Liquid Argon Molecular Dynamics

Dr. Axel Kohlmeyer
Senior Scientific Computing Expert
Information and Telecommunication Section
The Abdus Salam International Centre for Theoretical Physics

http://sites.google.com/site/akohlmey/

akohlmey@ictp.it
The Model for Liquid Argon

- Cubic box of particles with a Lennard-Jones type pairwise additive interaction potential

\[ U(r) = \sum_{i,j} \begin{cases} 4\varepsilon \left[ \left( \frac{\sigma}{r_{ij}} \right)^{12} - \left( \frac{\sigma}{r_{ij}} \right)^{6} \right], & r_{ij} < r_c \\ 0, & r_{ij} \geq r_c \end{cases} \]

- Periodic boundary conditions to avoid surface effects
Newton's Laws of Motion

- We consider our particles to be *classical objects* so Newton's laws of motion apply:
  1. In absence of a force a body rests or moves in a straight line with constant velocity
  2. A body experiencing a force $\mathbf{F}$ experiences an acceleration $\mathbf{a}$ related to $\mathbf{F}$ by $\mathbf{F} = m\mathbf{a}$, where $m$ is the mass of the body.
  3. Whenever a first body exerts a force $\mathbf{F}$ on a second body, the second body exerts a force $-\mathbf{F}$ on the first body (Bonus Law)
Velocity-Verlet Algorithm

- The Velocity-Verlet algorithm is used to propagate positions and velocities of the atoms

\[ \begin{align*}
\vec{x}_i(t + \frac{\Delta t}{2}) &= \vec{x}_i(t) + \frac{1}{2} \vec{v}_i(t) \Delta t + \frac{1}{2} \vec{a}_i(t) (\Delta t)^2 \\
\vec{v}_i(t + \frac{\Delta t}{2}) &= \vec{v}_i(t) + \frac{1}{2} \nabla V(\vec{x}_i(t + \frac{\Delta t}{2})) \Delta t \\
\vec{a}_i(t + \Delta t) &= \vec{a}_i(t) + \frac{1}{2} \nabla V(\vec{x}_i(t + \Delta t)) \Delta t \\
\vec{x}_i(t + \Delta t) &= \vec{x}_i(t) + \vec{v}_i(t + \frac{\Delta t}{2}) \Delta t + \frac{1}{2} \vec{a}_i(t) (\Delta t)^2 \Delta t \\
\vec{v}_i(t + \Delta t) &= \vec{v}_i(t + \frac{\Delta t}{2}) + \frac{1}{2} \vec{a}_i(t + \frac{\Delta t}{2}) (\Delta t)^2 \Delta t \\
\end{align*} \]


![Force calculation](image)
What Do We Need to Program?

1. Read in parameters and initial status and compute what is missing (e.g. accelerations)

2. Integrate Equations of motion with Velocity Verlet for a given number of steps
   a) Propagate all velocities for half a step
   b) Propagate all positions for a full step
   c) Compute forces on all atoms to get accelerations
   d) Propagate all velocities for half a step
   e) Output intermediate results, if needed
Initial Serial Code: Velocity Verlet

```c
void velverlet(mdsys_t *sys) {
    for (int i=0; i<sys->natoms; ++i) {
        sys->vx[i] += 0.5*sys->dt / mvsq2e * sys->fx[i] / sys->mass;
        sys->vy[i] += 0.5*sys->dt / mvsq2e * sys->fy[i] / sys->mass;
        sys->vz[i] += 0.5*sys->dt / mvsq2e * sys->fz[i] / sys->mass;
        sys->rx[i] += sys->dt*sys->vx[i];
        sys->ry[i] += sys->dt*sys->vy[i];
        sys->rz[i] += sys->dt*sys->vz[i];
    }

    force(sys);

    for (int i=0; i<sys->natoms; ++i) {
        sys->vx[i] += 0.5*sys->dt / mvsq2e * sys->fx[i] / sys->mass;
        sys->vy[i] += 0.5*sys->dt / mvsq2e * sys->fy[i] / sys->mass;
        sys->vz[i] += 0.5*sys->dt / mvsq2e * sys->fz[i] / sys->mass;
    }
}
```
Initial Code: Force Calculation

```c
for(i=0; i < (sys->natoms); ++i) {
    for(j=0; j < (sys->natoms); ++j) {
        if (i==j) continue;
        rx=pbc(sys->rx[i] - sys->rx[j], 0.5*sys->box);
        ry=pbc(sys->ry[i] - sys->ry[j], 0.5*sys->box);
        rz=pbc(sys->rz[i] - sys->rz[j], 0.5*sys->box);
        r = sqrt(rx*rx + ry*ry + rz*rz);
        if (r < sys->rcut) {
            ffac = -4.0*sys->epsilon*(-12.0*pow(sys->sigma/r,12.0)/r
                                +6*pow(sys->sigma/r,6.0)/r);
            sys->epot += 0.5*4.0*sys->epsilon*(pow(sys->sigma/r,12.0)
                                           -pow(sys->sigma/r,6.0));
            sys->fx[i] += rx/r*ffac;
            sys->fy[i] += ry/r*ffac;
            sys->fz[i] += rz/r*ffac;
        }
    }
}
```

