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

Parallel-computing basics using MD as an example

Parallel Computing

Glossary

- *Parallel algorithm design = decomposition (who does what?)*
  – **Task:** Units of computation into which the main computation is subdivided
  – **Decomposition:** Dividing a computation into subsets of tasks that may be executed in parallel

- *Goal of parallel algorithm design = maximize concurrency & minimize task dependency/interaction*

  – **Concurrency:** The maximum number of tasks that can be executed simultaneously in parallel (limited by task dependency/interaction)

  – **Task dependency:** A task depends on another task, if the former uses data produced by the latter; represented by a directed acyclic graph called **task-dependency graph**

  – **Task interaction:** Tasks share inputs, outputs or intermediate data

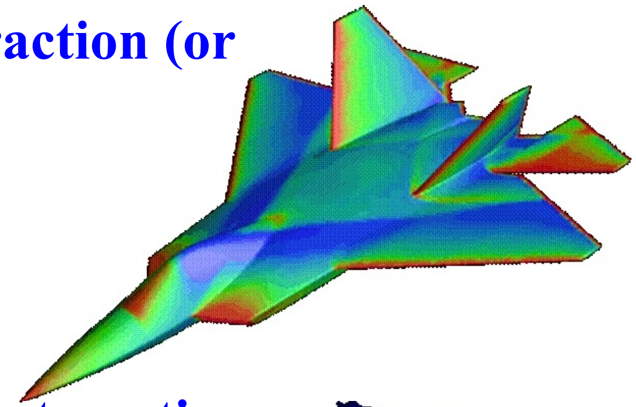
- **Granularity:** Size of decomposed tasks: **fine-grained** = a large number of small tasks; **coarse-grained** = a small number of large tasks

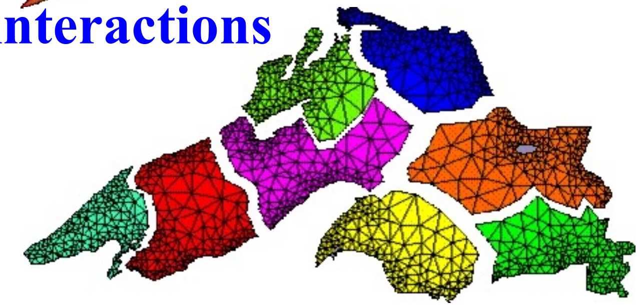
- **Mapping:** Assign tasks (or processes = running programs to perform the tasks) to processors

A. Grama, A. Gupta, G. Karypis, & V. Kumar,
Introduction to Parallel Computing, 2nd Ed. (Addison-Wesley, ’03) Chap. 3

Parallel Algorithm Design

- **Decomposition** (example: molecular dynamics)
  - Spatial decomposition (≈ domain decomposition)—coarse-grained
  - Particle decomposition—single-instruction multiple-data (SIMD) computers
  - Force decomposition—fine-grained
- **Maximal-concurrency algorithm:** Expose data locality in the problem (*e.g.*, divide-&-conquer)
- **Scalability:** Achieve a large fraction of perfect speed-up (= number of processors) on a large number of processors
- **Load balancing:** Keep all processors equally busy
- **Optimization:** Optimal mapping to minimize task interaction (or communication between processes)
  - Owner-computes rule
  - Minimize the volume & frequency of data exchanges
  - Computation-communication overlapping
  - Data & computation replication
- **Issues:** Regular *vs*. irregular & static *vs*. dynamic task interactions





Parallel Supercomputers

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	1,110,144	151.90	214.35	2,942
4	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096
5	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94.64	125.71	7,438

Measured performance (in Pflop/s)

Theoretical performance

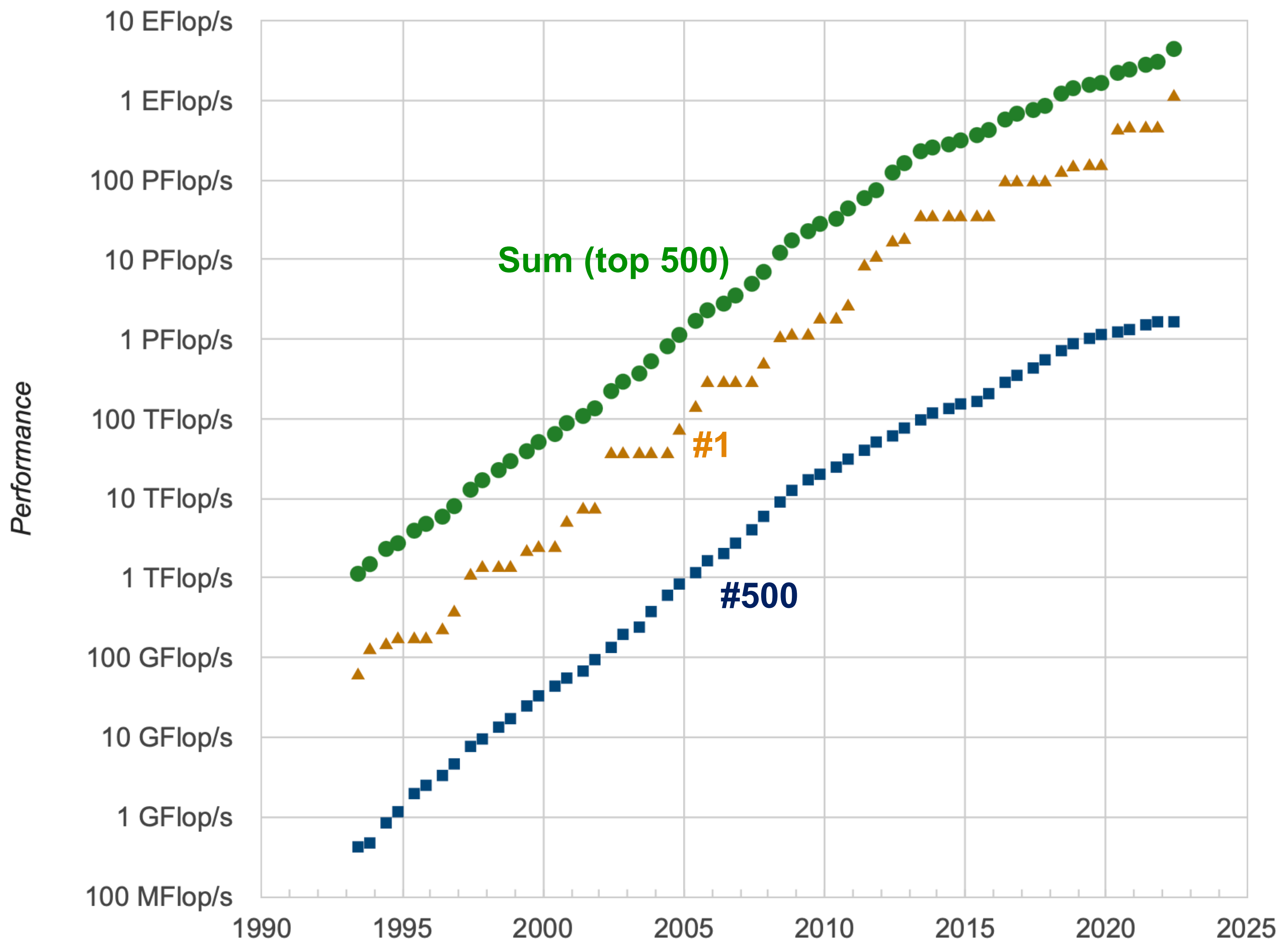
http://www.top500.org (June '22)

flops = floating-point operations/second

M (mega)	= 10^6
G (giga)	= 10^9
T (Tera)	= 10^{12}
P (Peta)	= 10^{15}
E (Exa)	= 10^{18}
Z (Zetta)	= 10^{21}
Y (Yotta)	= 10^{24}

1.1 exaflop/s Frontier

Performance Development



Message Passing Interface

MPI (Message Passing Interface): A standard message passing system that enables us to write & run applications on parallel computers (http://www.mcs.anl.gov/mpi**).**

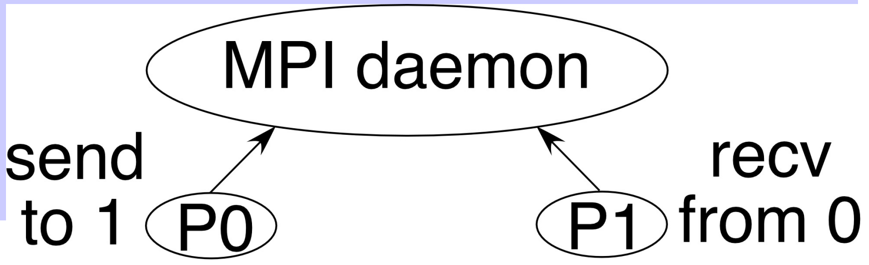
```
#include "mpi.h"
#include <stdio.h>
main(int argc, char *argv[]) {
  MPI_Status status;
  int myid;
  int n;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  if (myid == 0) {
    n = 777;
    MPI_Send(&n, 1, MPI_INT, 1, 10, MPI_COMM_WORLD);
  }
  else {
    MPI_Recv(&n, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, &status);
    printf("n = %d\n", n);
  }
  MPI_Finalize();
}
```

MPI rank

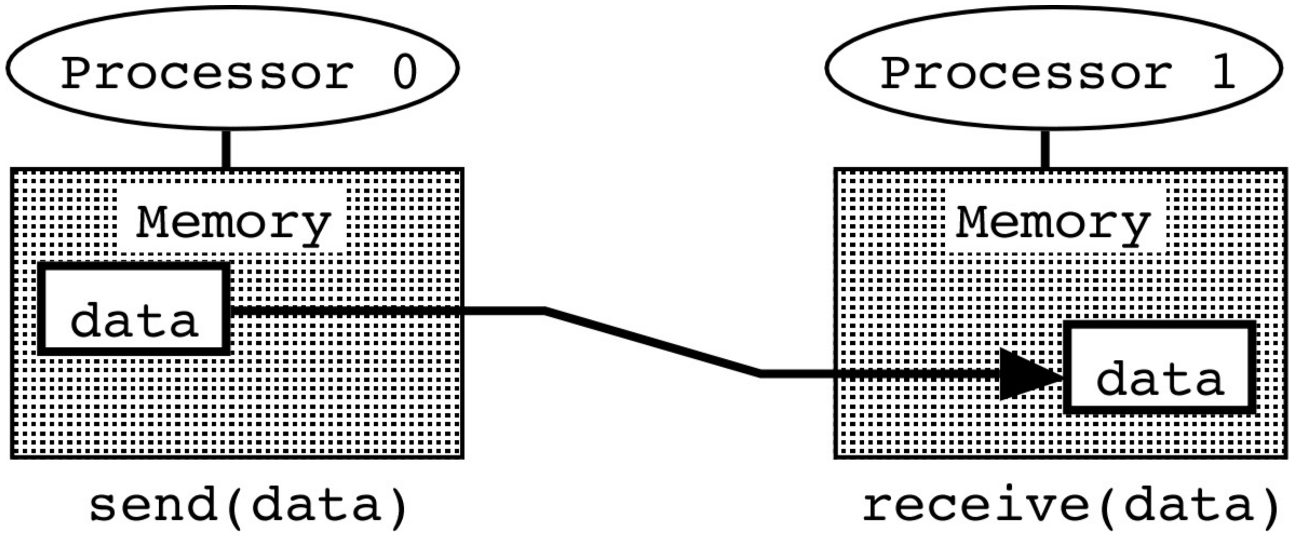
Matching message labels

Data triplet

To/from whom



Single Program Multiple Data (SPMD)



Process 0

```
if (myid == 0) {
  n = 777;
  MPI_Send(&n,...);
}
else {
  MPI_Recv(&n,...);
  printf(...);
}
```

Process 1

```
if (myid == 0) {
  n = 777;
  MPI_Send(&n,...);
}
else {
  MPI_Recv(&n,...);
  printf(...);
}
```

```
CSCI653_to_do()
{
 if (I == student)
 {
   do_assignment();
   MPI_Send(...);
 }
 else if (I == teacher)
 {
   MPI_Recv(...);
   grade();
 }
}
```

Parallel programming = choreography of “who does what”?

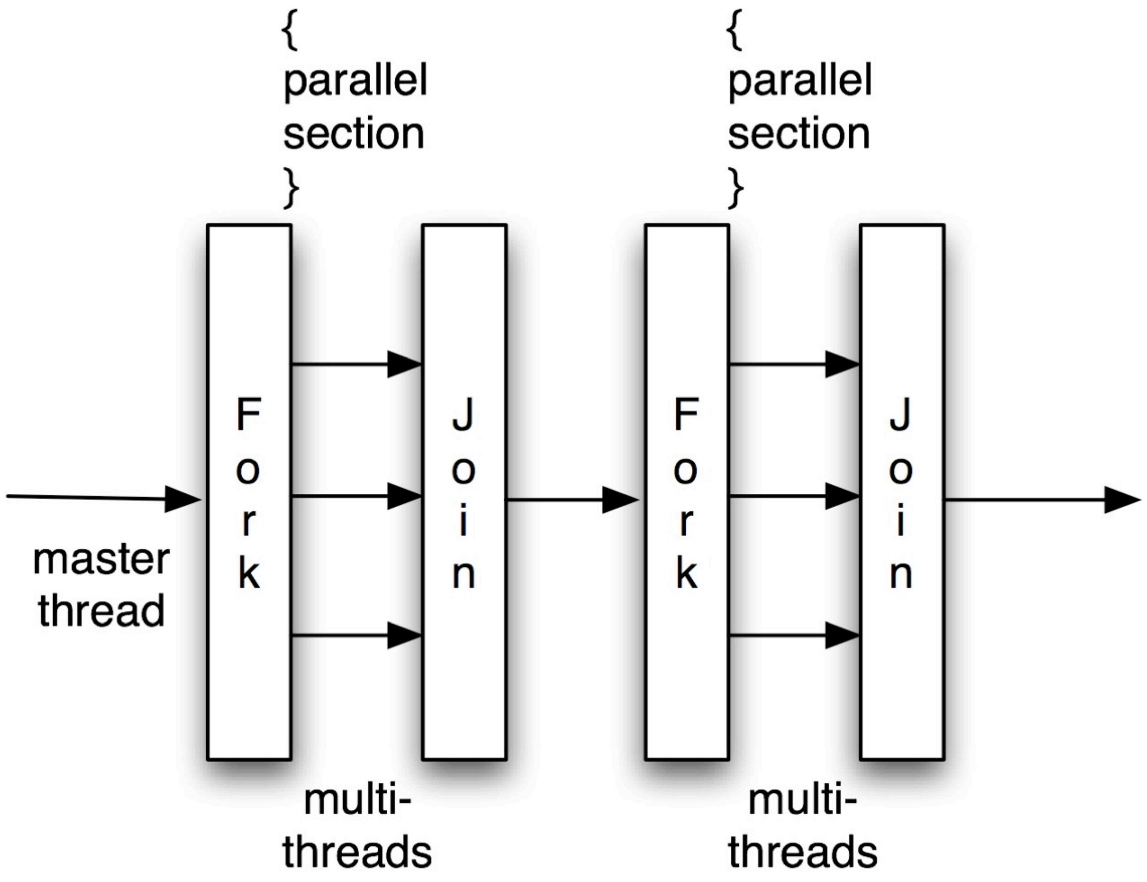
OpenMP

- **OpenMP (Open specifications for Multi Processing): Portable application program interface (API) for shared-memory parallel programming based on multi-threading by compiler directives** (http://www.openmp.org)
- **Fork-join parallelism:**
  > **Fork: Master thread spawns a team of threads as needed**
  > **Join: When the team of threads complete the statements in the parallel section, they terminate synchronously, leaving only the master thread**

  processes (= running programs) sharing resources
- **OpenMP is typically used to parallelize loops**
- **OpenMP threads communicate by sharing variables**

