Parallel Computing 67 (2017) 20–37

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

A scalable interface-resolved simulation of particle-laden flow using the lattice Boltzmann method

Nicholas Geneva[a], Cheng Peng[a], Xiaoming Li[b], Lian-Ping Wang[a,c,*]

[a] Department of Mechanical Engineering, University of Delaware, Newark, Delaware 19716-3140, USA
[b] Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware 19716-3140, USA
[c] State Key Laboratory of Coal Combustion, Huazhong University of Science and Technology, Wuhan, PR China

A R T I C L E I N F O

Article history:
Received 7 September 2016
Revised 5 July 2017
Accepted 18 July 2017
Available online 19 July 2017

Keywords:
Lattice-Boltzmann method
Particle-laden turbulent flow
Interface-resolved direct numerical simulation
Scalable performance
Data structure
Code optimization

A B S T R A C T

We examine the scalable implementation of the lattice Boltzmann method (LBM) in the context of interface-resolved simulation of wall-bounded particle-laden flows. Three distinct aspects relevant to performance optimization of our lattice Boltzmann simulation are studied. First, we optimize the core sub-steps of LBM, the collision and the propagation (or streaming) sub-steps, by reviewing and implementing five different published algorithms to reduce memory loading and storing requirements to boost performance. For each, two different array storage formats are benchmarked to test effective cache utilization. Second, the vectorization of the multiple-relaxation-time collision model is discussed and our vectorized collision and propagation algorithm is presented. We find that careful use of Intel's Advance Vector Extensions and appropriate array storage formats can significantly enhance performance. Third, in the presence of many finite-size, moving solid particles within the flow field, three different communication schemes are proposed and compared in order to optimize the treatment of fluid-solid interactions. These efforts together lead to a very efficient LBM simulation code for interface-resolved simulation of particle-laden flows. Overall, the optimized scalable code of particle-laden flow is a factor of 4.0-to-8.5 times faster than our previous implementation.

1. Introduction

The lattice Boltzmann method (LBM) is a kinetic approach based on solving the Boltzmann equation, whose moments are governed by the same continuity and Navier–Stokes equations for continuum fluids. Over the last three decades, LBM has been rapidly developed as an effective tool to numerically solve a variety of complex flows, including multiphase flows with fluid-fluid or fluid-solid interfaces [1–3]. The method is attractive due to the simplicity of the lattice Boltzmann equation (LBE), the locality of data communication, and the great feasibility to handle different boundary conditions. LBM tracks the distribution functions of a set of fluid lattice-particles, which evolve in two sub-steps: collision and propagation (or streaming). While simple in its mathematical form, LBM requires a large number of distribution functions to convert complex macroscopic physics to simpler mesoscopic physics, resulting in a significant amount of floating-point calculations and high memory usage. This intense computational requirement places a restriction on LBM's overall effectiveness. For large three-dimensional flow problems that consume a large amount of computational resources, high-performance computing

* Corresponding author at : Department of Mechanical Engineering, University of Delaware, Newark, Delaware 19716-3140, USA.
E-mail addresses: ngeneva@udel.edu (N. Geneva), cpengxpp@udel.edu (C. Peng), xli@udel.edu (X. Li), lwang@udel.edu (L.-P. Wang).

http://dx.doi.org/10.1016/j.parco.2017.07.005
0167-8191/© 2017 Published by Elsevier B.V.

must be employed in which the simulation is executed in parallel. Computationally optimized and scalable implementation of LBM is essential in order to make it competitive with conventional Computational Fluid Dynamics (CFD) methods based on solving directly the Navier–Stokes equations.

Many advances have been made to optimize LBM simulations on both CPU and more recently GPU clusters as reviewed in [4]. For CPU architectures, computational performance is heavily impacted by maximizing spatial locality and exploiting the full potential of cache hierarchy. Several groups have studied the impacts of the data storage format in which fluid lattice-particle distributions are saved and have found that this can have a major influence on performance [5–8]. Additionally various algorithms have been proposed to execute the two fundamental steps of LBM, namely, collision and propagation, more efficiently. Examples include data blocking, more optimal data access patterns, and tailoring of the overall update process for modern CPU architectures [9–11]. Another topic that has been studied in the past is the optimization of data communication, data storage, or optimal domain decomposition of complex flow geometry where standard approaches would result in wasted memory or poor partitioning of the fluid domain among parallel processes [12–15]. However these optimizations were not designed for multiphase flows in which complex fluid-fluid or fluid-solid interfaces move relative to the fluid lattice.

In this paper, we are interested in turbulent flows laden with finite-size solid particles which are exceedingly common among engineering applications and naturally-occurring phenomena. Examples include fluidized bed reactors, sediment transport, volcanic ash eruptions, and dust storms. One of the key advantages of LBM is its ability to directly resolve a moving fluid-solid interface through interpolated bounce back of fluid lattice-particles [16–18], allowing complex boundaries to be implemented with relative ease compared to conventional CFD methods. The representation of a solid particle can be approached in many different ways, however, most implementations boil down into three main components: bounce back of lattice-particles at the fluid-solid interface, specification of distribution functions at a newly uncovered fluid node or refilling, and calculation of hydrodynamic force and torque acting on the solid particle. Due to its algorithm complexity and additional data communication needs, the resulting codes for particle-laden flows are often more difficult to optimize than their single-phase counterparts.

There is very little past work done on optimizing simulation codes for flows laden with finite-size solid particles. Stratford et al. [19] discussed their optimizations for their particle-laden simulation that uses the bounce-back scheme [20] which resolves the solid particle boundary through mid-link bounce back. In their implementation, fluid nodes inside the solid particle are considered as part of fluid domain, which eliminates the need for a refilling scheme. Additionally each process or thread tracks fluid lattice-particles and solid particles that only reside in its local sub-domain plus an additional layer of ghost nodes or halo around the sub-domain. This allows the mid-link bounce back to be executed on the boundaries of solid particles with no additional data communication. While this implementation was able to achieve low solid particle overhead and good scalability, it is well known that more complex and expensive bounce back or refilling schemes involving interpolation or extrapolation are often required to ensure better physical accuracy and numerical stability [21].

Xiong et al. [22] presented their optimizations made for their LBM particle-laden code designed for general-purpose GPU computing. In their work, the single-relaxation-time or Bhatnagar–Gross–Krook (BGK) collision model with the D3Q19 lattice is used. The solid particle boundary is treated by the immersed boundary method (IBM). Several optimizations are proposed in this work including executing the propagation sub-step before the collision sub-step with the use of a propagation optimized data structure to achieve fast data transfer between the global memory and GPU. Additionally, lattice node control volumes are made spherical when calculating the local solid volume fraction for the IBM process as opposed to cuboidal cells to reduce computational load and complexity. Other optimizations are mentioned such as implementing concurrent kernel execution, however, such optimizations are specific to GPUs and do not apply to CPU clusters. While IBM is more popular in conventional methods which solve directly the Navier–Stokes equations, in our LBM simulation we prefer the interpolated bounce back scheme for three reasons: (a) to keep our method fully mesoscopic, (b) to eliminate any local smoothing (or regularization) errors which exist in IBM, and (c) to allow more precise calculation of local profiles conditioned on the solid particle surface [23].

Gao et al. [24] discussed the performance of their parallel simulation that uses a second-order interpolated bounce back scheme to resolve the solid particle boundary [16,25] and a non-equilibrium refilling scheme [26]. Using a one-dimensional domain decomposition scheme they were able to achieve good scalability. Additionally Gao et al. benchmarked the respective solid particle overhead of the simulation for two different domain resolutions and solid particle radii with a solid particle volume fraction of 10%. While the solid particle overhead varied depending on both volume fraction and particle size, in general the solid particle overhead was found to be between 20% and 26% for most test cases. This code was later updated to allow both 2D and 3D domain decompositions, in order to utilize more MPI tasks. Similar scalability and performance were observed [27].

In this work we will discuss the optimization of our parallel simulation code for particle-laden channel flow previously developed and validated in [28,29]. In Section 2 we briefly describe the LBM implementation used and the general structure/decomposition of our simulation. Next, in Section 3 we discuss the storage of the fluid lattice-particles in respect to memory addressing and array formats. We then focus on optimization of the collision and propagation process of LBM in Section 4 by reviewing algorithms proposed in past works to increase computational performance, conduct benchmarking and discuss our findings. Following, in Section 5 we investigate the vectorization of the multiple-relaxation-time collision model and look at the performance gains from the use of SSE (Streaming SIMD [single instruction multiple data] Extensions) and AVX (Advanced Vector Extensions) instructions. Lastly in Section 6 we focus on the optimization of the data communication

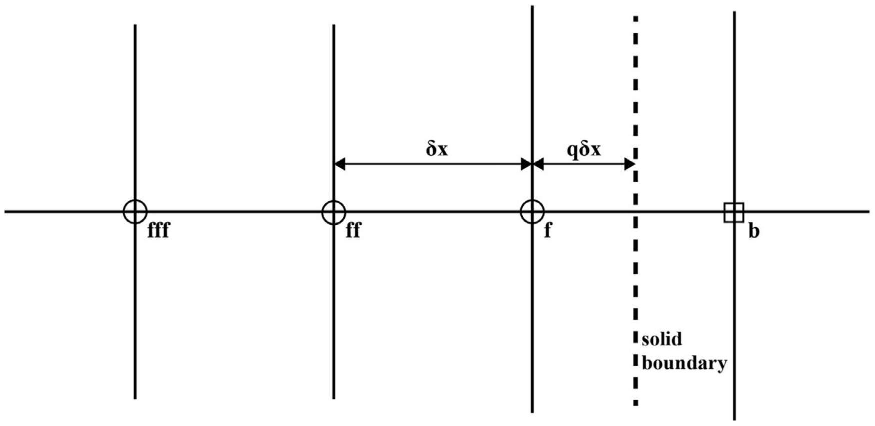


Fig. 1. Sketch to describe the details of the interpolated bounce back scheme.

related to both the bounce back at the moving fluid-solid interfaces and the refilling scheme. We then summarize the results of this work and discuss the potential future work. Throughout the entire paper, we present benchmarking results to critically examine the overall performance and scalability of the improved codes are presented. All simulations were conducted on the National Center for Atmospheric Research's (NCAR) supercomputer Yellowstone equipped with 2.6-GHz Intel Xeon E5-2670 (Sandy Bridge) processors.

