The Journal of Supercomputing
https://doi.org/10.1007/s11227-019-02860-3

FastNBL: fast neighbor lists establishment for molecular dynamics simulation based on bitwise operations

Kun Li[1,2] · Shigang Li[1] · Shan Huang[1,2] · Yifeng Chen[3] · Yunquan Zhang[1]

Abstract
In the molecular dynamics simulation, an important step is the establishment of neighbor list for each particle, which involves the distance calculation for each particle pair in the simulation space. However, the distance calculation will cause costly floating-point operations. In this paper, we propose a novel algorithm, called Fast Neighbor List, which establishes the neighbor lists mainly using the bitwise operations. Firstly, we design a data layout, which uses an integer value to represent the three-dimensional coordinates of a particle. Then, a bunch of bitwise operations and two subtraction operations are used to judge whether the distance between a pair of particles is within the cutoff radius. We demonstrate that our algorithm can deal with the periodic boundary seamlessly. We also use single instruction multiple data (SIMD) instructions to further improve the performance. We implement our algorithm on Intel Xeon E5-2670, ARM v8, and Sunway many-core processors, respectively. Compared with the traditional method, our algorithm achieves on average 1.79x speedup on Intel Xeon E5-2670 processor, 3.43x speedup on ARM v8 processor, and 4.03x speedup on Sunway many-core processor. After using SIMD instructions, our algorithm achieves on average 2.64x speedup and 14.43x speedup on Intel Xeon E5-2670 and ARM v8 processors, respectively.

Keywords Neighbor list · Bitwise operations · SIMD · Molecular dynamics

✉ Shigang Li
shigangli.cs@gmail.com

[1] State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China

[2] School of Computer and Control Engineering, University of Chinese Academy of Sciences, Beijing, China

[3] HCST Key Lab at School of EECS, Peking University, Beijing, China

Published online: 29 April 2019

1 Introduction and related work

Molecular dynamics simulation is widely used to investigate the physical properties, biological structures, or chemical processes for a large number of particles [25]. In the past decades, significant time and resources have been devoted to the development of such molecular dynamics packages, including GROningen MAchine [1, 26] (GROMACS), Large-scale Atomic/Molecular Massively Parallel Simulator [10, 23, 29, 30] (LAMMPS) and Nanoscale Molecular Dynamics [19, 28] (NAMD), among many other commercial and open-source options. These packages provide robust molecular dynamics implementations for massively parallel computers. A more recent addition is HOOMD-blue [5, 6], which was developed and optimized for GPUs. HOOMD-blue performs an order of magnitude faster than a multi-core CPU in typical benchmarks on a single NVIDIA GPU [16]. Despite these advances in hardware and software, molecular dynamics simulation still remains challenging because the computation of interaction forces is extremely time-consuming [20]. This is mainly because the force computation requires to calculate the interactions between each pair of particles in the system, giving rise to $O(N^2)$ evaluations of the interaction in each time step, where N is the total number of particles. It is very costly to carry out such a calculation when a great quantity of particles are simulated. Some typical optimization methods are designed for reducing the cost in this process.

Typically, the interaction forces decrease rapidly as the distance between particles increases, which are called short-range forces [27, 35]. Based on the above assumption, a cutoff radius r_{cut} is introduced to reduce the cost of force computation. If the distance between particles is greater than r_{cut}, the interaction forces are neglected. This means a central particle only has interaction forces with its neighbor particles, which are located in a globe with the central particle as its center and r_{cut} as its radius [3]. Thus, the cost of force computation is reduced significantly. Typically, the neighbor particles of a central particle are detected and stored in the neighbor list, and this process is called neighbor lists establishment [34]. Consequently, the performance bottleneck moves from force computation to neighbor lists establishment. Two typical algorithms are used to establish the neighbor lists. One is Verlet table algorithm [32] and the other is cell linked list algorithm [21].

The basic idea of the Verlet table algorithm is to construct a list of neighboring particles for every particle. The definition of r_{skin} is introduced, which is an extension of r_{cut} [31]. r_{skin} produces a skin surrounding the globe, of which the radius is r_{cut}. Let $r_{\text{ext}} = r_{\text{cut}} + r_{\text{skin}}$. A particle is added in a neighbor list of a central particle if the distance between them is less than r_{ext} [24, 35]. It is critical for setting r_{skin} large enough, so that the exact neighbor particles (those which are less than r_{cut} away from the central particle) are all included in the neighbor lists in a presented number of time steps. In this way, the neighbor lists remain unchanged for a certain number of time steps [2, 32], which reduces the frequency of neighbor lists reconstruction. Since each pair of particles have to be evaluated during the neighbor lists establishment, the computational complexity is $O(N^2)$, where N is the total number of particles [14, 35]. The Verlet table algorithm is appropriate

for the situation where the total number of particles is relatively small and the particles move slowly in the space [35].

The cell linked list algorithm is characterized by its cell partition and linked lists. The simulation space is partitioned into cells, the edge of which is usually equal to or larger than r_{cut}. All particles are distributed to these cells according to their positions. For each cell, it has 8 neighbor cells for a 2D simulation space or 26 neighbor cells for a 3D simulation space [17, 35]. Since the edge of each cell is equal to or larger than r_{cut}, the neighbor particles for a central particle can only locate in the cell where the central particle locates, or locate in its neighbor cells. Each cell maintains a link list to store the particles located in the cell. Assume that the molecular dynamics space is divided into $M_x \cdot M_y \cdot M_z$ cells, the average number of particles in a cell is $N_c = N/(M_x \cdot M_y \cdot M_z)$. Thus, the computational complexity for neighbor lists establishment using the cell linked list algorithm is $O(27NN_c)$ in a 3D simulation space. The cell linked list algorithm constructs the neighbor lists faster than the Verlet table algorithm. However, it has to update the particles located in each cell ($O(N)$ complexity) and reconstruct the neighbor lists after each time step, which is more frequent than the Verlet table algorithm.

The cell linked list algorithm constructs neighbor lists faster and efficiently but a large quantity of particles need to be scanned in every fixed-time interval, the cost of which is extremely expensive in the whole computation process. The Verlet table algorithm reduces the frequency of the neighbor lists reconstruction, while the cell linked list algorithm reduces the computational complexity for each time of neighbor lists construction. However, both of them adopt the Euclidean Metric [9] to calculate the distance between two particles, and then put one particle into the other particle's neighbor list if the distance is less than the cutoff radius. For two particles $P_0\ (x_0, y_0, z_0)$ and $P_1\ (x_1, y_1, z_1)$, the distance between them is calculated by

$$\text{Dis} = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2 + (z_1 - z_0)^2} \tag{1}$$

Equation (1) involves some expensive operations in terms of the clock cycles cost by the processor, such as the square root operation.

To remove the expensive operations in the traditional method, we propose a *Fast N*eigh*B*or *L*ists establishment algorithm, abbreviated to *FastNBL*, which only involves bitwise and subtraction operations. Typically, the bitwise and subtraction operations run much faster than the square root operation and a little faster than the multiplication operation on modern processors [7, 13]. Firstly, we propose a new data layout to transform the three-dimensional coordinates information of each particle into a single integer value, which consists of three bit segments corresponding to three-dimensional coordinates, respectively. Then, based on the data layout, we use a series of bitwise operations and two subtractions to rapidly bypass the non-neighbor particles. The periodic boundary condition judgment is also achieved easily using *FastNBL*. We further exploit the instruction-level parallelism by Single Instruction Multiple Data (SIMD) instructions. Performance evaluation is conducted on Intel Xeon E5-2670, ARM v8, and Sunway many-core

processors. Compared with the traditional method, our algorithm achieves on average 1.79x speedup on Intel Xeon E5-2670 processor, 3.43x speedup on ARM v8 processor, and 4.03x speedup on Sunway processor. After using SIMD instructions, our algorithm achieves on average 2.64x speedup and 14.43x speedup on Intel Xeon E5-2670 and ARM v8 processors, respectively.

