Parallel Molecular Dynamics

Aiichiro Nakano

Collaboratory for Advanced Computing & Simulations
Department of Computer Science
Department of Physics & Astronomy
Department of Chemical Engineering & Materials Science
Department of Biological Sciences
University of Southern California
Email: anakano@usc.edu

Objective: Operationally understand spatial decomposition (who does what) & message passing using a real-world application (pmd.c)

http://cacs.usc.edu/education/cs596/src/pmd
https://github.com/KenichiNomura/binary-LJ-pmd
Parallel Molecular Dynamics

Spatial decomposition (short-ranged):

1. Divide the physical space into subspaces of equal volume
2. Assign each subspace to a computing node (more generally, to a process) in a parallel computer
3. Each node computes forces on the atoms in its subspace & updates their positions & velocities

Who does what

Will learn other decomposition schemes later:
http://cacs.usc.edu/education/cs596/NT.pdf

or MPI rank
Spatial Decomposition

- **Process ID**
  - **Vector**
    - \( p_x = p / (P_y P_z) \)
    - \( p_y = (p / P_z) \mod P_y \)
    - \( p_z = p \mod P_z \)
  - **Scalar**
    - \( p = p_x P_y P_z + p_y P_z + p_z \)

Which 3D subspace?

\[
\text{In pmd.h}
\]
\[
\text{int vproc[3] = \{1,1,2\}, nproc = 2;}
\]

\[
\text{In pmd.c}
\]
\[
\text{MPI_Comm_rank(MPI_COMM_WORLD, \&sid);} \\
\text{vid[0] = sid / (vproc[1] * vproc[2]);} \\
\text{vid[1] = (sid / vproc[2]) \% vproc[1];} \\
\text{vid[2] = sid \% vproc[2];}
\]
Neighbor Process ID

\[ p'_{\alpha}(\kappa) = [p_{\alpha} + \delta_{\alpha}(\kappa) + P_{\alpha}] \mod P_{\alpha} \quad (\kappa = 0, \ldots, 5; \quad \alpha = x, y, z) \]

\[ p'(\kappa) = p'_{x}(\kappa) \times P_{y}P_{z} + p'_{y}(\kappa) \times P_{z} + p'_{z}(\kappa) \]

<table>
<thead>
<tr>
<th>Neighbor ID, ( \kappa )</th>
<th>( \tilde{\delta} = (\delta_{x}, \delta_{y}, \delta_{z}) )</th>
<th>( \tilde{\Delta} = (\Delta_{x}, \Delta_{y}, \Delta_{z}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (east)</td>
<td>(-1, 0, 0)</td>
<td>(-L_{x}, 0, 0)</td>
</tr>
<tr>
<td>1 (west)</td>
<td>(1, 0, 0)</td>
<td>(L_{x}, 0, 0)</td>
</tr>
<tr>
<td>2 (north)</td>
<td>(0, -1, 0)</td>
<td>(0, -L_{y}, 0)</td>
</tr>
<tr>
<td>3 (south)</td>
<td>(0, 1, 0)</td>
<td>(0, L_{y}, 0)</td>
</tr>
<tr>
<td>4 (up)</td>
<td>(0, 0, -1)</td>
<td>(0, 0, -L_{z})</td>
</tr>
<tr>
<td>5 (down)</td>
<td>(0, 0, 1)</td>
<td>(0, 0, L_{z})</td>
</tr>
</tbody>
</table>

- \( L_{x}, L_{y} \) & \( L_{z} \) are the box lengths *per process* in the \( x, y \) & \( z \) directions
- Atom coordinates are in the range \([0, L_{\alpha}] \) (\( \alpha = x, y, z \)) in each process

