CS420 – Lecture 14

Marc Snir

Fall 2018

Application Examples

Particle Code – Molecular Dynamics

Simulate movement of bunch of atoms, each with its charge
Highly simplified description: Repeat, for millions of steps

- Compute forces between atoms
  - Coulomb forces, due to atom charge
  - Bond forces
- Update velocity
- Update location

No need to maintain velocity:

$$\frac{\Delta^2 \vec{x}_n}{\Delta t^2} = \frac{\frac{\vec{x}_{n+1}-\vec{x}_n}{\Delta t} - \frac{\vec{x}_n-\vec{x}_{n-1}}{\Delta t}}{\Delta t} = \frac{\vec{x}_{n+1} - 2\vec{x}_n + \vec{x}_{n-1}}{\Delta t^2} = \vec{F}_n \cdot mass$$

$$\vec{x}_{n+1} = 2\vec{x}_n - \vec{x}_{n-1} + \vec{F}_n \cdot mass\, \Delta t^2.$$

Usually compute forces only for nearby atoms, use another method to handle long-range interactions

Algorithm outline

```
foreach pair of atoms x,y {
dist = ||x.loc − y.loc||;
if dist < C {
   direction = (x.loc − y.loc)/dist;
   force = −(x.charge · y.charge)/dist^2;
   x.force− = force · direction;
   y.force+ = force · direction;
}
}
foreach atom x {
temp = x.loc;
x.loc = 2 · temp − x.oldloc + x.force · x.mass;
x.oldloc = temp;
}
```

Will assume world is 2D – to shorten code

C does not have vector operations – need to define them. We do this with C macros

```
#define v_assign(x,y) {y[0]=x[0]; y[1]=x[1];} /* vec=vec */
#define s_times_v(s,x,y) {y[0]=s*x[0]; y[1]=s*x[1];} /* vec=scalar*vec */
#define v_plus_v(x,y,z) {z[0]=x[0]+y[0]; z[1]=x[1]+y[1];} /* vec=vec+vec */
#define v_minus_v(x,y,z) {z[0]=x[0]-y[0]; z[1]=x[1]-y[1];} /* vec=vec-vec */
#define v_norm(x)  sqrt(x[0]*x[0]+x[1]*x[1]) /* vector norm */
```

Can use functions – but then would want the compiler to inline them

```
static inline void
assign (double x[2], double y[2])
{
y[0] = x[0]; y[1] = x[1];
}
```

Inlining: No function call; the code of the function is inserted in place of a function call.

C compiler compiles each file speparately; can inline a static function – it will not be invoked from another file; otherwise need to inline at link time (-flto – link time optimization)

Data Structure

```
typedef struct Atom {
double loc[2]; /* current location */
double oldloc[2]; /* previous location */
double mass; /* mass */
double charge; /* charge */
double force[2]; /* force operating on atom */
} Atom;
```

Compute interaction between two atoms

```
void interact(Atom *x, Atom *y) {
double force, dist, direction[2];

v_minus_v( x->loc, y->loc, direction)
dist = v_norm(direction);
if (dist < C) {
 s_times_v( (1/dist), direction, direction);
 force = -(x->charge * y->charge)/(dist*dist);
 s_times_v(force, direction, direction);
 v_minus_v(x->force, direction, x->force);
 v_plus_v(y->force, direction, y->force);
 }
}
```

Move atom

Assume $\Delta t = 1$.

```
void move(Atom *x) {
 double temp[2];
 for (i=0; i<2; i++) {
  temp[i] = x->loc[i];
  x->loc[i] = 2*temp[i] - x->oldloc[i] + x->force[i]*x->mass;
  x->oldloc[i] = temp[i];
  }
 }
```

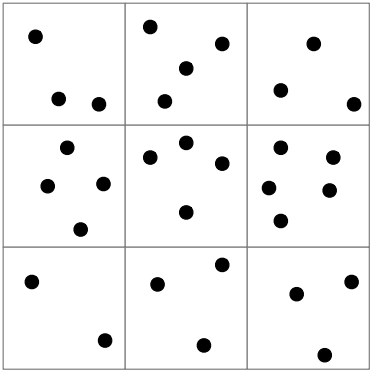
or

```
...
v_assign_v(x->loc, temp);
v_plus_v(temp, temp, x->loc);
v_minus_v(x->loc, x->oldloc, x->loc);
s_times_v(x->mass, x->force, part);
v_plus_v(x->loc, part, x->loc);
V_assign_v(temp,x->oldloc)
```

