

Efficient Implementations of Molecular Dynamics Simulations for Lennard-Jones Systems

Hiroshi WATANABE,^{1,2,*} Masaru SUZUKI³ and Nobuyasu ITO⁴

¹*Institute for Solid State Physics, The University of Tokyo,
Kashiwa 277-8581, Japan*

²*Information Technology Center, The University of Tokyo, Tokyo 113-8558, Japan*

³*Department of Applied Quantum Physics and Nuclear Engineering,
Kyushu University, Fukuoka 819-0395, Japan*

⁴*Department of Applied Physics, School of Engineering, The University of Tokyo,
Tokyo 113-8656, Japan*

(Received January 24, 2011; Revised June 24, 2011)

Efficient implementations of the classical molecular dynamics (MD) method for Lennard-Jones particle systems are considered. Not only general algorithms but also techniques that are efficient for some specific CPU architectures are also explained. A simple spatial-decomposition-based strategy is adopted for parallelization. By utilizing the developed code, benchmark simulations are performed on a HITACHI SR16000/J2 system consisting of IBM POWER6 processors which are 4.7 GHz at the National Institute for Fusion Science (NIFS) and an SGI Altix ICE 8400EX system consisting of Intel Xeon processors which are 2.93 GHz at the Institute for Solid State Physics (ISSP), the University of Tokyo. The parallelization efficiency of the largest run, consisting of 4.1 billion particles with 8192 MPI processes, is about 73% relative to that of the smallest run with 128 MPI processes at NIFS, and it is about 66% relative to that of the smallest run with 4 MPI processes at ISSP. The factors causing the parallel overhead are investigated. It is found that fluctuations of the execution time of each process degrade the parallel efficiency. These fluctuations may be due to the interference of the operating system, which is known as OS Jitter.

Subject Index: 020

§1. Introduction

The classical molecular dynamics (MD) simulation was first performed by Alder and Wainwright.¹⁾ The original MD was used to investigate the hard-particle system, and an event-driven algorithm was adopted for the time evolution, which was soon followed by a time-step-driven algorithm.²⁾ Owing to the recent increase in computational power, MD is now a powerful tool for studying not only the equilibrium properties but also the nonequilibrium transportation phenomena of molecular systems³⁾⁻⁶⁾ as well as biomolecular systems.^{7),8)} Electronic degrees of freedom can also be considered by ab initio methods on the basis of quantum mechanics.⁹⁾ The increase in computational power also allows us not only to use more realistic and complex interactions but also to treat a larger number of particles. The recent increase in computational power has mainly been achieved by increasing the number of processing cores. This paradigm is called massively parallel processing (MPP), and

*) E-mail: hwatanabe@issp.u-tokyo.ac.jp

most recent high-end computers have been built on the basis of this paradigm.¹⁰⁾ The number of cores is typically from several thousand to several hundred thousand. In an MPP system, a single task is performed by a huge number of microprocessors, which operate simultaneously and communicate with each other as needed. Therefore, parallelization is now unavoidable in order to utilize the computational power of such machines effectively. While parallelization itself is of course important, the tunings for a single core have become more important for huge-scale and long-time simulations, since the overall performance of MD mainly depends on the performance on the single cores. In particular, optimizing memory access is important. The latency and bandwidth of memory access are sometimes slow compared with the execution speed of processors, and the data supply often cannot keep up with the requests by the processors. Therefore, data must be suitably arranged so that necessary data is located near a processor. Tuning for specific architectures is also important to achieve high performance. Since design concepts differ from one processor to another, it is necessary to prepare codes for each architecture, at least for the hot spot of the simulation where most time is spent during the execution, which is usually force calculation in MD.

The Next-Generation Supercomputer project is currently being carried out by RIKEN.¹¹⁾ The system is designed on the basis of the MPP paradigm and will use scalar CPUs with 128 GFlops, will consist of over 80 000 nodes, and is expected to achieve a total computational power of 10 petaflops. This supercomputer is now under development and will be ready in 2012. Therefore, we must prepare parallelized codes that can utilize the full performance of this system by then. The parallelization of MD has been discussed over several decades with the aims of treating larger systems and performing simulations for longer timescales,¹²⁾ and a huge MD simulation involving one trillion particles has been performed on BlueGene/L, which consists of 212 992 processors.¹³⁾ Despite this success, it will not be easy to obtain a satisfactory performance on the Next-Generation Supercomputer since the properties of the systems are considerably different from those of BlueGene. BlueGene has relatively slow processors with a 700 MHz clock speed for BlueGene/L, and consequently, it achieves a good balance between the processor speed and the memory bandwidth. In comparison, the Next-Generation Supercomputer will have a relatively high peak performance of 128 GFlops per CPU (16 GFlops for each core and each CPU consists of eight cores). Therefore, the memory bandwidth and latency may be a significant bottleneck.

The purpose of the present article is to describe efficient algorithms and their implementation for the MD method, particularly focusing on the memory efficiency and the parallel overheads. We also provide some tuning techniques for a couple of the specific architectures. While the Lennard-Jones potential is considered throughout the manuscript, the techniques described here are generally applicable to other simulations with more complicated potentials. This manuscript is organized as follows. In §2, algorithms for finding interacting particle pairs are described. Some further optimization techniques whose efficiency depends on architecture of the computer are explained in §3. Parallelization schemes and the results of benchmark simulations are described in §4. The factors causing the parallel overhead for the developed

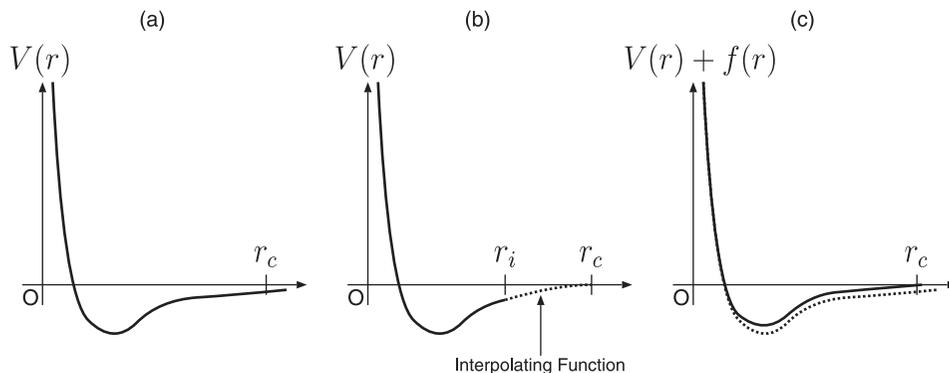


Fig. 1. Truncation of the Lennard-Jones potential function. (a) Original form. The interaction range extends to infinity. (b) Truncation by interpolation. An interpolating function is introduced for the region from the interpolation point r_i to the truncation point r_c . (c) Truncation by adding extra terms. The additional function $f(r)$ is chosen so that both the value and the derivatives of $V(r) + f(r)$ are zero at the truncation point.

MD code are discussed in §5, and a summary and a discussion of further issues are given in §6.

§2. Pair list construction

2.1. Truncation of potential function

The common expression for the Lennard-Jones potential is

$$V(r) = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^6 \right], \quad (2.1)$$

for the particle distance r , well depth ϵ , and atomic diameter σ . In the following, we will use the physical quantities reduced by σ , ϵ , and k_B , for example., the length scale is measured in the unit of σ , and so forth. The first term of the right-hand side of Eq. (2.1) describes the repulsive force, and the other term describes the attractive force. This potential form has been widely used for a standard particle model involving phase transitions since this system exhibits three phases: solid, liquid, and gas. While the original form of the Lennard-Jones potential spans infinite range, it is wasteful to consider an interaction between two particles at a long distance since the potential decays rapidly as the distance increases. To reduce computation time, truncated potentials are usually used instead of the original potential.³⁾ There are several ways of introducing truncation. One way is to use cubic spline interpolation for the region from some distance to the cutoff length^{14),15)} as shown in Fig. 1 (b). In this scheme, there are two special distances, the interpolating distance r_i and the truncation distance r_c . The potential is expressed by a piecewise-defined function that switches from $V(r)$ given in Eq. (2.1) to an interpolating function $V_i(r)$ at $r = r_i$. The interpolation function is chosen so that values and the derivatives of the potential are continuous at the interpolation and the truncation points. These conditions require four adjustable parameters, and therefore, a third-order polynomial

is usually chosen as the interpolating function. While this interpolation scheme was formerly popular, it is now outdated since it involves a conditional branch, which is sometimes expensive for current CPU architectures.

Nowadays, truncation is usually introduced by adding some extra terms to the potential. At the truncation point, both the potential and the force should become continuous, and therefore, at least two additional terms are necessary. One such potential is given by¹⁶⁾

$$V(r) = 4\varepsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^6 + c_2 \left(\frac{r}{\sigma} \right)^2 + c_0 \right], \quad (2.2)$$

with two additional coefficients c_2 and c_0 , which are determined so that $V(r_c) = V'(r_c) = 0$ for the cutoff length r_c . Here, we use a quadratic term instead of a linear term such as $c_1 r + c_0$, since the latter involves a square-root operation in the force calculation, which is sometimes expensive. While the computational cost of the force computation is more expensive than that for the interpolating method, this method is usually faster than the interpolating method since conditional branches can be hazards in the pipeline that obstruct the instruction stream. If accuracy is less crucial, one can truncate the potential by adding one constant, i.e., $c_2 = 0$ and c_0 is determined by $V(r_c) = 0$. In this case, the calculation of force is identical to that using the original potential and the modification only appears in the calculation of the potential energy. Note that this scheme sometimes involves some problems regarding the conservation of energy since the force is not continuous at the truncation point. The truncation changes the phase diagram. One should check the phase diagram and compute phase boundaries for each truncated potential before performing production runs. In particular, the gas-liquid coexistence phase becomes drastically narrower as the cutoff length decreases. To investigate phenomena involving the gas-liquid phase transition, the cutoff length should be made longer than 3.0σ .

2.2. Grid search method

To perform the time evolution of a system, we first have to find particle pairs such that the distance between the particles is less than the cutoff length. Since the trivial computation leads to $O(N^2)$ computation, where N is the number of particles, we adopt a grid algorithm, which is also called the linked-list method,^{17),18)} to reduce the complexity of the computation to $O(N)$. The algorithm finds interacting pairs within a grid by the following three steps: (i) divide a system into small cells. (ii) register the indices of particles on cells to which they belong, and (iii) search for interacting particle pairs using the registered information. There are two types of grid, one is exclusive and the other is inexclusive (see Fig. 2). An exclusive grid allows only one particle to occupy a cell,¹⁹⁾ and the inexclusive grid allows more than one particle to occupy one cell simultaneously.^{3),15),20),21)} Generally, an exclusive grid is efficient for a system with short-range interactions such as hard particles, and an inexclusive grid is suitable for a system with relatively long interactions such as Lennard-Jones particles. However, the efficiency strongly depends not only on the physical properties such as density, polydispersity, and interaction length but also on the hardware architecture such as the cache size and memory bandwidth. Therefore,

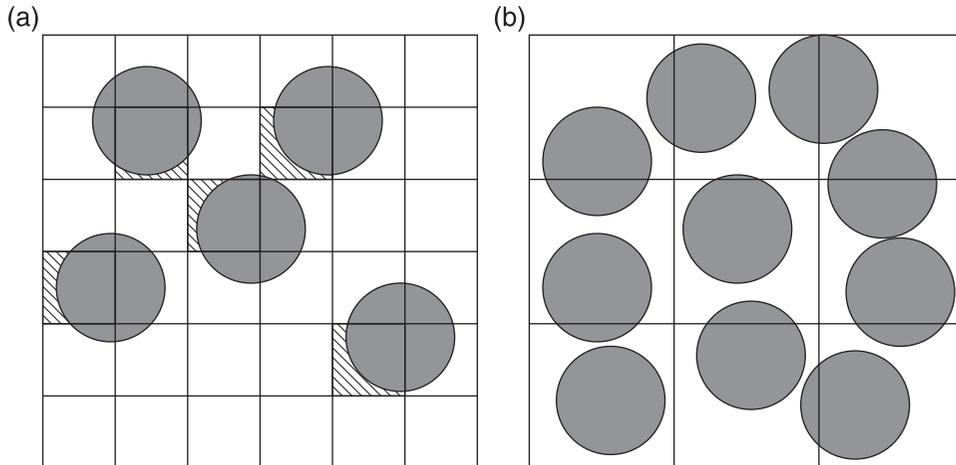


Fig. 2. Searching for interacting pairs in a grid. (a) Exclusive grid. The size of each cell is determined so that only one particle can be placed in each cell. (b) Inexclusive grid. The interaction length determines the size of cells. More than one particle is allowed to occupy one cell simultaneously.

it is difficult to determine which is better before implementation. We have found that an inexclusive grid is more efficient than an exclusive grid for the region in which a Lennard-Jones system involves a gas-liquid phase transition. Therefore, we adopt an inexclusive grid in the following. Note that the length of each cell should be larger than the search length r_s , which is longer than the cutoff length as described later, so that there are no interactions occurring through three or more cells, in other words, interacting particle pairs are always in the same cell or adjacent cells.

