Parallel Molecular Dynamics

Aiichiro Nakano

Collaboratory for Advanced Computing & Simulations
Department of Computer Science
Department of Physics & Astronomy
Department of Quantitative & Computational Biology
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)

https://aiichironakano.github.io/cs596/src/pmd
https://github.com/KenichiNomura/binary-LJ-pmd

Parallel Molecular Dynamics

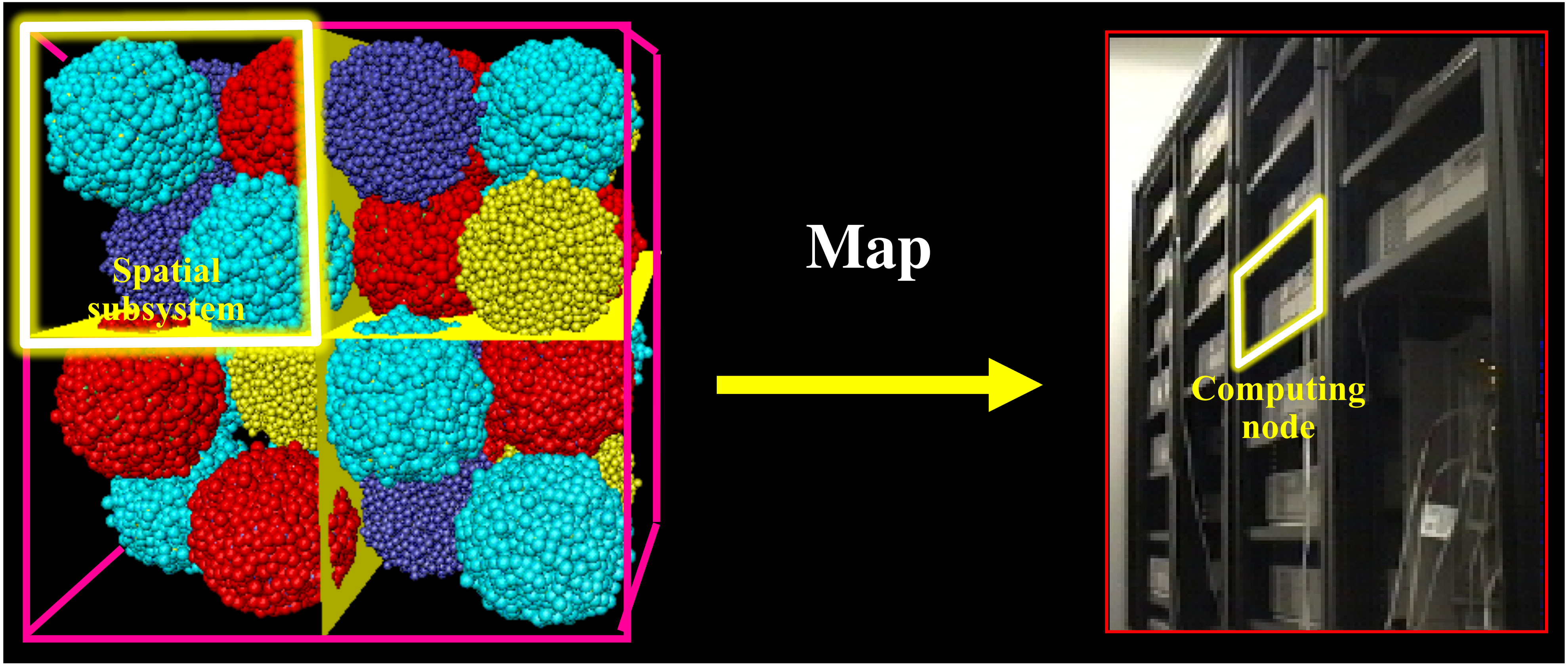
Spatial decomposition (short-ranged):

Will learn other decomposition schemes later:
https://aiichironakano.github.io/cs596/NT.pdf

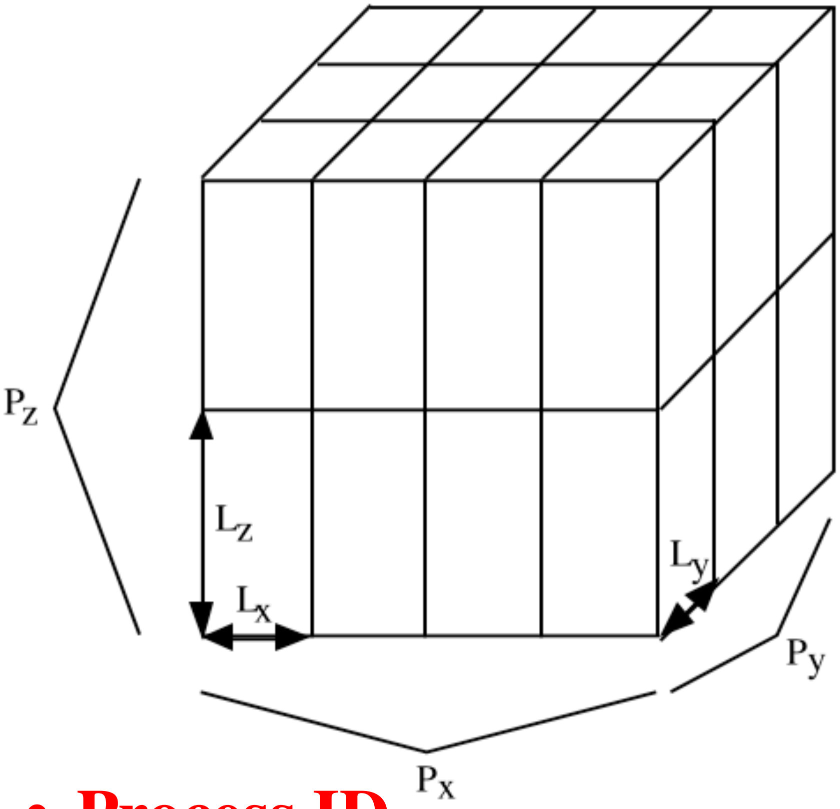
1. **Divide the physical space into subspaces of equal volume**
2. **Assign each subspace to a computing node (more generally, to a process) in a parallel computer**

   or MPI rank
3. **Each node computes forces on the atoms in its subspace & updates their positions & velocities**

   Who does what

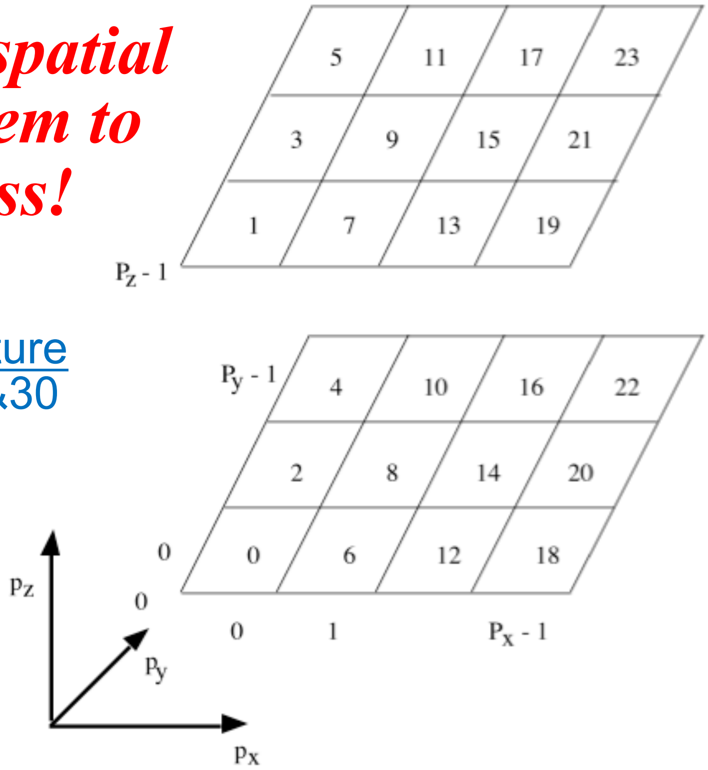


Spatial Decomposition



Map a spatial subsystem to a process!

cf. MD lecture slides 29 &30



- **Process ID**

  Vector

  $$p_x = p/(P_yP_z)$$
  $$p_y = (p/P_z) \bmod P_y$$
  $$p_z = p \bmod P_z$$

  Which 3D subspace?

  Scalar

  $$p = p_x \times P_yP_z + p_y \times P_z + p_z$$

  Rank

nproc = vproc[0]×vproc[1] ×vproc[2]

P_x P_y P_z

In pmd.h

```
int vproc[3] = {1,1,2}, nproc = 2;
```

In pmd.c

rank

```
MPI_Comm_rank(MPI_COMM_WORLD, &sid);
vid[0] = sid/(vproc[1]*vproc[2]);
vid[1] = (sid/vproc[2])%vproc[1];
vid[2] = sid%vproc[2];
```

Neighbor Process ID

$p'_\alpha(\kappa) = [p_\alpha + \delta_\alpha(\kappa) + P_\alpha] \bmod P_\alpha \quad (\kappa = 0,\ldots,5;\ \alpha = x, y, z)$ neighbor's vector process ID

$p'(\kappa) = p'_x(\kappa)\times P_y P_z + p'_y(\kappa)\times P_z + p'_z(\kappa)$ neighbor's rank

