ACCELERATED IN-SITU WORKFLOW OF MEMORY-AWARE LATTICE BOLTZMANN SIMULATION AND ANALYSIS

by

Yuankun Fu

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

Department of Computer Science

West Lafayette, Indiana

May 2021

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Fengguang Song, Co-Chair

Department of Computer and Information Science

Dr. Zhiyuan Li, Co-Chair

Department of Computer Science

Dr. Yao Liang

Department of Computer and Information Science

Dr. Xavier Michel Tricoche

Department of Computer Science

Approved by:

Dr. Kihong Park

This dissertation is dedicated to my parents who gave me the most solid support and a dream that I can pursue with my passion and endeavor.

ACKNOWLEDGMENTS

First and foremost, I would like to express my sincere appreciation for my major academic advisor, Professor Fengguang Song for his expert advice, valuable guidance, and kind support during my Ph.D. journey at Purdue University. He has taught me, both consciously and unconsciously, how solid research is done. I would like to thank my co-chair, Professor Zhiyuan Li, for his valuable guidance and suggestions. Besides, I want to thank Professor Luoding Zhu, for his guidance and support to lead me into the area of numerical analysis and computational fluid dynamics. With their help and advice, I have learned a lot and accomplished more than I could have imagined. Moreover, I want to thank my committee members, Professor Yao Liang and Professor Xavier Tricoche for serving as my committee members for their feedback and great help in my studies. This dissertation is based upon research supported by the National Science Foundation (NSF) Award No. 1522554 and Award No. 1835817.

I also would like to thank my lab mate, Feng Li, to give me generous help to design the in-situ workflow benchmark and the 2D sequential memory-aware LBM. I also want to thank another lab mate, Weijian Zheng, for his help and support in both life and academic aspects.

Finally, I would like to express special thank to my parents, Qihong Zang and Shaojun Fu, and families for their love, support and gracious understanding. It is difficult to imagine how I could have done this without them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

AI	Arithmetic Intensity
BC	Boundary condition
BGK	Bhatnagar-Gross-Krook
CARM	Cache-aware Roofline Model
CFD	Computational fluid dynamics
CHA	Cache/Home Agent
CPU	Central Processing Unit
HPC	High-performance computing
LB(M/E)	Lattice Boltzmann (method/equation)
DdQq	q velocities per cell in d-Dimension
MCDRAM	Multi-Channel DRAM
MD	Molecular dynamics
MPI	Message-passing interface
NS(E)	Navier-Stokes (equations)
PCIe	Peripheral Component Interconnect express
QPI	Intel QuickPath Interconnect

ABSTRACT

As high performance computing systems are advancing from petascale to exascale, scientific workflows to integrate simulation and visualization/analysis are a key factor to influence scientific campaigns. As one of the campaigns to study fluid behaviors, computational fluid dynamics (CFD) simulations have progressed rapidly in the past several decades, and revolutionized our lives in many fields. Lattice Boltzmann method (LBM) is an evolving CFD approach to significantly reducing the complexity of the conventional CFD methods, and can simulate complex fluid flow phenomena with cheaper computational cost. This research focuses on accelerating the workflow of LBM simulation and data analysis.

I start my research on how to effectively integrate each component of a workflow at extreme scales. Firstly, we design an in-situ workflow benchmark that integrates seven state-of-the-art in-situ workflow systems with three synthetic applications, two real-world CFD applications, and corresponding data analysis. Then detailed performance analysis using visualized tracing shows that even the fastest existing workflow system still has 42% overhead. Then, I develop a novel minimized end-to-end workflow system, Zipper, which combines the fine-grain task parallelism of full asynchrony and pipelining. Meanwhile, I design a novel concurrent data transfer optimization method, which employs a *multi-threaded work-stealing algorithm* to transfer data using both channels of network and parallel file system. It significantly reduces the data transfer time by up to 32%, especially when the simulation application is stalled. Then investigation on the speedup using *OmniPath network tools* shows that the network congestion has been alleviated by up to 80%. At last, the scalability of the Zipper system has been verified by a performance model and various large-scale workflow experiments on two HPC systems using up to 13,056 cores. Zipper is the fastest workflow system and outperforms the second-fastest by up to 2.2 times.

After minimizing the end-to-end time of the LBM workflow, I began to accelerate the memory-bound LBM algorithms. We first design novel parallel 2D memory-aware LBM algorithms. Then I extend to design 3D memory-aware LBM that combine features of single-copy distribution, single sweep, swap algorithm, prism traversal, and merging multiple temporal time steps. Strong scalability experiments on three HPC systems show that 2D and

3D memory-aware LBM algorithms outperform the existing fastest LBM by up to 4 times and 1.9 times, respectively. The speedup reasons are illustrated by theoretical algorithm analysis. Experimental roofline charts on modern CPU architectures show that memory-aware LBM algorithms can improve the arithmetic intensity (AI) of the fastest existing LBM by up to 4.6 times.

1. INTRODUCTION

1.1 Motivation and Objectives

As high performance computing (HPC) systems are advancing from petascale to exascale, scientific workflows, composed of coupled simulations along with analytics or visualization components, will facilitate scientific communities to explore the extreme-scale computational tasks in multi-disciplines (e.g., climate, nuclear energy, cosmology, astrophysics, chemical sciences and those proposed by the US Exascale Computing Program (ECP) [1]). By the year 2030, it is projected that high-fidelity computational fluid dynamic (CFD) simulations (to solve the problems, e.g., aerodynamics, climate, marine, energy transformation, gas turbines, combustion, the spread of COVID-19, 3D printing, etc.) will be at the grid resolution of 100's of millions of points with a large ensemble of parameter variations [2]. The fine resolution and transient nature of these simulations will generate the entire volumetric solution dataset of 100's of terabytes or even petabytes [3], [4]. However, the computation-I/O gap on leadership-class systems becomes even larger. For example, the peak CPU performance is at least five orders of magnitude faster than the I/O bandwidth on the world's fastest supercomputer Fugaku in the top500 list of 2020 [5]. [1] Thus, moving the data between the simulation and analysis components of scientific workflows becomes a serious bottleneck.

For decades, the dominant paradigm is the *"post hoc"* workflow, i.e., writing the whole dataset to persistent file system, then later reading them for post analysis/visualization. Although the *post hoc* paradigm is easier for human interaction and exploratory investigation, there are several critical constraints. Firstly, the primary issue is performance, i.e., the low I/O throughput and massive data movement across networks can become a bottleneck in the overall scientific workflow [7], [8]. The second issue is capacity, i.e., storing the total dataset can easily exceed the available storage space. Thirdly, it is unnecessary to store huge amounts of whole datasets, which slows down some scientific discovery campaigns. For example, only features of interest (e.g., shocks or vortices, which reside mainly on the wet surfaces of the objects being studied [7]), or rare events detection – used in deep learning,

[1]↑Fugaku's peak LINPACK benchmark performance is 0.442 exaFlops, total memory bandwidth is 163 PB/s, Tofu-D 6D Torus network bandwidth is 6.49 PB/s. Its single node contains local L1 NVMe storage and PCIe Gen3 x16 I/O with 100 Gbps I/O network endpoint into Lustre [6].

graph analysis, or experimental data analysis – need to be saved to storage. The fourth issue is economics, i.e., moving data around disks and I/O fabric is expensive in terms of energy and money.

To address these issues, the *in-situ* workflow has been studied over the past three decades. The word in-situ originates from Latin, and means "on-site", or "in-place". The in-situ workflow literally means analyzing or visualizing data as data is generated [9]. This term evolves as the in-situ research goes on, and recently a group of over fifty experts convened to standardize its definition [9], which is followed in this dissertation. We define the in-situ workflow as a workflow whose tasks are coupled by exchanging data over the memory, storage hierarchy, or network in the same HPC system during the same scheduled execution of a job [10], [11]. [2] Thus, it includes both on-node proximity (i.e., tasks performed only in the same node of the compute resources [12], alias names are synchronization or co-processing [13], [14]) , and off-node proximity (i.e., offloading computations to a set of secondary resources using asynchronous data transfers [15], [16], alias names are in-transit or data staging). There are several advantages to use in-situ workflows. The first point is the I/O cost savings, where data is analyzed/visualized while being generated, without first storing in a file system. The second point is higher fidelity and accuracy, which leads to better science. Since in-situ workflows could perform fine temporal sampling of transient analysis without throttled by I/O , the *post hoc* workflow has to use coarse temporal sampling to avoid the excessive I/O cost. The third point is the economical availability to use all computing resources on the same site for both simulation and visualization/analysis routines without moving to another site. Above all, it is more appealing to use in-situ workflows to combine the exascale simulation with big data analysis to create a virtuous cycle to amplify their collective effects [17]–[20].

However, it is exceedingly challenging to build high performance in-situ workflows at extreme scales, which motivates **the first objective** of my Ph.D. research — **accelerating massively parallel in-situ workflows of simulation with big data analysis**. Several basic questions are still hard to answer: Did any unusual phenomena happen or not during

[2]↑Conversely, a distributed workflow is one whose tasks are more loosely coupled through files, and executed on geographically distributed clusters, clouds, grids, etc. [10].

the workflow? When and where did they occur? Will they affect the original performance of the simulation or analysis application if not using the workflow method? How to circumvent the bottlenecks to achieve optimal workflow performance?

On the other hand, computational fluid dynamics (CFD) simulations emerged five decades ago and have been paced by advances in the HPC systems [21]. They revolutionized the design process in various scientific, engineering, industrial, medical fields, etc. For example, CFD is the main tool for preliminary aerodynamic design and is supported with wind tunnel testing. Other applications include the design of Formula 1 racing cars, red blood cell flow in the vessels of the heart, etc. Currently, we can use Reynolds averaged Navier-Stokes (RANS) solvers to calculate the steady viscous flow from low speed to transonic and supersonic flow, but they are not able to reliably predict for turbulent-separated flows [22].

Lattice Boltzmann method (LBM) is a young and evolving CFD approach to solving these problems since the late 1980s. It originates from a mesoscale description of the fluid, can integrate physical terms of molecule interaction, and apply to complex geometries and flows. Solving isothermal and weakly compressible flows can be relatively difficult for Navier-Stokes equations [23] [3], but is simple for LBM. Later, many collision models for LBM are proposed to improve its stability to the second order of numerical accuracy when simulating high Reynolds number flows [25]. Now exascale LBM has shed light on its capability to solve future CFD problems. For the large eddy simulation (LES, e.g., a full-powered aircraft configuration across full flight envelopes [4]), LES-based LBM models have shown their strengths to simulate thermal and compressible flows in transonic and supersonic regimes [22], [25]–[29]. At moderate Reynolds regimes, exascale hemodynamics with LBM can advance precision medicine by simulating a full heartbeat at the red-blood cell resolution in about half an hour [26]. At particle regimes, LBM potentially promotes medical therapies against neurological diseases by millisecond simulations of protein dynamics within the cell [26].

However, it is also challenging to achieve high performance for exascale LBM simulations, which motivates **the second objective** of my Ph.D. research — **accelerating the LBM**

[3]↑On May 24, 2000, Clay Mathematics Institute proposed seven millennium problems (e.g., Riemann Hypothesis, P vs NP Problem, etc). The Navier–Stokes existence and smoothness problem is one of them [24].
[4]↑This is the first of the four grand challenges proposed by the NASA CFD Vision 2030 Study [2]

algorithm. This is because LBM is a heavily data-intensive and notoriously memory-bound algorithm [30]. LBM simulation can be generally viewed as an iterative *collision-streaming* cycle. The *collision* step is purely local computation at each lattice node and can be adapted to different collision models according to stability and applied circumstances (incompressible/compressible). The *streaming* step requires intensive data exchange among neighboring lattice nodes. Thus, how to efficiently improve the memory access pattern and the arithmetic intensity (AI) of LBM will be the key to accelerate LBM algorithms.

1.2 Research Challenges

Scientific workflow research can be generally divided into two categories: in-situ workflow and distributed workflow. For the existing distributed workflow solutions, such as Swift/T [31], Tigres [32], Kepler[33], Pegasus [34], Vistrails [35], Galaxy [36], Taverna [37], etc. , they target high productivity and have been widely used in different scientific domains. They provide orchestrating, executing, and monitoring *coarse-grain steps* in a workflow. Each step runs an application program or cloud/web service [10], [38]. However, those participant steps are often *loosely coupled* such that the resultant workflows have higher latencies (i.e., milliseconds or much more) than the MPI-based HPC applications (i.e., microseconds).

To achieve higher performance within an HPC system, in-situ systems have been developed to reduce the I/O bottleneck by many researchers (e.g., MPI-IO [39], ADIOS [40], DataSpaces [41], FlexPath [42], Decaf [11], etc). As for the first objective to accelerate the performance of in-situ workflows, there are several challenges to solve:

1. "*Benchmarks and community data sets*" is one of the six research challenges for future scientific workflows [10]. To the best of our knowledge, there exists no such in-situ workflow benchmark available to compare the performance of different in-situ systems in scientific workflow communities. Due to different design purposes, hardware configurations, software stacks, and HPC system environment compatibility, it is nontrivial to design such benchmarks to integrate simulation and analysis with those existing in-situ systems to make workflow function properly and then evaluate fairly their best achievable performance with appropriate configurations.

2. What could be the minimum end-to-end time-to-solution for an in-situ workflow of simulation with analysis applications? How to achieve it?

3. Simulation and data analysis applications work as an interactive producer-consumer system, thus how can we reduce the simulation stall time if the analysis is slow?

4. How can we reduce the I/O or data transfer time between simulation and analysis applications?

As for the second objective to accelerate the LBM algorithm, due to the iterative nature of LBM to reach convergence, publications have mainly focused on improving the performance of the collision-streaming cycle within one iterative time step. For instance, a few LBM algorithms (e.g., as swap [43], shift [44], AA-pattern [45], esoteric twist [46], etc.) retain a single copy of the particle distribution, and optimize the memory access pattern in the LBM streaming kernel, but each of the algorithms needs to follow a set of constraints (e.g., swap requires predefined order of discrete cell velocities, etc.) Some work replaces distribution representation with moment representation to further reduce the storage cost [30]. Some work explores spatial locality techniques (e.g., loop fusion, loop tiling, loop skewing, etc.), but produces limited improvement [47]. Some hides the inter-process communication cost on multicore accelerators [48], and achieve large-scale parallelization on HPC systems [49] and GPUs [50].

Although the LBM community has achieved fruitful results, we find that the existing techniques show that the state-of-the-art performance of LBM is still under the memory-bound ceilings in the Roofline model [30]. To further improve the performance of LBM, a novel design of LBM is necessary to improve its Arithmetic Intensity (AI) and eventually alleviate its memory-bound limitation.

Our intuition is to increase data reuse across multiple time steps of collision-streaming cycles. There exists research using wavefront parallelism to merge multiple time steps, but they enforce frequent synchronization among threads in every time step [47], [51]. Instead, we aim to achieve higher performance by minimizing synchronization costs. We decompose the simulation domain to each thread with as much data as possible, but how to handle the intersection or overlapping area involves uneasy issues, especially when merging multiple time steps. Thus, we start by exploring the effectiveness of our intuition on 2D cases with the sequential and parallel versions, and then extend them to the more complicated 3D cases. However, there exist nontrivial challenges related to both thread safety and performance efficiency.

1. For the 2D sequential LBM, how to arrange the memory access pattern to correctly merge multiple time steps of the collision and streaming in a tile of the simulation domain, meanwhile to keep the data integrity and dependency among multiple time steps, to handle the boundary conditions correctly, and without involving extra data storage?

2. For the parallel 2D LBM, how to handle the lattice points on the intersection area between different threads? How to minimize synchronization cost? And how to justify the parallel performance improvement?

3. For the 3D sequential LBM, as the geometry changes from $O(n^2)$ to $O(n^3)$, data storage has increased by an order of magnitude, and data dependency of lattice model becomes more complicated (from D2Q9 to D3Q19 or D3Q27). We need to consider the huge increase of data storage and use the existing data storage optimization for LBM (e.g., swap algorithm, etc). Although these methods have the advantage to reduce the total storage cost by half, they also require to follow specific traversal orders (e.g., instead of streaming standard total populations of each cell, swap algorithm streams only half of the populations to the nearest neighbors). Based on this condition, how can we still combine the idea of merging multiple time steps with these existing methods and utilizing the spatial and temporal locality?

4. For the parallel 3D LBM, how to handle the intersection layers among threads correctly and efficiently? How to reduce synchronization cost in parallel?

1.3 Research Overview

1.3.1 Accelerating the massively parallel in-situ workflow

To achieve the first objective of my Ph.D. research, I start by designing an in-situ workflow benchmark using the latest high-level I/O libraries and in-situ systems (e.g., MPI-IO [39], ADIOS [52], DataSpaces [41], DIMES [53], Flexpath [42] and Decaf [11]) to "glue" simulation and analysis applications. The benchmark has developed seven different in-situ workflow implementations to combine a LBM simulation with a turbulence flow analysis application. Each workflow implementation employs a different I/O transport method. From the experimental results in Chap. 3, we can conclude that these workflows' end-to-end time is significantly larger than the essential simulation or analysis time. More detailed performance analysis identifies a set of performance inefficiencies, such as synchronization with data staging servers, coarse-grain critical sections, interlock and barriers between applications, network bandwidth contention, and application stalls.

Instead of only concentrating on the workflow's data transfer cost, my second step is to design a novel in-situ system, *Zipper*, to focus on optimizing the end-to-end workflow's time-to-solution. Its design inspiration originates from an in-situ workflow performance model, which estimates an in-situ workflow's time-to-solution, and guide improvement on the most time-consuming component to achieve the minimal end-to-end time of a workflow. Specifically, Zipper uses the parallelism of fine-grain tasks, pipelining and asynchrony to tightly integrate simulation and analysis applications. To overcome the above inefficiencies incurred by the existing in-situ systems, Zipper is driven by data availability, and has no data staging server cost and no artifactual data dependency (e.g., barriers) between tasks. Moreover, Zipper can transport simulation output by two concurrent channels: low-latency HPC network and file-based deep memory storage (e.g., NVME, local storage, parallel file system, etc.). At last, it supports two common scientific discovery scenarios: 1) Users can use the *Preserve* mode to store the intermediate simulation results, which can be used to verify the correctness of the simulation, or for post exploratory investigation; 2) Users can use the *NoPreserve* mode to discard the intermediate simulation output and the analysed data to speedup the discovery workflow circle. Chap. 4 introduces the design details of Zipper.

Then my third step is to conduct workflow experiments to assess the performance of Zipper and compare it with existing in-situ systems on two HPC systems. We started with two groups of synthetic experiments: the first group validates whether Zipper conforms to the analytical performance model; the second group confirms that the concurrent dual-channel data transfer optimization method can reduce data transfer time and the simulation application stall time, followed with the investigation on the speedup reason using network congestion analysis. Next, we evaluate the scalability performance of Zipper using two real-world applications: the LBM simulation is coupled with an online statistical turbulence analysis, and the LAMMPS simulation is coupled with the Mean-Squared Displacement (MSD) data analysis. Based on the experimental results, Zipper workflow can outperform the existing fastest state-of-the-art in-situ systems by up to 2.2 times on 13,056 cores. At last, the performance benefits have been studied and analysed by collecting and comparing different workflow implementations' traces.

1.3.2 Accelerating the lattice Boltzmann method

To achieve the second objective of my Ph.D. research, I start by accelerating LBM in 2D cases. The Roofline performance model is an emerging visual performance analysis tool to offer insight for applications on multicore architectures. I firstly use it to evaluate the sequential performance of the Original LBM algorithms and two other LBM algorithms, i.e., Fuse LBM (only using loop fusion) and Fuse tile LBM (combining loop fusion and loop tiling). We observe that the streaming step in the three LBM algorithms suffers low arithmetic intensity, spends the longest proportion of total running time, and is bounded by the DRAM bandwidth ceiling. This discovery motivates us to target improving the memory access pattern by merging multiple time steps of collision-streaming cycles to explore the temporal locality.

We start by designing the 2D sequential memory-aware LBM algorithm which merges two time steps of collision-streaming cycles, and then study handling various boundary conditions (BCs) under this new computation pattern, since BCs are fundamental to maintain the accuracy of LBM stable. Then to alleviate the synchronization cost incurred by the wavefront algorithms used by existing research, I design the 2D parallel memory-aware LBM, which targets that each thread computes more time steps within each local data domain and reduces the synchronization costs. Then I discuss how to handle the intersection area among threads to keep thread safety. Next, I design the sequential and parallel memory-aware LBM in 2D cases to merge three or more time steps of collision-streaming cycles. To analyse the potential performance gain, we compare the amount of data usage in these new LBM algorithms. At last, three groups of experiments are conducted on three different manycore architectures. The first group evaluates the sequential performance of seven LBM algorithms (i.e., three baseline LBM algorithms and four new ones), followed by Roofline analysis. We observe that the two-step tile LBM is the best and outperforms the state-of-the-art Fuse LBM by up to 30% on a Haswell CPU and 20% on a Skylake CPU. The second group compares the strong scalability performance of seven parallel LBM algorithms. We find that the k-step parallel memory-aware LBM is the best and can outperform the Fuse LBM by up to 4.4 times on the Haswell node with 28 cores, 5 times on the Skylake node with 48 cores, and 2.6 times on the Knight Landing node with 68 cores. Roofline analysis is used again to investigate the performance gain. The third group uses Paraview [54] and Catalyst [14] to visualize the data generated by our new algorithms, which validate the simulation results.

My next step is to extend to the 3D cases with the idea of merging multiple time steps to accelerate LBM. Both sequential and parallel memory-aware 3D LBM algorithms are designed to merge two time steps of LBM collision-streaming cycles. Besides, together with swap and prism traversal, I reduce the storage cost from two copies of the particle distribution data per fluid lattice to one copy. Besides, I study how to handle the intersection planes between threads in parallel within 3D. At last, I conduct three groups of experiments on three different manycore architectures. The first group evaluates the sequential performance of three 3D LBM algorithms under the Palabos LBM framework. The sequential experiment shows that the two-step prism LBM outperforms the state-of-the-art Fuse prism LBM by up to 19% on a Haswell CPU and 15% on a Skylake CPU. The second group evaluates the strong scalability performance of the 3D LBM by two subgroup experiments: equivalent input and identical input. Based on the scalability experiments using the equivalent input, the two-step prism memory-aware LBM outperforms the Fuse prism LBM by up to 89% on a Haswell node with 28 cores, 85% on a Skylake node with 48 cores, and 39% on a KNL node with 68 cores.

1.4 Contributions

This dissertation is mostly drawn from prior works which have been peer-reviewed and published in high-quality HPC and CFD conferences. Our in-situ workflow research appeared at [55]–[57], the 2D memory aware LBM research appeared at [58] and the 3D memory aware LBM research is being reviewed. To the best of our knowledge, this dissertation makes the following contributions:

1. Design **an in-situ workflow benchmark** in the scientific workflow community. It integrates seven state-of-the-art in-situ systems with two real-world CFD simulations and corresponding data analyses. Then detailed performance analysis via visual tracing shows that even the fastest existing in-situ system still has 42% overhead, due to synchronization with staging servers, coarse-grain critical sections, interlock and barriers between applications, network bandwidth contention, and application stalls.

2. Develop a novel **minimized end-to-end** in-situ system, **Zipper**, and propose an in-situ workflow performance model. Zipper utilizes both hybrid proximity, combines the fine-grain task parallelism of fully *asynchrony and pipeline*, and supports *Preserve* and *No-Preserve* mode. Scalability workflow experiments on two HPC systems using up to 13,056 cores show that Zipper is the fastest, and outperforms the second fastest by up to **2.2 times**.

3. Design a novel **concurrent data transfer optimization method**, which is embedded in the Zipper in-situ workflow. It employs a *multi-threaded work-stealing algorithm* to transfer data using both channels of network and deep memory hierarchy (burst buffer, NVMe, local storage, parallel file system, etc.). When the method combines network and parallel file system, it significantly reduces the data transfer time by up to **32%**, therefore reducing the simulation stall frequency. Then investigation on the

speedup using *OmniPath network tools* shows that the network congestion has been alleviated by up to **80%**.

4. Design novel **sequential/parallel 2D & 3D memory-aware LBM** algorithms to accelerate LBM algorithms efficiently. It combines both spatial and temporal locality (merging multiple time steps of computation). Specifically, we reduce half of the storage cost in 3D cases. Strong scalability experiments on three manycore architectures show that 2D & 3D memory-aware LBM outperforms the existing fastest LBM by up to **5 times** and **89%**, respectively.

5. Explain the speedup reason using both theoretical algorithm analysis and Roofline performance model. The 2D memory-aware LBM can improve the arithmetic intensity (AI) of the Fuse tile LBM by up to **4.6 times**.

1.5 Dissertation Organization

The rest of this dissertation is organized as follows.

Chap. 2 introduces the background of in-situ and CFD research. It gives a brief introduction of in-situ history and concepts, and summarizes different state-of-the-art in-situ systems. Then various numerical methods of fluid dynamics are compared and introduced.

Chap. 3 designs the in-situ workflow benchmark, compares different in-situ workflows' performance, and then investigates their inefficiencies and bottlenecks.

Chap. 4 presents the Zipper and proposes an in-situ workflow performance model. The implementation details of Zipper and the concurrent dual-channel data transfer optimization method are described. Then we run experiments and evaluate the performance using the existing in-situ systems and Zipper.

Chap. 5 designs the 2D sequential and parallel k-step memory-aware LBM algorithms, and compares their results with the existing LBM algorithms.

Chap. 6 moves forward to extend the idea to design the 3D sequential and parallel k-step memory-aware LBM algorithms, and compares their performance with the existing 3D LBM algorithms.

Chap. 7 summarizes the dissertation and presents the future work.

2. BACKGROUND

In this chapter, Sect. 2.1 gives an overview of in-situ research, i.e., its concept, history, categories, followed by a summary of existing in-situ workflow systems. Sect. 2.2 gives an overview of the basic theory of fluid dynamics and existing various numerical methods. Then since fluid can be also viewed by either a continuum or particle description, we separately introduce numerical methods under each scope. Both sections are backgrounds that motivate my Ph.D. research.

2.1 Overview of In-situ Processing

The idea and history of performance analysis, producing images without first outputting data to storage, dates back to the 1960s. The NCAR Graphics Library [59] which contains a group of subroutines for producing images/plots is one of the "in situ methods and infrastructure" in the early time and is still used today.

However, due to the exascale computing-I/O gap, in-situ research has advanced and many in-situ systems are developed separately from the *storage* and *visualization* communities. Understanding their difference and features will be the key to help scientists choose the most appropriate to speed up their scientific discovery.

2.1.1 Categories of In Situ Systems

The in-situ terminology team [9] proposed six axes to categorize them, but we can distill them into four axes from end users' perspective as follows.

1. **Proximity**: how close the simulation are to the analysis/visualization. We can further divide the proximity into two sub-categories: ***On Node*** vs ***Off Node***, i.e. whether the analysis/visualization share the same resources on the node of simulation or not. In some in-situ systems, visualization/analysis *directly* accesses the memory of simulation on the *same* resources, thereby the advance of the simulation and visualization/analysis is alternated on these resources. Aliases of on-node proximity are *time division*, *coprocessing*, or *direct access*. On the other hand, visualization/analysis on distinct

resources uses *indirect* access to the simulation by the network, burst buffer, local file system, or specific connections (e.g., PCI , NVLink, etc.). Aliases of Off Node proximity are *space division* or *indirect access*. Some systems allow in-situ routines to be broken down into separate pieces and deployed onto both compute nodes and staging nodes, i.e., **Hybrid**.

2. **Access**: **Deep Copy** vs **Shallow Copy**. Deep Copy makes a copy of the simulation data, and can prevent the simulation stall if in-situ routines are slow. Shallow Copy can save memory space and adapt data to simulation routine by adding extra data representation. But it may stall the simulation, if in-situ routines haven't finished using the simulation data.

3. **Integration Type**: how the simulation code is integrated to the analysis/visualization code. Under this axis, we can further distinguish by whether the simulation code knows the API of the integration. The majority of in situ systems use ***Application-Aware*** mode (e.g., dedicated API or multipurpose API). On the other hand, some new research uses ***Application-Unaware*** mode to integrate simulation and analysis e.g., *interposition* (i.e., using dynamically-loaded library to replace the expected routines by the custom in situ routines), *inspection* (i.e., inspecting memory to deduce data patterns and attach in-situ functions).

4. **Operation Controls**: the (***interactive*** vs ***automatic***) schemes for choosing the executed operations during run-time. Some in-situ systems let users interactively change operations (human-in-the-loop), while others fix the operations at the beginning and refuse users to change during execution.

The first axis can be used to distinguish different in-situ systems in most cases. Both On Node and Off Node have potential benefits and pitfalls.

On Node (time division) has less costs of synchronization and data transfer. Since it can directly access the memory space of simulation, Deep copy and Shallow copy can be used. But since it shares the same resources of simulation, the performance of simulation can be influenced. Also since it alternates the execution of resources between simulation and in-situ routines, we cannot concurrently execute both of them to achieve seamless integration. Moreover, the data decomposition used in simulation can be unfavorable for analysis/visualization (e.g., handling ghost layer data, which is more easily addressed by Off Node mode).

Off Node (space division) can offload the in-situ routines to specific resources to help both the efficient execution of the simulation and in-situ processing. However, it may incur over-subscription if configured poorly. A solution can be the dynamic management of in-situ system resources at run time. Besides, the cost synchronization and data transfer may involve more infrastructure and potential overhead. At last, in-situ routines under Off Node mode can also possibly block the simulation if they run slowly due to insufficient resources.

2.1.2 State-of-the-art In-Situ Systems

In this section, we briefly introduce several state-of-the-art in-situ systems as follows.

1. **Paraview/Catalyst** [14] and **VisIt/Libsim** [60] were both developed firstly as post hoc tools by the visualization communities around 2000. Now, they also support *On Node* in-situ processing. Paraview/Catalyst defines an interface between simulation and visualization applications, which requires users to implement three subroutines: initialize, coprocess, and finalize. The co-process subroutine is responsible for converting raw simulation data to ready-to-visualize data, and computing visualization functions in each time step. Similar functions are also provided by VisIt/LibSim. Both of them support shallow-copy, and allow human-in-the-loop control. Besides, the shared HDF5 virtual file layer can be used to support Off node mode, when running the simulation and the ParaView servers in separate jobs to read and write data [61].

2. **Damaris** [12] is originally designed as a middleware to support I/O operations for Catalyst and VisIt. Damaris uses an XML-based system to describe the data movement and control of in-situ routines. It splits the global MPI communicator of a simulation to dedicated cores (On Node) or nodes (Off Node) in order to run the analysis concurrently with the application.

3. **SCIRun** [62] is another *On Node* in-situ framework. It uses a dataflow model to support interaction with a simulation application at run time. It also uses Shallow copy by incorporating templates to adapt to the simulation code at compile time. This allowed SCIRun to minimize its memory footprints.

4. **GLEAN** [13] uses an *interposition* interface aiming for minimal modifications to integrate in situ code with the simulation code. It is implemented in C++ leveraging MPI, pthreads, and higher-level I/O libraries such as MPI-IO, Parallel-netCDF [63] or HDF5 [61]. The simulation can call GLEAN's I/O libraries to move data to storage or staging servers. In-situ analysis routines can be either On Node or Off Node. Then GLEAN uses higher level I/O libraries to output data from staging servers to storage asynchronously.

5. **MPI-IO** is a parallel low-level I/O library that allows multiple processes of an MPI application to write or read parts of a shared common file [39], [64]. It can map I/O reads and writes to message-passing sends and receives to improve the I/O performance. But in-situ processing with MPI-IO is a bespoken approach that requires users to manually write specific code to integrate with producers and inform consumers when new data is available on the storage. MPI-IO workflow can be either On Node or Off Node by users' choice.

6. The Adaptable I/O System (**ADIOS**) [40], [52] was developed by the storage communities as a high-level data processing library for storage, staging, compression, and reduction of data. It uses an XML file or a multi-purpose API to describe data by a prior generalization. This API can also integrate with a range of in-situ systems, e.g., DataSpaces [41], DIMES[53], and Flexpath [42] to stage data, or Ascent [65], VisIt [60], and ParaView [14] for analysis/ visualization. But there is a performance cost due to the over-generalization of the interface. Due to its flexibility, users could use ADIOS to be On Node or Off Node as their own choice.

7. **DataSpaces** offers a distributed shared-memory space across a number of central data staging servers [41], [66]. Each participant application has its own failure domain by

launching separate *mpirun* or *aprun* commands, and then connects to the data staging server via a publisher/subscriber interface (*put* and *get*). DataSpaces indexes data based on a space-filling curve. Data are then distributed among data servers based on their index. The distributed index is used both for pushing data into DataSpaces and for retrieving data efficiently from it using one-sided RDMA communications. Reader-writer locks are provided to coordinate accesses to shared data among different applications.

8. DIstributed MEmory Space (**DIMES**) [53] is an successor of DataSpaces. DIMES uses DART [67], a set of RDMA communication primitives, to asynchronously transfer data. Instead of exchanging data from the central data servers as Dataspaces does, DIMES uses a peer-to-peer (P2P) model, and stores data to RDMA memory buffers located in the producers' nodes directly (On Node), and then consumers read data directly from the producers' memory (Off Node). Before establishing P2P connections, producers need to communicate with the central *metadata* staging servers which manages the location (index) of data. Locking schemes are required to keep the data consistency.

9. **Stacker** [68] is another extension of DataSpaces. It utilizes emerging storage architectures (e.g., deep memory hierarchies and burst buffers) as data staging solutions. It also can handle application-aware data movement by application hints and machine learning methods.

10. **Flexpath** [42] (former named FlexIO [69]) is a data transport method based on EVPath [70] in ADIOS. It uses a publisher/subscriber paradigm and leverages both RDMA and IP-based networking protocols (TCP, reliable UDP, multicast, etc.). With Flexpath, different software components can be connected by event channels and source-to-sink event communications at runtime to perform Hybrid in-situ process sing. Each publisher or subscriber is executed as an independent application by running *mpirun* or *aprun*. Hence, Flexpath has multiple failure domains. To transfer data, a publisher uses an output epoch (i.e., open, write, close) to save data to its

buffer. Later on, a subscriber sends to each of the event publishers a fetch message to request its desired data.

11. **Decaf** [11] is a dataflow system for parallel communication of participant applications in workflows. It can be regarded as a "coupling service", which allows users to describe nodes and links as serial entities while Decaf takes care of their parallelism. It provides a simple put/get API that utilizes MPI, and can implement a workflow system by using a Python API. Different from the above DataSpaces, DIMES and Flexpath, Decaf creates a single MPI_Comm_World for all the participant applications. Data coupling between applications is defined during the compile time. Also, it requires existing MPI-based programs to replace their MPI_COMM_WORLD by the communicator provided by Decaf. Therefore, there is a single failure domain in Decaf workflows.

Tab.2.1 summarizes the in-situ systems using the aforementioned four axes. There are other in-situ systems summarized in [9], [71], and readers who have interest can explore more. Although there exist many in-situ systems, one may ask: which one has the best performance? Do they integrate each component of a workflow efficiently and seamlessly? Chap.3 will answer the questions by building a benchmark to compare the performance when using some of the above in-situ systems.

2.2 Overview of Computational Fluid Dynamics

This section gives an overview of the basic theory of fluid dynamics and existing various numerical methods. A substance exists in three primary phases: solid, liquid, and gas. A substance in the liquid or gas phase (including plasma) is referred to as a *fluid*. Fluid has the following three features: 1) continually deforms under stresses, 2) resists deformations only lightly because of viscosity, and 3) can adopt the shape of any container into which it flows.

Mechanics is categorized into *statics* and *dynamics*, which handles stationary and moving objects under the influence of forces, respectively. Thus, *fluid dynamics* (**FD**) is the study of fluids in motion under forces. More specifically, *hydrodynamics* is the study of the motion of the incompressible or isochoric fluid. The material density of the fluid is constant within a

Table 2.1. Types of existing in-situ systems using four axes.

Name	**Proximity**	**Access**	**Integration Type**	**Operation Control**
Paraview/Catalyst	On Node	SC	D-API	Interactive/Batch
VisIt/Libsim	On Node	SC	D-API	Interactive/Batch
SCIRun	On Node	SC/DC	D-API	Interactive/Batch
GLEAN	On Node	SC/DC	Interposition	Batch
Damaris	On/Off Node	SC/DC	D-API	Interactive/Batch
ADIOS	On/Off Node	SC/DC	MP-API	Interactive/Batch
Dataspaces	On/Off Node	DC	MP-API	Batch
DIMES	Hybrid	DC	MP-API	Batch
Stacker	On Node	DC	MP-API	Batch
Flexpath	Hybrid	DC	D-API	Batch
Decaf	Hybrid	DC	D-API	Batch

SC = Shallow Copy; DC = Deep Copy
D-API = Dedicated API; MP-API = Multi-Purpose API

fluid parcel, such as liquids, especially water, or gases at low speeds. Through fluid dynamics, we understand how objects interact with the media they are immersed in. Fluid dynamics is everywhere in science (e.g., physics, biology and medicine, chemistry, geology, etc.) and engineering (e.g., mechanical, civil, household, etc.).

2.2.1 Computational Fluid Dynamics

Generally, there are three analysis methods to study fluid dynamics.

The first is the *analytical method*, in which people basically use paper and pen to figure out a group of generalized equations. However, it only works for simple and limited geometries, and is not feasible for solving the non-linearity of FD equations and boundary conditions of complex shapes.

The second is the *experimental method*, in which people build facilities in a lab to measure and monitor the experimental results. This method is accurate but limited to experimental scale and expensive to build specialized facilities. Besides, its main objective is to test what we knew, but not to explore new possibilities.

The third is the *computational method*, namely *Computational Fluid Dynamics* (**CFD**). It can be just as accurate as the experimental method and give more information cheaper.

According to viewing fluid as continuum model or particle model, many numerical methods are developed to solve FD equations, which are briefly introduced in Sect. 2.2.3. However, they can be difficult to implement and to parallelize using exascale computing, and it is generally agreed that there is no one method better than the others. More details about each method and background are presented in [72].

2.2.2 Continuum Governing Equations

From the perspective of macroscopic phenomena of fluid, people can consider fluid as a continuum model. We need to follow three conservation equations of mass, momentum, and energy.

The first **Continuity Equation** follows the *conservation of mass.* The mass of a fluid element with density ρ and volume V_0 is $\int_{V_0} \rho \mathrm{d}V$. The fluid flow with velocity $\boldsymbol{u}$ into or out of the volume V_0 results in the change of the mass of the fluid element per unit time, Due to the conservation of mass, together with the divergence theorem, we have the following equation:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}) = 0 \tag{2.1}$$

The second equation is the **Navier-Stocks Equation** (NSE, 1822), which follows the *conservation of momentum.* The flow of momentum into or out of the fluid volume V_0, changes of pressure p, and external forces $\boldsymbol{F}$ can result in the change of net momentum. For the incompressible flow (constant ρ), NSE can be written in its most common form:

$$\rho \left(\frac{\partial u}{\partial t} + u \cdot \nabla u \right) = -\nabla p + \eta \Delta u + F, \tag{2.2}$$

Here, p is pressure, η is viscosity, $\Delta = \nabla \cdot \nabla = \frac{\partial^2}{\partial x_\beta \partial x_\beta}$ is Laplace operator, and $\boldsymbol{F}$ is the external body force. The NSE has the following features: non-linearity, simultaneous (i.e., velocity has 3 components in 3D case), extremely complicated math, and no general analytical solution existed.

Moreover, since there are five unknown variables (density ρ, velocity (u_x, u_y, u_z), and pressure p), the above system of two equations has not been closed yet.

Thus, we need to introduce the third equation, **State Equation**, which follows the *conservation of energy* and the state principle of equilibrium thermodynamics [73]. It adds the thermodynamic state variables (e.g., pressure p, density ρ, temperature T, internal energy e, and entropy s) to the above system of equations.

$$\frac{p}{p_0} = \left(\frac{\rho}{\rho_0}\right)^{\gamma} \mathrm{e}^{\frac{(s-s_0)}{c_V}} \tag{2.3}$$

Here, c_V is the heat capacities at constant volume, c_p is the constant pressure, and γ is their ratio.

$$c_V = \left(\frac{\partial \mathrm{e}}{\partial T}\right)_V, \quad c_p = \left(\frac{\partial(\mathrm{e} + p/\rho)}{\partial T}\right)_p, \quad \gamma = \frac{c_p}{c_V} \tag{2.4}$$

With suitable approximations, such as p_0, ρ_0, and s_0 at some constant reference state, we can simplify equation(2.3) as follows:

$$p = p_0 \left(\frac{\rho}{\rho_0}\right)^{\gamma} \tag{2.5}$$

Dimensionless number

Dimensionless numbers in fluid mechanics are used to describe ratios of the relative magnitude of fluid and physical system characteristics, such as density, viscosity, speed of sound, flow speed, etc. They play an important role in analyzing the behavior of fluids.

The first is the **Reynolds number**. l is the macroscopic length scale, e.g., flow move from one site to another. u is the flow velocity, and ν is viscosity. Thus, the shortest time scale is either $t_{\text{conv}} \sim \ell/u$ or $t_{\text{diff}} \sim \ell^2/\nu$ in the advective (inertial) regime or diffusive (viscous) regime, respectively. The ratio between t_{conv} and t_{diff} is *Reynolds number*:

$$Re = \frac{t_{diff}}{t_{conv}} = \frac{u * l}{v} \tag{2.6}$$

For the "thick" fluid (e.g., honey), it has low Re and easily forms laminar (steady) flow, which typically is seen in areas such as microfluidics, biophysics, and others. Conversely, for the “thin” fluid (e.g., water), it has higher Re and easily forms the turbulent flow, which is widely used in aerodynamics, nuclear weapons, and other applications.

The second is the **Mach number**, which is the ratio between the acoustic time scale t_{sound} and advective time scale t_{conv}:

$$Ma = \frac{t_{sound}}{t_{conv}} = \frac{u}{c_s} \tag{2.7}$$

The speed of compression waves transfer in the fluid is determined by $t_{\text{sound}} \sim l/c_{\text{s}}$, where c_s is the speed of sound in the fluid. When the acoustic time scale t_{sound} of the fluid is faster than the advective time scale t_{conv}, we can assume that the fluid has a similar behavior as an incompressible fluid with $Ma \leq 0.1$.

The third is the **Knudsen number**, which defines the ratio between the mean free path l_{mfp} [1] and the macroscopic length scale l within which flow occurs.

$$Kn = \frac{l_{mfp}}{l} = \alpha \frac{Ma}{Re} \tag{2.8}$$

If $0.1 < Kn < 1$, hydrodynamic flows fall into this category and can be solved by Navier-Stokes equations. For $Kn \geq 1$, microscale and nanoscale flows are valid, and we use particle-based methods (kinetic theory description), which are introduced in Sect.2.2.3.

2.2.3 Existing Numerical Methods for Fluid Dynamics

The analytical solutions to the continuum governing equations in Sect. 2.2.2 can only apply to simple geometries, but they are difficult or impossible to be found due to the non-linearity and the boundary conditions (BCs) for complex shapes. We can numerically convert them into a system of partial differential equations (PDE), then computationally and iterative solve them until their convergence is ensured.

Macroscale methods are general methods to solve PDE by directly discretizing the macroscale fluid equations with minor adaptions, e.g., Finite Difference Method (FDM), Finite Volume Method (FVM, 1980), and Finite Element Method (FEM, 1956). Others are

[1]↑The average distance traveled by a molecule to collide with another molecule.

particle-based methods to describe fluid at microscale or mesoscale, in which a particle can be viewed as an atom, a molecule, a cluster of molecules, or a portion of the macroscale fluid, etc. Microscale methods include Molecular dynamics (MD). Mesosale methods include Lattice Gas Models (LGA), Lattice Boltzmann Method (LBM, 1988), Dissipative Particle Dynamics (DPD, 1992), Multi-particle Collision Dynamics (MPC, 1999), Direct Simulation Monte Carlo (DSMC, the 1960s), Smoothed-Particle Hydrodynamics (SPH, 1970).

Conventional CFD Methods

There are three conventional CFD methods (i.e., Finite Difference Method, Finite Volume Method, and Finite Element Method) to directly solve the coupled system of fluid dynamics equations introduced in Sect.2.2.2. But they use different discrete approximation methods, i.e., how to represent the unknown variables (e.g., velocity $\mathbf{u}$ and pressure p) by spatial derivatives throughout the entire simulation domain. The process of discretization can be viewed as the matrix equation $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is a sparse matrix, $\mathbf{x}$ is the vector of unknown discretized variables, and $\mathbf{b}$ is the source terms and influence of boundary conditions. Finding the solution of the matrix equation either by inverting $\mathbf{A}$ or some efficient method is the key to them. In this section, we briefly introduce the fundamentals of the above methods.

Finite Difference Method (FDM)

Finite Difference Method divides the simulation domain into a **regular square/cube grid** of nodes, and uses "*finite differences*" of λ_j to approximate the derivatives of λ. To find them, we consider Taylor expansion of $\lambda(x)$ about x_j:

$$\lambda\left(x_\mathrm{j}+n\Delta x\right)=\lambda\left(x_\mathrm{j}\right)+(n\Delta x)\frac{\partial\lambda\left(x_\mathrm{j}\right)}{\partial x}+\frac{(n\Delta x)^2}{2}\frac{\partial^2\lambda\left(x_\mathrm{j}\right)}{\partial x^2}+\ldots \tag{2.9}$$

Then, *forward* difference, *central* difference, and *backward* difference can be derived to approximate for the first-order derivative as follows:

$$\left.\frac{\partial\lambda}{\partial x}\right|_{x_\mathrm{j}}\approx\frac{\lambda_{\mathrm{j}+1}-\lambda_\mathrm{j}}{\Delta x},\quad\left.\frac{\partial\lambda}{\partial x}\right|_{x_\mathrm{j}}\approx\frac{\lambda_{\mathrm{j}+1}-\lambda_{\mathrm{j}-1}}{2\Delta x},\quad\left.\frac{\partial\lambda}{\partial x}\right|_{x_\mathrm{j}}\approx\frac{\lambda_\mathrm{j}-\lambda_{\mathrm{j}-1}}{\Delta x}. \tag{2.10}$$

Next, FDM can use any of them to describe the derivatives in fluid dynamics equations.

FDM has the following advantages and disadvantages. It is simple in principle and can achieve second or higher-order accuracy. Besides, it can be used explicitly or implicitly in time. However, it cannot handle complex geometric flexibility on irregular grids, and is not perfectly conservative for quantities, e.g., mass, momentum, and energy, due to the truncation error of Taylor expansion [74]. Moreover, it belongs to the advective FD schemes, thereby has false diffusion issue [75]. At last, it also needs special methods to handle the checkboard instabilities [72].

Finite Volume Method (FVM)

Finite Volume Method is mainly used to solve conservation equations of mass, momentum, etc. It divides the simulated domain V into **smaller volumes** V_{i}, then uses divergence theorem to transform volume integrals to surface integrals, . FVM can handle complex geometries with different sizes and shapes. Typical software using FVM are OpenFoam [76], Ansys Fluent [77], Ansys CFX [78], Starccm+ [79], etc.

FVM has the following advantages and disadvantages. It is simple to implement and fast, and has a second or higher-order accuracy when using linear interpolation. Besides, it can be explicitly solved in time. However, the reconstruction for complex geometries by appropriate grids adds complexity. Besides, higher-order FVM is not straightforward to handle 3D irregular grids [74]. At last, FVM also suffers from checkboard instabilities.

Finite Element Method (FEM)

Finite Element Method solves PDE by the *weak form* integral, i.e., it multiplies the original PDE with a weight function $w(x)$ and integrates the product through the simulation domain. In particular, FEM uses space discretization to divide the simulation domain into smaller, simpler parts, i.e., finite elements λ_{i}. Then FEM interpolates λ_{i} with basis functions $\phi_{\mathrm{i}}(x)$, which determines the order of accuracy of FEM. FEM is successful in solid structure analysis, e.g., commercial software COMSOL [80].

FEM has the following advantages and disadvantages. It has high-order accuracy, and apply to complex unstructured geometries. However, it is an implicit method in time. Besides, it is not strictly conservative by default like FVM. Moreover, when compared with FDM and FVM, it adds complexity due to the integrals over unstructured grids. The checkboard instabilities may also appear and needs special handling.

Summary of conventional CFD methods

The three conventional CFD methods belong to generic numerical methods to solve the fluid continuum governing equations by approximating the derivatives in the PDEs. They describe fluid properties (e.g., velocity and pressure) by values in continuous fields of cells throughout the domain, but interpret them in different ways: the continuous field of FDM, FVM and FEM is defined on a square/cube grid of cells, the average of the fluid variables in a small volume around the cell, and an interpolation of the cell values, respectively. Due to the non-linearity, simultaneity, and the implicit pressure term in the incompressible NS equation, these methods will introduce complex iterative methods (e.g., SIMPLE, SIMPLER [81], etc.). They also add complexities to handle the flows in complex shapes, checkerboard instabilities, etc.

Particle-Based Methods

Instead of viewing fluid at the *macroscopic* scale and direct discretization on the continuum governing equations, this section briefly presents several particle-based methods to view the fluid at the *microscopic* or the *mesoscopic* scale.

Molecular Dynamics

Molecular dynamics (MD) uses atoms or molecules to track the position of particles, which interact by inter-molecular forces $f_{\mathrm{ij}}(t)$. It follows the Newton's second law and integrator algorithms (e.g., Verlet algorithm [82]). MD is mainly used to simulate microscale phenomena, e.g., phase changes, protein folding, chemical reactions, etc. However, since it tracks every individual molecule, e.g., a single gram of water contains 10^{22} molecules, it is far too detailed to be used for macroscopic phenomena.

Lattice Gas Automata (LGA)

As the predecessor of LBM, *Lattice gas automata* (LGA) [83] is first designed in 1973 as a simple 2D gas dynamics model with 4 velocity directions c_{i}, and is developed later with 6 velocity directions to simulate fluids in 1986 [84]. LGA introduces the occupations number n_{i}

(Boolean variables), and the concept of **collision** and **streaming**. During the collision step, particles distribution is redistributed by the conservation of mass and momentum. During the streaming step, particles move to a neighboring position.

The advantage of LGA is that its collision doesn't have the round-off error, and can be massively paralleled However, the occupations number is a Boolean number, representing that the presence of a particle *there* or *not there*, respectively. Thus, it has statistical noise and fluctuates strongly with a large number of time steps computations.

Lattice Boltzmann Method (LBM)

Lattice Boltzmann method (LBM, 1988) [72] is a relatively modern method and historically emerged from LGA in the 1980s. LBM eliminates LGA's statistical noise, and can be derived from the kinetic theory of gases. It is now proven to be of particular interest to CFD communities. More details about the theoretical fundamentals of LBM are described in Sect. 5.1.

1. LBM easily accommodates complex physics and boundary conditions (BC), so that it can also be widely used in multi-phase flow, multi-component flows , and reactive and suspension flows.

2. LBM is easy to parallelize to utilize current and future architectures, since its heaviest computation is local.

3. LBM has a strong physical basis since it originates from the Boltzmann equation. Since Boltzmann equation can represent non-hydrodynamic fluids with large molecular mean free paths, LBM can cover more physical phenomena than Navier-Stokes solvers.

4. LBM is simple and efficient to implement. LBM is an explicit scheme, involves cheaper computational cost [85], and takes out the advection operator.

5. LBM can handle transport phenomena (e.g., diffusion, temperature transport, etc.).

However, LBM has a drawback: it is a memory-bound algorithm. The common LBM implementation requires two copies of **probability distribution function** stored in the memory to prevent memory overwrite during collision and streaming cycles, which requires

18 and 38 double floating point variables per lattice cell in D2Q9 and D3Q19 model, respectively. This design requires more memory storage than the Navier-Stokes methods. And this motivates our second goal is to design the memory-aware LBM in Chap.5 and Chap.6.

Dissipative Particle Dynamics (DPD)

Dissipative Particle Dynamics (DPD, 1992) [86] is a young mesoscopic MD approach using Lagrangian approach instead of grids. DPD uses conservative, dissipative, and random forces to describe the interaction among clusters of particles. DPD usually applies to complex hydrodynamic fluids at the mesoscale, e.g., multiphase flows in complex geometries, suspended biological cells or polymers, etc. But since it requires careful selection of a large number of parameters (e.g., radial weight functions), the emergent hydrodynamic behavior can be influenced [72].

Direct Simulation Monte Carlo (DSMC)

Direct Simulation Monte Carlo (DSMC, the 1960s) [87] primarily solves high Knudsen number flows with a large l_{mfp}, e.g., dilute gases. It randomly selects pairs of statistically representative particles to collide on a collision model. However, its statistical error is inversely proportional to the number N of simulation particles, thereby its main drawback is the high computational cost.

Multi-particle Collision Dynamics (MPC)

Multi-particle Collision Dynamics (MPC, 1999) [88] is a modification of DSMC and usually applies to situation with a small l_{mfp}. It considers hydrodynamics and thermal fluctuation naturally. Besides, it is simple and easy to parallelize, and widely used in colloids, polymers, etc. But it is not simple in BCs for pressure, and not well controlled on the no-slip boundary.

Smoothed-Particle Hydrodynamics (SPH)

Smoothed-Particle Hydrodynamics (SPH, the 1970s) [89] is an interpolation sch111111111eme using overlapping blobs (SPH particles) that influence their vicinity. It follows the conservation of mass and momentum. However, SPH has issues with handling BCs, accuracy, and its mathematical proof of numerical consistency with hydrodynamics equations [72].

Summary of Particle-based CFD Methods

Particle-based methods represent fluid by atoms, molecules, clusters of molecules, etc. They vary a lot and have their own difficulties and specific applied domains.

3. PERFORMANCE ANALYSIS OF WORKFLOWS WITH STATE-OF-THE-ART IN-SITU SYSTEMS

A version of this chapter has been published in *HPDC '18, Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing* doi.org/10.1145/3208040.3208049.

This chapter presents an in-situ workflow benchmark that uses seven state-of-the-art in-situ systems or libraries to integrate simulation applications with analysis applications. [1] Then we compare their performance and use tracing tools to investigate the inefficiencies hidden in these in-situ workflows.

3.1 In-Situ Workflow Benchmark Setup

We use seven in-situ systems to build in-situ workflows, i.e., MPI-IO, ADIOS/DataSpaces, ADIOS/Flexpath, ADIOS/DIMES, native DataSpaces, native DIMES, and Decaf. The "ADIOS/name" indicates that we use ADIOS's interface and the specific data transport method of *name*. Otherwise, we call it a "native" method when using the intrinsic in-situ systems directly.

The workflow's producers use a parallel Lattice Boltzmann method (LBM) based CFD application, which simulates a flow in 3D channel iteratively in time steps. The simulation output is then transferred to the workflow's consumers, which run a parallel $n-th$ moment turbulence data analysis application [90].

The workflow experiments are performed on the *Bridges* system at the Pittsburgh Supercomputing Center. There are 752 regular compute nodes on *Bridges*, and each node has two Intel Haswell 3.3 GHz 14-core CPUs and 128GB memory. All nodes are connected with the Intel Omni-Path network, and use the Lustre parallel file system. More detailed system information is provided in Sect.4.3.

Tab. 3.1 presents the experimental setup information of the workflow experiments. We launch 256 simulation processes on 16 nodes to start a CFD simulation with an input of $16384 \times 64 \times 256$ channel and running 100 time steps. Thus each simulation process gets the

[1]↑The design of the in-situ benchmark is a joint work with Feng Li.

Table 3.1. Experimental setup of the in-situ workflow benchmark.

Global input size of 3D grid	$16384 \times 64 \times 256$ ($64 \times 64 \times 256$ per process)
#Simulation processes	256 processes on 16 nodes
#Analysis processes	128 processes on 8 nodes
Compute node information	Each node has 28 cores, 128GB of memory
#Data staging processes	DataSpaces: 32 server processes on 8 nodes DIMES: 32 server processes on 8 nodes Decaf: 64 Decaf-link processes on 8 nodes
#Time steps in the simulation	100 (every time step require a data analysis)
The *n*-th moment turbulence analysis	n=4
Total amount of data moved	400GB

subgrid 3D grid of $64 \times 64 \times 256$ and generates 16 MB data in each time step. [2] Therefore, for the global grid, all 256 simulation processes generate 40GB data in each time step, and 400 GB in 100 time steps. Then, the generated "source data" in each time step is sent to 128 analysis processes on 8 nodes, which means each analysis process receives data from two simulation processes, and executes the fourth-moment turbulence analysis. For DataSpaces and DIMES related experiments, 32 additional data server processes on 8 nodes are used. For the Decaf experiment, 64 link processes on 8 nodes are also in use.

All of our workflow implementations have been designed to overlap simulation with analysis time steps to obtain the best performance. For instance, Fig.3.1 illustrates how our workflow implementation can hide the analysis time when the simulation time is greater than the analysis time. A similar figure can also be drawn when the analysis time is greater than the simulation time. By using such a software design, either the simulation time or the analysis time can be totally hidden from the workflow execution time.

Tab. 3.2 particularly lists the software versions and configuration options that are used to install and build the tested benchmark. On Bridges, all the software and libraries are built

[2]↑Fore each fluid point, two velocities with the double-precision floating-point value on the X-axis and Y-axis are stored. Thus, in each time step, each process generates $64 \times 64 \times 256 \times 8B \times 2 = 16MB$ data per step. The reason that we don't store the velocities in Z-axis is that they are nearly 0 and changes little due to the property of the incompressible fluid.