Functions should be inlined, if possible

parallel (message-passing) algorithm

- Divide the (2D) domain into cells; assign a cell to each process.
- Results in OK load balancing if atom density is close to constant. (True of molecular simulations, not of cosmology simulations.)
- Usually, cell dimension $\gg$ cutoff radius. Atoms may interact only with atoms in nearby cells.
- Array is periodic

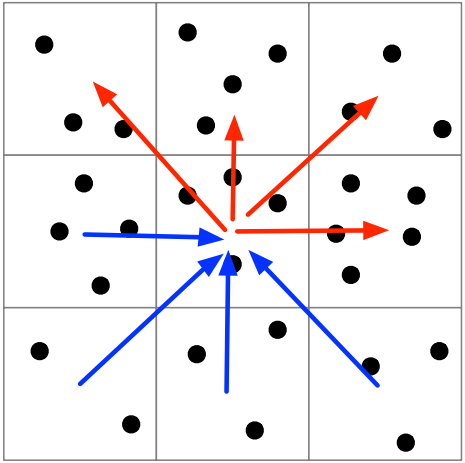


Algorithm outline

Repeat

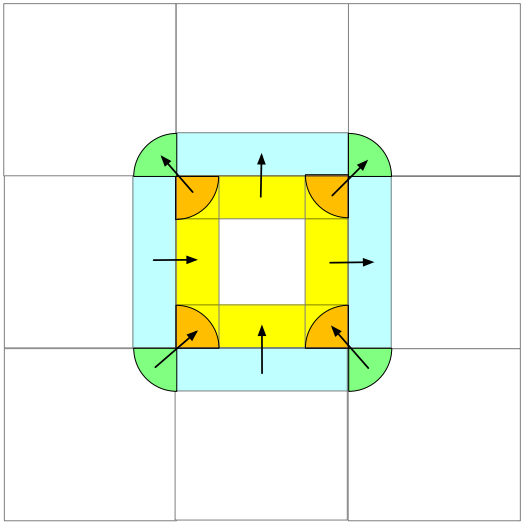
1. Broadcast atoms to neighbor cells
2. Compute atom-atom interactions
3. Collect back computed forces
4. Update locations
5. Move particles that crossed cell boundaries

To compute each force only once, send in 4 directions and receive from 4 directions



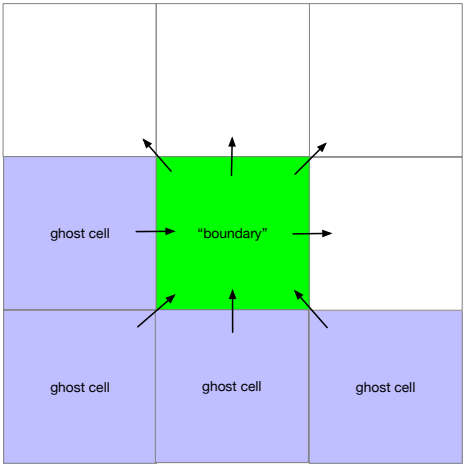
Interactions

- Atoms at the boundary of a cell interact with atom on the corresponding boundary of the neighboring cell
- Can use "ghost cell" pattern – but need to recompute boundary at each iteration (atom moves!)
- Have a pretty good estimate of number of atoms in each slice ($\sim$ constant density)



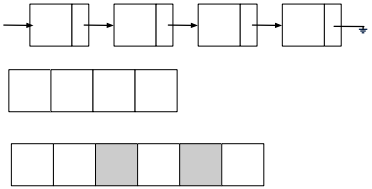
Simplified algorithm

- Assume all atoms could interact with atoms in neighboring cells (they are all "on the boundary")
- Superflous work and communication but correct, since check cuttoff before computing atom-atom interaction
- "boundary" communicated to four other processess (NE, N, NW, W)
- Process stores 4 ghost cells, where it receives atoms from E, SE, S, SW