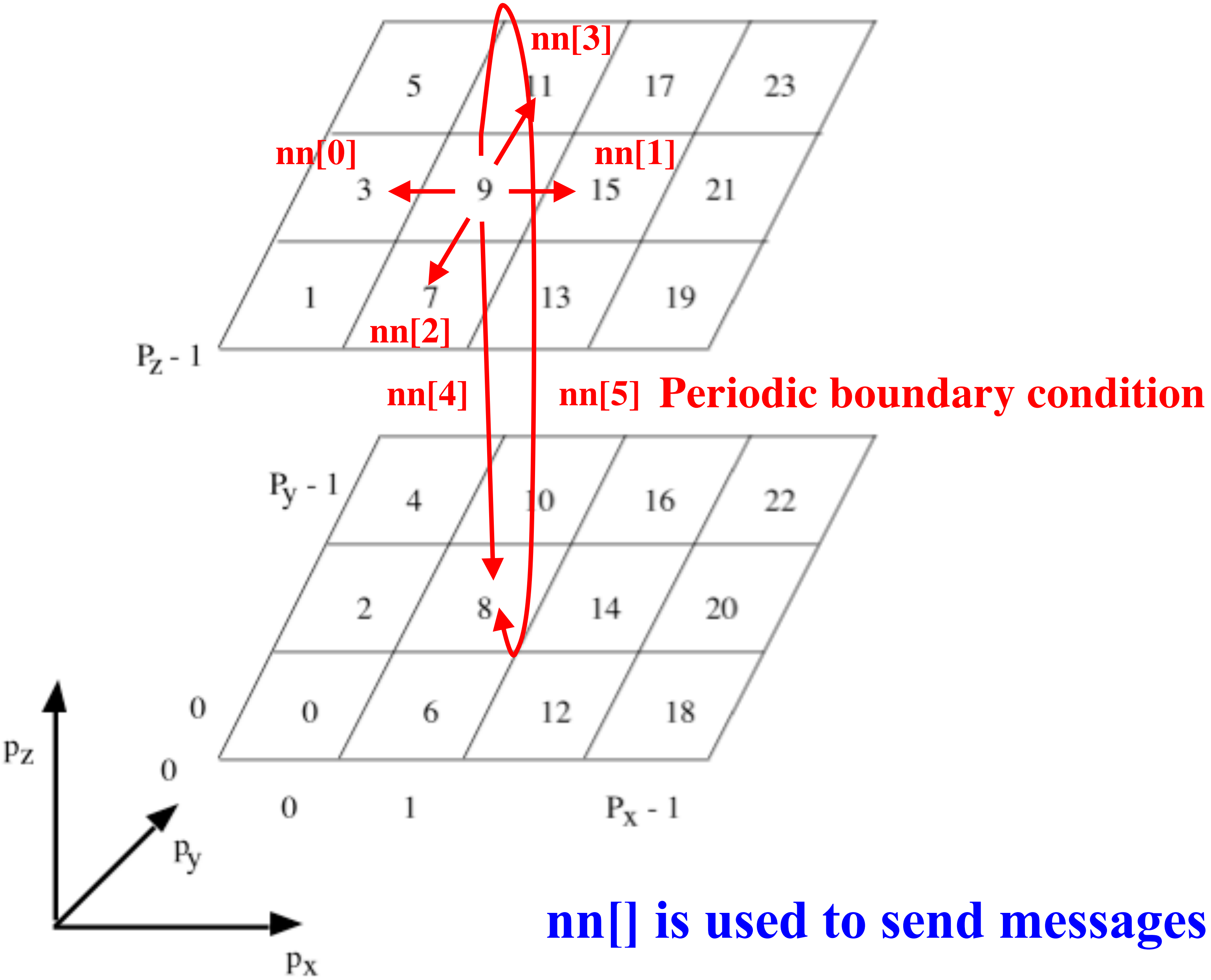
Neighbor ID, κ	$\vec{\delta} = (\delta_x, \delta_y, \delta_z)$	$\vec{\Delta} = (\Delta_x, \Delta_y, \Delta_z)$
0 (east)	(-1, 0, 0)	$(-L_x, 0, 0)$
1 (west)	(1, 0, 0)	$(L_x, 0, 0)$
2 (north)	(0, -1, 0)	$(0, -L_y, 0)$
3 (south)	(0, 1, 0)	$(0, L_y, 0)$
4 (up)	(0, 0, -1)	$(0, 0, -L_z)$
5 (down)	(0, 0, 1)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

- **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

```
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]; wrap around
  nn[ku] = k1[0]*vproc[1]*vproc[2]+k1[1]*vproc[2]+k1[2];destination rank
  for (a=0; a<3; a++) sv[ku][a] = al[a]*iv[ku][a]; coordinate shift for
}                                                   self-centric parallelization
```

Neighbor Process ID Example

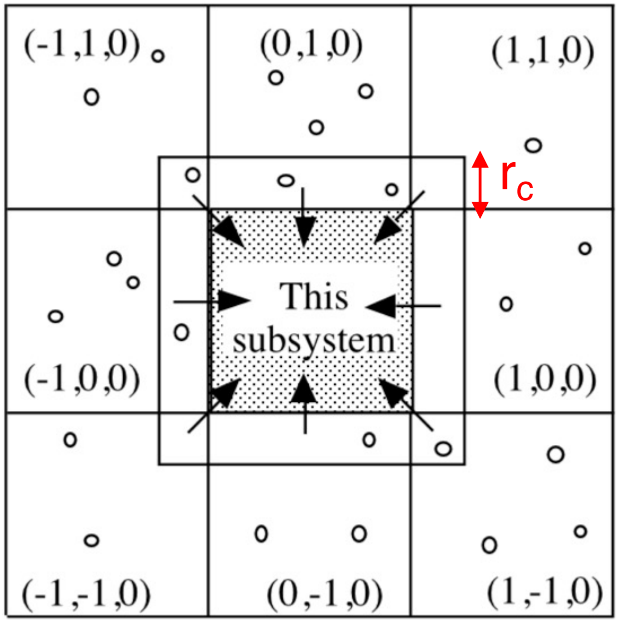


Periodic boundary condition

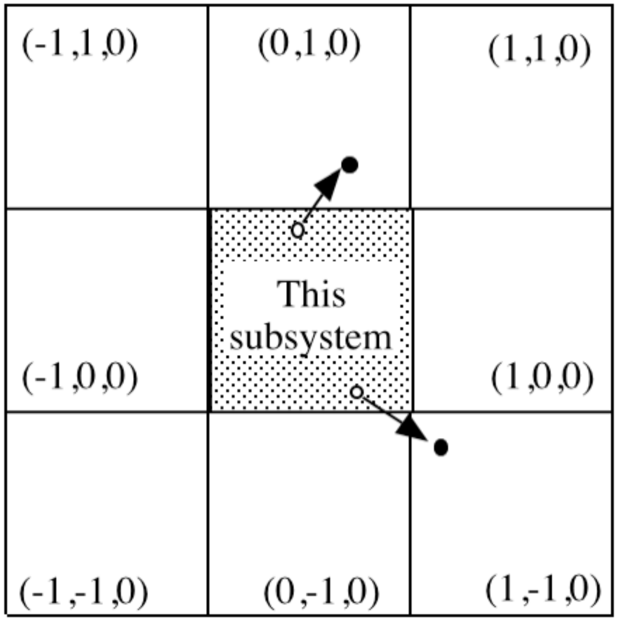
nn[] is used to send messages

Parallel MD Concepts

Atom caching

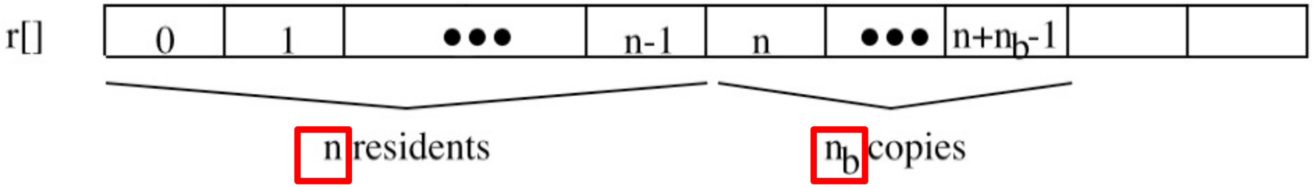


Atom migration



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)

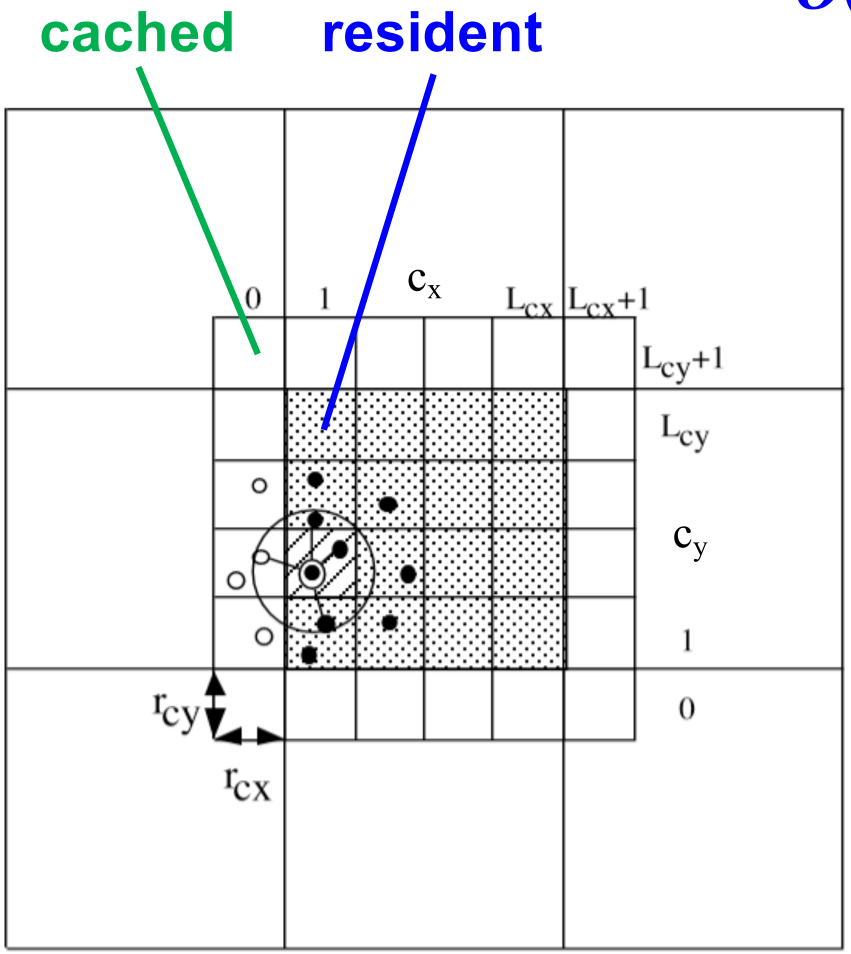
Data structure



Linked-List Cell Method

Search for pairs only within the nearest neighbor cells:

$$O(N^2) \rightarrow O(N)$$



- **Cell size**

  $L_{c\alpha} = \lfloor L_\alpha / r_c \rfloor$

  $r_{c\alpha} = L_\alpha / L_{c\alpha}$ (α = x, y, z)

- **Cell index**

  $c = c_x(L_{cy}+2)(L_{cz}+2) + c_y(L_{cz}+2) + c_z$

  $c_x = c/[(L_{cy}+2)(L_{cz}+2)]$

  $c_y = [c/(L_{cz}+2)] \bmod (L_{cy}+2)$

  $c_z = c \bmod (L_{cz}+2)$

- **Atom → cell mapping**

  $c_\alpha = \lfloor (r_\alpha + r_{c\alpha})/r_{c\alpha} \rfloor$ (α = x, y, z)

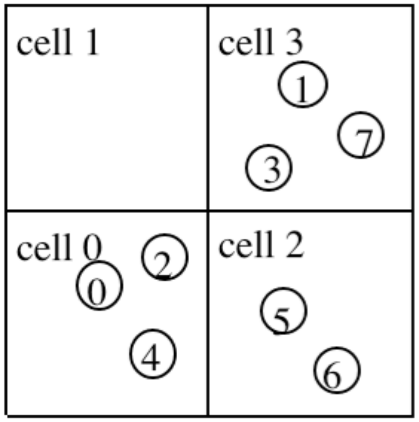
Only change from serial lmd.c in green:
Augmented cells to include cached atoms

