CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2016; **28**:1678–1692
Published online 6 November 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3717

Parallelizing and optimizing large-scale 3D multi-phase flow simulations on the Tianhe-2 supercomputer

Dali Li[1], Chuanfu Xu[1,2,*,†], Yongxian Wang[1,2], Zhifang Song[1], Min Xiong[1], Xiang Gao[1] and Xiaogang Deng[3]

[1]*College of Computer, National University of Defense Technology, ChangSha 410073, China*
[2]*National Laboratory for Parallel and Distributed Processing, National University of Defense Technology, ChangSha 410073, China*
[3]*National University of Defense Technology, ChangSha 410073, China*

SUMMARY

The lattice Boltzmann method (LBM) is a widely used computational fluid dynamics method for flow problems with complex geometries and various boundary conditions. Large-scale LBM simulations with increasing resolution and extending temporal range require massive high-performance computing (HPC) resources, thus motivating us to port it onto modern many-core heterogeneous supercomputers like Tianhe-2. Although many-core accelerators such as graphics processing unit and Intel MIC have a dramatic advantage of floating-point performance and power efficiency over CPUs, they also pose a tough challenge to parallelize and optimize computational fluid dynamics codes on large-scale heterogeneous system.

In this paper, we parallelize and optimize the open source 3D multi-phase LBM code *openlbmflow* on the Intel Xeon Phi (MIC) accelerated Tianhe-2 supercomputer using a hybrid and heterogeneous MPI +OpenMP+Offload+single instruction, mulitple data (SIMD) programming model. With cache blocking and SIMD-friendly data structure transformation, we dramatically improve the SIMD and cache efficiency for the single-thread performance on both CPU and Phi, achieving a speedup of 7.9X and 8.8X, respectively, compared with the baseline code. To collaborate CPUs and Phi processors efficiently, we propose a load-balance scheme to distribute workloads among intra-node two CPUs and three Phi processors and use an asynchronous model to overlap the collaborative computation and communication as far as possible. The collaborative approach with two CPUs and three Phi processors improves the performance by around 3.2X compared with the CPU-only approach. Scalability tests show that *openlbmflow* can achieve a parallel efficiency of about 60% on 2048 nodes, with about 400K cores in total. To the best of our knowledge, this is the largest scale CPU-MIC collaborative LBM simulation for 3D multi-phase flow problems.

Received 11 August 2015; Revised 12 October 2015; Accepted 12 October 2015

KEY WORDS: heterogeneous system; intel xeon phi; Tianhe-2; multi-phase flow; LBM

1. INTRODUCTION

The lattice Boltzmann method (LBM) is an alternative to classical Navier–Stokes solvers for computational fluid dynamics (CFD) simulations [1]. From the microscopic perspective of view, LBM regards fluids as Newtonian fluids, divides flow field into small lattices (mass points), and simulates fluid evolution dynamics through collision models (lattices' collision and streaming). LBM can simulate a variety of sophisticated fluid problems, including multi-phase,

*Correspondence to: Chuanfu Xu, College of Computer, National University of Defense Technology, ChangSha 410073, China.
†E-mail: xuchuanfu@nudt.edu.cn

multi-component and thermal fluids [2]. Meanwhile, LBM can be easily applied to complex geometries with various boundary conditions as well. LBM codes have also been used in problems such as geofluidic flows, or magma flow through porous media, macro-scale solute transport, the dispersion of airborne contaminants in an urban environment, impact effects of tsunamis on near-shore infrastructure, melting of solids and resultant fluid flow in ambient air, and to determine permeability of materials [3].

Large-scale LBM simulations with increasing resolution and extending temporal range require massive high-performance computing resources. Thus, it is essential and practical to port LBM codes onto modern supercomputers, often featuring many-core accelerators/coprocessors (GPU[3], Intel MIC [4, 5], or specialized ones). These heterogeneous processors can dramatically enhance the overall performance of HPC systems with remarkably low total cost of ownership and power consumption. Although key kernels and procedures in LBM codes are naturally discrete, completely calculation independent, and ready for parallel processing, researchers often need to use various programming models/tools (e.g., Offload [4] for Intel MIC, OpenMP and MPI for intra-node/inter-node parallelization) on heterogeneous supercomputers, thus making the development and optimization of large-scale LBM applications exceptionally difficult. This is especially true when collaborating host CPUs and accelerators/coprocessors to maximize application performance. Taking Tianhe-2 (Milky Way-2), the present No. 1 supercomputer, as an example, each compute node contains 2 Xeon E5-2692 v2 CPUs and three Xeon Phi 31S1P coprocessors. The 2 CPUs and 3 MICs contribute 422 GFLOPS and 3 TFLOPS double-precision floating point performance, respectively. Our experience shows that the realistic performance of LBM codes on two Xeon E5-2692 CPUs is comparable with that of single Xeon Phi 31S1P, because the host CPU has advanced instruction issuing and scheduling mechanism with higher average memory bandwidth per core [6]. Thus, instead of using a CPU-only or MIC-only approach, it is necessary to perform CPU/MIC collaborative computing on Tianhe-2. However, to achieve efficient CPU/MIC collaboration, besides the aforementioned programming challenge, developers need to map a hierarchy of parallelism in problem domains to heterogeneous devices with different processing capabilities, memory availability, and communication latency. As for Tianhe-2, we must carefully balance the use of two CPUs and three Phi coprocessors, design mechanisms to effectively overlap and minimize the interaction costs between them, and finally scale LBM codes to large scale compute nodes.

Based on our previous experience of CFD codes on graphical processing unit (GPU)-accelerated supercomputers[6], in this paper, we parallelize and optimize a popular open source LBM code *openlbmflow* for large-scale 3D multi-phase flow simulations on the Tianhe-2 supercomputer. We tackle the challenge of heterogeneous CPU/MIC programming and collaboration for LBM codes. In particular, we make the following main contributions:

- With an in-depth analysis of both characteristics of LBM simulations and multi-core/many-core processors on Tianhe-2, we substantially enhance the cache efficiency and single instruction, mulitple data (SIMD) utilization of key LBM kernels in *openlbmflow* using cache-blocking and SIMD-friendly data layout transformation. As a result, we achieve a speedup of 7.9X and 8.8X on Xeon and Xeon Phi, respectively.
- On a single Tianhe-2 node, we collaborate CPU and Phi using offload and OpenMP. To do so, we propose a load-balance scheme to distribute workloads among CPUs and Phi processors and use an asynchronous model to overlap collaborative computation and communication on both devices. The intra-node CPU-MIC collaborative performance for *openlbmflow* is improved by 3.2X compared with the CPU-only approach.
- We further scale *openlbmflow* up on Tianhe-2 for massively parallel LBM simulations. Scalability tests show that the final MPI+OpenMP+Offload+SIMD parallelized code achieves a parallel efficiency of above 60% on 2048 Tianhe-2 nodes, with around 400K cores in total. To the best of our knowledge, this is the largest scale CPU-MIC collaborative LBM simulation for 3D multi-phase flow problems.

The remainder of the paper is organized as follows: Section 2 briefly describes LBM method and *openlbmflow* implementation. Section 3 presents the overall parallelization of *openlbmflow* on Tianhe-2. Section 4 presents hybrid MPI+OpenMP implementation and optimization for *openlbmflow*.

