2015 44th International Conference on Parallel Processing

LBM-IB: A Parallel Library to Solve 3D Fluid-Structure Interaction Problems on Manycore Systems

Prateek Nagar*, Fengguang Song*, Luoding Zhu† and Lan Lin‡
*Department of Computer Science
Indiana University-Purdue University, Indianapolis, Indiana, USA
Email: {pnagar, fgsong}@iupui.edu
†Department of Mathematics
Indiana University-Purdue University, Indianapolis, Indiana, USA
Email: lzhu@math.iupui.edu
‡Department of Computer Science
Ball State University, Muncie, Indiana, USA
Email: llin4@bsu.edu

***Abstract*—Deformable structures are abundant in various domains such as biology, medicine, life sciences, and ocean engineering. Our previous work created a numerical method, named *LBM-IB* method [1], to solve the fluid-structure interaction (FSI) problems. Our LBM-IB method is particularly suitable for simulating flexible (or elastic) structures immersed in a moving viscous fluid. Fluid-structure interaction problems are well known for their heavy demands on computing resources. Today, it is still challenging to resolve many real-world FSI problems. In order to solve large-scale fluid-structure interactions more efficiently, in this paper, we design a parallel LBM-IB library on shared memory manycore architectures. We start from a sequential version, which is extended to two different parallel versions. The paper first introduces the mathematical background of the LBM-IB method, then uses the sequential version as a ground to present our implemented computational kernels and the algorithm. Next, it describes the two parallel programs: an OpenMP implementation and a cube-based parallel implementation using Pthreads. The cube-based implementation builds upon our new cube-centric algorithm where all the data are stored in cubes and computations are performed on individual cubes in a data-centric manner. By exploiting better data locality and fine-grain block parallelism, the cube-based parallel implementation is able to outperform the OpenMP implementation by up to 53% on 64-core computer systems.**

***Keywords*-High performance computing; computational fluid dynamics (CFD); fluid-structure interactions (FSI); immersed boundary methods (IB); manycore systems**

I. Introduction

Computational fluid dynamics (CFD) is a crucial area with various numerical methods to solve a wide range of important scientific, engineering, and life sciences applications. Among all the different CFD applications, simulation of the puzzling and intricate fluid-structure interactions (FSI) is an active field of research, where development of any faster and more scalable parallel library will enable more accurate numerical simulations of important real-world FSI problems. This paper presents a parallel multithreaded library, called LBM-IB, to solve the FSI problems on shared memory manycore architectures.

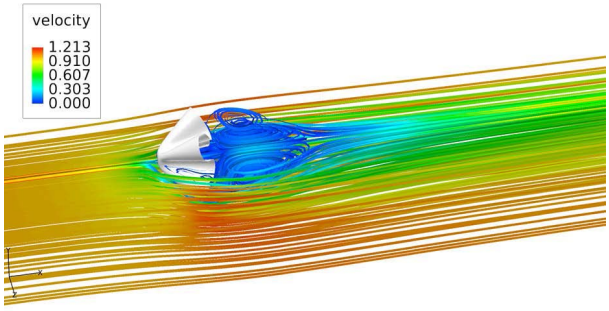

Figure 1. A 3D simulation of fluid and flexible-structure interaction using the LBM-IB method [1].

The fluid-structure interaction problems are occurring in many physical and industrial environments as well as in Mother Nature, for instance, parachutes dropping from aircrafts, ships cruising in oceans, blood cells flowing past human vessels, sea creatures swimming in the water, and so on [2]–[4]. Our scope of research on fluid-structure interactions focuses on the complicated and detailed interplay among viscous fluids, deformable bodies, and the free moving boundaries separating the fluid and the bodies. Due to the complexity of the type of problems, analytic solutions are almost impossible to obtain and hence computational methods are typically used by researchers.

There are quite a few computational methods for solving fluid-structure interaction problems: the Arbitrary Lagrangian Eulerian (ALE) method [5], the fictitious domain method [6], the material point method [7], and the immersed boundary (IB) method, to name a few. Our paper uses the immersed boundary method, which was the first method to address the full interactions of a viscous fluid and *flexible (or elastic) structures*. The immersed boundary method is

0190-3918/15 $31.00 © 2015 IEEE
DOI 10.1109/ICPP.2015.14

comprised of three basic elements: 1) solving the motion of the viscous incompressible fluid, 2) solving the motion of the flexible structures, and 3) handling interactions of the fluid and the structure by the Dirac delta function. Note that the third element is the key strength of the immersed boundary method, which makes the IB method attractive.

Our previous work [1] developed an immersed boundary method, which utilizes the 3-D lattice Boltzmann method (LBM) [8], [9] to model the fluid flow and integrates the elastic boundary forces with LBM. We refer to our new immersed boundary method as the *LBM-IB* method. Figure 1 shows our 3D simulation of a flexible circular plate fastened in the middle region and immersed in a fluid flow using this LBM-IB method. In our LBM-IB method, we execute the following operations in every time step: 1) compute the elastic forces coming from the immersed structure, 2) spread the elastic forces from the immersed structure to the fluid, 3) solve the fluid's velocity via LBM using the elastic boundary forces, 4) derive the structure's velocity based on the velocity of the nearby fluid, and finally 5) update the location of the structure (i.e. the structure is moving).

However, to date, there is no parallel software to support the LBM-IB method. The goal of this paper is to develop a parallel LBM-IB library for shared memory manycore architectures. To design general-purpose software for many different applications, we have implemented the library from scratch. The LBM-IB library includes an easy-to-use application programming interface, new data structures to represent fluid grids and flexible structures, and parallel algorithms tailored for manycore systems. This paper presents three programs: 1) a sequential version which focuses on algorithm correctness and efficient data structures, 2) a parallel OpenMP implementation, and 3) a block-based implementation using Pthreads. Although the paper is focused on parallel software, the sequential program is used as a foundation to describe the modified LBM-IB algorithm and the related computational kernels.

