SW_GROMACS: Accelerate GROMACS on Sunway TaihuLight

Tingjian Zhang[1,4], Yuxuan Li[2,4], Ping Gao[1,4], Qi Shao[1,4], Mingshan Shao[1,4]
Meng Zhang[1,4], Jinxiao Zhang[1], Xiaohui Duan[1,4], Zhao Liu[2,4], Lin Gan[2,4]
Haohuan Fu[3,4], Wei Xue[2,4], Weiguo Liu[1,4], Guangwen Yang[2,4]

1. School of Software, Shandong University, China
2. Department of Computer Science and Technology, Tsinghua University, China
3. Ministry of Education Key Lab for Earth System Modeling, and Department of Earth System Science, Tsinghua University, China
4. National Supercomputer Center in Wuxi, China

ABSTRACT

GROMACS is one of the most popular Molecular Dynamic (MD) applications and is widely used in the field of chemical and bimolecular system study. Similar to other MD applications, it needs long run-time for large-scale simulations. Therefore, many high performance platforms have been employed to accelerate it, such as Knights Landing (KNL), Cell Processor, Graphics Processing Unit (GPU) and so on. As the third fastest supercomputer in the world, Sunway TaihuLight contains 40960 *SW26010* processors and *SW26010* is a typical many-core processor. To make full use of the superior computation ability of TaihuLight, we port GROMACS to *SW26010* with following new strategies: (1) a new deferred update strategy; (2) a new update mark strategy; (3) a full pipeline acceleration. Furthermore, we redesign GROMACS to enable all possible vectorization. Experiments show that our implementation achieves better performance than both Intel KNL and Nvidia P100 GPU when using appropriate number of SW26010 processors for a fair comparison.

CCS CONCEPTS

• **Computing methodologies → Shared memory algorithms**; **Vector / streaming algorithms**; *Self-organization*; Massively parallel algorithms; • **Applied computing → Chemistry**.

ACM Reference Format:
Tingjian Zhang, Yuxuan Li, Ping Gao, Qi Shao, Mingshan Shao, Meng Zhang, Jinxiao Zhang, Xiaohui Duan, Zhao Liu, Lin Gan, Haohuan Fu, Wei Xue, Weiguo Liu and Guangwen Yang. 2019. SW_GROMACS: Accelerate GROMACS on Sunway TaihuLight. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3295500.3356190

1 INTRODUCTION

Molecular dynamics (MD) simulation [14] is a very popular application on supercomputers [2]. Various MD applications have been used in different fields such as materials science, chemistry and biology. Computer simulation methods are widely adopted in those applications to simulate the movements of molecules and particles according to the Newtonian equations of motion. There are many frequently-used MD software applications, such as LAMMPS[21], AMBER [23] and so on. Most of them have been well implemented in many different platforms, such as CPU [22], GPU [28], KNL[15].

As one of the most popular MD application, GROMACS is mainly designed for the simulation of proteins, liquid, and nucleic acids where there exist lots of complicated bonded interactions between the molecules. But because of its extremely fast calculation of the non-bonded interactions, more and more research groups use it to simulate non-biological systems. Like some other MD applications, it is a free open-source software application, which is supported by many different groups.

Till now, it has been a pressing need for a long time to accelerate GROMACS on TaihuLight. Now, we implement *SW_GROMACS* on TaihuLight by rebuilding the code of GROMACS.

TaihuLight [11] is a supercomputer with a peak performance of 125.3 PFLOPS. It is composed of 40960 *SW26010* chips, which are placed in 40 cabinets and connected by a 2-level fat-tree topology network[27].

Each *SW26010* includes four core groups (CGs), which are connected via the network on chip (NoC). Each CG is composed of one Management Processing Element (MPE) and 64 Computing Processing Elements (CPEs) arranged in an 8 * 8 grid. The MPE is designed for handling management and communication functions. It contains 8 G DDR3 memory with a 32 KB L1 instruction cache, a 32 KB L1 data cache and a 256 KB L2 cache for both data and instruction. In contrast, the CPE is a Reduced Instruction Set Computer (RISC) core with only 64 KB fast local device memory (LDM). The 64 CPEs is arrayed as an 8 * 8 mesh structure. CPEs in the same row or column could transmit data with the communication bus rapidly. All CPEs could access the MPE memory by DMA, which could get the data in a contiguous region of the memory efficiently. Otherwise, CPEs have to access parameters in MPE memory by global load/store instructions (gld/gst) with high latency.

As for the computation ability, both the MPE and CPEs could use the 256-bit vector instructions, with a 1.45 GHZ running frequency. Every chip could provide a peak performance of 3.06 TFlops.

In fact, it is hard to make full use of *SW26010*. There are many constraints for porting GROMACS to *SW26010*, for example, the weak MPE, the small LDM in CPE, the low bandwidth of DMA on TaihuLight and so on. To overcome those challenges, we come

SC '19, November 17–22, 2019, Denver, CO, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6229-0/19/11…$15.00
https://doi.org/10.1145/3295500.3356190

MD workflow	Kernel	Case1	Case2
Initialize	Domain decomp.	NULL	0.7%
Calculate interaction	Neighbor search	2.5%	2.3%
	Force	95.5%	74.8%
	Wait + comm. F	NULL	1.1%
	NB X/F buffer ops	0.1%	0.2%
Update configuration	Update	0.3%	0.2%
	Constraints	0.6%	1.7%
	Comm. energies	NULL	18.7%
Output	Write traj	0.5%	0.1%
	Rest	0.5%	NULL

Table 1: Time ratio of different kernels in the two case. The case 1 is 48,000 particles water case with 1 CG. And Case 2 is the 3,000,000 particles water case is simulated by 512 CGs. And the detail of the benchmark will be introduce in the evaluation section

up with some strategies in our *SW_GROMACS* to reduce those restrictions of *SW26010*. Those works will be introduced in this paper:

(1). Designing a new data structure for the calculation of short-range interaction in the GROMACS. proposing deferred update strategy .

(2). In the original method, to avoid the write conflict every CPE is assigned a array to accumulate the interaction of all atoms. However, the method require the initial of the arrays and reduction, which cost lots of time. SO we come up with the update mark method to reduce meaningless transmission in the calculation step and the reduction step and desert the the initialization step.

(3). To make full use of the calculation of *SW26010*. We carefully optimize the vectorization of the short-range interaction calculation. We change the data layout and use the simd operation to accelerate the calculation as fast as possible.

(4). After the optimization of the calculation of the short-range interaction. The other parts of the GROMACS occupy too much time. So that, the Optimization of other important steps of the workflow is of great importance to achieve better entirety performance.

In this paper, we will introduce the basic information of GROMACS and some previous work in Section 2. And in Section 3, some of our optimization works in our *SW_GROMACS* will be introduced. The performance of *SW_GROMACS* will be discussed in Section 4. In the end, we talk about some conclusions and future work in Section 5.

2 BACKGROUND

In this section, the basic algorithm of GROMACS and some related work are given as follows.

2.1 Algorithms

The algorithm used in GROMACS is similar to other MD applications. As shown in Figure 1, the workflow of GROMACS consists of initial conditions input, forces computation, configuration update, and result output. In most cases, the calculation of interactions is the most time-consuming portion (as we can see in the Table 1). So we mainly talk about the calculation of particles interactions in

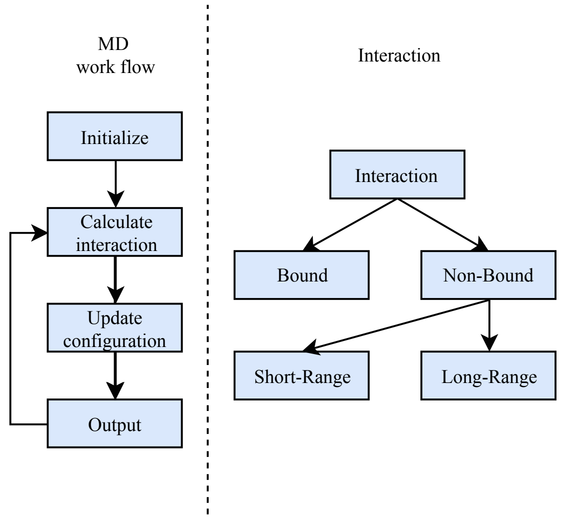

Figure 1: The left side of the figure is a standard workflow of MD applications. The application will repeat calculate interaction step, update configuration step, output step as they need. And the right side is the specific classifications of the interaction force.

this section. And the detail of the algorithm could be seen at [2] [6] [25] [18].

As shown in Figure 1, there are different kinds of interactions between particles, which can be divided into the bounded interaction and the non-bounded interaction. The non-bounded interaction contains not only short-range interaction but also long-range interaction.

In the calculation of short-range interactions, GROMACS employs a pair list that contains those particle pairs for which non-bounded interactions must be calculated. The pair list contains all the particle pairs that the distance between them is within $R_{cut-list}$. The non-bounded interaction should be calculated only if the distance between them r_{ij} is less than the given cut-off radius $R_{cut-off}$ ($R_{cut-off} < R_{cut-list}$). Because the position of the particles is constantly changing, we should regenerate the neighbor list every *nstlist* steps to ensure that all possible pairs are in the list, where the *nstlist* is typically 10.

After that, the short-range interaction can be calculated according to the pair list. The calculation is mainly based on the Lennard-Jones (L-J) interaction [13]. In the L-J interaction, the potential V_{LJ} between two particles can be defined in Equation (1). The r_{ij} means the distance of atoms. The parameters C_{ij}^{12} and C_{ij}^{6} depend on pairs of atom types.

$$V_{LJ}(r_{ij}) = \frac{C_{ij}^{12}}{r_{ij}^{12}} - \frac{C_{ij}^{6}}{r_{ij}^{6}} \tag{1}$$

So the force F_{ij} between the two particles could be calculated by Equation (2).

$$F_i(\mathrm{d}r_{ij}) = (12\frac{C_{ij}^{12}}{r_{ij}^{13}} - 6\frac{C_{ij}^{6}}{r_{ij}^{7}}) - \frac{\mathrm{d}r_{ij}}{r_{ij}} \tag{2}$$

As the calculation process shown in Algorithm 1, we will calculate the interactions of those particle pairs where the distance between particles in the pair is within $R_{cut-off}$. Then the interactions have been calculated will be added to the interaction data of two particles (as shown in Line 9 and Line 13 of Algorithm 1). In the implementation of GROMACS, particles in the $Particles\_List$ and the $Neighbor\_List$ are not contiguous in the memory, so the data of different atoms could not be got at one time. The fine-grained frequently memory access will make the memory access worse.