In `pmd.c`

```c
int iv[6][3]={{-1,0,0}, {1,0,0}, {0,-1,0}, {0,1,0}, {0,0,-1}, {0,0,1}};
...
for (ku=0; ku<6; ku++) {
    for (a=0; a<3; a++)
        k1[a] = (vid[a]+iv[ku][a]+vproc[a])%vproc[a];
    for (a=0; a<3; a++) sv[ku][a] = al[a]*iv[ku][a];
} wrap around```


Neighbor Process ID Example

Periodic boundary condition
1. First half kick to obtain $v_i(t+Dt/2)$
2. Update atomic coordinates to obtain $r_i(t+Dt)$
3. `atom_move()`: Migrate the moved-out atoms to the neighbor processes
4. `atom_copy()`: Copy the surface atoms within distance $r_c$ from the neighbors
5. `compute_accel()`: Compute new accelerations, $a_i(t+Dt)$, including the contributions from the cached atoms
6. Second half kick to obtain $v_i(t+Dt)$
Search for pairs only within the nearest neighbor cells:  
\[ O(N^2) \rightarrow O(N) \]

- **Cell size**
  \[ L_{c\alpha} = \left\lfloor \frac{L_\alpha}{r_c} \right\rfloor \]
  \[ r_{c\alpha} = \frac{L_\alpha}{L_{c\alpha}} (\alpha = x, y, z) \]

- **Cell index**
  \[ c = c_x (L_{cy}+2)(L_{cz}+2) + c_y (L_{cz}+2) + c_z \]
  \[ c_x = \left\lfloor \frac{c}{(L_{cy}+2)(L_{cz}+2)} \right\rfloor \]
  \[ c_y = \left\lfloor \frac{c}{(L_{cz}+2)} \right\rfloor \mod (L_{cy}+2) \]
  \[ c_z = c \mod (L_{cz}+2) \]

- **Atom → cell mapping**
  \[ c_\alpha = \left\lfloor \frac{(r_\alpha+r_{c\alpha})}{r_{c\alpha}} \right\rfloor (\alpha = x, y, z) \]

Only change from serial lmd.c in green:  
Augmented cells to include cached atoms
/* Reset the headers, head */
for (c=0; c<lcyz2; c++) head[c] = EMPTY;
/* Scan atoms to construct headers, head, & linked lists, lscl */
for (i=0; i<n+nb; i++) {
  Consider \( n_b \) cached atoms
  /* Vector cell index to which this atom belongs */
  for (a=0; a<3; a++) mc[a] = (r[i][a]+rc[a])/rc[a];
  /* Translate the vector cell index, mc, to a scalar cell index */
  c = mc[0]*lcyz2+mc[1]*lc2[2]+mc[2];
  /* Link to the previous occupant (or EMPTY if you're the 1st) */
  lscl[i] = head[c];
  /* The last one goes to the header */
  head[c] = i;
}

In the above:
1. \( lcyz2 = lc2[1]*lc2[2] \)
2. where \( lc2[a] = lc[a]+2 \) (a = 0,1,2)
3. \( lcyz2 = lcyz2*lc2[0] \)

Change from serial lmd.c in green
Interaction Computation

/* Scan inner cells (resident) */
for (mc[0]=1; mc[0]<=lc[0]; (mc[0])++)
for (mc[1]=1; mc[1]<=lc[1]; (mc[1])++)
  /* Calculate a scalar cell index */
  c = mc[0]*lcyz2+mc[1]*lc2[2]+mc[2];
  /* Scan the neighbor cells (including itself) of cell c (resident + cached) */
  for (mc[0]=mc[0]-1; mc[0]<=mc[0]+1; (mc[0])++)
        /* Calculate the scalar cell index of the neighbor cell */
        c1 = mc[0]*lcyz2+mc[1]*lc2[2]+mc[2];
        /* Scan atom i in cell c */
        i = head[c];
        while (i != EMPTY) {
          /* Scan atom j in cell c1 */
          j = head[c];
          while (j != EMPTY) {
            ...
            if (i<j && r_{ij}<r_c^2) Process pair (i, j)
            ...
            j = lslc1[j];
          }
          i = lslc1[i];
        }
      }
}

Change from serial lmd.c in green

Who does what: Each rank computes forces on the resident atoms in its subspace & updates their positions & velocities

Resident atoms may interact with cached atoms
Parallel Interaction Computation

**SPMD:** Who does what?

Each process computes:
1. The forces on its resident atoms
2. The potential energy between resident pairs & 1/2 of that between resident-cached pairs

