Performance Evaluation and Vectorization of a basic DL_POLY Algorithm

Bobo Shi
Biophysics/Chemistry
100 W. 18th Ave
Columbus, OH, 43210
shi.224@osu.edu

ABSTRACT

Molecular Dynamics (MD) simulations are becoming more and more important in physics, chemistry, biological modeling and material science. It can capture the atomistic details that are hard to obtain through experiments. However, as the simulation system becomes large, the computing period is often beyond months. Parallelization of the MD simulations code is widely studied to accelerate the computing. DL_POLY is a MD simulation package, which is available for three body potentials and is used in my research. There are two main sections in this paper. Firstly the performance of DL_POLY as a function of different number of processors, nodes and system sizes is reported. Program details, such as cache misses, MPI communication cost and so on, are used to explain the performance. Isoefficiency of the DL_POLY is roughly analyzed. Secondly, a new vectorization algorithm for coulombic interactions is developed. The performance of the new algorithm is compared to the normal MD code and a known vectorization algorithm.

Categories and Subject Descriptors

J.3 [**Computer Applications**]: Life and medical sciences; L.5.0 [**Science and Technology of Learning**]: Simulation/Games—*simulation*; D.1.3 [**Software Engineering**]: Programming Techniques—*concurrent programming*; I.1.2 [**Computing Methodologies**]: Symbolic and Algebraic Manipulation—*algorithms*

General Terms

Parallel computing

Keywords

DL_POLY, parallel computing, performance evaluation, vectorization

1. INTRODUCTION

DL_POLY is a general-purpose classical molecular dynamics (MD) simulation software[7]. Generally speaking, it uses Newtonian equation to solve the motions of the particles at atomistic level. In molecular dynamics simulation, the position and velocity of each atom in each time step are calculated. Then based on the positions and velocities of atoms at the current step, the positions and velocities of each atom at next step are updated by calculating the force on each atom.

When the system is large, it is necessary to use multiple processors. A Domain Decomposition[6, 5] strategy is used in DL_POLY to realize parallelization. The simulated system is divided into equi-geometrical spatial blocks or domains, each of which is allocated to a specific processor of a parallel computer. For example, the arrays defining the atomic coordinates r, velocities v, potentials u and forces f, for all N atoms in the simulated system, are divided into sub-arrays of approximately size N/P, where P is the number of processors. The sub-arrays are allocated to specific processors. When using multiple nodes, MPI will be used to communicate with different nodes. Also, GPU port is also implemented in DL_POLY but has some bugs for now. The function of GPU is different from the MPI. MPI works together with Domain Decomposition and it contributes to the spatial. GPU mainly works on computing forces, updating velocities and coordinates at each time step.

2. PERFORMANCE EVALUATION OF DL_POLY

The parallelism of DL_POLY and other popular MD packages, such as GROMACS[8] and LAMMPS[3], is mainly implemented using message passing interface technique combined with the Domain Decomposition strategy. The performance of the parallelization for the system is very important, which can save scientists and engineers tons of time.

2.1 Methods

All the evaluation tests are performed on Oakley Cluster in Ohio Supercomputer Center (OSC). Oakley cluster has 8328 cores (12 cores/node and 48 gigabytes of memory/node) with Xeon x5650 CPUs. The test systems are water molecules with NVT ensemble of tempreture $T = 300K$. The fixed volume is chosen to obtain pressure of 1bar.

4 sets of experiments are performed: 1) To study how the performance is improved with multiple cores within the same node, a single system (number of water molecules = 222264) with different number of cores (number of core(s) is: 1, 2, 4, 6, 8, 10, 12) with the same node is tested. 2) The communications of cores within the same node and cores

in different nodes might be different since the nodes don't share the same cache and memory. Therefore the same system (number of water molecules = 222264) with different number of nodes (number of nodes is: 1, 2, 4, 6, 8, 10, 12 with 1 core/node) are evaluated. 3) Multiple nodes with 12 cores/node are used to study the speedup and efficiency with more nodes. 4) The speedup also depends on the size of the system. Therefore the different sizes of the system (N=5184, 41472, 139968, 222264) with different number of cores (1, 2, 4, 6, 8, 10, 12) are tested in the same node.

All the above experiments are running 1000 steps to get reliable performance results. HPCToolkit[2] is used to monitor and analyze the performance details such as cache misses, floating point cycles completed and so on.

2.2 Definition

Here we define some metrics: Let $T(n, p)$ be the time to finish 1000 steps with number of molecules n using p processors, then:

$$\text{Speedup:} \quad S(n,p) = T(n,1)/T(n,p) \tag{1}$$
$$\text{Efficiency:} \quad E(n,p) = S(n,p)/p \tag{2}$$

2.3 Performance of cores within the same node and different nodes

In order to explore different performances for different nodes-requested schemes, two cases are performed and the results are analyzed. The first case uses the cores from the same nodes. The second case uses cores from the different nodes. In the second case, only one core is used from each requested node and the number of nodes are changed to get different cores.