Algorithm 1 Calculate the Short-Range Force

```
Require:
    $FETCH$ : get the data of particles from main memory;
    $UPDATE$ : update the interaction (force) of particles in main memory;
    $CAL\_F$ : calculate the interaction (force) between two particles;
    $Particle\_List$ : particles we need to traverse;
    $Neighbor\_List$ : Neighbor lists of different particles, it is a half neigbor list;
    $F$ : the array store the interaction (force) of every particle;
 1: for $A_{particle} \in Particles\_List$ do
 2:     $FETCH(DATA(A_{particle}))$
 3:     $FETCH(Neighbor(A_{particle}))$
 4:     $F_A = 0$
 5:     for $B_{particle} \in Neighbor(A_{particle})$ do
 6:         if $DISTANCE(A_{particle}, B_{particle}) < R_{cut-off}$ then
 7:             $FETCH(DATA(B_{particle}))$
 8:             $F_{A,B} = CAL\_FORCE(DATA(A_{particle}), DATA(B_{particle}))$
 9:             $UPDATE\_FORCE(F(B_{particle}), F_{A,B})$
10:             $F_A = F_A + F_{A,B}$
11:         end if
12:     end for
13:     $UPDATE\_FORCE(F(A_{particle}), F_A)$
14: end for
```

As for MD, the calculation of short-range interactions can suffer from inaccuracy because of the existence of the long-range interaction. To improve the accuracy of computation, GROMACS incorporates some lattice sum methods such as Ewald Sum method [12], Particle Mesh Ewald(PME) method [10] and Particle-Particle Particle-Mesh(PPPM) method [5]. Among these methods, PME is used because of its low computational complexity. To parallelize PME, the Fast Fourier Transformation(FFT)[26] is supposed to be used in many processes, causing heavy-duty communication.

As for the bounded interactions, the calculation is based on a fixed list of particles. And they are not only the pair interaction but also the bound interaction include 3 and 4-body interaction. There are bond stretch (2-body), bound angle (3-body) and dihedral angle (4-body) interactions.

2.2 Previous work on optimizing GROMACS

MD has been a popular research field since its inception. These years, many useful optimization methods [24] for different platforms have been proposed for different MD applications. These platforms range from the laptops to the supercomputers [2] [9]. The processors include CPU, GPU [3] [4], KNL [19], MIC [20], *SW26010* [8] [7] and so on. In this subsection, we will introduce the related work about MD applications.

With the appearance and development of various processors, porting MD applications to different processors with fine-grained optimization has attracted wide attention. Many different strategies have been proposed in the past two decades to accelerate the calculation of the short-range interaction, which is the most time-consuming kernel in most MD applications. We will introduce some related strategies in this subsection.

In 2013, a flexible algorithm [22] has been used to accelerate the calculation of pair interactions on SIMD architectures. In this algorithm, GROMACS could group a fixed number of particles, e.g. 2, 4, or 8, into spatial clusters. By this way, GROMACS calculates all interactions between particles in a pair of such clusters, which will improve data reuse compared to the traditional scheme and result in a more efficient SIMD parallelization.

Another most common challenge is to solve the write-write conflict in the process of updating interactions between particles. For instance, if different cores update the interactions of different particles in the shared memory, the write-write conflicts will occur.

To tackle the problem, different methods have been proposed. In 2007, to port GROMACS to cell [17], each core keeps an interaction array for every particle. The core will add the interactions to its own interaction arrays, which will avoid write-write conflicts. In the end, the original interactions of every particle could be updated by the interaction arrays kept by those cores. It is a very simple way to solve the write-write conflicts. But the initialization step and the CPE updating step may occupy too much time and bring bad effects on the performance. After that, a more complex strategy has been used in *SW26010* in 2015 [29]. In this strategy, the idle MPE is used to receive the interaction calculated by CPEs and add them to the interaction of different particles. At that time, only the MPE could update the interaction data. The write-write conflict will disappear. However, the implementation is so complex and the idle time always exists in CPEs and MPE because of the unbalance computation ability between CPEs and MPE. And we will talk about it in detail later. Both of those two strategies are used to parallelize the short-range interaction calculation, however, with non-negligible performance loss. Therefore, it motivates us to develop a more effective strategy to tackle the challenge and implement it on *SW26010*.

3 OPTIMIZATION

We first port the code of GROMACS on TaihuLight and complied GROMACS for the *SW26010*. As the architecture of *SW26010* shown before, every CG of *SW26010* supports one MPI thread. Then, we find the calculation of the short-range interaction in GROMACS is the most time-consuming part. As we can see from Table 1, the time spend on the Force kernel (the calculation of short-range interaction) is more than 90% in the Case 1 and about 75% in the Case 2.

The algorithm used to compute the short-range interaction is shown in Algorithm 1. The algorithm is partitioned by the outer loop (in Line 1 of Algorithm 1) across all CPEs. There exist frequent memory accesses to get the data of particles to calculate the short-range interaction. However, in CPEs, the LDM is too small, only 64 KB, to keep the data of all the particles, which means CPEs have to access the MPE memory frequently to fetch the essential data. However, it can hardly meet the requirements because of the low bandwidth between CPEs and the MPE. So the low bandwidth

Access Data Size	DMA Bandwidth
8 B	0.99 GB/s
128 B	15.77 GB/s
256 B	28.88 GB/s
512 B	28.98 GB/s
2048 B	30.48 GB/s

Table 2: The bandwidth change with the assess data size. The first line is the memory access size. The second line is the bandwidth changes according to the memory access size.

between CPEs and MPE becomes a key bottleneck for our . Thus, to make full use of the computation ability of *SW26010*, we should firstly reduce the restriction of memory bandwidth. Then, we will also accelerate the part of calculation, communication, I/O, and so on. The detailed optimization methods are listed in this section.

3.1 Fetch Strategy

During the computation process of short-range interactions, many data elements of different particles are needed, for example, the position, type, and amount of charge, which are placed in different arrays. Among all these elements, apart from the interaction and the position coordinate of each particle in 3-dimensional space, all the other elements are not stored in a contiguous area of memory.

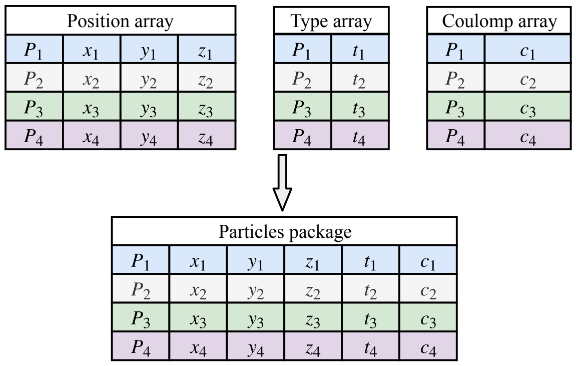
Position array
Type array
Coulomp array
Particles package
P_1 x_1 y_1 z_1 t_1 c_1
P_2 x_2 y_2 z_2 t_2 c_2
P_3 x_3 y_3 z_3 t_3 c_3
P_4 x_4 y_4 z_4 t_4 c_4

Figure 2: Aggregate the data from different arrays to a particle package. The P means different particles. The x, y, z means the three different position elements. The t means the type of this particle. The C mean the column of the particle.

To achieve the peak bandwidth and reduce the frequency of memory access, we try to aggregate the different data elements of the same particle as a new data structure. As mentioned in the background section, in GROMACS, every four contiguous particles are put in one group and particles in the same group is always calculated simultaneously. So we could aggregate the data of four particles in one structure. We call it the particle package and it will increase the size of the particle package as well as reduce the memory access frequency. Finally, we get a particle package as shown in Figure 2.

In this way, the data block size for one access increases from 4 B to 108 B, and as shown in Table 2, the bandwidth in our *SW_GROMACS* increases from less than 0.99 GB/s to almost 15.77 GB/s. Moreover, we can get the data of four particles pipeline, reduce multiple memory access and avoid the DMA conflicts.

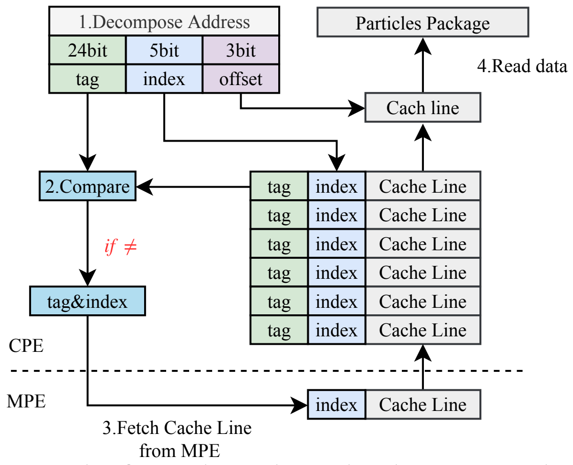

Figure 3: This figure shows the read cache operation during the short-range interaction calculation. As it shows that we will first decompose the index id into tag number, line number and offset number. The tag number means a unique id of a cache line in MPE. The line number is the cache line index in CPE. And the offset number is the address of the particle in the cache line. The second step is to compare the tag of the line with the original line's tag. If they are the same, it means that the cache line is what we want. So we could do the fourth step, fetching data and calculating. And if that tag is not the same with the original one, which means cache miss, we have to do the third step, fetching data. And then do the fourth step.

As mentioned above, the cache architecture in *SW26010* is not integrated as other platforms. As similar to another work on *SW26010* [8], we design a read cache strategy to fetch particles data from the main memory. The operation of the read cache is shown in Figure 3. As a result, the data of different particles could be reused and we could fetch eight particle packages in pipeline. In this way, the accessing block size in one DMA is about 800 B and the peak bandwidth is almost achieved, according to the Table 2.

3.2 Deferred Update

As shown in Algorithm 1, after every calculation of particle pairs, the interaction of B particle will be updated. It means every calculation will occupy an interaction update, which is too frequent for the low bandwidth between MPE and CPEs. In the previous work on *SW26010* [8], the redundant computation approach Algorithm 2 [16] [8] has been used. They change the generation of pair list and make every particle pair exits in both neighbor lists of the two particles. As Algorithm 2 shows, it only updates the interactions of A particles in Line 10, which will reduce the update frequency. However, this strategy will double the computation and the memory access load.