```c
for resident cells, c {
    for neighbor (resident or cached) cells, c1 {
        scan atom i in cell c using c’s linked list {
            scan atom j in cell c1 using c1’s linked list {
                ...
                if (i<j && r_ij<r_c^2) {
                    compute pair force a_ij & potential u(r_ij)
                    bintra = j < n; /* j is resident? */
                    a_i += a_ij; if (bintra) a_j -= a_ij;
                    if (bintra) lpe += u(r_ij); else lpe += u(r_ij)/2;
                }
            }
        }
    }
    MPI_Allreduce(&lpe, &potEnergy,...,MPI_SUM,...);
}
```
Atom Caching: \texttt{atom\_copy()}

Caching from 26 neighbors in 6 steps

Reset the number of received cache atoms, \(\text{nbnew} = 0\)

for \(x, y,\) and \(z\) directions

Make boundary-atom lists, \(\text{lsb},\) for lower and higher directions \textit{including both resident, n, and cache, nbnew, atoms} (within \(r_c\) from boundary)

for lower and higher directions

Send/receive boundary-atom coordinates to/from the neighbor

Increment \(\text{nbnew};\)

endfor

def
endfor

\(\text{nb} = \text{nbnew}\)
Implementing Atom Caching

### Copying condition

```cpp
bbd(ri[], ku) {
    kd = ku / 2 (= 0|1|2) x|y|z
    kdd = ku % 2 (= 0|1) lower|higher
    if (kdd == 0)
        return ri[kd] < RCUT
    else
        return al[kd] – RCUT < ri[kd]
}
```

#### 3 phases of message passing

1. Message buffering: `dbuf ← r–sv (shift), gather`
2. Message passing: `dbufr ← dbuf`
   - Send `dbuf`
   - Receive `dbufr`
3. Message storing: `r ← dbufr`, append after the residents

---

Self-centric coordinate systems

- La Canada
- Pasadena
- Arcadia
Deadlock Avoidance

Cyclic dependence

<table>
<thead>
<tr>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

vproc[0|1|2] must be 1 or even

3-phase (deadlock-free) message passing

1. Message buffering: dbuf ← r, gather
2. Message passing: dbufr ← dbuf
   /* Even node: send & recv, if not empty */
   if (myparity[kd] == 0) {
      MPI_Send(dbuf,3*nsd,MPI_DOUBLE,inode,120,MPI_COMM_WORLD);
      MPI_Recv(dbufr,3*nrc,MPI_DOUBLE,MPI_ANY_SOURCE,120,
                MPI_COMM_WORLD,&status);
   }
   /* Odd node: recv & send, if not empty */
   else if (myparity[kd] == 1) {
      MPI_Recv(dbufr,3*nrc,MPI_DOUBLE,MPI_ANY_SOURCE,120,
                MPI_COMM_WORLD,&status);
      MPI_Send(dbuf,3*nsd,MPI_DOUBLE,inode,120,MPI_COMM_WORLD);
   }
   /* Single layer: Exchange information with myself */
   else
      for (i=0; i<3*nrc; i++) dbufr[i] = dbuf[i];
3. Message storing: r ← dbufr, append

Cyclic dependence

vproc[0|1|2] must be 1 or even
11. Q: My parallel program runs on other parallel machines but seems to deadlock on the SP-1 when using EUI, EUI-H, or Chameleon.

A: The following parallel program can deadlock on any system when the size of the message being sent is large enough:

```c
    send( to=partner, data, len, tag )
    recv( from=partner, data, maxlen, tag )
```

where these are blocking send’s and receives (mp_bsend in EUI/EUI-H and PIBsend in Chameleon). For many systems, deadlock does not occur until the message is very long (often 128 KBytes or more). For EUI, the size is (roughly) 128 bytes (not KBytes) and for EUI-H, the size if (again roughly) 4 KBytes. The limit for Chameleon is the same as the underlying transport layer (i.e., the EUI or EUI-H limits).

To fix this you have several choices:

**pmd.c**
- Reorder your send and receive calls so that they are pair up. For example, if there are always an even number of processors, you could use