The paper makes the following contributions:

- We propose the *FastNBL* algorithm to accelerate the neighbor lists establishment. *FastNBL* only adopts two subtractions and several bitwise operations to accomplish the judgment of neighbor particles, which significantly outperforms the traditional methods.
- We use SIMD instructions to exploit the instruction-level parallelism on ARM v8 and Intel Xeon E5-2670 processors.
- We demonstrate that the proposed *FastNBL* algorithm can deal with the periodic boundary condition easily.

In the next section, we discuss the motivation of our work. Section 3 introduces the detailed procedure for the *FastNBL* algorithm. The periodic boundary condition processing and the SIMD optimization are also discussed in this section. Experimental results and analysis on ARM v8, Intel Xeon E5-2670 and Sunway processors are presented in Sects. 4 and 5 concludes.

2 Motivation

2.1 The verlet table and the cell linked list algorithms

Algorithm 1 lists the Verlet table algorithm for creating neighbor lists. From line 1 and line 3, we can see the computational complexity is $O(N^2)$, since two for-loop structures exist, where N is the total number of particles [4, 35]. The computation overhead mainly comes from lines 5 and 6, namely the distance calculation based on Euclidean Metric.

Algorithm 1 The Verlet table algorithm

```
1: for i = 1 to N do
2:     nbn[i] = 0
3:     for j = 1 to N do
4:         if (i != j) then
5:             d_x = x_j − x_i, d_y = y_j − y_i, d_z = z_j − z_i
6:             d = sqrt (d_x^2 + d_y^2 + d_z^2)
7:             if (d ≤ r_ext) then
8:                 nblist[i][nbn[i]] = j
9:                 nbn[i]++
10:            end if
11:        end if
12:    end for
13: end for
```

Algorithm 2 lists the cell linked list algorithm. The cell edge is usually set as the cutoff distance r_{cut} so that all particles in 27 cells, or in the volume of 27 r_{cut}^3, will be scanned in evaluation procedure [17, 35]. The computational cost of building the neighbor list is highly decreased by restricting the neighbor search for each particle in a set of several cells. The particles, which locate in a cell, are stored in a linked list maintained by the cell. The head pointers for all the linked lists maintained by the cells are stored in the *headPointers* array. As can be seen from Algorithm 2, the first for-loop perform a traversal for N particles. For each particle, it loops over the 27 neighbor cells (line 4) to check the neighbor particles rather than the whole space like verlet algorithm. Assuming the average number of particles that each cell has is N_c, then the computational complexity for the cell linked list algorithm is $O(27NN_c)$. Although it has lower computational complexity than the Verlet table algorithm, it has to be called every time step to update the neighbor lists. The main computation overhead also comes from the distance calculation based on Euclidean Metric (lines 9 and 10).

Algorithm 2 The cell linked list algorithm

```
1: for i = 1 to N do
2:     nbn[i] = 0
3:     //Loop over 27 neighbor cells: Cell_0 to Cell_26
4:     for Cell_k = Cell_0 to Cell_26 do
5:         pointer = headPointers(Cell_k)
6:         j = pointer-> particleID
7:         while do(j ≠ 0)
8:             if j ≠ i then
9:                 d_x = x_j − x_i, d_y = y_j − y_i, d_z = z_j − z_i
10:                d = sqrt (d_x^2 + d_y^2 + d_z^2)
11:                if (d ≤ r_cut) then
12:                    nblist[i][nbn[i]] = j
13:                    nbn[i]++
14:                end if
15:            end if
16:            pointer = pointer-> next
17:            j = pointer-> particleID
18:        end while
19:    end for
20: end for
```

2.2 Performance problems in the existing algorithms

Apparently, the frequent distance calculation is the main cost for constructing the neighbor lists for both algorithms. As shown in lines 5 and 6 in Algorithm 1 and lines 9 and 10 in Algorithm 2, the distance calculation based on Euclidean Metric needs five addition or subtraction operations, three multiplication operations, and one square root operation. On modern processors, it commonly costs 1 clock cycle for addition/subtraction and 10 clock cycles for multiplication operations. Addition or subtraction operation is a little faster than the multiplication operation. For the square root operation, it usually takes up thousands of clock cycles, which is quite

time-consuming for massive computation in molecular dynamics simulation. On the contrary, it only costs 1 clock cycle for a bitwise operation. In this paper, we propose the *FastNBL* algorithm, which only needs two subtractions and thirteen bitwise operations. As a result, *FastNBL* improves the computation efficiency remarkably by employing fast bitwise operations to replace traditional distance calculation in the procedure of neighbor lists establishment for either the Verlet table algorithm or the cell linked list algorithm.

3 Fast neighbor lists establishment based on the bitwise operations

In this section, we discuss the *FastNBL* algorithm in detail. Figure 1 presents the key steps of the *FastNBL* algorithm. For the first three steps, *FastNBL* transforms the original three-dimensional coordinates of each particle into an integer value, called "integer coordinate". An integer coordinate consists of three bit segments, which represent the three-dimensional lattice coordinates of a particle, respectively. The last six steps include a series of bitwise operations and two subtraction operations performed on the two integer coordinates for each pair of particles, and output whether one particle is a neighbor of the other particle.

3.1 The key steps for the *FastNBL* algorithm

Step 1. Build the lattice coordinates

Figure 2a exhibits an example of the particles distribution in a 3D simulation space. The length of each dimension is L_x, L_y, and L_z, respectively. The original coordinates of each particle are represented by three floating-point values. For example, (x_0, y_0, z_0) are the original three-dimensional coordinates for a central particle P_0; (x_1, y_1, z_1) are the original three-dimensional coordinates for particle P_1.

The *FastNBL* algorithm adopts an integer value to store the position information of each particle. However, the original coordinates are represented by floating-point values. If the rounding operations are used to change the original floating-point coordinates to integer coordinates, it will make two particles have the same position with a high probability. Thus, we propose lattice coordinates to avoid two particles having the same position. We partition the 3D simulation space into small lattices (as exhibited in Fig. 2b), whose edge length is less than the theoretical minimum of the distance between two particles. Thus, there is at most one particle in a lattice, as shown in Fig. 2c, d. We can use the coordinates (integers) of a lattice to approximately represent the coordinates of the particle in the lattice. As a result, we transform the original floating-point coordinates into the new lattice (integer) coordinates.

