CS 598 Final Project

Scott High and Erin Molloy

May 5, 2015

1 Introduction

Molecular dynamics simulations are important for researching physical phenomena from the electronic structures of metals to the folding trajectories of proteins. In our final project, we create a performance expectation for a computationally intensive function in a mini-application, called `CoMD`, which is a code base designed to expose core features of molecular dynamics simulations while being simple to understand and modify [1].
Our contributions include

(1) A **performance expectation** for the Lennard-Jones force computate function.

(2) Simple code **modifications** which improve performance by 25-27%.

(3) An illustration that **compiler choice** impacts compute function performance.

2 Background

A molecular dynamics simulation models the movement and interactions of a discrete set of simulation particles over time. In an N-particle simulation, the force acting on particle i at $\boldsymbol{r}_i$ is given by

$$\mathbf{F}_i = m_i \ddot{\boldsymbol{r}}_i = -\frac{\partial}{\partial \boldsymbol{r}_i} U(\boldsymbol{r}_1, \ldots, \boldsymbol{r}_N)$$

where U is the potential energy from particle-particle interactions [2]. The resulting system of first order ODEs is integrated to find the positions of particles at each time step. The `CoMD` mini-application computes potentials using the truncated Lennard-Jones potential, where the the potential between a pair of particles is given by

$$U_{LJ_{trunc}}(r_{ij}) = \begin{cases} U_{LJ}(r_{ij}) - U_{LJ}(r_c) & r_{ij} \leq r_c \\ 0 & r_{ij} > r_c \end{cases} \tag{1}$$

where

$$U_{LJ}(r) = 4\epsilon \left\{ \left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right\} \tag{2}$$

is the Lennard-Jones potential and $r_{ij} = \|\mathbf{r}_i - \mathbf{r}_j\|_2$ is the distance between particles. The potential 1 decays rapidly, so interactions are calculated only within a cutoff radius $r_c = 2.5\sigma$. Material parameters ϵ and σ are the potential well depth and distance at which the pair potential becomes zero, respectively [3].

3 Performance baseline

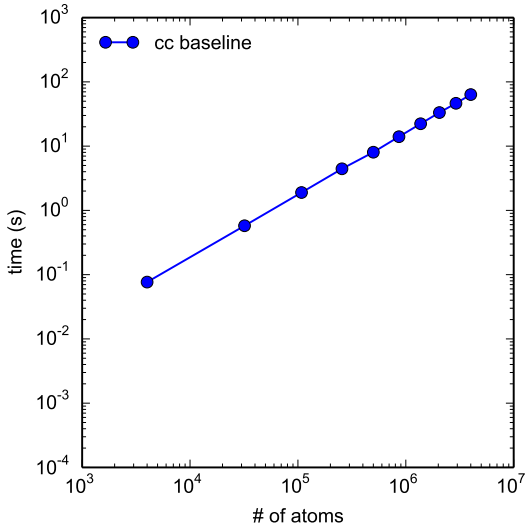

Figure 1: Baseline results with Cray compiler.

`CoMD` includes a default simulation of Copper atoms in a face-centered cubic (FCC) lattice with a fully periodic domain and a spacing of 3.615 Å. Material parameters are defined as $\epsilon = 0.167$ eV and $\sigma = 2.315$ Å [1]. Thus, there exist

$$\left(\frac{4}{3}\pi(2.5 \cdot 2.315)^3 \text{ Å}^3\right) \times \left(\frac{4 \text{ particles}}{3.615 \text{ Å} \times 3.615 \text{ Å} \times 3.615 \text{ Å}}\right) \approx 69 \text{ particles} \tag{3}$$

within the cut-off radius of each particle at lattice initialization.

In our scaling studies, we use default parameters for temperatures of 0, 600, and 3000 K. Strong scaling studies with 16,384,000 atoms are performed on 64, 512, and 4096 processors. Weak scaling studies are performed with 32000 atoms per processor for 1, 8, 64, 512, and 4096 processors. All scaling and baseline results are obtained on Blue Waters by compiling `CoMD` with the default Cray compiler (optimization flags: `-g -O3`). Table 1 shows the difference in runtimes for the force computation function and the halo exchange. The baseline results are shown in figure 1. As the former dominated total runtime, we target the force computation function in order to improve the overall performance. In our performance baseline, when default parameters are used with a temperature of 0 K the particles remain in their initial positions throughout the simulation. This enables us to model the number of

particle-particle interactions as constant.