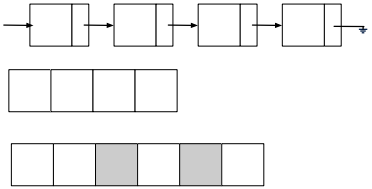
Implementation choices

- No need to communicate all atom fields: Need to broadcast location and charge and receive back (reduce) forces– might be worthwhile to keep these fields separately
- How do we store atom list?
  - Don't care the order in which they are stored
  - Need to insert and delete atoms in the list (when they move from cell to cell)
  - But few atoms change cell at each iteration (pragmatic knowledge)



linked list

- Easy to insert or delete
- Data not contiguous and links take space

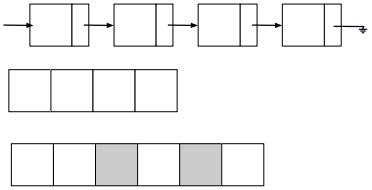


linked list

- Easy to insert or delete
- Data not contiguous and links take space

contiguous array

- Easy to insert (at array end)
- Expensive to delete, if deletes are at arbitrary locations

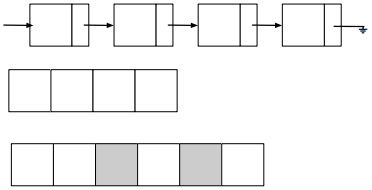


linked list

- Easy to insert or delete
- Data not contiguous and links take space

contiguous array

- Easy to insert (at array end)
- Expensive to delete, if deletes are at arbitrary locations
- Easy to delete (and compress) if deletes done while traversing the array sequentially



linked list

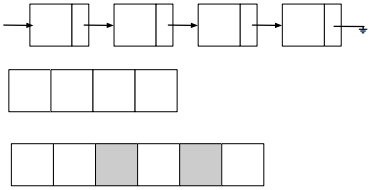
- Easy to insert or delete
- Data not contiguous and links take space

contiguous array

- Easy to insert (at array end)
- Expensive to delete, if deletes are at arbitrary locations
- Easy to delete (and compress) if deletes done while traversing the array sequentially

Array with "holes": Deleted items are marked invalid

- Easy to insert or delete
- OK if deletes are small fraction of accesses: Can compress periodically



- Will use array
- Will split parts to send and to receive
- "struct of arrays", rather than "array of structs", so that parts sent and received are contiguous, and vectorization easier.

```
/* atom data structures */
double loc[MAXATOMS][2];
double oldloc[MAXATOMS][2];
double mass[MAXATOMS];
double charge[MAXATOMS];
double force[MAXATOMS][2];
```

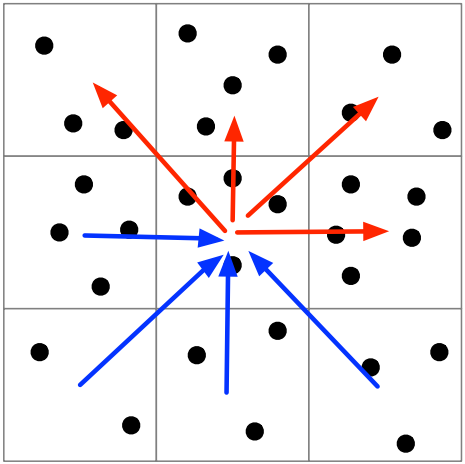
Not sure this is right choice; can decide via trial and error

Communication

How do we communicate?

- 4 sends & 4 receives (of atom lists) followed by 4 receives and 4 sends (of forces) (13 in 3D)
- 1 broadast as root and 4 broadcasts as receiver, followed by one reduce as root and 4 reduces as senders (adding forces) (1 and 13, in 3D)

Not sure which is better – will use collectives in example



Create Cartesian topology

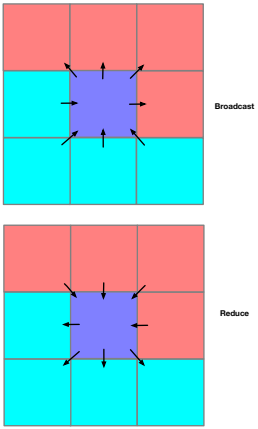
```
/* create 2D torus */
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Dims_create(size, 2, dims);
if (int dim = dims[0] != dims[1]) exit(1); /* should be square */
period[0]=period[1]=true;
MP_Cart_create(MPI_COMM_WORLD,2,dims,periods,true, &torus);
MPI_Comm_rank(torus, &myrank);
MPI_Cart_coords(torus, myrank, 2, mycoords); /* returns x,y coordinates */

...
```