Compute distance between atoms i & j

Compute energy and force

Add force contribution of atom j on atom i
How Well Does it Work?

- Compiled with:
  ```
gcc -o ljmd.x -pg ljmd.c -lm
  ```

Test input: 108 atoms, 10000 steps: 49s

Let us get a profile (using gprof):

```plaintext
<table>
<thead>
<tr>
<th>% cumulative time</th>
<th>self seconds</th>
<th>calls</th>
<th>self ms/call</th>
<th>total ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>73.70</td>
<td>13.87</td>
<td>10001</td>
<td>1.39</td>
<td>1.86</td>
<td>force</td>
</tr>
<tr>
<td>24.97</td>
<td>18.57</td>
<td>346714668</td>
<td>0.00</td>
<td>0.00</td>
<td>pbc</td>
</tr>
<tr>
<td>0.96</td>
<td>18.75</td>
<td>0.18</td>
<td></td>
<td></td>
<td>main</td>
</tr>
<tr>
<td>0.37</td>
<td>18.82</td>
<td>0.07</td>
<td>10001</td>
<td>0.01</td>
<td>ekin</td>
</tr>
<tr>
<td>0.00</td>
<td>18.82</td>
<td>30006</td>
<td>0.00</td>
<td>0.00</td>
<td>azzero</td>
</tr>
<tr>
<td>0.00</td>
<td>18.82</td>
<td>101</td>
<td>0.00</td>
<td>0.00</td>
<td>output</td>
</tr>
<tr>
<td>0.00</td>
<td>18.82</td>
<td>12</td>
<td>0.00</td>
<td>0.00</td>
<td>getline</td>
</tr>
</tbody>
</table>
```
Compiler Optimization

- Use of pbc() is convenient, but costs 25% time => compiling with -O3 should inline it
- Loops should be unrolled for superscalar CPUs => compiling with -O2 or -O3 should do it for us
  
  Time now: 39s (1.3x faster)  
  Only a bit faster than 49s

- Now try more aggressive optimization options: -ffast-math -fexpensive-optimizations -msse3
  
  Time now: 10s (4.9x faster)  
  Much better!

- Compare to LAMMPS: 3.6s => need to do more
Now Modify the Code

- Use physics! Newton's 3\textsuperscript{rd} law: \( F_{ij} = -F_{ji} \)

```c
for(i=0; i < (sys->nats)-1; ++i) {
    for(j=i+1; j < (sys->nats); ++j) {
        rx=pbc(sys->rx[i] - sys->rx[j], 0.5*sys->box);
        ry=pbc(sys->ry[i] - sys->ry[j], 0.5*sys->box);
        rz=pbc(sys->rz[i] - sys->rz[j], 0.5*sys->box);
        r = sqrt(rx*rx + ry*ry + rz*rz);
        if (r < sys->rcut) {
            ffac = -4.0*sys->epsilon*(-12.0*pow(sys->sigma/r,12.0)/r
            +6*pow(sys->sigma/r,6.0)/r);
            sys->epot += 4.0*sys->epsilon*(pow(sys->sigma/r,12.0)
            -pow(sys->sigma/r,6.0));
            sys->fx[i] += rx/r*ffac; sys->fx[j] -= rx/r*ffac;
            sys->fy[i] += ry/r*ffac; sys->fy[j] -= ry/r*ffac;
            sys->fz[i] += rz/r*ffac; sys->fz[j] -= rz/r*ffac;
        }
    }
}
```

