Received: 27 July 2017 | Revised: 4 February 2018 | Accepted: 4 February 2018
DOI: 10.1002/cpe.4468

WILEY

RESEARCH ARTICLE

Implementing molecular dynamics simulation on the Sunway TaihuLight system with heterogeneous many-core processors

[1]College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China
[2]National Supercomputing Center in Changsha, Changsha 410082, China
[3]Department of Computer Science, State University of New York, New Paltz, NY 12561, USA

Correspondence
Kenli Li, College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China.
Email: lkl510@263.net

Funding information
Key Program of National Natural Science Foundation of China, Grant/Award Number: 61432005; National Outstanding Youth Science Program of National Natural Science Foundation of China, Grant/Award Number: 61625202; National Natural Science Foundation for the Youth of China, Grant/Award Number: 61602166

Summary

In various research of atom and molecule physical movements, molecular dynamics (MD) simulation is a common tool to simulate and investigate the real molecular motion. However, it introduces a significant penalty in performance, power, electricity, and running time. Consequently, once the simulation size scales up and computing demands keep growing, it comes at substantial costs in performance and energy usage. In this paper, an optimized MD implementation on the Sunway TaihuLight supercomputer with heterogeneous many-core processors is developed to address the abovementioned issues. The Sunway TaihuLight is a totally independently designed and developed Chinese supercomputer with a profusely customized integration approach and a brand new many-core processor, the SW26010. The computing power mainly supported by the homegrown many-core SW26010 processors differs from other existing heterogeneous supercomputers. The Sunway TaihuLight is a heterogeneous supercomputer that ranks first in the world with its peak performance over 100 PFLOPS. Firstly, we propose a new algorithm based on cluster particles to fit the special architecture of SW26010. Then, three optimization methods of the MD simulation are implemented step by step: parallelized extensions to SW26010, memory-access optimizations, and vectorization. After these optimization processes, a 14x speedup is achieved on a single computing node and yields significant performance and energy improvements. Superiorly, almost 24,000 computing nodes (6,000,000 cores) are applied in our experiments, which gain an almost linear speedup. Besides, the proposed methods also can be adapted and fit to other molecular dynamics codes, even other similar scientific applications.

KEYWORDS

computing science, heterogeneous many-core processors, high performance computing, molecular dynamics, simulation, the Sunway TaihuLight, the SW26010

1 | INTRODUCTION

1.1 | Motivation

The extremely powerful technique of molecular dynamics (MD) simulation[1,2] abstracts the study of matter at the atomistic level to address the classical many-body problems in context relevant. The MD simulation is a general used tool to study the physical movements of atoms and molecules in chemical physics,[2] materials science,[3] biomolecules modeling, and other scientific research fields. In the meanwhile, it is an exceedingly time-consuming application.[4,5] Researchers have been used to spend weeks or months for a single simulation as the system size scales up and computing demands keep growing. Therefore, it is imperative to seek more efficient methods to solve the problem.

The Sunway TaihuLight, the fastest supercomputer, is also the third most energy-saving supercomputer ranked in the TOP500 list.[6] The Sunway TaihuLight supercomputer is designed for large-scale applications in scientific computing and industrial areas.[7] It is one of the new generation Chinese supercomputers provided services for public user in China and even across the world. The supercomputers theoretical peak performance reaches 125.4 PFLOP/s provided by 10,649,600 cores and 1.31 PB of primary memory.[8,9] The Sunway TaihuLight supercomputer

 Copyright © 2018 John Wiley & Sons, Ltd.

is built up based on a brand new powerful processor, ie, the SW26010. The SW26010 many-core processor comprises the Management Processing Elements (MPEs) and Computing Processing Elements (CPEs) in one chip, which is totally different from other existing heterogeneous processors, like CPU-GPU. The SW26010 is a tailored multi-core processor for the Sunway TaihuLight with a custom programming language and compiling environment. Normal C/C++ or FORTRAN codes can directly run on the supercomputer but cannot obtain the best performance. As the saying goes, both sharpening and chopping wood do well. Therefore, adapting MD code for the Sunway TaihuLight is of significant importance.

In addition, how to make a combination of efficiency and large-scale in computing process of molecular motion is a problem faced in implementing and running applications in heterogeneous supercomputers. In this paper, a highly scalable MD simulation application is implemented with an effective and efficient parallelization and optimization in the Sunway TaihuLight supercomputer. The many-core SW26010 processors bring high speedup ratio and parallel efficiency. The MPEs pay a pivotal role in performance management, communication, and computation, and CPEs, which are relatively rich in compute capacity, are more suitable for parallel computing. The MD simulation is time consuming due to its huge calculation scale and dense floating-point calculation. Thanks to the weak correlation between particles, the MD simulation gets a strong parallelism. As the Sunway TaihuLight supercomputer is equipped with strong computing units, ie, the SW26010, and designed for large-scale applications of scientific computing, it is adequate to fit the demands of MD simulation.

Besides, on the path to parallelize computing processes in the MD simulation, one of major obstacles is the difficulty of data feeding. The immense computing power, due to the parallel collaboration of multi-cores, requires transportation capacity to keep pace. Fortunately, each CPE is equipped with 16 kB L1 instruction cache and 64 kB Scratch Pad Memory (SPM), which can serve as user-controlled buffer or automatic data buffering. The SPM enhances the efficiency of data transportation and reduces the waiting time while transportation is delay. However, exploiting the fullest potential of SPM is one of the most serious challenges, as the storage is still low for the computing process.

1.2 | Contributions

An in-depth research on transplanting and optimizing the MD simulation application to the Sunway TaihuLight is executed in this paper. The design, implementation, and performance about large-scale simulations of MD are also presented. The contributions of this paper are summarized as follows.

- Data structure and associated algorithms are meticulously designed to take advantages of the Sunway TaihuLight. Through implementing and parallelizing the code of MD simulation, a general open-source MD simulation application, or large-scale simulations are conducted effectively and efficiently.
- Optimizations are carefully explored in allusion to the new heterogeneous processer, ie, SW26011 processor. To optimize large-scale simulations of MD on SW26011, which supports several levels of parallelization, a hybrid programming model is adopted. Experimental results demonstrate that our methods reach a 14x speedup on a single node.
- The scalability of MD codes on the Sunway TaihuLight and the performance of short-range force calculation are evaluated, which is the most time-consuming part of MD simulation. For parallelization between nodes, Message Passing Interface (MPI) is applied. Communication between nodes via MPI is at 12 GB/s and a latency of about 1 μs. The explosive growth of data transmission speeds paves the way for parallel computing efficiency and accelerating large-scale computing capacity.

The remainder of this paper is organized as follows. Section 2 elaborates a review of the classical parallelization approaches on the MD simulation. Section 3 provides an in-depth introduction to the Sunway TaihuLight supercomputer and MD simulation. Section 4 demonstrates the basic algorithm of the neighbor list building and force calculation and provides a new method, fits better to the SPM. Section 5 gives detailed descriptions of more optimization methods. Section 6 provides extensive experiments to evaluate the optimization performance. Section 7 concludes this paper and puts forward future works.

2 | RELATED WORK

As an important application in the field of scientific computing, MD simulation has been researched and developed deeply. The idea behind the MD simulation is very simple: Newton's classical equations of motion, $F = ma$, are solved directly to evolve the positions and velocities of a large collection of atoms comprising the fluid or crystal of interest. The problem becomes serious as the number of atoms scales up. Theoretically, there are forces between every pair of atoms, and all these forces are independent to each other. In the past decades, the most direct solution adopted is calculating $N(N-1)/2$ forces, switching the MD simulation into an $O(N^2)$ problem. When involving more sorts of forces or various sizes and

structures of atom, the cost surged to $O(N^3)$ or $O(N^4)$. Numerous smart algorithms have been developed to reduce the complexity[10,11] while it still need powerful computing capability to solve large-scale MD simulations.