Basic algorithm

repeat:

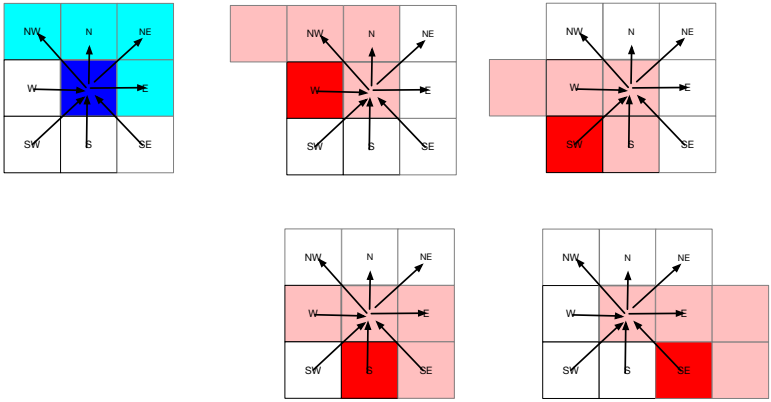
1. Send atom list to neighbors in directions NE, N, NW, W
2. Receive atom list from neighbors in directions E, SE, S, SW
3. Compute all interactions between local atoms and other atoms at process (local-local, local-E, local-SE, local-S,local-SW
4. Gather and sum forces acting on local atoms from neighbors in directions NE, N, NW, W and from local-local interactions
5. Compute new positions for local atoms
6. Communicate atoms that crossed boundaries

Need four ghost cells



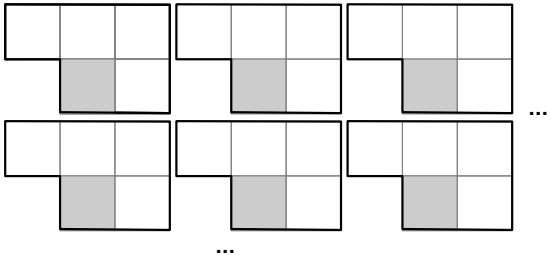
Communicators for broadcast-reduce

- process is root in the group containing itself, and neighbors in directions NW, N, NE, E.
- It participates in four other groups rooted at neighbors in directions W, SW, S, SE



Splitting communicators

- It is "easy" to split a communicator into sub-communicators with disjoint process groups; not possible to create overlapping communicators.
- Need to create the new communicators in successive phases, where communicators created at the same phase do not overlap
- Can do it in 6 phases, assuming `dimm` divides by 6: Slide basic template across 3×2 positions.



Splitting Communicators

`MPI_Comm_split(comm, color, key, newcomm)`

- Called collectively by all processes in `comm`
- One new communicator is created for each distinct value of `color`
- All processes that provides the same 'color' (an integer) end up in the same communicator
- They are ordered in increasing value of key (with ties broken according to the old rank)

Splitting Communicators

`MPI_Comm_split(comm, color, key, newcomm)`

- Called collectively by all processes in `comm`
- One new communicator is created for each distinct value of `color`
- All processes that provides the same 'color' (an integer) end up in the same communicator
- They are ordered in increasing value of key (with ties broken according to the old rank)

comm communicator being split

Splitting Communicators

`MPI_Comm_split(comm, color, key, newcomm)`

- Called collectively by all processes in `comm`
- One new communicator is created for each distinct value of `color`
- All processes that provides the same 'color' (an integer) end up in the same communicator
- They are ordered in increasing value of key (with ties broken according to the old rank)

comm communicator being split

color integer used to partition the processes

Splitting Communicators

`MPI_Comm_split(comm, color, key, newcomm)`

- Called collectively by all processes in `comm`
- One new communicator is created for each distinct value of `color`
- All processes that provides the same 'color' (an integer) end up in the same communicator
- They are ordered in increasing value of key (with ties broken according to the old rank)