We conduct detailed performance analysis using gprof, OmpP [10], and PAPI [11]. Although the OpenMP implementation works well for a small number of cores, we find the bottlenecks of load imbalance and poor locality in the program when using more than 8 CPU cores. To improve its performance, we design a cube-based LBM-IB algorithm to maximize the degree of parallelism, obtain load balance, and increase data locality. In the cube-based algorithm, an input of fluid grid is divided into a set of fine-grain 3D blocks (also called "cubes"). Each cube is then stored in a continuous block of memory. During the program execution, the algorithm first distributes the set of fine-grain cubes to each thread. Next, each thread applies the LBM-IB computational kernels to its assigned subset of cubes iteratively. We also minimize the number of barriers in each time step to reduce the global synchronization overhead.

The experimental results on a 32-core AMD machine show that the OpenMP implementation achieves good scalability from 1 to 8 cores with a parallel efficiency of 75%. Another experiment on a 64-core machine show that the new cube-based parallel implementation can further improve the OpenMP program by up to 53%.

This paper has made the following contributions:

- It is for the first time to realize a parallel library to enable the numerical LBM-IB method on manycore architectures.
- We propose a new cube-based parallel LBM-IB algorithm to exploit parallelism and locality to obtain higher performance. The same idea can also be adapted and applied to other different 3D CFD applications.
- We present detailed performance analysis and provide an insight into identifying bottlenecks and performing code optimizations to develop CFD libraries from sequential version to different parallel versions on manycore systems.

The rest of the paper is organized as follows. Next section introduces the basic idea of our LBM-IB method. Section III describes the sequential implementation of the LBM-IB method, computational kernels, and its performance profiles. Section IV describes the parallel OpenMP implementation, and Section V describes the cube-based algorithm and implementation. Section VI shows the experimental results. Finally, Section VII presents the related work and Section VIII summarizes the paper.

II. Background

A. Immersed Boundary Method

The immersed boundary (IB) method, the first method created for fluid-structure interaction problems, was originated by CS Peskin [12], [13] in 1970s for numerical investigation of flow patterns of blood flow around human heart valves. It has since become a generic method to numerically solve problems involving fluid-structure interactions. Compared to other approaches for fluid-structure interactions, the IB method is more attractive because it has significantly reduced the complexity of the complicated interactions between the fluid and the structure by using the Dirac delta function.

The IB method is both a unique mathematical formulation and a numerical method (for its mathematical formulation). In the IB method, fluid-related variables (such as velocity, mass density, pressure, and elastic force passed from structure to fluid) are defined on a fixed uniform Eulerian grid. Structure-related variables (such as position, velocity, and elastic force to the fluid) are defined on a moving Lagrangian array of points which do not necessarily coincide with the fixed uniform Eulerian grid of the fluid.

The Navier-Stokes equations which describes the motion of fluid are discretized on the fixed Eulerian grid, while the flexible-structure equations are discretized on the moving Lagrangian grid. The transfers between the Lagrangian

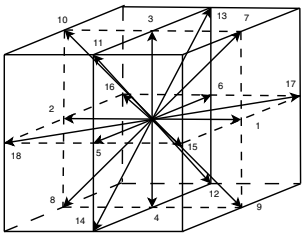

Figure 2. The Lattice Boltzmann D3Q19 model. A particle at the center can move along 18 different directions.

variables and the corresponding Eulerian variables are performed through the Dirac delta function. The mathematical formulation of IB method using Dirac delta function has been rigorously proved to be exact (i.e., no modeling errors from idealization or simplification).

The IB method essentially consists of three components: fluid, solid, and the interaction. While the solid and interaction parts are more or less the same within the IB method, there are different approaches to handle and solve the fluid motions. In our work, the Navier-Stoke equations are solved by the lattice Boltzmann method (described in the next subsection).

B. Lattice Boltzmann Method

The Lattice Boltzmann method (LBM) employs a mesoscale description to deal with the velocity distribution function that obeys an approximate Boltzmann equation. Instead of solving for the macroscopic quantities of velocity and pressure directly, LBM deals with the single particle velocity distribution functions $g(\mathbf{x}, \alpha, t)$ ($\mathbf{x}$ represents the spatial coordinate, α is the particle velocity, and t is the time variable) based on the Boltzmann equation.

In our work, we take advantage of the LBM method and adapt the LBM D3Q19 model to model the fluid motion. Base on the D3Q19 model, at any spatial node, the fluid particles may move along 18 different directions (as shown in Figure 2). The particles are also allowed to stay at the center. The LBM method is of second-order accuracy in both time and space. Compared to conventional methods, LBM is relatively simpler to use, easier to parallelize, and more convenient to incorporate additional physics to simulate new flow phenomena, particularly in 3D. These advantages have been fully utilized by this work.

C. The LBM-IB Coupling for Fluid-Structure Interactions

The two-way full interaction between the fluid and the immersed structure is handled by a smoothed approximation of the Dirac delta function. This is mediated by interpolating the structure velocity from the fluid velocity and spreading the elastic force from the strucutre to the fluid. This way the fluid "feels" the existence/influence of the structure because it receives the elastic force from the structure meanwhile the structure "feels"the existence/influence of the fluid because it must move with the fluid (i.e. its velocity is dictated by the fluid). In other words, the two-way interaction of fluid and structure is by means of force spreading and velocity interpolation, both of which build upon the smoothed Dirac delta function.

Our special LBM-IB coupling method works as follows: 1) compute the elastic force from the configuration of the immersed structure (i.e. the flexible structure that has been stretched or bent); 2) spread the elastic force from the structure to fluid by the Dirac delta function; 3) solve the fluid equations with the LBM using the elastic force from structure; 4) compute the velocity of the structure based on the velocity of the fluid by the Dirac delta function; 5) update the configuration of the structure; 6) time_step = time_step + 1; go to step 1) until termination time.