Over the past 20 to 30 years, there were affluent considerable works concentrating on developing and applying various kinds of parallel schemes in MD simulations. More interesting studies of MD can be found in other works.[12-15]

Furthermore, with the Graphics Processing Units (GPUs) became programmable, which offered an alternative to the distributed memory clusters in MD simulations, there were more optimizations concentrating on hardware of parallel molecular dynamics programs.[16,17] It was essential and practical to port it onto modern supercomputers like the TianHe-1A, which is equipped with powerful multi-core CPUs.[18,19] Heterogeneous devices performed different parallel processes in the CPU-GPU heterogeneous systems. Hence, the GPU was responsible for those computationally intensive parts of MD, whereas the CPU handled the rest. These heterogeneous processors dramatically improved the total performance of computation with low cost of ownership and power consumption, but the development and optimization of large-scale applications were also becoming exceptionally difficult. To address these problems, researchers often use various programming models/tools, including CUDA for NVIDIAs GPUs, OpenMP, and MPI for intra-/inter-node parallelization.[20-22] Abundant efforts on parallelizing MD simulation and porting MD codes are contributed about homogeneous systems' shared memory space or distributed memory space featuring a mix of OpenMP and MPI.[23-26] In the meanwhile, a single Intel Xeon Phi coprocessor was developed, offering double precision performance over TFLOPS, to overcome the fine-grained parallel obstacles.[27,28] Nowadays, the mainstream High Performance Computing (HPC) platforms are most heterogeneous clusters, generally equipped with many-core accelerators/coprocessors (GPUs, Intel Xeon Phi, or specialized ones).[15,29,30]

There also exist some implementations of molecular dynamics simulations on special-purpose machines. To explore high performance, Anton's tightly integrated hardwired pipelines and programmable cores are exploited.[31] Moreover, Anton 2 is a second-generation special-purpose supercomputer for molecular dynamics simulations that achieves significant gains in performance, programmability, and capacity compared to its predecessor, Anton 1.[32] MDGRAPE-3 accelerators can also provide solutions with high performance and high power efficiency, especially about the calculation of nonbonded interactions.[33]

However, even in these high performance computing platforms and special-purpose machines, there still exist various problems, such as a hierarchy of capabilities, memory availability, and communication latency.

3 | SYSTEM ARCHITECTURE AND MODELS

3.1 | The Sunway TaihuLight supercomputer

As the fastest supercomputer in the world, with a LINPACK benchmark rating of 93 PFLOPS, the Sunway TaihuLight consists of several subsystems. Supercomputer hardware system mainly consists of high-speed computing subsystem, a high-speed interconnection communication network subsystem, a storage subsystem, a power subsystem, a cooling subsystem, etc. The software system mainly includes an operating subsystem, a compiling subsystem, parallel program development environment, and a scientific computing visualization subsystem. This paper mainly focuses on the performance of MD simulation, which is related to the computing subsystem, the network subsystem, the compiling subsystem, and the parallel program development environment of the Sunway TaihuLight.

The computing subsystem is built with 4 levels. The computing nodes are the basic units of the computing system. Each super node contains 256 computing nodes. Each cabinet consists of 4 super nodes. And the entire computing system contains 40 cabinets.

The network of the Sunway TaihuLigh is named *Sunway Network*. Sunway Network takes a multi-level Fat-Tree-Cross Topology. There are 3 levels in the network system, ie, a central switching network, a super node network, and a resource-sharing network. In a super node, all of 256 computing nodes are fully connected by the super node network, which supports all-to-all communications between all of the $256 \times 4 \times 65 = 66,560$ processing cores. And the fundamental switching network is in charge of the communication between super nodes. An upper connection between super nodes is carried out through a central switching network. There are 64 optical fibers for every super node and each optical fiber is shared by 4 computing nodes, respectively.

3.2 | SW26010 processor

The Sunway TaihuLight supercomputer installs 40,960 SW26010 processors. As shown in Figure 1, there are four Core Groups (CGs) in each processor. Each CG contains an MPE and 64 CPEs. Both the MPE and CPEs run with a speed of 1.45 GHz. Every CG has one memory controller (MC) and 8 GB physical memory. Each MPE has a 32 kB L1 instruction cache and a 32 kB L1 data cache. The MPE has a 256 kB L2 cache for instruction or data. Each CPE has a 16 kB L1 instruction cache and a user-controlled scratch pad memory (SPM). The SPM is to be configured as either a fast buffer, which enable supporting precise user control, or a software-emulated cache, which enable achieving automatic data cache. However, generally a user-controlled buffering scheme provides better performance. The SW26010 processor is relatively power energy saving. The power consumption of the entire system is 15.371 MW, means 373.8 W/node.

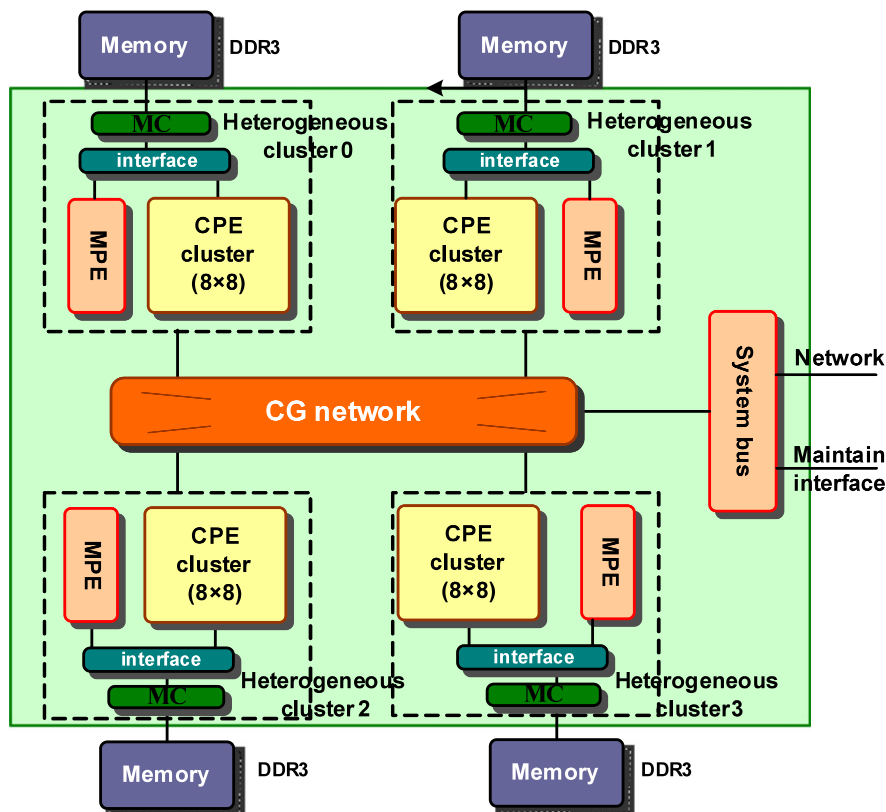

FIGURE 1 SW26010 processor architecture

The SW26010 processor is engineered in two user modes: (1) CG private mode and (2) Chip-sharing mode. The main difference between the two modes is the memory division pattern. The CG private mode is a general used mode. Each of the four CGs takes an independent address space under CG private mode. Applications run with MPI on four MPEs or four hybrid combinations (every MPE binds 64 CPEs). The Chip-sharing mode is a special offer for applications when need a larger memory. Processors enable providing services with a form of a NUMA Architecture in the Chip-sharing mode. Four CGs share a 32 GB physical memory and a global memory address space (see Figure 1). In this research, applications run with a hybrid-MPI-OMP mode.

Sunway TaihuLight equipped with an integrated programming mode for SW26010 processors. This system supports basic languages (eg, C, C++, and FORTRAN) and provides the parallel programming standard support for management processing element (MPE), like MPI3.0, OpenMP3.1, and OpenACC2.0. Programming on computing processing elements (CPEs) is based on a special accelerate thread library, named *athread*. MPE is just like a normal CPU, which takes the responsibility for communication, IO, job management, etc. CPEs take the role as accelerate cards, like GPU, which provide the majority of computing power. The athread for CPEs is like CUDA for GPU. While the difference is that, for now, *athread* is the only programming method for the CPE. There is another important difference between the CPE and general accelerate cards: the CPE shares the main memory with MPE instead of its own DRAM. So, there is no extra data transfer between the CPE and MPE during programming and running.

The MD simulation is a compute-intensive application, which needs excessive computing power. To run the simulation with an ideal performance, adapting the source codes to CPEs is indispensable. In Sections 4 and 5, there is a detailed introduction about the implementation and optimization of an open source MD codes, ie, LAMMPS on the Sunway Taihulight supercomputer.

3.3 | Molecular dynamics

Molecular simulation is a very powerful toolbox in modern molecular modeling and enables us to follow and understand a model dynamics structure where the motions of individual atoms are tracked. All motions of atoms follow the established physical laws. The main factor affecting an atomic motion is the interactions between atoms. Plenty of interactions are taken into account to estimate the system. Physically, potentials can be divided into 2 categories, ie, bond forces and non-bond forces. Bond forces refer to forces among atoms that form the physical-chemical contact. The rest interactions without a physical-chemical contact are the non-bond forces. The computation of non-bond forces is the tough aspect for time consuming.