2.3.1 Results

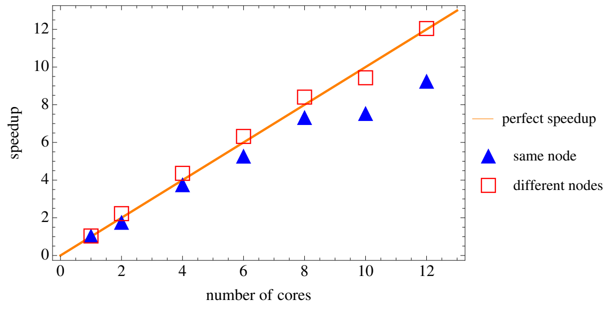

Figure 1: Speedup vs. number of cores in two cases: blue triangle represents cores within the same node and red square represents cores in different nodes

The speedup for two cases is shown in Figure 1. Orange line stands for the perfect speedup. A better speedup is obtained for the case with cores from different nodes. By looking at the efficiency of the two cases, a very high efficiency (higher than 1) is achieved for the case with cores from different nodes. The corresponding efficiency is shown in Figure 2. Efficiency and Speedup show the same result that simulation performed on different nodes gives better performance than simulation performed on the same node.

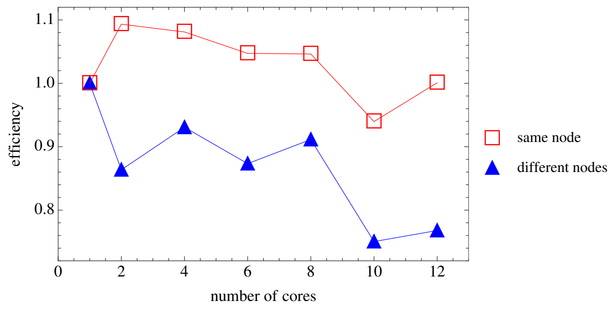

Figure 2: Efficiency vs. number of cores in two cases: cores are in the same node and cores are in different nodes

2.3.2 Explanation

Figure 3 shows two schemes: (a) requested cores are in one node. In this scheme, the communication between different cores has shorter path since they share the same memory. However, all the cores(CPU) share the same bandwidth between cache and memory. When the communication message between cores are large, there will be a delay of communications because of the bandwidth. Furthermore, in this scheme, many cores share the same L2 cache and L3 cache, which indicates higher chances of potential cache misses. (b) requested cores are in different nodes. Only one core is requested on one node. In this case, the communication between each core has longer path, which increases the communicating time. Each core owns their whole L2 and L3 cache as well as the memory. There may be less cache misses.

when the communication message size is large enough, the communication for scheme (a) will be delayed because of the limit of the bandwidth. On the contrary, communication for scheme (b) is not likely to be delayed since the bandwidth is large enough for one core in each node.

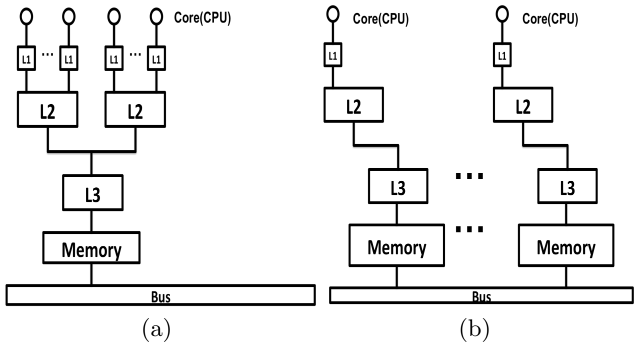

Figure 3: Two different core-requested scheme. (a) Requested cores are in the same node (b) Requested cores are in different nodes.

From Figure 1 and 2, cores in different nodes gives the better performance. From the analysis of the above paragraph, the reasons could be (1) limited bandwidth in first scheme and (2) less cache misses in second scheme.

In order to know whether the limited bandwidth slows down the communication, two MPI functions are monitored by hpctoolkit [2]. MPID_Send() and MPIC_Wait() are the two functions to represent the MPI sampling. Those two functions are used frequently while MPI is used and can basically indicate the overall MPI communication times. The total cycles spent on those two functions are shown in Figure 4. From Figure 4(a), the time spent on MPID_Send() are slightly higher for using cores on the same nodes. From Figure 4(b), MPIC_Wait() for using cores on different nodes costs more time. From only MPID_Send() and MPIC_Wait() functions, we can't conclude that the reason is because the limit of the bandwidth. We may need to monitor more MPI functions to get a good statistics.