On HPC, compile as

```
> cc     ... -fopenmp
> mpicc ... -fopenmp
```



OpenMP Programming

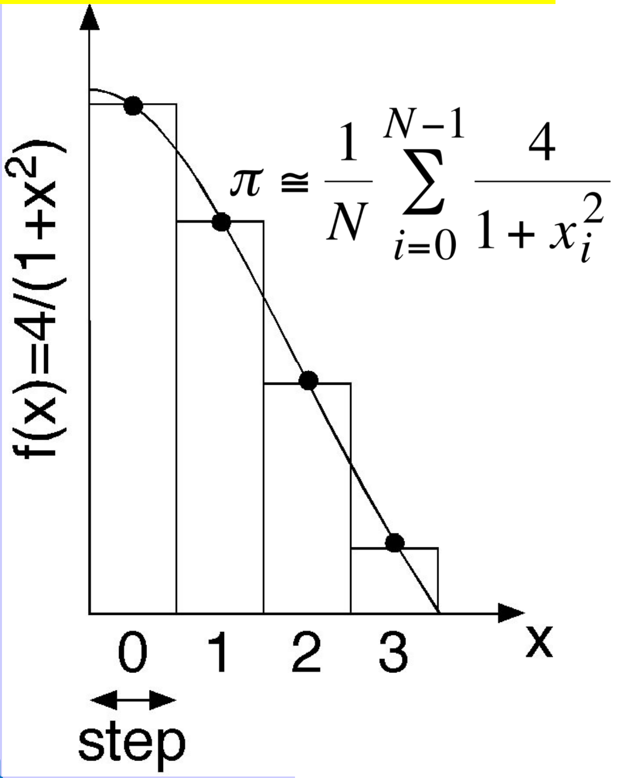
```
#include <stdio.h>
#include <omp.h>
#define NBIN 100000
#define MAX_THREADS 8
void main() {
  int nthreads,tid;
  double step,sum[MAX_THREADS]={0.0},pi=0.0;
  step = 1.0/NBIN;
#pragma omp parallel private(tid)
  {
    int i;
    double x;
    nthreads = omp_get_num_threads();
    tid = omp_get_thread_num();
    for (i=tid; i<NBIN; i+=nthreads) {
      x = (i+0.5)*step;
      sum[tid] += 4.0/(1.0+x*x);}
  }
  for(tid=0; tid<nthreads; tid++) pi += sum[tid]*step;
  printf("PI = %f\n",pi);
}
```

parallel section

Array of partial sums for multi-threads

data privatization to avoid race condition

thread reduction

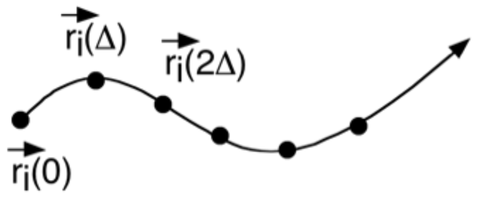


- **Obtain the number of threads & my thread ID**
- **By default, all variables are shared unless selectively changing storage attributes using private clauses**

Molecular Dynamics Algorithm

Time discretization

$$\begin{cases} \vec{r}_i(t+\Delta) = \vec{r}_i(t) + \vec{v}_i(t)\Delta + \frac{1}{2}\vec{a}_i(t)\Delta^2 \\ \vec{v}_i(t+\Delta) = \vec{v}_i(t) + \frac{\vec{a}_i(t) + \vec{a}_i(t+\Delta)}{2}\Delta \end{cases} \qquad \vec{a}_i = -\frac{1}{m}\frac{\partial V}{\partial \vec{r}_i}$$



Time stepping: Velocity Verlet algorithm

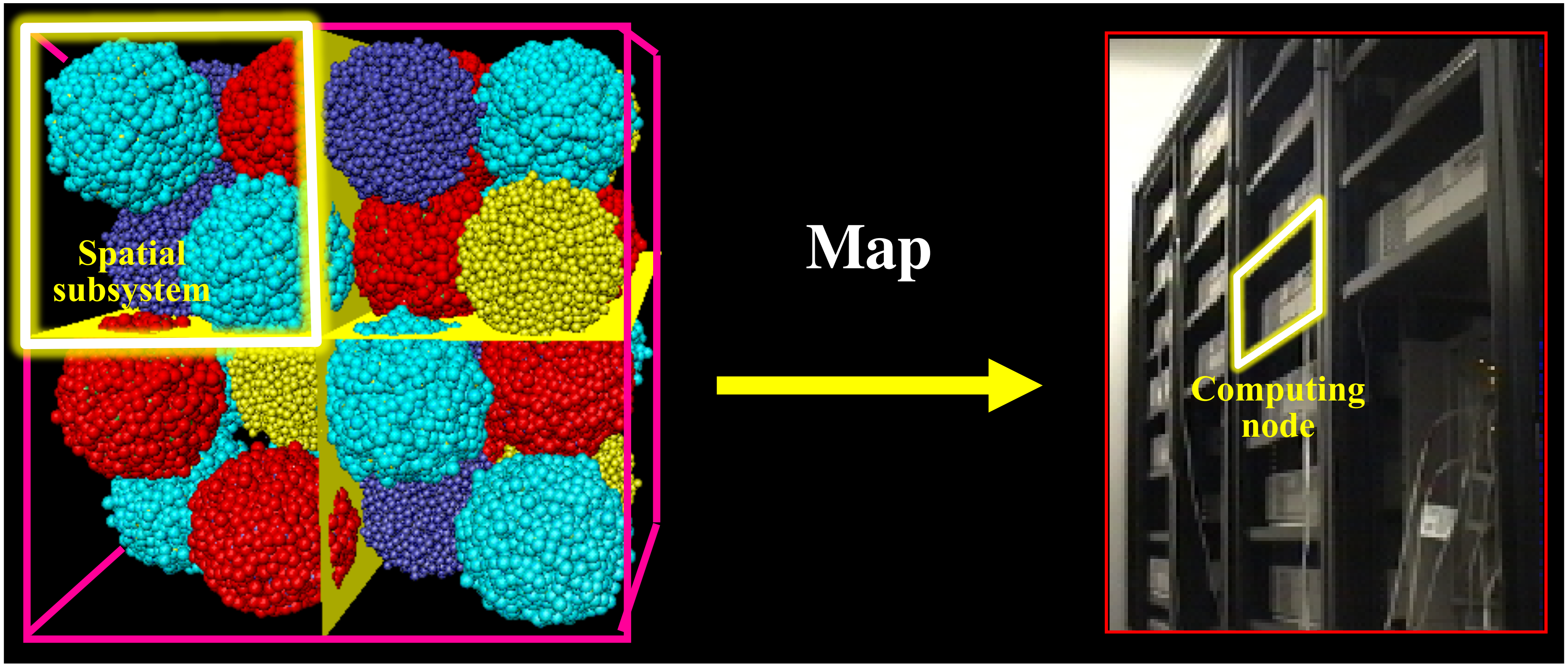
Given $\left(\vec{r}_i(t), \vec{v}_i(t)\right)$,

1. (Compute $\vec{a}_i(t)$ as a function of $\{\vec{r}_i(t)\}$)
2. $\vec{v}_i\left(t+\frac{\Delta}{2}\right) \leftarrow \vec{v}_i(t) + \frac{\Delta}{2}\vec{a}_i(t)$
3. $\vec{r}_i(t+\Delta) \leftarrow \vec{r}_i(t) + \vec{v}_i\left(t+\frac{\Delta}{2}\right)\Delta$
4. Compute $\vec{a}_i(t+\Delta)$ as a function of $\{\vec{r}_i(t+\Delta)\}$
5. $\vec{v}_i(t+\Delta) \leftarrow \vec{v}_i\left(t+\frac{\Delta}{2}\right) + \frac{\Delta}{2}\vec{a}_i(t+\Delta)$

Parallel Molecular Dynamics

Spatial decomposition (short ranged):

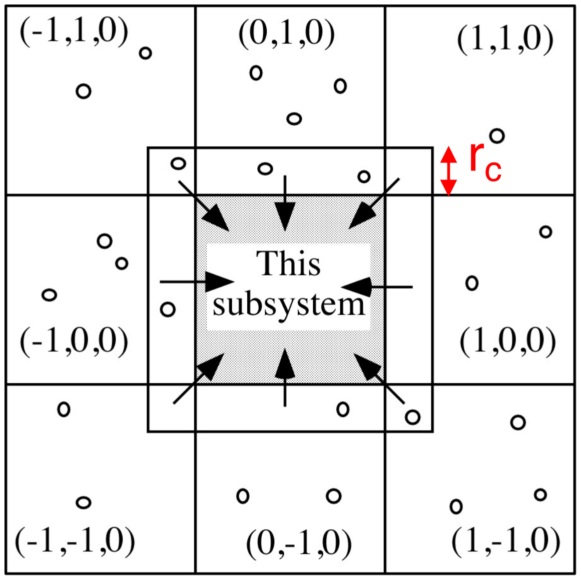
1. Divide the physical space into subspaces of equal volume
2. Assign each subspace to a compute node (more generally, to a process) in a parallel computer
   or MPI rank
3. Each node computes forces on the atoms in its subspace & updates their positions & velocities Who does what

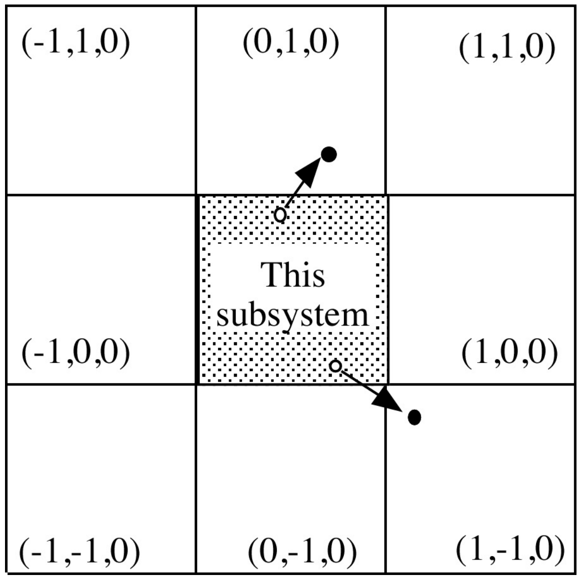


Parallel MD Algorithm

```
1.  $\vec{v}_i\left(t+\frac{\Delta}{2}\right) \leftarrow \vec{v}_i(t) + \frac{\Delta}{2}\vec{a}_i(t)$
2.  $\vec{r}_i(t+\Delta) \leftarrow \vec{r}_i(t) + \vec{v}_i\left(t+\frac{\Delta}{2}\right)\Delta$
3.  atom_move()     // migrate moved-out atoms
4.  atom_copy()     // cache surface atoms
5.  Compute $\vec{a}_i(t+\Delta)$ as a function of $\{\vec{r}_i(t+\Delta)\}$
6.  $\vec{v}_i(t+\Delta) \leftarrow \vec{v}_i\left(t+\frac{\Delta}{2}\right) + \frac{\Delta}{2}\vec{a}_i(t+\Delta)$
```

atom_copy()

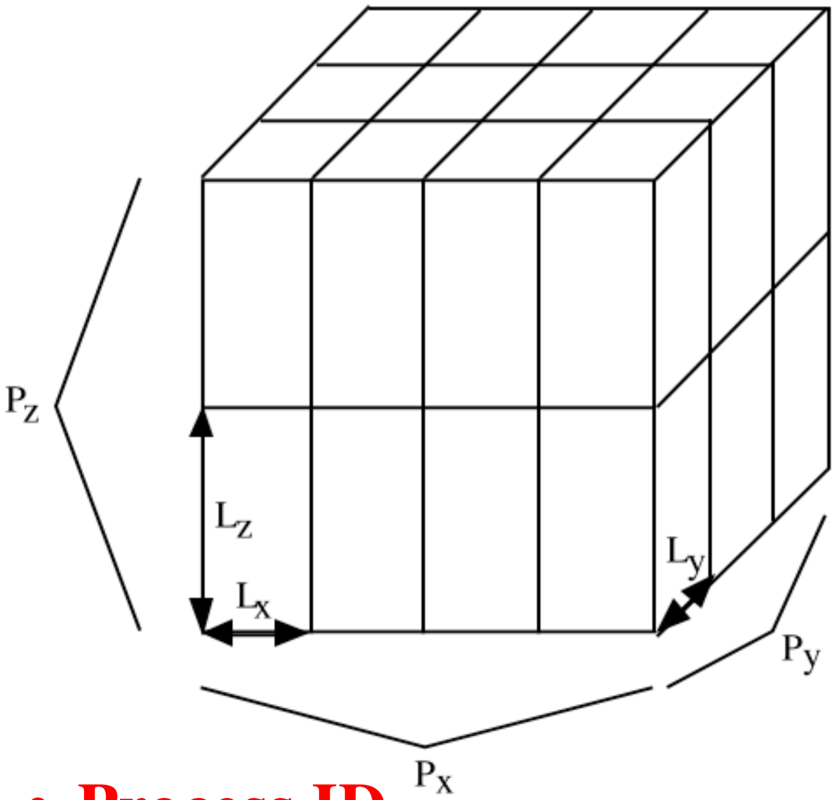




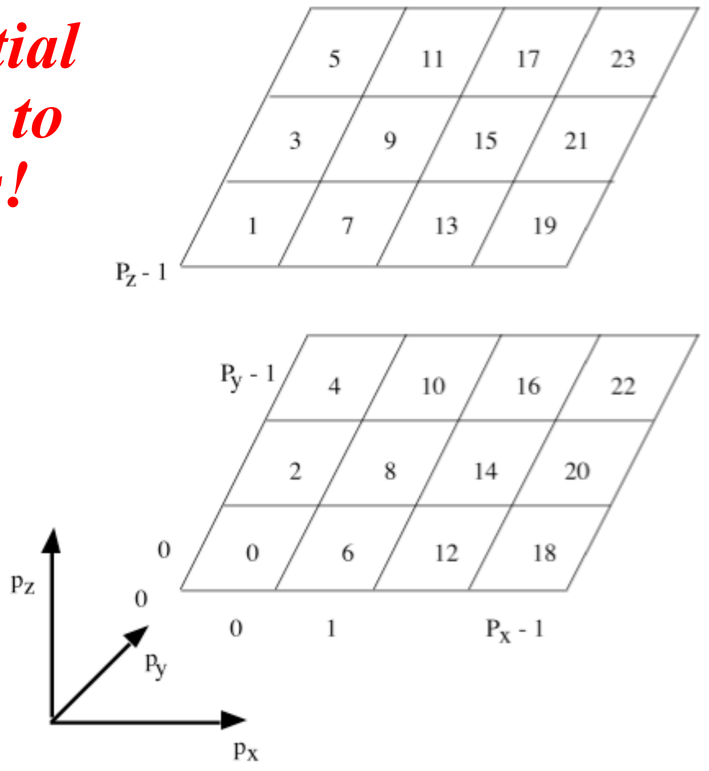
atom_move()

https://aiichironakano.github.io/cs653/src/parMD/

Spatial Decomposition



Map a spatial subsystem to a process!



