Implementation of classical molecular dynamics algorithm using Intel Concurrent Collections

Riyaz Haque
haque1@llnl.gov
Lawrence Livermore National Laboratory

Abstract

We discuss an implementation of the classical molecular dynamics (MD) simulation algorithm using Intel Concurrent Collections [1]. For this work, we primarily consider materials represented using short-range forces - specifically the Lennard-Jones potential, although the model can certainly be extended to consider more complex forces. We decompose the problem by a *linked-cell* approach wherein the simulation space is divided into fixed-sized cells. The cell size is chosen to be greater than the cutoff distance for particle interaction, which ensures that a particle can interact only with particles located either in its own cell or its neighboring cells. During each simulation timestep, a cell computes forces due to interactions within itself and with its neighboring cells and accordingly updates particle velocities and positions. A valid simulation conserves overall system energy in each timestep.

By construction, each cell proceeds independently of other cells as long as its data and control dependencies with (only) its neighboring cells are satisfied. Our code is written as a set of step-collections. Each iteration creates an *integration* step per cell for updating the velocities and positions of all particles within a cell, a *redistribution* step for exchanging particles between neighboring cells and an *interaction* step for force calculation per neighboring cell-pair. A *reduction* step is used to calculate the total force on each particle per iteration. Another reduction step computes the total system energy at regular intervals while an *output* step writes the simulation output to a file. Dependencies are specified using item- and tag-collections. Use of CnC tuners allows optimizations such as optimally co-locating data and computation to be carried out separately from the application logic.

1 Introduction

Molecular dynamics simulation [3] finds widespread use in many scientific and engineering applications. Several parallel algorithms exist for solving this problem [2, 4]. This makes it an important HPC benchmark for evaluating parallel and distributed computing paradigms.

Classical molecular dynamics is the N-body simulation of the physical movement of atoms and molecules. For a given time interval (timestep), the algorithm first calculates forces between particles governed by a potential function and uses them to advance particle positions accordingly. Besides particle motion, thermodynamic properties of the whole system, such as overall temperature and energy are also tracked during each timestep. A typical simulation either runs for a fixed number of timesteps or terminates when a system property converges to some desired value.

We now discuss the computational model and decomposition strategies for our code.

1.1 Computational Model

1.1.1 Force Equation

We only consider short-range interactions between homogeneous particles based on the commonly used Lennard-Jones potential [ref]. We believe that with some effort, the work can be extended to handle more complex and long-range interactions.

The Lennard-Jones (LJ) [6] potential approximates interaction between a pair of neutral particles. For a distance r between a pair of particles the LJ-potential $V_{LJ}(r)$ is given by

$$V_{LJ}(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{1}$$

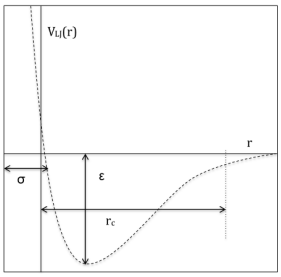

Figure 1: Lennard-Jones potential

where ϵ the depth of the potential well and σ is a finite distance at which the inter-particle potential is zero as shown in Fig. 1.

The r^{12} term is the repulsive force due to Paulis exclusion principle [ref], while the r_6 term represents the attractive van der Waals force[ref]. To reduce the computational effort, we consider interactions only at distances less than some cutoff distance r_c beyond which the potential becomes negligible.

The corresponding LJ-force between two particles p and q is given by

$$F_i(p,q) = -24\epsilon \left[2\left(\frac{\sigma^{12}}{{r_{pq}}^{13}}\right) - \left(\frac{\sigma^{6}}{{r_{pq}}^{7}}\right)\right] \tag{2}$$

where r_{pq} is the distance between p and q

Let t_i denote the timestep i. Let $N_i(p)$ be the *neighborhood* of p i.e. the set of all particles p can interact with at t_i. If $x_i(p)$ is the position of particle p at t_i, then $N_i(p)$ is a function of $x_i(p)$ defined as