According to a more subtle and technical subdivision, non-bond forces are divided into long-range forces and short-range forces. Long-range forces are a result of the charge of atoms, following the Coulomb's law. In fundamental algorithm, long-range forces are calculated by Equation (1)

$$U_{ele}(\overrightarrow{R}) = \sum_{i}\sum_{j\neq i}\frac{q_i q_j}{\epsilon r_{ij}}, \tag{1}$$

which is really complicated and expensive. Commonly, a long-range interaction that do not possess a singularity are more likely to be transformed into a short-range contribution. By calculated through a Fourier Transform, the processes of long-range forces are often transformed into calculations of short-range forces. Thus, the short-range forces are more widely used in MD simulations.

This paper involves two kinds of short range forces: 1) the Lennard-Jones (LJ) potential, which involves individual pairs of atoms and originally proposed in liquid argon, and 2) the Embedded-Atom Method (EAM) potential, which is a many-body potential and wildly used in metals.

A universally LJ potential calculation method is presented in Equation (2)

$$U(\overrightarrow{R}) = \begin{cases} \sum_{i}\sum_{j\neq i} 4\epsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right] & r_{ij} < r_c = 2^{1/6}\sigma \\ 0 & r_{ij} \geqslant r_c \end{cases}. \tag{2}$$

For a pair of atoms i and j located at $\overrightarrow{r_i}$ and $\overrightarrow{r_j}$, $\overrightarrow{r_{ij}} = \overrightarrow{r_i} - \overrightarrow{r_j}$ and $r_{ij} = |\overrightarrow{r_{ij}}|$. The parameter ϵ governs the strength of interaction and σ defines a length scale. For example, a pair of atoms exists an interaction of LJ potential. They repel each other at close range, then attract, and eventually, the force is cut off at limited separation r_c. The simulations are still scaled by $O(N^2)$ in Big O notation. Algorithms are developed to fix this problem. The complexity reduces to $O(N)$ with cell-list methods.

As a representative of density-independent pair potentials, the Lennard-Jones (LJ) potential is merely justified when electronic clouds remain closely to individual atoms. When it comes to metal, valence electrons might be shared among atoms. This calls for potentials that take the local electron density into account and that, consequently, have a many-body nature. The embedded-atom potential is involved for the following interactions. The EAM potential consists of two parts: 1) a pair interaction and 2)a many-body term. The pair interaction is a part that does not depend on density. The many-body term is a part that depends on a local value of density at the point where an atom is located. A calculation about the two parts of the potential energy is expressed as Equation (3)

$$U = \frac{1}{2}\sum_{i=0}^{N_m}\left(\chi\sum_{j\neq i}\phi(r_{ij}) + (1-\chi)\mu(\rho_i)\right). \tag{3}$$

Here, $\phi(r_{ij})$ is a density-independent pair potential and χ is a fractional contribution of this part. The embedding energy μ is a nonlinear function of a local atomic density ρ_i, which is defined as a sum over the neighbors of i of a local weighting function $w(r)$ as Equation (4)

$$\rho_i = \sum_{j\neq i}\omega(r_{ij}). \tag{4}$$

The nonlinearity of μ is necessary to ensure that it introduces many-body effects; otherwise, its contribution would be simply pairwise added. The pair potential is expressed as the form of Equation 5

$$\phi(r) = \begin{cases} \phi_{lj}(r) & r < r_s \\ \phi_{sp}(r) & r_s \leq r \leq r_m \\ 0 & r_{ij} \geqslant r_c. \end{cases} \tag{5}$$

$\phi_{lj}(r)$ is an LJ potential here, while other potential functions also can be used. The function $\phi_{sp}(r)$ is introduced to ensure that the pair interaction drops smoothly to zero over the range from r_s to r_m.

In conventional cell linked list algorithm, space is considered to be a rectangular region with periodic boundary conditions.[14,34] This region is subdivided into large cells that are assigned to the same processing node, then the subdivided region is further subdivided into small cells, and the edge of cells is equal to or larger than a cutoff distance. The neighbors of each atom can be found in the cell where the atom located and the neighboring cells. Hence, the appropriate construction of cell list is determined by the edge of the cell. The cutoff at r_m is introduced to reduce computational effort.

4 | BASIC ALGORITHM

The overall algorithm procedure of MD simulation is presented in Figure 2. Neighbor list building and force calculation are the most complicated and compute-intensive parts of the whole workload. On the basis of Amdahl's law, our work will focus on reducing the computation and effectively improving the performance of these compute-intensive payloads. We briefly introduce the neighbor list building part in Section 4.1 and the force

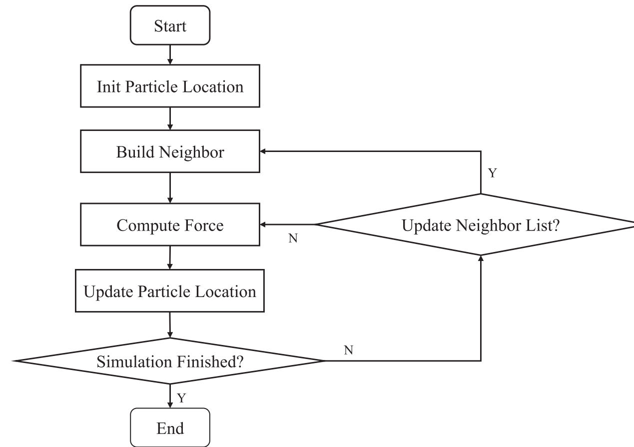

FIGURE 2 The flowchart of basic algorithm

calculation part in Section 4.2. Furthermore, a novel algorithm is proposed in Section 4.3 to mitigate the massive computing workload of the MD simulation.

4.1 | Neighbor list building

Neighbor list is a data structure, which is employed to store particle pairs with interactions between each other. With particles' positions changing, the neighbor list will be updated every several time steps to ensure the correctness. The update of the neighbor list is expensive, which means the neighbor list building and the force between neighbor particle pairs will be calculated repetitively. To obtain an ideal update frequency of neighbor list, the distance r_n, which can confirm whether particle pairs are neighbor or not, should be adjusted carefully.

Algorithm 1 Neighbor list building

Require: Location array *atom*; computing range *start*, *end*;
cutoff distance r_n.
Ensure: Neighbor list.
1: $r_n^2 \leftarrow r_n * r_n$;
2: **for** each local particle *i* **do**
3: $A \leftarrow$ getBinId(i);
4: **for** each neighbor cell *C* of *A* **do**
5: **for** each particle *j* of *C* **do**
6: **if** *i* and *j* are same particle **then**
7: continue;
8: **end if**
9: $d \leftarrow atomLocation[i] - atomLocation[j]$;
10: $d_{ij}^2 \leftarrow |d|^2$;
11: **if** $d_{ij}^2 < r_n^2$ **then**
12: push *j* into neighbor list;
13: **end if**
14: **end for**
15: **end for**
16: **end for**
17: **return** Neighbor list.

Algorithm 1 shows a general cell-list method. For each local particle of a process, the cell (bin) contains particles, which are located according to their coordinates *i* (line 3). Then, the neighbor cells are located with a stencil array, which is per-calculated according to the side length of the cell and cutoff distance. The algorithm traverses all particles (potential neighbors of *i*) contained by the neighbor cells and calculates the distance between *i* and potential neighbors then records the neighbors of *i* (line 4 to 15).

Algorithm 2 Force calculation

Require: Computing range *start*, *end*; cutoff distance r_{cut}; location array *atom*.

Ensure: Force array *F*, energy array *E*.

```
1:  if EAMSign then
2:      for i ranges from start to end do
3:          for all neighbors j of i do
4:              d_ij^2 ← |atomLocation[i] − atomLocation[j]|^2;
5:              if d_ij^2 < r_cut^2 then
6:                  calculate Density_ij;
7:                  TotalDensity_i ← TotalDensity_i + Density_ij;
8:              end if
9:          end for
10:         D[i] ← TotalDensity_i;
11:     end for
12: end if
13: for i ranges from start to end do
14:     for each neighbors j of i do
15:         d_ij^2 ← |atomLocation[i] − atomLocation[j]|^2;
16:         if d_ij^2 < r_cut^2 then
17:             calculate Energy_ij;
18:             TotalEnergy_i ← TotalEnergy_i + Energy_ij;
19:             calculate Force_ij;
20:             TotalForce_i ← TotalForce_i + Force_ij;
21:         end if
22:     end for
23:     F[i] ← TotalForce_i;
24:     if energySign then
25:         E[i] ← TotalEnergy_i;
26:     end if
27: end for
28: return F. E
```

4.2 | Force calculation

This part introduces a general force calculation algorithm. An integrated framework is presented in Algorithm 2. The framework consists of several steps. Firstly, all particles in the system need a force evaluation, and a traversal among all particles is taken at line 13. Secondly, the neighbor list is built on the basis of r_n, which is greater than the cutoff distance and rebuilt in every several time steps. Hence, the distance d between particle i and every corresponding neighbor recorded in the neighbor list should be checked. Finally, if $d \le r_{cut}$, there is a force evaluation according to the specified potential energy equation.