List Construction Algorithm

```
/* Reset the headers, head */
for (c=0; c<lcxyz2; c++) head[c] = EMPTY;
/* Scan atoms to construct headers, head, & linked lists, lscl */
for (i=0; i<n+nb; i++) {
  /* 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;
}
```

Consider n_b cached atoms

Position offset by one cell



In the above:

```
lcyz2 = lc2[1]*lc2[2]
```

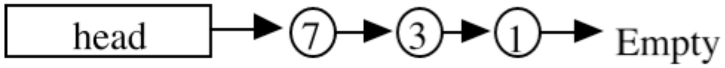
where

```
lc2[a] = lc[a]+2 (a = 0,1,2)
lcxyz2 = lcyz2*lc2[0]
```

	0	1	2	3
head	4	E	6	7

	0	1	2	3	4	5	6	7
lscl	E	E	0	1	2	E	5	3

Change from serial lmd.c in green



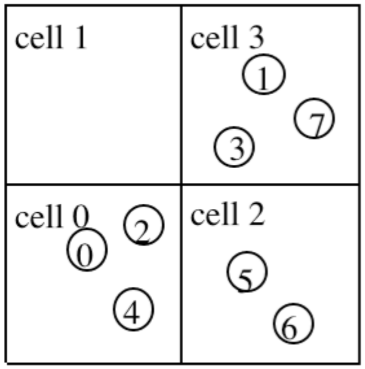
Interaction Computation

```
/* Scan inner cells (resident) */
for (mc[0]=1; mc[0]<=lc[0]; (mc[0])++)
for (mc[1]=1; mc[1]<=lc[1]; (mc[1])++)
for (mc[2]=1; mc[2]<=lc[2]; (mc[2])++) {
 /* 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 (mc1[0]=mc[0]-1; mc1[0]<=mc[0]+1; (mc1[0])++)
 for (mc1[1]=mc[1]-1; mc1[1]<=mc[1]+1; (mc1[1])++)
 for (mc1[2]=mc[2]-1; mc1[2]<=mc[2]+1; (mc1[2])++) {
  /* Calculate the scalar cell index of the neighbor cell */
  c1 = mc1[0]*lcyz2+mc1[1]*lc2[2]+mc1[2];
  /* Scan atom i in cell c */
  i = head[c];
  while (i != EMPTY) {
   /* Scan atom j in cell c1 */
   j = head[c1];
   while (j != EMPTY) {
    ...
    if (i<j && $r_{ij}<r_c^2$) Process pair (i, j)
    ...
    j = lscl[j];
   }
   i = lscl[i];
  }
 }
}
```

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

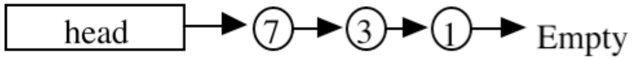
Resident atoms may interact with cached atoms (*cf.* slide 7)

Change from serial lmd.c in green



	0	1	2	3
head	4	E	6	7

	0	1	2	3	4	5	6	7
lscl	E	E	0	1	2	E	5	3

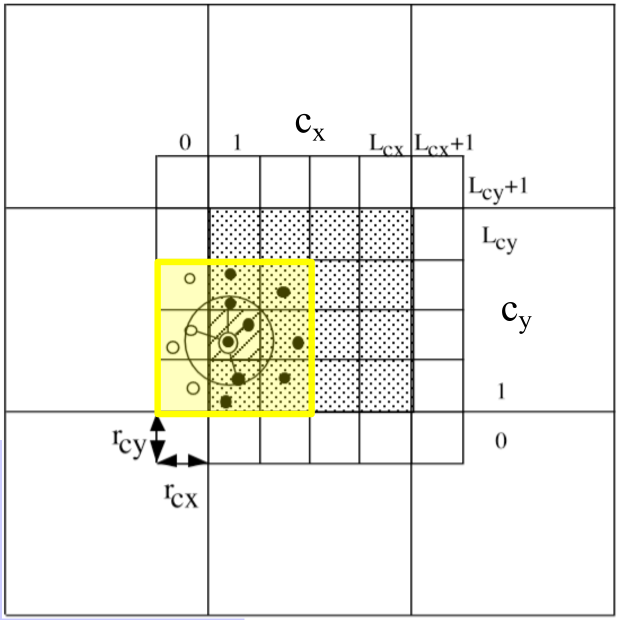


Parallel Interaction Computation

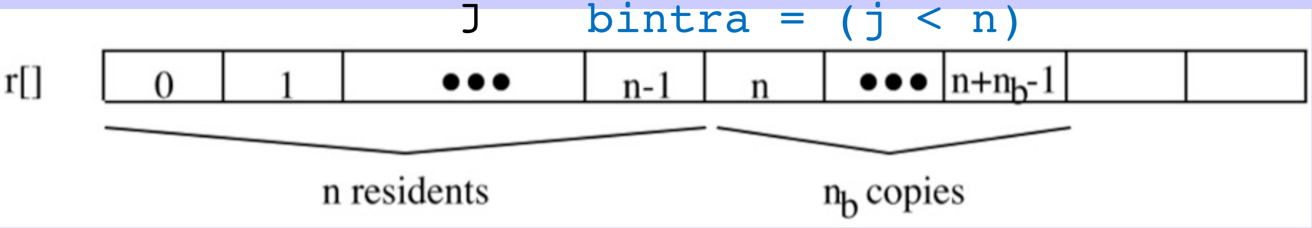
SPMD: Who does what?

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**

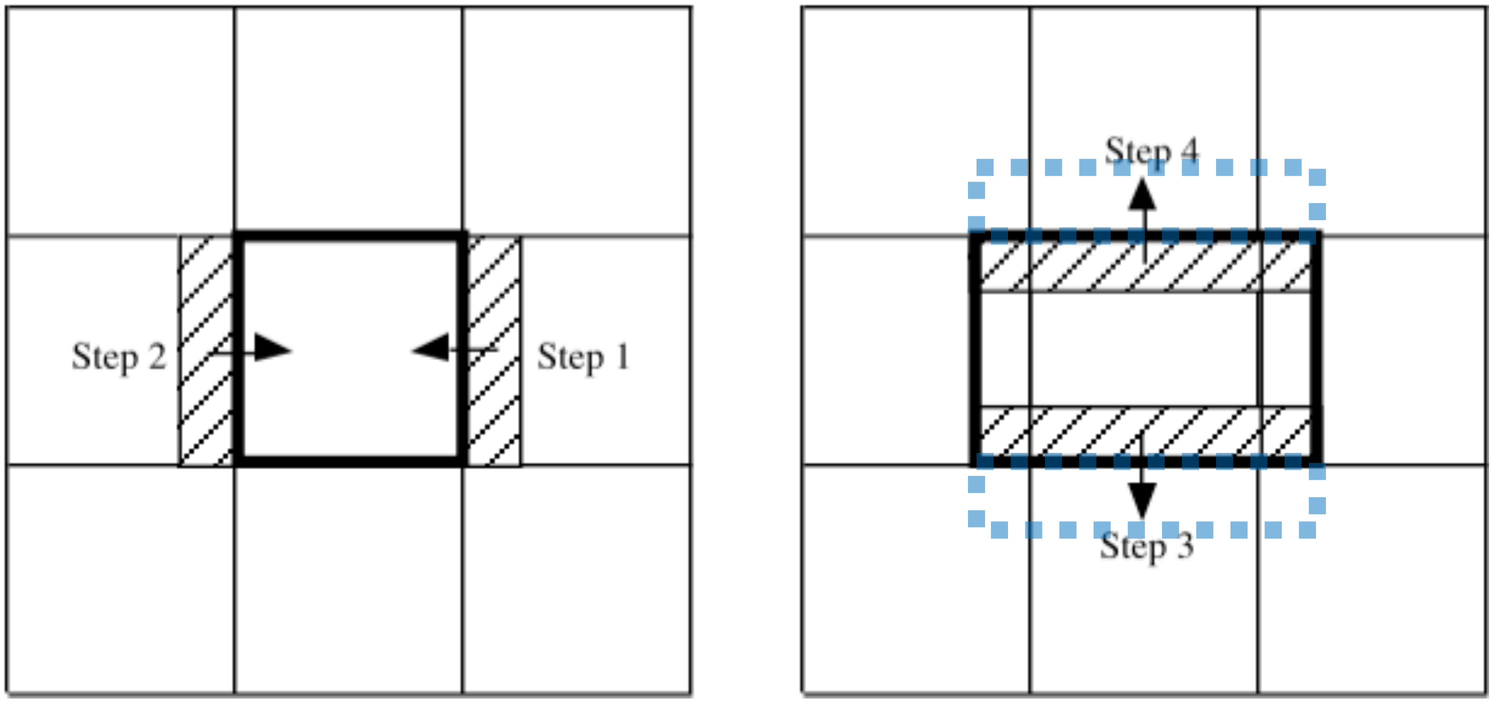


```
for resident cells, c {
  for neighbor (resident or cached) cells, c1 {
    scan atom i in cell c using c’s linked list {
      scan atom j in cell c1 using c1’s linked list {
        ...
        if (i<j && $r_{ij}<r_c^2$) {
          compute pair force $\mathbf{a}_{ij}$ & potential u($r_{ij}$)
          bintra = j < n; // j is resident?
          $\mathbf{a}_i$ += $\mathbf{a}_{ij}$; if (bintra) $\mathbf{a}_j$ -= $\mathbf{a}_{ij}$;
          if (bintra) lpe += u($r_{ij}$); else lpe += u($r_{ij}$)/2;
        }
      }
    }
  }
}
MPI_Allreduce(&lpe, &potEnergy,...,MPI_SUM,...);
```