2. Lattice Boltzmann method and flow configuration

Since this paper extends the methodology reported in our recent studies [27–29], here we shall only briefly describe LBM used in our simulations. For the flow evolution, the multiple-relaxation-time (MRT) LBE [30] is implemented. While the MRT collision model is computationally more expensive than the single-relaxation-time or BGK collision model, due to the calculation of the moments [31], MRT LBE provides greater control over relaxation parameters which allows better numerical stability [30]. In MRT LBE the fluid lattice-particle distributions are governed by

$$\boldsymbol{f}(\boldsymbol{x}+\mathbf{e}_\alpha\delta t, t+\delta t) = \boldsymbol{f}(\boldsymbol{x},\boldsymbol{t}) - \boldsymbol{M}^{-1}\mathbf{S}[\boldsymbol{m}-\boldsymbol{m}^{eq}] + \boldsymbol{Q}, \tag{1}$$

where $\boldsymbol{x}$ is lattice node location, δt is the time step size, the matrix $\boldsymbol{M}$ contains a set of orthogonal row vectors used to transform the distribution vector $\boldsymbol{f}$ into a set of independent moments $\boldsymbol{m}$, namely, $\boldsymbol{m} = \boldsymbol{Mf}$. The diagonal matrix $\boldsymbol{S}$ contains relaxation rates for all moments and the term $\boldsymbol{Q}$ denotes the mesoscopic forcing field. For this study we used the D3Q19 lattice model with 19 discrete velocities $\boldsymbol{e}_\alpha$ assigned to the following directions [30]

$$\boldsymbol{e}_\alpha = \begin{cases} (0,0,0), & \alpha = 0; \\ (\pm1,0,0),\ (0,\pm1,0),\ (0,0,\pm1), & \alpha = 1,2\ldots6; \\ (\pm1,\pm1,0),\ (\pm1,0,\pm1),\ (0,\pm1,\pm1), & \alpha = 7,8\ldots18. \end{cases} \tag{2}$$

The conserved macroscopic variables, density ρ, pressure p, and velocity $\mathbf{u}$, are obtained from the mesoscopic particle distribution functions for nearly incompressible flows [32] as

$$\begin{aligned} &\rho = \rho_0 + \delta\rho, \quad \rho_0 = 1, \quad \delta\rho = \textstyle\sum_\alpha f_\alpha, \\ &\rho_0\boldsymbol{u} = \textstyle\sum_\alpha f_\alpha \boldsymbol{e}_\alpha + \frac{\delta t}{2}\rho_0 \boldsymbol{F}(\boldsymbol{x},t), \quad p = \delta\rho c_s^2, \end{aligned} \tag{3}$$

where ρ_0 is the mean density, $\delta\rho$ is the local density fluctuation, c_s is the model speed of sound, and $\boldsymbol{F}$ is the physical-space forcing per unit mass [33].

To resolve the fluid-solid interface we implemented the quadratic interpolation scheme developed by Bouzidi et al. [16] due to its proven numerical stability and physical accuracy [21]. Bouzidi et al.'s scheme uses two different interpolation equations depending on the distance between the solid boundary and the fluid boundary node. Namely, if the percent distance q of the solid boundary along a link is less than or equal to 0.5, a pre-advection fluid particle distribution is interpolated as

$$f_{\bar\alpha}(x_f, t+\delta t) = q(2q+1)\tilde{f}_\alpha(x_f,t) + (1+2q)(1-2q)\tilde{f}_\alpha(x_{ff},t) - q(1-2q)\tilde{f}_\alpha(x_{fff},t) + 2w_\alpha\rho_0\frac{\mathbf{e}_{\bar\alpha}\cdot\mathbf{u}_w}{c_s^2}, \tag{4}$$

in which x_f, x_{ff}, and x_{fff} are depicted in Fig. 1, $\tilde{f}$ denotes distribution after the collision sub-step, $\mathbf{e}_\alpha$ points from the fluid node to the solid boundary, and $\mathbf{e}_{\bar\alpha} = -\mathbf{e}_\alpha$. The term $2w_\alpha\rho_0\mathbf{e}_{\bar\alpha}\cdot\mathbf{u}_w/c_s^2$ is derived from the equilibrium distribution that ensures the no-slip boundary condition when the solid boundary is moving, in which w_α the directional weighting factor and $\mathbf{u}_w$ is the wall velocity. If the percent distance q is larger than 0.5, a post-advection fluid particle distribution is interpolated as

$$f_{\bar{\alpha}}(x_f, t+\delta t) = \frac{1}{q(2q+1)}\left(\tilde{f}_{\alpha}(x_f, t) + 2w_{\alpha}\rho_0\frac{\mathbf{e}_{\bar{\alpha}}\cdot\mathbf{u}_w}{c_s^2}\right) + \frac{2q-1}{q}\tilde{f}_{\bar{\alpha}}(x_f, t) + \frac{1-2q}{1+2q}\tilde{f}_{\bar{\alpha}}(x_{ff}, t)\ . \tag{5}$$

With the current expressions, interpolation occurs in the post-collision, pre-streaming state. However, when using algorithms that fuse collision and streaming, this state is not always accessible. Thus the post streaming distributions are used instead, namely,

$$\begin{aligned} f_{\bar{\alpha}}(x_f, t+\delta t) &= q(2q+1)f_{\alpha}(x_b, t+\delta t) + (1+2q)(1-2q)f_{\alpha}(x_f, t+\delta t) \\ &\quad - q(1-2q)f_{\alpha}(x_{ff}, t+\delta t) + 2w_{\alpha}\rho_0\frac{\mathbf{e}_{\bar{\alpha}}\cdot\mathbf{u}_w}{c_s^2}, \end{aligned} \tag{6}$$

and

$$\begin{aligned} f_{\bar{\alpha}}(x_f, t+\delta t) &= \frac{1}{q(2q+1)}\left(f_{\alpha}(x_b, t+\delta t) + 2w_{\alpha}\rho_0\frac{\mathbf{e}_{\bar{\alpha}}\cdot\mathbf{u}_w}{c_s^2}\right) \\ &\quad + \frac{2q-1}{q}f_{\bar{\alpha}}(x_{ff}, t+\delta t) + \frac{1-2q}{1+2q}f_{\bar{\alpha}}(x_{fff}, t+\delta t), \end{aligned} \tag{7}$$

for q less than or equal to 0.5 and q greater than 0.5, respectively, with x_b, x_f, x_{ff}, and x_{fff} again depicted in Fig. 1.

When a solid particle moves relative to the fluid lattice, a new fluid node may emerge. The velocity-constrained refilling scheme recently developed by Peng et al. [21] is used to repopulate unknown distributions at these new fluid nodes. This approach ensures that the appropriate no-slip boundary conditions are met, which has been found to significantly reduce the unphysical pressure noise in particle-laden flow simulations [21]. To fill distributions at a new fluid node x_f, the discrete velocity direction closest to the outward normal, $\boldsymbol{e}_n$, is determined and the quadratic extrapolation is used to obtain the missing fluid particle distributions, namely,

$$\tilde{f}_{\alpha}(x_f, t+\delta t) = 3f_{\alpha}(x_f + e_n\delta t, t+\delta t) - 3f_{\alpha}(x_f + 2e_n\delta t, t+\delta t) + f_{\alpha}(x_f + 3e_n\delta t, t+\delta t). \tag{8}$$

Next the moments for these extrapolated distributions are calculated. The density moment is set equal to the average density of surrounding fluid nodes, and the velocity moments are constrained to the local wall velocity on the solid particle surface. These modified moments are then multiplied by the inverse transformation matrix $\boldsymbol{M}^{-1}$ to obtain the final distributions for the fluid node. The use of the MRT collision model allows all other moments (such as local shear stress tensor) to remain untouched while constraining the velocities to satisfy the no-slip boundary condition. The order in which these processes are executed within the simulation can be seen in Algorithm 1 below. Additional implementation details and validation are

Algorithm 1 LBM particle-laden main loop.

```
for $i = 0$ do nsteps
    CollisionMRT                              ▷ Execute MRT collision
    Streaming                                 ▷ Stream fluid lattice-particles
    if ipart then                             ▷ If solid particles are on
        sp-Collision                          ▷ Interpolated bounce-back
        sp-Lubforce                           ▷ Calculate hydro-dynamic forces
        sp-Move                               ▷ Update solid particle locations
        sp-Redistribute                       ▷ Transfer solid particle data between MPI tasks
        sp-Links                              ▷ Calculate solid particle boundary links
        sp-Filling                            ▷ Fill newly uncovered fluid nodes
    end if
    MacroVar                                  ▷ Calculate macroscopic variables
end for
```

thoroughly covered in our previous studies [21,27–29].

While the methods discussed in this paper can be extended to a variety of flows with and without moving solid particles, in this work we focus on improving the performance of DNS of a turbulent particle-laden channel flow. The computational domain is a cuboid, with the wall aligned normal to the x-direction (the wall-normal direction), the streamwise direction being in y, and spanwise in z. To represent the flat channel-wall boundaries, mid-link bounce back is used on the two faces normal to the x-direction. The remaining boundary conditions in y and z are periodic. A domain of the size $x = 2H$ (wall normal) by $y = 4H$ (streamwise) by $z = 2H$ (spanwise), is discretized by a uniform mesh. It is then decomposed into smaller sub-domains in the y-direction and z-direction. Since the domain size in y is set to be twice that in z, the number of divisions in y is also twice the number in z, as seen in Fig. 2. These sub-domains are then distributed to different processors for computation, with all data communications handled by the Message Passing Interface (MPI) library. Each MPI task is assigned to an individual CPU core, meaning that 8 MPI tasks would run on a single 8 core Intel Xeon Sandy Bridge CPU.

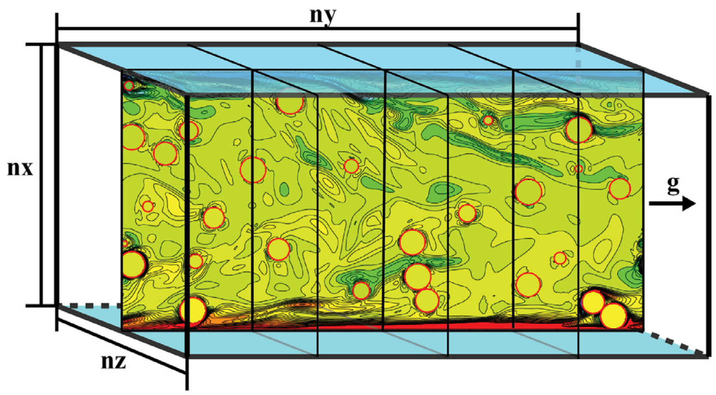


Fig. 2. An example of MPI decomposition for turbulent particle-laden channel flow. Blue faces indicate the solid boundaries using mid-link bounce back. The MPI topology breaks down the streamwise and spanwise directions into smaller sub-domains. *g* indicates the direction of the uniform driving force.

3. Data storage

With respect to LBM, the addressing and the formatting of the structure that stores the lattice-particle distributions can have a significant impact on the simulation's computational performance. When considering approaches to address the fluid distributions in memory there are two different approaches, the first being direct addressing in which a multi-dimensional array is used to read and write all lattice-particle distributions in both the fluid and solid phases. The drawback for using multi-dimensional array storage is one is limited to a cuboid domain shape. For simulations in which the boundary geometry is complex with a low porosity ratio, a full multi-dimensional array can waste a significant amount of memory since many nodes will lie in the solid phase and become unused. The second approach is indirect addressing where a sparse reference array typically containing pointers to a one-dimensional array is used to store only the particle distributions on active fluid nodes [14,15,34]. This allows indirect addressing to have the potential to save a significant amount of memory depending on the simulation. Concepts from both direct and indirect addressing can be combined to create hybrid addressing schemes as presented by Mattila et al. [35] that could be better suited for certain simulations.