III. The Sequential LBM-IB Implementation

This section introduces how to create the input, the LBM-IB computational kernels, the sequential implementation and its performance profile.

A. Input of the Algorithm

The LBM-IB algorithm takes as input a 3D fluid grid and an immersed structure, which consists of a set of flexible fibers. As shown in Figure 3, the 3D fluid grid is built as a structured $N_x \times N_y \times N_z$ mesh, where each fluid coordinate (x, y, z) (also known as fluid node) records the characteristics of the fluid particles near the location of (x, y, z). On the other hand, Figure 4 shows an example of a flexible structure of 2D sheet. The 2D sheet structure is comprised of an array of fibers, each of which consists of a list of fiber nodes. For a 3D flexible structure, it can be comprised of a number of 2-D sheets.

When executing CFD simulations, the LBM-IB algorithm applies a series of steps to each fluid node and each fiber node to compute new properties of the fluid and the fiber nodes in every time step. All the time steps in the end compose a complete CFD simulation. The properties of a fluid node contain information such as velocity, pressure,

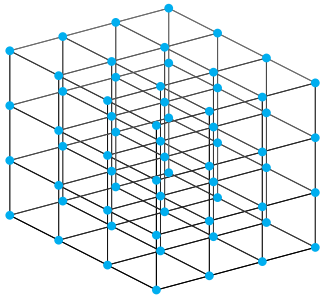

Figure 3. A 3D fluid grid of dimension $4 \times 4 \times 4$. Each coordinate (x,y,z) describes the characteristics of the fluid particles in the location of (x,y,z).

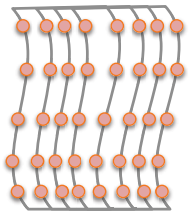

Figure 4. A flexible fiber sheet consisting of 8 fibers. Each fiber has 5 fiber nodes.

vorticity, shear stress, and so on. The properties of a fiber node include information of bending force and stretching force to model flexible or elastic fibers.

B. Computational Kernels

We have implemented 9 kernels to compute the numerical simulation results. Here we briefly introduce the execution patterns of these kernels as follows. (for mathematical formulas, please refer to our previous work [1]).

1) *compute_bending_force_in_fibers()*: visits every fiber node and computes the bending force of the fiber node, which depends on the locations of its 8 neighbor fiber nodes in a 2D surface (i.e., two nodes on the left, two nodes on the right, two nodes above, and two nodes below).
2) *compute_stretching_force_in_fibers()*: visits every fiber node and computes the stretching force based on the distances between the current fiber node and its four neighbors on the left, right, top, and bottom, respectively.
3) *compute_elastic_force_in_fibers()*: The elastic force of each fiber node is the sum of the node's bending and stretching forces.
4) *spread_force_from_fibers_to_fluid()*: It first finds the set of fluid nodes located in the $4 \times 4 \times 4$ space (named influential domain) around a given fiber node, then the center fiber node's elastic force will be exerted onto the surrounding fluid nodes within the influential domain.
5) *compute_fluid_collision()*: It applies the lattice Boltzmann method to solve the collision at each fluid grid node in 19 directions defined by the D3Q19 model.
6) *stream_fluid_velocity_distribution()*: After computing the collision on a fluid node, it streams (or copies) the newly computed velocity distribution of the fluid node to its 18 immediate neighbors (as shown in Figure 2).
7) *update_fluid_velocity()*: It updates the new velocity for each fluid node given the new velocity distribution (from kernel 6) and the fiber's exerted elastic force (from kernel 4).
8) *move_fibers()*: Similar to *spread_force* (kernel 4), it first finds the set of fluid nodes in the influential domain for each fiber node; then computes the fiber node's new position based on the velocity of those surrounding fluid nodes.
9) *copy_fluid_velocity_distribution()*: There are two buffers to store the present velocity distribution and the new velocity distribution, respectively. Before going from the i-th time step to the (i+1)-th time step, the function copies data from the buffer of new distributions to the buffer of present distributions so that the space of the new distribution buffer can be reused.

C. The Sequential Algorithm

Algorithm 1 shows the LBM-IB algorithm. It first creates an immersed structure and a 3-D fluid grid. Next it executes the 9 computational kernels repeatedly to simulate each time step. We can category the kernels into three classes: the IB-related kernels, the LBM-related kernels, and the coupling-related kernels. In our implementation, the structure is represented by a number of 2D sheets, each of which consists of a set of flexible fibers.

As soon as the simulation starts, each fiber starts to interact with its surrounding fluid particles, and may move, bend, stretch, or even rotate. At the same time, the velocity of a fluid particle is changed dynamically by the elastic forces applied by its surrounding fibers.

Algorithm 1 Sequential LBM-IB Algorithm

```
immersed_struct = create_fiber_shape();
fluid = create_fluid_grid();
for timestep ← 0 to N-1 do
  /* IB related */
  1) compute_bending_force_in_fibers( immersed_struct );
  2) compute_stretching_force_in_fibers( immersed_struct );
  3) compute_elastic_force_in_fibers( immersed_struct );
  4) spread_force_from_fibers_to_fluid(immersed_struct,fluid)
  /* LBM related */
  5) compute_fluid_collision( fluid );
  6) stream_fluid_velocity_distribution( fluid );
  /* FSI-coupling related */
  7) update_fluid_velocity( fluid );
  8) move_fibers( immersed_struct, fluid );
  9) copy_fluid_velocity_distribution( fluid );
end for
```

D. Performance Analysis

Before parallelizing the code, we conduct performance analysis to identify bottlenecks and hot regions of the program. The analysis is done on a Linux machine with two AMD Opteron 16-core Abu Dhabi 2.9GHz CPUs and memory of 64 GB. The sequential LBM-IB program takes as input a 3D fluid grid of dimension $124 \times 64 \times 64$ and an immersed 2D sheet of dimension 20×20 with 52×52 fiber nodes. The execution time is around 967 seconds to simulate the fluid-structure interaction for 500 time steps.

