Computer Physics Communications 211 (2017) 31–40

Contents lists available at ScienceDirect

Computer Physics Communications

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

Kernel optimization for short-range molecular dynamics

Hu Changjun [a], Wang Xianmeng [a], Li Jianjiang [a,*], He Xinfu [b], Li Shigang [c], Feng Yangde [d], Yang Shaofeng [a], Bai He [a]

[a] University of Science and Technology Beijing, Beijing, China
[b] China Institute of Atomic Energy, Beijing, China
[c] State Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[d] Computer Network Information Center, Chinese Academy of Science, Beijing, China

ARTICLE INFO

Article history:
Received 19 February 2016
Received in revised form
25 May 2016
Accepted 2 July 2016
Available online 26 July 2016

Keywords:
Molecular Dynamics
OpenMP
SIMD
MIC

ABSTRACT

To optimize short-range force computations in Molecular Dynamics (MD) simulations, multi-threading and SIMD optimizations are presented in this paper. With respect to multi-threading optimization, a Partition-and-Separate-Calculation (PSC) method is designed to avoid write conflicts caused by using Newton's third law. Serial bottlenecks are eliminated with no additional memory usage. The method is implemented by using the OpenMP model. Furthermore, the PSC method is employed on Intel Xeon Phi coprocessors in both native and offload models. We also evaluate the performance of the PSC method under different thread affinities on the MIC architecture. In the SIMD execution, we explain the performance influence in the PSC method, considering the "*if-clause*" of the cutoff radius check. The experiment results show that our PSC method is relatively more efficient compared to some traditional methods. In double precision, our 256-bit SIMD implementation is about 3 times faster than the scalar version.

1. Introduction

The use of atomistic simulations to understand the microstructure of materials has a long history [1]. With respect to the radiation effect of steel, neutron radiation is capable of displacing atoms from their lattice sites, causing point defects [2]. The migration and clustering of point defects leads to microstructure evolution under service conditions. Molecular Dynamics (MD) simulation, a common numerical technique, simulates the simultaneous motion of atoms (molecules) under their mutual interactions [3]. It plays a significant role in the investigation of radiation effects because it provides a methodology for accurate microscopic modeling at the molecular scale. The investigation of the radiation effect of steel by means of MD simulations typically requires simulation times that are nanoseconds or longer, and thus a great deal of CPU time is necessary [4]. Therefore, the efficiency of MD packages is of great significance. In a typical application, more than 90% [5] of the runtime of MD simulations is consumed by calculating interaction forces. In this paper, we define the force computation component as the calculation kernel. The optimization work of the kernel improves statistical convergence [6] and makes it possible to study events that occur on a longer timescale.

Multi-threading accelerates programs. Single-Instruction-Multiple-Data (SIMD) execution is an effective way to increase peak performance. To reach exascale performance, computing nodes are supplied with accelerators such as GPUs and, recently, many-integrated-cores (MIC) coprocessors. All [7–9] of these methods have been exploited to speed-up MD simulations. Liu et al. [10] proposed a multi-step computation method to optimize long-range force computations in MD simulations. This method divides atoms into groups and then partitions the real-space Ewald summation into steps based on these groups. Inspired by Liu's [10,11] strategy, we put forward a PSC (Partition-and-Separate-Calculation) method. This method avoids write conflicts among threads when multi-threading is used to optimize short-range force computations. S.J. Pennycook et al. [12,13] explored SIMD for MD simulations on Xeon Phi coprocessors. We also utilize our PSC method on Xeon Phi coprocessors and make related improvements to achieve higher SIMD performance.

The main contributions of our work are as follows:

- We design a PSC method to optimize the force calculation kernel of large-scale MD simulations in a hybrid MPI-OpenMP

* Corresponding author.
E-mail address: lijianjiang@ustb.edu.cn (J. Li).

http://dx.doi.org/10.1016/j.cpc.2016.07.010
0010-4655/

scheme. Our method eliminates write conflicts caused by using Newton's third law in a simple and efficient way. Our method merely requires several implicit barriers. Therefore, unlike in the case of certain traditional solving strategies, synchronization costs, lock contentions, repeated computations and extra memory usage are lessened.
- We utilize the PSC method on Intel Xeon Phi coprocessors. We analyze the effect of thread affinity on PSC performance. In both native and offload models, we observed performance gains using up to 240 threads.
- We propose a modified pre-searching neighbors strategy to increase the meet ratio of the cut-off radius "*if* clause" in SIMD implementation. Using AVX and AVX2 to process double precision variables, we achieve a speedup that is approximately three times greater than that of the scalar version.
- We analyze and compare the performance of the original program with our accelerated version using Intel Vtune [14]. The experiment results show that our optimized version executes much faster than the original one. This means that in the same environment platform and limited time, our version is able to simulate a longer physical process.

The rest of this paper is organized as follows. Section 2 presents related work and background knowledge. Section 3 details the specific multi-threading and vectorization optimization work. Section 4 provides the results and evaluations of the experiments. Section 5 concludes the paper.

2. Related work and background

We review related work, including multi-threading, SIMD, and coprocessor optimizations of MD simulations in this section. The Embedded Atom Method (EAM) potential is introduced briefly because it is used as an example to explain our optimization strategies. Because our experiments are based on Crystal MD, we also provide information about Crystal MD.

2.1. Related work on optimizing MD simulations

2.1.1. Multi-threading for MD simulations

MD force calculation typically employs the nature symmetry of pair forces, (i.e., $f_{ij} = -f_{ji}$) by computing the forces of a pair of atoms only once and then adding f_{ij} and $-f_{ji}$ to atom i and atom j separately [4]. Although applying Newton's third law reduces the force calculation task by half, it will probably create write conflicts among threads. Different pair interactions may involve some common atoms (in Fig. 1), which possibly results in a write conflict in memory (i.e., force array) if Newton's third law is adopted.

Some solutions have been put forward to eliminate the write conflict caused by adapting Newton's third law. Among them, the simplest way is to abandon using Newton's third law. However, this method duplicates the force calculation [15,16]. The second solution is to make use of critical sections [4]: modifications of force arrays must be achieved in a critical section. The critical section in this method brings about a severe synchronization cost, and the bottleneck worsens with the increasing number of CPU cores. The third solution is creating thread private work areas to store partial forces. The private data are reduced later. However, the memory cost of private work areas is extraordinarily expensive when a large number of threads are used. To reduce the memory cost of private work areas, M. Kunaseth et al. [17] proposed data-privatization thread scheduling algorithms using nucleation-growth allocation. However, the extra memory usage still cannot be ignored.

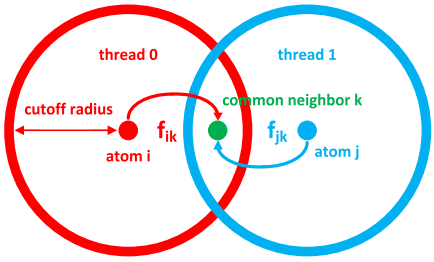

Fig. 1. The write conflict among threads caused by the application of Newton's third law.

Hu et al. [11] proposed a Spatial Decomposition Coloring (SDC) method. This method splits the spatial domain into subdomains and colors the subdomains with a set of different colors. A similar method is used to compute Ewald summation on multi-core platforms [10]. The SDC method is scalable but may cause load imbalances. Load imbalances are caused not only by Spatial Decomposition but also by the task allocation mechanism. The mechanism directly splits the domain and then assigns subdomains to each thread. In most instances, the number of subdomains cannot be divided by the number of threads, so a load imbalance among threads results. Another disadvantage of the task allocation mechanism is that the task load cannot adjust according to the number of threads. If the number of threads is too high, for example, more than 200 threads in MIC, the subdomain number may not be enough.