This challenge has been addressed in a different way. We find that many particles are reused in different inner loops (Line 5 of Algorithm 1). It means a certain particle may be updated by a CPE more than once. So the changes of different particles could be accumulated in the CPE and the interaction of this particles in main memory could be updated in one time.

So we have every CPE keep a certain size of LDM as the update buffer to accumulate the interaction changes of every particle. In the update buffer, every particle will map a certain address. The change of every particle's interaction will firstly be accumulated in the update buffer instead of in the main memory. The update of interaction in main memory will only occupy at the time that a particle in the update buffer will be replaced by another. We call this strategy as deferred update. To implement this strategy efficiently, we use the ways simulated to the direct-map cache. Every data change is base on eight particle package. For convenience, we also call it cache line. And the detail operation is shown in Figure 4. In this way, many DMA access could be reduced to one access. Thus we will achieve better DMA performance.

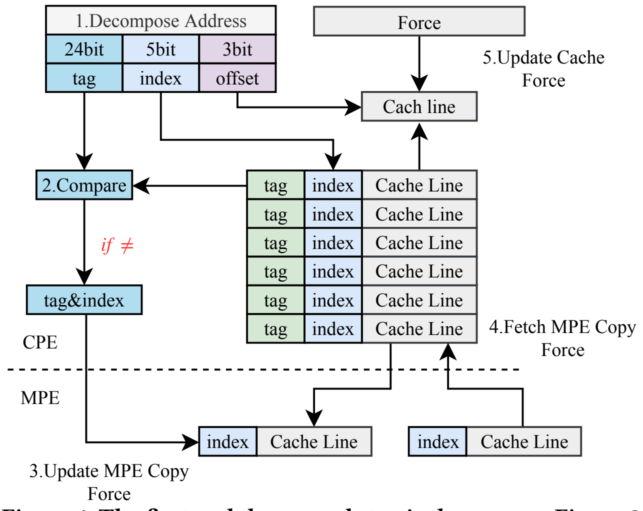

Figure 4: The first and the second step is the same as Figure 3. If the tag is the same as the original one, we can update the data in the update buffer. While if the tag is different from the original one. CPEs should update the interaction in the main memory and fetch the interaction of this particle in the main memory.

3.3 Bit-Map

To parallelize the calculation of the short-range interaction, there exists another challenge, the write conflict. As we mentioned in the related work, there are some solutions for the write conflict.

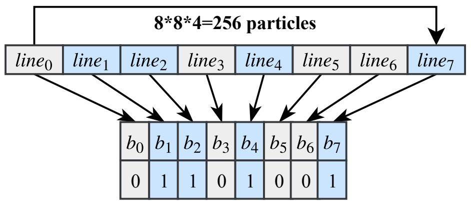
8*8*4=256 particles
$line_0$ $line_1$ $line_2$ $line_3$ $line_4$ $line_5$ $line_6$ $line_7$
b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7
0 1 1 0 1 0 0 1

Figure 5: We use each bit to mark the update state of a cache line. For 1 byte memory there are 8 bits. For one cache line, there is eight particle-package. So for one Byte size memory we could record the update state of 256 ($8 \times 8 \times 4$) particles.

Among them, we chose the approach of keeping an interaction array for every CPE. In this paper, this strategy is called as the redundant memory approach (*RMA*). The redundant interaction

Algorithm 2 Calculate the shorter-range interaction in RCA method

```
Require:
    FETCH : get the data from MPE;
    UPDATE : update the interaction data in MPE;
    CAL_F : calculate the interaction between two particles;
    Particle_List : particles we need to traverse;
    Neighbor_List : the neighbor particles of every particles in particle_List array, it is a full neighbor list;
    F : the array store the change data of interaction;
 1: for $A_{particle} \in Partic\_List$ do
 2:     $FETCH(DATA(A_{particle}))$
 3:     $FETCH(Neighbor(A_{particle}))$
 4:     $F_A = 0$
 5:     for $B_{particle} \in Neighbor(A_{particle})$ do
 6:         $FETCH(DATA(B_{particle}))$
 7:         $F_{A,B} = CAL\_F(DATA(A_{particle}), DATA(B_{particle}))$
 8:         $F_A = F_A + F_{A,B}$
 9:     end for
10:     $UPDATE(F(A_{particle}), F_A)$
11: end for
```

arrays are called as the copies of the original interaction. The steps to gather up copies, get the summation of them, and write back to memory are called as reduction step. To use RMA, all of the copies should be initialized before the calculation, which almost consumes the same time with calculation time. To achieve a better efficient DMA, we propose a new strategy, the Bit-Map strategy.

During the calculation process, most of the particles will update their interactions only in some of the 64 CPEs while few of them will update their interaction in all CPEs. If the data of a particle is not updated in some CPEs, the data copies of this particle in these CPEs will never change during the calculation. So the initialization step and the reduction step become meaningless for these particles' copies. We call those copies of particles meaningless copies. These meaningless copies exist widely, which occupy much time in initialization step and reduction step. The Bit-Map could reduce meaningless cost with little performance loss.

The main idea of the Bit-Map strategy is to record the update status of every copy in its own CPE. To save the memory and work with the deferred update strategy, every CPE will record the update status of every cache line in its copy. In this way, the initialization step could be deserted. The Algorithm 3 and Algorithm 4 show the update mark operation in the calculation step and the reduction step. As Line 8 of Algorithm 3 shows, if the cache line is not updated, the value of its data must be zero (the initialized value). So the data does not need to be fetched and can be set as zero at CPEs. At the reduction step, as Figure 5 shows, if a cache line is not updated, it will not need to be added to the original data. So it will not be fetched.

To implement the Bit-Map strategy, as we show in Algorithm 3, we use 1-bit memory in every CPE to mark the update status of a cache line, which includes eight particle-packages with 32 particles. It means an integer parameter could record the update status of 1024 particles. And all of those operations could be done by the bit operations as Algorithm 3 and Algorithm 4 show.

Algorithm 3 The deferred update behavior with mark

```
Require:
    $F$ : the interaction we calculate;
    $I$ : the index of the particles we update;
    $C\_L$ : the cache line array in CPE;
    $T\_C$ : the array store the tags of cache line in CPE;
    $C\_M$ : the array store the marks of cache line in CPE;
    $F\_MPE$ : the copy array of the interaction in MPE;
    $nLines \leftarrow 2^n$ : the number of cache lines;
    $Line \leftarrow 2^m$ : the long of the cache lines;
1:  $Tag \leftarrow I >> (n + m)$
2:  $Line \leftarrow (I\&(1 << (n + m) - 1)) >> m$
3:  $Offset \leftarrow I\&((1 << m) - 1)$
4:  $Cache\_Begin \leftarrow I >> m$
    calculate the tag, line index, offset and the cache address by bit
    operation.
5:  if $T\_C(Line) = Tag$ then
6:      $UPDATE(C\_L(line, Offset))$
7:  else
8:      if $T\_C(Line) \geq 0$ then
9:          $PUT\_MPE(C\_L(Line), F\_MPE(Cache\_Begin))$
            If the cache data has been updated the MPE data, put the
            data back to the MPE copies;
10:     end if
11:     if $(C\_M >> Cache\_Begin)\&1 = 1$ then
12:         $Address \leftarrow tag << n + Line$
13:         $FETCH\_MPE(F\_MPE(Address), C\_L(Line))$
            If the cache line has been updated, we fetch it from MPE;
14:     else
15:         $INIT(C\_L(Line))$
16:         $C\_M = C\_M|(1 << Cache\_Begin)$
            If the cache line has not been updated, we just initial it,
            and make the mark 1;
17:     end if
18: end if
```

Algorithm 4 The reduction step with mark

```
Require:
    $Cache\_Line\_cnt$ : the number of cache lines;
    $Force$ : the interaction array we should update;
    $Force\_Copy$ : the copy of interaction array;
    $CPE\_Num$ : the number of CPE;
    $Cache\_Mark$ : the marks of Cache line we store
    $F$ : the arrays in CPE to calculate the update of interaction;
1:  for $n \leftarrow 1$ to $Cache\_Line\_cnt$ do
2:      $INIT(F)$
        initial the array we will use;
3:      for $m \leftarrow 1$ to $CPE\_Num$ do
4:          if $Cache\_Mark(m) >> n = 1$ then
5:              $F\_GET \leftarrow FETCH(Force\_Copy(m, n))$
6:              $F \leftarrow F + F\_GET$
                If the mark of the cache line is 1, we fetch and reduce it.
                If it is 0, we do not fetch it.
7:          end if
8:      end for
9:  end for
```

3.4 Vectorization

Since the memory access has been optimized carefully, we obtain good performance in the short-range interaction kernel. As a result, the computation in this kernel now becomes the new hot spot. So we try to vectorize it. In *SW26010*, CPEs support 256-bit SIMD vector registers. It supports floatv4 which could calculate four floats once.

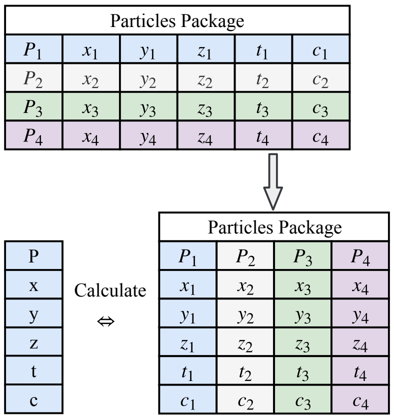

Figure 6: The data layout has been changed to make the same position elements contiguous.

In GROMACS, there exist challenges to implement vectorization efficiently, because some of operations could not be accelerated by vectorization easily. In GROMACS, the particles in the outer loop (Line 1 of Algorithm 1) are fixed, while the particles in the inner loops (Line 5 of Algorithm 1) are often changing. With this in mind, it will be more efficient to vectorize every four particles in the outer loop and compute the interactions with the particles in the inner loop.

After the vectorization, the pre-treatment and the post-treatment seem to occupy lots of time. At the pre-treatment step, every four floats should be transformed into a floatv4 parameter for the later computation. In the original particle package, the same element of different particles is not contiguous, which makes the elements unable to be fetched and transformed into vectors efficiently. As shown in Figure 6, we change the data layout to make them contiguous, which could accelerate the pre-treatment step.

