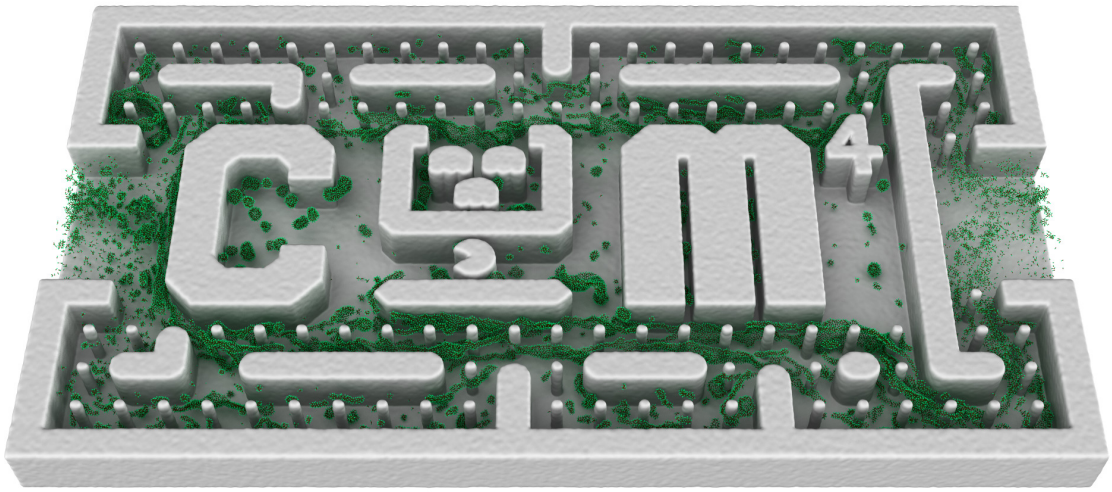
```
#include <ermine.h>
using namespace std;
using namespace ermine;
int main()
{
    ParserLammps<ObjectWithBond> parser;
    Sandbox     <ObjectWithBond> sandbox;

    auto star = parser.read( "star.data" );
    sandbox.set_bbox( InBox( 0, Vector3D(100,100,80) ) );
```



(a) Red blood cells in a solvent box with all cells facing the center of box.



(b) Colloid particle grafted with polymer and immersed in a solvent box.



(c) Polymer solution flowing through a microfluidic channel of complex geometry.

```
11    sandbox.set_overlap_cutoff( 0.3 );
12    fill( sandbox, star, 100, Rectangle(100,100,80), RandomTilt(2*M_PI) );
13    parser.write( "star_melt.data", sandbox );
14 }
```

Explanation of the code:

- lines 1-3: includes the ermine header. This is the only file that has to be included, and no pre-compilation is needed. All ermine functionalities are defined in **`namespace ermine`**.
- lines 7: a parser is a class that reads particle models from files. Current only the LAMMPS data format and the PDB format are supported. The LAMMPS parser is defined as a template of the molecule type in order to handle variations between different subtypes.
- line 8: a sandbox is where the actual construction work is done. It is a container for storing intermediate particle models, and can also efficiently performs the task of overlap checking.
- line 10: load the template model file as an C++ object. The C++11 **`auto`** keyword allows us to omit the type of the model, which in this case would be **`ObjectWithBond`**.
- lines 11-12: Set the global bounding geometry as a rectangular box of dimension $100 \times 100 \times 80$, while also set the overlap checking cutoff to be 0.3. The overlap checking feature will be detailed later.
- line 13: fill the sandbox using 100 copies of the loaded template at random positions and random orientations by applying the corresponding rigid-body transformations to the template molecule. The global bounding geometry and cutoff rules will be used to make sure no molecules will either lie outside of the rectangular box or overlap with other molecules.

· line 14: write the content of the sandbox to a file

A.3 Programming Model

Ermine seeks to fight against the combinatorial complexity in building particle systems by orthogonalizing functionalities of its components. In this way complicated construction strategies may be described by combining basis features. There are five major categories of classes in ermine: **`objects`**, **`parsers`**, **`sandbox`**, **`tools`** and **`functors`**.

A.3.1 Object

`Object` is the generalization of the concept of molecule. They are essentially C++ classes that act as containers for holding a variable numbers of simple entities such as atoms, bonds, angles, etc. Ermine allows users to define custom interaction types and objects through a declarative attribute system, which also supports semi-automatic data serialization and deserialization.

All the functional intrinsics, *e.g.* parsers, sandbox, and utility codes, in **Ermine** that operates on molecule-like objects are templaterized. Therefore, they are designed to handle any, instead of one, object as long as the object class contains all the necessary information for carrying out the given task. This is implemented using an attribute-based design pattern and static dispatch, rather than inheritance and virtual methods, and thus does not incur any polymorphism overhead. The attribute-based design pattern will be detailed in later publications.

A.3.2 Type remapping

Suppose that we have two input molecules. The first molecule contains two different types of atoms, e.g. carbon and oxygen, labeled as type 1 and 2. The second molecule also contains two different types of atoms labeled as type 1 and 2, but in this case they are hydrogen and nitrogen. In other words the types with the same numeric id in fact refers to different types of elements. Now considering the scenario of randomly placing the two template molecules in a sandbox. If the type of the particle were just copied mechanically, we end up getting a system with particles labeled either as type 1 or 2, despite the fact that the system should contain four different types of atoms. This would of course cause confusion when, for example, preparing force field files to set up a simulation.

One plausible solution would be to change the second model such that the hydrogen and nitrogen atoms get labeled as type 3 and 4. However this would compromise the reusability of the template file, since with every new combination we would need to relabel the particles repeatedly.

Ermine solves the problem by applying family-based type remapping at the time of writing a sandbox into a file. In the example above, in the final output file, carbon and oxygen atoms will still be kept as type 1 and 2, while hydrogen and oxygen atoms will be remapped as type 3 and 4 automatically. This is implemented by having each input object (molecule) carrying a family name. All objects with the same family name thus occupy the same type space. Particles with the same numeric type id but different family names are treated as having different types. The types will be consolidated and numbered according to their order of appearance in the sandbox when being written to a file.

A.3.3 Parser

Parsers control the information exchange between various realizations of **`Object`**s and disk files. The public interface that a **`Parser`** exposes is very minimal: 1) a **`read`** method that can either read from a file, as specified by its file name, or an input stream; and a **`write`** method that writes either an entire sandbox, or a single object, to either a file, as specified by its file name, or an output stream.

A.3.4 Sandbox

`Sandbox`es are container classes that store what has been placed in a system. It is templaterized, just like containers such as **`std::vector`**, using the type of the object as an argument. In this way, it is able to automatically handle new object classes defined by users that carry customized information. It implements all the access semantics of **`std::vector`**, and as such can be used in pretty much the same way as an STL container. **`Sandbox`** provides overlap checking functionalities with an amortized $O(1)$ complexity. This overlap checking functionality is key to the efficient implementation of many **Ermine** functionalities.

A.4 Tools

As mentioned previously, **Ermine** provides intrinsics, i.e. algorithmic building blocks, that can be assembled into particle system builders using minimal coding labor. For example, to place 100 replicates of a template molecule at random locations but with a fixed orientation on a z-parallel plane from (0,0) to (50,50), we

can translate the following algorithm:

```
template = load( file )
system   = empty
count    = 0
while count < 100
  instance = template
  move instance to random point within (0,0)-(50,50)
  if no overlap between system and instance
    insert instance into system
    count = count + 1
  endif
endwhile
```

into:

```
#include <ermine/ermine.h>
using namespace ermine;
int main() {
    ParserLammps<ObjectWithBond> parser;
    Sandbox     <ObjectWithBond> sandbox;
    auto star = parser.read( "star.data" );
    int count = 0;
    sandbox.set_overlap_cutoff( 0.3 );
    while( count < 100 ) {
        auto instance = star;
        star.moveto( uniform() * 50, uniform() * 50, 0 );
        if( !sandbox.overlap( instance ) ) {
            sandbox.insert( instance );
            count++;
        }
```

```
16      }
17  }
```

However, this is not the end of the story. **Ermine** includes a collection of tool functions that packs some of the most frequently used generation algorithms so that the tools can be used out-of-the-box. Users can check out the source files prefixed with **`tool_`** to locate the tools.

A.5 Functors

Ermine relies on functors to implement a rich set of configurable options. In the C++ syntax, functors are classes equipped with the **`operator ()`** method. Instances of such classes can thus be used as if they were simply functions. For example:

```
1  struct Add {
2    int inc;
3    int operator () (int x) { return x + inc; }
4  };
5
6  int main() {
7    Add add;
8    add.inc = 5;
9    printf( 3  + 5 = %d\n , add(3) );
10 }
11
12 output:
```

```
13 8
```

The usefulness of functors comes from the fact that they can be passed around between program components just like ordinary objects, and thus can store additional information to configure the function behavior dynamically.

A.5.1 Predicates

Predicates are functors that returns Boolean values. In fact, the global bounding geometry feature is Ermine is entirely built on top of predicates whose inputs are 3D coordinates. There are other predicates defined in Ermine as well. For example, particle predicates, i.e. predicates whose input are particles, serves to screen particles that gets crosslinked in the cross link tool.

A.5.2 Site generators and Spatial predicates

The ability to generate random vectors from a distribution in 3D is core to the placement capability of Ermine. For example, a uniform distribution on a spherical shell is a reasonable fit for creating vesicles using surfactant molecules. Furthermore, to model a microfluidic devices, we may need to put solvent particles in a cylindrical channel while avoiding a few specific locations (e.g. poles).

The entire collection of distributions that may be used in various simulation scenarios is beyond enumeration. Ermine adopts a rejection sampling approach to fight against such complexity. It works by letting users to compose generalized bounding geometries with Boolean combination of geometric primitives. These

bounding geometries can then be used to reject points that fall outside of the specified geometry. The detection algorithm is executed hierarchically along the suffix tree that represents the geometry description. The geometry primitives are in fact spatial predicates as mentioned previously.

A.5.3 Options

Options are functors that are used by tools such as **genpoly** and **crosslink** for encoding customized parameters.