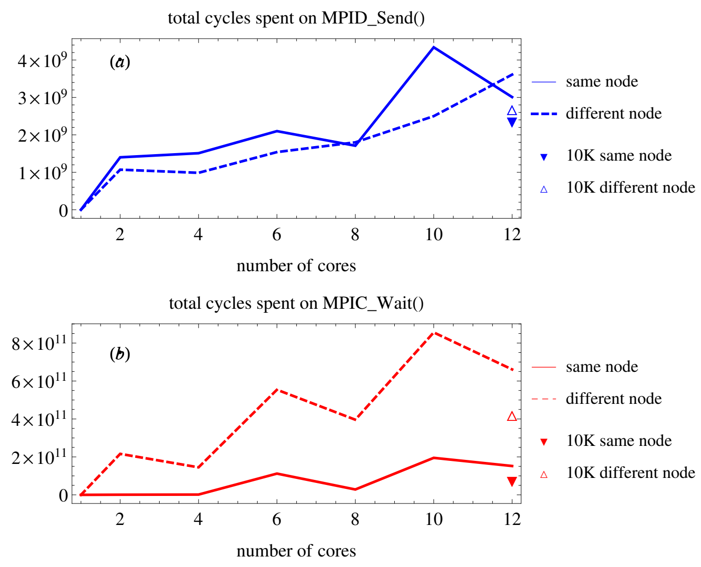

Figure 4: Total cycles spent on MPI functions (a) cycles spent on MPID_Send() for two different scheme. (b) cycles spent on MPIC_Wait() for two different scheme.

Another way to test if the bandwidth is the factor to slow down the communication is to change MPI message size. When temperature of the system is much lower, the particles moves more slowly. For example,for a 300K temperature system, assume there are 100 particles exchanges between each decomposed domain in each 10 steps. When the temperature drops to 10K, there will be much less particles exchanges. One processor take charge of one domain. So decrease of the temperature will decrease the message passing size. 10K simulations are performed for two different schemes. Figure 4 and Table 1 shows the two MPI functions cost on 2 schemes. The time cost on two functions at 10K for both scheme decrease. However, the ratio of two functions time cost of 300K to 10K still remains the same. If the bandwidth is the factor, the same node scheme should have the better performance with the small message-passing size. However, Table 2 shows same node still need longer time to finish. Furthermore, the ratio of computing time of same node to different node for 300K, 12 cores is 1.303 while, for 10K, it is 1.454. It is even larger, which means for smaller message size, performance of same node is even worse. So the bandwidth should not be a factor to slow down the communication of 300K simulations (Figure 1).

Table 1: parameters of 10K, 12 cores simulations

	MPID_Send	MPIC_Wait	L3 cache miss	time(s)
same node	3.37e+09	7.9e+10	4.12e+10	884
different node	2.71e+09	4.25e+11	1.02e+10	608

Another possible reason could be the less cache misses of the second scheme. HPCToolkit[2] is used to detect the cache misses in L1, L2, L3 level. Figure5 shows the cache misses in L1, L2 and L3 level. From the figure, L1 and L2 cache misses are similar for two different schemes and for different number of cores. L3 level cache miss is different for two different schemes. When the number of cores are more than 1, the cache misses in L3 level for scheme 1 is higher than it for scheme 2. That is one of the reasons that scheme 2 gives the better performance. L3 cache misses for the cores in the same node keeps similar value as the number of cores increase. This can be easily explained by the structure of the cluster. Figure 3(a)(a) shows that all 12 cores share the same L3 cache. So no matter how many cores in the same node are used, they use the same L3 cache. So L3 cache misses maintains a similar value as number of cores increases. We can see that the L3 cache misses for cores in different nodes from 1 cores to 2 cores dramatically reduces and then keep the similar value as the number of cores increase. That indicates that the need of the L3 capacity to avoid further cache misses should be between L3 cache size and two times of L3 cache size. Let's use Q3 stead for the need of L3 cache size to just avoid further cache misses. Let's use C3 stead for the L3 cache size from one node of OSC oakley cluster. Then C3 < Q3 < 2*C3. When the number of cores equals 1, only C3 of L3 cache size is available. Since C3 < Q3, there will be lots of L3 cache misses. When the number of cores equals p, p>1. p*C3 L3 cache size is available. p*C3 > Q3, so the L3 cache misses is reduced and remains a similar value for p>2. The similar value L3 cache misses for p>2 is inevitable for a certain level of cache size since L3 cache size is much less than the memory size.

10K data also shows that cache miss is the reason that scheme2 gives the higher performance than scheme1. Table 1 shows that L3 cache miss of scheme1 is 4 times of the scheme2, resulting in longer computing time.

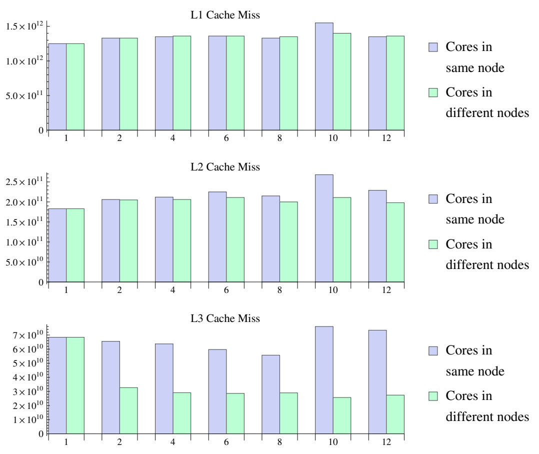

Figure 5: Cache miss for two schemes

2.4 performance vs. nodes

2.4.1 *Results*

The performance on larger cores and nodes are tested. Here, all the cores on each node are used and only number of nodes are changing. Speedup and efficiency are shown in Figure 6. As expected, the efficiency goes down as the number of nodes increase.