In the post-treatment, the vector should be transformed into four float numbers and added to the three position elements. To perform the summation operation more efficiently, we design a convert operation including six *simd_vshulff* operations to transform the vector. (The *simd_vshulff*, one of the fastest SIMD instructions, could combine two vectors into a new one. It chooses two float numbers in the first vector as the first two float numbers of the new vector and the other two float numbers of the new vector are from the second vector.) As shown in Figure 7, the vector could be added to the arrays without decomposition, so that the post-treatment can be more efficient.

3.5 Acceleration of the Pair Lists Generation

After the calculation of the short-range interaction has been carefully optimized, the establishment of the pair list becomes the new hottest spot. As is introduced in the background section, the pair

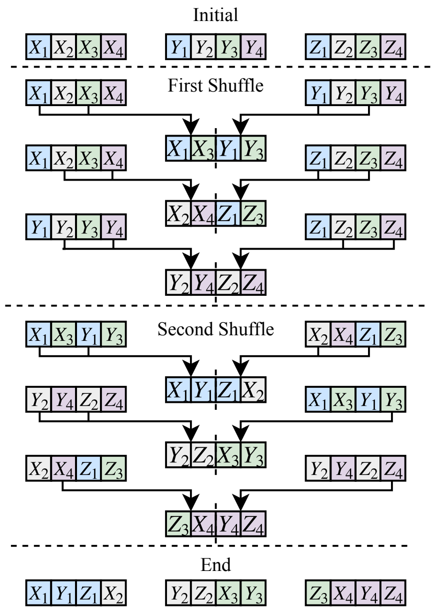

Figure 7: In the post-treatment we use the vector shuffle in CPEs to make three position elements of the same particle continuous. And we spend six simd operations on it as shown above.

list will be regenerated in every *nslist* step. On account of its complicated code, researchers seldom accelerate the establishment of the pair list by CPEs, which inspires us to tackle the challenge and accelerate it.

For the pair list in GROMACS, it comprises the neighbor lists of every particle. For every particle, it keeps the start and the end index of its neighbors. To implement this in a many cores system, different cores will generate the neighbor lists of different particles. Because of the different length of different neighbor lists, it is impossible to get the start index of the first neighbor list in a CPE. To solve the problem, every CPE keeps a temporary memory in the main memory to store neighbor lists which are calculated by corresponding CPEs. Finally, the pair list will be formed by gathering all these neighbor lists. The start and end index of every particle's neighbor list are calculated at the same time.

What's more, During the establishment of the neighbor list, it needs to access memory randomly for lots of things, which is something like the memory access in calculation kernel.
In the calculation of short-range interaction, the performance of the direct-map cache is excellent. Most of the time, the cache miss ratio is less than 10%. But in the neighbor list establishment kernel, the performance of the direct-map cache is undesirable. The cache miss ratio is more than 85%, because of serious cache thrashing. To eliminate the cache thrashing, the two-way associative Cache has been used in this kernel. By this means, we make the achievement of reducing the cache miss ratio from more than 85% to 10%.

3.6 Acceleration of Communication

In GROMACS, the overhead of th e communication increases with the number of processes and the communication is high frequency with small message size. To accelerate communication, we re-implement communication by RDMA.

At the sender side of the MPI communication, the application will first create the data to be sent in the user space. Then the data in the user space will be copied to kernel space, where the data is packed into a new TCP segment(a data packet). After that, the Network Interface Card (NIC) will copy the packet from the kernel memory to its own memory and then send it via the network. At the receiver side, it will receive the packet in NIC, and then move the data from device memory to kernel memory. The kernel will unpack the packet into data. Finally, the data will be copied from kernel memory to user memory. Following these steps above, the data has to be copied four times and we have to pay extra CPU time to the packing and unpacking operation.

However, the memory of a computer could access the memory of another computer directly by RMDA technology. It moves the data without any memory copying and the kernel time of the CPU. When an application performs an RDMA communication, the data will be delivered to NIC and NIC sent the data to network directly from the user memory. And receiver application could get the data from the NIC directly. All those behavior could be done without CPUs, caches or context Switching. And there is no memory copy and Kernel bypass in RMDA which can't be avoided in MPI. Compare to the MPI the RDMA could get deliver the data more quickly.

3.7 Some Other Optimization

Under some circumstances, the users may ask for the position of every particle. In this case, GROMACS have to output the particles position file, of which the size is huge. Therefore, sometimes, the I/O step occupies lots of time, which motivates us to accelerate the I/O operation. In the Large-scale case, the I/O cost almost accounts for 30% of the overall run time. In the original code, GROMACS use the $fwrite$ and $fread$ function to do the I/O operation. To accelerate it, these functions are reimplemented by *read* and *write* with a 20M buffer, which is much quicker than before. After that, however, there is still a lot of time spent on I/O mainly because of the format function, which plays the role of converting double data elements to characters for subsequent printing. To accelerate these steps, we develop another function to transform the float data into characters and then generate the output data. In the new implementation, concise methods are adopted to convert data type, from double, float or integer type to character type. Compared to the C standard library, it saves so much time in dealing with special cases such as illegal input, other format requests and so on. In these ways, the time spent on I/O has been significantly reduced with little accuracy sacrifice.

In fact, for most memory access, if the data address is in the alignment of 128 bit, the memory access tends to be more efficient than before. To achieve better performance, we make the address of all parameters and arrays in the alignment of 128bit. By this means, we speed up the entirety performance significantly.

3.8 Portability of Our Optimization

Although most of our optimizations are designed for *SW26010*, many of them could also be used on other platforms. Firstly, the cache and deferred update could be used in some platforms that lack for an integrated memory architecture. The calculation ability of these platforms is always restricted by memory bandwidth. By this way, those platforms achieve the peak bandwidth. Secondly, the update mark strategy could also work in different many-core processors, multi-core processors and even GPU. In many Parallel optimizations, the original methods to deal with the write conflict problem always bring lots of performance loss. Someone use the multi-copy strategy, which we introduce in the related work. But the time spent on the initialization step and reduction step always restrict the performance of this strategy. Our update mark could reduce those time, and it could be widely used in many different platforms. Thirdly, the optimization in I/O could be used in many other platforms to solve the huge I/O problem.

4 EVALUATION

In this section, we will evaluate the performance of our optimization of GROMACS in TaihuLight. And the evaluations include Benchmark, The performance of our optimization on Short-range Interaction Calculation, Comparison with Other Strategies, The entirety Performance, Comparison with Different Platforms, Accuracy and Scalability.

4.1 Benchmark

Our work is based on the version of the GROMACS 5.1.5 [6]. And we use the water case [1] as the standard case to evaluate the performance of GROMACS in different platforms. To get better performance, we use the mixed precision in every platforms to do the evaluation. And the input parameter is show in Table 3.

Key Variable	Value
$particles\ number$	$0.9K \backsim 3,000K$
$nstlist$	10
ns_type	$grid$
$coulombtype$	PME
$rlist$	1.0
$coulombtype$	100
$cutoffscheme$	$verlet$

Table 3: The benchmark of the case

4.2 The performance of our optimization on Short-range Interaction Calculation

As we mentioned above, the short-range calculation is the most time-consuming kernel. So we evaluate the performance of our optimization on it. As we show in the optimization section, we have come out with lots of novel optimizations to accelerate short-range calculation. In the first optimization step, we just use data aggregation to accelerate it. At that time, we just get 3 times speed-up. The calculation is restricted by the memory access bandwidth. By the write cache and the read cache, we partly reduce the restriction in memory access. We get 20 times speed-up by it. As we evaluate that the cache-miss rate in both write cache and read cache are under 15%. The DMA bandwidth is more than 30 G/s in each CG, which almost achieve the theoretical peak bandwidth. The vectorization optimization will reduce the calculation time and speed up the calculation almost 2 times from the cache version. Finally, we decrease the lots of meaningless translation by the update mark. We get another 2 times speed-up compared to the last version. We accelerate the short-range calculation, the Lennard-Jones potential, 64 times. The different cases shown in Figure 8 seems that the speed-up ratio will not change by the number of particles in each CG.

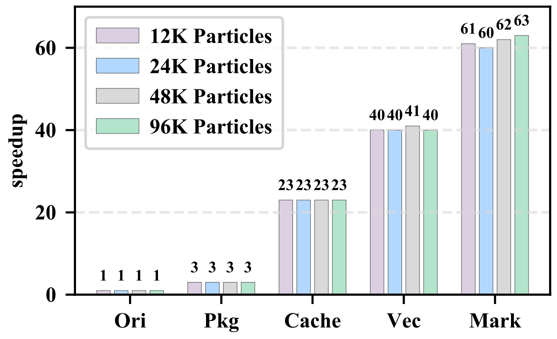

Figure 8: The original is the original version of the GROMACS. It just runs on the MPE. The Pkg is the version uses the data aggregation. The Cache is the version that is implemented with the read & write cache. The Vec is the version accelerates computation by vectorization. The last version Mark is the version uses update mark strategy.

4.3 Comparison with Other Strategies

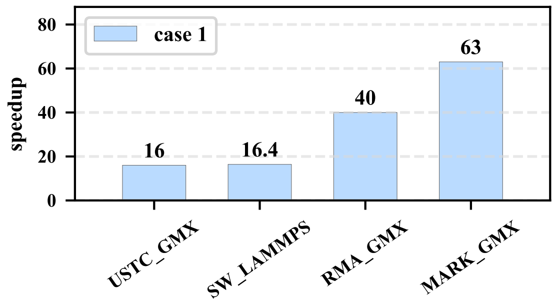

Figure 9: The speed-up of different strategies. The version USTC_GMX is the version that is implemented by USTC on *SW26010* [29]. The version SW_LAMMPS is the LAMMPS implemented on *SW26010* with the RCA strategy [8]. The version RMA_GMX is the GMX that is accelerated by us with the RMA strategy. The MARK_GMX is the version implemented by our update mark strategy.

To accelerate the calculation of the short-range interaction in different many-core accelerators, the write conflict has to be solved and many different strategies have been proposed before. Compared with those previous strategies, the strategy we come up with in this paper is a more efficient way to deal with the write conflict. We will discuss those strategies in this subsection.