Appendix B

Examples for Multiscale Universal Interface

B.1 Couette flow

Listing B.1: The MUI functionality was implemented as a LAMMPS *fix* for coupling two SPH simulations of different resolution.

```
#include "fix.h"
#include <mui/mui.h>

// LAMMPS custom fix declaration
class FixMUI : public Fix {
public:
  FixMUI(class LAMMPS *, int, char **);

  virtual ~FixMUI() {
    if ( interface ) delete interface;
  }
  int setmask() {
    return POST_INTEGRATE | END_OF_STEP;
  }

  virtual void post_integrate();
  virtual void end_of_step();

protected:
  mui::uniface3d *interface;
  double send_upper, send_lower;
  double recv_upper, recv_lower;
  double sample_rc;
};

// initialize MUI main object
// parsing interface region description from LAMMPS command
FixMUI::FixMUI(LAMMPS *lmp, int narg, char **arg) :
  Fix(lmp, narg, arg) {
```

```
30   if (narg < 9) error->all(FLERR,"Illegal fix mui command");
31
32   // arg[3] example: mpi://sph_fine/sph_sph_interface
33   //                 mpi://sph_coarse/sph_sph_interface
34   interface = new mui::uniface<default_config>( arg[3] );
35   send_upper = atof( arg[4] );
36   send_lower = atof( arg[5] );
37   recv_upper = atof( arg[6] );
38   recv_lower = atof( arg[7] );
39   sample_rc  = atof( arg[8] );
40
41   nevery = 1;
42 }
43
44 // the PUSH part
45 void FixMUI::post_integrate()
46 {
47   for (int i = 0; i < atom->nlocal; i++) {
48     // check whether particle in interface region
49     if ( atom->x[i][1] >= send_lower && atom->x[i][1] <= send_upper ) {
50       // push data points of type double under name "v_x"
51       interface->push( "v_x", mui::point3d(atom->x[i]), atom->v[i][0] );
52     }
53   }
54   // commit points generated in this step
55   // MUI handles delivery of the points via MPI
56   double t = update->ntimestep * update->dt;
57   interface->commit( t );
58   // barrier to bound solver progress discrepancy within 1
59   // prevent frames accumulating in case of load imbalance
60   interface->barrier( t-1 );
61   // remove obsolete frames to free up memory
62   interface->forget( t-1 );
```

```
}

// the FETCH part
void FixMUI::end_of_step()
{
  // define spatial and temporal sampler
  mui::sampler_shepard_quintic<double> quintic(sample_rc);
  mui::chrono_sampler_exact<>          texact(0);

  double t = update->ntimestep * update->dt;
  for (int i = 0; i < atom->nlocal; i++) {
    // determine whether particle in interface region
    if ( x[i][1] >= recv_lower && x[i][1] <= recv_upper ) {
      // interpolate value, note v_x here is from other solver
      double r = interface->fetch( "v_x", mui::point3d(atom->x[i]), t, quintic,
    texact );
      // apply coupling
      atom->v[i][0] = r;
    }
  }
}
```

Listing B.2: MUI Exact-point temporal sampler

```
template<typename CONFIG=default_config> class chrono_sampler_exact {
public:
  using REAL       = typename CONFIG::REAL;
  using INT        = typename CONFIG::INT;
  using time_type  = typename CONFIG::time_type;

  chrono_sampler_exact( time_type tol = time_type(0) ) {
    tolerance = tol;
  }

```

```
11    template<typename TYPE>
12    TYPE filter( time_type focus, const std::vector<std::pair<time_type, TYPE> >
        &points ) const {
13      for( auto i: points ) {
14        if ( std::abs(i.first - focus) <= tolerance ) {
15          return i.second;
16        }
17      }
18      return TYPE(0);
19    }
20    time_type get_upper_bound( time_type focus ) const {
21      return focus + tolerance;
22    }
23    time_type get_lower_bound( time_type focus ) const {
24      return focus - tolerance;
25    }
26
27  protected:
28    time_type tolerance;
29  };
```

Listing B.3: MUI SPH sampler using the quintic spline

```
1  template<typename O_TP, typename I_TP = O_TP, typename CONFIG = default_config>
2  class sampler_sph_quintic
3  {
4  public:
5      using OTYPE      = O_TP;
6      using ITYPE      = I_TP;
7      using REAL       = typename CONFIG::REAL;
8      using INT        = typename CONFIG::INT;
9      using point_type = typename CONFIG::point_type;
10     const static int D = CONFIG::D;
11
```

```
12      sampler_sph_quintic( REAL r_ ) : r( r_ ), hinv( REAL( 3 ) / r_ )
13      {
14          static_assert( D == 1 || D == 2 || D == 3, "Quintic kernel for dimension
        other than 1,2,3 not defined." );
15          REAL sigma;
16          switch( D ) {
17          case 1:
18              sigma = 1.0 / 120.0;
19              break;
20          case 2:
21              sigma = 7.0 / 478.0 / PI;
22              break;
23          case 3:
24              sigma = 1.0 / 120.0 / PI;
25              break;
26          }
27          norm_factor = sigma * power<D>( hinv );
28      }
29
30      template<template<typename, typename> class CONTAINER>
31      inline OTYPE filter( point_type focus, const CONTAINER<ITYPE, CONFIG>
        &data_points ) const
32      {
33          OTYPE vsum = 0;
34          for( INT i = 0 ; i < data_points.size() ; i++ ) {
35              auto dist2 = ( focus - data_points[i].first ).normsq();
36              if( dist2 < r * r ) {
37                  REAL w = quintic_polynomial( sqrt( dist2 ) );
38                  vsum += data_points[i].second * w;
39              }
40          }
41          return vsum;
42      }
```

```
43
44      inline geometry::any_shape<CONFIG> support( point_type focus ) const
45      {
46          return geometry::sphere<CONFIG>( focus, r );
47      }
48
49  protected:
50      REAL r, hinv, norm_factor;
51
52      inline REAL quintic_polynomial( const REAL dist ) const
53      {
54          REAL s, w;
55          REAL s1, s2, s3;
56          REAL s1_5, s2_5, s3_5;
57          s = dist * hinv;
58          s1 = 1.0 - s;
59          s2 = 2.0 - s;
60          s3 = 3.0 - s;
61          s1_5 = 15.0 * power<5>( s1 );
62          s2_5 = -6.0 * power<5>( s2 );
63          s3_5 = power<5>( s3 );
64          if( s < 1.0 ) {
65              w = s3_5 + s2_5 + s1_5;
66          } else if( s < 2.0 ) {
67              w = s3_5 + s2_5;
68          } else if( s < 3.0 ) {
69              w = s3_5;
70          } else {
71              w = 0.0;
72          }
73          w *= norm_factor;
74          return w;
75      }
```

```
};
```

B.2 Soft Matter

Listing B.4: MUI Temporal summation sampler

```
template<typename CONFIG=default_config> class chrono_sampler_sum {
public:
  using REAL       = typename CONFIG::REAL;
  using INT        = typename CONFIG::INT;
  using time_type  = typename CONFIG::time_type;

  chrono_sampler_sum( time_type newleft = time_type(0), time_type newright =
    time_type(0) ) {
    left   = newleft;
    right  = newright;
  }

  template<typename TYPE>
  TYPE filter( time_type focus, const std::vector<std::pair<time_type, TYPE> >
    &points ) const {
    TYPE sum = TYPE(0);
    for( auto i: points ) {
      if ( i.first <= focus + right && i.first >= focus - left ) {
        sum += i.second;
      }
    }
    return sum;
  }
  time_type get_upper_bound( time_type focus ) const {
    return focus + right;
  }
```

```
25   time_type get_lower_bound( time_type focus ) const {
26     return focus - left;
27   }
28
29 protected:
30   time_type left, right;
31 };
```

B.3 Conjugate Heat Transfer

Listing B.5: MUI spatial averaging sampler based on the voronoi diagram of a given set of vertices

```
1  template<typename O_TP, typename I_TP = O_TP, typename CONFIG = default_config>
2  class sampler_voronoi_mean
3  {
4  public:
5      using OTYPE      = O_TP;
6      using ITYPE      = I_TP;
7      using REAL       = typename CONFIG::REAL;
8      using INT        = typename CONFIG::INT;
9      using point_type = typename CONFIG::point_type;
10     static int const D = CONFIG::D;
11
12     sampler_voronoi_mean( const std::vector<point_type > &v, const
       std::vector<std::set<INT> > &n ):
13         vertices( new std::vector<point_type >( v ) ),
14         neighbors( new std::vector<std::set<INT> >( n ) )
15     {
16         hmax = 0;
17         for( INT i = 0 ; i < neighbors->size() ; i++ ) {
18             auto xi = vertices->operator[]( i );
19             for( auto &j : neighbors->operator[]( i ) ) {
```

```
20                hmax = std::max( hmax, ( xi - vertices->operator[]( j ) ).norm()
    );
21            }
22        }
23    }
24
25    sampler_voronoi_mean( sampler_voronoi_mean &s ) :
26        neighbors( s->neighbors ),
27        vertices( s->vertices ),
28        hmax( s->hmax ) {}
29
30
31    template<template<typename, typename> class CONTAINER>
32    inline OTYPE filter( point_type focus, const CONTAINER<ITYPE, CONFIG>
    &data_points ) const
33    {
34        INT my_vertex = find_closest( focus );
35        auto sum = OTYPE( 0 );
36        auto n   = INT( 0 );
37        auto v   = vertices->operator[]( my_vertex );
38        auto &neigh_list = neighbors->operator[]( my_vertex );
39        // for each data point, check whether its closest to my vertex
40        for( int i = 0; i < data_points.size(); i++ ) {
41            auto u = data_points[i].first;
42            REAL rmine = ( u - v ).normsq();
43            bool is_mine = true;
44            for( auto &j : neigh_list ) {
45                REAL r = ( u - vertices->operator[]( j ) ).normsq();
46                if( r < rmine ) {
47                    is_mine = false;
48                    break;
49                }
50            }
```

```
51              if( is_mine ) {
52                  sum += data_points[i].second;
53                  n++;
54              }
55          }
56          return sum / n;
57      }
58      inline geometry::any_shape<CONFIG> support( point_type focus ) const
59      {
60          point_type closest = vertices->operator[]( find_closest( focus ) );
61          return geometry::sphere<CONFIG>( closest, hmax );
62      }
63  protected:
64      // flyweights
65      std::shared_ptr<std::vector<point_type > > vertices;
66      std::shared_ptr<std::vector<std::set<INT> > > neighbors;
67      REAL hmax;
68
69      // locate closest vertex
70      inline INT find_closest( point_type focus ) const
71      {
72          INT closest = -1;
73          REAL rmin = std::numeric_limits<REAL>::infinity();
74          for( INT i = 0; i < vertices->size(); i++ ) {
75              auto &v = vertices->operator[]( i );
76              REAL r = ( v - focus ).normsq();
77              if( r < rmin ) {
78                  closest = i;
79                  rmin = r;
80              }
81          }
82          return closest;
83      }
```

```
84 };
```