DOI: 10.1002/cpe

Section 5 presents the collaboration of CPU and Phi and scaling to large scale on Tianhe-2. Section 6 presents performance results. Finally, in Section 7**,** we introduce some related work and conclude in Section 8.

2. LBM METHOD AND *OPENLBMFLOW* IMPLEMENTATION

2.1. LBM method in openlbmflow

The *openlbmflow* program uses the popular D2Q9/D3Q19 (as shown in Figure 1) Lattice Boltzmann discretization model and Shan-Chen BGK single relaxation time collision model for multi-phase flows. It can simulate both 2D/3D single-phase or multi-phase flow problems with periodic and/or bounce-back boundary conditions. In this paper, we use 3D multi-phase flow problem as our simulation case. External-body-force feature enables *openlbmflow* to simulate gravity-like effects, and top/bottom-wall-speed feature is used for lid-driven flow. Further information about *openlbmflow* is available from its official website [7].

The particle distribution function of LBM obeys the Lattice Boltzmann equation (LBE). LBE performs in a micro-kinetics point of view and considers the macroscopic flow phenomenon as statistically averaged outcome of massive microscopic particles' movements. The particle collision in *openlbmflow* multi-phase flow is a simplified Shan-Chen BGK model. The particle distribution function $f(r,u,t)$ represents the density of particles at the moment t and spatial point r, whose speed is between u and $u+du$according to statistical kinetics. The evolution equation of f is

$$\frac{\partial f}{\partial t}+u\frac{\partial f}{\partial r}=\Omega(f)+F, \tag{1}$$

where F and $\Omega(f)$ stand for external force and collision term, respectively. In linear BGK collision model, $\Omega(f)$ satisfies the following formula:

$$\Omega(f)=-\frac{1}{\tau}(f-f^{eq}), \tag{2}$$

where f^{eq} is the equilibrium distribution function, and τ is the dimensionless relaxation time. And f is discretized as follows:

$$f_i(x+e_i,t+\Delta t)-f_i(x,t)=-\frac{1}{\tau}[f_i(x,t)-f_i^{eq}(x,t)], \tag{3}$$

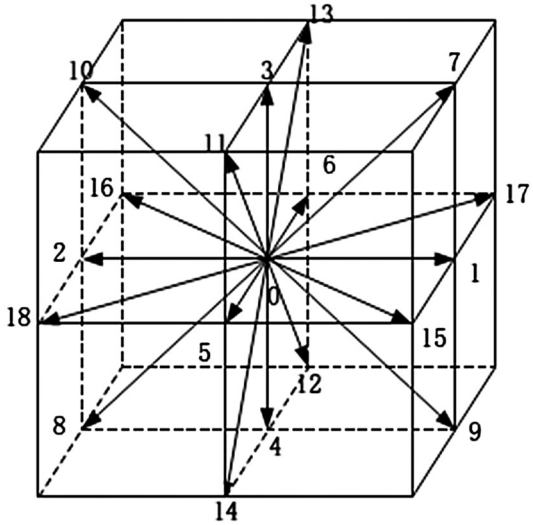

Figure 1. D3Q19 LBM discretization model.

DOI: 10.1002/cpe

where i represents the moving direction of a given particle, Δt is the time step, e_i is the unit length vector of each velocity direction of particle movements. Fluid density ρ and momentum ρu are described in the following formula, and the pressure $\mathbf{p}$ can be obtained through density.

$$\rho = \sum_i f_i = \sum_i f_i^{eq} \tag{4}$$

$$\rho u = \sum_i e_i f_i = \sum_i e_i f_i^{eq}, \tag{5}$$

the equilibrium distribution function f_i^{eq} and e_i are calculated as follows:

$$\begin{aligned} f_0^{eq} &= \frac{1}{3}\rho\left(1 - \frac{3}{2c^2}u^2\right) \\ f_i^{eq} &= \frac{1}{18}\rho\left[1 + \frac{3}{c^2}(e_i u) + \frac{9}{2c^4}(e_i u)^2 - \frac{3}{2c^2}u^2\right] \quad (i = 1, 2\cdots 6) \\ f_i^{eq} &= \frac{1}{36}\rho\left[1 + \frac{3}{c^2}(e_i u) + \frac{9}{2c^4}(e_i u)^2 - \frac{3}{2c^2}u^2\right] \quad (i = 7, 8\cdots 18), \end{aligned} \tag{6}$$

$$e_i = \begin{cases} (0,0,0) & i = 0 \\ (\pm 1,0,0)c, (0,\pm 1,0)c, (0,0,\pm 1)c & i = 1, 2\cdots 6 \\ (\pm 1,\pm 1,0)c, (\pm 1,0,\pm 1)c, (0,\pm 1,\pm 1)c & i = 7, 8\cdots 18 \end{cases}, \tag{7}$$

where $c = \frac{\Delta x}{\Delta t}$, Δx is the spatial grid stride, Δt is the temporal stride.

As for boundary conditions, periodic boundary lattices are treated as normal inner lattices, while bounce-back boundary lattices are updated after inner lattices and periodic boundary lattices during collision and streaming procedure. Figure 2 shows the bounce-back boundary mechanism in a 2D case. The gray boxes and white boxes are bounce-back boundary lattices and near boundary inner lattices, respectively; the arrows represent the distribution function components of each lattice. First, regular inner lattices' components are all updated when inner lattice collision and streaming are finished, as shown in Figure 2(a); meanwhile, the outward components (bold-black arrow) of bounce-back boundary lattices are also updated, but the back boundary components of near boundary lattices remain unchanged. Next, during the solid boundary lattice collision step in Figure 2(b), the outward components change their direction in a reflex manner, imitating the bounce-back effect of solid surface. At last the reversed components of bounce-back boundary lattices migrate to the corresponding components of near boundary lattices, see Figure 2(c).

2.2. *openlbmflow* implementation

The serial baseline LBM code *openlbmflow* is one of the most popular open-source LBM packages. We use the latest version (v1.0.1) in this work as of July 2015. Figure 3 shows the flowchart of *openlbmflow*. It mainly consists of three phases: initialization, time iteration, and post-processing. During initialization phase, the geometry of the flow field, flow density, and the distribution function are initialized. *Time* iteration phase includes three important procedures: inter-particular force calculation (as well as velocity and density), collision, and streaming. Because of the bounce-back boundary mechanism, procedures for boundary and inner lattices are implemented in separate kernels.

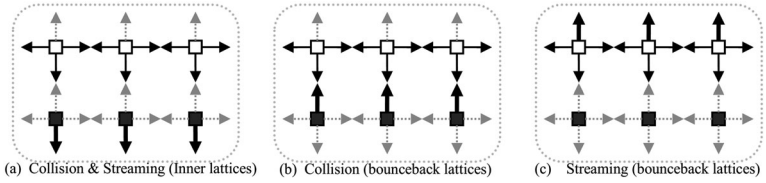

Figure 2. Example of 2D bounceback boundary.