Table I lists the nine kernels that are ranked in the order of execution time. The first column displays the index of

Table I
PERFORMANCE ANALYSIS OF THE SEQUENTIAL LBM-IB PROGRAM WITH `gprof`. TOTAL EXECUTION TIME = 967 SECONDS.

Kernel Index	Kernel Name	Percentage of Total Time
5)	compute_fluid_collision	73.2%
7)	update_fluid_velocity	12.6%
9)	copy_fluid_velocity_distribution	5.9%
6)	stream_fluid_velocity_distribution	5.4%
4)	spread_force_from_fibers_to_fluid	1.4%
8)	move_fibers	0.7%
1)	compute_bending_force_in_fibers	0.03%
2)	compute_stretching_force_in_fibers	0.02%
3)	compute_elastic_force_in_fibers	0.00%

each kernel, which is the same as that used in Algorithm 1. From the table, we can see that the time spent by the top four kernels take up 97% of the total execution time. By inspecting the compute patterns of the 9 kernels (see Section III-B), we discover that the top four kernels are the only kernels that have visited every single fluid node and computed results on every fluid node.

Due to the large 3D fluid space, any kernel that attempts to visits every fluid node will be both compute intensive and memory intensive. For instance, the third-ranked kernel *copy_fluid_velocity_distribution* simply copies the velocity distribution from one buffer to another buffer for all the fluid nodes. However, this simple memory operation has taken noteworthy 5.9% of the total execution time. The fourth-ranked kernel *stream_fluid_velocity_distribution* is similar to the third-ranked kernel *copy_fluid_velocity_distribution* and also takes 5.4% of the total execution time.

Although this paper does not cover how to optimize kernel code, the performance profiling provides an insight into the cost of the kernels as well as the need for optimization on memory efficiency.

IV. THE PARALLEL IMPLEMENTATION USING OPENMP

A. OpenMP Implementation

OpenMP is a portable shared-memory programming model that can quickly convert a sequential program to a multi-threaded parallel program with high performance [14]. To use the OpenMP programming model, we have inspected every *for* loop and determined what and where data dependencies are and inserted OpenMP pragmas into the sequential code to parallelize it.

As shown in Algorithm 1, the sequential program mainly consists of two types of kernels: one type visits every fluid node resident in a 3-D space; and the other type visits every fiber node resident in a 2-D space. The four most expensive kernels in Table I (e.g. compute_fluid_collision and update_fluid_velocity) belong to the first type. The remaining five less expensive kernels (e.g. spread_force_from_fibers_to_fluid and compute_bending_force_in_fibers) belong to the second type. To present the OpenMP implementation, we use two pseudocodes to describe the two types of kernels: one pseudocode for fluid-node computing (Algorithm 2), and the other for fiber-node computing (Algorithm 3).

Algorithm 2 Fluid-computing kernels with OpenMP

```
/* Compute for every fluid node */
#pragma omp parallel for default(shared) private (x, y, z, direction)
for x ← 0 to N_x do
  for y ← 0 to N_y do
    for z ← 0 to N_z do
      for direction ← 0 to 18 do
        fluid_nodes[x,y,z].distri_freq[direction]
        ← function(properties of fluidnodes[x,y,z]);
      end for
    end for
  end for
end for
```

As shown in Algorithm 2, a specific computational function is applied to every fluid node to compute either new distribution function or velocity, or perform streaming. In our implementation, we use the static scheduling policy to divide a 3D fluid grid into multiple contiguous segments of 2D surfaces, where each surface is aligned with the y-z coordinate and vertical to the x axis. Next, each segment is assigned to a different thread to achieve parallelism. We have also tried the dynamic scheduling policy but obtained the same performance.

Algorithm 3 shows the pseudocode to calculate various forces between a fiber node and its neighbor fiber nodes (e.g. bending and stretching forces). The computation goes through two stages: it first goes along each fiber to compute a partial force, then it visits the fiber nodes along each column which is vertical to the fibers.

B. Performance Analysis

To evaluate the performance of the OpenMP program, we run experiments on the same Linux machine with two AMD Opteron 16-core 2.9GHz CPUs and a memory of 64GB. The experiments take the same input as that used by the sequential implementation and execute 200 time steps on a varying number of CPU cores from 1 to 32. As shown in Figure 5, the speed up is good till 8 cores for which the parallel efficiency is 75%. However, when using 16 and 32 cores, the parallel efficiency drops quickly to 56% and 38%, respectively. Figure 5 also displays the ideal speedup, which is equal to the number of CPU cores.

To understand why the performance degrades as the number of CPU cores increases. We use an OpenMP profiler called OmpP [10] and the PAPI [11] library to investigate if there are bottlenecks and where they are. Table II lists the

Algorithm 3 Fiber-computing kernels with OpenMP

```
/* Compute along each fiber */
#pragma omp parallel for default(shared) private (fiber, node)
for fiber ← 0 to num_fibers do
  for node ← 0 to num_nodes do
    fiber_nodes[fiber,node].certain_force ← function(left
    and right neighbors of fiber_nodes[fiber,node]);
  end for
end for

/* Compute vertically to all fibers */
#pragma omp parallel for default(shared) private (fiber, node)
for node ← 0 to num_nodes do
  for fiber ← 0 to num_fibers do
    fiber_nodes[fiber,node].certain_force += function'(upper
    and lower neighbors of fiber_nodes[fiber,node]);
  end for
end for
```

measured L1 data cache miss rate, the L2 data cache miss rate, and load imbalance with respect to the whole program. From the table, we can see that the L1 cache miss rate is good, but the L2 cache miss rate (>25%) is significantly high. This indicates that the program has a poor locality and a large memory footprint. Furthermore, the load imbalance ratio (relative to the whole program) keeps increasing from 8 cores to 32 cores. To eliminate the bottlenecks, we redesign a new parallel algorithm, called cube-based algorithm, to increase both data locality and the degree of parallelism.