		Strong scaling		Weak scaling	
temperature	# ranks	force computation	halo exchange	force computation	halo exchange
$T = 0K$	1	-	-	5.1e-1/97.4%	-
	8	-	-	6.2e-1/96.5%	1.2e-3/0.2%
	64	4.8e-0/96.8%	2.2e-2/0.4%	6.2e-1/96.3%	3.9e-3/0.6%
	512	6.2e-1/85.9%	8.2e-2/11.4%	6.2e-1/91.4%	5.9e-3/0.9%
	4096	8.3e-2/72.3%	1.8e-2/15.7%	6.2e-1/91.9%	9.0e-3/1.3%
$T = 600K$	1	-	-	5.0e-1/97.4%	-
	8	-	-	6.1e-1/96.4%	1.4e-3/0.2%
	64	4.8e-0/94.9%	9.2e-2/1.8%	6.1e-1/96.2%	3.6e-3/0.6%
	512	6.1e-1/95.9%	5.7e-3/0.9%	6.1e-1/95.9%	5.9e-3/0.9%
	4096	8.1e-2/67.9%	3.6e-3/3.0%	6.1e-1/92.0%	9.1e-3/1.4%
$T = 3000K$	1	-	-	5.0e-1/97.3%	-
	8	-	-	6.3e-1/96.5%	1.0e-3/0.2%
	64	4.7e-0/96.7%	2.3e-2/0.5%	6.0e-1/96.2%	4.2e-3/0.7%
	512	6.0e-1/95.9%	6.1e-3/1.0%	6.0e-1/95.9%	6.3e-3/1.0%
	4096	8.1e-2/66.3%	3.0e-3/2.4%	6.1e-1/96.4%	9.6e-3/1.5%

Table 1: Scaling Studies (seconds spent in function/percentage of total runtime)

4 Performance model

All analysis assumes a one-level cache model and approximates read and write times as equal. The constants used in the performance expectations are determined by the processor clock rate (2.3 GHz) and the results of the `STREAM` benchmark on Blue Waters.

4.1 Computation-based performance model

We first develop a performance model that represents the best performance a force calculation algorithm can possibly achieve. Our model is based on the cost of updating variables associated with particle i due to interactions with each of it's n neighboring atoms, j. Specifically, the potential energy U_{LJ} between the pair of atoms is computed as

$$U_{LJ}(r_{ij}) = A\left(\frac{1}{r_{ij}}\right)^6\left\{\left(\frac{1}{r_{ij}}\right)^6 - 1\right\}, \tag{4}$$

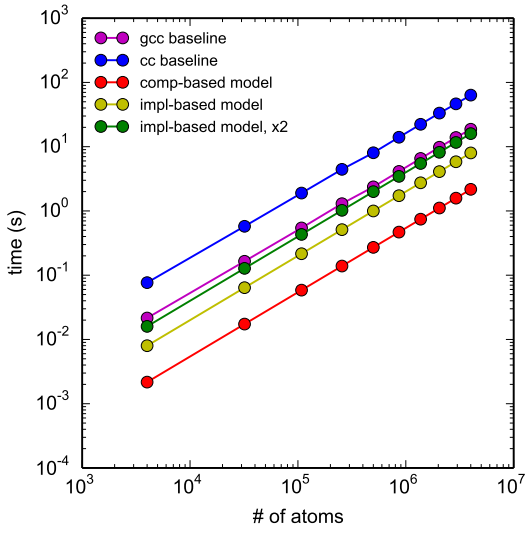

Figure 2: Performance models and baseline results

and the resulting force $\mathbf{F}$ is computed as

$$\begin{aligned}\mathbf{F}(r_{ij}) &= -U'_{LJ}(r_{ij})\hat{r}_{ij} \\ &= 24\frac{\epsilon}{r_{ij}}\left\{2\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right\}\hat{r}_{ij} \\ &= A\frac{1}{r_{ij}^2}\left(\frac{1}{r_{ij}}\right)^{6}\left\{2\left(\frac{1}{r_{ij}}\right)^{6} - 1\right\}\mathbf{r}_{ij}, \end{aligned} \tag{5}$$