DOI: 10.1002/cpe

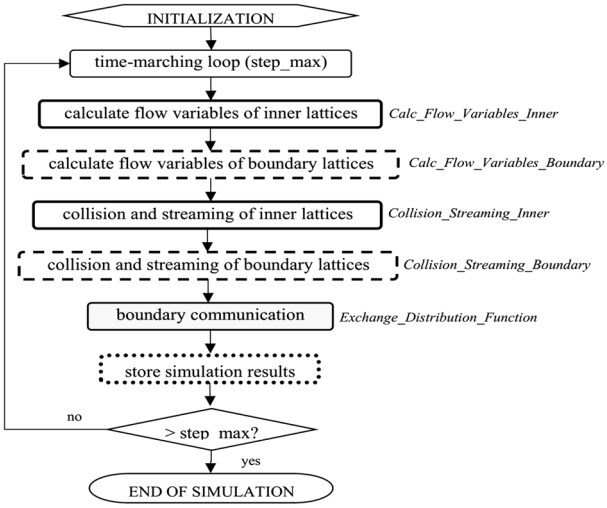

Figure 3. Flowchart of *openlbmflow*.

Procedures for collision and streaming are implemented together (*Collision_Streaming_Inner* for inner lattices and *Collision_Streaming_Boundary* for boundary lattices), and the equilibrium distribution calculation is also implemented in the collision code. Similarly, flow variables are updated for inter-particular force calculation in *Calc_Flow_Variables_Inner* and *Calc_Flow_Variables_Boundary*, respectively. In post-processing phase, simulation results are stored. *Collision_Streaming_Inner* and *Calc_Flow_Variables_Inner* for inner lattices are the most time-consuming kernels, and the time cost for boundary lattice kernels are negligible compared with inner-lattice kernels.

3. OVERALL HETEROGENEOUS AND COLLABORATIVE PARALLELIZATION

3.1. The Tianhe-2 supercomputer

Tianhe-2 (Milky Way-2) [8] is a heterogeneous supercomputer developed by National University of Defense Technology (NUDT), China. Tianhe-2 has been ranked No.1 since June, 2013. It consists of 16,000 compute nodes in total. As shown in Figure 4, each compute node has 2 Xeon E5-2692 v2 (ivy bridge) CPUs with 64 GB shared memory and 3 Xeon Phi 31S1P MIC coprocessors with 8 GB memory for each. All nodes are connected through self-developed THExpress-2 interconnection with a peak network bandwidth of 160 Gbps. Xeon Phi is connected to the host CPU using PCI-e 2.0 with a data transfer rate of 10 Gbps. Each Xeon CPU has 12 cores and the two CPUs deliver 422 GFLOPS peak performance in double precision. Each Xeon Phi coprocessor has up to 57 cores, with 4 hardware threads on each core, delivering 1 TFLOPS peak performance in double precision. Note that both processors have extended math unit (EMU) and vector processing unit (VPU) with long

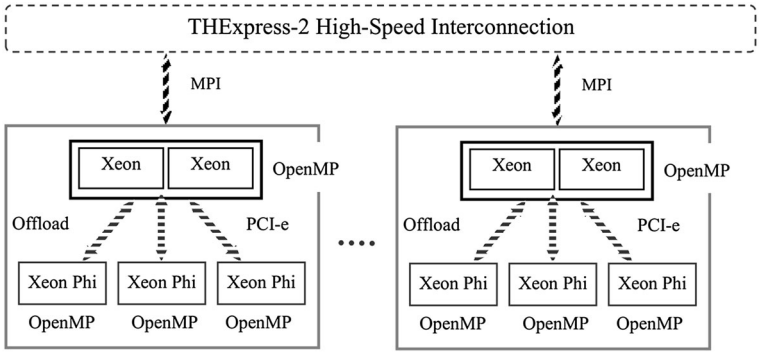

Figure 4. Overall System Architecture of Tianhe-2.

DOI: 10.1002/cpe

vector width. The vector length for Xeon CPU and Xeon Phi are 256 and 512 bit, respectively. Thus, it is fundamental to efficiently utilize the wide vector unit to tap the full potential of both processors.

Xeon has 3 cache levels: L1 cache (32KB for data and instruction, respectively) and L2 cache (256KB) are both core-private; L3 cache (30MB) is shared by all the 12 cores within a socket. Xeon Phi has 2 cache levels: L1 cache (32KB for data and 32KB for instruction) is core-private; the total capacity for L2 cache is 28.5MB and shared by all 57 cores using Core-Ring Interface (i.e., 512KB per core) with a bidirectional ring bus. The peak memory bandwidth of the 2 socket Xeon CPUs and a Xeon Phi within a node are about 204.8GB/s and 352GB/s, respectively.

We use the *roofline* model and operation intensity [9] to briefly analyze the performance characteristics of *openlbmflow* on Tianhe-2. According to the *roofline* model, if the operation intensity of a kernel is larger than the operation intensity of a processor, then the kernel is expected to be compute-bounded on the processor, otherwise it is most likely to be memory-bounded. With in-depth analysis, we find that the operational intensity of *Calc_Flow_Variables_Inner* and *Collision_Streaming_Inner* are 0.14 FLOP/Byte and 0.12 FLOP/Byte, respectively, while the operational intensity of Xeon is nearly 2.07 FLOP/Byte, and that of Xeon Phi is up to 2.84 FLOP/Byte. Therefore, *openlbmflow* is expected to be severely memory-bounded on both Xeon CPU and Xeon Phi coprocessor of Tianhe-2.

3.2. Overall parallel decomposition on Tianhe-2

Figure 5 shows the overall domain decomposition for MPI+OpenMP+Offload heterogeneous parallelization of *openlbmflow* on Tianhe-2. We decompose the original computational domain into many sub-blocks evenly and distribute them among MPI processes. For simplicity, we create one MPI process per node and assign one sub-block to it. We extend each sub-block with two layers of ghost lattices for data exchanging. On each node, we decompose the sub-block into four block chunks along X dimension by a load balancing factor (See Section 5 for more detail), and one block chunk is calculated by the 2 Xeon CPUs, and the other three chunks are offloaded to Xeon Phi coprocessors, with each coprocessor calculating one chunk. The load-balance scheme can successfully avoid frequent data movements between CPU and Phi, and only the boundary regions need to be updated after each time-step iteration.

The calculation of each lattice is completely independent; therefore, we use OpenMP to exploit the multi-core/many-core performance of both processors. On CPU the workload is parallelized in a piece-wise manner, the computation of block chunk is processed piece by piece along x dimension. While on Xeon Phi coprocessor, the workload is parallelized in a fine-grained line-wise manner to fulfill its massive amount of concurrent threads. To exploit fine-grained parallelism, we use the *collapse(2)* OpenMP clause to unfold the outer two layers of the loop region.

4. HYBRID MPI+OPENMP IMPLEMENTATION

4.1. Optimizing cache and SIMD efficiency

As described in Section 3, *openlbmflow* is severely memory-bounded on both Xeon and Xeon Phi coprocessor. Before performing heterogeneous parallelization and collaboration, we focus on improving the single core/thread performance of *openlbmflow* by optimizing Cache and SIMD efficiency.