A simple way to express the grid information is to use multidimensional arrays. Two types of array are necessary, one is for the number of particles in each cell, and the other is for the indices of the particles in each cell. Suppose the total number of cells is $n_g = n_{gx} \times n_{gy} \times n_{gz}$, then the arrays can be expressed by

$$\begin{aligned} \text{integer} &: \text{GridParticleNumber}[n_{gx}][n_{gy}][n_{gz}], \\ \text{integer} &: \text{GridIndex}[n_{gx}][n_{gy}][n_{gz}][g_{\max}], \end{aligned} \quad (2.3)$$

where g_{\max} is the capacity of one cell. The number of particles in a cell at (x, y, z) is stored as $\text{GridParticleNumber}[x][y][z]$ and the index of the i -th particle at the cell is stored as $\text{GridIndex}[x][y][z][i]$. Although this scheme is simple, the required memory can be dozens of times larger than the total number of particles in the system, and therefore, the indices of the particles are stored sparsely, which causes a decrease in cache efficiency. To improve the efficiency of memory usage, the indices of particles should be stored sequentially in an array. This method is proposed by Kadau et al. and implemented in SPaSM.²²⁾ An algorithm to construct grid information using an array whose size is the total number of particles N is as follows.

- 1: Divide a system into small cells, and assign a serial number to each cell.
- 2: Label particles with the serial numbers of the cells that they belong to.

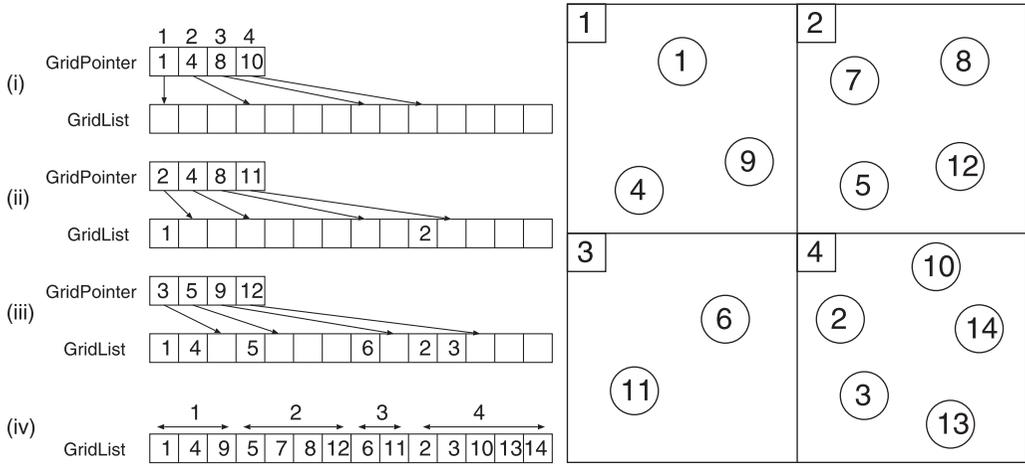


Fig. 3. How to construct GridList, which is a linear array of size N storing grid information. Suppose there are four cells and 14 particles in a system ($N = 14$). The left figure shows the stored data in the array and the right figure shows the configuration of the particles in the system. The numbers with arrows denote pointers indicating where the indices of particles should be stored (GridPointer). The particles have indices from 1 to 14 and the cells are labeled from 1 to 4. First, prepare an array of size N , and label particles with the serial number of the cells to which they belong. (i) Count the number of particles in each cell, store the number as GridParticleNumber, and set a pointer for each cell at the position where the index of the first particle of each grid should be stored when the particles are sorted by their labels. (ii) Place the index of a particle at the position where the pointer corresponding to its label points, and move the pointer to the right. (iii) Repeat this procedure for all particles. The status of the GridList just after the particle number 6 is located into the GridList. (iv) The completed array after above (i) – (iii) procedures. The particles are sorted in GridList in order of the grid labels.

3: Count the number of particles in each cell.

4: Sort the particles by their labels.

The details of the implementation are shown in Fig. 3. Suppose that i -cell denotes the cell labeled with i and the grid information is stored in the linear array GridList. Then GridParticleNumber[i] denotes the number of particles in i -cell and GridPointer[i] denotes the position of which the first particle of i -cell should be stored in GridList. The complexity of this sorting process is $O(N)$ instead of $O(N \log N)$ for usual Quicksort since it does not perform a complete sorting. The memory usage is effective since the indices of particles in each grid are stored contiguously in the array.

2.3. Bookkeeping method

After constructing GridList, we have to find interacting particle pairs using the grid information. Suppose the particle pairs whose distance is less than the cutoff length are stored in PairList. PairList is an array of pairs which hold two indices of interacting particles. All particles are registered in cells and labeled by the serial numbers of the cells. There are two cases, interacting particles share or do not share the same label, which correspond to inner-cell interaction and cell-crossing

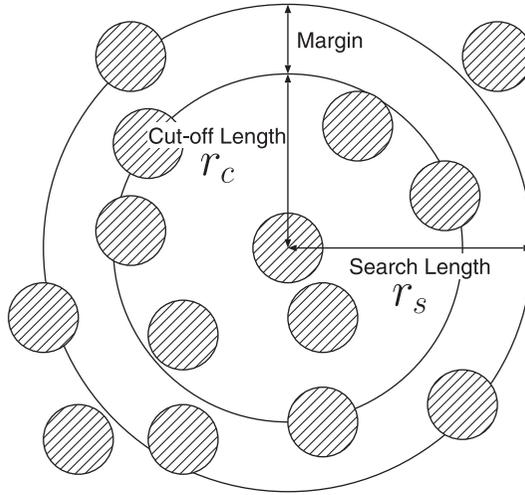


Fig. 4. Cutoff length and search length. The search length r_s is chosen so that $r_s > r_c$ for the cutoff length r_c . A pair list can be reused unless a particle located outside of search length across half of the margin $r_s - r_c$.

Algorithm 1 Finding the interacting particle pairs

```

1: for all  $i$ -cell do
2:    $s_i \leftarrow \text{GridPointer}[i]$ 
3:    $e_i \leftarrow s_i + \text{GridParticleNumber}[i] - 1$ 
4:    $L_c \leftarrow \text{GridList}[s_i..e_i]$ 
5:   for all  $j$  such that  $j$ -cell is the neighbor of  $i$ -cell do
6:      $s_j \leftarrow \text{GridPointer}[j]$ 
7:      $e_j \leftarrow s_j + \text{GridParticleNumber}[j] - 1$ 
8:     Append  $\text{GridList}[s_j..e_j]$  to  $L_c$ 
9:   end for
10:  for  $k = 1$  to  $\text{GridParticleNumber}[i]$  do
11:    for  $l = k + 1$  to  $|L_c|$  do
12:       $m \leftarrow L_c[k]$ 
13:       $n \leftarrow L_c[l]$ 
14:      if  $m$ - and  $n$ -particles interact each other then
15:        Register a pair  $(m, n)$  to PairList
16:      end if
17:    end for
18:  end for
19: end for

```

interaction. Let $\text{GridList}[s..e]$ denote a subarray of GridList whose range is from s to e with the data of the pair list stored in PairList . The pseudocodes used to find interacting pairs are shown in Algorithm 1. Only 13 of the 26 neighbors should be checked for each cell because of Newton's third law.

As the number of particles increases, the cost of constructing the pair list in-

creases drastically since it involves frequent access to memory. While it depends on the details of the system, constructing a pair list is sometimes dozens of times more expensive than the force calculation. In order to reduce the cost of constructing a pair list, a well-known bookkeeping method was proposed.²³⁾ The main idea of the bookkeeping method is to reuse the same pair list for several time steps by registering pairs within a search length r_s which is set at longer than the cutoff (interaction) length r_c (see Fig. 4). The margin $r_s - r_c$ determines the lifetime of a pair list. While a longer margin gives a longer lifetime, the length of the pair list also increases, and consequently, the computational time for the force calculation becomes longer. Therefore, there is an optimal length for the margin that depends on the details of the system, such as its density and temperature. Several steps after the construction of the pair list, the list can become invalid, i.e., some particle pair that is not registered in the list may be at a distance shorter than the interaction length r_c . Therefore, we have to calculate the expiration time of the list and can check the validity of the pair list for each step as the following simple method. The expiration time t_e of a pair list constructed at time t is given by

$$t_e = \frac{r_s - r_c}{2v_{\max}} + t, \quad (2.4)$$

where v_{\max} is the absolute value of the velocity of the fastest particle in the system. The factor 2 in the denominator corresponds to the condition that two particles undergo a head-on collision. It is necessary to update the expiration time t_e when the maximum velocity changes to

$$t_e \leftarrow (t_e - t) \frac{v_{\text{old}}}{v_{\text{new}}} + t, \quad (2.5)$$

where v_{old} and v_{new} denote the previous and current maximum speeds, respectively. Obviously, the expiration time will be brought forward in the case of a faster particle, and vice versa. This technique for calculating the expiration time is called the dynamical upper time cutoff (DUTC) method and was proposed by Isobe.²⁴⁾

Although the validity check by considering the maximum velocity is simple, the expiration time t_e determined by the velocities of the particles is shorter than the *actual* expiration time of the pair list, since the pair list is valid as long as the particles migrate short distances. Therefore, we propose a method which extends the lifetime of a pair list by utilizing the displacements of the particles. The strict condition for the expiration of a pair list is that there exists a particle pair (i, j) such that

$$|\mathbf{q}_i^{\text{old}} - \mathbf{q}_j^{\text{old}}| > r_s \quad \text{and} \quad |\mathbf{q}_i - \mathbf{q}_j| < r_c, \quad (2.6)$$

where $\mathbf{q}_i^{\text{old}}$ is the position of the i -particle when the pair list was constructed and \mathbf{q}_i is the current position of the i -particle. The lifetime of a pair list can be extended if we use the strict condition for the validity check, but the computational cost of checking this condition is $O(N^2)$. We therefore extend the lifetime of the pair list by using a less strict condition. The main idea is to prepare a list of particles that have moved by large amounts and check the distance between all pairs in the list. The algorithm is as follows: (i) Calculate the displacements of the particles from

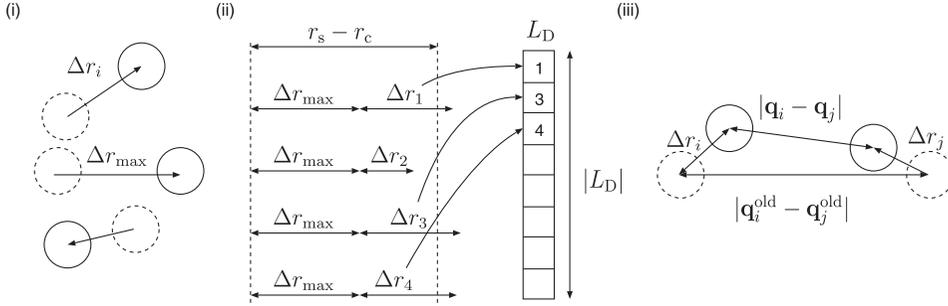


Fig. 5. The large displacement check (LDC) method. (i) Calculate the displacements Δr_i for the particle i and find the largest displacement Δr_{\max} . (ii) Register a index of a particle i such that $\Delta r_i > r_s - r_c - \Delta r_{\max}$ to a list L_D whose size is denoted by $|L_D|$. (iii) Old and current position of particles i and j . The dashed circles denote the old position which was registered in the previous construction of the pair list. The solid circles denote the current positions of particles. The expiration of the pair list is determined by the old and the current distance of the particles.

their position when the pair list was constructed and find the largest displacement Δr_{\max} . (ii) If there exists a particle such that $\Delta r_i > r_s - r_c - \Delta r_{\max}$, then append the particle to the list of large displacement L_D which size is denoted by $|L_D|$. If the list is longer than the predetermined maximum length of the list N_l , that is, $|L_D| > N_l$, then the pair list is invalidated. (iii) If the old distance is longer than the search length and the current distance is shorter than the cutoff length, it means that there are interacting particles which are not registered in a pair list. Therefore, the pair list is invalidated if there exists a pair (i, j) in the list of large displacement such that $|\mathbf{q}_i^{\text{old}} - \mathbf{q}_j^{\text{old}}| > r_s$ and $|\mathbf{q}_i - \mathbf{q}_j| < r_c$. We call this method the large displacement check (LDC) method. See Fig. 5 for a schematic description of the LDC method.