Time now: 5.4s (9.0x faster)  Another big improvement
More Modifications

- Avoid expensive math: pow(), sqrt(), division

```c
        c12=4.0*sys->epsilon*pow(sys->sigma, 12.0);
        c6 =4.0*sys->epsilon*pow(sys->sigma, 6.0);
        rcsq = sys->rcut * sys->rcut;
        for(i=0; i < (sys->natoms)-1; ++i) {
            for(j=i+1; j < (sys->natoms); ++j) {
                rx=pbc(sys->rx[i] - sys->rx[j], 0.5*sys->box);
                ry=pbc(sys->ry[i] - sys->ry[j], 0.5*sys->box);
                rz=pbc(sys->rz[i] - sys->rz[j], 0.5*sys->box);
                rsq = rx*rx + ry*ry + rz*rz;
                if (rsq < rcsq) {
                    double r6,rinv; rinv=1.0/rsq;
                    r6=rinv*rinv*rinv;
                    ffac = (12.0*c12*r6 - 6.0*c6)*r6*rinv;
                    sys->epot += r6*(c12*r6 - c6);
                    sys->fx[i] += rx*ffac; sys->fx[j] -= rx*ffac;
                    sys->fy[i] += ry*ffac; sys->fy[j] -= ry*ffac;
                    sys->fz[i] += rz*ffac; sys->fz[j] -= rz*ffac;
                }
            }
        }
```

=> 108 atoms: 4.0s (12.2x faster) still worth it
Improvements So Far

- Use the optimal compiler flags => ~5x faster but some of it: inlining, unrolling could be coded
- Use our knowledge of physics => ~2x faster since we need to compute only half the data.
- Use our knowledge of computer hardware => 1.35x faster. (could be more: SSE/AVX)

We are within 10% (4s vs. 3.6s) of LAMMPS.

- Try a bigger system: 2916 atoms, 100 steps
  Our code: 13.3s  LAMMPS: 2.7s  => Bad scaling with system size
Making it Scale with System Size

- Lets look at the algorithm again: We compute all distances between pairs.

- But for larger systems not all pairs contribute and our effort is $O(N^2)$.

- So we need a way to avoid looking at pairs that are too far away.

$\Rightarrow$ Sort atoms into cell lists, which is $O(N)$. 
The Cell-List Variant

- At startup build a list of lists to store atom indices for atoms that “belong” to a cell.
- Compute a list of pairs between cells which contain atoms within cutoff. **Doesn't change!**
- During MD sort atoms into cells.
- Then loop over list of “close” pairs of cells $i$ and $j$.
- For pair of cells loop over pairs of atoms in them.
- Now we have linear scaling with system size at the cost of using more memory and an O(N) sort.
Cell List Loop

```c
for(i=0; i < sys->npair; ++i) {
    cell_t *c1, *c2;
    c1=sys->clist + sys->plist[2*i];
    c2=sys->clist + sys->plist[2*i+1];

    for (int j=0; j < c1->natoms; ++j) {
        int ii=c1->idxlist[j];
        double rx1=sys->rx[ii];
        double ry1=sys->ry[ii];
        double rz1=sys->rz[ii];

        for(int k=0; k < c2->natoms; ++k) {
            double rx, ry, rz, rsq;
            int jj=c2->idxlist[k];
            rx=pbc(rx1 - sys->rx[jj], boxby2, sys->box);
            ry=pbc(ry1 - sys->ry[jj], boxby2, sys->box);
            ...
```

- 2916 atom time: 3.4s (4x faster), LAMMPS 2.7s
Scaling with System Size

- Cell list does not help (or hurt) much for small inputs, but is a huge win for larger problems