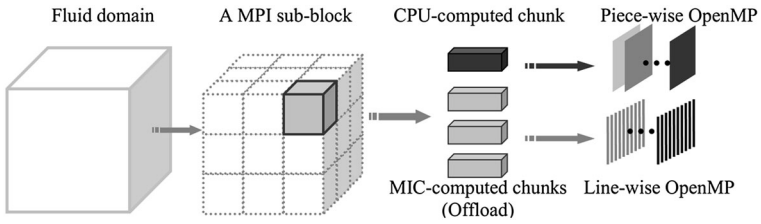

Figure 5. Overall domain decomposition for MPI+OpenMP+Offload parallelization of *openlbmflow* on Tianhe-2.

DOI: 10.1002/cpe

Data layout, that is, data structure, has significant impacts on SIMD performance and cache efficiency. In *openlbmflow*, it originally uses multi-dimensional pointers for flow variables. Thus, flow variables might be non-contiguous in memory resulting in poor data locality and further degrading cache efficiency. It will also cause extra vector gather/scatter operations when the SIMD feature is enabled and thus degrade the vectorization performance of Xeon and Xeon Phi. We reconstruct the initial multi-dimensional pointer using a plain one-dimensional pointer for flow variables. The SIMD-friendly data structure can achieve better data locality and higher cache efficiency and also dramatically reduces gather and scatter operations.

Furthermore, we employ 3D cache-blocking in memory-intensive kernels such as *Collision_Streaming_Inner* to enhance Cache efficiency. As shown in Figure 6, the original tri-loop is divided into many small chunks on each dimension (BX, BY, and BZ for X, Y, and Z dimension, respectively). By modifying BX, BY, and BZ, we can fit the small chunk into L3/L2 cache perfectly. As a result, the expensive memory access operations can be reduced significantly. In our tests, the optimal cache-blocking dimension size is 48 (BX = BY = BZ = 48) for *openlbmflow* program on both Xeon and Xeon Phi.

To take the full advantage of VPU, especially for the Xeon Phi coprocessor with 512-bit vectors, we enable the SIMD capability with mandatory SIMD directive (*!$DIR SIMD*) and eliminate the fake vector dependence between pointers of distribution function among two iterations using *!$DIR ivdep* directive.

4.2. Hybrid MPI/OpenMP parallelization

Before collaborating CPU and MIC, we implement a hybrid MPI/OpenMP parallelization on CPUs. In the MPI implementation, the original computational domain is decomposed for large amounts of nodes. For the test case in this work, we divide the cubic flow field domain with periodic or bounce-back boundaries in a progressive bisection manner, that is, dividing the domain along each coordinates orderly and repeatedly. Each sub-block is distributed to one Tianhe-2 node/ MPI process. As described in Section 2, a lattice requires communicating with its direct neighbors. Domain decomposition for parallelization will change the processing of original boundary conditions and introduce new ghost boundaries among sub-blocks. We expand sub-blocks with two layers of ghost lattices for data exchanges among neighboring sub-blocks. The reason of using two-layer ghost lattices rather than one layer is that the periodic boundary ghost lattices streaming procedure needs an extra layer of ghost lattices.

After each iteration, ghost lattices need to be updated. Figure 7 expresses the data exchanges of boundary and corner ghost lattices in 2D case. Figure 7(a) shows the data exchange of boundary ghost lattices. Source sub-block boundary lattices (denoted by solid squares) will exchange and update the ghost lattices of its destination sub-block (denoted by shadowed squires). Figure 7(b) shows the data exchange of domain corner lattices. Domain corner lattices (solid color squares) will exchange and update the ghost corner lattices (shadowed squares) that denoted by the same number. For instance, the left-bottom shadowed square (3) is updated by the right-top solid square (3). The corner lattices and corner ghost lattices data exchange relations among sub-blocks are shown in Figure 7(c).

```
#pragma omp parallel for
for(i = xs;i<xe;i++) //outer loop for thread parallel
for(j = ys;j<ye;j++) //middle loop for thread collapse
for(k = zs;k<ze;k++) // inner loop for SIMD
{
      fn = F(fp, phi,...); //collision and streaming
}
```

(a) Original Tri-loop

```
for(b_xs=xs;b_xs<xe;b_xs+=BX)
{   b_xe = min(b_xs+BX,xe);
    for(b_ys=ys;b_ys<ye;b_ys+=BY)
    {   b_ye = min(b_ys+BY,ye);
        for(b_zs=zs;b_zs<ze;b_zs+=BZ)
        { b_ze = min(b_zs+BZ,ze);
          #pragma omp parallel for
          for (x=b_xs; x<b_xe; x++)
          for (y=b_ys; y<b_ye; y++)
          for (z=b_zs; z<b_ze; z++)
          {
              fn = F(fp, phi,...);
}   }   } }
```

(b) Tri-loop with cache-blocking

Figure 6. Cache-blocking for tri-loop kernels.

DOI: 10.1002/cpe

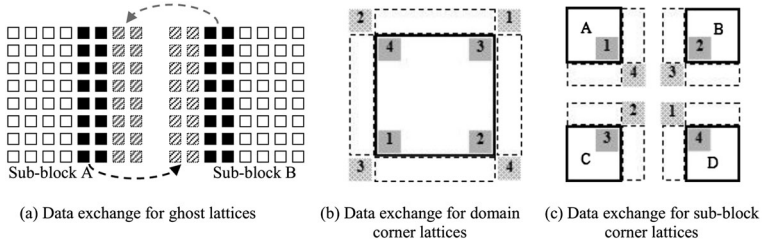

Figure 7. Data exchange after domain decomposition.

Within a node, shared-memory thread-level parallelization using OpenMP directives for time-consuming kernels is implemented on both Xeon CPU and Xeon Phi. As shown in Figure 8, OpenMP directives are added in the outer *for* loop. On Xeon CPU, the 24 cores/threads can be easily satisfied by the outer loop range (xs to xe). However, on Xeon Phi, the amount of concurrency threads (4 per core, up to 228 in total) is close to the outer loop range, so we have to make further efforts to exploit the parallelism in the middle loop, whereas the inner loop should be reserved for vectorization, to fully saturate Xeon Phi and avoid potential load imbalance problem. This can be achieved by using OpenMP *collapse* clause on Xeon Phi, as demonstrated in Figure 8(b).

5. COLLABORATING CPU AND XEON PHI

5.1. Intra-node collaborative programming

As described in 3.2, on each sub-block, one chunk is calculated by OpenMP threads on CPUs, and another three chunks are offloaded to the three Phi coprocessors. Figure 9 illustrates the intra-node

```
#pragma omp parallel for
for(i = xs;i<xe;i++) //outer loop for thread parallel
for(j = ys;j<ye;j++)
for(k = zs;k<ze;k++) // inner loop for SIMD
{
      fn = F(fp, phi,...) //collision and streaming
}
```

(a) OpenMP code on Xeon

```
#pragma omp parallel for collapse(2)
for(i = xs;i<xe;i++) // outer loop for thread parallel
for(j = ys;j<ye;j++) // middle loop for thread collapse
for(k = zs;k<ze;k++) // inner loop for SIMD
{
      fn = F(fp, phi,...) //collision and streaming
}
```

(b) OpenMP code with loop collapse on Xeon Phi

Figure 8. OpenMP implementation on Xeon and Xeon Phi.

```
// MIC calculation
for(d_id=0;d_id<MICNUM;d_id++){
   #pragma offload target (mic:d_id) in() signal(&sig[d_id])
//Offload asynchronously
   { .....
      #pragma omp parallel for collapse(2) //MIC threads
      Tri-loop-kernel;
      .....}
}; // end for & return immediately
// CPU calculation
#pragma omp parallel for    // CPU threads
Tri-loop-kernel;
.....
// MIC-CPU synchronization
for(d_id=0;d_id<MICNUM;d_id++){
   #pragma offload_wait target(d_id) wait(&sig[d_id])
   #pragma offload_transfer target(d_id) out()
}; //end for
```

Figure 9. Pseudo-code illustrating collaborative programming.