`comm` communicator being split

`color` integer used to partition the processes

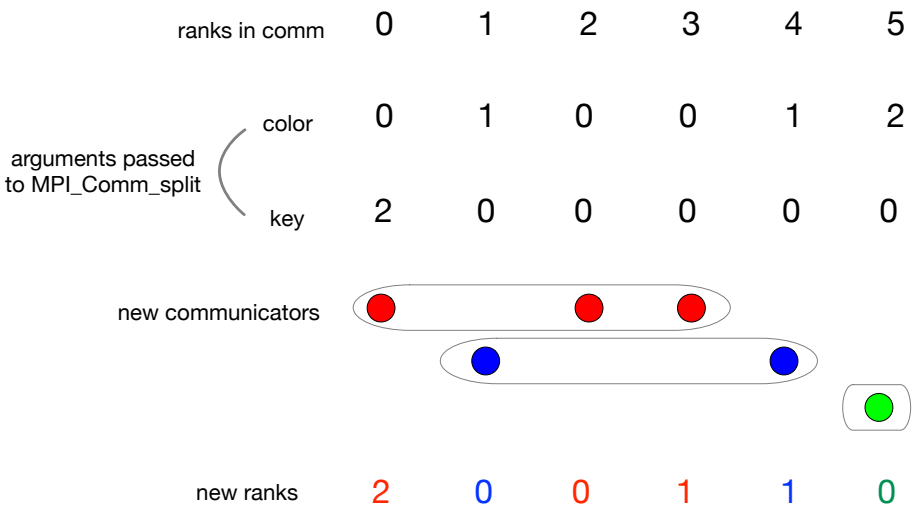
`key` integer used to specify process order in new communicator

Splitting Communicators

`MPI_Comm_split(comm, color, key, newcomm)`

- Called collectively by all processes in `comm`
- One new communicator is created for each distinct value of `color`
- All processes that provides the same 'color' (an integer) end up in the same communicator
- They are ordered in increasing value of key (with ties broken according to the old rank)

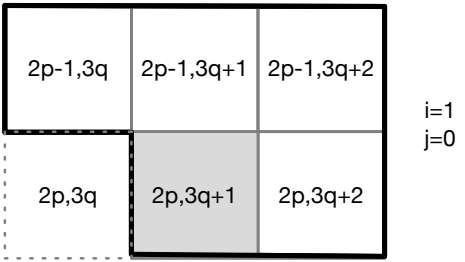
comm	communicator being split
color	integer used to partition the processes
key	integer used to specify process order in new communicator
newcomm	new communicator


ranks in comm
0 1 2 3 4 5
arguments passed to MPI_Comm_split
color
0 1 0 0 1 2
key
2 0 0 0 0 0
new communicators
new ranks
2 0 0 1 1 0

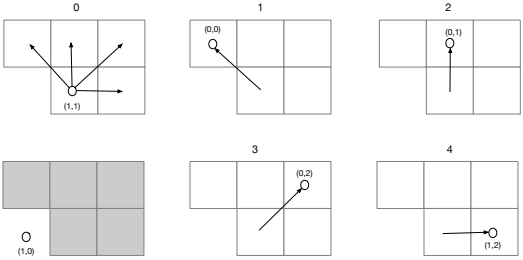
Assume `dimm` is a multiple of 6

- Let (r, s) be the coordinates of a cell.
- Compute $p = r/2$ and $q = s/3$.
- Six processes, in the 2×3 rectangle with cell $2p, 3q$ in the upper left corner, will compute the same (p, q) values.
- The plane will be tesselated by (`dimm`/2)×(`dimm`/3) 2×3 rectangles

- Compute, for $i = 0, 1$ and $j = 0, 1, 2$
  $p = ((r + i) \mod dimm)/2$ and
  $q = ((s + j) \mod dimm)/3$.
- Six processes, in the 2×3 rectangle with cell $(2p - i, 3q - j)$ in the upper left corner, will compute the same (p, q) values.
- Each 2×3 rectangle will obtain once
- Only five of the processes in the rectangle should be in the newly created communicator



- Let $m = (r + i) \mod 2$ and $n = (s + j) \mod 2$
- If $(m, n) = (1, 1)$ then process (r, s) is the root of the broadcast
- If $(m, n) = (0, 0)$ then process (r, s) receives a broadcast from SE process
- If $(m, n) = (0, 1)$ then process (r, s) receives a broadcast from S process
- If $(m, n) = (0, 2)$ then process (r, s) receives a broadcast from SW process
- If $(m, n) = (1, 2)$ then process (r, s) receives a broadcast from W process
- If $(m, n) = (1, 0)$ then process should not be in communicator.