Step 2. Compute the integer coordinates

In this step, we compute the integer coordinate utilizing the lattice coordinates obtained in Step 2. As shown in Fig. 1, the original coordinate of the central particle P_0 is transformed into the lattice coordinate (x'_0, y'_0, z'_0) after lattice partition. Here, we assume that the maximal number of lattices for each dimension of the

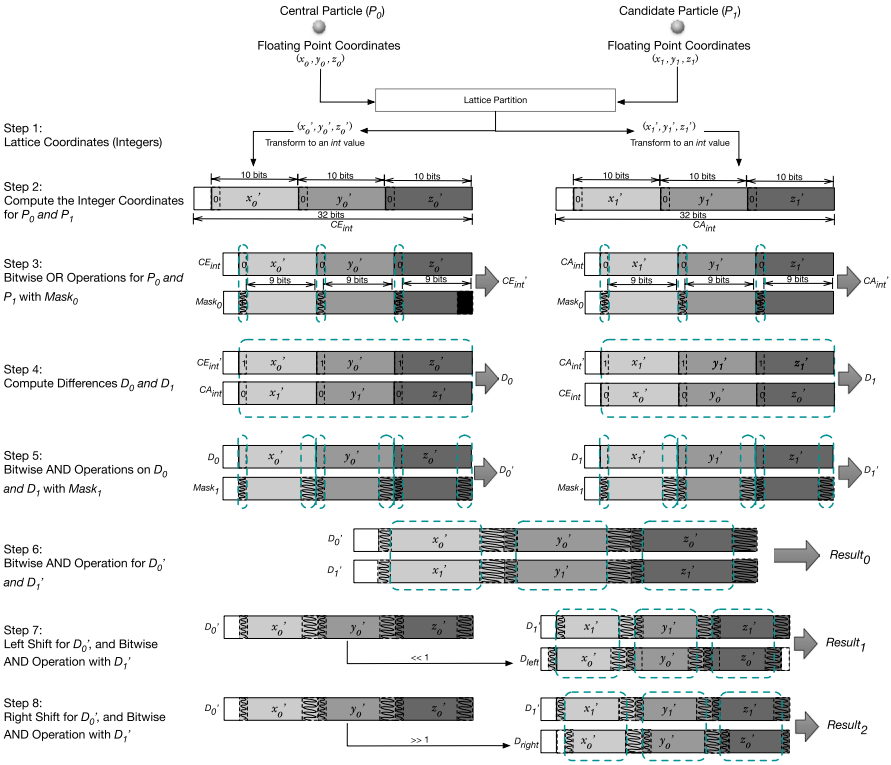
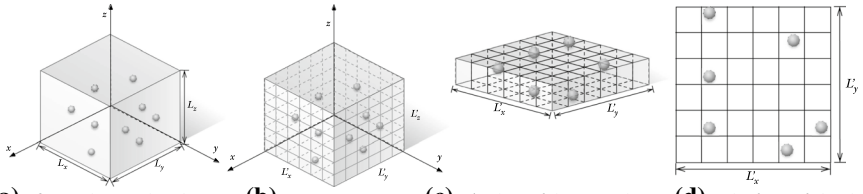


Fig. 1 The steps of the *FastNBL* algorithm

3D simulation space is 512. We use a segment of 9 binary bits to represent the lattice coordinate for each dimension. In addition, a reserved bit, which is set to "0", is placed in the front of each segment. Thus, the three segments occupy total 30 bits. We use an integer value with *int* type, which occupies 32 bits, to contain the three segments. The highest 2 bits of the *int* value are untapped and set to "0". We call this single *int* data "integer coordinate". As shown in Fig. 1, z_0' is placed from the first bit to the ninth bit of the integer coordinate; y_0' is placed from the eleventh bit to the nineteenth bit using left shift and bitwise OR operations; x_0' is placed from the twenty-first bit to the twenty-ninth bit using left shift and bitwise OR operations. It is worth noting that the tenth bit, the twentieth bit and the thirtieth bit of the integer coordinate are the reserved bits. The integer coordinates of P_0 and P_1 obtained after Step 3 are defined as $\mathrm{CE_{int}}$ and $\mathrm{CA_{int}}$, respectively. If any dimension of the 3D simulation space has more than 512 lattices, we will use the data type with more bits to represent the integer coordinate, such as *long long* with 64 bits.

Step 3. Execute bitwise OR operation with $\mathbf{Mask_0}$

This step executes a bitwise OR operation with a preset value $\mathrm{Mask_0}$ on the integer coordinates of P_0 and P_1. The three reserved bits of $\mathrm{Mask_0}$ are "1" and the



(a) Original particles distribution **(b)** Lattice partition **(c)** A slice of lattices along the horizontal direction **(d)** Planform of the slice

Fig. 2 The lattice coordinates for particles

other bits are "0". As a result, all three reserved bits of each integer coordinate are set to "1". Since we will calculate the difference for each segment between the integer coordinates of P_0 and P_1, this step guarantees that the difference is always a positive value. For the central particle P_0 and the neighbor particle P_1, the integer coordinates, whose reserved bits are set to "1", are defined as $\mathrm{CE}'_{\mathrm{int}}$ and $\mathrm{CA}'_{\mathrm{int}}$, respectively.

Step 4. Compute the differences

Step 5 in Fig. 1 exhibits the process of computing the differences between the integer coordinates of P_0 and P_1. Note that all the reserved bits of $\mathrm{CE}'_{\mathrm{int}}$ and $\mathrm{CA}'_{\mathrm{int}}$ are set to "1"; on the contrary, all the reserved bits of $\mathrm{CE}_{\mathrm{int}}$ and $\mathrm{CA}_{\mathrm{int}}$ are "0". We subtract the integer coordinate $\mathrm{CA}_{\mathrm{int}}$ of P_1 from $\mathrm{CE}'_{\mathrm{int}}$ and obtain the first difference value D_0. Similarly, D_1 is obtained by subtracting the integer coordinate $\mathrm{CE}_{\mathrm{int}}$ of P_0 from $\mathrm{CA}'_{\mathrm{int}}$.

Step 5. Execute bitwise AND operation with $\mathbf{Mask}_1$

Mask_1 is an *int* value with 32 bits. The three reserved bits of Mask_1 are set to "0". We use R to denote the cutoff radius. Except for the reserved bits, the lower $\lceil \log_2(R+1) \rceil$ bits of each segment of Mask_1 are also set to "0". In Fig. 1, we assume the cutoff radius is equal to 3. Thus, the lower 2 bits of each segment of Mask_1 are set to "0". All the other bits of Mask_1 are set to "1". When D_0 is executed with Mask_1 by the bitwise AND operation, we obtain a new value D'_0. In this way, the reserved bits and the lower 2 bits of each segment of D'_0 are set to "0", and the other bits of D'_0 are the same as D_0. Similarly, we obtain D'_1 when D_0 is executed with Mask_1 by the bitwise AND operation. We will use D'_0 and D'_1 in the following steps to judge whether the distance between the two particles is within the cutoff radius.

Step 6. Execute bitwise AND operation between D'_0 ***and*** D'_1

For each segment of the three dimensions, if either D'_0 or D'_1 contains all "0" bits, P_1 is a neighbor of P_0. For other cases, P_1 is not a neighbor of P_0. Firstly, we perform a bitwise AND operation between D'_0 and D'_1, and obtain a result Result_0. If Result_0 is equal to zero, the algorithm continues to the next step for further judgment. However, if Result_0 is larger than zero, P_1 is not the neighbor of P_0 and the algorithm returns. For example, Result_0 is larger than zero for Case_1 in Fig. 3. For this case, P_1 is not the neighbor of P_0. Result_0 is equal to zero for the Case_0, Case_2 and Case_3 in Fig. 3, which need further judgment.