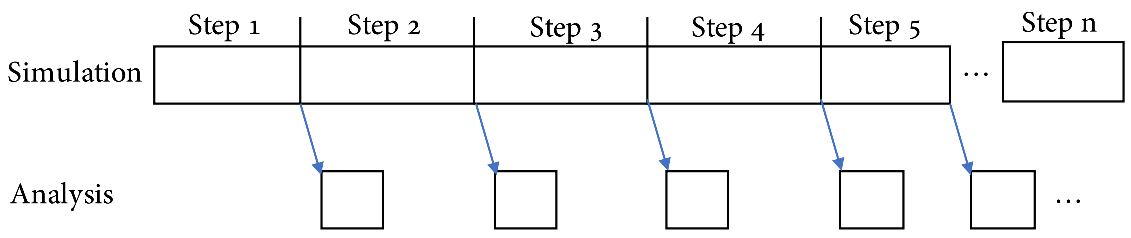


Figure 3.1. Our workflow implementations can overlap simulation and analysis using I/O transport libraries. In this example, we assume data analysis is faster than simulation for each time step.

Table 3.2. Configurations of different in-situ systems.

Software Tested	**Version**	**Build Configurations**	**Runtime Configurations**
ADIOS/DataSpaces and ADIOS/DIMES	DataSpaces 1.6.2, ADIOS: 1.13	Default ADIOS auto-config script	lock_type=1, hash_version=2
Native DataSpaces and Native DIMES	DataSpaces 1.6.2	–with-ib-interface=ib0, –with-dimes-rdma-buffer-size=1024	lock_type=2, hash_version=2
ADIOS/MPI-IO	ADIOS 1.13	Default ADIOS auto-config script	xml: type="MPI", without *time aggregation*
Flexpath	EVPath, ADIOS 1.13	perl chaos_bootstrap.pl adios-1.13	CMTransport=socket, CM_Interface=ib0
Decaf	https://bitbucket.org/tpeterka1/decaf, Git commit version used: 637eb58	mpi_transport=on	redist="count"

with gcc 4.8.5 and the Intel MPI library (2017 Update 3). [3] ADIOS 1.13 is used to combine with the in-situ (data staging) software. Furthermore, we perform large scale experiments using the MPI-IO, Flexpath and Decaf libraries on 13,056 cores as shown in Sect. 4.3.3 (cf. Fig. 4.13 and 4.15).

[3]↑The Bridges system's default Intel compiler and default Intel MPI library is not used due to DataSpaces's compatibility issues.

3.2 Experimental Evaluation

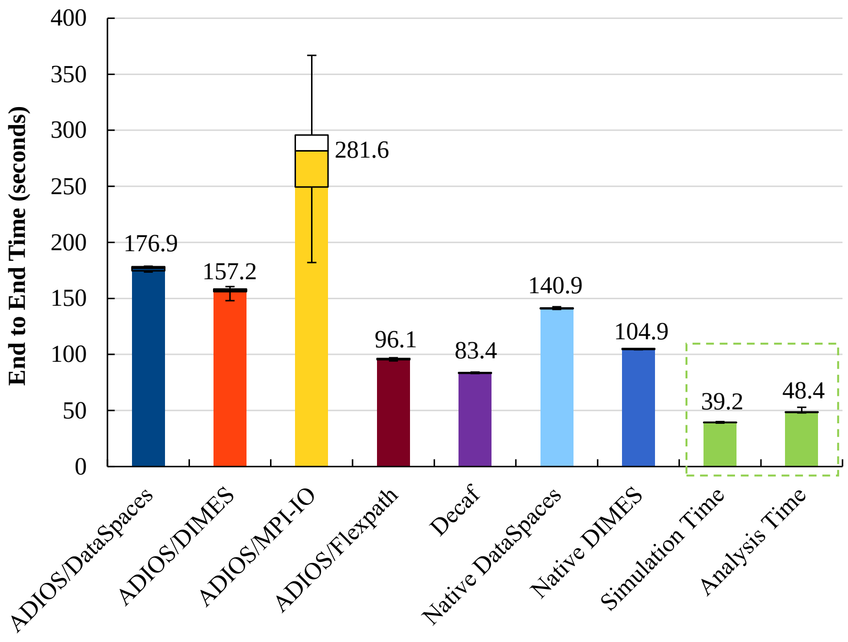


Figure 3.2. Performance of the CFD workflow application using 7 different I/O transport libraries, in comparison with the simulation time and analysis time.

Fig.3.2 shows the various end-to-end time of the CFD workflow experiments using different I/O libraries. The two green columns bounded by the rightmost block are the time when we running the LBM CFD simulation application alone and the analysis application alone. Among all workflows, Decaf achieves the best end-to-end time of 83.4s, followed by ADIOS/Flexpath of 96.1s. Then, the four workflow experiments using the I/O transport libraries (i.e., ADIOS/{MPI-IO, DataSpaces, DIMES, Flexpath}) combined with ADIOS. Although MPI-IO is the highly optimized file-based I/O method and also the most mature transport method provided by ADIOS, MPI-IO performs the worst among workflow experiments: it gives the longest and most variational end-to-end time. This is anticipated because MPI-IO writes data to a file system, which is also shared by many other users. However, MPI-IO in the fastest case can surprisingly achieve a performance that is comparable to the data staging methods (e.g., ADIOS/DataSpaces). Compared to the other three data staging methods with ADIOS (i.e., DataSpaces, DIMES, Flexpath), ADIOS/Flexpath achieves the best end-to-end time of 96.1s, the ADIOS/DIMES ranks second place with 157.2s, and the ADIOS/Data ranks the third place with 176.9s.

To investigate the problem of DataSpaces, we turn to the *native* DataSpaces and DIMES libraries. As a result, the native libraries give a 1.3X speedup for DataSpaces, and a 1.5X speedup for DIMES. The speedup reasons are as follows. Because ADIOS introduces a uniform interface for all transport methods including file-based and staging methods, users can switch to different I/O methods easily. However, to achieve this goal, low-level details in certain transport methods have to be hidden in this common interface, since configurations for one method might not be available in the other methods. For instance, *native* DataSpaces provides a customized lightweight lock strategy to enforce synchronizations among applications (e.g., dspaces_lock_on_write), but the native lock strategy is not exposed by the ADIOS interface and not supported by other methods (e.g., Flexpath). The *native* DataSpaces and DIMES workflow implementations take advantage of the multiple *native* locks, so that simulation processes can have finer control of locks, and continuously send multiple versions of data to staging servers with less interruption. By tuning the *version* and *lock* configurations in the native DataSpaces and DIMES (cf. Sect.3.2.1), a separate 30% and 50% speedup is achieved in the end-to-end time, compared with the general interface provided by ADIOS.

3.2.1 Performance Analysis of In-Situ Workflow Experiments

Based on the results in Fig.3.2, we can find there is still a big gap in the end-to-end time between those staging methods and the case of running simulation alone. For example, the best experimental result using Decaf workflow in Fig.3.2 show that the workflow execution time is 2.3X slower than the simulation-alone time, and 1.72X slower than the analysis-alone time. This implies that the performance of the simulation application has been considerably dragged down by using the combined simulation and analysis workflow model. This is not what workflow designers expected, since they expect the simulation application and analysis application are combined as seamlessly as possible, so that the total end-to-end time should be either simulation time or analysis time at the best case. To investigate why and where the performance is lost, We use TAU [91] and Intel Trace Analyzer and Collector (ITAC [92]) to collect traces of each workflow experiment. Here we only show the trace analysis results for the three fastest methods (i.e., the native DIMES, Flexpath, and Decaf) to reveal their major performance inefficiencies.

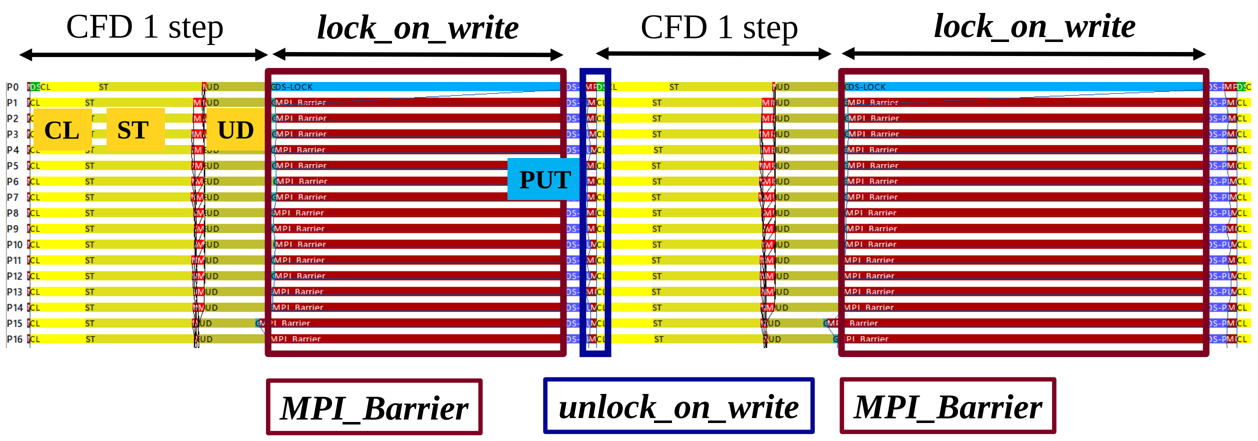


Figure 3.3. A trace of native DIMES with a snapshot of 2 seconds.

Fig.3.3 shows the trace for the native DIMES workflow during two consecutive time steps of CFD simulation. All 100 time steps have a similar performance pattern, we only display a snapshot to show details. Since the CFD simulation application launches 256 processes, to view the problem clearly, we zoom in on the first 16 simulation processes, and the other 240 processes synchronize at the same time frame in each time step. Within each CFD step, there are three phases marked with yellow blocks: collision (CL), streaming(ST), and updating local population (UD). There is a fourth “boundary condition” phase, but it is not shown in the trace, since it is a very short period compared with the other three phases. In the DIMES workflow, the simulation application needs to synchronize between metadata servers and computing processes, and then inserts results into the DIMES buffer. This insertion is marked with the blue blocks “PUT”, and then discharged by “unlock_on_write”. Notice that there is a lengthy “lock” period (lock_on_write), when each simulation process is performing data insertions. The lock period happens when the sender tries to insert data into DIMES, while the receiver (analysis process) has not fetched previous data yet. As a result, simulation is stalled with MPI_Barrier, and the stall time is almost equal to the time to execute one time step of CFD simulation.

The DIMES implementation is presented as follows: it uses the type-2 customized lock of DIMES, which is a collective lock and enforces strict synchronization between producers and consumers. To better overlap simulation with data analysis, and efficiently utilize the RDMA memory in DIMES, the DIMES workflow uses multiple locks. We use ($step \% num\_slots$) as the lock name so that we keep reusing a circular queue of multiple locks with a fixed size of num_slots, where $step$ is the time step index of the CFD simulation, and num_slots is the number of slots the CFD simulation can use to buffer its output data in a FIFO manner. When the analysis application is slower, the simulation application will be stalled in order to make sure the previous data are not overwritten. This explains why the simulation application stall time (MPI_Barrier) in Fig.3.3 is almost equal to one step of simulation time. Therefore, the end-to-end workflow time nearly doubles. [4]

Next, Fig.3.4 presents the trace of the Flexpath-based workflow implementation using TAU with a snapshot during three seconds for two different cases: 1) running simulation alone, and 2) running the Flexpath workflow. The orange stripes represent the time to execute the MPI_Sendrecv function, which performs the inter-process communication in the *streaming* (ST) phase of the LBM CFD simulation. We can see that after combining the Flexpath data staging method, the MPI_Sendrecv time in the LBM CFD simulation takes much longer, which results in the increased end-to-end time of workflow. Because both LBM's streaming computation and Flexpath's event channel involve intensive communications among different processes, Flexpath's data-staging operations will compete with the CFD simulation's MPI communication. In particular, when staging a large slab of simulation data (e.g., 16 MB per time step per process in this workflow experiment), the chances to have communication interference are much higher. However, we don't see such intrusion in DIMES because the data access pattern: instead of routing data immediately after a com-

[4]↑DIMES workflow implementation can be further optimized by using an additional thread in the consumer application to fetch a newer version of data while the main thread is analyzing the data of previous time steps. But this requires additional modification and instrumentation when comparing to other methods.

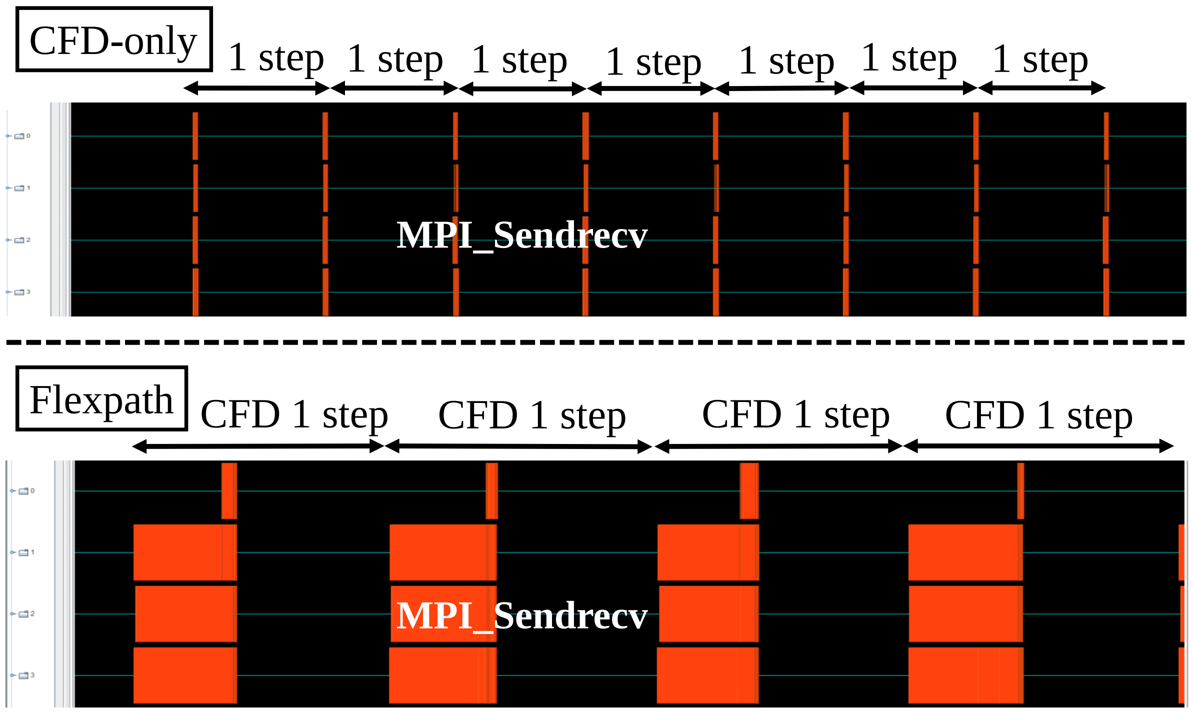


Figure 3.4. Comparison between running CFD simulations only and running Flexpath based workflows. This figure shows a snapshot of 3 seconds.

putation step, DIMES only buffers data in a sender's local memory, and the transmission won't start until a receiver issues a "get" command.

Finally, we compare the fastest workflow implementation using the Decaf method to the experiment that runs simulation only. We are not able to use TAU for the tracing purpose, because the latest TAU library (version 2.27) cannot filter out the huge number of inline *Boost* serialization function calls made by Decaf. The inline function calls make the trace files too large to generate. To circumvent the tracing problem, we manually instrument the workflow source code, and use the Intel Trace Analyzer and Collector (ITAC) to collect execution traces.

Fig.3.5 shows the two traces for CFD simulation only, and Decaf-based workflow, respectively. In the trace snapshot of CFD simulation only, CFD simulation itself can execute 3 time steps during 0.9 seconds. By contrast, in the lower Decaf-based workflow trace, there is an additional PUT function invoked by simulation processes to transfer output data to 64 Decaf-link processes. We can first observe that the PUT function utilizes a collective

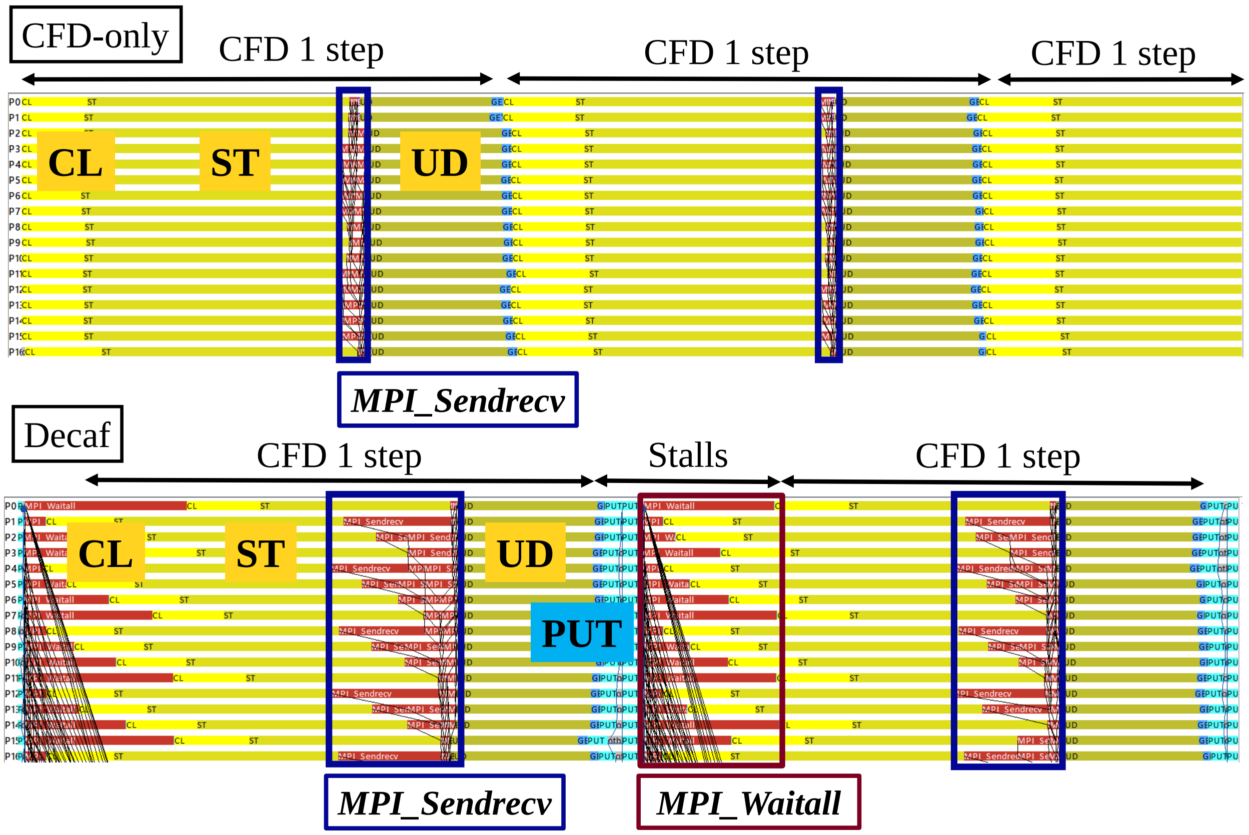


Figure 3.5. Comparison between running CFD simulations only and running Decaf-based workflows. This figure shows a snapshot of 0.9 seconds.

"MPI_Waitall" function (marked with red blocks) and cause all simulation processes to stall. This is because Decaf has to make sure data is safely stored in the link processes before it can proceed to the next step. Besides, we observe that the "MPI_Sendrecv" time in the streaming (ST) phase increases significantly once Decaf is added to the workflow. This indicates that using Decaf also has affected the MPI communication performance of the original LBM CFD simulation application.

4. ZIPPER IN-SITU SYSTEM

A portion of this chapter was previously published in *HPDC'18, Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing* [55] doi.org/10.1145/3208040.3208049 and *ICCS'16, International Conference on Computational Science* [56] doi.org/10.1016/j.procs.2016.05.297

This chapter will first introduce the design and implementation of the Zipper runtime system in Sect. 4.1, and then the related work of in-situ or data staging libraries in Sect. 4.2, and evaluate the experimental results among Zipper and other libraries in Sect. 4.3.

4.1 Design and Implementation

From the performance analysis of workflow experiments in Chap.3, we can find several performance issues and optimization opportunities:

1. The cost of locking service and the staging-server access including the server query and data transfer can be reduced or removed (e.g., DataSpaces and DIMES have such a cost).

2. The enforced global barriers, which are used by all simulation processes to insert (write) data to data staging processes, and are also used by all analysis processes to retrieve (read) from data staging processes, can be reduced or removed (e.g., Decaf and Flexpath have such barriers).

3. The I/O data transfer time from simulation to analysis processes between consecutive simulation steps can be decreased or potentially hidden by the computation time of the simulation, if an early-start fine-grain pipelining approach is used (e.g., I will increase the degree of task-level parallelism and use pipelining to overlap all simulation, analysis, and I/O tasks).

4. To interfere less with the simulation's communication, instead of transfer a burst of large data block (e.g., Decaf and Flexpath have experienced increased MPI communication time in the simulation application), asynchronous fine-grain-block data transfer can be used, so that we can have more balanced network traffic.

The rest of this section will introduce a new runtime system called **Zipper** to improve the above identified performance inefficiencies.

4.1.1 System Overview

From the design view of Zipper system design, both simulation and analysis applications are executed in parallel and located on different compute nodes of an HPC system or cloud system, without data staging servers on extra compute nodes. For instance, m compute nodes are used to execute the simulation application with M processes, and n compute nodes are used to execute the data analysis application with N processes simultaneously. There are two reasons to separate them. 1) simulation and analysis applications may have different resource requirements and scaling capabilities, we had better not let them interfere with each other, especially for large-scale scientific applications, which might be compute-bound, memory-bound, or communication-bound. 2) multiple failure domains can be supported if separated, since the failure of the simulation or analysis application may not interfere with the others.

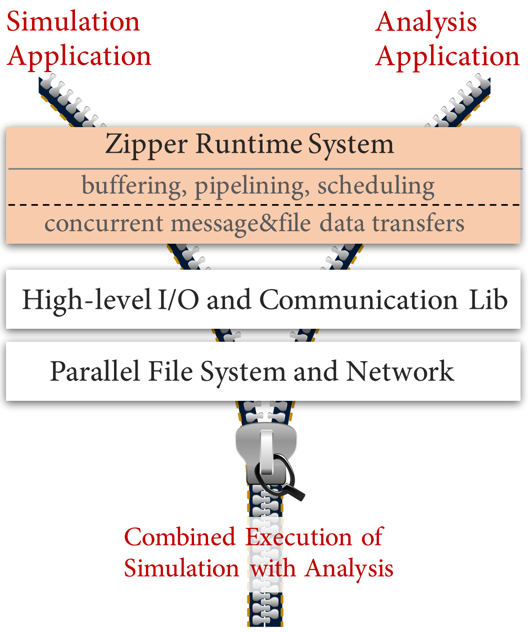

Simulation Application
Analysis Application
Zipper Runtime System
buffering, pipelining, scheduling
concurrent message&file data transfers
High-level I/O and Communication Lib
Parallel File System and Network
Combined Execution of Simulation with Analysis

Figure 4.1. The Zipper runtime system.

Fig.4.1 conceptually shows that the Zipper runtime system is located below the application layer, and above the high-level I/O and communication libraries. The Zipper runtime system itself has two strata: 1) The upper stratum provides the functions of buffering data in memory, pipelining data blocks from simulation to analysis applications, and scheduling data transfer operations and data analysis tasks; 2) The lower stratum is an optimization layer, which can transport computed results by two concurrent channels: low-latency HPC network and file-based parallel file system.

Zipper library has two separate parts, and each part is embedded at the node of simulation or analysis processes. It will first slice the large source data into fine-grain blocks, and then asynchronously transfer them using both network and file I/O to the analysis application. As a result, the simulation application can push the generated source data to the analysis application continuously and seamlessly. Therefore, the analysis application is driven by data-availability, i.e., whenever a new data block arrives, it can be immediately read and processed.

Fig.4.2 presents the architecture of the Zipper. Thus, simulation processes write source data to Zipper, while analysis processes input (read) data from Zipper. The interface provided by the Zipper runtime is simple: `Zipper.write(block_id, void* data, block_size)` and `Zipper.read(block_id, void* data, block_size)`. Simulation processes call the Zipper.write() method to pass the generated source data to the *Producer Runtime Module*. The producer runtime module is multi-threaded and provides the essential functionalities of buffer management, asynchronous I/O, data prefetching, communication with consumers, and the concurrent data transport optimization (cf. Sect. 4.1.3). On the other side, each analysis process works as a consumer and calls Zipper.read() to interact with its *Consumer Runtime Module* to retrieve, analysis and optionally store data constantly. Both producer and consumer runtime modules can utilize the low-latency HPC network and high-performance parallel file system (e.g., Lustre) to transport and store computed results.

The Zipper runtime system offers two modes to users: *Preserve* mode and *No Preserve* mode. A user may choose the *Preserve* mode to keep the computed results for future analysis, validation, and verification. On the other hand, one may also choose the *No Preserve* mode

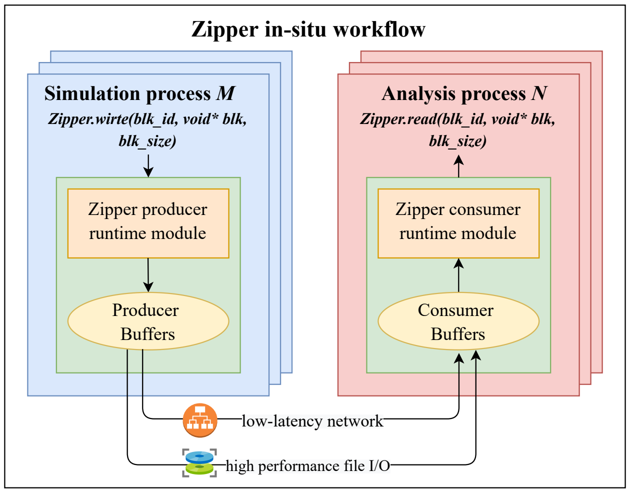

Zipper in-situ workflow
Simulation process M
Zipper.wirte(blk_id, void* blk, blk_size)
Zipper producer runtime module
Producer Buffers
Analysis process N
Zipper.read(blk_id, void* blk, blk_size)
Zipper consumer runtime module
Consumer Buffers
low-latency network
high performance file I/O

Figure 4.2. The architecture of the Zipper workflow framework to integrate a parallel simulation application with a parallel analysis application.

to save time and storage space without writing the output data to the file system, and only perform faster online experiments.

4.1.2 Implementation

Fig.4.3 presents the producer runtime module, whose function is buffer management, asynchronous I/O, data prefetching, communication with consumers, and the concurrent data transport optimization. It consists of a producer ring buffer, a sender thread, and a writer thread. The sender thread is responsible for sending data blocks to the consumer processes via the HPC network. The writer thread is responsible for storing computed results in a parallel file system. More specifically, when the simulation application calls the Zipper.write() and passes the source data into the prodcuer module, the data will be sliced into fine-grain data blocks with a user-defined $block\_size$, packed with a $blkid$ at the time step t, and inserted into the producer buffer. Once completing the insertion, the simulation process can return back to its computation. Note that each data block can also be packed with all the necessary information to support the analysis later performed in the analysis application, e.g., the simulation process ID that sends the block, the coordinates of each data point, etc. Next, the sender thread will fetch a data block from the producer buffer, and check whether there are blocks stored on disks by reading an array of "*block IDs on disk*".

1. If no blocks have stored on disk by the writer thread, the sender thread will form a "pure data" message directly send it through HPC network to the statically mapped analysis process.
2. If some blocks are fetched by the writer thread and have completely stored on disk during the period when the sender thread is sending the messages or fetching a new data block, the sender thread will smartly detect and append the on-disk block IDs to form a *mixed message*, and then send it to the analysis process.
3. If the producer buffer is empty and some data had been stored to disk (e.g., at the last step of simulation), the sender thread will form an "on-disk block IDs" message through HPC network the analysis process.

The reason to bring in the writer thread is that when the analysis application is slower than the simulation, the simulation application will not be blocked or stalled since the writer thread is also moving data to the parallel file system. In Sect. 4.1.3, we will describe how the writer thread can help the sender thread to increase data transfer rate by using a concurrent dual data-path method.

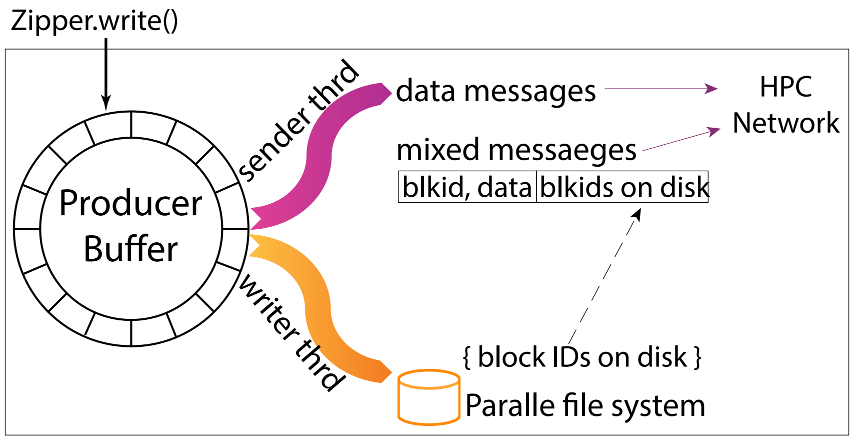


Figure 4.3. The producer runtime module.

Fig.4.4 presents the consumer runtime module, which consists of a consumer buffer, a receiver thread, a reader thread, and an output thread. For the receiver thread, it will perform three different actions according to the message type:

1. If the receiver thread gets a mixed message from the HPC network, and parses it into a data block and a list of block IDs. The data block will be picked up and directly inserted into the consumer buffer, while the block IDs will be copied to an array of "*block IDs on disk*".

2. If the receiver thread gets a "pure data" message, it will directly be inserted into the consumer buffer.

3. If the receiver thread gets an "on-disk block IDs" message, it will copy the block IDs to the "*block IDs on disk*" array.

According to the *block IDs* in the array, the reader thread will read the blocks from the parallel file system one by one, pack an additional flag *on_disk = true*, and then insert them into the consumer buffer. Whenever a data block is in the consumer buffer, it will be pushed to the analysis process by Zipper.read(). Since each data block contains all the necessary data, the analysis process can apply appropriate data analysis to it, and then mark it with the flag *is_analyzed=true*.

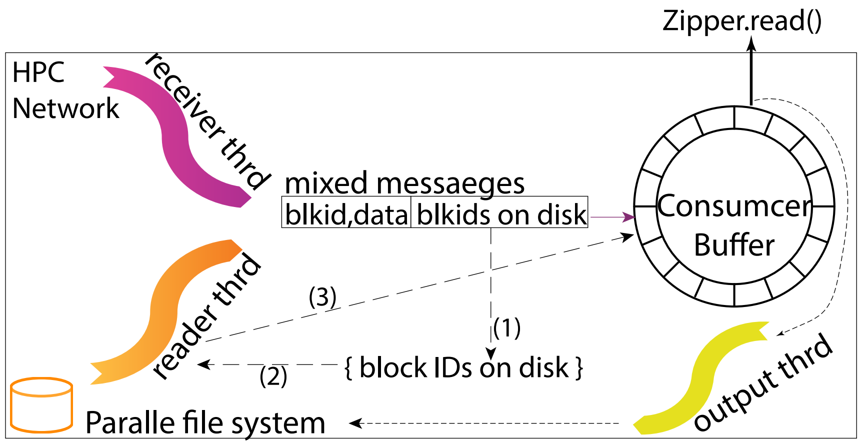


Figure 4.4. The consumer runtime module.

The output thread in Fig.4.4 is dedicated to supporting the *Preserve* mode, when users need not only to analyze the computed results, but also to preserve (store) the original source data from simulation processes for future analysis. The output thread constantly fetches data blocks in the consumer buffer, which have been analyzed by the consumer process. 1) If the fetched data block has a flag of *on_disk* = *false*, the output thread will store the data block to the file system. After storing the block, it sets the block's flag to *on_disk* = *true*. 2) If the fetched data block has a flag of *on_disk* = *true*, the output thread does nothing and fetches the next data block. A data block in the consumer buffer can be freed from the system only if the block has been both analyzed by the analysis process and stored in the file system by the output thread. To free a data block from the system, we utilize two flags of *on_disk* and *is_analyzed* associated with each block. When both *on_disk* = *true* and *is_analyzed* = *true*, Zipper.free() can be called by the analysis process, so that the data block will be released by the Zipper runtime system. Note that the output thread will not be created by Zipper in the *No Preserve* mode.

4.1.3 Optimization of Concurrent Message and File Data Transfers

The Zipper runtime system utilizes two data transport paths: 1) message passing via a low-latency HPC network, and 2) parallel I/O via a parallel file system. The reason to use the parallel file system is that to mainly alleviate the simulation stall issue. In the case of without using it, when the analysis application is relatively complex and slow, or users choose the Preserve mode to store data into the parallel file system, a data block that reaches the analysis node will take longer time to be consumed, thus the producer buffer at the simulation node will be full, then the simulation application will be blocked to progress. This is especially common when the simulation application is data-intensive (e.g., CFD simulation). The large-scale scientific application could generate huge data in every time step, e.g., the experiment in Sect. 3.2 generates 400GB in 100 time steps. Therefore, when the producer buffer nearly reaches a "threshold", if a writer thread helps to continually fetch data blocks from the producer buffer and store them into the parallel file system, the producer buffer will not be full to stall the simulation. Thus, the simulation can be interrupted much less frequently than the state-of-the-art data staging libraries that only uses HPC network.

On the other hand, using two data paths has the potential to increase the data transfer rate if a portion of the data movement work is offloaded to parallel file I/O. Fig.4.5 explains how the concurrent transfer optimization works. The top part presents that all data blocks are only transferred by the network. The bottom part shows that most blocks are transferred by the network while a portion of blocks is transferred by parallel file I/O. Considering that emerging HPC systems will deploy much faster non-volatile memory (NVM) technologies and separate I/O networks among I/O servers, the Zipper workflow on future HPC systems will benefit more from this optimization.

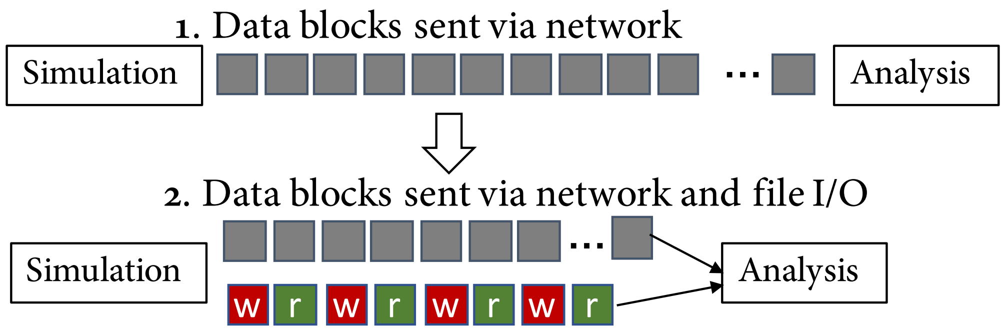

1. Data blocks sent via network
Simulation
Analysis
2. Data blocks sent via network and file I/O
Simulation
Analysis
w
r
w
r
w
r
w
r

Figure 4.5. The concurrent data transfer method can reduce the data transfer time by converting a portion of message passing time to certain overlapped parallel file I/O time.

The *concurrent data transfer optimization method* implementation also uses the *work-stealing* algorithm, which allows data blocks to be transferred through the parallel file system path only when it is necessary. The writer thread in the producer runtime module will do this job as a helper. When detecting the producer buffer is almost full (defined by a "high water mark" threshold), the writer thread will fetch a data block from the buffer and write it to the file system. Algorithm 1 shows the pseudocode of the writer thread. This strategy can automatically adapt to either the message-passing-only method or the mixed network&file-IO method depending on how full or empty the producer buffer is. For example, if the buffer is constantly near-empty, Zipper will always choose the fastest HPC network to send data to

Algorithm 1 Writer Thread Work-stealing Algorithm

```
1: while true do
2:   block ← StealBlock(ProducerBuffer)
3:   store the block to the parallel file system
4:   place the block's ID into the in-memory data structure of block IDs on disk
5: end

6: function StealBlock(ProducerBuffer)
7:   while true do
8:     acquire the lock of ProducerBuffer
9:     if number of Blocks in ProducerBuffer > Threshold then
10:      fetch the address of the first block in ProducerBuffer
11:      release the lock of ProducerBuffer
12:      return the address of the block
13:    else
14:      wait on a condition variable and release the lock
15:      /* Note: the generator thread that produces fine-grain data blocks will signal the condition
         variable when number of Blocks in ProducerBuffer > Threshold. */
```

the analysis application (Sect. 11 shows the experiments and effect of using the concurrent data transfer optimization).

To monitor network traffic and verify the speedup reason by using the concurrent data transfer optimization method, hardware performance counters can be used. If an HPC system has two separate networks (i.e., one for message passing and the other for I/O traffic), the concurrent data transfer optimization can be expected to increase the data transfer rate. If an HPC system does not have a separate interconnect network and I/O network (such as the Bridges HPC and the Stampede2 HPC in Sect. 4.3), the concurrent data transfer optimization may not be able to achieve its potential best benefits. However, a significant speedup can still be observed on the two HPC (detailed experiments are shown in Sect. 4.3). The reason is briefly explained as follows. Because most interconnect networks (e.g., InfiniBand and Omni Path Architecture (OPA)) have network congestion control mechanisms, when many simulation processes simultaneously attempt to transfer data to many analysis processes, network congestion control in network switches will play a key role in the communication performance. The concurrent data transfer optimization method is more efficient in working with the congestion control mechanism because the dual paths allow messages (i.e., the data blocks) to arrive at the receiver side out of order when using different network paths, and to take advantage of multiple network links/switches for improved bandwidth. In-depth network performance analysis will be presented in Sect. 11.

Summary of Zipper's features

1. Zipper uses fine-grain data blocks to create a higher degree of task parallelism which accelerates the pipeline execution. All the other data-staging workflow systems transfer one huge data block during each time step.
2. Zipper does not impose strict barriers between time steps, and deploys a data-flow-driven approach to minimizing application stalls. The other workflow systems often force the insertion of strict writer-reader interlocks and collective global operations (e.g., MPI_wait_wall, global locks).
3. The overhead of the data staging server is not involved, which is different from DataSpaces, DIMES and Decaf.
4. Zipper provides multiple failure domains (similar to DataSpaces, DIMES, and Flexpath, but Decaf doesn't support them).
5. Zipper supports both Preserve mode and No-Preserve mode, and introduces a concurrent data transfer optimization, which is based on an adaptive work-stealing algorithm.

4.1.4 Performance Model

To evaluate the efficiency of Zipper theoretically, we use a simplified analytical performance model to estimate the workflow end-to-end time. The performance model uses the following notation. All the simulation processes use totally P processor cores, and all the analysis processes use totally Q processor cores. Given that the total data generated by the simulation during all time steps is D and each fine-grain data block size is B, there will be $n_b = \frac{D}{B}$ blocks requiring transmission. [1]

[1]↑Block size between 1MB and 8MB is used in the later experiments.

To keep the performance model simple, each simulation processor core computes $\frac{n_b}{P}$ blocks, and each analysis processor core analyses $\frac{n_b}{Q}$ blocks. However, the model can be also adapted to support load imbalance situations by considering the specific process with the maximum workload. The performance model is based on the time spent on each data block. Since the workflow uses the pipelining parallelism to couple simulation and analysis applications, a whole source block in one time step will go through 4 different stages in the non-integrated design (upper) of Fig.4.6: Simulation (compute) $\rightarrow$ Transfer data blocks (including output and input) $\rightarrow$ Analyze .

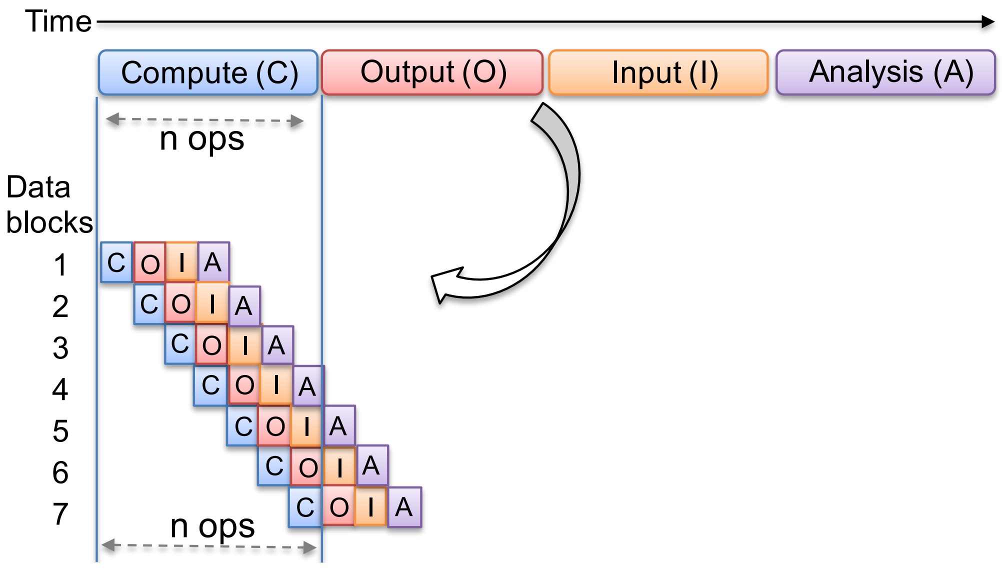


Figure 4.6. Non-integrated design (upper) vs. integrated design (lower). In the (lower) integrated design, at any time, four stages (C, O, I, and A) are working on four distinct data blocks. The four data blocks could be sequentially dependent, but can still be processed in parallel due to the data pipelining parallelism.

In the Zipper workflow, let t_{comp}, $t_{transfer}$, and t_{analy} denote the time on a fine-grain data block to perform simulation (computation), transfer, and analyze, respectively. Thus, the parallel computation time is $T_{\text{comp}} = t_{comp} \times \frac{n_b}{P}$, and the parallel analysis time is $T_{\text{analysis}} = t_{analy} \times \frac{n_b}{Q}$. Because each pipeline stage works independently from other stages, the end-to-end time-to-solution T_{t2s} can be expressed as follows:

$$T_{t2s} = \max(T_{\text{comp}}, T_{\text{transfer}}, T_{\text{analysis}}) \tag{4.1}$$

This formula is under the assumption that the number of data blocks is much larger than the number of pipeline stages so that the pipeline *startup* time and *drainage* time can be ignored. The simplified T_{t2s} formula can be easily derived from the integrated design (lower) of Fig.4.6: different stages are overlapped such that the end-to-end time is almost equal to the time of the slowest stage. Based on the model, if the simulation application and analysis application are scalable, the Zipper workflow can scale well accordingly. Note that the data transfer time T_{transfer} can be controlled by the frequency of outputting the simulation data (e.g., one data output per k time steps) to reduce the I/O time. Therefore, if every stage is seamlessly combined , the performance model shows the end-to-end time should be theoretically equal to the time of one stage. Sect.4.3 performs a variety of experiments to verify the model.

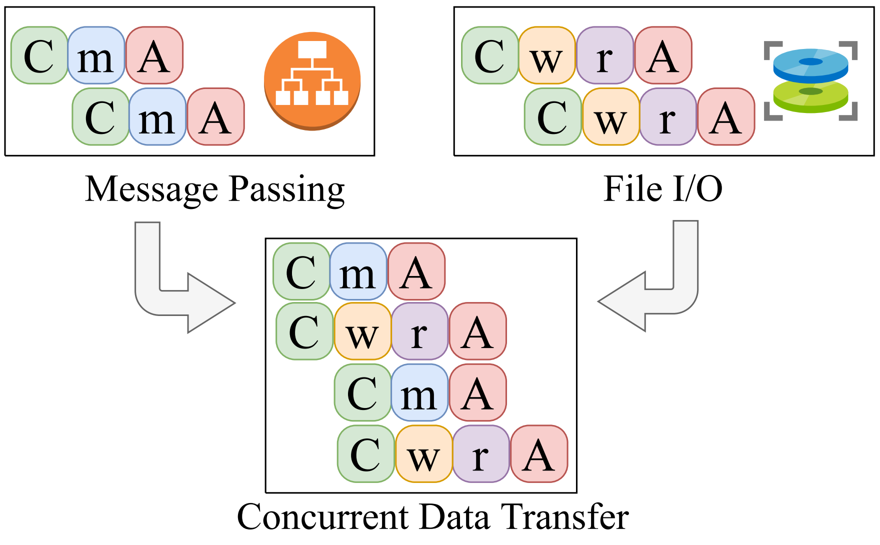


Figure 4.7. The Zipper performance model in *No Preserve* mode when using the concurrent data transfer method.

Furthermore, the more detailed performance model for Zipper in the *No Preserve* and *Preserve* can be derived when using the concurrent data transfer method. In the *No Preserve* mode, as shown in Fig.4.7, suppose $p\%$ of blocks are transferred through *files* (disk write $\boxed{\texttt{w}}$ and read $\boxed{\texttt{r}}$ on the upper right), meanwhile $(1 - p\%)$ of blocks are transferred through *messages* ($\boxed{\texttt{m}}$ on the upper left). Let t_{msg}, t_{wr}, t_{rd} denote the time to transfer a block by using networks, and a pair of file writing and reading, respectively. Then, the message passing time is $T_{msg} = t_{msg} \times \frac{n_b}{P} \times (1 - p\%)$, disk write time is $T_{wr} = t_{wr} \times \frac{n_b}{P} \times p\%$, and disk read time is $T_{rd} = t_{rd} \times \frac{n_b}{Q} \times p\%$. Thus, the time-to-solution for the $No\,Preserve$ pipeline can be expressed as follows:

$$T_{t2s}^{NoPreserve} = \max(T_{comp}, T_{msg}, T_{wr}, T_{rd}, T_{analy}). \tag{4.2}$$

If $p\% = 0\%$, the two terms T_{wr} and T_{rd} can be ignored from the above formula, indicating the message passing only method is used.

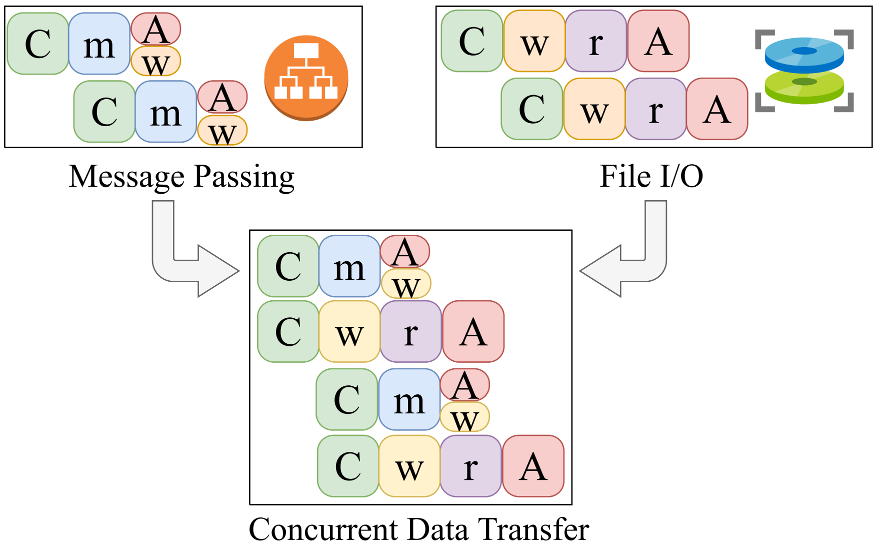


Figure 4.8. The Zipper performance model in *Preserve* mode when using the concurrent data transfer method.

Fig.4.8 shows the Zipper performance model in *Preserve* mode when using the concurrent data transfer method. Since the *Preserve* mode requires the additional operation of storing the intermediate data to disks, we let T_{analy_wr} denote the time to store data in the analysis processes, $T_{analy\_wr} = t_{wr} \times \frac{n_b}{Q} \times (1 - p\%)$. The time-to-solution in the *Preserve* mode can be extended as follows:

$$T_{t2s}^{Preserve} = \max(T_{comp}, T_{msg}, T_{analy\_wr}, T_{wr}, T_{rd}, T_{analy}) \tag{4.3}$$

Here, T_{msg} and T_{analy_wr} are proportional to $(1 - p\%)$, and T_{wr}, T_{rd} are proportional to $p\%$ of all the blocks.

Notice that $T_{t2s}^{Preserve}$ is more generic than the model of $T_{t2s}^{No\,Preserve}$, since by turning off the function of storing data to disks in analysis processes, $T_{t2s}^{Preserve}$ will become the same as $T_{t2s}^{No\,Preserve}$.

4.2 Related Work

As an alternative to in-situ approaches, *data staging* approaches can enable co-analysis pipelines by using a loosely coupled integration model. ADIOS [40], PreDatA [93], GLEAN [13], DataStager [15], DataSpaces [41], DIMES [53], and Flexpath [42] leverage advanced I/O infrastructure to reduce the I/O cost. In particular, PreDatA [93] realizes in-transit data processing on a data flow. It moves data from compute nodes to staging nodes through two passes: the first pass of sending data-fetch requests to the staging nodes, followed by the second pass of *pulling* packed data chunks from the compute nodes. We use a single pass to move data to the analysis processes rapidly. DataSpaces [41] and DIMES [53] allow different applications to store data into and extract data from dedicated servers (or metadata servers) simultaneously. Our Zipper system does not use dedicated servers and has no accompanying server access overhead. Sun et al. [94] use DataSpaces and asynchronous coupling of workflows as a use case to develop scheduling policies for placing data to different staging cores. GLEAN [13] and DataStager [15] deploy a data staging service on analysis nodes of a cluster to support in-situ processing. FlexIO [69] uses local memory and RDMA to support co-analysis either on the same compute nodes or on different staging nodes. Our research shares the data-staging philosophy of these libraries (e.g., data coupling at runtime and multiple failure domains), but uses fine-grain data blocks, asynchronous task parallelism, and holistic end-to-end level pipelining to minimize application idle time, reduce network contention, and overlap all workflow stages (i.e., simulation, data write, data read, and data analysis).

Our concurrent data transfer optimization method improves the communication throughput by taking advantage of the network congestion control and multiple switches and links. Our deployed network congestion measurement is inspired by the work of Alali et al. [95], which conducts a study to understand whether network congestion occurs on production

HPC systems. There are also studies that investigate how to use Quality of Service (QoS) mechanisms to enhance communication. Reinemo et al. compare a list of QoS capabilities on InfiniBand, Advanced Switching, and Ethernet [96]. Gonsiorowski et al. create a model to analyze the use of QoS lanes to reduce the impact of the RAID *rebuild* traffic by assigning different traffic quotas to read, write, and rebuild operations. [97]. Kim et al. design an OpenSM (Open SubnetManager) based scheme to adjust the QoS level dynamically by considering the estimated bandwidth and requirement to increase the overall bandwidth of multiple concurrent traffic [98].

Workflow systems such as Pegasus [34], Kepler [33], Taverna [37], and Condor/DAGMan [99] use files to communicate data and target coarse job-level meta-scheduling.

Decaf [11] is a workflow middleware that uses multiple overlapping MPI communicators and a special staging area called "link" to transfer data between a producer and a consumer. The communication among Decaf producer, link, and consumer are inter-locked, and all data must arrive in the link before they can be forwarded to the next application. Also, slower consumers will block the producers from running. Swift/T [31] uses a Swift-Turbine compiler to translate a Swift program to an ADLB [100] MPI program, and executes it with a master-worker model. Differently, we target fine-grain tasks and asynchronous computing, and use data-staging to minimize the workflow latency.

4.3 Experimental Evaluation

This section designs experiments to evaluate the Zipper runtime system by the performance model, the effect of Zipper's concurrent message and file transfer optimization method, and comparing the scalability performance between the Zipper and other state-of-the-art data staging libraries. The first two experiments are conducted on *Bridges*, and the third experiment is on *Stampede2*.

The *Bridges* system in the Pittsburgh Supercomputer Center (briefly mentioned in Sect.3.2) contains 752 regular compute nodes, 42 large shared-memory nodes (3TB memory each), and 4 extreme shared-memory nodes (12TB memory each). Each regular compute node has 28 Intel Haswell cores on 2 sockets and 128GB DRAM. The *Bridges* system deploys a 100

Gbps Intel Omni-Path Architecture, which connects all compute nodes with a 10PB high performance Lustre parallel file system $Pylon5$.

The *Stampede2* system in the Texas Advanced Computing Center entered full production in August 2017. It has 4,200 Knights Landing (KNL) nodes. Each KNL node has 68 cores on one socket, 16GB of MCDRAM (Multichannel DRAM), and 96GB DRAM. *Stampede2* also deploys an Intel Omni-Path Architecture and has a 30PB Lustre parallel file system.

The experiments performed in this section are three synthetic applications that are used to verify the accuracy of the Zipper analytical mode and evaluate the effectiveness of Zipper's concurrent data transfer method, and two real-world scientific computing applications which are used to compare the real workflow performance between Zipper and other libraries . Their description is presented in Tab.4.1.

Table 4.1. Description of the applications used in the experiments.

Workflow	**Simulation**	**Data analysis**
Synthetic $O(n)$	To emulate $T(n) = O(n)$ linear algorithms	Standard variance computation
Synthetic $O(n \log n)$	To emulate $T(n) = O(n \log n)$ such as divide & conquer algorithms	Standard variance computation
Synthetic $O(n^{3/2})$	To emulate $T(n) = O(n^{3/2})$ algorithms such as matrix computations	Standard variance computation
CFD application	Use the Lattice Boltzmann method to compute 3D channel flows	Turbulence analysis
LAMMPS application	Use LAMMPS to compute 3D Lennard-Jones atoms melt dynamics	Atoms movement statistics

4.3.1 Evaluation of the Performance Model

The performance model described in Sect.4.1.4 shows that the Zipper workflow can ideally achieve the end-to-end time to solution $T\_t2s = \max(T_{\text{comp}}, T_{\text{transfer}}, T_{\text{analysis}})$. The first group of experiments is intended to verify whether the performance model conforms to the actual Zipper workflow's performance. The experiments were performed on *Bridges* with 784 simulation process using 1,568 CPU cores, and 392 analysis processes using 784 CPU cores in both *No Preserve* and *Preserve* modes. In these experiments, a total amount of 3,136GB of data are transferred from simulation to analysis.

Fig. 4.9 shows the *No Preserve* mode's time breakdown for six synthetic workflows, which use the $O(n)$, $O(n \log n)$, and $O(n^{3/2})$ applications listed in Tab.4.1 with 1MB and 8MB block sizes. In each synthetic workflow, each data block is analyzed and its standard variance is reduced to one double-precision floating point value. For each block size (i.e., 1MB and 8MB), we measure the total time in three separate stages, i.e., simulation (blue), data transfer (orange), analysis (yellow), and then the workflow's end-to-end time (green). Let's first look at the left group with 1MB block size, as the application's time complexity T(n) increases from $O(n)$ to $O(n^{3/2})$, the simulation time (blue) on each data block becomes longer from 2.1s to 64s. Thus, the dominant stage switches from data transfer time (orange) in the $O(n)$ workflow to the simulation time (blue) in the $O(n^{3/2})$ workflow. However, regardless of the distinct time complexity in each synthetic application, the workflow's end-to-end time is always close to the maximum stage time, which empirically validates the performance model. The same phenomena can be found at the right group of three workflow experiments with 8MB block size.

Next, the experiments using the *Preserve* mode with the same configurations are performed. Fig. 4.10 shows the corresponding time breakdown and total time. An extra column of store data (dark red) is added. These experiments show that the end-to-end workflow time is almost equal to the time spent on storing computed results to the file system. Since all 784 simulation processes generate a total amount of 3,136 GB of data, storing data to the parallel file system takes the longest time.

Moreover, we evaluate the performance model with two real-world applications of CFD and LAMMPS. Their results are shown together with the weak-scalability experiment (cf. Sect.4.3.3). For the CFD and LAMMPS applications, the Zipper workflow end-to-end time is nearly the same as the dominant simulation time.

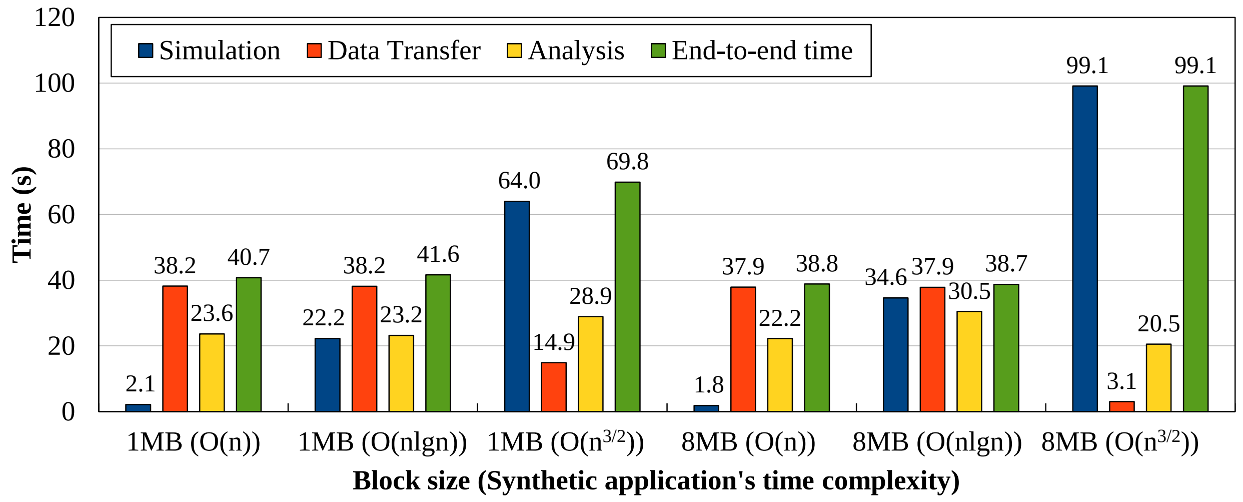


Figure 4.9. Time breakdown of the execution time for three different synthetic applications in the *No Preserve* mode.

