A short tutorial on parallel molecular dynamics

Joerg Rottler

- Introduction
- Basic principles and ideas of molecular dynamics
- Numerical integration (velocity verlet)
- Parallelization strategy (domain decomposition) and implementation using MPI
- Parallel scalability for MD
- An example parallel MD program

Introduction

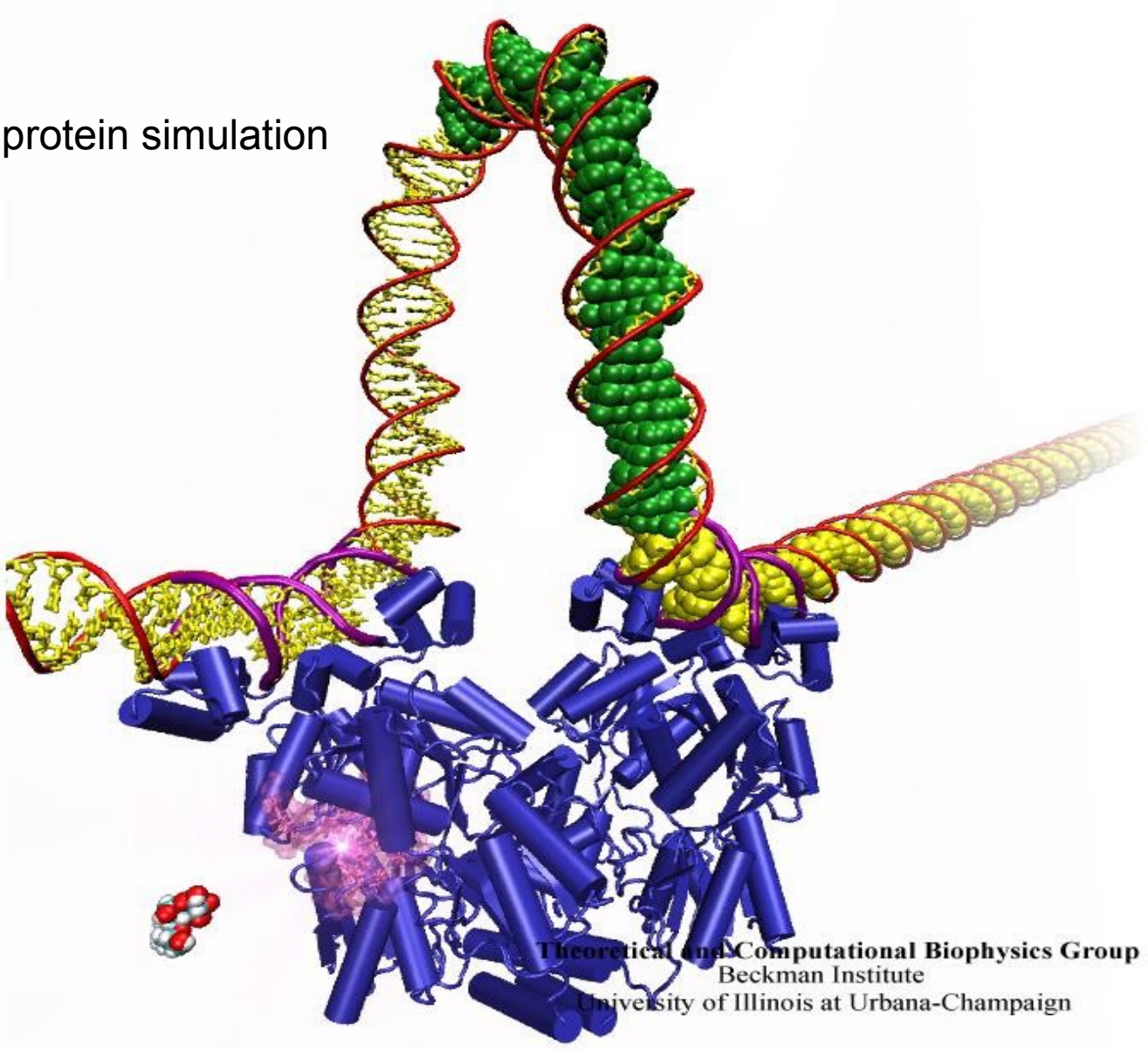
- Molecular dynamics (MD) is a very versatile research tool with crossdisciplinary applications in:

- Condensed matter physics
- Chemistry and chemical engineering
- Biophysics
- Materials science

- Basic idea: study the evolution of many particle systems based on classical or quantum mechanical equations of motion in or out of equilibrium
- Conceptually straightforward, but computationally very demanding
- Can study complex systems with few assumptions, but get relatively short trajectories (< microseconds).