Step 7. Left Shift and bitwise AND operations for D'_0 ***and*** D'_1

"$\text{Result}_0 == 0$" is not a sufficient condition to state that P_1 is the neighbor of P_0. We need further judgment. This is because even "$\text{Result}_0 == 0$" is satisfied, it is not guaranteed that the segment of each dimension of either D'_0 or D'_1 is zero. For instance, we will obtain all "0" bits for the segment of Y dimension in Result_0 when the bits from the third bit to the ninth bit of D'_0 are exactly opposite to the corresponding bits of D'_1; however, neither the segment of Y dimension of D'_0 nor the segment of Y dimension of D'_1 is equal to zero, such as Case_2 and Case_3 in Fig. 3. In this step, we use two bitwise operations to determine that P_1 is not the neighbor of P_0 for Case_3 in Fig. 3. We carry out an 1-bit left shift operation on D'_0. The lowest bit of D'_0 is filled by "0" and the highest bit of D'_0 is removed out. As a result, a new shifted value D_{left} is obtained. We perform an bitwise AND operation on D_{left} and D'_1, and obtain the result Result_1. If Result_1 is equal to zero, such as Case_0 and Case_2 in Fig. 3, the algorithm continues to the next step for further judgment. However, if Result_1 is larger than zero, P_1 is not the neighbor of P_0 and the algorithm returns, such as Case_3 in Fig. 3.

Step 8. Right Shift and bitwise AND operations for D'_0 and D'_1

Similarly, we use two bitwise operations to determine that P_1 is not the neighbor of P_0 for Case_2 in Fig. 3. We carry out an 1-bit right shift operation on D'_0. The highest bit of D'_0 is filled by "0" and the lowest bit of D'_0 is removed out. As a result, a new shifted value D_{right} is obtained. We perform an bitwise AND operation on D_{right} and D'_1, and obtain the result Result_2. If Result_2 is equal to zero, P_1 is the neighbor of P_0 and the algorithm returns, such as Case_0 in Fig. 3. However, if Result_2 is larger than zero, P_1 is not the neighbor of P_0 and the algorithm returns, such as Case_2 in Fig. 3.

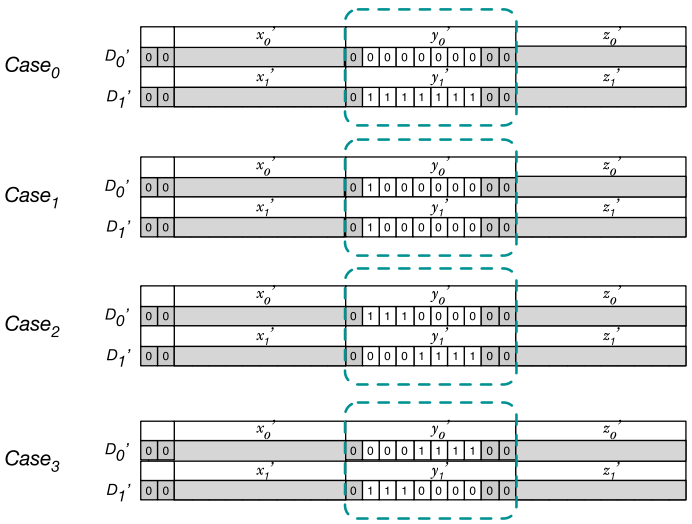


Fig. 3 Different cases to judge whether the distance between a pair of particles is within the cutoff radius (equal to 3). Only the segment of Y dimension is shown. Suppose the segments of other dimensions are the same as the Y dimension

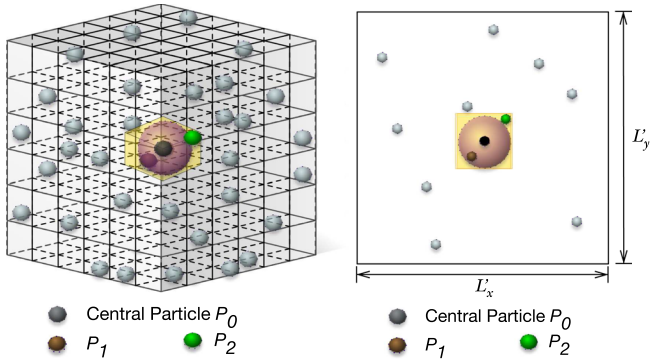
In fact, both D'_0 and D'_1 are obtained by bitwise AND operation for D_0 and D_1 with Mask_1, respectively, in Step 5, and D_0 and D_1 are not used in the following steps after that. We can notice that the candidate particle P_1 is a neighbor of the central particle P_0 only if Result_0, Result_1 and Result_2 are all equal to zero.

The overhead is reduced dramatically using the *FastNBL* algorithm, since almost all the operations are bitwise (except two subtraction operations). On the contrary, both the Verlet table algorithm and the cell linked list algorithm adopt a traditional distance calculation method, which has to perform costly floating-point operations.

3.2 The precise neighbor list

From the perspective of the *FastNBL* algorithm, one particle is a neighbor of another particle if the distances in three dimensions between the two particles are all within the cutoff radius. Actually, the *FastNBL* algorithm recognizes a cubic neighbor region, whose side length is equal to two times the length of the cutoff radius. On the contrary, the traditional method recognizes a spherical region, whose radius is equal to the length of the cutoff radius. The purple sphere in Fig. 4a and the purple circle in Fig. 4b are the neighbor regions recognized by the traditional method, while the yellow cube in Fig. 4a and the yellow square in Fig. 4b are the neighbor regions recognized by *FastNBL*. We can notice that the neighbor region recognized by *FastNBL* is a little larger than the traditional method.

As shown in Fig. 4, *FastNBL* recognizes that P_2 is a neighbor particle of P_0, but the traditional method not. Thus, we first use *FastNBL* to obtain an approximate neighbor list quickly. Then, we carry out the traditional distance calculation, i.e., Equation (1), on the particles in the approximate neighbor list to further obtain a precise neighbor list. In this way, P_2 is eliminated from the approximate neighbor list of P_0. The number of particles in the approximate



(a) A 3D simulation space **(b)** A 2D planform of the simulation space

Fig. 4 The neighbor region comparison between the traditional distance calculation method and the *FastNBL* algorithm

neighbor list is much less than that in the whole simulation region. Therefore, compared with traditional method which calculates the distances between the central particle and all other particles in the simulation region, the overhead of the precise neighbor list calculation carried out after *FastNBL* is much lower.

The whole workflow of *FastNBL* algorithm is shown in Algorithm 3. The N_{space} in line 7 is equal to N or $27 \times N_{\text{c}}$ when it is applied to Verlet table algorithm and cell linked algorithm, respectively. For-loop at line 1 is the step 1 in *FastNBL* algorithm. Line 9 to line 13 correspond to step 3 to step 8. According to the results by line 14, a neighbor judgment is made. Of course, if a higher accuracy is required, traditional distance calculation will be performed after *FastNBL* algorithm. When the traditional Verlet table algorithm is used, the workflow of finishing a neighbor judgment requires at least 9 thousands clock cycles roughly. However, since almost all operations are bitwise, only 15 clock cycles are required by our *FastNBL* algorithm. Thus, a theoretical 600x speedup is obtained if precise neighbor list is not required.

Algorithm 3 *FastNBL* algorithm

Require:
N: The total number of particles.
P_0: The current central particle.
P_1: The current candidate neighbor particle.