In the three-dimensional particle-laden simulation considered in this paper, the fluid lattice-particle distributions are stored using direct addressing in a four-dimensional double-precision floating point array. One dimension for the set discrete velocities on each node, and the remaining for spatial coordinates. Although the geometry of suspended solid particles can be compared to that of a porous media, direct addressing was used in our simulation for two reasons. First, since the solid particles are continually moving over the lattice, the number of solid particles, and thus number of fluid nodes, residing inside a local MPI sub-domain is unpredictable and can vary significantly depending on the state of the flow. This means that if indirect addressing were used, we must account for the *worst* case in which all nodes inside sub-domain are fluid nodes which eliminates any potential memory savings. Additionally, unlike most simulations that use indirect addressing, such as flow through a porous media, the solid boundaries in this particle-laden simulation are moving relative to the lattice resulting in lattice nodes continually changing phases. This would essentially require an indirect addressing scheme to be remapped potentially every time step and more importantly destroy spatial locality without rearranging the entire lattice-particle distribution array each time a lattice node switches phase resulting in a large performance drop. Finally, when compared to porous media flows with solid volume fractions often above 30%, our simulation is designed for lower volume fractions below 20% potentially resulting in the memory savings not being able to out weigh the additionally memory requirement of the pointer array. For example, Vidal et al. observed an increase in memory usage from indirect addressing for solid volume fractions less than 20%, and only significant memory savings with volume fractions greater than 30% [14].

As indicated by multiple groups in the past, the format of the fluid lattice-particle array also plays a key role in the simulation's performance [6–8,35]. While past works have proposed unique storage schemes to maximize spatial locality or allow for more efficient vectorization [8,35,36], the most optimal array format is heavily dependent on the specific collision and propagation algorithms used. Thus in the scope of the work we shall focus on the two most common formats, and then rigorously and systematically compare each format's performance on a variety of different collision and streaming algorithms. The first format is the array of structures (AOS), a collision optimized format, in which all discrete velocities for a given node are stored next to each other on the physical memory. AOS allows a node's local fluid-particle distributions to be efficiently loaded into cache memory which allows a more optimized collision process. The second structure is the structure of arrays (SOA), a propagation optimized format, where a discrete velocity of all nodes are stored next to each other. SOA allows neighboring fluid-particle distributions of a specific velocity to be better loaded into cache memory, resulting in a more efficient streaming process and making it better suited for vectorization. The difference between these two data structures can be seen visually in Fig. 3. The key idea behind these different formats is to take advantage of spatial locality and cater the program to take full advantage of CPU cache line loading and cache associativity which can significantly accelerate a program [37].

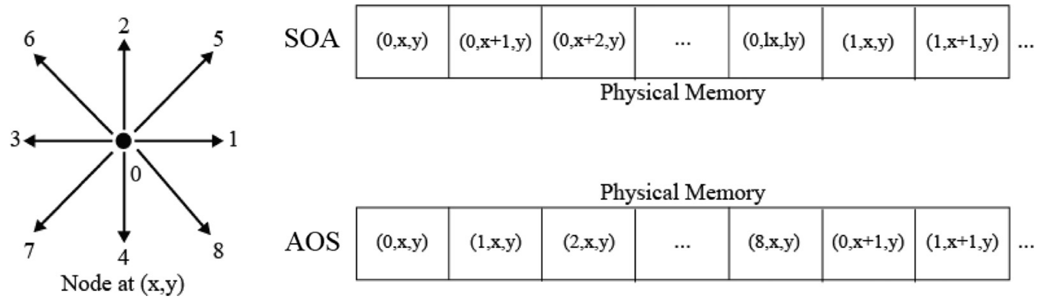


Fig. 3. A comparison of two data formats using the D2Q9 lattice model. The first is propagation optimized or structure of arrays (SOA) data format. The second is the collision optimized or array of structures (AOS) data format.

4. Collision and propagation methods

In the past 10 years, multiple groups have proposed a variety of different algorithms to make the collision and streaming process more computationally optimized in LBM [9–11]. These different algorithms primarily focus on reduction of data read/writes along with reducing memory requirements. Given that the collision and streaming sub-steps are present in every LBM simulation, optimizing these fundamental steps can yield performance gains across multiple simulations. In the following section we will first briefly review several different algorithms that have been proposed by several different groups and then rigorously compare their performance on the same machine using both data formats described previously in Section 3.

We have selected a total of five different algorithms to implement and test using both the SOA and AOS array structures. The first algorithm is the two-step algorithm in which all fluid nodes undergo collision, are placed back into their pre-stream location, and then the entire lattice is streamed at once. While intuitive and requiring the minimal array size for storing the fluid lattice-particles, this approach is far from computationally optimal. Fusion of the collision and propagation sub-steps into a single process can allow for a significant reduction in amount of data movement required [6]. The simplest fused method is the two-array algorithm in which distributions are accessed from one array, collided and then placed in their post-streaming state in a second array to eliminate false data dependencies introduced from storing the fluid lattice-particles in a single array. During the next time step this process is reversed. While the two-array method is very easy to implement, it does require doubling the memory footprint relative to what would normally be required.

To reduce the memory requirements of two-array, the grid compression or shift algorithm was developed [9,10]. Shift maintains the fusion of the collision and streaming steps while using an expanded lattice to shift the entire lattice each time-step to avoid data dependencies. Mattila et al. [9] presented an efficient swap algorithm that does not require an expanded lattice and requires less loads and stores from memory than the two-array or shift methods. Although swap introduces much more complexity into the collision and streaming process, the clever swapping of specific fluid-lattice distributions with neighboring nodes allow for the propagation process to be carried indirectly while dancing around data dependencies. The final algorithm implemented is called AA-Pattern, originally proposed by Bailey et al. [11]. The key advantage to AA-Pattern is that it has minimal memory requirements, just like two-step and swap, and has optimal data accessing by only reading and writing to the same location [6]. Additionally all lattice nodes can be updated in parallel, just like two-step and two-array, making it ideal for GPUs. That being said, AA-Pattern alternates between two different lattice states for which the standard lattice formation is only present every other time-step. This makes processes such as boundary conditions much more difficult since two versions are then required to be implemented for the two different lattice states.

Although serial optimizations are traditionally benchmarked on a single core, we have completed tests in parallel since our simulation is designed to be executed on multiple CPU cores and this amplifies the impact of a minor performance gain. Since our focus here is on just the streaming and collision, we first consider a benchmark test using a single-phase turbulent channel flow with a uniform mesh of grid resolution $nx * ny * nz = 200 * 400 * 200$, exactly the same as that used for DNS of single-phase turbulent channel flow at a frictional Reynolds number of $Re_\tau = Hu^*/\nu = 180$, discussed in [28,29], where u^* is the wall frictional velocity, H is the channel half width, and ν is the fluid kinematic viscosity. Such a grid resolution provides a grid spacing of about $1.18\nu/u^*$, sufficient to resolve the near-wall region of the single-phase turbulent channel flow [38]. By the overall force balance, the frictional velocity can be related to the driving body force g as $u^* = \sqrt{gH}$.

We first conducted a strong scaling test by increasing the number of MPI tasks used to decompose the global domain resulting in an overall speed up of the program as more resources are devoted to the same problem size. *MLUPS* (Mega Lattice Updates Per Second) is used to quantify the performance, this effectively measures how many fluid node points undergo both collision and propagation every second. Fig. 4 displays the performance of each algorithm in our single phase strong scaling test. Swap using the AOS data structure has the best performing overall for our CPU based implementation and two-step AOS performing being the worst. In general, these results are in congruent with serial benchmarks completed in the past [6,9]. We do note that although, based on data read/write requirements alone, AA-pattern should be the most ideal, our implementation of swap yielded better performance. We attribute this to us not fully exploiting the full massively

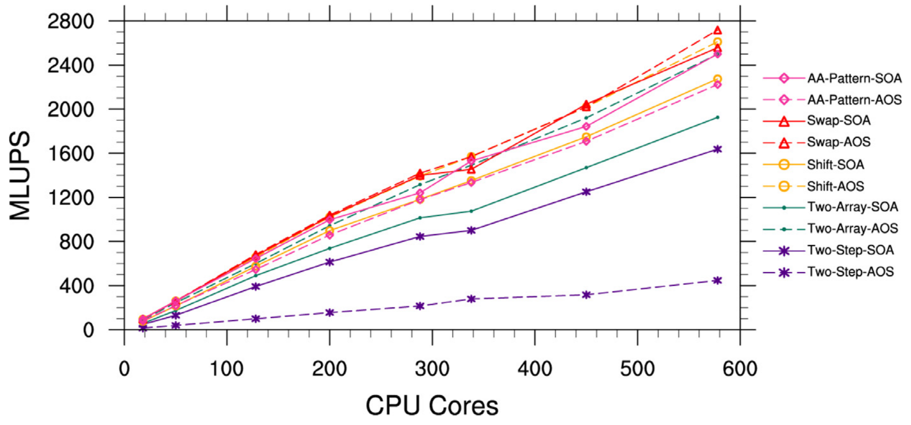


Fig. 4. The strong scaling performance of LBM simulation of single-phase turbulent channel flow, held at a fixed mesh resolution of $nx * ny * nz = 200 * 400 * 200$ with varying MPI tasks.

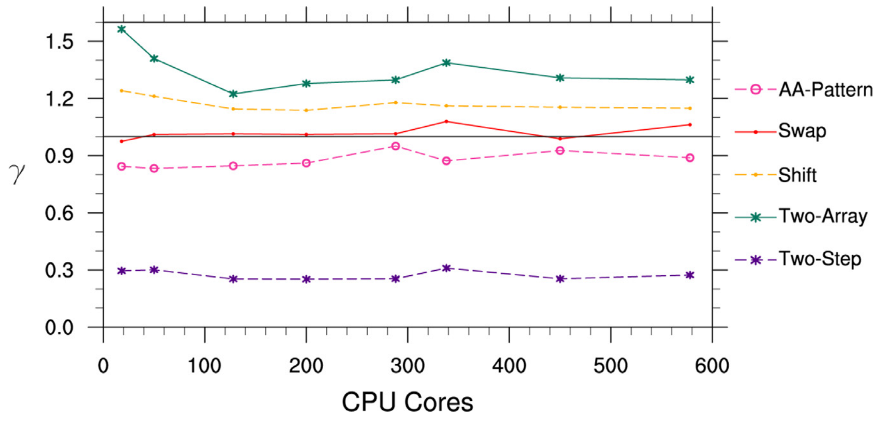


Fig. 5. The data format performance ratio defined as $\gamma = (AOS)/(SOA)$ for strong scaling of the single-phase turbulent channel flow.

parallel potential of AA-pattern, that is seen when using GPUs, which results in added complexity without fully reaping the benefits.