In theory, Newton's third law can cut the calculation in half. However, with a relatively poor performance in memory access, the Newton's third law could not bring any improvements. Contrary, it requests a mass of atomic operations. The law shows a really bad efficiency in our experiences. Finally, we give up on it.

4.3 | The particle-cluster method

Either the neighbor list building or the force calculation, there is a traversal among all local particles and their neighbors (or potential neighbors). In the beginning, particles are stored in the main memory according to their physical positions in space. With the simulation going on, particles are moving in the space, and this leads to a mismatch between the memory positions and the simulation physical positions of particles. Gradually, some particles that belong to a same cell may move long distances in different directions and belong to different cells. Which means that these particles in a same cell are randomly spread everywhere in the memory. As a result, here is a high performance penalty due to the high cache miss rate.

There is a strategy to solve this problem. After every several time steps, there is a reordering of particles in memory according to their physical positions to ensure that the memory positions of particles are consistent with the physical positions within a certain range. With introducing of

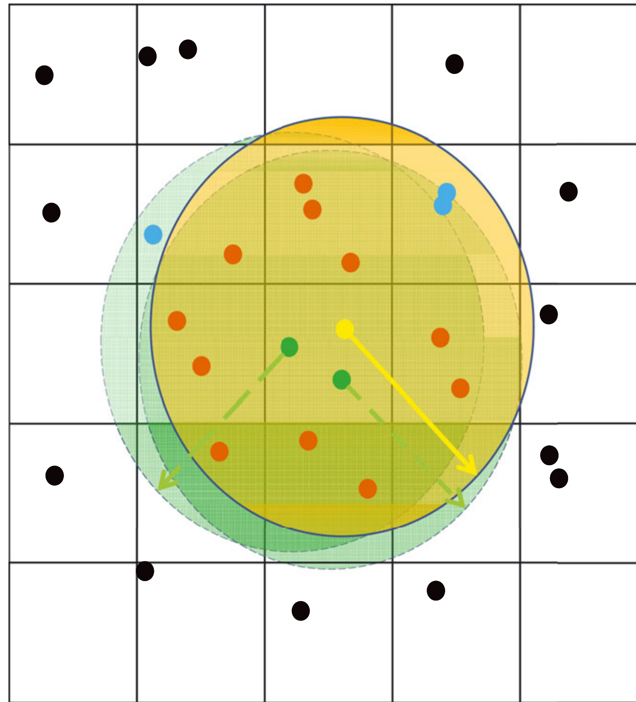

FIGURE 3 With the cluster method, the yellow and green particles are taken as a particle cluster and share the red particles as neighbors. For other three blue particles, each is the neighbor of at least one particle of the particle cluster. During the building of the neighbor list, all of the red and blue particles are written into the neighbor list as neighbors of the particle cluster

reordering, it ensures the data locality and solves the problem of high cache miss rate. However, the memory positions of particles are still random even in the same cell or neighbors of a same particle. Conducting a reordering strategy before every rebuilding step of neighbor list can avoid the randomness of memory positions during the rebuilding. But, there are only some (generally not all) particles in one cell that are neighbors of each other, and the randomness of memory positions still exists during the force calculation.

Here, we involve an adjacent particle-cluster algorithm, and implement this algorithm into neighbor list building and force calculation with cluster. A cluster means that some particles are adjacent to each other. How to define the "adjacent"? For a particle (the yellow particles in Figure 3), the particle and its several adjacent particles in same cell (the green particles in Figure 3) constitute a whole cluster. In other words, the adjacent particles are the whole particles exist in the same cell, and this is a cluster. Because the adjacency of these particles, they will share most of the neighbors (as the red particles in Figure 3).

Algorithm 3 Cluster neighbor list building

Require: Location array *atom*; computing range *start*, *end*;
cutoff distance r_n.
Ensure: NL: neighbor list.
1: Reorder the particles;
2: $r_n^2 \leftarrow r_n * r_n$;
3: **for** each local cell *A* **do**
4: **for** each particle cluster *i* in cell *A* **do**
5: **for** each neighbor cell *C* of *A* **do**
6: **for** each particle *j* of *C* **do**
7: **if** *j* is a neighbor of any particle in *i* **then**
8: push *j* into neighbor list;
9: **end if**
10: **end for**
11: **end for**
12: **end for**
13: **end for**
14: **return** Neighbor list.

TABLE 1 The changes of neighbor lines and max neighbor number of each line. The neighbor lines increase, whereas the max neighbor number decreases. There are generally no more than 12 particles in each cell, so there are almost no change after the cluster size is bigger than 12

Cluster Size	Neighbor Lines Number	Max Neighbor Number	t_force	t_neigh
1	131,072	94	2.82	0.49
2	69,786	148	1.70	0.24
4	39,547	188	1.53	0.17
8	22,635	224	1.43	0.15
12	17,832	245	1.37	0.12
16	17,576	245	1.37	0.12

As shown in Algorithm 3, the adjusted algorithm will reorder the memory positions of particles every time before the neighbor list building. The neighbor list is built in units of cluster instead of a single particle. All neighbors of each particle belonging to a cluster are recorded in a corresponding line in the neighbor list without repetition. Thus, the size of neighbor list will change from

$$S = P \times Max(|A_i|) \tag{6}$$

to

$$S = \sum_{i=1}^{P} \left\lceil \frac{cell_i}{p} \right\rceil \times Max(|B_i|). \tag{7}$$

Here, P is the local particles' number, A_i is the set of neighbor particles for local particle i, $cell_i$ is the particle number of cell i, p is the particle number of each cluster, and B_i is the set of neighbor particles for particle cluster i ($B_i = \bigcup A_j$, where j is the id of each particle in cluster i). The line number of neighbor list will reduce, whereas the element number of each line will increase. The total time of neighbor list building should reduce for $|B_i| \leq \sum |A_j|$ (j is the id of each particle in cluster i). Table 1 shows the changes of neighbor lines and the max neighbor number of every line. The range of time reduce depends on the number of shared neighbor particles, which is related to the particle density.

Algorithm 4 Cluster force calculation

Require: Computing range *start*, *end*; cutoff distance r_{cut}; location array *atom*.

Ensure: Force array F, energy array E.

```
1:  for cluster ii ranges from start to end do
2:      for all neighbors j of cluster ii do
3:          for each particles i in cluster ii do
4:              d_ij^2 ← |atomLocation[i] − atomLocation[j]|^2;
5:              if d_ij^2 < r_cut^2 then
6:                  calculate Energy_ij;
7:                  TotalEnergy_i ← TotalEnergy_i + Energy_ij;
8:                  calculate Force_ij;
9:                  TotalForce_i ← TotalForce_i + Force_ij;
10:             end if
11:         end for
12:     end for
13:     F[i] ← TotalForce_i;
14:     if energySign then
15:         E[i] ← TotalEnergy_i;
16:     end if
17: end for
18: return F. E
```

On the other hand, during the force calculation (as shown in Algorithm 4), each particle in a cluster will match its neighbors in the corresponding line of the neighbor list. The distance will be calculated. Once a new neighbor of cluster *ii* is loaded, the forces of all particles in cluster *ii* are evaluated. The adjusted algorithm still accesses the memory during the force calculation, but the number of memory accesses is reduced with the introduction of a cluster. Assuming that there are n particles in a cluster m, in the general method, there are q_i potential neighbor particles of the particle q in neighbor list and Q_i real neighbors, whereas p potential neighbor particles of m in total. Then, there are $\sum_{i=1}^{n} q_i$ times memory accesses in a general

TABLE 2 Memory access bandwidth

Size	1 CPE, GB/s	64 CPEs, GB/s
8 bytes	0.05	0.99
16 bytes	0.10	1.99
32 bytes	0.20	3.92
64 bytes	0.41	7.96
128 bytes	0.80	15.77
256 bytes	1.47	28.88
512 bytes	2.65	28.98
1,024 bytes	4.28	27.97
2,048 bytes	6.33	30.48
4,096 bytes	8.14	30.18
8,192 bytes	9.47	30.78
16,384 bytes	10.60	30.94

TABLE 3 Performances of neighbor list building and force calculation in different methods

Methods	Neighbor Lines Number	Max Neighbor Number	t_force	t_neigh
org	131,072	94	11.05	2.38
org_parallel	131,072	94	2.37	1.14
MPE cluster	131,072	94	23.04	0.99
CPE cluster	17,832	245	1.37	0.12

method, and p times memory accesses in the cluster method. And there are $\sum_{i=1}^{n} q_i$ times distance calculations and $\sum_{i=1}^{n} Q_i$ times force evaluation in a general method, whereas only p times distance calculations and $\sum_{i=1}^{n} Q_i$ times force evaluation in the cluster method. Therefore, the theoretical ratio of calculation time between a general method and a cluster method is presented in Equation 10

$$ratio = \frac{MemAccess\left(\sum_{i=1}^{n} q_i\right) + Distance\left(\sum_{i=1}^{n} q_i\right) + Force\left(\sum_{i=1}^{n} Q_i\right)}{MemAccess(p) + Distance(n \times p) + Force\left(\sum_{i=1}^{n} Q_i\right)}. \quad (8)$$

The cluster method reduces the time of MemAccess while increasing the time of Distance. In the case of using cache, miss rate is low because of reordering. Afterward, the increase of memory accessing performance cannot offset the time costs on distance calculation. But, in the SPM case, the performance of data exchange is related to data size (Table 2 shows test results of data exchange bandwidth). Because the randomness of neighbor particles' distribution and the SPM is without an auto address mapping mechanism, the data of only a single neighbor particle will be brought in the SPM every time (about 32 bytes), which makes the memory accessing the main limitation of performance. And the cluster method can relieve this problem.