1: Build the lattice coordinates.
2: **for** $i = 0$ to N **do**
3: Compute an integer coordinate for each particle.
4: **end for**
5: **for** $i = 0$ to N **do**
6: Choose an integer coordinate CE_{int} for P_0.
7: **for** $i = 0$ to N_{space} **do**
8: Choose an integer coordinate CA_{int} for P_1.
9: $D_0^{'} = [(CE_{int} \mid Mask_0) - CA_{int}] \& Mask_1$
10: $D_1^{'} = [(CA_{int} \mid Mask_0) - CE_{int}] \& Mask_1$
11: $Result_0 = D_0^{'} \& D_1^{'}$
12: $Result_1 = (D_0^{'} \ll 1) \& D_1^{'}$
13: $Result_2 = (D_0^{'} \gg 1) \& D_1^{'}$
14: **if** $(Result_0 == 0) \&\& (Result_1 == 0) \&\& (Result_2 == 0)$ **then**
15: **if** Precise neighbor list is required. **then**
16: Perform traditional distance calculation.
17: **else**P_1 is a neighbor for P_0 by *FastNBL*.
18: **end if**
19: **else**P_1 is not a neighbor for P_0.
20: **end if**
21: **end for**
22: **end for**

3.3 A case study for the *FastNBL* algorithm

In this section, we present a case study, as shown in Fig. 5, to further illustrate how does *FastNBL* work and demonstrate the accuracy of the algorithm. We suppose the cutoff radius is equal to 3 in this case. In *Step 1*, we transform the original

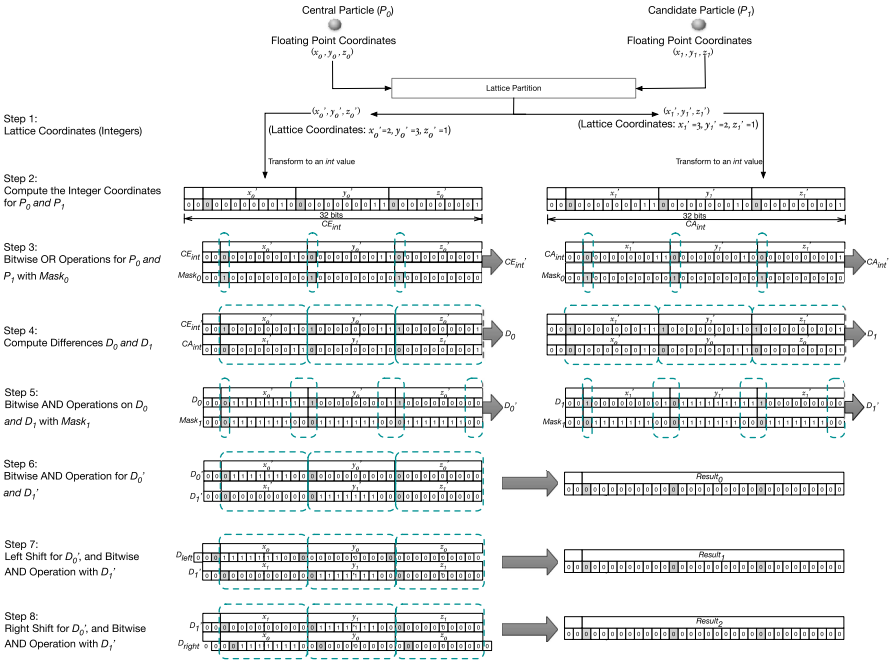


Fig. 5 A case study for the *FastNBL* algorithm. Suppose the cutoff radius is equal to 3

floating-point coordinates into the lattice coordinates. The lattice coordinates of the central particle P_0 and the candidate particle P_1 are (2, 3, 1) and (3, 2, 1), respectively. In *Step 2*, we compute the integer coordinates, $\mathrm{CE}_{\mathrm{int}}$ and $\mathrm{CA}_{\mathrm{int}}$, for P_0 and P_1 using left shift and bitwise OR operations, respectively. Note that the bit segments for three dimensions are separated by the reserved bits. In *Step 3*, we set all three reserved bits of $\mathrm{CE}_{\mathrm{int}}$ and $\mathrm{CA}_{\mathrm{int}}$ to "1", and obtain $\mathrm{CE}'_{\mathrm{int}}$ and $\mathrm{CA}'_{\mathrm{int}}$. In *Step 4*, we compute two differences, D_0 and D_1, where $D_0 = \mathrm{CE}'_{\mathrm{int}} - \mathrm{CA}_{\mathrm{int}}$ and $D_1 = \mathrm{CA}'_{\mathrm{int}} - \mathrm{CE}_{\mathrm{int}}$. In *Step 5*, we obtain D'_0 and D'_1 by setting the reserved bits and the lower 2 bits of each segment of D_0 and D_1 to "0". In *Step 6*, we obtain Result_0 by conducting bitwise AND operation on D'_0 and D'_1. As shown in Fig. 5, Result_0 is equal to 0. In *Step 7*, we obtain Result_1 if we right shift D'_0 by 1 bit and then conduct bitwise AND operation with D'_1. In *Step 8*, we obtain Result_2 if we left shift D'_0 by 1 bit and then conduct bitwise AND operation with D'_1. Note that Result_1 and Result_2 are also equal to 0. Thus, we recognize that P_1 is a neighbor particle of P_0 using *FastNBL*. The distance between P_0 and P_1, calculated by Eq. (1), is also less than the cutoff radius, which demonstrates the correctness of *FastNBL*.

3.4 Periodic boundary condition

Periodic boundary conditions are a set of boundary conditions which are often chosen for approximating a large or infinite system by using a small simulation region.

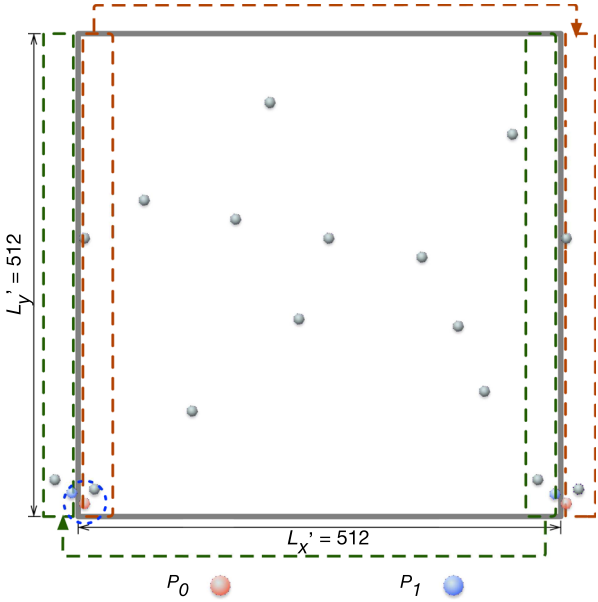


Fig. 6 A 2D planform of periodic boundary situation. The edge length of simulation space is 512

The simulation space is extended in each dimension periodically [27]. As shown in Fig. 6, the rightmost strip region along the *X* dimension will be the periodic boundary for the leftmost region, and vice versa. In Fig. 6, suppose the edge length of simulation space is 512 and the cutoff radius is 3, and the lattice coordinates for P_0 and P_1 are (1, 3, 2) and (511, 4, 1), respectively. After calculating the distance between P_0 and P_1 using Eq. (1), we can see that P_1 is not a neighbor particle of P_0. However, in the periodic boundary, the coordinates of P_1 can be changed to $(-1, 4, 1)$, and then the distance between P_0 and P_1 is less than 3. Thus, P_1 is a neighbor particle of P_0 under the periodic boundary condition. Both Verlet table algorithm and cell linked algorithm need to adopt additional calculations to support the periodic boundary condition, which will bring extra overhead inevitably [22].