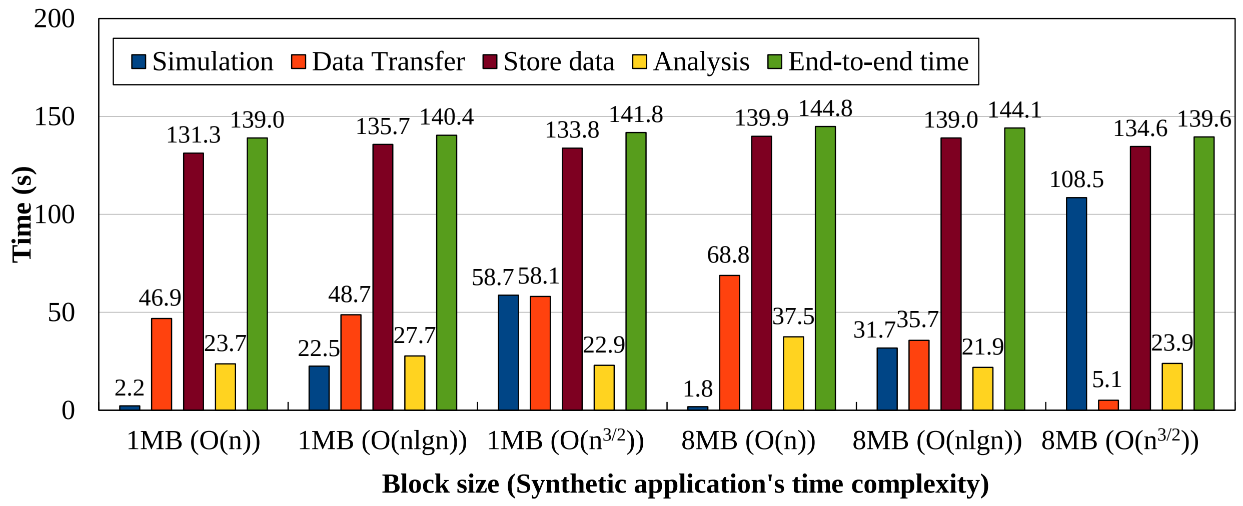


Figure 4.10. Time breakdown of the execution time for three different synthetic applications in the *Preserve* mode.

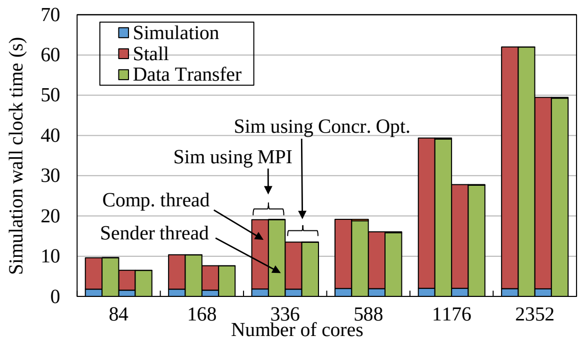
4.3.2 Effect of the Concurrent Message and File Transfer Optimization

The second experiment evaluates the effect of using the concurrent message and file data transfer optimization method in the Zipper runtime system. [2] Since the motivation to design the concurrent method is to alleviate the simulation stall issue and achieve faster data transfer time, We first compare the simulation wallclock time when using the message-only method

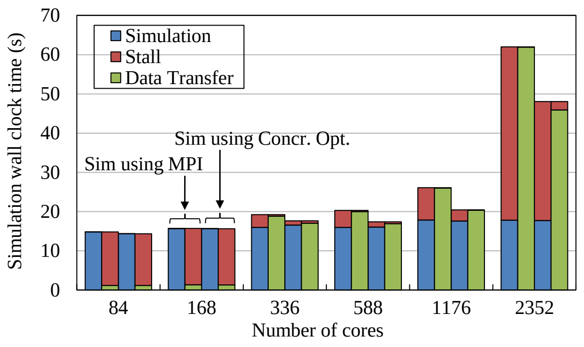
[2]↑In the later description, we use the term "concurrent method" for short.

and concurrent method, and then investigate the speedup reason behind it. We perform the weak scale Zipper workflow experiments in the *No Preserve* mode on *Bridges*, and uses the three synthetic applications in Tab.4.1. The reasons are as follows: 1) The reason to test only in the *No Preserve* mode is that if testing with the *Preserve* mode, the workflow with the message-passing-only method uses the N analysis process to store data, while the concurrent method will utilize both M simulation processes and the N analysis process to store data, although the concurrent method is much faster than the message-passing-only method, this is not a fair comparison. But the *No Preserve* mode only considers the data transfer and doesn't require storing data at all, thus it doesn't have an unfair issue. 2) Because the three applications with different time complexities can generate data blocks from fast to slow, we can see when the Zipper workflow can benefit from the concurrent method, and when it becomes the pure message-passing-only method. Each simulation process uses two physical cores and generates total 4GB data, thus 14 processes are located on a compute node. Same configuration for analysis processes. The largest scale is the workflow of 784 simulation processes transferring data to 392 analysis processes, which uses 2352 cores on 84 compute nodes and generates 3,136 GB data.

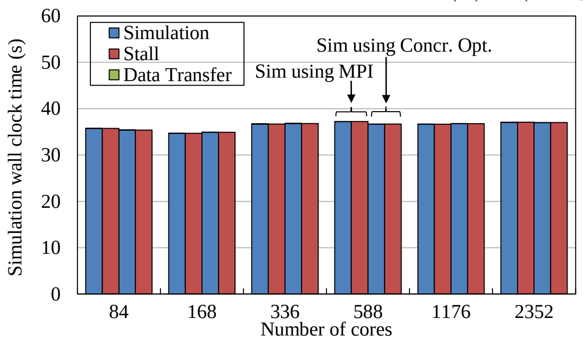
The applications' source code is instrumented by timers to measure the time spent on the two parallel threads of each simulation process: the computation thread and the sender thread. Since the computation thread will be either computing simulations or stalled due to a full producer buffer, its measured time breakdown is shown as a column with stacked *simulation* (blue) and *stall* (red) in Fig. 4.11). Similarly, the sender thread will be either sending messages or waiting for new data, and its measured time is shown as a column with stacked *data transfer* (green) and *stall* (red). As seen in Fig. 4.11, the weak scaling experiments increase the number of CPU cores from 84 to 2,352. For a specific number of cores, we compare the workflow that uses the message-passing-only method to the workflow that uses the concurrent message&file transfer optimization. Given n cores, there is a group of four columns in the figure. The left two columns show the workflow performance of the message-passing-only implementation, and the right two columns show the workflow performance of the concurrent transfer optimization.



(a) $O(n)$ application.



(b) $O(n \log n)$ application.



(c) $O(n^{3/2})$ application.

Figure 4.11. Effect of the concurrent data transfer optimization using different numbers of cores on three synthetic applications.

In Fig. 4.11.a for the $O(n)$ application, from 84 to 2352 cores, the simulation application's wallclock time has been reduced by 32.4%, 26.3%, 29.2%, 16.1%, 29.4%, and 20.2% when using the concurrent method, respectively. This improvement is mainly due to the reduced stall time. For this $O(n)$ application, the measured average of each process's computation time is 1.7s, thus the data generation rate from each compute node is $4\,GB/process \times 14\,processes/node \div 1.7s = 56GB/s$, while the point-to-point network bandwidth for each port on the compute node is 10.2GB/s. As a result, the sender thread cannot move data out in time, therefore the producer buffer becomes full and the simulation thread is blocked. In this case, the work-stealing writer thread detects that the threshold is reached and starts to steal blocks (stolen 47% $\sim$ 62.4% of total blocks) in the above cases.

In Fig. 4.11.b for the $O(n \log n)$ application, the concurrent transfer optimization has reduced the simulation stall time and data transfer time by 8.1%, 14.2%, 21.7%, and 22.5%, from 336 to 2352 cores, respectively. The work-stealing doesn't improve the two smaller-scale cases of using 84 and 168 cores, because the producer buffer is mostly empty and thus there are no blocks to steal during the execution. But later using more than 336 cores, more nodes, more switches, and longer routing distance are involved, thus network switches result in longer network latency, so that the data transfer time starts to dominates the workflow.

Fig. 4.11.c shows the time for the computation-intensive $O(n^{3/2})$ application. Since this application has the slowest data generation rate and longer computation time than the network data transfer, the producer buffer is almost always empty such that the work-stealing in the writer thread is never activated. In this case, the concurrent transfer optimization falls back to the message-passing method.

Based on the performance results in Fig. 4.11, we can find that the concurrent optimization method is always as good or better than the message-passing-only method. The reason is that the concurrent optimization deploys an adaptive stealing-based approach such that it lends a hand only if there exist appropriate opportunities to steal. If there is no stealing opportunity, its performance will be the same as the original performance.

Why the concurrent optimization can improve performance?

The HPC system of *Bridges* uses the Intel Omni Path Architecture (OPA) network, where each compute node is connected to a leaf edge switch (42 ports, max bandwidth 12.5 GB/s/port). Then all leaf switches are connected through a set of core edge switches [101]. [3] At first glance, it seems impossible to gain any benefit by using the concurrent method because there is only one link from a compute node to one port of a leaf switch.

To dig into the reason, we use the PAPI network component [102] and OPA network analysis tools to measure network related performance events. Specifically, the performance counters of `XmitData`, `XmitPkts`, `RcvData`, `RcvPkts`, and `XmitWait` are measured when comparing the message-passing only method and the concurrent method. Since users do

[3]↑There is a single OPA card per compute node. A parallel job sharing the nodes on a single OPA switch enjoys the full OPA bandwidth. Beyond a switch, the effective bandwidth decreases by an empirical factor of 7.

not have privileges to access the counters on switches, we can only collect the performance counters on the network adapter on each compute node.

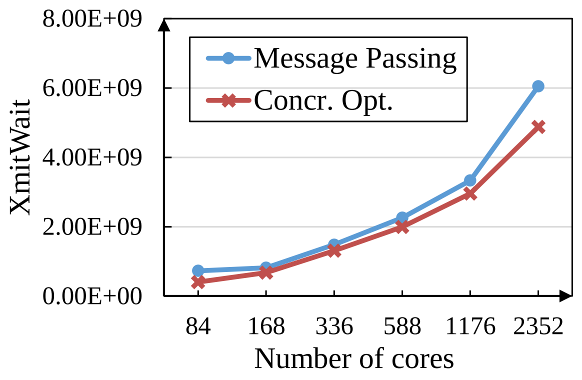
Among all the network events, we find that the `XmitWait` counter shows the biggest difference between using the message-passing only method and using the concurrent method. The `XmitWait` counter is used to count the number of events (in FLIT[4]) when any virtual lane had data but was unable to transmit [103], for reasons such as no transmission credits available, or the link was busy sending non-data packets. Therefore, this counter is often used to measure the extent of network congestion [95].

We use the Linux command "`opapmaquery -o getportstatus`" to collect the values of the counters on each compute node periodically. Whenever 10% of the total number of blocks are generated, the sender thread will query the counters and calculate the difference between the current query and the previous query. This measured difference indicates how many messages are attempted to send out but rejected due to the network congestion control mechanism. The larger the `XmitWait` value is, the more times the network adapter is unable to transmit, and the more congested the network is.

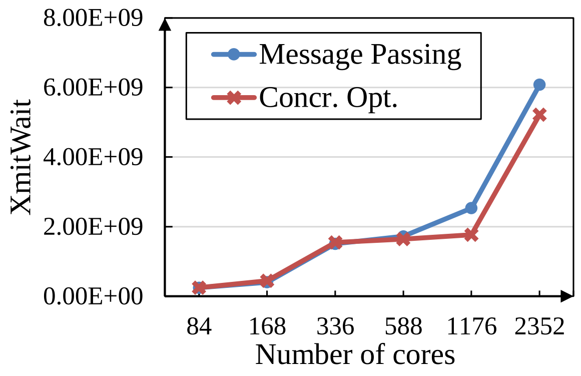
We use the measured `XmitWait` counter to show the relationship between the degree of network congestion and data transfer time. As shown in Fig. 4.12.a dedicated for the $O(n)$ application, We observe that the counter of `XmitWait` using message-passing-only is larger than that using the concurrent method by 80%, 21%, 13%, 13%, 13%, and 24% from 84 to 2,352 cores, respectively. This suggests that when we use the message-passing-only method, more messages are not able to transmit than when we use the concurrent method. Since `XmitWait` is an indication of the degree of network congestion, we can say that the concurrent method has less serious congestion than the message-passing-only method. Also due to the reduced network congestion, the concurrent method can send data more quickly and has a shorter transfer time, which is confirmed by Fig. 4.11.a correspondingly.

Measurement of the `XmitWait` counter for the $O(n \log n)$ application is shown in Fig. 4.12.b. On 84 and 168 cores, the `XmitWait` counter is less than 0.5×10^9, which implies light net-

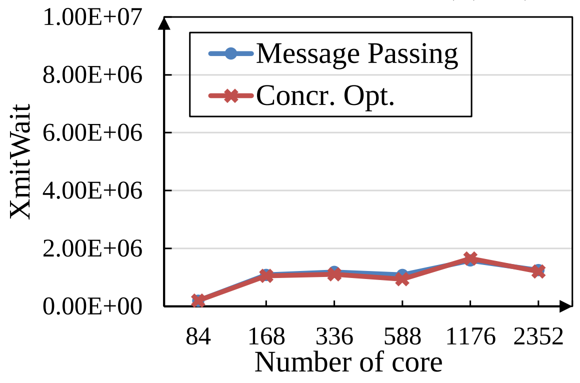
[4]↑In Omni Path, the Link Transfer (LT) layer segments the end-to-end Fabric Packets (FPs) into 64 bit Flow Control Digits (FLITs), and groups 16 FLITs into a Link Transfer Packet (LTP) to reliably transport FP FLITs and control information on the link[101].



(a) $O(n)$ application.



(b) $O(n \log n)$ application.



(c) $O(n^{3/2})$ application.

Figure 4.12. Network Congestion of the concurrent data transfer optimization using different numbers of cores on three synthetic applications. *XmitWait* counts the number of occurrences when any virtual lane had data but was unable to transmit.

work congestion and all data can be sent out rapidly without waiting. The other sign of light network congestion is that the producer's message buffer is almost empty all the time. Therefore, the writer thread does not steal any data blocks so that the concurrent method becomes the message-passing-only method. Hence, Fig. 4.11.b shows equal data transfer time on 84 and 168 cores. However, for larger scales starting from 336 cores, the `XmitWait` counter rises up significantly (i.e., 3X $\sim$ 12X larger than that on 168 cores). This suggests a higher degree of congestion, and the producer's buffer becomes full and the writer thread starts stealing and eases the congestion again. The reduced congestion also justifies the shorter data transfer time by using the concurrent method from 588 to 2352 cores (see Fig. 4.11.b).

In Fig. 4.12.c, for the slowest $O(n^{3/2})$ producer application, the value of the `XmitWait` counter is around 10^6 (i.e., three orders of magnitude less than the previous two applications). The congestion degree is constantly low for all different numbers of cores, and the producer's buffer is almost empty such that the concurrent method becomes the message-passing-only method. Therefore, the corresponding Fig. 4.11.c shows that the message-passing-only and concurrent methods have equal data transfer time.

4.3.3 Scalability Performance

The last experiment is to evaluate the scalability performance of the Zipper system. We perform experiments with two real-world applications of CFD and LAMMPS on the larger *Stampede2* system, which allows up to 17,408 cores per job, while *Bridges* only allows 4,704 cores per job. [5]

The CFD application uses the Lattice Boltzmann method to compute 3D simulations of viscous incompressible fluid sliding down 3D hydrophobic microchannel walls [104], [105]. Its corresponding analysis component computes the n-th moment of the velocity distribution: $E(u(x,t)^n)$, where $u(x,t)$ is the velocity at a spatial point x at time t. The statistics can help scientists understand the properties of the turbulent flow with high *Reynolds* numbers. When all n-th moments are available, the probability density function of $u(x,t)$ can be evaluated to give the complete information of the velocity fluctuation of a turbulent flow [106], [107].

The LAMMPS application simulates clusters of Lennard-Jones atoms. We use the application to study the melting process of materials from a low-energy solid structure at low temperatures to a set of higher energy liquid structures at high temperatures. The Lennard-Jones model is a mathematical model for approximating interactions between neutral atoms or molecules. The counterpart data analysis application will compute MSD (mean squared displacement). MSD calculates the deviation time between the position of a particle and a reference position, in order to analyze the spatial extent of random motions.

[5]↑Stampede2 allows up to 256 $nodes/job \times 68\, cores/node = 17408 coresjob$ Bridges allows up to 168 $nodes/job * 28\, cores/node = 4704 cores/job$

Remark: The reason we choose the CFD and LAMMPS workflows to do experiments is that *simulation-time data analyses* are common in scientific and engineering domains, and achieving high performance is crucial to most domain scientists [108], [109]. The data analysis application in our workflows receives data blocks and analyzes them accordingly, followed by asynchronous reduction operations.

The CFD application

In the CFD workflow experiments, each simulation process is allocated with a fluid subgrid of dimension $64 \times 64 \times 256$. When doubling the numbers of CPU cores, the total input size also doubles (i.e., weak scaling). Among the total number of cores, two-thirds of the cores are used for CFD simulations and one-third is used for the n-th moment analysis.

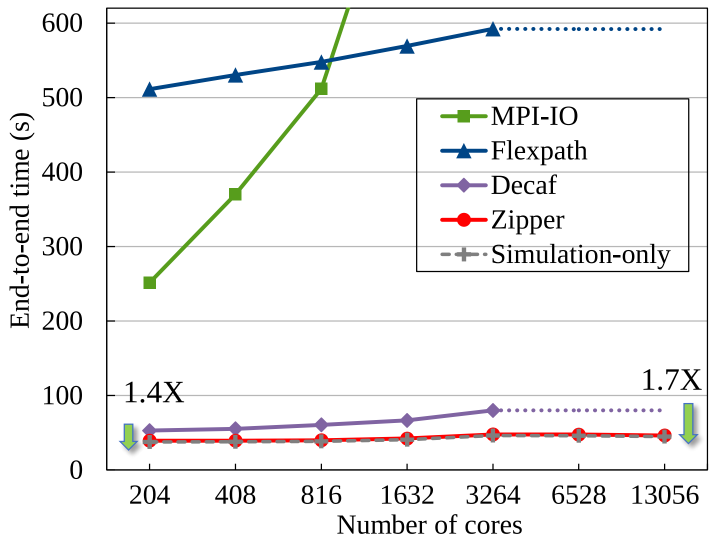


Figure 4.13. Scalability performance of the CFD workflows using MPI-IO, Flexpath, Decaf, and Zipper, respectively.

Fig. 4.13 shows the end-to-end time using MPI-IO, Flexpath, Decaf, and Zipper, as well as the simulation-only time in the *No Preserve* mode. On Stampede2, when the number of compute nodes is larger than 8, DataSpaces and DIMES aborted with "rpc_bind_addr" error in the DataSpaces/DIMES initialization function. The error is related to "an issue related to OPA and KNL processors", and has been confirmed by the DataSpaces team.

Hence, we could not test DataSpaces/DIMES on Stampede2. Nevertheless, the fastest library is Decaf, which we choose to compare with Zipper.

Simulation-only time is the time spent only by the simulation program's computational kernels (excluding any I/O, idle time, and data staging related cost). It works as a lower bound of the workflow end-to-end time. Fig. 4.13 shows that using MPI-IO is not scalable: as the number of cores increases from 3264 to 13,056, larger MPI-IO experiments take too long to finish. On the other hand, Flexpath and Decaf scale well from 204 cores to 3,264 cores. However, Flexpath and Decaf crashed with software faults on 6,528 and 13,056 cores. In particular, Decaf has segmentation faults due to integer overflows. We have reported the issue to Decaf developers and they have confirmed the error. Flexpath is terminated by segmentation faults when the number of cores reaches 6,528. We have also reported the problem to Flexpath developers.

In order to show complete experimental results for Flexpath and Decaf, let's assume that both methods have perfect scalability on 6,528 and 13,056 cores, and show their ideal end-to-end time (denoted by dotted lines). As shown in Fig. 4.13, Zipper's end-to-end time is almost equal to the simulation-only time, and is 11.5X faster than Flexpath, and 1.7X faster than Decaf.

One might wonder why Flexpath is slow. We conducted a set of investigations to find out the reason. Based on my experiments, Flexpath's data transfer time becomes significantly slower as we increase the number of processes per node (each process uses Flexpath to transport data). We find that Flexpath does not have optimized support for multiple processes per node. Flexpath utilizes a socket interface and all communications (even within the same node) have to go through the socket interface. However, the communication between processes on one node can use shared memory to achieve higher performance (e.g., MPI uses this optimization). In order to show the *ideal* performance of Flexpath, We attempt one-process-per-node to rerun the 204-core experiment (although wasting many cores on each node). In the new experiment, Flexpath using 102 processes on 102 nodes (i.e., 6,936 cores) only takes 46 seconds, but is still slower than Zipper using 102 processes on 3 nodes (i.e., 204 cores) by 16.8%. Besides using a smaller number of processes per node, another Flexpath optimization is to use a "Master" process on each node to aggregate data from all processes of the node to reduce the communication cost. However, this method requires significant code modifications.

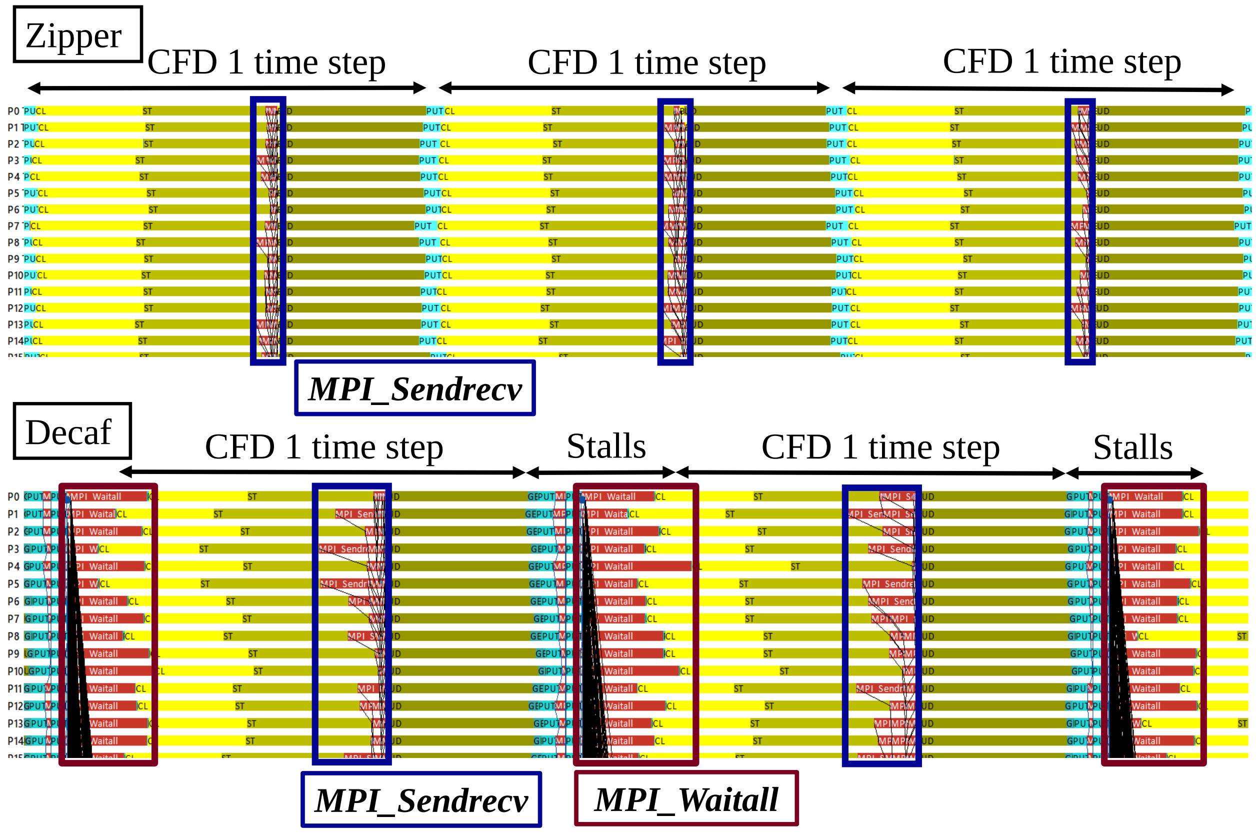


Figure 4.14. Trace comparison between Zipper and Decaf for the CFD application on 204 cores. This figure shows a snapshot of 1.3 seconds when using 204 cores, which is taken from the experiment shown in Fig.4.13.

In order to illustrate why Zipper is faster than Decaf, Fig. 4.14 shows that Zipper and Decaf's traces within a time interval of 1.3 seconds on 204 cores. To take the snapshot, we zoom in the entire trace, and then cut out a trace segment of 1.3 seconds. Note that showing the entire trace all at once will make the figure too dense to view any details. During the same interval, Zipper is able to run three simulation steps, while Decaf is able to run two steps with a significant amount of stall time. This speedup of 1.4X is almost the same as the speedup shown in Fig. 4.13 on 204 cores.

The reason for the performance inefficiency is as follows (also reported in Chap. 3): 1) Decaf has significant simulation stall time caused by MPI_Waitall, and 2) the simulations application's MPI_Sendrecv time becomes longer due to Decaf's interference. Since Zipper uses fine-grain data blocks and asynchronous pipelining data transfers, both the network traffic interference and the collective MPI cost have been reduced.

The LAMMPS application

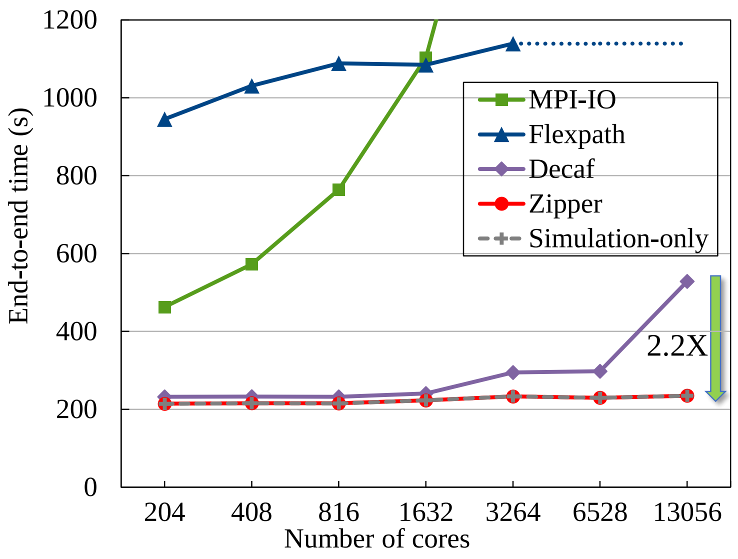


Figure 4.15. Scalability performance of the LAMMPS workflows using MPI-IO, Flexpath, Decaf, and Zipper, respectively.

Fig. 4.15 shows the experimental results for the LAMMPS workflow application. Again, we perform weak scaling experiments. Fig. 4.15 shows that Flexpath scales well from 204 to 3,264 cores but is 7.1X slower than Zipper. Because the data size in LAMMPS does not reach the integer limit, We am able to execute Decaf on 6,528 and 13,056 cores successfully without integer overflows. From the figure, we can see that Decaf scales well from 204 to 1,632 cores, but becomes 128% slower from 1,632 to 6,528 cores. Eventually, its end-to-end time increases by 177% from 6,528 to 13,056 cores.

To study why Decaf is 2.2X slower than Zipper in the largest experiment, We specifically collect two very large traces for Decaf and Zipper using 13,056 cores, respectively. Visualizing the large-scale trace itself requires us to use a dedicated compute node from the *Stampede2* HPC system for 2 hours.

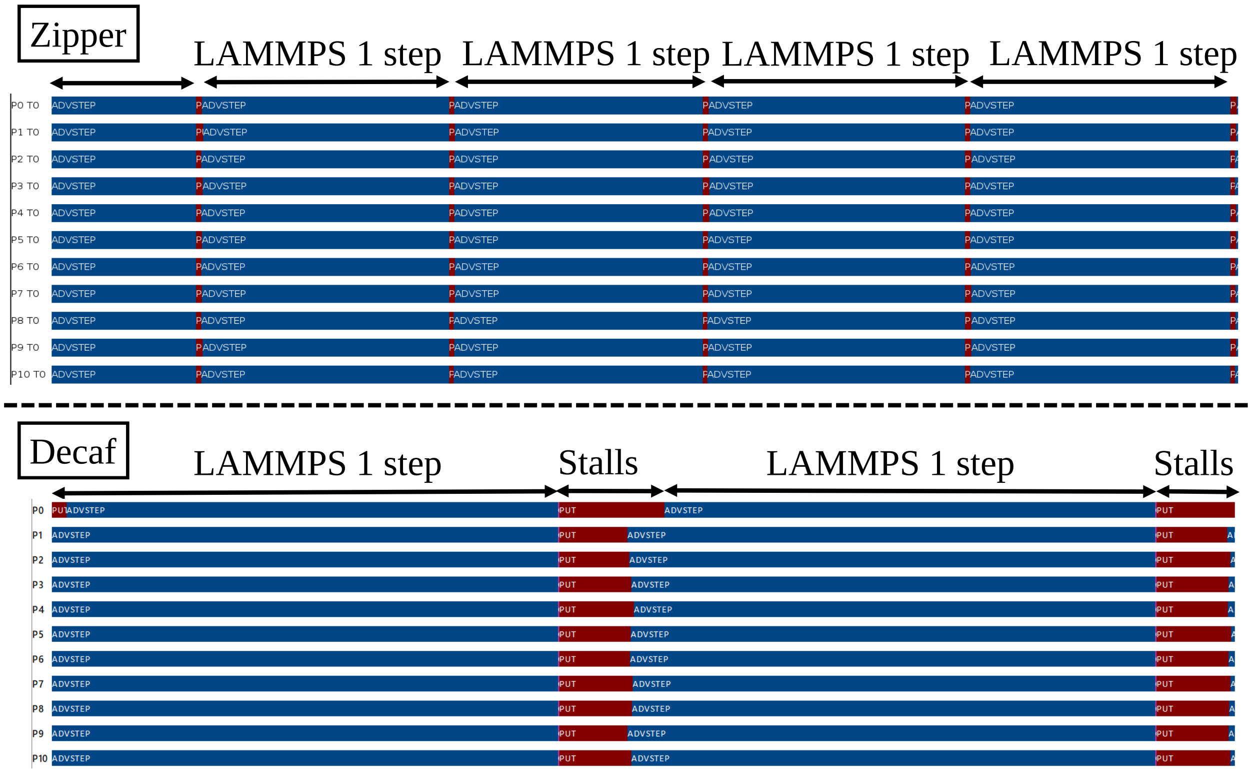


Figure 4.16. Trace comparison between Zipper and Decaf for the LAMMPS application on 13,056 cores. This figure shows a snapshot of 9.1 seconds when using 13,056 cores, which is taken from the experiment shown in Fig.4.15.

Fig. 4.16 shows a snapshot of the two traces in an interval of 9.1 seconds. During the same time interval, LAMMPS using Zipper runs around 4.4 time steps. On the other hand, LAMMPS using Decaf runs around 2 time steps. Notice that the Decaf trace has a significant stall time at the end of each step. Also, the LAMMPS simulation time using Decaf becomes much longer than that using Zipper. In this LAMMPS workflow experiment, each LAMMPS process generates approximately 20MB of data in each time step. While Decaf directly sends a message of 20MB to destination processes, Zipper divides the contiguous 20MB data into many small blocks of size 1.2MB. Such an asynchronous fine-grain-block data transfer method has managed to keep network traffic more balanced with lesser interference to the LAMMPS simulation processes.

5. 2D PARALLEL MEMORY-AWARE LBM ON MANYCORE SYSTEMS

A portion of this chapter was previously published in *SBAC-PAD'18, 30th International Symposium on Computer Architecture and High Performance Computing* [58] doi.org/10.1109/CAHPC.2018.8645909.

Chap.4 has designed the Zipper in-situ system, which can minimize the end-to-end time-to-solution for in-situ workflows to be its longest stage, either simulation, data transfer, or analysis. In particular, for simulation-bound Zipper workflows, if the simulation can be accelerated by a factor, the time-to-solution of the whole workflow can potentially speed up by the same factor. This inspires me to focus on the source of simulation-bound workflows, i.e., simulation applications.

Sect.2.2 has introduced the computational fluid dynamics (CFD) simulations, which have revolutionized the design process in various scientific, engineering, industrial, and medical fields. The current Reynolds averaged Navier-Stokes (RANS) methods can solve steady viscous transonic and supersonic flows, but are not able to reliably predict turbulent separated flows [21]. Lattice Boltzmann method (LBM) is a young and evolving approach to solving these problems in the CFD community [25]. It originates from a mesoscale description of the fluid, is based on the Boltzmann equation, and can integrate physical terms from molecule interaction. Besides, many collision models for LBM have been developed to improve its stability to the second order of numerical accuracy when simulating high Reynolds number flows [25].

However, it is challenging to achieve high performance for LBM, since LBM has large data storage costs and is highly memory-bound on current architectures [26]. In general, this chapter discusses how to merge multiple collision-streaming cycles (or time steps) in 2D. Sect.5.1 introduces LBM and its pros and cons. Then I briefly outline the Original LBM algorithm in Sect.5.2.1 and the Fuse LBM algorithm in Sect. 5.2.2. Sect.5.3 uses the "Roofline" model to pinpoint the bottleneck of the two algorithms and introduces an improvement algorithm, Fuse tile LBM. However, their parallel performances are hugely bounded by memory accesses in current multi-core CPU architectures. To reduce the memory bottleneck, the two-step and k-step memory-aware LBM algorithms are designed. Sect.5.5 introduces how to merge "two" time steps sequentially and in parallel, how to handle the boundary condition, and how to handle thread safety efficiently. Sect.5.6 merges "three" time steps sequentially and in parallel with optimization on handling boundary conditions. A thorough performance evaluation and analysis of seven algorithms in Tab.5.1 are given in Sect. 5.8.

Table 5.1. Seven LBM algorithms discussed in this chapter. Each algorithm has its sequential version and parallel version.

Algorithms	**Description**
Original LBM	Standard LBM implementations with two copies of distribution and two sweeps
Fuse LBM	Use loop fusion
Fuse tile LBM	Use loop fusion and loop tiling
2-step LBM	Use loop fusion and merge two steps
2-step tile LBM	Use loop fusion, loop tiling, and merge two steps
3-step LBM	Use loop fusion and merge three steps
3-step tile LBM	Use loop fusion, loop tiling, and merge three steps

5.1 Background of Lattice Boltzmann Method

Based on the Boltzmann equation, Lattice Boltzmann method (LBM) originates from a mesoscale description of the fluid. It views the gas or fluid as clusters of small particles moving with random motions on a fixed Cartesian velocity lattice. Then these clusters exchange momentum and energy through particle streaming and billiard-like particle collision.

5.1.1 The Lattice Boltzmann Equation

The central variable of LBM is the particle distribution function $f(\vec{x}, \vec{\xi}, t)$, which is a generalization of density ρ, at the position $\vec{x}$, and particle velocity $\vec{\xi}$, and time frame t. The Boltzmann equation describes the evolution of $f(\vec{x}, \vec{\xi}, t)$ in time:

$$\frac{\partial f}{\partial t} + \xi_\beta \frac{\partial f}{\partial x_\beta} + \frac{F_\beta}{\rho} \frac{\partial f}{\partial \xi_\beta} = \Omega(f) \tag{5.1}$$

Here, we use one generic index $\beta = \{x, y, z\}$ to represent Boltzmann equation on three directions of physical space. The first two terms in Equation (5.1) shows that the $f(\vec{x}, \vec{\xi}, t)$ is advected with the velocity $\vec{\xi}$, which is affected by the forces $\frac{F_\beta}{\rho}\frac{\partial f}{\partial \xi_\beta}$. The collision operator $\Omega(f)$ works as the source term to locally redistribute f on the right of Equation (5.1).

While there exist various collision operators $\Omega(f)$ available, the simplest one that can be used for Navier-Stokes simulations is the Bhatnagar-Gross-Krook (BGK) operator [110]:

$$\Omega(f) = -\frac{1}{\tau}\left(f - f^{\mathrm{eq}}\right) \tag{5.2}$$

The relaxation time constant τ determines the speed towards the equilibrium distribution f^{eq} and the transport coefficients such as viscosity and heat diffusivity.

By discretizing Equation (5.1) in time, physical space, and velocity space, we derive the lattice Boltzmann equation with BGK operator as follows:

$$f_{\mathrm{i}}(\vec{x} + \vec{\xi}_{\mathrm{i}}\Delta t, t + \Delta t) = f_{\mathrm{i}}(\vec{x}, t) - \frac{\Delta t}{\tau}[f_{\mathrm{i}}(\vec{x}, t) - f_{\mathrm{i}}^{\mathrm{eq}}(\rho(\vec{x}, t), \vec{u}(\vec{x}, t))] \tag{5.3}$$

where f_{i} is the particle density distribution corresponding to the discrete velocity direction $\vec{\xi}_{\mathrm{i}}$, and $\vec{x}$ and Δt are the discrete location and time, respectively. Besides, the macroscopic density and the momentum density $\rho\vec{u}$ can be derived by weighted summation as follows [72]:

$$\rho(\boldsymbol{x}, t) = \sum_{\mathrm{i}} f_{\mathrm{i}}(\boldsymbol{x}, t), \quad \rho\boldsymbol{u}(\boldsymbol{x}, t) = \sum_{\mathrm{i}} \boldsymbol{\xi}_{\mathrm{i}} f_{\mathrm{i}}(\boldsymbol{x}, t) \tag{5.4}$$

Fig.5.1 presents the D2Q9 velocity vectors, which are widely used in 2D LBM simulations. Each fluid cluster has eight neighbors, and it may move along 9 different directions including staying at the center.

5.1.2 LBM Pros & Cons

Here is a summary of LBM's pros and cons, and interested readers should refer to [72] and [111].

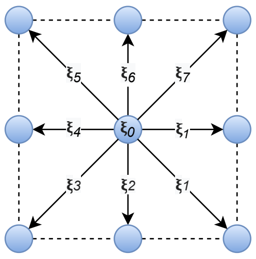


Figure 5.1. D2Q9 velocity sets for each lattice cell.

In terms of **simplicity** and **efficiency**, LBM gains simplicity and scalability by allowing artificial compressibility for solving the incompressible Navier-Stokes equation. Secondly, the Poisson equation is not included. Thirdly, the most computations in the LBM are local in each cells, thereby easy to design its parallel algorithms. However, LBM is memory-intensive and time-dependent.

In terms of **geometry**, LBM can handle complex geometries (e.g., porous media [111]). LBM also particularly works well with moving boundaries that conserve mass, such as soft matter simulations [112].

In terms of **thermal effects**, LBM can mesoscopically incorporate Thermal fluctuations, which are averaged out on the macroscale [113], [114].

In terms of **multiphase** and **multicomponent** flows, there exists a broad range of LBM-based algorithms to simulate these flows in complex geometries [111].

At last, in terms of **sound** and **compressibility**, LBM can handle sound and flow interaction simulations [115]. Besides, some cutting-edge research on LES-based LBMs and Entropic LBMs (ELBMs) have shed light on simulating thermal and compressible flows in transonic and supersonic regimes [22], [25]–[29].

5.2 Baseline 2D LBM Algorithms

This section introduces the baseline 2D LBM algorithm, i.e., Original LBM, and its two improved versions, Fuse LBM and Fuse tile LBM. Next, I use roofline analysis to investigate the inefficiencies that reside in the three algorithms.

5.2.1 2D Original LBM

The Original LBM follows Equation (5.3). Each cell owns two buffers (*buf1* and *buf2*) to store the particle distributions at the time step $f(t)$ and $f(t+1)$, respectively. Alg. 2 presents the Original LBM running N time steps. Generally, in each time step t, it sweeps over the entire simulation lattice twice. During the *collision* sweep, each cell uses a specific collision model (e.g., single relax term, multiple relax terms, boundary conditions, etc.) to calculate the "intermediate" particle distribution at the time step t. During the *streaming* sweep, each cell copies the "intermediate" data to its own and 8 neighbors in the directions of D2Q9 velocity sets. After a cell collects all the dependencies, it is ready to compute the next time step $t+1$. The streaming kernel can be viewed as a communication step and exchange data $buf1$ and $buf2$ among its 8 neighbors.

Algorithm 2 Original LBM

```
 1: //N: total time steps, $lx$: domain width, $ly$: domain length.
 2: **for** iT = 0; iT < N; ++iT **do**
 3:   // *Collision kernel*
 4:   **for** iX = 1; iX $\leq lx$; ++iX **do**
 5:     **for** iY = 1; iY $\leq ly$; ++iY **do**
 6:       compute (iX, iY) collision using *buf1*
 7:   // *Streaming kernel*
 8:   **for** iX = 1; iX $\leq lx$; ++iX **do**
 9:     **for** iY = 1; iY $\leq ly$; ++iY **do**
10:       // stream (iX, iY) *buf1* to its own and 8 neighbors' *buf2*
11:       **for** iPop = 0; iPop < 9; ++iPop **do**
12:         nextiX = iX + c[iPop][0];
13:         nextiY = iY + c[iPop][1];
14:         *buf2*[nextX][nextY].fPop[iPop] = *buf1*[iX][iY].fPop[iPop];
15:   Swap $buf1$ and $buf2$
```

In the first *collision* kernel (*lines 3~6*), a cell uses the particle distribution data in *buf1* to locally compute the collision, macroscopic attributes, and equilibrium state. Then the "intermediate" state is stored back into the *buf1*. Once the *collision* kernel sweeps the domain, the *streaming* kernel (*lines 7~10*) starts. The "intermediate" data in *buf1* of a cell now streams (copies) to *buf2* of its own and its neighbors. Once the *streaming* sweep completes, swapping $buf1$ and $buf2$ in line 11 is necessary before the next time step

$t + 1$. Since every cell's $buf1$ and $buf2$ are referenced by two pointer variables [1], the swap operation is $O(1)$. For simplicity, we will use "stream (iX, iY)'s *buf1* to its own and 8 neighbors' *buf2*" to replace the fourth loop from line 11 to 14 in Alg. 2. The referential Original LBM is an ANSI C implementation from Palabos [116] which can be accessed at http://wiki.palabos.org/numerics:codes.

5.2.2 2D Fuse LBM

Algorithm 3 Fuse LBM

```
1: // Use loop fusion to combine collision and streaming in one sweep:
2: for iT = 0; iT < N; ++iT do
3:   for iX = 1; iX ≤ $lx$; ++iX do
4:     for iY = 1; iY ≤ $ly$; ++iY do
5:       compute (iX, iY) collision using buf1
6:       stream (iX, iY)'s buf1 to its own and 8 neighbors' buf2
7:   Swap $buf1$ and $buf2$
```

An improvement on the Original LBM is to use *loop fusion* (combining the collision and streaming cycle) to increase data reuse [44], [117]. Instead of sweeping through the whole lattice twice per time step, after calculating the distribution function values in *buf1* in the collision operation, Fuse LBM immediately streams the "intermediate" data to the neighbors' *buf2* (*lines 5∼6*). Besides, the Fuse LBM works as the "baseline" to compare with later LBM algorithms, which adds more features, such as spatial tiling/blocking, temporal locality, optimized traversing order, and so on.

5.2.3 2D Fuse Tile LBM

To improve the Fuse LBM's data reuse, we can add the "loop tiling" feature as shown in Alg.4, which traverses the 2D lattice tiles by tiles instead of lines by lines. Each tile is

[1]↑We allocate two pieces of continuous memory chunk (`memChunk1` and `memChunk2`) for the whole lattice. Thus, a cell at coordinate (iX, iY) has two copies of data: one at `memChunk1[iX][iY]`, and the other at `memChunk2[iX][iY]`. Before simulation starts, we assign two pointer variables $buf1$ and $buf2$ to `memChunk1` and `memChunk2`. Thus the two copies of each cell can be accessed by `buf1[iX][iY]` and `buf2[iX][iY]`, respectively. Therefore,, swapping $buf1$ and $buf2$ after every time step in line 15 of Alg. 2 is just swapping the values of two pointer variables.

assumed to be small enough to fit into the last level cache to maximize data reuse. The "MIN" (minimum) statements (*lines∼5*) ensure that the chosen "tile" does not access data outside the domain boundary. Thus, users are allowed to select any non-negative "tile" parameter to produce the best performance on their architectures.

Algorithm 4 Fuse Tile LBM

```
1: for iT = 0; iT < N; ++iT do
2:   for outerX = 1; outerX ≤ $lx$; outerX += tile do
3:     for outerY = 1; outerY ≤ $ly$ ; outerY += tile do
         // Use loop tiling to explore spatial locality:
4:       for innerX = outerX; innerX ≤ MIN(outerX + tile - 1, $lx$); ++innerX do
5:         for innerY = outerY; innerY ≤ MIN(outerY + tile - 1, $ly$); ++innerY do
6:           compute (innerX, innerY) collision using buf1
7:           stream (innerX, innerY)'s buf1 to its own and 8 neighbors' buf2
8:   Swap $buf1$ and $buf2$
```

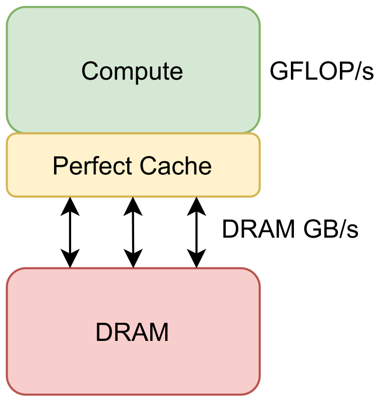
5.3 Roofline Analysis of 2D Baseline LBM Algorithms

To investigate the performance of 2D baseline LBM algorithms, we use the Roofline model [118] to provide insight into questions like: What the performance bottlenecks are, and which kernels are worth addressing firstly? Why the bottlenecks exist? How much potential performance can be improved?

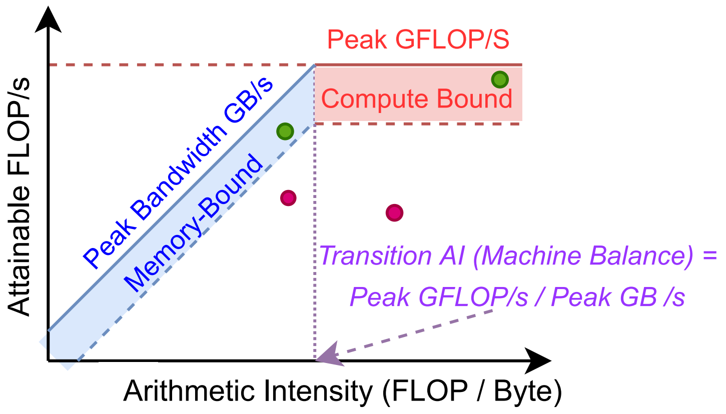
5.3.1 Brief Introduction of Roofline Model

Fig.5.2a presents the intuition of the standard Roofline model, which uses the bytes that access the main memory after they have been filtered by the cache hierarchy. The time-to-solution is limited by either time for floating-point computation (#FP ops) or data movement (#Bytes moved to/from DRAM).

$$\text{Time} = \max \begin{cases} \text{\#FP ops / Peak GFLOP/s} \\ \text{\# Bytes / Peak GB/s} \end{cases} \tag{5.5}$$



(a) Intuition.


Attainable FLOP/s
Peak GFLOP/S
Compute Bound
Peak Bandwidth GB/s
Memory-Bound
Transition AI (Machine Balance) =
Peak GFLOP/s / Peak GB /s
Arithmetic Intensity (FLOP / Byte)

(b) Roofline chart.

Figure 5.2. Standard DRAM Roofline model.

Next, we divide #FP ops on both sides of Eq.(5.5).

$$\frac{\text{Time}}{\#\text{ FP ops}} = \max \begin{cases} 1/\text{ Peak GFLOP/s} \\ \#\text{Bytes / \#FP ops / Peak GB/s} \end{cases} \tag{5.6}$$

Then, we take the reciprocal of Eq.(5.6) and get the following equation.

$$\frac{\#FP\text{ ops}}{\text{Time}} = \min \begin{cases} \text{Peak GFLOP/s} \\ (\text{\#FP ops / \#Bytes}) * \text{Peak GB/s} \end{cases} \tag{5.7}$$

Here, # FP ops / # Bytes is called *Arithmetic Intensity* (AI), which means # operations per byte of DRAM traffic on a particular computer. This metric directly ties to the algorithm and implementation of an application.

$$\text{GFLOP/s} \;= \min \begin{cases} \text{Peak GFLOP/s} \\ \text{AI} * \text{Peak GB/s} \end{cases} \tag{5.8}$$

Because logarithm function turns a multiplicative factor into a shift up (Y-axis intercept), and an exponential into a multiplicative factor (slope), we take the same base of the logarithm function on both sides of Equation (5.8) to turn it into straight lines, and get the most information from the slope and intercept.

$$\log(\text{GFLOP/s}) = \min \begin{cases} \log(\text{Peak GFLOP/s}) \\ \log(\text{AI}) + \log(\text{Peak GB/s}) \end{cases} \tag{5.9}$$

Thus, the standard roofline chart is plotted on the log-log scale in Fig.5.2b. The peak bandwidth and peak GFLOP/s are the two *"rooflines"*. we can see that the slope of $log(AI)$ is 1, which indicates the angle of memory bandwidth roofline is 45 degrees, and the $log(Peak\text{GB}/s)$ is shown as the intercept of the Y-axis. Fig.5.2b divides the locality performance plane into five regions. Out of boundaries of the peak bandwidth and the peak GFLOPS are unattainable performance. The purple transition of peak bandwidth and peak GFLOPS is called *machine balance*. Memory-bound kernels are on the left of machine balance, while compute-bound kernels are on the right, e.g., the two green dots in the blue and red region, respectively. The kernels that are far away from the two ceilings, e.g., the red dot in Fig.5.2b, generally perform poorly and need to be improved.

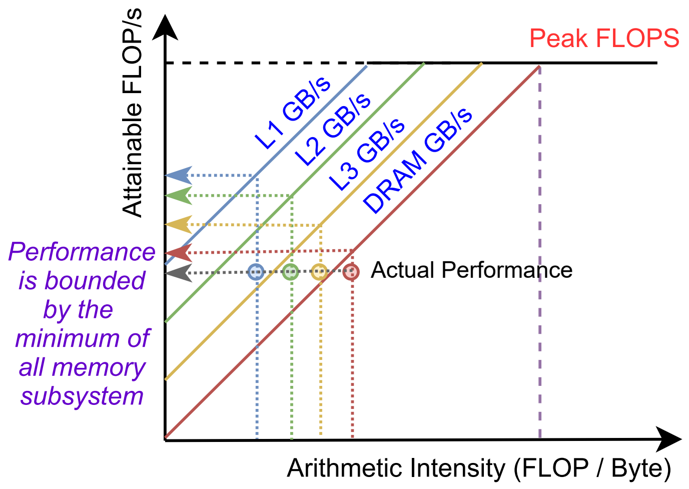

Attainable FLOP/s
Peak FLOPS
L1 GB/s
L2 GB/s
L3 GB/s
DRAM GB/s
Performance is bounded by the minimum of all memory subsystem
Actual Performance
Arithmetic Intensity (FLOP / Byte)

Figure 5.3. Hierarchical Roofline model [118]. An application's achieved performance and AI at each memory level of the machine.

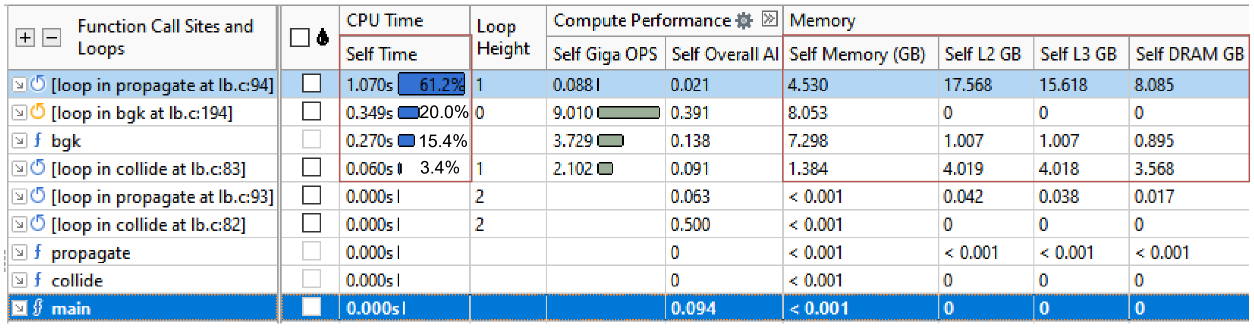
"Hierarchical Roofline" model [118] in Fig.5.3 adds not only computational peak (e.g., integer-based scalar, vector instructions, etc.) but also multi-level memory hierarchy (e.g., the data movement between CPU and L1 cache (CARM, Cache-Aware Roofline Model), L1 and L2 cache, L2 and LLC, and LLC and DRAM). Before collecting data on an application, Intel Advisor runs benchmarks (e.g., STREAM[119], dgemm, etc.) to measure the hardware limitations of the machine, and then plots the ceilings on the chart. We see that the AI of an application can be represented by four dots according to four memory levels. The performance of an application is bounded by the minimum of all memory subsystems. The widely separated space between two dots of AI indicates high reuse in the faster memory subsystem, while the narrow space indicates effectively low reuse. Moreover, the left to right order (CARM, L2, L3, DRAM) might not be respected if an application has inefficient memory accesses. For example, the CARM dot can be on the right side compared to the L2 dot. This can be due to a large amount of L1 misses, therefore the data transfer between L1 and L2 increases. According to $AI = \#FPops/\#Bytes$, we have smaller L2 AI than CARM AI. We will observe this phenomenon in the Original LBM, and their memory access patterns are further investigated by using the Memory Access Pattern Analysis (MAP) tool in Sect.5.8.2. The hierarchical Roofline model can be automatically collected and generated by Intel Advisor [120].

5.3.2 Roofline Analysis of Three 2D LBM algorithms

This section evaluates the three 2D baseline LBM algorithms with the same simulation domain of an 2D square lattice with edge size $L = 1024$ during 60 collision-streaming cycles. Meanwhile, Intel Advisor is used to collect data and generate roofline charts. The experiments are on a *Bridges* Haswell node, and its LLC is 33MB per socket. The memory consumption of the input is 160MB [2], which exceeds the LLC size and will incur frequent DRAM accesses. We use 30 time steps to warm up machines, and use the next 30 time steps for measurement. Thus, the data transfer to sweep the 2D grid is $160MB * 30 = 4.7GB$.

Fig.5.4 sorts the CPU time of each kernel in the Original LBM in descending order. The “loop in propagate” kernel (the first row) with “self overall AI = 0.021” computes the

[2]↑Each lattice cell takes 80B, which consists of 9 double floating numbers for particle distribution and 1 pointer for the action function, e.g., BGK collision or boundary condition computation. Besides, each cell has two copies of distribution, each point occupies $80B * 2 = 160B$. Thus 1024×1024 2D lattice takes up 160 MB.