- **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]

$P_x \qquad P_y \qquad P_z$

In pmd.h

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

In pmd.c

```
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

$$p'_\alpha(\kappa) = [p_\alpha + \delta_\alpha(\kappa) + P_\alpha] \bmod P_\alpha \quad (\kappa = 0,\ldots,5;\ \alpha = x, y, z)$$
$$p'(\kappa) = p'_x(\kappa) \times P_y P_z + p'_y(\kappa) \times P_z + p'_z(\kappa)$$

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)	$(0, 0, L_z)$

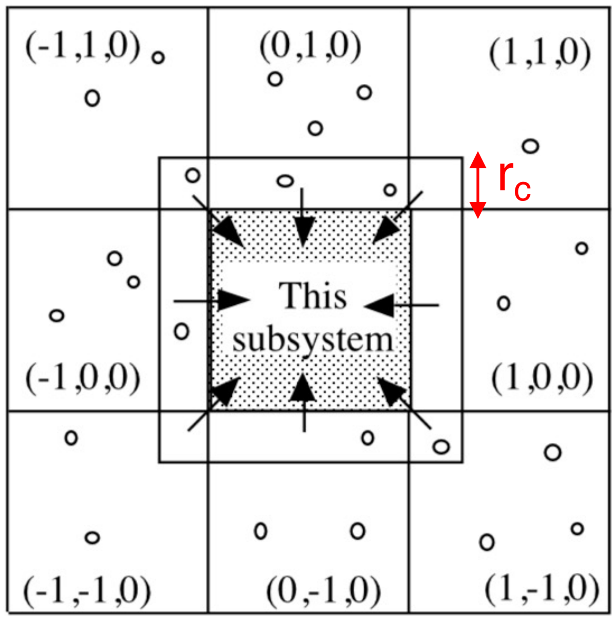
- L_x, L_y & L_z are the box lengths *per process* in the x, y & z directions
- Atom coordinates are in the range $[0, L_\alpha]$ ($\alpha = x, y, z$) in each process

In `pmd.c`

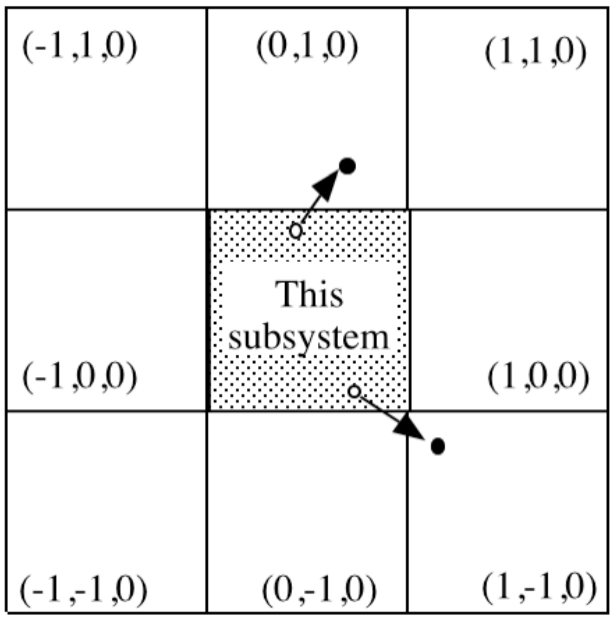
```
int iv[6][3]={{-1,0,0}, {1,0,0}, {0,-1,0}, {0,1,0}, {0,0,-1}, {0,0,1}};
...
for (ku=0; ku<6; ku++) {
  for (a=0; a<3; a++)
    k1[a] = (vid[a]+iv[ku][a]+vproc[a])%vproc[a]; Wrap around
  nn[ku] = k1[0]*vproc[1]*vproc[2]+k1[1]*vproc[2]+k1[2];destination rank
  for (a=0; a<3; a++) sv[ku][a] = al[a]*iv[ku][a]; coordinate shift for
}                                                   self-centric parallelization
```

Parallel MD Concepts

Atom caching

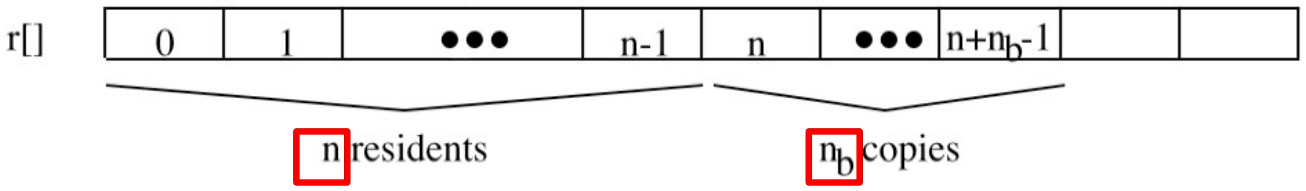


Atom migration



```
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

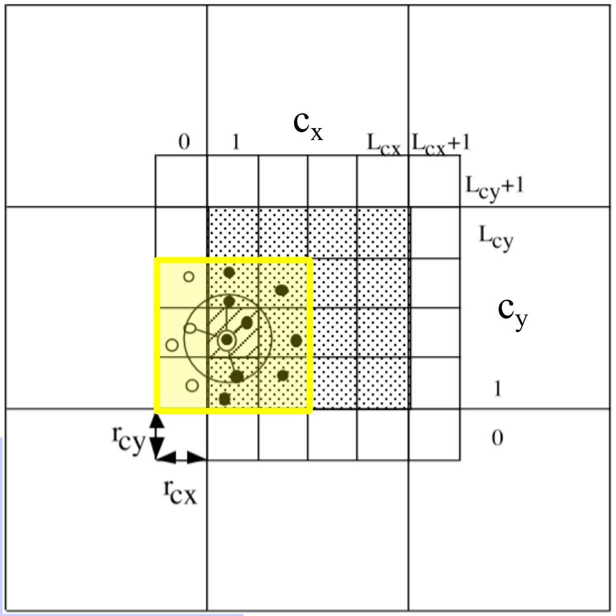


Parallel Interaction Computation

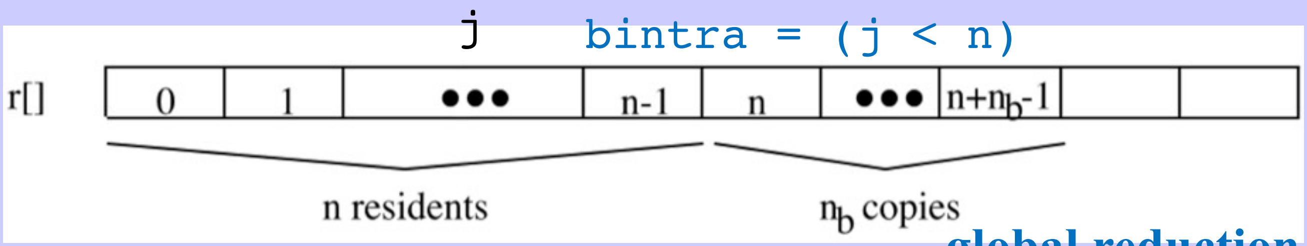
SPMD: Who does what?

Each process computes:

1. **The forces on its resident atoms** *Owner-computes rule*
2. **The potential energy between resident pairs & 1/2 of that between resident-cached pairs**

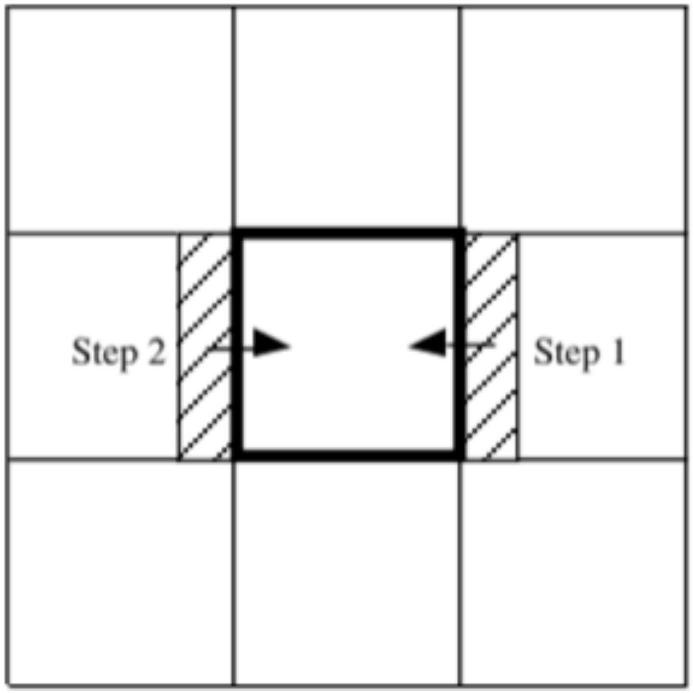


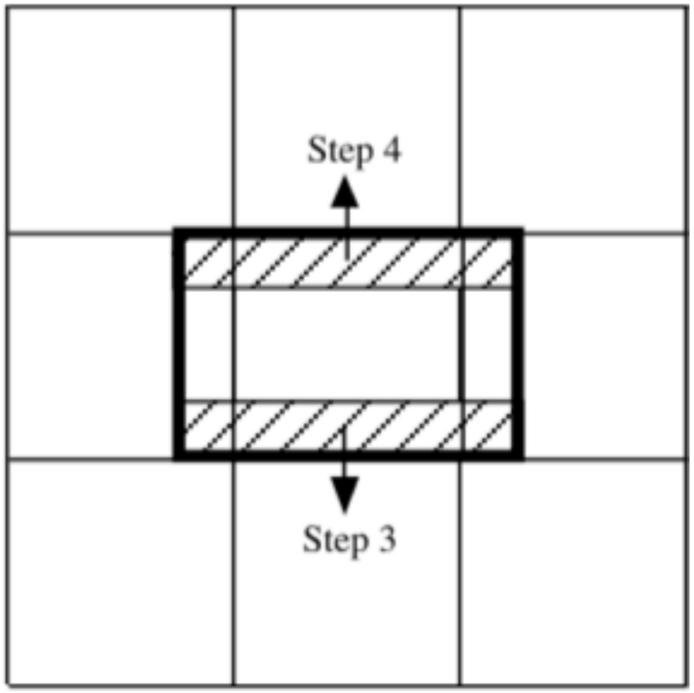
```
for resident cells, c {
  for neighbor (resident or cached) cells, c1 {
    scan atom i in cell c using c's linked list {
      scan atom j in cell c1 using c1's linked list {
        ...
        if (i<j && $r_{ij}<r_c^2$) {
          compute pair force $\mathbf{a}_{ij}$ & potential $u(r_{ij})$
          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,...);
```



global reduction over MPI ranks

Atom Caching: atom_copy()