=> Lesson: always pay attention to scaling
OpenMP Parallelization

- OpenMP is directive based
  => code (can) work without them
- OpenMP can be added incrementally
- OpenMP only works in shared memory
  => multi-socket nodes, multi-core processors
- OpenMP hides the calls to a threads library
  => less flexible, but much less programming
- Caution: write access to shared data can easily lead to race conditions
Naive OpenMP Version

```c
#if defined(_OPENMP)
#pragma omp parallel for default(shared) \
    private(i) reduction(+:epot)
#endif
for(i=0; i < (sys->natoms)-1; ++i) {
    double rx1=sys->rx[i];
    double ry1=sys->ry[i];
    double rz1=sys->rz[i];
    [...]
#endif
```

Each thread will work on different values of “i”

```c
    #if defined(_OPENMP)
    #pragma omp critical
    #endif
    {
        sys->fx[i] += rx*ffac;
        sys->fy[i] += ry*ffac;
        sys->fz[i] += rz*ffac;
        sys->fx[j] = rx*ffac;
        sys->fy[j] = ry*ffac;
        sys->fz[j] = rz*ffac;
    }
```

The “critical” directive will let only one thread execute this block at a time

Race condition: “i” will be unique for each thread, but not “j” => multiple threads may write to the same location concurrently

Timings (108 atoms):
1 thread: 4.2s
2 threads: 7.1s
4 threads: 7.7s
8 threads: 8.6s
OpenMP Improvements

- Use **omp atomic** to protect one instruction
  => faster, but requires hardware support
  => some speedup, but serial is faster for 108,
  at 2916 atoms we are often beyond cutoff

- No Newton's 3\textsuperscript{rd} Law:
  => no race condition
  => better scaling, but
  we lose 2x serial speed
  => need 8 threads to
  be faster than **atomic**
MPI-like Approach with OpenMP

```c
#if defined(_OPENMP)
#pragma omp parallel reduction(+:epot)
#endif
{
    double *fx, *fy, *fz;
    #if defined(_OPENMP)
    int tid=omp_get_thread_num();
    #else
    int tid=0;
    #endif
    fx=sys->fx + (tid*sys->natoms);
    azzero(fx,sys->natoms);
    fy=sys->fy + (tid*sys->natoms);
    azzero(fy,sys->natoms);
    fz=sys->fz + (tid*sys->natoms);
    azzero(fz,sys->natoms);
    for(int i=0; i < (sys->natoms -1); i += sys->nthreads) {
        int ii = i + tid;
        if (ii >= (sys->natoms -1)) break;
        rx1=sys->rx[ii];
        ry1=sys->ry[ii];
        rz1=sys->rz[ii];
    }
    Thread Id is like MPI rank
    sys->fx holds storage for one full fx array for each thread => race condition is eliminated.
```
MPI-like Approach with OpenMP (2)

- We need to write our own reduction:

```c
#if defined (_OPENMP)
#pragma omp barrier
#endif

i = 1 + (sys->nAtoms / sys->nThreads);
fromidx = tid * i;
toidx = fromidx + i;
if (toidx > sys->nAtoms) toidx = sys->nAtoms;

for (i=1; i < sys->nThreads; ++i) {
    int offs = i*sys->nAtoms;
    for (int j=fromidx; j < toidx; ++j) {
        sys->fx[j] += sys->fx[offs+j];
        sys->fy[j] += sys->fy[offs+j];
        sys->fz[j] += sys->fz[offs+j];
    }
}
```

- Need to make certain, all threads are done with computing forces
- Use threads to parallelize the reductions
More OpenMP Timings

- The `omp parallel` region timings
  2916: 1T: 103s, 2T: 53s, 4T: 19s, 8T: 10s
  => better speedup, but serial is faster for 108, at 2916 atoms we are often beyond cutoff

- This approach also works with cell lists
  => with 8 threads:
  4.1s = 6.8x speedup vs. serial cell list version (28s).
  That is 62x faster than the first naive serial version
Conclusions

- Make sure that you exploit the physics of your problem well => Newton's 3\textsuperscript{rd} law gives a 2x speedup for free (but interferes with threading!)
- Let the compiler help you (more readable code), but also make it easy to the compiler => unrolling, inlining can be offloaded
- Understand the properties of your hardware and adjust your code to match it
- Best strong scaling on current hardware with hybrid parallelization, e.g. MPI+OpenMP