On the contrary, the problem of periodic boundary condition can be solved by the *FastNBL* algorithm efficiently. The *FastNBL* algorithm can deal with the periodic boundary condition without extra operations. When the edge length of the simulation space is a power-of-two, the periodic boundary condition is solved by *FastNBL* the same as that shown in Fig. 1. When the edge length of the simulation space is a non-power-of-two, more bits of Mask_1 need to be set to "0". For instance, suppose the edge length of the simulation space is 18, which is a non-power-of-two, and the cutoff radius is equal to 3. Apart from the reserved bits and the lowest 2 bits corresponding to the length of the cutoff radius, the fifth bit in every segment of Mask_1 is also set to "0". This Mask_1 guarantees that if the distance between a particle pair for one dimension is located in a interval [15, 17], one particle may still be the neighbor of another particle. Here we use the example in Fig. 6 to show that *FastNBL* can deal with the periodic boundary condition seamlessly. Recall that the lattice coordinates of P_0 and P_1 are (1, 3, 2) and (511, 4, 1), respectively. Following the steps

shown in Fig. 1, we obtain the values of Result_0, Result_1 and Result_2, which are all equal to zero. Thus, P_1 is a neighbor of P_0 under the periodic boundary condition.

3.5 The SIMD optimization for *FastNBL*

Modern multi/many-core architectures [7, 15, 18] commonly support wide Single Instruction Multiple Data (SIMD) instructions. Utilizing the SIMD instructions efficiently is essential to achieve high performance on these architectures. We use the bitwise SIMD instructions, if they are supported on a specific platform, to exploit the instruction-level parallelism of *FastNBL* and further improve the performance. In the process of SIMD optimization, the vector data type which contains four integers is called *int4*. Similarly, *int8* refers to the vector data type which consists of eight integers. We put the integer coordinates of four or eight candidate particles into a *int4* or *int8* vector. Then, the *FastNBL* algorithm deals with four or eight candidate particles simultaneously when using the bitwise SIMD instructions on the *int4* or *int8* vector. At last, we obtain the vector values of Result_0, Result_1, and Result_2. We carry out SIMD OR instructions on Result_0, Result_1, and Result_2, and obtain a vector value $\text{Result}_{\text{final}}$. If all the four or eight scalar values in $\text{Result}_{\text{final}}$ are larger than zero, all the four or eight candidate particles are not the neighbor particles of the central particle. Otherwise, if any scalar in $\text{Result}_{\text{final}}$ is equal to zero, the corresponding candidate particle is the neighbor particle of the central particle.

4 Evaluation

4.1 Experimental environment introduction

The experiments are conducted on different architectures, including x86 [18], ARMv8 [7], and Sunway many-core architectures [15]. For the x86 architecture, we use Intel Xeon E5-2670 v3 as the experimental platform. It contains 2 processors connected by QPI, and each processor has 12 cores sharing a 30 MB unified L3 cache. Thus, there are total 24 cores with a frequency of 2.30 GHz on Xeon E5-2670. It supports Intel AVX2 vector instructions, the register width of which is 256 bits.

For the ARMv8 architecture, we use ARM Cortex-A57 as the experimental platform. The ARM Cortex-A57 is a microarchitecture implementing the ARMv8-A 64-bit instruction set designed by ARM [8]. It contains 16 cores with a frequency of 2.1 GHz. Each core has a 48 KB L1 instruction cache and a 32 KB L1 data cache [12]. It supports NEON vector instructions, the register width of which is 128 bits.

We also conduct the experiments on the Sunway many-core architecture, i.e., SW26010 processor. The general architecture of SW26010 is shown in Fig. 7. It

contains four core-groups (CGs). Each CG includes one management processing element (MPE), one computing processing element (CPE) cluster with eight by eight CPEs, and one memory controller (MC). The processor connects to other outside devices through a system interface (SI). For convenience, we call MPE as master core and call CPE as slave core in this paper. The master core has a 32 KB L1 instruction cache, a 32 KB L1 data cache, and a unified 256 KB L2 cache. Each slave core has a 16 KB L1 instruction cache, and a 64 KB local store (user-controlled scratch pad memory) [33]. Both the master core and the slave cores work at 1.45 GHz and support 256-bit vector instructions. However, it does not support bitwise vector instructions.

We use the implementation of the Verlet table algorithm, which adopts the traditional distance calculation method, as our baseline. The baseline is named as *Naive* in the following discussion. We use *FastNBL* to denote the fast neighbor lists establishment algorithm proposed in this paper. We use *FastNBL (SIMD 4)* to denote the *FastNBL* algorithm using *int4* SIMD instructions. We use *FastNBL (SIMD 8)* to denote the *FastNBL* algorithm using *int8* SIMD instructions. The size of the 3D simulation space is $512 \times 512 \times 512$. The simulation space contains total 4096 particles. The cutoff radius is set to 3. All the evaluations are run for 1024 times and we present the average runtime in the following figures.