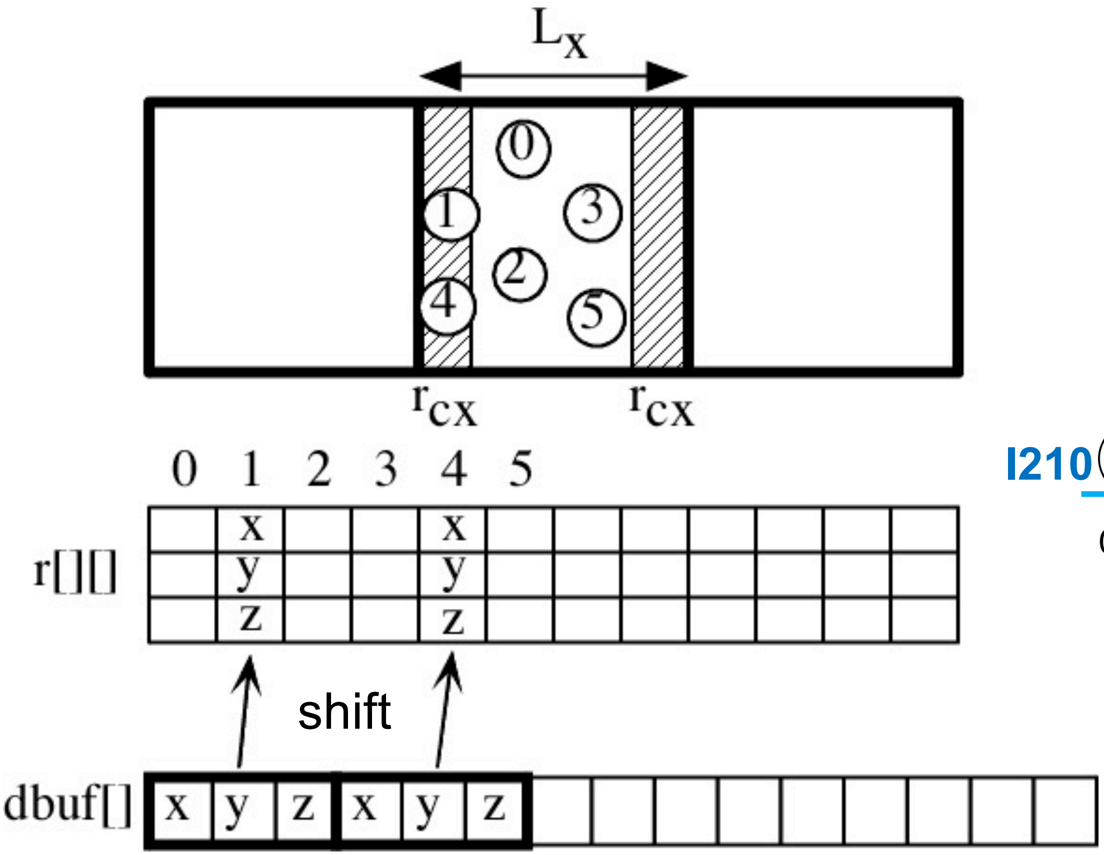
Atom Caching: `atom_copy()`

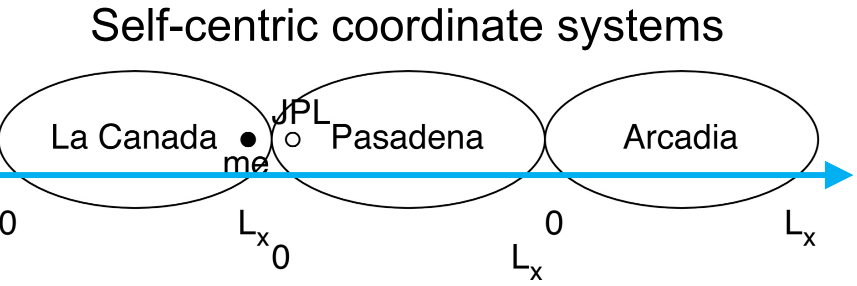
Caching from 26 neighbors in 6 steps (by forwarding)



```
Reset the number of received cache atoms, nbnew = 0
for x, y, and z directions
  Make boundary-atom lists, lsb, for lower and higher directions 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 nbnew;
  endfor
endfor
nb = nbnew
```

Implementing Atom Caching





Copying condition

```
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]
}
```

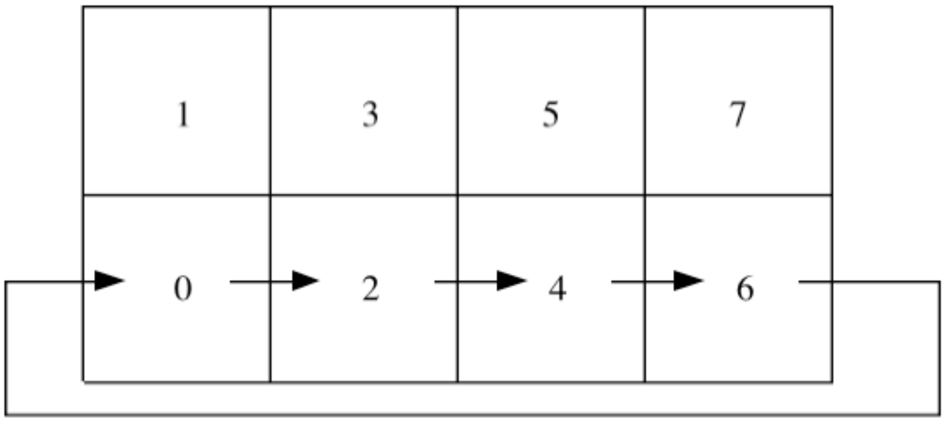
3 phases of message passing

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

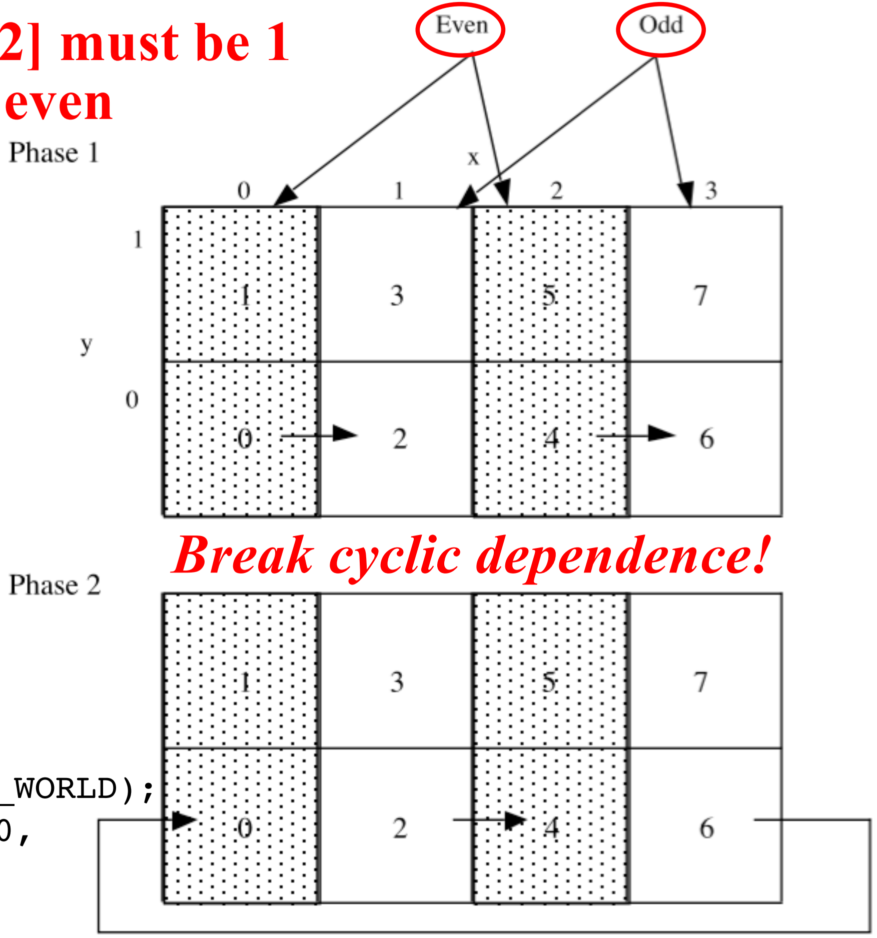
See atom_copy() in pmd.c

Deadlock Avoidance

Cyclic dependence



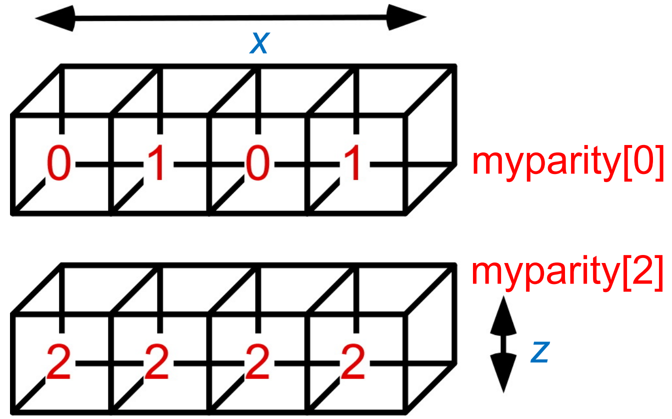
vproc[0|1|2] must be 1 or even



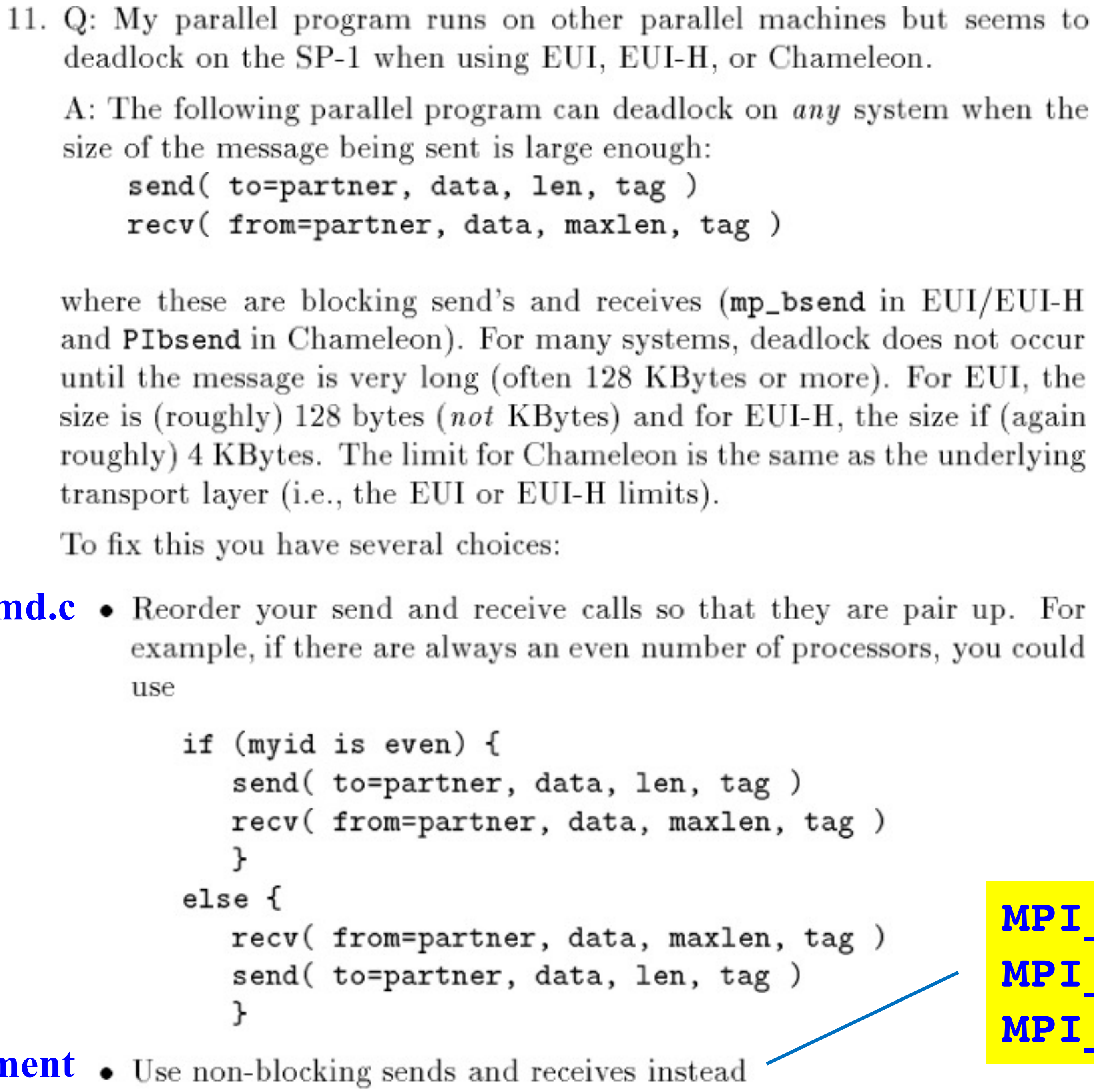
Break cyclic dependence!