where N is the total number of particles in the simulation and n is the number of particles within a sphere defined by the cut-off radius, r_c. In addition, the total energy of the system E_{tot}

$$E_{tot} = \sum_{ij} U_{LJ}(r_{ij}), \tag{6}$$

is updated within the innermost loop of the compute function. The resulting cost is

(A) 3 loads to get particle i's position, $\mathbf{r}_i$

(B) For each particle j where $r_{ij} < r_c$

1. 3 loads to get particle j's position, $\mathbf{r}_j$
2. 3 subtractions, 3 multiplications, 2 additions, and 1 division to calculate $\frac{1}{r_{ij}^2}$
3. 3 multiplications to calculate $\left(\frac{1}{r_{ij}^2}\right)^3 = \left(\frac{1}{r_{ij}}\right)^6$
4. 2 subtractions and 2 multiplications to calculate the potential U_{LJ} (1 subtraction is for the cutoff potential)

5. 3 multiplications and 1 subtraction to calculate magnitude of the force, $F(r_{ij})$
6. 3 multiplications and 3 additions to calculate and update the force vector, $\mathbf{F}$

(C) 1 write to save $U_{LJ}(r_{ij})$, assuming the $U_{LJ}(r_{ji})$ term is saved for free

(D) 3 writes to save $\mathbf{F}$

(E) 1 addition to update E_{tot}

The total cost is then

$$N\left(A + \frac{nB}{2}\right) + C + D + E \tag{7}$$

where the factor of 1/2 accounts for double counting calculations of the symmetric pairwise interactions. Let c be the cost of a floating point operation, w be the cost of a write, and r be the cost of a read. Then the cost equation 7 becomes

$$N\left(\frac{n}{2}\left(26c + 3w\right) + 3w\right) + c + 4r, \tag{8}$$

which reduces to

$$N\left(\frac{n}{2}\left(26c + 3m\right) + 3m\right) + c + 4m \tag{9}$$

assuming equal read and write times m. Specifically, we approximate $c = 4.35 \times 10^{-10}$ flops/s using the processor clock rate and $m = 1.43 \times 10^{-9}$ B/s using the STREAM benchmark on Bluewaters. We estimate $n = 69$ from the cut-off radius and lattice parameters. The results of this performance model are shown in figure 2. Both our results and the computation-based performance model scale linearly with the number of atoms, however our performance expectation is nearly 2 orders of magnitude less than our experimental results.

4.2 Implementation-based model

The computation-based performance model from section 4.1 represents a best case scenario for the number of operations required to compute the inter-particle forces. To better understand our baseline results, we also develop a performance model that captures the algorithm's implementation.

The data structure used for storing the computational particles is a struct of linear C arrays. Each linear array is divided into equal sized sections which store information for particles in a region of the simulation domain. Section size is the maximum number of particles per region of space, and does not change throughout the simulation. This ordering of particles is used to efficiently determine, which particle pairs are within the cutoff radius. However, this still requires work to "check" whether a particle is within the cut-off radius. Specifically, the inter-particle distance is computed as $\|\boldsymbol{r}_i - \boldsymbol{r}_j\|_2^2$, which requires 3 reads, 3 subtractions, 3 multiplications, and 2 additions. If we denote the number of particles checked which are not within the sphere defined by r_c as k, then our performance model becomes

$$N\left(\frac{n}{2}\left(26c + 3m\right) + \frac{k}{2}\left(8c + 3m\right) + 3m\right) + c + 4m \tag{10}$$

In addition, the actual implementation does not compute interactions and distance checks once but rather does so for each particle. This doubles the number of interactions and distance checks in our the performance model 10 to give

$$N\left(n\left(26c+3m\right)+k\left(8c+3m\right)+3m\right)+c+4m \tag{11}$$

The results of these implementation-based performance models are shown in figure 2. These models scale linearly with the number of atoms and appear much closer to our experimental results; however, our performance expectation is still nearly an order of magnitude less than our baseline.