Listing B.6: MUI linear nearest 2 points sampler

```
1  template<typename O_TP, typename I_TP=O_TP, typename CONFIG=default_config>
2  class sampler_pseudo_nearest2_linear {
3  public:
4    using OTYPE      = O_TP;
5    using ITYPE      = I_TP;
6    using REAL       = typename CONFIG::REAL;
7    using INT        = typename CONFIG::INT;
8    using point_type = typename CONFIG::point_type;
9
10   sampler_pseudo_nearest2_linear( REAL h_ ) : h(h_) {}
11
12   template<template<typename,typename> class CONTAINER>
13   inline OTYPE filter( point_type focus, const CONTAINER<ITYPE,CONFIG>
       &data_points ) const {
14     REAL r2min_1st = std::numeric_limits<REAL>::max();
15     REAL r2min_2nd = std::numeric_limits<REAL>::max();
16     OTYPE value_1st = 0, value_2nd = 0;
17     for(INT i = 0 ; i < data_points.size() ; i++) {
18       REAL dr2 = ( focus - data_points[i].first ).normsq();
19       if ( dr2 < r2min_1st ) {
20         r2min_2nd = r2min_1st;
21         value_2nd = value_1st;
22         r2min_1st = dr2;
23         value_1st = data_points[i].second ;
24       } else if ( dr2 < r2min_2nd ) {
25         r2min_2nd = dr2;
26         value_2nd = data_points[i].second ;
27       }
28     }
29     REAL r1 = std::sqrt( r2min_1st );
```

```
30      REAL r2 = std::sqrt( r2min_2nd );
31      return ( value_1st * r2 + value_2nd * r1 ) / ( r1 + r2 );
32    }
33    inline geometry::any_shape<CONFIG> support( point_type focus ) const {
34      return geometry::sphere<CONFIG>( focus, h );
35    }
36  protected:
37    REAL h;
38  };
```

Appendix C

A Simple Physics-Based Parallel Ray Tracer

C.1 Ray tracing

Ray tracing stands out among a plethora of graphics rendering technologies thanks to its unique capabilities in generating photo-realistic images using a relatively simple algorithm. In this method, a digital image, just like its real-world analog — a picture recorded on a film, is obtained by accumulating rays that travel from the scene into a virtual camera. However, instead of numerically tracing the path of rays that originates from some light source into the camera, the reverse direction is taken and rays are shot from the camera into the scene. This strategy is motivated by the observation that only a small fraction of the rays that radiate from a light source will eventually enter the camera before being absorbed after repeatedly bouncing between material surfaces. The intersection between the rays and scene objects are solved for and used to determine the color of the incoming rays that lights up the image pixels.

This algorithm carries the following advantages:

1. The intersection of rays with many geometric objects can be solved analytically. In fact, sphere is the fastest object to render in ray tracing. In contrast, with traditional tessellation techniques, a sphere has to be rasterized using a large number of triangles to ensure surface smoothness, thus incurring significant overhead.

2. By manipulating the (recursive) generation of rays, it is straightforward to implementation various optical effects such as soft shadows, transparency, depth-of-field and motion blur.

3. The computation for tracing each individual ray is embarrassingly parallel

and hence can be accelerated easily with hybrid computer clusters with heterogeneous accelerators.

C.2 Implementation Overview

Collaborating with Lu Lu and Dongkun Zhang, I developed from scratch a very compact ray tracer with the following features:

- Native support for spheres, triangles, smoothed triangles, and cylinders.
- Multiple directional light with adjustable color, luminance and hardness.
- Diffusive, reflective, transparent & refractive, illuminating materials.
- Soft shadows
- Adaptive ray depth control.
- log(N) ray-scene intersection detection.
- MPI + OpenMP hybrid parallelization.
- Dynamic load balancing.

The code is written with C++11 using only 700 lines of code. The following classes were implements for encapsulating the basic concepts of ray tracing:

```
struct Timer    ; // performance monitoring and assists dynamic load balancing
struct Ray      ; // ray originating point and direction
struct Material ; // description of surface optical properties
struct Object   ; // base class for scene objects
```

```
5  struct Sphere   ; // scene object primitive
6  struct Triangle ; // scene object primitive
7  struct Striangle; // scene object primitive
8  struct Cylinder ; // scene object primitive
9  struct RNG      ; // thread-safe parallel random number generator
10 struct FastTree ; // log(N) fast scene traversal structure
11 struct Scene    ; // containers of scene objects and lighting information
```

The tracer can be downloaded from `http://www.cosx-isinx.org/`

C.3 Intersection Algorithm

The program can deal with a variety of basic geometric primitives, while more complicated structures has to be partitioned into a combination of the geometric primitives. The primitives are: sphere, triangle, smoothed triangle (s-triangle), and cylinder. The intersection method takes in a ray as the input, and returns a real number, which is the distance from the light source to the incident point, as well as a vector, which is the unit normal vector at the incident point, pointing to the direction of the light source. If the ray does not intersect with any object, a distance of -1.0 will be returned.

C.3.1 Sphere

Sphere is the simplest geometric primitive in ray-tracing. A sphere is uniquely defined by its center (a vector) and radius. The following plot shows the reflection of a ray on a ball in a 2D plane.

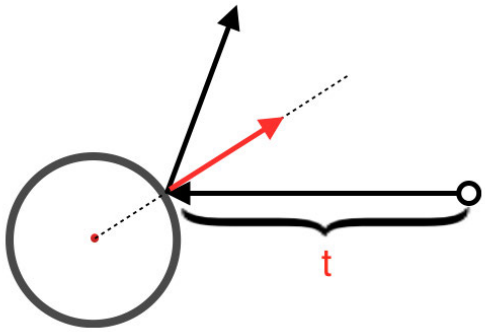


Figure C.1: Illustration of ray-sphere intersection.

C.3.2 Triangle

Another basic geometric primitive is triangle, which can be defined by its three vertices (3 vectors). The following plot shows the reflection of a ray on a triangle.

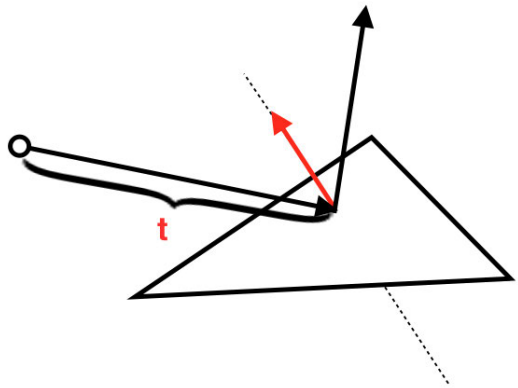


Figure C.2: Illustration of ray-triangle intersection.

C.3.3 Smoothed Triangle

Among all the basic objects, smoothed triangle is the most important one and can be used to approximate complicated 3D shapes. It is a curved surface with three vertices, whose curvature is defined a a linear interpolation of unit normal vectors

that collocates with the vertices. Given the coordinates (vectors) of these vertices and the unit normal vector at each vertex, we can approximate the normal vector at the incident point by interpolating the normals at vertices, using barycentric coordinates. Since the actual incident point is very close to the flat triangle, in order to simplify the model, we replace the actual incident point by the intersection of the extension of the ray with the flat triangle as determined by the vertices.

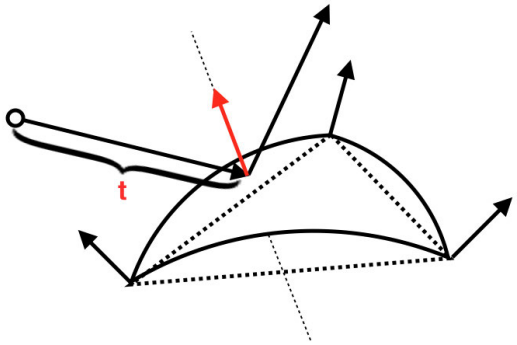


Figure C.3: Illustration of ray-smoothed triangle intersection.

C.3.4 Cylinder

A cylinder can be defined by either of the two ways: 1) by specifying the center of either its upper or lower surface, its radius, its height, and the direction of its axis; 2) by specifying the centers of both its upper and lower surfaces, and its radius. The cylinder primitive is particularly useful for visualizing chemical bonds between atoms.

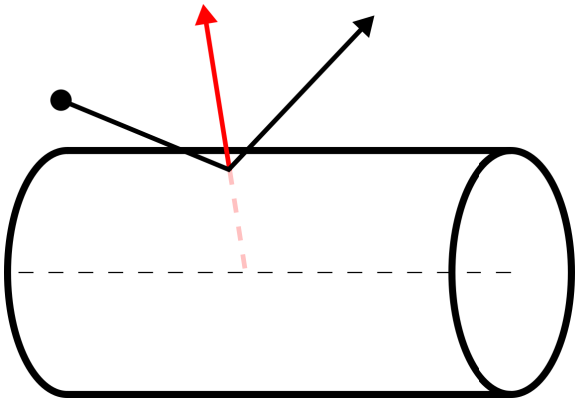


Figure C.4: Illustration of ray-cylinder intersection.

C.4 Directional Light

Directional lights are global illumination entities whose interactions with scene objects only depend on the angle of incidence but not on the absolute location. The amount of directional light absorbance for a given point on a surface is determined by the inner product between the rays that originate from that point and the direction of the directional light by

$$I_{\mathrm{d}} = \max(0, -e_{\mathrm{d}} \cdot e_{\mathrm{ray}})^{h}, \tag{C.1}$$

where h is a hardness factor controlling how fast the luminance decays if the ray does not align with the directional light.

C.5 Recursive Tracing Model and algorithm

In our render equation, after a ray hits the surface of an object, it may spawn several types of secondary rays. We consider three different kinds of secondary rays to account for the effects of specular reflection, diffusive reflection, and refractive reflection. The specular ray and the incident ray are symmetric about the normal line of the surface; the diffusive rays consists of multiple rays of random directions uniformly distributed in the hemisphere centered around the surface normal; the refractive ray lies on the other side of the incidence plane and is bent according to the Snell's law.