As shown in Table 3, the size of neighbor list is significantly declined, and the performance of neighbor list building is improved with the cluster method on MPE. As for the force calculation, the performance of the cluster method on MPE is lower than the general method because the MPE has caches. But, the performance of the cluster method is higher than the general method on CPEs, which uses SPM.

5 | OPTIMIZATION

5.1 | Adaption codes to CPE cluster

The CPEs provide the most efficient calculation power of computing nodes. However, CPEs engineered with their own methods and a special accelerate thread library, typical applications are unable to be run on CPEs without manual modifications. Transplanting kernel codes to CPEs is the first step to achieve our goal.

The *athread* library is specially designed for CPE programming. The intent is to control and schedule workloads among CPEs. The library bonds threads to CPEs. It is a one-one mapping between threads and CPEs, and the allocation and management of CPEs should be based on the usages of threads.

Algorithm 5 Org neighbor list builder call

Require: Force function id array *blist*;
force function array *pair_build*;
neighbor list id arrays *blist*; neighbor list arrays *lists*;
number of different forces *nblist*; cutoff distance r_{cut}.
Ensure: Neighbor lists.
1: **for** $i = 0; i < nblist; i$++ **do**
2: (this→ *pair_build[blist[i]])(lists[blist[i]])
3: **end for**
4: **return**

Algorithm 6 New neighbor list builder call

Require: Force function id array *blist*;
force function array *pair_build*;
neighbor list id arrays *blist*; neighbor list arrays *lists*;
number of different forces *nblist*; cutoff distance r_{cut}.
Ensure: Neighbor lists.
1: athread_init();
2: athread_set_num_threads(NUM_THRADS);
3: neigh_parameter para;
4: **for** $i = 0; i < nblist; i$++ **do**
5: para← parameters of neigh list building
6: athread_spawn(neigh_Slave,¶);
7: athread_join();
8: **end for**
9: athread_halt();
10: **return**

As the example showed in Algorithms 5 and 6, general CPE jobs are brought up by the MPE. Program needs to initialize CPEs, by calling athread_init function, prior to starting any tasks. Function athread_set_num_threads(int num) sets the maximum available number of threads to “num”. Function athread_spawn launches *num* CPE threads, and each thread is bonded to one CPE. Each thread will start from the function pointer, which is supplied as the first parameter of function athread_spawn. The second parameter of athread_spawn is passed to CPE threads as the only parameter. It is typically a pointer pointing to a parameter structure. Function athread_join will block MPE threads until all CPE threads have been accomplished. After all the tasks of threads groups in CPEs have been completed, "athread_halt()" must be called to terminate CPE cores.

5.2 | Parallelization

Since a CPE cluster contains 64 processors, the parallelization is an essential process. In the perspective of our algorithm, neighbor list building and force calculation modules calculate the distances and forces between pairs. It also gets input from an array for storing particle's locations and writes data into a neighbor list or a force array in accordance with particle's id. There are no location updating and no data collision (there will exist data collision if using the Newton's third law, but we give it up due to the low memory access efficiency) during the process. It is a normal SIMD process without any data-dependent collision. The codes get intrinsic parallelism. We take a fine-grained data level parallelization. Particles are assigned to different threads at the top iteration of a parallel region.

Parallelized code snippet is called parallel region. The entire parallel region is parallelized, and no serial part is left. So, the efficiency of parallelization can be measured by Equation (9)

$$parallelEff = \frac{S}{N_{cpu}}, \tag{9}$$

where N_{cpu} refers the number of CPU cores, and S refers the speedup.

Under the aforementioned parallelization method, the program costs T running time on a single core, then the time consuming with *CoreNumber* cores should be $T/CoreNumber$ if an extra parallelization cost is ignored. Sign memory access costs as P, total memory access for N times iteration is

TABLE 4 Parallel efficiency in 131k particles and 1m particles systems

	131k		1m	
	Time, s	Speedup	Time, s	Speedup
1	10.09	1.00	80.60	1.00
2	5.17	1.95	41.68	1.93
4	2.76	3.65	22.93	3.52
8	1.46	6.90	11.81	6.83
16	0.69	14.61	5.28	15.27
32	0.50	20.31	4.69	17.19
64	0.42	23.99	4.33	18.61

$P \times N$. Then, the speedup of the parallelization method can be calculated by Equation (10)

$$\begin{aligned} Speedup &= \frac{SerialTime}{ParallelTime} \\ &= \frac{T}{T/CoreNumber + P \times N}. \end{aligned} \tag{10}$$

Table 4 shows the parallel efficiency in two different size systems. A parallel slowdown shows up when core number reaches to 8 and becomes even more forceful with the rising of the core number. As there is no data collision among CPE cores. Therefore, the issue of performance loss should be the communication problems. Next, we concentrated on reducing memory access costs in CPE parallel region.

5.3 | Memory access optimization

CPE cores do not apply the traditional design method, which apply caches to remit the performance imbalance between calculation and memory access. Instead, the CPE replaces caches with the SPM. The novel design supplies more subjective initiative compared to the cache way. However, it also brings more problems about programming complexity. The compiling system of the Sunway TaihuLight supplies collective communication interface, DMA instruction, in C language to support asynchronous transmission between the SPM and the main memory.

In particular, the DMA operations can only be set up by the CPE. The length of read and write instruction is 128 bytes. When data is aligned with integral multiple of 128 bytes, DMA operations will reach the best performance. There are five kinds of DMA modes, ie, a single CPE mode, a broadcast mode, a single row mode, a row broadcast mode, and an array mode.

In both processes of force calculation and neighbor list building, all particles are traversed in the outermost iteration of the parallel region, and then particles are allocated to different threads in this layer. In each single secondary iteration, there are $2 \times numNeighbor$ location array reading operations and one force array writing operation. All these memory accesses only corresponded to specific particles. Since there is no definable data sharing among threads, all data transfer is taken place under a single CPE mode. Comparing to reading data from main memory directly, the DMA way yields a certain degree of performance improvement. But, various problems still exist in these models. Neighbor list building and force calculation read memory in a random pattern and are determined by the distribution of the particles. There are several methods sorting particles every several time steps to keep a relative order of particles. Such improvements provide extra performance with hardware caching mechanism. However, as the continuity is physical, it cannot be asserted that the address of accessed particles will always be continuous in memory. So, the reordering offers a minor improvement in SPM mechanism. Irregularity memory access puts forward high demands on memory performance; however, memory access is a short board of the system, and the short fetch size (4 double variables each time) further limits the performance of the play. To overcome this defect, we utilize an approach of asynchronies of DMA. Hence, the time of transfer is hidden by taking a manual prefetching.

5.4 | Manual prefetching

In CPE parallel region, especially in the neighbor-list building part, memory accessing often largely restrict the parallel efficiency. The manual prefetch is realized by taking advantage of an intrinsic characteristic of asynchronous DMA. Furthermore, the parallel efficiency is improved.

A double-buffering mechanism is adopted. The double-buffering mechanism is a technique designed to hide memory-access performance. When there are multi-cycle DMA read/write operations, CPEs allocate double size memory space in the SPM to accommodate two sets of data. These two sets are buffered for each other. When one set is calculated, the other one serves as a message buffer. During the process, memory accessed time and calculation time is overlapped, and the time cost is reduced by half while the operation costs are similar in time consuming. Fortunately, the simulation does not take much memory space. As analyzed in Section 5.3, communication size is limited in four double variables (32 bytes).