Create communicators

```
for(i=0; i<2; i++)
 for(j=0; j<3 ; j++) {
  p= ((mycoords[0]+i)%dimm)/2; m = mycoords[0]%2;
  q= ((mycoords[1]+j)%dimm)/3; n = myccords[1]%3;
  color = 3*p+q;
  if ((m==1) && (n==0())) color = MPI_UNDEFINED;
  k = 3*m+n+1;
  if(k==4) k=0; /* find communicator index */
  if(k==6) k=4;
  MPI_Comm_split(torus, color, 0, &comm[k]);
  }
```

- Each process belongs to 5 new communicators
- In one of the 6 calls, the color was `MPI_UNDEFINED` and the call returned `MPI_COMM_NULL`
- In each new communicator the processes are ordered in row major order (according to previous rank), so root has rank 3 in each of the new communicators
- Communicators are numbered as indicated in Figure

communication

```
/* broadcast number of atoms at root process */
numreq=0;
MPI_Ibcast(&numatom,1, MPI_INT, 3, req[numreq++]);
for (k=1; k<5; k++)
 MPI_Ibcast(&count[k], 1, MPI_INT, 3, comm[k], req[numreq++]);
MPI_Waitall(5,req, MPI_STATUSES_IGNORE);
count[0]=numatoms;
```

Assume that locations are stored in an array `loc[5][MAXATOMS][2]`: One MAXATOMS$\times$2 array for local coordinates and 4 for coordinates of atoms in neighbor cells. Charges are stored in array `charge[5][MAXATOMS]`

```
/* broadcast atoms */
numreq=0;
for(k=0;k<5;k++)
  MPI_Ibcast(loc[k][MAXATOMS], 2*count[k], MPI_DOUBLE, 3,
      comm[k], req[numreq++]);
MPI_Waitall(15, req, MPI_STATUSES_IGNORE);
```

Reduce

Broadcast is followed by force computations, next by the gathering of the computed forces

```
/* collect forces*/
numreq=0;
for(k=0;k<5;k++)
 MPI_Ireduce(force[k][MAXATOMS], 2*count[k], MPI_DOUBLE, 3, comm[k],
 req[numreq++]);
MPI_Waitall(5, req, MPI_STATUSES_IGNORE);
```

Algorithm steps

1. Broadcast atom list to neighbors in directions NE, N, NW, W
2. Receive atom list from neighbors in directions E, SE, S, SW
3. Compute all interactions between local atoms and other atoms at process (local-local, local-E, local-SE, local-S,local-SW
4. Gather and sum forces acting on local atoms from neighbors in directions NE, N, NW, W and from local-local interactions.
5. Compute new positions for local atoms
6. Communicate atoms that crossed boundaries

Algorithm steps

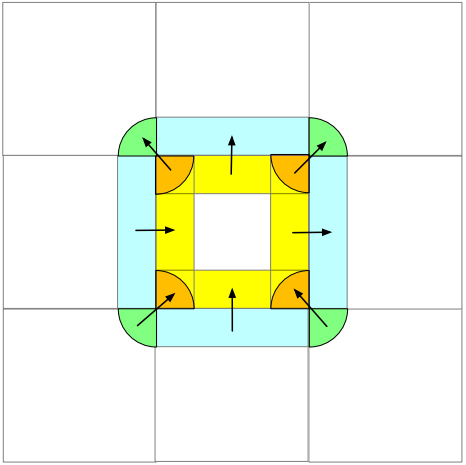
Shown how to implement the following:

1. Broadcast atom list to neighbors in directions NE, N, NW, W
2. Receive atom list from neighbors in directions E, SE, S, SW
3. Compute all interactions between local atoms and other atoms at process (local-local, local-E, local-SE, local-S,local-SW
4. Gather and sum forces acting on local atoms from neighbors in directions NE, N, NW, W and from local-local interactions.
5. Compute new positions for local atoms
6. Communicate atoms that crossed boundaries