V. The Cube-Centric Algorithm

This section describes the new cube-based algorithm and our new implementation using Pthreads.

A. The Algorithm

Being designed to be data centric, the cube-based algorithm first divides an input of 3D fluid grid into a 3D array of sub-grids (we refer to them as cubes). Each cube consists

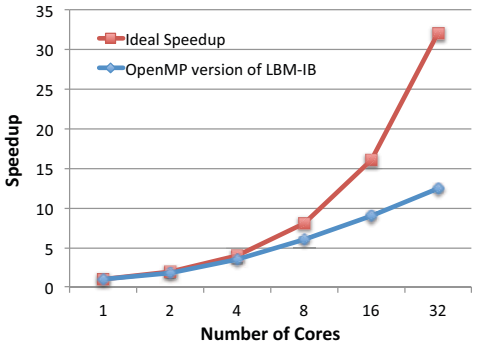

Figure 5. Performance of the OpenMP implementation of LBM-IB on a 32-core system. Speedup is relative to the execution time of one-core experiment.

Table II
Performance metrics data collected for the OpenMP implementation of LBM-IB.

Cores	L1 miss rate	L2 miss rate	Load imbalance
1	1.76%	26.1%	0%
2	1.75%	26.1%	1.8%
4	1.75%	26.1%	1.4%
8	1.75%	26.2%	5.1%
16	1.74%	27.1%	11%
32	1.76%	27.6%	13%

of $k \times k \times k$ individual fluid nodes, which are stored in a contiguous memory block. This implies a much smaller working set size and a better locality than the original algorithm. A fluid grid of $N_x \times N_y \times N_z$ is now considered as $\frac{N_x}{k} \times \frac{N_y}{k} \times \frac{N_z}{k}$ cubes.

Given a number of n threads, the set of $\frac{N_x}{k} \times \frac{N_y}{k} \times \frac{N_z}{k}$ cubes will be statically mapped to different threads using a user-defined data distribution function. The set of n threads will be laid out in a 3D mesh such that $n = P \times Q \times R$, where P, Q, R are dimensions of the thread mesh. The distribution function is defined as a function of *int cube2thread(cube$_x$, cube$_y$, cube$_z$)*, which calculates the mapped thread ID for a given cube whose coordinate is (cube$_x$, cube$_y$, cube$_z$). Similarly, *int fiber2thread(fiber$_i$)* maps fibers to different threads. The distribution function may define different data distribution methods such as block distribution, cyclic distribution, or block cyclic distribution. For instance, Figure 6 shows how to map a 3D fluid grid of $4 \times 4 \times 4$ nodes (i.e., $2 \times 2 \times 2$ cubes of dimension 2) to a 3D mesh of $2 \times 2 \times 2$ threads using a block distribution method. In the thread grid, threads T_0, T_1, T_2, T_3 are on the first layer and threads T_4, T_5, T_6, T_7 are on the second layer. In the example, after data distribution, each thread is assigned a single cube of dimension 2.

During program execution, every thread is only responsible for computing its own subset of cubes. The thread will follow the same 9 steps described in Section III-B such as *compute_bending_force_in_fibers()*, *com-*

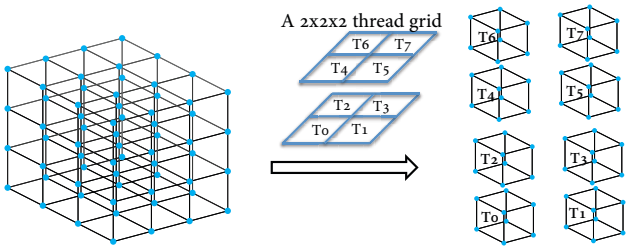

Figure 6. Mapping a $4 \times 4 \times 4$ fluid grid to a $2 \times 2 \times 2$ thread grid. After distribution, each thread T_i owns a single cube of dimension 2.

Algorithm 4 Multithreaded cube-centric LBM-IB algorithm

```
main() function
fibers[Num_fibers] = create_fiber_shape();
cubes[N_x,N_y,N_z] = create_fluid_grid();
for tid ← 0 to num_threads-1 do
   create_thread(Thread_entry_fn, tid, fibers, cubes)
end for

Thread_entry_fn(int tid, void* fibers, void* cubes)
for timestep ← 0 to N-1 do
   for each fiber i /* 1st loop */ do
      if fiber2thread(i) == tid then
         1) compute_bending_force_in_fibers(fibers[i]);
         2) compute_stretching_force_in_fibers(fibers[i]);
         3) compute_elastic_force_in_fibers(fibers[i]);
         4) spread_force_from_fibers_to_fluid(fibers[i]);
      end if
   end for
   for each cube of cubes[I,J,K] /* 2nd loop */ do
      if cube2thread(I,J,K) == tid then
         5) compute_fluid_collision(cube);
         6) stream_fluid_velocity_distribution(cube);
      end if
   end for
   thread_barrier_wait();
   for each cube of cubes[I,J,K] /* 3rd loop */ do
      if cube2thread(I,J,K) == tid then
         7) update_fluid_velocity(cube);
      end if
   end for
   thread_barrier_wait();
   for each fiber i /* 4th loop */ do
      if fiber2thread(i) == tid then
         8) move_fibers(fibers[i]);
      end if
   end for
   for each cube of cubes[I,J,K] /* 5th loop */ do
      if cube2thread(I,J,K) == tid then
         9) copy_fluid_velocity_distribution(cube);
      end if
   end for
   thread_barrier_wait();
end for
```

pute_stretching_force_in_fibers(), and so on. However, all the new computational kernels will take cubes as input and follow the cube-centric algorithm to compute results.

To enforce mutual exclusions among threads, every thread has a private lock to protect its subset of cubes. In other words, a cube will be protected by its owner thread's private lock. If a cube can be modified by different threads, all the threads will try to acquire the cube's owner lock (which is unique across all the threads) before reading or writing the cube.