Although the lifetime of the pair list increases with N_l , the computational cost also increases as $O(N_l^2)$. Therefore, there is an optimal value for N_l . Since the LDC method is more expensive than DUTC method, we adopt a hybrid algorithm involving both methods, that is, first we use the DUTC method for the validity check, and then we use LDC method after the DUTC method decides that the pairlist is expired. The full procedure of the validity check is shown in Algorithm 2.

2.4. Sorting of pair list

After constructing a pair list, we can calculate the forces between interacting pairs of particles. Hereafter, we call the particle of index i the i -particle for convenience. Suppose the positions of particles are stored in $\mathbf{q}[N]$. A simple method for calculating the force using the pair list is shown in Algorithm 3. This simple algorithm is, however, usually inefficient since it involves random access to the memory. Additionally, it fetches and stores the data of both particles of each pair, which is wasteful. Therefore, we construct a sorted list to improve the efficiency of memory usage by calculating particles together that interact with the same particles. For convenience, we call the lower-numbered particle in a pair the *key* particle and the other the *partner* particle. The *partner* particles with the same *key* particle are grouped together. The detail of the implementation is shown in Fig. 6. Suppose the

Algorithm 2 Checking the validity of a pair list

1. When a pair list is constructed
 - 1: **for all** i -particle **do**
 - 2: $\mathbf{q}_i^{\text{old}} \leftarrow \mathbf{q}_i$ {Keep the positions}
 - 3: **end for**
 - 4: $\text{buffer_length} \leftarrow r_s - r_c$
2. Validity check by DUTC method
 - 1: $v_{\text{max}} \leftarrow$ the maximum velocity of the particle
 - 2: $\text{buffer_length} \leftarrow \text{buffer_length} - 2v_{\text{max}}\Delta t$
 - 3: **if** $\text{buffer_length} > 0$ **then**
 - 4: the pair list is valid.
 - 5: **else**
 - 6: Check the validity by considering displacements.
 - 7: **end if**
3. Validity check by LDC method
 - 1: **for all** i -particle **do**
 - 2: $\Delta r_i \leftarrow |\mathbf{q}_i^{\text{old}} - \mathbf{q}_i|$
 - 3: **end for**
 - 4: $\Delta r_{\text{max}} \leftarrow \max\{\Delta r_i\}$
 - 5: Prepare a list L_D
 - 6: **for all** i -particle **do**
 - 7: **if** $\Delta r_i > r_s - r_c - \Delta r_{\text{max}}$ **then**
 - 8: Append i -particle to L_D
 - 9: **if** $|L_D| > N_i$ **then**
 - 10: the pair list is invalidated.
 - 11: **end if**
 - 12: **end if**
 - 13: **end for**
 - 14: **for all** (i, j) in L_D **do**
 - 15: **if** $|\mathbf{q}_i^{\text{old}} - \mathbf{q}_j^{\text{old}}| > r_s$ and $|\mathbf{q}_i - \mathbf{q}_j| < r_c$ **then**
 - 16: the pair list is invalidated.
 - 17: **end if**
 - 18: **end for**
 - 19: the pair list is invalidated.

data of the sorted list and the position of the first *partner* particle of the i -particle in the list are stored in SortedList and KeyPointer[i], respectively. Then the algorithm used to calculate the force using the sorted pair list is shown in Algorithm 4. The variables \mathbf{q}_{key} and \mathbf{f}_{key} are intended to be stored in CPU registers. The amount of memory access decreases compared with that for Algorithm 3, since the fetching and storing are performed only for the data of *partner* particles in the inner loop. The amount of memory access becomes half when the *key* particles have an enough number of *partner* particles. The force acting on the *key* particle is accumulated

Algorithm 3 Calculating the force in a simple manner

```

1: for all pairs  $(i, j)$  in PairList do
2:   if  $|\mathbf{q}[i] - \mathbf{q}[j]| < r_c$  then
3:     Calculate force between  $i$ - and  $j$ -particles.
4:   end if
5: end for

```

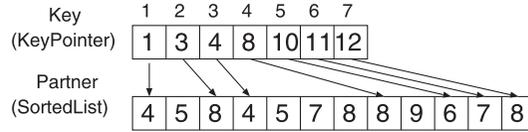


Fig. 6. A sorted list. A list sorted by the indices of *key* particles is constructed by the following procedure. (i) Count the number of *partner* particles for each *key* particle for the particle pairs are stored in the pair list. (ii) Prepare a linear array of size N_{pair} , and set a pointer for each *key* particle where the index of the first *partner* particle will be stored. (iii) Similarly to in the grid construction (Fig. 3), place the index of a particle at the pointer of its *key* particle and move the position of the pointer to the next position. (iv) Repeat this procedure for all pairs to construct the sorted list. The figure shows two lists, KeyPointer and SortedList. KeyPointer[i] stores the first position of the interacting particles in SortedList.

Algorithm 4 Calculating the force using the sorted pair list

```

1: for  $i = 1$  to  $N - 1$  do
2:    $\mathbf{q}_{\text{key}} \leftarrow \mathbf{q}[i]$ 
3:    $\mathbf{f}_{\text{key}} \leftarrow 0$ 
4:   for  $k = \text{KeyPointer}[i]$  to  $\text{KeyPointer}[i + 1] - 1$  do
5:      $j \leftarrow \text{SortedList}[k]$ 
6:      $r \leftarrow |\mathbf{q}_{\text{key}} - \mathbf{q}[j]|$ 
7:      $f \leftarrow -V'(r)$ 
8:     Update momenta of the  $j$ -particle using  $f$ .
9:     Update  $\mathbf{f}_{\text{key}}$  using  $f$ .
10:  end for
11:  Update momenta of  $i$ -particle using  $\mathbf{f}_{\text{key}}$ .
12: end for

```

during the inner loop, and the momentum of the *key* particle is updated at the end of the inner loop. Note that the loop variable i takes values from 1 to $N - 1$, instead of 1 to N , since this is the loop for the *key* particle, and the N -particle cannot be a *key* particle. The value of KeyPointer[N] should be $|\text{PairList}| + 1$, where $|\text{PairList}|$ is the number of pairs stored in PairList.

Algorithm 5 Calculating the force with if-branch elimination

```

1: for  $k = 1$  to  $N_{\text{pair}}$  do
2:    $(i, j) \leftarrow \text{PairList}[k]$ 
3:    $r \leftarrow |\mathbf{q}[i] - \mathbf{q}[j]|$ 
4:    $f \leftarrow -V'(r)$ 
5:   if  $r \geq r_c$  then
6:      $f \leftarrow 0$ 
7:   end if
8:   Update momenta of  $i$ - and  $j$ -particles using  $f$ .
9: end for

```

§3. Further optimization*3.1. Elimination of conditional branch*

Since we adopt the bookkeeping method, a pair list contains particle pairs whose distances are longer than the interaction length. Therefore, it is necessary to check whether or not the distance between each pair is less than the interaction length before the force calculation (Algorithm 3). These conditional branches (so-called, “if then else” of “case” in the high-level language statement) can sometimes be expensive for RISC(Reduced Instruction Set Computer)-based processors such as IBM POWER6, and it causes a significant decrease in execution speed. To prevent the decrease in speed due to conditional branches, we calculate the forces without checking the distance and assign zero to the calculated value for the particle pairs whose distances are longer than the interaction length. This method is shown in Algorithm 5. While it appears to be wasteful, it can be faster than the original algorithm when the penalty due to the conditional branches is more expensive than the additional cost of calculating forces. Additionally, this loop can be performed for all pairs, which makes it easy for a compiler to optimize codes, such as that for prefetching data. Note that, some CPU architectures prepare the instruction for the conditional substitution. For example, PowerPC architecture has `fsel` (Floating-Point Select) instruction which syntax is `fsel FRT, FRA, FTC, FRB` where `FRT`, `FRA`, `FTC`, and `FRB` are registers. If `FRA > 0` then `FRT = FRB`, otherwise `FRT = FRC`. We found that this optimization provides us with more than double the execution speed on IBM POWER6 in the benchmark simulation whose details are described in §4.5.

3.2. Reduction of divisions

The explicit expression of Algorithm 3 is shown in Algorithm 6, where c_2 is the coefficient for the truncation defined in Eq. (2-2) and dt is the time step. For simplicity, the diameter of the particle σ is set to unity and only the force calculations are shown. As shown in Algorithm 6, the force calculation of the Lennard-Jones potential involves at least one division operation for each pair. Divisions are sometimes expensive compared with multiplications. Therefore, computations can be made faster by reducing the number of divisions. Suppose there are two independent divisions as follows;

Algorithm 6 Explicit expression of force calculation

```

1: for  $k = 1$  to  $N_{\text{pair}}$  do
2:    $(i, j) \leftarrow \text{PairList}[k]$ 
3:    $\mathbf{r} \leftarrow \mathbf{q}[j] - \mathbf{q}[i]$ 
4:    $r^2 \leftarrow |\mathbf{r}|^2$ 
5:    $r^6 \leftarrow r^2 \times r^2 \times r^2$ 
6:    $r^{14} \leftarrow r^6 \times r^6 \times r^2$ 
7:    $f dt \leftarrow [(24r^6 - 48)/r^{14} + 8c_2] \times dt$ 
8:    $\mathbf{p}[i] \leftarrow \mathbf{p}[i] + f dt \times \mathbf{r}$ 
9:    $\mathbf{p}[j] \leftarrow \mathbf{p}[j] - f dt \times \mathbf{r}$ 
10: end for

```

$$A_1 \leftarrow 1/B_1,$$

$$A_2 \leftarrow 1/B_2.$$

We can reduce the number of divisions by transforming them into

$$C \leftarrow 1/(B_1 \times B_2),$$

$$A_1 \leftarrow C \times B_2,$$

$$A_2 \leftarrow C \times B_1.$$

Here, the number of divisions decreases from two to one, while three multiplications appear. This optimization can be effective for architecture in which the penalty for divisions is large. We apply this optimization technique to the force calculation of the Lennard-Jones system by utilizing loop unrolling, that is, we first obtain two independent divisions by unrolling and then we apply this method of reducing the number of divisions to the unwound loop. Although one can apply this optimization technique to Algorithm 6, it is more efficient to use it together with the sorted array described in §2.4. The algorithm for the force calculation with a reduced number of divisions is shown in Algorithm 7. The term $\lfloor x \rfloor$ denotes the largest integer less than or equal to x and subscriptions a and b correspond to the loop unrolled twice. This optimization achieves 10% speedup in the condition described in §4.5.