A recursive tracing algorithm is used to sum up the contribution of the secondary rays into the primary ray. Since each ray will generate even more rays after interacting a surface, a maximum depth of reflection is enforced in order to prevent the number of rays from exploding. A ray with a reflection count higher than a given threshold will not be able to generate any secondary rays regardless of its interaction with the scene. To further speed up computation, rays whose weight of contributions to the original primary ray fall below a certain threshold will also be discarded during the tracing process.

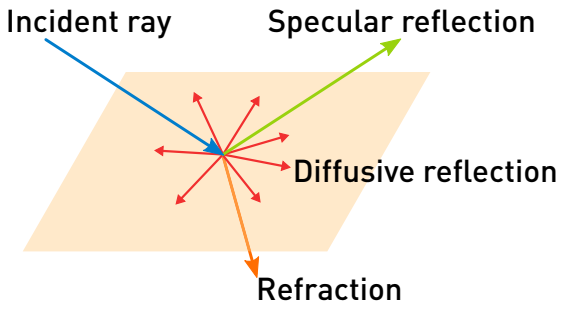


Figure C.5: Recursive ray tracing

C.6 Antialiasing

As with any rasterization algorithm, the breaking down of the continuous projection of the scene onto the image plane into discrete pixels introduces aliasing errors. In addition, the computation of soft shadows is built on top of Monte Carlo sampling, where the direction of reflecting rays are chosen stochastically from a given distribution. The combined effect is that the final image may contain significant noise that will negatively impact the appearance of the image. Therefore, antialiasing techniques were used to shoot multiple rays at randomly jittered subpixel locations within each pixel. The final color of each pixel is an average of the subpixel results. This can greatly reduce aliasing artifacts and improve visual quality of the image.

C.7 Parallelization

C.7.1 Parallel RNG

To spawn the diffusive rays, we need to generate many random rays with a uniform distribution on a disc in parallel. However, the built-in random number generator (RNG) in C++ cannot work in parallel. Therefore, we must set up independent RNG for each thread. We implemented the RNG class with high efficiency based on the RNG library in C++ 11.

C.7.2 Division of work

To parallelize the program, we adopted a hybrid approach using both MPI and OpenMP. We illustrate the parallelization scheme in Figure C.6 using an example image of 1920×1080 pixels. Here, a *tile* denotes a small piece of the picture, usually a block of 8×8 pixels, and is the smallest and unbreakable unit of task parallelization. The entire picture can be treated as a 1D array of tiles that expands in a row-major fashion. A *section* is a continuously collection of tiles, and is used to map the workloads onto MPI tasks. Thus, the number of sections always equals to the number of MPI ranks, and each MPI rank only takes care of only one section.

C.7.3 Load Balancing

The distribution of the objects in the scene may not project evenly onto the screen space. Thus, what might happen is that in some sections all rays will hit nothing but the background canvas, thus entailing minimal computation workload at runtime. Meanwhile, in some other sections, the rays may need to bounce many times between several objects, and the computation workload can be orders of

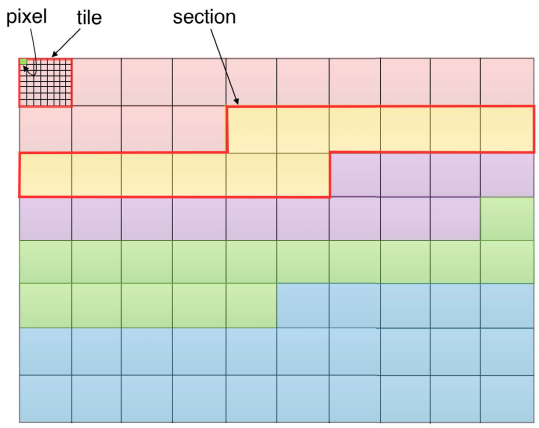


Figure C.6: Decomposition of image pixels among parallel tasks.

magnitude higher than tracing a ray that hit nothing. As a consequence, each pixel in the rendered image will consume dramatically different amounts of processor time, and the time spent for ray tracing will be determined by the slowest pixel without proper load balancing strategies. This could result in severe imbalance of the utilization of computation resource.

Therefore, we implemented a load balancing algorithm to dynamically adjust the partition of sections to optimize computational efficiency. Due to the need of antialiasing, the entire image will be traced several times, and this gives us the chance to re-partition the pixel sections among passes of antialiasing. To determine the adjusted size of the sections, we time the computation of each tile, and use the timing result to build a cumulative tile-wise CPU time distribution. Note this curve, as a function, is monotonically increasing and hence is guaranteed to be invertible. The start and end points of the partitions are then determined by evenly dividing the curve along the y-axis, and then finding the corresponding division points on the x-axis. In this way, each of the newly determined sections should consume an approximately equal amount of computation time. This process can be demonstrated by Figure C.7, where a total of 1024 tiles is divided into 4 section.

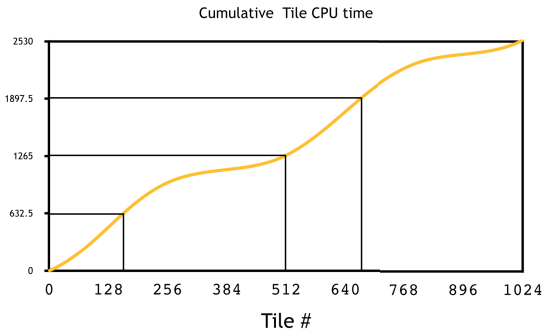


Figure C.7: Loading balancing using inverse cumulative timing.

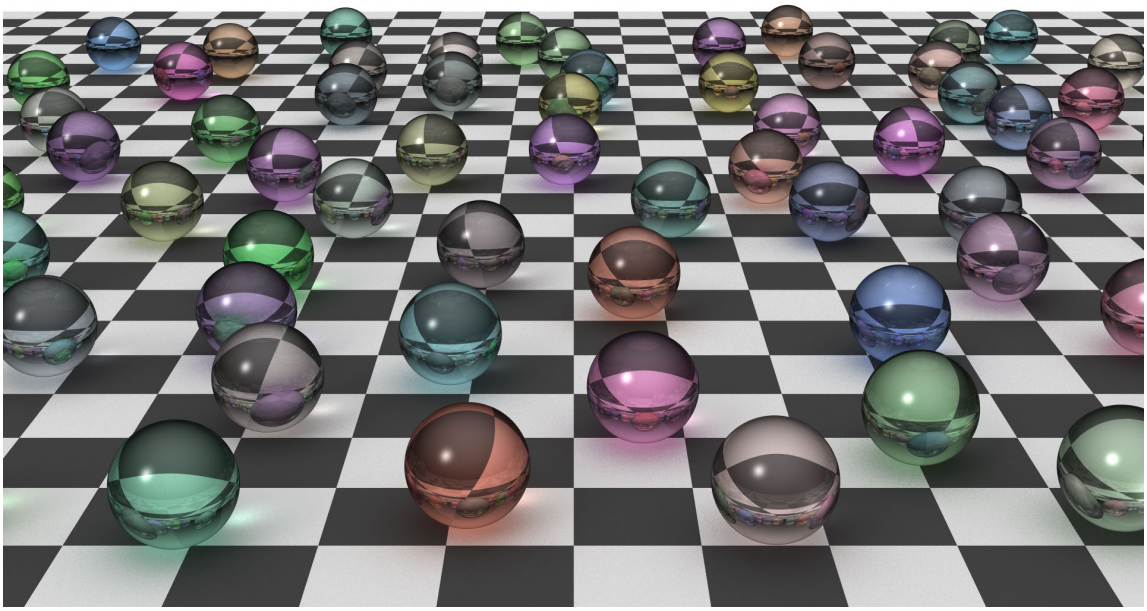
The re-partitioning process will typically reach a steady state after several iterations, where the program can then achieve near-optimal performance. The

pseudo-code for the load balancing algorithm is

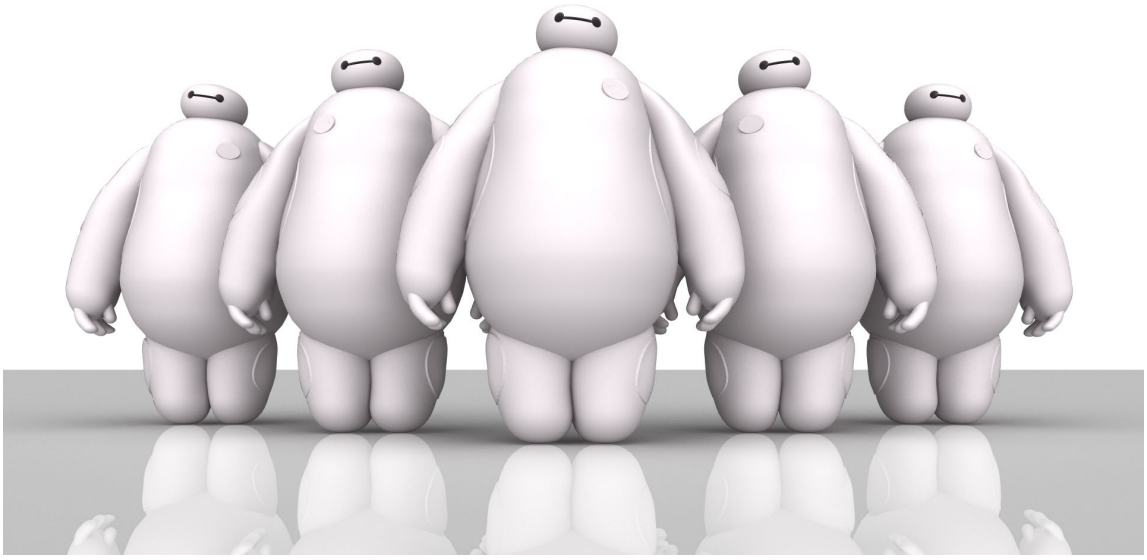
```
double times[total number of tiles];
for (int k = 0; k < num_antialiasing; k++) {
  define timer;
  for (int nt = tile_start; nt < tile_end; nt ++) {
    reset timer;
    Process image in tile nt;
    end timer;
    times[nt] = processing time of tile nt;
  }
  calculate tile_start and tile_end for my rank based on times;
}
```

C.8 Result and benchmark

Case 1: Glass Spheres on A Checkerboard



Case 2: Baymax



SMT efficiency: We ran this test on the Mira supercomputer located at the Argonne Leadership Computing Facility. Each node of the computer has 16 cores, and each core has 4 simultaneous hardware threads. From Figure C.8, we can see that the wall time of computation decreases with the number of threads significantly.