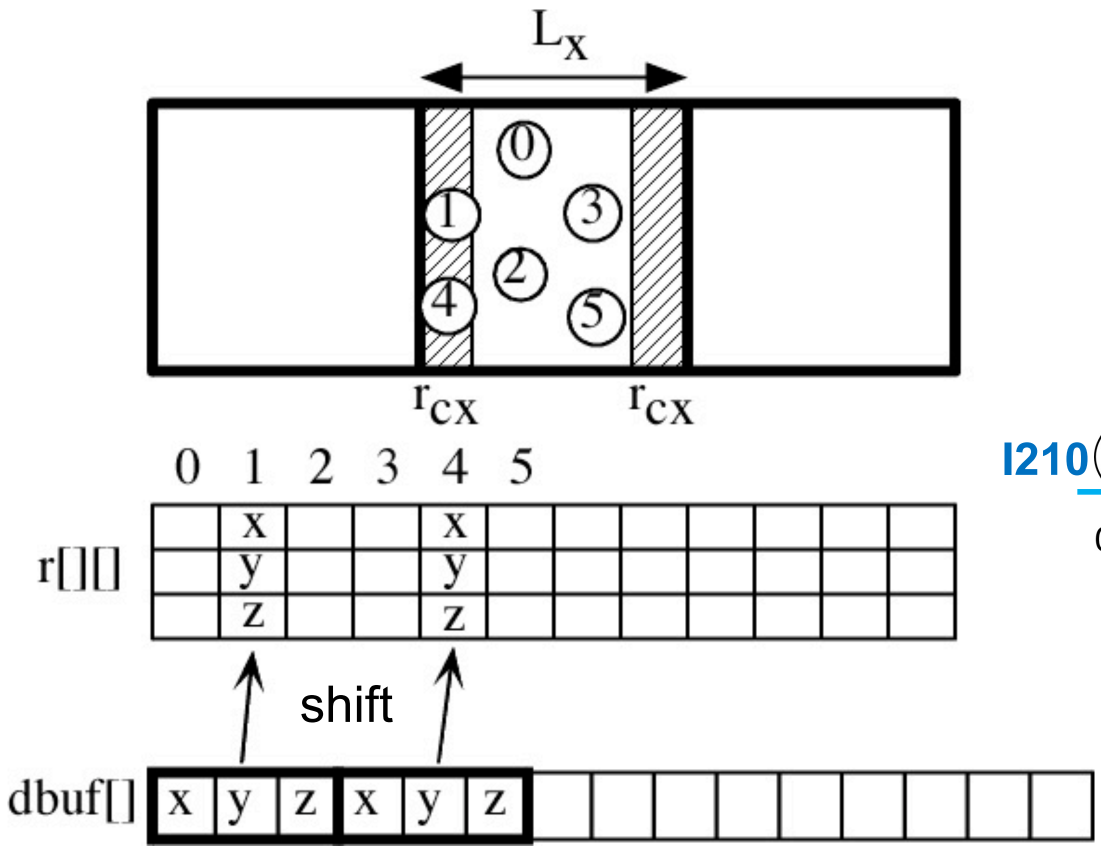


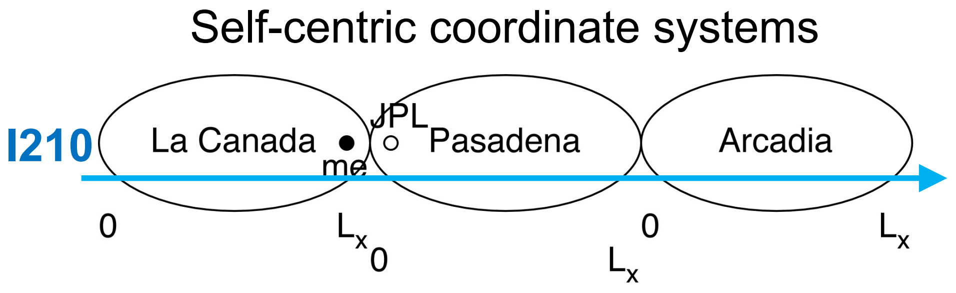


26-step → 6-step communication by message forwarding

```
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
  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

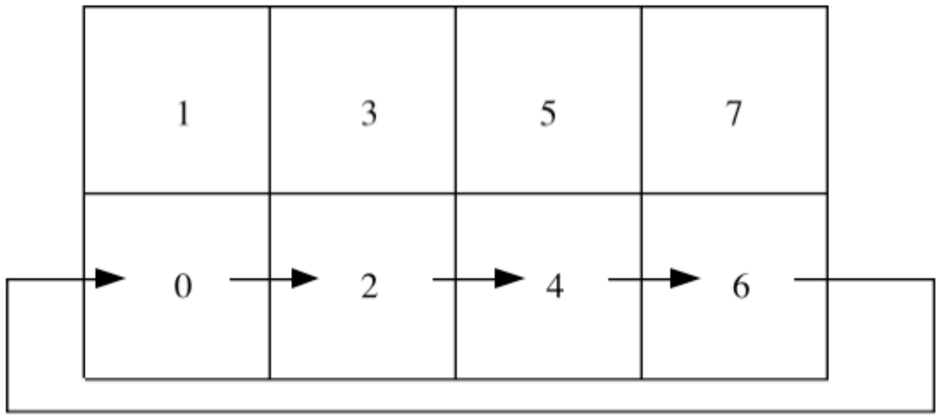
```
bbd(ri[],ku) {
  kd = ku / 2 (= 0|1|2) x|y|z
  kdd = ku % 2 (= 0|1) lower|higher
  if (kdd == 0)
    return ri[kd] < RCUT
  else
    return al[kd] – RCUT < ri[kd]
}
```

3 phases of message passing

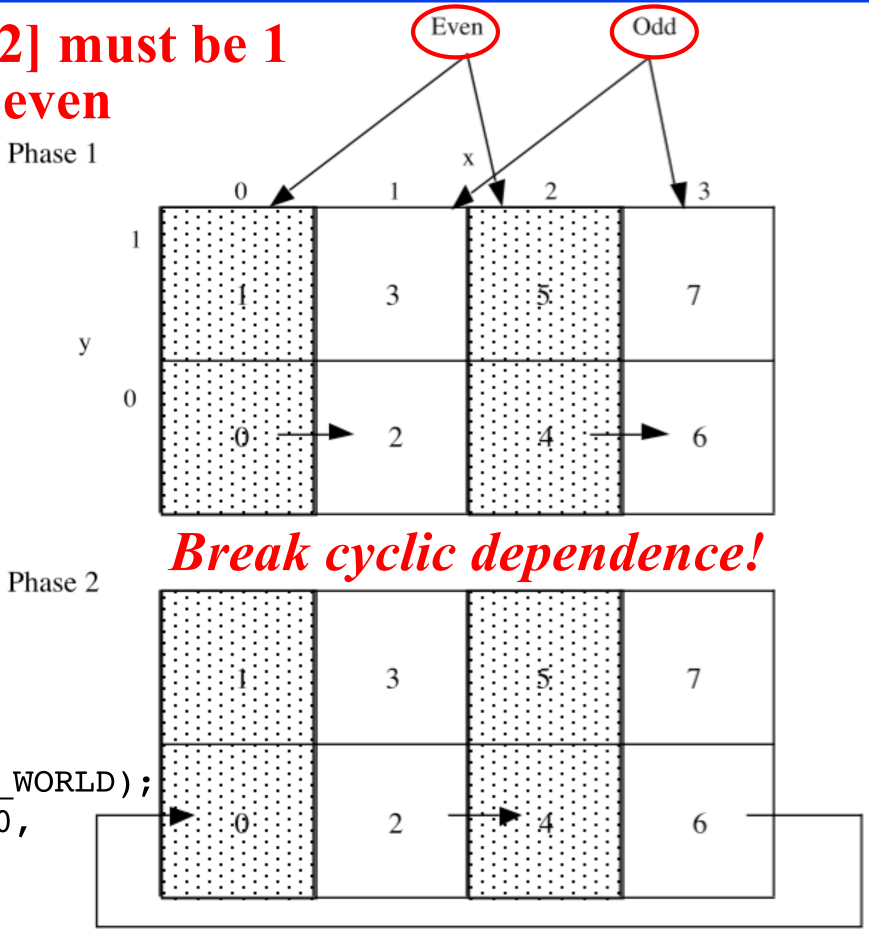
1. Message buffering: dbuf ← r–sv (shift), gather
2. Message passing: dbufr ← dbuf
   Send dbuf
   Receive dbufr
3. Message storing: r ← dbufr, append after the residents

Deadlock Avoidance

Cyclic dependence

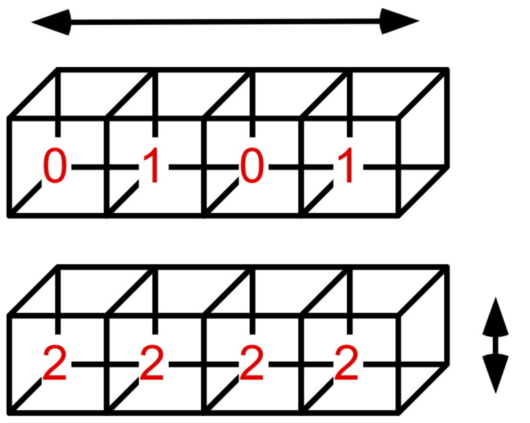


vproc[0|1|2] must be 1 or even

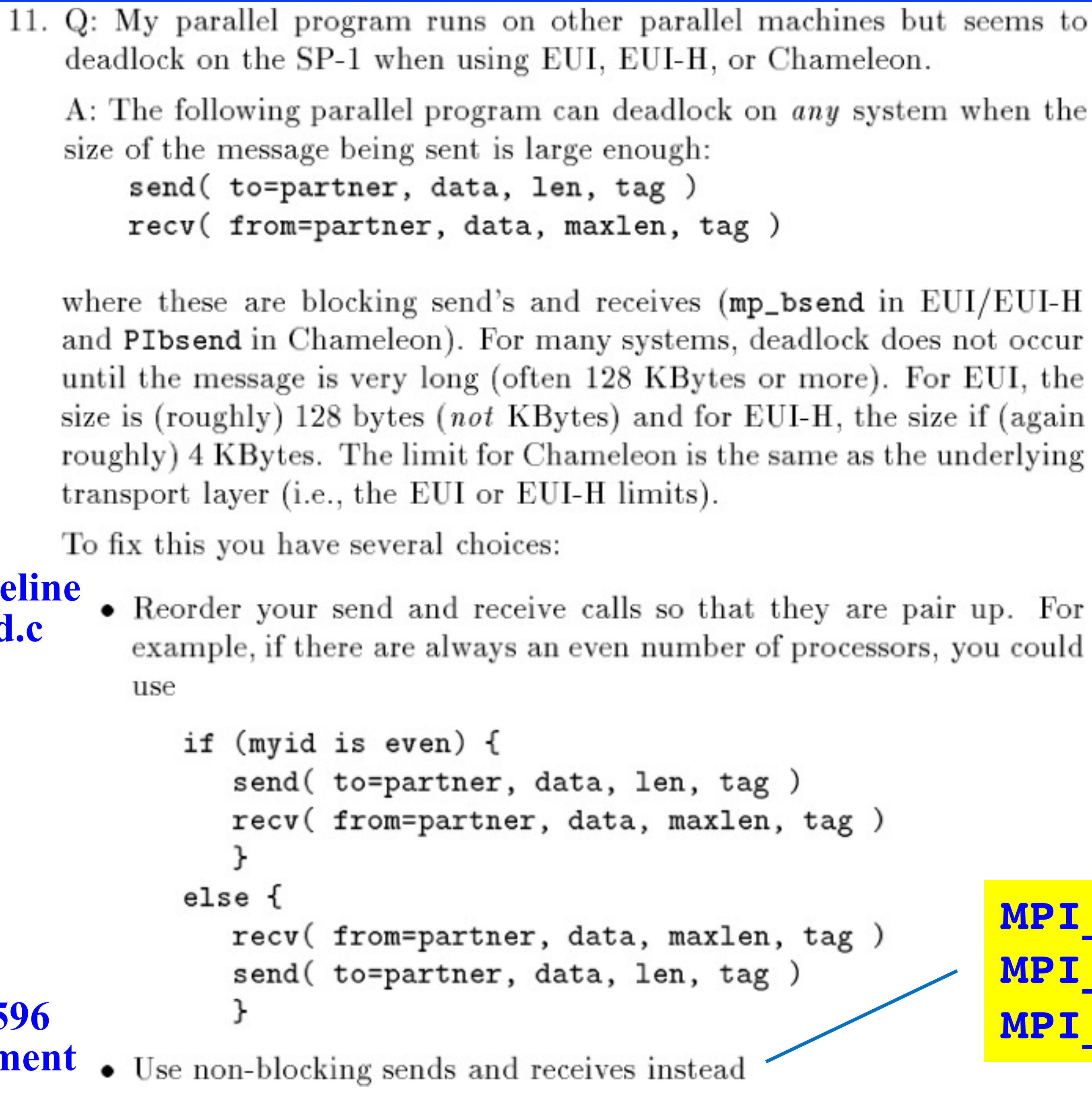


3-phase (deadlock-free) message passing

```
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
```



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:

Baseline 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

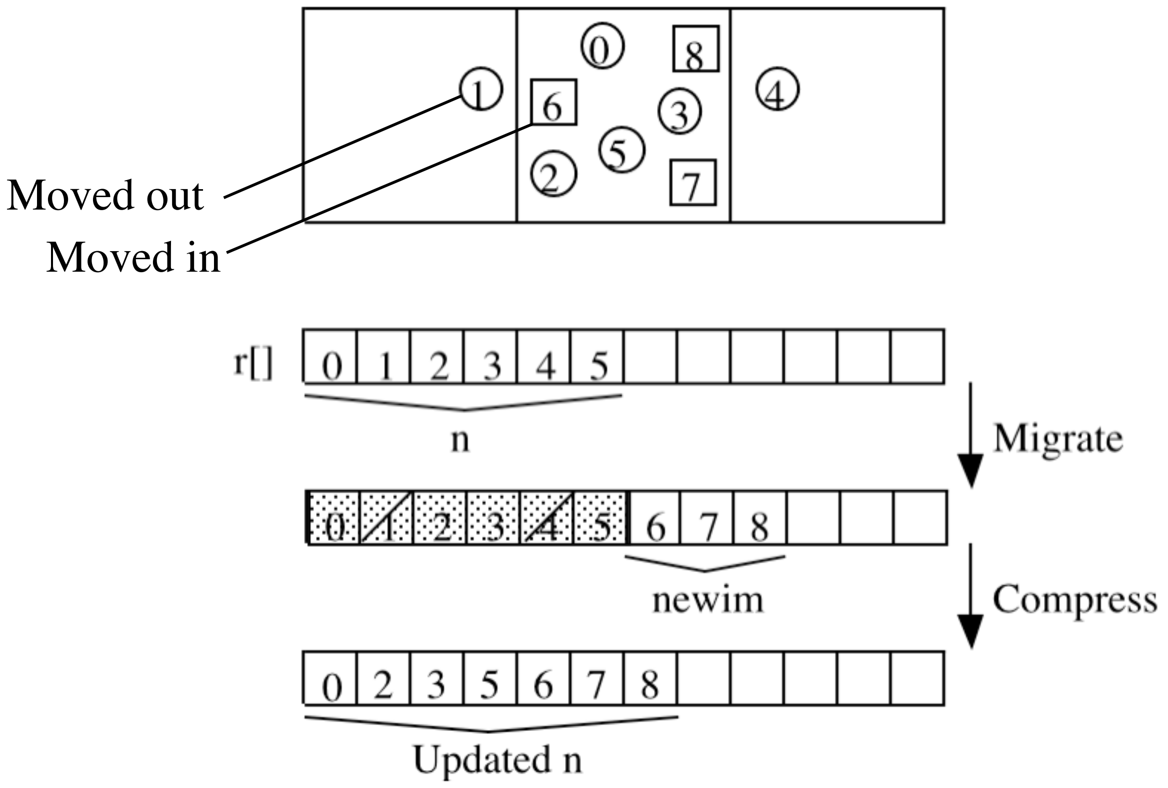
```
    if (myid is even) {
        send( to=partner, data, len, tag )
        recv( from=partner, data, maxlen, tag )
        }
    else {
        recv( from=partner, data, maxlen, tag )
        send( to=partner, data, len, tag )
        }
```

CSCI 596 assignment

- Use non-blocking sends and receives instead

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

Atom Migration: `atom_move()`

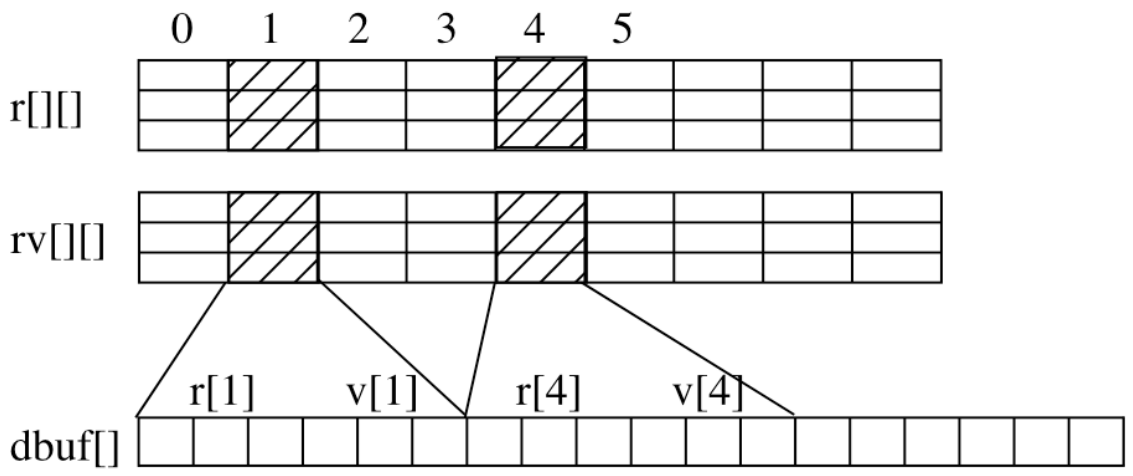


```
Reset the number of received new immigrants, newim = 0
for x, y, and z directions
  Make moving-atom lists, mvque, for lower and higher directions including both
  resident, n, and immigrant, newim, atoms but excluding those already moved out
  for lower and higher directions
    Send/receive moving-atom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

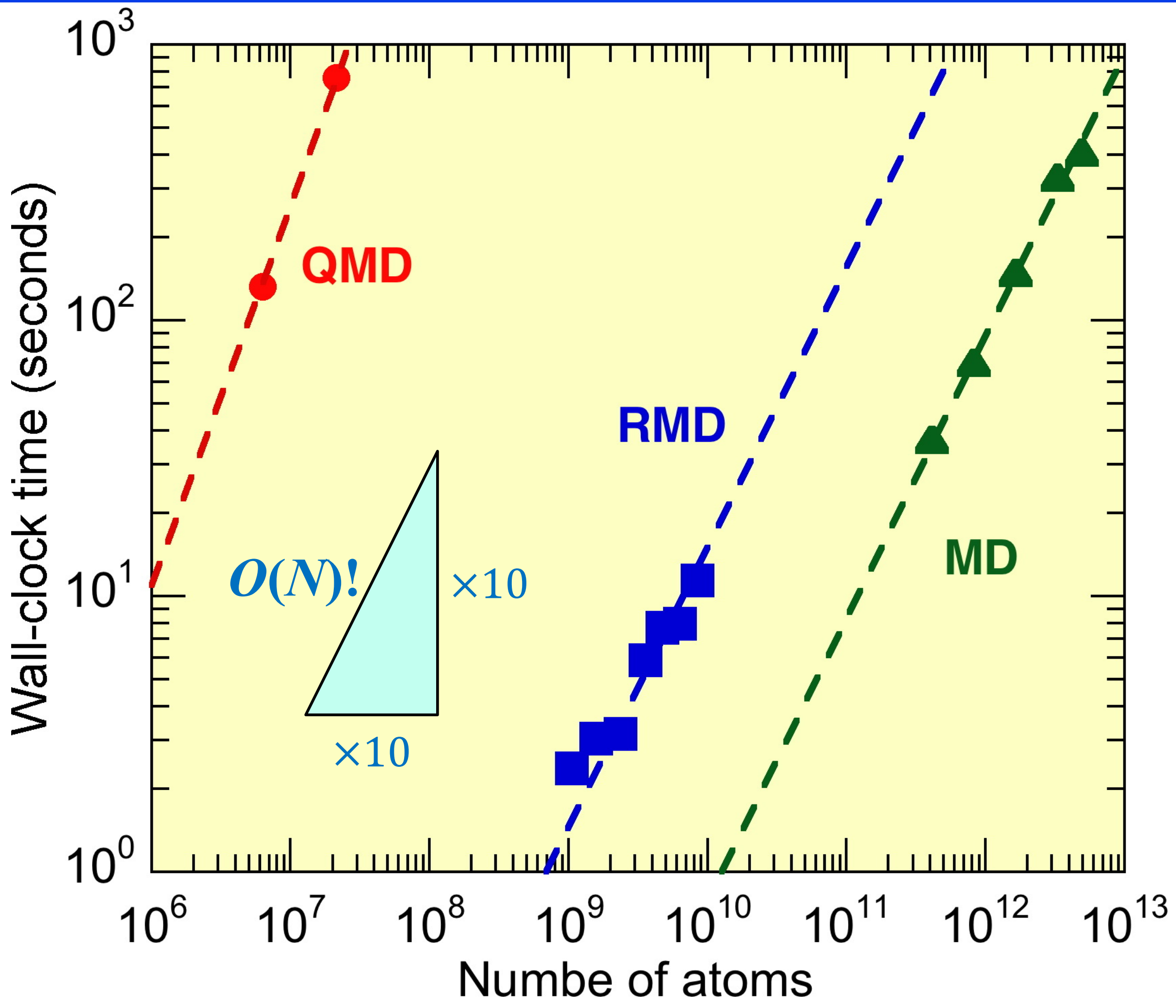
```
bmv(ri[],ku) {
  kd = ku / 2 (= 0|1|2)
  kdd = ku % 2 (= 0|1)
  if (kdd == 0)
    return ri[kd] < 0.0
  else
    return al[kd] < ri[kd]
}
```