Our method is inspired by the SDC method [10,11], so it has the same advantages as the SDC method. We made modifications to the task allocation mechanism in SDC and implemented our method in a hybrid MPI-OpenMP schema. Our task allocation mechanism partitions each MPI execution area into equal sized slabs according to the thread number. Our method has a balanced load, and the amount of thread tasks in our method can adjust according to the thread number.

2.1.2. Intel Xeon Phi acceleration for MD simulations

The Xeon Phi [18] coprocessor is typically composed of about 60 cores clocked at 1 GHz or more. The Intel Xeon Phi supports three usage models [19] for interfacing the coprocessor. In the native model, programs are able to execute exclusively on the device; in the offload model, programs can transfer compute-intensive work to the device via compiler directives; and in the symmetric model, programs can treat the Xeon Phi as a standalone message-passing interface (MPI) node. In CoMD [15], the offload model is applied to target the force computation kernel. W. Michael Brown et al. [20] made modifications to the LAMMPS package to enable concurrent calculations on CPUs and coprocessors. The recently released GROMACS has supported the Intel Xeon Phi coprocessor in the native/symmetric model. It provides a 16-way [21] neighbor list for enabling 512-bit vector registers (KNC/KNL) as well as optimizations for native computations. Amber Molecular Dynamics [6] has observed a 2.83× speedup using the MIC coprocessor over the original host only version.

2.1.3. SIMD for MD simulations

Much effort has been devoted to achieving better performance in MD using SIMD technology. Gromacs [22–24] provided some SIMD acceleration options on the most compute-expensive parts. S.J. Pennycook et al. [12,13] explored SIMD utilization on Sandia's MiniMD benchmark using three SIMD widths (128-, 256- and 512-bit). Studies on improving [25] vectorization by loop-blocking techniques and vectorization pragmas have been conducted by the ExMatEx team [26].

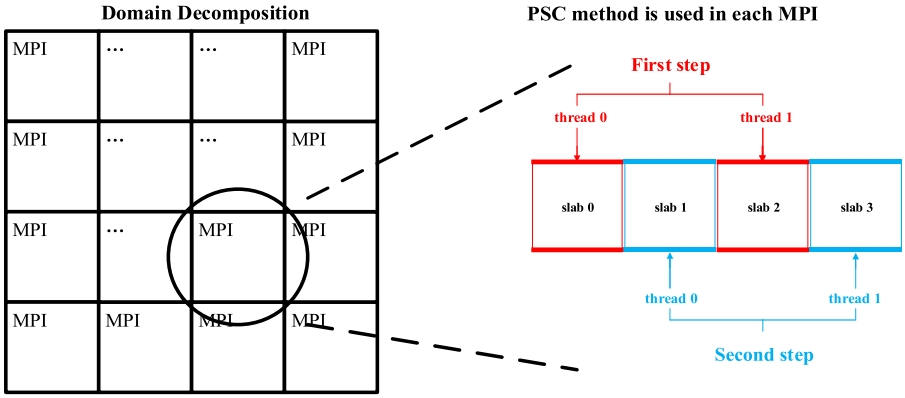

Fig. 2. Hybrid MPI-OpenMP scheme. Inside each MPI, the PSC method is used.

2.2. Background

Although our work is exemplified by optimizing EAM potential calculation, its capabilities are broadly applicable to other short-range potential computations. The EAM potential was introduced by Daw and Baskes [27] as a new means of studying ground-state properties of metal systems. The total potential is characterized as the addition of a many-body embedding energy term to a standard pair potential interaction [28,27]. The total potential is given by Eq. (1) [27].

$$E_{tot} = \sum_{i}^{n} e_i + \sum_{i}^{n} F(\rho_i) \tag{1}$$

where e is the pair potential. The term F is the embedding energy. The local electron density term ρ is a superposition of contributions f from neighboring atoms [29]

$$e_i = \frac{1}{2} \sum_{i \neq j} \Phi_{ij}(r_{ij}) \quad \text{and} \quad \rho_i = \sum_{i \neq j} f_{ij}(r_{ij}) \tag{2}$$

where the term r_{ij} represents the distance between atom i and atom j.

The implementation of our optimization work is based on Crystal MD [30]. Crystal MD is an MD package designed for metal with a BCC Structure. It puts forward a lattice neighbor list to calculate atoms' neighbor indexes according to their positions. It is indispensable to note that the optimization methods we have proposed in this paper can also apply to other MD simulations, although the basis of our experiment is Crystal MD. If this PSC method is not used for crystal structure material, load imbalances among threads may occur.

3. Kernel optimization of MD simulations

3.1. Efficient multi-threading implementation

Our PSC method is implemented in a hybrid MPI-OpenMP scheme. MPI processes executions in distributed memories, and OpenMP threads executions in shared memories. The entire domain is decomposed into several subdomains, and each MPI is responsible for a subdomain. As shown in Fig. 2, the PSC multi-threading method is used inside every MPI execution. In the MPI execution, the subdomain is partitioned into equally sized slabs according to the number of threads in the parallel region. Afterward, potential and corresponding forces are computed in two steps.

3.1.1. Principle of PSC method

The PSC method acts as follows:

(1) The OpenMP threads in parallel regions are set to M. The simulation area is partitioned into $2M$ slabs. Figs. 3 and 4 illustrate examples in which M is 2 in 2-Dimension and 3-Dimension situations. The simulation areas are partitioned into 4 slabs (slab 0 ∼ slab3) correspondingly.

(2) The $2M$ slabs are then divided into two groups. Each slab is separated from other slabs in the same group. Figs. 3 and 4 show that the 4 slabs are divided into a red group and a blue group. Slabs 0 and 2 are in the red group and are separated from each other. Slabs 1 and 3 are in the blue group.

(3) The interaction forces are calculated using M threads in two steps. In the first step, all the threads deal with the red group. Afterward, the same M threads continue to process the blue group. In our examples, thread 0 and thread 1 are assigned to compute slab 0 and 2 separately in the first step. Next, thread 0 and 1 turn to process slab 1 and slab 3.

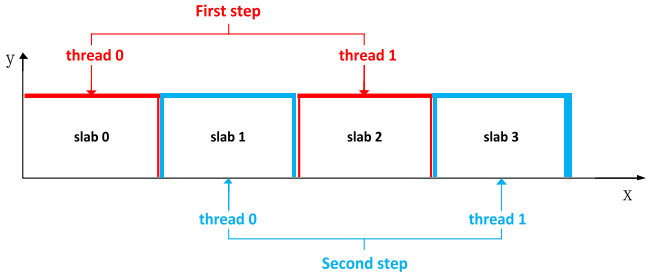

Fig. 3. Partition of the 2D simulation area into $2M$ slabs ($M = 2$). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

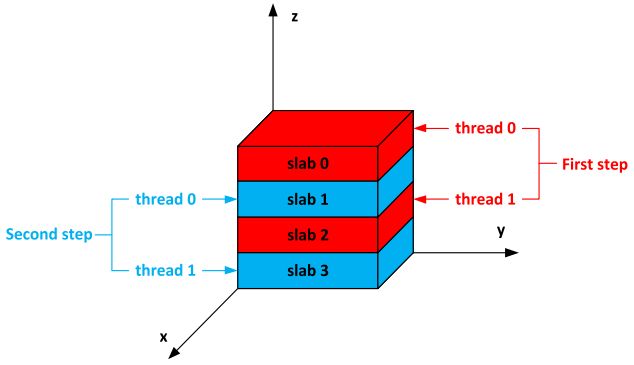

Fig. 4. Partition of the 3D simulation area into $2M$ slabs ($M = 2$). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

To avoid the write conflict caused by using Newton's third law, the distance between two slabs should be larger than the cutoff radius. This is the limitation of the PSC method, which will be discussed in detail in Section 3.1.3.