Additionally, it is clear that data storage format plays a critical role in the performance due to different levels of cache utilization efficiency. Fig. 5 plots the ratio, γ, of each algorithm's AOS computational performance in MLUPS over the corresponding SOA performance. We can suggest that both two-array and shift can be considered collision based methods that favor the collision optimized AOS array format. On the other hand, algorithms such as AA-pattern and two-step favor the streaming optimized SOA array format indicating that the streaming process or data movement is the bottle neck in these methods. Finally we see that swap just slightly favors AOS but still has very good performance with SOA format. This result deviates from Mattila et al.'s [35] benchmarks which have a swap γ of approximately 1.25 likely due to implementation variation. Regardless, the general lack of array format preference in our implementation of swap opens up the option to use array formats better suited for reduction of cache misses, vectorization and other potential optimizations.

In Fig. 6 the speed-up versus the core count is plotted. The speed-up is defined by $S = t_0/t$ where t_0 is the wall-clock time of the run with the lowest parallelism, in this case 18 cores, and t is the wall clock time when the processor count is further increased, using the same algorithm. The black straight line represents the ideal speed-up curve. We notice that the speed-up ratios are close to the ideal curve for all algorithms, which is to be expected since we are focusing on serial optimizations. However, more optimized algorithms have a slight drop in scaling performance while slower methods such of two-step have ideal scaling. This is attributed to the poor computational performance masking the communication overhead of the slower methods. Based off the results here, we will be using the swap method for its speed, low memory requirements and its ability to retain the standard lattice structure every time-step.

5. Vectorization of LBM collision

For computationally dense problems with highly repetitive calculations, single instruction multiple data (SIMD) computations can allow for a significant boost in computational performance. The introduction of Streaming SIMD Extensions (SSE) intrinsics from Intel allowed SIMD operations using 128 bit vector registers which, when using double precision floats

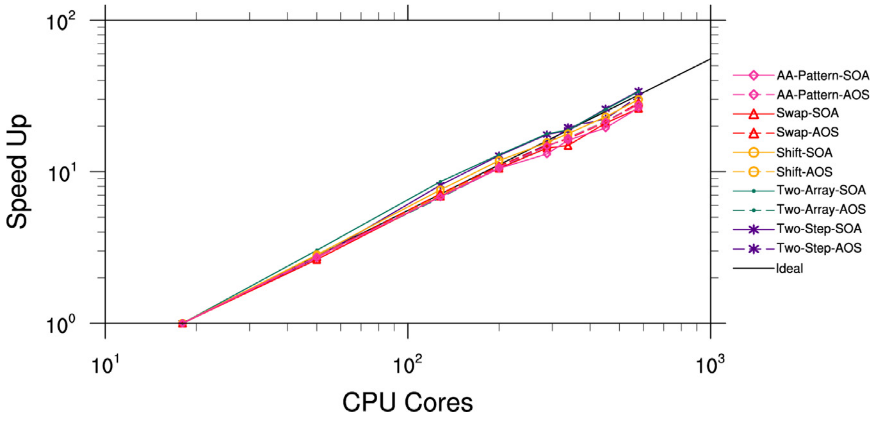


Fig. 6. The strong scaling speed-up of LBM simulation of single-phase turbulent channel flow with a fixed mesh resolution of $nx * ny * nz = 200 * 400 * 200$. The black straight line denotes the ideal speed-up curve.

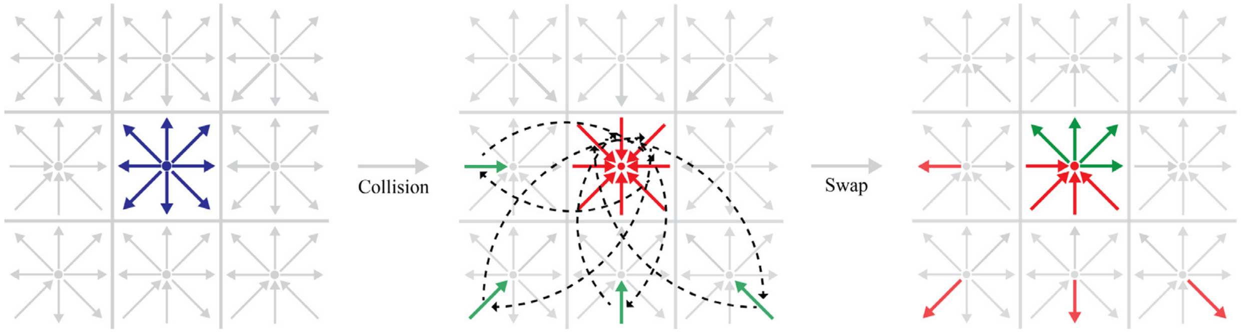


Fig. 7. D2Q9 push implementation for the swap algorithm that allows the conservation of the standard lattice format.

as we are in this paper, equates to 2 simultaneous floating point calculations. In recent years with new modern architectures, Intel has unveiled Advanced Vector Extensions (AVX), supporting larger 256 bit registers [39]. While functions related to solving solid particles are tricky to vectorize due to a large amount of conditional statements, single-phase LBM lends itself very well to SIMD computing due to its highly repetitive arithmetic. Multiple works in the past have delved into exploiting SSE and AVX calculations to boost performance [6,8,40]. However, in these past works simpler collision models such as BGK and two-relaxation-time (TRT) [41] are used which allow for a much easier implementation of LBM into SIMD computations. Although more complex and computationally demanding, MRT collision used in this work reins superior for numerical stability and simulation tuning due to its additional relaxation parameters [30]. Thus in this section we shall focus on integrating SIMD computations into our simulation that is using a more complex collision model.

The simulation considered here was originally programmed in FORTRAN and while FORTRAN can often outperform C with large array manipulation, explicit SSE/AVX intrinsics are not supported by FORTRAN compilers. This is not ideal when one wishes to perform complex vectorization manually which is the case here. Fortunately, one can interoperate between C and FORTRAN with FORTRAN's *iso_c_binding* module which opens up the potential use of SSE and AVX intrinsics by interfacing with C functions from FORTRAN. As we mentioned in the previous section, we will be using the swap method for its speed, low memory consumption, and ability to retain the standard lattice structure. Our implementation of swap can be seen in Fig. 7 and details on its functionality can be referenced in Mattila et al. [9]. While the swap algorithm requires fluid-particle distributions to be moved to neighboring nodes in serial during the actual swap process, the local collision of nodes can certainly be carried out in parallel which results in almost a hybrid swap/two-step implementation.

MRT collision requires two matrix operations, one of which translating the fluid lattice-particle distributions into moment space and the other transforming the post collided distributions out of moment space. Upon inspection of the transformation matrix for D3Q19, seen in Appendix A of d'Humières et al. [30], one can notice that the transformation matrix, $\boldsymbol{M}$, is relatively sparse thus the first step was to eliminate the zero and repeated operations by explicitly writing out the matrix arithmetic and identifying common sums. However this alone still results in operations that are much too complex for automatic vectorization in a compiler, thus the only way to utilize SIMD based operations is to employ SSE/AVX instructions manually. Even after explicit matrix arithmetic and eliminating redundant calculations the collision process still executes over 900 SSE/AVX commands to calculated forcing values, transform distributions in and out of moment space, and calculate equilibrium values for a *single* fluid node in D3Q19. Of these instructions only a few are vector loads and stores

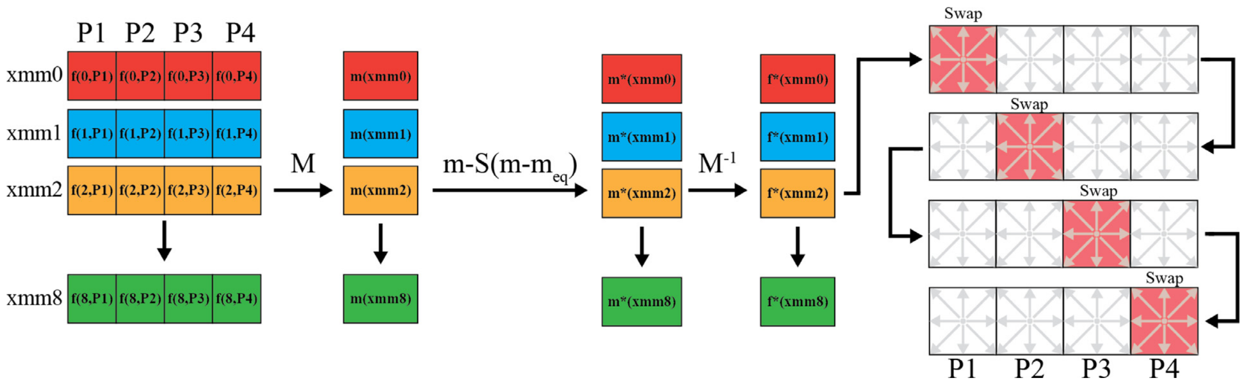


Fig. 8. Diagram of the proposed collision and streaming process with D2Q9 using AVX SIMD commands. MRT collision is executed using AVX commands, colliding four nodes at a time. The set of four nodes are then streamed in series using the swap algorithm in Fig. 7.

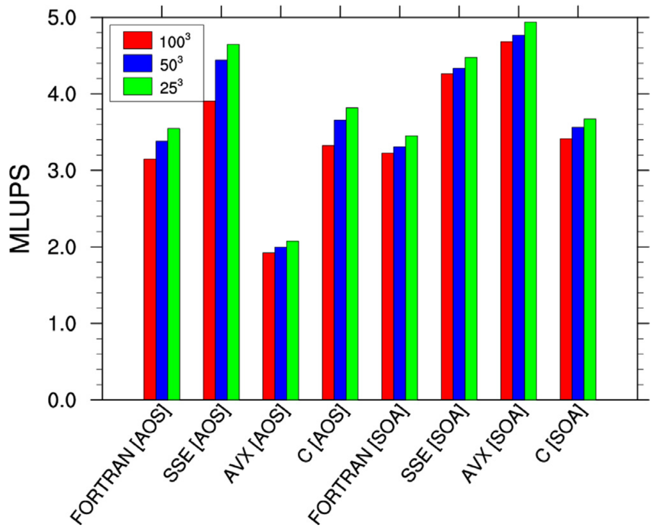


Fig. 9. A performance comparison of FORTRAN, SSE (128 bit vector registers), AVX (256 bit vector registers), and standard C implementations of MRT collision and swap streaming on a single processor core using both the AOS and SOA array structures.

including the loading of pre-collision lattice distributions and macroscopic variables such as velocity, density, and forcing values for the calculation of $\boldsymbol{m_{eq}}$. Operations are then restricted to the $m128$ or $m256$ aligned data types until the post collided distributions are placed back into their post stream location. The core idea is to collide 2 (for 128 bit vector registers) or 4 (for 256 vector registers) nodes at the same time and then sequentially stream them using the swap method illustrated in Fig. 8.