```c
    if (myid is even) {
        send( to=partner, data, len, tag )
        recv( from=partner, data, maxlen, tag )
    }
    else {
        recv( from=partner, data, maxlen, tag )
        send( to=partner, data, len, tag )
    }
```

**assignment**
- Use non-blocking sends and receives instead
Digress: Polyacetylene & Peierls Distortion

Fig. 1. Electron dispersion and a band pattern of one-dimensional molecular system: a). metallic and b). insulator state, ($\rho(z)$—electronic density, $a$—lattice period).

Nature’s spontaneous even-odd symmetry breaking
Atom Migration: atom_move()

Reset the number of received new immigrants, newim = 0
for x, y, and z directions
Make moving-atom lists, mvque, for lower and higher directions including both resident, n, and immigrant, newim, atoms but excluding those already moved out for lower and higher directions
Send/receive moving-atom coordinates to/from the neighbor
(When moving, r[][0] ← MOVED_OUT = -10^{10})
Increment newim
endfor
endfor
Compress the r array to eliminate the moved-out atoms
Implementing Atom Migration

Moving condition

\[ \text{bmv(ri[], ku)} \{ \]
\[ \quad \text{kd} = \text{ku} / 2 \ (= 0|1|2) \]
\[ \quad \text{kdd} = \text{ku} \mod 2 \ (= 0|1) \]
\[ \quad \text{if} \ (\text{kdd} == 0) \]
\[ \quad \quad \text{return} \ \text{ri[kd]} < 0.0 \]
\[ \quad \text{else} \]
\[ \quad \quad \text{return} \ \text{al[kd]} < \text{ri[kd]} \]
\[ \} \]

3 phases of message passing

1. Message buffering: \( \text{dbuf} \leftarrow r - \text{sv} \) (shift) & \( \text{rv} \), gather
   - Mark \text{MOVED_OUT} in \( r \)
2. Message passing: \( \text{dbufr} \leftarrow \text{dbuf} \)
   - Send \text{dbuf}
   - Receive \text{dbufr}
3. Message storing: \( r \) & \( \text{rv} \leftarrow \text{dbufr} \), append after the residents
Parallel computing:
Specifies who does what — decomposition

Parallel molecular dynamics (spatial decomposition):
Who does what = each processor computes forces on only resident atoms in the subspace assigned to it & update their positions & velocities
Scalability Metrics for Parallel Molecular Dynamics

Aiichiro Nakano

Collaboratory for Advanced Computing & Simulations
Department of Computer Science
Department of Physics & Astronomy
Department of Chemical Engineering & Materials Science
Department of Biological Sciences
University of Southern California

Email: anakano@usc.edu

Objective: Consolidate your understanding of scalability analysis (e.g., fixed-problem vs. isogranular scaling) using a real-world example of pmd.c
Recap: Parallel Efficiency

Parallel computing = solving a big problem ($W$) in a short time ($T$) using many processors ($P$)

- **Execution time:** $T(W,P)$
  - $W$: Workload
  - $P$: Number of processors

- **Speed:**
  
  $$S(W,P) = \frac{W}{T(W,P)}$$

- **Speedup:**
  
  $$S_P = \frac{S(W_P,P)}{S(W_1,1)} = \frac{W_P T(W_1,1)}{W_1 T(W_P,P)}$$

- **Efficiency:**
  
  $$E_P = \frac{S_P}{P} = \frac{W_P T(W_1,1)}{PW_1 T(W_P,P)}$$

How to scale $W_P$ with $P$?
Fixed Problem-Size (Strong) Scaling

Solve the same problem faster

\( W_P = W \) — constant (strong scaling)

- **Speedup:**
  \[
  S_P = \frac{W_P T(W_1,1)}{W_1 T(W_P,P)} = \frac{T(W,1)}{T(W,P)}