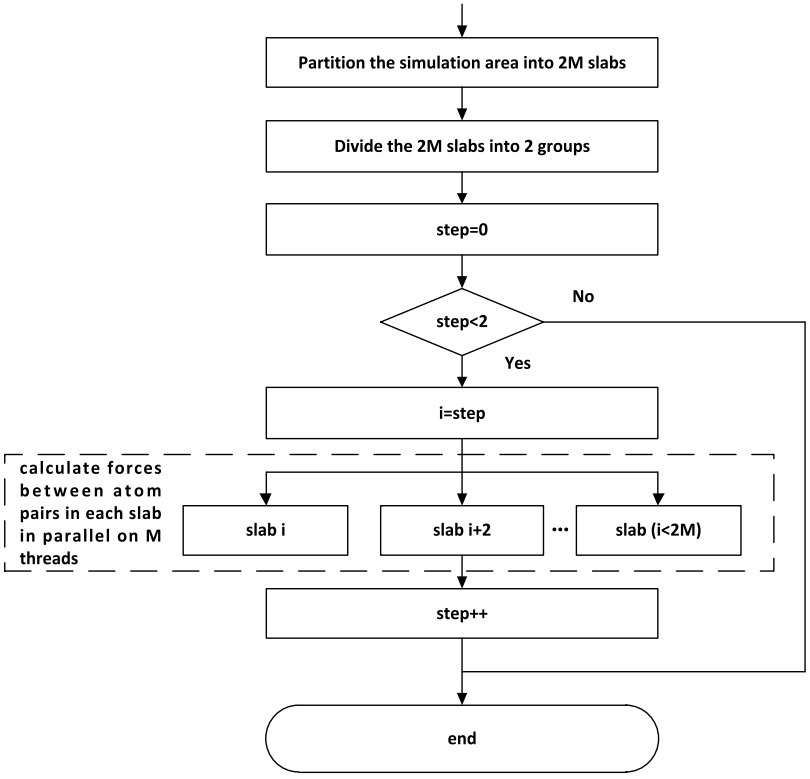

Fig. 5. A flowchart of the PSC method.

The workflow of the PSC method is illustrated in Fig. 5. In step 0, M threads process slabs (slab 0, slab 2…slab $2M-2$) that belong to the first group concurrently. In step 1, the M threads continue to deal with slabs (slab 1, slab 3…slab $2M-1$) that belong to the second group. We use the OpenMP [31] model to implement this method.

3.1.2. Implement the PSC method using OpenMP

Three loops are required in the classical computation of EAM potential and force. First, a loop is used to compute Φ (pair potential) and obtain ρ (electron density) from r (distance between atoms) using an interpolation function. Afterward, the second loop is used to obtain F (embedding energy) and its derivative. F (embedding energy) must then be communicated, as the force computation requires terms from adjacent simulation areas [27]. Finally, a loop computes the embedding energy contribution to the force and adds the result to the two-body force [26]. The approach in this case is to utilize the existing MPI framework and then further decompose the workload of each MPI task using OpenMP [32,31,33] multi-threading. Fig. 6 shows the partial pseudo of the PSC method using OpenMP.

3.1.3. Limitation of PSC method

To avoid write conflicts caused by using Newton's third law, the distance d, as shown in Fig. 7 in the 2-Dimension case, is supposed to be larger than the cutoff radius. Therefore, different threads are guaranteed not to write the same neighbor atom in the force array simultaneously.

Because our PSC method is designed for large-scale MD simulations, this restriction is overcome in most situations. A check is still supported in our program: if the distance d is greater than the cutoff radius, the PSC method is applied; otherwise, a traditional method is provided. Because only half of the neighbors are calculated, the distance d does not need to be larger than the $2 *$ cutoff radius.

3.2. Utilizing PSC on Xeon Phi

The Intel Xeon Phi is an x86-based many-core coprocessor based on Intel's Many Integrated Core (MIC) architecture [34–36]. Xeon Phi [18] is designed for massively parallel workloads, which offers a highly theoretical computational performance and memory bandwidth. These attributes of Xeon Phi [37] make it a commendable option for exascale computing.

3.2.1. Utilize PSC on Xeon Phi using a native model

The Intel Xeon Phi has a full-service Linux OS and possesses its own network interface, so it can act as an independent device. Programs can be solely executed on coprocessors without involving the host CPUs. The native model is a fast way to make existing programs execute with minimal code changes. In the native model, the MPI processes reside only inside the coprocessors. The application [18], MPI libraries, and other necessary libraries must be uploaded to the coprocessors.

To utilize the PSC method on the Xeon Phi coprocessor in the native model, it is not necessary to make modifications on the host version code. It is simply necessary to invoke the offload compiler with the "-mmic" flag. In addition, we copy the executable file, input file, and required libraries to the coprocessor. Then, the native executable file is ready to run directly on the MIC coprocessor. What calls for special attention is that when programs run on coprocessors, as many as 200 threads or more are typically used. Thus, the simulation scale should be large enough to meet the requirement mentioned in 3.1.3.

To illustrate the principle in a simple way, we only partition the simulation area from the z dimension. If the area is cut from three dimensions (x, y, and z), more steps and groups are needed, but the principle is the same as partitions from only the z dimension. Fig. 8 represents a partition of the simulation area from the y and z dimension, in which four colors and 4 steps are used.

```
thread_number = M ;                                   //set the thread number in parallel region to M
divide the simulation area into 2M slabs;

/*In step 0, M threads process the red slabs in parallel; In step 1, process blue slabs*/
#pragma omp parallel private(step)
For step from 0 to 2 by 1
    #pragma omp for
    For i from step to 2M by 2                                  //i is the slab index
        compute distance r using M threads;
        compute ρ from r using interpolation function in slab i using M threads;
    End for

    communicate ρ using 1 thread;

    #pragma omp for
    For i from step to 2M by 2
        compute F and its derivative using interpolation function in slab i using M threads;
    End for

    communicate the derivation of F using 1 thread;

    #pragma omp for
    For i from step to 2M by 2
        compute the Φ in slab i;
        calculate force using Φ and F in slab i using M thread;
    End for
End for
```

Fig. 6. Partial pseudo of the PSC method using OpenMP model.

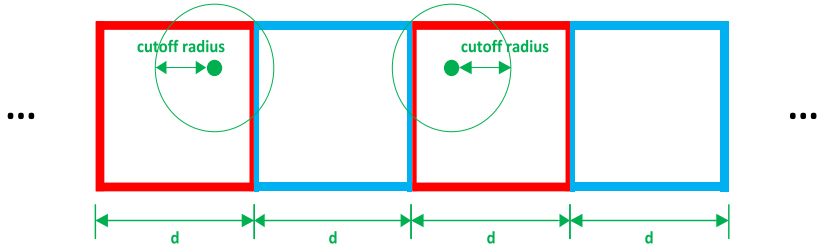

Fig. 7. If the distance d between two slabs (belonging to the same group) is greater than the cutoff radius, no common neighbor atoms will be operated simultaneously among different threads.