We implemented a total of four MRT collision versions: one using FORTRAN, one using SSE intrinsics, one using AVX intrinsics, and the last just being vanilla C. For each version the arithmetic is identical in each, the only difference being the SSE/AVX are manually programmed to perform SIMD calculations while vectorization of the FORTRAN/C versions is left up to the compiler. We again tested using the AOS and SOA array format at three different domain sizes of $nx * ny * nz = 100^3$, 50^3 and 25^3. The physical flow used was a simple laminar channel flow with walls normal to the x direction and periodic boundaries in the y and z. Intel's compilers were used for both FORTRAN and C compilation with the optimization level set to $O3$ or aggressive; the results can be seen below in Fig. 9. Even though the collision arithmetic is quite extensive, vectorizing does result in a significant performance gain. Implementing AVX instructions with SOA array format results in a speed up close to 1.5 for all domain sizes. The performance of vanilla FORTRAN and C is consistently lower than both SSE and AVX implementations which indicates the compiler did fail at providing the same degree of vectorization. SOA is the superior array format for vector calculations as expected. However, the SSE implementation with AOS does yield a very good performance gain compared to the standard FORTRAN or C implementation. It is also seen that using larger 256 bit registers with AVX and the AOS array format results in significant cache thrashing and a sharp decline in performance. Thus when using industry standard 256 bit registers or larger sizes such as 512 bit in the future, AOS should be avoided. Although, theoretically AVX should allow the number of calculations to be reduced by a factor of three we do not see such

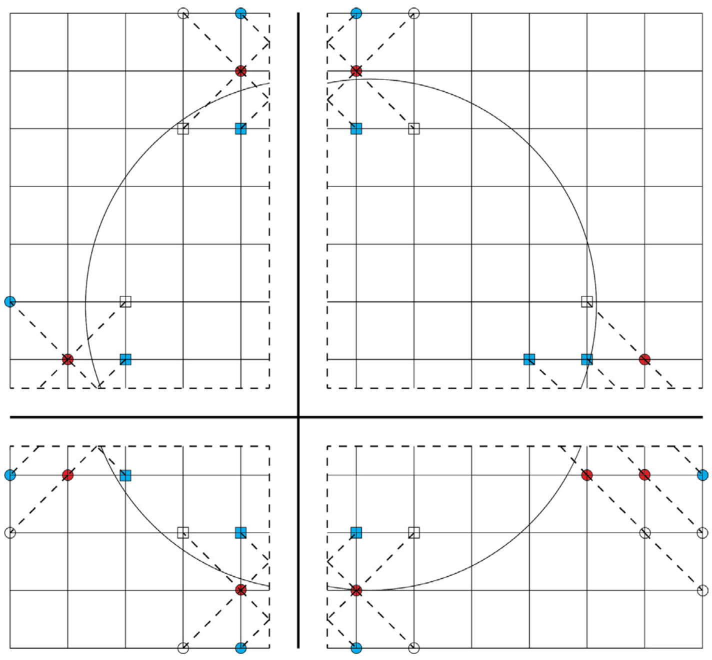


Fig. 10. An illustration when a solid particle resides in multiple sub-domains, resulting in distributions being needed from neighboring tasks to execute interpolation. In this example the red nodes indicate the location at which interpolation needs to occur and blue nodes indicate the location of distributions that are needed for interpolation but reside in a different sub-domain. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

a significant speed up for several reasons including: overhead from data alignment, swap/streaming overhead and vector register loading/storing. All of the above can potentially be improved to obtain the full potential of SIMD calculations.

6. Optimization of solid particle overhead

One of the significant challenges of optimizing a LBM code for interface-resolved simulation of particle-laden flow is reducing the solid particle computational overhead, often defined as the amount of additional wall clock time a multi-phase simulation takes compared to a single-phase flow for a prescribed lattice grid resolution. While the bounce back and refilling schemes implemented here provide better numerical stability and physical accuracy, they are significantly more computationally demanding compared to simpler methods due to their need for higher-order spatial interpolation or extrapolation and additional arithmetic. If programmed naively, solid particle related functions can have a devastating effect on a simulation's computational performance as the naive implementation would constantly invoke global and random loading of data. In contrast to the previous sections, the focus is now on algorithmically improving data communication associated with the treatment of moving fluid-solid interfaces.

It is well known that data communication between hardware components is a frequent bottleneck for massively parallel programs. Optimizing data communication between these components is critical for good scalability and overall performance. As previously stated, our LBM simulation is broken down into a 2D MPI topology. Thus, due to the construction of the D3Q19 lattice, data between a processor and its 8 neighboring tasks must be communicated during each time step. Communication first occurs during the streaming process in which fluid lattice-particle distributions are moved out of a local task into the appropriate neighbor. Since communication delays increase as message size increases, one should only transfer the fluid particle distributions that are actually leaving the local sub-domain as opposed to all distributions associated with the fluid node at the boundary to minimize amount of data transferred. This can be further refined for simulation with large stationary obstacles where it is known in advanced which distributions need to be exchanged and which do not due to a solid boundary [14]. However for moving solid particles, such optimization is not possible due to the changing location of the solid particle boundary relative to the fixed lattice.

When a solid particle is close to the edge of a given sub-domain, one must consider if information is needed from a neighboring MPI task to properly carry out either interpolation bounce-back or extrapolation during the refilling step. For example in Bouzidi's quadratic interpolation bounce back, as shown in Eqs. (6) and (7), if node x_f is on the edge of the local MPI task's sub-domain, x_b, x_{ff} or x_{fff} could reside in the neighboring sub-domain. Thus distributions will need to be obtained from the respective neighboring processes as illustrated in Fig. 10 in which a solid particle resides in 4 different MPI tasks. Depending on the boundary/refilling scheme used, pre-caching fluid distributions needed for particle related processes can

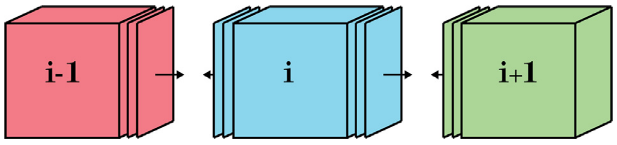


Fig. 11. An illustration of the slice-based algorithm for a 1D MPI topology for task *i*, in which a layer all distributions that could be needed are exchanged between neighboring tasks.

eliminate a part of the local communication. However, the effectiveness of pre-caching depends largely on the scheme used, the information available at the boundary, and the amount of conditional statements one wishes to introduce into the collision sub-loop. For example, through pre-caching Bouzidi et al.'s linear interpolation bounce back can be fully local [31], however, this is not possible with quadratic interpolation and other methods. The required data communication for both bounce back and refilling also depends on factors including local sub-domain size, flow speed, volume fraction and particle radii. Thus our goal is to create a robust data communication scheme that adjusts with these factors to avoid the passing of excess data. Below we discuss three different algorithms to potentially handle this data communication need.

6.1. Data communication schemes at the moving fluid-Solid interface

6.1.1. Slice-based data communication

The first method discussed, which also happens to be the most naive, is to blindly exchange every fluid distribution that could possibly be required for either interpolation or extrapolation. This method is in essence very similar to the streaming communication such that a task exchanges the entire outer face with its neighboring process as seen in Fig. 11. For example, a bounce-back/refilling scheme that uses either linear extrapolation refilling scheme or quadratic interpolation bounce back requires a slice containing two lattice layers. A quadratic extrapolation refilling scheme, such as the one being used, would require three lattice layers to be exchanged. Depending on the bounce back or refilling scheme, all distributions may need to be exchanged, such is this case with the contained velocity refilling scheme. This method makes both fluid-solid particle interaction and refilling the most communication intensive operations in the entire simulation.

6.1.2. Flag-based data communication

Another possible and relatively simple approach is to implement a flag system in which a task locally determines which MPI neighboring processors it needs to receive data from to execute either interpolation or extrapolation. A set of flags is established for each respective MPI neighbor, if data is required from a given MPI neighbor its flag is set to true, as illustrated Fig. 12(a). These flags can be very small data types such as a boolean since it simply needs to be either true or false. The flags can then be dispersed globally or just to the local MPI neighbors, as shown in Fig. 12(b). The small data type allows this messaging to be small and quick. Each MPI task is then enabled to determine which neighbors it needs to send data to and what data will be received (Fig. 12c). While the task must send a large slice like the previous algorithm, the use of flags allows the amount of communication to significantly decrease for lower volume fractions of solid particles with relative ease.

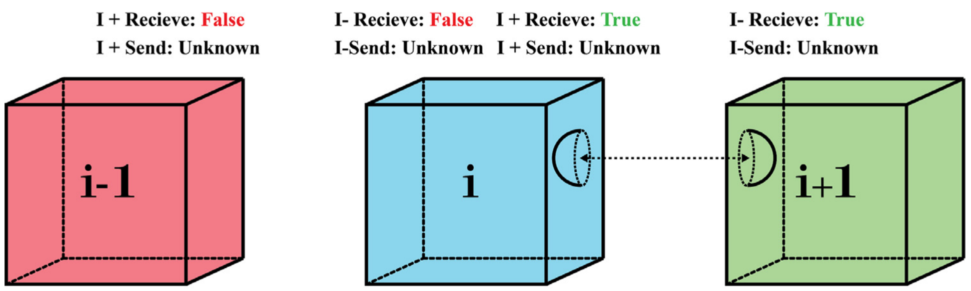
6.1.3. Direct-request data communication

The final algorithm presented is designed to only exchange data that is absolutely needed. We break this algorithm down into a series of steps using Fig. 13 as a simple example to illustrate the process.

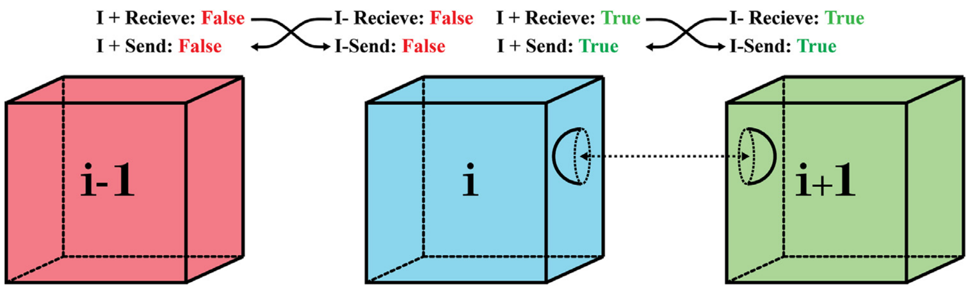
1. The process starts by determining which distributions (or nodes) are needed from all of its neighboring tasks. In Fig. 13 task i would determine that it needs distributions from task $i+1$ to complete interpolation. Information including velocity direction and spatial location are stored in a *request* buffer.
2. Once the entire local sub-domain has been processed each request buffer is sent to its respective neighbor (Fig. 13a).
3. Upon the receiving of a request buffer, the process then parses this request and retrieves the requested distributions in the *same* order in which they were requested.
4. These distributions are then sent back to the original process (Fig. 13b), in which they are received, and used appropriately.