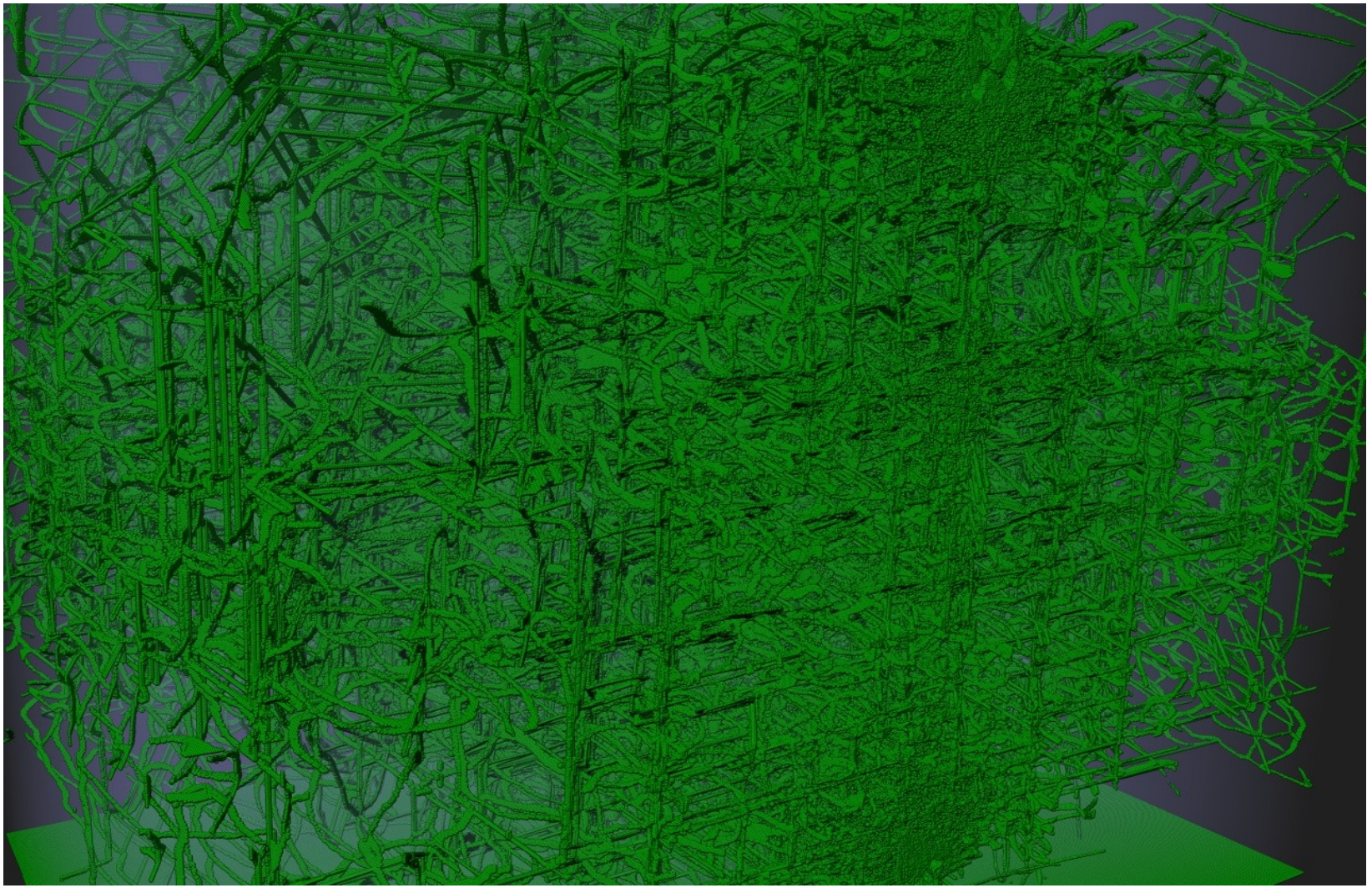
Application 1: Biophysics

314,000-atom protein simulation



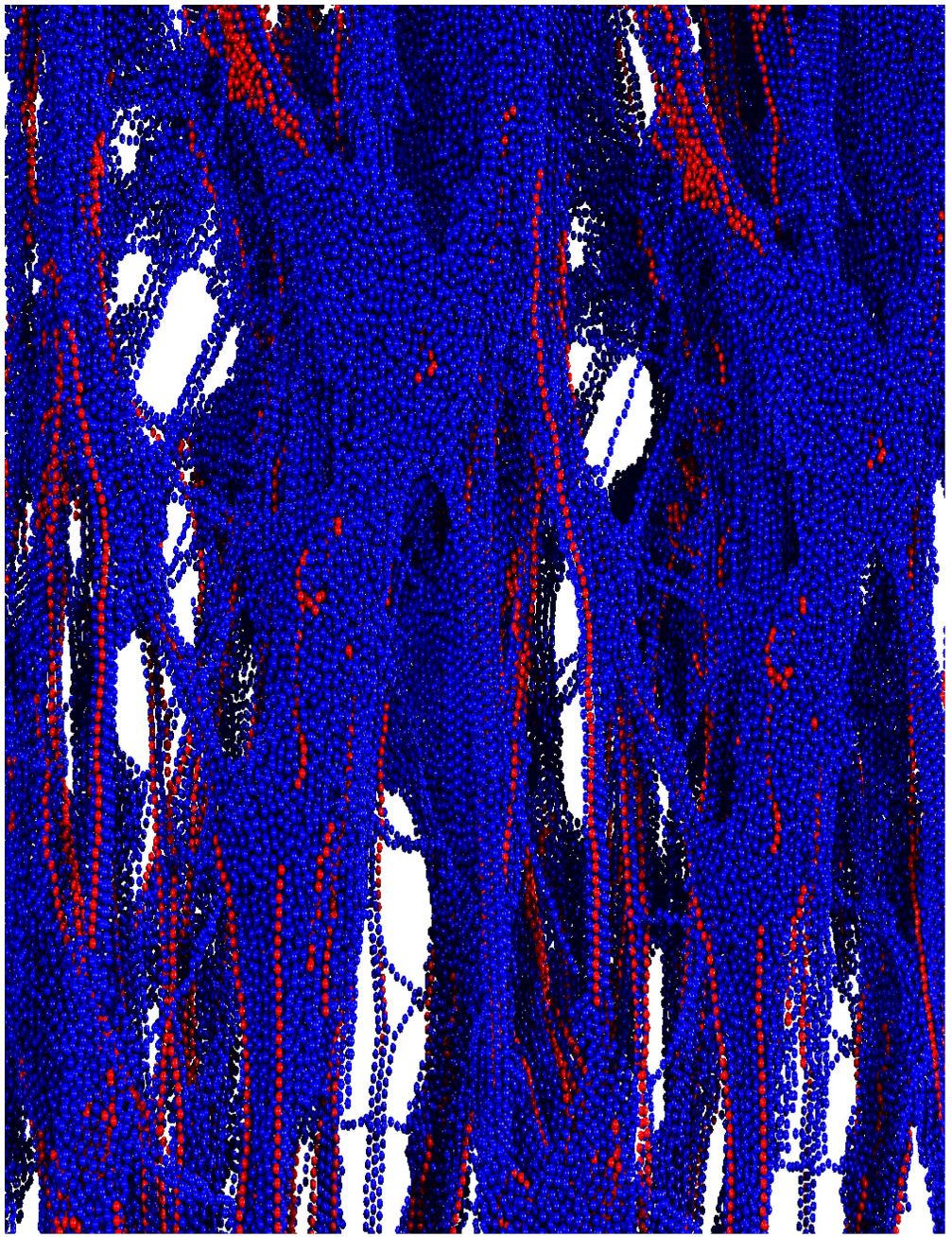
Application 2: deformation of metals

Dislocation dynamics with a billion copper atoms



http://www.llnl.gov/largevis/atoms/ductile-failure/

Application 3: polymer fracture



Deformation of glassy polymers into a dense network of fibrils near crack tip.

~ 250,000 atoms

MD basics

- Particle positions $\mathbf{r}_i$,

  velocities $\mathbf{v}_i = d\mathbf{r}_i/dt$

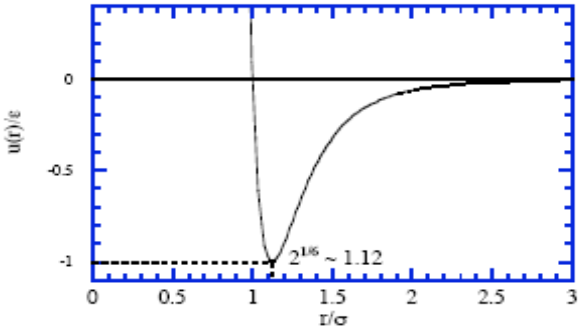
  accelerations $\mathbf{a}_i = d\mathbf{v}_i/dt$

- Equation of motion: Newton's 2nd law $\mathbf{F}_i = m\mathbf{a}_i$
- Initial value problem: given positions and velocities at time t, compute trajectories of interacting particles at later times.
- Forces $\mathbf{F}_i$ on particle i arise from interactions with other particles j
- An MD simulation consists of
  (a) compute all particle interactions efficiently
  (b) propagate the particles by numerical integration of the equation of motion
  (c) interate ...

Interaction potentials

- All the physics is in the force law
- Can come from classical potentials (as in examples) or quantum mechanics (ab-initio or Car-Parinello MD)