DOI: 10.1002/cpe

collaborative programming model. We use an asynchronous model to overlap CPU and MIC computation. Before starting calculations on CPU, we launch the Offload code on Phi, and it will execute asynchronously and return to the CPU code immediately. We perform synchronization to make sure that both devices have finished their computations. In this way, we overlap the computation on both sides.

The key issue for intra-node collaboration is to balance workloads among multiple CPUs and Xeon Phi coprocessors. We use a load balancing factor α to adjust workloads according to realistic performance of *openlbmflow* for different problem sets on both processors. Here, α is defined as follows:

$$\begin{aligned} \alpha &= \frac{P_{mic}}{P_{cpu} + N_{mic} * P_{mic}} \\ W_{cpu} &= W_{total} * (1 - N_{mic} * \alpha), \\ W_{mic} &= W_{total} * \alpha \end{aligned} \tag{8}$$

where W_{total}, W_{cpu}, and W_{mic} denote the total workload, CPU workload, and MIC workload, respectively. The number of Xeon Phi coprocessors is denoted as N_{mic}. The practical performance of CPU(P_{cpu}) and MIC(P_{mic}) can be obtained by sample running. The offload data transfer mechanism requires the transferred data to be in a continuous buffer, we divide the workload along the **x** dimension, that is, the outer layer of global array, eliminating additional memory copy operations for offload implementation.

The workload on each processor will keep constant during simulations to avoid frequent expensive data movements, and only boundary lattices need to be exchanged in each iteration. We collect the outer two-layer lattices of each block chunk into a temporal memory buffer for data transfers between CPU and Phi. This can dramatically reduce the quantity and frequency of PCI-e data transfers.

Figure 10 shows the collaborative CPU and multi-MIC computation and communication of synchronous and asynchronous Offload. Compared with the synchronous Offload, the asynchronous Offload can successfully overlap the computation of CPU and MICs and dramatically improve the intra-node performance. We also overlap the PCI-e data transfers between CPU and Phi with CPU end computation to further alleviate the intra-node communication costs.

We use a double buffering mechanism to update distribution function. Two duplicate distribution function arrays, *fp* for the present time step and *fn* for the next time step, need to exchange pointer addresses after each time-step iteration. The two pointers are associated to memory addresses on CPU and Xeon Phi during data transferring. Therefore, after exchanging pointer addresses, the pointers might be associated with wrong addresses; this is especially tricky when involving multiple Xeon Phi coprocessors. We use a delicate combined virtual pointer and solid pointer method by associating the pointers on Xeon Phi memory space with a solid pointer on CPU memory space for data exchanges and then only exchanging the addresses of virtual pointers for time iterations.

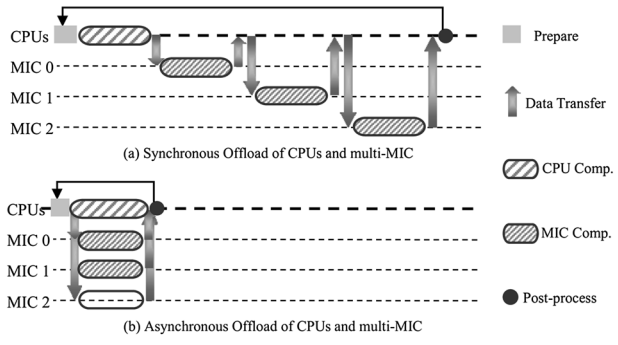

Figure 10. Overlapping the collaborative CPU-MIC computation and communication.

DOI: 10.1002/cpe

5.2. Scaling openlbmflow on large-scale Tianhe-2 nodes

To scale *openlbmflow* on large-scale Tianhe-2 nodes, we need to tackle the communication challenge among CPUs and Phi coprocessors on different nodes. The communication is especially complex for Tianhe-2 when utilizing multiple CPUs and Phi coprocessors on each node. Generally, running large-scale *openlbmflow* simulations on Tianhe-2 will involve the following communication patterns: (1) intra-processor communication within CPU/MIC, (2)intra-node communication of CPU-MIC and MIC-MIC, and (3)inter-node communication of CPU-CPU, CPU-MIC, and MIC-MIC. To simplify the MPI+OpenMP+Offload parallel implementation, we design a two-level hierarchical communication procedure and it contains only two explicit communications: the intra-node CPU-MIC communication (PCI-e data transfer) using offload directives and the inter-node CPU-CPU communication using MPI. Intra-processor communications are accomplished using shared OpenMP. Other types of communication can be implemented using the aforementioned two basic types of communications. Firstly, Xeon Phi processors transfer their updated boundaries to host CPU through the PCI-e channel. Then, all nodes update their ghost layers from neighbors by MPI communication. Finally, CPUs transfer the updated ghost layers to Xeon Phi processors.

6. PERFORMANCE RESULTS

6.1. Experimental setup and metrics

We use Intel *icc v11.1* for *openlbmflow* with the -O3 option. We use MPICH2-GLEX for MPI communication, and the Xeon Phi coprocessor shares the same compiler tool-chain with the Xeon CPU. In this work, we use the test case from *openlbmflow* website for performance tests. The case simulates the 3D drop on drop impact multi-phase (liquid and gases) problem with gravity effect. As shown in Figure 11, the fluid domain is a cubic space, the invisible background is filled with air, and there are two drops inside: one is on the floor, and the other one is in the air (Figure 11(a)). Because of the effect of gravity, the upper drop will fall down (Figure 11(b)) and fuses with the drop on the floor into one bigger drop (Figure 11(c)).

The problem size for single-core and node-level performance test is $512 \times 256 \times 256$. For large-scale tests, we extend the three dimensions of the fluid domain orderly. All the performance results are obtained using the average value of 10 effective sample simulations with 10-time iterations.

6.2. Single-core performance

In this section, we evaluate the performance optimization for *openlbmflow* on a single core of Xeon CPU and Xeon Phi. As shown in Figure 12, the 'OPT_Layout' denotes the optimization of LBM data layout using cache blocking and data structure transformation, and this will benefit both Cache and SIMD efficiency. Compared with the baseline code, we achieve a speedup of 3.0X and 3.4X on Xeon CPU and Xeon Phi, respectively. The performance is further improved by 1.7X on Xeon CPU and more significantly 2.6X on Xeon Phi when the SIMD feature is enabled (denoted as 'OPT_SIMD' in Figure 12). This is because the vector unit of Xeon Phi 31S1P is two times wider than that of Xeon E5-2692. As a result, the single-core performance for optimized LBM code on Xeon Phi is comparable with that of a Xeon CPU.