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
```



ANL IBM SP1 User’s Guide (’94)



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:

```
    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

```
  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

```
MPI_Irecv();
MPI_Send();
MPI_Wait();
```

Digress: Polyacetylene & Peierls Distortion

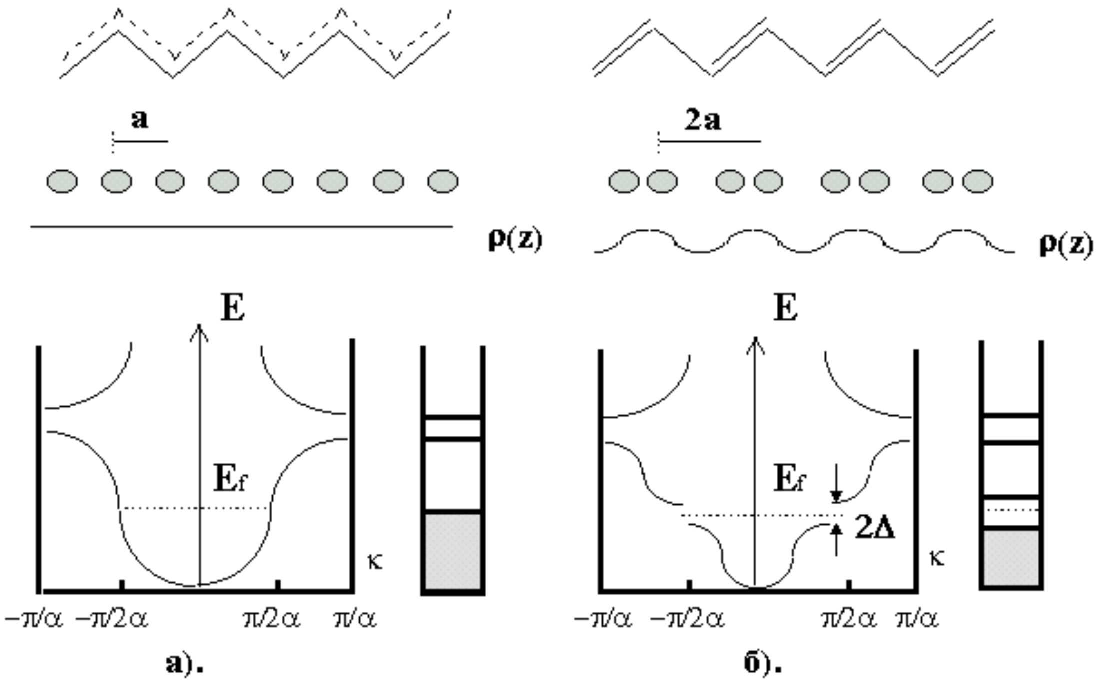
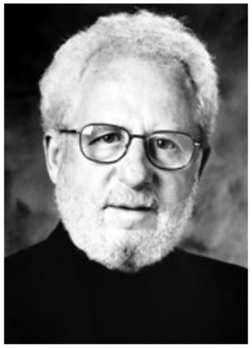


Fig. 1. Electron dispersion and a band pattern of one-dimensional molecular system: a). metallic and b). insulator state, (**ρ(z)**–a electronic density, **a**–a lattice period).

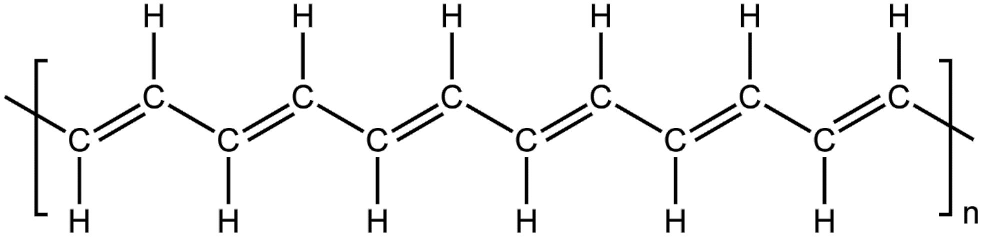


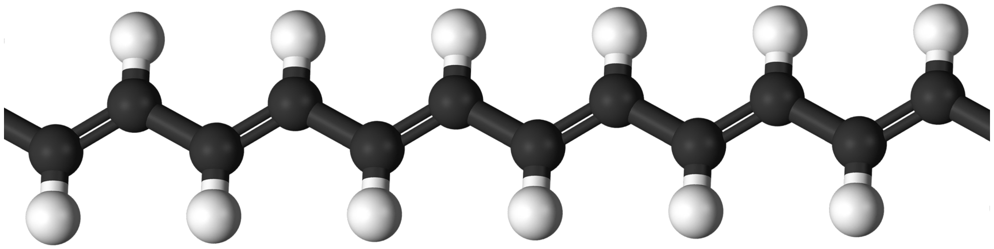
Alan J. Heeger
Prize share: 1/3

Alan G. MacDiarmid
Prize share: 1/3

Hideki Shirakawa
Prize share: 1/3

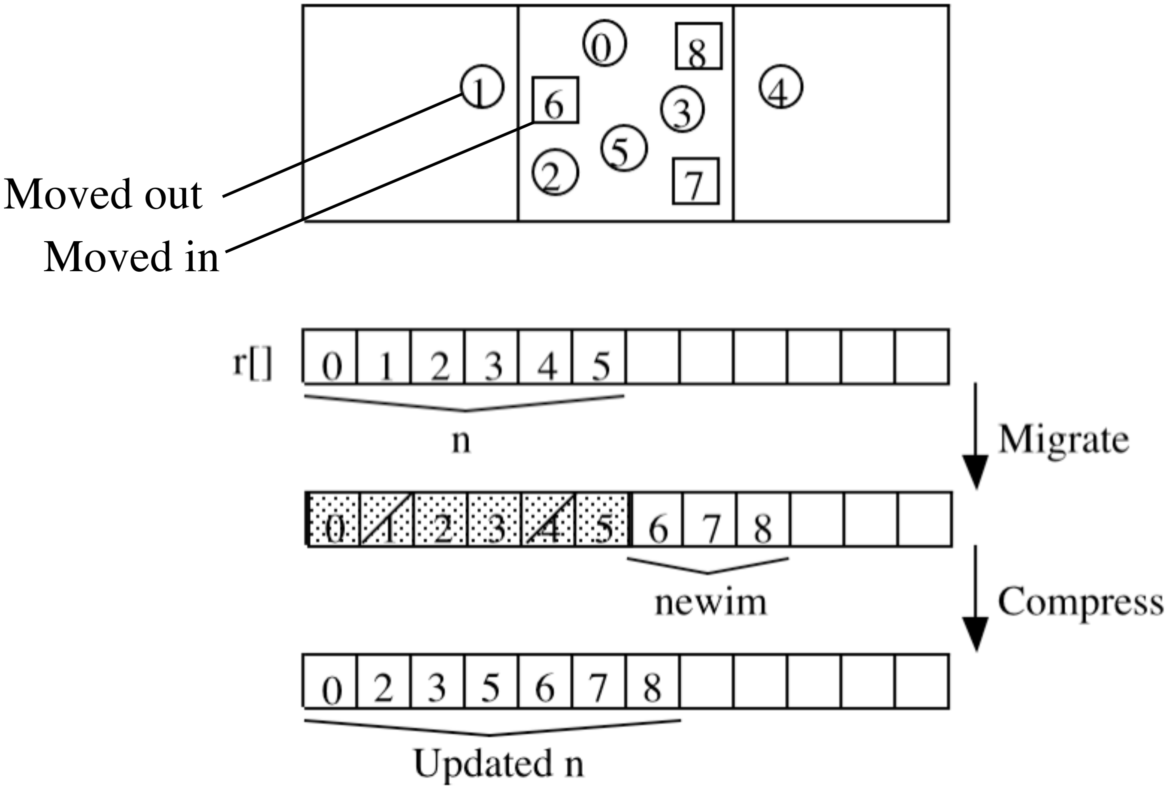
The Nobel Prize in Chemistry 2000 was awarded jointly to Alan J. Heeger, Alan G. MacDiarmid and Hideki Shirakawa *"for the discovery and development of conductive polymers"*.





Nature's spontaneous even-odd symmetry breaking

Atom Migration: atom_move()

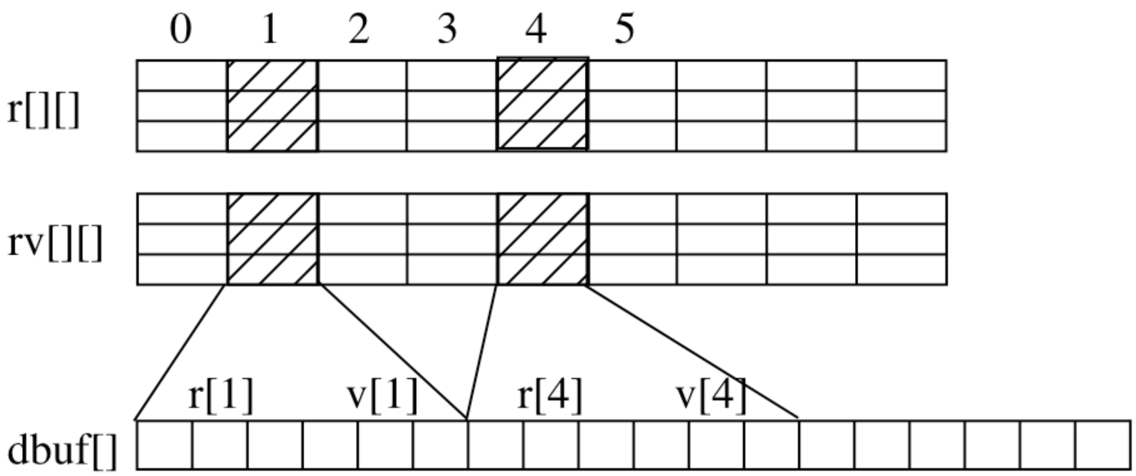


```
Reset the number of received new immigrants, newim = 0
for x, y, and z directions
  Make moving-atom lists, mvque, for lower and higher directions including both
  resident, n, and immigrant, newim, atoms but excluding those already moved out
  for lower and higher directions
    Send/receive moving-atom 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

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



3 phases of message passing

1. **Message buffering: dbuf ← r−sv (shift) & rv, gather**
   Mark MOVED_OUT in r
2. **Message passing: dbufr ← dbuf**
   Send dbuf
   Receive dbufr
3. **Message storing: r & rv ← dbufr, append after the residents**

See atom_move() in pmd.c

Bottom Line: Parallel MD

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 & updates 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 Quantitative & Computational Biology
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: $$E_P = \frac{S_P}{P} = \frac{W_P T(W_1,1)}{P W_1 T(W_P,P)}$$

How to scale W_P with P?