3.3. Cache efficiency

As particles move, the indices of the partner particles of each key-particle become random. Then the data of the interacting particles, which are spatially close, may be widely separated in memory. This severely decreases the computational efficiency since the data of particles that are not stored in the cache are frequently required. To improve the situation, reconstruction of the order of the particle indices in arrays is proposed.²⁵⁾ This involves sorting so that the indices of interacting particles are as close together as possible. This can be implemented by using the grid information described in §2.2. After an array of grid information is constructed, the particles are arranged in the order in the array. The sorting algorithm is shown in Fig. 7. After sorting, the indices of the particles in the same cell become sequential, which improves the cache efficiency. Note that the pair list is made invalid by this procedure; therefore, the pair list construction and the sorting should be performed at the same time.

Algorithm 7 Calculating the force using the sorted array and reduction of divisions

```

1: for  $i = 1$  to  $N - 1$  do
2:    $\mathbf{q}_{\text{key}} \leftarrow \mathbf{q}[i]$ 
3:    $\mathbf{f}_{\text{key}} \leftarrow 0$ 
4:    $n \leftarrow \text{KeyPointer}[i + 1] - \text{KeyPointer}[i]$ 
5:   for  $k = 1$  to  $\lfloor n/2 \rfloor$  do
6:      $k_a \leftarrow \text{KeyPointer}[i] + 2(k - 1)$ 
7:      $k_b \leftarrow \text{KeyPointer}[i] + 2(k - 1) + 1$ 
8:      $j_a \leftarrow \text{SortedList}[k_a]$ 
9:      $j_b \leftarrow \text{SortedList}[k_b]$ 
10:     $\mathbf{r}_a \leftarrow \mathbf{q}[j_a] - \mathbf{q}_{\text{key}}$ 
11:     $\mathbf{r}_b \leftarrow \mathbf{q}[j_b] - \mathbf{q}_{\text{key}}$ 
12:     $r_a^2 \leftarrow |\mathbf{r}_a|^2$ 
13:     $r_b^2 \leftarrow |\mathbf{r}_b|^2$ 
14:     $r_a^6 \leftarrow r_a^2 \times r_a^2 \times r_a^2$ 
15:     $r_b^6 \leftarrow r_b^2 \times r_b^2 \times r_b^2$ 
16:     $r_a^{14} \leftarrow r_a^6 \times r_a^6 \times r_a^2$ 
17:     $r_b^{14} \leftarrow r_b^6 \times r_b^6 \times r_b^2$ 
18:     $D \leftarrow 1/(r_a^{14} \times r_b^{14})$ 
19:     $f dt_a \leftarrow [(24r_a^6 - 48) \times D \times r_b^{14} + 8c_2] \times dt$ 
20:     $f dt_b \leftarrow [(24r_b^6 - 48) \times D \times r_a^{14} + 8c_2] \times dt$ 
21:     $\mathbf{f}_{\text{key}} \leftarrow \mathbf{f}_{\text{key}} + f dt_a \times \mathbf{r}_a + f dt_b \times \mathbf{r}_b$ 
22:     $\mathbf{p}[j_a] \leftarrow \mathbf{p}[j_a] - f dt_a \times \mathbf{r}_a$ 
23:     $\mathbf{p}[j_b] \leftarrow \mathbf{p}[j_b] - f dt_b \times \mathbf{r}_b$ 
24:  end for
25:  if  $n$  is odd then
26:    calculate force for the last partner particle
27:  end if
28:   $\mathbf{p}[i] \leftarrow \mathbf{p}[i] + \mathbf{f}_{\text{key}}$ 
29: end for

```

A comparison between the computational speed with and without sorting is shown in Fig. 8. The simulations were performed on a Xeon 2.93 GHz machine with 256 KB for an L2 cache and 8 MB for an L3 cache. We performed the sorting every ten constructions of a pair list, which is typically every few hundred time steps, since the computational cost of sorting is not expensive but not negligible. To mimic a configuration after a very long time evolution without sorting, the indices of the particles were completely shuffled at the beginning of the simulation, that is, the labels of the particles were exchanged randomly while keeping their positions and momenta. The computational speed is measured in the unit of MUPS (millions update per second), which is unity when one million particles are updated in one second. As the number of particles increased, the computational efficiency was greatly improved by sorting. The stepwise behavior of the data without sorting reflected the cache hierarchy. Since the system is three dimensional, amount of memory re-

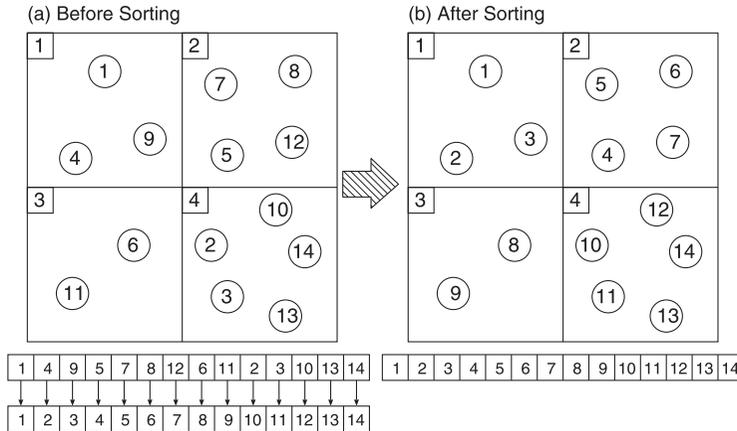


Fig. 7. Sorting the indices of the particles to increase cache efficiency. There are four cells and 14 particles in the system. (a) Status of the system before sorting. The array of grid information corresponding to this configuration is shown below (see Fig. 3). (b) Indices of the particles are sorted using the grid information.

quired to store information of one particle is 48 bytes. Therefore, L2 cache (256 KB) corresponds to $256 \text{ KB}/48 \text{ bytes} \sim 5.3 \times 10^3$ and L3 cache (8MB) corresponds $8 \text{ MB}/48 \text{ bytes} 1.7 \times 10^5$. When the number of particles was smaller than 5.3×10^3 , the difference in speed between the data with and without sorting was moderate. While the number of particles was larger than 5.3×10^3 , we found that the efficiency was improved by 30 to 40% by sorting. When the number of particles was larger than 1.7×10^5 , the computational speed without sorting became much lower than that with sorting. We found that the effect of the sorting in efficiency was about a factor of 3 in the case of 10^6 particles.

3.4. Software pipelining

The elimination of the conditional branch can be inefficient in some types of CPU architectures such as Intel or AMD, since the cost of the conditional branch is not expensive. In such CPU architectures, software pipelining is effective in improving the computational speed. Recent CPUs can handle four or more floating-point operations simultaneously if they are independent. The force calculation of one pair, however, consists of a sequence of instructions that are dependent on each other. Therefore, CPU sometimes has to wait for the previous instruction to be completed, which decreases the computational speed. Software pipelining is a technique to reduce such dependencies between instructions in order to increase the efficiency of the calculation. The purpose of the software pipelining is increasing a number of instructions which can be executed simultaneously, and consequently, increasing the instruction throughputs in a hardware pipeline. While the software-pipelining technique is often used in combination with loop unrolling, the simple loop unrolling can cause a shortage of registers leading to access to the memory which can be expensive. The main idea of software pipelining is to calculate independent parts of the force calculation simultaneously by shifting the order of the instructions. The calculation

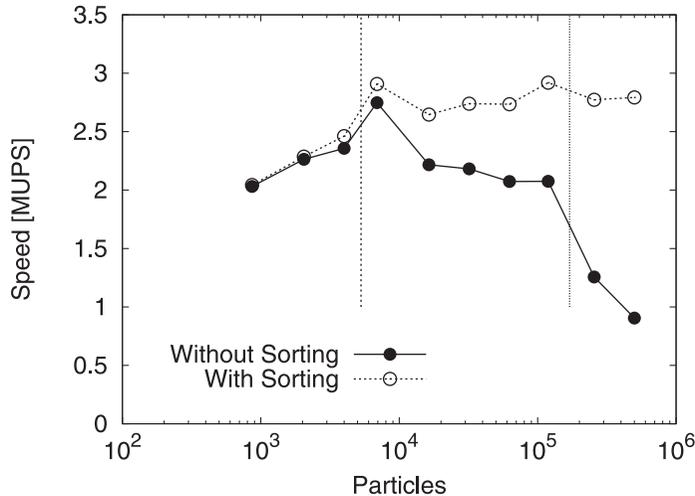


Fig. 8. Computational speed with and without sorting. The speed is greatly improved by sorting for a large number of particles N . Dashed vertical lines denote $N = 5.3 \times 10^3$ and $N \sim 1.7 \times 10^5$. The behavior changes at the lines reflecting the cache hierarchy.

of force can be broken into the following five stages;

DIS calculate distance between two particles.

SQR calculate the square of the distance.

CMP compare the distance and the cutoff length.

CFD calculate the force using the distance.

UMP update the momentum of the particle.

The schematic image of the software pipelining is shown in Fig. 9. The number after a stage denotes the index of the pair (the value of the loop counter), e.g., DIS1 denotes the calculation of the distance of the first particle pair. Figure 9 (a) denotes the calculation without software pipelining. Suppose the distances between third and fifth pairs are longer than the cutoff length. Then the calculation of the force and the update the momentum are unnecessary. We refer this to **NOP** (no operation). All calculations are performed sequentially. For instance, SQR1 is performed after DIS1 is completed, CMP1 is performed after SQR1 is completed, and so on. Figure 9 (a) denotes the calculation with software pipelining. The order of the calculations is shifted so that the number of independent calculations is increased. For instance, DIS4, SQR3, and UMP1 can be performed simultaneously since there are no dependencies between them (see the column where the value of the loop counter is three). Increase of independent instructions increases the efficiency of the hardware pipeline, and consequently, improves the computational speed. The pseudocode corresponding to Fig. 9 is shown in Algorithm 8. As far as we know in our experience, the software pipelining without loop unrolling described here is the best choice for recent architecture such as Intel Xeon processors.

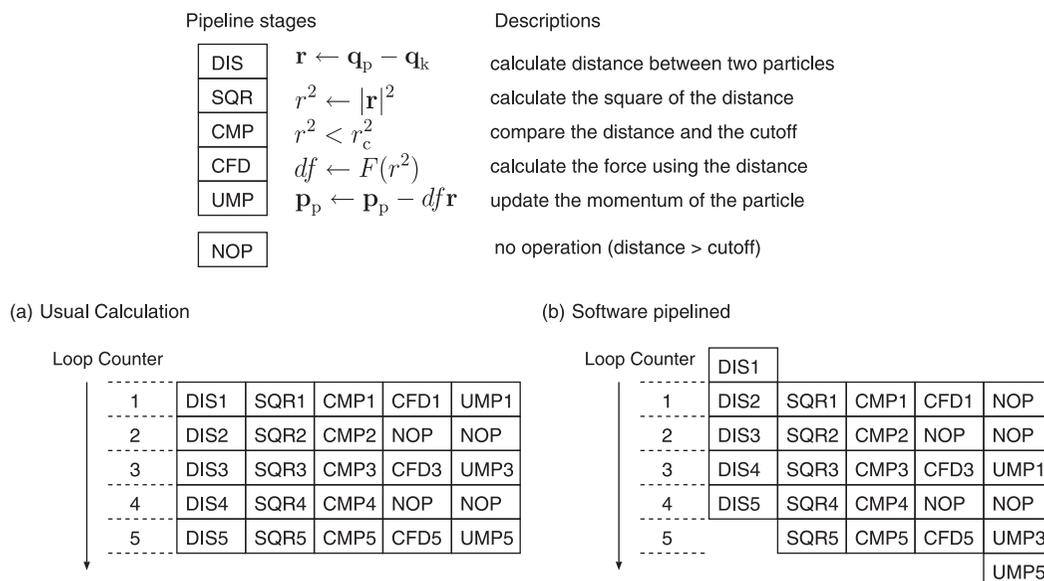


Fig. 9. Software pipelining. Calculation of the force consists of five parts: DIS, SQR, CMP, CFD, and UMP. (a) Implementation without software pipelining. All calculation are performed sequentially. (b) Implementation with software pipelining. The order of the instructions is arranged in order to improve the throughput of the instructions.