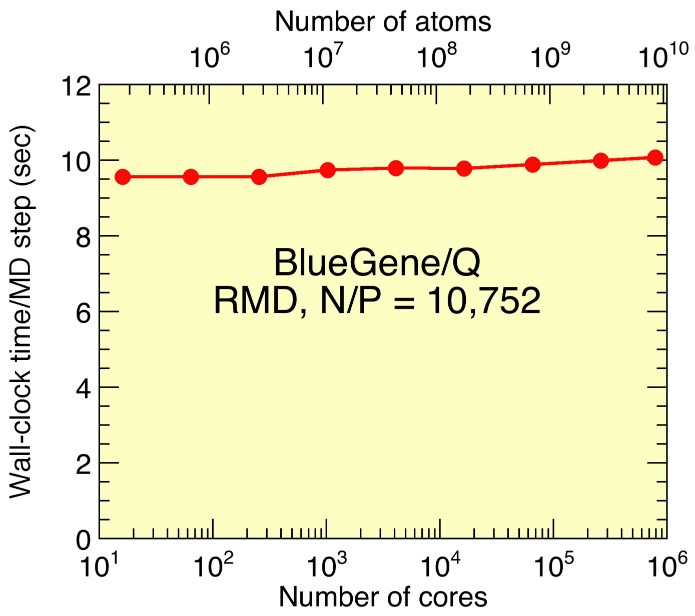


3 phases of message passing

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**

Spatial Decomposition Benchmark





QMD (quantum molecular dynamics): DC-DFT

RMD (reactive molecular dynamics): F-ReaxFF

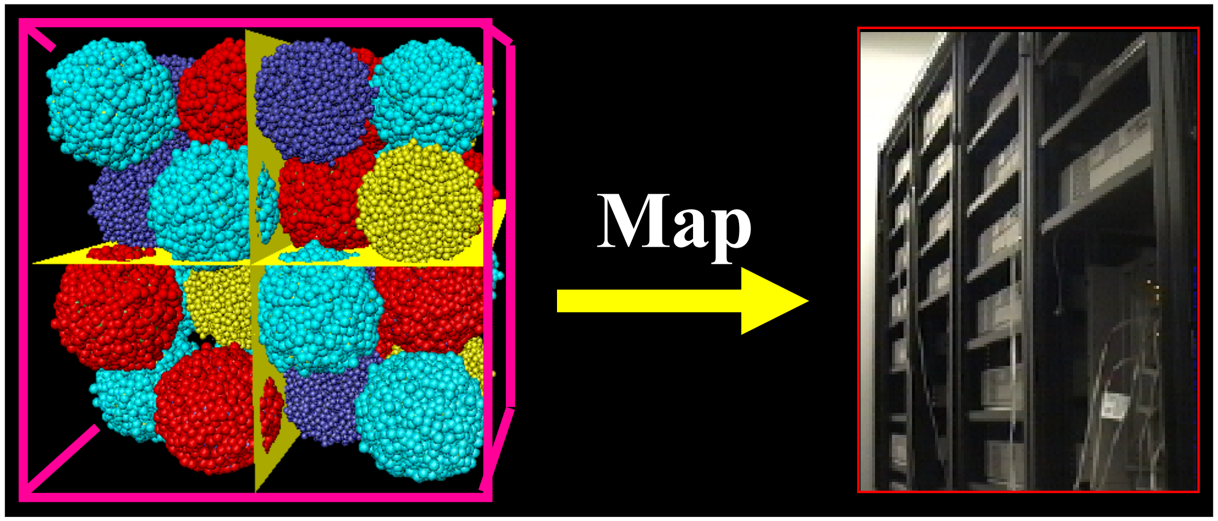
MD (molecular dynamics): MRMD

- **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**
- **1.9 trillion grid points (21.2 million-atom) DC-DFT QMD of SiC**

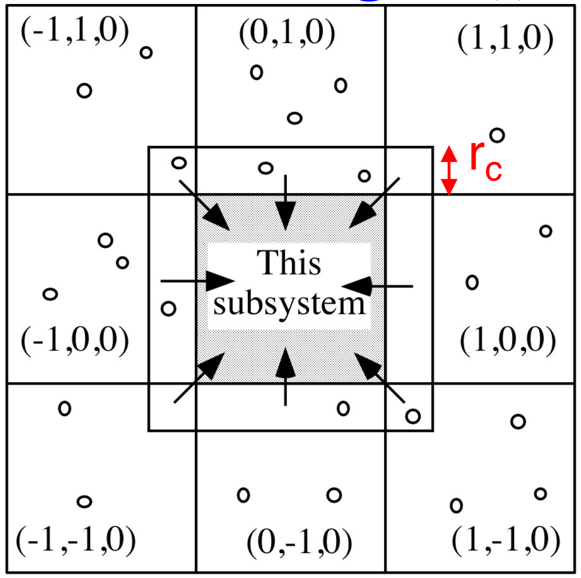
parallel efficiency 0.98 on 786,432 BlueGene/Q cores

Cost of Spatial Decomposition MD

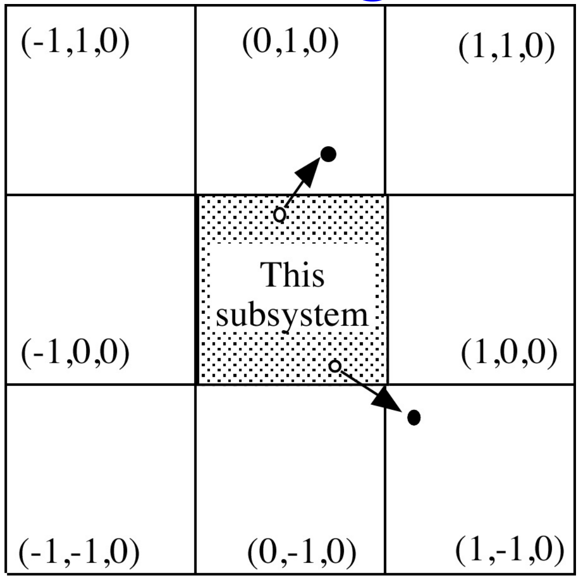
Spatial decomposition (short ranged): $O(N/P)$ computation



Atom caching: $O((N/P)^{2/3})$



Atom migration



Large overhead & lack of parallelism for small N/P

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

See Grama'03, Chap. 5

- **Efficiency:** $E_P = \dfrac{S_P}{P} = \dfrac{W_P T(W_1,1)}{P W_1 T(W_P,P)}$
- **How to scale W_P with P?**
  - > **Isogranular (weak) scaling:**
    $W_P = Pw$; w **= constant workload per processor (granularity)**
  - > **Constant problem-size (strong) scaling:**
    $W_P = W$ **—constant**

Analysis of Parallel MD

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

MPI_Allreduce()

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

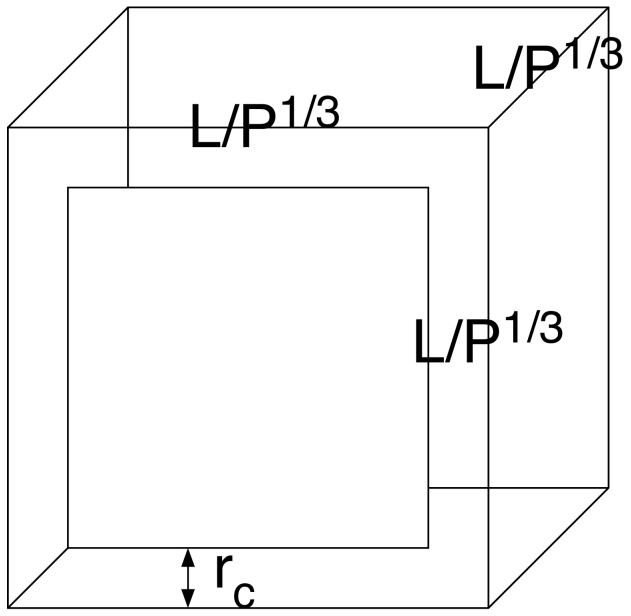
$$\underbrace{6}_{\text{facets}}\ \underbrace{\frac{L^2}{P^{2/3}}r_c}_{\text{cached volume}}\ \underbrace{\rho}_{\text{atom density}}$$

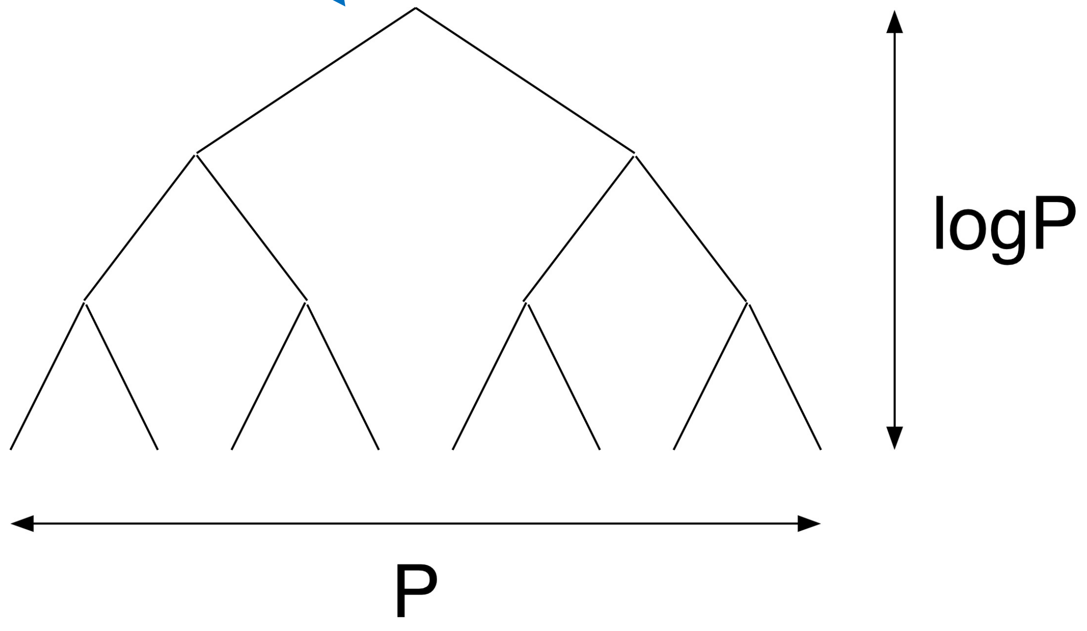
$$= 6r_c\frac{N^{2/3}/\rho^{2/3}}{P^{2/3}}\rho$$

$$= 6r_c\rho^{1/3}\left(\frac{N}{P}\right)^{2/3}$$

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

Eliminate *L* by expressing it in terms of *N*





Fixed Problem-Size Scaling

- **Speedup:**

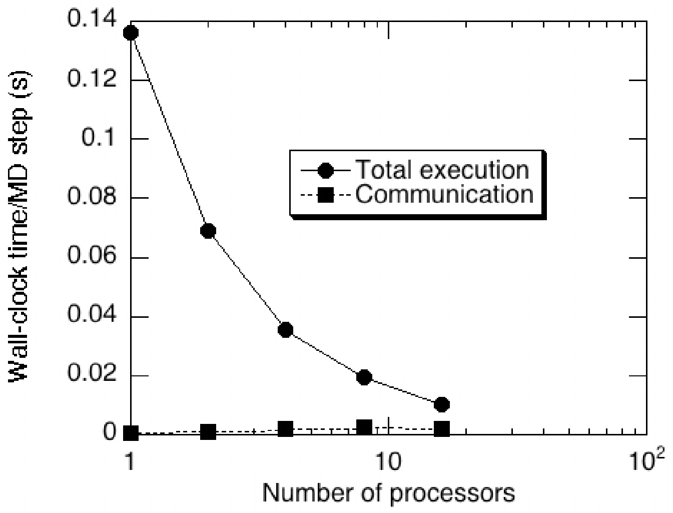
$$S_P = \frac{T(N,1)}{T(N,P)} = \frac{aN}{aN/P + b(N/P)^{2/3} + c\log P}$$

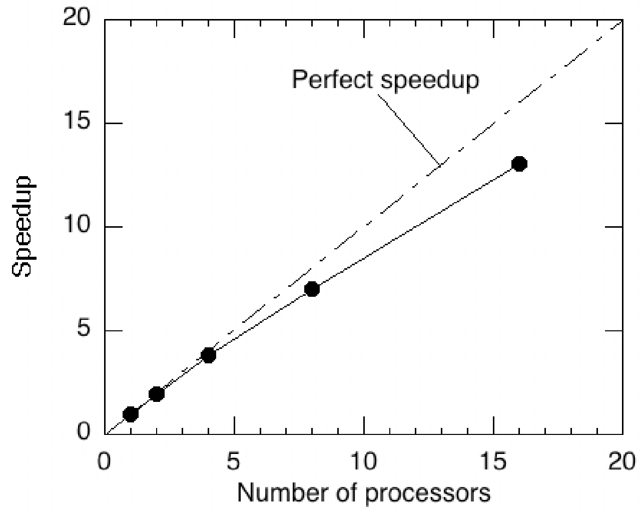
$$= \frac{P}{1 + \frac{b}{a}\left(\frac{P}{N}\right)^{1/3} + \frac{c}{a}\frac{P\log P}{N}}$$

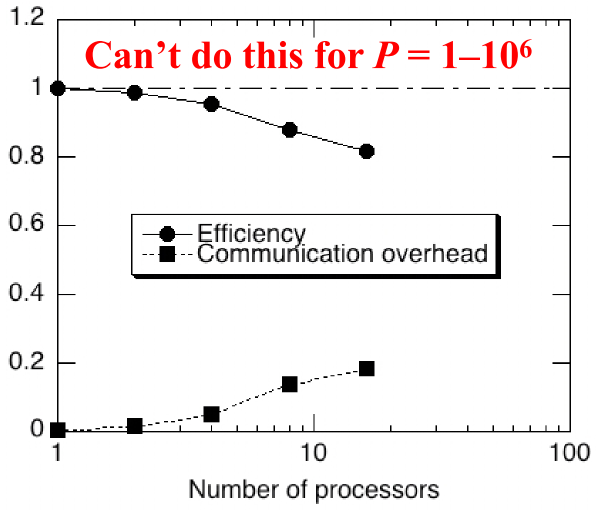
$$S_P = \frac{S(W_P,P)}{S(W_1,1)} = \frac{\cancel{W_P}T(W_1,1)}{\cancel{W_1}T(W_P,P)}$$