Fixed Problem-Size (Strong) Scaling

Solve the same problem faster using more processors

$W_P = W$—constant (strong scaling)

$$S_P = \frac{T(W,1)}{T(W,P)} \leq P$$

- **Speedup:**

$$S_P = \frac{W_P T(W_1,1)}{W_1 T(W_P,P)} = \frac{T(W,1)}{T(W,P)}$$

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

- **Efficiency:**

$$E_P = \frac{T(W,1)}{PT(W,P)}$$

$$E_P = \frac{S_P}{P} = \frac{W_P T(W_1,1)}{P W_1 T(W_P,P)}$$

- **Amdahl's law:** f (= sequential fraction of the workload) limits the asymptotic speedup

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

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

$$\therefore S_P \rightarrow \frac{1}{f} \quad (P \rightarrow \infty)$$

Isogranular (Weak) Scaling

Solve a larger problem within the same time duration using more processors $E_P = \frac{T(w,1)}{T(Pw,P)} \leq 1$

$W_P = Pw$ **(weak scaling)**
w **= constant workload per processor (granularity)**

- **Speedup:** $S_P = \dfrac{S(P \bullet w, P)}{S(w,1)} = \dfrac{P \bullet w / T(P \bullet w, P)}{w / T(w,1)} = \dfrac{P \bullet T(w,1)}{T(P \bullet w, P)}$

- **Efficiency:** $E_P = \dfrac{S_P}{P} = \dfrac{T(w,1)}{T(P \bullet w, P)}$

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

$$E_P = \frac{S_P}{P} = \frac{W_P T(W_1, 1)}{P W_1 T(W_P, P)}$$

Analysis of Parallel MD

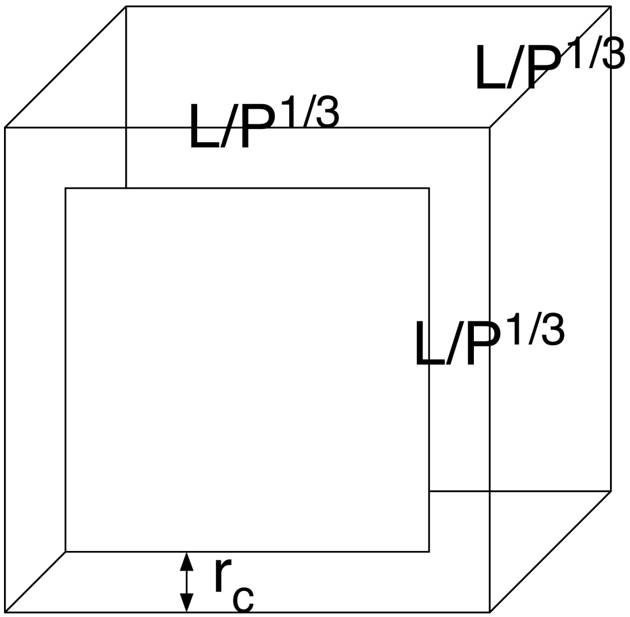
- **Parallel execution time:**
  Workload ∝ Number of atoms, *N* (linked-list cell algorithm)

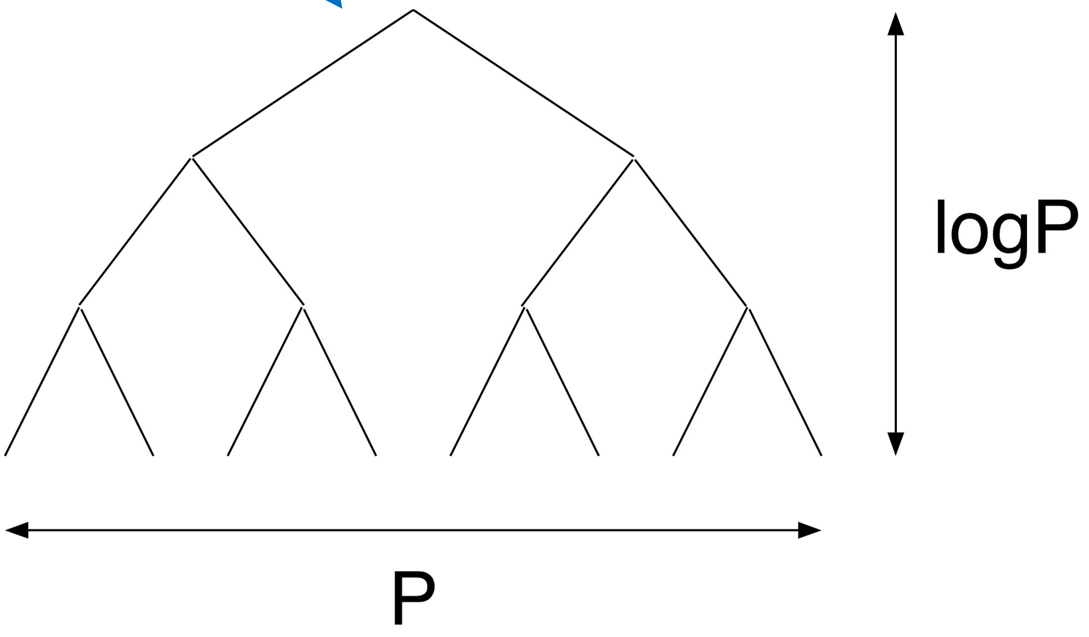
MPI_Allreduce()

$$T(N,P) = T_{\text{comp}}(N,P) + T_{\text{comm}}(N,P) + T_{\text{global}}(P)$$
$$= a\frac{N}{P} + b\left(\frac{N}{P}\right)^{2/3} + c\log P$$

$$\overbrace{6}^{\text{facets}}\;\overbrace{\frac{L^2}{P^{2/3}}r_c}^{\text{cached volume}} \times \overbrace{\rho}^{\text{atom density}}$$
$$= 6r_c\frac{N^{2/3}/\rho^{2/3}}{P^{2/3}}\rho$$
$$= 6r_c\rho^{1/3}\left(\frac{N}{P}\right)^{2/3}$$

$$\left(\because \frac{N}{L^3} = \rho \Rightarrow L^2 = \frac{N^{2/3}}{\rho^{2/3}}\right)$$





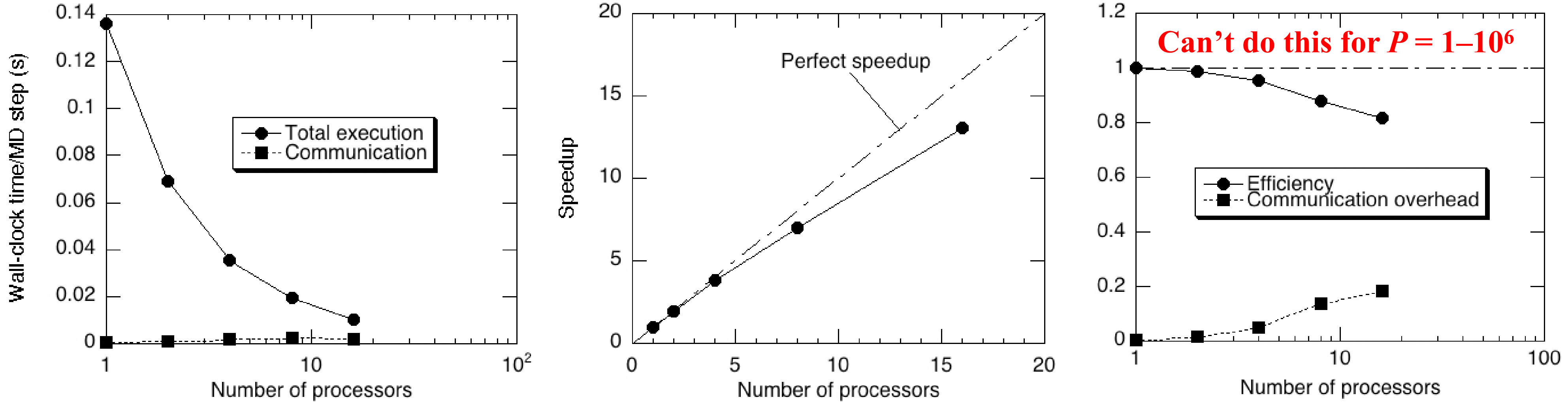
Fixed Problem-Size Scaling

- **Speedup:**

$$S_P = \frac{T(N,1)}{T(N,P)} = \frac{aN}{aN/P + b(N/P)^{2/3} + c\log P} = \frac{P}{1 + \frac{b}{a}\left(\frac{P}{N}\right)^{1/3} + \frac{c}{a}\frac{P\log P}{N}}$$

- **Efficiency:**

$$E_P = \frac{S_P}{P} = \frac{1}{1 + \frac{b}{a}\boxed{\left(\frac{P}{N}\right)^{1/3}} + \frac{c}{a}\frac{\boxed{P\log P}}{N}}$$