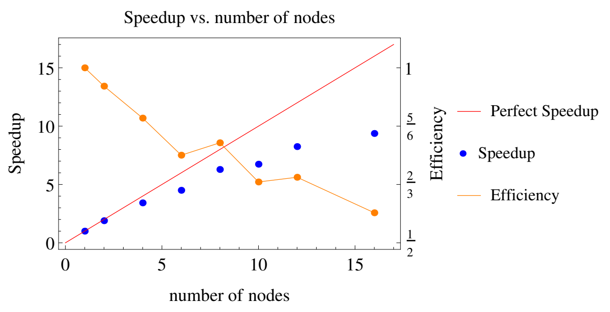

Figure 6: Speedup vs. number of nodes

2.4.2 *Analysis*

In order to explore why the efficiency goes down, total cycles spent on two MPI functions and cache misses information are plotted (Figure 7 and 8). From the cache misses information, L1 cache misses slight goes up when number of nodes is 10 and 12. L2 cache misses slightly goes down when number of processors from 2 to 8. L3 cache miss drops when number of nodes increase from 1 to 6 and then keep a similar value. We can see that when number of nodes equals 1, the L3 cache misses has similar value as the L3 cache misses for cores in same nodes. When there are more nodes, there are more L3 cache to share. In other words, when there are more nodes, each node will take charge of a smaller portion of whole simulation. The smaller portion calculation will need less cache size to store certain data. When number of nodes increases to 5, the L3 cache size is large enough to hold that certain data. So when number of nodes continue to increase from 6, the L3 cache misses never go down. The cache analysis doesn't provide the sufficient evidence to show that the lower efficiency at high number of nodes is due to the cache misses. Actually, when the number of nodes increases, the whole system share more memories so there should be less cache misses. So cache miss could be the reason of low efficiency at high number of nodes.

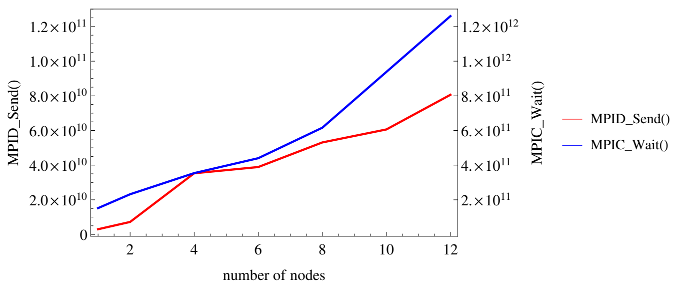

Figure 7: Total cycles spent on MPID_Send() and MPIC_Wait() for many nodes

Figure 7 shows the cycles spent on two MPI functions. Those two functions (MPID_Send() and MPIC_Wait()) are used to represent the overall MPI communication time during the simulation. As we can see, both functions takes more cycles when number of nodes increases. So the MPI communication overhead could be one of the reason of the low efficiency at high number of nodes.

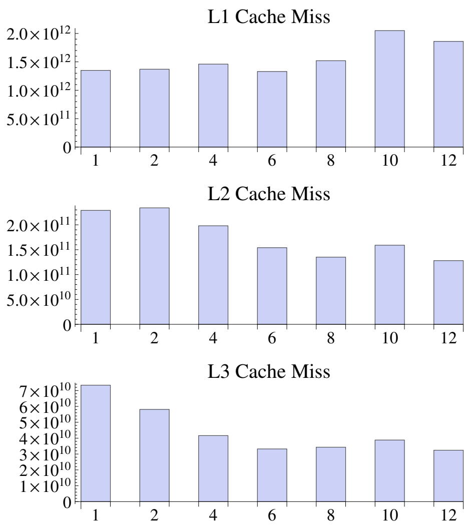

Figure 8: Cache misses for many nodes

2.5 Performance vs. size of system

The performance over different size of the systems is studied in this section. Four different number of molecules are chosen (N = 5184, 41472, 139968, 222264). To simply test the system, simulations are performed in 1 cores to 12 cores in the same node. Figure 9 shows the efficiency of the four

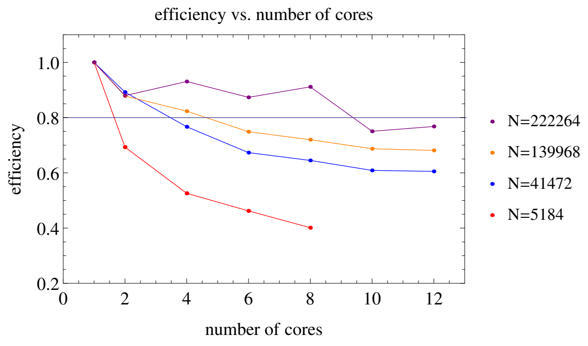

Figure 9: Efficiency for different sizes of system

systems for number of cores from 1 to 12. The trend is reasonable that larger number size of system gives higher speedup. The constant efficiency $eff = 0.8$ is drawn here as a reference line.