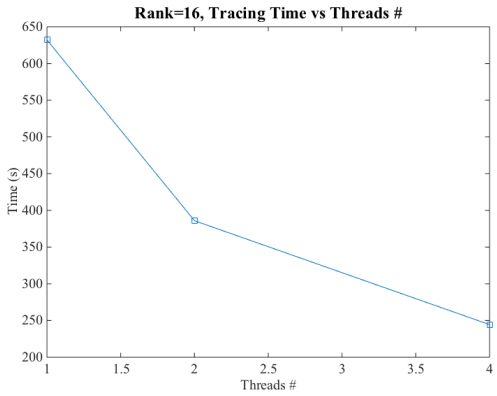


Figure C.8: SMT efficiency.

Load balancing: To test load balancing, we ran the program using 2048 ranks,

with and without load balancing enabled. From Figure C.9, we show that the load balance algorithm can dramatically reduce the running time.

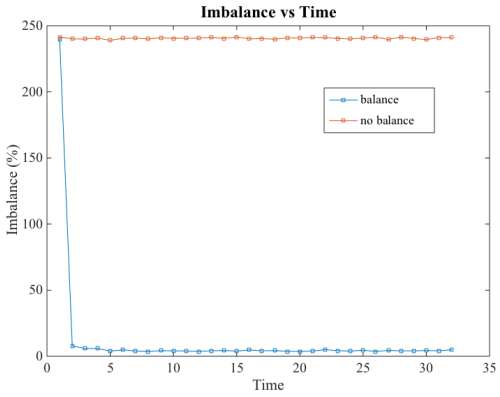


Figure C.9: Dynamic load balancing.

Strong scaling: We ran the program under different number of ranks from 16 to 16384. The figures below are the tracing time and wall time versus the number of ranks. We can see from Figure C.10 that:

1. The tracing time always decreases with the number of ranks;

2. The wall time decrease with the number of ranks at first. However, when the number of ranks is already large enough, the wall time will not decrease again. This is because the communication costs most of the wall time.

Overall, the tracer demonstrates good strong scaling efficiency.

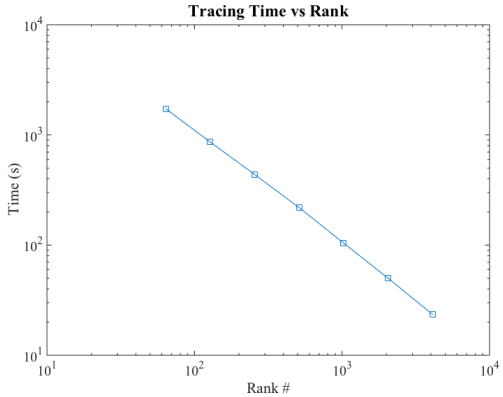


Figure C.10: Strong scaling.

C.9 Section Summary

We successfully implemented a parallel ray tracer, which can generate photo-realistic images using a hybrid MPI/OpenMPI parallel algorithm. The tracer scales strongly up to 10,000 MPI ranks with 4 threads per rank. A inverse-histogram based load balancing algorithm is the primary enabling technique for achieving this level of scalability. The tracer is light weight and hence can be incorporated easily into existing solvers for the purpose of in-situ visualization.

Bibliography

[1] Introducing Titan. `http://www.olcf.ornl.gov/titan/`.

[2] Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C Smith, Berk Hess, and Erik Lindahl. Gromacs: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 1:19–25, 2015.

[3] B J Alder and T E Wainwright. Studies in Molecular Dynamics. I. General Method Studies in Molecular Dynamics. I. General Method*. *The Journal of Chemical Physics Journal of Chemical Physics The Journal of Chemical Physics The Journal of Chemical Physics The Journal of Chemical Physics Journal of Chemical Physics*, 31(21):2384–1087, 1959.

[4] B.A. Allan, S. Lefantzi, and J. Ray. ODEPACK++: refactoring the lsode fortran library for use in the cca high performance component software architecture. In *High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings. Ninth International Workshop on*, pages 109–119, April 2004.

[5] Benjamin A. Allan, Robert C. Armstrong, Alicia P. Wolfe, Jaideep Ray, David E. Bernholdt, and James A. Kohl. The CCA core specification in a distributed memory SPMD framework. *Concurrency and Computation: Practice and Experience*, 14(5):323–345, 2002.

[6] Joshua A. Anderson, Chris D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, 227(10):5342 – 5359, 2008.

[7] Michel Arotçaréna, Bettina Heise, Sultana Ishaya, and André Laschewsky. Switching the inside and the outside of aggregates of water-soluble block copolymers with double thermoresponsivity. *Journal of the American Chemical Society*, 124(14):3787–3793, 2002.

[8] J. A. Backer, C. P. Lowe, H. C. J. Hoefsloot, and P. D. Iedema. Poiseuille flow to measure the viscosity of particle model fluids. *The Journal of Chemical Physics*, 122(15):154503, 2005.

[9] A.D. Bangham and R.W. Horne. Negative staining of phospholipids and their structural modification by surface-active agents as observed in the electron microscope. *Journal of Molecular Biology*, 8(5):660 – IN10, 1964.

[10] Albert P. Bartók, Risi Kondor, and Gábor Csányi. On representing chemical environments. *Physical Review B*, 87(18):184115, 2013.

[11] Albert P. Bartók, Mike C. Payne, Risi Kondor, and Gábor Csányi. Gaussian Approximation Potentials: The Accuracy of Quantum Mechanics, without the Electrons. *Physical Review Letters*, 104(13):136403, apr 2010.

[12] JONATHAN BARZILAI and JONATHAN M. BORWEIN. Two-Point Step Size Gradient Methods. *IMA Journal of Numerical Analysis*, 8(1):141–148, 1988.

[13] Jörg Behler. Atom-centered symmetry functions for constructing high-dimensional neural network potentials. *J. Chem. Phys. J. Chem. Phys. THE JOURNAL OF CHEMICAL PHYSICS*, 134(134), 2011.

[14] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[15] Martin Berzins, Justin Luitjens, Qingyu Meng, Todd Harman, Charles A. Wight, and Joseph R. Peterson. Uintah: A scalable framework for hazard analysis. In *Proceedings of the 2010 TeraGrid Conference*, TG '10, pages 3:1–3:8, New York, NY, USA, 2010. ACM.

[16] Timo Betz, Martin Lenz, Jean-François Joanny, and Cécile Sykes. Atp-dependent mechanics of red blood cells. *Proceedings of the National Academy of Sciences*, 106(36):15320–15325, 2009.

[17] Jürgen Blumm and André Lindemann. Characterization of the thermophysical properties of molten polymers and liquids using the flash technique. *High Temperatures-High Pressures*, 35(36):6, 2007.

[18] Sander Boonstra, Patrick R. Onck, and Erik van der Giessen. CHARMM TIP3P Water Model Suppresses Peptide Folding by Solvating the Unfolded State. *The Journal of Physical Chemistry B*, 120(15):3692–3698, apr 2016.

[19] V. Botu and R. Ramprasad. Learning scheme to predict atomic forces and accelerate materials simulations. 92, 2015.

[20] George EP Box and Mervin E Muller. A note on the generation of random normal deviates. *The Annals of Mathematical Statistics*, 29(2):610–611, 1958.

[21] Daniel Braun, Stefan Boresch, and Othmar Steinhauser. Transport and dielectric properties of water and the influence of coarse-graining: Comparing BMW, SPC/E, and TIP3P models. *The Journal of Chemical Physics*, 140(6):064107, feb 2014.

[22] W Michael Brown, Peng Wang, Steven J Plimpton, and Arnold N Tharrington. Implementing molecular dynamics on hybrid high performance computers - short range forces. *Computer Physics Communications*, 182(4):898–911, 2011.

[23] Hung-Yu Chang, Yung-Lung Lin, Yu-Jane Sheng, and Heng-Kwong Tsao. Multilayered polymersome formed by amphiphilic asymmetric macromolecular brushes. *Macromolecules*, 45(11):4778–4789, 2012.

[24] Anindya Chatterjee. An introduction to the proper orthogonal decomposition. *Current science*, 78(7):808–817, 2000.

[25] Zhen Chen, Shan Jiang, Yong Gan, Hantao Liu, and Thomas D Sewell. A particle-based multiscale simulation procedure within the material point method framework. *Computational Particle Mechanics*, 1(2):147–158, 2014.

[26] Tao Cheng, Andrés Jaramillo-Botero, William A Goddard, and Huai Sun. Adaptive Accelerated ReaxFF Reactive Dynamics with Validation from Simulating Hydrogen Combustion. *Journal of the American Chemical Society*, 136(26):9434–9442, jul 2014.

[27] R R Coifman, S Lafon, A B Lee, M Maggioni, B Nadler, F Warner, and S W Zucker. Geometric diffusions as a tool for harmonic analysis and structure definition of data: diffusion maps. *Proceedings of the National Academy of Sciences of the United States of America*, 102(21):7426–31, may 2005.

[28] F-Xabier Contreras, Lissete Sánchez-Magraner, Alicia Alonso, and Félix M Goñi. Transbilayer (flip-flop) lipid motion and lipid scrambling in membranes. *FEBS letters*, 584(9):1779–1786, 2010.

[29] Wendy D. Cornell, Piotr Cieplak, Christopher I. Bayly, Ian R. Gould, Kenneth M. Merz, David M. Ferguson, David C. Spellmeyer, Thomas Fox, James W. Caldwell, and Peter A. Kollman. A second generation force field for the simulation of proteins, nucleic acids, and organic molecules. *Journal of the American Chemical Society*, 117(19):5179–5197, may 1995.

[30] J. Davison de St.Germain, J. McCorquodale, S.G. Parker, and C.R. Johnson. Uintah: a massively parallel problem solving environment. In *High-Performance Distributed Computing, 2000. Proceedings. The Ninth International Symposium on*, pages 33–41, 2000.

[31] Rafael Delgado-Buscalioni, Kurt Kremer, and Matej Praprotnik. Concurrent triple-scale simulation of molecular liquids. *The Journal of Chemical Physics*, 128(11):114110, 2008.

[32] Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the royal statistical society. Series B (methodological)*, pages 1–38, 1977.

[33] Herbert Edelsbrunner. Voronoi and delaunay diagrams. In *A Short Course in Computational Geometry and Topology*, pages 9–15. Springer, 2014.