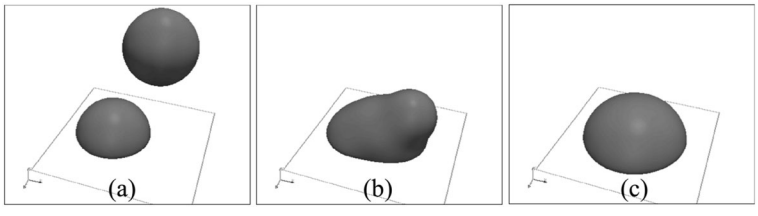

Figure 11. Test case - A 3D drop on drop impact multi-phase problem with gravity effect.

DOI: 10.1002/cpe

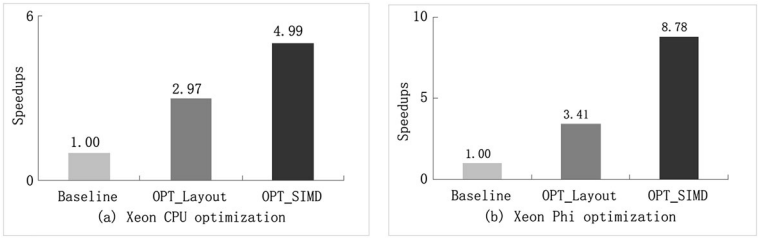

Figure 12. Performance optimization on a single core.

6.3. *Node-level performance*

In this section, we evaluate the shared-memory OpenMP performance for *openlbmflow* on both Xeon and Xeon Phi, as well as the collaborative performance on a single Tianhe-2 node. Figure 13 shows the percentage of execution time for the four kernels in each iteration on the Xeon CPU, with the number of OpenMP thread scaled to 24. *Collision_Streaming_Inner* is the most time-consuming kernel in *openlbmflow*, and it takes more than 60% execution time of the whole iteration. The percentage rises slightly with more than 8 threads, because of the increasing memory bandwidth requirement compared with other LBM kernels.

Figure 14 shows the speedup for the LBM code and the four kernels, respectively, on Xeon CPU with up to 24 OpenMP threads. Both *Calc_Flow_Variables_Inner* and *Collision_Streaming_Boundary* can achieve a maximum speedup of 11 using 16 OpenMP threads, demonstrating a good shared memory parallel scalability. And the *Calc_Flow_Variables_Boundary* kernel has a maximum speedup of 7.7 at 24 threads. However, limited by the memory bandwidth and its irregular memory footprint, the *Collision_Streaming_Inner* kernel achieves the maximum speedup of merely 6.5 at 8 threads and

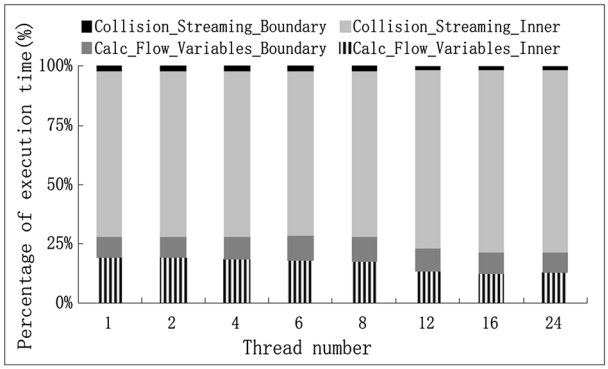

Figure 13. Percentages of different kernels on Xeon CPU.

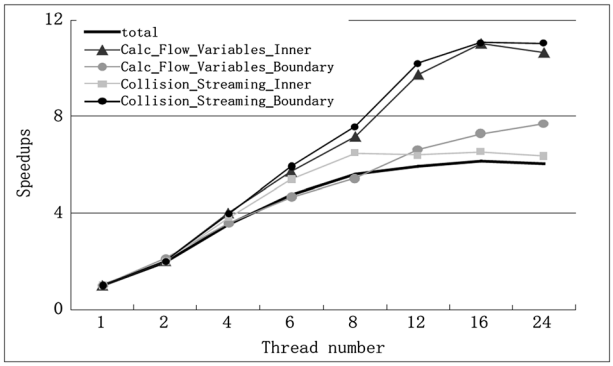

Figure 14. Speedup of different kernels on Xeon CPU.

DOI: 10.1002/cpe

decreases slightly afterwards. As a result, the speedup for the whole LBM code is about 6 with 24 threads. Besides these four primary kernels, there are some sequential sections such as initialize stage, and boundary data exchange stage. Limited by these sequential codes, the total speedup here is unsurprisingly lower than these four primary kernels.

We also evaluate the performance of Offload calculation on multiple Xeon Phi processors within a Tianhe-2 node. In the test, we use the performance on a single Xeon Phi as our baseline, and all the LBM calculations are offloaded to Xeon Phi. As shown in Figure 15, we achieve a speedup of 1.7 and 2.1 using two and three Xeon Phi processors, respectively. We further evaluate the collaborative speedup using both Xeon CPUs and Xeon Phis on a Tianhe-2 node. As shown in Figure 16, compared with the CPU-only approach, which merely utilizes both two CPUs, the collaborative approach can achieve a speedup of 1.9, 2.7, and 3.2 when we use one, two, and up to all three Xeon Phi processors, respectively. The results notably validate the effectiveness of our collaborative LBM simulation. The efficiency decreases of multi-MIC in previous tests are mainly caused by sequential sections of the program including initialize state, boundary data preparation stage before offloading, and boundary data harvest stage after asynchronous executing.

6.4. *Large-scale performance*

In the following large-scale parallel scalability tests, we use the performance on two nodes as our baseline, and the problem set is the same as before. Figure 17 shows the weak scalability and parallel efficiency of the MPI+OpenMP hybrid *openlbmflow* implementation using up to 4096 Tianhe-2 compute nodes. On each node, we create one MPI process and 24 OpenMP threads. As shown in Figure 17(a), although the MPI communication time changes a lot with the varying of node number, the computational time is always the dominating part of *openlbmflow* simulations. The communication time also keeps stable when we use more than 64 nodes and increases obviously after 512 nodes. The parallel efficiency of MPI+OpenMP approach is presented in Figure 17(b), the hybrid parallel version obtains a parallel efficiency of nearly 70% when scaling up to 4096 nodes, which demonstrates fairly good hybrid MPI+OpenMP parallel scalability.

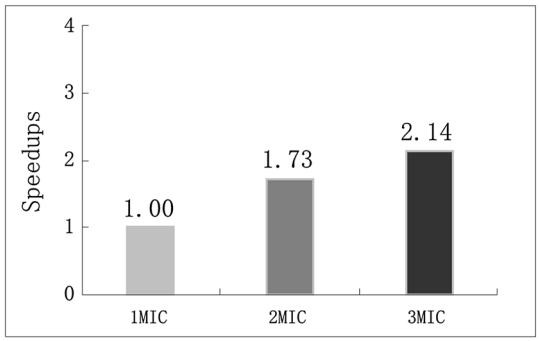

Figure 15. Intra-node multi-MIC speedups.