Algorithm 7 Programming framework prefetching

Require: Location array *atom*; computing range *start*, *end*;
cutoff distance r_{cut}.
1: $r_{cut}^2 \leftarrow r_{cut} * r_{cut}$;
2: **for** *i* ranges from *start* to *end* **do**
3: $atom_i \leftarrow atom[i]$;
4: $dataId \leftarrow$ getIndex(0)
5: DMA_get(j(*dataId*), *atom*[0], reply(getIndex(0)));
6: **for** $jId \leftarrow 1; jId < nNeighbor[i]; jId{+}{+}$ **do**
7: $dataId \leftarrow$getIndex(*jId*);
8: DMA_get(j(*dataId*), *atom*[*jId*], reply(*dataId*));
9: DMA_barrier(reply(*dataId*));
10: calculate with prefetched data;
11: **end for**
12: DMA_barrier(reply(getIndex(*jId*-1)))
13: calculate with final data;
14: **end for**
15: **return**

Double-buffering mechanism is realized in software way. By adding practical experience, the mechanism gets a robust programming framework. As an example in Algorithm 7, buffered variables are extended from $j(x, y, z)$ to $j(0 \;:\; 1, x, y, z)$. This is usually an identification to distinguish current smoothed data and prefetching data. Inline function getIndex() is considered to get the identification based on cyclic variable. The first data prefetching is per-performed and the final calculation is being picked out for correctness.

In this approach, the run time of CPE is divided into an overlapped part and an unsheltered part. The unsheltered part is the process to get data for the first turn and calculation for the final turn. All of the rest is the overlapped part. Signing communication time in a single cycle is indicated as p, and computing time is labeled as q. Overlapped part-time is defined as $\max(p \times (N-1), q \times (N-1))$. The speedup of the CPE is defined in Equation (9)

$$SpeedUp = \frac{T}{max(p \times (N-1), q \times (N-1)) + p + q}. \tag{11}$$

The speedup equation indicates that a double-buffering mechanism can be realized through mutually overlapping between computing and communication. The parallel region that has a longer computing time should benefit more from the mechanism. Figure 4 shows the parallel efficiency before and after the data transfer optimization in two different sizes. Parallel efficiency is obviously improved with data transfer optimization.

5.5 | Vectorization

Each CPE integrates a 256-bit wide SIMD floating point unit, of which the instruction pipeline is fixed as long as 8 stages. Each CPE core can process 8 floating point operations in a single instruction cycle. However, this puts forward more requirements about programming. There should be eight or more independent instructions to fulfill the pipeline. Although enough float operations are available in the innermost loop, there are dependencies among each other. As a result, original automatic vectorization of compiler system cannot produce an efficient binary file. To overcome the problem,

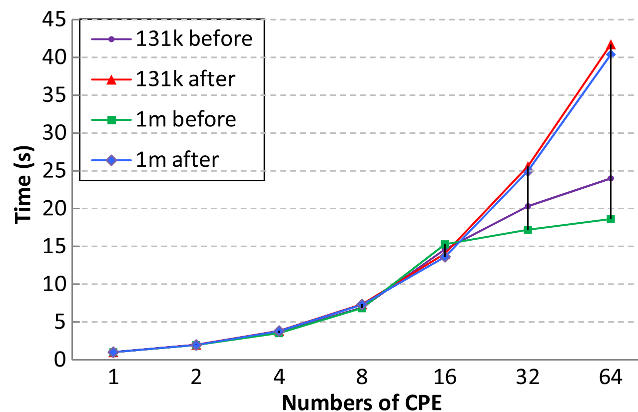

FIGURE 4 Speedup changes with different number of CPEs

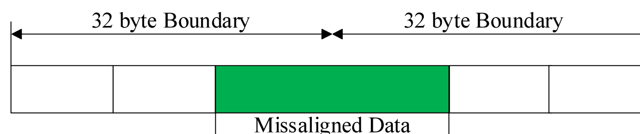

FIGURE 5 Memory-access alignment

we take an inter-loop vectorization[35] method to expand the innermost loop manually and make use of function simd_load and simd_store at the same time to further optimize fetch. Because CPU's vectorization size is just 256 bit (4 double), there is no need for intra-loop optimization. Each loop will fetch position of eight atomics in the neighbor table and reorder them for subsequent calculations. SW26010 processor offers a specialized SIMD data processing unit and corresponding instructions. It can satisfy the reordering needs of reading data.

While variable mapping operations occur in function, simd_load or simd_store, it is needed to ensure that the standard type of variable is 32 bytes boundary alignment (for floatv4, only 16 bytes boundary alignment).

As compiler environment does not provide a specific function interface for boundary alignment memory allocation, as seen in Figure 5, it is necessary to manually create spaces and reposition in the array initial position to guarantee the alignment at the start position in a block of memory. In the actual process, the method of data padding is taken to make each atomic structure filled just as 32 bytes(from three elements to four), and make every memory-access natural alignment.

6 | PERFORMANCE

6.1 | Experimental setup

For the Lennard-Jones simulations, the LJ benchmark of LAMMPS is taken into practice. Parameters are described in dimensionless units. Our initial configurations consist of four sizes, which scale from 131m to 536m particles. Benchmark simulations are engineered using the microcanonical ensemble with a series of parameter, ie, a cutoff distance of 2.5σ, a neighbor skin of 0.3 σ, and a reduced density of 0.8442 for each 100 time steps in liquid simulations. The size and time steps of EAM benchmark is the same as the LJ benchmark, whereas the EAM benchmark takes a cutoff of 4.95 ANG and a neighbor skin of 1.0 ANG for metal simulations with a lattice vector length of 3.84406. In CPU-only simulations, forces for ghost atoms are communicated in order to save computational time (this is a default setting in the LJ benchmark). For accelerated simulations, if two interacting particles are on different processors, then both processors compute their interaction and resulting to force information that is not communicated. This allows our implementation of full neighbor lists without special treatment for ghost atoms.

All of the experiments are carried out on the Sunway TaihuLight to validate the adaptability of our algorithm and optimization on the Sunway system. An overview of system configuration is showed in Table 5.

The memory-access bandwidth of CPEs is tested. The measured performance is showed in Table 2. Bandwidth differs with different fetch sizes and CPE numbers. The more data delivered from each fetch, the better memory access performed. The peak performance of a single CPE is 10.60 GB/s, and 30.94 GB/s for the entire 64 CPE clusters.

6.2 | Performance evaluation

To evaluate the performance of the particle-cluster algorithm proposed in Section 4, we design an experiment to compare an original version and an optimized version step by step with different system sizes.

This part mainly illustrate the performance results about each optimized step proposed in Section 5. To underpin the reliability and scalability, there are two force types and data sizes are tested, and per node vertices numbers are set as 131k and 1m. The test case runs on a single node (4 CGs).

TABLE 5 The Sunway TaihuLight supercomputer system configuration

CPU	SW26010 Processor
Instruction set	Shenwei-64 Instruction Set
Node processor cores	256 CPEs 4 MPEs
Memory	8 GB DDR3 per CG
OS	Sunway Raise OS 2.0.5 (based on Linux)
Compile language	C, C++, Fortran

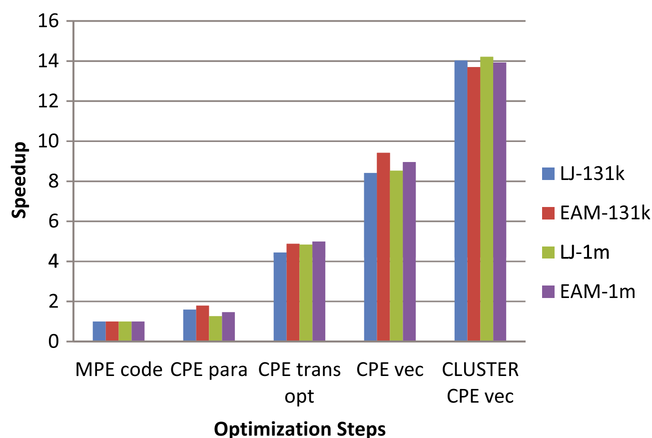

FIGURE 6 Speedup with different optimization steps

The speedup of every optimized step is illustrated in Figure 6. Performance increment is stable between both system sizes (131k vs 1m) and both benchmarks (LJ vs EAM). Comparing MPE code and CLUSTER CPE vec, we can easily draw a conclusion that more than 14x speedup in the configuration is achieved according to our optimized algorithm in experiments. The performance changes among different kinds of cutoff distance and particle density, as these two parameters determine the ratio of calculation and memory access. Especially, the speedup increases drastically through the operations of CPE trans opt (memory-access optimizations), CPE vec (vectorization), and CLUSTER CPE vec (cluster optimized algorithm).