[34] Mahmoud Elsabahy and Karen L Wooley. Design of polymeric nanoparticles for biomedical delivery applications. *Chemical Society Reviews*, 41(7):2545–2561, 2012.

[35] Pep Espanol and Patrick Warren. Statistical mechanics of dissipative particle dynamics. *EPL (Europhysics Letters)*, 30(4):191, 1995.

[36] Evan A Evans. Bending resistance and chemically induced moments in membrane bilayers. *Biophysical journal*, 14(12):923, 1974.

[37] James Evans, Walter Gratzer, Narla Mohandas, Kim Parker, and John Sleep. Fluctuations of the red blood cell membrane: relation to mechanical properties and lack of atp dependence. *Biophysical journal*, 94(10):4134–4144, 2008.

[38] Dmitry A Fedosov, Huan Lei, Bruce Caswell, Subra Suresh, and George E Karniadakis. Multiscale modeling of red blood cell mechanics and blood flow in malaria. *PLoS Comput Biol*, 7(12):e1002270, 2011.

[39] Dmitry A. Fedosov, Wenxiao Pan, Bruce Caswell, Gerhard Gompper, and George E. Karniadakis. Predicting human blood viscosity in silico. *Proceedings of the National Academy of Sciences*, 108(29):11772–11777, 2011.

[40] Scott E Feller. Molecular dynamics simulations of lipid bilayers. *Current opinion in colloid & interface science*, 5(3):217–223, 2000.

[41] Feng Feng and William S Klug. Finite element modeling of lipid bilayer membranes. *Journal of Computational Physics*, 220(1):394–408, 2006.

[42] E Fermi, J Pasta, S Ulam, and M Tsingou. Los Alamos report LA-1940: Studies of Nonlinear Problems. I. Technical report, 1955.

[43] Grégoire Ferré, Terry Haut, and Kipton Barros. Learning molecular energies using localized graph kernels. *The Journal of Chemical Physics*, 2017.

[44] Daan Frenkel and Berend Smit. *Understanding molecular simulation: from algorithms to applications*, volume 1. Academic press, 2001.

[45] Mark S. Friedrichs, Peter Eastman, Vishal Vaidyanathan, Mike Houston, Scott Legrand, Adam L. Beberg, Daniel L. Ensign, Christopher M. Bruns, and Vijay S. Pande. Accelerating molecular dynamic simulation on graphics processing units. *Journal of Computational Chemistry*, 30(6):864–872, 2009.

[46] S-P Fu, Z Peng, H Yuan, R Kfoury, and Y-N Young. Lennard-jones type pair-potential method for coarse-grained lipid bilayer membrane simulations in lammps. *Computer Physics Communications*, 210:193–203, 2017.

[47] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Yao Zhang, and V. Volkov. Parallel computing experiences with cuda. *Micro, IEEE*, 28(4):13–27, 2008.

[48] A. A. Gavrilov, A. V. Chertovich, and E. Yu. Kramarenko. Conformational behavior of a single polyelectrolyte chain with bulky counterions. *Macromolecules*, 49(3):1103–1110, 2016.

[49] Benoit Gibert and David Mainprice. Effect of crystal preferred orientations on the thermal diffusivity of quartz polycrystalline aggregates at high temperature. *Tectonophysics*, 465(1):150–163, 2009.

[50] N. Goga, S. Marrink, R. Cioromela, and F. Moldoveanu. Gpu-sd and dpd parallelization for gromacs tools for molecular dynamics simulations. In *Bioinformatics Bioengineering (BIBE), 2012 IEEE 12th International Conference on*, pages 251–254, 2012.

[51] Andreas W Götz, Mark J Williamson, Dong Xu, Duncan Poole, Scott Le Grand, and Ross C Walker. Routine Microsecond Molecular Dynamics Simulations with AMBER on GPUs. 1. Generalized Born. *Journal of Chemical Theory and Computation*, 8(5):1542, 2012.

[52] Scott Le Grand, Andreas W Götzx, and Ross C Walker. SPFP: Speed without compromise - a mixed precision model for gpu accelerated molecular dynamics simulations. *Computer Physics Communications*, 2012.

[53] Robert D Groot and KL Rabone. Mesoscopic simulation of cell membrane damage, morphology change and rupture by nonionic surfactants. *Biophysical journal*, 81(2):725–736, 2001.

[54] Robert D Groot and Patrick B Warren. Dissipative particle dynamics: Bridging the gap between atomistic and mesoscopic simulation. *The Journal of chemical physics*, 107:4423, 1997.

[55] Jonathan L. Gross and Jay Yellen. *Graph theory and its applications*. Chapman & Hall/CRC, 2005.

[56] Yuanyuan Han, Haizhou Yu, Hongbo Du, and Wei Jiang. Effect of selective solvent addition rate on the pathways for spontaneous vesicle formation of aba amphiphilic triblock copolymers. *Journal of the American Chemical Society*, 132(3):1144–1150, 2009.

[57] John F Hart. *Computer approximations*. Krieger Publishing Co., Inc., 1978.

[58] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[59] Wolfgang Helfrich. Elastic properties of lipid bilayers: theory and possible experiments. *Zeitschrift für Naturforschung C*, 28(11-12):693–703, 1973.

[60] Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2010. Version 1.7.0.

[61] Bingbing Hong, Feng Qiu, Hongdong Zhang, and Yuliang Yang. Dissipative particle dynamics simulations on inversion dynamics of spherical micelles. *The Journal of Chemical Physics*, 132(24):244901, 2010.

[62] P. J. Hoogerbrugge and J. M. V. A. Koelman. Simulating microscopic hydrodynamic phenomena with dissipative particle dynamics. *EPL (Europhysics Letters)*, 19(3):155, 1992.

[63] John F Hughes, Steven K Feiner, James D Foley, Kurt Akeley, Morgan McGuire, Andries van Dam, and David F Sklar. Computer graphics: principles and practice. 2013.

[64] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[65] Florian D Jochum and Patrick Theato. Temperature-and light-responsive smart polymer materials. *Chemical Society Reviews*, 42(17):7468–7483, 2013.

[66] Marie-Christine Jones and Jean-Christophe Leroux. Polymeric micelles a new generation of colloidal drug carriers. *European Journal of Pharmaceutics and Biopharmaceutics*, 48(2):101 – 111, 1999.

[67] Laxmikant Kal, Robert Skeel, Milind Bhandarkar, Robert Brunner, Attila Gursoy, Neal Krawetz, James Phillips, Aritomo Shinozaki, Krishnan Varadarajan, and Klaus Schulten. Namd2: Greater scalability for parallel molecular dynamics. *Journal of Computational Physics*, 151(1):283 – 312, 1999.

[68] Smita Kashyap and Manickam Jayakannan. Thermo-responsive and shape transformable amphiphilic scaffolds for loading and delivering anticancer drugs. *Journal of Materials Chemistry B*, 2(26):4142–4152, 2014.

[69] Alireza Khorshidi and Andrew A. Peterson. Amp: A modular approach to machine learning in atomistic simulations. *Computer Physics Communications*, 207:310–324, 2016.

[70] Sun Dal Kim, Sang Youl Kim, and Im Sik Chung. Unprecedented lower critical solution temperature behavior of polyimides in organic media. *Macromolecules*, 47(24):8846–8849, 2014.

[71] Yuzo Kitazawa, Takeshi Ueki, Lucas D. McIntosh, Saki Tamura, Kazuyuki Niitsuma, Satoru Imaizumi, Timothy P. Lodge, and Masayoshi Watanabe. Hierarchical solgel transition induced by thermosensitive self-assembly of an abc triblock polymer in an ionic liquid. *Macromolecules*, 49(4):1414–1423, 2016.

[72] Christopher M Kolodziej and Heather D Maynard. Shape-shifting micro- and nanopatterns controlled by temperature. *Journal of the American Chemical Society*, 134(30):12386–12389, 2012.

[73] Joseph Kost and Robert Langer. Responsive polymeric delivery systems. *Advanced Drug Delivery Reviews*, 64, Supplement:327 – 341, 2012.

[74] L. B. Lucy. A numerical approach to the testing of the fission hypothesis. *THE ASTRONOMICAL JOURNAL*, 82(12), 1977.

[75] Mohamed Laradji and P. B. Sunil Kumar. Dynamics of domain growth in self-assembled fluid vesicles. *Phys. Rev. Lett.*, 93:198105, Nov 2004.

[76] Andrew R Leach. *Molecular modelling: principles and applications*. Pearson education, 2001.

[77] DN Lebedev, VI and Laikov. A quadrature formula for the sphere of the 131st algebraic order of accuracy. *Doklady. Mathematics*, 59(3):477–481, 1999.

[78] S. Lefantzi, J. Ray, and H.N. Najm. Using the common component architecture to design high performance scientific simulation codes. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 10 pp.–, April 2003.

[79] Huan Lei and George Em Karniadakis. Predicting the morphology of sickle red blood cells using coarse-grained models of intracellular aligned hemoglobin polymers. *Soft matter*, 8(16):4507–4516, 2012.

[80] B. Leistedt and J. D. McEwen. Exact Wavelets on the Ball. *IEEE Transactions on Signal Processing*, 60(12):6257–6269, 2012.

[81] He Li and George Lykotrafitis. Two-component coarse-grained molecular-dynamics model for the human erythrocyte membrane. *Biophysical journal*, 102(1):75–84, 2012.

[82] He Li and George Lykotrafitis. Erythrocyte membrane model with explicit description of the lipid bilayer and the spectrin network. *Biophysical journal*, 107(3):642–653, 2014.

[83] He Li and George Lykotrafitis. Vesiculation of healthy and defective red blood cells. *Physical Review E*, 92(1):012715, 2015.

[84] He Li, Yihao Zhang, Vi Ha, and George Lykotrafitis. Modeling of band-3 protein diffusion in the normal and defective red blood cell membrane. *Soft matter*, 12(15):3643–3653, 2016.

[85] Xuejin Li, E Du, Huan Lei, Yu-Hang Tang, Ming Dao, Subra Suresh, and George Em Karniadakis. Patient-specific blood rheology in sickle-cell anaemia. *Interface focus*, 6(1):20150065, 2016.

[86] Xuejin Li, He Li, Hung-Yu Chang, George Lykotrafitis, and George Em Karniadakis. Computational biomechanics of human red blood cells in hematological disorders. *Journal of Biomechanical Engineering*, 2016.

