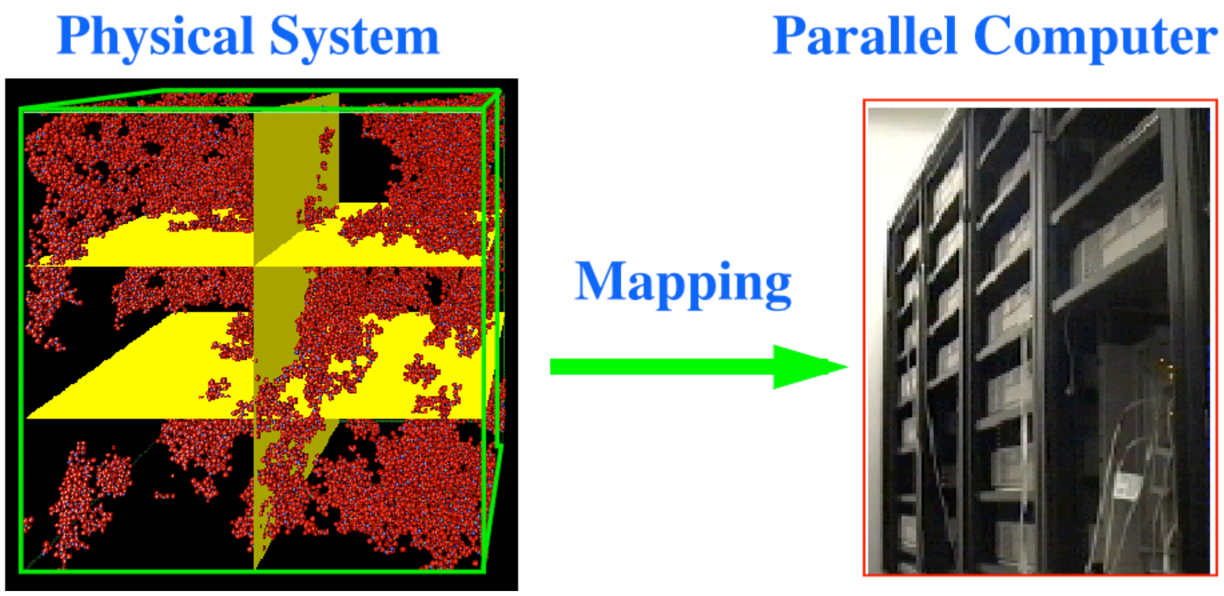
Parallel Molecular Dynamics

This chapter explains the example parallel MD program, `pmd.c`, in detail.