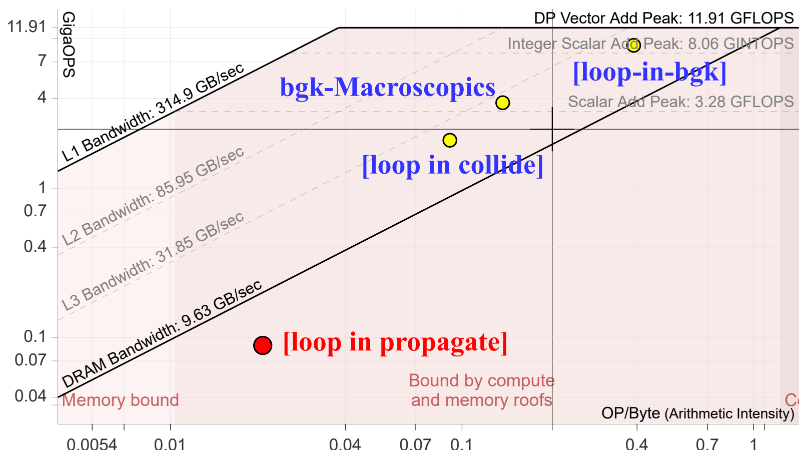


Function Call Sites and Loops		CPU Time	Loop Height	Compute Performance		Memory			
		Self Time		Self Giga OPS	Self Overall AI	Self Memory (GB)	Self L2 GB	Self L3 GB	Self DRAM GB
[loop in propagate at lb.c:94]	☐	1.070s 61.2%	1	0.088	0.021	4.530	17.568	15.618	8.085
[loop in bgk at lb.c:194]	☐	0.349s 20.0%	0	9.010	0.391	8.053	0	0	0
bgk		0.270s 15.4%		3.729	0.138	7.298	1.007	1.007	0.895
[loop in collide at lb.c:83]	☐	0.060s 3.4%	1	2.102	0.091	1.384	4.019	4.018	3.568
[loop in propagate at lb.c:93]	☐	0.000s	2		0.063	< 0.001	0.042	0.038	0.017
[loop in collide at lb.c:82]	☐	0.000s	2		0.500	< 0.001	0	0	0
propagate		0.000s			0	< 0.001	< 0.001	< 0.001	< 0.001
collide		0.000s			0	< 0.001	0	0	0
main	☐	0.000s			0.094	< 0.001	0	0	0

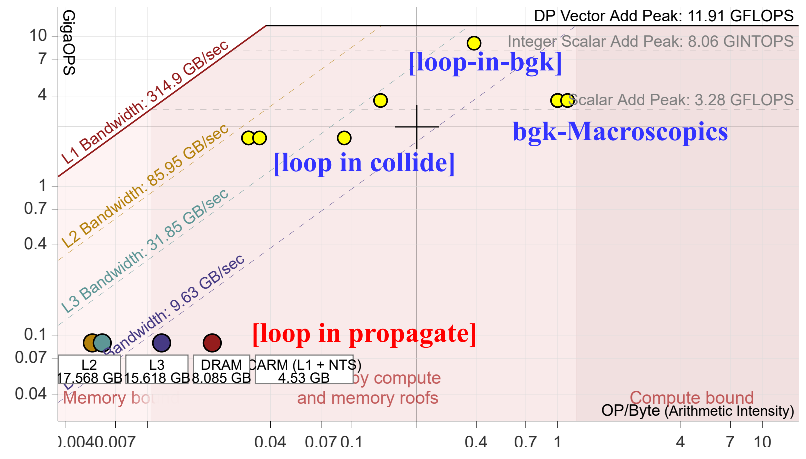
Figure 5.4. Time and data transfer among each memory level of Original LBM. "Self" means not including functions called in the current loop or function. The columns of "Self Memory", "Self L2", "Self L3" , and "Self DRAM" collect the data transfer between CPU and L1 cache, L1 and L2 cache, L2 and L3 cache, L3 cache and DRAM, respectively.

streaming sweep. Its "Self CPU time" takes 61.2% of the total CPU time and ranks the most time-consuming kernel. We find that "Self L2" of "loop in propagate" kernel is 17.5GB, and is 3.9X of its 4.5GB "Self Memory", which indicates that the Original LBM has huge L1 cache misses, mostly likely due to inefficient memory access patterns. We will see these numbers again in the later Roofline chart. Note that the "loop in collide" (the fourth kernel) also suffers a similar problem: 4GB self L2 $>$ 1.3GB self L1. The "loop-in-bgk" spends 20% of the total CPU time and is the second hotspot kernel. Since this kernel locally computes the equilibrium states using the data in each cell, its "Self L2 , L3 and DRAM" are all zero. At last, the "bgk" kernel takes 15.4% of total CPU time and is the third hotspot kernel. It updates the macroscopic value, e.g., density and velocity norm. Similar to "loop-in-bgk", it is bounded by L1 cache.

Fig.5.5a shows the cache-aware Roofline model (CARM) of the Original LBM. Firstly, the "+" mark shows the whole performance of the Original LBM. Secondly, the yellows dots are the loops or functions in the Original LBM, and their size and color shows how many portions of total running time they take. Small dots that spends less than 1% of the total time are likely not worth optimization. The medium and yellow dots spend 1% $\sim$ 20%, while large and red dots spend larger than 20% and are the best candidates for optimization. In this example, the large red dot is the "loop in propagate", while the yellow dots are "loop in collide", "bgk-macroscopic" and "loop in bgk". Thirdly, Fig.5.5b presents the Hierarchical



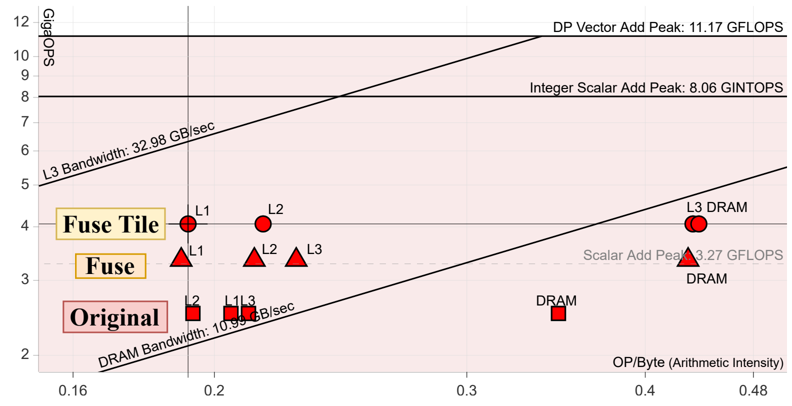
(a) Cache-aware Roofline model of loops and kernels in the Original LBM.



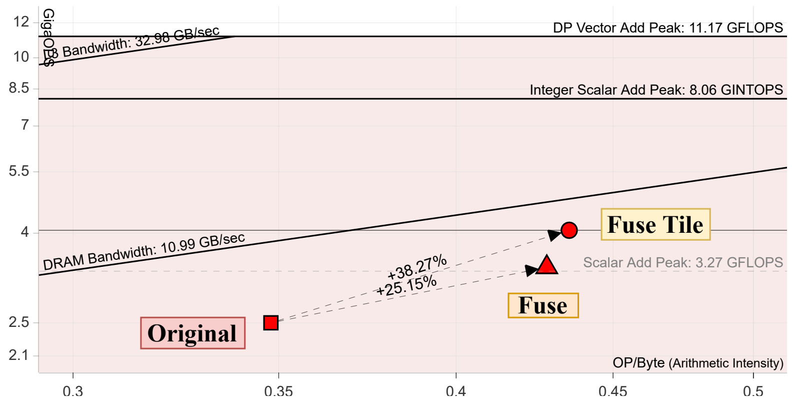
(b) Hierarchical Roofline model of the “loop in propagate”. The purple dot (DRAM AI) is closest to the DRAM roofline, which indicates it is a DRAM-bound kernel.

Figure 5.5. Sequential performance of the Original LBM on a square lattice with $L = 1024$.

Roofline model of the “loop in propagate” kernel, which indicates that it is bounded by DRAM, since the purple dot (DRAM AI) gives the minimum of GigaOPS and has the closest distance to its memory ceiling when comparing with other memory levels. Due to the memory bound feature, directly increasing parallelism on this kernel will not give benefit. Thus, we need to increase its DRAM AI, normally by redesigning the algorithm, e.g., increasing data reuse in the cache, ensuring memory affinity, restructuring loops for more unit stride memory access, etc.



(a) Hierarchical Roofline model comparison.



(b) DRAM Roofline model comparison.

Figure 5.6. Callstacks Roofline comparison of the Original, Fuse, and Fuse tile LBM on a 2D square lattice with edge size $L = 1024$.

Next, we compare the performance among the Original, Fuse, and Fuse tile LBM ($tile = 256$). Firstly, Fig.5.6a shows that they are all bouned by DRAM, In particular, the Original LBM is below the "Scalar Add Peak", whereas the Fuse LBM and Fuse tile LBM are above it and have higher GigaOPS. Secondly, the Original LBM's L1 AI is on the right of its L2 AI, which indicates it has large L1 misses and flushes data to L2 and L3 cache, whereas Fuse LBM and Fuse tile LBM don't have the phenomenon. Thirdly, the AI of Fuse tile LBM's L1, L2, and DRAM is a little larger than its counterpart in Fuse LBM, but its L3 AI contains a big increase. This indicates that the "loop tiling" technique used in Fuse tile LBM take effects and optimizes the L2 memory access, so that Fuse tile LBM has less L2 misses, less data transfer between L2 and L3 cache, thereby results in higher L3 AI. Fourthly, Fig.5.6b zooms in on the comparison of three algorithms using DRAM Roofline model, since they are all bounded by DRAM. We see that both Fuse LBM and Fuse tile LBM are on the right of Original LBM, and have 38% and 25% speedup, respectively. Although Fuse tile LBM has used the "spatial" blocking, it is still under the DRAM ceiling. If we want to continue improving its performance, "temporal" blocking to merge multiple time steps of computation can be chosen to increase the data reuse in cache and improve the AI.

5.4 Related Works

5.4.1 Optimization of Data Storage and Streaming Patterns

The standard LBM implementation follows the particle distribution at lattice cell x and time t with discrete velocities. The original LBM presented in Sect.5.2.1 uses two copies of the particle distribution per lattice cell and *two* distinct collision and streaming kernels, which is also called "AB2k" or "ping-pong buffering". Although the strategy doubles the total memory allocation, it simplifies the streaming kernel to prevent overwritten issues, as the distributions $f_k(t)$ and $f_k^{out}(t)$ are stored at source memory location *buffer A*, while the distributions $f_k(t+1)$ stored at destination memory location *buffer B*, then the two buffers are alternated at each time-step. But since the collision and streaming kernels are separated, there is no effective data reuse between them. The Fuse LBM (or "AB1k") presented in Sect.5.2.1 improves performance by 'fusing' collision and streaming into a single kernel, thereby allows for data reuse between the two kernels. Next, the Fuse tile LBM continues to combine loop tiling with Fuse LBM, and benefits from the spatial locality. But the two optimized methods still use two distribution copies.

Newer single kernels (e.g., swap [43], AA [45], shift [44], and esoteric twist [46], etc.) retains only a single copy of the particle distribution and optimizes the data reuse in streaming kernel, but each needs to follow some constraints.

The swap method [43], [121] observes that when a cell sends a population to its neighbors during streaming, it also receives a population from the same neighbor. Thus, the two forth and back copy operations between a cell and its neighbor in the streaming kernel can be fused into a single value swap. As a result, the swap algorithm is in-place and doesn't require the second memory allocation. However, while combining swap with a fused collision-streaming cycle, we must guarantee that the populations of the neighbor involved in the swap are already in a post-collision state, in case of violating thread safety. A work around solution is to adjust the traversal order of the simulation domain with a predefined order of discrete cell velocities [121], [122].

The shift method [44], also known as compressed grid method, allocates an extra temporary space for one line (for 2D LBM) or one surface (for 3D LBM) which contains nearest-neighbors. The shift method reuses the temporary space in streaming operation. This method reduces the storage to $q * (N^d + N^{d-1})$, but it requires diagonally alternating traversing pattern in even and odd time steps to avoid violating data dependencies, which makes the implementation complex. Besides, they don't provide parallel implementation.

The AA method [45] offers a fused collision-streaming step and a single-memory implementation by storing data in different locations at two subsequent even and odd time steps. At the even time step, we perform a single collision at t without streaming. At the odd time step, there are three sequential operations: a `Pull` operation, a collision at $t+1$, and a `Push` operation. The `Pull` operation gathers the populations at t from the neighbor cells to a local and temporary array for collision at $t+1$. The `Push` operation eventually writes the post-collision variables back to the same locations at the neighbors.

Similar to the swap method, the esoteric twist (ET) method [46] also observes the esoteric (unintuitive) pattern that the streaming step can be eliminated if the distributions are written back in twisted (opposite) order compared to the reading before the collision. Similar to AA, it can be implemented using odd and even steps but not necessarily. It requires a structure of array (SoA) data layout for accessing the distributions. A node's local distributions are now inside a virtual cell that is shifted by half a lattice cell's diameter in each direction. Then distributions are read, collided, and written back to the opposite direction. After each node has been updated, the pointers of opposing discrete velocity directions in the control structure are exchanged. When combined with indirect addressing, it requires fewer ghost nodes than the AA-pattern. Consequently, ET requires only a single read and write operation for each datum in each time step and only a single place in main memory.

Vardhan et al. [30] and Argentini et al. [123] choose another path to reduce the memory footprint by only storing macroscopic, moment-based data (density ρ, velocity u, and the symmetric stress tensor Π) instead of distributions. Therefore, the moment representation of Regularized LBM in [30] reduces the 19 distribution components of a single copy in 3D to 10 moment variables. Since we cannot recompute moments of a cell until all adjacent cells stream to its distribution, this method requires temporary layer storage and enforces ordering layer by layer in 3D. Their implementation is in HARVEY[124] and adopts indirect addressing to solve sparse simulation.

Although the above single kernels reduce to one distribution copy and even less, they mainly focus on optimizing the memory access pattern within one time step, and haven't considered merging two or more time steps to further explore temporal data reuse together with the spatial tiling or blocking. Our memory-aware LBM starts to explore the effectiveness of the idea by using the two distribution copies in 2D, and then combine this idea with the swap method to use one distribution copy in 3D.

5.4.2 Difference with Wavefront Related Algorithms

Wavefront algorithms are characterized by a dependency in the processing order of cells within a spatial domain [125]. Each cell in a multidimensional spatial grid can only be processed when previous cells in the direction of processing flow have been processed. *Pipelined wavefront parallelism* generally groups many threads to compute on the same spatial domain. Then successive wavefronts computation by each thread are executed in a shared last level cache to reduce cache misses, thereby improving memory access performance. However, a thread can only start computation on the domain of a time step after a previous thread completes, and thus there exists intensive synchronization cost among threads in every time step.

Song et al. present a shared-memory wavefront LBM only in 2D, and also utilizes loop fusion, loop bump, loop skewing, and loop tiling [47]. To alleviate the implicit barriers in wave-front parallelism, they propose a synchronization strategy based on the semaphore operations of `POST` and `WAIT`. However, their thread synchronization costs in every time step are still high when each thread is assigned a small sub-domain, they only achieve 10% parallel performance speedup on average.

Habich et al. present a shared-memory wavefront LBM in 3D [126] to explore the temporal locality, but they don't combine spatial locality techniques, e.g. loop fusion or loop blocking. Wellein et al. present a shared-memory pipelined wavefront parallelization approach combined with spatial blocking for the 3D Jacobi method, which is a 6-neighbors stencil-based computation [51]. It has simpler dependencies than the 19 or 27 neighbors in 3D LBM. Besides, both these two work contain wavefront synchronization costs.

By contrast, our 2D and 3D memory-aware LBM do not use the wavefront parallelism, but judiciously contains two or three light-weight synchronization barriers every two or more collision-streaming cycles. In addition, we partition the simulation domain and assign a local sub-domain to every thread, rather than all threads work on the same sub-domain in wavefront parallelism. In each sub-domain, each thread in our algorithm computes multiple time steps at once, rather than one thread computes one time step at a time in wavefront parallelism. Each of our threads also utilizes tiling or prism techniques to optimize spatial locality. This strategy in particular favors new manycore architecture designs, which tend to have increasingly larger cache sizes on each core.

5.4.3 Difference with Cache Oblivious Algorithms

T.Zeiser et al. introduce a parallel cache-oblivious blocking algorithm (COLBA) [127] for the LBM in 3D. COLBA is based on a cache-oblivious algorithm [128], and divides the space-time domain using *space cut* and *time cut*, thus tries to remove the explicit dependency on the cache size. However, it comes at the cost of irregular block access patterns, which causes many cache misses and branch-prediction misses. Due to the recursive structure of the algorithm, they also use an unconventional parallelism scheme to map the virtually decomposed domain to a tree. This work is quite different from ours since it not only uses the recursive method but also has irregular data accesses.

5.4.4 State-of-the-art CFD and LBM Software Packages

The Parallel Lattice Boltzmann Solver (Palabos) [116] is an MPI and C++ open-source library for general-purpose CFD, and developed since 2010 as a research and engineering tool. The library uses MPI and C++ templates to support a wide scope of collision models. Palabos adopts a matrix-based array-of-structure (AOS) memory organization at the cell level, so that the neighbors can be easily accessed by index arithmetics, but have a limitation to regular and rectangular shapes. To overcome the drawback, the irregular shapes in Palabos are decomposed into multiple blocks, which are simply smaller matrix pieces.

OpenLB [129] is a C++ LBM package and is the successor of the VLADYMIR [130] library. Palabos developers broaden OpenLB to the simulation with coupled physics and complex geometries. HemeLB [131] and the Palabos-based solver HemoCell [132] focus on the field of computational biomedicine simulation.

Other packages adopt adjacent list data structure (indirect addressing), e.g. Musubi [133] and waLBerla [134]. HARVEY [124] is designed for simulations in complex vascular geometries. They are often used for simulating domains with sparse and irregular geometries, but their cells require additional memory of pointers, and double the memory consumption in the worst case.

Open Field of Operation And Manipulation (OpenFOAM) [76] is a C++ open-source library released in December 2004 and widely used in the CFD community. Its original development started in 1989 at Imperial College, London. It focuses on the unstructured grid and uses pressure correction methods. OpenFoam provides *utilities* as functional tools to pre/post-processing, e.g. blockMesh, sampling tool, and its *solvers* calculate the numerical solution of PDEs. Since its theoretical background is the Finite Volume Method (FVM), it doesn't provide LBM implementation. In a finite scheme, functional values usually have to be gathered in a finite neighborhood of a grid node, and the same data is required to update several grid nodes. However, in LBM, the number of input variables of the local time integration scheme equals the number of output variables, thus each input data is required only once.

In this study, we choose the more widely-used matrix-based data structure in the LBM community, and select the state-of-the-art Palabos library as the baseline, since Palabos offers a broad modeling framework, targets applications with complex physics, and exhibits solid computational performance.

5.5 Two-step Memory-aware LBM Algorithm

To further improve LBM performance, we aim to increase data reuse across two or multiple time steps of collision-streaming cycles in the sequential LBM, and then design its parallel version. However, there are two main challenges:

1. For the 2D sequential LBM, how to correctly arrange the access pattern to merge two time-steps of the collision and streaming cycle, meanwhile combining loop tiling, and handling the boundary conditions?

2. For the parallel 2D LBM, how to keep thread safety on the intersection area between different threads? How to minimize synchronization cost? And how to explain the parallel performance improvement?

5.5.1 Sequential Two-step Memory-aware LBM

We start from the Fuse LBM presented in Sect.5.2.2, and explore data reuse across *two* time steps. Assuming a block of cells fit in the last level cache, when traversing the domain first from along Y-axis and then from bottom to top along X-axis, we observe that after the first collision and streaming operation of a cell $(\mathrm{i}x, \mathrm{i}y)$ at the time step t, its neighbor $(\mathrm{i}x-1, \mathrm{i}y-1)$ fulfills the data dependency and is ready for the collision at the time step $t+1$. Indeed, we can perform the second computation on $(\mathrm{i}x-1, \mathrm{i}y-1)$ to increase the cache reuse as long as the last level cache can hold all the data of one line of Y-axis. This idea is essentially simple, which leads to the Alg.5. [3] Note that we don't need alternate *buf1* and *buf2* at the end of every two collision-streaming cycles.

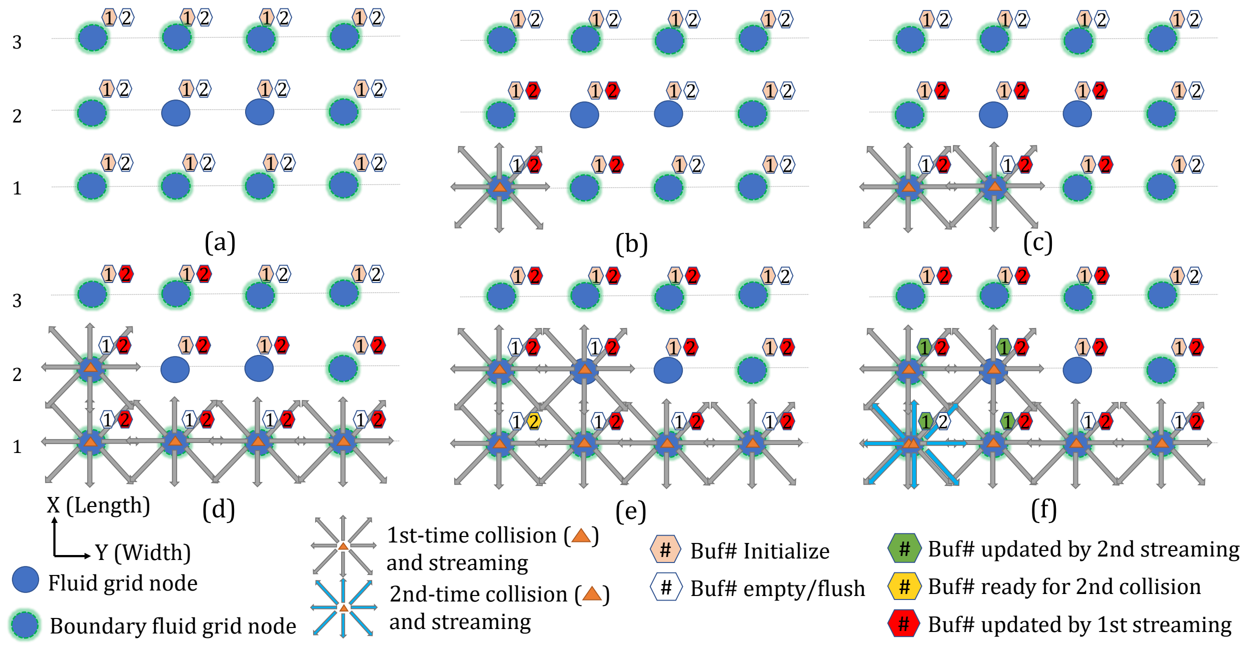
Now we illustrate the 2D sequential two-step LBM Alg.5 by an example of 3×4 lattice in Fig.5.7. To conform with Alg.5, the horizontal direction in Fig.5.7 is defined as Y-axis, while the vertical direction is X-axis. The later figures will follow this definition. Cells are placed from the bottom left corner $(1, 1)$ to top right corner $(3, 4)$. The algorithm works as follows.

1. Fig.5.7.a shows the initialization state of all cells at the current time step t.

[3]↑The design of sequential 2D memory-aware LBM is a joint work with Feng Li.

Algorithm 5 2D Sequential Two-step Memory-aware LBM

```
1:  for iT=0; iT < N; iT += 2 do
2:    for ix=1; ix ≤ $lx$; ix++ do
3:      for iy=1; iy ≤ $ly$; iy++ do
4:        /*First Fused Collision and Streaming*/
5:        compute (ix, iy) collision using buf1
6:        stream (ix, iy)'s buf1 to its own and 8 neighbors' buf2
7:        /*Second Fused Collision and Streaming*/
8:        if ix>1 and iy>1 then
9:          compute (ix-1, iy-1) collision using buf2
10:         stream (ix-1, iy-1)'s buf2 to its own and 8 neighbors' buf1
11:   Compute the second fused collision and streaming on rightmost column $lx$.
```



3
2
1
(a)
(b)
(c)
(d)
(e)
(f)
X (Length)
Y (Width)
Fluid grid node
Boundary fluid grid node
1st-time collision (▲) and streaming
2nd-time collision (▲) and streaming
Buf# Initialize
Buf# empty/flush
Buf# updated by 2nd streaming
Buf# ready for 2nd collision
Buf# updated by 1st streaming

Figure 5.7. Sequential memory-aware algorithm. Note that the horizontal direction in this figure is the channel's width (Y axis in Fig.5.23), while the vertical direction is the channel's length (X axis in Fig.5.23). This notation will be used in the later algorithm illustration diagram. **(a)** Initialization. **(b)** First collision and streaming on $(1, 1)$. **(c)** First collision and streaming on $(1, 2)$. **(d)** Continue computing the first collision and streaming through $(2, 1)$. **(e)** First collision and streaming on $(2, 2)$ and fulfill the data dependency of $(1, 1)$ to compute the second collision and streaming. **(f)** Second collision and streaming on $(1, 1)$.

2. In Fig.5.7.b, we start computing the first fused collision and streaming on the cell $(1, 1)$ at the left corner. We use the data in *buf1* to perform collision. Then in the streaming operation, we propagate the data in the *buf1* of cell $(1, 1)$ to the *buf2* of itself and

its neighbors. Thus, we change these *buf2* to red, which indicates that these *buf2* are updated but still lack other dependent data to compute the collision at the time step $t+1$. Since *buf1* of cell $(1,1)$ has been computed completely, we change it to white, indicating that this *buf1* is flushed and can be updated by the data at the time step $t+1$.

3. In Fig.5.7.c, we compute the cell $(1,2)$ by performing the fused collision and streaming operation at the time step t.
4. After completion of computing the bottom row 1, we move up one row and compute the first fused collision and streaming on cell $(2,1)$ on row 2 in Fig.5.7.d.
5. In Fig.5.7.e, we compute the first fused collision and streaming on cell $(2,2)$. Note that after cell $(2,2)$'s streaming phase, since the *buf2* of cell $(1,1)$ has collected all the particle distributions from its neighbors, the data dependency of *buf2* in cell $(1,1)$ is fulfilled and ready for computing the second collision in the time step $t+1$. Thus we change its *buf2* to yellow, indicating that the data within the buffer is ready for the second fused collision at the time step $t+1$.
6. In Fig.5.7.f, we perform the second fused collision and streaming at the time step $t+1$ on cell $(1,1)$ using *buf2*. After the streaming phase, since the *buf1* of cell $(1,1)$ and its neighbors are flushed previously, we can safely propagate and store the "intermediate" data at the time step $t+1$ to those *buf1*. Thus we change these *buf1* to green, indicating that they store the data at the time step $t+1$ but still lack other dependent data to compute the collision at the time step $t+2$.

Next, we can use the loop tiling to combine spatial and temporal cache blocking together, which leads to Alg.6. The MIN statement again allows Alg.6 to support any non-negative tile size to choose the best tile according to different CPU's cache size. Alg.6 accesses the whole 2D lattice tiles by tiles using a constant stride, and then accesses cells in each tile line by line using a unit stride. With such a memory access pattern and data reuse among multiple time steps, the sequential two-step memory-aware LBM algorithm can improve performance significantly.

Algorithm 6 2D Sequential Two-step Memory-aware LBM with Loop-tiling

```
for iT=0; iT < N; iT += 2 do
  for outerX = 1; outerX $\leq lx$; outerX += tile do
    for outerY = 1; outerY $\leq ly$; outerY += tile do
      for innerX = outerX; innerX $\leq$ MIN(outerX + tile - 1, lx); ++innerX do
        for innerY = outerY; innerY $\leq$ MIN(outerY + tile - 1, ly); ++innerY do
          // First fused collision and streaming:
          compute (innerX, innerY) collision using buf1
          stream (innerX, innerY)'s buf1 to its own and 8 neighbors' buf2
          // Second fused collision and streaming:
          if innerX>1 and innerY>1 then
            if innerX == $lx - 1$ or $lx$ then // Handle boundary condition
              save $\rho$ at column $lx - 1$ & $lx - 2$
            compute (innerX-1, innerY-1) collision using buf2
            stream (innerX-1, innerY-1)'s buf2 to its own and 8 neighbors' buf1
  Compute the second fused collision and streaming on the rightmost column $lx$ and handle
boundary conditions. // see section 5.5.2
```

5.5.2 Special Handling of Boundary Conditions

Boundary conditions (BCs) are complex and affect the correctness, stability, and accuracy of LBM. The discrete distribution functions on the boundary have to be taken care of to reflect the macroscopic BCs of the fluid. Fig.5.8 shows the four BCs used in our LBM simulation experiments. The horizontal and vertical axis are in the same direction as Fig.5.23:

1. The upper (red) and lower (green) boundaries use regularized BC.

2. At the inlet (yellow left boundary), a parabolic Poiseuille profile is imposed on the velocity.

3. At the outlet (blue right boundary), an outflow condition: $\nabla u = 0$ is implemented. At every time step, we compute a second-order extrapolation on the right boundary to ensure a zero-gradient BC on the pressure. Thus the velocity is constrained to be perpendicular to the outflow surface.

4. On the surface of the cylinder obstacle and within its interior, the bounce-back BC is used.

Other different BCs can also be applied, as long as we follow the procedure in the next paragraph.

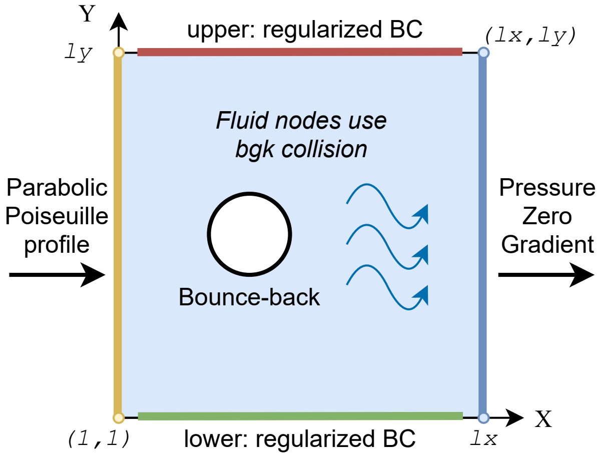
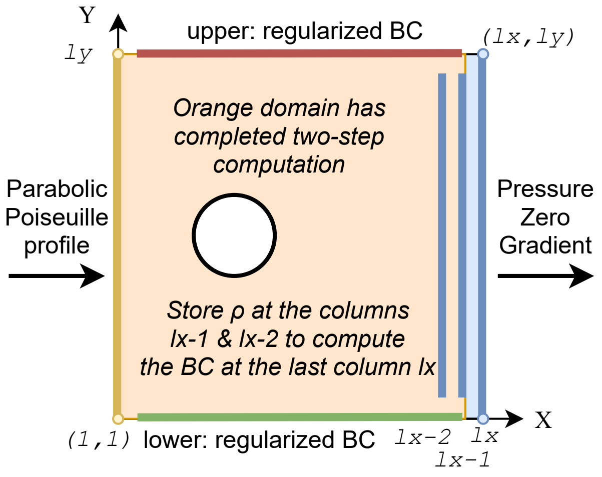


Figure 5.8. Four boundary conditions used in the LBM simulation.



Y
ly
upper: regularized BC
(lx,ly)
Orange domain has completed two-step computation
Parabolic Poiseuille profile
Store ρ at the columns lx-1 & lx-2 to compute the BC at the last column lx
Pressure Zero Gradient
(1,1)
lower: regularized BC
lx-2
lx
lx-1
X

Figure 5.9. Handle the right outlet BC. When Alg.6 reaches innerX = lx, the orange domain has completed two-step computation. We use two extra arrays to store the density ρ of cells at column $lx-1$ and $lx-2$ after the first fused computation but before the second fused computation, so that the right outlet BC at the time step $t+1$ can be correctly handled.

Handling the Four BCs.

The upper, lower and left BCs are all local computation and do not use neighbors' data, but the right outlet BC (zero-gradient on pressure) is not local, and it requires data from its two neighbors on the left. Specifically, we have to conform to the formula $\rho_{x,y} = 4/3 \times \rho_{x-1,y} - 1/3 \times \rho_{x-2,y}$. This indicates the density ρ of cells at column lx depends on the ρ of cells at column $lx-1$ and $lx-2$. But every time a cell executes collide function, its density ρ will change. To ensure the correctness of the right BC at the time step $t+1$, we need two extra arrays to store the density ρ of cells at column $lx-1$ and $lx-2$ after the first fused computation but before the second fused computation. This is reflected in lines 8 and 9 in Alg.6. Thus, with the ρ stored in the two arrays, we can successfully compute the second fused collision and streaming, and handle the right outlet BC correctly at the time step $t+1$.

5.5.3 Parallel Two-step Memory-aware LBM Algorithm

To further improve performance and support manycore systems, threading can be used to parallelize the 2D sequential two-step memory-aware LBM. OpenMP [135] is a portable shared-memory programming model that can quickly convert a sequential program to a multi-threaded parallel program with high performance. However, to design the 2D two-step LBM with OpenMP, there are several challenges:

1. To reduce the frequent synchronization cost, how to divide the computational domain among multiple threads?

2. To avoid race conditions and keep thread safety, how to handle the "intersection domain" (overlapped cells) between two threads?

3. How to add tiling to increase spatial locality into the new parallel algorithm?

We will resolve these challenges in this section.

Fig.5.10 illustrates the idea of 2D parallel two-step memory-aware LBM Alg.7 on a 6×4 lattice. The whole lattice is evenly distributed to two threads along the channel's length (X-axis), thus each thread computes a 4×3 sub-lattice. I define the top row of each thread's sub-lattice as the "intersection", e.g., row 3 is the intersection. The algorithm generally works in three stages and seven steps:

1. Stage I (prepossessing): In Fig.5.10.a$\sim$c, each thread computes the first fused collision and streaming on the "intersection" (row 3 & 6), as mapped to lines 3$\sim$7 in Alg.7.

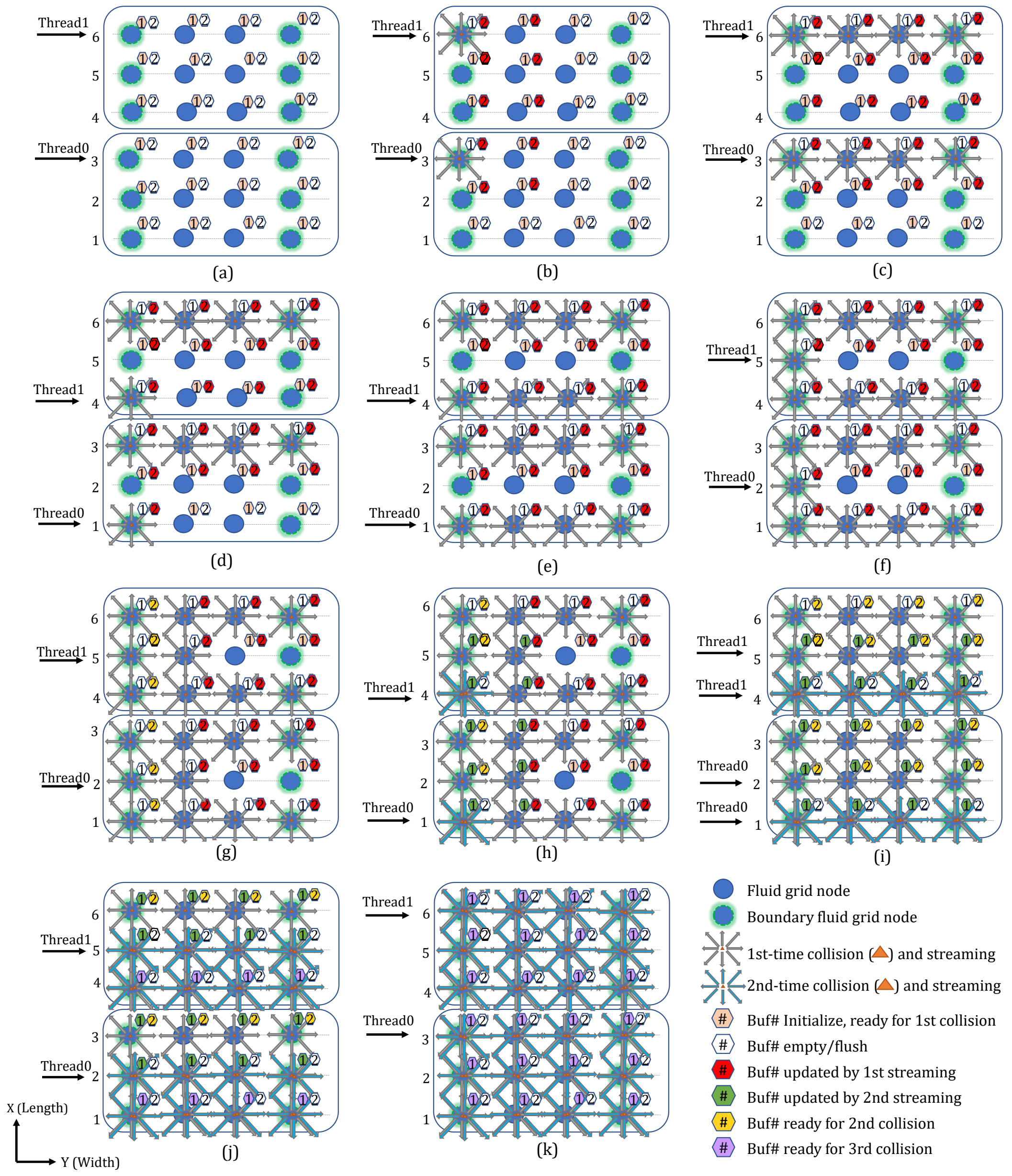

Thread1
Thread0
(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
(k)
X (Length)
Y (Width)
Fluid grid node
Boundary fluid grid node
1st-time collision ( ) and streaming
2nd-time collision ( ) and streaming
Buf# Initialize, ready for 1st collision
Buf# empty/flush
Buf# updated by 1st streaming
Buf# updated by 2nd streaming
Buf# ready for 2nd collision
Buf# ready for 3rd collision

Figure 5.10. Parallel two-step memory-aware algorithm in Y-X axis. **(a)** Initialization. **(b∼c)** First computation on the "intersection" (row 3 & 6) at time step t. **(d∼e)** First computation on row 1 and 3 at time step t. **(f)** First computation on the leftmost cell on row 2 and 5. **(g)** When thread 0 and 1 complete the first computation on $(2, 2)$ and $(5, 2)$ respectively, the *buf2* in $(1, 1)$ and $(4, 1)$ fulfill the data dependency for the second computation at time step $t + 1$. **(h)** Second fused computation on $(1, 1)$ and $(4, 1)$. **(i∼j)** Do the first computation on row 2 and 5, meanwhile do the second computation on row 1 and 4. **(k)** Second computation on the "intersection" (row 3 & 6).

2. Stage II (main computation in each thread's sub-lattice): In Fig.5.10.d∼e., each thread computes the first fused collision and streaming on the bottom row of their sub-lattice (row 1 & 4) as mapped to lines 14∼16 in Alg.7.

3. In Fig.5.10.f, each thread computes the first fused collision and streaming on the second row (row 2 and 5) in each sub-lattice.

4. In Fig.5.10.g, the *buf2* of some cells fulfill the data dependency for the second collision, we change them to yellow.

5. In Fig.5.10.h, each thread computes the second fused collision and streaming on these cells in step 4, as mapped to lines 20∼22 in Alg.7.

6. In Fig.5.10.i∼j, each thread repeats steps 3∼5 and completes two-step computation for the rest of the cells in their own sub-lattice except for the "intersection".

7. Stage III: In Fig.5.10.k, each thread computes the second fused collision and streaming on "intersection", as mapped to lines 24∼29 in Alg.7.

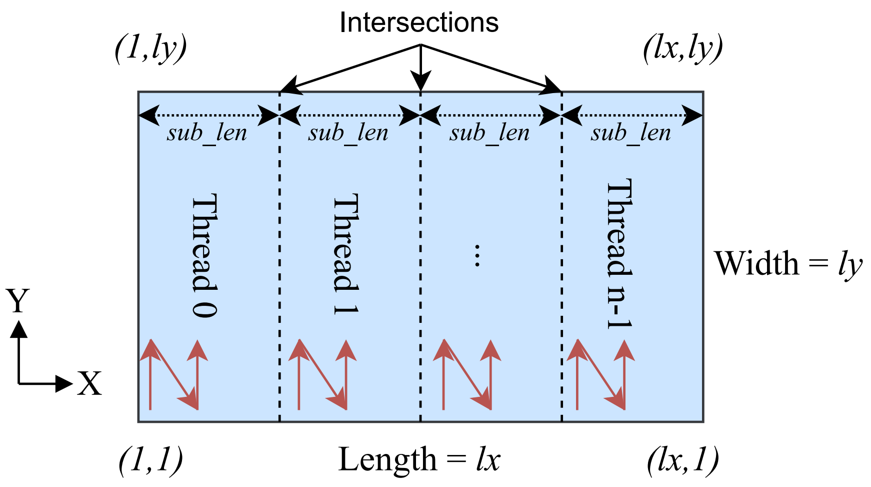


Figure 5.11. Partition of a 2D lattice by parallel memory-aware LBM.

Alg.7 presents the parallel two-step memory-aware LBM with loop tiling. The 2D lattice is evenly partitioned by n threads along X-axis (length lx) in Fig.5.11, and columns at

Algorithm 7 2D Parallel Two-step Memory-aware LBM with Loop-tiling

```
for iT=0; iT < N; iT += 2 do
  #pragma omp parallel{
  // tid: each thread id; sub_len: length of each thread's sub-lattice
  mylx[0] = 1 + tid * sub_len; // the lowest row of the sub-lattice
  mylx[1] = (tid + 1) * sub_len; // "intersection", i.e., the highest row of the sub-lattice
  /* Stage I: First fused collision and streaming on the "intersection": */
  for iy=1; iy≤ $ly$; ++iy do
    compute (mylx[1], iy) collision using buf1
    stream (mylx[1], iy) buf1 to its own and 8 neighbors' buf2
  #pragma omp barrier
  /* Stage II: Main computation in each thread's sub-lattice: */
  for outerX = mylx[0]; outerX ≤ $mylx$[1]; outerX += tile do
    for outerY = 1 ; outerY ≤ $ly$; outerY += tile do
      for innerX=outerX; innerX ≤ MIN(outerX + tile - 1, $mylx$[1]); ++innerX do
        for innerY=outerY; innerY ≤ MIN(outerY + tile - 1, ly); ++innerY do
          // First fused collision and streaming at time step $t$:
          if innerX != mylx[1] then
            compute (innerX, innerY) collision using buf1
            stream (innerX, innerY)'s buf1 to its own and 8 neighbors' buf2
          // Second fused collision and streaming at time step $t + 1$:
          if innerX != mylx[0] and innerY > 1 then
            if innerX == $lx - 1$ or $lx$ then // Handle boundary condition
              save $\rho$ at column $lx - 1$ & $lx - 2$
            compute (innerX-1, innerY-1) collision using buf2
            stream (innerX-1, innerY-1)'s buf2 to its own and 8 neighbors' buf1
  #pragma omp barrier
  Stage III: Second fused collision and streaming on "intersection" mylx[1] and row $ly$, meanwhile handle boundary conditions.
  }
```

$lx/n, 2*lx/n, ..., lx$ are *"intersections"*. Thus, each thread computes a local $lx/n \times ly$ sub-lattice. The red arrows describe the tiling traversal in each thread's sub-lattice. The "MIN" statements in 13 and 14 allow users to select the best non-negative $tile$ based on the size of the last level cache on their architectures. Since line 22 stores the density ρ at cells on column $lx-1$ and $lx-2$ in the post-collision state at time step t, we can compute the zero gradient BCs at time step $t+1$, as mentioned in Sect. 5.5.2.

Handling Thread Safety on Intersection Lines

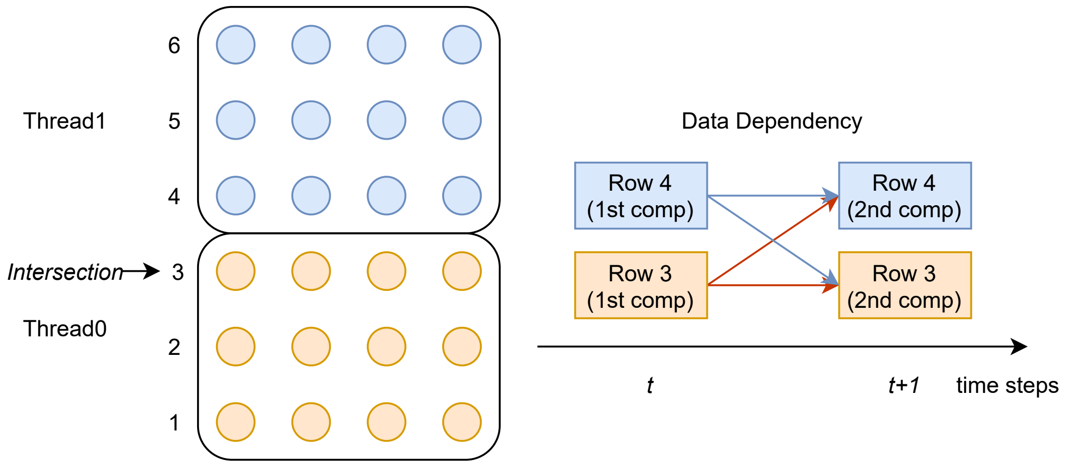


Figure 5.12. Handle the intersection line. To keep thread safety, the second computation on row 4 should be delayed after the first computation on row 3, and the second computation on row 3 should also be delayed after the first computation on row 4.

We handle the thread safety on the "intersection layer" between two threads in Stage I (step 1) and Stage III (step 7). As shown in Fig.5.12, the second computation on the bottom row in thread 1's sub-lattice (row 4) needs the data streamed from the *intersection* (i.e., row 3, the top row in the thread 0's sub-lattice). In other words, the second computation on row 4 should be delayed after the first computation on row 3. Thus we need to compute the first fused collision and streaming on row 3 in advance, which is pre-processed by Step 1, mapping to lines 6∼8 of Alg.7. And the first barrier is inserted in line 9 of Alg.7 to ensure the delay.

In Stage II (step 2∼5), each thread starts computation from the lowest row in their own sub-lattice. Main computation happens here, mapping to line 10∼24 of Alg.7. When thread 0 reaches the highest row of its sub-lattice (row 3), since the first computation has already been completed on these row, we ignore the first computation in line 15 in Alg.7, but compute the second computation on its lower one row. The second computation on row 3 should be delayed after the first computation on row 4 in Fig.5.12, as mapped to line 25 in Alg.7.

In Stage III, we compute the second computation on the intersection row ly (the rightmost column in Fig.5.10). Above all, since the majority of computation happens in Stage II of each thread's sub-domain, we avoid the frequent "line-wise" thread synchronizations that occur in the wave-front parallelism. Besides, we only synchronize at the intersection lines every two time steps, hence the overhead of two barriers of Alg.7 becomes much less.

5.6 k-step Memory-aware LBM Algorithm

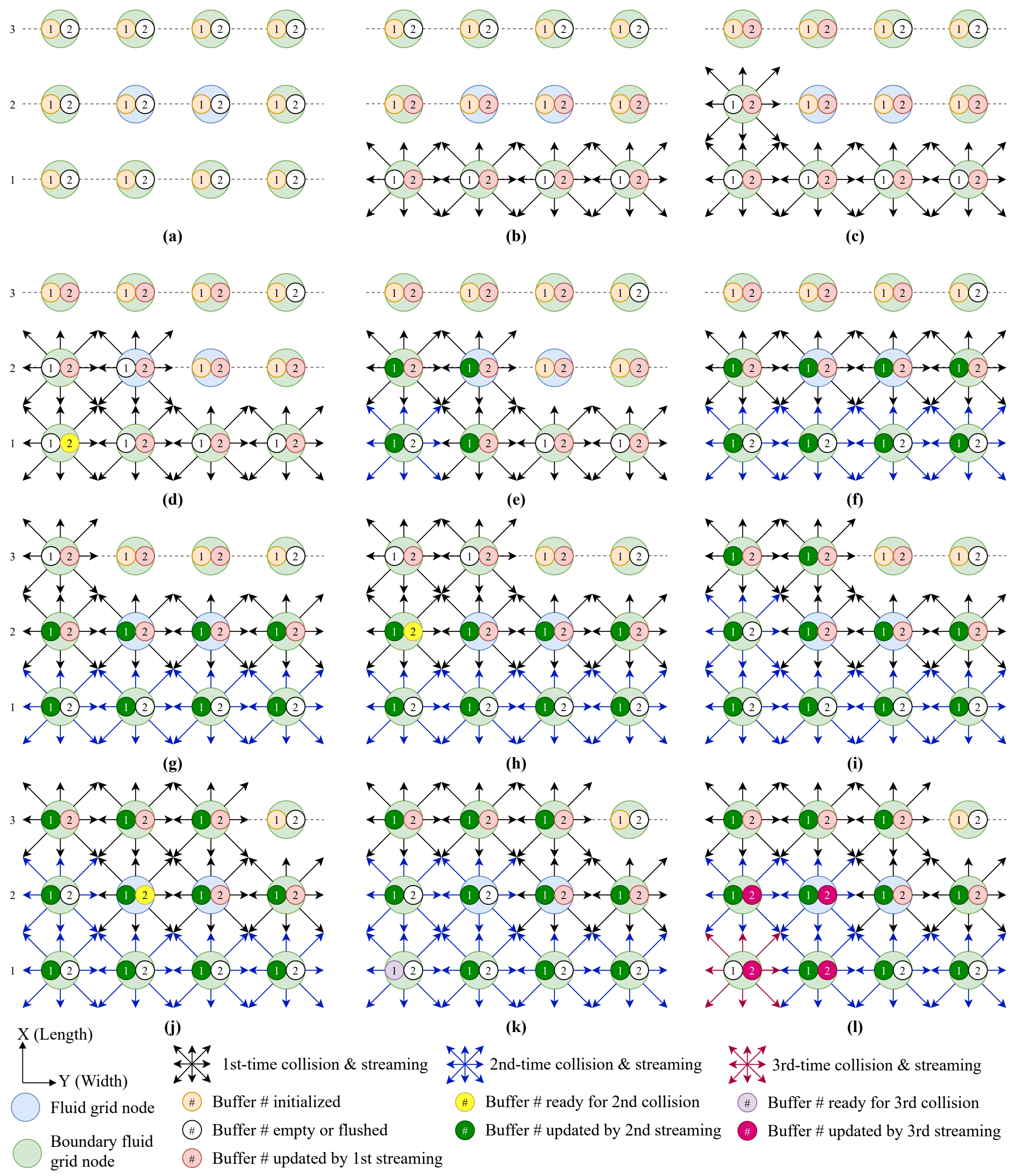
Sect.5.5 designs the two-step memory-aware LBM, and we ask ourselves: can we merge three steps or more? This section extends the two-step LBM by more temporal and spatial data reuse to further increase performance.

5.6.1 Sequential k-step Memory-aware LBM Algorithm

The basic idea of k-step memory-aware LBM is to merge k time steps into one iteration. Since the key observation of the two-step memory-aware LBM is that when completing the first computation on (ix, iy), we can execute the second computation on (ix-1, iy-1). Similarly, when we are at row k, is it possible to compute the k-th computation on (ix-k, iy-k)? Fig.5.13 illustrates how to merge k=3 time steps of LBM computation.

1. Fig.5.13.a∼f do the same operation as Fig.5.7.a∼f to complete the first computation on line 2, and the second computation on line 1.

2. From Fig.5.13.g∼i, we continue using the two-step LBM until cell (3,3).



3
2
1
(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
(k)
(l)
X (Length)
Y (Width)
Fluid grid node
Boundary fluid grid node
1st-time collision & streaming
2nd-time collision & streaming
3rd-time collision & streaming
Buffer # initialized
Buffer # empty or flushed
Buffer # updated by 1st streaming
Buffer # ready for 2nd collision
Buffer # updated by 2nd streaming
Buffer # ready for 3rd collision
Buffer # updated by 3rd streaming

Figure 5.13. Sequential k-step memory-aware algorithm. (k=3) **(a)** Initialization. **(b)** First computation on line 1. **(c)** First computation on $(2, 1)$. **(d)** First computation on $(2, 2)$. The data dependency of $(1, 1)$ is fulfilled to compute the second computation. **(e)** Second computation on $(1, 1)$. **(f)** First computation on line 2. Second computation on line 1. **(g $\sim$ j)** First computation on line 3. Second computation on line 2. **(k)** Second computation on $(2, 2)$. The data dependency of $(1, 1)$ is fulfilled to compute the third computation. **(l)** Third computation on $(1, 1)$.

3. In Fig.5.13.j, we compute the first fused computation on cell $(3, 3)$. Note that after cell $(3, 3)$'s streaming phase, since the *buf2* in cell $(2, 2)$ has collected all the dependent data from its neighbors, cell $(2, 2)$ can compute the second collision at the time step $t + 1$. Thus we change its *buf2* to yellow.

4. In Fig.5.13.k, we perform the second fused computation on cell $(2, 2)$ using *buf2* at the time step $t + 1$. The streaming phase of cell $(2, 2)$ propagates the "intermediate" $t + 1$ data to its own and its neighbors' $buf1$, meanwhile its *buf2* can be flushed, thus we change it to blank. At this moment, cell (1,1)'s *buf1* has fulfilled all the $t + 1$ data dependency to compute at the time step $t + 2$. Therefore, we change cell (1,1)'s *buf1* to purple, indicating that it is ready for the third computation at the time step $t + 2$.

5. In Fig.5.13.l, we perform the third fused computation on cell $(1, 1)$ using *buf1* at the time step $t+2$. After the streaming phase, since the *buf2* of cell $(1, 1)$ and its neighbors are flushed, we can safely propagate the "intermediate" $t + 2$ data to its own and its neighbors' *buf2*. Thus, we change these *buf2* to pink, indicating that they are updated by the third computation at the time step $t + 2$ but still need more dependent data to ready for the fourth computation at the time step $t + 3$.

The 2D sequential three-step memory-aware LBM method with loop tiling is shown in Alg. 8. It supports any non-negative tile size. More details about how to handle BCs both sequentially and in parallel and are discussed in Sect. 5.6.3. Meanwhile, I will present how to avoid the extra two arrays used in Sect. 5.5.2 when supporting the zero-gradient BC.

Furthermore, loop-unrolling can be used to extract the two if-branches in lines 10 and 14 of Alg.8 outside the four-level nested for-loop. In Fig.5.13, we can prepossess row 1 till the time step $t + 2$, row 2 till the time step $t + 1$, and row 3 till the time step t. Thus, the innermost of the nested for-loop can remove the if-branches which break CPU pipeline, in order to achieve more unit strides. For k > 4 (future work), we can use additional if-branches after line 16 of Alg.8 to compute the fourth fused collision, and streaming by swapping *buf1* and *buf2*, and so on. Besides, we need to handle the second, third, and fourth computation on the last few lines near the boundaries of the simulation domain, which is similar to *lines*

Algorithm 8 2D Sequential k-step Memory-aware LBM with Loop-tiling. (k=3)

```
1: for iT=0; iT < N; iT += 3 do
2:   for outerX = 1; outerX ≤ $lx$; outerX += $tile$ do
3:     for outerY = 1; outerY ≤ $ly$; outerY += $tile$ do
4:       for innerX = outerX; innerX ≤ MIN(outerX + tile - 1, lx); ++innerX do
5:         for innerY = outerY; innerY ≤ MIN(outerY + tile - 1, ly); ++innerY do
6:           /*First fused collision and streaming:*/
7:           collide on (innerX, innerY) using buf1
8:           stream (innerX, innerY)'s buf1 to its own and 8 neighbors' buf2
9:           // /*Second fused collision and streaming:*/
10:          if innerX>1 and innerY>1 then
11:            collide on (innerX-1, innerY-1) using buf2
12:            stream (innerX-1, innerY-1)'s buf2 to its own and 8 neighbors' buf1
13:            // /*Third fused collision and streaming:*/
14:            if innerX>2 and innerY>2 then
15:              collide on (innerX-2, innerY-2) using buf1
16:              stream (innerX-2, innerY-2)'s buf1 to its own and 8 neighbors' buf2
17:   /* Handle remaining incomplete boundaries*/
18:   Second collide using buf2 and stream to neighbors' buf1 on row $ly$ and column $lx$.
19:   Third collide using buf1 and stream to neighbors' buf2 on row $ly-1$ & $ly$, and column $lx-1$
      & $lx$.
20:   Swap $buf1$ and $buf2$.
```

18∼19 in Alg.8. Since k=3 is an odd number, we need to swap the two pointers $buf1$ and $buf2$ at every three time steps, as shown in line 20 of Alg.8.

5.6.2 Parallel k-step Memory-aware LBM Algorithm

This section presents the 2D parallel k-step memory-aware LBM algorithm 9, which is generally divided into three stages. Stage I in Fig.5.16 is the pre-processing stage. which handles the intersection between threads in advance because of data dependency. Stage II in Fig.5.17 and 5.17 are the main computation in each thread's sub-lattice. Stage III is to compute the remaining incomplete cells and to handle BCs.