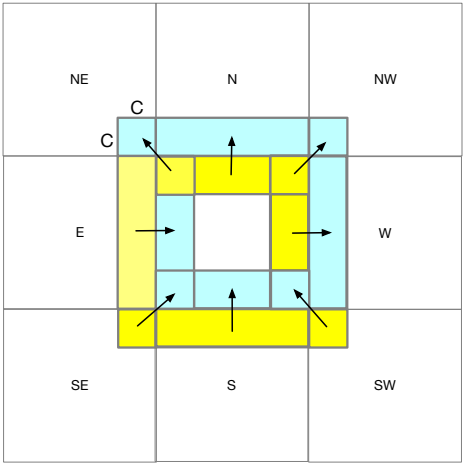
Possible Improvements (outline)

If cutoff radius is $\ll$ cell size, will want to communicate only cells close to boundarty

Boundary regions



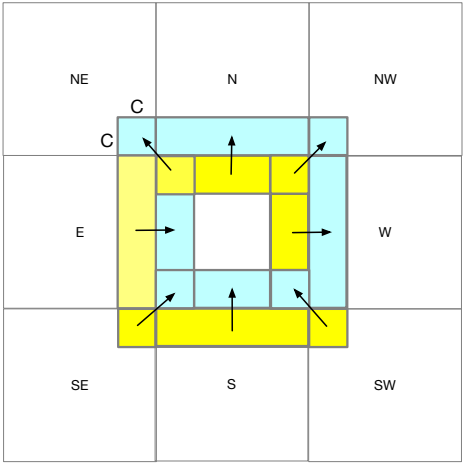
Simplified view (cutting corners)



Possible implementation

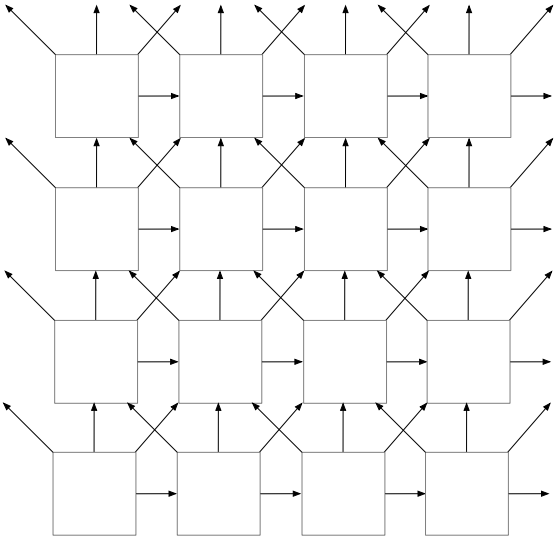
Keep 9 lists for atoms in each of the boundary regions and in center

1. Start send or receive of atoms in boundary regions
2. Compute interactions of atoms in center region
3. complete send/receives
4. Compute interactions of atom in adjacent boundary regions
5. Send/receive back forces
6. Sum forces
7. Update local atom locations
8. Move atoms crossing boundaries



Alternative Approach – Use Sparse Collectives

- Can create a communicator that has a directed graph topology
- Can execute a *Neighborhood Gather* collective call that sends data on each outgoing edge and receives data on each incoming edge
- Need a graph to send atoms and a reverse graph to receive forces (no reduce)



Find neighbors

```
/* assume comm has a Cartesian topology */
...
enum directions {NW, N, NE, E, SE, S, SW, W};
int neighbors[8]; /* ranks of neighbors */
int coords[2], ncoords[2], dims[2], periods[2], neighbor[8];
MPI_Cart_get(comm, 2, dims, periods, coords)
                /* returns info on Cartesian topology */
ncoords[0]=(coords[0]-1)%dims[1];
ncoords[1]=(coords[1]-1)%dims[1];
MPI_Cart_rank(comm, ncoords, &neighbors[NW]);
                /* translates Cartesian coordinates into ranks */
ncoords[0]= (ncoords[0]+1)%dims[0];
MPI_Cart_rank(comm, ncoords, &neighbors[N]);
ncoords[0]= (ncoords[0]+1)%dims[0];
MPI_Cart_rank(comm, ncoords, &neighbors[NE]);
ncoords[1]= (ncoords[1]+1)%dims[1];
MPI_Cart_rank(comm, ncoords, &neighbors[E]);
...
```

```
MPI_Cart_get(comm, maxdims, dims, periods, coords)
```

```
MPI_Cart_get(comm, maxdims, dims, periods, coords)
```

comm communicator with Cartesian structure

```
MPI_Cart_get(comm, maxdims, dims, periods, coords)
```

comm communicator with Cartesian structure

maxdims number of dimensions

```
MPI_Cart_get(comm, maxdims, dims, periods, coords)
```