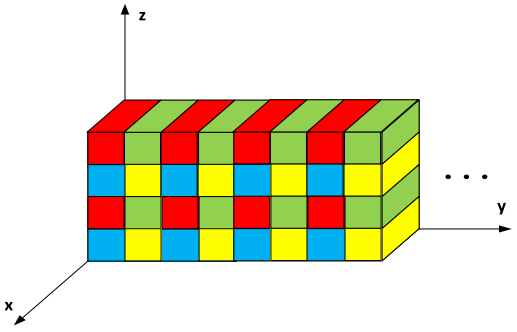

Fig. 8. Partition of the simulation area from the y and z dimensions [38].

Thread affinity [19,37,39] provides a general technique to improve performance on MIC architectures. It restricts execution [19] of certain threads (virtual execution units) to a subset of the physical processing units in multiprocessor computers. OpenMP [40,41] threads can be bound to physical processing units through the affinity environment variable. Depending on the operating systems [42], application and topology of the machine, thread affinity may have a dramatic effect on the application performance. The thread affinities investigated in our work are compact, scatter, and balanced. In the compact affinity mode, the maximum number of threads (which is 4 in Xeon Phi) is assigned to a core before being assigned to another core. This mode keeps the thread grouped tightly together, so it may benefit if there are exchanged data or shared data among threads. However, this mode may cause a load imbalance problem in some cases. In the scatter mode [42], threads are evenly distributed among the entire system in a round-robin way. Therefore, threads with the neighboring IDs [39] are not guaranteed to be physically adjacent. When there is no dependence among threads (especially neighboring threads), this mode can lead to good performance. In the balanced affinity mode, threads are also evenly distributed. This mode [39] makes full use of all available cores while keeping the thread-adjacent logical IDs physically close to each other. In most instances, the balanced mode can achieve good results. If the thread affinity is not set, it will be specified to none by default.

3.2.2. Utilize PSC on Xeon Phi using offload model

The offload model [36] is the standard solution for combining the capabilities of multi-core CPUs with highly parallel accelerators such as Intel MIC into a single application. Although CPUs [36] offer a small number of general purpose cores, accelerators have many specialized execution units, which can attain high floating-point performance. The offloading method is usually used to program the Xeon Phi because the majority of current applications [36] cannot fully utilize hundreds of hardware threads for all parts of the software but instead need the host CPU's massive single thread

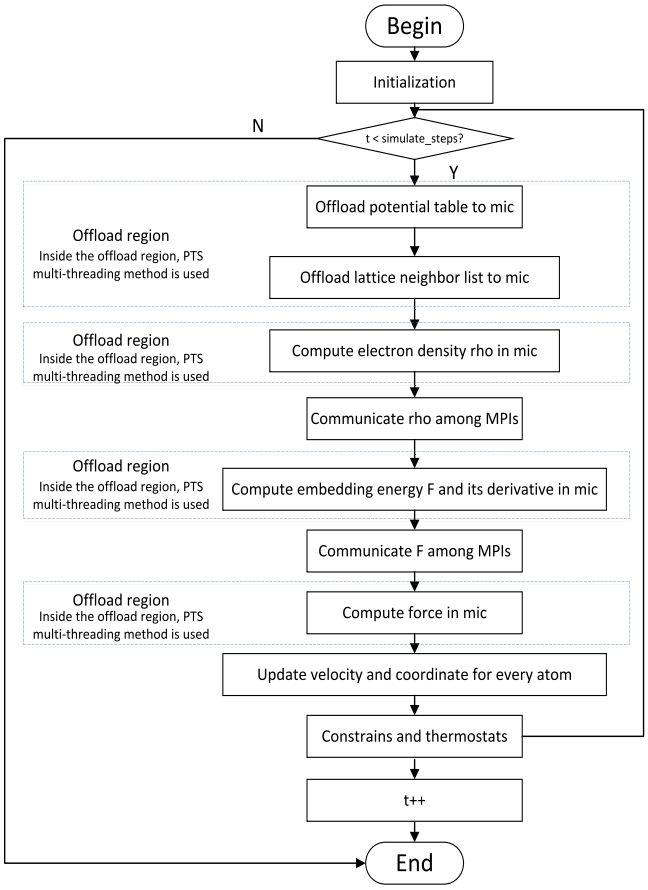

Fig. 9. Offload the EAM computation the kernel to mic.

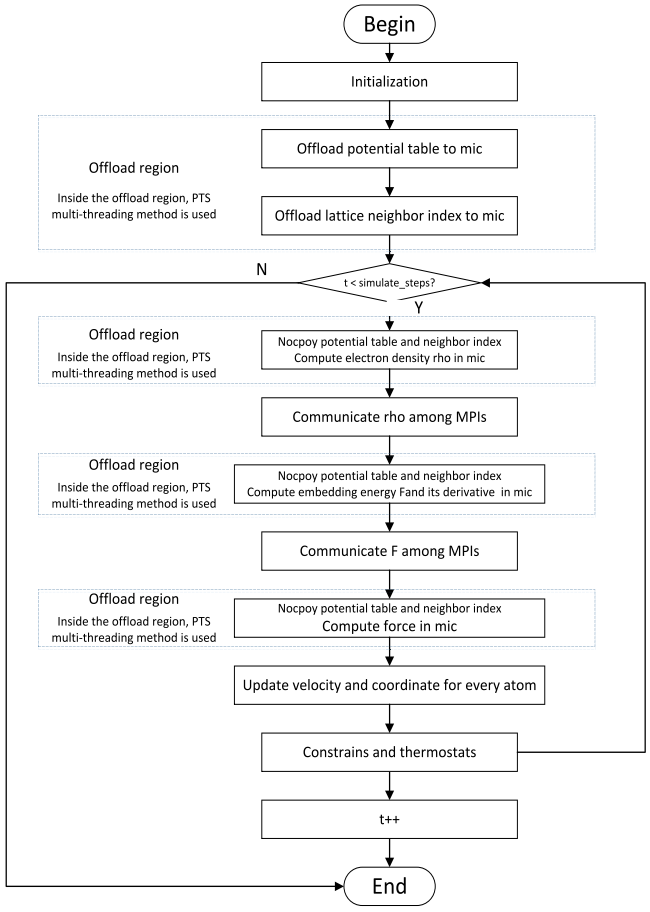

Fig. 10. Using nocopy to reduce time for transferring data to mic.

performance. The vendor recommended offload solutions for the Xeon Phi are Intel's Language Extensions for Offload (LEO) [40] and OpenMP 4.0 [43]. The non-shared memory model and virtual-shared memory model are supported in the offload compiler. The non-shared memory model is chosen in our implementation.

With the offload model [44], the highly parallel compute-intensive part and related data are transferred from the host to the coprocessor through the PCI Express. The offload can be initialed with pragmas and directives by the programmer. Excessive data-transfer time across coprocessors and host can reduce program scalability. Only when the offload computation time compensates the offload overhead cost will an overall improvement in performance be attained. The offload [36] cost includes the data/code transfer between host and coprocessors, synchronization and other forms of offload framework overhead.