$$N_i(p) = \bar{N}_i(x_i(p)) = \{q \mid |x_i(p) - x_i(q)| \leq r_c\} \tag{3}$$

For any $q \notin N_i(p)$, $F_i(p,q) = 0$

Then at t_i the overall force acting on a particle p is given by

$$F_i(p) = \bar{F}_i(p, N_i(p)) = \sum_{q \in N_i(p)} F_i(p,q) \tag{4}$$

1.1.2 Timestep Integration

We use the standard leapfrog integration [5] technique to calculate the velocity $v_i(p)$ and position $x_i(p)$ of particle p at timestep t_i as follows

$$v_{i-1/2}(p) = v_{i-1}(p) + \frac{F_{i-1}(p)}{m(p)} \times \Delta t \tag{5}$$

$$x_i(p) = x_{i-1}(p) + v_{i-1/2}(p) \times \Delta t \tag{6}$$

$$N_i(p) = \bar{N}_i(x_i(p)) \tag{7}$$

$$F_i(p) = \bar{F}_i(p, N_i(p)) \tag{8}$$

$$v_i(p) = v_{i-1/2}(p) + \frac{F_i(p)}{m(p)} \times \Delta t \tag{9}$$

where $m(p)$ is the mass of p. Equation (8) calculates the new force interactions based on the updated position of p

1.1.3 Energy Equation

At each t_i we can calculate the total potential and kinetic energy of the system. Potential energy of a particle p, $PE_i(p)$ easily follows from equation (1) during force computation (equation (8)). The kinetic energy $KE_i(p)$ is calculated as

$$KE_i(p) = \frac{1}{2} \times m(p) \times {v_i(p)}^2 \tag{10}$$

The total energy E_i for the system is calculated as

$$E_i = \sum_{\forall\ p} PE_i(p) + KE_i(p) \tag{11}$$

The simulation is valid only if the total energy is conserved per timestep i.e. for any timesteps t_i and t_j, $E_i \approx E_j$

The high-level algorithm for our MD simulation is as follows
For each particle p:

1. Read $x_0(p)$ and $v_0(p)$ from the input file input file
2. At t_0: Compute $N_0(p)$ and $F_0(p)$
3. For $t_1 \leq t_i \leq t_n$
   (a) Calculate $v_{i-1/2}(p)$
   (b) Calculate $x_i(p)$ and $N_i(p)$
   (c) Calculate $F_i(p)$ and $PE_i(p)$
   (d) Calculate $v_i(p)$ and $KE_i(p)$
   (e) Calculate E_i

As an optimization, we compute $PE_i(p)$, $KE_i(p)$ and E_i only at intervals where an output has to be generated.

1.2 Problem Decomposition

For computing its neighborhood $N_i(p)$, a particle p needs find out other particles within its cutoff distance. In a naive implementation, every particle checks the entire particle list for locating other atoms within its cutoff distance. To avoid this inefficient $O(n^2)$ search of the whole simulation box, we decompose the space around each particle and limit our search to only this small region. There are two widely used decomposition approaches.

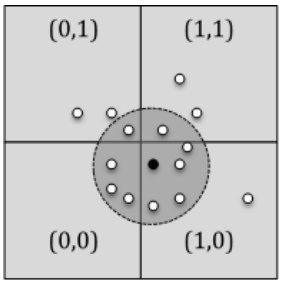

Figure 2: Linked-cell decomposition

1.2.1 Linked-Cell

In this approach, the entire simulation space is spatially divided into fixed-sized cells. Each cell maintains a list of particles that lie within it. The cell size is chosen to be greater than the cutoff distance. Thus every particle is guaranteed to interact only with other particles within its own cell or one of its neighboring cells as shown in Fig. 2. This reduces our search space to a much smaller finite-sized neighborhood around a particles cell. This method is useful in cases where particles move a lot during the simulation. The obvious disadvantage of it is that each particle has to search through all the particles in its cell's neighborhood even though it actually interacts with a small number of them.

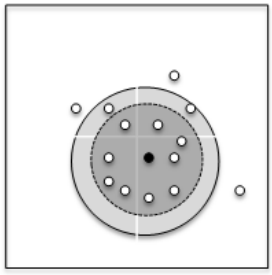

Figure 3: Neighbor-list decomposition

1.2.2 Neighbor-List

This method maintains two lists for each particle; a *neighbor-list* comprising of all particles it actually interact with and another list called the *halo region*, consisting of particles lying within some small width region outside the cutoff as seen in Fig. 3. This approach reduces the search space for each particle to its valid interactions. In a timestep, if any particle moves more than half the width of the halo region, the neighbor lists are recalculated for all particles. Clearly, this works best if particles do not move significantly. This approach however typically requires some external mechanism like a space-filling curve for ensuring locality.