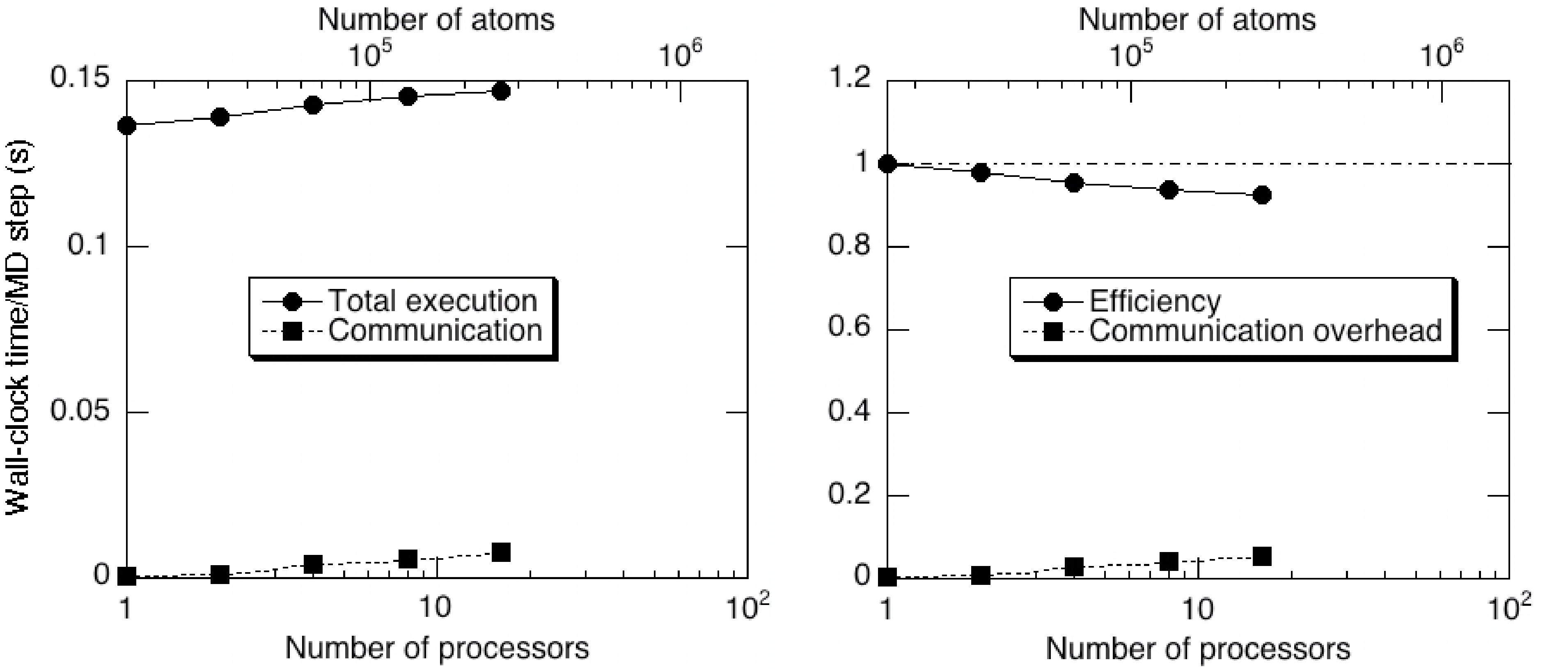
Can't do this for $P = 1–10^6$

pmd.c: *N* = 16,384, on HPC (predecessor of CARC)

Isogranular Scaling of Parallel MD

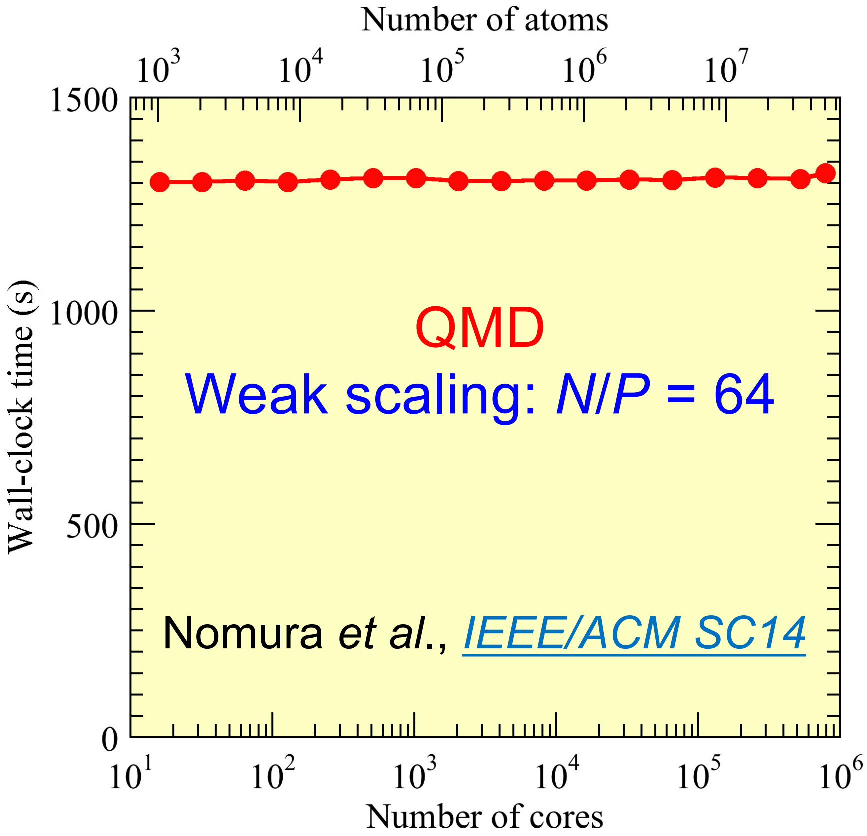
- ***n* = *N*/*P* = constant:** **doable for arbitrarily large *P***
- **Efficiency:**

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



pmd.c: *N*/*P* = 16,384, on HPC (predecessor of CARC)

High-End Parallel MD



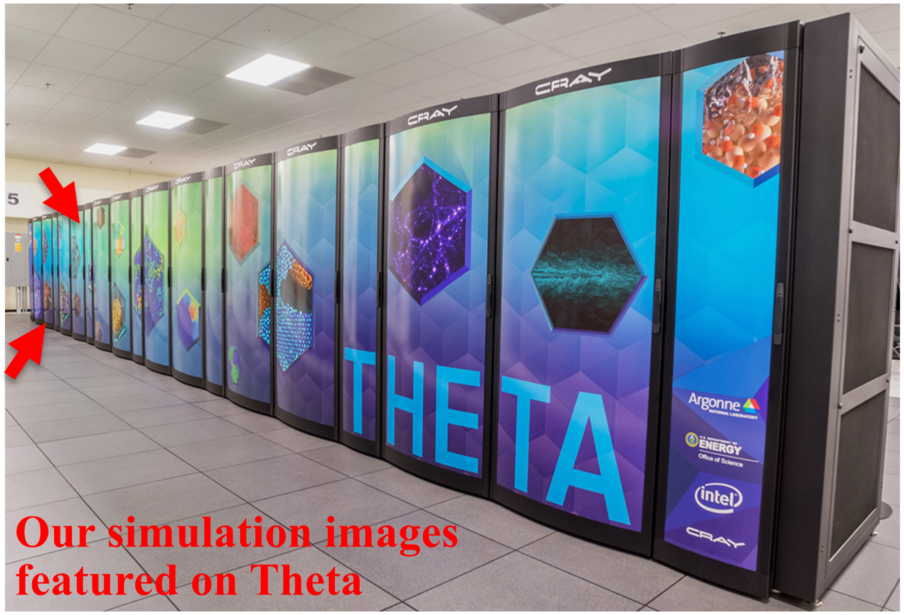
- **4.9 trillion-atom space-time multiresolution MD (MRMD) of SiO_2**
- **8.5 billion-atom fast reactive force-field (F-ReaxFF) RMD of RDX**
- **39.8 trillion grid points (50.3 million-atom) DC-DFT QMD of SiC**

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

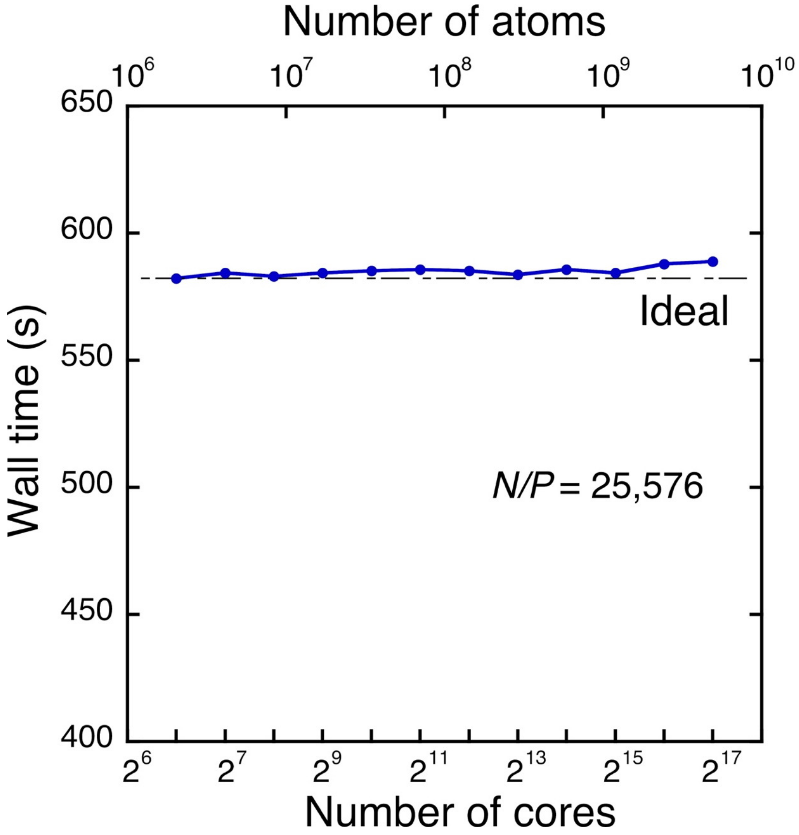
MD (molecular dynamics): MRMD
RMD (reactive molecular dynamics): F-ReaxFF
QMD (quantum molecular dynamics): DC-DFT

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 on Theta supercomputer at Argonne National Laboratory**



Our simulation images featured on Theta



K. Liu *et al*., Shift-collapse acceleration of generalized polarizable reactive molecular dynamics for machine learning-assisted computational synthesis of layered materials, *Proc. ScalA18* (IEEE, '18)

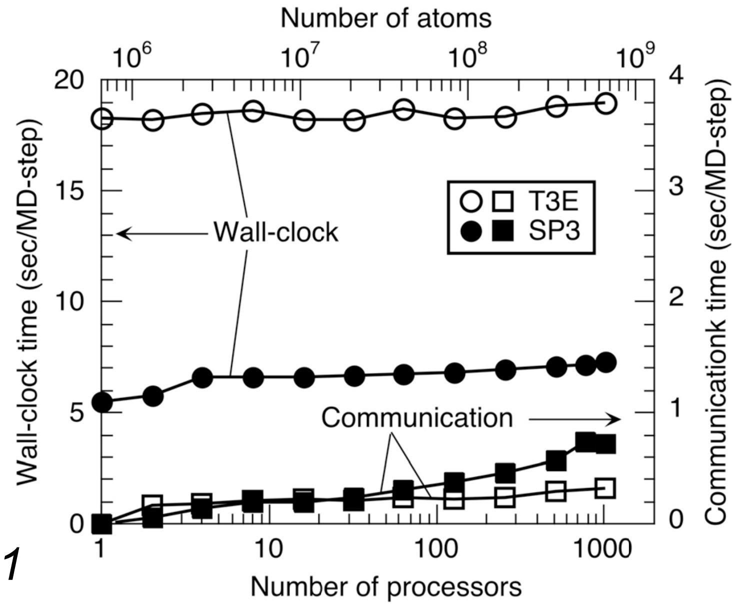
Quantum MD@Scale

Quantum dynamics at scale: ultrafast control of emergent functional materials

S. C. Tiwari, P. Sakdhnagool, R. K. Kalia, A. Krishnamoorthy, M. Kunaseth, A. Nakano, K. Nomura, P. Rajak, F. Shimojo, Y. Luo & P. Vashishta