While the fundamentals of this algorithm seem comparable to the previously described methods, this approach has many hidden challenges and pitfalls that make it difficult to implement. Here we describe a few difficulties that we encountered:

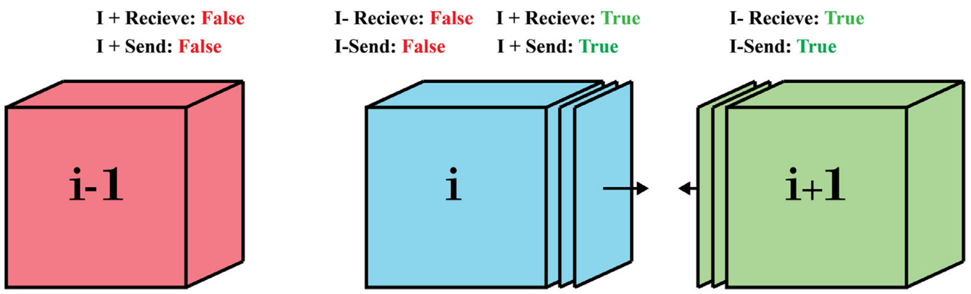
- The length of the request buffer array is changing every time step, the amount of data each task is receiving is unknown for Step 2. For our MPI implementation we utilized MPI_PROBE followed by MPI_GET_COUNT functions to determine the message sizes that we were receiving without having extra data communication.
- It is critical that each task sends back distributions in the same order in which they were requested. This allows each task to use its request buffer to correctly identify the distributions received from each neighboring task. Returning the



(a) Step 1: Determine receive flags



(b) Step 2: Exchange flags



(c) Step 3: Exchange distributions

Fig. 12. The three steps in the flag-based scheme for MPI task i on a 1D MPI topology for interpolation bounce back. In this example a solid particle is partially in the MPI task i and $i+1$ local sub-domains. Thus both require data from the neighbor to execute interpolation on the nodes near the local sub-domain boundary.

requested distributions in a different order would require additional information to be sent between each processor with details of the distributions including spatial location and direction.

- Although not fully exploited in our implementation, direct-request has the unique advantage to be approached similar to a network scheduler. Namely, all requests are sent to neighboring processors at once, and requests are processed and returned as they are received. Although here we use a first in, first out approach other algorithms such as a round-robin technique can be used to process incoming requests at a given MPI task [42]. This could help drop communication delays between CPUs during this process.

6.2. Results

We now apply the three different data communication schemes for treating moving fluid-solid interfaces. The physical flow problem is the turbulent channel flow laden with finite-size solid particles, discussed in detail in [28,29]. The background turbulent channel flow is identical to single-phase test noted in Section 4 with a uniform grid resolution $nx * ny * nz = 200 * 400 * 200$ and $Re_\tau = 180$. For the suspended solid particle phase, we consider the neutrally buoyant par-

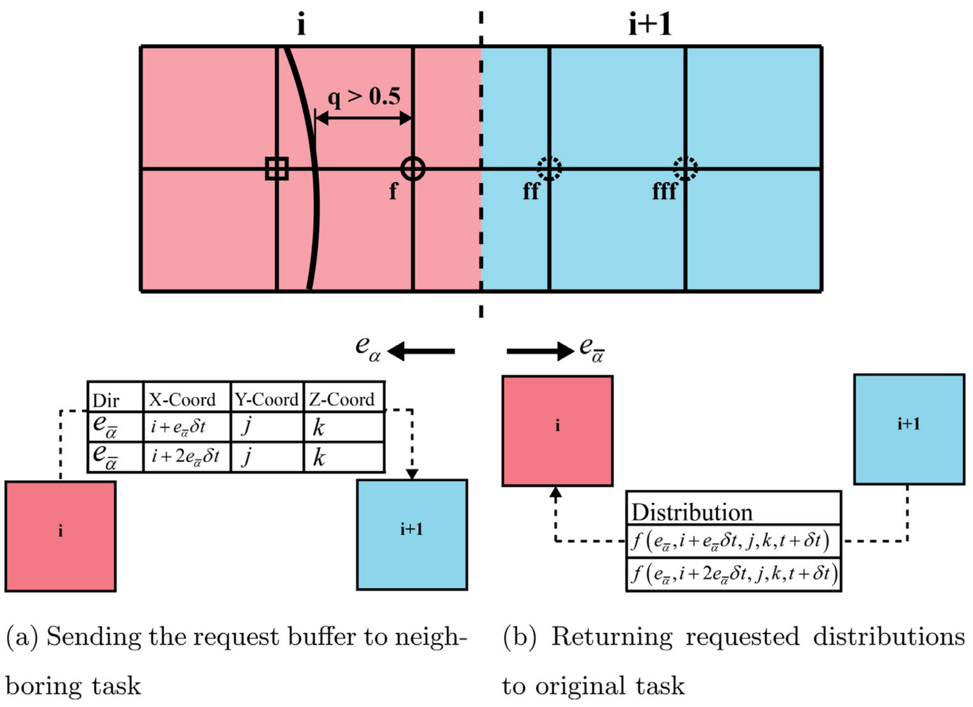

i
i+1
q > 0.5
f
ff
fff
e_α
$e_{\bar{\alpha}}$
Dir | X-Coord | Y-Coord | Z-Coord
$e_{\bar{\alpha}}$ | $i+e_{\bar{\alpha}}\delta t$ | j | k
$e_{\bar{\alpha}}$ | $i+2e_{\bar{\alpha}}\delta t$ | j | k
Distribution
$f\left(e_{\bar{\alpha}}, i+e_{\bar{\alpha}}\delta t, j, k, t+\delta t\right)$
$f\left(e_{\bar{\alpha}}, i+2e_{\bar{\alpha}}\delta t, j, k, t+\delta t\right)$
(a) Sending the request buffer to neighboring task
(b) Returning requested distributions to original task

Fig. 13. An illustration of the direct-request algorithm for obtaining the distributions needed for the Bouzidi et al.'s interpolated bounce-back distribution on the fluid node that is right on the MPI sub-domain boundary.

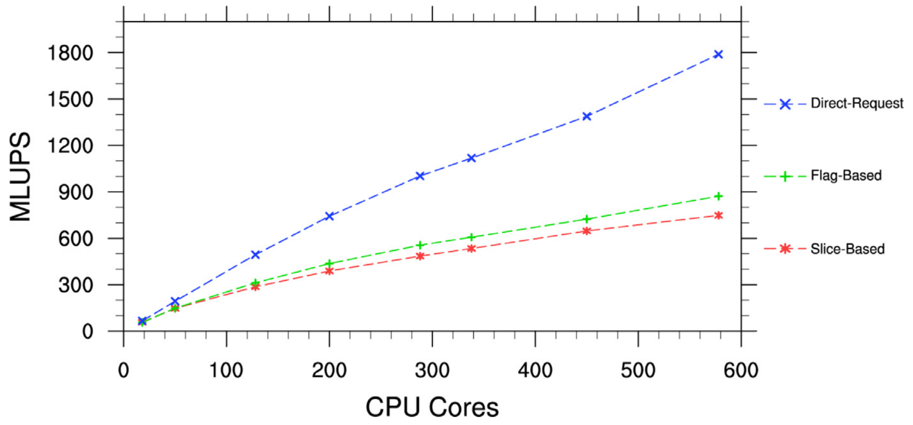


Fig. 14. The strong scaling performance of the each communication algorithm with a constant global domain of $nx * ny * nz = 200 * 400 * 200$ and a solid volume fraction of 7%.

ticles, namely, the density of the solid particles being the same as that of the fluid. In this case, the characteristics of the solid phase is described by two physical parameters: the relative particle size r_p/H and the average solid volume fraction $\phi_v = N_p 4\pi r_p^3/(3 \cdot nx \cdot ny \cdot nz)$, where r_p is the particle radius and N_p is the total number of solid particles. For the following benchmark test, $N_p = 270$, $r_p = 10$, and $\phi_v \approx 7\%$, if not specified otherwise.

Three separate simulations were performed, each using one of the three previously described communication schemes to obtain data to conduct both interpolation for fluid lattice-particle distribution at the fluid-solid interface and extrapolation for constrained velocity refilling. To test the performance of each communication scheme we conducted a series of scaling tests similar to Section 4. Fig. 14 shows the performance in MLUPS of the three communication schemes and Fig. 15 shows relative speed-up. It's clear that the data communication for solid particle related functions play an important role in a program's parallel performance. We observe that not only does the direct-request data communication scheme have superior computational performance, but it also has better parallel performance since its speed up is much closer to the ideal performance line in Fig. 15. Although we do see some minor deviation with greater MPI tasks, this could be most likely improved with a three dimensional MPI decomposition. The MLUPS level is roughly doubled using direct-request compared to slice-based and flag-based schemes.

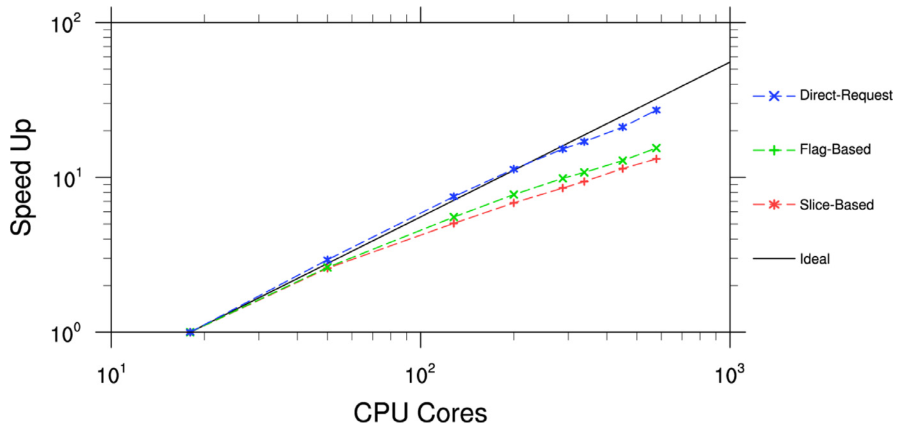


Fig. 15. The strong scaling speed-up of the each communication algorithm with a constant global domain of $nx * ny * nz = 200 * 400 * 200$ and a volume fraction of approximately 7%. The black line marks the ideal speed-up curve.