In CnC, choice of good tags is essential for programmability and optimization. Here linked-cell offers a number of advantages over neighbor-list. Each cell has a unique cell-id based on its position within the simulation space; tags are based on cell-ids. This enables optimal co-location of items and their consumer steps in a straightforward manner. It also facilitates implementation of communication-optimal distribution strategies for mapping neighboring cells on physically proximate nodes. This would be tedious to accomplish with neighbor-list. Moreover, since each cell has a finite number of neighbors, specifying data and control dependencies is easier. This is especially true for reductions, which in CnC prefer the reduction counts to be known ahead of time. Lastly, but importantly, cell-size also provides a simple control on code granularity. Hence we use the linked-cell approach in our case.

We now describe our MD implementation in CnC. The rest of the report is organized as follows: Section 2 gives an overview of the code and the tuners, Section 3 evaluates preliminary performance results and Section 4 discusses conclusions and future work.

2 Implementation

Our implementation solves the MD problem for a 3-dimensional grid of particles. The computation proceeds along the same lines as the computational model described in Section 1.1 albeit at the granularity of a cell (set of particles) rather than a single particle. The simulation box dimensions and the initial positions and velocities of particles are read from an input file. Runtime configurable program properties are the number of cells in each dimension (and thus the cell size), number of iterations, timestep length, epsilon, sigma, cutoff distance and output interval. Configurable cell-size allows for changing the code granularity

depending on input size. In this section, we summarize the important aspects of our code. Appendix A gives a complete dependency graph for the code.

2.1 Implementation Overview

Following are the primary program components:

Force Computation

Force computation is the most significant aspect of entire algorithm. It is invoked first from the initialization step and subsequently from the redistribution step. It proceeds in two stages:

1. **Computing partial forces between neighbors:** Given a cell c, let $P_i(c)$ denote the set of particles within the cell. Given two cells $c1$ and $c2$ we compute the force of interaction $F_i(c1, c2)$ as follows

$$F_i(c1, c2) = \left\{ \sum F_i(p, q) \mid p \in P_i(c1) \wedge q \in P_i(c2) \right\}$$

In other words, $F_i(c1, c2)$ is the set of individual forces on the particles of $c1$ due to interaction with particles of cell $c2$. Since the cell-size is greater than the cutoff-distance, for any non-neighbor cells, the force of interaction is zero. For every cell, the force of interaction with each individual neighbor is computed in parallel by a separate *interaction step*. The case $F_i(c, c)$, is handled locally on the same thread as the step that invokes force computation.

Half-neighbor computation: Since LJ-forces are symmetric, for any cell-pair $c1$ and $c2$, $F_i(c1, c2) = -F_i(c2, c1)$. It is therefore sufficient to calculate the force of interaction only once per neighboring cell-pair. Aided by the fact that each cell has a unique global id, we accomplish this by creating the interaction step only if a cell's index is greater than its neighbor's index.

Periodic boundary conditions: If a particle moves outside the simulation box in a periodic dimension, it reappears on the other side of the box; in other words, in that dimension, the simulation box wraps around itself. Along such boundaries we need to also consider interaction between cells on the box extremities. This is done by treating such cells as neighbors and appropriately shifting particle co-ordinates by the length of the simulation box in the periodic dimension. For each neighboring cell, we use a 32-bit integer called the *shift flag* that stores -1, 0 or +1 per dimension depending on whether the neighboring cell lies to the left of the -ve face, inside or to the right of the +ve face of the simulation box in that dimension. The least significant 8-bits of the shift vector are flags for the z-dimension and next significant 8 bits for y and then the x-dimension respectively. The most significant 8-bits of a shift flag are always set to 0.