[87] Xuejin Li, Igor V. Pivkin, Haojun Liang, and George Em Karniadakis. Shape transformations of membrane vesicles from amphiphilic triblock copolymers: A dissipative particle dynamics simulation study. *Macromolecules*, 42(8):3195–3200, 2009.

[88] Zhen Li, Yu-Hang Tang, Huan Lei, Bruce Caswell, and George Em Karniadakis. Energy-conserving dissipative particle dynamics with temperature-dependent properties. *Journal of Computational Physics*, 265(0):113–127, 2014.

[89] Zhen Li, Yu-Hang Tang, Xuejin Li, and George Em Karniadakis. Mesoscale modeling of phase transition dynamics of thermoresponsive polymers. *Chem. Commun.*, 51:11038–11040, 2015.

[90] Zhenlong Li and Elena E. Dormidontova. Kinetics of diblock copolymer micellization by dissipative particle dynamics. *Macromolecules*, 43(7):3521–3531, 2010.

[91] Zhenwei Li, James R. Kermode, and Alessandro De Vita. Molecular Dynamics with On-the-Fly Machine Learning of Quantum-Mechanical Forces. *Physical Review Letters*, 114(9):096405, mar 2015.

[92] Xing Liang, Fei Liu, Veronika Kozlovskaya, Zachary Palchak, and Eugenia Kharlampieva. Thermoresponsive micelles from double lcst-poly (3-methyl-n-vinylcaprolactam) block copolymers for cancer therapy. *ACS Macro Letters*, 4(3):308–311, 2015.

[93] Kresten Lindorff-Larsen, Paul Maragakis, Stefano Piana, and David E. Shaw. Picosecond to Millisecond Structural Dynamics in Human Ubiquitin. *The Journal of Physical Chemistry B*, 120(33):8313–8320, aug 2016.

[94] Pierre-Louis Lions. On the Schwarz alternating method. i. In *First international symposium on domain decomposition methods for partial differential equations*, pages 1–42. Paris, France, 1988.

[95] Fei Liu, Veronika Kozlovskaya, Srikanth Medipelli, Bing Xue, Fahim Ahmad, Mohammad Saeed, Donald Cropek, and Eugenia Kharlampieva. Temperature-sensitive polymersomes for controlled delivery of anticancer drugs. *Chemistry of Materials*, 27(23):7945–7956, 2015.

[96] Futian Liu and Adi Eisenberg. Preparation and ph triggered inversion of vesicles from poly (acrylic acid)-b lock-polystyrene-b lock-poly (4-vinyl pyridine). *Journal of the American Chemical Society*, 125(49):15059–15064, 2003.

[97] M.B. Liu, G.R. Liu, and K.Y. Lam. Constructing smoothing functions in smoothed particle hydrodynamics with applications. *Journal of Computational and Applied Mathematics*, 155(2):263–284, jun 2003.

[98] Weiguo Liu, Bertil Schmidt, Gerrit Voss, and Wolfgang Mller-Wittig. Accelerating molecular dynamics simulations using graphics processing units with CUDA. *Computer Physics Communications*, 179(9):634 – 641, 2008.

[99] Ryan Longenecker, Tingting Mu, Mark Hanna, Nicholas A. D. Burke, and Harald D. H. Stver. Thermally responsive 2-hydroxyethyl methacrylate polymers: Solubleinsoluble and solubleinsolublesoluble transitions. *Macromolecules*, 44(22):8962–8971, 2011.

[100] Weicong Mai, Bin Sun, Luyi Chen, Fei Xu, Hao Liu, Yeru Liang, Ruowen Fu, Dingcai Wu, and Krzysztof Matyjaszewski. Water-dispersible, responsive, and carbonizable hairy microporous polymeric nanospheres. *Journal of the American Chemical Society*, 137(41):13256–13259, 2015.

[101] LoisCurfman McInnes, BenjaminA. Allan, Robert Armstrong, StevenJ. Benson, DavidE. Bernholdt, TamaraL. Dahlgren, LoriFreitag Diachin, Manojkumar Krishnan, JamesA. Kohl, J.Walter Larson, Sophia Lefantzi, Jarek Nieplocha, Boyana Norris, StevenG. Parker, Jaideep Ray, and Shujia Zhou. Parallel PDE-based simulations using the common component architecture. In AreMagnus Bruaset and Aslak Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51 of *Lecture Notes in Computational Science and Engineering*, pages 327–381. Springer Berlin Heidelberg, 2006.

[102] KM Mohamed and AA Mohamad. A review of the development of hybrid atomistic–continuum methods for dense fluids. *Microfluidics and Nanofluidics*, 8(3):283–302, 2010.

[103] J.J. Monaghan. Smoothed particle hydrodynamics and its diverse applications. *Annual Review of Fluid Mechanics*, 44(1):323–346, 2012.

[104] Hazime Mori. Transport, collective motion, and brownian motion*. *Progress of theoretical physics*, 33(3):423–455, 1965.

[105] Hideharu Mori, Ikumi Kato, Shoko Saito, and Takeshi Endo. Proline-based block copolymers displaying upper and lower critical solution temperatures. *Macromolecules*, 43(3):1289–1298, 2010.

[106] Lauri Mkinen, Divya Varadharajan, Heikki Tenhu, and Sami Hietala. Triple hydrophilic ucstlcst block copolymers. *Macromolecules*, 49(3):986–993, 2016.

[107] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, March 2008.

[108] Mu-Ping Nieh, Paul Dolinar, Norbert Kuerka, Steven R. Kline, Lisa M. Debeer-Schmitt, Kenneth C. Littrell, and John Katsaras. Formation of kinetically trapped nanoscopic unilamellar vesicles from metastable nanodiscs. *Langmuir*, 27(23):14308–14316, 2011.

[109] Robert Osada, Thomas Funkhouser, Bernard Chazelle, and David Dobkin. Shape distributions. *ACM Transactions on Graphics (TOG)*, 21(4):807–832, 2002.

[110] YongKeun Park, Monica Diez-Silva, Gabriel Popescu, George Lykotrafitis, Wonshik Choi, Michael S Feld, and Subra Suresh. Refractive index maps and membrane dynamics of human red blood cells parasitized by plasmodium falciparum. *Proceedings of the National Academy of Sciences*, 105(37):13730–13735, 2008.

[111] Steven G. Parker. A component-based architecture for parallel multi-physics PDE simulation. *Future Generation Computer Systems*, 22(12):204 – 216, 2006.

[112] Alexander Patronis and Duncan A Lockerby. Multiscale simulation of non-isothermal microchannel gas flows. *Journal of Computational Physics*, 270:532–543, 2014.

[113] Zhangli Peng, Xuejin Li, Igor V Pivkin, Ming Dao, George E Karniadakis, and Subra Suresh. Lipid bilayer and cytoskeletal interactions in a red blood cell. *Proceedings of the National Academy of Sciences*, 110(33):13356–13361, 2013.

[114] Carolyn L. Phillips, Joshua A. Anderson, and Sharon C. Glotzer. Pseudo-random number generation for Brownian Dynamics and Dissipative Particle Dynamics simulations on GPU devices. *Journal of Computational Physics*, 230(19):7191 – 7201, 2011.

[115] S Plimpton, P Crozier, and A Thompson. LAMMPS-large-scale atomic/-molecular massively parallel simulator. *Sandia National Laboratories*, 2007.

[116] Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117(1):1–19, 1995.

[117] Thomas R Powers, Greg Huber, and Raymond E Goldstein. Fluid-membrane tethers: minimal surfaces and elastic boundary layers. *Physical Review E*, 65(4):041901, 2002.

[118] Matej Praprotnik, Luigi Delle Site, and Kurt Kremer. Multiscale simulation of soft matter: From scale bridging to adaptive resolution. *Annu. Rev. Phys. Chem.*, 59:545–571, 2008.

[119] Sander Pronk, Szilrd Pll, Roland Schulz, Per Larsson, Pr Bjelkmar, Rossen Apostolov, Michael R. Shirts, Jeremy C. Smith, Peter M. Kasson, David van der Spoel, Berk Hess, and Erik Lindahl. Gromacs 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. *Bioinformatics*, 29(7):845–854, 2013.

[120] Philip Rabinowitz and George Weiss. Tables of Abscissas and Weights for Numerical e^{-x} x^n f(x) dx.

[121] A Rahman. Correlations in the Motion of Atoms in Liquid Argon*.

[122] F. Rampf, W. Paul, and K. Binder. On the first-order collapse transition of a three-dimensional, flexible homopolymer chain model. *EPL (Europhysics Letters)*, 70(5):628, 2005.

[123] A. K. Rappe, C. J. Casewit, K. S. Colwell, W. A. Goddard, and W. M. Skiff. UFF, a full periodic table force field for molecular mechanics and molecular dynamics simulations. *Journal of the American Chemical Society*, 114(25):10024–10035, dec 1992.

[124] C E Rasmussen, C K I Williams, Richard S Sutton, Andrew G Barto, Peter Spirtes, Clark Glymour, Richard Scheines, Bernhard Schölkopf, and Alexander J Smola. Gaussian Processes for Machine Learning. 2006.

[125] Jamie A Reed, Adrianne E Lucero, Steve Hu, Linnea K Ista, Mangesh T Bore, Gabriel P López, and Heather E Canavan. A low-cost, rapid deposition method for smart films: applications in mammalian cell release. *ACS Applied Materials & Interfaces*, 2(4):1048–1051, 2010.

[126] Diego Rossinelli, Yu-Hang Tang, Kirill Lykov, Dmitry Alexeev, Massimo Bernaschi, Panagiotis Hadjidoukas, Mauro Bisson, Wayne Joubert, Christian Conti, George Karniadakis, Massimiliano Fatica, Igor Pivkin, and Petros Koumoutsakos. The in-silico lab-on-a-chip: Petascale and high-throughput simulations of microfluidics at cell resolution. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 2:1–2:12, New York, NY, USA, 2015. ACM.

[127] Debashish Roy, William LA Brooks, and Brent S Sumerlin. New directions in thermoresponsive polymers. *Chemical Society Reviews*, 42(17):7214–7243, 2013.

[128] Matthias Rupp, Alexandre Tkatchenko, Klaus-Robert Robert Müller, O Anatole Von Lilienfeld, and O. Anatole von Lilienfeld. Fast and Accurate Modeling of Molecular Atomization Energies with Machine Learning. *Physical Review Letters*, 108(5):058301, jan 2012.