2.5.1 *Analysis*

In order to get detailed performance, we could still analyze the cache miss information and MPI functions. To compare the performance over different size of the system, we define cache misses and cycles cost on MPI functions per molecules. Assume $C(N)$ and $F(N)$ are the cache misses and total cycles spent on MPI functions over system size N. The modified cache misses $C'(N)$ and cycles cost $F'(N)'$ on MPI functions are,

$$C'(N) = C(N)/N \quad (3)$$
$$F'(N) = F(N)/N \quad (4)$$

Figure 10 compares the cache misses per system size of L1, L2 and L3 level cache for different system size. For L1 and L2 level, cache misses per system size are similar among different system size. L3 cache misses per system size for larger system (N = 139958 and 222264) is apparently higher than it for smaller system (N = 41472). This means the L3 cache size is enough to hold a certain set of data for N = 41472. But for larger system, L3 cache is not large enough to hold that data and result in more cache misses per system size.

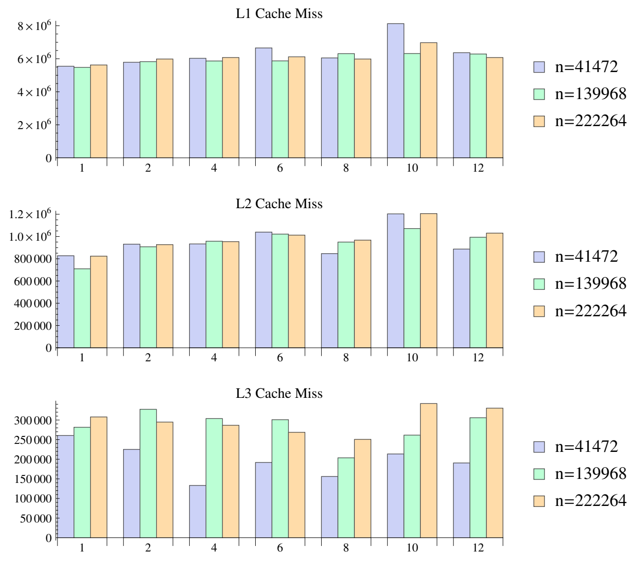

Figure 10: Cache misses for many nodes

Figure 11 indicates the cycles spent on MPI communications. Two functions (MPID_Send() and MPIC_Wait()) are illustrated here. From MPID_Send(), the smallest system (N = 41472) has highest communication time. This means the overhead per system size for the smaller system is higher.

From both Figure 11 (a) and (b), the cycles spent on MPI functions have the same pattern depend on the number of cores. For example, when number of cores equals 6 or 10, the communication time is high. The topology of how the CPUs are assigned to the physical location of the simulation box could be a reason. For example, 6 cores, 2 × 3 × 1 and For 10 cores, 2 × 5 × 1, make the topology of each region flat or thin, which may make the communication less efficient. But for like 8 cores, 2 × 2 × 2 and 12 cores, 2 × 2 × 3, the domain for each processor is more cube-like, which makes the communication more efficient.

2.5.2 *Isoefficiency*

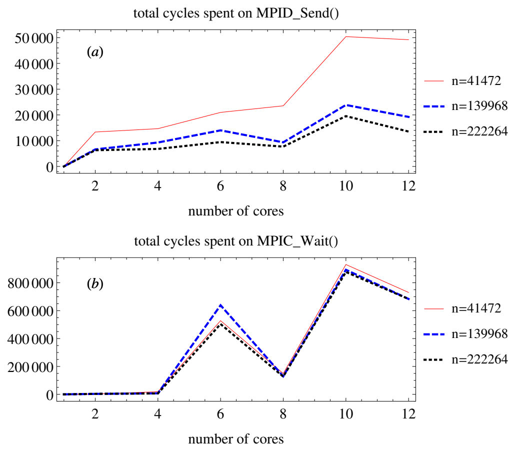

Figure 11: Cycles cost on MPI functions (a) MPID_Send() (b) MPIC_Wait() for different sizes of the systems

To study the isoeffciency of the DL_POLY simulation package, 4 points are taken from efficiency plot (Figure 9) by choosing the efficiency of each size system equal to 0.8. The four points are listed in the Table 2.

Table 2: The number of processors and particles parameters for isoefficiency analysis

number of processors(p)	number of particles(N)
1.8	5184
3.5	41472
4.8	139968
9	222264

To check the isoefficiency, we want to know as we increase number of processors (p), is it feasible to maintain a desired level of efficiency by suitably increasing the problem size (N)? and how many should we increase. Three common relations,

$$N = \Theta(p) \quad (5)$$
$$N = \Theta(p \log p) \quad (6)$$
$$N = \Theta(p^2) \quad (7)$$

are used to fit the the curve, which are shown in Figure 12. From the R^2 value, $N = \Theta(p)$ is the most closed one. So we may say that in order to maintain a desired level of efficiency (0.8), number of particles should go at least as fast as number of processors (p). However, the size of data to analyze the isoefficiency is so small that I can't give a clear conclusion. $N = \Theta(p)$ is just a very roughly estimated solution. Much more work is still need to do to get the sufficient data to analyze.