From the perspective of speedup, we make a great contribution on reducing memory-access costs on the data communication problems. Each density gets an optimum cutoff value, while the results have a requirement about the accuracy of cutoff distance. So, there should be a trade-off between the performance and correctness.

6.3 | Multi-node performance

The scalability of both strong and weak scalings about the code is measured. According to intrinsic attributes of strong and weak scalings, we only set the system scale size as a variable in our experiments. Physical parameters of the system are configured as same as the inner node test.

To test the strong scalability of the codes, we conduct the first experiment in a single node. As shown in Table 6, four different data sizes are tested, and the per particle numbers are 131m, 226m, 359m, and 539m. While for these different particle numbers, we also scale up node numbers to keep track with the scaling. We set that node numbers are proportional to particle numbers, and on an average, each node is responsible for 26k particles. The results demonstrate quite good parallelism efficiency. It has been shown that these total times of simulation are on the verge of 3.5 (from 3.50 to 3.57). The times mainly are the same among different nodes and particle numbers, which means our code can be extended with a strong scaling capability. According to the intrinsic attributes of architecture, there will be a decline during small scales and become stable when reach a relatively large scale. Meanwhile, we need to achieve more raw experimental data to support our theory, especially experiences on large scales.

The weak scalability of the codes is checked by testing same data size simulated with different node numbers. To evaluate the efficiency of our application, we use three atom group of different data sizes, ie, 131m, 226m and 537m. We scale the simulation size from 2.25k to 22.5k to verify the weak scalability of our code. The results about the MD simulation of different particle groups are shown in Figure 7. Universally, the statistical result of most code tests shows amazing reduction of execution time when employing more cores in experiments. It has been shown that the performance of simulation is dramatically improved due to the increasing computer power of core. In particular, allowing for data size communication problem, data transfer in small scales can be different from the execution in large scales.

TABLE 6 Parallelism efficiency on multi-nodes

Particles	Nodes	Total	Force	neigh	comm	Other
131m	5,000	3.53	1.90	0.81	0.43	0.39
226m	8,640	3.52	1.87	0.79	0.49	0.37
359m	13,720	3.57	1.87	0.83	0.53	0.34
536m	20,480	3.50	1.88	0.83	0.42	0.37

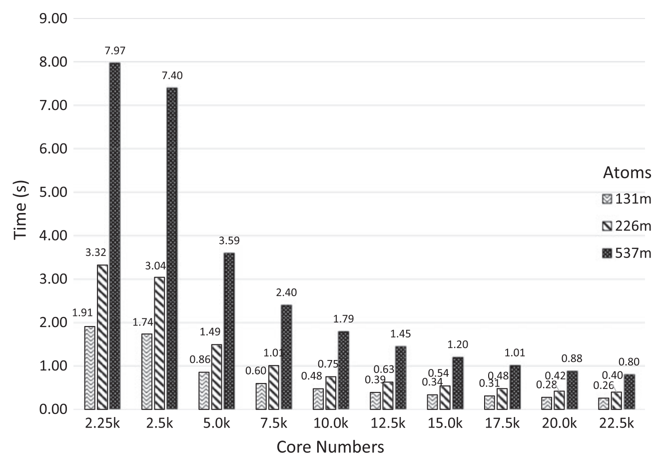

FIGURE 7 Times changes with node numbers

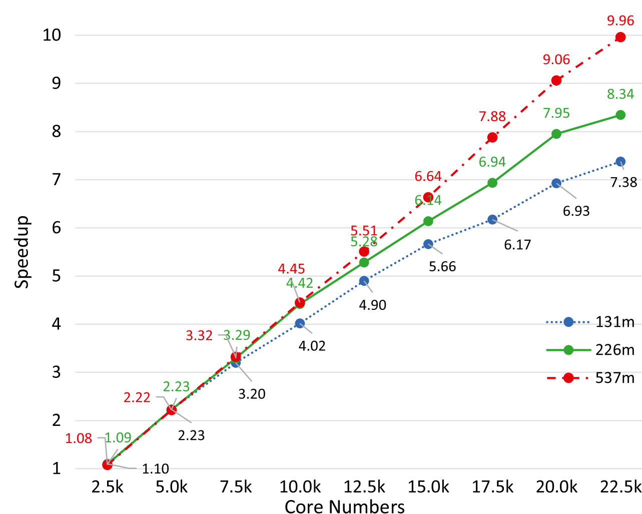

FIGURE 8 Speedup changes with node numbers

However, we still achieve a great performance on large scales, according to the proportion of the core number. In addition, the phenomenon illustrates our algorithm, and optimizations alleviate the data translation. This is particularly significant to evaluate the efficiency and adaptability of the weak scalability of our code.

We calculate speedup of multi-node execution, on the basis of 2.25k cores. In order to increase the effective scale of the experiments, we run 10 sets of groups with a range of a 10-fold span. Shown in Figure 8, with the nodes number scaling from 2.5k to 22.5k, is an almost linear speedup about weak scalability. All experiments show that our optimized operations and the cluster algorithm can be effectively adapted and extended on the TaihuLight system. We also believe that the weak scaling speedup will maintain the linear state when the system size is still scaling up. In each vector of atom gradation, this is an obvious tendency of linear growth about speedup. Especially, the larger scale of the particle size is, the more outstanding performance we achieve.

To make our experimental results more visualized, we normalize our speedup radio through a normalization percentage. According to the proportion of codes' scale, the normalization percentage N_i is calculated by

$$N_i = \frac{Speedup_{measure}}{S_i/S} \times 100\%. \tag{12}$$

In this equation, S_i means the core number that used to be normalized, and S is an initial size used to calculate the relative radio of scale. We set 2.25k as the initial size in our experiments. For example, as the ratio of the current core number (22.5k) and the initial size is 10, the normalization

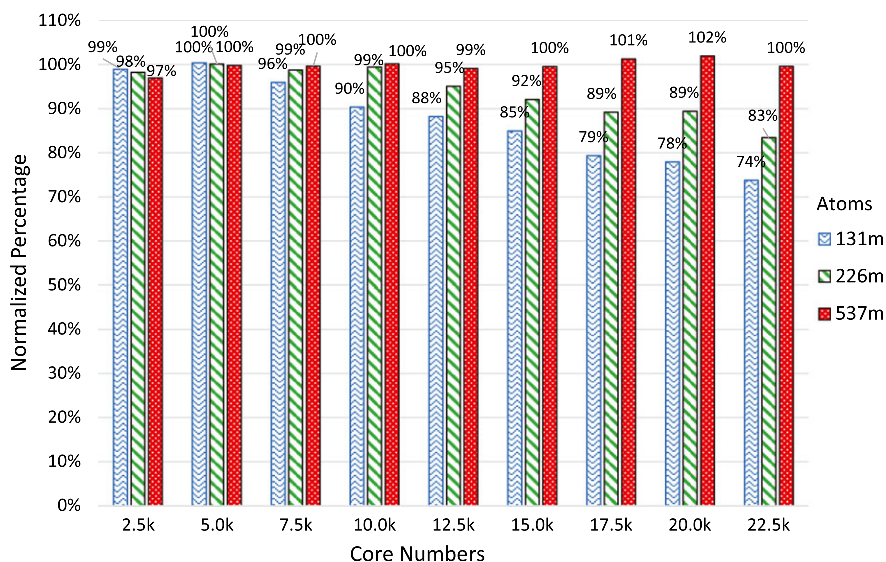

FIGURE 9 Speedup normalization

percentage of 537m atoms at the core scale of 22.5k is the quotient of the measured speedup (showed in Figure 8) and the initial size of 10. From Figure 9, the normalization percentage supports us a precise and straightforward view to appraise the performance of speedup. The Figure illustrates that the biggest size of atom number perfectly fits the linear tendency, even exceeds the expectation of linear speedup sometimes (at the scale of 17.5k and 20.0k). In contrast, small sizes show a decline of speedup normalization with core number scales. So far, we account the decline for the penalty of data communication. It does not mean that the time of communication is not decreased in larger scales, whereas the communication time is not reduced sharply with increasing the work nodes by almost one order of magnitude. While for the part of calculation, ie, the force calculation and neighbor list building, efficient computing power is supported without energy losses dramatically. Consequently, for compute-intensive applications, the effect is less pronounced in the data translation buffer and is negligible for the store buffer.