Let's start with an example to illustrate how the parallel three-step memory-aware LBM works. Similar to the parallel two-step LBM, Fig.5.14.a shows that a 10×4 lattice is assigned to two threads along X-axis. Thus each thread has a 5×4 sub-lattice. The intersection area is the overlapped cells between two threads, i.e., row 5 and 6, and we need to carefully handle the data dependencies when merging three time steps. Fig.5.14.b shows the data

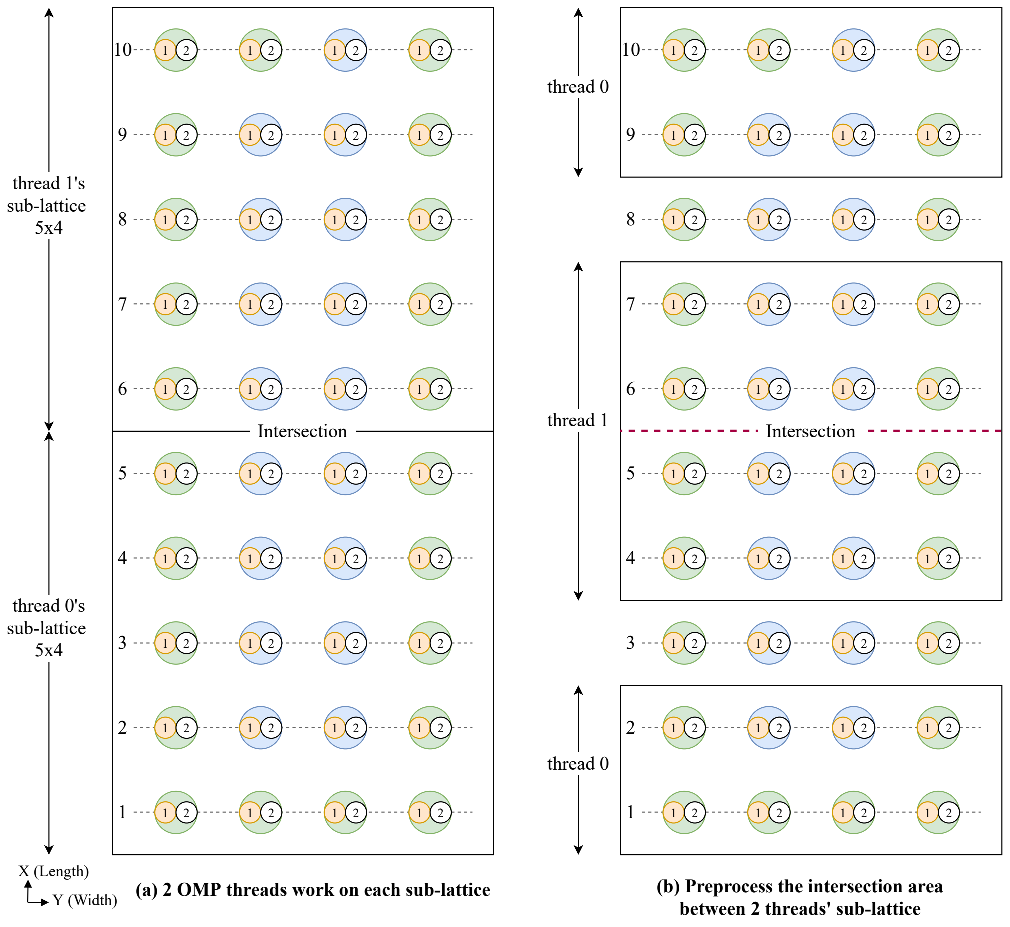

thread 1's sub-lattice 5x4
thread 0's sub-lattice 5x4
Intersection
X (Length)
Y (Width)
(a) 2 OMP threads work on each sub-lattice
thread 0
thread 1
thread 0
Intersection
(b) Preprocess the intersection area between 2 threads' sub-lattice

Figure 5.14. A 10×4 lattice is distributed to 2 threads. **(a)** Each thread owns a 5×4 sub-lattice. **(b)** Pre-processing intersection area: thread 1 computes the data domain from row 4 to 7; thread 0 computes the bottom 2 rows (row 1 & 2) and top 2 rows (row 9 & 10).

domain where each thread will preprocess during Stage I. Thread 1 will preprocess 4 rows from row 4 to 6, while thread 0 will also preprocess 4 rows, which are the bottom two rows (row 1 & 2) and the top two rows (row 9 & 10). For a general case, $lx \times ly$ lattice is distributed to n threads along X-axis. Let $lx/n = h$, $mylx_0 = 1 + thread_{\text{id}} \times h$ and $mylx_1 = (thread_{\text{id}} + 1) \times h$, thus each thread's sub-lattice is $h \times ly$ from $(mylx_0, 1)$ to $(mylx_1, ly)$. In Stage I, $thread_{\text{id}=1\sim(n-1)}$ pre-processes 4 lines $mylx_0 - 2 \sim mylx_0 + 1$, while thread 0 pre-processes lines 1, 2, lx, and $lx - 1$. Fig.5.15 shows the legends in use to describe the parallel k-step memory-aware algorithm.

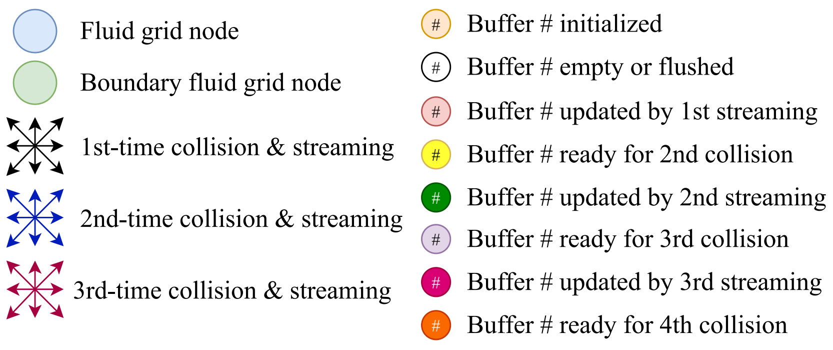
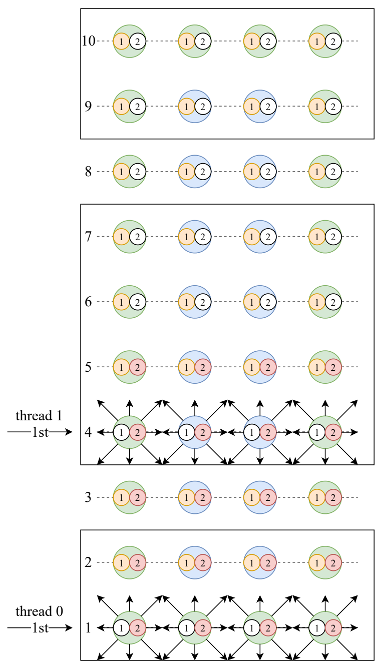

Fluid grid node
Boundary fluid grid node
1st-time collision & streaming
2nd-time collision & streaming
3rd-time collision & streaming
Buffer # initialized
Buffer # empty or flushed
Buffer # updated by 1st streaming
Buffer # ready for 2nd collision
Buffer # updated by 2nd streaming
Buffer # ready for 3rd collision
Buffer # updated by 3rd streaming
Buffer # ready for 4th collision

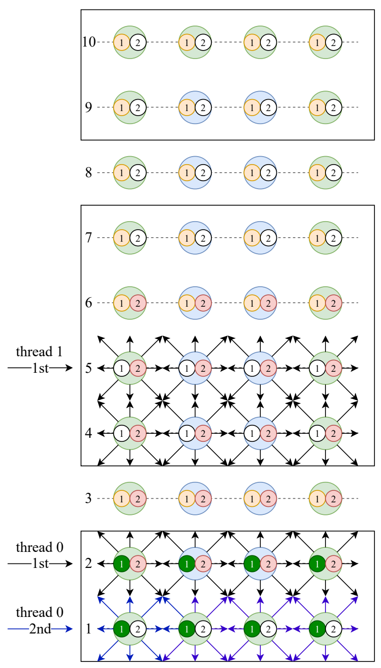
Figure 5.15. Legend used to describe parallel k-step memory-aware LBM.

Fig.5.16 shows Stage I to preprocess the intersection area between the 2 thread's sub-lattice.

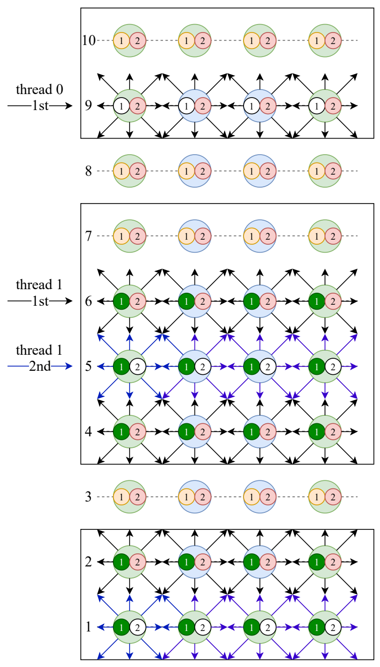
1. In Fig.5.16.a, thread 0 and 1 starts the first fused collision and streaming on row 1 and 4 at time step t, respectively.
2. In Fig.5.16.b, thread 0 starts the first fused computation on row 2, whenever the data dependency for row 1 is fulfilled, we immediately execute the second collision and streaming on row 2 at time step $t+1$. Meanwhile, thread 1 starts the first fused computation on row 5. However, row 4 still needs data propagated from row 3, thus row 4 cannot perform the second computation.
3. In Fig.5.16.c, thread 0 starts first fused computation on row 9. Meanwhile, thread 1 starts the first fused computation on row 6, whenever the data dependency for row 5 is fulfilled, we immediately execute the second collision and streaming on row 5 at time step $t+1$.
4. in Fig.5.16.d, thread 0 starts first fused computation on row 10. For the case without using zero gradient BC, since row 10 only needs data dependency from row 9, at this moment of completion of row 9's first computation, whenever a cell's right neighbor completes the first computation, it can perform the second computation. For example, when cell (10, 2) completes the first computation, the second computation on (10, 1) is ready to do. Meanwhile, thread 1 starts the first fused computation on row 7, whenever



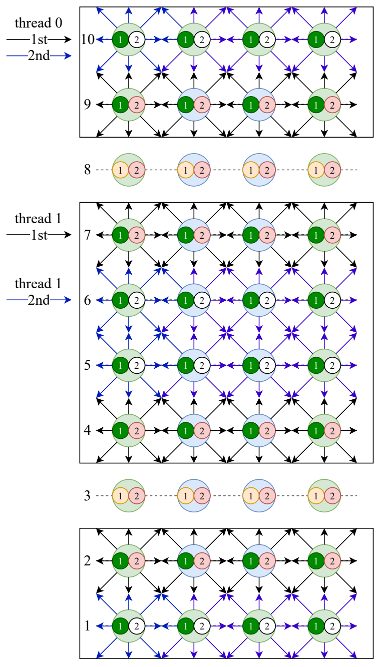
(a) Thread 0 first computation on Line 1; Thread 1 first computation on Line 4.



(b) Thread 0 second computation on Line 1; Thread 1 first computation on Line 4.



(c) Thread 0 first computation on Line 9; Thread 1 second computation on Line 5.



(d) Thread 0 second computation on Line 10; Thread 1 second computation on Line 6.

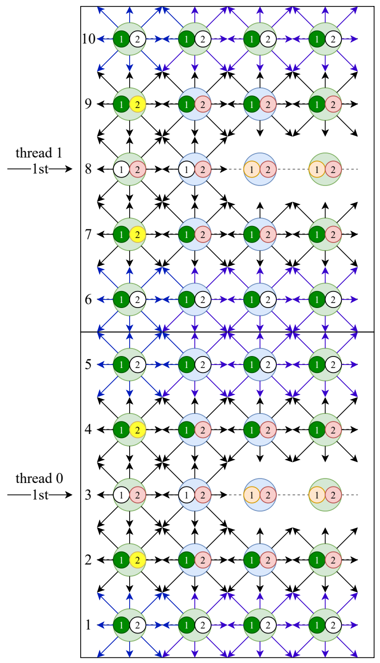
Figure 5.16. Stage I (preparation): handle the intersection.

the data dependency for row 6 is fulfilled, we immediately execute the second collision and streaming on row 6 at time step $t + 1$.

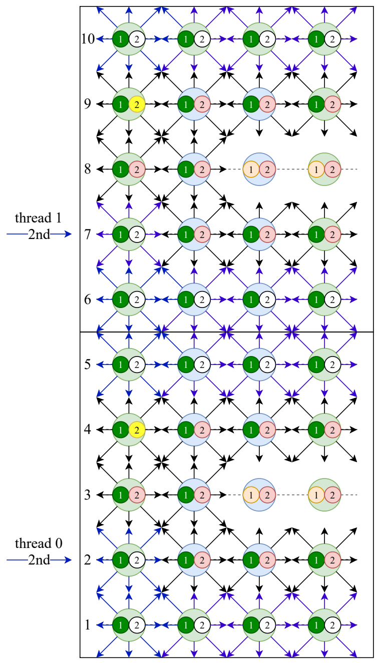
5. At the end of Stage I, a barrier is necessary. Since each thread computes 4 rows, they have the same workload and normally will end at the same time. Thus, the overhead is quite low and we ensure that all threads can start the next stage simultaneously.

Fig.5.17 and 5.17 shows Stage II, which is the main computation in each thread's sub-lattice.

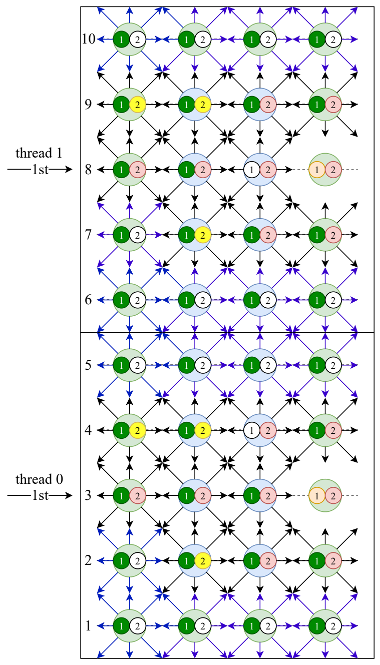
1. In Fig.5.17.a, at time step t, thread 0 computes on cell (3,1) and (3,2) in row 3, while thread 1 computes on cell (8,1) and (8,2) in row 8. They both use $buf1$ to execute the first fused collision and streaming to the $buf2$ of its own and neighbors. Therefore, we change the two cells' $buf1$ from beige to blank, and change the $buf2$ of their own and neighbors to red. Since we have pre-processed row 4 and 9 at time step t during Stage I, after the streaming phase of cell (3,2) and (8,2), cell (2,1) and (7,1) are ready for the second computation at the time step $t + 1$. Thus we change these four $buf2$ to yellow.
2. In Fig.5.17.b, on (3,1) in row 3 and (8,1) in row 8, thread 0 and 1 use $buf2$ to do the second fused collision and streaming to their neighbors' $buf1$ at time step $t + 1$, respectively. Then $buf2$ of (2,1) and (7,1) change to blank. We annotate the second streaming by dark blue streaming arrows, so the $buf1$ of the neighbors of(2,1) and (9,2) change to dark green, which indicates that they are updated by the results from the second computation at time step $t + 1$, but still lack data dependency to compute at the time step $t + 2$.
3. In Fig.5.17.c, thread 0 and 1 use $buf1$ to execute the first fused collision on (3,3) and (8,3) and streaming to their neighbors' $buf2$, so we change these $buf1$ to blank. After the streaming phase of the two cells, the $buf2$ of cell (2,2), (4,2), (7,2), and (9,2) are ready for the second computation. Hence, we change these four $buf2$ to yellow.
4. In Fig.5.17.d, thread 0 and 1 use $buf2$ to execute the second fused collision and streaming on (2,2) and (7,2), respectively. Note that at the end of the second streaming phase



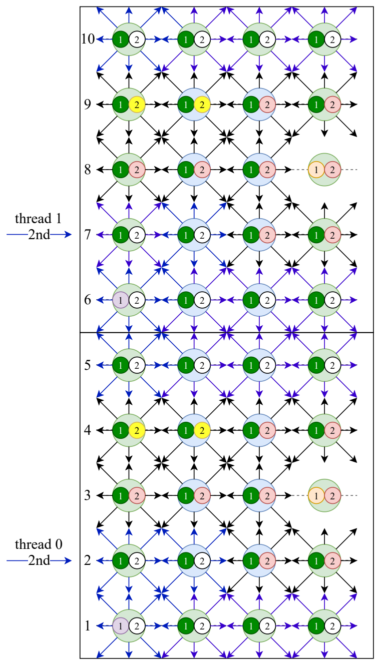
(a) Thread 0 first computation on (3,2); Thread 1 first computation on (8,2).



(b) Thread 0 second computation on (2,1); Thread 1 second computation on (7,1).



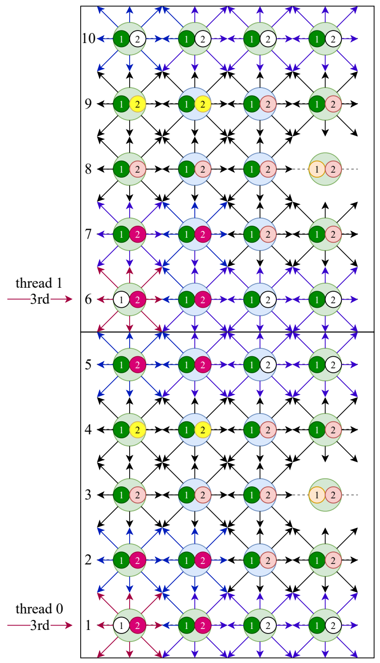
(c) Thread 0 first computation on (3,3); Thread 1 first computation on (8,3).



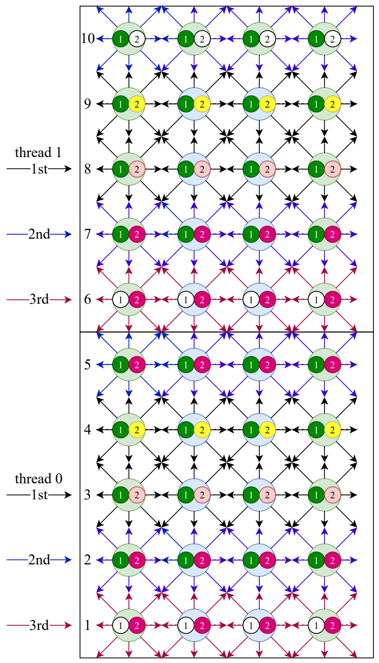
(d) Thread 0 second computation on (2,2); Thread 1 second computation on (7,2).

Figure 5.17. Stage II: main computation in each thread's sub-lattice.

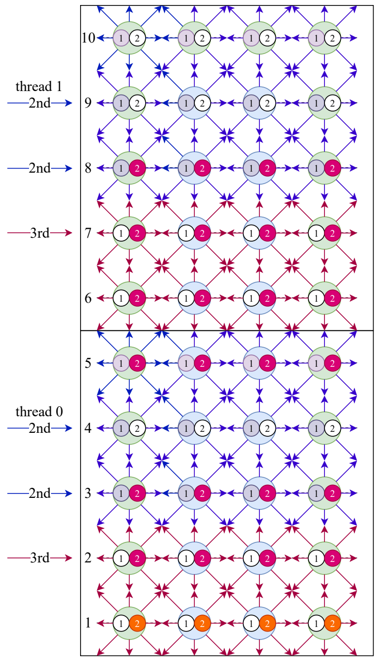
Figure 5.17. Continued.



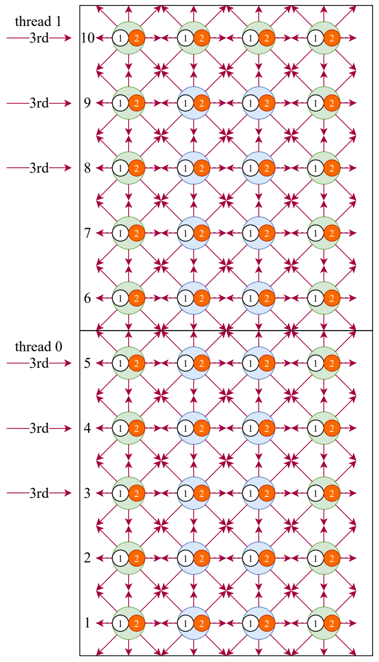
(a) Thread 0 third computation on (1,1); Thread 1 third computation on (6,1).



(b) Thread 0 first computation on (3,4); Thread 1 first computation on (8,4).



(c) Thread 0 second computation on Line 4; Thread 1 second computation on Line 9.



(d) Thread 0 third computation on Line 5; Thread 1 third computation on Line 10.

of cell (2,2) and (7,2), since we have pre-processed the row 1 and 7 at time step $t+1$ during Stage I, $buf1$ of cell (1,1) and (6,1) fulfill the dependency for the third computation at $t+2$. Hence, we change the two $buf1$ to purple.

5. In Fig.5.17.e, thread 0 and 1 use $buf1$ to execute the third fused collision and streaming on (1,1) and (6,1), respectively. Thus, the two $buf1$ change to blank. The third streaming is annotated by pink streaming arrows, so the $buf1$ of the neighbors of (2,1) and (9,2) also change to pink, which indicates that they are updated by the results from the third computation at the time step $t+2$, but still lack data dependency to compute at the time step $t+3$.

6. In Fig.5.17.f, similarly to step 3 to 5, thread 0 and 1 use $buf1$ to execute the first fused collision and streaming on (3,4) and (8,4), respectively. Since the two cells are the last cells in each row, the last two cells in row 2 and 7 have fulfilled the dependency for the second computation. Because of pre-processing in stage I, the last two cells in row 4 and 9 are also ready for the second computation at the time step $t+1$. ii) Next, thread 0 and 1 use $buf2$ to execute the second fused collision and streaming on cell (2,3) and (2,4) in row 2, and cell (7,3) and (7,4) in row 7, respectively. After the second streaming phase, the last three cells in row 1 and 6 are ready for the third computation at the time step $t+2$. iii) At last, thread 0 and 1 use $buf1$ to execute the third fused collision and streaming from (1,2) to (1,4) in row 2, and cell (6,2) to (6,4) in row 6, respectively. In the real problem, each thread's sub-lattice is far larger than the current dimension, step 3 to 6 will repeat and this is where the main computation happens.

7. In Fig.5.17.g, thread 0 and 1 reach row 4 and 9, respectively. i) During stage I, since row 4 and 9 have already been computed by the first fused computation till the time step t, thread 0 and 1 only need to use $buf2$ to perform the second computation on row 4 and 9. ii) Next, thread 0 and 1 use $buf1$ to perform the second computation on row 3 and 7. After the second streaming, the $buf1$ of row 3 to 5 and row 8 to 10 are all ready for the third computation, so these $buf1$ change to purple. iii) At last, thread 0 and 1 use $buf1$ to execute the third computation on row 2 and 7. After the

third streaming phase, the $buf2$ in row 1 are ready for the fourth computation at the time step $t + 3$, so we change them to orange.

8. In Fig.5.17.h, thread 0 and 1 reach row 5 and 10, i.e., the highest row in their own sub-lattice. During Stage I, since row 5 and 10 have already been computed by the second fused computation till the time step $t+1$, we are left to only perform the third computation on these rows and the two rows below. Specifically, thread 0 computes the third computation from row 5 to 3, while thread 1 computes the third computation from row 10 to 8. Finally, the $buf2$ of all cells changes to orange.

Alg.9 presents the parallel k-step memory-aware LBM algorithm. In Fig.5.17.f, when a thread computes the first computation on the last cell in a row, we can execute the second computation on the last two cells in the lower one row, and the third computation on the last three cells in the lower two rows. This operation will involve extra if-branches within the nested loop. But to avoid this overhead, loop unrolling can be used. Thus, there is a Stage III from lines 28∼31 in Alg.9 to compute the remaining lines, which haven't completed the computation at the time step $t + 1$ and $t + 2$.

5.6.3 Special Handling of Boundary Conditions

Except for the zero-gradient BCs, all other BCs are local computation. The boundary cells just call the related BC function, same as the cells which call the BGK collision function.

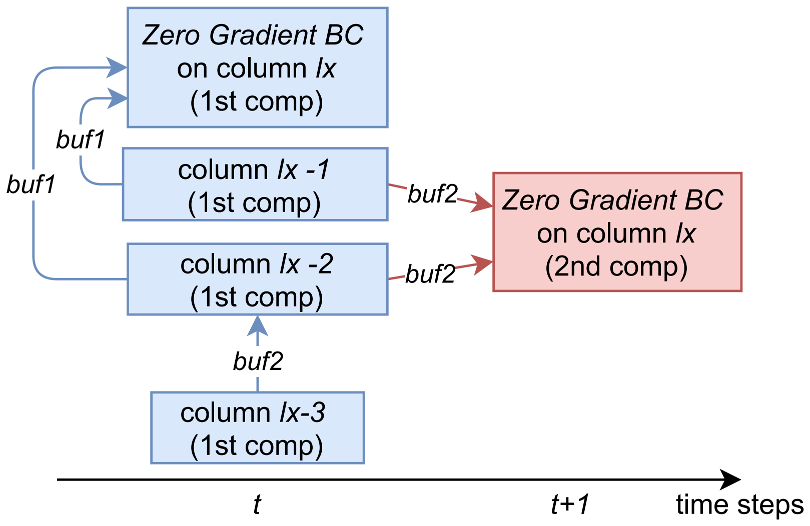
To support the zero-gradient BCs, we make the following changes in Alg.9.

1. On line 8 of Alg.9 in Stage I, thread 0 has to compute the first computation on extra two columns $lx - 2$ and $lx - 3$, so that $buf2$ of $lx - 1$ and $lx - 2$ collects all data dependency for the second computation. Fig.5.18 shows that: a) the first zero gradient BC on column lx is dependent on the $buf1$ from columns $lx - 1$ and $lx - 2$. b) the second zero gradient BC on column lx is dependent on the $buf2$ from columns $lx - 1$ and $lx - 2$. However, the $buf2$ on $lx - 2$ is depend on the $buf2$ from column $lx - 3$ at the time step t, and that's why we need to compute the extra $lx - 3$. Thus line 9 remains unchanged, and thread 0 can use the zero gradient BC to correctly execute the second computation on column lx. Therefore in Stage I, since we have computed

Algorithm 9 2D Parallel k-step Memory-aware LBM with Loop-tiling

```
1: for iT=0; iT < N; iT += 3 do
2:   #pragma omp parallel default(shared) {
3:   // tid: each thread id; sub_len: length of each thread's sub-lattice
4:   mylx[0] = 1 + tid * sub_len; // the lowest row of the sub-lattice
5:   mylx[1] = (tid + 1) * sub_len; // "intersection", i.e., the highest row of the sub-lattice
6:   /* Stage I: preprocess on the "intersection": */
7:   if tid == 0 then
8:     First fused collision and streaming on column 1, 2, $lx - 1$ & $lx$ using $buf1$.
9:     Second fused collision and streaming on column 1 & $lx$ using $buf2$.
10:  else
11:    First fused collision and streaming on column $mylx[0] - 2$ to $mylx[0] + 1$ using $buf1$.
12:    Second fused collision and streaming on column $mylx[0]$ to $mylx[0] - 1$ using $buf2$.
13:  #pragma omp barrier
14:  // Stage II: Main computation in each thread's sub-lattice:
15:  for outerX = mylx[0]+2; outerX ≤ $mylx[1]$; outerX += $tile$ do
16:    for outerY = 1; outerY ≤ $ly$; outerY += $tile$ do
17:      for innerX=outerX; innerX ≤ MIN(outerX + $tile$ - 1, $mylx[1]$); ++innerX do
18:        for innerY = outerY; innerY ≤ MIN(outerY + $tile$ - 1, ly); ++innerY do
19:          // First fused collision and streaming at the time step $t$:
20:          if innerX != mylx[1] && innerX != mylx[1] - 1 then
21:            Collide on (innerX, innerY)'s buf1 & stream to its own and 8 neighbors' buf2
22:          // Second fused collision and streaming at the time step $t + 1$:
23:          if innerY > 1 then
24:            Collide on (innerX-1, innerY-1)'s buf2 & stream to its own and 8 neighbors' buf1
25:            // Third fused collision and streaming at the time step $t + 2$:
26:            if innerY > 2 then
27:              Collide on (innerX-2, innerY-2)'s buf1 & stream to its own and 8 neighbors' buf2
28:  // Stage III: compute the remaining incomplete cell and handle BCs
29:  Second fused collision and streaming on row $ly$.
30:  Third fused collision and streaming on column mylx[1] - 1, then on the "intersection" mylx[1], mean-
     while handle BCs.
31:  Third fused collision and streaming on row $ly - 1$, then on row $ly$.
32:  }
33:  Swap $buf1$ and $buf2$
```

lx in advance till the time step $t+1$, we no longer need the extra two arrays to store the density from the column $lx-1$ and $lx-2$ afterwards.



Zero Gradient BC on column *lx* (1st comp)
buf1
buf1
column *lx -1* (1st comp)
buf2
Zero Gradient BC on column *lx* (2nd comp)
column *lx -2* (1st comp)
buf2
buf2
column *lx-3* (1st comp)
t
t+1
time steps

Figure 5.18. Zero gradient BC data dependency in k-step memory-aware LBM (k=3).

2. On line 20 in Stage II, we need to change the if-branch accordingly as follows:

$$\textbf{If}\ innerX\ != \ mylx[1]\ \ \&\&\ \ innerX\ != \ mylx[1]-1$$
$$\&\&\ \ innerX\ != \ lx-2\ \ \&\&\ \ innerX\ != \ lx-3$$

   This is because we have computed the first computation on all these columns in Stage I to avoid computation twice at the time step t.

3. Stage III remains unchanged, and we just follow line 31 in Alg.9. Now, thread $n-1$ uses the zero gradient BC to compute the third computation on column $lx-1$, then on column lx.

5.7 Analysis of the 2D LBM Algorithms

In this section, we analyze the algorithms by counting how many data reuses are in the Original, Fuse, two-step, k-step memory-aware LBM.

5.7.1 Data Reuse in Original LBM & Fuse LBM

Fig.5.19 shows the times of data reuses in the Original LBM on a 3×4 lattice. We assume that a core's private cache can hold all the data.[4] In each time step, the Original LBM sweeps the lattice twice. During the first sweep, since each cell only uses its *buf1* to collide and does not access neighbors' data, there is no data reuse. However, the second streaming sweep has 8 data reuse per cell, as shown in the grey region. For example, when the cell $(2, 2)$ starts streaming to the red $buf2$ of its own and its neighbors, except for (3,3), all the other $buf2$ in its neighbors have just been streamed. Cell (2,1) has streamed to $buf2$ of (3,1) and (3,2) in row 3, (2,1) and (2,2) in row 2, and (1,1) and (1,2) in row 1. Cell (1,2) has streamed to $buf2$ of (2,3). Thus, 7 neighbors and its own $buf2$ are in the cache. Therefore, we count 8 $buf2$ reuse in the Original LBM.

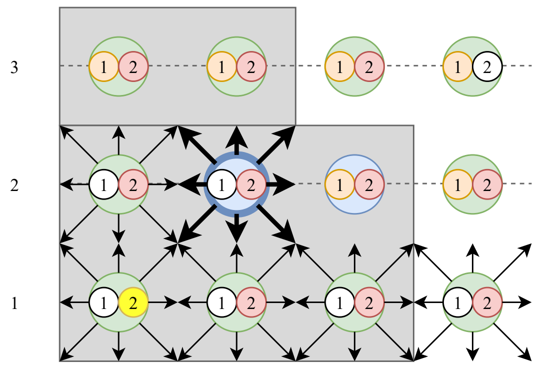


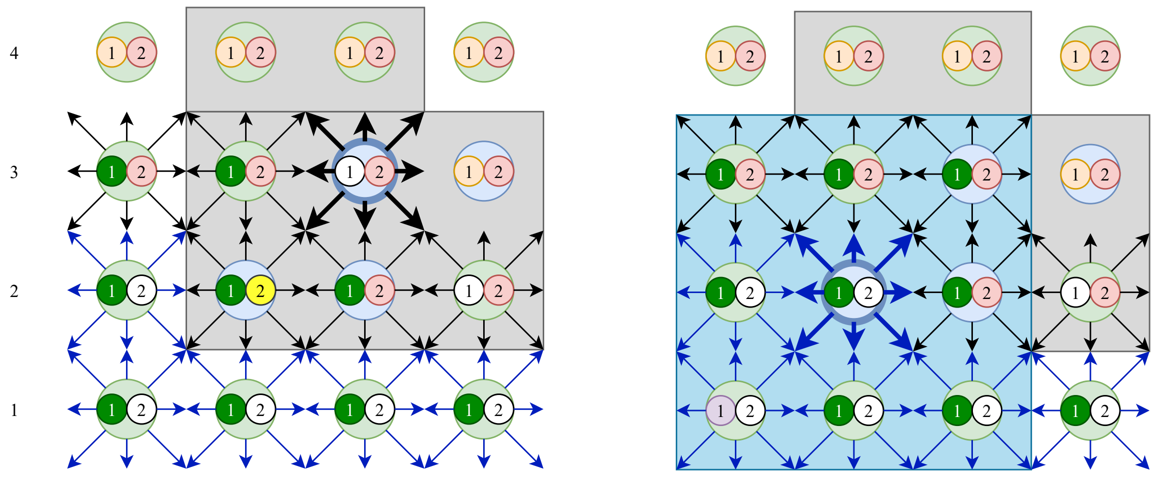
Figure 5.19. The Original LBM has 8 $buf2$ reuses per cell (in the grey region) during the streaming stage, while the Fuse LBM has 9 data reuses per cell (including 1 $buf1$ reuse).

Since the Fuse LBM uses loop fusion, it uses a cell's $buf1$ to execute the collision, then immediately propagates the cell's *buf1* to its neighbors. Hence, a cell's *buf1* is reused during the streaming phase. The streaming phase has the same reuse as the Original LBM. Then there are $1 + 8 = 9$ data reuses per cell in the Fuse LBM.

[4]↑Each cell's buf is (9 * double + 1 pointer)* 8B = 80B, and each cell has two buffers. Data moves around most of the cache hierarchy are 64-byte cache lines. Thus, using $buf\#$ to compute the collision on a cell requires at least two cache line loads if it is not in the cache.

5.7.2 Data Reuse in k-step Memory-aware LBM

Fig.5.20 shows the times of data reuse in the two-step memory-aware algorithm on a 4×4 lattice. We still assume that a core's private cache can hold all the data here. Same as the Fuse LBM, in Fig.5.20a, when cell $(3, 3)$ use $buf1$ to execute the first computation, there are 9 data reuses in the grey region. Next, in Fig5.20b, cell (2,2) uses $buf2$ to execute the second fused collision and streaming to the $buf1$ of its own and neighbors. Hence, its *buf2* is reused in the second collision, while 9 *buf1* in the blue region are also reused during the second streaming. Therefore, there are $9 + 1 + 9 = 19$ reuses per two cells in every two time steps.

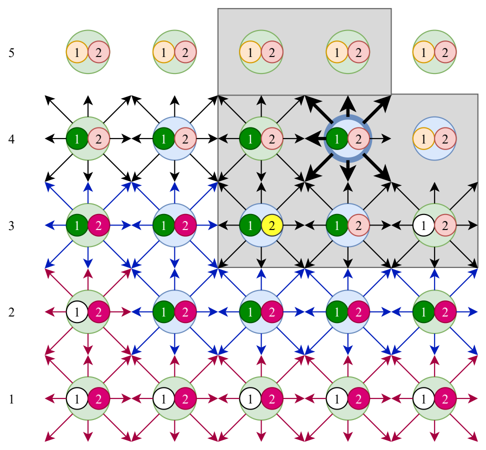


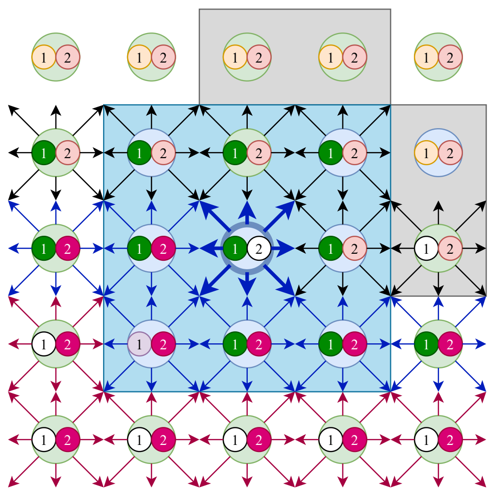
(a) First computation on (3,3).

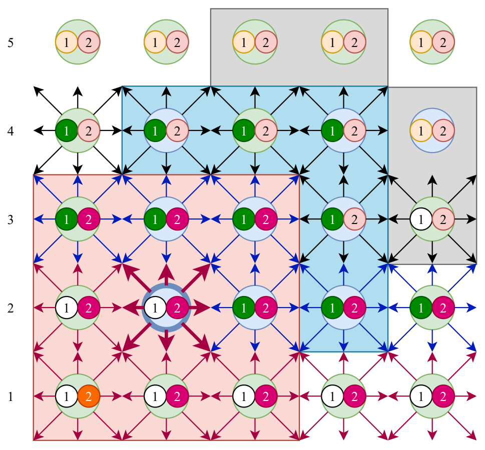
(b) Second computation on (2,2).

Figure 5.20. The two-step memory-aware LBM has 19 data reuses per two cells in two time steps.

Fig.5.21 shows the times of data reuse in the k-step memory-aware algorithm (k=3) on a $\times$ lattice. Same as the Fuse LBM, in Fig.5.21a, when cell $(4, 4)$ use $buf1$ to execute the first fused collision and streaming, there are 9 data reuses in the grey region. Next, in Fig5.21b, cell (3,3) uses $buf2$ to execute the second fused collision and streaming to the $buf1$ of its own and neighbors. Hence, its *buf2* is reused in the second collision, while 9 *buf1* in the blue region are also reused during the second streaming. At last, cell (2,2) uses $buf1$ to execute the second fused collision and streaming to the $buf2$ of its own and neighbors. So its *buf1* is reused in the third collision, while 9 *buf2* in the pink region are also reused during the


(a) First computation on (4,4).


(b) Second computation on (3,3).


(c) Third computation on (2,2).]

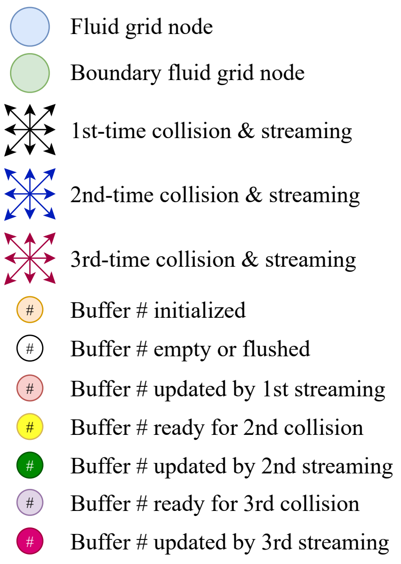

Fluid grid node
Boundary fluid grid node
1st-time collision & streaming
2nd-time collision & streaming
3rd-time collision & streaming
Buffer # initialized
Buffer # empty or flushed
Buffer # updated by 1st streaming
Buffer # ready for 2nd collision
Buffer # updated by 2nd streaming
Buffer # ready for 3rd collision
Buffer # updated by 3rd streaming
(d) Legend used in (a)∼(c).

Figure 5.21. The three-step memory-aware LBM has 29 per three cells in three time steps.

second streaming. Therefore, there are $9 + (1 + 9) * 2 = 29$ reuses per three cells in every three time steps.

5.8 Experimental Evaluation

In this section, we first evaluate the sequential and parallel performance of the seven LBM algorithms, namely Original LBM, Fuse LBM (with/without tile), two-step memory-aware LBM (with/without tile), and three-step memory-aware LBM (with/without tile) on three Intel CPU architectures deployed in two supercomputers: Haswell on *Bridges*, Skylake and Knight Landing on *Stampede2*. Secondly, we visualize and validate the results using Paraview and Catalyst.

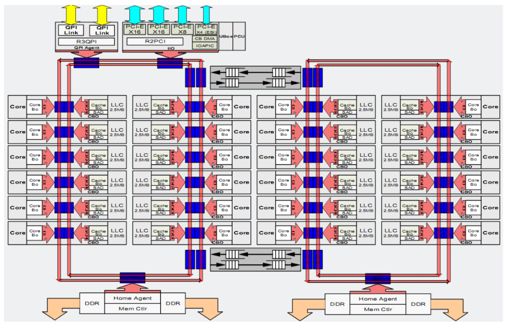
The *Bridges* HPC system in the Pittsburgh Supercomputer Center (PSC) has 752 regular nodes ("RM"). Each node has 28 Haswell physical cores on 2 sockets. The *Stampede2* HPC system in the Texas Advanced Computing Center (TACC) has 1,736 Skylake nodes (SKX) and 4,200 Knight Landing nodes (KNL). Each SKX node has a total of 48 Skylake physical cores on 2 sockets, while each KNL node has a total of 68 physical cores on a single socket. More details about the HPC system and the CPU architectures used in our experiments are given in Tab.5.2.

Table 5.2. Details of the experimental platforms.

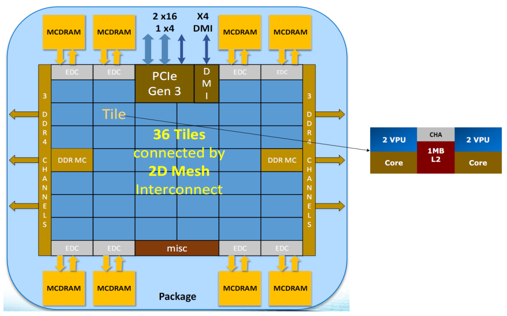
HPC System	*PSC Bridges*	*TACC Stampede2*	
Microarchitecture	*Haswell'14*	*Skylake'17*	*Knight Landing'16*
Intel CPU product code	Xeon E5-2695v3	Xeon Platinum 8160	Xeon Phi 7250
Total #Cores/node	28 on 2 sockets	48 on 2 sockets	68 on a single socket
Clock rate (GHz)	2.1∼3.3	2.1 nominal (1.4∼3.7)	1.4
L1 cache/core	32KB	32KB	32KB
L2 cache/core	256KB	1MB	1MB per two-core tile
L3 cache/socket	35MB	33MB (Non-inclusive)	16GB MCDRAM
DDR4 Memory(GB)/node	128 (2133MHz)	192 (2166MHz)	96 (2166MHz)
Compiler	icc/19.5	icc/18.0.2	
AVX extension	AVX2	AVX512	

The Haswell microarchitecture released in 2014 uses the two red ring buses in Fig.5.22a to connect the L2 caches to portions of the L3 cache, as well as QPI, PCIe links, and the home agents for the memory controllers. The Knight Landing (Xeon Phi) processor released in 2016 uses a 2D mesh in Fig.5.22b. It does not have an L3 cache, but has a Multi-Channel DRAM (MCDRAM) instead. The 2D mesh is composed of tiles, which includes two cores, 2 Vector Processing Unit (VPU, supporting AVX512), and a shared L2 cache per tile. The Skylake (Xeon Platinum) processor released in 2017 also uses a 2D mesh in

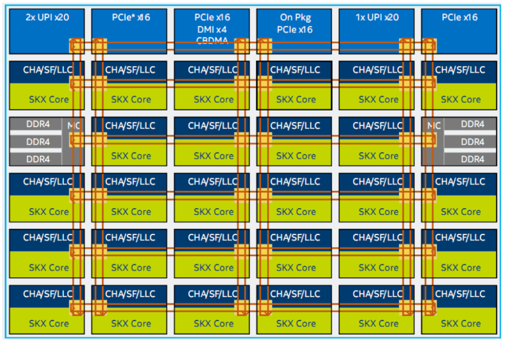
Fig.5.22c. The major improvements are larger private L2 cache, non-inclusive L3 cache and Snoop Filter (SF), and QPI updated to Ultra Path Interconnect (UPI). Prior architectures use the "Inclusive" L3, which has copies of all lines in L2. But in the new "Non-inclusive" L3 (Skylake only), lines in L2 may not exist in L3. The new design results in reducing SKX local memory latency significantly, and multi-threaded workloads can operate on larger data per thread (due to increased L2 size) and reduce interconnect and L3 activity.



(a) Haswell (2014) has two bidirectional rings. [136]



(b) Knight Landing (2016) has a 2D mesh of cores. [137]



(c) Skylake (2017) has a 2D mesh of cores. [138]

Figure 5.22. On-chip Interconnect of CPUs used in experiments.

All the LBM algorithms are compiled with "`-O3 -AVX2`" flag on *Bridges*, and "`-O3 -xCORE-AVX512 -MIC-AVX512`" on *Stampede2* and to enable vectorization and AVX instructions. The compiler optimization has resulted in significant speedup in the local computation (e.g. BGK collision) due to using vector instructions. All implementations use double precision. Besides, we use MFLUPS (millions of fluid lattice node updates per second), which is a widely-used metric in CFD community, to evaluate the performance of LBM algorithms.

5.8.1 Experiment Setup

We choose to simulate a classic 2D CFD benchmark, namely the steady Poiseuille fluid flowing past a circular cylinder as shown in Fig.5.23. As time goes by, the flow will eventually become unstable and generate vortexes. The horizontal direction (X-axis) is the channel's length lx, while the vertical direction (Y-axis) is the channel's width ly. Thus, the left bottom corner is (1,1), while the right top corner is (lx, ly). We use "row" to represent the horizontal direction in Fig.5.23, e.g., "row ly" is the top line (1,ly) → (lx,ly), and use "column" to represent the vertical direction, e.g., "column lx" is the rightmost line (lx,1) → (lx, ly). The particles on the cylinder has bounce-back boundary conditions (BC) and the four boundaries of the channel have regularized BCs, as discussed more in detail in Sect. 5.5.2.

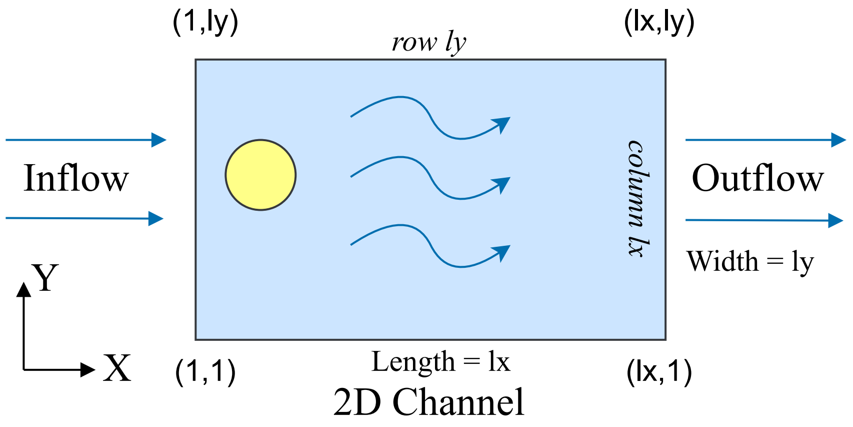
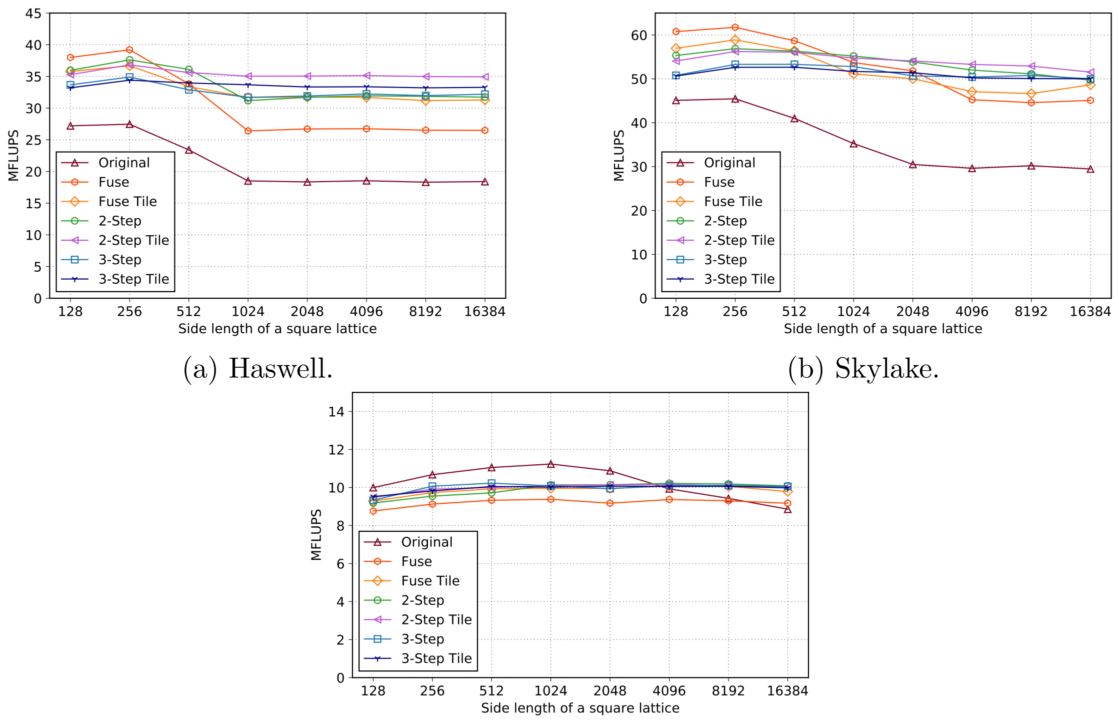


Figure 5.23. A steady Poiseuille fluid flowing around a cylinder in a 2D channel.

5.8.2 Sequential Experiments and Performance Analysis

The first experiment intends to compare the sequential performance of the seven LBM algorithms. The experiment uses a single core, and takes the input of square lattice ranging with side length $L = 128 \sim 16384$ on 3 different CPUs. Each algorithm runs by five times, and we calculate the average MFLUPS. To get accurate results, the same time steps to warm up the machine for each algorithm is given before measurement. For the "tile" version algorithms, we test with different tile size (ranging from 8, 16, 32, ..., to 4096) to find the best tile which gives the best performance and presents in the results.



(a) Haswell.

(b) Skylake.

(c) Knight Landing.

Figure 5.24. Sequential performance using seven LBM algorithms on three Intel CPUs.

Fig.5.24 shows the sequential performance of the seven LBM algorithms on the Intel Haswell, Knight Landing, and Skylake CPU, respectively. Let's start with the Haswell CPU in Fig.5.24a. When the lattice is small, the Fuse LBM is the fastest, and is up to 44% faster than the Original LBM on 128×128 and 256×256. This is because the memory of the whole small lattice can fit into a Haswell CPU's L3 cache (35MB per socket). Tab.5.3 presents the memory allocation size for each square lattice in the sequential experiments. Thereby, memory-aware LBM won't help much.

Table 5.3. Memory allocation size for each square lattice in the sequential experiments.

Side length	128	256	512	1024	2048	4096	8192	16384
Memory(MB)	2.5	10	40	160	640	2056	10240	40960

Besides, the Fuse LBM doesn't contain the if-branch within the loop to break the instruction pipeline, so that it achieves more unite strides. But when the length of a square lattice is 512 and larger, the L3 cache cannot hold the whole lattice. So the Fuse LBM drops 16% from the length increase from length 256 to 512, and drops 28% from length 512 to 1024. This suggests that spatial and temporal tiling are necessary, if seeking to increase more cache locality and data reuse. As the lattice grows larger and larger than L3 can hold, more DRAM accesses involves. We find that the 2-step-tile LBM performs the best, achieves up to 35.3 MFLUPS, and is up to 32.7% faster than the Fuse LBM when $L = 1024$. When $L = 16384$, the 2-step-tile LBM acquires 34.95 MFLUPS and is 31.9% faster than the fused LBM, while the second-fastest 3-step-tile LBM achieves 33.3 MFLUPS and is 25.6% faster than the fused LBM. And I will explain the reason why the 3-step LBM is 4.75% faster than 2-step-tile LBM later.

Secondly, Fig.5.24b shows that Skylake gives similar ranking results as Haswell since they both have an on-chip L3 cache. Fuse LBM is still the fastest with $L \leq 512$ and up to 43% faster than the Original LBM. Then we see the same phenomenon again that when $L > 512$, which exceeds the Skylake's 33 MB L3 cache, the Fuse LBM starts to slump. As the lattice grows even larger, the 2-step-tile LBM again performs the best, and achieves up to 54.7 MFLUPS and up to 18.7% faster than the Fuse LBM when $L = 8192$. When $L = 16384$, the 2-step-tile memory-aware LBM achieves 51.5 MFLUPS and is 14.3% faster than the fused LBM, while the second-fastest 3-step-tile LBM achieves 50.0 MFLUPS and is 10.9% faster than the fused LBM. Besides, we also find that for all the algorithms, performance on Skylake is better than Haswell, although the frequency of Skylake and Haswell are both around 2.1GHz. [5] For example, the 2-step-tile LBM on Skylake gives an average of 52.5% better performance than on Haswell. This is because Skylake's larger L2 cache helps to increase L2 hit, and non-inclusive L3 helps to reduce DRAM memory latency.

At last, on Knight Landing CPU in Fig.5.24c, we see that the Original LBM is the fastest till $L = 4096$ lattice. But as lattice grows larger than that, the 2-step LBM without tiling

[5]↑The actual clock of the SKX CPU depends on the vector instruction set, the number of active cores, and other factors affecting power and temperature limits. A single core serial code using the AVX2 instruction set may run at 3.7GHz, while a large, fully-threaded MKL dgemm may run at 1.4GHz.

is the best and can be up to 10.5% faster than the Fuse LBM. When $L = 16384$, the 2-step LBM achieves 10.08 MFLUPS and is 9.6% faster than the fused LBM, while the second-fastest 3-step no-tile LBM achieves 10.05 MFLUPS and is 8.83% faster than the fused LBM. The reason why on KNL 2-step-tile and 3-step-tile LBM don't gain the same speedup as Haswell and Skylake is that the LLC on KNL is 1 MB L2 cache and no L3 cache exists. Thus even $L = 128$ lattice cannot fit into it, which results in fewest data reuse and the most DRAM memory accesses.

Above all, considering that LBM simulation usually runs on a large scale, on three different CPUs, we can still claim that the k-step memory-aware LBM will outperform other methods in most use cases.

Performance Analysis of Sequential LBM Experiments

To investigate the reason why the performance of the seven algorithms differ, the "Call-stacks Roofline" can be again used to measure the total performance and total AI of the seven sequential LBM algorithms. I use the same input and configuration as in the Sec.5.3, and choose the same following metrics: to measure the (INT+FLOAT) operations of DRAM data transfer on a 1024×1024 square lattice running 30 time steps on a Haswell node. Since the 33MB LLC cache cannot hold the 160MB memory allocation size of the whole lattice, we can evaluate how each algorithm performs on the DRAM memory access.

Fig.5.25 shows the DRAM Roofline comparison for the 7 sequential LBM algorithms. The point information of each algorithm in the figure are in Tab.5.4. Firstly, we see that only the Original LBM is below the scalar double floating add peak (3.28 GFLOPS), while the other six are above it, but all of them are under the integer scalar add peak (8.06 GINTOPS). Secondly, the machine balance between DRAM and integer scalar add peak is 0.84 OP/Byte. We use it to decide a kernel belonging to memory-bound or compute-bound. Thus, the Original, Fuse, and Fuse tile LBMs are on the left of the machine boundary (0.84 OP/Byte), so they are memory bounded by DRAM. Fuse LBM improves the AI of the Original LBM by 1.22X, and the Fuse tile LBM has the largest AI and GOPs among the three. However, the Fuse tile LBM almost sits on the DRAM roof, suggesting that it is

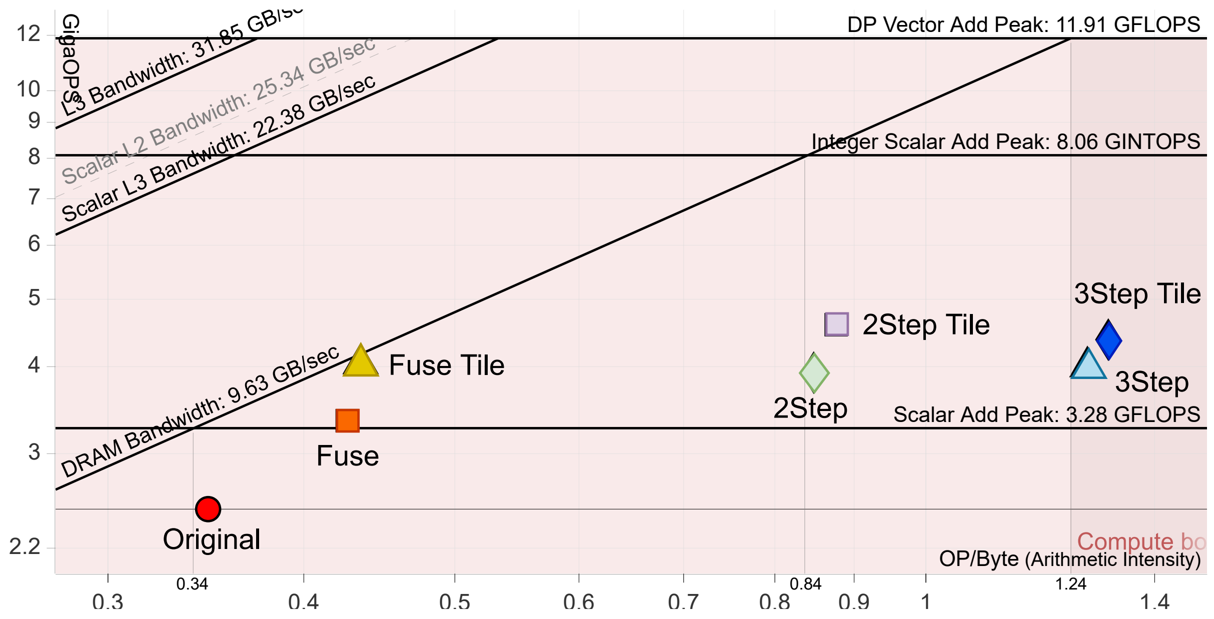


Figure 5.25. DRAM roofline comparison for the 7 sequential LBM algorithms with $L = 1024$ on a Haswell node.

impossible to use vectorization and threading on this kernel to vertically move it to higher GOPs, since there is no more room between its position and the DRAM roof, thus only changing the algorithm to get higher AI in order to move right can help. Secondly, we see that 2-step LBM improves AI by 1.98X compared to the Fuse LBM, indicating that the 2-step LBM executes twice the number of the (INT+FLOAT) operations for every byte transferred from DRAM to L3 cache. Besides, 2-step LBM's kernel is on the right side of the machine boundary and stays in the compute-bound region. The 2-step-tile LBM takes advantage of spatial tiling and has the best GOPS and MFLUPS among the seven algorithms. At last, the 3-step LBM moves its kernel further right and even enters into the compute-bound region of the roofs using DRAM and DP (double floating-point) vector add peak (11.91 GFLOPS). Its AI is 3X of the Fuse LBM's AI, also meaning that it has 3X data reuse. The GOPS and MFLUPS of 3-step-tile LBM is only 4.6% less than 2-step-tile, and ranks the second in the sequential experiment. Thus since both 2-step-tile and 3-step-tile LBMs are in the compute-bound region, we can increase parallelism to improve their performance.

Next, I use the Memory Access Pattern (MAP) tool of Intel Advisor to analyze the collision-streaming loop of each LBM algorithm. Tab.5.5 presents the stride distribution of

Table 5.4. Point information of each algorithm on the Roofline Fig.5.25.