2.6 Summary

The cores in different nodes give better performance than the cores in the same node. Less L3 cache misses of cores in different nodes is the main reason for the better performance. No evidence shows bandwidth affects the performance of two schemes.

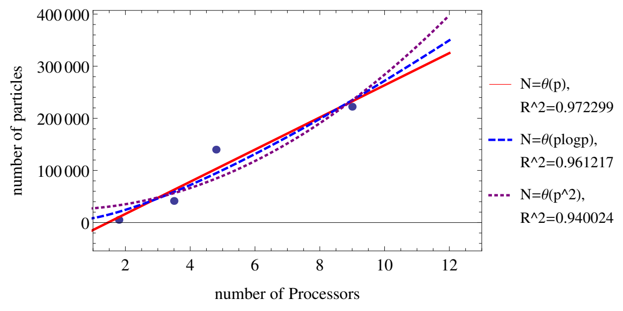

Figure 12: Isoefficiency fit

When the number of processors increases, the effciency decreases. Although the cache miss decreases with more processors, the communication cost increases and becomes the key limiting factor.

Small size system will have less cache miss per system size in a same node. How the processors are assigned to the system domain affects the communication cost. Isoefficiency analysis shows that to maintain the desired level of efficiency, such as 0.8, the number of particles should increase as fast as the number of processors, $N = \Theta(p)$.

3. VECTORIZATIONS OF MD CODE

3.1 Introduction

Basic MD simulation proceeds in a series of time steps. At each step, the potential and force on each particles due to the interaction with other particles are calculated and then the velocity and position of the particles are updated. From the new position, new potential and force will be calculated. The force evaluation takes the majority ($\sim 98\%$) of the computing time.

Vectorization method is widely used in MD simulation code to accelerate the computing. In this section, two vectorization algorithms are performed to compare with the non-vectorized version. The first vectorization algorithm is widely discussed[1, 4], which makes the position coordinate in x, y, z direction separated stored so that the coordinates in each direction located in the physical memory continuously (rx[0], rx[1], rx[2]... are continuous in physical memory). Let's call it column-wise algorithm. The second algorithm is row-wise like. It means the x, y, z coordinate of one particle are continuous, $(r_i[x], r_i[y], r_i[z]$ are continuous in physical memory). As far as I know, there is not a research work about the second method.

To simplify the problem, I make the following simplification for the model to test with the two vectorisation methods:
- Only consider the Coulomb interaction.
- Treat each particle as a point
- Do not consider the periodic boundary conditions.
- Just calculate force and potential and don't do the position integral.

The Coulomb interaction has the following formulas,

$$f_i x = \sum_{j=1, j\neq i}^{N} f_0 \frac{q_i q_j}{r_{ij}^3} rx \quad (8)$$

$$f_i y = \sum_{j=1, j\neq i}^{N} f_0 \frac{q_i q_j}{r_{ij}^3} ry \quad (9)$$

$$f_i z = \sum_{j=1, j\neq i}^{N} f_0 \frac{q_i q_j}{r_{ij}^3} rz \quad (10)$$

$$u_i = \sum_{j=1, j\neq i}^{N} f_0 \frac{q_i q_j}{r_{ij}} \quad (11)$$

Algorithm 1 shows the normal algorithm to compute force and potential due to coulomb interaction. Both r and f are a N by 3 matrix, where N is number of particles and 3 means the system is 3 dimensional (x, y, z). In step 7, the square of the distance between two particles are calculated and will be used to get the force f_{ij} and potential u_{ij} in the next line.

Algorithm 1 normal algorithm
```
 1: Begin
 2: Data $r : N \times 3$, $f : N \times 3$
 3: Data $u : N$
 4: for $i \leftarrow 2, N$ do
 5:     for $j \leftarrow 1, i-1$ do
 6:         $\vec{r}_t \leftarrow \vec{r}(i) - \vec{r}(j)$
 7:         $rsq \leftarrow |r^2|$
 8:         evaluate force $f_{ij}$ and potential $u_{ij}$
 9:         for $k \leftarrow 1, 3$ do
10:             $f_{ik} = f_{ik} + f_{ij}$
11:             $f_{jk} = f_{jk} - f_{ij}$
12:         $u_i = u_i + u_{ij}$
13:         $u_j = u_j + u_{ij}$
14: End
```

3.2 Column-wise method

Column-wise like vetorization algorithm is widely developed[1, 4]. Given a N particle system, the coordinates of all the particles are stored in a $3 \times N$ matrix, $r[3][N]$. The first row, second row and third row are for rx, ry, rz. For the model processor available for SSE4, one floating point can hold 4 float type numbers. To calculate forces, fx, fy, fz can be stored in the same pattern, $f[3][N]$. u is calculated in the normal way. In the single precision 4-way vector, Figure 13 shows how the scalar mapped to the vector. fxt, fyt, fzt, ut are SSE variables and each of these hold 4 single precision corresponding values. For example, $fxt[i]$ holds $fx[4i], fx[4i+1], fx[4i+2], fx[4i+3]$. When looping over each index of $fkt, (k = x, y, z)$, 4 concurrent calculation of $f_k, (k = x, y, z)$ are performed simultaneously.