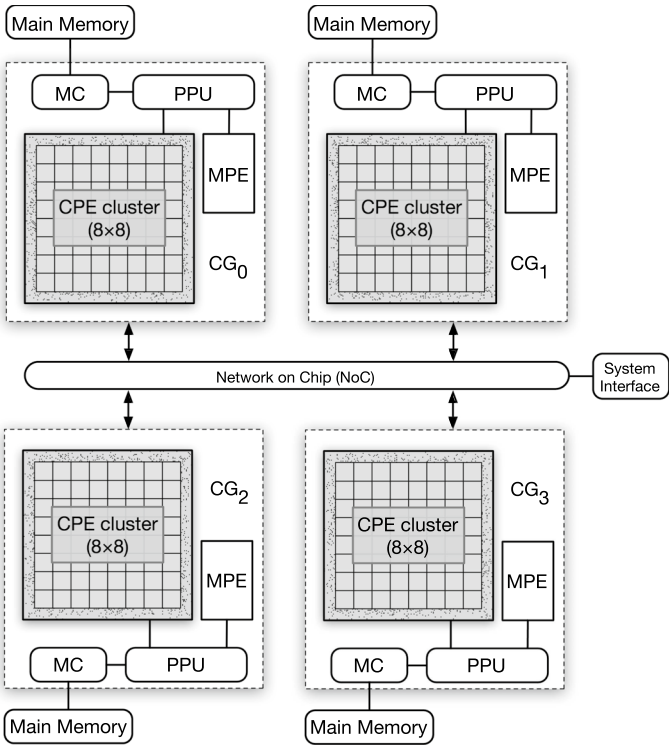


Fig. 7 Sunway many-core architecture

4.2 Performance evaluation on different architectures

The performance comparison on Intel Xeon E5-2670 v3 is shown in Fig. 8. To utilize multiple cores on the processor, we simply use the OpenMP [11] compiler directives to partition the workload and fork multiple threads. Compared with the *Naive* algorithm, the *FastNBL* algorithm achieves on average 1.79x speedup for the core numbers from 1 to 24, which illustrates that the overhead of the neighbor lists establishment is significantly decreased by *FastNBL*. In addition, bitwise SIMD instructions are supported on Intel Xeon E5-2670 v3. Since the program control is predictable and the application for neighbor list establishment is massively data parallel, SIMD is a good option to exploit the parallelism. However, if the *Naive* algorithm is performed by SIMD operations directly, the complex multiplication and square root operations bring extra intermediate results and redundant SIMD instructions, which is much more programmer-unfriendly. Even though a modern compiler can and will implement some complex calculations using a series of bitwise instructions, the Naïve algorithm cannot be optimized by compiler thoroughly. The proposed FastNBL algorithm is designed for solving this problem. Since we have achieved establishing neighbor list by several bitwise operations, no redundant bitwise optimizations are required by compiler and the algorithm is implemented with SIMD instructions efficiently. After using the *int4* and *int8* SIMD instructions, *FastNBL (SIMD 4)* and *FastNBL (SIMD 8)* further improve the performance to some extent. Compared with the *Naive* algorithm, *FastNBL (SIMD 4)* and *FastNBL (SIMD 8)* achieve on average 2.31x speedup and 2.64x speedup, respectively. This demonstrates that *FastNBL* is SIMD-friendly and can fully exploit the instruction-level parallelism. *FastNBL (SIMD 8)* performs better than *FastNBL (SIMD 4)*. This is because the SIMD instructions on Xeon E5-2670 have a 256-bit register width, and the *int8* data type can fully utilize the 256-bit registers. Finally, the best performance is achieved when *FastNBL (SIMD 8)* running on all 24 cores.

The performance comparison on ARM Cortex-A57 is shown in Fig. 9. Compared with the *Naive* algorithm, the *FastNBL* algorithm achieves on average 3.43x speedup

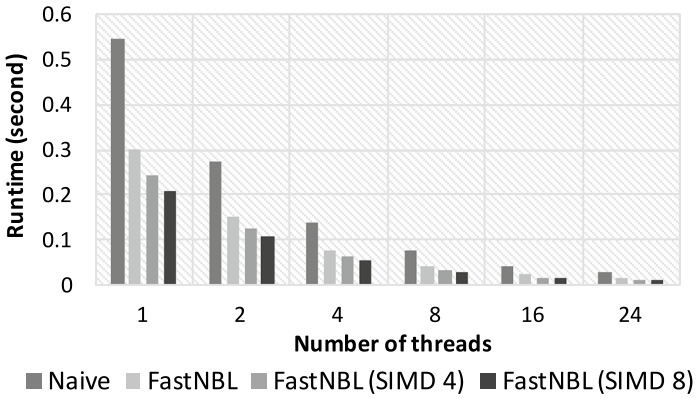


Fig. 8 The runtime comparison on Intel Xeon E5-2670 v3, using up to 24 cores

for the core numbers from 1 to 16. Compared with the *Naive* algorithm, *FastNBL (SIMD 4)* and *FastNBL (SIMD 8)* achieve on average 13.37x speedup and 14.43x speedup, respectively. The SIMD optimization on ARM Cortex-A57 achieves significant performance improvement compared with the scalar implementation of *FastNBL*. However, compared with *FastNBL (SIMD 4)*, *FastNBL (SIMD 8)* almost has no performance advantage. This is because the register width of the NEON vector instructions is 128 bits, and *FastNBL (SIMD 4)* has already fully utilized the 128-bit registers. Thus, using *int8* SIMD instructions brings no more speedup compared with *int4*.

The performance comparison on Sunway many-core processor is shown in Fig. 10. To utilize multiple slave cores on SW26010, we use the Sunway Athread library (similar to Pthread) to fork multiple slave threads. Compared with the *Naive* algorithm, the *FastNBL* algorithm achieves on average 4.03x speedup for the core numbers from 1 to 64. Since SW26010 many-core processor does not support bitwise vector instructions, only the runtime of the scalar version of *FastNBL* is presented in Fig. 10. Overall, *FastNBL* and its SIMD versions significantly outperform the *Naive* algorithm on different multi/many-core architectures, which demonstrates the performance portability of *FastNBL*.

The runtime comparison between different algorithms when using only one core for each architecture is shown in Table 1. We can see that the x86 architecture achieves good performance for all four algorithms. Meanwhile, the SIMD optimization effect for *FastNBL* on the x86 architecture is not significant. This is probably because the compiler has done the automatic vectorization for the x86 architecture. The Sunway slave core achieves the worst performance among all different processors/cores. This is because the local store of each slave core cannot hold the information of all the particles, which has to be stored in the main memory. It would cause frequent data transfer between the main memory and the local store during the calculation.

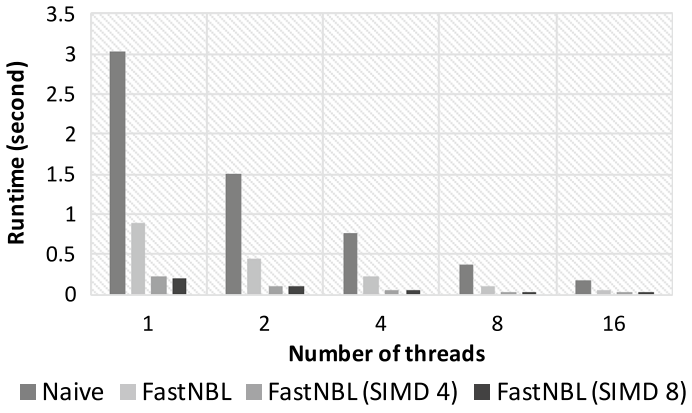


Fig. 9 The runtime comparison on ARM Cortex-A57, using up to 16 cores

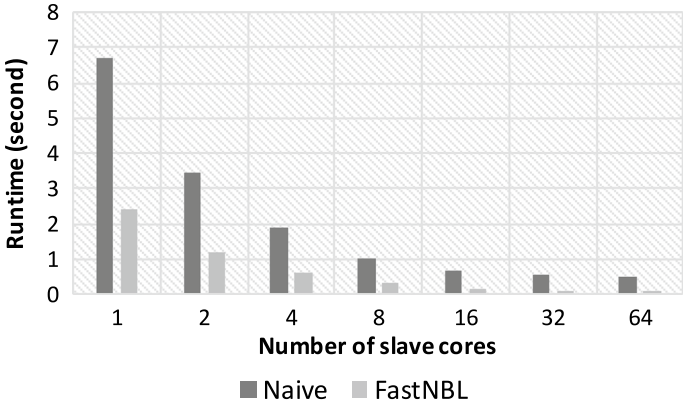


Fig. 10 The runtime comparison on SW26010 many-core processor, using up to 64 slave cores

Table 1 The runtime (s) comparison on different architectures

	x86	ARMv8	Sunway master	Sunway slave
Naive	0.5445	3.0264	1.0978	6.6730
FastNBL	0.3013	0.8823	0.2632	2.4479
FastNBL (SIMD 4)	0.2153	0.2255	–	–
FastNBL (SIMD 8)	0.2064	0.2089	–	–

Only one core is used for each architecture

5 Conclusion

Neighbor lists establishment is an essential module in many molecular dynamics simulations. The main runtime overhead of the neighbor lists establishment comes from the distance calculation for all the particle pairs, which involves costly floating-point operations. We propose *FastNBL* to establish the neighbor lists mainly using the bitwise operations. *FastNBL* transforms the three-dimensional coordinates of a particle into a single integer value, based on which a bunch of bitwise operations and two subtraction operations are applied to judge whether the distance between a pair of particles is within the cutoff radius. We demonstrate that our algorithm can deal with the periodic boundary seamlessly. SIMD instructions are used to exploit the instruction-level parallelism of *FastNBL*. Experimental results show that *FastNBL* significantly outperforms the traditional method on Intel Xeon E5-2670, ARM v8, and Sunway many-core architectures.