- **Efficiency:**

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







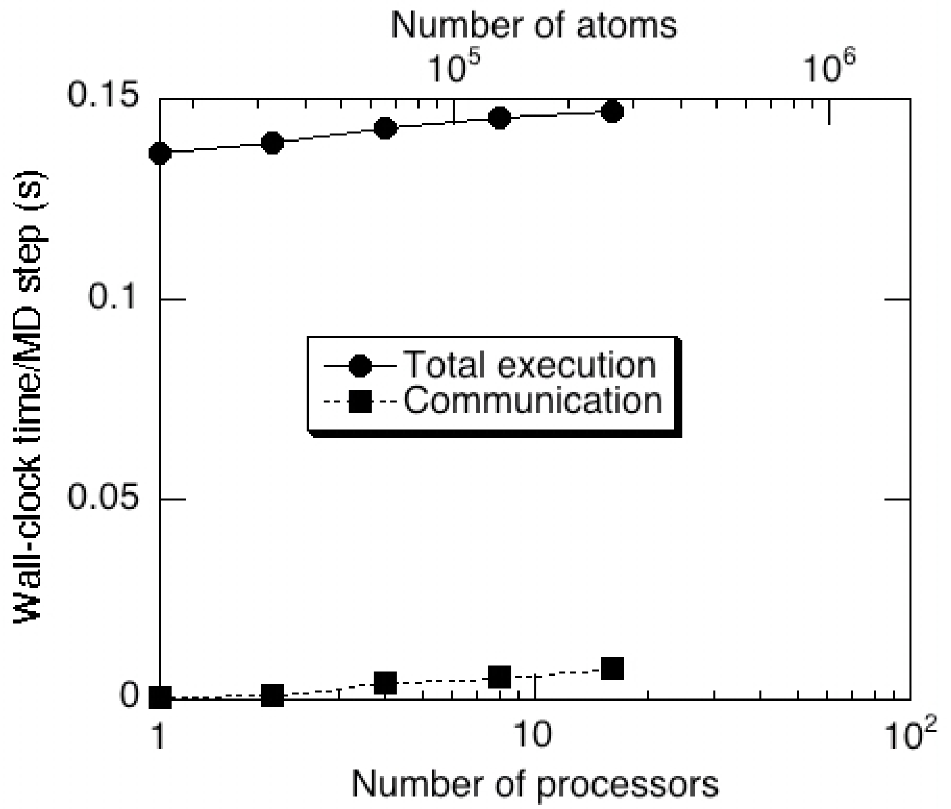
pmd.c: *N* = 16,384, on CARC

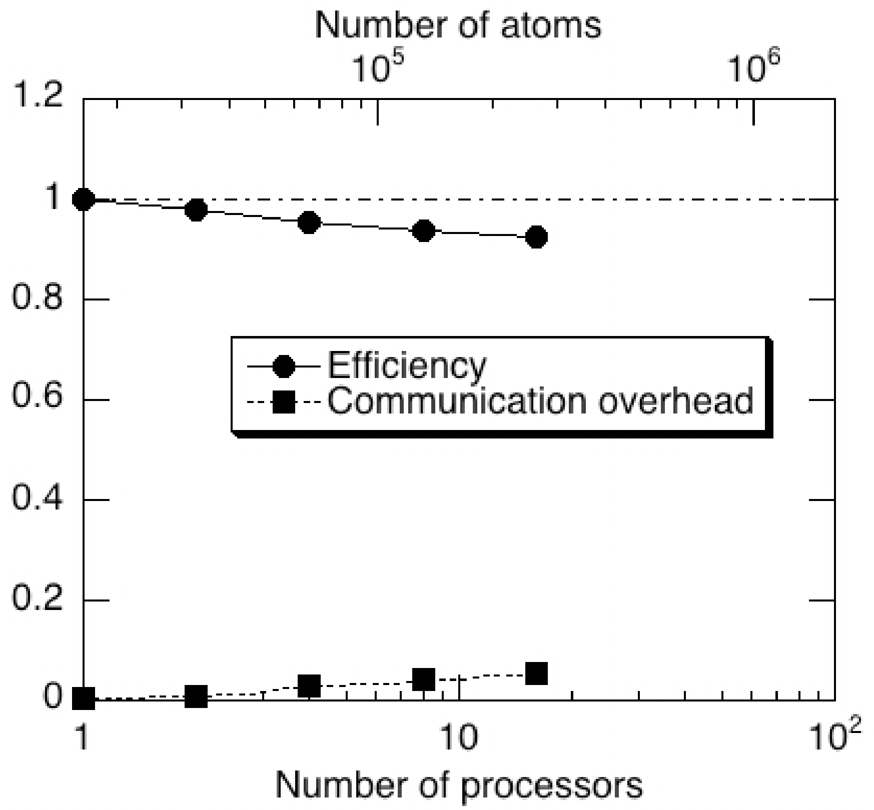
Isogranular Scaling of Parallel MD

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

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

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

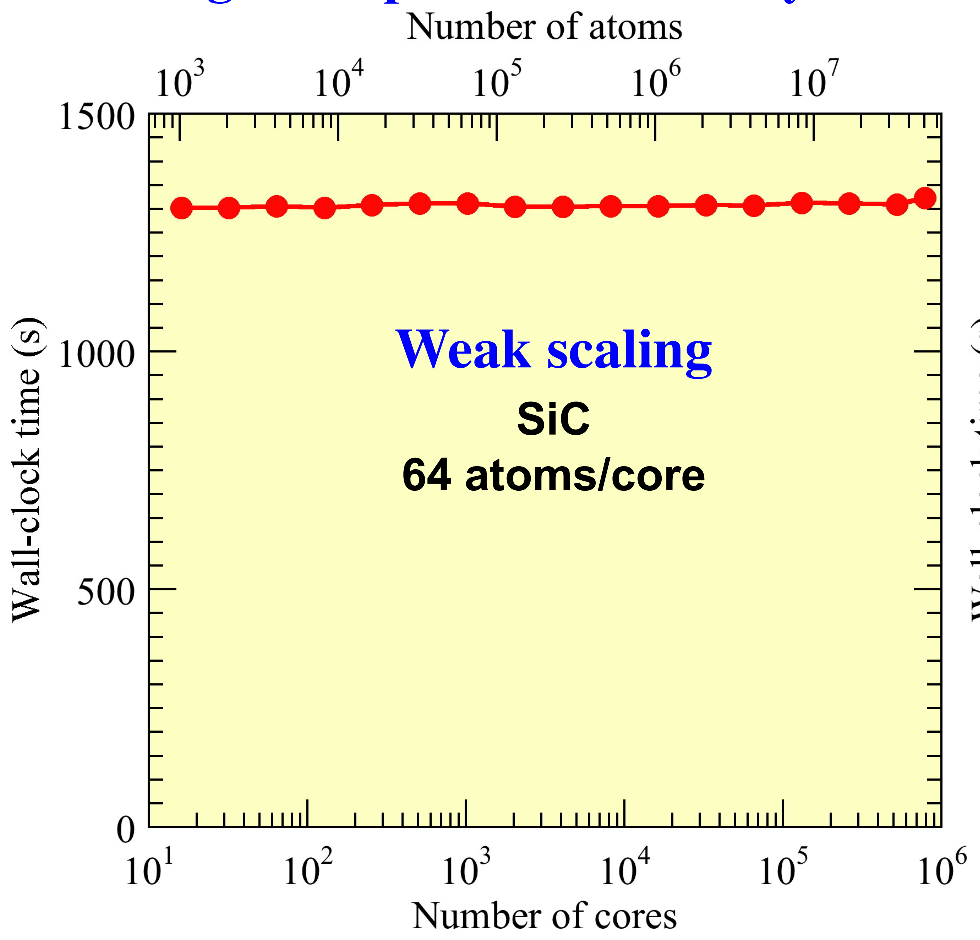


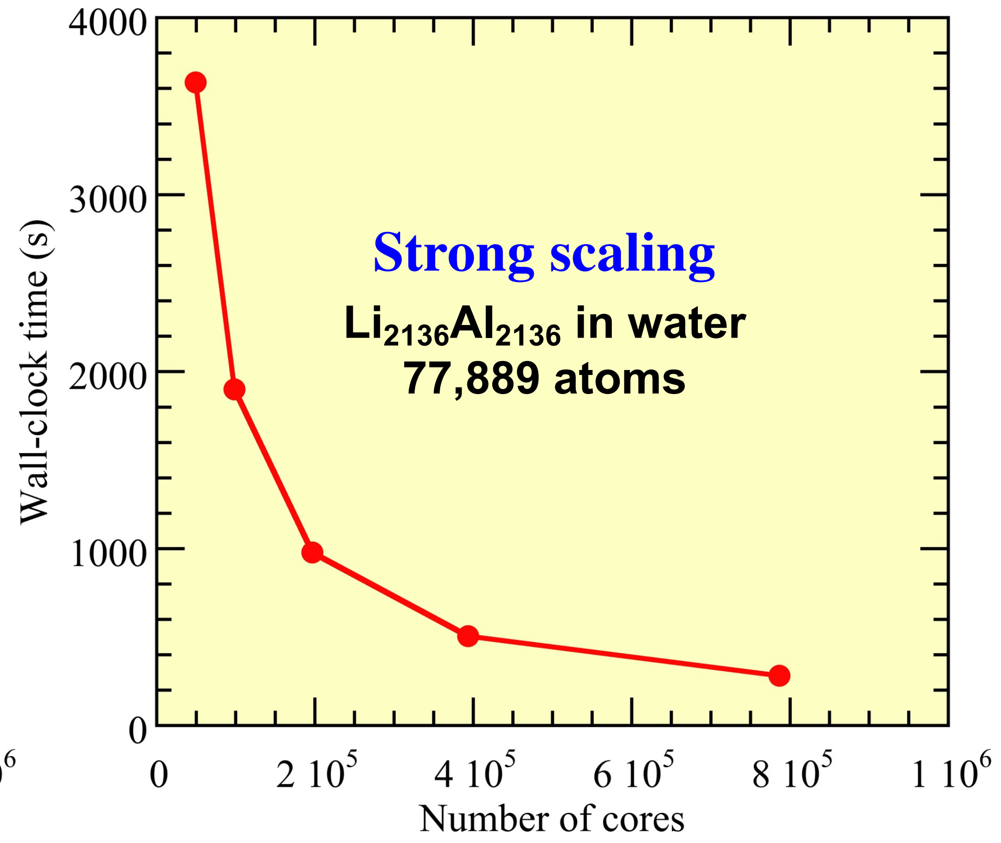


pmd.c: *N*/*P* = 16,384, on CARC

Parallel Performance of Quantum MD

- **Weak-scaling parallel efficiency is 0.984 on 786,432 Blue Gene/Q cores for a 50,331,648-atom SiC system**
- **Strong-scale parallel efficiency is 0.803 on 786,432 Blue Gene/Q cores**

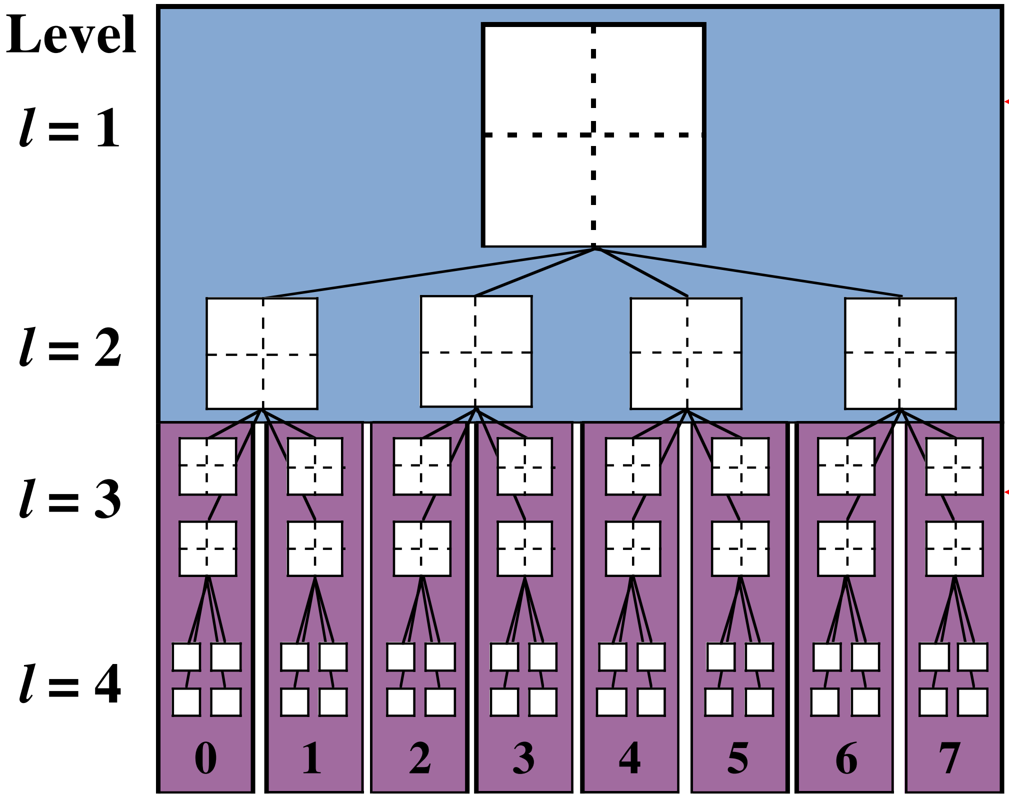




- **62-fold reduction of time-to-solution** [441 s/SCF-step for 50.3M atoms] **from the previous state-of-the-art** [55 s/SCF-step for 102K atoms, Osei-Kuffuor *et al.*, *PRL* ’14]

K. Nomura *et al.*, *IEEE/ACM Supercomputing, SC14* (’14)

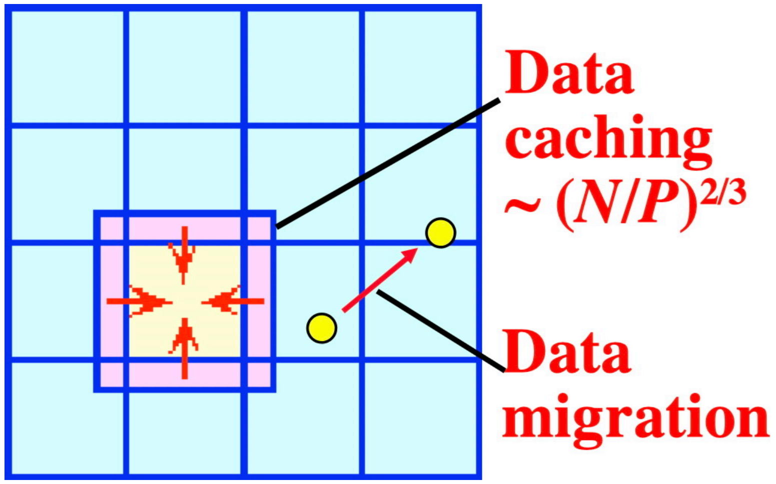
Parallel Fast Multipole Method



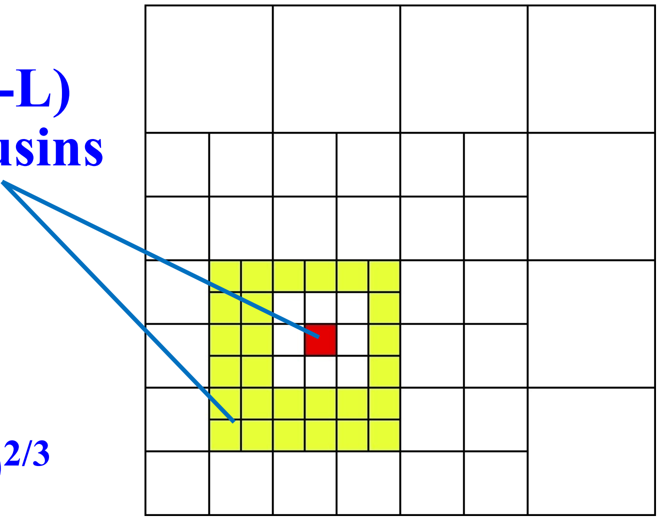
Upper levels:
Global to all processors
Overhead: $O(\log P)$

A. Nakano *et al.*, Comput. Phys.Commun. **83**, 197 (1994)

Lower levels:
Spatial decomposition
Computation: $O(N/P)$



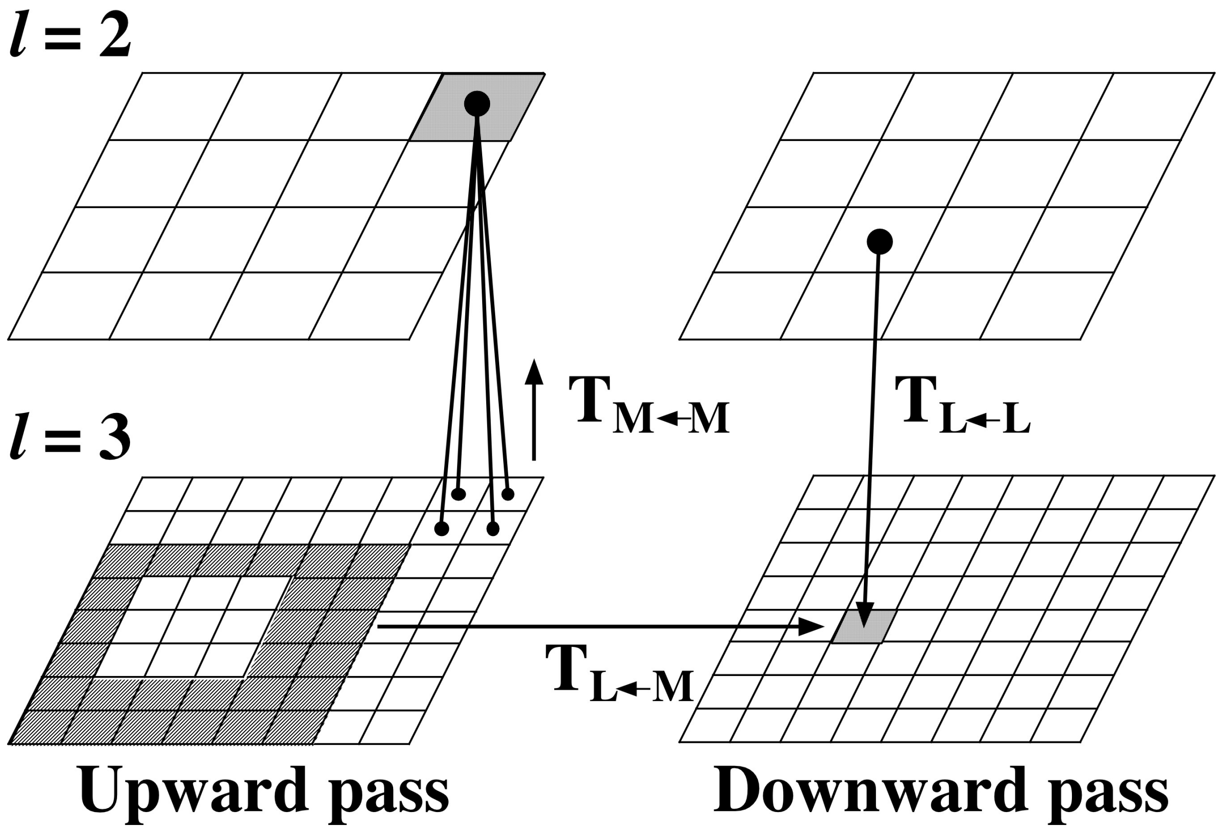
Level-by-level short-ranged (M-to-L) interaction with cousins