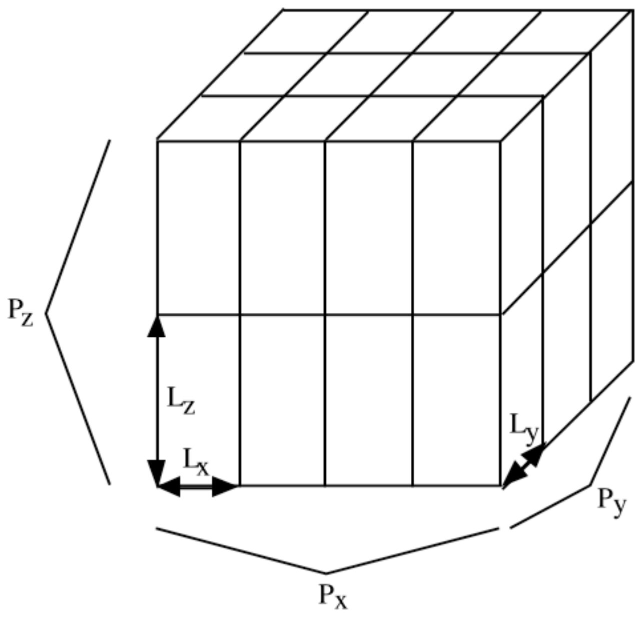
Spatial Decomposition

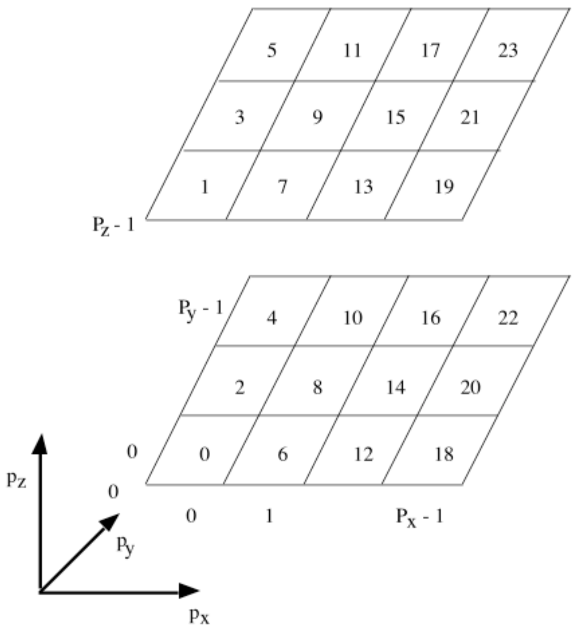
- **Spatial decomposition**: The physical system to be simulated is partitioned into subsystems of equal volume. Processors in a parallel computer are logically arranged according to the topology of physical subsystems. Atoms that are located in a particular subsystem are assigned to the corresponding processor.



We use a simple 3D mesh (or torus because of the periodic boundary conditions) decomposition. Subsystems are (and accordingly processors are logically) arranged in a 3D array of dimensions $P_x \times P_y \times P_z$ (`int vproc[3]` in `pmd.c`). Each subsytem is a parallel-piped of size $L_x \times L_y \times L_z$ (`double al[3]`).

PROCESSOR ID





Each processor is given a unique processor ID, p, the range of which is $[0, P\text{-}1]$ (`int sid`) where $P = P_x P_y P_z$ is the total number of processors (`int nproc`). We also define a vector processor ID $\vec{p} = (p_x, p_y, p_z)$ (`int vid[3]`), where $p_x = 0, ..., P_x\text{-}1$; $p_y = 0, ..., P_y\text{-}1$; and $p_z = 0, ..., P_z\text{-}1$.

The relation between the sequential and vector ID's is

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

or

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

(Example) $(P_x, P_y, P_z) = (4,3,2)$

Sequential processor ID, p	Vector ID, $\vec{p}$ p_x	p_y	p_z
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	0	2	0
5	0	2	1
6	1	0	0
7	1	0	1
8	1	1	0
9	1	1	1
10	1	2	0
11	1	2	1
...	...	...	...

NEIGHBOR PROCESSOR ID

For each processor, the six face-shared neighbor are identified by a sequential index ($\kappa = 0, ..., 5$). For each neighbor, κ, the shift-length vector $\vec{\Delta} = (\Delta_x, \Delta_y, \Delta_z)$ denotes the position of the neighbor subsystem relative to itself (`double sv[6][3]`). The following table lists the six neighbors, where an integer vector $\vec{\delta} = (\delta_x, \delta_y, \delta_z)$ specifies the relative location of each neighbor.

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)	$(0, 0, L_z)$

The sequential processor ID, $p'(\kappa)$ (`int nn[6]`), is obtained by

$$p'_\alpha(\kappa) = [p_\alpha + \delta_\alpha(\kappa) + P_\alpha] \bmod P_\alpha \; (\alpha = x, y, z)$$

and

$$p'(\kappa) = p'_x(\kappa) \times P_yP_z + p'_y(\kappa) \times P_z + p'_z(\kappa)$$

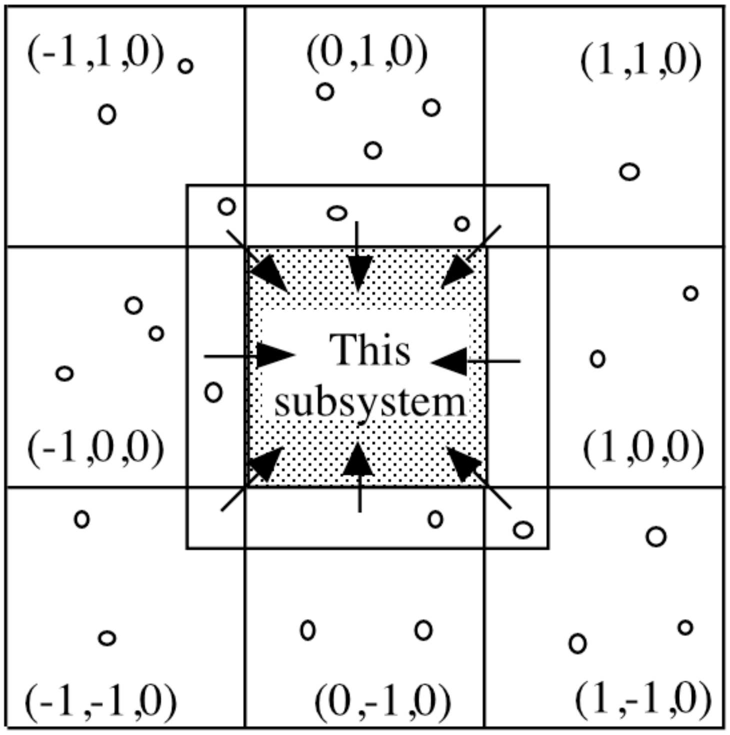
In an SPMD program using the MPI, each process knows its process ID through the returned `rank` value from an `MPI_Comm_rank(MPI_Comm comm, int *rank)` call. We identify the `rank` with `sid`.