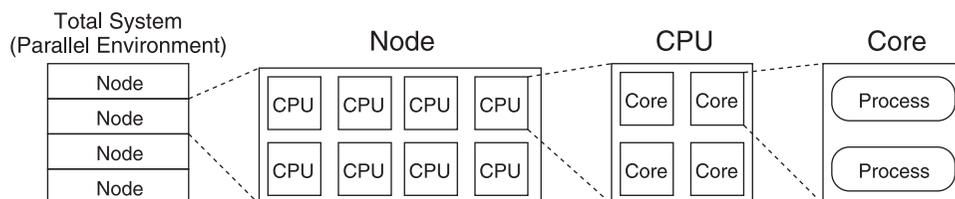


Fig. 10. Parallel environment. A total system of a supercomputer consists of several nodes. A node consists of several CPUs. A CPU consists of several cores. One or two processes are executed on a core.

§4. Parallelization

4.1. Parallelization scheme

Since the size of the system or computational steps are limited on the single process job, parallel computation is required in order to simulate large systems or long time steps. Parallel computation is executed on a parallel environment such as a supercomputer. A total system of a supercomputer has hierarchic structure, i.e., a total system consists of nodes, a node consists of CPUs, a CPU consists of cores (see Fig. 10). One or two processes are usually executed on a single core, and a set of processes works concertedly on the same task.

There has been considerable effort devoted to the parallelization of MD simulations. The parallel algorithms used for MD can be classified into three strategies, domain decomposition, particle decomposition, and force decomposition.¹²⁾ In domain

Algorithm 8 Calculating the force with software pipelining

```

1: for  $i = 1$  to  $N - 1$  do
2:    $\mathbf{q}_{\text{key}} \leftarrow \mathbf{q}[i]$ 
3:    $\mathbf{p}_{\text{key}} \leftarrow \mathbf{0}$ 
4:    $f dt \leftarrow 0$ 
5:    $k \leftarrow \text{KeyPointer}[i]$ 
6:    $j_a \leftarrow \text{SortedList}[k]$ 
7:    $\mathbf{r}_a \leftarrow \mathbf{q}[j_a] - \mathbf{q}_{\text{key}}$ 
8:    $\mathbf{r}_b \leftarrow \mathbf{0}$ 
9:    $j_b \leftarrow 0$ 
10:  for  $k = \text{KeyPointer}[i]$  to  $\text{KeyPointer}[i + 1] - 1$  do
11:     $\mathbf{r} \leftarrow \mathbf{r}_a$ 
12:     $r^2 \leftarrow |\mathbf{r}|^2$ 
13:     $j \leftarrow j_a$ 
14:     $j_a \leftarrow \text{SortedList}[k+1]$ 
15:     $\mathbf{r}_a \leftarrow \mathbf{q}[j_a] - \mathbf{q}_{\text{key}}$ 
16:    if  $r^2 < r_c^2$  then
17:       $\mathbf{p}_{\text{key}} \leftarrow \mathbf{p}_{\text{key}} + f dt \times \mathbf{r}_b$ 
18:       $\mathbf{p}[j_b] \leftarrow \mathbf{p}[j_b] - f dt \times \mathbf{r}_b$ 
19:       $f dt \leftarrow [(24r^6 - 48)/r^{14} + 8c_2] \times dt$ 
20:       $j_b \leftarrow j$ 
21:       $\mathbf{r}_b \leftarrow \mathbf{r}$ 
22:    end if
23:  end for
24:   $\mathbf{p}[j_b] \leftarrow \mathbf{p}[j_b] - f dt \times \mathbf{r}_b$ 
25:   $\mathbf{p}[i] \leftarrow \mathbf{p}[i] + \mathbf{p}_{\text{key}} + f dt \times \mathbf{r}_b$ 
26: end for

```

decomposition (also called spatial decomposition) the simulation box is subdivided into small domains, and each domain is assigned to each process. In particle decomposition, the computational workload is divided and distributed on the basis of the particles. Different processes have different particles. In force decomposition the computational workload is divided and distributed on the basis of the force calculation. Consider a matrix F_{ij} that denotes the force between i - and j -particles. Force decomposition is based on the block decomposition of this force matrix. Each block of the force matrix is assigned to each process. Generally, domain decomposition is suitable for systems with short-range interactions and simulations with a large number of particles,^{26),27)} and force decomposition is suitable for systems with long-range interactions such as electrostatic force and simulations with a large number of steps. In the present paper, we adopt a simple domain decomposition method for parallelization, i.e., we divide a system into small cubes of identical size. We use a Message Passing Interface (MPI) for all communication. To perform the computation, three types of communication are necessary: (i) exchanging particles that stray from the originally associated process, (ii) exchanging information of particles near

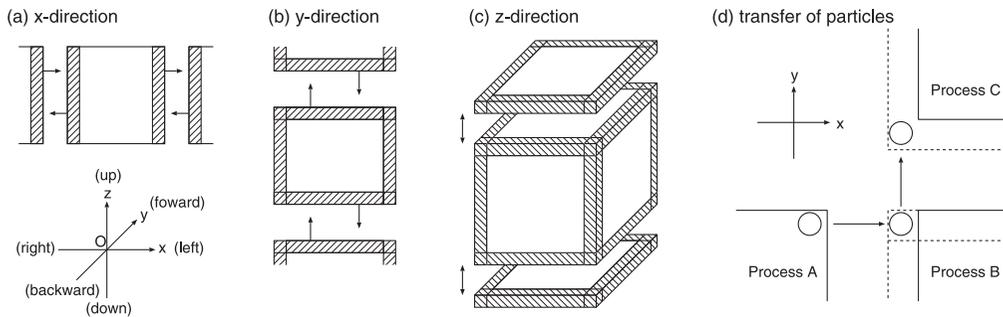


Fig. 11. Sending and receiving of positions of particles that are close to boundaries. (a) Sending along the x -direction. First, each process sends/receives the positions of particles near the right boundary to/from the right neighbor. Similarly, the positions of particles near the left boundary is transmitted. (b) Sending along the y -direction. Positions of particles near the boundaries are sent to neighbors. (c) Sending along the z -direction is performed similarly. (d) Transferring particles. Suppose there are three neighboring domains associated with Process A, B, and C. First Process A sends a particle to Process B along x -axis. Process B sends the particle received from Process A to Process C. Then Process C receives particles from Process A via Process B. Thus, each process obtains the information of particles from its diagonal neighbors without direct communication.

boundaries to consider the cross-border interactions, and (iii) synchronization of the validity check of pair lists.

4.2. Exchanging particles

Although all particles are initially in the domains to which they are placed, they tend to stray from their original domains as a simulation progresses. Therefore, each process should pass the migrated particles to an appropriate process at an appropriate time. There is no need for exchanging particles each step since it will not cause problems while migration distances are short. Therefore, migrated particles are exchanged simultaneously with a reconstruction of pair list.

4.3. Cross-border interactions

To calculate forces between particle pairs which are assigned to different processes, communication between neighboring processes is necessary. While a naive implementation involves communication in 26 directions since each process has 26 neighbors, the number of communications can be reduced to six by forwarding information of particles sent by other processes. Suppose a process is assigned the cuboidal domain $s_x \leq x < e_x$, $s_y \leq y < e_y$, and $s_z \leq z < e_z$, and the search length is denoted by r_s , which is defined in §2.3. The procedures to send and receive positions of particles on the boundaries are as follows.

- 1: Left: Send positions of particles in the region $s_x \leq x < s_x + r_s$ to the left process, and receive positions of particles in the region $s_x - r_s \leq x < s_x$ from the left process.
- 2: Right: Send the positions of particles in the region $e_x - r_s \leq x < e_x$, and receive positions of particles in the region $e_x \leq x < e_x + r_s$ from the right process.

- 3: Backward: Send positions of particles in the region $s_y \leq y < s_y + r_s$ to the backward process, including the particles received from the left and the right process. Receive positions of particles in the region $s_y - r_s \leq y < s_y$ from the backward process.
- 4: Forward: Send the positions of particles in the region $e_y - r_s \leq y < e_y$ to the process in front, including the particles received from left and the right processes. Receive positions of particles in the region $e_y \leq y < e_y + r_s$ from the process in front.
- 5: Down: Send the positions of particles in the region $s_z \leq z < s_z + r_s$ to the lower process, including the particles received from other processes. Receive positions of particles in the region $s_z - r_s \leq z < s_z$ from the lower process.
- 6: Up: Send positions of particles in the region $e_z - r_s \leq z < e_z$ from the upper process, including the particles received from other processes. Receive positions of particles in the region $e_z \leq z < e_z + r_s$ from the upper process.

After the above six communications, exchanging positions of particles close to boundaries are completed including between diagonal processes. The details are illustrated in Fig. 11. Note that the simple implementation of the communication may cause deadlock which is a situation that two or more processes will wait response forever. Suppose there are two processes, A and B. If A tries to send a message to B, and B also tries to send to A with MPI blocking communications, then the communication will not be completed. Similarly, suppose there are three processes, A, B, and C. If the communication path is $A \rightarrow B$, $B \rightarrow C$, and $C \rightarrow A$, then the system is getting into a deadlock. In the domain decomposition parallelism with periodic boundary condition, it is possible that a deadlock occurs. There are several techniques to avoid this problem, such as reversing the order of sending and receiving and using nonblocking operations. A simple way to avoid such a deadlock problem is to use the `MPI_Sendrecv` function. This function does not cause deadlock for closed-path communications. The pseudocode using `MPI_Sendrecv` is shown in Algorithm 9. *SendDir* and *RecvDir* denote the direction that the process should send to or receive from. There are six directions to complete the communication. The number of particles to be sent in each direction is fixed until the expiration of a pair list. Suppose N_s is a number of particles to send. Then the number of data sent is $3N_s$, since only coordinates are required to calculate forces. The message tag denoted by *Tag*, which is an arbitrary non-negative integer but it should be the same for sender and receiver.

4.4. Synchronization of pair list constructions

Each process maintains its own pair list and has to reconstruct it when it expires. If each process checks the validity of the pair list and reconstructs it independently, since it causes idle time, and the calculation efficiency decreases. To solve this problem, we synchronize the validity check of pair lists for all processes, that is, we reconstruct all the pair lists even when one of the pair lists has expired. While this reduces the average lifetime of the pair lists, the overall speed of the simulation is greatly improved since the idle time is eliminated (see Fig. 12). This synchronization process can be implemented with `MPI_Allreduce`, which is a global-reduction

Algorithm 9 Communications for cross border interactions

-
- 1: **for** $SendDir$ in [left, right, backward, forward, down, up] **do**
 - 2: $RecvDir \leftarrow$ the opposite direction of $SendDir$
 - 3: $SendBuf \leftarrow$ coordinates of particles to send
 - 4: $N_s \leftarrow$ number of particles to send
 - 5: $N_r \leftarrow$ number of particles to receive
 - 6: Prepare $RecvBuf$
 - 7: $DestRank \leftarrow$ rank of neighbor on the $SendDir$
 - 8: $SrcRank \leftarrow$ rank of neighbor on the $RecvDir$
 - 9: Call $MPI_Sendrecv(SendBuf, 3N_s, MPI_DOUBLE, DestRank, Tag, RecvBuf, 3N_r, MPI_DOUBLE, SrcRank, Tag, MPI_COMM_WORLD)$
 - 10: Update coordinates of particles using $RecvBuf$
 - 11: **end for**
-

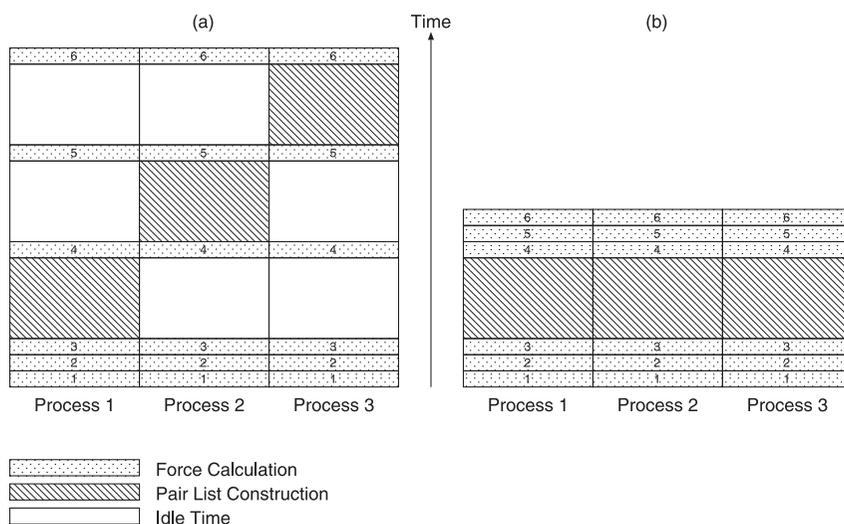


Fig. 12. Synchronization of pair lists. The numbers in the boxes of force calculation denote the time steps. The size of the boxes of force calculation and the pair list construction reflect the computational time required for them, i.e., the size of the box is proportional to the computational time. Suppose the lifetime of the pair list of Process 1 is 3 steps, that is, the pair list should be reconstructed after three force calculations. Similarly, the lifetime of the pairlists is 4 and 5 for Process 2 and 3, respectively. (a) Without synchronization, if the processes reconstruct pair lists independently, then other processes must wait for the reconstruction resulting in a huge amount of idle time. (b) With synchronization, if the processes synchronize their own pair lists with others, then the idle time vanishes while the average lifetime of pair lists also decreases.

function of the MPI. Note that the global synchronization of pair lists can have a serious effect on the parallel efficiency when a system is highly inhomogeneous such as a system involving phase transitions or a heat-conducting system. In such cases, further advanced treatment that manages pair lists locally at each location of grid would be required.