Force calculation is the most compute-intensive part of molecular dynamics simulation, so the force kernel is offloaded to the coprocessor. The mechanism employed in this case was to utilize the existing MPI framework and offload the force computation to the coprocessor. Next, we decompose the task of each offloading MPI using the PSC method. Three loops are needed in EAM force computation, and there are MPI communications after every loop. Thus, in each iteration [37] of the EAM force computation kernel, data must be transferred between the host and Xeon Phi in three offload events. W. Michael Brown et al. [20] have offloaded the neighbor list build and short-range calculation to MIC. Because the lattice neighbor list remains the same through the entire simulation time in Crystal MD, offloading for the lattice neighbor list in every iteration is unnecessary. The offload flow chart is illustrated in Fig. 9.

The time involved in transferring data between the host and coprocessor causes significant overhead. Although some data must be updated every iteration, some data remain the same throughout the entire simulation. An excessive amount of time spent on transferring data from coprocessor to coprocessor slows down the execution. Transferring static data before the iteration begins could lead to certain improvements. After the initialization phase, we launch one offload region for instantiation and memory allocation. Potential tables and the lattice neighbor list are transferred to the Xeon Phi in this offload region. Inside the iteration, the coprocessor can utilize nocopy, alloc_if() and free_if() to reuse previously transferred data. The workflow for less transfer time is given in Fig. 10.

In this work, we use LEO directives to perform data allocation and offload computation on MIC. The program can be compiled [45] by any C++ compiler and can even be used on machines without coprocessors.

3.3. Improved SIMD exploitation

As discussed in Section 3.1.2, the EAM potential and force computation requires three loops. In consideration of the constrained space and the similarity of the three loops, we choose the ρ (electron density) calculation loop to introduce our optimization strategies.

3.3.1. Cutoff radius "if clause"

There is a cutoff distance check (indicated on line 7 in Fig. 10) to estimate if the distance r is shorter than the cutoff radius in the short-range potential computation. As to the neighbors whose distances between atom i are larger than the cutoff radius, their contributions of electron density ρ should be ignored, not

```
1.    for all atoms i do
2.      for all neighbors j do
3.        delx = xi - pos[j]+[0];
4.        dely = yi - pos[j]+[1];
5.        delz = zi - pos[j]+[2];
6.        rsq = (delx * delx) + (dely * dely) + (delz *delz);
7.          if (rsq <= Rc) then              //Rc=curoff*cutoff
8.           r=sqrt(rsq);
9.           ρ= Interpolate(r);
10.          rho_i+=ρ;
11.          rho[j]-= ρ;
12.         end if
13.     end for
14.   end for
```

Fig. 11. Computing electron density ρ using distance r.

vice versa. S.J. Pennycook et al. [12,46] addressed this issue by setting the adding value of neighbors that fail in the cutoff to zero via blending/masking. The resembling technique is used in Gromacs [22,23].

Nevertheless, these solutions bring about redundant calculation: both neighbors that satisfy and fail the cutoff check execute lines 8–11 in Fig. 11. The amount of redundant calculation depends on the proportion of neighbors that fail in the cut-off check. Pennycook et al. [12] used a neighbor list in miniMD and Gromacs [23] used Verlet pair-lists to control the ratio of failing atoms in the cut-off check. Therefore, the efficiency loss is tied to the updating frequency of the neighbor list. The more accurate the neighbor list is, the greater the amount of time spent on updating the neighbor list.

3.3.2. Modified pre-searching neighbor method

We reference the lattice neighbor list in Crystal MD [30], which finds neighbors according to their positions. Modifications have been made to the neighbor searching method to achieve a higher meet ratio of the cutoff radius check. Fig. 12 illustrates the principle of the lattice neighbor list: to find the neighbors of atoms i (colored red), a red rectangle whose width is N (calculated using Eq. (3)) times the lattice constant is used. In Eq. (3), ceil is used to return an integer that is greater than or equal to the value of the expression within the parentheses. The yellow atoms in this rectangle are considered neighbors of atom i. In most cases, this algorithm makes about 30% of the neighbors meet the cutoff radius check, which leads to a significant performance loss when SIMD is used.

$$N = \text{ceil}\,(\text{cutoff radius/lattice constant}). \tag{3}$$

The modifications of the lattice neighbor list are detailed as follows: pre-calculate the lattice position distance between atom i and the yellow atoms; if the distance is shorter than variable R (calculated using Eq. (4)), the yellow atoms are determined as a neighbor of atom i; otherwise, the yellow atoms are removed from the lattice neighbor list. In our application background, atoms generally do not derivate from their lattice position more than twice the lattice constant. Here, we use Eq. (4) to determine variable R, and variable R can be adjusted according to other actual applications. The pre-searching work only needs to be carried out once during the entire simulation process, so time consumption in this work can be ignored. After the modified pre-searching neighbor strategy is used, up to approximately 70% of neighbors can meet the cutoff check. For MD packages that use a neighbor list, to achieve the same meet ratio, a large amount of time is necessary to update the neighbor list.

$$R = \text{cutoff} + 2 * (\text{lattice constant}). \tag{4}$$

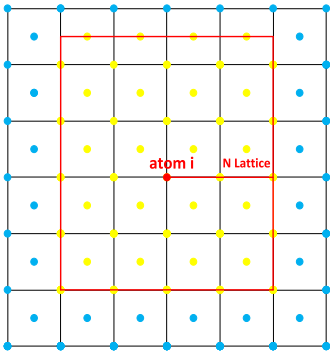

Fig. 12. Neighbor searching method in Crystal MD. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

The simulation variables, such as positions and forces, are double-precision in Crystal MD. We use 256-bit SIMD [46–49] for vectorization implementation, so we process four [13] neighbors simultaneously. The left part in Fig. 13 [12] illustrates the AVX and AVX2 implementation outline for ρ computation. 256-SIMD is used to process four neighbors simultaneously. The right part is a 512-SIMD implementation outline in which eight neighbors are operated at the same time.

4. Experiments results and analysis

Three test cases were used in our experiments: the small simulation case (549,250 atoms), the middle simulation case (1,024,000 atoms), and the large simulation case (8,192,000 atoms).

4.1. Experiment result and analysis of multi-threading optimization

In these experiments, we use the Red Hat 4.8.2-16 operating system and Intel(R) Xeon(R) CPU E7-8890 v3. Given that the EAM calculation kernel is the most compute-intensive part and is where we make the optimizations, the execution time in the experiment results indicates the EAM computation time.

The comparisons of the speedup of the PSC, RC, and CS methods are presented in Fig. 14. The RC in Fig. 14 refers to the redundant computation method used in LAMMPS (set the Newton command to *off* [50] in the input file, i.e., Newton's 3rd law is turned to *off* for pairwise interactions). The CS is the critical section method used in LAMMPS. It is clear that our PSC method has the best speedup compared with the other two methods over the three test cases.

Fig. 14 indicates that the CS method experiences bad speedup when the thread number is 16 and 20. The critical method reduces the global properties serially using a "critical" directive, so that only one thread at a time can access the global variables [50]. Thus, the speedups of the critical method are worse than those of our PSC method with an increasing number of threads. The redundant computation method (RC) exhibits better speedup than the CS method. Although the speedup is good, the execution time of the RC method is the longest of all the methods. Because the RC method duplicates the computation, it executes slower when simulating the same number of atoms.