Parallel MD Concepts

Using the periodic boundary conditions, every subsystem has 26 neighbor subsystems, which share either a corner, an edge, or a face with it. (Or there are 6 face-sharing neighbor subsystems.) We express atomic coordinates relative to the origin of each subsystem, i.e., $0 < x_\alpha < L_\alpha$ (α = x, y, z). (Every subsystem thinks it is the center of the world.)

ATOM CACHING

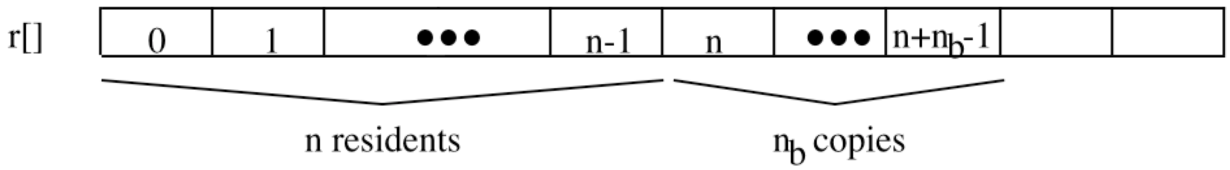
- **Augmented system**: In order to compute interatomic interaction with cut-off length r_c (`RCUT`), atomic coordinates of 26 neighbor subsystems which are located within r_c from the subsystem boundary are copied to "this" processor (**atom caching**).



- **Cache coherence** is maintained by copying the latest neighbor surface atoms every time before atomic accelerations are computed.

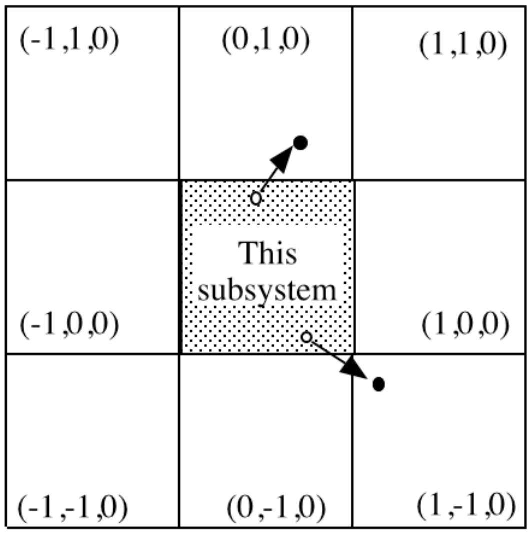
DATA STRUCTURES

`n`: The number of resident atoms which reside in "this" subsystem.
`nb`: The number of copied surface atoms from the neighbors.
`r[NEMAX][3]`: Atomic coordinates relative to the subsystem origin. `r[0:n-1][]` hold the atoms reside in this subsystem, while `r[n:n+nb-1][]` hold the cached neighbor atoms.



ATOM MIGRATION

After the atomic coordinates are updated according to the velocity Verlet algorithm, some resident atoms may have moved out of the subsystem boundary. Such **migrated** atoms must be **moved** to proper processors.