Table I. Summary for HITACHI SR16000/J2 at the National Institute for Fusion Science (NIFS) and SGI Altix ICE 8400EX at the Institute for Solid State Physics (ISSP). Location, name, CPU, number of cores per node, number of processes per node, and memory per node are shown. (*) Using simultaneous multithreading (SMT), 64 processes are executed on each node. (**) The single-node run is performed with four processes on one CPU and two-or-more-node runs are performed with 8 processes per node on two CPUs because of the queue design at ISSP.

Facility	Name	CPU	Cores	Processes	Memory
NIFS	HITACHI SR16000/J2	IBM POWER6 (4.7 GHz)	32	64 (*)	128GB
ISSP	SGI Altix ICE 8400EX	Intel Xeon X5570 (2.93 GHz)	8	4 or 8 (**)	24GB

4.5. Results of benchmark simulations

To measure the efficiency of the developed code, we perform benchmark simulations on two different parallel computers located at the different facilities, which are NIFS and ISSP. The details of the computers are listed in Table I. In the actual analysis we used the so-called weak scaling analysis, which is the dependence of the execution time in terms of node number under the condition of the number of particle per process keeping constant. The conditions for the benchmark simulations are as follows. Note that, all quantities are normalized by well depth ε and atomic diameter σ of the Lennards-Jones potential defined in §2.1.

- System size: $100 \times 100 \times 100$ for each process.
- Number of particles: 500,000 for each process (number density is 0.5).
- Initial condition: Face-centered-cubic lattice.
- Boundary condition: Periodic for all axes.
- Integration scheme: Second-order symplectic integration.
- Time step: 0.001.
- Cutoff length: 2.5.
- Search length: 2.8.
- Cutoff scheme: Add constant and quadratic terms to potential as shown in Eq. (2.2).
- Initial velocity: The absolute value of the velocities of all particles are set to be 0.9 and the directions of the velocities are given randomly.
- After 150 steps, we measure the calculation (elapsed) time for the next 1000 steps.

When a system with N particles is simulated for k steps in t seconds, then the number of MUPS is given by $\times Nk/10^6t$. At NIFS, we placed $4 \times 4 \times 4$ cubes at each node, and assigned one cube to each process (see Fig. 13), since each node has 32 cores, and we placed 64 processes on each node utilizing simultaneous multithreading (SMT). At ISSP, we placed $2 \times 2 \times 2$ cubes at each node, and assigned one cube to each process, since one node consists of two CPUs (8 cores) at ISSP.

The results obtained at NIFS and ISSP are summarized in Tables II and III, respectively. Both results are shown in Fig. 14. While the elapsed time of the 2-node calculation at NIFS is 302.6 s, that of the 128-node calculation is 412.7 s. Similarly, while the elapsed time of the single node calculation at ISSP is 183.13 s, that of the 1024-node calculation is 272.478 s. Since we perform weak scaling, the elapsed time

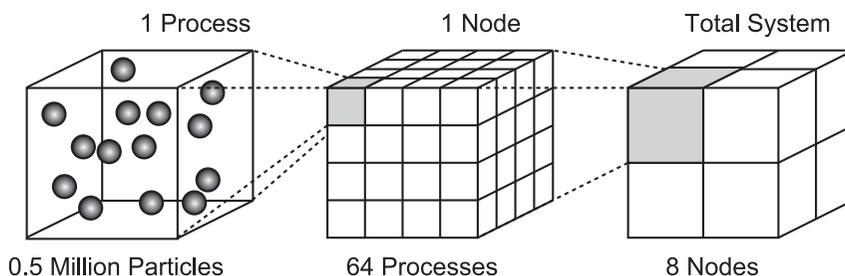


Fig. 13. Domain decomposition for parallelization at NIFS. Each process is assigned a cube with a length of 100 that contains 500,000 particles. Each node is assigned to a large cube made up of $4 \times 4 \times 4$ small cubes. The case of eight nodes is shown as an example.

Table II. Results of benchmark simulations at NIFS. Numbers of nodes, numbers of processes, numbers of particles, elapsed times [s], and speeds [MUPS] are listed. Efficiency is estimated based on the elapsed time obtained by the case of the 2-node calculation, since a single-node calculation is not permitted at NIFS.

Nodes	Processes	Particles	Elapsed Time [s]	Speed [MUPS]	Efficiency
2	128	64,000,000	302.623	211.484	1.00
4	256	128,000,000	305.469	419.027	0.99
8	512	256,000,000	309.688	826.638	0.98
16	1024	512,000,000	325.744	1571.79	0.93
32	2048	1,024,000,000	333.140	3073.78	0.91
64	4096	2,048,000,000	385.765	5308.93	0.78
128	8192	4,096,000,000	412.717	9924.48	0.73

Table III. Same as Table II for ISSP. Efficiency is determined relative to the elapsed time for the single-node calculation.

Nodes	Processes	Particles	Elapsed Time [s]	Speed [MUPS]	Efficiency
1	4	2,000,000	183.13	10.9212	1.00
2	16	8,000,000	186.296	42.9424	0.98
4	32	16,000,000	187.751	85.2194	0.98
8	64	32,000,000	190.325	168.133	0.96
16	128	64,000,000	194.932	328.32	0.94
32	256	128,000,000	203.838	627.95	0.90
64	512	256,000,000	224.028	1142.71	0.82
128	1024	512,000,000	228.485	2240.84	0.80
1024	8192	4,096,000,000	272.478	15032.4	0.66

should be independent of the number of processes provided the computations are perfectly parallelized. Therefore, the increase in elapsed time of the calculation with larger nodes is regarded as being due to the parallel overhead.

§5. Parallel overhead

The benchmark results described in the previous section show that the parallelization efficiency decreases as a number of processes increases, while the amount of communication in one step seems to be negligible compared with the computa-

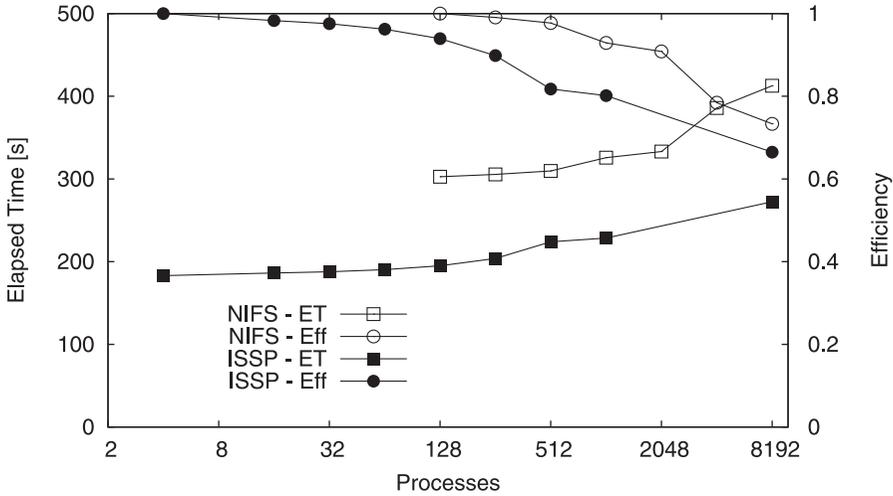


Fig. 14. Results of benchmark simulations. The results at NIFS are denoted by open symbols and ISSP by filled symbols. ET and Eff denotes elapsed time [s] and efficiency, respectively. The increase in elapsed time is due to the parallel overhead.

tions. In the present section, we investigate the cause of the parallel overhead of our codes in order to identify the factor of the decrease in parallelization efficiency. All simulations presented here are performed under the same condition as that used for the benchmarks in §4.5.

5.1. Granularity

In parallel computing, the ratio of computation to communication is important. This ratio is called the *granularity*. Fine-grain parallelism refers to the situation that data are transferred frequently and computational tasks are relatively light compared with communication. Coarse-grained parallelism, on the other hand, refers to the situation that computational tasks are relatively heavy compared with communication. In general, a program having coarse granularity exhibits better performance for a large number of processes. Here, we estimate the granularity of our code. Although the interaction length is set to 2.5, each process has to send and receive information of particles whose distances from boundaries are less than 2.8, since we adopt the DUTC method and the search length is set to 2.8. Each process is assigned to a cube of length 100 and the number density is set to 0.5. The communications are performed along the x -, y -, and z -directions, and two communications are involved for each direction. The amount of transmitting data increases in order of the x -, y -, and z -directions, since the received data of particles are forwarded. The position of each particle is expressed by three double-precision floating-point numbers, requiring $3 \times 8 = 24$ bytes. The amount of communication along y -axis is larger than that along x -axis, since the information of particles received along x -axis is transferred to y -axis (see Fig. 11). The number of particles transmitted along the x -axis is $100 \times 100 \times 2.8 \times 0.5 = 14,000$. Therefore, 336 KB are transmitted along the x -axis. Similarly, $(100 + 2.8 \times 2) \times 100 \times 2.8 \times 0.5 \times 24 = 355$ KB of

Table IV. Breakdowns of the computational time at NIFS. Total, Force, pair list, and Communication denote the total time of computation [s], the time spent on force calculations [s], the time required to construct pair lists [s], and communication time [s], respectively.

Nodes	Total [s]	Force [s]	pair list [s]	Communication [s]
2	304.609	217.403	38.26566	12.54
128	404.335	217.733	41.85143	14.39

Table V. Same as Table IV for ISSP.

Nodes	Total [s]	Force [s]	pair list [s]	Communication [s]
1	180.945	144.92	14.388425	4.12
256	228.614	144.638	16.299467	7.92

data are transmitted along the y -axis and 375 KB of data are transmitted along the z -axis. Since inner-node communication is performed on shared memory and should be sufficiently faster than internode communication, we only consider the internode communication. Each node at NIFS has $4 \times 4 \times 4 = 64$ cubes, and therefore, 16 processes involve internode communication. The largest amount of communication is along the z -axis, which is about $375 \text{ KB} \times 16 = 6 \text{ MB}$. The bandwidth between each node at NIFS is about 5 GB/s. The total amount of internode communication is at most 36 MB, and therefore, the time spent on communication will be about 7.2 ms for each step. The computational time without communication is about 0.3 s, which is 500 times longer than the estimated communication time. This implies that this code has coarse-grained parallelism and should be less affected by increasing the number of processes. The benchmark results, however, show a significant decrease in parallel efficiency for large-scale runs.

We have also investigated the communication set up time at Kyushu University. The supercomputer at Kyushu University is HITACHI SR16000/J2 which is the same as that at NIFS. But it is 42 nodes while NIFS has 128 nodes. We observed the communication set up time for MPI_Barrier and MPI_Allreduce over full of nodes, and found that the set up times are at most 50 us and 130 us, respectively. Therefore, communication cannot be a main factor of the parallel overhead.