2. **Force reduction:** The total force on particles of a cell is given by

$$F_i(c) = \left\{ \sum F_i(c, c')[p] \mid p \in P_i(c) \wedge c' \in N_i(c) \right\}$$

where let $N_i(c)$ denotes the neighbors of c.

We use the CnC reduction graph for this part. Since a cell's neighbors are known, the reduction counts can be provided easily. The final force value is used to update velocities in the integration step.

Timestep Integration

This part iterates of the cell's particles and computes the new velocity and position for each of them. As an optimization, we perform the calculation of $v_{i-1}(p)$, $v_{i-1/2}(p)$ and $x_i(p)$ in the same step to avoid redundant data movement and step creation. Additionally if system parameters need to be copied to a file, this step invokes the necessary output step.

Redistribution

This step is invoked after the integration step to redistribute particles that have moved to their new cells. We assume that a particle will never move more than once cell distance in a single timestep. Hence we exchange particles between neighboring cells only.

Cell Initialization

Cell initialization is invoked from the environment and is executed for each cell. It constructs the cell's neighbor list and initializes the list of particles within the cell. Based on the starting positions of the particles, it also invokes initial force calculation for the cell particles.

Output

This step is invoked on specified intervals to write the generated data to an output file. Written to the output are the particle positions and velocities and total energy for that timestep.

2.2 Tuners

Tuners are indispensable for good performance. Preliminary results indicate that tuned code runs on the order of 25X faster than untuned code. We tune the code primarily for optimal co-location of data and computation and garbage collection.

Co-location: Since tags are based on cell-ids, tuning for co-location essentially becomes a problem of mapping cells to processors. We abstract this mapping as an interface. This decouples the underlying cell distribution specification from the application logic. We then implement the tuner methods `compute_on` and `consumed_on` as callbacks to this interface using the cell-id extracted from the tag. This ensures that all data a step needs will be always located locally, without actually knowing where it is allocated. Also, once data is guaranteed to be local, we can store smart pointers in the item collections rather than the actual values. We store a cell's neighbor list and particle list in this manner.

Garbage Collection: Almost all of the collections are accessed a constant number of times (mostly once per timestep). Hence specifying the `get_count` is trivial.

Scheduling dependencies: In most cases, scheduler dependencies can be specified using `depends_on()`. However, at two places - the integration and redistribution steps - this is not possible due to dynamic dependencies on accessing the cell neighbors.

3 Performance

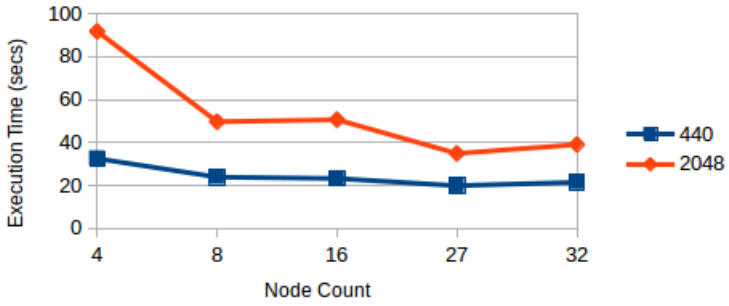

Figure 4: CnC MD Performance

Currently, we have a very preliminary performance analysis for our code. We tested the distributed version of our code on two simulations, containing 440 and 2048 particles, on an Intel Xeon EP X5660 cluster using mpavich2-intel-1.7 as the MPI library. The simulation test cases were decomposed into 3 cells in each dimension (total 27 cells). Fig. 4 shows the execution time for 4, 8, 16 , 27 and 32 nodes. The numbers show measurable performance improvement for the 2048-atom case as the number of nodes approaches the number of cells. For the 440-atom case, the performance improvement is quite nominal. This is most probably due to the small input size. We are currently in the process of running the code for larger input sizes.

4 Conclusions

We implemented a version of the classical molecular dynamics simulation in CnC. This model differs from other conventional programming models and some effort is required to specify the appropriate dependencies. However, once that is done the actual coding effort required is fairly low. The model also allows for writing application logic in a runtime-agnostic manner. We are currently working on evaluating code performance for larger inputs. For future, we want to further optimize the code and analyze its performance in comparison with reference implementations. We also plan to use data-parallel techniques in conjunction with CnC to parallelize local sections of our code. Lastly, we plan to implement distribution strategies for load balancing, optimal cell placement etc.