PARALLEL MD ALGORITHM

In the parallel MD program, `pmd.c`, a single time step of the velocity-Verlet algorithm consists of the following steps.

```
1. First half kick to obtain vi(t+Dt/2)
2. Update atomic coordinates to obtain ri(t+Dt)
3. atom_move():  Migrate the moved-out atoms to the neighbor processors
4. atom_copy():  Copy the surface atoms within distance rc from the neighbors
5. compute_accel():  Compute new accelerations, ai(t+Dt), including
   the contributions from the cached atoms
6. Second half kick to obtain vi(t+Dt)
```

Linked-List Cell Method for Interatomic Force Computation `function compute_accel()`

Recall that a naive double-loop implementation to compute pair interactions scales as $O(N^2)$. In fact, with a finite cut-off length, r_c, an atom interacts with only a limited number of atoms ~ $(4\pi/3)\ r_c^3\ (N/V)$. The linked-list cell algorithm computes the entire interaction with $O(N)$ operations.

CELLS

First divide the system consisting of the resident and cached atoms into small cells of equal size. The edge lengths of each cell, (r_{cx}, r_{cy}, r_{cz}) (`double rc[3]`), must be at least r_c; we use $r_{c\alpha} = L_\alpha/L_{c\alpha}$, where $L_{c\alpha} = \lfloor L_\alpha/r_c \rfloor$. An atom in a cell interacts with only other atoms in the same cell and its 26 neighbor cells. Including the cached atoms, the range of atomic coordinates is $[-r_c, L_\alpha + r_c]$ (α = x, y, z). The number of cells to accommodate all these atoms is $(L_{cx}+2)\times(L_{cy}+2)\times(L_{cz}+2)$, where $L_{c\alpha} = L_\alpha/r_{c\alpha}$ (α = x, y, z) (`int lc[3]`). We identify a cell with a vector cell index, $\vec{c} = (c_x, c_y, c_z)$ $(0 \le c_x \le L_{cx}+1;\ 0 \le c_y \le L_{cy}+1;\ 0 \le c_z \le L_{cz}+1)$, and a sequential cell index,

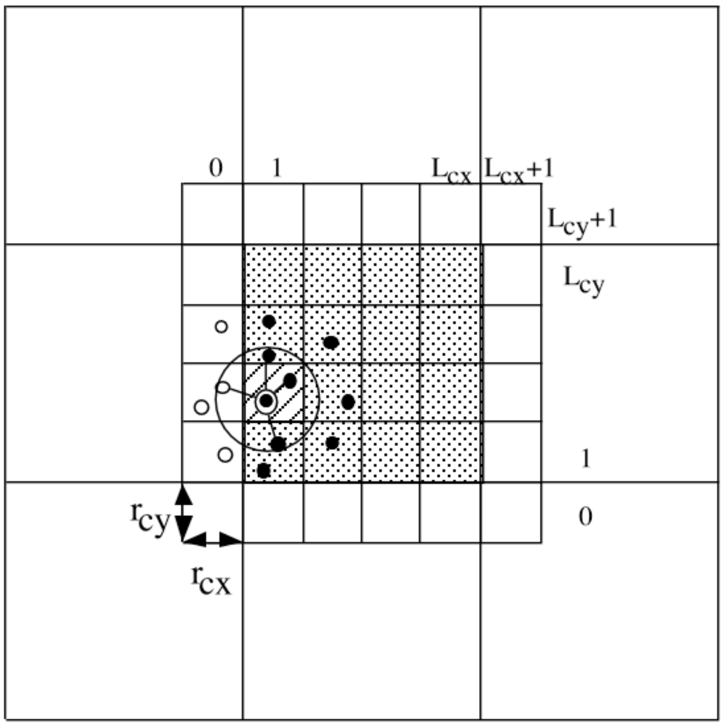
$$c = c_x(L_{cy}+2)(L_{cz}+2) + c_y(L_{cz}+2) + c_z$$

or

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

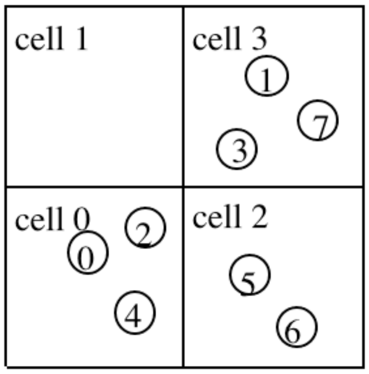
An atom with coordinate $\vec{r}$ belongs to a cell with the vector cell index,

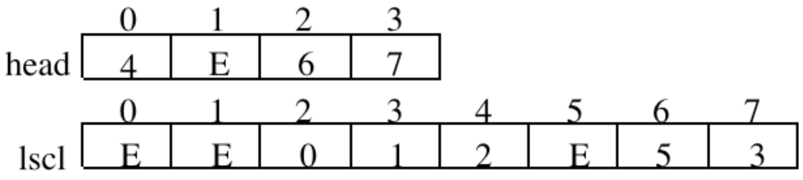
$$c_\alpha = \lfloor (r_\alpha + r_{c\alpha})/r_{c\alpha} \rfloor\ (\alpha = \text{x, y, z}).$$

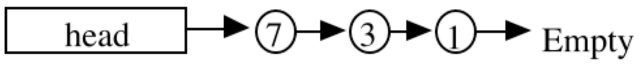


LISTS

The atoms belonging to a cell is organized as a linked list.