5 Compiler choice

The baseline results using the Cray compiler compare poorly with the performance models as shown in figure 2. We were unable to determine why the Cray compiler has such poor performance and ran an additional baseline using the `CoMD` code compiled with gcc 4.7.1. This compiler change alone resulted in a 3.5 times speed-up. The gcc baseline and performance models are also shown in figure 2. From this we see that our implementation based performance model is consistent with the baseline run using the gcc compiler.

6 Modifications

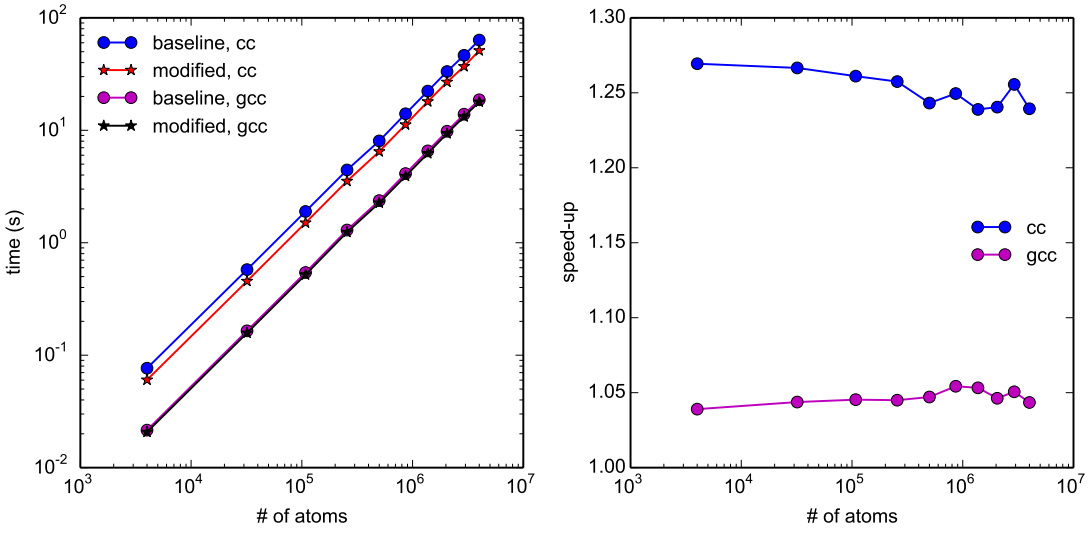

Figure 3: Runtime and speedups from the modifications described in section 6

The performance baseline revealed a large discrepancy between our performance model and experimental results for the Cray compiler. We have focused our modifications on attempting to improve the results using the Cray compiler. The results using the gcc compiler agree well

with our performance model, and further improvements would require changes to the underlying data structures or a complete rewrite of the force calculation routines, both of which are beyond the scope of this project. The results from our modifications are shown in figure 3.

The force compute function uses a large number of pointer dereferences to access the arrays storing particle information. For example, accessing the m^{th} component of the position vector corresponding to particle i requires two pointer dereferences: `s->atoms->r[iOff][m]`. From the code report generated by the Cray compiler, it was clear that these dereferences were being done **each time** the pointers were accessed in the innermost loop of the force calculation. Thus, a simple optimization is to dereference these pointers outside the computation loops. In addition, small loops over a set of 3 real variables are frequently used to work with position and force vectors. Not only does this prevent the pipelining of instructions, but this also requires additional clock cycles to execute branch instructions. Accordingly, we unrolled all of these loops by hand. The inner loop of the force calculation routine is shown before and after optimization in appendix A.

The Cray compiler reports tell us that no vectorization occurs during the force calculation. After some investigation, we determined that this is, in fact, the correct behavior as the data structure does not guarantee that pointers to arrays associated with particle data are not aliased. Aliasing does occur when calculating the interactions between particles within the same cell. We considered modifying the data structure to remove aliasing; however, we decided that such substantial modifications were beyond the scope of this project.