One of previous strategies has been used in Power Cell processor [17], the work we introduce in the related work. They overcome the write conflict by having every core accumulate local interaction totals for all particles, which is something like the strategy we implement without update mark. As mentioned above, this strategy contains lots of meaningless transmissions. The update mark strategy we come up with achieves almost 2 times speedup compare to it Figure 8. After that, a more complex method is proposed. People from USTC use MPE to collect the interaction calculated by CPEs and update the interaction array, at the time of CPEs calculation. However, it is hard to strike a computation balance between CPEs and MPE. In the implementation of their GROMACS [29], they finally achieved 24 times speedup in the optimization of short-range calculation, which is far from our implementation. There is also another strategy, the RCA strategy, which is implemented on *SW26010* [8]. To avoid the write conflict, the pair list has been changed and all the interactions will be calculated twice and every core will only update the particles in the outer loop. As a result, every interaction will be calculated twice and particles every core update are totally different. It is obvious that this strategy doubles the computation, which causes performance loss. Finally, int the L-J interaction performance, they get 24 times speed-up.

As for our implementation, by the update strategy, the initiation step is deserted and the reduction time is only about the 1.2% of the calculation time. So the performance loss is very small. We finally achieve 63 times speed up in the L-J (short-range) interaction. As shown in Figure 9, our strategy is much better than other strategies.

4.4 The Overall Performance

Besides the optimization on the calculation of the short-range interaction, we also do some other optimization in neighbor searching, I/O and communication. And because the performances of some optimization in different cases are different. So we use two cases in different scales to evaluate the performance better. The first case, which contains 48,000 particles, is in the single CG. The second case contains 3,072,000 particles and we use 512 CGs to simulate it. In the single CG case, most of the time spend on neighbor searching and short-force calculation. As Figure 10 shows that we could get a better speed-up ratio in version 1 and version 2. While optimizations in version 3 and version 4 seem useless. Especially communication optimization, since there is no communication in single CG simulation. While the time in 512 CG scale is spent in different aspects. So the speed-up ratio if case 2 in version 1 and version 2 is not as good as case 1. The speed-up ratio in version 3 and version 4 is better than case1. Finally, we get 32 times speed-up in case 1 and 18 times speed-up in case 2.

4.5 Comparison with Different Platforms

To compare the performance with other platforms, we use the case with more than 3,000,000 particles. And we compare our version with KNL and GPU. As for GPU, we have used P100. The platform information is shown in Table 4. As we mentioned above, the bandwidth of our implementation is more than 30 G/s, which almost achieve the peak bandwidth. It means that *SW_GROMACS* is restricted by the memory bandwidth as some other MD problems [8]. To evaluate the performance of our implementation, we have decided to use the time to fulfill (TTF) [8] value to do fair performance comparison with other platforms. We first compare *SW26010*

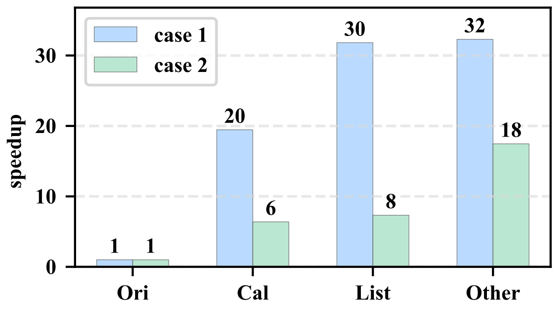

Figure 10: The performance of different optimizations. In case 1, one CG simulates about 48,000 particles. In case 2, 512 CGs simulate about 3,000,000 particles. The Ori version is the version without any optimization and simulates only by MPE. The Cal version is the version optimizes the calculation of short-range interaction. And List version optimizes the generation of pair list. The Other version contains other optimizations we implement.

with KNL. In KNL, every two cores have 1 MB L2 cache. The cache miss rate of L1 cache on KNL is about 2%, which is almost half of the cache miss rate on *SW26010*. And the L2 cache miss rate of KNL is less than 4%. So the total miss rate of the cache in KNL is less than 0.08%, which is about 2.5% of the cache miss rate on *SW26010*. And the bandwidth of the KNL is about three times of *SW26010*. Based on these number we can compute Equation (3), from it we can see that SW26010's TTF is about 150 times as many as KNL's. As for P100, the cache miss ratio of L1 is 6%. The cache miss ratio of its L2 is 15%. Thus the total cache miss ratio is about 0.9%. The bandwidth of P100 is 720 G/s. Based on these number we can compute Equation (4), from it we can see that SW26010's TTF is about 24 times as many as P100's.

	Knights Landing	*SW26010*	P100
Flops	6 T	3 T	10 T
Bandwidth	400 G/s	132 G/s	720 G/s
Cache	32 KB+1 MB	64 KB LDM	64 KB+4 MB

Table 4: The base information different platforms.

$$\frac{TTF_{SW}}{TTF_{KNL}} = \frac{\frac{LAA \cdot MR_{SW}}{BW_{SW}}}{\frac{LAA \cdot MR_{KNL}}{BW_{KNL}}} = \frac{MR_{SW} \cdot BW_{KNL}}{MR_{KNL} \cdot BW_{SW}} \approx 150 \quad (3)$$

$$\frac{TTF_{SW}}{TTF_{P100}} = \frac{\frac{LAA \cdot MR_{SW}}{BW_{SW}}}{\frac{LAA \cdot MR_{P100}}{BW_{P100}}} = \frac{MR_{SW} \cdot BW_{P100}}{MR_{P100} \cdot BW_{SW}} \approx 24 \quad (4)$$

Based on Equation (3) and Equation (4), Figure 11 shows performance comparisons for our implementation running on 150 SW26010, GROMACS 5.1.5 running on 1 KNL, and GROMACS 5.1.5 running on 1 P100. From it we can see that the performance of 150 *SW26010* is much better than 1 KNL. And the performance of 24 *SW26010* is also comparable with P100. We can also see that the scalability of our implementation is better than GROMACS 5.1.5 running on GPU (the performance of 48 *SW26010* is better than 2 P100s).

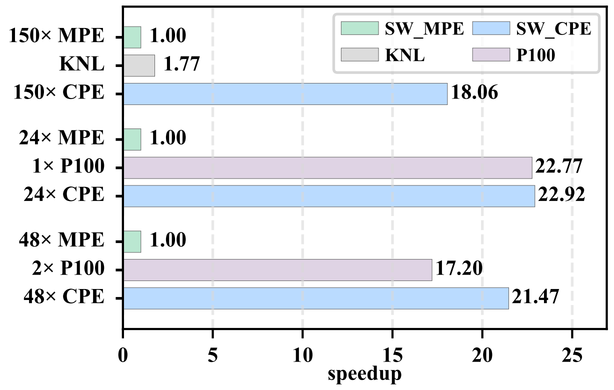

Figure 11: The performance of different platforms. The KNL and P100 means the performance of GROMACS in them. The MPE means the performance of GROMACS only in the MPE. The CPE means the version we accelerat GROMACS in CPEs. And all the performances are entirety performance.

4.6 Scalability

In this subsection, we will discuss our study on the scalability of our optimization version of GROMACS. In the evaluation, we use the water case with 48,000 particles as the case in the strong scalability test. And we expend our simulation from 4 CG to 512 CG. As for the weak scalability, we make each CG simulation over 10,000 particles and make the scale from 4 CG to 512 CG. To calculate the parallel efficiency, we use the two equation Equation (5) and Equation (6). In the equation Equation (5), the $Eff_{strongly}$ is the parallel efficiency of the strong scalability. The T_4 is the time we simulate the case1 by 4 CG (one *SW26010*). And in the equation Equation (6), the T_N is the time we simulate case1 by N CG. And the T_4 is the same as that in the equation Equation (5).

As we can see from Figure 12, we get great weak scalability. There is almost no performance loss as the scale increase at the beginning. As shown in Figure 12, the performance of strong scalability seems not very bad. The parallel efficiency goes down to 0.60 at 512 CGs. It seems that the number of particles in every CG is too small, so more time has to be spent on communication. It is very bad for the parallel efficiency.

$$Eff_{strongly}(N) = \frac{T_4}{\frac{N}{4} \cdot T_N} \quad (5)$$

$$Eff_{weakly}(N) = \frac{T_4}{T_N} \quad (6)$$

4.7 Accuracy

Because many optimizations have been used in our implementation, it may bring some changes in the result. So we should evaluate the accuracy of our implementation. As we show in Figure 13 We compare the result of *E5-2680-v3* and our final version on *SW26010*. The main parameters we compare is the temperature and the total energy of the simulation in every 100 steps in a 500,000 steps long test. Although there is some deviation between the outcome of *E5-2680-v3* and our implementation. But it seems that the deviation

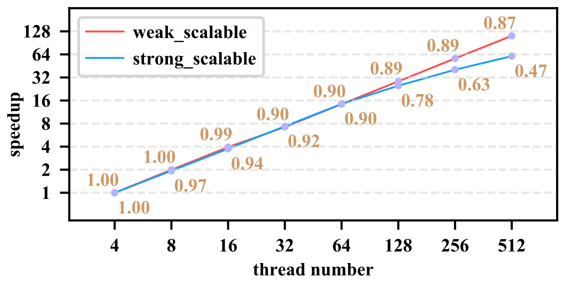

Figure 12: Weak & Strong Scalability

could be contained in a certain range and our implementation is stable enough to simulate a long-running step.

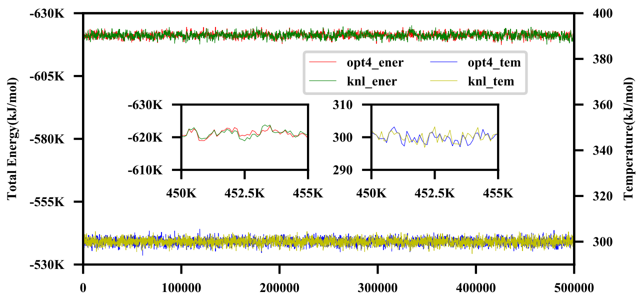

Figure 13: The energy and temperature difference between our version and X86 version

5 CONCLUSION AND FUTURE WORK

GROMACS is a classical scientific application as well as an excellent representative of various MD applications [25]. The calculation of the short-range interaction is the most frequently used kernel in GROMACS. To achieve an efficient implementation on *SW26010*, we have proposed a series of new strategies and achieved more than 60 times speedup for the calculation of the short-range interaction, which is much better than any other implementations of the short-range interaction on *SW26010*. Our strategies are general and could also be implemented in other manycore and multicore processors. Experiments show that our implementation achieves better performance than both Intel KNL and Nvidia P100 GPU when using appropriate number of SW26010 processors for a fair comparison.