DATA STRUCTURES

`lscl[NEMAX]`: An array implementation of the linked list. `lscl[i]` holds the atom index to which the i-th atom points.
`head[NCLMAX]`: `head[c]` holds the index of the first atom in the *c*-th cell, or `head[c]` = `EMPTY` (= -1) if there is no atom in the cell.

ALGORITHM 1: LIST CONSTRUCTOR

```
/* 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;
}
```

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

ALGORITHM 2: INTERACTION COMPUTATION

```
/* Scan inner cells */
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 */
  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];
    }
  }
}
```

Note that the central cells, *c*, and hence the central atoms, *i*, are residents, whereas the pair atoms, *j*, are either residents or cached. The parallelization strategy in `pmd.c` is that each MPI process is responsible for computing the forces on its resident atoms. Accordingly, the force contribution, $\vec{a}_{ij}$, between pair (*i*, *j*) is added on the force on atom *j*, only if *j* is resident. Since all cached atoms are appended after the *n* resident atoms, atom *j* is resident if $j < n$:

```
if (j < n) ra[j][a] -= the a-th component of $\vec{a}_{ij}$
```

Regarding the potential energy calculation, note that if *j* is cached, the same pair, (*i*, *j*), is also considered by the process of which *j* is a resident. Our parallelization strategy is that each process adds 1/2 of the pair contribution:

```
if (j < n) potential energy += $u(r_{ij})$; else potential energy += $u(r_{ij})$/2;
```

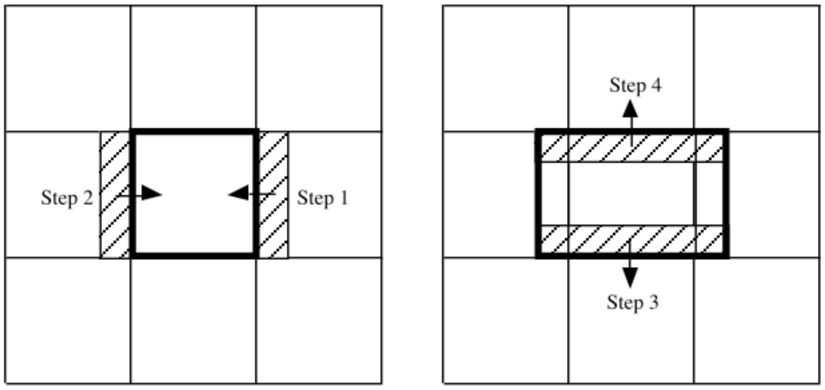
Note that each process first computes the local contribution to the potential energy, and subsequently `MPI_Allreduce()` is used to compute the global summation of the local potential energies.

Atom Caching—`function atom_copy()`

DATA STRUCTURES

`lsb[6][NBMAX]: lsb[ku][0]` is the total number of boundary atoms to be sent to neighbor `ku`; `lsb[ku][k]` is the atom ID, used in `r`, of the `k`-th atom to be sent.
`logical function bbd(r,ku): true` if the coordinate vector `r` is within distance `rc` from the face this subsystem shares with the `ku`-th neighbor.

The coordinates of the boundary atoms to the six face-sharing neighbors are sent to these nodes in six steps. Copies to the non-face-sharing neighbors (26 – 6 = 20) are done by message forwarding.