- Simple molecular potential: 6-12 Lennard Jones (noble gases)

$$V(r) = 4\epsilon\left[(\sigma/r)^{12} - (\sigma/r)^{6}\right]$$



Very popular, well behaved, models vanderWaals interactions
Short-ranged due to $1/r^6$ tail, can be truncated without artefacts

- Other interactions: chemical (covalent) bonds
  bending forces (for polymers)
  coulombic forces (if charges present)
- For metals, use "embedded atom" potentials, mimicks atom core in electron sea

Integrating the eqs. of motion (EOM)

- Want to predict r(t+Δt), v(t+Δt) from r(t), v(t)
- Idea: Taylor-expand the EOM

$$r(t+\Delta)=r(t)+v(t)\Delta+\Delta^2 F/2\mathrm{m}+\ldots$$

$$r(t-\Delta)=r(t)-v(t)\Delta+\Delta^2 F/2\mathrm{m}+\ldots$$

and add: $r(t+\Delta)=2\mathrm{r}(t)-r(t-\Delta)+\Delta^2 F/m+O(\Delta^4)$

- Verlet scheme: given r(t-Δt) and r(t), compute forces F(t), predict r(t+Δ) and iterate
  also compute velocities: $v(t)=(r(t+\Delta)-r(t-\Delta))/2\Delta+O(\Delta^2)$

- Problem: cannot calculate v(t) until r(t+Δt) is known

Velocity Verlet

- Alternative but equivalent integration scheme

$$v(t+\Delta/2) = v(t) + \Delta(F(t))/2\mathrm{m}$$
$$r(t+\Delta) = r(t) + v(t)\Delta + \Delta^2 F(t)/2\mathrm{m} = r(t) + v(t+\Delta/2)\Delta$$
$$v(t+\Delta) = v(t) + \Delta(F(t) + F(t+\Delta))/2\mathrm{m}$$