Algorithm	Original	Fuse	Fuse Tile	2Step	2Step Tile	3Step	3Step Tile
DRAM AI (OP/Byte)	0.35	0.43	0.44	0.85	0.88	1.27	1.31
Giga OPs	2.5	3.36	4.05	3.93	4.60	4.01	4.38
MFLUPS	18.6	26.4	31.8	31.2	35.3	31.7	33.8
Speedup	baseline	1.42X	1.71X	1.68X	1.9X	1.71X	1.82X

OP = INT + FLOAT operations.

Table 5.5. Memory Access pattern analysis of the innermost loop within each sequential algorithm on a 1024×1024 square lattice on a Bridges Haswell node.

Algorithm	**Original**		**Fuse**	**Fuse Tile**	**2Step**	**2Step Tile**	**3Step**	**3Step Tile**
	Collide	Stream						
Unit Stride%	62%	0%	61%	63%	59%	63%	62%	64%
Constant Stride%	36%	100%	38%	37%	32%	37%	38%	36%
Irregular Stride%	2%	0%	2%	0%	2%	0%	0%	0%
Mem. addr. range	80KB		80KB	20KB	80KB	488B	80KB	20KB

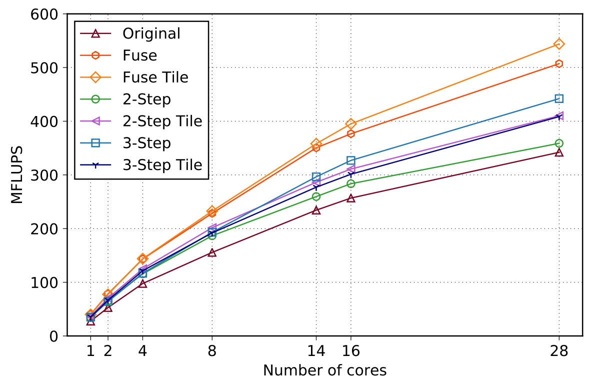
each kernel, and their memory address range. If a kernel has more unit stride and small memory footprint, it will have effective SIMD instructions, and no latency or bandwidth bottlenecks, which is the ideal case. Firstly, a kernel's stride distribution shows portions of different types of memory access strides during its loop execution. The tool reports unit (stride 0/1), constant (stride N), and irregular (variable/random) stride accesses. More unit stride percentage has better effective SIMD performance. Since the Original LBM sweeps the whole lattice twice in each time step, it is separated by the *collide* and *stream* sweep. We see that the 3-step-tile LBM has the highest 64% unit stride and 36% constant stride. The 2-step-tile LBM ranks the second place, and has 63% unit stride and 37% constant stride. Secondly, the memory address range represents the maximum distance between minimum and maximum memory address values accessed by instructions in this loop, leading to the memory footprint. For the algorithms without tiling, they all have 80 KB address range. For the Fuse tile LBM and 3-step-tile LBM, they both reach their best performance with

$tile = 256$, and both access in the 20KB address range. The 2-step-tile LBM gives the best result when $tile = 8$, and its innermost loop only accesses 488 B address range. Above all, with more unit strides and the smallest memory footprint, the sequential 2-step-tile LBM is 4.75% better 3-step-tile LBM, but the two algorithms are far better than the other five algorithms. The reason why the 2-step-tile and 3-step-tile LBM is not 2X or 3X faster than the Fuse tile LBM in terms of MFLUPS is that the BGK collision computation has spent 85.4% and 83.9% of their total time in the sequential experiment, respectively. According to Amdahl's law, the theoretical speedup is always limited by the part of the task that cannot benefit from the improvement. Our optimization on the memory access pattern spatially and temporarily can improve a kernel's AI, but since k-step memory-aware LBM is now compute-bound, our future work is to improve the performance of the collision kernel by using vector instructions.

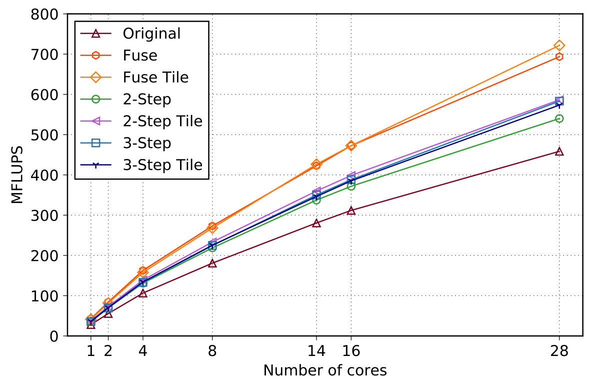
5.8.3 Strong Scalability and Performance Analysis

Our second experiment evaluates the strong scalability performance of the seven parallel LBM algorithms with the edge size of 2D lattice $L = 112 \sim 14336$ on a *Bridges Haswell node*, , $L = 192 \sim 24576$ on a *Stampede2 Skylake node* and $L = 272 \sim 17408$ on a *Stampede2 KNL node*. The parallel version of original, Fuse, and Fuse tile LBM distribute the whole lattice to n threads according to the channel's length direction (X-axis), just as the parallel k-step algorithms, so that we have a fair comparison among them. The number of threads n is picked from 1 to the maximum total number of the physical cores on each node of a system, namely 28 threads on Haswell, 48 threads on Skylake, and 68 threads on KNL. The OpenMP thread affinity sets "spread" to bind each thread to a physical core, meanwhile evenly distributed on the two sockets of a node. The result of every algorithm is an average of five times execution on each lattice size. For the tile LBM implementations, $tile$ ranges from 8 to 512 and then we pick the fastest achieved result.

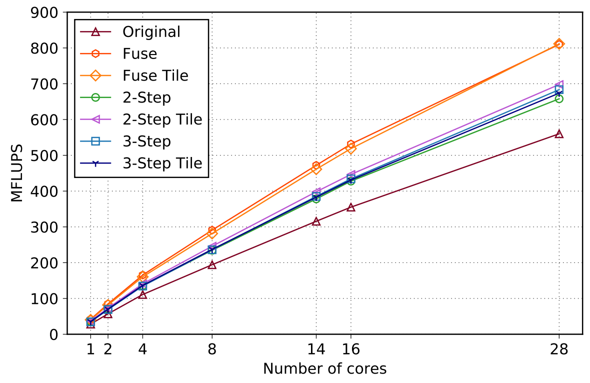
Fig.5.26 and Fig.5.26 shows the strong scalability of the 7 LBM algorithms on a Haswell node of *Bridges* with the edge size of 2D lattice $L = 112 \sim 14336$. Besides, the memory consumption of each lattice size is in Tab.5.7, and the memory allocation of the largest



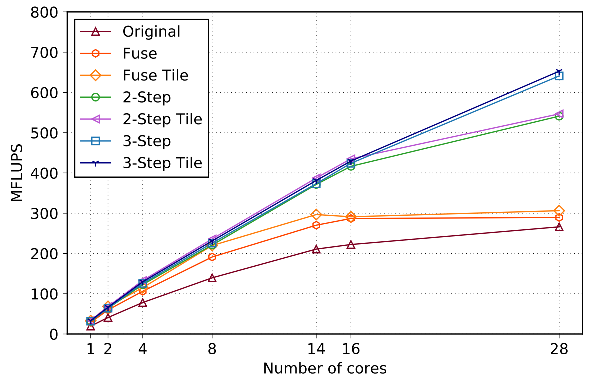
(a) $L = 112$.



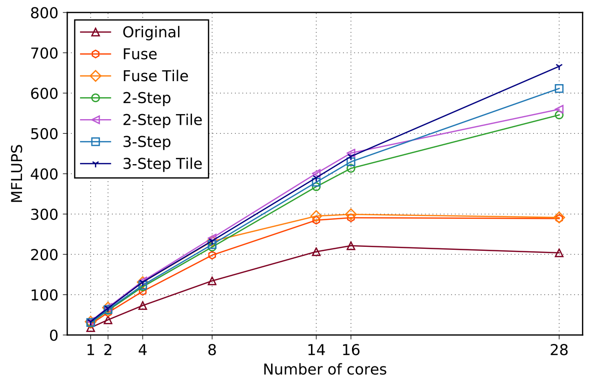
(b) $L = 224$.



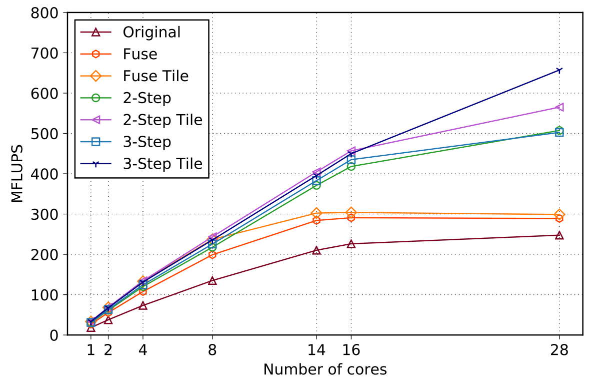
(c) $L = 448$.



(d) $L = 896$.



(e) $L = 1792$.

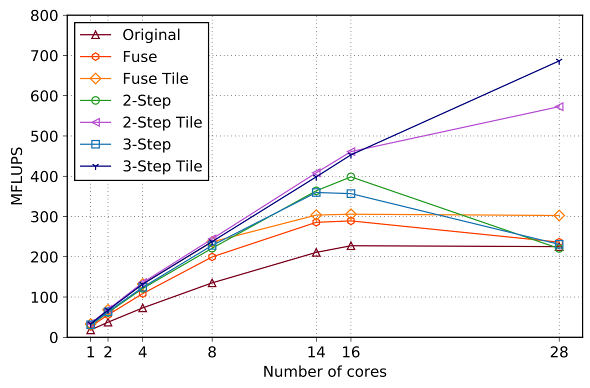


(f) $L = 3584$.

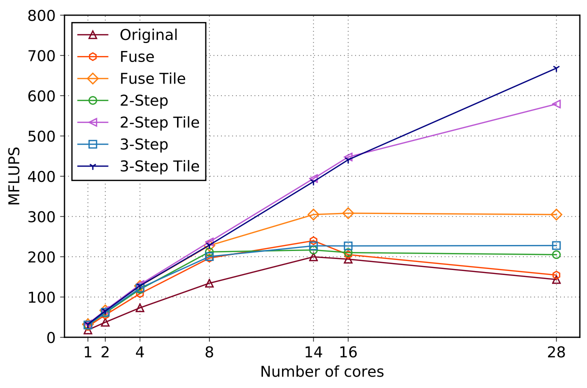
Figure 5.26. Haswell Strong Scalability performance.

$L = 14336$ lattice in this group of experiment nearly reaches the max memory 128 GB of each RM node in *Bridges*. Firstly, Fig.5.26a to Fig.5.26c shows that when $L \leq 448$, the memory size of the whole lattice fits into the LLC ($35MB/socket \times 2sockets/node = 70MB/node$), the Fuse tile LBM gives the best parallel performance. However, when the lattice grows

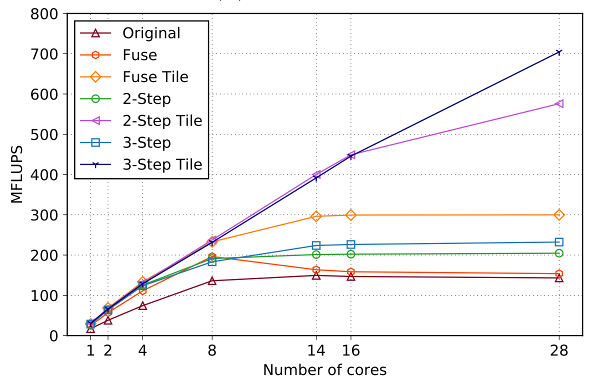
Figure 5.26. Continued.



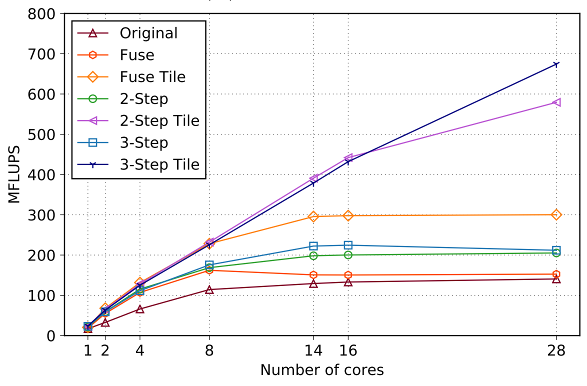
(a) $L = 7168$.



(b) $L = 14336$.



(c) $L = 20720$.



(d) $L = 27104$.

larger, its performance slumps a lot and doesn't scale well, because each core in the Fuse tile LBM sits the DRAM bound, analyzed in Sect.19. Secondly, among the algorithms without tiling when $L \geq 896$, 3-step LBM generally ranks the first by up to 18.5% faster than the second place 2-step LBM, Fuse LBM ranks the third, and the original LBM is the baseline. The performance of k-step LBM using 28 cores drops significantly and is worse than the Fuse tile LBM when the side length is larger than 7168. This indicates that tiling is necessary to reduce the socket level data transfer (e.g., SNOOP request, QPI and etc.) and explore more data reuse especially as the number of threads increases to all the cores on a node. Thirdly, when $L \geq 896$, the 3-step-tile LBM performs the best with up to 704.5 MFLUPS and achieves the nearly ideal linear strong scalability. Using 28 threads, it is up to 22.3%

Table 5.6. Memory consumption for each 2D lattice in the parallel experiments on a *Bridges* Haswell node.

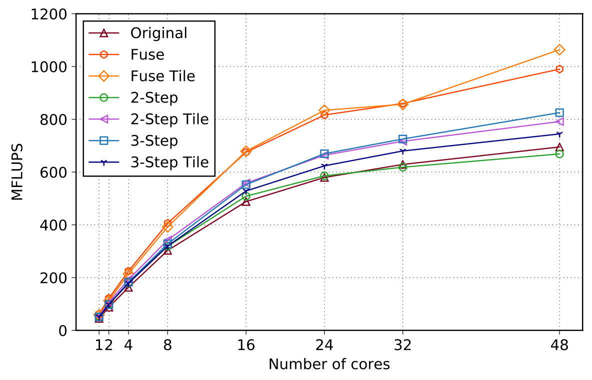
Side length L	112	224	448	896	1792	3584	7168	14336	20720	27104
Memory(B)	1.9M	7.7M	30.6M	122.5M	490M	1.9G	7.7G	30.6G	64.0G	109.5G

faster 2-step-tile LBM, 135% faster than Fuse tile LBM, and 358% faster than Fuse LBM. On the other hand, the 2-step-tile ranks the second when using 28 threads, with up to 93% faster than Fuse tile LBM, and 280% faster than Fuse LBM. The result that the 2-step-tile outperforms the Fuse-tile by up to 1.93X strongly claims that merging two steps of computation significantly improves the actual performance and the data reuse by nearly twice. The 3-step-tile LBM gets up to 2.35X but not nearly 3X faster than the Fuse tile LBM, suggesting that the local collision computation has become the new bottleneck and the whole algorithm is compute-bounded, which will later be shown in the Roofline graph.

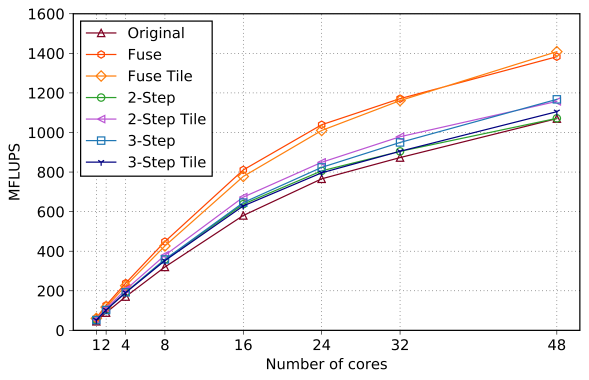
Table 5.7. Memory consumption for each 2D lattice in the parallel experiments on a *Stampede*2 SKX node.

Side length	192	384	786	1536	3072	6144	12288	24576	28800	33600
Memory(B)	1.9M	22.5M	90M	360M	1.4G	5.6G	22.5G	90G	123.6G	168.2G

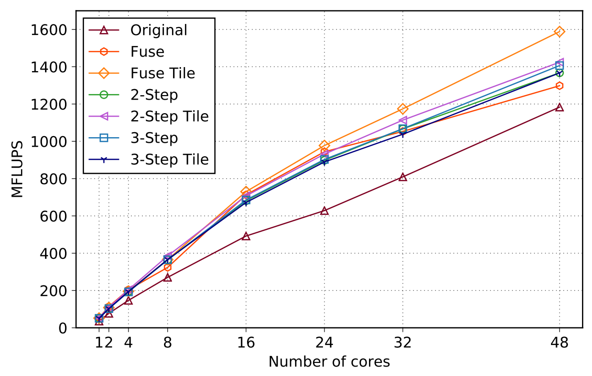
Fig.5.27 and Fig.5.27 shows the strong scalability of the 7 LBM algorithms on a Skylake node of *Stampede2* with edge size $L = 192 \sim 33600$. Tab.5.7 presents the memory allocation size of each lattice. Firstly, Fig.5.27a to Fig.5.27c shows that when $L \leq 768$, the Fuse tile LBM is the best, since at least half of the memory consumption can fit into LLC size ($33MB/socket \times 2sockets/node = 66MB/node$). However, when the lattice grows larger, Fuse tile slumps a lot again due to the DRAM bound. Secondly, among the algorithms without tiling when $L \geq 1536$, 3-step LBM generally ranks the first by up to 13.5% faster than the second-place 2-step LBM, Fuse LBM ranks the third, and the original LBM is the baseline. The performance of k-step LBM using 48 cores drops significantly and is worse than the Fuse tile LBM when the side length of the lattice is larger than 6144. Thirdly, when $L \geq 1536$, the 3-step-tile LBM performs the best and get up to 1514.6 MFLUPS.



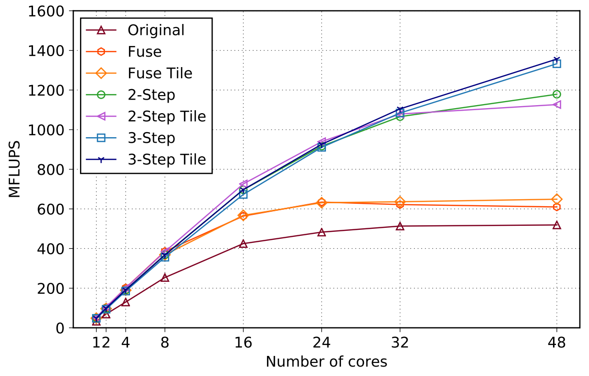
(a) $L = 192$.



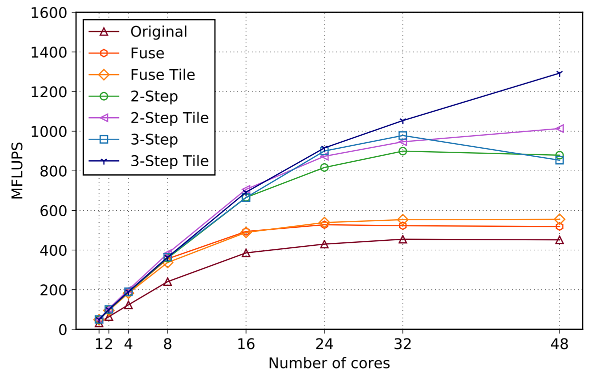
(b) $L = 384$.



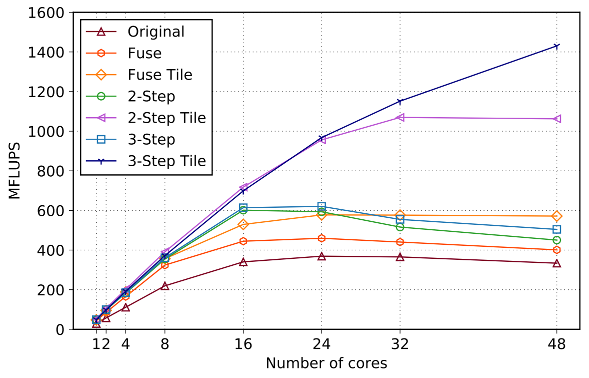
(c) $L = 768$.



(d) $L = 1536$.



(e) $L = 3072$.

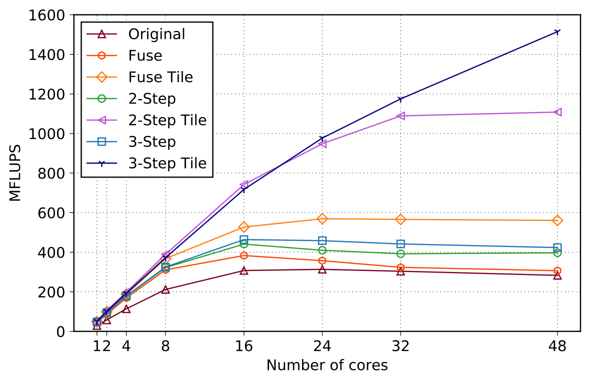


(f) $L = 6144$.

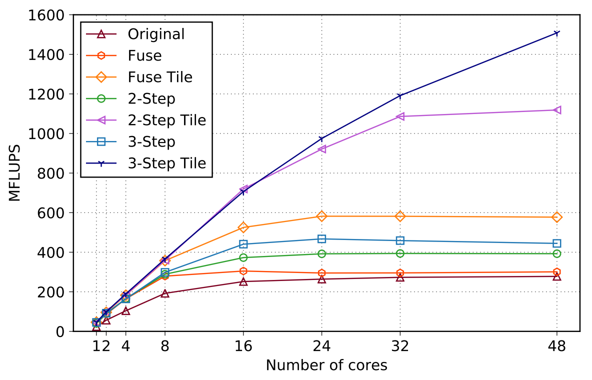
Figure 5.27. Skylake Strong Scalability performance.

Using 48 threads, it is up to 36.7% faster than the 2-step-tile LBM, 170% faster than the Fuse tile LBM, and 401% faster than the Fuse LBM. On the other hand, the 2-step-tile ranks the second when using 48 threads, with up to 97.7% faster than the Fuse tile LBM, and 275% faster than the Fuse LBM. At last, when we compare the results using 16 threads

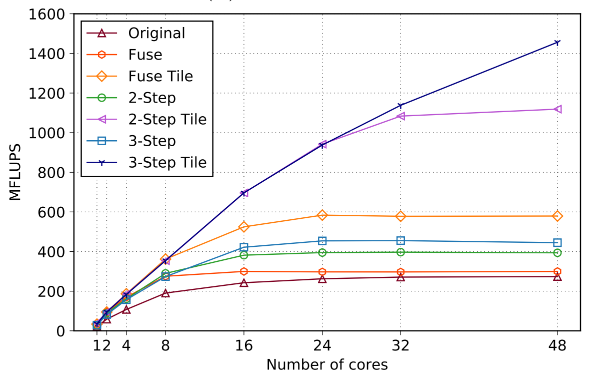
Figure 5.27. continued.



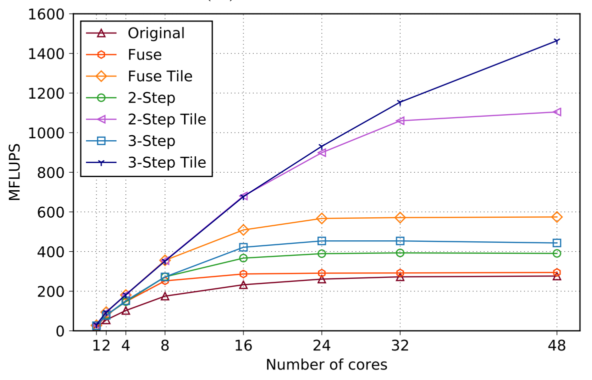
(a) $L = 12288$.



(b) $L = 24576$.



(c) $L = 28800$.

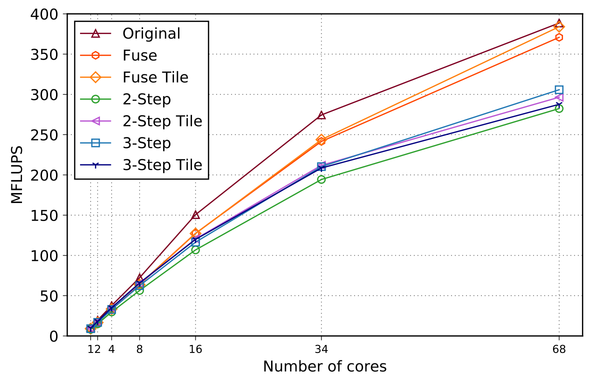


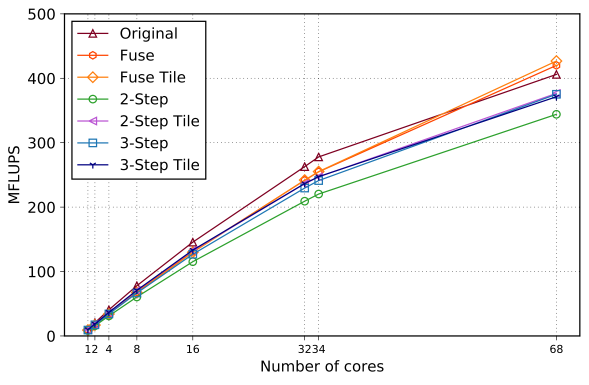
(d) $L = 33600$.

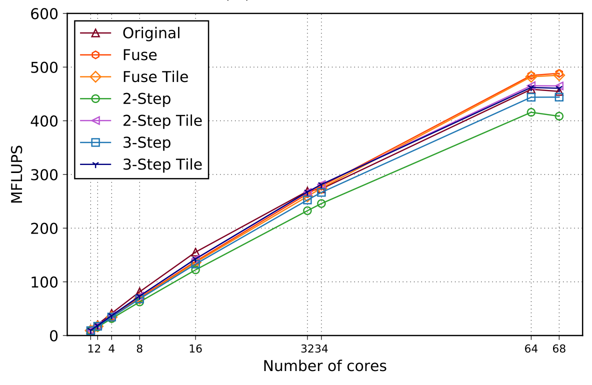
between Haswell and Skylake, we find Skylake has around 60% better performance on all algorithms. For example, the 2-step-tile LBM on Skylake with $L = 12288$ is 1.65X faster than on Haswell with $L = 14336$. This is highly related to 4X larger size of L2 cache, the 2D mesh structure of cores, and the non-inclusive L3 cache used in Skylake to reduce the memory latency significantly.

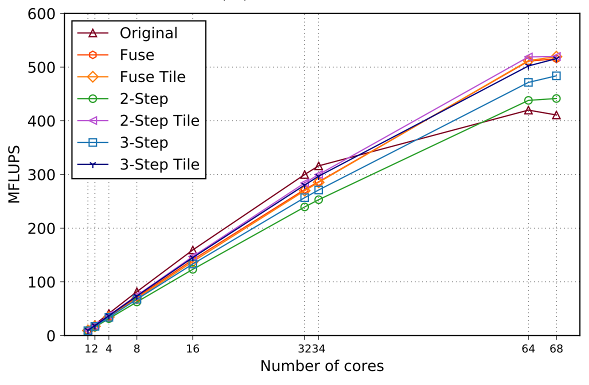
Table 5.8. Memory consumption for each 2D lattice in the parallel experiments on a *Stampede*2 KNL node.

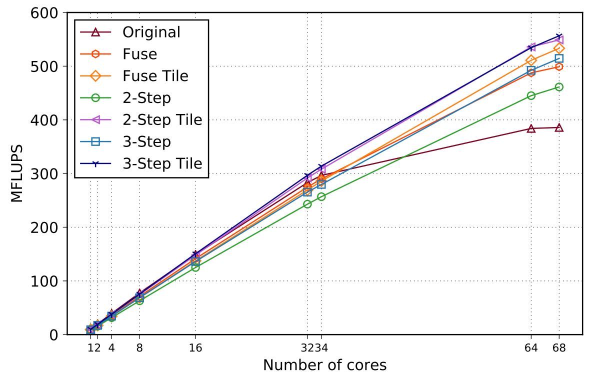
Edge size L	272	544	1088	2176	4352	8704	17408	20400	21760
Memory(B)	11.3M	45.2M	180.6M	722.5M	2.8G	11.3G	45.2G	62G	70.6G


(a) $L = 272$.


(b) $L = 544$.


(c) $L = 1088$.


(d) $L = 2176$.


(e) $L = 4352$.

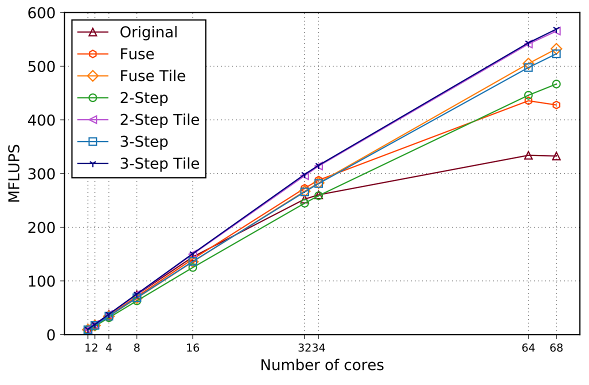
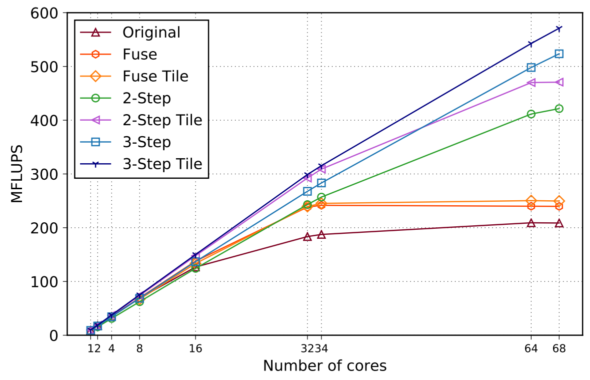

(f) $L = 8704$.

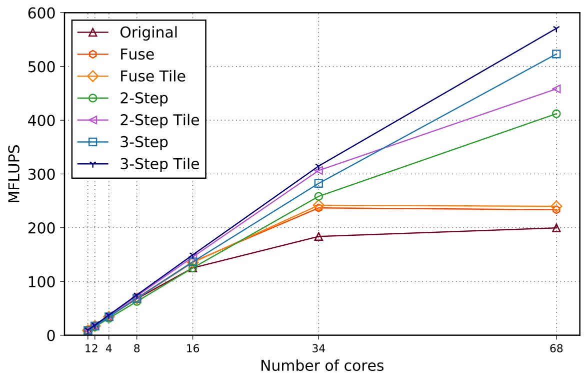
Figure 5.28. Knight Landing Strong Scalability performance.

Fig.5.28 and Fig.5.28 show the strong scalability of the 7 LBM algorithms on a KNL node of *Stampede2* with edge size $L = 272 \sim 21760$. Tab.5.8 presents the memory consumption of each lattice. The KNL nodes on Stampede2 are configured in cache mode, meaning that MCDRAM is configured as an "L3 cache" and the operating system transparently uses the

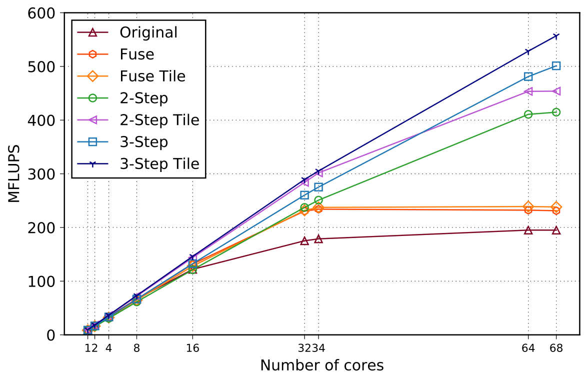
Figure 5.28. Continued.



(a) $L = 17408$.



(b) $L = 20400$.



(c) $L = 21760$.

MCDRAM to move data from main memory. But comparing to the real on-chip L1, L2 and L3, MCDRAM is still too low speed. Thus a high L1 and L2 cache hit rate is necessary for KNL to run at full speed. Fig.5.28a shows that the original performs the best for all threads cases. Fig.5.28b to Fig.5.28c shows the Fuse tile ranks the first when using 68 cores. Then after that, the k-step LBM performs the best, especially when $L \leq 17408$, the total memory size of the whole lattice cannot fit into the MCDRAM (16 GB). Since there is no on-chip L3 cache on KNL, the k-step LBM without tiling doesn't drop too much as the results on Haswell and Skylake. Fig.5.28d to Fig.5.29c shows that with the lattice $L \geq 1536$, the 3-step-tile LBM performs the best with up to 571.1 MFLUPS and achieves the nearly ideal linear strong scalability. Using 68 threads, it is up to 27% faster 2-step-tile LBM, 124%

faster than the Fuse tile LBM, and 162% faster than the Fuse LBM. On the other hand, the 2-step-tile ranks the second when using 48 threads, with up to 79.4% faster than the Fuse tile LBM, and 112.7% faster than the Fuse LBM.

Performance Analysis of Parallel LBM Experiments

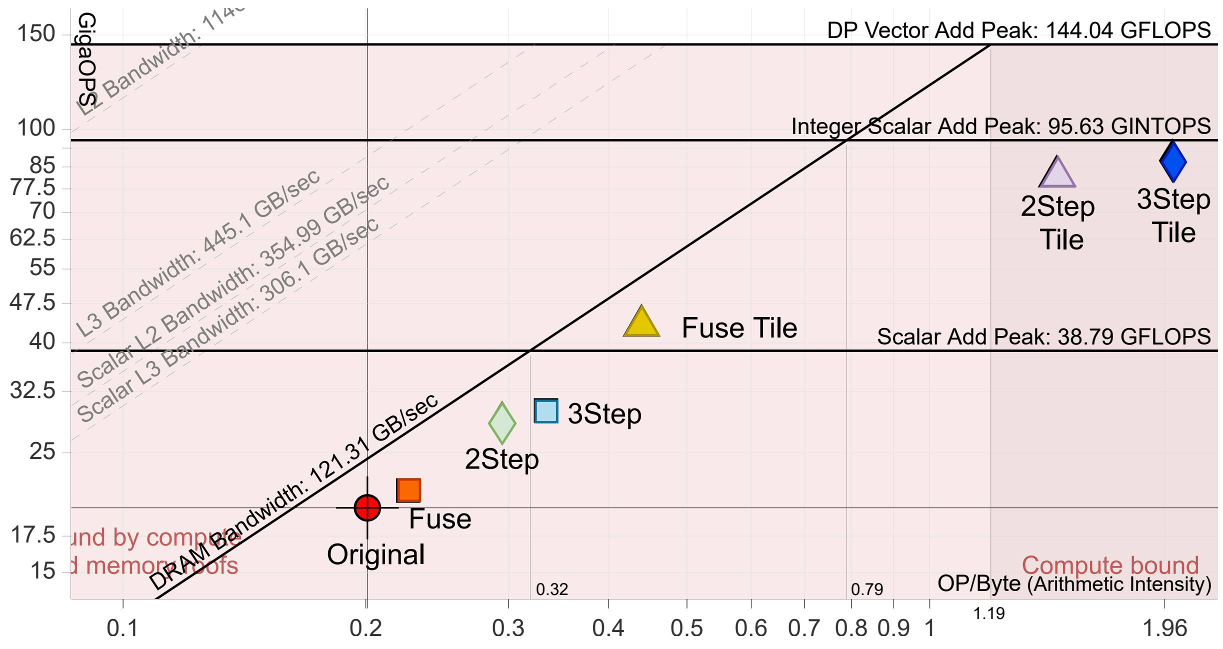


Figure 5.30. DRAM Roofline comparison for the 7 parallel LBM algorithms with 28 threads on the 14336 × 14336 square lattice on a Haswell node.

Table 5.9. Point information of each algorithm on the Roofline Fig.5.30

Algorithm	Original	Fuse	Fuse Tile	2Step	2Step Tile	3Step	3Step Tile
DRAM AI(OP/Byte)	0.2	0.225	0.44	0.295	1.432	0.334	2.002
Giga OPs	19.73	21.28	43.22	28.49	81.72	29.97	87.81
MFLUPS	143.48	154.48	304.81	205.25	579.37	227.88	668.44
Speedup	baseline	1.08X	2.124X	1.43X	4.04X	1.59X	4.66X

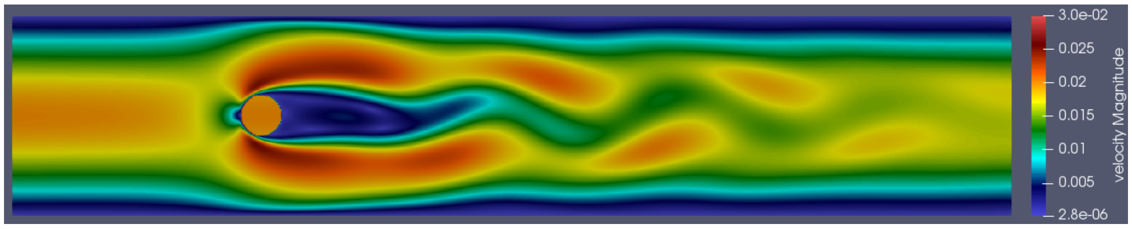
OP = INT + FLOAT operations.

Fig.5.30 shows the DRAM Roofline comparison for the 7 parallel LBM algorithms with 28 threads with $L = 14336$ on a Haswell node. The point information of each algorithm in the figure are in Tab.5.9. Firstly, we see that all the parallel algorithms without tiling are below the scalar double floating add peak (38.79 GFLOPS), while the other three with tiling are above it, but all of them are under the integer scalar add peak (95.63 GINTOPS). Secondly, in terms of the crosspoint of the roofs of DRAM and integer scalar add peak, the AI machine balance is 0.79 OP/Byte. Thus we see that the original, Fuse, Fuse tile, 2-step, and 3-step LBM are on the left side of the boundary, so they are memory-bounded by DRAM roof. Secondly, 2-step-tile LBM and 3-step-tile LBM are on the right side of the machine balance (0.79 OP/Byte), thus they are now compute-bounded, and reside under the ceilings of "DP vector add peak" (144.04 GFLOPS) and DRAM. We see that the thee-step tile and 2-step-tile LBM improves AI by 4.55X and 3.25X compared to Fuse tile LBM, respectively. At last, the parallel 3-step-tile LBM has the best GOPS and MFLUPS among the seven algorithms, and improves 4.66X compared to the original LBM.

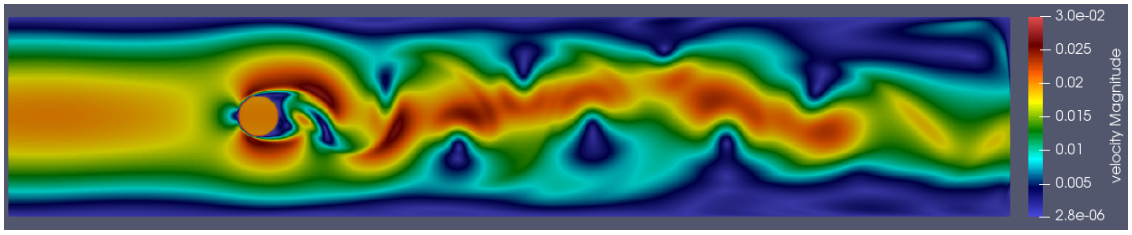
5.8.4 Visualization

The last experiment is used to visualize and validate the memory-aware LBM algorithm. Our simulation examines the widely known and extended test scenario, a flow past a cylinder placed in a channel, which dates back to the design of wings of an aircraft and understanding the behavior of the flow past them in the early 20th century. It turns out that the *Reynolds number* (i.e., the ratio of a fluid's inertial force to its viscous force) plays an important role in characterizing the behavior of the flow. As the Reynolds number increases to 100 or higher, an unstable periodic pattern is created, which is called the *Karman vortex street.*

Our algorithm can compute and output the velocity of each fluid point. We use Catalyst to convert those outputs to VTK files. Next, Paraview reads the VTK files and generates figures and videos. The simulation is a flow past a 1280×256 channel. An uncompressed cylinder at location (320,128) with a radius equaling 26 makes the steady-state symmetrical flow unstable. In Fig.5.31a, after running 100,000 steps, a Karman vortex street is generated. Fig.5.31a and 5.31b show the Karman vortex street when the Reynolds number equals 100 and 400, respectively. We can observe that more vortices are generated when the Reynolds number equals 400. This is because inertial forces dominate the viscous forces at higher



(a) Reynolds number = 100.



(b) Reynolds number = 400.

Figure 5.31. Vorticity plot of flow past a cylinder, a Karman vortex street is generated

Reynolds numbers, which tend to produce more chaotic eddies and induce flow instabilities [139]. The full simulation videos are published at https://youtu.be/C5IqsZVPV0Y and https://youtu.be/hyNN6yxdn18.

6. 3D PARALLEL MEMORY-AWARE LBM ON MANYCORE SYSTEMS

A version of this chapter is a pending publication in *Euro-Par'21, 27th International European Conference on Parallel and Distributed Computing*.

Chap. 5 merges multiple collision-streaming cycles (or time steps) in 2D, this chapter aims to augment the memory-awareness idea to support parallel 3D LBM. Sect.6.2 presents the baseline 3D LBM algorithms. Sect.6.3 design the sequential 3D memory-aware LBM algorithms that combine five features: single-copy distribution, loop fusion (single sweep), swap algorithm, prism traversal, and merging two collision-streaming cycles. Sect.6.4 presents the parallel 3D memory-aware LBM, which aims to keep the thread safety on the intersection layers among threads and reduce the synchronization cost in parallel. Sect.6.5 conducts two groups of experiments on three different manycore architectures, followed by performance analysis. The first group of sequential experiments (i.e., using a single CPU core) shows that our memory-aware LBM outperforms the state-of-the-art Palabos (Fuse swap prism LBM solver)[116] by up to 19% on a Haswell CPU and 15% on a Skylake CPU. The second group evaluates the performance of parallel algorithms. The experimental results show that our parallel 3D memory-aware LBM outperforms Palabos by up to 89% on a Haswell node with 28 cores, 85% on a Skylake node with 48 cores, and 39% on a Knight Landing node with 68 cores.

Table 6.1. Four LBM algorithms discussed in this chapter. Each algorithm has its sequential version and parallel version.

Algorithms	**Description**
Fuse (swap) LBM	Use a single copy of distribution, swap algorithm and loop fusion
Fuse (swap) prism LBM	Use a single copy of distribution, swap algorithm, loop fusion and prism traversal
2-step (swap) LBM	Use a single copy of distribution, swap algorithm, loop fusion, and merge two steps
2-step (swap) tile LBM	Use a single copy of distribution, swap algorithm, loop fusion, prism traversal and merge two steps

6.1 Introduction

Although it seems to be simple to move from the 2D space to 3D space, it is significantly much more difficult to design an efficient 3D memory-aware LBM algorithm. In this chapter, we focus on solving the following three main challenges.

1. As geometries change from 2D to 3D, the required data storage increases from $O(N^2)$ to $O(N^3)$, and the data dependency of the lattice model becomes much more complicated (from D2Q9 to D3Q19 as shown in Fig.6.1. [1] There exist single-copy distribution methods to reduce data storage cost by half, but they require following a particular traversal order. Can we combine one of the best single-copy distribution methods with our idea of merging multiple collision-streaming cycles to design a 3D memory-aware LBM with higher performance?

2. If the combination is possible, since normal 3D tiling [140] doesn't apply to this case, how to additionally explore the spatial locality?

3. When designing the parallel 3D memory-aware LBM, a non-trivial interaction occurs at the boundaries between threads, how to guarantee the thread safety and avoid race conditions? Although some existing works use wavefront parallelism to explore the temporal locality, they insert frequent layer-wise synchronizations among threads every time step [47], [51]. In this chapter, we also aim to reduce the synchronization cost in parallel threads.

6.2 Baseline 3D LBM Algorithm

The baseline 3D LBM algorithm in this chapter is called *Fuse swap LBM* as shown in Alg.10, which involves three features: single-copy distribution, swap algorithm, and loop fusion. To reduce the high data storage cost in 3D, Sect.5.4.1 introduces a few single copy distribution kernels such as swap [43], AA [45], shift [44], and esoteric twist [46], but each needs to follow some constraints. For example, swap requires predefined order of discrete

[1]↑3D LBM has a sequence of velocity sets with increasingly higher accuracy and computational complexity, i.e., D3Q13, D3Q15, D3Q19, D3Q27, or customized lattice sets.

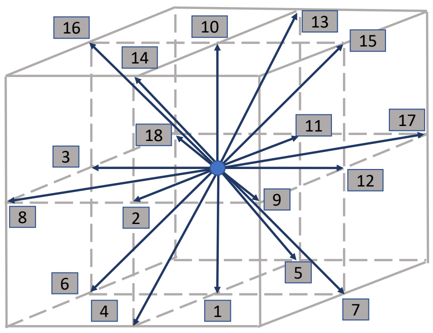
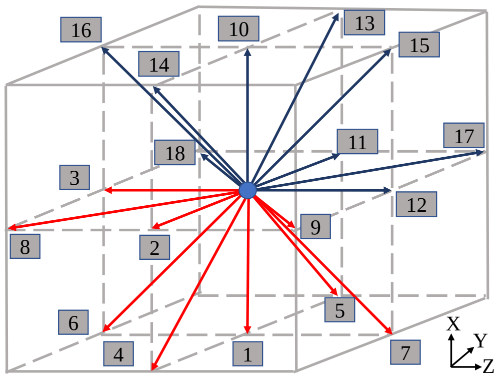


Figure 6.1. The D3Q19 velocity sets of each cell in 3D LBM.

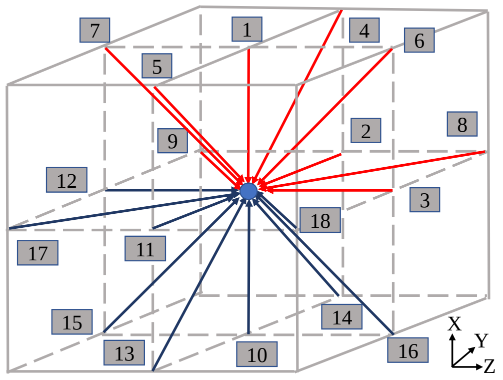
cell velocities, AA requires distinguishing between even and odd time steps, shift requires extra 2D layer space. We choose the swap algorithm since it is relatively simpler than the other single-copy distribution methods, and is more efficient to use simple index arithmetic to access neighbors in the matrix-based memory organization. The swap algorithm [43] replaces the copy operations between a cell and its neighbors in the streaming kernel by a value swap, thereby it is in-place and doesn't require the second copy. But when combining it with loop fusion, we must guarantee that the populations of neighbors involved in the swap are already in a post-collision state to keep thread safety.

The work-around solution is to adjust the traversal order of simulation domains with a predefined order of discrete cell velocities [121], [122]. Thus each cell can stream its post-collision data by swapping values with half of its neighbors pointed by the red arrows ($1 \sim 9$ directions for D3Q19 in Fig.6.2a), if those neighbors are already in post-collision and have "reverted" their distributions. We define this operation as "*swap_stream*". The "*revert*" operation in Fig.6.2b lets a cell locally swap its post-collision distributions to opposite directions. We will explain the behavior of the two operations later in detail.

To make the Fuse swap LBM more efficient, Palabos pre-processes and post-processes the boundary cells on the bounding box at line 2 and 7, respectively, so that we can remove the boundary checking operation in the inner bulk domain. Fig.6.3a presents the big picture of three stages of Fuse swap LBM traversing on a cuboid box from (1,1,1) to (lx, ly, lz). Thus Alg.10 is divided into three stages in every time step as follows.

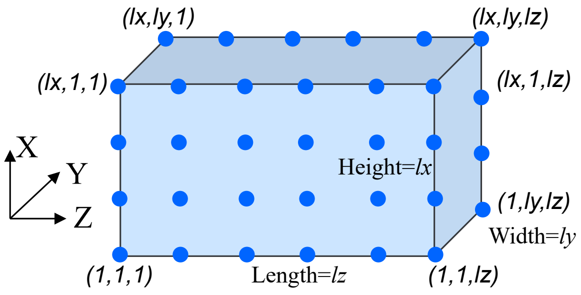


(a) *swap_stream*: the post-collision distribution will be swapped with half of its neighbors pointed by red arrows.

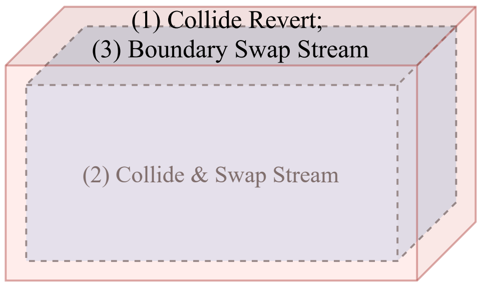


(b) *revert*: a cell locally swaps its post-collision data to its opposite direction.

Figure 6.2. Two operations used in sequential 3D Fuse swap LBM.



(a) A 3D cuboid simulation domain.



(b) Three stages in 3D Fuse swap LBM.

Figure 6.3. Big picture of 3D Fuse swap LBM.

1. Stage I (line 2): *collide* followed by "**revert**" operations are performed at the cells located on the bounding box (i.e., six red surfaces of the 3D block domain in Fig.6.3b).

2. Stage II (line 3 $\sim$ 6): *collide* followed by "**swap_stream**" operations are performed at the cells located in the center blue bulk domain from (2,2,2) to $(lx-1, ly-1, lz-1)$ in Fig.6.3b. This stage contains the most workload.

3. Stage III (line 7): "**boundary_swap_stream**" on the bounding box.

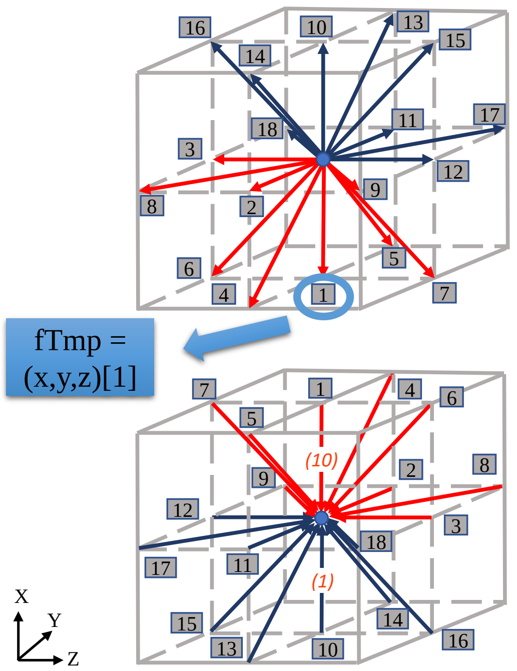
Now, we explain why we need "*revert*" operation performed at the cells on the bounding box in Stage I. Fig.6.2b shows that a cell locally swaps its post-collision distribution data to its opposite directions. Thus, the red arrows with 1 $\sim$ 9 directions are reverted to the top

Algorithm 10 3D Fuse swap LBM

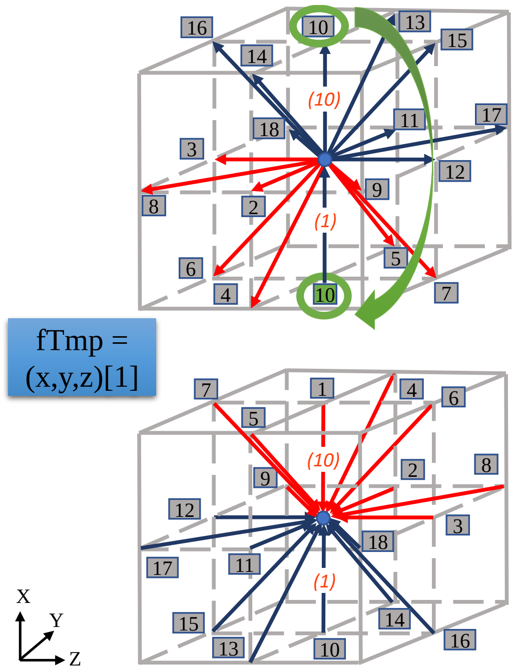
```
1: for iT = 0; iT < N; ++iT do
2:   Stage I: collide and revert on the bounding box, i.e., 6 surfaces of cuboid (1,1,1) to $(lx, ly, lz)$
     // Stage II: bulk domain computation
3:   for iX = 2; iX $\leq lx - 1$; ++iX do
4:     for iY = 2; iY $\leq ly - 1$; ++iY do
5:       for iZ = 2; iZ $\leq lz - 1$; ++iZ do
6:         collide & swap_stream on (iX, iY, iZ) to half of its neighbors
7:   Stage III: boundary_swap_stream on the bounding box
```

part of a cell, while the blue arrows are reverted to the bottom part. We remain heads of red and blue arrows the same directions as in Fig.6.2a, indicating that we should originally stream (copy) the post-collision data into neighbor's storage along with those directions. Due to the nature of the four-value swap in *swap_stream*, the cells in the inner bulk domain do not need to check whether their neighbors are on the boundary or not. Therefore, we can safely remove the if-statements for inner cells to check out-of-bound errors, thereby speed up the bulk domain computation. But since we have not executed *stream* on these boundary cells, we leave them in Stage III.

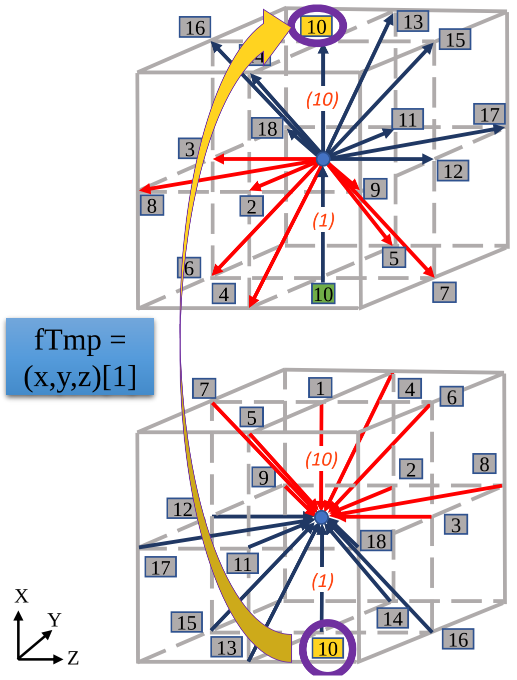
In Stage II, the "*swap_stream*" operation performed on the inner cells streams their post-collision data to half of their neighbors. Fig.6.4 illustrates this process between two cells along the first and tenth direction (vertical X axis) as an example, and *swap_stream* on other pairs of directions are similar. Fig.6.4a is the initial state: cell (x, y, z) on the top of the figure is at layer 1, and has just completed *collide*, and is ready to stream its post-collision distribution to its neighbors; cell $(x-1, y, z)$ on the bottom is at layer 0, and has completed the *collide* and *revert* operation. We can see that because of the *revert*, the storage of $(x-1, y, z)$ at the first direction (marked by "(1)" with orange color) now stores the post-collision distribution data which is originally at the tenth direction (navy blue arrow), indicating that the data currently stored in the first direction should be streamed upward. Since we aim to copy the post-collision data in (x, y, z) in the first direction downward into the storage of $(x-1, y, z)$ at the first direction, meanwhile to copy the post-collision data in $(x-1, y, z)$ at the tenth direction upward into the storage of (x, y, z) at the tenth direction.



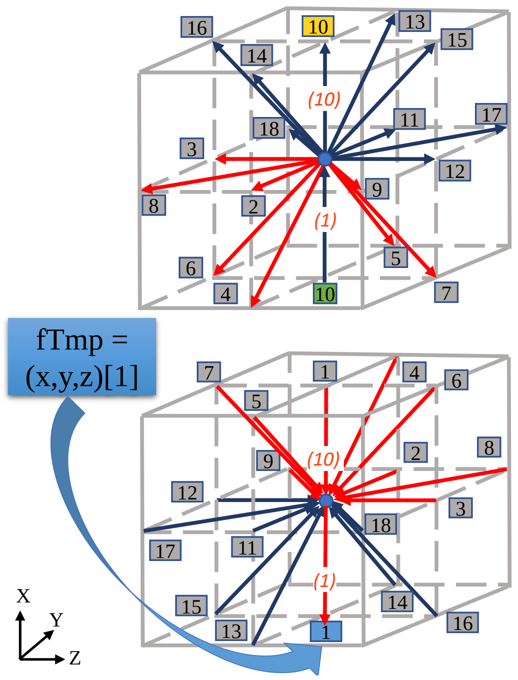
(a) $fTmp \leftarrow (x, y, z)[1]$. $(Push)$



(b) $(x, y, z)[1] \leftarrow (x, y, z)[10]$. $(Push)$



(c) $(x, y, z)[10] \leftarrow (x - 1, y, z)[1]$. $(Pull)$



(d) $(x - 1, y, z)[1] \leftarrow fTmp$. $(Push)$

Figure 6.4. Swap Stream on (x, y, z) and $(x-1, y, z)$ along with the first and tenth directions.

To achieve such a goal, we use a temporary variable $fTmp$, thereby *swap_stream* operation can be divided into 4 copy instructions as follows.

1. $fTmp \leftarrow (x, y, z)[1]$. Cell (x, y, z) *push* (copy) its post-collision distribution at the first direction into the $fTmp$, as shown by the thick blue arrow in Fig.6.4a.

2. $(x, y, z)[1] \leftarrow (x, y, z)[10]$. Cell (x, y, z) *push* the data in its tenth direction into its first direction, as shown by the thick green arrow in Fig.6.4b. We mark the storage in (x, y, z)'s first direction with green color and the number 10, indicating that it currently stores the cell's post-collision distribution at the tenth direction.