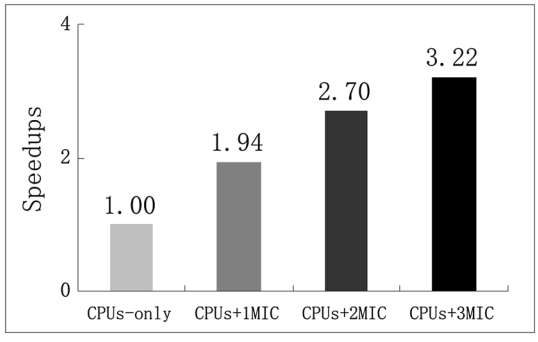

Figure 16. Intra-node CPU+MIC speedups.

DOI: 10.1002/cpe

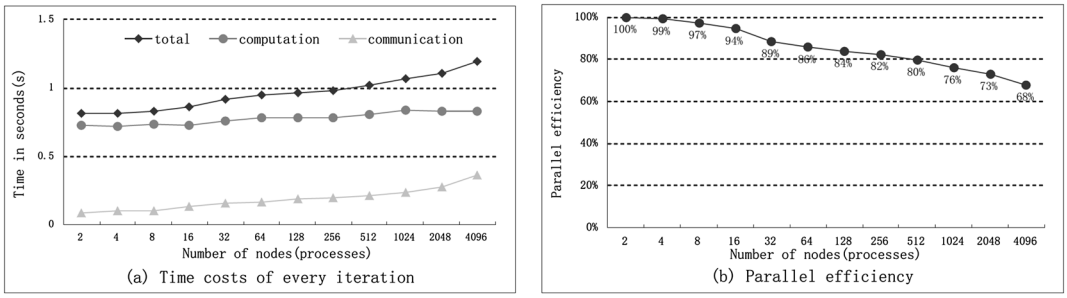

Figure 17. Weak scalability for MPI+OpenMP hybrid parallel implementation of *openlbmflow*.

Further, we present the weak scalability of MPI+OpenMP+Offload+SIMD *openlbmflow* on up to 2048 Tianhe-2 compute nodes, and each node utilizes all 2 CPUs and 3 MIC coprocessors. From Figure 18(a), we can see that the *openlbmflow* program has a good scalability at large scale. The computation performance remains steady with increasing node number, while the total simulation time increases remarkably with communication performance when compute nodes scale up. The communication time starts to overtake computation time and play a dominant role when scales up to 1024 and beyond. However, Figure 18(b) shows that the parallel efficiency of heterogeneous approach remains 60% with 2048 nodes, which demonstrates the effectiveness of overlapping collaborate computation and communication. Based on the obtained test results, we can infer that when the number of compute nodes scales beyond 2048 the parallel efficiency will surely decrease gradually but will still be acceptable for practical engineering applications.

7. RELATED WORK

Lattice Boltzmann method is considered to be very promising for large-scale computing. There have been many studies in parallelizing and optimizing LBM codes on HPC systems. Williams S. *et al.* [10] presented an auto-tuning approach to optimize LBM on multi-core architectures including Intel, AMD as well as Sun CPU processors. They developed a code generator to identify the specific CPU platform and generate highly optimized LBM program, with a speedup of over 15 times. Liu Z. *et al.* [11] evaluated the performance of a MPI+OpenMP hybrid parallel LBM program performance on the 'Ziqiang 4000' cluster in Shanghai University. Mountrakis L. *et al.* [12] evaluated the MPI performance for the open-source LBM framework Palabos. Their results show excellent weak and strong scalability with 8192 CPU cores. Pananilath I. *et al.* [13] developed an automated code generator for LBM simulations, featuring optimization techniques of tiling, load balancing, SIMD, etc. to boost LBM codes' performance. Generally their optimization can outperform Palabos by 3 times on Intel Xeon Sandy-bridge CPU. To summarize, previous studies on multi-core CPUs show that LBM has excellent parallel potential and achieves fairly well performance scalability.

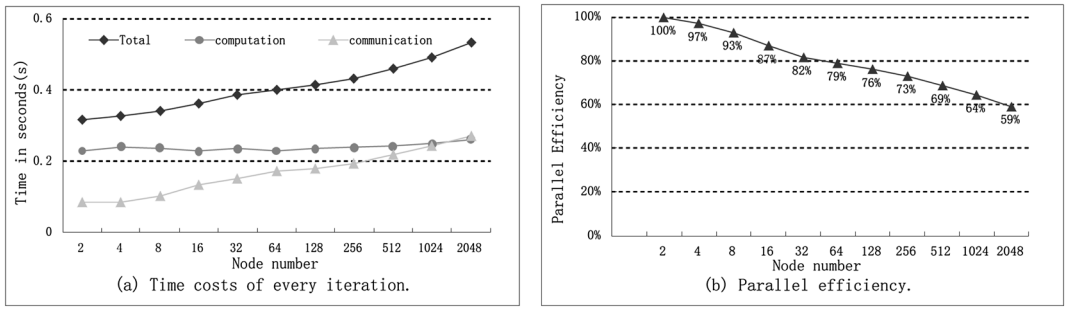

Figure 18. Weak scalability of MPI+OpenMP+Offload+SIMD heterogeneous parallel *openlbmflow*.

DOI: 10.1002/cpe

Much prior work has shown the experience of porting various LBM codes to many-core processors such as GPUs, with impressive speed-ups. Wei C. *et al.* [14] implemented an improved LBM program to utilize both CPUs and GPUs of Tianhe-1A, and they proposed a new load-balance scheme on the heterogeneous system. Kanoria A. A. *et al.* [15] implemented a parallel LBM program using MATLAB, and they evaluated the implementation three different GPU products (i.e., C2070, GTX 680, and K20). Feichtinger C. *et al.* [16] implemented a LBM code on GPU-accelerated Tsubame 2.0 supercomputer. They proposed a performance model for the communication overhead and evaluated the LBM program performance on more than 1000 GPUs. Koda Y. *et al.* [17] developed a LBM code to simulate turbulent flow problems using large eddy simulations (LES) on GPUs. They achieved the speedup of over 150 using 2 GPUs. LBM codes achieve remarkable speedups using GPU accelerators, and researches about large scale CPU-GPU cluster are also presented and gain good parallel efficiency.

Intel MIC products were available until the year of 2013, there are very few related studies involving porting and optimizing scientific codes including LBM codes on MIC architecture, especially for large-scale applications. Crimi G. *et al.* [18] ported a GPU-accelerated 2D LBM code to Xeon Phi coprocessor and compared with previous GPU implementation and state-of-the-art CPUs by sustained performance and a defined ξ metric that provides a fair comparison of performances across architectures with widely different number of cores and vector sizes. MIC shows higher sustained performance but lower ξ parameter value. Rosales C. *et al.* [19] demonstrated the challenges such as tradeoff between performance and portability when porting and optimizing codes on Xeon Phi with several micro-benchmarks including a LBM code. Their work also shows that data transfers using Offload and MPI exchanges are a fundamental bottleneck for large-scale codes on HPC systems accelerated by Xeon Phi. McIntosh-Smith S. *et al.* [20] implemented a performance-portable OpenCL version of LBM program and evaluated its performance on modern processors including CPUs, NVIDIA GPUs as well as Intel Xeon Phi. Results show that the OpenCL LBM code achieves both good performance portability and competitive performance compared with the hand-tuned version on different architectures. Bortolotti G. *et al.* [21] evaluated the performance of a LBM code on both Xeon Phi (Knight) and NVIDIA K20X (Kepler) GPU. Specific optimizations on both architectures were stated in detail, and the optimized program on accelerators achieved 2 to 3 times speedups compared with Intel Sandy Bridges CPUs. Calore E. *et al.* [22] also demonstrated the tradeoff of portability and performance on different high-performance processors, including NVIDIA GPUs, Intel Xeon Phi, and Intel Ivy Bridge and Opteron CPUs, using a OpenCL LBM code. To the best of our knowledge, optimization study of LBM program on heterogeneous CPU and multi-MIC system at large scale has not been noted.