6 ACKNOWLEDGEMENT

We would like to thank all the anonymous reviewers for their insightful comments and suggestions, also, my workmate Bin Yang from Shandong University provided a number of suggestions for developing GROMACS on the TaihuLight. This work is partially supported by the National Natural Science Foundation of China (Grant No. U1806205, 51761135015), National Key R&D Program of China (2017YFA0604500), Center for High Performance Computing and System Simulation, Pilot National Laboratory for Marine Science and Technology (Qingdao). The corresponding authors are Xiaohui Duan (sunrise.duan@mail.sdu.edu.cn), Haohuan Fu (haohuan@tsinghua.edu.cn), Wei Xue (xuewei@tsinghua.edu.cn), and Weiguo Liu (weiguo.liu@sdu.edu.cn).

REFERENCES

[1] [n. d.]. The Benchmark of water case. ftp://ftp.gromacs.org/pub/benchmarks/water_GMX50_bare.tar.gz.

[2] Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C Smith, Berk Hess, and Erik Lindahl. 2015. GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX* 1 (2015), 19–25.

[3] Sadaf Alam and Ugo Varetto. 2014. GROMACS on hybrid CPU-GPU and CPU-MIC clusters: Preliminary porting experiences, results and next steps.

[4] Joshua A Anderson, Chris D Lorenz, and Alex Travesset. 2008. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of computational physics* 227, 10 (2008), 5342–5359.

[5] Markus Deserno and Christian Holm. 1998. How to mesh up Ewald sums. II. An accurate error estimate for the particle–particle–particle-mesh algorithm. *The Journal of Chemical Physics* 109, 18 (1998), 7694–7701.

[6] GROMACS development team. [n. d.]. GROMACS 5.1.5 version. http://manual.gromacs.org/documentation/5.1.5/download.html.

[7] Wenqian Dong, Kenli Li, Letian Kang, Zhe Quan, and Keqin Li. 2018. Implementing molecular dynamics simulation on the Sunway TaihuLight system with heterogeneous many-core processors. *Concurrency and Computation: Practice and Experience* 30, 16 (2018), e4468.

[8] Xiaohui Duan, Ping Gao, Tingjian Zhang, Meng Zhang, Weiguo Liu, Wusheng Zhang, Wei Xue, Haohuan Fu, Lin Gan, Dexun Chen, et al. 2018. Redesigning LAMMPS for peta-scale and hundred-billion-atom simulation on Sunway TaihuLight. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 148–159.

[9] Maria Eleftheriou, Blake Fitch, Aleksandr Rayshubskiy, TJ Christopher Ward, and Robert Germain. 2005. Performance measurements of the 3d FFT on the Blue Gene/L supercomputer. In *European Conference on Parallel Processing*. Springer, 795–803.

[10] Ulrich Essmann, Lalith Perera, Max L Berkowitz, Tom Darden, Hsing Lee, and Lee G Pedersen. 1995. A smooth particle mesh Ewald method. *The Journal of chemical physics* 103, 19 (1995), 8577–8593.

[11] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Yang Chao, Xue Wei, Fangfang Liu, and Fangli Qiao. 2016. The Sunway Taihu Light supercomputer:system and applications. *Science China Information Sciences* 59, 7 (2016), 072001.

[12] Gerhard Hummer. 1995. The numerical accuracy of truncated Ewald sums for periodic systems with long-range Coulomb interactions. *Chemical physics letters* 235, 3-4 (1995), 297–302.

[13] Christian Kriebel, Matthias Mecke, Jochen Winkelmann, Jadran Vrabec, and Johann Fischer. 1998. An equation of state for dipolar two-center Lennard–Jones molecules and its application to refrigerants. *Fluid phase equilibria* 142, 1-2 (1998), 15–32.

[14] J Andrew McCammon, Bruce R Gelin, and Martin Karplus. 1977. Dynamics of folded proteins. *Nature* 267, 5612 (1977), 585.

[15] William McDoniel, Markus Höhnerbach, Rodrigo Canales, Ahmed E Ismail, and Paolo Bientinesi. 2017. LAMMPSâĂŹPPPM Long-Range Solver for the Second Generation Xeon Phi. In *International Supercomputing Conference*. Springer, 61–78.

[16] Trung Dac Nguyen. 2017. GPU-accelerated Tersoff potentials for massively parallel molecular dynamics simulations. *Computer Physics Communications* 212 (2017), 113–122.

[17] Stephen Olivier, Jan Prins, Jeff Derby, and Ken V. Vu. 2007. Porting the GROMACS Molecular Dynamics Code to the Cell Processor. In *IEEE International Parallel & Distributed Processing Symposium*.

[18] Szilárd Pall, Mark James Abraham, Carsten Kutzner, Berk Hess, and Erik Lindahl. 2014. Tackling exascale software challenges in molecular dynamics simulations with GROMACS. In *International Conference on Exascale Applications and Software*. Springer, 3–27.

[19] Conor Parks, Lei Huang, Yang Wang, and Doraiswami Ramkrishna. 2017. Accelerating multiple replica molecular dynamics simulations using the Intel® Xeon PhiâĎć coprocessor. *Molecular Simulation* 43, 9 (2017), 714–723.

[20] Shaoliang Peng, Xiaoyu Zhang, Yutong Lu, Xiangke Liao, Lu Kai, Canqun Yang, Liu Jie, Weiliang Zhu, and Dongqing Wei. 2017. mAMBER: A CPU/MIC collaborated parallel framework for AMBER on Tianhe-2 supercomputer. In *IEEE International Conference on Bioinformatics & Biomedicine*.

[21] Steve Plimpton. [n. d.]. lammps website. https://lammps.sandia.gov/index.html.

[22] SzilÃąrd PÃąll and Berk Hess. 2013. A flexible algorithm for calculating pair interactions on SIMD architectures. *Computer Physics Communications* 184, 12 (2013), 2641–2650.

[23] Romelia Salomon-Ferrer, David A Case, and Ross C Walker. 2013. An overview of the Amber biomolecular simulation package. *Wiley Interdisciplinary Reviews: Computational Molecular Science* 3, 2 (2013), 198–210.

[24] Frank Suits, MC Pitman, Jed W Pitera, William C Swope, and Robert S Germain. 2005. Overview of molecular dynamics techniques and early scientific results from the Blue Gene project. *IBM Journal of Research and Development* 49, 2.3 (2005), 475–487.

[25] David Van Der Spoel, Erik Lindahl, Berk Hess, Gerrit Groenhof, Alan E Mark, and Herman JC Berendsen. 2005. GROMACS: fast, flexible, and free. *Journal of computational chemistry* 26, 16 (2005), 1701–1718.

[26] Peter Welch. 1967. The use of fast Fourier transform for the estimation of power spectra: a method based on time averaging over short, modified periodograms. *IEEE Transactions on audio and electroacoustics* 15, 2 (1967), 70–73.

[27] Bin Yang, Xu Ji, Xiaosong Ma, Xiyang Wang, Tianyu Zhang, Xiupeng Zhu, Nosayba El-Sayed, Haidong Lan, Yibo Yang, Jidong Zhai, et al. 2019. End-to-end I/O Monitoring on a Leading Supercomputer. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 379–394.

[28] Juekuan Yang, Yujuan Wang, and Yunfei Chen. 2007. GPU accelerated molecular dynamics simulation of thermal conductivities. *J. Comput. Phys.* 221, 2 (2007), 799–804.

[29] Yang Yu, Hong An, Junshi Chen, Weihao Liang, Qingqing Xu, and Yong Chen. 2017. Pipelining Computation and Optimization Strategies for Scaling GROMACS on the Sunway Many-Core Processor. In *International Conference on Algorithms & Architectures for Parallel Processing*.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We accelerate the GROMACS v5.1.5 on Sunway TaihuLight supercomputer. As described in the paper, we use the case of water with different atom numbers. Every could get by: $ wget ftp://ftp.gromacs.org/pub/benchmarks/water_GMX50_bare.tar.gz This data includes example "water" data of different sizes with folder names like 0384, 0768, and 1536. The name of the folder corresponds to the number of atoms in thousands. And we to run it on the Taihu-light use the cmake command: LD=mpiCC CC=mpicc CXX=mpiCC cmake .. -DGMX_FFT_LIBRARY=fftpack -DGMX_MPI=on -DGMX_BUILD_MDRUN_ONLY=ON -DBUILD_SHARED_LIBS=off -LH bsub -I -n <thread number> bin/mdrun_mpi -s <case name> -v

ARTIFACT AVAILABILITY

Software Artifact Availability: Some author-created software artifacts are NOT maintained in a public repository or are NOT available under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: All author-created data artifacts are maintained in a public repository under an OSI-approved license.

Proprietary Artifacts: There are associated proprietary artifacts that are not created by the authors. Some author-created artifacts are proprietary.

List of URLs and/or DOIs where artifacts are available:

`https://github.com/BEYHHH/SW\_GROMAC`

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Sunway TaihuLight supercomputer, with SW26010 processor