3. $(x, y, z)[10] \leftarrow (x-1, y, z)[1]$. Cell (x, y, z) *pull* the data in $(x-1, y, z)$'s first direction (which now stores the post-collision data at the tenth direction because of "*revert*" in Stage I) into the storage in (x, y, z)'s tenth direction, as shown by the thick yellow arrow in Fig.6.4c. We mark the storage in (x, y, z)'s tenth direction with yellow color and the number 10, indicating that it currently stores the $(x-1, y, z)$'s post-collision data at the tenth direction.

4. $(x-1, y, z)[1] \leftarrow fTmp$. $fTmp$ *push* its data into $(x-1, y, z)$'s first direction, as shown by the thick blue arrow in Fig.6.4d. We mark the storage in $(x-1, y, z)$'s first direction with blue color and the number 1, indicating that it currently stores the (x, y, z)'s post-collision data at the first direction.

At last, the "*boundary_swap_stream*" in Stage III is performed at the cells on the bounding box. Since the inner cells in the blue bulk domain of Fig.6.3b have already *swap_stream* the data with half of post-collision data on the boundary cells during stage II, the other half of post-collision data on the boundary cells (red arrows in Fig.6.2b) remains to be streamed to neighbors. During this operation, we add extra boundary-checking statements when accessing the neighbors' coordinate in case of out-of-bound errors.

6.2.1 3D Fuse Swap Prism LBM Algorithm

To further increase data reuse, a combination with spatial locality like loop blocking in a small region can be used. Otherwise, we need to go through a whole line or a whole layer in a large domain, which results in data eviction from the cache. However, the normal 3D tiling [140] doesn't apply to the case that combines single-copy distribution and swap algorithm. But when cutting Fig.6.2a (*swap_stream*) along the Y-Z plane, we have a planar slice as shown in Fig.6.5. We observe that a cell (star) swaps with its lower right neighbor (orange) at direction 9. In other words, when the orange cell swaps with the upward row, its neighbor "shifts" one cell *leftward*. Similarly, if cutting Fig.6.2a (*swap_stream*) along the X-Y plane, when a cell swaps data with the upward row, its neighbor "shifts" one cell *forward*. This access pattern is named "*prism traversal*", as we will see later that the shape of traversal order is either a prism or parallelepiped shape.

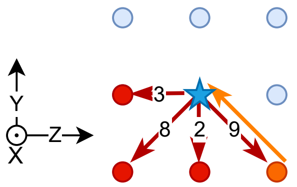


Figure 6.5. Planar slice when cutting Fig.6.2a (swap stream operation) along Y-Z plane.

Fig.6.6 gives an example of prism traversal on a prism with four layers. On layer i$X = 1$, we can traverse four rows, and the last coordinate of the first row is 4. Thus, the last coordinate on the second row is 3, and the length of other rows gradually decreases by one. On layer i$X = 2$, we can traverse three rows, and the last coordinate of the first row is 3. On layer i$X = 3$, we can traverse two rows, and the last coordinate of the first row is 2. On layer i$X = 4$, only one cell is accessed.

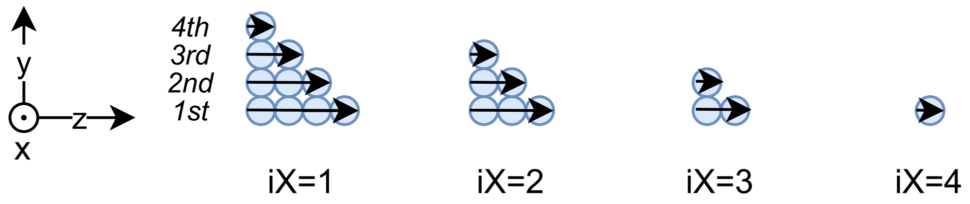


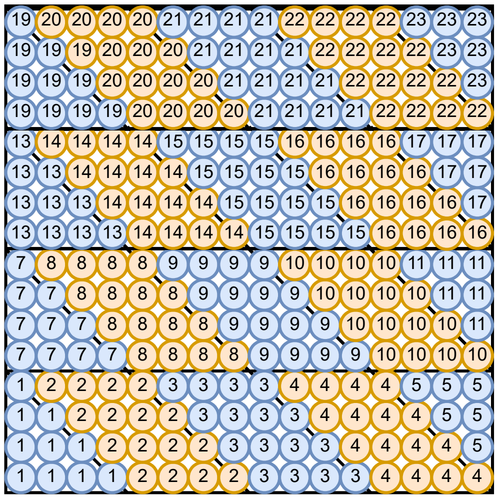
Figure 6.6. Prism traversal with four layers when *tile* = 4.

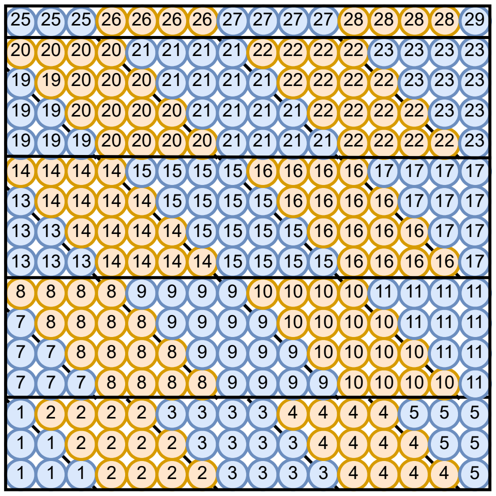
Next, we use an example to explain its access pattern in a $4 \times 16 \times 16$ cuboid with stride *tile* = 4. Fig.6.7a∼6.7d are the four separate 16×16 layers of the cuboid from bottom to top. The cells with the same number on the four layers construct a *prism* (e.g., the cells with number 1 in Fig.6.7a∼6.7d construct a pyramid-shape "Prism 1"). In each prism, we still firstly go along Z-axis, then along Y-axis, and upward along X-axis at last. Then we traverse prism-wise from Prism 1 to Prism 30. Finally, if a cuboid is much larger than this example, the majority of prisms are "parallelepiped " shapes like Prism 9 and 10 in Fig.6.7e. Thus when we traverse *tile* number of cells on Z-axis at row iY, they can swap with *tile* number of cells but shifted one cell leftward at row i$Y + 1$, thereby we get parallelograms in Fig.6.7a∼6.7d. When the shift encounters domain boundaries, we truncate the parallelograms and get isosceles right triangles or part of parallelograms.

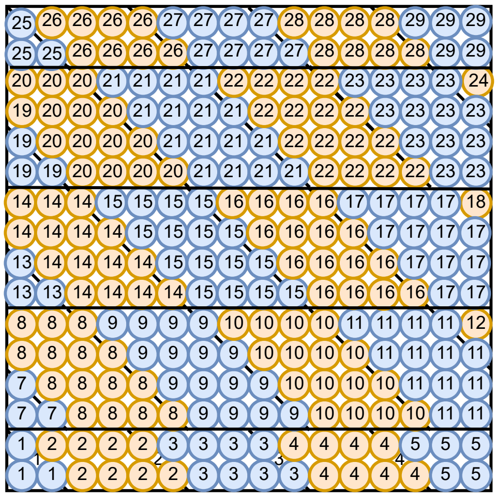
Alg.11 presents the 3D Fuse swap LBM algorithm with prism traversal, short for Fuse prism LBM. It also contains three stages, and the first and third stage are the same as Fuse swap LBM. The bulk domain computation in stage II is different. This part uses `dy` and `dx` to shift on the Y axis and X axis, respectively. As a result, the `outerY` range is from 2 to $ly - 1 + tile - 1 = ly + tile - 2$, while the `outerZ` range is from 2 to $lz - 1 + 2 * (tile - 1) = lz + 2 * tile - 3$. The combination of `innerX, innerY` and `innerZ` accesses the cells, so the "MIN" and "MAX" statements from line 7 to 11 are used to ensure the user-chosen "tile" not to access data outside the domain boundary. This also allows users to select any non-negative *tile* parameter to produce the best performance on their architectures.

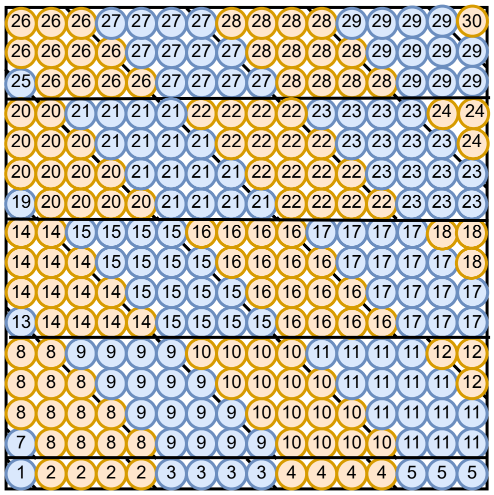
Algorithm 11 3D Fuse swap LBM with Prism Traversal

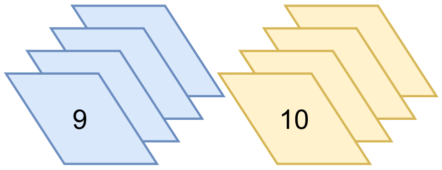
```
 1: for iT = 0; iT < N; ++iT do
 2:   Stage I: collide and revert on the bounding box
    // Stage II: bulk domain computation
 3:   for outerX = 2; outerX ≤ $lx - 1$; outerX += tile do
 4:     for outerY = 2; outerY ≤ $ly - 1 + tile - 1$; outerY += tile do
 5:       for outerZ = 2; outerZ ≤ $lz - 1 + 2 * (tile - 1)$; outerZ += tile do
 6:         for innerX=outerX; innerX ≤ MIN(outerX+tile-1, $lx - 1$); ++innerX,++dx do
 7:           minY = outerY - dx; maxY = minY + tile - 1; dy = 0;
 8:           for innerY=MAX(minY,2); innerY≤MIN(maxY, $ly - 1$); ++innerY,++dy do
 9:             minZ = outerZ - dx - dy; maxZ = minZ + tile - 1;
10:             for innerZ=MAX(minZ, 2); innerZ ≤ MIN(maxZ, $lz - 1$);++innerZ do
11:               collide & swap_stream on (innerX, innerY, innerZ) to half of its neighbors.
12:   Stage III: boundary_swap_stream on the bounding box
```


(a) Layer iX = 1.


(b) Layer iX = 2.


(c) Layer iX = 3.


(d) Layer iX = 4.



(e) Prism 9 and 10 are parallelpiped shapes. Layer iX = 4 is on the top.

Figure 6.7. Fuse swap prism traversal on a 4 × 16 × 16 cuboid block.

6.3 Sequential 3D Memory-aware LBM Algorithms

Sect.6.2.1 has added four features to the original 3D LBM: one copy distribution, swap algorithm, loop fusion, and prism traversal. Based on them, we design and develop the sequential 3D memory-aware LBM by adding the temporal locality feature, i.e., merging two collision-streaming cycles. We start adding it by dropping off the prism traversal feature first. Next, we add back the prism traversal and make the complete version.

Fig.6.8 shows an example of how to merge two collision-streaming cycles given a $4 \times 4 \times 4$ cube: if a cell at (iX, iY, iZ) completes its first collide-streaming cycle, we expect to execute the second computation of cell at (iX-1, iY-1, iZ-1) if it fulfills dependencies, so that we can increase data reuse when data is still in the cache.

1. Fig.6.8a shows the initial state of all cells at the current time step t. Green cells are on boundaries, and blue cells are located in the inner bulk domain.

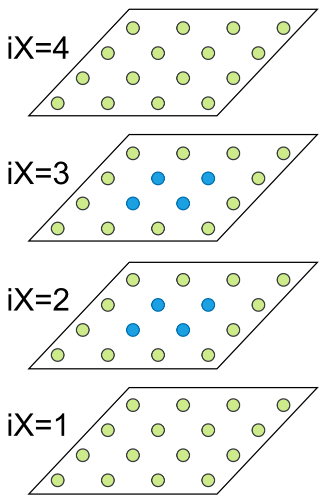
2. In Fig.6.8b, we compute the first *collide*, *revert*, and *boundary_swap_stream* row by row on the bottom layer iX = 1. After a cell completes the first computation, we change it to orange.

3. In Fig.6.8c, we compute the first *collide* and *boundary_swap_stream* row by row till cell (2,2,1) on the second layer iX = 2.

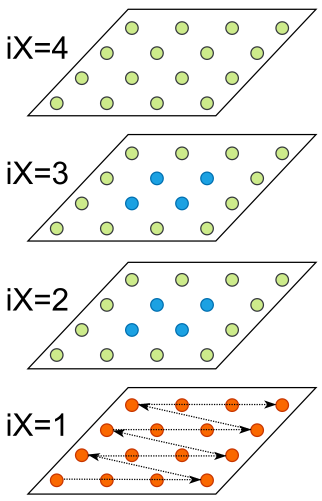
4. In Fig.6.8d, cell (2,2,2) completes its first *collide* and *swap_stream*, so we change it to red since they are inner cells. Then we observe that cell (1,1,1) is ready for the second *collide*, so we change it to yellow.

5. In Fig.6.8e, we execute the second *collide* and *boundary_swap_stream* on cell (1,1,1), and change it to purple.

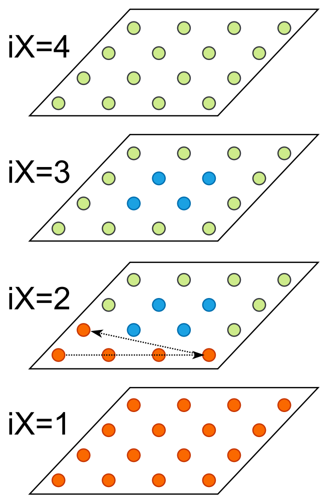
Alg.12 presents the sequential 3D memory-aware LBM algorithm without prism traversal. The computation of boundary cells and inner bulk cells for two time steps are combined inside the innermost loop from line 4 to 8, and is divided into three sub-steps. For simplicity of description, we define three helper functions. The *boundary_cell_comp* function executes three sequential operations: *collide*, *revert*, and *boundary_swap_stream* at a boundary cell.



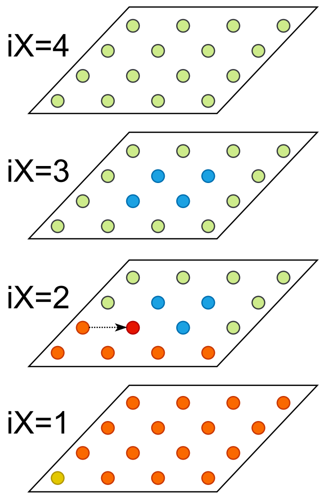
(a) Initialization



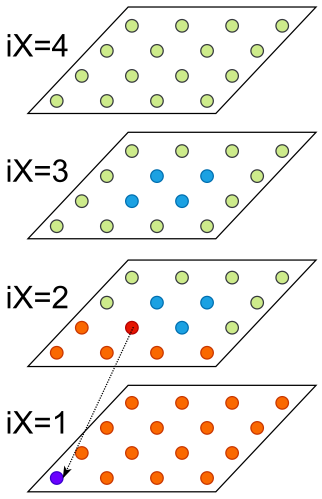
(b) First *collide*, *revert*, *boundary_swap_stream* on layer iX = 1



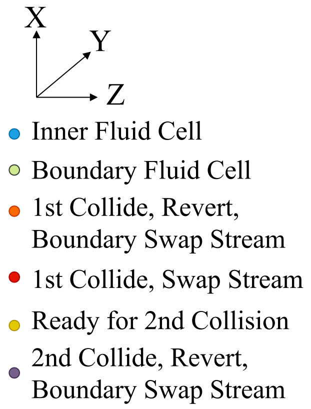
(c) First *collide*, *revert*, *boundary_swap_stream* on surface iX = 2



(d) First *collide*, *swap_stream* on cell (2,2,2).



(e) Second *collide*, *revert*, *boundary_swap_stream* on cell (1,1,1)


X
Y
Z
● Inner Fluid Cell
○ Boundary Fluid Cell
● 1st Collide, Revert, Boundary Swap Stream
● 1st Collide, Swap Stream
● Ready for 2nd Collision
● 2nd Collide, Revert, Boundary Swap Stream

(f) Legends.

Figure 6.8. 3D sequential two-step memory-aware LBM on a $4 \times 4 \times 4$ cube lattice.

The *adaptive_collide_stream* function executes either *boundary_cell_comp* at a boundary cell, or *collide* and *swap_stream* at an inner bulk cell. The *boundary_neighbor_handler* function handles the second computation of (iX, iY, iZ)'s neighbors at certain locations. The three helper functions are also used in later algorithms.

Algorithm 12 Sequential 3D Memory-aware LBM

```
1: for iT = 0; iT < N; iT+=2 do
2:   for iX = 1; iX ≤ $lx$; ++iX do
3:     for iY = 1; iY ≤ $ly$; ++iY do
4:       for iZ = 1; iZ ≤ $lz$; ++iZ do
            /* (1) First computation at time step $t$. */
5:         adaptive_collide_stream(iX, iY, iZ);
            /* (2) Second computation at time step $t + 1$. */
6:         if iX > 1 && iY > 1 && iZ > 1 then
7:           adaptive_collide_stream(iX-1, iY-1, iZ-1);
            /* (3) Second computation of neighbors at certain locations.*/
8:         boundary_neighbor_handler(iX, iY, iZ);
9:   Second collide, revert & boundary_swap_stream on the top layer iX = $lx$.
10: function boundary_cell_comp(iX, iY, iZ)
11:   collide, revert, & boundary_swap_stream on (iX, iY, iZ) to half of its neighbors;
12: function adaptive_collide_stream(iX, iY, iZ)
13:   if (iX, iY, iZ) is on the boundary then
14:     boundary_cell_comp(iX, iY, iZ);
15:   else
16:     collide & swap_stream on (iX, iY, iZ) to half of its neighbors;
17: function boundary_neighbor_handler(iX, iY, iZ)
    // Handle the second computation of (iX, iY, iZ)'s neighbors at certain locations.
18:   if iZ == $lz$ then // (iX, iY, iZ) is the last cell of a row.
19:     boundary_cell_comp (iX-1, iY-1, iZ);
20:   if iY == $ly$ && iZ > 1 then // (iX, iY, iZ) is in the last row of a layer.
21:     boundary_cell_comp(iX-1, iY, iZ-1);
22:   if iY == $ly$ && iZ == $lz$ then // (iX, iY, iZ) is the last cell on a layer.
23:     boundary_cell_comp(iX-1, iY, iZ);
```

1. For the first computation at time step t, we do not separately compute the bounding box and the inner bulk domain like Fuse swap LBM. Because if a cell is ready to compute the second computation at time step $t + 1$, it must get the first post-collision data at time step t from its neighbors, who can be either inner cells or boundary cells. Thus we must start from the the bottom layer iX = 1 and complement the *boundary_swap_stream* on the cells after their first *collide* and *revert*. Differently, Fuse swap LBM that leaves the *stream* in the post-processing stage. Contrarily, inner cells still compute their first *collide* and *swap_stream* operation. Therefore, we call

adaptive_collide_stream function in line 5 for all cells to adaptively execute the first computation at time step t.

2. For the second computation at time step $t + 1$, the if-statement in line 6 ensures that the cell to compute at time step $t + 1$ are in post-collision state, which can apply to the standard D3Q15, D3Q19 and D3Q27 velocity sets. [2] Besides, we still need to distinguish cells on the boundary or not. Line 7 calls the *adaptive_collide_stream* function to execute the second computation at time step $t + 1$.

3. We call *boundary_neighbor_handler* to compute the second computation of cell (iX, iY, iZ)'s neighbors at certain locations. When (iX, iY, iZ) is at the last cell of a row (line 9 $\sim$ 10), the last row of a layer (line 11 $\sim$ 12), or the last cell of a layer (line 13 $\sim$ 14), we need to complete execute *boundary_cell_comp* on (iX-1, iY-1, iZ), (iX-1, iY, iZ-1) or (iX-1, iY, iZ) at time step $t + 1$, respectively.

4. After the iX loop of line 2 $\sim$ 8, line 9 wraps up the second computation on the top layer lx of the domain.

In the real implementation, we use loop unrolling to move some if-branches outside the innermost loop to avoid breaking the instruction pipeline. For example, we can unroll the innermost iZ loop by iZ = 1, iZ = 2 $\sim$ $lz - 1$, and iZ = lz to separately compute the boundary cells, which requires *boundary_swap_stream* and the inner bulk cells requiring *swap_stream*. Besides, if we compute the first two layers beforehand, we can remove the if-branch at line 6, since the cells from layer 3 don't need to check this condition. Details about implementation are presented in the Github repository [141].

6.3.1 Sequential 3D Prism Memory-aware LBM Algorithm

We can safely combine "prism traversal" with merging two collision-streaming cycles, since the cell at the left forward down corner has been in a post-collision state and ready to compute the second computation when following the above traversal order. Line 6 $\sim$ 10

[2]↑For D3Q19 and D3Q15 dynamics, when (iX, iY, iZ) completes first computation, we can start the second computation on (iX-1, iY-1, iZ). But to generalize the algorithm on D3Q27, we shift one cell and compute (iX-1, iY-1, iZ-1).

Algorithm 13 Sequential 3D Prism Memory-aware LBM

```
1: tile := stride of the prism traversal
2: for iT = 0; iT < N; iT += 2 do
3:   for outerX = 1; outerX ≤ $lx$; outerX += tile do
4:     for outerY = 1; outerY ≤ $ly$ + tile - 1; outerY += tile do
5:       for outerZ = 1; outerZ ≤ $lz$ + 2* (tile - 1); outerZ += tile do
6:         for innerX=outerX; innerX ≤ MIN(outerX+tile-1, $lx$); ++innerX, ++dx do
7:           minY = outerY - dx; maxY = minY + tile - 1; dy = 0; /* forward shift */
8:           for innerY=MAX(minY, 1); innerY ≤ MIN(maxY, $ly$); ++innerY, ++dy do
9:             minZ = outerZ - dx - dy; maxZ = minZ + tile - 1; /* leftward shift */
10:            for innerZ=MAX(minZ, 1); innerZ ≤ MIN(maxZ, $lz$); ++innerZ do
       /* (1) First computation at time step $t$. */
11:              adaptive_collide_stream(innerX, innerY, innerZ);
       /* (2) Second computation at time step $t + 1$. */
12:              if innerX > 1 && innerY > 1 && innerZ > 1 then
13:                adaptive_collide_stream(innerX-1, innerY-1, innerZ-1);
       /* (3) Second computation of neighbors at certain locations. */
14:              boundary_neighbor_handler(innerX, innerY, innerZ);
15:   Second collide, revert & boundary_swap_stream on the top layer iX = $lx$.
```

in Alg.13 traverse the 3D cuboid domain prism by prism with stride *tile* = 4. Within each prism, it accesses cells regularly row by row, and then layer by layer. The innermost operations of line 11 $\sim$ 14 merge two-step computation, which follows the three sub-steps in Alg.12.

In the implementation [141], similar loop unrolling methods as mentioned before are also used to ease the “if-branch” inefficiency. The “MIN” and “MAX” statements allow users to choose any non-negative “tile” size to produce the best performance on their architectures.

6.4 Parallel 3D Memory-aware LBM

To support manycore systems, we use OpenMP to realize the parallel 3D memory-aware LBM algorithm. Fig.6.9 shows the way to partition the data domain by our parallel 2-step prism LBM. A 3D cuboid domain is decomposed along the X-axis (*height*) by n threads. Let $sub\_h = lx/n$, thus each thread has a 3D sub-cuboid domain with $sub\_h \times ly \times lz$. Although it can be optimal to choose the axis depending on the dimensions of a given domain, our current implementation partitions only along the X-axis.

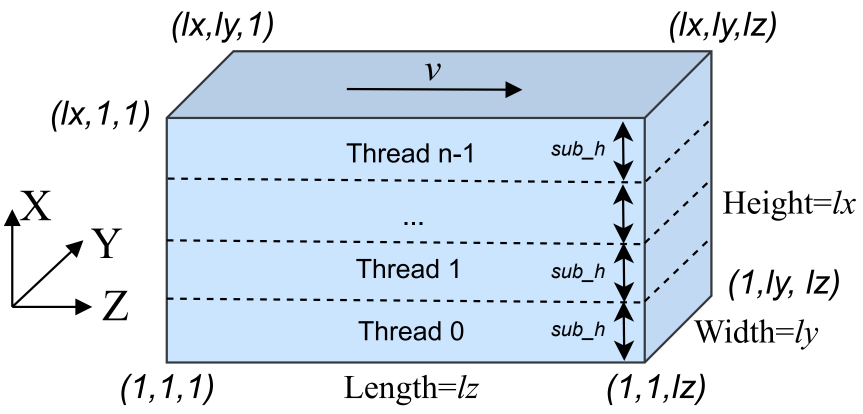


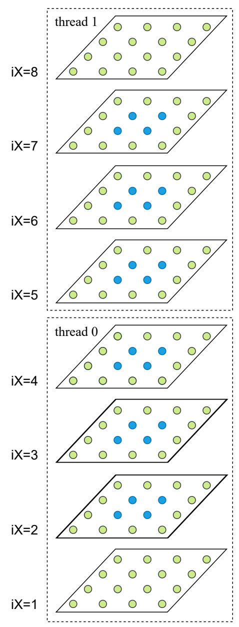
Figure 6.9. Partition of a 3D cuboid domain by n threads.

Fig.6.10 illustrates its idea on an $8 \times 4 \times 4$ cuboid, which is evenly partitioned by two threads along the X-axis (*height*). Then each thread traverses a $4 \times 4 \times 4$ sub-domain with prism stride $tile = 4$. Line 4 in Alg.14 defines the start and end layer index of each thread's sub-domain, thus the end layers $myEndX$ are "*intersections*" (e.g., layer 4 and 8). Fig.6.10a shows the initial state at time step t. In addition, the parallel 3D memory-aware Alg.14 consists of three stages: Preprocessing, Sub-domain computation, and Post-processing.

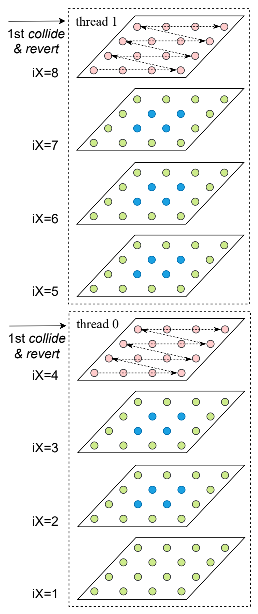
Fig.6.10 illustrates the idea of 3D parallel memory-aware LBM Alg.14 on a $8 \times 4 \times 4$ 3D cuboid domain. The cuboid domain is evenly decomposed by two threads along the X-axis (*height*), so each thread traverses a $4 \times 4 \times 4$ sub-domain with prism stride $tile = 4$. We define the **top layer of each thread's sub-domain** as the "**intersection**", e.g., layer 4 and 8. Fig.6.10a shows the initial state at time step t. We use an extra variable $layer\_id$ at line 12 of Alg.14 as the internal index of each thread's sub-domain. The algorithm generally is divided into three stages, i.e., preprocessing, main bulk computation (step $2 \sim 9$), and post-processing.

1. **Stage I (Preprocessing)** *line 5 in Alg.14*: In Fig.6.10b, thread 0 and 1 compute the first *collide* and *revert* on the "intersection" layers 4 and 8, respectively, and then change them to pink.

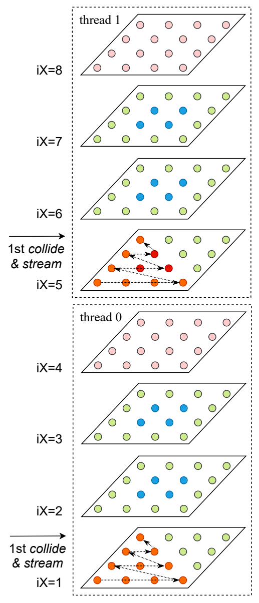
2. **Stage II (Sub-domain computation)** handles five cases from step 2 to 7. In *case 0* (*lines 15∼17 in Alg.14*), when thread 0 and 1 access the cells on the first



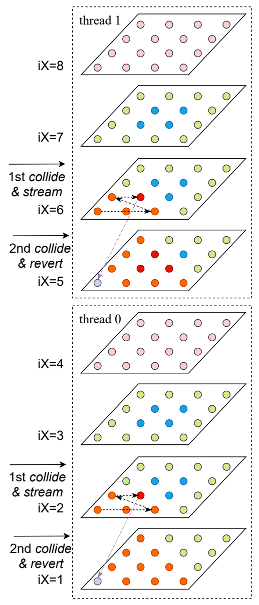
(a) Initialization



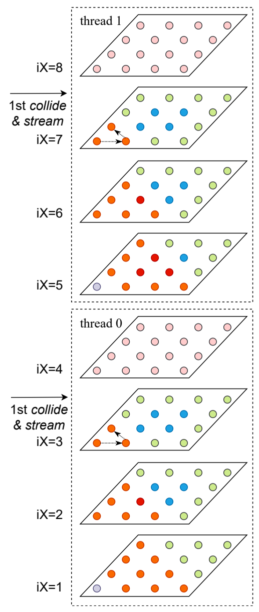
(b) Stage I.



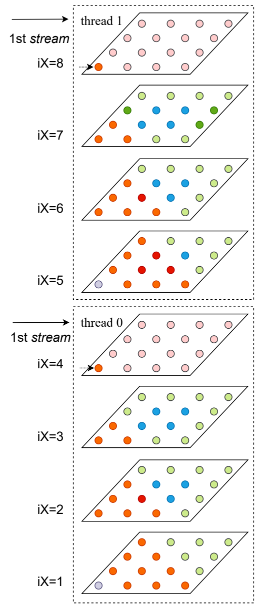
(c) Stage II: Case 1.



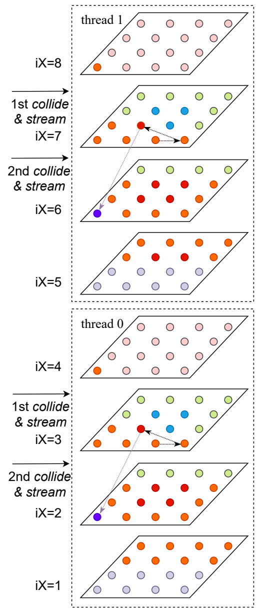
(d) Stage II: Case 2.



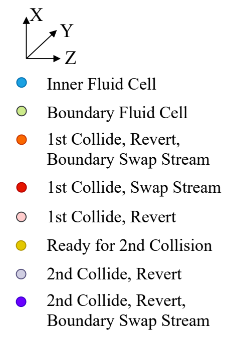
(e) Stage II: Case 0.



(f) Stage II: Case 3.



(g) Stage II: Case 4.


X
Y
Z
● Inner Fluid Cell
● Boundary Fluid Cell
● 1st Collide, Revert, Boundary Swap Stream
● 1st Collide, Swap Stream
● 1st Collide, Revert
● Ready for 2nd Collision
● 2nd Collide, Revert
● 2nd Collide, Revert, Boundary Swap Stream

(h) Legends.

Figure 6.10. 3D parallel two-step memory-aware LBM on a $8 \times 4 \times 4$ cuboid.

row and column of each layer except the "intersection" layers, we execute the first *boundary_cell_comp* on them and change them to orange.

Algorithm 14 3D Parallel Two-step Memory-aware LBM with Prism Traversal

```
 1: for iT = 0; iT < N; iT += 2 do
 2:   #pragma omp parallel default(shared){
 3:   $sub\_h = lx/nthreads$; // height of each thread's sub-domain
 4:   myStartX = $1 + thread\_id \times sub\_h$; myEndX = $(thread\_id + 1) \times sub\_h$;
    /* Stage I: First collide & revert on the intersection layer.*/
 5:   collide & revert on all $ly \times lz$ cells on layer iX = myEndX;
 6:   #pragma omp barrier
    /* Stage II: Main computation in each thread's sub-domain.*/
 7:   for outerX = myStartX; outerX ≤ myEndX; outerX += tile do
 8:     for outerY = 1; outerY ≤ $ly$ + tile - 1 ; outerY += tile do
 9:       for outerZ = 1; outerZ ≤ $lz$ + 2 * (tile - 1); outerZ += tile do
10:         for innerX=outerX; innerX≤MIN(outerX+tile-1, myEndX); ++innerX, ++dx do
11:           minY = outerY - dx; maxY = minY + tile - 1; dy = 0; /* forward shift */
12:           for innerY=MAX(minY, 1); innerY≤MIN(maxY, $ly$); ++innerY, ++dy do
13:             minZ = outerZ - dx - dy; maxZ = minZ + tile - 1; /* leftward shift */
14:             for innerZ = MAX(minZ, 1); innerZ ≤ MIN(maxZ, $lz$); ++innerZ do
    // Case 0: First collide & stream on the first row and column of each layer except the intersection layers.
15:               if innerX != myEndX && (innerX == 1 or innerY == 1 or innerZ == 1) then
16:                 First $boundary\_cell\_comp(innerX, innerY, innerZ)$;
17:                 continue;
    // Case 1: First collide & stream on layer myStartX:
18:               if innerX == myStartX then
19:                 First adaptive_collide_stream(innerX, innerY, innerZ);
    // Case 2: First collide & stream on myStartX + 1; Second collide & revert on myStartX:
20:               else if innerX == myStartX + 1 then
21:                 First adaptive_collide_stream(innerX, innerY, innerZ);
22:                 Second collide & revert on (innerX-1, innerY-1, innerZ-1);
23:                 Handle the second collide & revert of neighbors at certain boundary locations;
    // Case 3: First stream on layer myEndX; Second collide & stream under one layer:
24:               else if innerX == myEndX then
25:                 First adaptive_stream(innerX, innerY, innerZ);
26:                 Second adaptive_collide_stream(innerX-1, innerY-1, innerZ-1);
27:                 boundary_neighbor_handler (innerX, innerY, innerZ);
    // Case 4: first collide & stream on other layers; Second collide & stream under one layer:
28:               else
29:                 First adaptive_collide_stream(innerX, innerY, innerZ);
30:                 Second adaptive_collide_stream(innerX-1, innerY-1, innerZ-1);
31:                 boundary_neighbor_handler(innerX, innerY, innerZ);
32:   #pragma omp barrier
    /* Stage III: second collide & stream on the intersection; then second stream on the layer myStartX. */
33:   adaptive_collide_stream at all $ly \times lz$ cells on layer iX = myEndX;
34:   #pragma omp barrier
35:   stream at all $ly \times lz$ cells on layer iX = myStartX;
36:   }
```

3. Fig.6.10c shows *case 1* (*lines 18∼19 in Alg.14*). When thread 0 and 1 access the cells on layer $myStartX$ (iX = 1 & 5), respectively, we execute the *adaptive_collide_stream* on them to compute at time step t, and then change the boundary cells to orange and the inner cells to red.

4. Fig.6.10d shows *case 2* (*lines 20∼23 in Alg.14*). When thread 0 and 1 are on layer $myStartX+1$ (iX = 2 & 6), respectively, we execute the first *adaptive_collide_stream* at time step t and change boundary cells to orange and inner cells to red. Meanwhile, cell (5,1,1) and (1,1,1) have collected the data dependencies to *collide* at time step $t + 1$, we execute the second *collide* and *revert* but without *stream* on them, and change to light purple.

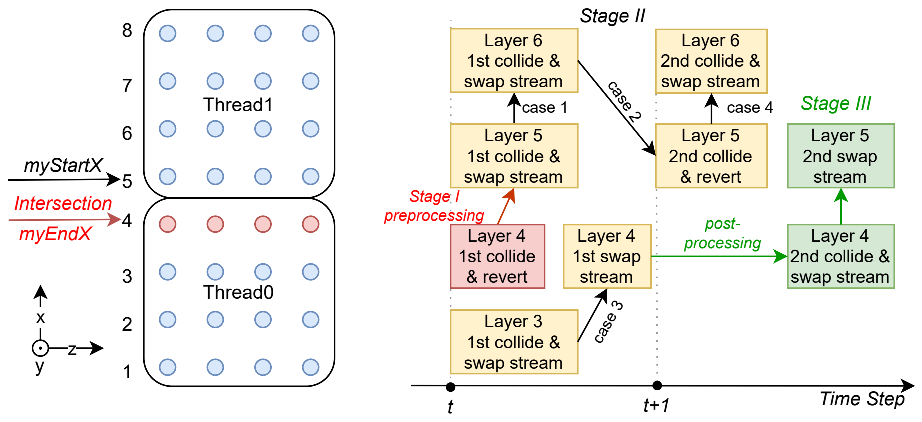
5. Fig.6.10e shows that when continuing traversal in Prism 1, thread 0 and 1 are on layer iX = 3 & 6. Since the cells traversed in this figure are in the first row and column, case 0 is used here, otherwise, case 4 is used.

6. Fig.6.10f shows *case 3* (*lines 24∼27 in Alg.14*). When thread 0 and 1 are on the intersection layers (iX = 4 & 8), we execute the remaining first *stream* at time step t due to preprocessing in Stage I. Then if cells under one layer (iX = 3 & 7) collect their data dependency at time step $t + 1$, we execute the second *adaptive_collide_stream* on them.

7. Fig.6.10g shows *case 4* (*lines 28∼31 in Alg.14*). When thread 0 and 1 are on the other layers of sub-domain, we conduct the first *adaptive_collide_stream* on (innerX, innerY, innerZ) at time step t, and then the second *adaptive_collide_stream* on (innerX-1, innerY-1, innerZ-1) at time step $t + 1$. Then we call *boundary_neighbor_handler* to compute the neighbors of (innerX, innerY, innerZ) at certain locations at time step $t + 1$.

8. **Stage III (Post-processing)** *lines 33∼35 in Alg.14*: Firstly, since Stage I and case 3 have completed the first computation on intersection layers, we wrap up the second *collide* and *stream* on intersections. Secondly, since case 2 have executed the second *collide* and *revert* on the first layers $myStartX$ of each sub-domain, the second *stream* remains to be executed.

6.4.1 Handle Thread Safety on Intersection Layers



8
7
6
5
4
3
2
1
Thread1
Thread0
myStartX
Intersection
myEndX
x
z
y
Stage II
Layer 6 1st collide & swap stream
case 1
Layer 5 1st collide & swap stream
Stage I preprocessing
Layer 4 1st collide & revert
Layer 4 1st swap stream
case 3
Layer 3 1st collide & swap stream
case 2
Layer 6 2nd collide & swap stream
case 4
Layer 5 2nd collide & revert
post-processing
Stage III
Layer 5 2nd swap stream
Layer 4 2nd collide & swap stream
t
t+1
Time Step

Figure 6.11. Handle thread safety on intersection layers. To keep thread safety: (1) the first *swap_stream* on layer 5 during stage II should be delayed after the first *revert* on layer 4 during stage I; (2) during stage II, the second *swap_stream* on layer 6 should be delayed after the second *revert* on layer 5. (3) during stage III, the second *swap_stream* on layer 5 should be delayed after the second *swap_stream* on layer 4 during stage II.

We aim to keep thread safety and minimize the synchronization cost during parallel executions. To this end, we need to carefully design the initial state of each thread so that the majority of computation stays in each threads' local sub-domain. The left part of Fig.6.11 shows the view of Fig.6.10 along X-Z axis, and layer 4 is the intersection layer that partitions two threads' sub-domains. The right part shows the data dependencies near the intersection layer in two time steps. In the figure, the red block represents Stage I of Alg.14, yellow blocks Stage II, and green blocks Stage III . The arrows indicate that data are transferred from layer A to B by using a procedure (or B depends on A). There are three non-trivial dependencies requiring to handle thread safety near intersection layers. (1) Since the swap algorithm only streams data to half of the neighbors under one layer, the *swap_stream* on layer 5 —the first layer of thread 1's sub-domain— should be delayed after the *revert* on layer 4 in thread 0's sub-domain. Thus, in Stage I, we pre-process *collide* and *revert* at time step t but without *stream* on layer 4, since *stream* on layer 4 depends on the post-collision on layer 3, which has not been computed yet. (2) In Stage II, the second *swap_stream* on layer 6 called by the case 4 procedure should be delayed after the second *revert* but without *swap_stream* on layer 5. This is because thread 1 cannot guarantee that thread 0 has completed the second $swap\_steam$ on layer 4. To keep thread safety, $swap\_stream$ on layer 5 is delayed to Stage III. (3) Thus, in Stage III, the second $swap\_stream$ on layer 5 is delayed after the second $swap\_stream$ on layer 4. Above all, since the majority of computation happens in Stage II of each thread's sub-domain, we avoid the frequent "layer-wise" thread synchronizations that occur in the wave-front parallelism. Besides, we only synchronize at the intersection layers every two time steps, hence the overhead of three *barriers* of Alg.14 becomes much less.

6.5 Experimental Evaluation

In this section, we first present the experimental setup and validations on our 3D memory-aware LBM. Then we evaluate its sequential and parallel performance.

6.5.1 Experiment Setup and Verification

The details of our experimental hardware platforms are provided in Tab.5.2. To evaluate the performance of our new algorithms, we use the 3D lid-driven cavity flow simulation in Fig.6.12 as an example. The 3D cavity has a dimension of $lz \times ly \times lx$, and its top lid moves with a constant velocity v. As time goes by, the flow inside the cavity becomes unstable and generates vortexes. Our 3D memory-aware LBM algorithms have been implemented as C++ template functions, which are then added to the Palabos framework. For verification, we construct a *cavity* with the same procedure, and then separately execute four algorithms on it, i.e., Palabos solvers $fuse()$ and $fuse\_prism()$ for N time steps, and our memory-aware algorithms $two\_step\_prism()$ and $two\_step\_prism\_omp()$ for $N/2$ time steps. Then, we compute the velocity norm of each cell and write to four separate logs. At last, we verify that our algorithms produce the same result as Palabos for guaranteeing software correctness.

6.5.2 Performance Analysis of Sequential 3D Memory-aware LBM

The first set of experiments with 3D cavity flows compare the sequential performance of four different LBM algorithms, which are the Fuse swap LBM (with/without prism traversal),

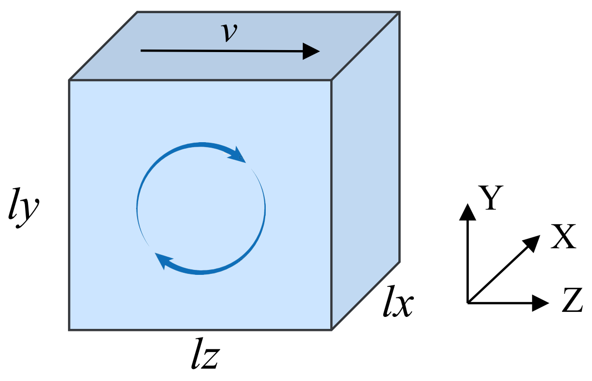


Figure 6.12. 3D lid-driven cavity benchmark: the top lid moves with a constant velocity v.

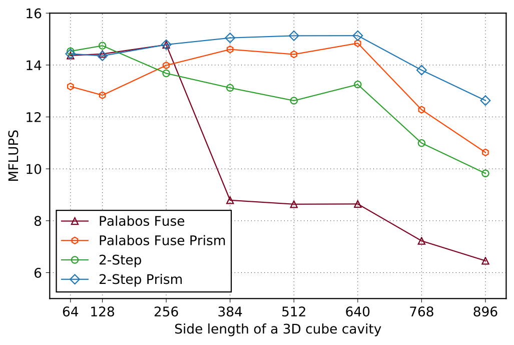
and the two-step memory-aware LBM (with/without prism traversal). For simplicity, we use the abbreviations of Fuse LBM, Fuse prism LBM, 2-step LBM, and 2-step prism LBM, respectively. The simulation input is 3D cubes with edge size $L = 64 \sim 896$. Every algorithm is executed five times, and the average MFLUPS (millions of fluid lattice node updates per second) is calculated. For the "prism" algorithms, different prism strides ("tile" ranging from 8, 16, 32, ..., to 448) are tested, and we select the best performance achieved.

Fig.6.13a shows the sequential performance on the Haswell CPU. When we use a small edge size, e.g. $L = 64, 128$, the 2-step LBM is the fastest. But when edge size $L \geq 256$, the 2-step prism LBM performs the best and achieves up to 15.1 MFLUPS. At the largest $L = 896$, the 2-step prism LBM acquires 12.6 MFLUPS and is 18.8% faster than the second-fastest Fuse prism LBM.

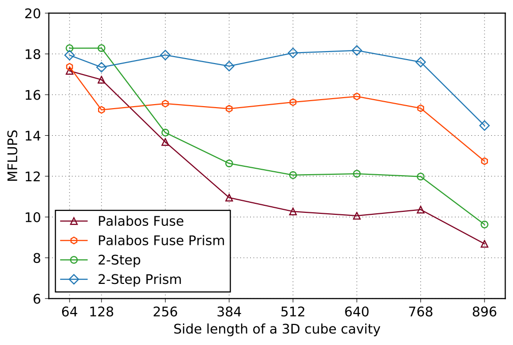
Table 6.2. Memory allocation size for each 3D cube lattice in the sequential experiments.

Edge size L	64	128	256	384	512	640	768	896
One layer memory (MB)	0.7	2.7	10.7	23.9	42.3	66	95	129
Cube memory (GB)	0.045	0.34	2.7	9.0	21.2	41.4	71.4	113.3

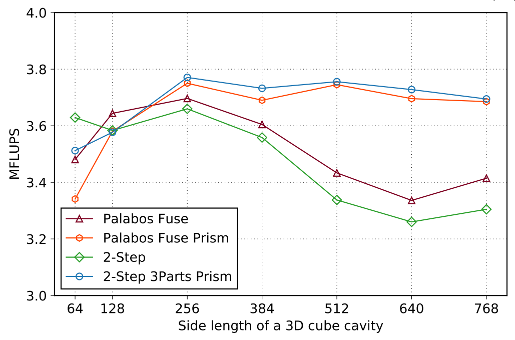
We observe that the performance of algorithms without prism traversal starts to drop when edge size $L \geq 768$. Especially, Fuse swap LBM plummets 40.5% from 14.8 MFLUPS to 8.79 MFLUPS when L increases from 256 to 384.The reason is that the L3 cache cannot hold



(a) Haswell.



(b) Skylake.



(c) Knight Landing.

Figure 6.13. Sequential performance using four LBM algorithms on three types of CPUs.

two layers of cells. Tab.6.2 shows the memory consumption of one layer and total domain used in the experiments. Since Fuse swap LBM *collides* on (iX, iY, iZ) and *swap_stream* to half of its neighbors on layer iX and iX-1, it accesses cells on two layers without spatial data reuse. When $L \geq 384$, two layers of cells $23.9 \times 2 = 47.8MB$ exceeds L3 cache (35 MB per socket on Haswell). Similarly, since 2-step LBM accesses cells on three layers, its performance also drops when $L \geq 384$, Meanwhile, without prism traversal, we notice that 2-step LBM is up to 53.3% faster than Fuse swap LBM. With prism traversal, Fuse prism LBM is up to 71.7% faster than Fuse swap LBM, and 2-step prism is LBM up to 28.6% faster than 2-step LBM. Hence, we can conclude that prism traversal and merging two steps both significantly increase data reuse on a large domain.

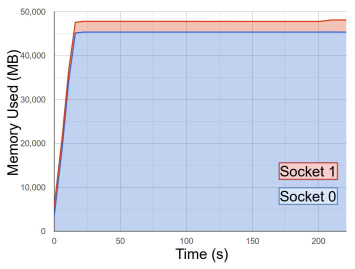
Secondly, Fig.6.13b shows that the sequential performance on the Skylake CPU. The 2-step LBM is still the fastest when $L = 64, 128$, and up to 9.3% faster than Fuse swap LBM. Similarly, when $L \geq 256$, 2-step prism LBM performs the best again, and achieves up to 18.2 MFLUPS and up to 15.5% faster than Fuse prism LBM at when $L = 512$. Meanwhile, without prism traversal, we notice that 2-step LBM is up to 20.5% faster than Fuse swap LBM. With prism traversal, Fuse prism LBM is up to 58.2% faster than Fuse swap LBM, and 2-step prism LBM is up to 50.4% faster than 2-step LBM. In the largest $L = 896$, 2-step prism LBM achieves 14.5 MFLUPS and is 13.7% faster than Fuse prism LBM. Besides, we also find that for all the algorithms with every L, the performance on Skylake is better than on Haswell. For example, the 2-step prism LBM on Skylake is on average 20.5% faster than on Haswell. This suggests that the larger L2 cache (1 MB on Skylake vs 256KB on Haswell) helps to increase L2 hit, and non-inclusive L3 helps to reduce DRAM memory latency. Another interesting result is that when $L = 256$, 2-step LBM drops more on Skylake than on Haswell. This is because three layers of cells ($10.7 \times 3 = 32.1MB$) nearly exhaust 33 MB L3 cache per socket on Skylake, but they can fit in 35 MB L3 cache per socket on Haswell.

At last, Fig.6.13c shows the sequential performance on Knight Landing CPU, we see that when $L \geq 256$, 2-step prism LBM is the best and can be up to 1.15% faster than Fuse prism LBM. The reason why 2-step prism LBM on KNL does not gain the same significant speedup as with the previous two CPUs is that the LLC (last level cache) on KNL is 1 MB shared L2 cache on every two cores, and no large L3 cache exists. Let's assume each core occupies 0.5 MB L2 cache exclusively, but it still cannot hold a 64×64 layer of cells, which results in few data reuse and many DRAM accesses.

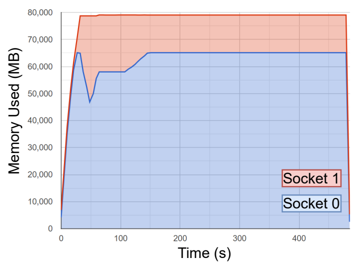
Above all, we need to use spatial locality by adding the feature of prism traversal. Consequently, on Haswell and Skylake, fuse tile LBM is up to 71.7% and 58.2% faster than Fuse swap LBM, and 2-step tile LBM is up to 28.6% and 50.4% faster than 2-step LBM. When only adding the feature of merging two steps, 2-step LBM is faster than Palabos (Fuse) by up to 53.3% on Haswell and 20.5% on Skylake. Hence, we conclude that both prism traversal and merging two steps significantly increase cache reuse on the large domain.

Why do all sequential performance drop when L is very large?
In Fig.6.13, we observe that the performance of all algorithms starts to drop when $L \geq 768$ on Haswell and $L = 896$ on Skylake, but doesn't drop on KNL. To investigate the reason, we use Remora[142] to monitor the memory free and used on each socket of a compute node every 5 seconds. In Fig.6.14, a group of stacked charts presents the memory free and used on each socket, when we run the above benchmark with edge size $L = 640, 768, 896$ on a

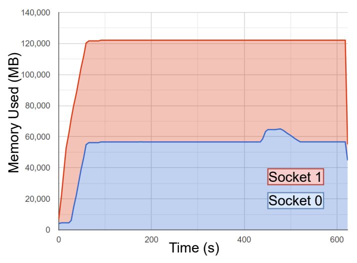
Haswell node. If memory usage exceeds the DRAM capacity per socket, NUMA memory accesses are involved and causes longer foreign memory access latency than local ones.



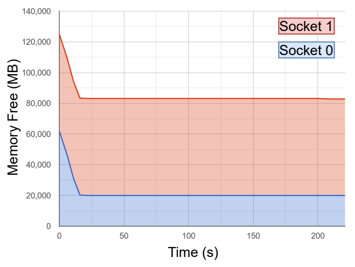
(a) Memory used ($L = 640$)



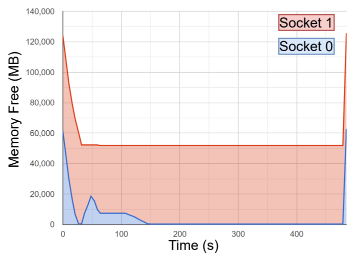
(b) Memory used ($L = 768$)



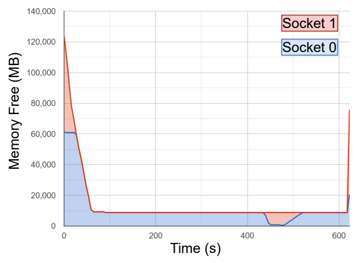
(c) Memory used ($L = 896$)



(d) Memory free ($L = 640$)



(e) Memory free ($L = 768$)



(f) Memory free ($L = 896$)

Figure 6.14. Memory used and free on two sockets of a Haswell node.

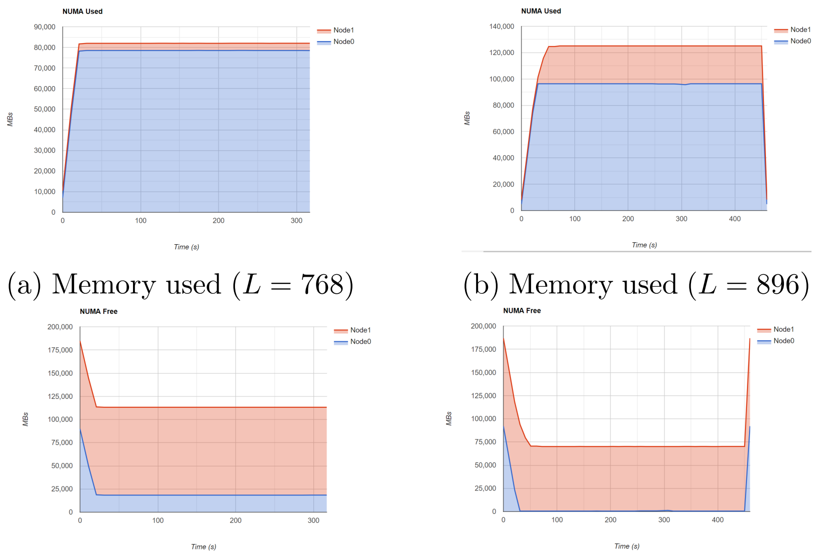
1. When $L = 640$, the allocated memory is 41.4GB and smaller than the $128GB/node \div 2sockets = 64GB/(socket \cdot node)$ DRAM capacity on a Haswell node. In Fig.6.14a, 44 GB memory of socket 0 (blue area) is mainly in use, while socket 1 only uses 2.4 GB memory. Meanwhile, Fig.6.14d also shows that socket 1 (red area) has 61.6 GB of free memory. Since NUMA access is relatively small in this setup, we don't see a significant performance drop when $256 \leq L \leq 640$.

2. When $L = 768$, the allocated memory is 71.4GB $>$ 64GB DRAM per socket. In Fig.6.14b, we see that socket 1 has average 13.6 GB memory used, which is 5.7X more NUMA memory accesses than $L = 640$. Meanwhile, Fig.6.14e shows that socket 1 has average 50.4 GB of free memory.

3. When $L = 896$, the allocated memory is 113.3 GB and 1.8X larger than 64GB DRAM per socket. Socket 1 has used 63.9 GB memory in Fig.6.14c, and almost none free

memory in Fig.6.14f. This indicates that socket 1 in this case has 4.7X more NUMA access than $L = 768$.

We can conclude that given a 3D lattice whose memory allocation is larger than the DRAM capacity per socket, because of large amounts of NUMA memory access involved, sequential performance will reduce significantly, which motivates the needs of parallel memory-aware LBM algorithms.



(a) Memory used ($L = 768$) (b) Memory used ($L = 896$)

(c) Memory free ($L = 768$) (d) Memory free ($L = 896$)

Figure 6.15. Memory free and usage on two sockets of a Skylake node.

Fig.6.15 presents the memory free and used when $L = 768$ and 896 on Skylake. Each socket on a Skylake node has $192GB/node \div 2sockets = 96GB/(socket \cdot node)$ DRAM. Similar conclusion can also be achieved. When $L = 768$, the allocated memory is 71.4GB $<$ 96GB DRAM per socket. In Fig.6.15c, socket 1 (red area) has used 3.4 95GB free memory, while socket 0 (blue area) is mainly in use and consumes 78.5GB memory in Fig.6.15a.When $L = 896$, the allocated memory is 113.3GB $>$ 96GB DRAM per socket. In Fig.6.15b, socket 1 has used average 28 GB memory, while in Fig.6.15d, the free memory on socket 1 has decreased from 92.6 GB to an average of 67.8 GB. This indicates that 8.2X more NUMA memory accesses have been involved, which incurs the performance slump of all algorithms at $L = 896$ in Fig.6.13b.

Table 6.3. Allocated memory of cubes and speedup in the strong scalability experiments on a Haswell node.

Edge size L	112	224	336	448	560	672	784	840
Cube memory (GB)	0.23	1.8	5.9	14	27.5	47.5	75.4	92.7
Max Speedup1	89.2%	48.1%	46.0%	37.9%	31.6%	31.4%	18.7%	23.8%
Max Speedup2	19.7%	14.1%	18.3%	21.1%	20.4%	14.2%	4.7%	9.3%

Speedup1 = (2-step prism eqv / Palabos Fuse prism - 1) ×100%
Speedup2 = (2-step prism / Palabos Fuse prism - 1) ×100%

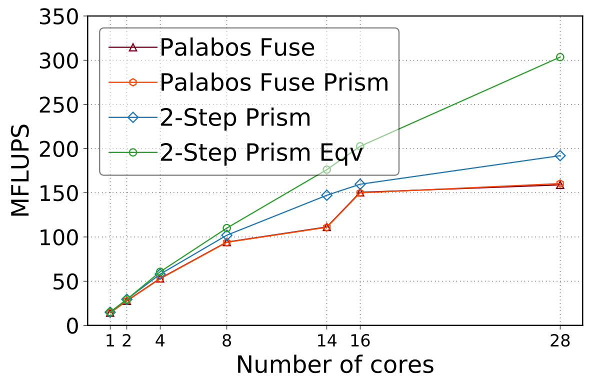
6.5.3 Performance of Parallel 3D Memory-aware LBM

Given N cores, Palabos LBM solvers partition the simulation domain evenly along three axes by $N_z \times N_y \times N_x = N$ MPI processes, which follows the underlying memory layout of cells along the axis of Z, then Y, and X at last. But our 3D memory-aware LBM partitions a domain only along X-axis by N OpenMP threads. Hence, Palabos LBM solvers have a smaller Y-Z layer size per core than our algorithm, thus have closer memory affinity, especially in a large domain. To exclude the factor caused by different partition methods, when the input of Palabos LBM solvers still uses cubes, 3D memory-aware LBM will take two different inputs. Firstly, it takes the input of the "equivalent dimension" of those cubes, such that a thread in our algorithm and a process in Palabos will compute a sub-domain with the same dimension after the respective partition method. Secondly, it simply takes the identical input of those cubes.