8. CONCLUSION AND FUTURE WORK

In this paper, with MPI+OpenMP+Offload+SIMD, we have parallelized and optimized an open-source LBM code *openlbmflow* on the Intel Xeon Phi-accelerated Tianhe-2 supercomputer. We have performed particular optimizations to boost achievable performance for D3Q19 LBM kernels on leading-edge multi-core and many-core architectures with wide vector units and complex Cache hierarchy. To achieve a greater speedup and fully tap the potential of Tianhe-2 compute node, we have collaborated CPU and Xeon Phi for *openlbmflow* instead of using a naive MIC-only approach. We have presented a flexible load-balance scheme to distribute the loads among intra-node CPUs and Phi processors and an asynchronous model to overlap the collaborative computation and communication as far as possible. With our method, we successfully simulate a 3D multi-phase flow problem on the Tianhe-2 supercomputer using 2048 compute nodes with more than 400K cores and obtain a parallel efficiency of 60%.

With the increasing popularity of Xeon Phi in HPC systems, it is practical and essential to port and optimize real-world complex LBM codes on the many-core architecture. The approach and implementation of CPU-MIC collaboration as well as the balancing scheme presented in this work provide a fairly general experience of similar efforts for LBM codes on heterogeneous supercomputers using Xeon Phi.

DOI: 10.1002/cpe

ACKNOWLEDGEMENTS

This paper was supported by the Basic Research Program of National University of Defense Technology under Grant No. ZDYYJCYJ20140101, the Open Research Program of China State Key Laboratory of Aerodynamics under Grant No. SKLA20140104, the IAPCM Application Research Program for High Performance Computing under Grant No. R2015-0402-01, and the National Science Foundation of China under Grant No. 11502296. We would like to thank Guangzhou Supercomputer Center for providing us with Tianhe-2 system for performance evaluation and giving us technical supports.

REFERENCES

1. Godenschwager C, Schornbaum F, Bauer M, Köstler H, Rüde U. A framework for hybrid parallel flow simulations with a trillion cells in complex geometries. In Proceedings of the *International Conference on High Performance Computing, Networking, Storage and Analysis* (p. 35). ACM, 2013.
2. Biferale L, Mantovani F, Pivanti M, Pozzati F, Sbragaglia M, Scagliarini A, … Tripiccione R. Optimization of multi-phase compressible lattice Boltzmann codes on massively parallel multi-core systems. *Procedia Computer Science* 2011; **4**: 994–1003.
3. Bailey P, Myre J, Walsh SD, Lilja DJ, Saar MO. Accelerating lattice Boltzmann fluid flow simulations using graphics processors. In *Parallel Processing*, 2009. ICPP'09. International Conference on (pp. 550–557). IEEE, 2009.
4. Fang J, Sips H, Zhang L, Xu C, Che Y, Varbanescu AL. Test-driving intel xeon phi. In Proceedings of the *5th ACM/SPEC International Conference on Performance Engineering* (pp. 137–148). ACM, 2014.
5. Fang J. Towards a Systematic Exploration of the Optimization Space for Many-Core Processors (*Doctoral dissertation,* TU Delft*,* Delft University of Technology), 2014.
6. Xu C, Deng X, Zhang L, Fang J, Wang G, Jiang Y, Cheng X. Collaborating CPU and GPU for large-scale high-order CFD simulations with complex grids on the TianHe-1A supercomputer. *Journal of Computational Physics* 2014; **278**:275–297.
7. Official website of *openlbmflow*. http://www.lbmflow.com. [10 August 2015]
8. Introduction of Tianhe-2 on Top500 website. http://www.top500.org/system/177999. [10 August 2015]
9. Williams S, Waterman A, Patterson D. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM* 2009; **52**(4):65–76.
10. Williams S, Carter J, Oliker L, Shalf J, Yelick K. Optimization of a lattice boltzmann computation on state-of-the-art multicore platforms. *Journal of Parallel and Distributed Computing* 2009; **69**(9):762–777.
11. Liu Z, Song A, Xu L, Feng W, Zhou L, Zhang W. A High Scalable Hybrid MPI/OpenMP Parallel Model of Multiple-relaxation-time Lattice Boltzmann Method*. *Journal of Computational Information Systems* 2014; **10**(20):10147–10157.
12. Mountrakis L, Lorenz E, Malaspinas O, Alowayyed S, Chopard B, Hoekstra AG. Parallel performance of an IB-LBM suspension simulation framework. *Journal of Computational Science* 2015; **9**:45–50.
13. Pananilath I, Acharya A, Vasista V, Bondhugula U. An Optimizing Code Generator for a Class of Lattice-Boltzmann Computations, 2014.
14. Wei C, Zhenghua W, Zongzhe L, Lu Y, Yongxian W. An improved LBM approach for heterogeneous GPU-CPU clusters. In *Bio]medical Engineering and Informatics (BMEI)*, 2011 4th International Conference on (Vol. 4, pp. 2095–2098). IEEE, 2011.
15. Kanoria AA, Damodaran M. Parallel Matlab implementation of the lattice boltzmann method on GPUs, 2014.
16. Feichtinger C, Habich J, Köstler H, Rüde U, Aoki T. Performance Modeling and Analysis of Heterogeneous Lattice Boltzmann Simulations on CPU-GPU Clusters. *Parallel Computing*, 2014.
17. Koda Y, Lien FS. The lattice Boltzmann method implemented on the GPU to simulate the turbulent flow over a square cylinder confined in a channel. *Flow, Turbulence and Combustion*, 2014, 1–18.
18. Crimi G, Mantovani F, Pivanti M, Schifano SF, Tripiccione R. Early experience on porting and running a lattice Boltzmann code on the Xeon-Phi co-processor. *Procedia Computer Science* 2013; **18**:551–560.
19. Rosales C. Porting to the intel Xeon phi: opportunities and challenges. In *Extreme Scaling Workshop (XSW)*, 2013 (pp. 1–7). IEEE, 2013.
20. McIntosh-Smith S, Curran D. Evaluation of a performance portable lattice Boltzmann code using OpenCL. In Proceedings of the *International Workshop on OpenCL 2013 &* 2014 (p. 2). ACM, 2014.
21. Bortolotti G, Caberletti M, Crimi G, Ferraro A, Giacomini F, Manzali M, … Zanella M. Computing on Knights and Kepler Architectures. In *Journal of Physics: Conference Series* 2014; **513**(5):52032–52038.
22. Calore E, Schifano SF, Tripiccione R. A portable OpenCL lattice Boltzmann code for multi-and many-core processor architectures. *Procedia Computer Science* 2014; **29**:40–49.

DOI: 10.1002/cpe