ALGORITHM

```
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
```

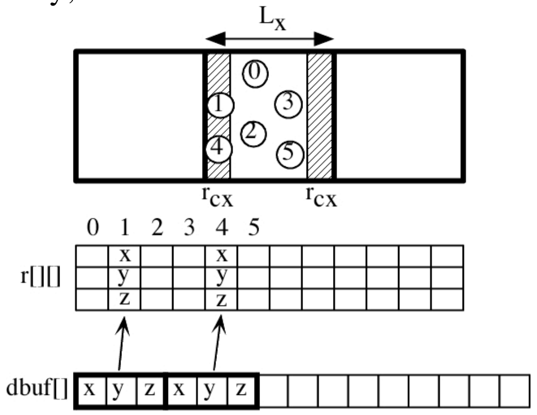
*Copying condition

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

#Three-phase message passing

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

Variable `nsd` is the number of atoms to be copied to the neighbor currently considered. The `3*nsd` coordinates are packed in a 1D array, `dbuf`.

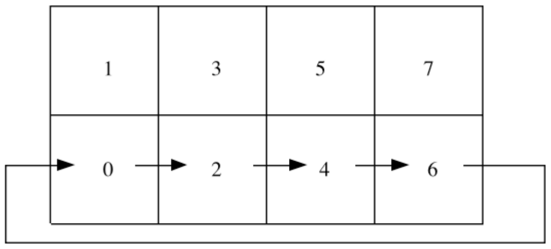


DEADLOCK AVOIDANCE

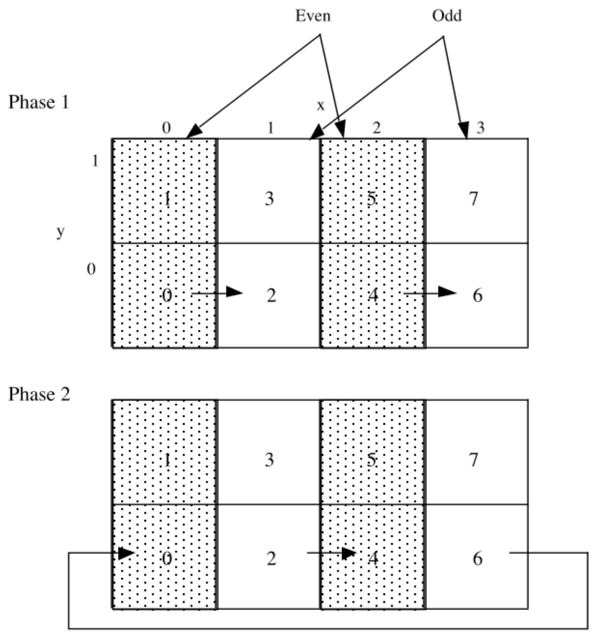
A circular send and receive relation could cause a deadlock. (Circular wait can occur if blocking.) In the following code, suppose that the destination node, `inode`, is defined as in the figure below.

```
MPI_Send(dbuf,3*nsd,MPI_DOUBLE_PRECISON,inode,10,MPI_COMM_WORLD);
MPI_Recv(dbufr,3*nrc,MPI_
DOUBLE_PRECISON,MPI_ANY_SOURCE,10,MPI_COMM_WORLD,&status);
```

With a finite buffer size in the receiver's communication system, a sender blocks until the receiver's buffer is cleared. However in the example above, each processor cannot start receiving until its send operation is completed.



To avoid this deadlock, we classify the processors into even and odd processors in each of the x, y, and z directions.



Data Structures

`myparity[0|1|2]` = Parity of vector processor ID in the x|y|z direction. `myparity[a]` is 0|1 if the vector processor ID, `vid[a]`, is even|odd. If there is only one processor in the a direction, `myparity[a]` = 2. In this case, no message passing is required.

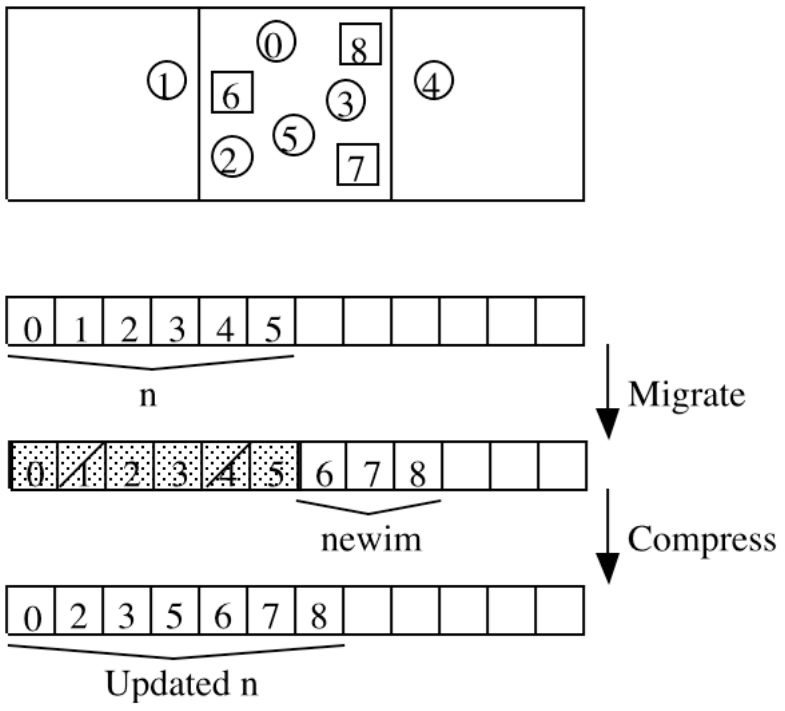
ALGORITHM

```
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
```