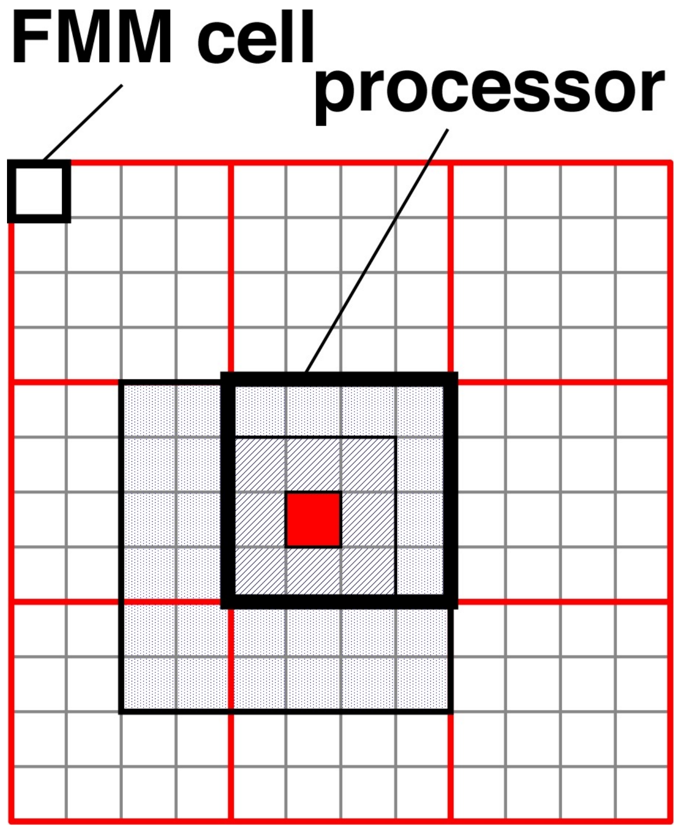


Coarse grain:
$N/P \sim 10^6$; $P \leq 10^3$
$\downarrow$
$N/P \gg \log P, (N/P)^{2/3}$

S. Ogata *et al.*, *Comput. Phys. Commun.* **153**, 445 ('03)

Caching Interactive Cells





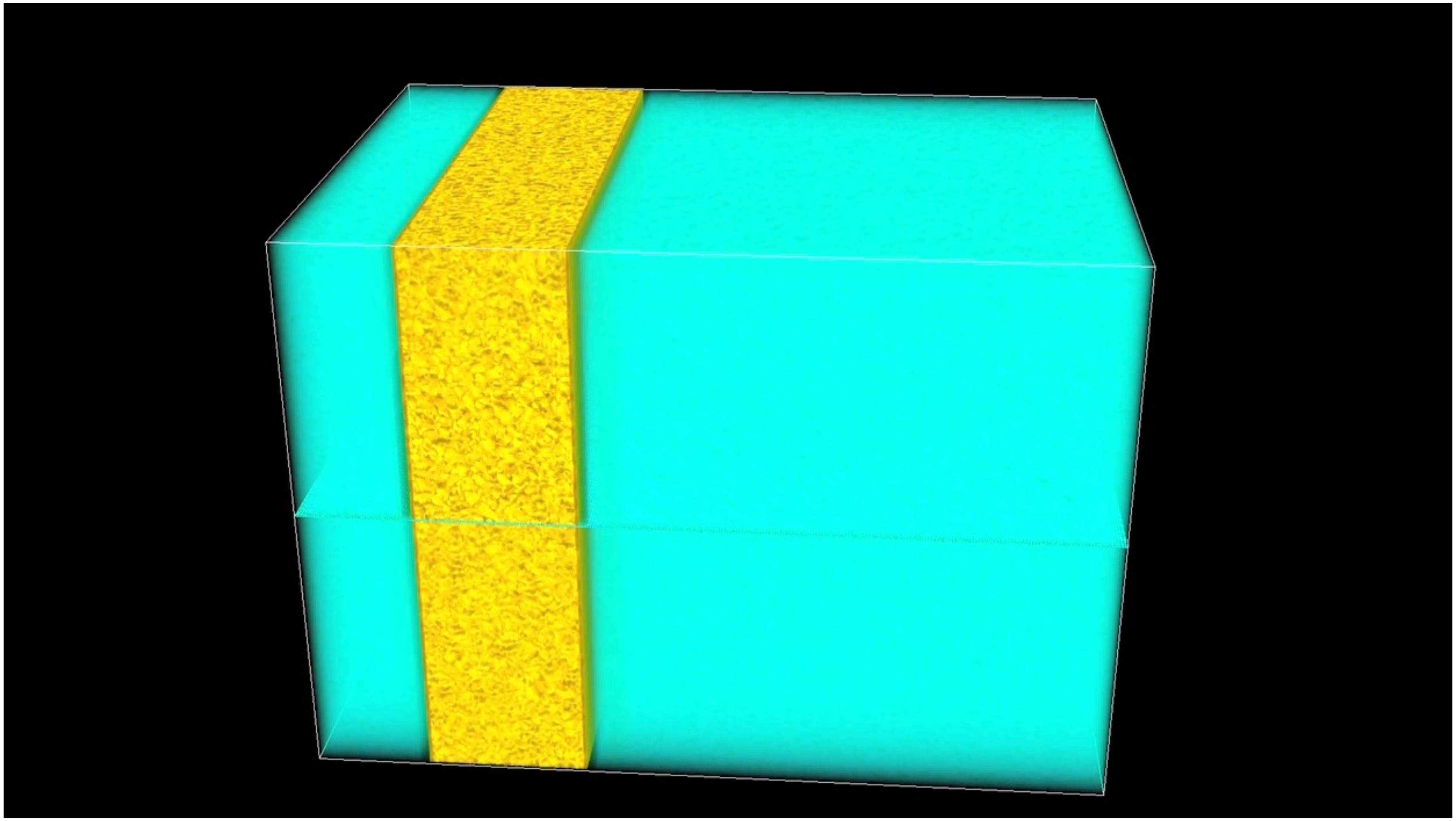
- $T_{M\leftarrow M}$ & $T_{L\leftarrow L}$: local at lower octree levels
- $T_{L\leftarrow M}$: cache 2 boundary layers of cells at each level

See lecture note on "scalability analysis of parallel molecular-dynamics & fast-multipole-method algorithms"

https://aiichironakano.github.io/cs653/02-2Scalability.pdf

Billion-Atom Molecular Dynamics

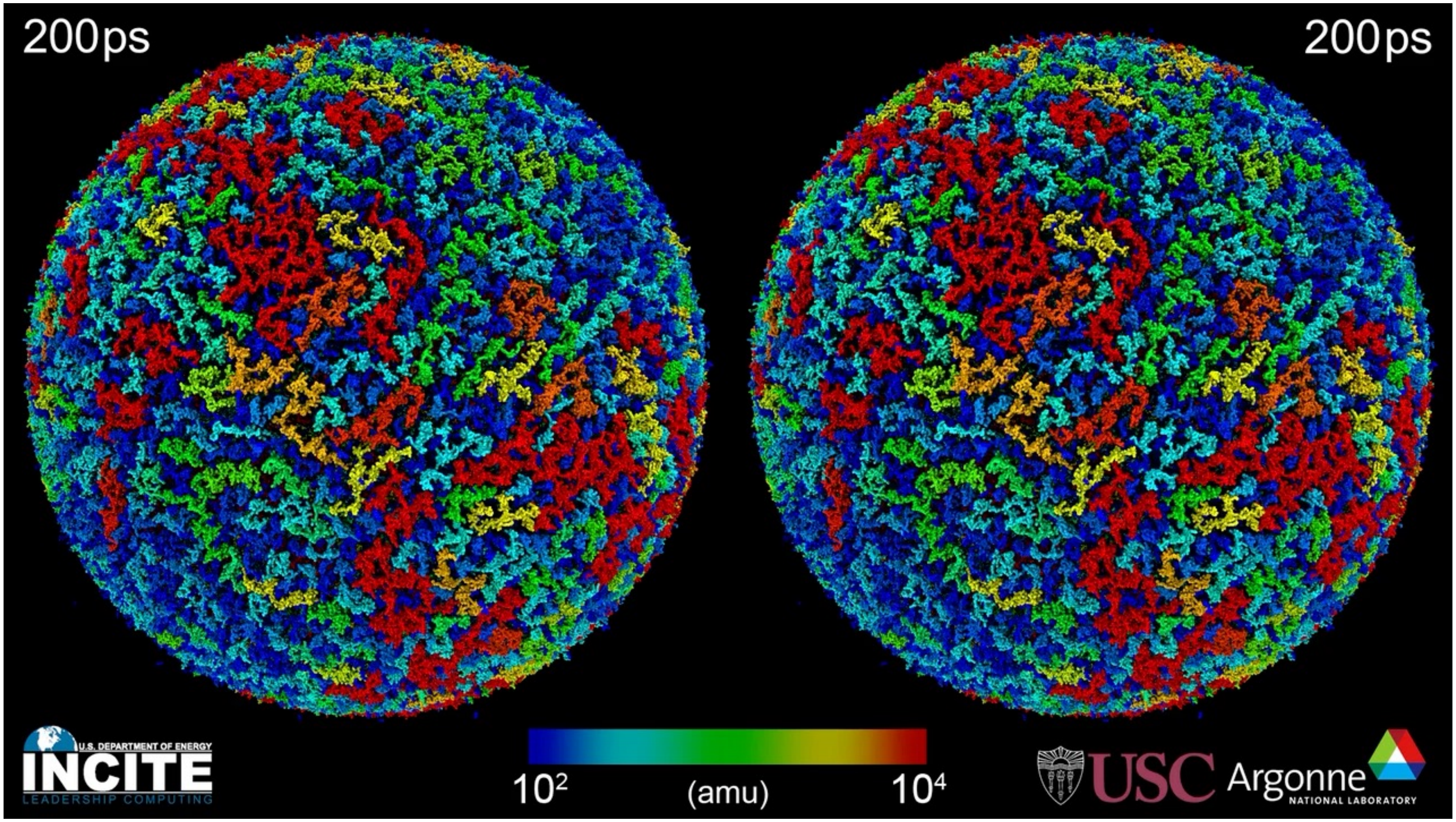
- **Billion-atom MD simulation of shock-induced nanobubble collapse in water near silica surface (67 million core-hours on 163,840 Blue Gene/P cores)**



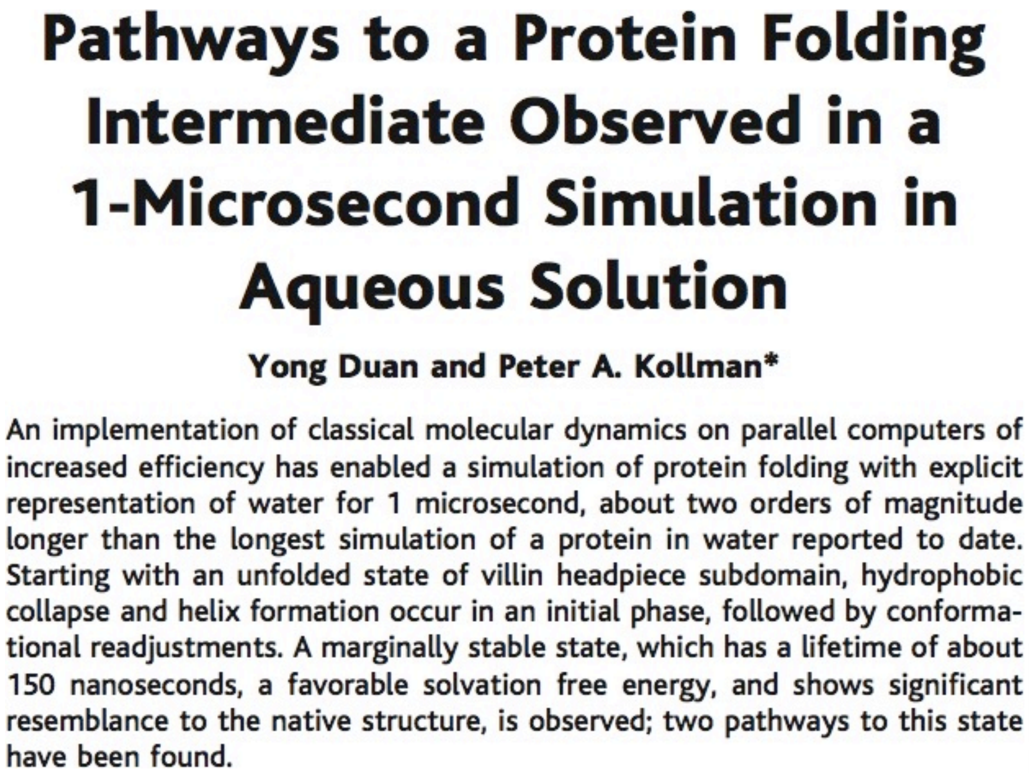
- **Water nanojet formation and its collision with silica surface**

112 Million-Atom Reactive MD

- 112 million-atom reactive MD simulation to study nanocarbon synthesis by high-temperature oxidation of SiC nanoparticle (410 million core-hours on 786,432 Blue Gene/Q cores)



Fine-Grained Parallel MD

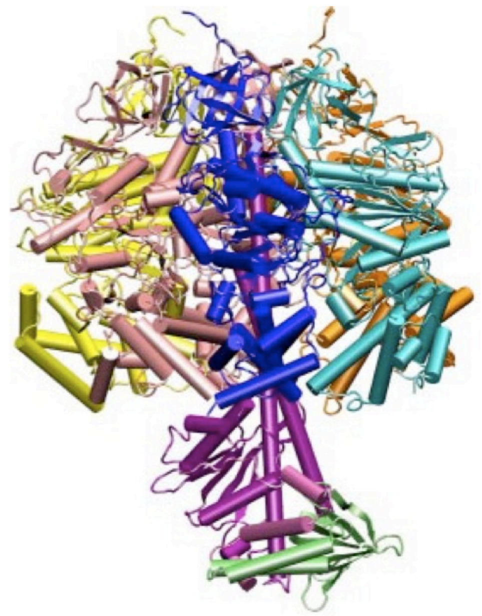


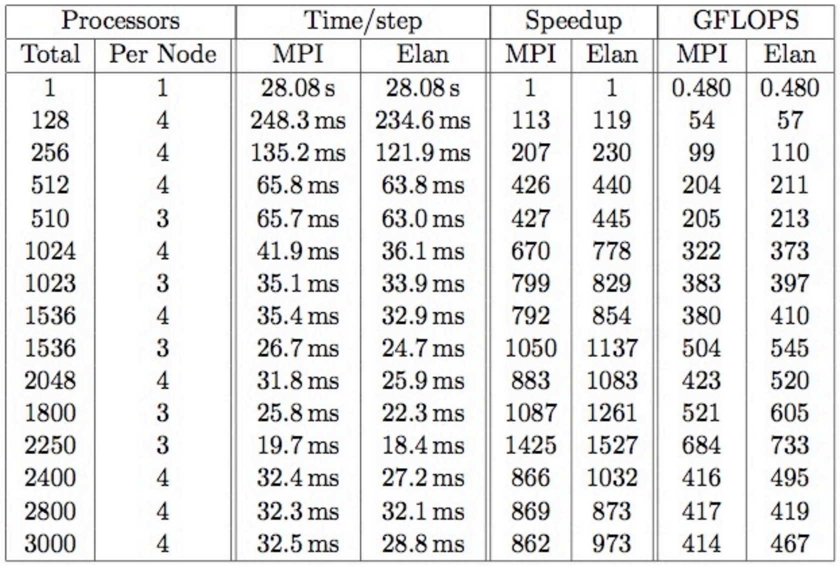
Pathways to a Protein Folding Intermediate Observed in a 1-Microsecond Simulation in Aqueous Solution

Yong Duan and Peter A. Kollman*

An implementation of classical molecular dynamics on parallel computers of increased efficiency has enabled a simulation of protein folding with explicit representation of water for 1 microsecond, about two orders of magnitude longer than the longest simulation of a protein in water reported to date. Starting with an unfolded state of villin headpiece subdomain, hydrophobic collapse and helix formation occur in an initial phase, followed by conformational readjustments. A marginally stable state, which has a lifetime of about 150 nanoseconds, a favorable solvation free energy, and shows significant resemblance to the native structure, is observed; two pathways to this state have been found.