Operating systems and versions: Customized Linux with kernel version 3.8.0

Compilers and versions: SWCC Compilers: Version 5.421-sw-500

Applications and versions: GROMACS 5.1.5

Libraries and versions: athread/mvapich-2.2

Key algorithms: molecular dynamics

Input datasets and versions: The water case support by GROMACS

Paper Modifications: Makelist , docemdo, nbnxn_kernels, I/O,

Output from scripts that gathers execution environment information.

```
USER=swsduhpcLD_LIBRARY_PATH=/usr/local/ora_cli/prod⌋
↪  uct/10.2.0/lib:
/usr/local/ora_cli/product/10.2.0/lib32:/gfspenvhome⌋
↪  /penvSvr/lib/:/usr/sw-cluster/intel/composer_xe_⌋
↪  2013_sp1.4.211/compiler/lib/intel64:/usr/sw-clus⌋
↪  ter/intel/composer_xe_2013_sp1.4.211/mpirt/lib/i⌋
↪  ntel64:/usr/sw-cluster/intel/composer_xe_2013_sp⌋
↪  1.4.211/compiler/lib/intel64:/usr/sw-cluster/int⌋
↪  el/composer_xe_2013_sp1.4.211/mkl/lib/intel64:/u⌋
↪  sr/sw-cluster/intel/composer_xe_2013_sp1.3.174/c⌋
↪  ompiler/lib/intel64:/usr/sw-cluster/intel/compos⌋
↪  er_xe_2013_sp1.3.174/mpirt/lib/intel64:/usr/sw-c⌋
↪  luster/intel/composer_xe_2013_sp1.3.174/ipp/../c⌋
↪  ompiler/lib/intel64:/usr/sw-cluster/intel/compos⌋
↪  er_xe_2013_sp1.3.174/ipp/lib/intel64:/usr/sw-clu⌋
↪  ster/intel/composer_xe_2013_sp1.3.174/compiler/l⌋
↪  ib/intel64:/usr/sw-cluster/intel/composer_xe_201⌋
↪  3_sp1.3.174/mkl/lib/intel64:/usr/sw-cluster/inte⌋
↪  l/composer_xe_2013_sp1.3.174/tbb/lib/intel64/gcc⌋
↪  4.4:/usr/sw-cluster/slurm-16.05.3/lib:/usr/sw-cl⌋
↪  uster/mpi2/lib:/home/export/online1/systest/swpe⌋
↪  rf/wd/BLCR/lib::
/home/export/online1/systest/swsduhpc/local/libHOME=⌋
↪  /home/export/online1/systest/swsduhpcTERM=xtermP⌋
↪  ATH=/home/export/online1/swmore/local/bin:/home/⌋
↪  export/online1/swmore/release/bin:/usr/sw-mpp/bi⌋
↪  n/:/usr/bin:/home/export/online1/swmore/minicond⌋
↪  a3/bin:/home/export/online1/swmore/miniconda3/co⌋
↪  ndabin:/usr/sw-mpp/bin:/usr/java/jdk1.6.0_07/bin⌋
↪  :/usr/sw-cluster/intel/composer_xe_2013_sp1.4.21⌋
↪  1/bin/intel64:/usr/sw-cluster/intel/composer_xe_⌋
↪  2013_sp1.4.211/mpirt/bin/intel64:/usr/sw-cluster⌋
↪  /intel/composer_xe_2013_sp1.4.211/debugger/gdb/i⌋
↪  ntel64_mic/py26/bin:/usr/sw-cluster/intel/compos⌋
↪  er_xe_2013_sp1.4.211/debugger/gdb/intel64/py26/b⌋
↪  in:/usr/sw-cluster/intel/composer_xe_2013_sp1.4.⌋
↪  211/bin/intel64:/usr/sw-cluster/intel/composer_x⌋
↪  e_2013_sp1.4.211/bin/intel64_mic:/usr/sw-cluster⌋
↪  /intel/composer_xe_2013_sp1.4.211/debugger/gui/i⌋
↪  ntel64:/usr/sw-cluster/intel/composer_xe_2013_sp⌋
↪  1.3.174/bin/intel64:/usr/sw-cluster/intel/compos⌋
↪  er_xe_2013_sp1.3.174/mpirt/bin/intel64:/usr/sw-c⌋
↪  luster/intel/composer_xe_2013_sp1.3.174/debugger⌋
↪  /gdb/intel64_mic/py26/bin:/usr/sw-cluster/intel/⌋
↪  composer_xe_2013_sp1.3.174/debugger/gdb/intel64/⌋
↪  py26/bin:/usr/sw-cluster/intel/composer_xe_2013_⌋
↪  sp1.3.174/bin/intel64:/usr/sw-cluster/intel/comp⌋
↪  oser_xe_2013_sp1.3.174/bin/intel64_mic:
```

```
/usr/sw-cluster/intel/composer_xe_2013_sp1.3.174/deb⌟
↪  ugger/gui/intel64:/usr/lib64/qt-3.3/bin:/usr/ker⌟
↪  beros/sbin:/usr/kerberos/bin:/opt/clusconf/bin:/⌟
↪  usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr⌟
↪  /sbin:/sbin:/opt/ibutils/bin:/usr/sw-cluster/slu⌟
↪  rm-16.05.3/sbin:/usr/sw-cluster/slurm-16.05.3/bi⌟
↪  n:/usr/sw-cluster/slurm-16.05.3/bin:/usr/sw-clus⌟
↪  ter/mpi2/bin:/bin::/usr/sw-mpp/bin:/usr/kerberos⌟
↪  /bin/:/usr/sw-mpp/swcc/sw5gcc-binary/bin/:/home/⌟
↪  export/online1/systest/swsduhpc/local/binMV2_HAN⌟
↪  G_WHEN_ERROR=1SHELL=/bin/bashPWD=/home/export/on⌟
↪  line1/systest/swsduhpc/dxh/workspace/testcollect⌟
↪  TZ=CST-8RMS_RANK=0RMS_MYID=0SSH_CLIENT=192.168.1⌟
↪  67.11 38041
↪  22QTDIR=/usr/lib64/qt-3.3QTINC=/usr/lib64/qt-3.3⌟
↪  /includeORACLE_BASE=/usr/local/ora_cliMAIL=/var/⌟
↪  spool/mail/swsduhpcKDE_IS_PRELINKED=1LANG=en_US.⌟
↪  UTF-8MODULEPATH=/usr/share/Modules/modulefiles:/⌟
↪  etc/modulefiles:/usr/sw-cluster/Modules/modulefi⌟
↪  lesLOADEDMODULES=KDEDIRS=/usrSSH_ASKPASS=/usr/li⌟
↪  bexec/openssh/gnome-ssh-askpassSHLVL=2LOGNAME=sw⌟
↪  sduhpcQTLIB=/usr/lib64/qt-3.3/libCVS_RSH=sshCLAS⌟
↪  SPATH=.:SSH_CONNECTION=192.168.167.11 38041
↪  41.0.0.188 22MODULESHOME=/usr/share/ModulesLESSO⌟
↪  PEN=||/usr/bin/lesspipe.sh
↪  %sORACLE_HOME=/usr/local/ora_cli/product/10.2.0G⌟
↪  _BROKEN_FILENAMES=1BASH_FUNC_module()=() {  eval
↪  `/usr/bin/modulecmd bash $*`
}_=/usr/sw-mpp/bin/bsubRMS_USER=swsduhpcRMS_USER_HOM⌟
↪  E=/home/export/online1/systest/swsduhpcMKLROOT=/⌟
↪  usr/sw-cluster/intel/composer_xe_2013_sp1.4.211/⌟
↪  mklMANPATH=/usr/sw-cluster/intel/composer_xe_201⌟
↪  3_sp1.4.211/man/en_US:/usr/sw-cluster/intel/comp⌟
↪  oser_xe_2013_sp1.4.211/man/en_US:/usr/sw-cluster⌟
↪  /intel/composer_xe_2013_sp1.3.174/man/en_US:/usr⌟
↪  /sw-cluster/intel/composer_xe_2013_sp1.3.174/man⌟
↪  /en_US:/usr/kerberos/man:/opt/clusconf/man:/usr/⌟
↪  local/share/man:/usr/share/man/overrides:/usr/sh⌟
↪  are/man/en:/usr/share/man:::/usr/sw-mpp/mpi2/sha⌟
↪  re/man/PROJ_LIB=/home/export/online1/swmore/mini⌟
↪  conda3/share/projLINK_F64=-Wl,-defsym,athread_sp⌟
↪  awn64_=athread_spawn64,-defsym,athread_join64_=a⌟
↪  thread_join64,-defsym,athread_enter64_=athread_e⌟
↪  nter64,-defsym,athread_leave64_=athread_leave64H⌟
↪  OSTNAME=psn013SLURM_ROOT=/usr/sw-cluster/slurm-1⌟
↪  6.05.3IPPROOT=/usr/sw-cluster/intel/composer_xe_⌟
↪  2013_sp1.3.174/ippINTEL_LICENSE_FILE=/usr/sw-clu⌟
↪  ster/intel/composer_xe_2013_sp1.3.174/licenses:
/usr/sw-cluster/intel/licenses:/home/export/online1/⌟
↪  systest/swsduhpc/intel/licenses:/usr/sw-cluster/⌟
↪  intel/composer_xe_2013_sp1.4.211/licenses:/usr/s⌟
↪  w-cluster/intel/licenses:/home/export/online1/sy⌟
↪  stest/swsduhpc/intel/licensesHISTSIZE=5000CATALI⌟
↪  NA_HOME=/usr/local/tomcat/apache-tomcat-6.0.20MV⌟
↪  2_MEMORY_OPTIMIZATION=0GDBSERVER_MIC=/usr/sw-clu⌟
↪  ster/intel/composer_xe_2013_sp1.4.211/debugger/g⌟
↪  db/target/mic/bin/gdbserverLIBRARY_PATH=/usr/sw-⌟
↪  cluster/intel/composer_xe_2013_sp1.4.211/compile⌟
↪  r/lib/intel64:/usr/sw-cluster/intel/composer_xe_⌟
↪  2013_sp1.4.211/compiler/lib/intel64:/usr/sw-clus⌟
↪  ter/intel/composer_xe_2013_sp1.4.211/mkl/lib/int⌟
↪  el64:/usr/sw-cluster/intel/composer_xe_2013_sp1.⌟
↪  3.174/compiler/lib/intel64:/usr/sw-cluster/intel⌟
↪  /composer_xe_2013_sp1.3.174/ipp/../compiler/lib/⌟
↪  intel64:/usr/sw-cluster/intel/composer_xe_2013_s⌟
↪  p1.3.174/ipp/lib/intel64:/usr/sw-cluster/intel/c⌟
↪  omposer_xe_2013_sp1.3.174/compiler/lib/intel64:/⌟
↪  usr/sw-cluster/intel/composer_xe_2013_sp1.3.174/⌟
↪  mkl/lib/intel64:/usr/sw-cluster/intel/composer_x⌟
↪  e_2013_sp1.3.174/tbb/lib/intel64/gcc4.4CONDA_SHL⌟
↪  VL=1LINK_OTRACE=-Wl,--whole-archive,-wrap,openat⌟
↪  64,-wrap,openat,-wrap,__openat,-wrap,__openat64,⌟
↪ -wrap,__libc_openat,-wrap,__libc_openat64,-wrap,o⌟