  This is called "velocity-verlet" or leapfrog scheme

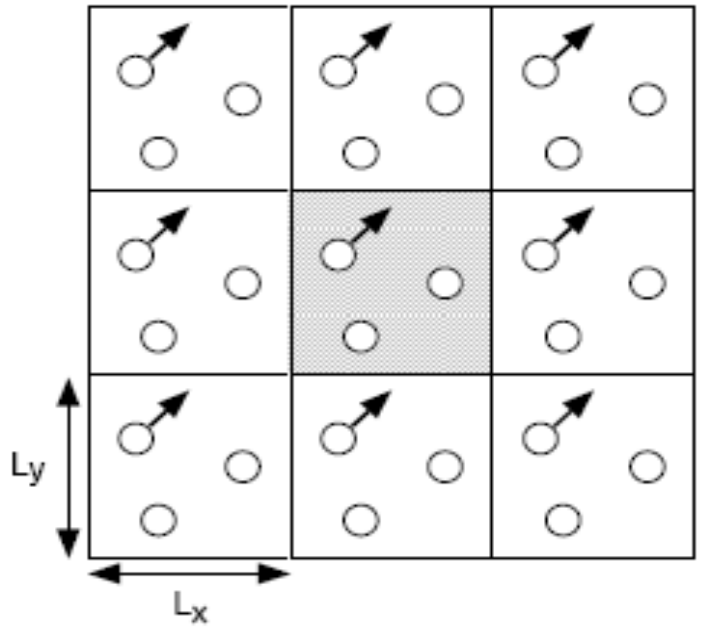
- First advance velocities to midpoint
  Then advance positions by full timestep
  (knowing the midstep velocities)
  Finally complete velocity move
  "velocities leap over positions"
- Can prove that fully equivalent to Verlet scheme
- Currently most popular algorithm in large scale MD packages

Computational expenses

- Integrator is relatively cheap, simple O(N) loop
- Force computation: fairly easy if forces are short ranged as in Lennard Jones. For O(N) algorithm, need to construct "neighbor list" that contains only particles within "cutoff radius" of a given particle, otherwise force loop becomes too expensive
- Several well established methods available (linked lists)
- In dense systems, neighbor list construction can be as expensive as force computation
- If forces are long-ranged (eg. Coulombic, Gravitational), neighbor lists don't help. Need other tricks and a separate lecture for those.

Periodic boundary conditions

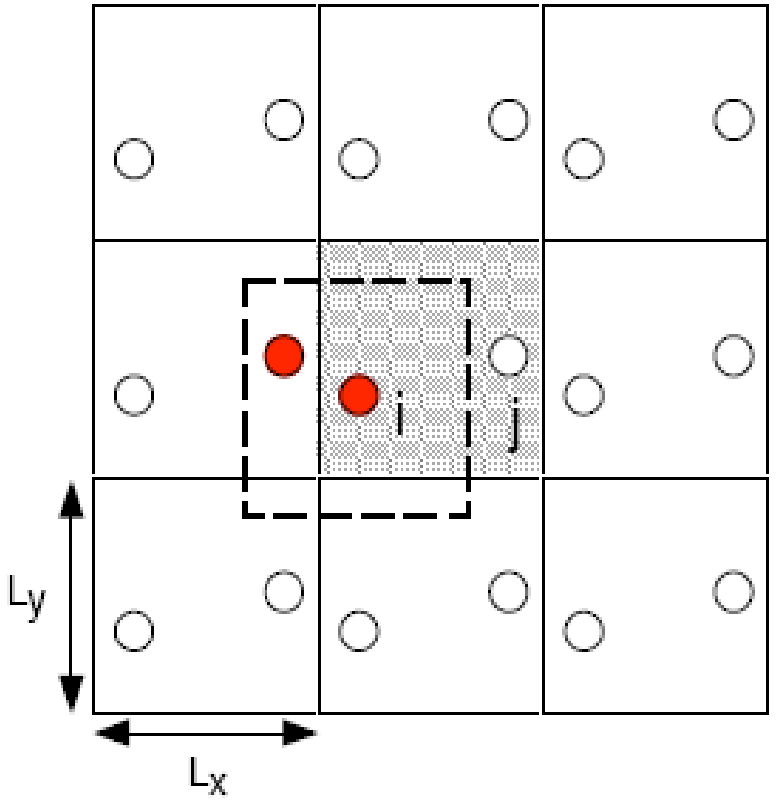
- Eliminate surface and boundary effects and simulate bulk solid



- Atoms leaving the simulation box on one side reenter on the other so that the box “looks” infinite
- All coordinates between 0 and L

Minimum image convention

- Make simulation box large enough so that each atom interacts only with the nearest image of another atom



Energy, temperature, etc

$$E = E_{kin} + E_{pot} = \sum_i m v_i^2 / 2 + \sum_{i<j} V(r_i)$$

- Energy is conserved up to the accuracy of the integrator
- Typical timestep for Lennard Jones systems Δ=0.005

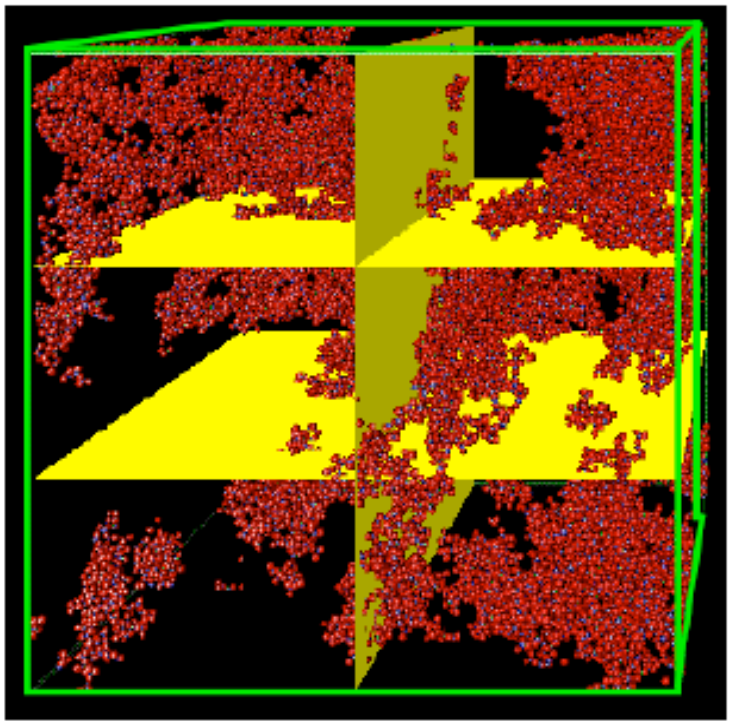
- Temperature T is computed from equipartition:

$$3/2\, N k_B T = 1/2 \sum_i m_i v_i^2$$

- Typical MD run:
  choose initial conditions, set external conditions (T, P,V),
  iterate
  monitor key parameters (T,E,P,V)
  periodically write out quantities of interest and analyse

Parallelization

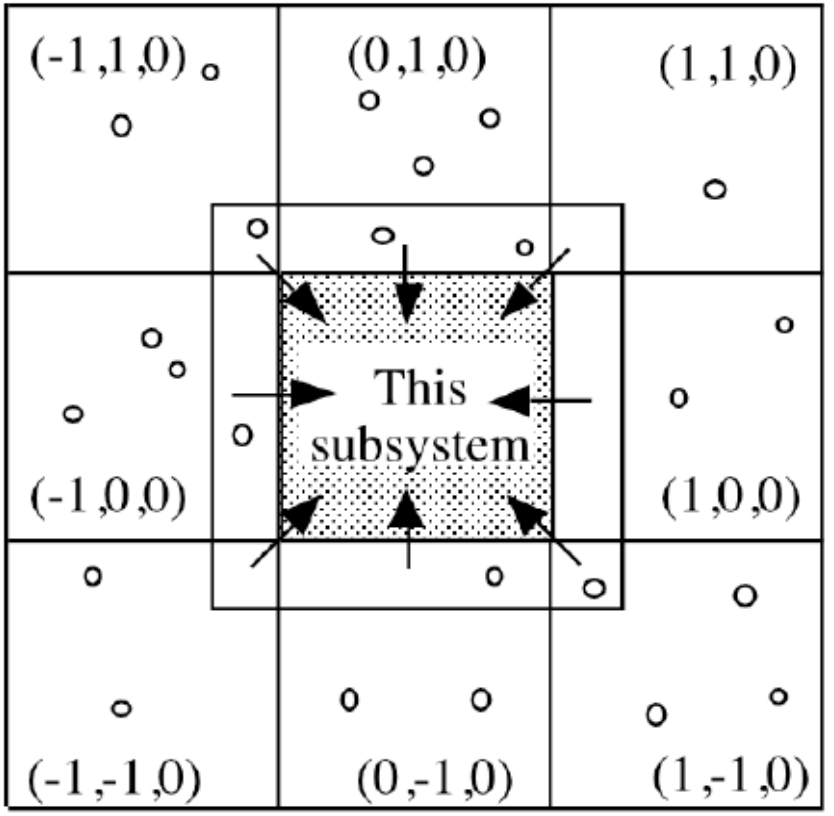
- How can we speed up the calculation by using many CPUs in parallel?
- Answer: partition the big simulation cell into many smaller ones and give each CPU a small piece to work on simultaneously.
- This is called **domain decomposition**



- Each CPU deals only with its local subregion
- Must exchange information at boudaries via MPI
- Works well for short range interactions that are local

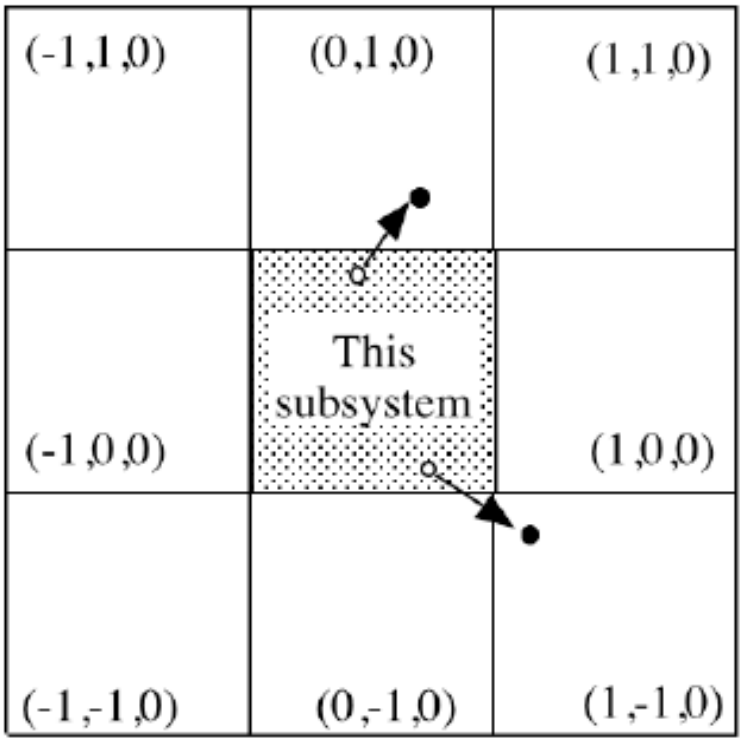
What do we need?