Actually, the *FastNBL* algorithm recognizes a cubic neighbor region, whose side length is equal to two times the length of the cutoff radius. However, the traditional method recognizes a spherical region, whose radius is equal to the length of the cutoff radius. Thus, the neighbor region recognized by *FastNBL* is a little larger than the traditional method. For the future work, we would manage to recognize a

more accurate region by *FastNBL* algorithm, then embed it into molecular dynamics applications, and further optimize the performance of the molecular dynamics simulation process on large-scale supercomputers, such as Sunway TaihuLight or Tianhe-2.

Acknowledgements This work was supported by National Natural Science Foundation of China under Grant Nos. 61502450 and 61432018; National Key R&D Program of China under Grant Nos. 2017YFB0202302 and 2016YFB0200800; State Key Laboratory of Computer Architecture Foundation under Grant No. CARCH3504.

References

1. Abraham MJ, Murtola T, Schulz R, Páll S, Smith JC, Hess B, Lindahl E (2015) Gromacs: high performance molecular simulations through multi-level parallelism from laptops to supercomputers. SoftwareX 1:19–25
2. Allen MP, Tildesley DJ (2017) Computer simulation of liquids. Oxford University Press, Oxford
3. Allen MP et al (2004) Introduction to molecular dynamics simulation. Comput Soft Matter Synth Polym proteins 23:1–28
4. Andersen HC (1983) Rattle: a velocity version of the shake algorithm for molecular dynamics calculations. J Comput Phys 52(1):24–34
5. Anderson JA, Glotzer SC (2013) The development and expansion of hoomd-blue through six years of gpu proliferation. arXiv:1308.5587
6. Anderson JA, Lorenz CD, Travesset A (2008) General purpose molecular dynamics simulations fully implemented on graphics processing units. J Comput Phys 227(10):5342–5359
7. ARM (2015) ARM Cortex-A Series: Programmer's Guide for ARMv8-A
8. ARM (2017) ARM architecture reference manual. ARMv8, for ARMv8-A architecture profile
9. Blumenthal LM (1970) Theory and applications of distance geometry. Chelsea, New York
10. Brown WM, Wang P, Plimpton SJ, Tharrington AN (2011) Implementing molecular dynamics on hybrid high performance computers-short range forces. Comput Phys Commun 182(4):898–911. https://doi.org/10.1016/j.cpc.2010.12.021
11. Dagum L, Menon R (1998) OpenMP: an industry standard API for shared-memory programming. IEEE Comput Sci Eng 5(1):46–55
12. Flur S, Gray KE, Pulte C, Sarkar S, Sezgin A, Maranget L, Deacon W, Sewell P (2016) Modelling the ARMv8 architecture, operationally: concurrency and ISA. In: ACM SIGPLAN notices, vol 51. ACM, pp 608–621
13. Fog A (2008) Optimizing subroutines in assembly language: an optimization guide for x86 platforms. Copenhagen University College of Engineering
14. Frenkel D, Smit B (2001) Understanding molecular simulation: from algorithms to applications, vol 1. Elsevier, London
15. Fu H, Liao J, Yang J, Wang L, Song Z, Huang X, Yang C, Xue W, Liu F, Qiao F et al (2016) The sunway taihulight supercomputer: system and applications. Sci China Inf Sci 59(7):072001
16. Glaser J, Nguyen TD, Anderson JA, Lui P, Spiga F, Millan JA, Morse DC, Glotzer SC (2015) Strong scaling of general-purpose molecular dynamics simulations on gpus. Comput Phys Commun 192:97–107
17. Hockney RW, Eastwood JW (1988) Computer simulation using particles. CRC Press, London
18. Intel (2018) Intel® 64 and IA-32 architectures software developers manual. Volume 3B: System programming Guide, Part 2
19. Jiang W, Hardy DJ, Phillips JC, MacKerell AD Jr, Schulten K, Roux B (2010) High-performance scalable molecular dynamics simulations of a polarizable force field based on classical drude oscillators in namd. J Phys Chem Lett 2(2):87–92
20. Liu W, Schmidt B, Voss G, Müller-Wittig W (2007) Molecular dynamics simulations on commodity GPUs with CUDA. In: International Conference on High-Performance Computing. Springer, pp 185–196

21. Mattson W, Rice BM (1999) Near-neighbor calculations using a modified cell-linked list method. Comput Phys Commun 119(2–3):135–148
22. Niethammer C, Becker S, Bernreuther M, Buchholz M, Eckhardt W, Heinecke A, Werth S, Bungartz HJ, Glass CW, Hasse H et al (2014) ls1 mardyn: the massively parallel molecular dynamics code for large systems. J Chem Theory Computation 10(10):4455–4464
23. Plimpton S (1995) Fast parallel algorithms for short-range molecular dynamics. J Comput Phys 117(1):1–19. https://doi.org/10.1006/jcph.1995.1039
24. Plimpton S, Crozier P, Thompson A (2007) LAMMPS-large-scale atomic/molecular massively parallel simulator. Sandia Natl Lab 18:43–43
25. Potter D (1973) Computational physics. Wiley
26. Pronk S, Páll S, Schulz R, Larsson P, Bjelkmar P, Apostolov R, Shirts MR, Smith JC, Kasson PM, Van Der Spoel D et al (2013) Gromacs 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. Bioinformatics 29(7):845–854
27. Rapaport DC, Rapaport DCR (2004) The art of molecular dynamics simulation. Cambridge University Press, Cambridge
28. Stone JE, Phillips JC, Freddolino PL, Hardy DJ, Trabuco LG, Schulten K (2007) Accelerating molecular modeling applications with graphics processors. J Comput Chem 28(16):2618–2640
29. Tang YH, Karniadakis GE (2014) Accelerating dissipative particle dynamics simulations on gpus: algorithms, numerics and applications. Comput Phys Commun 185(11):2809–2822. https://doi.org/10.1016/j.cpc.2014.06.015
30. Trott CR (2011) Lammpscuda—a new gpu accelerated molecular dynamics simulations package and its application to ion-conducting glasses. Ph.d. thesis, Universitätsbibliothek Ilmenau
31. Tuckerman M, Berne BJ, Martyna GJ (1992) Reversible multiple time scale molecular dynamics. J Chem Phys 97(3):1990–2001
32. Verlet L (1967) Computer "experiments" on classical fluids. I. Thermodynamical properties of Lennard–Jones molecules. Phys Rev 159(1):98
33. Wang X, Xue W, Liu W, Wu L (2018) swSpTRSV: a fast sparse triangular solve with sparse level tile layout on sunway architectures. In: Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming. ACM, pp 338–353
34. Xing HJ, Khan MKR, Alnatsheh RH, Chirala RC, Bhattacharjee S (2012) Method and apparatus for neighbor list updates. US Patent 8,144,662
35. Yao Z, Wang JS, Liu GR, Cheng M (2004) Improved neighbor list algorithm in molecular simulations using cell decomposition and data sorting method. Comput Phys Commun 161(1–2):27–35