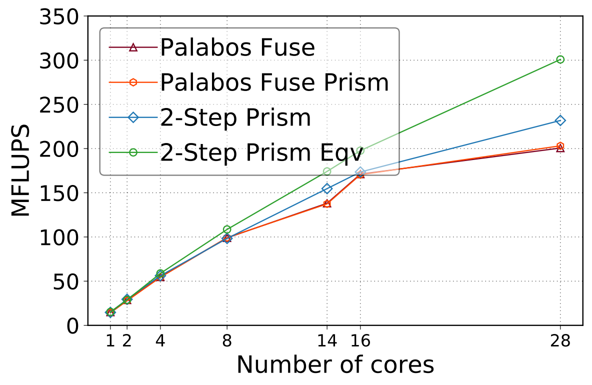
Fig.6.16 shows the strong scalability on a Haswell node. The input of Palabos LBM solvers uses cubes with edge size $L = 112 \sim 840$. Tab.6.3 presents the allocated memory of cubes and the maximum speedup achieved by 2-step prism LBM when using equivalent

Table 6.4. Equivalent input used by 2-step prism LBM when the input of Palabos LBM solvers is a cube with $L = 840$ on a Haswell node.

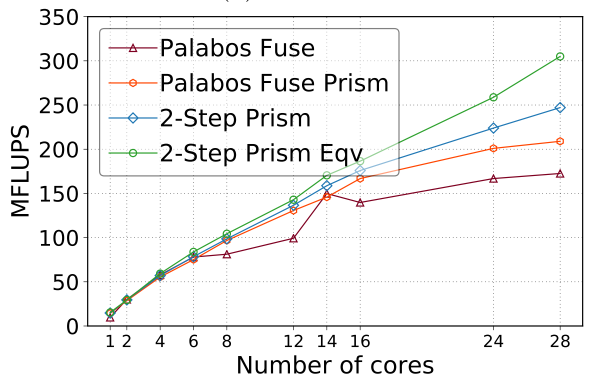
Cores	1	2	4	6	8	10	12	14	20	24	28
lx (height)	840	1680	3360	5040	3360	8400	5040	11760	8400	10080	11760
ly (width)	840	840	420	420	420	420	420	420	420	420	420
lz (length)	840	420	420	280	420	168	280	120	168	140	120



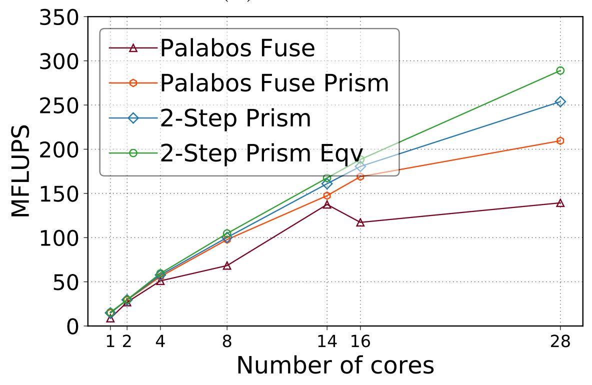
(a) $L = 112$.



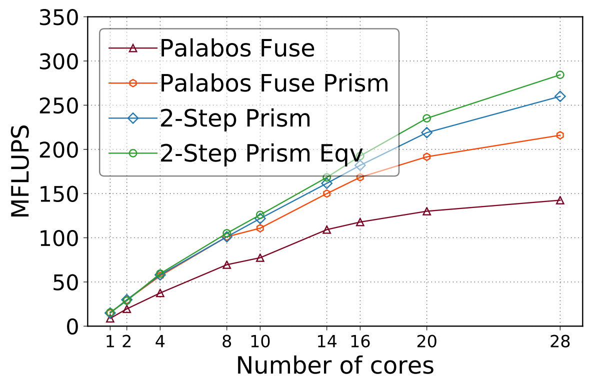
(b) $L = 224$.



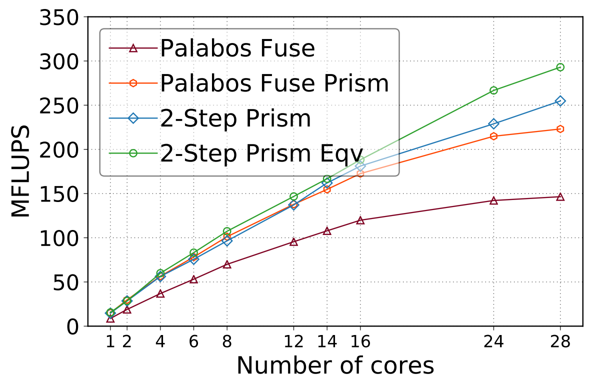
(c) $L = 336$.



(d) $L = 448$.



(e) $L = 560$.

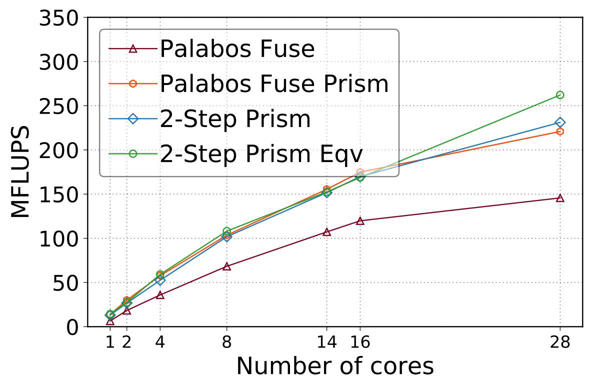


(f) $L = 672$.

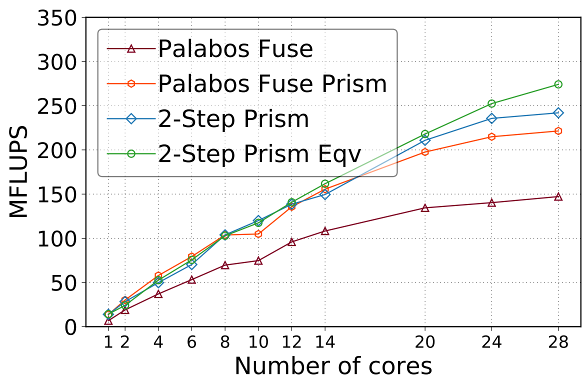
Figure 6.16. Haswell Strong Scalability performance. "2-step prism eqv" = Parallel 3D memory-aware LBM takes the equivalent input of cubes.

input and identical input. Tab.6.4 gives an example of the equivalent input used by 3D memory-aware LBM when Palabos LBM solvers use a cube with $L = 840$ on a Haswell node.

Figure 6.16. Continued.



(a) $L = 784$.



(b) $L = 840$.

Table 6.5. Allocated memory of cubes in the strong scalability experiments on a Skylake node.

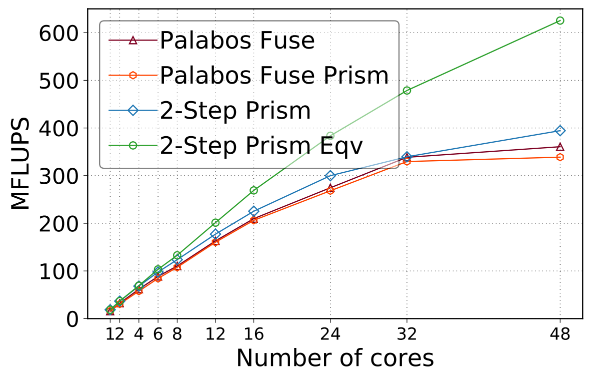
Edge size L	192	384	576	768	864	960
Cube memory (GB)	1.1	8.8	29.9	70.9	100.9	138.4
Max Speedup1	84.6%	51.2%	64.2%	70.7%	50.6%	34.2%
Max Speedup2	16.4%	22.8%	54.7%	54.3%	32.8%	28.7%

Speedup1 = (2-step prism eqv / Palabos Fuse prism - 1) ×100%
Speedup2 = (2-step prism / Palabos Fuse prism - 1) ×100%

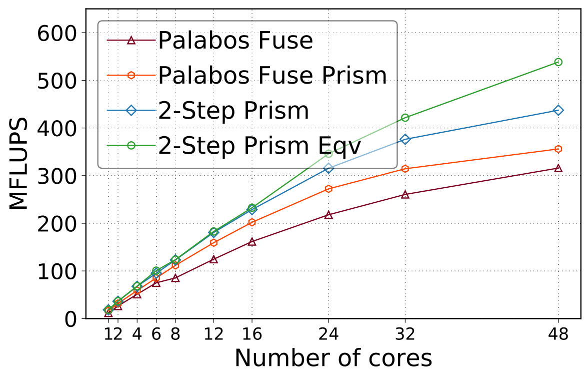
Table 6.6. Equivalent input used by the 2-step prism LBM when Fuse prism LBM is given a 960 × 960 × 960 cube on a Skylake node.

Cores	1	2	4	6	8	10	12	16	20	24	30	32	40	48
lx (height)	960	1920	3840	5760	3840	9600	5760	7680	9600	11520	14400	15360	19200	23040
ly (width)	960	960	480	480	480	480	480	480	480	480	320	240	480	240
lz (length)	960	480	480	320	480	192	320	240	192	160	192	240	96	160

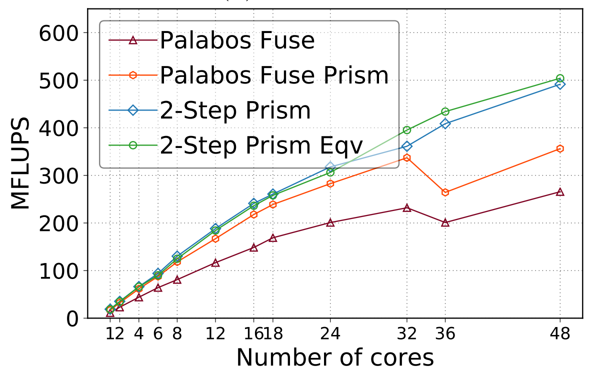
Fig.6.17 shows the strong scalability on a Skylake node. The input of Palabos LBM solvers use cubes with edge size $L = 192 \sim 960$. Tab.6.5 presents the allocated memory of cubes and the maximum speedup achieved by 2-step prism LBM when using equivalent input and identical input. Tab.6.6 gives an example of the equivalent input used by 3D memory-aware LBM when Palabos LBM solvers use a cube with $L = 960$ on a Skylake node.



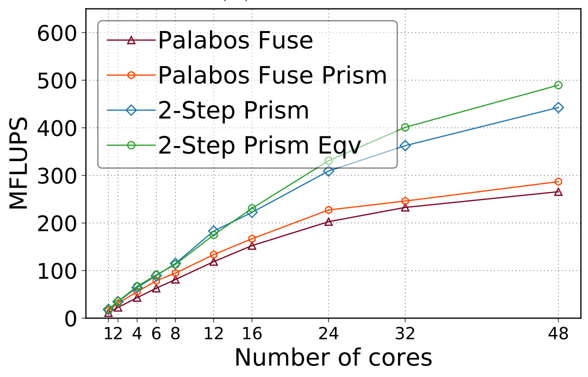
(a) $L = 192$.



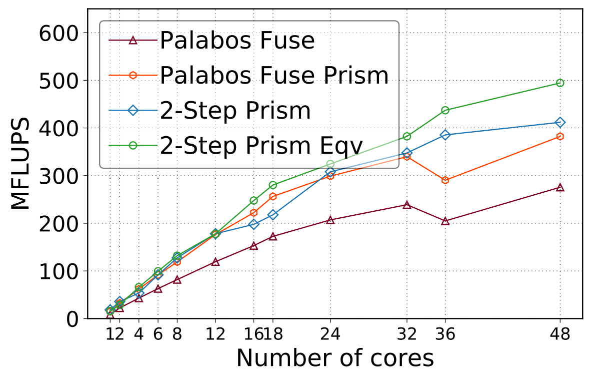
(b) $L = 384$.



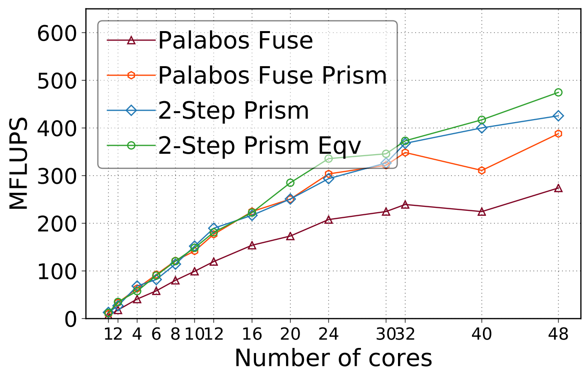
(c) $L = 576$.



(d) $L = 768$.



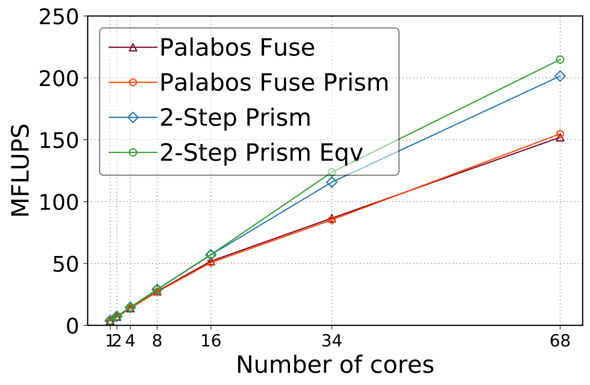
(e) $L = 864$.



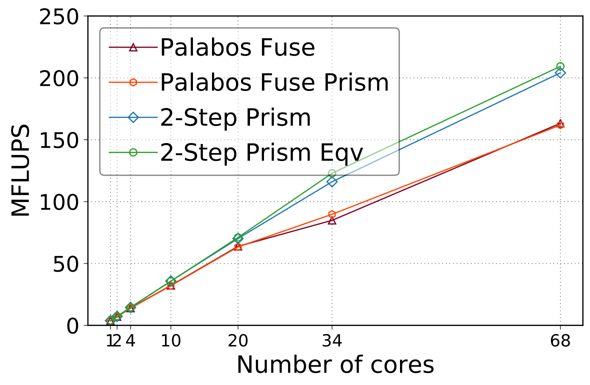
(f) $L = 960$.

Figure 6.17. Skylake Strong Scalability performance. “2-step prism eqv” = Parallel 3D memory-aware LBM takes the equivalent input of cubes.

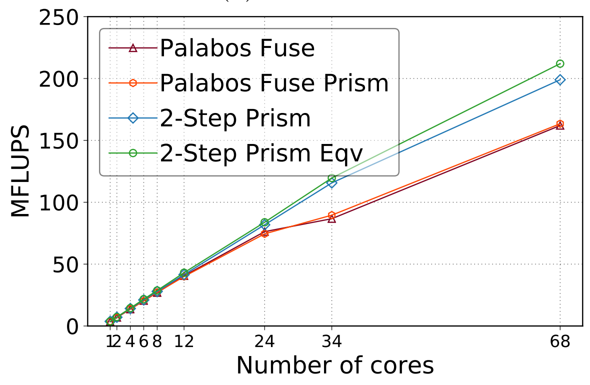
Fig.6.18 shows the strong scalability on a KNL node. The input of Palabos LBM solvers use cubes with edge size $L = 272 \sim 680$. Tab.6.7 presents the allocated memory of cubes and the maximum speedup achieved by 2-step prism LBM when using equivalent input and



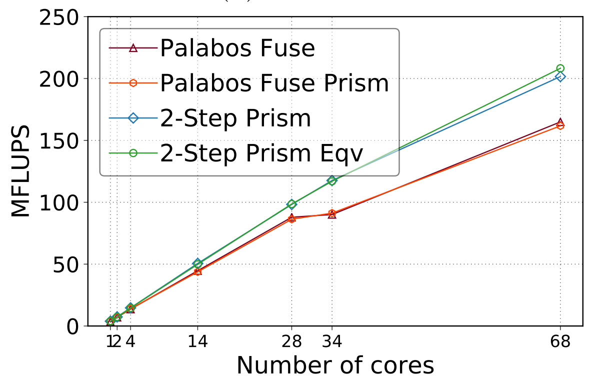
(a) $L = 272$.



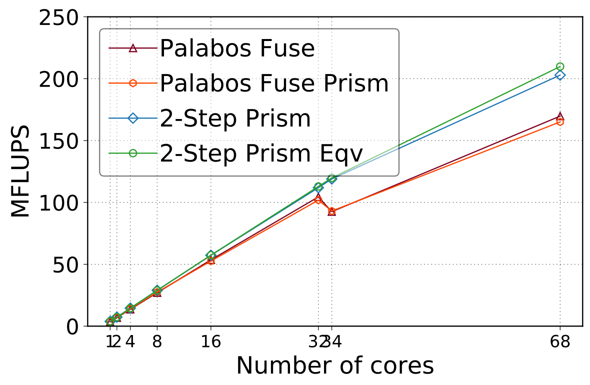
(b) $L = 340$.



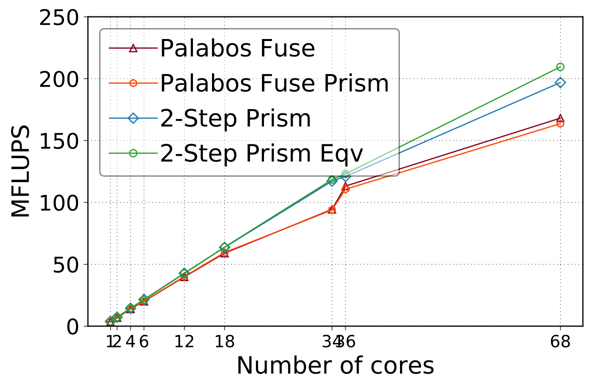
(c) $L = 408$.



(d) $L = 476$.



(e) $L = 544$.

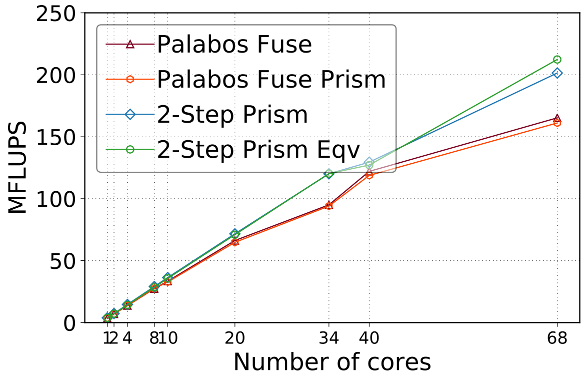


(f) $L = 612$.

Figure 6.18. Knight Landing scalability performance. "2-step prism eqv" = Parallel 3D memory-aware LBM takes the equivalent input of cubes.

identical input. Tab.6.6 gives an example of the equivalent input used by 3D memory-aware LBM when Palabos LBM solvers use a cube with $L = 680$ on a KNL node.

Figure 6.18. Continued.



(a) $L = 680$.

Table 6.7. Allocated memory size of cubes on a KNL node.

Edge size L	272	340	408	476	544	612	680
Cube mem(GB)	3.1	6.2	10.6	16.9	25.2	35.9	49.2
Max Speedup1	38.8%	29.3%	29.7%	28.8%	27.1%	28.0%	31.8%
Max Speedup2	35.7%	29.3%	29.1%	28.9%	27.9%	24.8%	27.7%

Speedup1 = (2-step prism eqv / Palabos Fuse prism - 1) $\times$100%
Speedup2 = (2-step prism / Palabos Fuse prism - 1) $\times$100%

Table 6.8. Equivalent input used by 2-step prism LBM when the Fuse prism LBM is given a $680 \times 680 \times 680$ cube on a Knight Landing node.

Cores	1	2	4	8	10	20	34	40	68
lx (height)	680	1360	2720	2720	6800	6800	23120	13600	23120
ly (width)	680	680	340	340	340	340	340	340	340
lz (length)	680	340	340	340	136	136	40	68	40

We observe that the 2-step prism LBM scales efficiently and always achieves the best performance in all cases. (1) When using the equivalent input of cubes on three compute nodes, for small scale cubes (with $L = 112, 192, 272$) in Fig.6.16a.6.17a.6.18a, 3D memory-aware LBM (green legend) is faster than the second-fastest Palabos (Fuse Prism) by up to 89.2%, 84.6%, and 38.8%, respectively. In Fig.6.16d.6.17c.6.18d, for the middle scale cubes (with $L = 448, 576, 476$), it is still faster than Palabos (Fuse Prism) by 37.9%, 64.2%, and

28.8% on three compute nodes, respectively. In Fig.6.17b.6.17f.6.19a, for the large scale cubes (with $L = 840, 960, 680$), it is still faster than Palabos (Fuse Prism) by 34.2%, 34.2%, and 31.8%, respectively. (2) When using the identical input of cubes, although our 3D memory-aware LBM has larger Y-Z layer sizes, it is still faster than Palabos (Fuse Prism) but with less speedup than before, i.e., by up to 21.1%, 54.7%, and 30.1% on the Haswell, Skylake and KNL node, respectively. The less speedup suggests our future work to partition a domain with smaller Y-Z layer sizes.

7. SUMMARY & FUTURE WORK

This dissertation firstly studies the scientific workflows that combine large-scale simulations with big data analysis using the present I/O and data transfer libraries. The trace analyses reveal that there are significant performance inefficiencies in the current practices (such as remote server and metaserver read/write time, coarse-grain critical sections, interlock between applications, barriers, and application stalls). With the aim of minimizing the end-to-end time of scientific workflows, a novel Zipper in-situ runtime system has been designed and implemented. It combines the parallelism of fine-grain task, pipelining, and asynchrony to seamlessly intertwine the simulation and analysis workflow such that the time-to-solution is merely one stage of time. Based on the experiment results, Zipper workflow obtains the fastest end-to-end time, and is verified by a proposed performance model. Additionally, the concurrent data transfer optimization method can reduce the stall time of the simulation application when the simulation is coupled with relatively slow data analysis. The experiments with the real-world CFD and LAMMPS workflows show that the Zipper approach is able to outperform the Decaf method — which is the second-fastest in-situ methods — by up to 2.2 times. A set of subsequent traces also reveal that the reduced idle/stall time, the lesser interference with the simulation time, and the full overlapping of each workflow stages have contributed the most to Zipper's enhanced end-to-end workflow time.

For the future works of Zipper, there is a new research direction called ”in situ algorithms”. Since we currently use a prior knowledge about what we want to analyze and where to send data, the visualization/analysis is scheduled beforehand. But for exploration scenarios that we do not know what to visualize/analysis, we can add the following features to Zipper: study the data in each cycle of workflows and decide when to trigger visualization/analysis, keep balance between reduction and data integrity, and complement traditional algorithm specifically for in-situ setting. Besides, the idea to asynchronously transfer fine-grain data could apply to the distributed workflows, e.g., between HPC to cloud, etc.

To address the memory bound limitation of Lattice Boltzmann method in manycore systems, we design the novel 2D and 3D memory-aware LBM algorithms. Then, we provide a detailed algorithm analysis to demonstrate how they enable more data reuses across multiple time steps. For the 2D case, the sequential and strong scalability experiments show that our 2D memory-aware LBM outperforms the Fuse LBM by up to 358% faster on the *Haswell* system, up to 401% faster on the Skylake system, and up to 162% on the Knight Landing system. In the 3D case, our 3D memory-aware LBM outperforms Fuse prism LBM by up to 89.2% faster on a *Hawell* node, up to 84.6% faster on a Skylake node and up to 38.8% on a Knight Landing node. Moreover, we use the Roofline model to give an insight into the speedup reasons. Our future work is to explore the 3D distributed memory-aware LBM across a large scale of nodes.

REFERENCES

[1] U.S. Department of Energy, Office of Science.Exascale Computing Project, *https://www.exascaleproject.org/exascale-computing-project/*, 2020.

[2] J. Slotnick, A. Khodadoust, J. Alonso, D. Darmofal, W. Gropp, E. Lurie, and D. Mavriplis, "CFD vision 2030 study: a path to revolutionary computational aerosciences," 2014.

[3] K.-L. Ma, "In situ visualization at extreme scale: Challenges and opportunities," *Computer Graphics and Applications, IEEE*, vol. 29, no. 6, pp. 14–19, 2009.

[4] J. Chen, A. Choudhary, S. Feldman, B. Hendrickson, C. Johnson, R. Mount, V. Sarkar, V. White, and D. Williams, "Synergistic challenges in data-intensive science and exascale computing," *DOE ASCAC Data Subcommittee Report, Department of Energy Office of Science*, 2013.

[5] Fugaku system overview, *https://www.top500.org/system/179807/*, 2020.

[6] Fugaku specifications, *https://www.fujitsu.com/global/about/innovation/fugaku/specifications/*, 2020.

[7] E. P. Duque, B. J. Whitlock, S. M. Legensky, C. P. Stone, R. Ranjan, and S. Menon, "The impact of in situ data processing and analytics upon scaling of cfd solvers and workflows," in *27th International Conference on Parallel Computational Fluid Dynamics, Montreal, Canada*, 2015.

[8] A. S. Szalay, "From large simulations to interactive numerical laboratories," *IEEE Data Eng. Bull.*, vol. 36, no. 4, pp. 41–53, 2013.

[9] H. Childs, S. D. Ahern, J. Ahrens, A. C. Bauer, J. Bennett, E. W. Bethel, P.-T. Bremer, E. Brugger, J. Cottam, M. Dorier, *et al.*, "A terminology for in situ visualization and analysis systems," *The International Journal of High Performance Computing Applications*, vol. 34, no. 6, pp. 676–691, 2020.

[10] E. Deelman, T. Peterka, I. Altintas, C. D. Carothers, K. K. van Dam, K. Moreland, M. Parashar, L. Ramakrishnan, M. Taufer, and J. Vetter, "The future of scientific workflows," *The International Journal of High Performance Computing Applications*, vol. 32, no. 1, pp. 159–175, 2018.

[11] M. Dreher and T. Peterka, "Decaf: Decoupled dataflows for in situ high-performance workflows," Argonne National Lab.(ANL), Argonne, IL (United States), Tech. Rep., 2017.

[12] M. Dorier, R. Sisneros, T. Peterka, G. Antoniu, and D. Semeraro, "Damaris/viz: A nonintrusive, adaptable and user-friendly in situ visualization framework," in *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, IEEE, 2013, pp. 67–75.

[13] V. Vishwanath, M. Hereld, M. Papka, R. Hudson, G. Jordan IV, and C. Daley, "In Situ Data Analysis and I/O Acceleration of FLASH Astrophysics Simulation on Leadership-Class System Using GLEAN," in *Proc. SciDAC, Journal of Physics: Conference Series*, 2011.

[14] N. Fabian, K. Moreland, D. Thompson, A. C. Bauer, P. Marion, B. Gevecik, M. Rasquin, and K. E. Jansen, "The paraview coprocessing library: A scalable, general purpose in situ visualization library," in *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, IEEE, 2011, pp. 89–96.

[15] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng, "Datastager: Scalable data staging services for petascale applications," *Cluster Computing*, vol. 13, no. 3, pp. 277–290, 2010.

[16] J. Bennett, H. Abbasi, P. Bremer, R. Grout, A. Gyulassy, T. Jin, *et al.*, "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, IEEE, 2012, pp. 1–9.

[17] M. Dorier, M. Dreher, T. Peterka, J. M. Wozniak, G. Antoniu, and B. Raffin, "Lessons learned from building in situ coupling frameworks," in *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, 2015, pp. 19–24.

[18] D. A. Reed and J. Dongarra, "Exascale computing and big data," *Communications of the ACM*, vol. 58, no. 7, pp. 56–68, 2015.

[19] E. M. National Academies of Sciences, *Future Directions for NSF Advanced Computing Infrastructure to Support U.S. Science and Engineering in 2017-2020.* Washington, DC: The National Academies Press, 2016. DOI: 10.17226/21886.

[20] G. Fox, J. Qiu, S. Jha, S. Ekanayake, and S. Kamburugamuve, "Big data, simulations and HPC convergence," in *Workshop on Big Data Benchmarks*, Springer, 2016, pp. 3–17.

[21] F. D. Witherden and A. Jameson, "Future directions in computational fluid dynamics," in *23rd AIAA Computational Fluid Dynamics Conference*, 2017, p. 3791.

[22] Towards Exascale Computing of Compressible Flows using LBM, *https://catalog.data.gov/dataset/towards-exascale-computing-of-compressible-flows-using-lbm-phase-i*, 2020.

[23] C. L. Rumsey, J. P. Slotnick, and A. J. Sclafani, "Overview and Summary of the Third AIAA High Lift Prediction Workshop," *Journal of Aircraft*, vol. 56, no. 2, pp. 621–644, 2019.

[24] Clay Mathematics Institute., *https://www.claymath.org/millennium-problems*, 2020.

[25] C. Coreixas, B. Chopard, and J. Latt, "Comprehensive comparison of collision models in the lattice Boltzmann framework: Theoretical investigations," *Physical Review E*, vol. 100, no. 3, p. 033 305, 2019.

[26] S. Succi, G. Amati, M. Bernaschi, G. Falcucci, M. Lauricella, and A. Montessori, "Towards exascale lattice Boltzmann computing," *Computers & Fluids*, vol. 181, pp. 107–115, 2019.

[27] H. Si, Y. Shi, and B. Wang, "LBM/LES for the study of fluid flow with high Reynolds numbers," in *Mechanics and Mechatronics (ICMM2015) Proceedings of the 2015 International Conference on Mechanics and Mechatronics (ICMM2015)*, World Scientific, 2016, pp. 142–150.

[28] X. Zhou, B. Dong, C. Chen, and W. Li, "A thermal LBM-LES model in body-fitted coordinates: Flow and heat transfer around a circular cylinder in a wide Reynolds number range," *International Journal of Heat and Fluid Flow*, vol. 77, pp. 113–121, 2019.

[29] A. Pradhan and S. Yadav, "Large eddy simulation using lattice Boltzmann method based on sigma model," *Procedia Engineering*, vol. 127, pp. 177–184, 2015.

[30] M. Vardhan, J. Gounley, L. Hegele, E. W. Draeger, and A. Randles, "Moment representation in the lattice Boltzmann method on massively parallel hardware," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–21.

[31] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster, "Swift/T: Large-scale application composition via distributed-memory dataflow processing," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2013, pp. 95–102.

[32] L. Ramakrishnan, S. Poon, V. Hendrix, D. Gunter, G. Z. Pastorello, and D. Agarwal, "Experiences with user-centered design for the tigres workflow api," in *2014 IEEE 10th International Conference on e-Science*, IEEE, vol. 1, 2014, pp. 290–297.

[33] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, "Scientific workflow management and the Kepler system," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1039–1065, 2006.

[34] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, *et al.*, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[35] L. Bavoil, S. P. Callahan, P. J. Crossno, J. Freire, C. E. Scheidegger, C. T. Silva, and H. T. Vo, "Vistrails: Enabling interactive multiple-view visualizations," in *VIS 05. IEEE Visualization, 2005.*, IEEE, 2005, pp. 135–142.

[36] J. Goecks, A. Nekrutenko, J. Taylor, G. Team, *et al.*, "Galaxy: A comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences," *Genome biology*, vol. 11, no. 8, R86, 2010.

[37] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, *et al.*, "The Taverna workflow suite: Designing and executing workflows of web services on the desktop, web or in the cloud," *Nucleic acids research*, vol. 41, no. W1, W557–W561, 2013.

[38] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers, "Examining the challenges of scientific workflows," *Computer*, vol. 40, no. 12, pp. 24–32, 2007.

[39] R. Thakur, W. Gropp, and E. Lusk, "On implementing MPI-IO portably and with high performance," in *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, ACM, 1999, pp. 23–32.

[40] H. Abbasi, J. Lofstead, F. Zheng, K. Schwan, M. Wolf, and S. Klasky, "Extending I/O through high performance data services," in *IEEE International Conference on Cluster Computing and Workshops (CLUSTER'09)*, IEEE, 2009, pp. 1–10.

[41] C. Docan, M. Parashar, and S. Klasky, "DataSpaces: An interaction and coordination framework for coupled simulation workflows," *Cluster Computing*, vol. 15, no. 2, pp. 163–181, 2012.

[42] J. Dayal, D. Bratcher, G. Eisenhauer, K. Schwan, M. Wolf, X. Zhang, H. Abbasi, S. Klasky, and N. Podhorszki, "Flexpath: Type-based publish/subscribe system for large-scale science analytics," in *The 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, IEEE, 2014, pp. 246–255.

[43] K. Mattila, J. Hyväluoma, T. Rossi, M. Aspnäs, and J. Westerholm, "An efficient swap algorithm for the lattice Boltzmann method," *Computer Physics Communications*, vol. 176, no. 3, pp. 200–210, 2007.

[44] T. Pohl, M. Kowarschik, J. Wilke, K. Iglberger, and U. Rüde, "Optimization and profiling of the cache performance of parallel lattice Boltzmann codes," *Parallel Processing Letters*, vol. 13, no. 04, pp. 549–560, 2003.

[45] P. Bailey, J. Myre, S. D. Walsh, D. J. Lilja, and M. O. Saar, "Accelerating lattice Boltzmann fluid flow simulations using graphics processors," in *2009 international conference on parallel processing*, IEEE, 2009, pp. 550–557.

[46] M. Geier and M. Schoenherr, "Esoteric twist: An efficient in-place streaming algorithmus for the lattice Boltzmann method on massively parallel hardware," *Computation*, vol. 5, no. 2, p. 19, 2017.

[47] S. Liu, N. Zou, Y. Cui, and W. Wu, "Accelerating the parallelization of lattice Boltzmann method by exploiting the temporal locality," in *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*, IEEE, 2017, pp. 1186–1193.

[48] G. Crimi, F. Mantovani, M. Pivanti, S. F. Schifano, and R. Tripiccione, "Early experience on porting and running a lattice Boltzmann code on the xeon-phi co-processor," *Procedia Computer Science*, vol. 18, pp. 551–560, 2013.

[49] J. Habich, C. Feichtinger, H. Köstler, G. Hager, and G. Wellein, "Performance engineering for the lattice Boltzmann method on GPGPUs: Architectural requirements and performance results," *Computers & Fluids*, vol. 80, pp. 276–282, 2013.

[50] N.-P. Tran, M. Lee, and S. Hong, "Performance optimization of 3D lattice Boltzmann flow solver on a GPU," *Scientific Programming*, vol. 2017, 2017.

[51] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske, "Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization," in *Computer Software and Applications Conference, 2009. COMPSAC'09. 33rd Annual IEEE International*, IEEE, vol. 1, 2009, pp. 579–586.

[52] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, and R. Oldfield, "Hello ADIOS: The challenges and lessons of developing leadership class I/O frameworks," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 7, pp. 1453–1473, 2014.

[53] F. Zhang, *Programming and runtime support for enabling data-intensive coupled scientific simulation workflows (Phd dissertation)*. Rutgers The State University of New Jersey-New Brunswick, 2015.

[54] U. Ayachit, "The paraview guide: A parallel visualization application," 2015.

[55] Y. Fu, F. Li, F. Song, and Z. Chen, "Performance analysis and optimization of in-situ integration of simulation with data analysis: Zipping applications up," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, ACM, 2018, pp. 192–205.

[56] Y. Fu, F. Song, and L. Zhu, "Modeling and Implementation of an Asynchronous Approach to Integrating HPC and Big Data Analysis," in *International Conference on Computational Science (ICCS-2016)*, San Diego, CA, Jun. 2016.

[57] Y. Fu and F. Song, "SDN helps Big Data to optimize access to data," in *Big Data and Software Defined Networks*, J. Taheri, Ed., Stevenage: The Institution of Engineering and Technology, 2018, ch. 14, pp. 297–318.

[58] Y. Fu, F. Li, F. Song, and L. Zhu, "Designing a parallel memory-aware lattice Boltzmann algorithm on manycore systems," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, IEEE, 2018, pp. 97–106.

[59] NCAR Graphics., *http://ngwww.ucar.edu/*, 2020.

[60] VisIt, *https://visit.llnl.gov*, 2018.

[61] J. Biddiscombe, J. Soumagne, G. Oger, D. Guibert, and J.-G. Piccinali, "Parallel computational steering and analysis for hpc applications using a paraview interface and the hdf5 dsm virtual file driver," 2011.

[62] C. Johnson, S. Parker, and D. Weinstein, "Large-scale computational science applications using the SCIRun problem solving environment," *Proceedings of Supercomputer*, 2000.

[63] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netCDF: A high-performance scientific I/O interface," in *SC'03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, IEEE, 2003, pp. 39–39.

[64] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2: Advanced features of the message-passing interface.* MIT press, 1999.

[65] Ascent, *https://ascent.readthedocs.io/en/latest/*, 2021.

[66] DataSpaces Project, *http://dataspaces.org*, 2018.

[67] C. Docan, M. Parashar, and S. Klasky, “Dart: A substrate for high speed asynchronous data io,” in *Proceedings of the 17th international symposium on High performance distributed computing*, 2008, pp. 219–220.

[68] P. Subedi, P. Davis, S. Duan, S. Klasky, H. Kolla, and M. Parashar, “Stacker: An autonomic data movement engine for extreme-scale data staging-based in-situ workflows,” in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2018, pp. 920–930.

[69] F. Zheng, H. Zou, G. Eisenhauer, K. Schwan, M. Wolf, J. Dayal, T. Nguyen, J. Cao, H. Abbasi, and S. Klasky, “FlexIO: I/O middleware for location-flexible scientific data analytics,” in *IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS)*, IEEE, 2013, pp. 320–331.

[70] G. Eisenhauer, M. Wolf, H. Abbasi, and K. Schwan, “Event-based systems: Opportunities and challenges at exascale,” in *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, ACM, 2009.

[71] A. C. Bauer, H. Abbasi, J. Ahrens, H. Childs, B. Geveci, S. Klasky, K. Moreland, P. O’Leary, V. Vishwanath, B. Whitlock, *et al.*, “In situ methods, infrastructures, and applications on high performance computing platforms,” in *Computer Graphics Forum*, Wiley Online Library, vol. 35, 2016, pp. 577–597.

[72] K. Timm, H. Kusumaatmaja, A. Kuzmin, O. Shardt, G. Silva, and E. Viggen, “The lattice Boltzmann method: Principles and practice,” *Springer International Publishing AG Switzerland, ISSN*, pp. 1868–4521, 2016.

[73] P. A. Thompson and G. Beavers, *Compressible-fluid dynamics*, 1972.

[74] J. H. Ferziger, M. Perić, and R. L. Street, *Computational methods for fluid dynamics.* Springer, 2002, vol. 3.

[75] B. Baliga and S. Patankar, “A new finite-element formulation for convection-diffusion problems,” *Numerical Heat Transfer*, vol. 3, no. 4, pp. 393–409, 1980.

[76] H. Jasak, A. Jemcov, Z. Tukovic, *et al.*, “OpenFOAM: A C++ library for complex physics simulations,” in *International workshop on coupled methods in numerical dynamics*, IUC Dubrovnik Croatia, vol. 1000, 2007, pp. 1–20.

[77] Ansys Fluent, *https://www.ansys.com/products/fluids/ansys-fluent*, 2021.

[78] Ansys CFX, *https://www.ansys.com/products/fluids/ansys-cfx*, 2021.

[79] Starccm+, *https://www.plm.automation.siemens.com/global/en/products/simcenter/STAR-CCM.html*, 2021.

[80] COMSOL, *https://www.comsol.com/products*, 2021.

[81] B. Baliga and S. Patankar, "A new finite-element formulation for convection-diffusion problems," *Numerical Heat Transfer*, vol. 3, no. 4, pp. 393–409, 1980.

[82] D. Frenkel and B. Smit, *From algorithms to applications*, 1996.

[83] J. Hardy, Y. Pomeau, and O. De Pazzis, "Time evolution of a two-dimensional model system. i. invariant states and time correlation functions," *Journal of Mathematical Physics*, vol. 14, no. 12, pp. 1746–1759, 1973.

[84] U. Frisch, B. Hasslacher, and Y. Pomeau, "Lattice-gas automata for the Navier-Stokes equation," *Physical review letters*, vol. 56, no. 14, p. 1505, 1986.

[85] E. Manoha and B. Caruelle, "Summary of the lagoon solutions from the benchmark problems for airframe noise computations-iii workshop," in *21st AIAA/CEAS aeroacoustics conference*, 2015, p. 2846.

[86] P. Hoogerbrugge and J. Koelman, "Simulating microscopic hydrodynamic phenomena with dissipative particle dynamics," *EPL (Europhysics Letters)*, vol. 19, no. 3, p. 155, 1992.

[87] G. Bird, "Approach to translational equilibrium in a rigid sphere gas," *The Physics of Fluids*, vol. 6, no. 10, pp. 1518–1519, 1963.

[88] A. Malevanets and R. Kapral, "Mesoscopic model for solvent dynamics," *The Journal of chemical physics*, vol. 110, no. 17, pp. 8605–8613, 1999.

[89] D. Violeau, *Fluid mechanics and the SPH method: theory and applications*. Oxford University Press, 2012.

[90] D. Ricot, V. Maillard, and C. Bailly, "Numerical simulation of unsteady cavity flow using Lattice Boltzmann Method," in *8th AIAA/CEAS Aeroacoustics Conference & Exhibit*, 2002, p. 2532.

[91] S. S. Shende and A. D. Malony, "The TAU parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[92] Intel. (2018). "Intel trace analyzer and collector," [Online]. Available: https://software.intel.com/en-us/intel-trace-analyzer.

[93] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf, "Predata-preparatory data analytics on peta-scale machines," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, IEEE, 2010, pp. 1–12.

[94] Q. Sun, M. Romanus, T. Jin, H. Yu, P. Bremer, S. Petruzza, S. Klasky, and M. Parashar, "In-staging data placement for asynchronous coupling of task-based scientific workflows," in *International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, IEEE, 2016, pp. 2–9.

[95] F. Alali, F. Mizero, M. Veeraraghavan, and J. M. Dennis, "A measurement study of congestion in an infiniband network," in *Network Traffic Measurement and Analysis Conference (TMA), 2017*, IEEE, 2017, pp. 1–9.

[96] S.-A. Reinemo, T. Skeie, T. Sodring, O. Lysne, and O. Trudbakken, "An overview of QoS capabilities in InfiniBand, advanced switching interconnect, and Ethernet," *IEEE Communications Magazine*, vol. 44, no. 7, pp. 32–38, 2006.

[97] E. Gonsiorowski, C. D. Carothers, J. LaPre, P. Heidelberger, C. Minkenberg, and G. Rodriguez, "Using quality of service lanes to control the impact of raid traffic within a burst buffer," in *2017 Winter Simulation Conference (WSC)*, IEEE, 2017, pp. 932–943.

[98] B. Kim and J.-D. Kim, "Dynamic QoS Scheme for InfiniBand-Based Clusters," in *Advances in Computer Science and Ubiquitous Computing*, Springer, 2016, pp. 573–578.

[99] S. Kalayci, G. Dasgupta, L. Fong, O. Ezenwoye, and S. M. Sadjadi, "Distributed and Adaptive Execution of Condor DAGMan Workflows.," in *SEKE*, 2010, pp. 587–590.

[100] E. L. Lusk, S. C. Pieper, R. M. Butler, *et al.*, "More scalability, less pain: A simple programming model and its implementation for extreme computing," *SciDAC Review*, vol. 17, no. 1, pp. 30–37, 2010.

[101] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak, "Intel® Omni-path architecture: Enabling scalable, high performance fabrics," in *The 23rd IEEE Annual Symposium on High-Performance Interconnects (HOTI)*, IEEE, 2015, pp. 1–9.

[102] PAPI project, *http://icl.utk.edu/papi/*, 2018.

[103] Intel, *Intel omni-path fabric suite fabric manager gui user guide*, version 1.0, Nov. 2015.

[104] Z. Guo and C. Shu, *Lattice Boltzmann method and its applications in engineering*. World Scientific, 2013.

[105] L. Zhu, D. Tretheway, L. Petzold, and C. Meinhart, “Simulation of fluid slip at 3D hydrophobic microchannel walls by the lattice Boltzmann method,” *Journal of Computational Physics*, vol. 202, no. 1, pp. 181–195, 2005.

[106] J. Schumacher, “Derivative moments in stationary homogeneous shear turbulence,” *Journal of Fluid Mechanics*, vol. 441, pp. 109–118, 2001.

[107] J. L. Lumley, *Stochastic tools in turbulence*. Courier Corporation, 2007.

[108] S. Dormido-Canto, J. Vega, J. Ramírez, A. Murari, R. Moreno, J. López, A. Pereira, and J.-E. Contributors, “Development of an efficient real-time disruption predictor from scratch on JET and implications for ITER,” *Nuclear Fusion*, vol. 53, no. 11, p. 113 001, 2013.

[109] T. Miyoshi, M. Kunii, J. Ruiz, G.-Y. Lien, S. Satoh, T. Ushio, K. Bessho, H. Seko, H. Tomita, and Y. Ishikawa, ““Big Data Assimilation” revolutionizing severe weather prediction,” *Bulletin of the American Meteorological Society*, vol. 97, no. 8, pp. 1347–1354, 2016.

[110] E. P. Gross and M. Krook, “Model for collision processes in gases: Small-amplitude oscillations of charged two-component systems,” *Physical Review*, vol. 102, no. 3, p. 593, 1956.

[111] R. R. Nourgaliev, T.-N. Dinh, T. G. Theofanous, and D. Joseph, “The lattice Boltzmann equation method: Theoretical interpretation, numerics and implications,” *International Journal of Multiphase Flow*, vol. 29, no. 1, pp. 117–169, 2003.

[112] G. Gompper, T. Ihle, D. Kroll, and R. Winkler, “Multi-particle collision dynamics: A particle-based mesoscale simulation approach to the hydrodynamics of complex fluids,” *Advanced computer simulation approaches for soft matter sciences III*, pp. 1–87, 2009.

[113] B. Dünweg, U. D. Schiller, and A. J. Ladd, “Statistical mechanics of the fluctuating lattice Boltzmann equation,” *Physical Review E*, vol. 76, no. 3, p. 036 704, 2007.

[114] A. J. Ladd, “Numerical simulations of particulate suspensions via a discretized Boltzmann equation. part 1. theoretical foundation,” *Journal of fluid mechanics*, vol. 271, pp. 285–309, 1994.

[115] E. M. Viggen, “The lattice Boltzmann method: Fundamentals and acoustics,” 2014.

[116] Palabos, *http://www.palabos.org/*, 2016.

[117] R. Allen and K. Kennedy, *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann San Francisco, 2002, vol. 1.

[118] S. Williams, "Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore," *ACM Communications*, 2009.

[119] STREAM, *https://www.cs.virginia.edu/stream/*, 2021.

[120] Intel. (2021). "Intel advisor," [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/advisor.html.

[121] J. Latt, "Technical report: How to implement your DdQq dynamics with only q variables per node (instead of 2q)," 2007.

[122] J. Latt, C. Coreixas, and J. Beny, "Cross-platform programming model for many-core lattice Boltzmann simulations," *arXiv preprint arXiv:2010.11751*, 2020.

[123] R. Argentini, A. Bakker, and C. Lowe, "Efficiently using memory in lattice Boltzmann simulations," *Future Generation Computer Systems*, vol. 20, no. 6, pp. 973–980, 2004.

[124] A. P. Randles, V. Kale, J. Hammond, W. Gropp, and E. Kaxiras, "Performance analysis of the lattice Boltzmann model beyond Navier-Stokes," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IEEE, 2013, pp. 1063–1074.

[125] D. J. Kerbyson, A. Hoisie, E. John, and J. Rubio, *A Performance Analysis of Two-Level Heterogeneous Processing Systems on Wavefront Algorithms*. CRC Press, 2007.

[126] J. Habich, T. Zeiser, G. Hager, and G. Wellein, "Enabling temporal blocking for a lattice Boltzmann flow solver through multicore-aware wavefront parallelization," in *21st International Conference on Parallel Computational Fluid Dynamics*, 2009, pp. 178–182.

[127] T. Zeiser, G. Wellein, A. Nitsure, K. Iglberger, U. Rude, and G. Hager, "Introducing a parallel cache oblivious blocking approach for the lattice Boltzmann method," *Progress in Computational Fluid Dynamics, an International Journal*, vol. 8, no. 1-4, pp. 179–188, 2008.

[128] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Foundations of Computer Science, 1999. 40th Annual Symposium*, IEEE, 1999, pp. 285–297.

[129] V. Heuveline and J. Latt, "The OpenLB project: An open source and object oriented implementation of lattice Boltzmann methods," *International Journal of Modern Physics C*, vol. 18, no. 04, pp. 627–634, 2007.

[130] J. Lätt and B. Chopard, "VLADYMIR—a C++ matrix library for data-parallel applications," *Future Generation Computer Systems*, vol. 20, no. 6, pp. 1023–1039, 2004.

[131] M. D. Mazzeo and P. V. Coveney, "HemeLB: A high performance parallel lattice-Boltzmann code for large scale fluid flow in complex geometries," *Computer Physics Communications*, vol. 178, no. 12, pp. 894–914, 2008.

[132] G. Zavodszky, B. van Rooij, V. Azizi, S. Alowayyed, and A. Hoekstra, "Hemocell: A high-performance microscopic cellular library," *Procedia Computer Science*, vol. 108, pp. 159–165, 2017.

[133] M. Hasert, K. Masilamani, S. Zimny, H. Klimach, J. Qi, J. Bernsdorf, and S. Roller, "Complex fluid simulations with the parallel tree-based lattice Boltzmann solver Musubi," *Journal of Computational Science*, vol. 5, no. 5, pp. 784–794, 2014.

[134] C. Feichtinger, S. Donath, H. Köstler, J. Götz, and U. Rüde, "WaLBerla: HPC software design for computational engineering simulations," *Journal of Computational Science*, vol. 2, no. 2, pp. 105–112, 2011.

[135] OpenMP, *http://www.openmp.org/*, 2018.

[136] Intel, *https://software.intel.com/content/www/us/en/develop/download/intel-xeon-processor-e5-and-e7-v3-family-uncore-performance-monitoring-reference-manual.html*, 2015.

[137] Avinash Sodani, *https://www.alcf.anl.gov/files/HC27.25.710-Knights-Landing-Sodani-Intel.pdf*, 2016.

[138] Akhilesh Kumar, *https://www.primeline-solutions.com/media/wysiwyg/news-presse/intel-xeon-scalable-architecture-deep-dive_1.pdf*, 2017.

[139] M. Van Dyke and M. Van Dyke, "An album of fluid motion," 1982.

[140] G. Rivera and C.-W. Tseng, "Tiling optimizations for 3d scientific computations," in *SC'00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, IEEE, 2000, pp. 32–32.

[141] Yuankun Fu, *https://github.com/qoofyk/mem-aware-lbm*, 2020.

[142] C. Rosales, A. Gómez-Iglesias, and A. Predoehl, "Remora: A resource monitoring tool for everyone," in *Proceedings of the Second International Workshop on HPC User Support Tools*, 2015, pp. 1–8.

A. Zipper Interfaces

We present how to use the interfaces of Zipper as follows.

A.1 Producer Module Interface

Fig.A.1 shows how to use the Zipper producer module to integrate with a simulation application. The first step is to setup the parameters of the in-situ workflow. `Graph g` contains the prior information about which group the process belongs to (e.g., simulation or analysis), `num_simulation_process`, `num_analysis_process`, and their mapping information (e.g., producer process rank 0 and 1 will send data to consumer process rank 0, etc.). We use `fine_grain_blk_size` and `producer_ringbuffer_size` to control the producer ring buffer size. The `writer_threshold` controls the threshold when the writer thread starts to steal blocks from producer ring buffer and then uses the dual channels to transfer data. The `preserve_flag` controls whether to store the intermediate simulation data. The second step is to start the simulation computation, when a block is generated in a time step t, users can call `Zipper.write(blk_id, blk, blk_size)`. Then Zipper chops this `blk` into to find grain blocks, and transfers the data to the corresponding analysis process. Finally, the simulation application calls its own clean-up code.

```
/* Step 1: In-situ workflow setup */
Graph g; // contains the priori workflow information
size_t fine_grain_blk_size;
size_t producer_ringbuffer_size, consumer_ringbuffer_size;
int writer_threshold;
bool preserve_flag; // whether or not store the simulation data

/* ... Simulation application initialization code ... */
Zipper.init(g, fine_grain_blk_size, producer_ringbuffer_size,
   consumer_ringbuffer_size, writer_threshold, preserve_flag)

/* Step 2: Simulation application starts */
for (long t = 0; t < total_time_steps; ++t) {
    /* ... Simulation computation code ... */
    /* Simulation generates a data block: long blk_id, void* blk, size_t
   blk_size*/
    Zipper.write(blk_id, blk, blk_size);
}

/* Step 3: Simulation application clean-up code */
```

Figure A.1. Use the Zipper Producer Module to integrate with a simulation application.

A.2 Consumer Module Interface

Fig.A.2 shows how to use the Zipper consumer module to integrate with an analysis application. Similar to the above producer procedure, there are three steps. Firstly, after setup the input parameters for the in-situ workflow, we use `fine_grain_blk_size` and `consumer_ringbuffer_size` to control the consumer ring buffer size. During the second step, users can call `blk = Zipper.read(&blk_id, &blk_size)` to get a block of data from the corresponding simulation process. Finally, the analysis application can call its own clean-up code.

```
/* Step 1: In-situ workflow setup */
Graph g;
size_t fine_grain_blk_size;
size_t producer_ringbuffer_size, consumer_ringbuffer_size;
int writer_threshold;
bool preserve_flag;

/* ... Analysis application initialization code ... */
Zipper.init(g, fine_grain_blk_size, producer_ringbuffer_size,
    consumer_ringbuffer_size, writer_threshold, preserve_flag)

/* Step 2: Analysis application starts */
for (long t = 0; t < total_time_steps; ++t) {
    void* blk;
    long blk_id
    size_t blk_size;
    blk = Zipper.read(&blk_id, &blk_size);
    long source_rank = ((long*)blk)[0]; // simulation process rank
    /* ... Analysis computation code ... */
}
/* Step 3: Analysis application clean-up code */
```

Figure A.2. Use the Zipper consumer module to integrate with an analysis application.

VITA

Yuankun Fu was born in Dalian, Liaoning, Peoples Republic of China in 1988. He graduated from Northeastern University (Shenyang, China) with B.S. degree in Electronic and Information Engineering in 2011. Then he graduated from Institute of Computing Technology, University of Chinese Academy of Sciences (Beijing, China) with M.S. degree in Computer Architecture in 2014. He is currently a Ph.D. candidate in the School of Computer Science at Purdue University. His Ph.D. research focuses on accelerating the integration of in-situ workflow and designing parallel algorithms for memory-bound CFD simulations.

PUBLICATIONS

1. Yuankun Fu, Fengguang Song, "*Designing a 3D Parallel Memory-Aware Lattice Boltzmann Algorithm on Manycore Systems*", **Euro-Par'21**, Lisbon Portugal, 09/2021.

2. Feng Li, Ranran Chen, Yuankun Fu, Fengguang Song, Yao Liang, Isuru Ranawaka, Sudhakar Pamidighantam, Daniel Luna, Xu Liang, "*Accelerating complex modeling workflows in CyberWater using on-demand HPC/Cloud resources*" (in submission).

3. Slaughter Elliott, Wei Wu, Yuankun Fu, Legend Brandenburg, Nicolai Garcia, Wilhem Kautz, Emily Marx, Kaleb S. Morris, Qinglei Cao, George Bosilca, Seema Mirchandaney, Wonchan Lee, Sean Treichler, Patrick McCormick, Alex Aiken, "*Task Bench: A Parameterized Benchmark for Evaluating Parallel Runtime Performance*", **SC'20**: International Conference for High Performance Computing, Networking, Storage and Analysis, Atlanta, Georgia, 11/2020. [PDF]

4. Yuankun Fu, Feng Li, Fengguang Song, Luoding Zhu, "*Designing a Parallel Memory-Aware Lattice Boltzmann Algorithm on Manycore Systems*", **SBAC-PAD'18**: 2018 30th International Symposium on Computer Architecture and High Performance Computing, Lyon, France, 09/2018. [PDF]

5. Yuankun Fu, Feng Li, Fengguang Song, Zizhong Chen, "*Performance Analysis and Optimization of In-situ Integration of Simulation with Data Analysis: Zipping Applications Up*", **HPDC'18**: Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing, Tempe, Arizona, 06/2018. [PDF]

6. Yuankun Fu, Fengguang Song, "*SDN helps Big-Data to optimize access to data*", chapter 14, 297-318(504), Big Data and Software Defined Networks, Stevenage, UK, 03/2018. [PDF]

7. Yuankun Fu, Fengguang Song, Luoding Zhu, "*Modeling and Implementation of an Asynchronous Approach to Integrating HPC and Big Data Analysi*", **ICCS'16**: 2016 International Conference on Computational Science, San Diego, CA, 06/2016. [PDF]