5.2. Lifetime of pair lists

Since we perform weak scaling analysis, the size of the system increases as the number of nodes increases. This can decrease the lifetime of pair lists, since the maximum speed of the particles increases in a larger system and faster particles reduce the lifetime of pair lists.*) The construction of a pair list takes much longer time than the force computation takes, and therefore, the simulation run time is strongly affected if the number of pair lists being constructed increases. This increases the simulation run time even if the simulation is perfectly parallelized, and therefore, it is inappropriate to regard this cost as the parallelization cost. Therefore, we investigate the system-size dependence on the time required for construction of pair

*) In equilibrium, a distribution for the velocity obeys the Maxwell-Boltzmann type. While there can be an infinitely fast particle in an infinite-size system, the expectation value of the maximum velocity is finite in the finite-size system and the value increases as the system size increases.

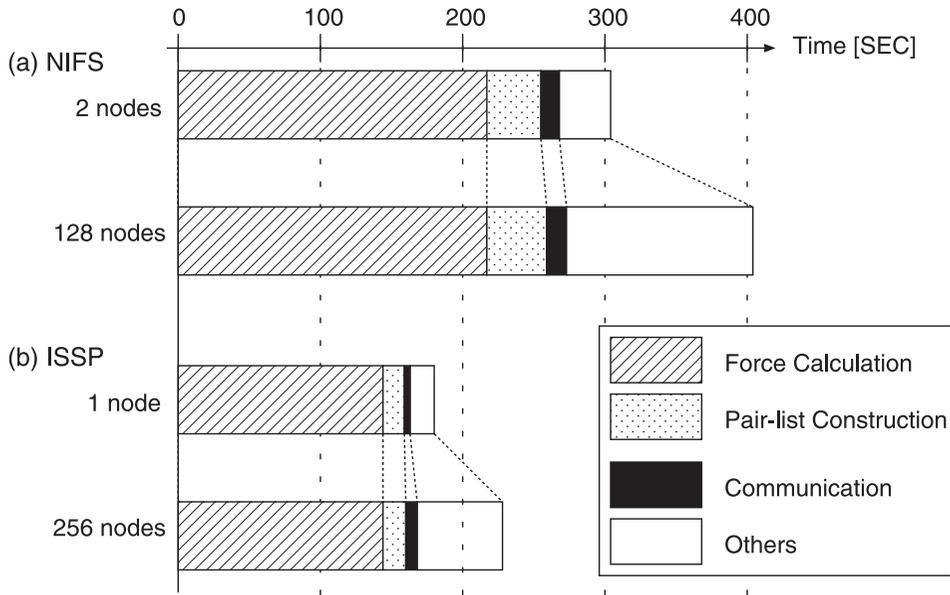


Fig. 15. Breakdowns of the computational time for (a) NIFS and (b) ISSP. The smallest and largest runs are shown for both cases. Although the communication time increases as the number of nodes increases, the increase in time for communication is at most 8% of the total increase, which cannot explain the observed parallel overhead. The idle time due to the synchronization is counted as “Others”.

lists. At NIFS, the pair list was reconstructed 20 times for the 2-node run while it was reconstructed 21 times for the 128-node run. The average life time is 51.2 steps for 2-node and 53.8 steps for 128-node calculations. Therefore, the computational cost to construct pair lists increases for larger number of nodes, but the difference is small. The breakdowns of the computational time for the smallest and the largest runs are listed in Tables IV and V and are also shown in Fig. 15. The additional cost of the pair list construction at NIFS is only 4 [s] which cannot explain the parallel overhead, which is over 100 [s]. Therefore, the lifetime of pair lists is not the main factor causing the reduced parallel performance.

5.3. Communication management

Each simulation involves two types of communication at each step. One is point-to-point communication for sending and receiving data of particles, the other is collective communication for the synchronization of pair lists. The point-to-point communication is implemented in a simple way, that is, all processes send data in one direction and receive data from the opposite direction. All processes send and receive data simultaneously in the same direction using `MPI_Sendrecv` (see Fig. 16 (a)). While the simple implementation of the communication may cause deadlock for the system with THE periodic boundary condition, the function `MPI_Sendrecv` avoid the deadlock implicitly by reordering of communication, which may cause some overhead in parallel efficiency for a large number of processes. Therefore, we examine different ways of communication which avoid the deadlock explicitly in order to investigate

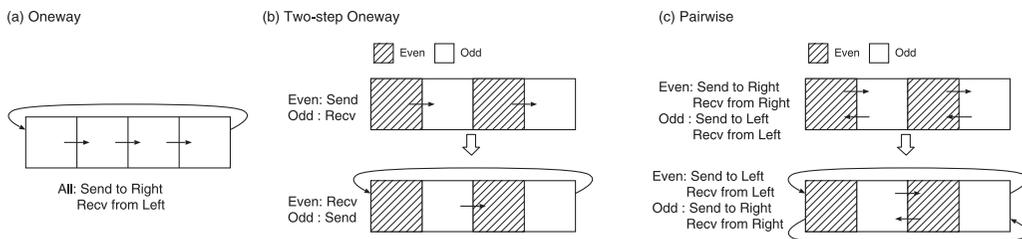


Fig. 16. Communication methods. (a) Oneway: All processes send information in the same direction, and receive the data simultaneously using `MPI_Sendrecv`. (b) Two-step Oneway: Although all processes send information in the same direction, the sending and receiving processes are explicitly separated using `MPI_Send` and `MPI_Recv`. (c) Pairwise: Each process is classified as even or odd. When the even processes exchange the positions of particles with their left neighbors, the odd processes exchange the positions of particles with their right neighbors. Then the directions of communication are reversed. Communication is implemented by `MPI_Sendrecv`.

whether the communication management by `MPI_Sendrecv` is the factor of the parallel overhead. Here we performed three methods. The first method, referred to as ‘Oneway’, is the simple implementation described in §4.3. The second method, referred to as ‘Two-step Oneway’, involves communications in 2 stages (see Fig. 16 (b)). All processes are classified into two groups, even and odd. In the first of the two stages, even processes send data and odd processes receive data. In the second stage, data are transmitted in the reverse direction. The communications are implemented by the blocking functions `MPI_Send` and `MPI_Recv`. The third method, referred to as ‘Pairwise’, arranges the order of communication explicitly. All processes are classified into two groups, even and odd, similarly to in the previous method. When the even processes exchange data with their right neighbors, the other processes exchange data with their left neighbors, and vice versa (see Fig. 16 (c)). The communications are implemented by `MPI_Sendrecv`. Note that, the behavior of ‘Pairwise’ at ISSP is not monotonic, i.e., the data of 128-node run is much faster than those of 8-, 16-, 32-, and 64-node runs. This may be due to the network topology at ISSP which is hypercube, however, the details should be investigated later.

A comparison of benchmark test for the above three methods is shown in Fig. 17. Figure 17 (a) shows the results at NIFS. The results of Pairwise only slightly differ from those of Oneway. While the results of Two-step Oneway show similar performance up to 32 nodes, those for larger nodes are significantly slower than the other methods. Figure 17 (b) shows the results at ISSP. There are major differences between the performances of the three methods even though the amounts of communication are identical, and Two-step Oneway considerably slower than the other methods. The reason why Two-step Oneway is the slowest method in both cases may be that double threads are executed by separating the sending and receiving processes instead of using `MPI_Sendrecv`. While it is unclear why the results of the three methods are almost the same for the largest number of nodes at ISSP, we can conclude that the overhead of communications strongly depends on the hardware environment. Considering the above results at NIFS and ISSP, the simple Oneway scheme is found to be less affected by the architecture of parallel machine and con-

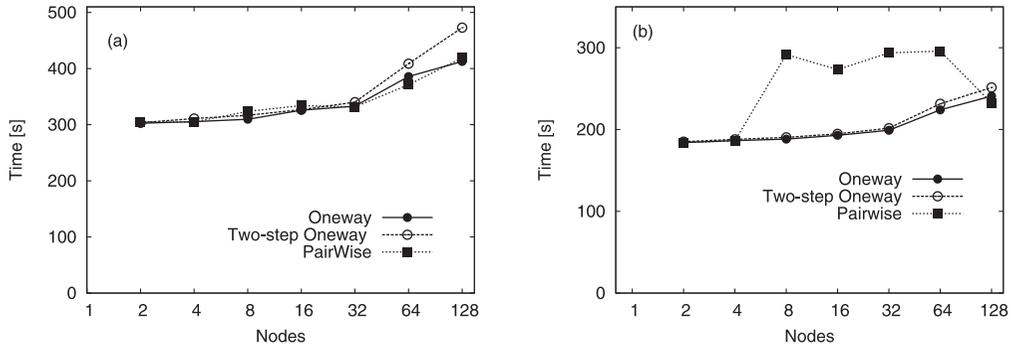


Fig. 17. Comparison between three communication schemes at (a) NIFS and (b) ISSP. The Oneway scheme exhibits the best performance at both facilities.

sistently has relatively good performance.

5.4. Synchronicity problem

Parallel programs adopting the single program multiple data (SPMD) model usually require synchronicity, and the parallel MD described here is one such example. The slowest process determines the overall performance, since other processes must wait for the slowest process owing to the synchronization. Therefore, the parallel efficiency is strongly affected by the fluctuation of the computation time of each process. One of the factors causing the fluctuation is load imbalance. Load imbalance refers to the inhomogeneity of workload among processes. Unequally assigned tasks cause the fluctuation of the computational time of each step and they decrease the parallel efficiency. The workload of parallel MD mainly depends on the number of interacting pairs in each process. To clarify this problem quantitatively, the number of interacting pairs is counted at the final state of the 128-node run. The parallel efficiency is determined by the slowest process which should be with the largest number of interacting pairs.

In our calculation, we found that the largest and the smallest number of pairs were 11,656,318 and 11,629,010, respectively. The difference is about 0.02%, which cannot be responsible for the reduction in parallel efficiency. Note that this small fluctuation originates from the fact that the initial condition is homogeneous. The load imbalance can be serious for inhomogeneous systems such as systems involving phase transitions. However, our benchmark results show that the parallel efficiency decreases as the number of processes increases, even for a homogeneous system with a uniform workload for each process. Therefore, we assumed that there are other factors that affect the parallel efficiency.

Next, we investigate the fluctuation of the computational time of force for each process. Suppose $t_{(p)}^{(i)}$ is the computational time spent for force calculation of p -th

process at i -th step. The average of the force calculation at i -th step is defined as

$$t_{\text{ave}}^{(i)} = \frac{1}{N_p} \sum_p^{N_p} t_{(p)}^{(i)}, \tag{5.1}$$

with the number of processes N_p . The time evolutions of $t_{\text{ave}}^{(i)}$ at NIFS and ISSP are shown in Fig. 18. The data of the smallest and the largest runs are presented. Note that, we observe the time spent only for force calculation which dose not involve communication. We found that the computational time for force show almost the same values and independent of the number of nodes. However, the computational time for each step itself might fluctuate, which can reduce the parallel efficiency. Therefore, we focus on the calculation time of force for each step. To investigate the fluctuation of the computational time for each step, we define the relative difference as

$$t_{\text{diff}}^{(i)} = \frac{t_{\text{max}}^{(i)} - t_{\text{min}}^{(i)}}{t_{\text{ave}}^{(i)}}, \tag{5.2}$$

where $t_{\text{max}}^{(i)} \equiv \max_p\{t_{(p)}^{(i)}\}$ is the maximum time, and $t_{\text{min}}^{(i)} \equiv \min_p\{t_{(p)}^{(i)}\}$ is the minimum time spent on the force calculation among all processes at i th step.*) $t_{\text{diff}}^{(i)}$ denotes the difference between the times spent on the slowest and fastest processes normalized by the average time at i th step. The time evolutions of the average and relative difference for force calculations at NIFS are shown in Fig. 19 (a). While the average times for force calculations are identical for 2- and 128-node runs, the relative differences of the large run are significantly larger than those of the small run. The relative difference of the large run sometimes fluctuates by 100% ($t_{\text{diff}}^{(i)}$ becomes 1.0), which implies that the computational time is doubled at the step. The same plots for ISSP are shown in Fig. 19 (b). Similar phenomena are observed, although the relative differences of the large run are at most 40%. The relative differences of the small run at ISSP are almost zero, since the run has only four processes, which is the reason why the fluctuation is relatively small.