Fig. 15 shows that our PSC method achieves a nearly linear speedup, which proves its satisfying scalability. In addition, the performance of our PSC method is improved with an increasing number of threads and atoms. The good performance of the PSC version benefits from several factors. First, as mentioned in Section 3.1, we partition the simulation area into equal sized slabs, which guarantees load balancing among multiple OpenMP threads. Second, as critical sections are not used in our PSC method, there

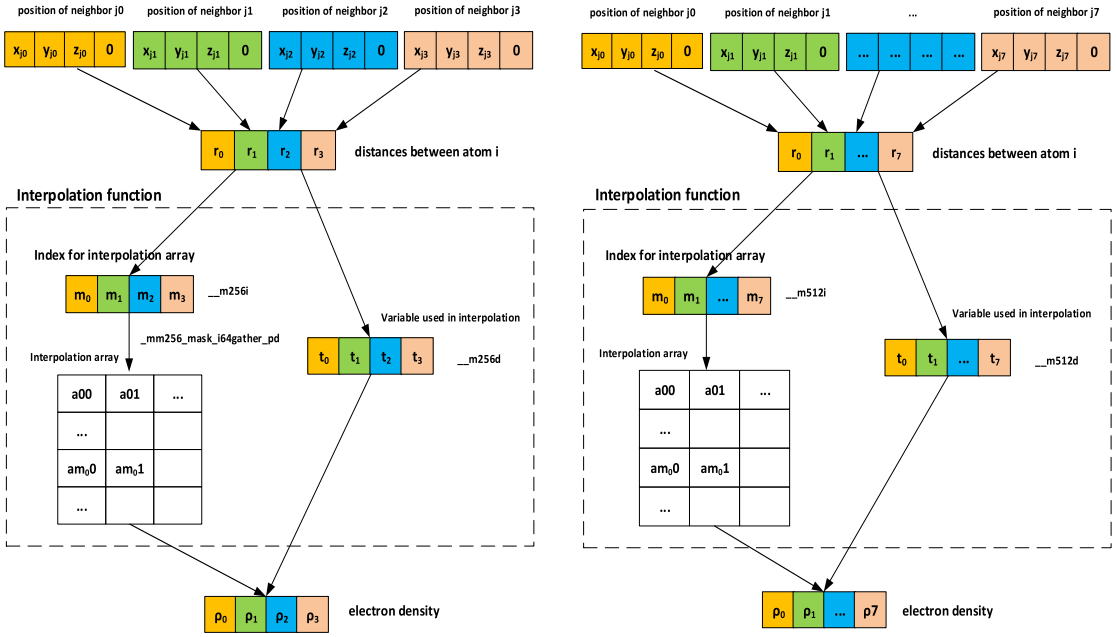

Fig. 13. SIMD implementation outline for calculating ρ.

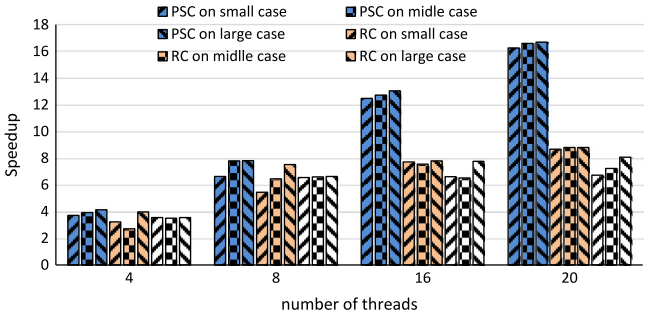

Fig. 14. Comparison of the PSC, RC and CS methods.

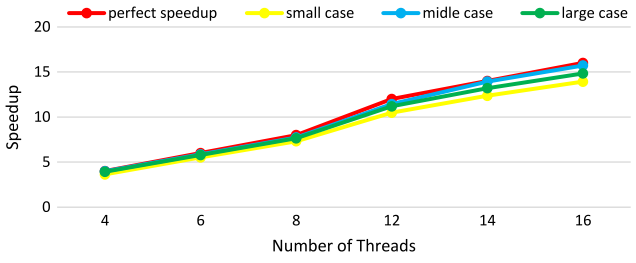

Fig. 15. Speedup of the PSC method.

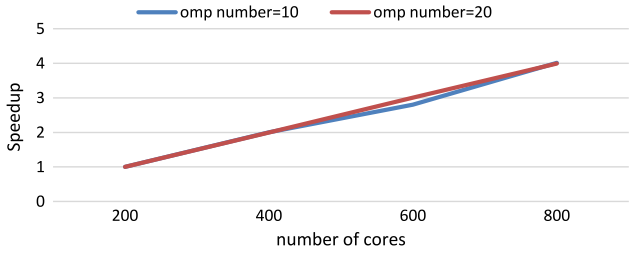

Fig. 16. Speedup of the PSC method simulating 4,000,000,000 atoms.

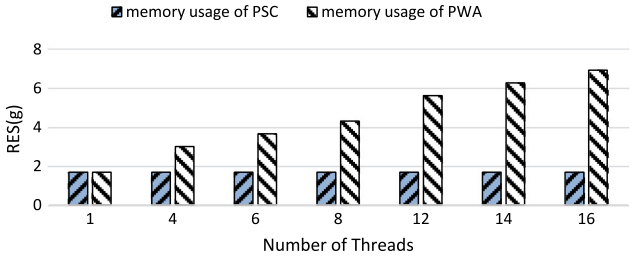

Fig. 17. Speedup of the PSC method in a larger scale.

are no severe serial bottlenecks. The only cost of the PSC method is several implicit barriers in every simulation time step. To avoid conflicts between two groups, the second group must be processed after the first one is completely finished.

To test the performance of the PSC method on a very large scale, we used a 4 billion-simulation case (4,000,000,000 atoms). The execution time for 200 cores is taken as the speedup basis. The PSC method shows a perfect speedup from 200 cores to 800 cores. The number of cores in the x-coordinate in Fig. 16 equals the MPI number $*$ OpenMP number in each MPI. In our experiment machine, there are 10 cores in a processor. When the OpenMP number is set to 20, the 20 threads are not in the same processor. As a consequence, the performance when the thread number is 20 is slightly worse than the performance when the thread number is 10.

To avoid write conflicts among threads, the private work area (PWA) method replicates the entire write-shared array and allocates a private copy to each OpenMP thread. The extra memory usage for the PWA method scales as $\Theta(np)$ [17], in which n is the number of simulation atoms and p is the number of threads. We made an estimate on the redundant memory allocation of the PWA method in CrystalMD. There are three write-share data structures in EAM computation: force f (3 $*$ double), electron density term ρ(1 $*$ double), and the derivative of embedding energy df (1 $*$ double). Fig. 17 estimates the memory usage of the PWA method in the large test case (8,192,000 atoms). The memory usages of the PSC method were obtained from experiments using the large test case. It is obvious that the memory usage of the PSC method remains almost the same among different numbers of threads, and it is much less than the PWA method.

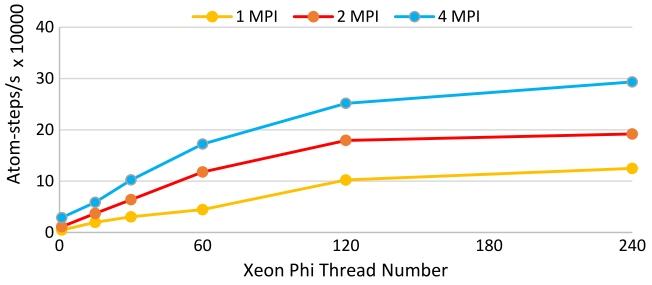

Fig. 18. Atom-steps/s for different coprocessor thread counts and host MPI in native mode.

