CS 598 Final Project

Scott High and Erin Molloy

May 4, 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 xx-xx%.

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

2 Background

A molecular dynamics simulation models the movements of individual particles over time. Particle motion is determined by Newton's well-known equation of motion, $F = ma$. In an N-particle simulation, the force acting on particle i at position $\boldsymbol{r}_i = (x_i, y_i, z_i)$ 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 force using the truncated Lennard-Jones potential, where the the potential between a pair of particles, call i and j, 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,

$$r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}, \tag{3}$$

is the distance between particles and $r_c = 2.5\sigma$ is the cut-off radius for particle-particle interactions. In particular, the cut-off radius defines a sphere with volume within which pair potentials must be computed. Material parameters ϵ and σ are the potential well depth and distance at which the pair potential becomes zero, respectively [3].

3 Performance baseline

`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

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

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

In a scaling study, we use these 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. Table 1 shows the difference in runtimes for the force compute function and the halo exchange. As the former dominated total runtime, we target the force computation function in order to improve the overall performance.

In our performance baseline, we use default parameters with a temperature of 0 K so that particles remain in their initial positions throughout the simulation. This enables us to model the number of particle-particle interactions as the constant value. Figure 1 shows the runtimes for the force computation on a single processor with the problem size increasing from 4000 to 4000000 atoms. Runtimes increase linearly in the number of atoms.

All scaling and baseline results are obtained on Blue Waters by compiling `CoMD` with the default Cray compiler (optimization flags: `-g -O3`).

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.

		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)

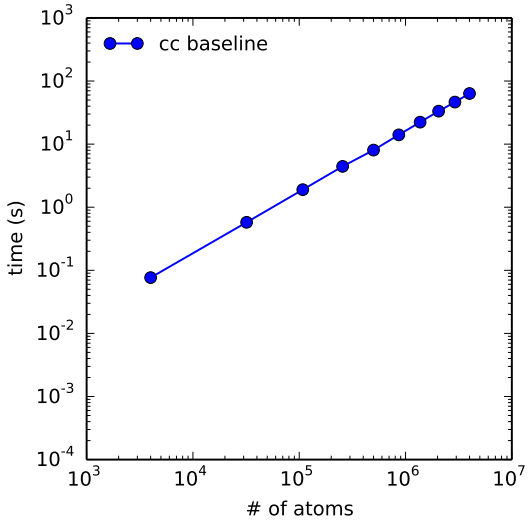

Figure 1: Time spent in force computation function on a single processor

4.1 Computation-based performance model

Our performance 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

$$U_{LJ}(r_{ij}) = A\left(\frac{1}{r_{ij}}\right)^6\left\{\left(\frac{1}{r_{ij}}\right)^6 - 1\right\}, \tag{5}$$

and the resulting force $\mathbf{F} = (F_x, F_y, F_z)$ is computed as

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

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

$$E_{tot} = \sum_{ij} U_{LJ}(r_{ij}), \tag{7}$$

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

(A) 3 loads to get particle i's position, $r_i = (x_i, y_i, z_i)$

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

1. 3 loads to get particle j's position, $r_j = (x_j, y_j, z_j)$
2. 3 subtractions, 3 multiplications, 2 additions, and 1 division to calculate the term, $\frac{1}{r_{ij}^2}$ (equation 2)
3. 3 multiplications to calculate the term, $\left(\frac{1}{r_{ij}^2}\right)^3 = \left(\frac{1}{r_{ij}}\right)^6$
4. 2 subtractions and 2 multiplications to calculate the potential U_{LJ} (equation 5), where 1 subtraction is for cutoff potential (equation 1)
5. 3 multiplications and 1 subtraction to calculate magnitude of the force, $F(r_{ij})$ (equation 6)
6. 3 multiplications and 3 additions to calculate and update the force vector, $\mathbf{F}$

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

(D) 3 writes to save $\mathbf{F} = (F_x, F_y, F_z)$

(E) 1 addition to update E_{tot} (equation 7)

The total cost is then

$$N\left(A + \frac{nB}{2}\right) + C + D + E, \tag{8}$$

where the factor of $1/2$ accounts for double counting particles in pairwise interactions, that is, the interaction between particle i and particle j is the same as the interaction between particle j and particle i making it unnecessary to compute both 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 can be approximated as

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

which reduces to

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

assuming equal read and write times, call m. Specifically, we approximate $c = 4.35 \times 10^{-10}$ flops/s using the processor clock rate, $m = 1.43 \times 10^{-9}$ B/s using the STREAM benchmark on Bluewaters, and $n = 69$ from the known cut-off radius and lattice parameters (equation 4). The results of this performance model are shown as model 1 in figure **??** for increasing N. Although both our results and the computation-based performance model scale linearly with the number of atoms. Our performance expectation is 3 orders of magnitude less than our results!