From the above, we can conclude that the main factor degrading the parallel efficiency is the fluctuation of the computational time at each step. Although the average time varies only slightly, the fluctuation of the computational time between processes increases as the number of processes increases. Then the delays of the slowest process accumulate as the overhead. There are several possible factors causing this fluctuation. One possible source is noise from an operating system (OS). An OS has several background processes called daemons, which interfere with the user execution time and decrease its efficiency.²⁸⁾ This noise from OS is referred to as OS

*) We investigated the difference between the maximum and the minimum time spent for force calculation among processes instead of the standard deviation, since the slowest process determines the global speed of the calculation and the quantity observed in the present manuscript is directly associated with the parallel overhead. It is also worth to be noted that the probability distribution function of $t_{(p)}^{(i)}$ is not the Gaussian type. Therefore, the standard deviation is not an appropriate index to characterize the fluctuation of the calculation time.

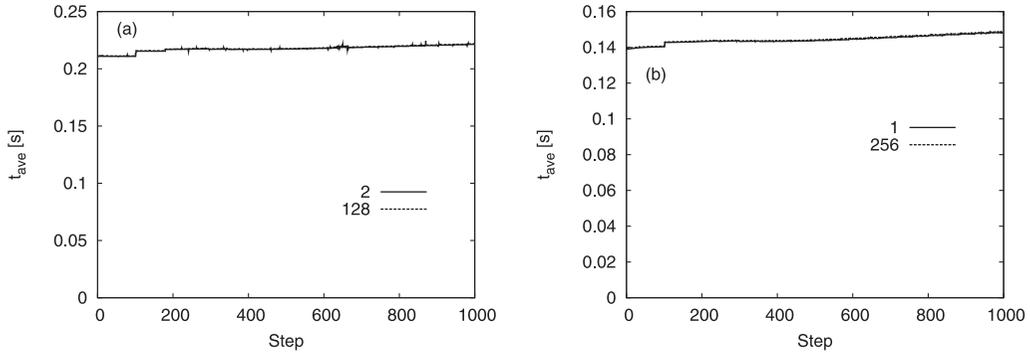


Fig. 18. Computational CPU time for force calculations averaged over all processes at each step. (a) The data obtained at NIFS. The data obtained from the smallest and largest runs are denoted by ‘2’ and ‘128’, respectively. (b) The data obtained at ISSP. The data obtained from the smallest and largest runs are denoted by ‘1’ and ‘256’, respectively. Time evolutions are almost independent of the number of nodes.

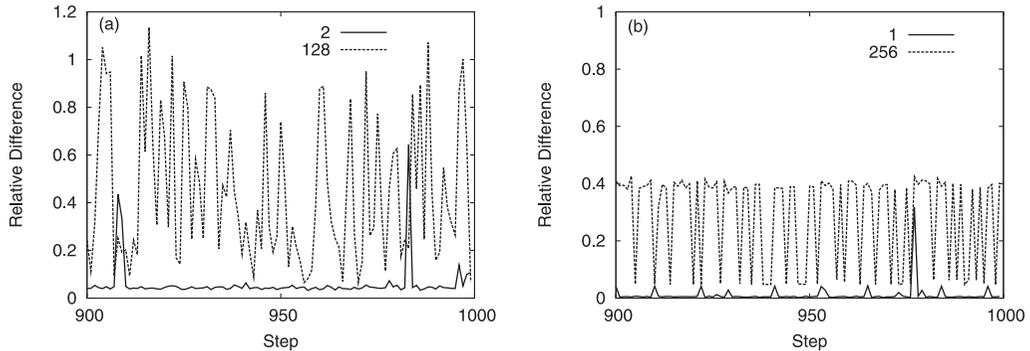


Fig. 19. Time evolutions of the relative difference at (a) NIFS and (b) ISSP. Only the final 100 steps are shown for the visibility. The fluctuations become larger as the number of nodes increases.

jitter or OS detour. Interruptions by an operating system occur randomly, and they cause the fluctuation of the execution time of each process. Not only interruptions themselves, but also side effects caused by the interruptions, such as pollution of the CPU cache, are also a source of noise.

§6. Summary and further issues

We have developed an MD code that is not only suitable for massively parallel processors but also exhibits good single-processor performance for IBM Power and Intel Xeon architectures. Most of all algorithms presented in this paper can be applied to other particle systems with short-range interactions including many-body interactions. There are three levels of optimization. At the highest level, we attempt to reduce the computational cost by developing or choosing appropriate algorithms such as the bookkeeping method. Next, we have to manage memory access. All data required by cores should be in the cache so that the cores make the best use of its

computational power. Finally, we have to perform architecture-dependent tuning, since the development paradigms are considerably different between different CPUs.

MD is suitable for MPP-type simulations since the computation-communication ratio is generally small. However, we have observed a significant degradation in parallel efficiency despite the use of conditions under which the communication cost should be negligible. We have found that the fluctuation of the execution time of processes reduces the parallel efficiency. The fluctuations become serious when the number of process becomes over one thousand. The OS jitter, which is the interference by the operating system, can be a source of the fluctuation. There are many attempts to handle the OS jitter, but it is difficult to be settled by user programs. One possibility is to adopt hybrid parallelization, i.e., use OpenMP for inner-node communications and use MPI for internode communications. Hybrid parallelization not only reduces the number of total MPI processes, but also averages out the fluctuations which can improve the parallel efficiency. Note that the affect due to the OS jitter is generally order of nanoseconds or shorter, and the time required to perform one step of MD simulations is order of millisecond. Therefore, there may be other sources of noise, such as translation lookaside buffer (TLB) miss. TLB is a kind of cache, which translates the address from virtual to real.²⁹⁾ While it makes computation faster, it takes long time to fetch the data if the required address is not cached in the buffer. The translation tables are sometimes swapped out into other storage such as hard-disk, then the penalty of TLB miss can be expensive. Investigation of the source of the fluctuation in execution time is one of the further issues.

In this paper, the load imbalance problem is not considered. Load balancing is important for parallelization with a huge number of processes since the fluctuation of the timing of each process greatly degrades the parallel efficiency as shown in §5.4. There are two approaches to load balancing, one is based on force decomposition and the other is based on domain decomposition. The load balancing in NAMD (Not Another Molecular Dynamics program)⁷⁾ is essentially achieved by the force decomposition, while NAMD adopts a hybrid strategy of parallelization. The processes of NAMD are under the control of a dynamic load balancer. The simulation box is divided into subdomains (called *patches*), and the total computational task is divided into subtasks (called *compute objects*). The load balancer measures the load on each process and reassigns *compute objects* to processors if necessary. GROMACS (Groningen Machine for Chemical Simulations)⁸⁾ adopts a domain-decomposition-based strategy for load balancing, that is, it changes the volume of domains assigned to processors to improve the load balance. The simulation box is divided into staggered grids whose volume is different. A load balancer changes the volume of each grid by moving the boundaries of cells to reduce load imbalance. There is also a method of non-box-type space decomposition that utilizes the Voronoi construction. Reference points are placed in a simulation box and each Voronoi cell is assigned to a process. Load balancing is performed by changing the positions of the reference points.^{30),31)} Generally speaking, force-decomposition-based load balancing is a better choice if a system contains long-range interactions such as electrostatic force, and domain-decomposition-based load balancing is better for a system with

short-range interactions. However, the choice strongly depends on the phenomenon to be simulated, and therefore, both strategies or some hybrid of them should be considered.

The source codes used in this paper have been published online.³²⁾ We hope that the present paper and our source codes will help researchers to develop their own parallel MD codes that can utilize the computational power of petaflop, and eventually exaflop machines.

Acknowledgements

The authors would like to thank Y. Kanada, S. Takagi, and T. Boku for fruitful discussions. Some parts of the implementation techniques are owing to N. Soda and M. Itakura. HW thanks M. Isobe for useful information of past studies. This work was supported by KAUST GRP (KUK-II-005-04), Grants-in-Aid for Scientific Research (Contracts No. 19740235), and the NIFS Collaboration Research program (NIFS10KTBS006). The computations were carried out using the facilities of National Institute for Fusion Science; the Information Technology Center, the University of Tokyo; the Supercomputer Center, Institute for Solid State Physics, University of Tokyo; and the Research Institute for Information Technology, Kyushu University.

References

- 1) B. J. Alder and T. E. Wainwright, *J. Chem. Phys.* **27** (1957), 1208.
- 2) A. Rahman, *Phys. Rev.* **136** (1964), A405.
- 3) M. P. Allen and D. J. Tildesley, *Computer Simulation of Liquids* (Clarendon Press, Oxford 1987).
- 4) D. Frenkel and B. Smit, *Understanding Molecular Simulation – From Algorithms to Applications* (Academic Press, New York, 2001).
- 5) D. Rapaport, *The art of molecular dynamics simulation* (Cambridge University Press, 2004).
- 6) W. G. Hoover, *Molecular Dynamics*, Lecture Notes in Phys. **258** (Springer-Verlag, Berlin, 1986).
- 7) L. Kalé, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan and K. Schulten, *J. Comput. Phys.* **151** (1999), 283.
- 8) B. Hess, C. Kutzner, D. van der Spoel and E. Lindahl, *J. Chem. Theory Comput.* **4** (2008), 435.
- 9) R. Car and M. Parrinello, *Phys. Rev. Lett.* **55** (1985), 2471.
- 10) <http://www.top500.org/>
- 11) <http://www.nsc.riken.jp/>
- 12) S. Plimpton, *J. Comput. Phys.* **117** (1995), 117.
- 13) T. C. Germann and K. Kadau, *Int. J. Mod. Phys. C* **19** (2008), 1315.
- 14) B. L. Holian, A. F. Voter, N. J. Wagner, R. J. Ravelo and S. P. Chen, *Phys. Rev. A* **43** (1991), 2655.
- 15) D. M. Beazley and P. S. Lomdahl, *Parallel Comput.* **20** (1994), 173.
- 16) S. D. Stoddard and J. Ford, *Phys. Rev. A* **8** (1973), 1504.
- 17) B. Quentrec and C. Brot, *J. Comput. Phys.* **13** (1975), 430.
- 18) R. W. Hockney and J. W. Eastwood, *Computer Simulation Using Particles* (McGraw-Hill, New York, 1981).
- 19) W. Form, N. Ito and G. A. Kohring, *Int. J. Mod. Phys. C* **4** (1993), 1085.
- 20) D. E. Knuth, *The Art of Computer Programming* (Addison-Wesley, Reading, MA, 1973).
- 21) G. S. Grest and B. Dünweg, *Comput. Phys. Commun.* **55** (1989), 269.
- 22) K. Kadau, T. C. Germann and P. S. Lomdahl, *Int. J. Mod. Phys. C* **17** (2006), 1755.

- 23) L. Verlet, *Phys. Rev.* **159** (1967), 98.
- 24) M. Isobe, *Int. J. Mod. Phys. C* **10** (1999), 1281.
- 25) S. Meloni, M. Rosati and L. Colombo, *J. Chem. Phys.* **126** (2007), 121102.
- 26) R. Hayashi, K. Tanaka, S. Horiguchi and Y. Hiwatari, *Parallel Processing Letters* **15** (2005), 481.
- 27) D. M. Beazley, P. S. Lomdahl, N. G. Jensen, R. Giles and P. Tamayo, "Parallel Algorithms for Short-Range Molecular Dynamics", in *Annual Reviews in Computational Physics 3* (World Scientific, 1995), p. 119.
- 28) P. Beckman, K. Iskra, K. Yoshii and S. Coghlan, *ACM SIGOPS Operating Systems Review* **40** (2006), 29.
- 29) A. S. Tanenbaum, *Modern Operating Systems* (Prentice Hall, 2001).
- 30) H. Yahagi, M. Mori and Y. Yoshii, *Astrophys. J. Suppl.* **124** (1999), 1.
- 31) R. Koradi, M. Billeter and P. Günter, *Comput. Phys. Commun.* **124** (2000), 139.
- 32) <http://mdacp.sourceforge.net/>