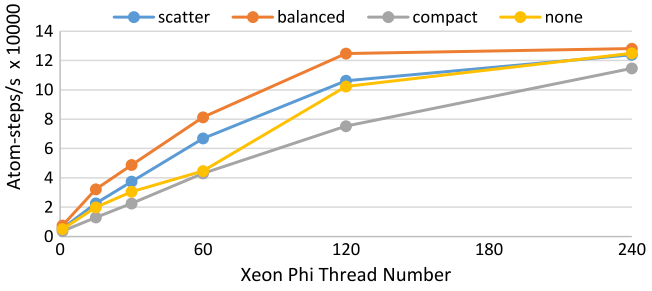

Fig. 19. Atom-steps/s for different thread affinities and coprocessor thread counts in native mode.

4.2. Experiment result and analysis of PSC on Xeon Phi

The hardware and software platform for the MIC experiment is detailed as follows:

The host is an Intel Xeon CPU E5-2670 @2.60 GHz, and the OS is Linux 2.6.32. The Intel Xeon Phi coprocessor is 7120P. The coprocessor has 8 GB DRAM (GDDR5), 61 cores each at 1.09 GHz and 4-way hardware multi-threading. The coprocessor is equipped with uOS 2.6.38.8 and mpss3.2.3. The Xeon Phi thread number and MPI task division are varied in the experiment. In the experiment with n MPIs, each MPI takes charge of $1/n$ of the simulation material. The middle test case (1,024,000 atoms) is used in this section.

4.2.1. Native model experiment

The application is compiled using mpiicpc – mmic and executed using mpiexec.hydra – n 1/2/4. The results presented in Fig. 18 give the absolute performance (i.e., (atoms ∗ time steps)/execution time) for different coprocessor thread counts and host MPI in native mode. A higher number is better.

Fig. 19 illustrates the Atom-steps/s for different thread affinity and coprocessor thread counts in native mode. There are 61 cores on the coprocessor, and one core is responsible for OS processes. Although each core can support four hard-threads, resource contention in the core may delay the speedup [51]. Despite the resource contention, the best performance is achieved when the Xeon Phi thread number is 240. Multi-threading gains more than the synchronization and contention cost [45,51].

The balanced and scatter mode make the load perfectly balanced among threads, so the application in these two modes performs well. There is no data sharing among iterations of loops in the PSC method. Therefore, the compact affinity mode, which binds the consecutive threads close together, does not gain good performance. The MD application [42] has some parallel regions that did not utilize all the available OpenMP threads, so it is desirable to avoid binding multiple threads to the same core while leaving other cores unused. A thread normally runs [42] faster on a core where it is not competing for resources with other threads. Thus, when KMP_AFFINITY = none the performance is better than compact.

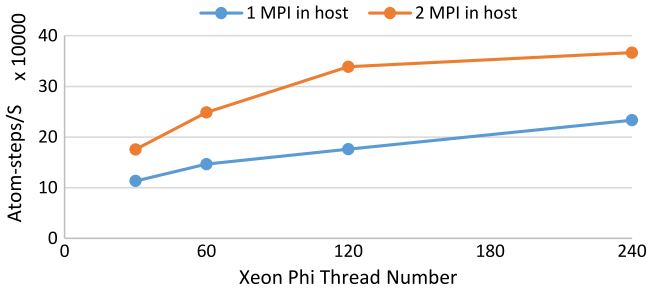

Fig. 20. Atom-step/s for different coprocessor thread counts and host MPI.

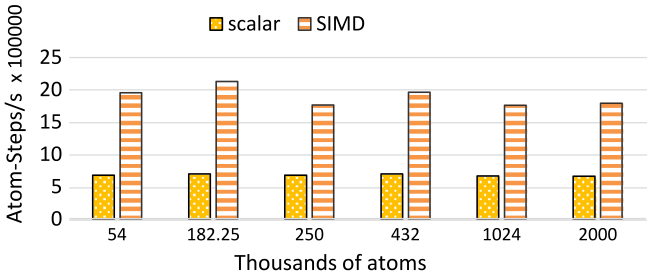

Fig. 21. Absolute performances of the original version and AVX version.

4.2.2. Offload model experiment

In this experiment, we use MPI at the cluster level, coupled with OpenMP and the offload model at the node level. Fig. 20 shows the absolute performance for different coprocessor thread counts and the host MPI.

Fig. 20 illustrates that the absolute performance is higher with an increasing Xeon Phi thread number. Moreover, the absolute performance reaches a peak when the thread number is the maximum 240. This experiment proves that the PSC method is also efficient using many threads.

4.3. Experiment result and analysis of improved SIMD implementation

Intel(R) Xeon(R) CPU E7-8890 v3 and Red Hat 4.8.2-16 operating system are used here. The simulation parameters were fixed as follows: the cutoff radius is 5.6, lattice constant is 2.855, and time steps is 30. We use absolute performance (i.e., (atoms ∗ time steps)/execution time) to compare the AVX and original versions. Because our optimization concentrates on the force calculation kernel, we only observe the execution time of EAM force calculation for both SIMD and the original versions.

Table 1 shows the execution time and speedups. Fig. 21 compares the absolute performances of our SIMD and original versions with different atom numbers. Fig. 21 also reveals that the Atom-Step/s of the scalar version remain almost the same across various problem sizes. Our optimized SIMD code is consistently about 3 times faster than the scalar version over different simulation scales.

5. Conclusions

We provide multi-threading and vectorization optimization of the MD force calculation kernel. Our optimization strategies accelerate the original MD version. Thus, in the same experimental platform and limited time, a longer physical process can be simulated using our optimized version.

We put forward a PSC method to avoid write conflicts when short-range force is calculated on shared-memory multi-core platforms. Our PSC method brings about neither extra memory usage, redundant computation, nor severe serial bottlenecks with increasing threads. Using both native and offload models, we utilize the PSC method on the MIC coprocessor.In the offload

Table 1
Execution time and speedup.

Thousands of atoms	54	182	250	432	1024	2000
Original execution time (s)	2.39	7.83	11.06	18.56	45.978	90.48
SIMD execution time (s)	0.83	2.57	4.26	6.61	17.48	33.53
Speedup	**2.88**	**3.05**	**2.60**	**2.81**	**2.63**	**2.70**

version, we offload the force calculation part to the coprocessor. We use nocopy and appropriate alloc_if as well as free_if() to reduce transfer time. Our experiment results demonstrate that the PSC method is scalable and efficient using up to 240 threads.

The cut-off radius "*if* clause" in short-range force calculation has a great influence on the MD package performance. We modify the lattice neighbor list in Crystal MD by adding a pre-searching procedure. The modified strategy leads to about 70% of atoms meeting the cutoff check, which decreases numerous redundant calculations. The optimized vectorization version is about 3 times faster than the scalar one.

Future research directions include auto tuning and data partitioning. Although EAM potential was used as an example, our optimization strategies are widely applicable for other short-range potentials. Both our multi-threading and vectorization optimizing methods are effective and straightforward to implement.