A simple vectorization algorithm similar to D. Fincham and B. J. Ralston[1] is shown in Algorithm 2, which I call column-wise algorithm. First of all, the coordinates matrix is transposed from Algorithm 1, which makes the vector unit to store the continuous elements of one physical variable. Because of the property of the vector processor, for 128bit 4-way single floating point vectors, 4 concurrent calculation

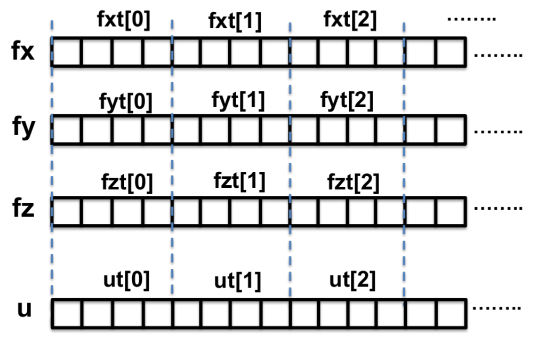

Figure 13: vectorization scheme of the first method

can be performed at the same time. By just transposing matrix is not enough to enable the vectorizatoin algorithm. In Algorithm 1, the inhibition of vectorization is in line 10 and line 12, which has the index i. To make it vectorisable, the column-wise vectorization (Algorithm 2) stores the pair forces (line 10) and potential (line 12), for a given i particles, in the additional arrays, fs and us. Those two arrays are later added to f_i and u_i, in a separate operation outside the inner loop.

Algorithm 2 Vectorization 1

```
 1: Begin
 2: Data $r : 3 \times N$, $f : 3 \times N$
 3: Data $u : N$
 4: for $i \leftarrow 2, N$ do
 5:     for $j \leftarrow 1, i-1$ do
 6:         $\vec{r}_t \leftarrow \vec{r}(i) - \vec{r}(j)$
 7:         $rsq \leftarrow |r^2|$
 8:         evaluate force $f_{ij}$ and potential $u_{ij}$
 9:         for $k \leftarrow 1, 3$ do
10:             $fs_{kj} = f_{ij}$
11:             $f_{kj} = f_{kj} - f_{ij}$
12:         $us_j = u_{ij}$
13:         $u_j = u_j + u_{ij}$
14:     for $k \leftarrow 1, 3$ do
15:         for $j \leftarrow 1, i-1$ do
16:             $f_{ki} = f_{ki} + fs_{kj}$
17:     for $j \leftarrow 1, i-1$ do
18:         $u_i = u_i + us_j$
19: End
```

With these changes, the force and potential summation can be vectorized. However, there are some penalties. One is that the summation over fs and us should be calculated separately, which cost extra computing time. Another one is that extra storage must be supplied for fs and us. But this is not usually a big problem.

3.3 Row-wise method

Row-wise algorithm need the way to store the data in the opposite way. To my knowledge, no work about row-wise algorithm has been reported yet. For 128bit 4-way single implementation, each vector unit can store 4 float type numbers. We define a vector unit array $ft[N]$, each of the vector element $ft[i]$ can be used to store u_i, $f_{i,x}$, $f_{i,y}$ and $f_{i,z}$ as Figure 14 shown.

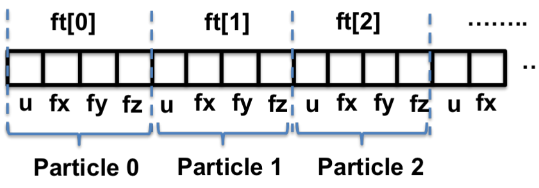

Figure 14: vectorization scheme of the second method

The main reason this algorithm works is the math expression of force and potential. Equations (12, 13) show that f_x, f_y, f_z and u has the same factor, A_0. So we can put f_x, f_y, f_z and u in a 4-way single precision vector. Then multiply the vector by the pre-calculated factor A_0.

$$f_k = \quad f_0 \tfrac{q_i q_j}{r^3} r_k \quad = A_0 r_k, \; (k = x, y, z) \qquad (12)$$

$$u = \quad f_0 \tfrac{q_i q_j}{r} \quad = A_0 r^2 \qquad (13)$$

Algorithm 3 illustrates the how row-wise algorithm works, where $\bigotimes$ means the vector multiplication and $\bigoplus$ means vector addition. The coordinate data doesn't need to be transposed. An array $ft[N]$, a SSE floating point 4-way single precision array, is used to store both forces and potentials. rt is a temporary SSE floating point 4-way single precision variable. Line 5 to line 7 in the Algorithm 3 are used to calculate the pre-factor A_0 and line 8 to line 11 are to do the vector operations.