Atom Migration—`function atom_move()`

Data Structures

`mvque[6][NBMAX]`: `mvque[ku][0]` is the # of to-be-moved atoms to neighbor `ku`; `MVQUE[ku][k>0]` is the atom ID, used in `r`, of the `k`-th atom to be moved.
`logical function bmv(r,ku)`: `.true.` if the coordinate vector `r` is out of the subsystem boundary plane which faces the `ku`-th neighbor.



The algorithm is basically the same as the one in atom_copy. During the 6-step loop, variable `newim` keeps the number of received new immigrants. These atoms are appended in the `r` array after the current resident. Suppose atom `i` has moved-out. When sending out `r[i]` to a proper neighbor, `r[i]` is marked as moved-out by putting constant `MOVED_OUT` (large negative value, -1e10, defined in `pmd.h`) in `r[i][0]`. When scanning `n+newim` atoms to sort out the moved-out atoms, these already-moved atoms are not considered. After the six move steps, the `r` array is compressed to remove the moved-out atoms.

ALGORITHM

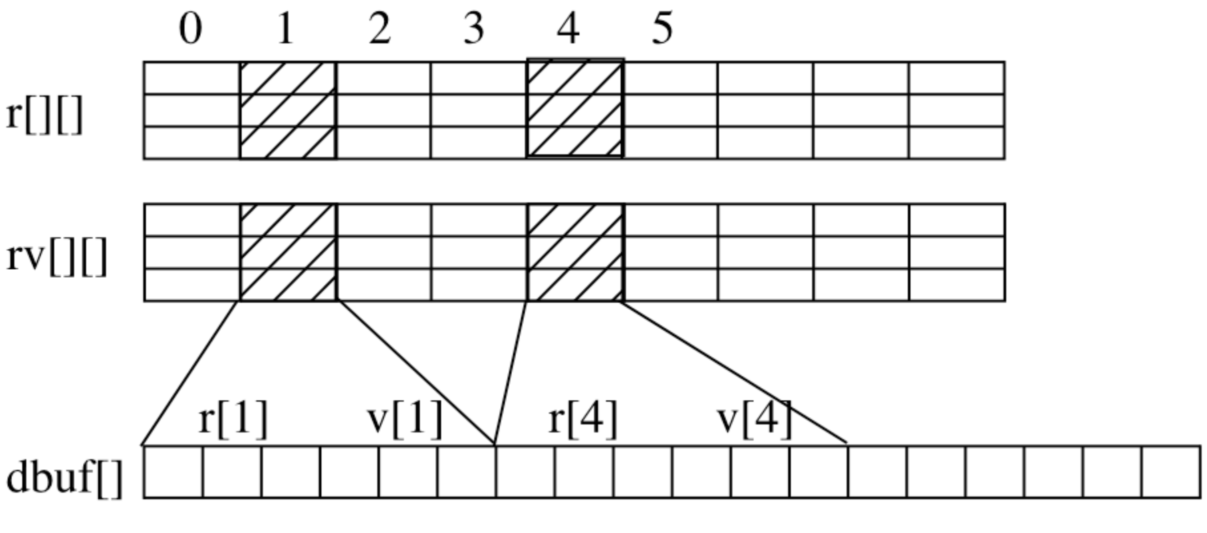
```
Reset the number of received new immigrants, newim = 0
for x, y, and z directions
  Make moving-atom lists, mvque, for lower and higher directions including both
  resident, n, and immigrant, newim, atoms but excluding those already moved out
  for lower and higher directions*
    Send/receive moving-atom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

*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]
   }
```