  \]

- **Efficiency:**
  \[
  E_P = \frac{T(W,1)}{PT(W,P)}
  \]

- **Amdahl’s law:** \( f (= \text{sequential fraction of the workload}) \) limits the asymptotic speedup

\[
T(W,P) = fT(W,1) + \frac{(1-f)T(W,1)}{P}
\]

\[
S_P = \frac{T(W,1)}{T(W,P)} = \frac{1}{f + (1-f)/P}
\]

\[
\therefore S_P \rightarrow \frac{1}{f} \quad (P \rightarrow \infty)
\]
Isogranular (Weak) Scaling

Solve a larger problem within the same time duration

\[ W_P = Pw \text{ (weak scaling)} \]
\[ w = \text{constant workload per processor (granularity)} \]

- **Speedup:**
  \[ S_P = \frac{S(P \cdot w, P)}{S(w, 1)} = \frac{P \cdot w / T(P \cdot w, P)}{w / T(w, 1)} = \frac{P \cdot T(w, 1)}{T(P \cdot w, P)} \]

- **Efficiency:**
  \[ E_P = \frac{S_P}{P} = \frac{T(w, 1)}{T(P \cdot w, P)} \]
Analysis of Parallel MD

- Parallel execution time:

  Workload $\propto$ Number of atoms, $N$ (linked-list cell algorithm)

  $$T(N, P) = T_{\text{comp}}(N, P) + T_{\text{comm}}(N, P) + T_{\text{global}}(P)$$

  $$= a \frac{N}{P} + b \left( \frac{N}{P} \right)^{2/3} + c \log P$$

  \[ \therefore \frac{N}{L^3 \rho} \Rightarrow L^2 = \frac{N^{2/3}}{\rho^{2/3}} \]

  \[ = 6r_c \frac{N^{2/3}}{P^{2/3}} \rho \]

  \[ = 6r_c \rho^{1/3} \left( \frac{N}{P} \right)^{2/3} \]
Fixed Problem-Size Scaling

- **Speedup:**
  \[
  S_P = \frac{T(N,1)}{T(N,P)} = \frac{aN}{aN/P + b(N/P)^{2/3} + c \log P} \\
  = \frac{b(P)^{1/3}}{a(N)} + \frac{c}{a} \frac{P \log P}{N}
  \]

- **Efficiency:**
  \[
  E_P = \frac{S_P}{P} = \frac{1}{1 + \frac{b(P)^{1/3}}{a(N)} + \frac{c}{a} \frac{P \log P}{N}}
  \]

pmd.c: \( N = 16,384 \), on HPC (predecessor of CARC)
Isogranular Scaling of Parallel MD

- $n = N/P = \text{constant}$
- Efficiency:

$$E_P = \frac{T(n,1)}{T(nP,P)} = \frac{an}{an + bn^{2/3} + c \log P} = \frac{1}{1 + \frac{b}{a} n^{-1/3} + \frac{c}{an} \log P}$$

pmd.c: $N/P = 16,384$, on HPC (predecessor of CARC)
High-End Parallel MD

- 4.9 trillion-atom space-time multiresolution MD (MRMD) of SiO$_2$
- 8.5 billion-atom fast reactive force-field (F-ReaxFF) RMD of RDX
- 39.8 trillion grid points (50.3 million-atom) DC-DFT QMD of SiC

parallel efficiency 0.984 on 786,432 Blue Gene/Q cores

MD (molecular dynamics): MRMD
RMD (reactive molecular dynamics): F-ReaxFF
QMD (quantum molecular dynamics): DC-DFT
Portable Parallel Efficiency

- Weak-scaling parallel efficiency of 0.989 for a new generation of reactive molecular dynamics (RMD) on 131,072 Intel Knights Landing cores

Quantum MD@Scale

Quantum dynamics at scale: ultrafast control of emergent functional materials


Best Paper in ACM HPC Asia 2020

19 years since Scalable atomistic simulation algorithms for materials research, A. Nakano et al., Best Paper, IEEE/ACM Supercomputing 2001, SC01
Neural MD@Scale

- Neural-network molecular dynamics (NNMD) could revolutionize atomistic modeling of materials, providing quantum-mechanical accuracy at a fraction of computational cost.

P. Rajak et al., Neural network molecular dynamics at scale, IEEE IPDPS-ScaDL’20

See also Pushing the limit of molecular dynamics with ab initio accuracy to 100 million atoms with machine learning

W. Jia et al., ACM/IEEE Supercomputing, SC20