- Each subsystem must know its neighboring domains. For a cubic lattice each box has 26 neighboring boxes
- Atom caching: each domain must not only know about its own atoms, but also about all atoms within interaction range from the boundary to compute the forces



What do we need?

- Atom migration: once atoms cross the boundary of the subsystem, they must be removed from it and placed into the neighboring subsystem.



- This step does not arise in lattice problems; there one only needs the caching of a few grid points beyond the boundary

Flow of a parallel MD program

- Update velocities to $v(t+\Delta t/2)$
- Update coordinates to $r(t+\Delta t)$
- Migrate moved-out atoms to the neighbor processors
- Copy the surface atoms within some distance from the neighbors
- Compute new forces $F(t+\Delta t)$ and accelerations, including the cached atoms
- Update velocites to $v(t+\Delta t)$

--> We need two subroutines that handle migration and atom copying, and MPI to for the communication

General structure of main.c

```
int main(int argc, char **argv) {

  MPI_Init(&argc,&argv); /* Initialize the MPI environment */
  MPI_Comm_rank(MPI_COMM_WORLD, &sid);  /* My processor ID */

  init_params(); /* simulation parameters */
  set_topology(); /* domain decomposition/processor grid */
  init_conf();   /* setup simulation */
  atom_copy(); /* first communication of boundary atoms */
  compute_accel(); /* Computes initial forces/accelerations */

  for (stepCount=1; stepCount<=StepLimit; stepCount++) single_step();

  MPI_Finalize(); /* Clean up the MPI environment */
}
```

integrate.c

```
 void single_step() {
/*--------------------------------------------------------------
r & rv are propagated by DeltaT using the velocity-Verlet scheme.
--------------------------------------------------------------*/
  int i,a;

  half_kick(); /* First half kick to obtain v(t+Dt/2) */
  for (i=0; i<n; i++) /* Update atomic coordinates to r(t+Dt) */
    for (a=0; a<3; a++) r[i][a] = r[i][a] + DeltaT*rv[i][a];
  atom_move();
  atom_copy();
  compute_accel(); /* Computes new accelerations, a(t+Dt) */
  half_kick(); /* Second half kick to obtain v(t+Dt) */
}
```

Domain decomposition

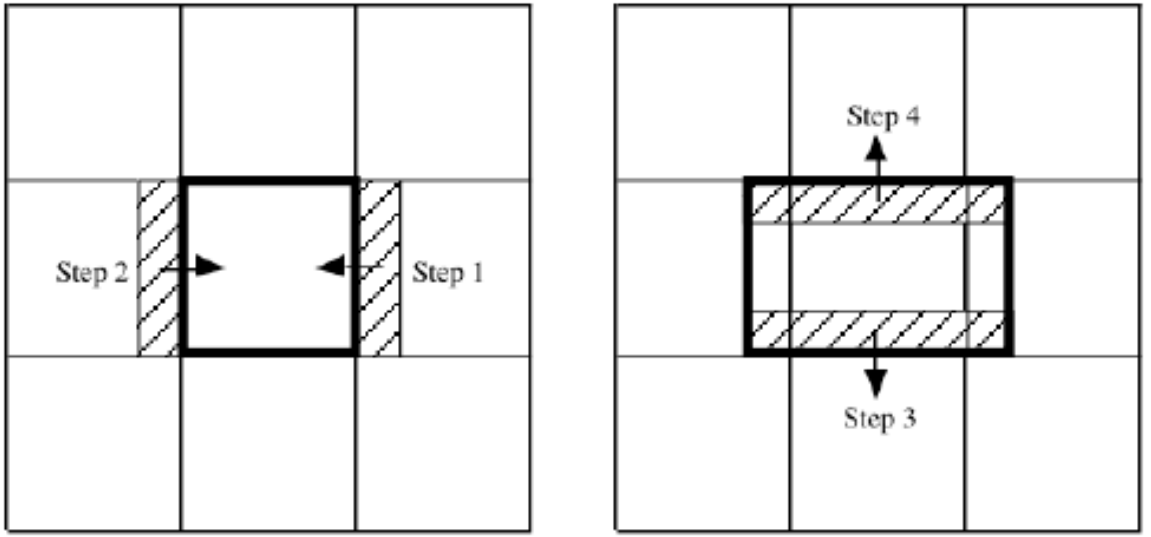
- 3D Mesh, processors P_x, P_y, P_z, total # of procs $P=P_x P_y P_z$
- Each processor has unique ID: $p=p_x \times P_y P_z + p_y \times P_z + p_z$
- Vector ID $\mathbf{p}=(p_x, p_y, p_z)$
- Each face-sharing neighbor can be reached via **δ:**

Neighbor ID, κ	$\vec{\delta}=(\delta_x, \delta_y, \delta_z)$	$\vec{\Delta}=(\Delta_x, \Delta_y, \Delta_z)$
0 (east)	(-1, 0, 0)	$(-L_x, 0, 0)$
1 (west)	(1, 0, 0)	$(L_x, 0, 0)$
2 (north)	(0, -1, 0)	$(0, -L_y, 0)$
3 (south)	(0, 1, 0)	$(0, L_y, 0)$
4 (up)	(0, 0, -1)	$(0, 0, -L_z)$
5 (down)	(0, 0, 1)	$(0, 0, L_z)$

- Processor ID can be obtained by MPI_Comm_Rank()