Acknowledgments

The research is partially supported by the Hi-Tech Research and Development Program (863) of China No. 2015AA01A303, Natural Science Foundation of China under Grant No. 61303050, and the Youth Innovation Promotion Association, CAS (2015375). National Natural Science Foundation of China under Grant No. 61502450.

References

[1] B.P. Uberuaga, L.J. Vernon, E. Martinez, et al., Sci. Rep. (2015) 5.
[2] H.E. Xinfu, P. Yang, Acta Metall. Sin. 47 (7) (2011) 954–957.
[3] D.C. Rapaport, The Art of Molecular Dynamics Simulation, second ed., Cambridge University Press, 2004.
[4] K.B. Tarmyshov, F. Müller-Plathe, J. Chem. Inf. Model. 45 (6) (2005) 1943–1952.
[5] zhangyue, Fundamentals of Computational Materials Science, Beihang University Press, 2007, p. 90.
[6] Ashraf Bhuiyan, Perri Needham, Ross C. Walker, Amber PME Molecular Dynamics Optimization - High Performance Parallelism Pearls - (Chapter 6).
[7] Jue Wang, Chun Liu, Yhuehui Huang, Future Gener. Comput. Syst. (2016).
[8] C. Hu, Y. Li, X. Cheng, et al., Future Gener. Comput. Syst. 54 (2016) 456–468.
[9] Jue Wang, Changjun Hu, Jilin Zhang, JianJiang Li, Future Gener. Comput. Syst. (4) (2010) Elsevier Publisher.
[10] Y. Liu, Comput. Phys. Comm. 182 (2011) 1111–1119.
[11] C. Hu, Y. Liu, J. Li, International Conference on Parallel Processing Workshops, IEEE Computer Society, 2009, pp. 121–129.
[12] S.J. Pennycook, C.J. Hughes, M. Smelyanskiy, S.A. Jarvis, Exploring SIMD for molecular dynamics, using intel xeon processors and intel xeon phi coprocessors, in: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, 2013, pp. 1085–1097.
[13] S.J. Pennycook, C.J. Hughes, M. Smelyanskiy, High Performance Parallelism Pearls, Elsevier Inc., 2015, (Chapter 8).
[14] Intel. http://software.intel.com/en-us/intel-vtune/.
[15] CoMD. http://www.exmatex.org/comd.html.
[16] L. Gary, S. Masha, Y.Z. Shen, Hardware-Software Co-Design for High Performance Computing, 2014.
[17] M. Kunaseth, D.F. Richards, J.N. Glosli, et al., J. Supercomput. 66 (1) (2013) 406–430.
[18] Intel Corporation, Intel Xeon Phi coprocessor system software developers' guide, 2013. [Online]. Available: http://www:intel:com/content/dam/www/public/us/en/documents/productbriefs/xeon-phi-coprocessor-system-software-developers-guide:pdf.
[19] Intel Developer Zone, Thread Affinity Interface (Linux and Windows), [Online]. Available: https://software.intel.com/en-us/node/522691.
[20] W.M. Brown, J.M.Y. Carrillo, N. Gavhane, et al., Comput. Phys. Comm. (2015).
[21] Intel. https://software.intel.com/en-us/articles/gromacs-for-intel-xeon-phi-coprocessor.
[22] GROMACS. http://www.gromacs.org/.
[23] E. Lindahl, B. Hess, D. van der Spoel, J. Mol. Model. 7 (2001) 306–317.
[24] B. Hess, C. Kutzner, D. van der Spoel, E. Lindahl, J. Chem. Theory Comput. 4 (3) (2008) 435–447.
[25] D. Mackay, Optimization and performance tuning for Intel Xeon Phi co-processors, part 1: Optimization essentials, 2012. [Online]. Available: https://software:intel:com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessorspart-1-optimization.
[26] ExMatEx. CoMD proxy application, 2012. http://exmatex.github.io/CoMD/.
[27] M.S. Daw, M.I. Baskes, Phys. Rev. B 29 (12) (1984) 6443–6453.
[28] S.M. Foiles, M.I. Baskes, M.S. Daw, Phys. Rev. B 33 (12) (1986) 7983.
[29] J.N. Glosli, K.J. Caspersen, Supercomputing (2007).
[30] He Bai, Changjun Hu, Xinfu He, Boyao Zhang, Jue Wang, Crystal MD:Molecular Dynamics Simulation Software for Metal with BCC Structure. Big Data Technology and Application 2015. Communications in Computer and Information Science. pp 247–258.
[31] OpenMP. http://www.openmp.org.
[32] Wang Xianmeng, Li Jianjiang, Wang Jue, et al., Big Data Technology and Applications, Springer, Singapore, 2015, pp. 269–281.
[33] Shigang Li, Changjun Hu, Implementation and optimization of OpenMP task parallelism on heterogeneous multi-core architecture, 2013.
[34] A. Heinecke, M. Klemm, H.J. Bungartz, Comput. Sci. Eng. 14 (2012) 78–83.
[35] James Jeffers, James Reinders, Intel Xeon Phi Coprocessor High Performance Programming, first ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2013.
[36] M. Noack, F. Wende, T. Steinke, et al., International Conference for High Performance Computing, Networking, Storage and Analysis, SC14, IEEE, 2014, pp. 203–214.
[37] G. Hager, G. Wellein, Introduction to High Performance Computing for Scientists and Engineers, CRC Press, 2011.
[38] Endong Wang, Qing Zhang, Bo Shen Guangyong Zhang, Xiaowei Lu, Qing Wu, Yajuan Wang, High-Performance Computing on the Intel® Xeon Phi™ how to Fully Exploit MIC Architectures, Springer, 2014.
[39] G. Lawson, M. Sosonkina, Y. Shen, 2014 International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW), IEEE, 2014, pp. 54–59.
[40] User and Reference Guide for the Intel C++ Compiler 14.0, Intel Corporation, 2014.
[41] R. Green, OpenMP thread affinity control, 2012. [Online]. Available: https://software:intel:com/en-us/articles/openmp-thread-affinity-control.
[42] Intel Developer Zone, Openmp thread affinity control. https://software.intel.com/en-us/articles/openmp-thread-affinity-control.
[43] OpenMP Application Program Interface, Version 4.0, OpenMP Architecture Review Board, July 2013.
[44] A. Das, D. Malav, S. Desai, S. Das, N. Kurkure, Analysis of Molecular Dynamics (MD_OPENMP) on Intel® Many Integrated Core Architecture. http://hpc-ua.org/hpc-ua-12/files/proceedings/1.pdf.
[45] James Reinder, An overview of Programming for Intel Xeon processor and Intel Xeon Phi coprocessors. by Intel Corporation, 2012.
[46] J. Reinders, J. Jeffers, High performance parallelism pearls, multicore and many-coreprogramming approaches, 2015.
[47] Intel Intrinsics Guide. http://software.intel.com/sites/landingpage/IntrinsicsGuide/.
[48] Intel® 64 and IA-32 architectures Software Developer's Manual, Volume 2: Instruction Set Reference, A-Z.
[49] Intel® Architecture Instruction Set Extensions Programming Reference, https://software.intel.com/sites/default/files/managed/68/8b/319433-019.pdf.
[50] Newton command in the manual of LAMMPS. http://lammps.sandia.gov/doc/newton.html?highlight=newton.
[51] Q. Yin, R. Luo, P. Guo, 2012 Eighth International Conference on Computational Intelligence and Security, IEEE Computer Society, 2012, pp. 209–213.