B. Parallel Implementation with Pthreads

Algorithm 4 displays the pseudocode of the cube-based LBM-IB algorithm implemented with Pthreads. In the main() function, a 1D array of fibers and a 3D array of cubes will be created first. Then the function will launch a number of threads, each of which executes the thread function Thread_entry_fn() in parallel.

When a thread starts, it will follow nine steps in a coordinated way. For each time step, every thread will go through five *for* loop nests, whose functions are described as follows: (i) the first *for* loop checks every fiber; if the fiber belongs to the current thread, it will compute the elastic force for the fiber. The data distribution function fiber2thread() guarantees that one fiber is only assigned to one thread. (ii) the second *for* loop checks every cube with a coordinate of [I,J,K]; if the cube belongs to the thread, it will compute the collision for each fluid node inside that cube, followed by velocity streaming. (iii) the third *for* loop checks every cube and computes new velocities for its assigned subset of cubes. (iv) the fourth *for* loop checks every fiber and moves the fiber's position if the fiber belongs to the thread. (v) in the fifth *for* loop, before entering the next time step, the thread copies velocity distributions from the new-distribution buffer to the present-distribution buffer for its assigned subset of cubes.

Note that there are several global barriers in the code. They are necessary because there exist data dependencies between the current kernel and the next kernel. For instance, the first thread_barrier_wait() is needed, because to compute velocity for a single fluid node in update_fluid_velocity() we will have to read 18 neighbor fluid nodes computed from the previous stream_fluid_velocity_distribution(). Since the 18 fluid nodes can reside in different threads, a barrier is inserted to make sure all the input are in place before executing the next kernel. The rest of the barriers are added to the code for the similar reason of data dependencies.

VI. Experimental Results

We conducted experiments on a 64-core AMD system (located at University of Tennessee Knoxville) to evaluate the performance of the parallel OpenMP LBM-IB implementation and the parallel cube-based LBM-IB implementation.

A. The Manycore System

Table III shows the manycore computer system named *thog* we used to do our experiments. The thog system consists of four AMD processors, each of which has 16 cores. On each processor, two cores share a unified L2 cache while eight cores share a unified L3 cache. The whole system has a total memory of 256 GB.

This manycore system has a deep NUMA memory hierarchy with eight NUMA (memory) nodes. As shown in the node distance table IV, the time to access a remote NUMA memory can be 2.2 times longer than the time to access a local NUMA memory. This characteristics makes data locality critical to achieve high performance on manycore architectures. For all the experiments, we have used "-O3" to compile and "numactl -interleave=all" to run the code to obtain the best performance. Also, all the numerical results

Table III
THE EXPERIMENTAL 64-CORE COMPUTER SYSTEM

	thog system
Processor type	AMD Opteron 6380 2.5 GHz
Cores per processor	16
L1 cache	16 KB per core
L2 unified cache	8 x 2 MB, each shared by two cores
L3 unified cache	2 x 12 MB, each shared by eight cores
Number of processors	4
Number of NUMA nodes	8
Cores per NUMA node	8
Memory per NUMA node	32 GB
OS	Linux 3.9.0
Compilers	gcc 64bit 4.6.3

have been verified to be correct by comparing the new result to that of the sequential implementation.

Table IV
NODE DISTANCE BETWEEN 8 DIFFERENT NUMA NODES ON *thog*, GENERATED BY COMMAND *"numactl –hardware"*.

node	0	1	2	3	4	5	6	7
0 :	10	16	16	22	16	22	16	22
1 :	16	10	22	16	22	16	22	16
2 :	16	22	10	16	16	22	16	22
3 :	22	16	16	10	22	16	22	16
4 :	16	22	16	22	10	16	16	22
5 :	22	16	22	16	16	10	22	16
6 :	16	22	16	22	16	22	10	16
7 :	22	16	22	16	22	16	16	10

B. Scalability Evaluation

We use weak scalability to evaluate the capability of our programs to solve potentially larger problems with more computing resources. For weak scalability experiments, each CPU core has a fixed amount of computation.

In our experiment, each CPU core owns a fixed number of fluid nodes. Whenever doubling the number of CPU cores, we double the total number of fluid nodes. As shown in figure 7, our experiments are used to simulate how a flexible sheet may flow and interact with a fluid in a 3D tunnel.

The fiber input size is kept the same, which consists of 104×104 fiber nodes. The fluid grid size will be increased when we increase the number of CPU cores. For instance, the input of the single core experiment takes as input $128 \times 128 \times 128$ fluid nodes. The two-core experiment takes as input $256 \times 128 \times 128$ nodes, the four-core experiment takes as input $512 \times 128 \times 128$ nodes, the eight-core experiment takes as input $256 \times 256 \times 256$ nodes, and so on.

Figure 8 shows the total execution time of the OpenMP LBM-IB implementation and the cube-based LBM-IB implementation using from one core to 64 cores. In an ideal case, the curve of execution time were to be a flat line

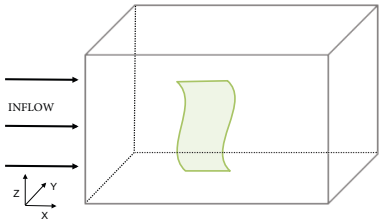

Figure 7. Experiment of a moving elastic sheet in a fluid.

because each core has a constant amount of workload and should take identical time to compute regardless of the number of cores. However, the overhead of thread synchronization such as locks and barriers, and hardware limitations such as memory bandwidth will increase the execution time gradually as the number of threads increases and as the shared memory link becomes more saturated.