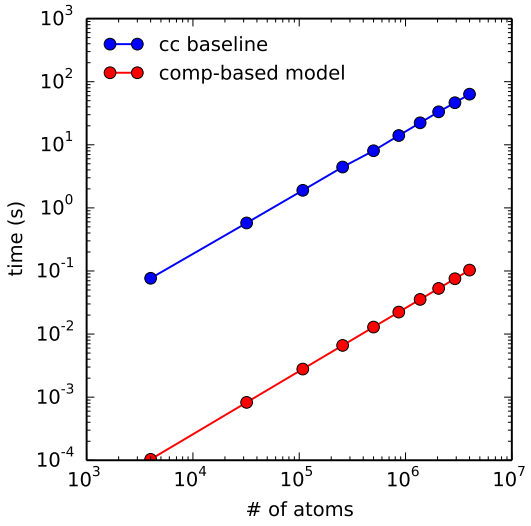

Figure 2: Computation-based performance model and baseline results

4.2 Implementation-based model

5 Compiler choice

The computation-based performance model 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 features of 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 an additional 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 can be updated

$$N\left(\frac{n}{2}\left(26c+3r\right)+\frac{k}{2}\left(8c+3r\right)+3r\right)+c+4r \tag{11}$$

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 11 to give

$$N\left(n\left(26c+3r\right)+k\left(8c+3r\right)+3r\right)+c+4r. \tag{12}$$

The results of these implementation-based performance model are shown as model # in figure for increasing N. These implementation-based performance models scale linearly with the number of atoms and appear much closer to our experimental results; however, our performance expectation is still 2-2.5 orders of magnitude less than our baseline!

6 Compiler choice

A baseline for the force compute routine is established by running `CoMD`, specifically using the default Cray compiler (with optimization flags: `-g -O3`). These baseline results compare poorly with the performance models as shown in figure **??**. We were unable to determine why the Cray compiler has such poor performance and obtained an additional baseline using the `CoMD` code compiled with gcc 4.7.1. This compiler change alone resulted in a 3.5 speed-up. The gcc baseline is only 2.39 orders of magnitude off from our computation-based model and 1.5-1.8 orders of magnitude off from our implementation-based models.

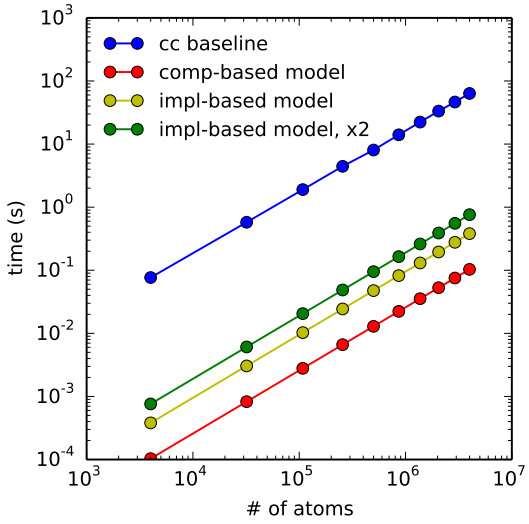

Figure 3: Implementation-based performance models and baseline results

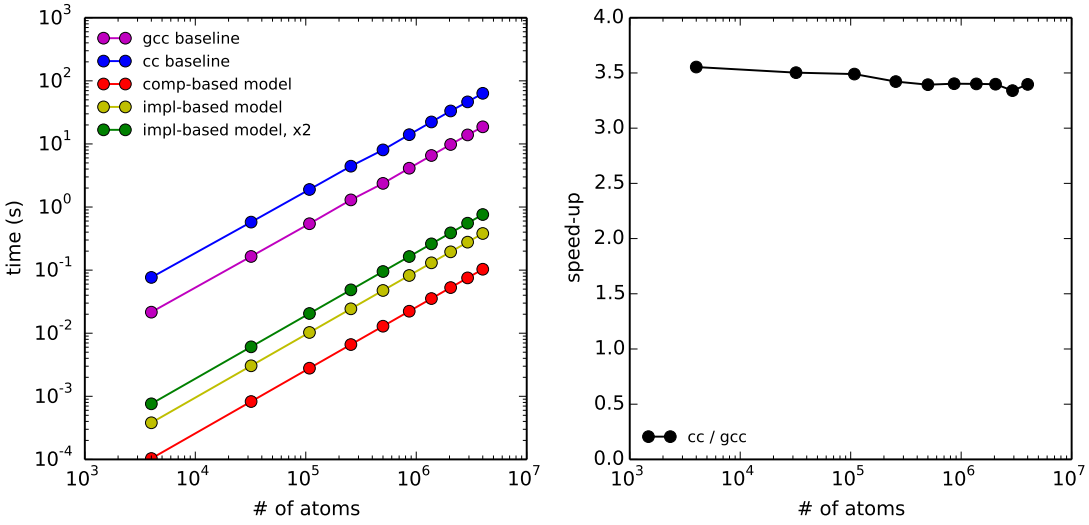

Figure 4: Effects of compiler choice

7 Modifications

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 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 (A).