Science **282**, 740 ('98)





Processors		Time/step		Speedup		GFLOPS	
Total	Per Node	MPI	Elan	MPI	Elan	MPI	Elan
1	1	28.08 s	28.08 s	1	1	0.480	0.480
128	4	248.3 ms	234.6 ms	113	119	54	57
256	4	135.2 ms	121.9 ms	207	230	99	110
512	4	65.8 ms	63.8 ms	426	440	204	211
510	3	65.7 ms	63.0 ms	427	445	205	213
1024	4	41.9 ms	36.1 ms	670	778	322	373
1023	3	35.1 ms	33.9 ms	799	829	383	397
1536	4	35.4 ms	32.9 ms	792	854	380	410
1536	3	26.7 ms	24.7 ms	1050	1137	504	545
2048	4	31.8 ms	25.9 ms	883	1083	423	520
1800	3	25.8 ms	22.3 ms	1087	1261	521	605
2250	3	19.7 ms	18.4 ms	1425	1527	684	733
2400	4	32.4 ms	27.2 ms	866	1032	416	495
2800	4	32.3 ms	32.1 ms	869	873	417	419
3000	4	32.5 ms	28.8 ms	862	973	414	467

Table 1: NAMD performance on 327K atom ATPase benchmark system with and multiple timestepping with PME every four steps for Charm++ based on MPI and Elan.

J.C. Phillips, G. Zheng, S. Kumar, & L.V. Kale, in *Proc. of IEEE/ACM SC2002*

Force Decomposition for Parallel MD

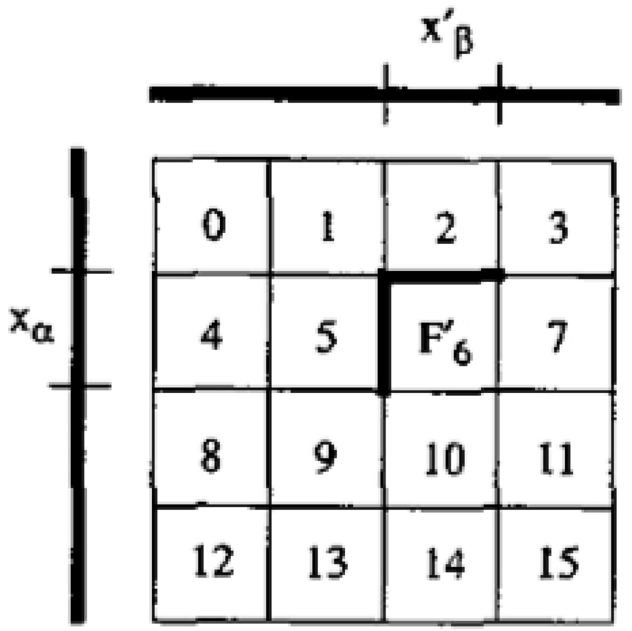
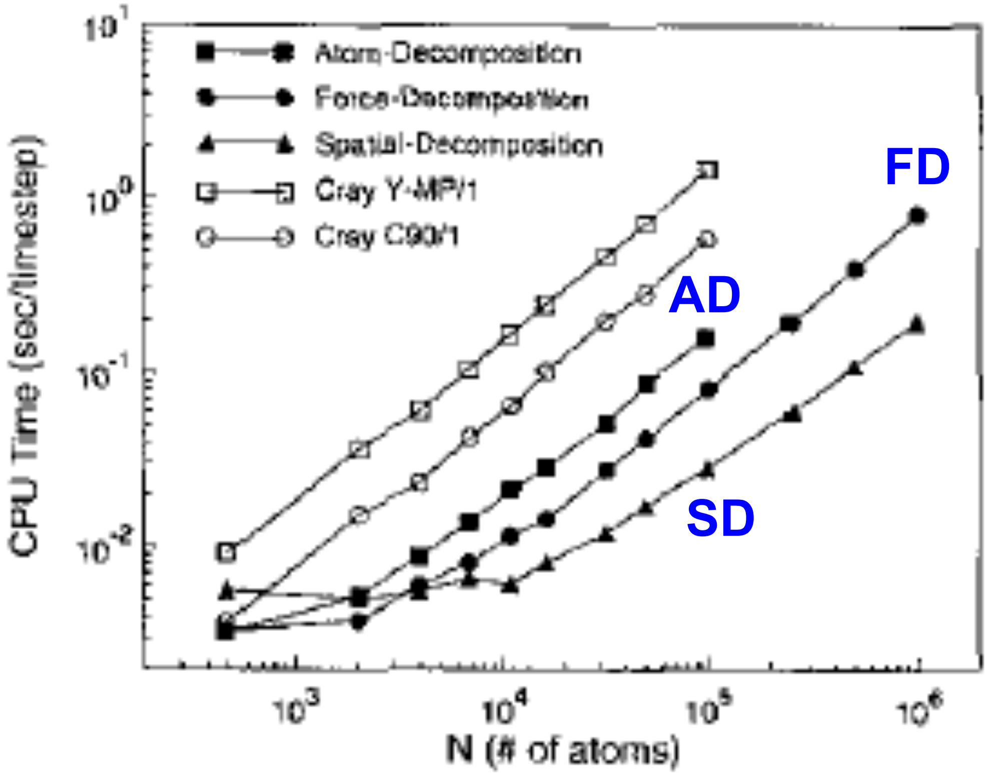


FIG. 5. The division of the permuted force matrix F' among 16 processors in the force-decomposition algorithm. Processor P_6 is assigned a sub-block F'_6 of size $N/\sqrt{P}$ by $N/\sqrt{P}$. To compute its matrix elements it must know the corresponding $N/\sqrt{P}$-length pieces x_α and x'_β of the position vector x and permuted position vector x'.

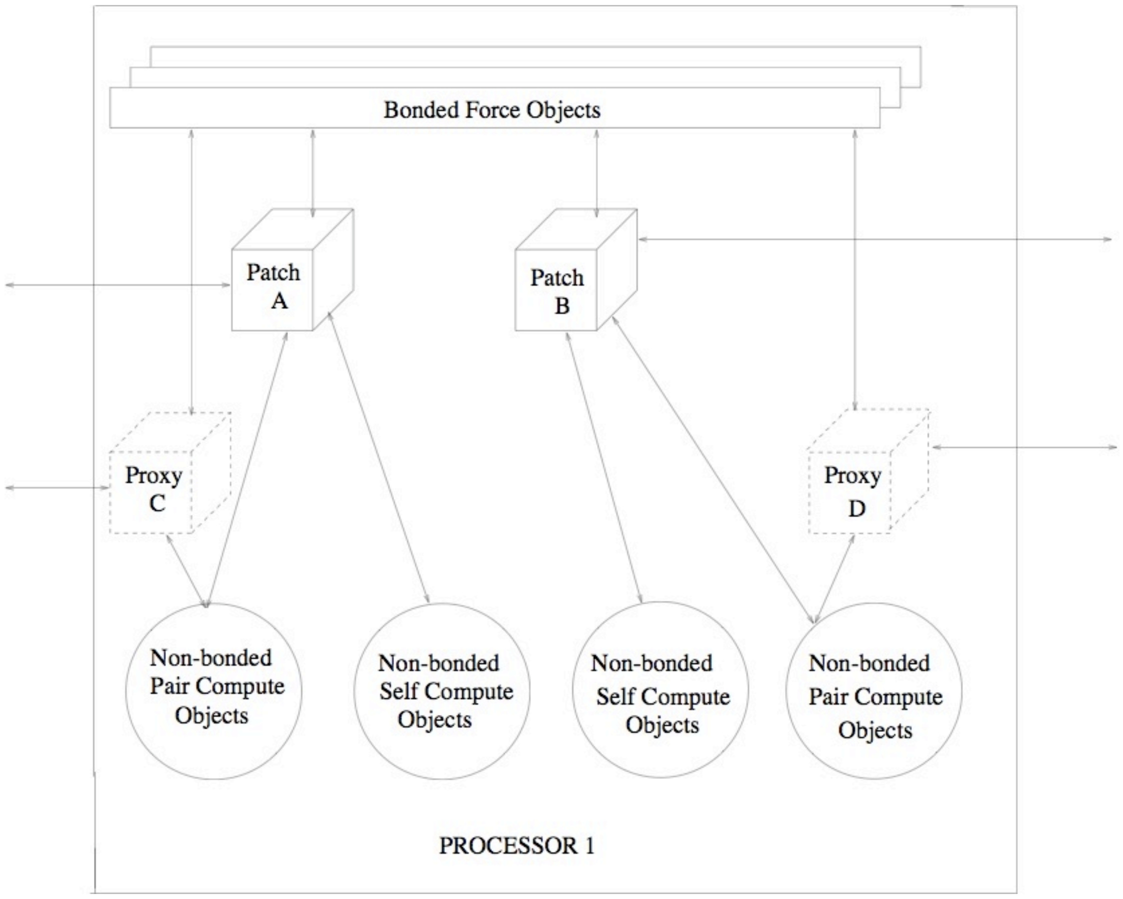


Runtime on 1,024-processor Intel Paragon

S. Plimpton, *J. Comput. Phys*. **117**, 1 (’95)

Hybrid Spatial+Force Decomposition

- **Spatial decomposition of patches (localized spatial regions & atoms within)**
- **Inter-patch force computation objects assigned to any processor**
- **Message-driven object execution**


Bonded Force Objects
Patch A
Patch B
Proxy C
Proxy D
Non-bonded Pair Compute Objects
Non-bonded Self Compute Objects
Non-bonded Self Compute Objects
Non-bonded Pair Compute Objects
PROCESSOR 1

L. Kale et al., *J. Comput. Phys*. **151**, 283 ('99); J. C. Phillips *et al*., *SC2002* (IEEE/ACM)

Quantum MD@Scale

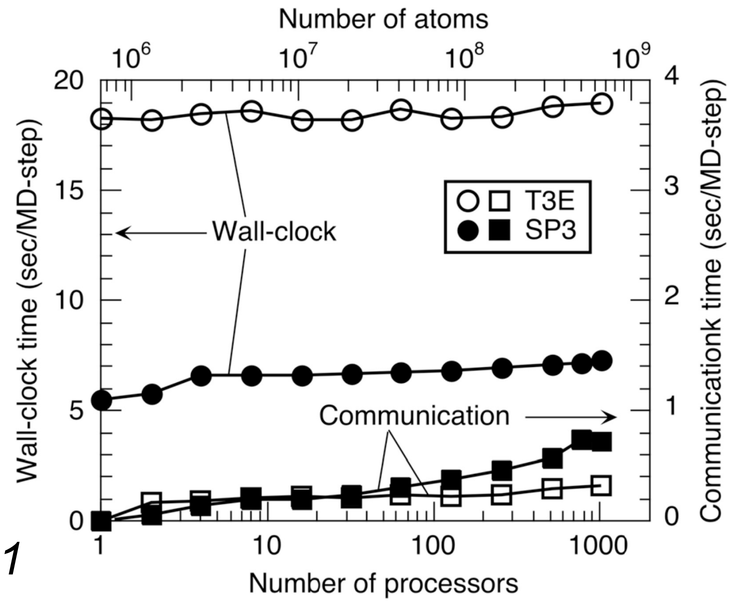
Quantum dynamics at scale: ultrafast control of emergent functional materials

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 HPCAsia 2020*

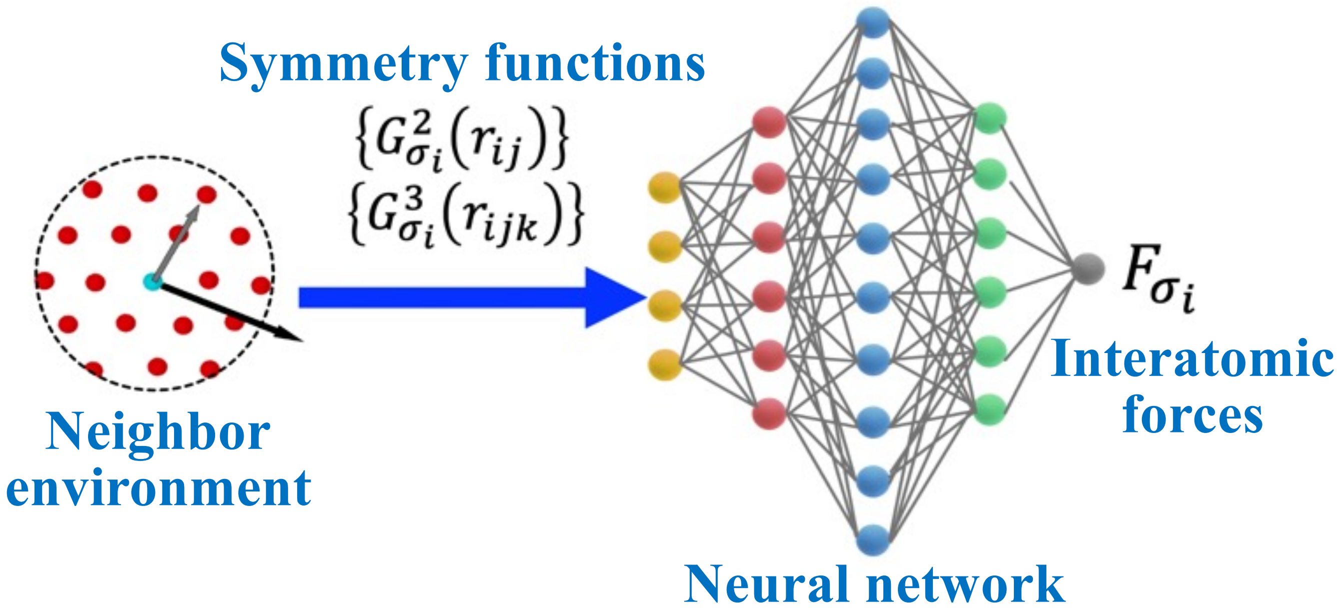
Scalable atomistic simulation algorithms for materials research, A. Nakano *et al.*, Best Paper, *IEEE/ACM Supercomputing 2001, SC01*

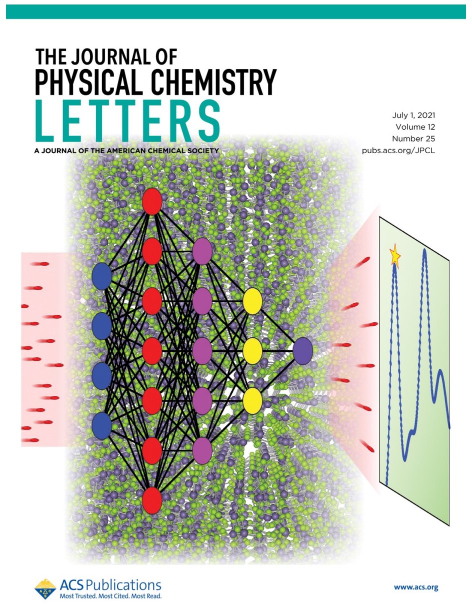
19 years since



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)]




THE JOURNAL OF PHYSICAL CHEMISTRY LETTERS
A JOURNAL OF THE AMERICAN CHEMICAL SOCIETY
July 1, 2021
Volume 12
Number 25
pubs.acs.org/JPCL
ACS Publications
Most Trusted. Most Cited. Most Read.
www.acs.org

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*

What We Have Learned Here

- Single program multiple data (SPMD) parallel programming for multicomputers based on message passing interface (MPI), using molecular dynamics (MD) as a prototypical example.
- Parallel computing = decomposition (who does what).
- Data locality-exposing data structure like linked-list cells leads to straightforward parallelization.
- Spatial, particle, force & hybrid decompositions.
- Scalability analysis based on analytical models.