Based on Figure 8, we can see that from two cores to 4 cores, the execution time of OpenMP LBM-IB increases by 25%; from 4 to 8 cores, the time increases by 36%; from 8 to 32 cores, it increase at a rate of 22% and from 32 to 64 cores, the time increases a lot by 42%. By contrast, the execution time of the cube-based LBM-IB grows more slowly than that of the OpenMP implementation. For instance, from one core to two cores, the execution time is increased by 3%. Then from two to 32 cores, the execution time increases at a constant rate of 13%; and eventually from 32 to 64 cores, the execution time increases by 18%. On 64 cores, the cube-based algorithm is able to outperform the OpenMP version by 53%. The major reason is because the cube-based algorithm is data-centric and has a better data locality (as well as a smaller working set) which alleviates the memory bandwidth bottleneck and improves the program performance.

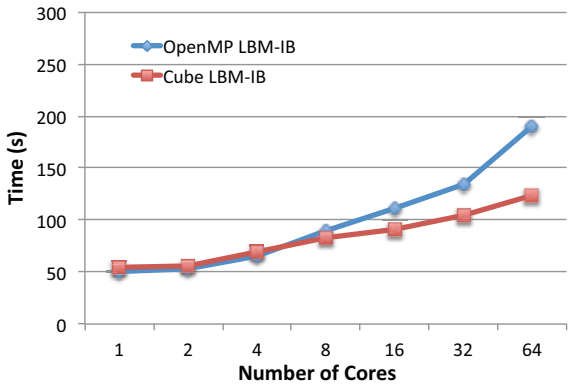

Figure 8. Weak scalability of the OpenMP-version LBM-IB implementation and the cube-based Cube LBM-IB implementation. The input size increases accordingly with the increment of the number of CPU cores.

VII. Related Work

Since Charles S. Peskin created the immersed boundary method in 1970s [12], researchers have tried to combine Peskin's IB method to other methods for more efficient numerical solutions of the fluid structure interactions. These include combinations with the level set method [15], the fictitious domain method [16], the immersed interface method [17], and the lattice-Boltzmann method. The LBM based IB method was first developed by Feng et al. [18] for particulate flows. After that, several other methods of this type have been developed. One is our LBM-IB method introduced here; another typical one was proposed by Shu et al. [19] which is particularly good for rigid-body-fluid interactions.

Although there are many different versions of the immersed boundary (IB) method, there are much less efforts in parallel implementations for the method. Here we describe the existing significant immersed boundary software and parallel algorithms as follows. IBAMR [20] is an open source parallel library of the immersed boundary method which supports adaptive mesh refinement. The IBAMR implements the versions of the IB method that uses either Fast Fourier Transforms (FFT) or the projection methods for solving the fluid motions (i.e. Navier-Stokes equations) [21]. Yelick et al. developed an IB method combined with a 3D FFT solver using the Titanium language on distributed memory systems [22]. Gotz et al. developed the LBM-FFD (fast frictional dynamics) algorithm for the simulation of particle laden flows, where a 3D lattice Boltzmann solver is used for the fluid flow and a rigid body physics engine is used for the treatment of the objects [23]. Valero-Lara investigated the type of solid-fluid interaction problem and proposed several optimization approaches [24] for multicore and GPU architectures. In this paper, we design and implement a parallel library to realize the new numerical LBM-IB method on manycore architectures.

Since the lattice Boltzmann method is a part of our LBM-IB method, we also describe the work related to LBM. Williams et al. applied an auto-tuning approach to optimize the lattice Boltzmann computational kernels on multicore systems [25]. Their work is actually complementary to our research for which we can use auto-tuning to optimize our individual cube-specific kernels. There are also implementations of the LBM method on GPU-like accelerators. For instance, Li et al. tested the LBM method on GPUs [26], Peng et al. implemented the pLBM library on a PlayStation3 cluster [27], and Tölke showed detailed implementation of the 2D LBM CUDA kernels [28].

The cube-based LBM-IB algorithm is designed to reduce the working set size of the program, increase data locality and alleviate the bottleneck on the memory bandwidth limitation. Its essential idea is similar to block/tile data layout and software blocking [29], which has been used in dense matrix, sparse matrix and CFD domains. Dongarra et al. designed a class of tile algorithms to solve linear algebra problems on multicore architectures [30], [31]. William et al. used a technique called sparse cache blocking to optimize memory-bound sparse matrix vector multiplications (SpMV) [32]. Giles et al. applied the tiling technique to an unstructured mesh CFD code that is particularly used for turbomachinery design [33]. In this work, we design a general-purpose cube-based data-centric algorithm to solve different fluid-structure interaction problems.

VIII. Conclusion and Future Work

As the number of cores per CPU increases, the shared resources per core (e.g., last-level cache, memory size, memory bandwidth, I/O bandwidth) become less and less. To design new parallel software that can scale on manycore computer systems, we must increase data locality and the degree of parallelism, and load balancing to achieve high performance. The parallel LBM-IB library presented in this paper targets an important domain of fluid-structure interaction problems. Our sequential program provides nine computational kernels and detailed performance analysis. Although our OpenMP implementation scales well on a small number of cores, its parallel efficiency drops quickly on a large number of cores. To improve locality and increase the memory access efficiency on deep NUMA memory hierarchies, we conceive a new data-centric cube-based parallel algorithm to realize the LBM-IB method. The outcome of this work is a parallel implementation that can deliver up to 53% better performance than the OpenMP implementation on many cores. While our work was focused on the LBM-IB method that operates on both 3D fluid grids and 2D fiber arrays, the same cube-centric techniques and algorithms can also be applied to other types of mesh- or grid-based parallel applications.

Our immediate future work along this line is to extend the cube-based implementation from shared memory manycore systems to extreme-scale distributed memory manycore systems. Other future work will include overlapping different time steps, removing the global synchronizations by using dynamic task scheduling, and performing auto-tuning and code optimizations on individual computational kernels.

Acknowledgment

This material is based upon work supported by the Institute for Mathematical Modeling and Computational Science (IM^2CS) at IUPUI, by NSF China Grant No. 11172219, and by Purdue Research Foundation.

References

[1] L. Zhu, G. He, S. Wang, L. Miller, X. Zhang, Q. You, and S. Fang, "An immersed boundary method based on the lattice Boltzmann approach in three dimensions, with application," *Computers & Mathematics with Applications*, vol. 61, no. 12, pp. 3506–3518, 2011.