#Three-phase message passing

1. Message buffering: `dbuf` ← `r & 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



Scalability Metrics for Parallel Molecular Dynamics

Parallel Efficiency

We define the efficiency of a parallel program running on P processors to solve a problem of size W. Let $T(W, P)$ be the execution time of this parallel program. ***Speed*** of the program is then $S(W, P) = W/T(W, P)$. ***Speedup***, S_P, on P processors is the speed of P processors divided by that of 1 processor, i.e., $S_P = S(W_P, P)/S(W_1, 1)$. To unambiguously define the speedup, we need to specify how the problem size, W_P, scales as a function of the number of processors, P (which will be discussed in the next few paragraphs). The ideal speedup on P processors is expected to be P, and therefore we define the ***parallel efficiency***, $E_P = S_P/P$.

CONSTANT PROBLEM-SIZE SCALING

In the constant problem-size scaling, the problem size $W_P = W$ is fixed constant. Therefore, the constant problem-size speedup is

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

and the parallel efficiency is

$$E_P = \frac{S_P}{P} = \frac{T(W,1)}{P \bullet T(W,P)}.$$

Amdahl's law: Consider a case, in which a fraction, f ($\in [0, 1]$), of the work W is inherently sequential and cannot be parallelized, but the rest, $1 - f$, can be divided and executed in parallel on P processors. Then, $T(W, P) = f \bullet T(\mathrm{W}, 1) + (1 - f)T(W, 1)/P$. Therefore, the speedup is

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

In the limit of large number of processors, $P \to \infty$, the asymptotic speedup is $S_P \to 1/f$. The constant problem-size speedup is thus limited by the fraction of the sequential bottleneck of the parallel program. For example, if 1% of the work is sequential, the maximum achievable speedup is 0.01/1 = 100, and it does not make sense to use more than ~100 processors to run this program.

ISOGRANULAR SCALING

In the isogranular scaling, the problem size W_P scales linearly with the number of processors: $W_P = P \bullet w$, where the ***granularity*** (or the work per processor), w, is constant. Therefore, the isogranular speedup is

$$S_P = \frac{S(P \bullet w,P)}{S(w,1)} = \frac{P \bullet w/T(P \bullet w,P)}{w/T(w,1)} = \frac{P \bullet T(w,1)}{T(P \bullet w,P)},$$

and the corresponding isogranular parallel efficiency is

$$E_P = \frac{S_P}{P} = \frac{T(w,1)}{T(P \bullet w,P)}.$$

Analysis of Parallel Molecular Dynamics Algorithm

Using the spatial decomposition and the $O(N)$ linked-list cell method, the parallel MD simulation of N atoms executes independently on P processors, and the computation time $T_{\text{comp}}(N, P) = aN/P$, where a is a constant. Here, we have assumed that atoms are on average distributed uniformly, so that the average number of atoms per processor is N/P. The dominant overhead of the parallel MD is atom caching, in which atoms near the subsystem boundary within a cutoff distance, r_c, are copied from the nearest neighbor processors and are processed. Since this nearest-neighbor communication scales as the surface area of each spatial subsystem, its time is $T_{\text{comm}}(N, P) = b(N/P)^{2/3}$, where b is a constant. Another major communication cost is for global summations, `MPI_Allreduce()`, which incurs $T_{\text{global}}(P) = c \log P$, where c is another constant.

The total execution time of the parallel MD program can thus be modeled as

$$
\begin{aligned}
T(N,P) &= T_{\text{comp}}(N,P) + T_{\text{comm}}(N,P) + T_{\text{global}}(P) \\
&= aN/P + b(N/P)^{2/3} + c\log P
\end{aligned}
.
$$

CONSTANT PROBLEM-SIZE SCALING

For constant problem-size scaling, the global number of atoms, N, is fixed, and the speedup is given by

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

and the parallel efficiency is

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

From this model, we can see that the efficiency is a decreasing function of P through both the $P^{1/3}$ and $P\log P$ dependences.

ISOGRANULAR SCALING

For isogranular scaling, the number of atoms per processor, $N/P = n$, is constant, and the isogranular parallel efficiency is

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

For a given number of processors, the efficiency E_P is larger for larger granularity n. For a given granularity, E_P is a weakly decreasing function of P, due to only the $\log P$ dependence.