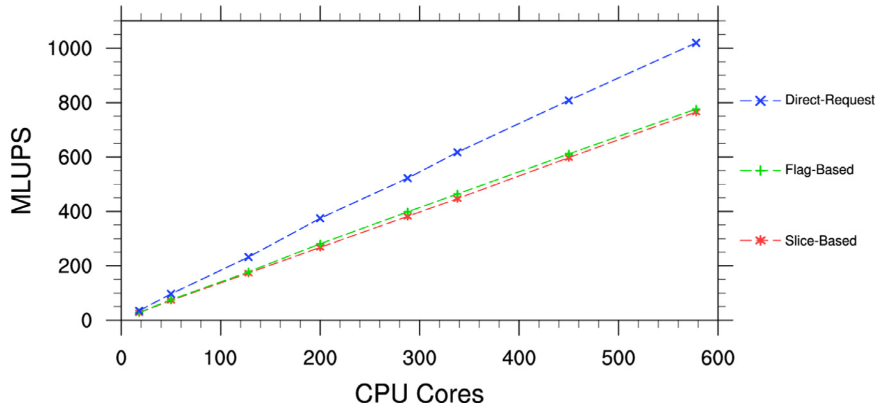


Fig. 16. The weak scaling performance of each communication scheme with each task assigned a local domain size of $nx * ny * nz = 200 * 40 * 40$ and a solid volume fraction of 7%.

For further comparison, a weak scalability test was performed to assess each communication algorithms ability to adapt to larger domains. Each MPI task's local domain is constrained to the same size of $nx * ny * nz = 200 * 40 * 40$, and the global volume fraction is constrained at 7%. As the number of MPI tasks used increases so does the global domain size and the number of solid particles, however the computational requirement from each process remains the same. A similar trend is seen in the performance of each communication scheme in Fig. 16 with direct-request exhibiting the best performance.

The weak scaling efficiency for each communication scheme, η_{weak}, is plotted in Fig. 17. Defined as the inverse of speed up, the ideal η_{weak} is 1 which indicates perfect scaling to larger problem sizes. Both slice and flag-based have a very steep decline in weak scaling due to the large message requirements resulting in very poor communication delays between processors. This results in a synchronization bottleneck between MPI tasks waiting on neighboring processors to send/receive the desired information due to the two-way communication requiring both tasks be synchronized. As the MPI topology expands the delay caused by synchronization is amplified until it reaches a saturation point at which the efficiency starts to decay at smaller constant rate. We do not observe this initial performance drop from direct-request because, although the communication is still blocking in a sense that the program cannot proceed until all requests have been fulfilled, the nature of direct-request allows each MPI task to send out requests to multiple neighbors and then process the requests/responses as they are received allowing for lower synchronization delays. Additionally, since direct-request messages are significantly smaller, the delay in sending/receiving messages is much quicker also allowing for increased communication speed.

Additional benchmarks were performed where the solid particle volume fraction was varied. For this test solid particles were given a radius of 10 fluid lattice cell lengths and a fluid domain of $nx * ny * nz = 200 * 400 * 200$, when 200 MPI tasks was used. In Fig. 18, the wall-clock time per time-step is shown as a function of the solid particle volume fraction along with wall clock time of a single-phase flow simulation with the same domain size. We again observe that direct-request has the best performance out of the three communication schemes. Since data communication needed for slice-based algorithm is independent of solid particle volume fraction, the wall clock increase is a direct result of increased computation needed to resolve the particle boundary and refilling of the new fluid nodes alone, which is roughly linearly proportional to the volume fraction. The computational overhead for the sliced-based scheme is between 150% and 220% for a solid particle volume

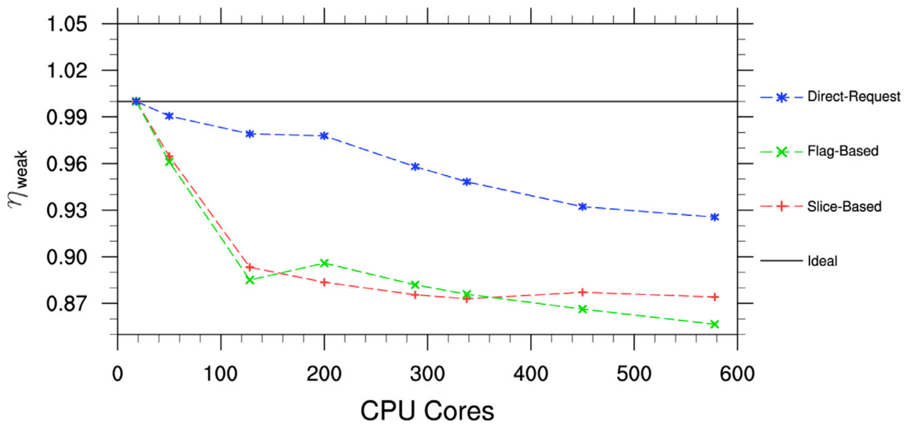


Fig. 17. The weak scaling efficiency, η_{weak}, of each communication scheme with each task assigned to a local domain size of $nx * ny * nz = 200 * 40 * 40$ and a solid volume fraction of 7%. The horizontal black line marks the ideal weak-scaling efficiency.

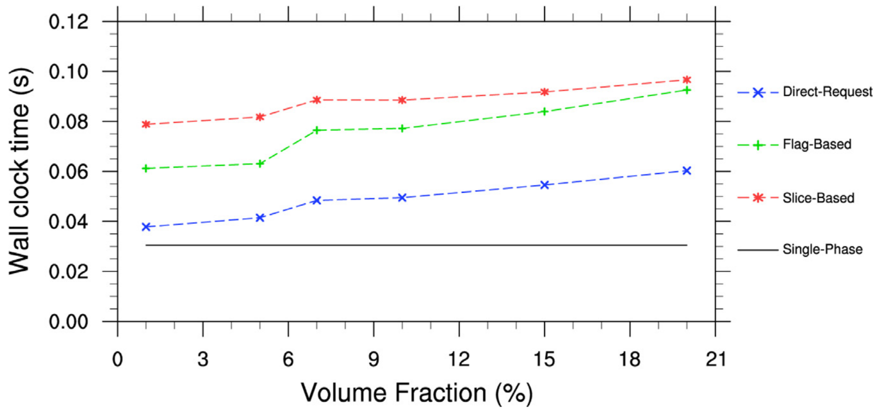


Fig. 18. The effects of solid volume fraction on the wall clock time per time step of each communication scheme using a domain size of $nx * ny * nz = 200 * 400 * 200$, 200 MPI tasks, and a particle radius of 10 lattice spacings.

fraction between 1% and 20% respectively. Both direct-request and flag-based methods exhibit greater change since both are designed to dynamically adjust their data communication requirements based off the precise distributions needed. At low volume fractions while both flag-based and direct-request perform significantly better than slice-based, direct-request is clearly superior with minimal overhead less than 24% for a volume fraction of 1%. At large volume fractions the flag-based algorithm approaches the same performance of the slice-based scheme since the solid particles are much more dense and almost all sub-domain edges must be exchanged. On the other hand, the direct-request scheme continues to perform significantly better, with an overhead of roughly 100% when the solid volume fraction reaches 20%.

We conclude by listing the wall-clock times and solid particle overhead for a set of arbitrary runs in Table 1 using all described optimizations including swap push implementation, AVX vectorization, and direct-request communication. Three different solid particle volume fractions are tested as close to 5%, 7%, and 10% as possible for three different particle radii. The computational load of the solid particle related functions is directly correlated with the total fluid-solid interface area along with the number of solid particles. Under a prescribed solid volume fraction, surface area is inversely proportional to the particle radius. Thus as the solid particle radius decreases, the number of boundary links increases, which raises the solid particle overhead. Additionally increasing the number of solid particles results in more fluid-solid boundary links crossing MPI boundaries, also increasing the communication requirement when using direct-request. This is why for small particle radii we see a spike in solid particle overhead, while the computational overhead for larger radii is much more reasonable being between 30% to 80%. However, it's evident that much more work must be done to reduce the solid particle overhead since the single-phase section is well optimized. While this solid particle overhead reported here is in fact larger than in our previous studies [24,27], our previously discussed optimizations for the collision and propagation sub-steps mask the true performance gain we have achieved. This point is illustrated in Fig. 19, in which we compare the performance gain of the three optimizations listed in this paper against our previous version used in our most recent particle-laden channel flow studies [28,29]. Fig. 19 emphasises that while the single phase optimizations yielded a significant performance gain, the optimization of the solid particle related functions is very important and yielded the largest performance gains. We achieved

Table 1
Runs to assess the solid particle overhead for our particle-laden turbulent channel flow code with a mesh resolution of 200*400*200 using 200 MPI tasks. r_p is the solid particle radius in lattice units, N_p is the number of solid particles used, ϕ_v is the respective solid-phase volume fraction, and WC is the wall-clock time per time step.

r_p	N_p	ϕ_v	WC(s)	MLUPS	Overhead
–	0	0%	0.024	676	–
5	1528	5.00%	0.049	326	104%
5	2139	7.00%	0.056	286	133%
5	3056	10.00%	0.067	239	179%
10	191	5.00%	0.036	442	50%
10	270	7.07%	0.038	420	58%
10	382	10.00%	0.043	368	79%
15	57	5.04%	0.032	500	33%
15	79	6.98%	0.033	478	37%
15	113	9.98%	0.036	441	50%

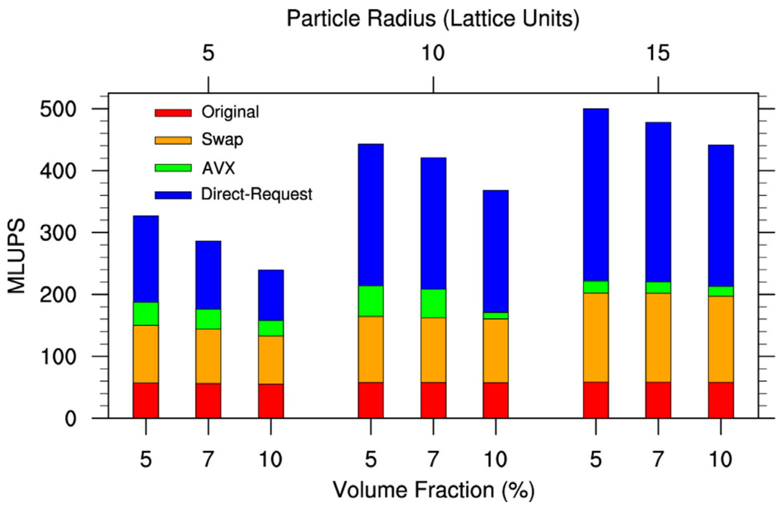


Fig. 19. A comparison in computational performance between the original code used in Wang et al. [29] and the performance gain from each optimization discussed in this paper for nine different simulations at varying solid particle radii and volume fractions. The red bar shows the original performance of the particle-laden turbulent channel flow code in terms of MLUPS. The yellow bar indicates the performance gain from swap over the original two-step implementation. The green bar describes the performance gain from vectorized collision implementation using AVX instructions. And finally, the blue bar marks the performance gain from the direct-request communication scheme over the original slice-based implementation. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

between a 4.0 and 6.0 times speed up over our previous simulation for a smaller solid particle radius of 5 lattice-cell units, and a 5.0–8.5 times speed up for larger solid particle radii between 10 and 15.