7 | CONCLUSIONS

In this paper, an optimized implementation of MD simulation is proposed to adapt the specific architecture of the Sunway TaihuLight supercomputer system. Our implementation uses a novel algorithm to alleviate the penalty of date transform and heavy computational load, which provides effective performance in the MD simulation. At the same time, we extended our particle-cluster algorithm with several optimized steps, ie, parallelization, memory-access optimization, and vectorization. The particle-cluster algorithm yields significant performance and energy improvements. By transplanting the entire developed application into the existing system, the optimization of short range potential is realized, which gains a 14x speedup in a single node and a nearly linear speedup among nodes. Meanwhile, our experiments show the platform of Taihulight system that provide outstanding performance for compute-intensive applications. Furthermore, our current work provides instructive information for other scientific applications to be implemented on the Sunway TaihuLight system or otherwise heterogeneous platform. In the future work, we will further perfect the application and extend our research and optimization about the long-range situations of MD. In addition, the Sunway Taihulight system owns up to 40,960 SW26010 processors, and it is promising to facilitate a deep investigation and utilization to the power of supercomputer to realize a larger scale implementation of MD simulation.

ACKNOWLEDGMENTS

A primary version of this paper was presented on the 18th IEEE International Conference on High Performance Computing and Communications (HPCC 2016),[36] 12-14 December 2016, Sydney, Astralia. The research was partially funded by the Key Program of National Natural Science Foundation of China (Grant 61432005), the National Outstanding Youth Science Program of National Natural Science Foundation of China (Grant 61625202), and the National Natural Science Foundation for the Youth of China (Grant 61602166).

ORCID

Wenqian Dong http://orcid.org/0000-0001-8823-4757

REFERENCES

1. Plimpton S. Fast parallel algorithms for short-range molecular dynamics. *J Comput Phys*. 1995;117(1):1-19.
2. Glikman E, Kelson I, Doan NV, Tietze H. An optimized algorithm for molecular dynamics simulation of large-scale systems. *J Comput Phys*. 1996;124(1):85-92.
3. Verlet L. Computer “experiments” on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules. *Phys Rev*. 1967;159(1):98.
4. Eastwood JW, Hockney RW. *Computer Simulation Using Particles*. New York, NY: Mc GrawHill; 1981.
5. Rapaport DC, Blumberg RL, McKay SR, Wolfgang C. The art of molecular dynamics simulation. *Comput Phys*. 1996;10(5):456-456.
6. Top500. Top500 supercomputer site. https://www.top500.org/lists/
7. Yang C, Xue W, Fu H, et al. 10m-core scalable fully-implicit solver for nonhydrostatic atmospheric dynamics. Paper presented at: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis; 2016; Salt Lake City, Utah.
8. Fu H, Liao J, Yang J, et al. The Sunway TaihuLight supercomputer: system and applications. *Science China Inf Sci*. 2016;59(7):072001.
9. Dongarra J. Report on the Sunway TaihuLight System [Technical Report]. Knoxville, TN: University of Tennessee; 2016. www.netlib.org
10. Mattson W, Rice BM. Near-neighbor calculations using a modified cell-linked list method. *Comput Phys Commun*. 1999;119(2-3):135-148.
11. Plimpton SJ, Hendrickson BA. Parallel molecular dynamics with the embedded atom method. *MRS Online Proc Libr*. 1992;291:37.
12. Phillips JC, Zheng G, Kumar S, Kalé LV. NAMD: Biomolecular simulation on thousands of processors. Paper presented at: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing; 2002; Baltimore, MD.
13. Shaw DE, Dror RO, Salmon JK, et al. Millisecond-scale molecular dynamics simulations on Anton. Paper presented at: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis; 2009; Portland, OR.
14. Lomdahl PS, Tamayo P, Gronbech-Jensen N, Beazley DM. 50 GFlops molecular dynamics on the Connection Machine 5. Paper presented at: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing; 1993; Portland, OR.
15. Glosli JN, Richards DF, Caspersen KJ, Rudd RE, Gunnels JA, Streitz FH. Extending stability beyond CPU millennium: A micron-scale atomistic simulation of Kelvin-Helmholtz instability. Paper presented at: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing; 2007; Reno, NV.
16. Trott CR. $\text{Lammps}_{\text{cuda}}$-A New GPU Accelerated Molecular Dynamics Simulations Package and Its Application to Ion-Conducting Glasses [PhD thesis]. Ilmenau, Germany: Technische Universität Ilmenau; 2011.
17. Phillips JC, Stone JE, Schulten K. Adapting a message-driven parallel application to GPU-accelerated clusters. Paper presented at: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing; 2008; Austin, TX.
18. Ma M, Hou J, Ye J, Arunachalam M, Gutierrez R. Optimizing non-contiguous memory access on Intel Xeon Phi coprocessors. Paper presented at: Proceedings of the 2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), and 2015 IEEE 12th International Conference on Embedded Software and Systems (ICESS) (HPCC-CSS-ICESS'15); 2015; New York, NY.
19. Harode A, Gupta A, Mathew B, Rai N. Optimization of molecular dynamics application for Intel Xeon Phi coprocessor. Paper presented at: 2014 International Conference on High Performance Computing and Applications (ICHPCA); 2014; New York, NY.
20. Tarmyshov KB, Müller-Plathe F. Parallelizing a molecular dynamics algorithm on a multiprocessor workstation using OpenMP. *J Chem Inf Model*. 2005;45(6):1943-1952.
21. Couturier R, Chipot C. Parallel molecular dynamics using OpenMP on a shared memory machine. *Comput Phys Commun*. 2000;124(1):49-59.
22. Glaser J, Nguyen TD, Anderson JA, et al. Strong scaling of general-purpose molecular dynamics simulations on GPUs. *Comput Phys Commun*. 2015;192:97-107.
23. Procacci P. Hybrid MPI/OpenMP implementation of the ORAC molecular dynamics program for generalized ensemble and fast switching alchemical simulations. *J Chem Inf Model*. 2016;56(6):1117-1121.
24. Jung J, Mori T, Sugita Y. Midpoint cell method for hybrid (MPI+OpenMP) parallelization of molecular dynamics simulations. *J Comput Chem*. 2014;35(14):1064-1072.
25. Pal A, Agarwala A, Raha S, Bhattacharya B. Performance metrics in a hybrid MPI–OpenMP based molecular dynamics simulation with short-range interactions. *J Parallel Distrib Comput*. 2014;74(3):2203-2214.
26. Kunaseth M, Richards DF, Glosli JN, Kalia RK, Nakano A, Vashishta P. Analysis of scalable data-privatization threading algorithms for hybrid MPI/OpenMP parallelization of molecular dynamics. *J Supercomput*. 2013;66(1):406-430.
27. Wu Q, Yang C, Tang T, Xiao L. MIC acceleration of short-range molecular dynamics simulations. Paper presented at: Proceedings of the First International Workshop on Code Optimisation for Multi and Many Cores; 2013; Shenzhen, China.
28. Anderson JA, Lorenz CD, Travesset A. General purpose molecular dynamics simulations fully implemented on graphics processing units. *J Comput Phys*. 2008;227(10):5342-5359.
29. Germann TC, Kadau K, Lomdahl PS. 25 Tflop/s multibillion-atom molecular dynamics simulations and visualization/analysis on BlueGene/L. Paper presented at: Proceedings of Supercomputing (SC05); 2005; Washington, US.
30. Narumi T, Susukita R, Koishi T, et al. 1.34 Tflops molecular dynamics simulation for NaCl with a special-purpose computer: MDM. Paper presented at: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing; 2000; Dallas, TX.
31. Scarpazza DP, Ierardi DJ, Lerer AK, et al. Extending the generality of molecular dynamics simulations on a special-purpose machine. Paper presented at: 2013 IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS); 2013; Boston, MA.
32. Shaw DE, Grossman J, Bank JA, et al. Anton 2: raising the bar for performance and programmability in a special-purpose molecular dynamics supercomputer. Paper presented at: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis; 2014; New Orleans, LA.

33. Kikugawa G, Apostolov R, Kamiya N, et al. Application of MDGRAPE-3, a special purpose board for molecular dynamics simulations, to periodic biomolecular systems. *J Comput Chem*. 2009;30(1):110-118.
34. Quentrec B, Brot C. New method for searching for neighbors in molecular dynamics computations. *J Comput Phys*. 1973;13(3):430-432.
35. Pennycook SJ, Hughes CJ, Smelyanskiy M, Jarvis SA. Exploring SIMD for molecular dynamics, using Intel® Xeon® processors and Intel® Xeon Phi coprocessors. Paper presented at: 2013 IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS); 2013; Boston, MA.
36. Dong W, Kang L, Quan Z, et al. Implementing molecular dynamics simulation on Sunway TaihuLight system. Paper presented at: 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS); 2016; Sydney, Australia.

How to cite this article: Dong W, Li K, Kang L, Quan Z, Li K. Implementing molecular dynamics simulation on the Sunway TaihuLight system with heterogeneous many-core processors. *Concurrency Computat Pract Exper*. 2018;e4468. https://doi.org/10.1002/cpe.4468