Atom caching

- n: # of local atoms, nb: # of copied surface atoms
- r[0:n-1]: coords of local atoms, r[n:n+nb-1]: coords of cached atoms
- Need function to determine if atom is near boundary
- Coords of boundary atoms are then sent to 6 face sharing neighbors, copies to non-face sharing neighbors are forwarded



Algorithm

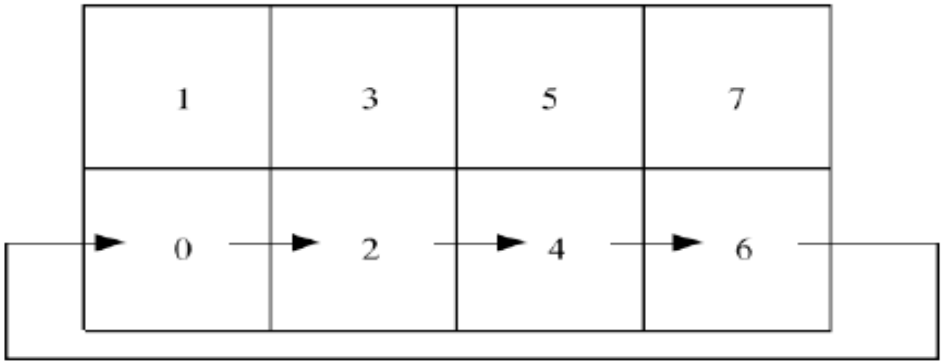
- Reset # of received cache atoms nbnew=0
- Loop over x,y,z directions
- Make boundary atom lists for lower and upper directions
- Loop over lower and upper directions
- Send/Receive # of boundary atoms to/from neighbor
- Send/Receive boundary atoms to/from neighbor
- Increment nbnew
- End for
- End for

<u>Three phase message passing:</u>

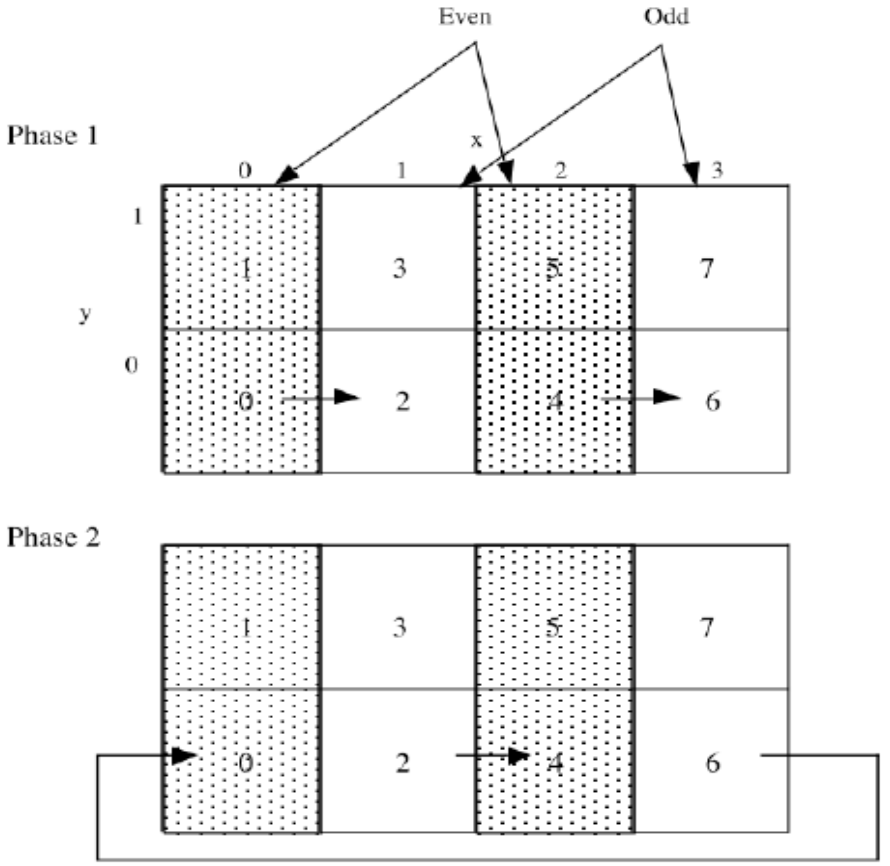
- Message buffering: coordinates --> dbuf
- Message passing: dbuf --> dbufr (send dbuf, receive dbufr)
- Messsage storing: coordinates <-- dbufr (append after local)

Deadlock avoidance

- Cannot send in circular fashion: sender blocks until receiver clears its buffer, but cannot receive until send is complete:



- Classify procs into EVEN/ODD and only have those send/receive pairs:



Communication algorithm

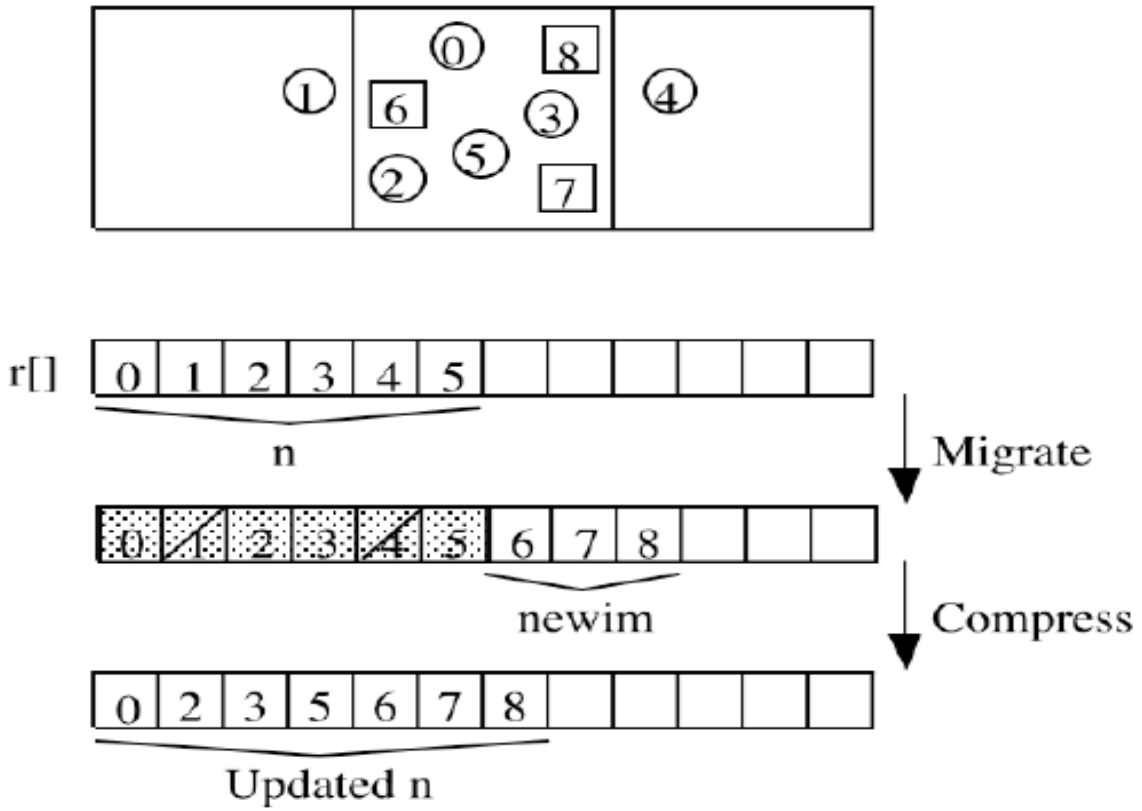
```
/* Message buffering */
   for (i=1; i<=nsd; i++)
     for (a=0; a<3; a++) /* Shift the coordinate origin */
       dbuf[3*(i-1)+a] = r[lsb[ku][i]][a]-sv[ku][a];

   /* Even node: send & recv */
   if (myparity[kd] == 0) {
     MPI_Send(dbuf,3*nsd,MPI_DOUBLE,inode,20,MPI_COMM_WORLD);
     MPI_Recv(dbufr,3*nrc,MPI_DOUBLE,MPI_ANY_SOURCE,20,
           MPI_COMM_WORLD,&status);
   }

   /* Odd node: recv & send */
   else if (myparity[kd] == 1) {
     MPI_Recv(dbufr,3*nrc,MPI_DOUBLE,MPI_ANY_SOURCE,20,
           MPI_COMM_WORLD,&status);
     MPI_Send(dbuf,3*nsd,MPI_DOUBLE,inode,20,MPI_COMM_WORLD);
   }
```

Atom migration

- Similar to atom cache: 6 step loop over face-sharing neighbors
- Need function to identify migrating atoms
- Variable newim keeps track of new atoms in cell
- New atoms are appended to r[i], v[i]; moved-out atoms are deleted and array is compressed at end of loop



Algorithm

- Newim=0
- Loop over x,y,z
- Make moving atom lists for lower and upper directions
- Loop over lower and upper directions
- Send/receive # of moving atoms
- Send/receive moving atom coords and velocities
- Mark moved out atoms
- End for
- End for
- Compress coordinate and velocity arrays to eliminate moved out atoms

Scalability metrics

- Problem size W, T(W,P) = execution time on P procs
- Speed S=W/T(W,P)
- Speedup S_p=S(W,P)/S(W,1)
- Parallel efficiency: E_p=S_p/P

Constant problem size speedup:

- S_p=S(W,P)/S(W,1)=T(W,1)/T(W,P)
- E_p=S_p/P=T(W,1)/(P T(W,P))="ideal time/actual time"
- Amdahl's law: fraction f is sequential, cannot be parallelized:

  S_p=T(W,1)/T(W,P)= 1/(f+(1-f)P) --> 1/f

Isogranular speedup: keep w=W/P const. (work per proc)

- S_p=S(P w,P)/S(w,1)= P T(w,1)/T(P w,P)
- E_p=T(w,1)/T(P w,P)

Efficiency of parallel MD

- T_{comp}=a N/P T_{comm}=b "area"= b $(N/P)^{2/3}$ T_{global}=c log P
- T_{total}= a N/P + b $(N/P)^{2/3}$ + c log P
- Speedup S_p=T(N,1)/T(N,P)= aN/(a N/P + b $(N/P)^{2/3}$ + c log P)

- Efficiency: $$E_p = \frac{S_p}{P} = \frac{1}{1 + \frac{b}{a}\left(\frac{P}{N}\right)^{1/3} + \frac{c}{a} P \log \frac{P}{N}}$$

  decreases with increasing P
- Isogranular speedup: granularity n=N/P

$$E_p = \frac{T(n,1)}{T(nP,P)} = \frac{1}{1 + \frac{b}{a} n^{-1/3} + \frac{c}{an} \log P}$$

larger for larger n, weakly decreasing with P due to log P