[129] Leonard M. C. Sagis and Hans Christian Öttinger. Dynamics of multiphase systems with complex microstructure. I. Development of the governing equations through nonequilibrium thermodynamics. *Phys. Rev. E*, 88:022149, Aug 2013.

[130] Leonor Saiz, Sanjoy Bandyopadhyay, and Michael L Klein. Towards an understanding of complex biological membranes from atomistic molecular dynamics simulations. *Bioscience reports*, 22(2):151–173, 2002.

[131] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10, 2009.

[132] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D Owens. Scan primitives for GPU computing. In *Graphics Hardware*, volume 2007, pages 97–106, 2007.

[133] Ronald A Siegel. Stimuli sensitive polymers and self regulated drug delivery systems: a very partial review. *Journal of Controlled Release*, 190:337–351, 2014.

[134] Leonardo Di G Sigalotti, Jaime Klapp, Eloy Sira, Yasmin Meleán, and Anwar Hasmy. SPH simulations of time-dependent Poiseuille flow at low Reynolds numbers. *Journal of Computational Physics*, 191(2):622–638, 2003.

[135] Eric A Simone, Thomas D Dziubla, and Vladimir R Muzykantov. Polymeric carriers: role of geometry in drug delivery. *Expert Opinion on Drug Delivery*, 5(12):1283–1300, 2008.

[136] W. Smith and T.R. Forester. DL_POLY_2.0: A general-purpose parallel molecular dynamics simulation package. *Journal of Molecular Graphics*, 14(3):136 – 141, 1996.

[137] D.F. Specht. A general regression neural network. *IEEE Transactions on Neural Networks*, 2(6):568–576, 1991.

[138] Randy S Sprague, Mary L Ellsworth, Alan H Stephenson, Mary E Kleinhenz, and Andrew J Lonigro. Deformation-induced atp release from red blood cells requires cftr activity. *American Journal of Physiology-Heart and Circulatory Physiology*, 275(5):H1726–H1732, 1998.

[139] WJ Starke, J Stuecheli, DM Daly, JS Dodson, F Auernhammer, PM Sagmeister, GL Guthrie, CF Marino, M Siegel, and B Blaner. The cache and memory subsystems of the IBM POWER8 processor. *IBM Journal of Research and Development*, 59(1):3–1, 2015.

[140] John E. Stone, James C. Phillips, Peter L. Freddolino, David J. Hardy, Leonardo G. Trabuco, and Klaus Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*, 28(16):2618–2640, 2007.

[141] Hong Yang Sun. *Learning over Molecules : Representations and Kernels*. PhD thesis, Harvard University, 2014.

[142] I-Jui Sung, G.D. Liu, and W.-M.W. Hwu. DL: A data layout transformation system for heterogeneous computing. In *Innovative Parallel Computing (InPar), 2012*, pages 1–11, 2012.

[143] William C. Swope, Hans C. Andersen, Peter H. Berens, and Kent R. Wilson. A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters. *The Journal of Chemical Physics*, 76(1):637–649, 1982.

[144] Yu-Hang Tang and George Em Karniadakis. Accelerating dissipative particle dynamics simulations on GPUs: Algorithms, numerics and applications. *Computer Physics Communications*, 185(11):2809–2822, 2014.

[145] Yu-Hang Tang, Shuhei Kudo, Xin Bian, Zhen Li, and George Em Karniadakis. Multiscale universal interface: A concurrent framework for coupling heterogeneous solvers. *Journal of Computational Physics*, 297:13–31, 2015.

[146] D Peter Tieleman, Siewert-Jan Marrink, and Herman JC Berendsen. A computer perspective of membranes: molecular dynamics studies of lipid bilayer systems. *Biochimica et Biophysica Acta (BBA)-Reviews on Biomembranes*, 1331(3):235–270, 1997.

[147] Brian P. Timko, Manuel Arruebo, Sahadev A. Shankarappa, J. Brian McAlvin, Obiajulu S. Okonkwo, Boaz Mizrahi, Cristina F. Stefanescu, Leyre Gomez, Jia Zhu, Angela Zhu, Jesus Santamaria, Robert Langer, and Daniel S. Kohane. Near-infrared-actuated devices for remotely controlled drug deliverys. *Proceedings of the National Academy of Sciences*, 111(4):1349–1354, 2014.

[148] Kechuan Tu, Michael L Klein, and Douglas J Tobias. Constant-pressure molecular dynamics investigation of cholesterol effects in a dipalmitoylphosphatidylcholine bilayer. *Biophysical journal*, 75(5):2147–2156, 1998.

[149] Viji Mary Varghese, Vidya Raj, K Sreenivasan, and TV Kumary. In vitro cytocompatibility evaluation of a thermoresponsive nipaam-mma copolymeric surface using l929 cells. *Journal of Materials Science: Materials in Medicine*, 21(5):1631–1639, 2010.

[150] Jens H. Walther, Matej Praprotnik, Evangelos M. Kotsalis, and Petros Koumoutsakos. Multiscale simulation of water flow past a C540 fullerene. *Journal of Computational Physics*, 231(7):2677 – 2681, 2012.

[151] Di Wang, Jun Yin, Zhiyuan Zhu, Zhishen Ge, Hewen Liu, Steven P. Armes, and Shiyong Liu. Micelle formation and inversion kinetics of a schizophrenic diblock copolymer. *Macromolecules*, 39(21):7378–7385, 2006.

[152] Sibo Wang, Junbo Xu, and Hao Wen. Accelerating dissipative particle dynamics with multiple {GPUs}. *Computer Physics Communications*, 184(11):2454 – 2461, 2013.

[153] Mark A Ward and Theoni K Georgiou. Thermoresponsive terpolymers based on methacrylate monomers: Effect of architecture and composition. *Journal of Polymer Science Part A: Polymer Chemistry*, 48(4):775–783, 2010.

[154] Peifa Wei, Timothy R. Cook, Xuzhou Yan, Feihe Huang, and Peter J. Stang. A discrete amphiphilic organoplatinum(ii) metallacycle with tunable lower critical solution temperature behavior. *Journal of the American Chemical Society*, 136(44):15497–15500, 2014.

[155] E Weinan. *Principles of multiscale modeling*. Cambridge University Press, 2011.

[156] E Weinan, Bjorn Engquist, and Zhongyi Huang. Heterogeneous multiscale method: a general methodology for multiscale modeling. *Physical Review B*, 67(9):092101, 2003.

[157] David J Wheeler and Roger M Needham. TEA, a tiny encryption algorithm. In *Fast Software Encryption*, pages 363–366. Springer, 1995.

[158] * William L. Jorgensen, , David S. Maxwell, and Julian Tirado-Rives. Development and Testing of the OPLS All-Atom Force Field on Conformational Energetics and Properties of Organic Liquids. 1996.

[159] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 235–246, 2010.

[160] Katherine C Wood, Miriam M Cortese-Krott, Jason C Kovacic, Audrey Noguchi, Virginia B Liu, Xunde Wang, Nalini Raghavachari, Manfred Boehm, Gregory J Kato, Malte Kelm, et al. Circulating blood endothelial nitric oxide synthase contributes to the regulation of systemic blood pressure and nitrite homeostasis. *Arteriosclerosis, thrombosis, and vascular biology*, 33(8):1861–1871, 2013.

[161] Chi Wu and Shuiqin Zhou. Laser light scattering study of the phase transition of poly(n-isopropylacrylamide) in water. 1. single chain. *Macromolecules*, 28(24):8381–8387, 1995.

[162] Hao Wu, JB Xu, SF Zhang, and Hao Wen. GPU accelerated dissipative particle dynamics with parallel cell-list updating. *IEIT Journal of Adaptive & Dynamic Computing*, 2011(2):26–32, 2011.

[163] Hsing-Lun Wu, Yu-Jane Sheng, and Heng-Kwong Tsao. Phase behaviors and membrane properties of model liposomes: Temperature effect. *The Journal of Chemical Physics*, 141(12), 2014.

[164] Jinchao Xu. Iterative methods by space decomposition and subspace correction. *SIAM review*, 34(4):581–613, 1992.

[165] Xin Yong, Olga Kuksenok, Krzysztof Matyjaszewski, and Anna C. Balazs. Harnessing interfacially-active nanorods to regenerate severed polymer gels. *Nano Letters*, 13(12):6269–6274, 2013.

[166] Fahad Zafar, Marc Olano, and Aaron Curtis. GPU random numbers via the tiny encryption algorithm. In *Proceedings of the Conference on High Performance Graphics*, pages 133–141. Eurographics Association, 2010.

[167] Zhanpeng Zhang, Ryan L. Marson, Zhishen Ge, Sharon C. Glotzer, and Peter X. Ma. Simultaneous nano- and microscale control of nanofibrous microspheres self-assembled from star-shaped polymers. *Advanced Materials*, 27(26):3947–3952, 2015.

[168] Zhenjiang Zhang, Jing Wang, Xin Nie, Tao Wen, Yinglu Ji, Xiaochun Wu, Yuliang Zhao, and Chunying Chen. Near infrared laser-induced targeted cancer therapy using thermoresponsive polymer encapsulated gold nanorods. *Journal of the American Chemical Society*, 136(20):7317–7326, 2014.

[169] Gongpu Zhao, Juan R Perilla, Ernest L Yufenyuy, Xin Meng, Bo Chen, Jiying Ning, Jinwoo Ahn, Angela M Gronenborn, Klaus Schulten, Christopher Aiken, and Others. Mature HIV-1 capsid structure by cryo-electron microscopy and all-atom molecular dynamics. *Nature*, 497(7451):643–646, 2013.

[170] Yannan Zhao, Xiaoxing Sun, Guannan Zhang, Brian G Trewyn, Igor I Slowing, and Victor S-Y Lin. Interaction of mesoporous silica nanoparticles with human red blood cell membranes: size and surface effects. *ACS nano*, 5(2):1366–1375, 2011.

[171] Yicheng Zhu, Rhiannon Batchelor, Andrew B. Lowe, and Peter J. Roth. Design of thermoresponsive polymers with aqueous lcst, ucst, or both: Modification of a reactive poly(2-vinyl-4,4-dimethylazlactone) scaffold. *Macromolecules*, 49(2):672–680, 2016.

[172] Robert Zwanzig. *Nonequilibrium statistical mechanics*. Oxford University Press, 2001.