In addition, small loops over a set of 3 real variables are frequently used to work with position $\boldsymbol{r}$ and force $\mathbf{F}$ 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. Specifically, all particle information is stored in contiguous arrays and 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.

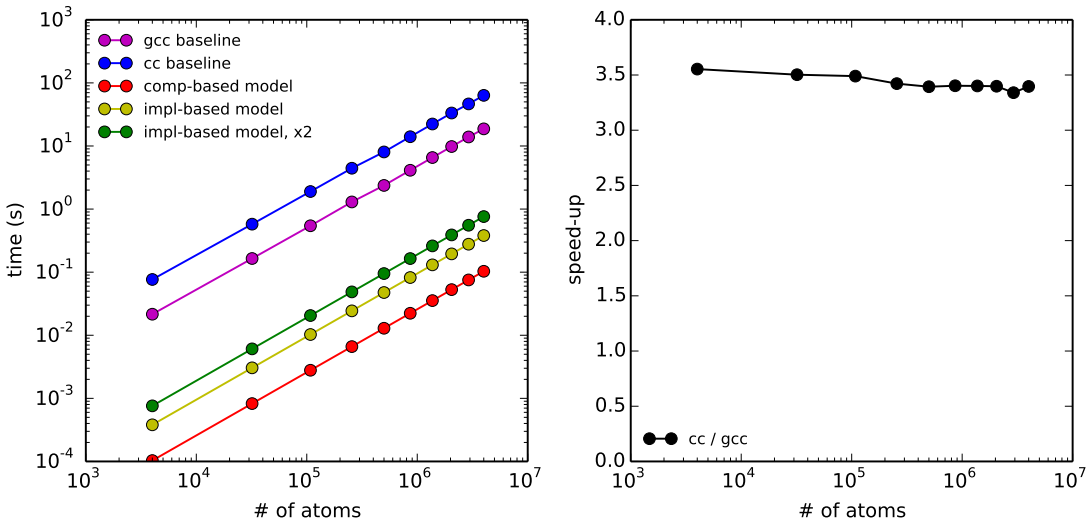

Figure 5: Runtime and speedups from the modifications described in section 7

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 several orders of magnitude higher than our performance expectations. As the gcc compiler only experienced a speed-up of 1.04-1.05, it is possible that the gcc compiler had already made some of our modifications; however, we have not been able to identify the causes of the remaining discrepancy between compilers as both compiler reports show no vectorization.
Even after these modifications, we still need speed-ups of 67-79 and 24-27 for the cc and gcc compilers, respectively, to obtain experimental runtimes close to our performance expectation.

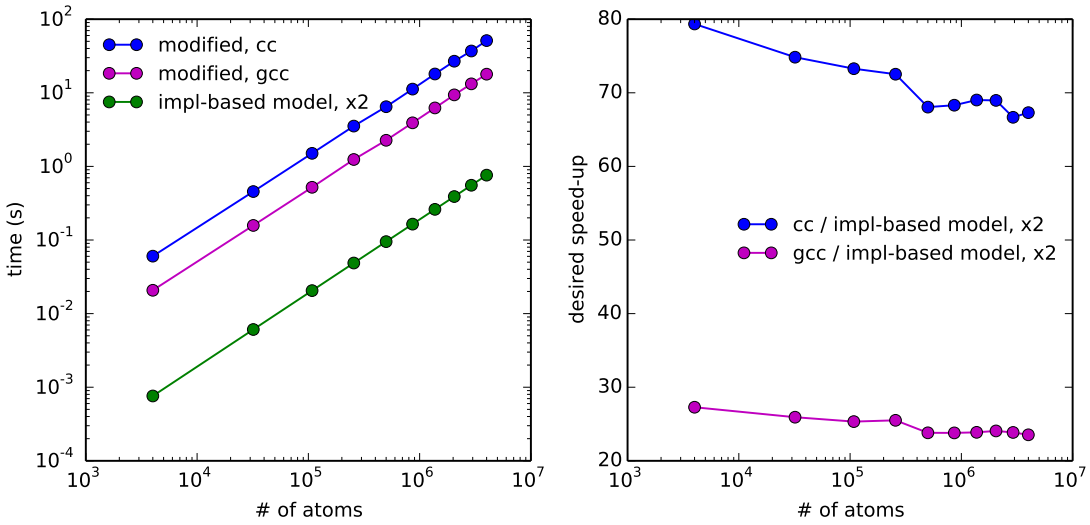

Figure 6: Desired speed-ups

8 Conclusion

In summary, we ran scaling studies and profiled the `CoMD` code base to determine a routine that could be targeted to enhance performance. We then created a **performance expectation** for the Lennard-Jones force compute function and used this performance model to make modifications to this computationally intensive routine. We obtained speed-ups of #-#; however, the biggest differences in performance were due to the choice of compiler.
Finally, we adapted our performance expectation to include implementation details, which could allow others to determine how changing the data structure to reduce accesses to particles outside the cut-off radius would impact performance before spending time implementing such substantial changes to `CoMD`.

References

[1] ADD ME

[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;
```