[2] D. Weihs, "Hydromechanics of fish schooling," 1973.

[3] M. Koehl, "The interaction of moving water and sessile organisms," *Sci. Am*, vol. 247, no. 6, pp. 124–134, 1982.

[4] J. Price, P. Patitucci, and Y. Fung, "Biomechanics: Mechanical properties of living tissues," 1981.

[5] T. J. Hughes, W. K. Liu, and T. K. Zimmermann, "Lagrangian-Eulerian finite element formulation for incompressible viscous flows," *Computer methods in applied mechanics and engineering*, vol. 29, no. 3, pp. 329–349, 1981.

[6] R. Glowinski, T. Pan, T. Hesla, D. Joseph, and J. Periaux, "A fictitious domain approach to the direct numerical simulation of incompressible viscous flow past moving rigid bodies: application to particulate flow," *Journal of Computational Physics*, vol. 169, no. 2, pp. 363–426, 2001.

[7] D. Sulsky, S.-J. Zhou, and H. L. Schreyer, "Application of a particle-in-cell method to solid mechanics," *Computer Physics Communications*, vol. 87, no. 1, pp. 236–252, 1995.

[8] R. Benzi, S. Succi, and M. Vergassola, "The lattice Boltzmann equation: theory and applications," *Physics Reports*, vol. 222, no. 3, pp. 145–197, 1992.

[9] S. Chen and G. D. Doolen, "Lattice Boltzmann method for fluid flows," *Annual review of fluid mechanics*, vol. 30, no. 1, pp. 329–364, 1998.

[10] OpenMP Profiler, "http://www.ompp-tool.com."

[11] PAPI Project, "http://icl.utk.edu/papi."

[12] C. S. Peskin, "Flow patterns around heart valves: a numerical method," *Journal of computational physics*, vol. 10, no. 2, pp. 252–271, 1972.

[13] ——, "The immersed boundary method," *Acta numerica*, vol. 11, pp. 479–517, 2002.

[14] OpenMP, "http://openmp.org."

[15] G.-H. Cottet and E. Maitre, "A level set method for fluid-structure interactions with immersed surfaces," *Mathematical models and methods in applied sciences*, vol. 16, no. 3, pp. 415–438, 2006.

[16] L. Shi, T.-W. Pan, and R. Glowinski, "Three-dimensional numerical simulation of red blood cell motion in poiseuille flows," *International Journal for Numerical Methods in Fluids*, vol. 76, no. 7, pp. 397–415, 2014.

[17] M.-C. Lai, "A hybrid immersed boundary and immersed interface method for two-phase electrohydrodynamic simulations," *Abstract of SIAM Annual Meeting 2014*, 2014.

[18] Z.-G. Feng and E. E. Michaelides, "The immersed boundary-lattice Boltzmann method for solving fluid–particles interaction problems," *Journal of Computational Physics*, vol. 195, no. 2, pp. 602–628, 2004.

[19] J. Wu and C. Shu, "An improved immersed boundary-lattice Boltzmann method for simulating three-dimensional incompressible flows," *Journal of Computational Physics*, vol. 229, no. 13, pp. 5022–5042, 2010.

[20] IBAMR, "https://github.com/IBAMR/IBAMR."

[21] B. E. Griffith, R. D. Hornung, D. M. McQueen, and C. S. Peskin, "Parallel and adaptive simulation of cardiac fluid dynamics," *Advanced computational infrastructures for parallel and distributed adaptive applications*, p. 105, 2010.

[22] E. Givelberg and K. Yelick, "Distributed immersed boundary simulation in Titanium," *SIAM Journal on Scientific Computing*, vol. 28, no. 4, pp. 1361–1378, 2006.

[23] J. Götz, K. Iglberger, C. Feichtinger, S. Donath, and U. Rüde, "Coupling multibody dynamics and computational fluid dynamics on 8192 processor cores," *Parallel Computing*, vol. 36, no. 2, pp. 142–151, 2010.

[24] P. Valero-Lara, "Accelerating solid–fluid interaction based on the immersed boundary method on multicore and GPU architectures," *The Journal of Supercomputing*, vol. 70, no. 2, pp. 799–815, 2014.

[25] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick, "Lattice Boltzmann simulation optimization on leading multicore platforms," in *IEEE International Symposium on Parallel and Distributed Processing (IPDPS), 2008.*, April 2008, pp. 1–14.

[26] W. Li, X. Wei, and A. Kaufman, "Implementing lattice Boltzmann computation on graphics hardware," *The Visual Computer*, vol. 19, no. 7-8, pp. 444–456, 2003.

[27] L. Peng, K.-i. Nomura, T. Oyakawa, R. K. Kalia, A. Nakano, and P. Vashishta, "Parallel lattice Boltzmann flow simulation on emerging multi-core platforms," in *Euro-Par 2008–Parallel Processing*. Springer, 2008, pp. 763–777.

[28] J. Tölke, "Implementation of a lattice Boltzmann kernel using the compute unified device architecture developed by NVIDIA," *Computing and Visualization in Science*, vol. 13, no. 1, pp. 29–39, 2010.

[29] N. Park, B. Hong, and V. K. Prasanna, "Tiling, block data layout, and memory hierarchy performance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 7, pp. 640–654, 2003.

[30] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. 35, no. 1, pp. 38–53, 2009.

[31] ——, "Parallel tiled QR factorization for multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 13, pp. 1573–1590, 2008.

[32] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix–vector multiplication on emerging multicore platforms," *Parallel Computing*, vol. 35, no. 3, pp. 178–194, 2009.

[33] M. B. Giles, G. R. Mudalige, C. Bertolli, P. H. Kelly, E. Laszlo, and I. Reguly, "An analytical study of loop tiling for a large-scale unstructured mesh application," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE, 2012, pp. 477–482.