Best Paper in *ACM HPC Asia 2020*

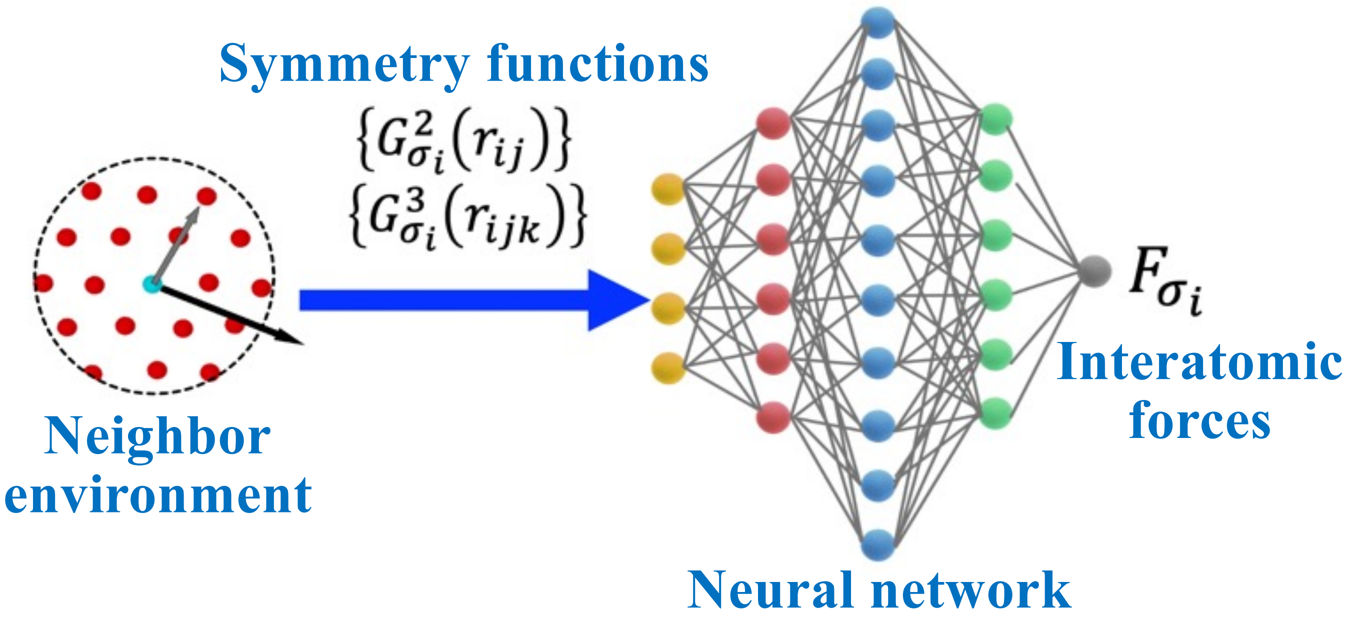
19 years since



Scalable atomistic simulation algorithms for materials research, A. Nakano *et al*., Best Paper, *IEEE/ACM Supercomputing 2001, SC01*

Neural MD@Scale

- **Neural-network quantum molecular dynamics (NNQMD) could revolutionize atomistic modeling of materials, providing quantum-mechanical accuracy at a fraction of computational cost** [*Phys. Rev. Lett.* **126**, 216403 ('21); *J. Phys. Chem. Lett.* **12**, 6020 ('21); *Nature Commun.* **15**, 3911 ('24)]



Symmetry functions

$\{G^2_{\sigma_i}(r_{ij})\}$

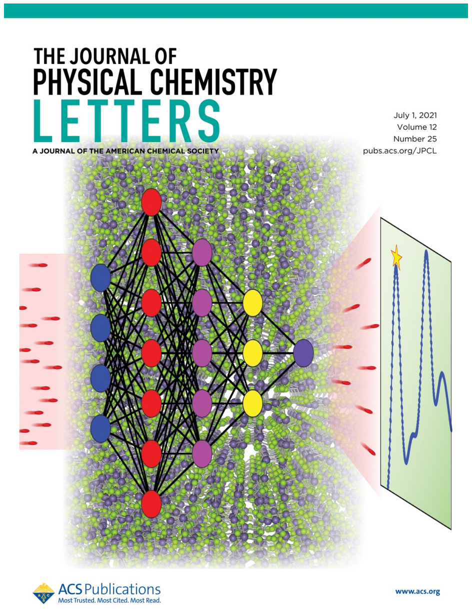
$\{G^3_{\sigma_i}(r_{ijk})\}$

Neighbor environment

Neural network

F_{σ_i}

Interatomic forces



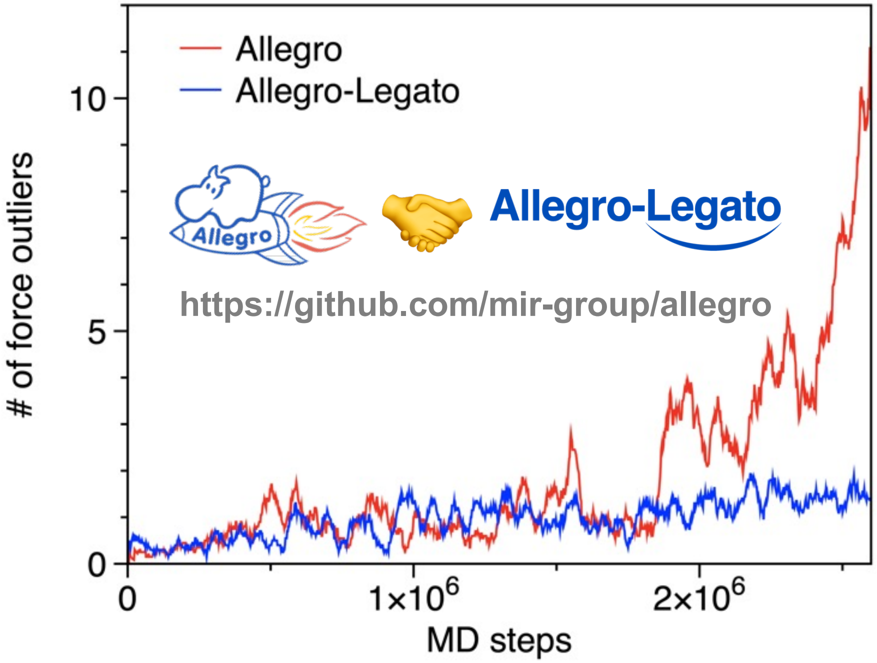
Neural network molecular dynamics at scale & Ex-NNQMD: extreme-scale neural network quantum molecular dynamics,
P. Rajak *et al.*, *IEEE IPDPS ScaDL 20 & 21*

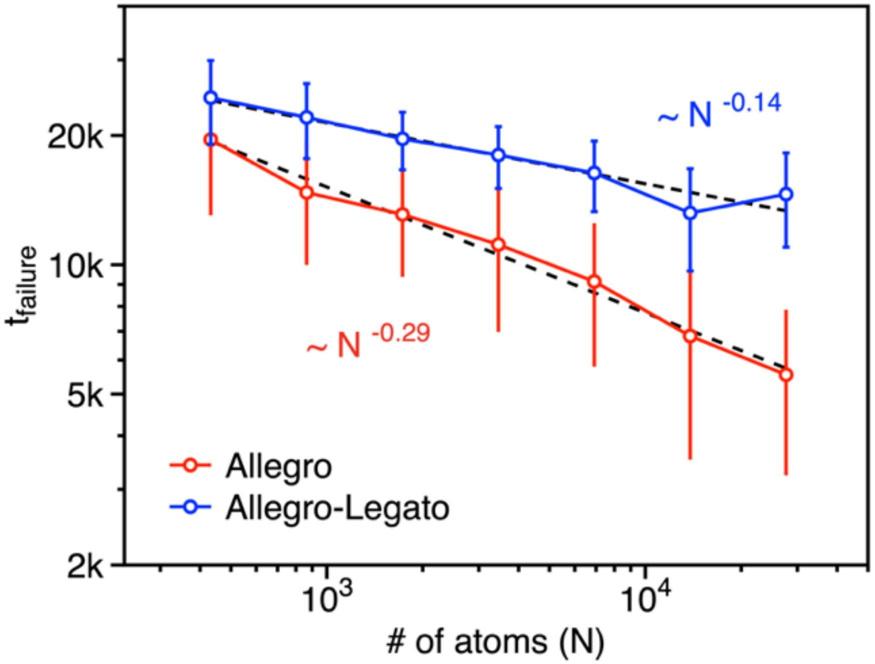
See also Pushing the limit of molecular dynamics with ab initio accuracy to 100 million atoms with machine learning
W. Jia *et al.*, *ACM/IEEE Supercomputing, SC20*

Fast, Robust & Scalable: Allegro-Legato

- **Allegro (fast) NNQMD: State-of-the-art *accuracy & speed* founded on group-theoretical equivariance & local descriptors** [Musaelian *et al*., *Nat. Commun.* **14**, 579 ('23)]
- **Fidelity-scaling problem: On massively parallel computers, growing number of unphysical (adversarial) force predictions prohibits simulations involving larger numbers of atoms for longer times**
- **Allegro-Legato (fast and "smooth"): *Sharpness aware minimization (SAM)* enhances the *robustness* of Allegro through improved smoothness of loss landscape**

  $\mathbf{w}_* = \mathrm{argmin}_{\mathbf{w}}\left[L(\mathbf{w}) + \max_{\|\boldsymbol{\epsilon}\|_2 \le \rho}\{L(\mathbf{w}+\boldsymbol{\epsilon}) - L(\mathbf{w})\}\right]$ **(*L*: loss; w: model parameters)**
- **Elongated time-to-failure scaling, $t_{\mathrm{failure}} = O(N^{-\beta})$, without sacrificing accuracy or speed, thereby achieving spectroscopically stable long-time Hamiltonian trajectory**





H. Ibayashi *et al*., *ISC23—LNCS* **13948**, 223 ('23); *arXiv*: 2303.08169