7. Summary and outlook

In this paper we have explored three different aspects to optimize our lattice Boltzmann interface-resolved simulation code of particle-laden turbulent channel flow, using the MRT LBE [30]. To realize the no-slip boundary condition at the moving fluid-solid interface, quadratic interpolated bounce back was employed [16]. The constrained velocity refilling scheme is used to repopulate distributions at new fluid nodes [21]. While these methods are more computationally intensive, they provide greater numerical stability and physical accuracy than simpler methods as demonstrated in our recent studies [21].

First, we looked into the performance of different collision and propagation methods previously proposed to increase computational performance by improving data accessing patterns. We implemented and benchmarked our results using both the array of structures (AOS) and structure of arrays (SOA) formats for storing the distributions of fluid lattice-particles. It was found that our swap implementation preformed the best and did not have any significant performance difference between AOS and SOA. We then transitioned to focusing on vectorizing our LBM code using Intel's AVX/SSE instruction set. Although the MRT collision is much more complex than other collision models, we were able to successfully vectorize the collision calculation and achieve a significant speed up of approximately 1.5 compared to non-vectorized C and FORTRAN versions. Third, we then discussed the optimization of communication between MPI tasks specifically related to suspended solid particles. We have proposed and compared three different data communication schemes, namely, slice-based, flag-based, and direct-request. While the most complicated of the three, the direct-request communication scheme exhibited the best scalability and performance under a variety of different volume fractions and particle radii.

When compared with our previous less optimized particle-laden simulation code used in our previous particle-laden channel flow studies [28,29], we found that a *4.0–8.5 times* speed up has been achieved. However, we believe that there is still plenty of opportunity to further improve the performance of our simulation and LBM in general by more efficiently using SIMD instructions or employing GPUs as many have done in the past. Additionally, significant work remains in optimizing particle related functions if one wishes to resolve a large number of solid particles with reasonable performance. These topics may be the focus of future works.

Acknowledgments

This work has been supported by the U.S. National Science Foundation (NSF) under grants CNS1513031, CBET-1235974, and AGS-1139743 and by Air Force Office of Scientific Research under grant FA9550-13-1-0213. Computing resources are provided by National Center for Atmospheric Research through CISL-P35751014, and CISL-UDEL0001 and by University of Delaware through NSF CRI 0958512.

References

[1] C.K. Aidun, J.R. Clausen, Lattice-Boltzmann method for complex flows, Annu. Rev. Fluid Mech. 42 (2010) 439–472.
[2] S. Succi, The Lattice Boltzmann Equation: For Fluid Dynamics and Beyond, Oxford university press, 2001.
[3] X. Shan, H. Chen, Lattice Boltzmann Model for Simulating Flows with Multiple Phases and Components, Phys. Rev. E 47 (3) (1993) 1815.
[4] W. Guo, C. Jin, J. Li, High performance lattice Boltzmann algorithms for fluid flows, in: Information Science and Engineering, 2008. ISISE'08. International Symposium on, volume 1, IEEE, 2008, pp. 33–37.
[5] J. Wilke, T. Pohl, M. Kowarschik, U. Rüde, Cache performance optimizations for parallel lattice Boltzmann codes, in: Euro-Par 2003 Parallel Processing, Springer, 2003, pp. 441–450.
[6] M. Wittmann, T. Zeiser, G. Hager, G. Wellein, Comparison of different propagation steps for lattice Boltzmann methods, Comput. Math. Appl. 65 (6) (2013) 924–935.
[7] G. Wellein, T. Zeiser, G. Hager, S. Donath, On the single processor performance of simple lattice Boltzmann kernels, Comput. Fluids 35 (8) (2006) 910–919.
[8] A.G. Shet, K. Siddharth, S.H. Sorathiya, A.M. Deshpande, S.D. Sherlekar, B. Kaul, S. Ansumali, On vectorization for lattice based simulations, Int. J. Modern Phys. C 24 (12) (2013) 1340011.
[9] K. Mattila, J. Hyväluoma, T. Rossi, M. Aspnäs, J. Westerholm, An efficient swap algorithm for the lattice Boltzmann method, Comput. Phys. Commun. 176 (3) (2007) 200–210.
[10] T. Pohl, M. Kowarschik, J. Wilke, K. Iglberger, U. Rüde, Optimization and profiling of the cache performance of parallel lattice Boltzmann codes, Parallel Process Lett. 13 (04) (2003) 549–560.
[11] P. Bailey, J. Myre, S.D. Walsh, D.J. Lilja, M.O. Saar, Accelerating lattice Boltzmann fluid flow simulations using graphics processors, in: Parallel Processing, 2009. ICPP'09. International Conference on, IEEE, 2009, pp. 550–557.
[12] A. Dupuis, From a lattice Boltzmann model to a parallel and reusable implementation of a virtual river, Diss. Thèse soutenue à Genève en Suisse (2002).
[13] L. Axner, J. Bernsdorf, T. Zeiser, P. Lammers, J. Linxweiler, A.G. Hoekstra, Performance evaluation of a parallel sparse lattice Boltzmann solver, J. Comput. Phys. 227 (10) (2008) 4895–4911.
[14] D. Vidal, R. Roy, F. Bertrand, On improving the performance of large parallel lattice Boltzmann flow simulations in heterogeneous porous media, Comput. Fluids 39 (2) (2010) 324–337.
[15] M. Wittmann, T. Zeiser, G. Hager, G. Wellein, Modeling and analyzing performance for highly optimized propagation steps of the lattice boltzmann method on sparse lattices, arXiv preprintarXiv:1410.0412(2014).
[16] M. Bouzidi, M. Firdaouss, P. Lallemand, Momentum transfer of a Boltzmann–lattice fluid with boundaries, Phys. Fluids (1994-present) 13 (11) (2001) 3452–3459.
[17] D. Yu, R. Mei, L.-S. Luo, W. Shyy, Viscous flow computations with the method of lattice Boltzmann equation, Prog. Aerosp. Sci. 39 (5) (2003) 329–367.
[18] B. Chun, A. Ladd, Interpolated boundary condition for lattice Boltzmann simulations of flows in narrow gaps, Phys. Rev. E 75 (6) (2007) 066705.
[19] K. Stratford, I. Pagonabarraga, Parallel simulation of particle suspensions with the lattice Boltzmann method, Comput. Math. Appl. 55 (7) (2008) 1585–1593.
[20] A. Ladd, R. Verberg, Lattice-Boltzmann simulations of particle–fluid suspensions, J. Stat. Phys. 104 (5–6) (2001) 1191–1251.
[21] C. Peng, Y. Teng, B. Hwang, Z. Guo, L.-P. Wang, Implementation issues and benchmarking of lattice Boltzmann method for moving rigid particle simulations in a viscous flow, Comput. Math. Appl. 72 (2) (2016) 349–374.
[22] Q. Xiong, B. Li, J. Xu, X. Wang, L. Wang, W. Ge, Efficient 3D DNS of gas–solid flows on Fermi GPGPU, Comput. Fluids 70 (2012) 86–94.
[23] L.-P. Wang, O.G. Ardila, O. Ayala, H. Gao, C. Peng, Study of local turbulence profiles relative to the particle surface in particle-laden turbulent flows, J. Fluids Eng. 138 (4) (2016) 041307.
[24] H. Gao, H. Li, L.-P. Wang, Lattice Boltzmann simulation of turbulent flow laden with finite-size particles, Comput. Math. Appl. 65 (2) (2013) 194–210.
[25] P. Lallemand, L.-S. Luo, Lattice Boltzmann method for moving boundaries, J. Comput. Phys. 184 (2) (2003) 406–421.
[26] A. Caiazzo, Analysis of lattice Boltzmann nodes initialisation in moving boundary problems, Prog. Comput. Fluid Dyn., Int. J. 8 (1–4) (2008) 3–10.
[27] L.-P. Wang, O. Ayala, H. Gao, C. Andersen, K.L. Mathews, Study of forced turbulence and its modulation by finite-size solid particles using the lattice Boltzmann approach, Comput. Math. Appl. 67 (2) (2014) 363–380.
[28] L.-P. Wang, C. Peng, Z. Guo, Z. Yu, Flow modulation by finite–size neutrally buoyant particles in a turbulent channel flow, J. Fluids Eng. 138 (4) (2016) 041306.
[29] L.-P. Wang, C. Peng, Z. Guo, Z. Yu, Lattice Boltzmann simulation of particle-laden turbulent channel flow, Comput. Fluids 124 (2016) 226–236.
[30] D. d'Humières, I. Ginzburg, M. Krafczyk, P. Lallemand, L.-S. Luo, Multiple–relaxation–time lattice Boltzmann models in three dimensions, Philos. Trans. R. Soc.London A 360 (1792) (2002) 437–451.
[31] C. Pan, L.-S. Luo, C.T. Miller, An evaluation of lattice Boltzmann schemes for porous medium flow simulation, Comput. Fluids 35 (8) (2006) 898–909.
[32] X. He, L.-S. Luo, Lattice Boltzmann model for the incompressible Navier–Stokes equation, J. Stat. Phys. 88 (3–4) (1997) 927–944.
[33] Z. Guo, C. Zheng, B. Shi, Discrete lattice effects on the forcing term in the lattice Boltzmann method, Phys. Rev. E 65 (4) (2002) 046308.
[34] M. Schulz, M. Krafczyk, J. Tölke, E. Rank, Parallelization strategies and efficiency of CFD computations in complex geometries using lattice Boltzmann methods on high-performance computers, in: High Performance Scientific and Engineering Computing, Springer, 2002, pp. 115–122.
[35] K. Mattila, J. Hyväluoma, J. Timonen, T. Rossi, Comparison of implementations of the lattice–Boltzmann method, Comput. Math. Appl. 55 (7) (2008) 1514–1524.

[36] A.G. Shet, S.H. Sorathiya, S. Krithivasan, A.M. Deshpande, B. Kaul, S.D. Sherlekar, S. Ansumali, Data structure and movement for lattice-based simulations, Phys. Rev. E 88 (1) (2013) 013314.
[37] P.J. Denning, The locality principle, Commun. ACM 48 (7) (2005) 19–24.
[38] P. Lammers, K. Beronov, R. Volkert, G. Brenner, F. Durst, Lattice BGK direct numerical simulation of fully developed turbulence in incompressible plane channel flow, Comput. Fluids 35 (10) (2006) 1137–1153.
[39] I. Corporation, Intel 64 and IA-32 architectures optimization reference manual, 2016, (http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html).
[40] S. Williams, J. Carter, L. Oliker, J. Shalf, K. Yelick, Lattice Boltzmann simulation optimization on leading multicore platforms, in: Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on, IEEE, 2008, pp. 1–14.
[41] I. Ginzburg, F. Verhaeghe, D. d'Humieres, Two-relaxation-time lattice Boltzmann scheme: about parametrization, velocity, pressure and mixed boundary conditions, Commun. Comput. Phys. 3 (2) (2008) 427–478.
[42] K.R. Baker, Introduction to Sequencing and Scheduling, John Wiley & Sons, 1974.