Appendix

A Complete dependency graph for CnC MD

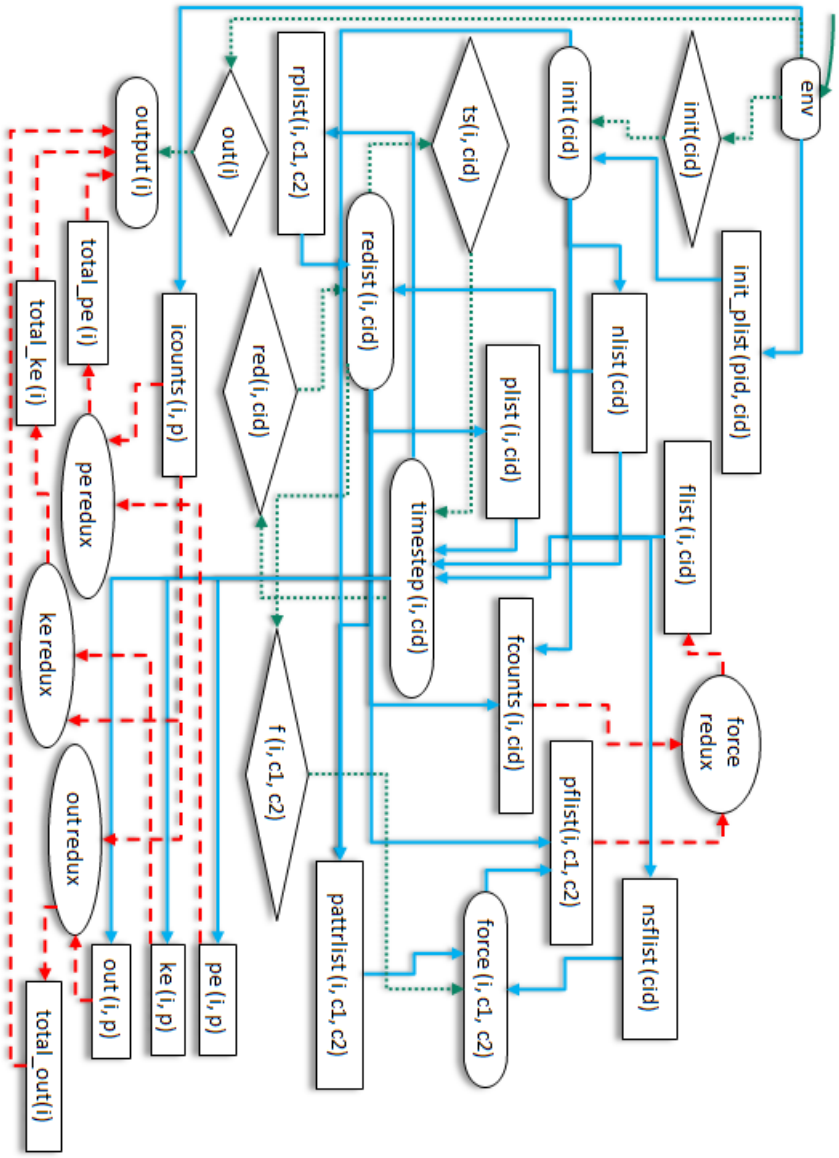

Figure 5: MD dependency graph

Fig. 5 shows the detailed dependency graph for the MD code. For brevity, variable and type names have been changed but a vis-à-vis comparison with the actual code should still be obvious.

Bibliography

[1] Intel. Concurrent collections. https://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc, 2014.

[2] Sandia. Lammps. lammps.sandia.gov/, 2014.

[3] Sandia. Molecular dynamics simulation. http://www.sandia.gov/~sjplimp/md.html, 2014.

[4] UIUC-Charm++. Namd. www.ks.uiuc.edu/Research/namd/, 2014.

[5] Wikipedia. Leapfrog integration. http://en.wikipedia.org/wiki/Leapfrog_integration, 2014.

[6] Wikipedia. Lennard-jones potential. http://en.wikipedia.org/wiki/Lennard-Jones_potential, 2014.