Another potential improvement considered was changing the order of particles within a cell to improve memory access patterns. However, as the current implementation already sorts the particles within a cell regularly, it is unlikely that we could substantially improve upon its current performance.

The above modifications led to a 1.24-1.27 speed-up using the Cray compiler; however, runtimes are still much higher than our performance expectations. With these changes the gcc compiler only experienced a speed-up of 1.04-1.05. It is likely that the gcc compiler had already made similar optimizations, so our manual editing made only minor improvements.

7 Conclusion

We ran scaling studies and profiled the `CoMD` code base to determine routines that could be most profitably optimized. We determined that the most likely place for performance improvement was the Lennard-Jones force compute function. We developed both ideal and implementation specific performance models for the Lennard-Jones force compute function. These models were used as guides while attempting optimization. We obtained speed-ups of 1.24-1.27; however, the biggest differences in performance were due to the choice of compiler. We tried a separate compiler (gcc) as a direct result of the failure of the Cray compiled code to meet performance expectations, a step we would likely not have taken otherwise.

References

[1] https://exmatex.github.io/CoMD/

[2] Allen, Michael P. Introduction to Molecular Dynamics Simulation. *Computational Soft Matter: From Synthetic Polymers to Proteins, Lecture Notes*, Norbert Attig, Kurt Binder, Helmut Grubmuller, Kurt Kremer (Eds.), John von Neumann Institute for Computing, Julich, NIC Series, Vol. 23, ISBN 3-00-012641-4, pp. 1-28, 2004.

[3] http://en.wikipedia.org/wiki/Lennard-Jones_potential

A Code changes

The changes made to the inner loop in the force cacluation are shown here.

A.1 Initial Inner Force Loop

```
real_t dr[3];
int jId = s->atoms->gid[jOff];
if (jBox < s->boxes->nLocalBoxes && jId <= iId )
continue; // don't double count local-local pairs.
real_t r2 = 0.0;
for (int m=0; m<3; m++)
{
  dr[m] = s->atoms->r[iOff][m]-s->atoms->r[jOff][m];
  r2+=dr[m]*dr[m];
}

if ( r2 > rCut2) continue;

r2 = 1.0/r2;
real_t r6 = s6 * (r2*r2*r2);
real_t eLocal = r6 * (r6 - 1.0) - eShift;
s->atoms->U[iOff] += 0.5*eLocal;
s->atoms->U[jOff] += 0.5*eLocal;

if (jBox < s->boxes->nLocalBoxes)
ePot += eLocal;
else
ePot += 0.5 * eLocal;

real_t fr = - 4.0*epsilon*r6*r2*(12.0*r6 - 6.0);
for (int m=0; m<3; m++)
{
  s->atoms->f[iOff][m] -= dr[m]*fr;
```

```
    s->atoms->f[jOff][m] += dr[m]*fr;
  }
```

A.2 Optimized Inner Force Loop

```
real_t dr[3];
int jId = gid[jOff];
if (jBox < nLocalBoxes && jId <= iId )
continue; // don't double count local-local pairs.

real_t r2 = 0.0;
dr[0] = r_iOff[0]-r[jOff][0];
r2+=dr[0]*dr[0];
dr[1] = r_iOff[1]-r[jOff][1];
r2+=dr[1]*dr[1];
dr[2] = r_iOff[2]-r[jOff][2];
r2+=dr[2]*dr[2];

if ( r2 > rCut2) continue;

r2 = 1.0/r2;
real_t r6 = s6 * (r2*r2*r2);
real_t eLocal = r6 * (r6 - 1.0) - eShift;
U[iOff] += 0.5*eLocal;
U[jOff] += 0.5*eLocal;

if (jBox < nLocalBoxes)
ePot += eLocal;
else
ePot += 0.5 * eLocal;

real_t fr = - 4.0*epsilon*r6*r2*(12.0*r6 - 6.0);
f_iOff[0] -= dr[0]*fr;
f[jOff][0] += dr[0]*fr;
f_iOff[1] -= dr[1]*fr;
f[jOff][1] += dr[1]*fr;
f_iOff[2] -= dr[2]*fr;
f[jOff][2] += dr[2]*fr;
```