Algorithm 3 Vectorization 2

```
 1: Begin
 2: Data $r : N \times 3$, $ft : N \times 4$
 3: for $i \leftarrow 2, N$ do
 4:     for $j \leftarrow 1, i-1$ do
 5:         $\vec{r}_t \leftarrow \vec{r}(i) - \vec{r}(j)$
 6:         $rsq \leftarrow |r^2|$
 7:         evaluate pre-factor $A_0$
 8:         set $rt \leftarrow (rsq, r_x, r_y, r_z)$
 9:         $ft_i \leftarrow ft_i \bigoplus A_0 \bigotimes rt$
10:         $rt \leftarrow rt \bigotimes (1, -1, -1, -1)$
11:         $ft_j \leftarrow ft_j \bigoplus A_0 \bigotimes rt$
12: End
```

With the method in Algorithm 3, some part of the normal algorithm can be vectorized. There are some limitations of this method. Firstly, it only vectorizes a part of the algorithm, so the acceleration can not be applied to all the code, which gives some penalty. Secondly, it can't do with the 2-way vectorization since here it needs four variables in one vector. However, it also has some advantages compared to column-wise algorithm. For column-wise algorithm, it has to deal with 4 pairs of particles together. If there is some selection mechanism in the code, which requires only 1 particle be evaluated or calculated at one time, column-wise algorithm will fail. For example, we consider the short-range interaction. Then the comparison between r_{ij} and r_{cut} will be performed. Only when r_{ij} is smaller than r_{cut}, we calculate the corresponding interaction. For the column-wise algorithm, it is hard to do this selection and also perform the afterwards algorithm vectorisably. For row-wise algorithm, this is not a problem since in each loop step, only one pair of particles is calculated.

3.4 Results and discussion

A c++ code with file name "MDvec-BoboShi.cpp" is attached. The code is tested with different number of particles. In the code, three algorithms are written corresponding to pesudocode 1, 2, 3 in the above sections. The speedup based on the normal algorithm is plotted in Figure 15. As we can see the column-wise algorithm, which gives a speedup of $1.0 \sim 1.2$, depends on the number of particles. However, the reported speedup for column-wise algorithm is around 2.0. The row-wise algorithm, which gives a speedup of $1.8 \sim 2.1$, is similar to the reported speedup of the column-wise algorithm and is much higher than the column-wise algorithm that is implemented in this paper.

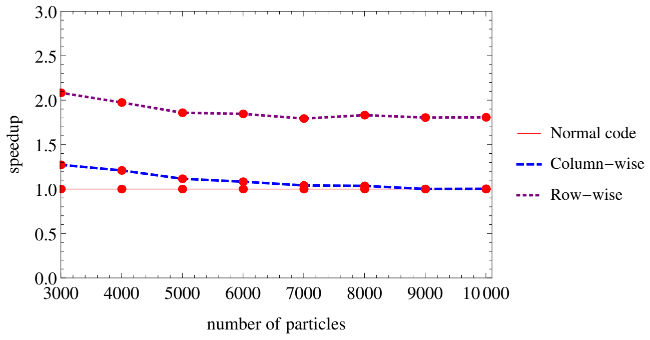

Figure 15: Speedup of two vectorization algorithms

For the column-wise algorithm, initially "`#pragam vector`" is attempted to have compiler vectorize the code but it turns out that more work has to be done. The SSE intrinsic code is explicitly put for both column-wise and row-wise algorithms. The main penalty for the column-wise is the transit between scalar and vector since the inner loop (line 5 to 18) has both vectorisable code and scalar code. After finishing each inner loop, the SSE intrinsic code has to be converted to scalar version so that the parts of force and potential got by intrinsic code and by scalar code can be summed up. The row-wise algorithm is a new approach since, to my knowledge, no work has been reported for this method. However, the main penalty for this method is that it only vectorizes part of the calculation. The advantage of row-wise algorithm is that it is not affected by treating atoms differently, which affects column-wise algorithm a lot.

4. REFERENCES

[1] D. Fincham and B. J. Ralston. Molecular dynamics simulation using the cray-1 vector processing computer. *Computer Physics Communications*, 23(2):127–134, July 1981.

[2] J. Mellor-Crummey, L. Adhianto, F. M., K. M., and T. N. Hpctoolkit usrs manual.

[3] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117(1):1–19, Mar. 1995.

[4] D. C. Rapaport. Large-scale molecular dynamics simulation using vector and parallel computers. *Computer Physics Reports*, 9(1):1–53, Dec. 1988.

[5] D. C. Rapaport. Multi-million particle molecular dynamics: II. design considerations for distributed processing. *Computer Physics Communications*, 62(2âĂŞ3):217–228, Mar. 1991.

[6] I. Todorov and W. Smith. DL_POLY_3: the CCP5 national UK code for moleculraâĂŞdynamics simulations. *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 362(1822).

[7] I. T. Todorov, W. Smith, K. Trachenko, and M. T. Dove. DL_POLY_3: new dimensions in molecular dynamics simulations via massive parallelism. *Journal of Materials Chemistry*, 16(20):1911–1918, May 2006.

[8] D. van der Spoel, E. Lindahl, B. Hess, A. R. van Buuren, E. Apol, P. J. Meulenhoff, D. P. Tieleman, A. L. T. M. Sijbers, K. A. Feenstra, R. van Drunen, and H. J. C. Berendsen. Gromacs user manual version 4.5.6 www.gromacs.org, 2010.