↪  pen,-wrap,open64,-wrap,__open,-wrap,__open64,-wr⌟
↪  ap,__libc_open,-wrap,__libc_open64
↪  /home/export/online1/swmore/release/lib/libotrac⌟
↪  e.a
↪  -Wl,--no-whole-archiveCONDA_PROMPT_MODIFIER=(bas⌟
↪  e)
↪  SW_CLUSTER_PATH=/usr/sw-clusterJBOSS_HOME=/opt/o⌟
↪  pev/jboss-4.2.3.GALINK_SPC=-Wl,--whole-archive,-⌟
↪  wrap,athread_init,-wrap,__expt_handler,-wrap,__r⌟
↪  eal_athread_spawn
↪  /home/export/online1/swmore/release/lib/libspc.a
↪  -Wl,--no-whole-archiveSSH_TTY=/dev/pts/33MIC_LD_⌟
↪  LIBRARY_PATH=/usr/sw-cluster/intel/composer_xe_2⌟
↪  013_sp1.4.211/compiler/lib/mic:/usr/sw-cluster/i⌟
↪  ntel/composer_xe_2013_sp1.4.211/mpirt/lib/mic:
```

```
/usr/sw-cluster/intel/composer_xe_2013_sp1.4.211/com⌟
↪  piler/lib/mic:/usr/sw-cluster/intel/composer_xe_⌟
↪  2013_sp1.4.211/mkl/lib/mic:/usr/sw-cluster/intel⌟
↪  /composer_xe_2013_sp1.3.174/compiler/lib/mic:/us⌟
↪  r/sw-cluster/intel/composer_xe_2013_sp1.3.174/mp⌟
↪  irt/lib/mic:/usr/sw-cluster/intel/composer_xe_20⌟
↪  13_sp1.3.174/compiler/lib/mic:/usr/sw-cluster/in⌟
↪  tel/composer_xe_2013_sp1.3.174/mkl/lib/mic:/usr/⌟
↪  sw-cluster/intel/composer_xe_2013_sp1.3.174/tbb/⌟
↪  lib/micCLUSCONF_HOME=/opt/clusconfLINK_MPI_DIAG=⌟
↪ -Wl,-wrap,MPI_Send,-wrap,MPI_Recv,-wrap,MPI_Isend⌟
↪  ,-wrap,MPI_Irecv,-wrap,MPI_Sendrecv,-wrap,MPI_Se⌟
↪  ndrecv_replace,-wrap,MPI_Barrier,-wrap,MPI_Bcast⌟
↪  ,-wrap,MPI_Gather,-wrap,MPI_Gatherv,-wrap,MPI_Sc⌟
↪  atter,-wrap,MPI_Scatterv,-wrap,MPI_Allgather,-wr⌟
↪  ap,MPI_Allgatherv,-wrap,MPI_Alltoall,-wrap,MPI_A⌟
↪  lltoallv,-wrap,MPI_Alltoallw,-wrap,MPI_Reduce,-w⌟
↪  rap,MPI_Allreduce,-wrap,MPI_Reduce_scatter,-wrap⌟
↪  ,MPI_Ibarrier,-wrap,MPI_Ibcast,-wrap,MPI_Igather⌟
↪  ,-wrap,MPI_Igatherv,-wrap,MPI_Iscatter,-wrap,MPI⌟
↪  _Iscatterv,-wrap,MPI_Iallgather,-wrap,MPI_Iallga⌟
↪  therv,-wrap,MPI_Ialltoall,-wrap,MPI_Ialltoallv,-⌟
↪  wrap,MPI_Ialltoallw,-wrap,MPI_Ireduce,-wrap,MPI_⌟
↪  Iallreduce,-wrap,MPI_Ireduce_scatter,-wrap,MPI_W⌟
↪  ait,-wrap,MPI_Waitall,-wrap,MPI_Init
/home/export/online1/swmore/release/lib/mpi_diag.oLS⌟
↪  _COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so⌟
↪  =01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31⌟
↪  ;01:mi=01;05;37;41:su=37;41:sg=30;43:ca=30;41:tw⌟
↪  =30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.⌟
↪  tgz=01;31:*.arj=01;31:*.taz=01;31:*.lzh=01;31:*.⌟
↪  lzma=01;31:*.tlz=01;31:*.txz=01;31:*.zip=01;31:*⌟
↪  .z=01;31:*.Z=01;31:*.dz=01;31:*.gz=01;31:*.lz=01⌟
↪  ;31:*.xz=01;31:*.bz2=01;31:*.tbz=01;31:*.tbz2=01⌟
↪  ;31:*.bz=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;3⌟
↪  1:*.jar=01;31:*.rar=01;31:*.ace=01;31:*.zoo=01;3⌟
↪  1:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.jpg=01;35⌟
↪  :*.jpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;3⌟
↪  5:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;3⌟
↪  5:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;⌟
↪  35:*.svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01⌟
↪  ;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=0⌟
↪  1;35:*.mkv=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v=0⌟
↪  1;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=0⌟
↪  1;35:*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.rmvb=0⌟
↪  1;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv=0⌟
↪  1;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;⌟
↪  35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.axv=01;⌟
↪  35:*.anx=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=01;⌟
↪  36:*.au=01;36:*.flac=01;36:*.mid=01;36:*.midi=01⌟
↪  ;36:*.mka=01;36:*.mp3=01;36:*.mpc=01;36:*.ogg=01⌟
↪  ;36:*.ra=01;36:*.wav=01;36:*.axa=01;36:*.oga=01;⌟
↪  36:*.spx=01;36:*.xspf=01;36:
CONDA_EXE=/home/export/online1/swmore/miniconda3/bin⌟
↪  /condaMIC_LIBRARY_PATH=/usr/sw-cluster/intel/com⌟
↪  poser_xe_2013_sp1.3.174/tbb/lib/micCPATH=/usr/sw⌟
↪ -cluster/intel/composer_xe_2013_sp1.4.211/mkl/inc⌟
↪  lude:
/usr/sw-cluster/intel/composer_xe_2013_sp1.3.174/ipp⌟
↪  /include:/usr/sw-cluster/intel/composer_xe_2013_⌟
↪  sp1.3.174/mkl/include:/usr/sw-cluster/intel/comp⌟
↪  oser_xe_2013_sp1.3.174/tbb/includeCLOUD_HOME=/op⌟
↪  t/cloudmon_CE_CONDA=NLSPATH=/usr/sw-cluster/inte⌟
↪  l/composer_xe_2013_sp1.4.211/compiler/lib/intel6⌟
↪  4/locale/%l_%t/%N:/usr/sw-cluster/intel/composer⌟
↪  _xe_2013_sp1.4.211/mkl/lib/intel64/locale/%l_%t/⌟
↪  %N:/usr/sw-cluster/intel/composer_xe_2013_sp1.4.⌟
↪  211/debugger/gdb/intel64_mic/py26/share/locale/%⌟
↪  l_%t/%N:/usr/sw-cluster/intel/composer_xe_2013_s⌟
↪  p1.4.211/debugger/gdb/intel64/py26/share/locale/⌟
↪  %l_%t/%N:/usr/sw-cluster/intel/composer_xe_2013_⌟
↪  sp1.4.211/debugger/intel64/locale/%l_%t/%N:/usr/⌟
↪  sw-cluster/intel/composer_xe_2013_sp1.3.174/comp⌟
↪  iler/lib/intel64/locale/%l_%t/%N:/usr/sw-cluster⌟
↪  /intel/composer_xe_2013_sp1.3.174/ipp/lib/intel6⌟
↪  4/locale/%l_%t/%N:/usr/sw-cluster/intel/composer⌟
↪  _xe_2013_sp1.3.174/mkl/lib/intel64/locale/%l_%t/⌟
↪  %N:/usr/sw-cluster/intel/composer_xe_2013_sp1.3.⌟
↪  174/debugger/gdb/intel64_mic/py26/share/locale/%⌟
↪  l_%t/%N:/usr/sw-cluster/intel/composer_xe_2013_s⌟
↪  p1.3.174/debugger/gdb/intel64/py26/share/locale/⌟
↪  %l_%t/%N:/usr/sw-cluster/intel/composer_xe_2013_⌟
↪  sp1.3.174/debugger/intel64/locale/%l_%t/%NNFSCON⌟
↪  F=/opt/clusconf/etc/nfs.cfgCPL_ZIP_ENCODING=UTF-⌟
↪  8TBBROOT=/usr/sw-cluster/intel/composer_xe_2013_⌟
↪  sp1.3.174/tbbCONDA_PREFIX=/home/export/online1/s⌟
↪  wmore/miniconda3JAVA_HOME=/usr/java/jdk1.6.0_07N⌟
↪  CARG_ROOT=/home/export/online1/swmore/miniconda3⌟
↪  IPMICONF=/opt/clusconf/etc/ipmi.cfgIDB_HOME=/usr⌟
↪  /sw-cluster/intel/composer_xe_2013_sp1.4.211/bin⌟
↪  /intel64GDB_CROSS=/usr/sw-cluster/intel/composer⌟
↪  _xe_2013_sp1.4.211/debugger/gdb/intel64_mic/py26⌟
↪  /bin/gdb-micAUTOCLUSCONF=/opt/clusconf/etc/autoc⌟
↪  onf.cfgGDAL_DATA=/home/export/online1/swmore/min⌟
↪  iconda3/share/gdalHISTCONTROL=ignoredupsMPM_LAUN⌟
↪  CHER=/usr/sw-cluster/intel/composer_xe_2013_sp1.⌟
↪  4.211/debugger/mpm/bin/start_mpm.sh_CE_M=PS1_BAC⌟
↪  KUP_SWMORE=[\u@\h \W]\$
↪  MPI_ROOT=/usr/sw-cluster/mpi2STARTWAITTIME=300OR⌟
↪  IG_PYTHON_PATH=/usr/binCONDA_DEFAULT_ENV=baseINC⌟
↪  LUDE=/usr/sw-cluster/intel/composer_xe_2013_sp1.⌟
↪  4.211/mkl/include:/usr/sw-cluster/intel/composer⌟
↪  _xe_2013_sp1.3.174/mkl/includeHISTFILE=/tmp/.his⌟
↪  tory/swsduhpc/192.168.167.11.history.2019:04:10:⌟
↪  18:05:44HISTTIMEFORMAT=[%Y.%m.%d %H.%M:%S]
↪  RMS_NODE_MPES=1RMS_RUN_TPROCS=1RMS_TPROCS=1RMS_N⌟
↪  NODES=1RMS_ONLY_MASTERCORE=1RMS_JOBLANG=seriesRM⌟
↪  S_FT_POLICY=rerunRMS_BIND_CPU=1RMS_CKPT_TYPE=sys⌟
↪  ckptRMS_STATE_DIR=/usr/sw-mpp/stateRMS_JOBID=454⌟
↪  63180RMS_JOBNAME=cpRMS_INTERACTIVE=1RMS_QUEUE=q_⌟
↪  sw_exprRMS_SUBCWD=/home/export/online1/systest/s⌟
↪  wsduhpc/dxh/workspace/testcollectRMS_NODE_EXCLUS⌟
↪  IVE=1RMS_TS_VERSION=ts_ver_ftRMS_PROGNAME=/bin/cp
```