- comm: communicator with Cartesian structure
- maxdims: number of dimensions
- dims: extent of each dimension

```
MPI_Cart_get(comm, maxdims, dims, periods, coords)
```

- comm: communicator with Cartesian structure
- maxdims: number of dimensions
- dims: extent of each dimension
- periods: periodicity in each dimension

```
MPI_Cart_get(comm, maxdims, dims, periods, coords)
```

comm	communicator with Cartesian structure
maxdims	number of dimensions
dims	extent of each dimension
periods	periodicity in each dimension
coords	Cartesian coordinates of calling process

```
MPI_Cart_rank(comm, coords, rank)
```

`MPI_Cart_rank(comm, coords, rank)`

`comm` communicator with Cartesian topology

`MPI_Cart_rank(comm, coords, rank)`

comm communicator with Cartesian topology

coords Cartesian coordinates of process

`MPI_Cart_rank(comm, coords, rank)`

- comm: communicator with Cartesian topology
- coords: Cartesian coordinates of process
- rank: rank of process

Create communicator with graph topology

```
MPI_Dist_graph_create_adjacent(comm, indegree, sources, outdegree,
destinations, weights, info, reorder, newcomm)
```

Create communicator with graph topology

```
MPI_Dist_graph_create_adjacent(comm, indegree, sources, outdegree,
destinations, weights, info, reorder, newcomm)
```

comm old communicator

Create communicator with graph topology

```
MPI_Dist_graph_create_adjacent(comm, indegree, sources, outdegree,
destinations, weights, info, reorder, newcomm)
```

comm old communicator

indegree number of incoming edges

Create communicator with graph topology

```
MPI_Dist_graph_create_adjacent(comm, indegree, sources, outdegree,
destinations, weights, info, reorder, newcomm)
```

- comm: old communicator
- indegree: number of incoming edges
- sources: ranks of source nodes

Create communicator with graph topology

```
MPI_Dist_graph_create_adjacent(comm, indegree, sources, outdegree,
destinations, weights, info, reorder, newcomm)
```

comm	old communicator
indegree	number of incoming edges
sources	ranks of source nodes
outdegree	number of outgoing edges

Create communicator with graph topology

```
MPI_Dist_graph_create_adjacent(comm, indegree, sources, outdegree,
destinations, weights, info, reorder, newcomm)
```

comm	old communicator
indegree	number of incoming edges
sources	ranks of source nodes
outdegree	number of outgoing edges
destinations	ranks of destination nodes

Create communicator with graph topology

```
MPI_Dist_graph_create_adjacent(comm, indegree, sources, outdegree,
destinations, weights, info, reorder, newcomm)
```

comm	old communicator
indegree	number of incoming edges
sources	ranks of source nodes
outdegree	number of outgoing edges
destinations	ranks of destination nodes
weights	weights of outgoing edges

Create communicator with graph topology

```
MPI_Dist_graph_create_adjacent(comm, indegree, sources, outdegree,
destinations, weights, info, reorder, newcomm)
```

comm	old communicator
indegree	number of incoming edges
sources	ranks of source nodes
outdegree	number of outgoing edges
destinations	ranks of destination nodes
weights	weights of outgoing edges
info	hints

Create communicator with graph topology

```
MPI_Dist_graph_create_adjacent(comm, indegree, sources, outdegree,
destinations, weights, info, reorder, newcomm)
```

comm	old communicator
indegree	number of incoming edges
sources	ranks of source nodes
outdegree	number of outgoing edges
destinations	ranks of destination nodes
weights	weights of outgoing edges
info	hints
reorder	`true` if can reorder processes, `false` otherwise

Create communicator with graph topology

```
MPI_Dist_graph_create_adjacent(comm, indegree, sources, outdegree,
destinations, weights, info, reorder, newcomm)
```

comm	old communicator
indegree	number of incoming edges
sources	ranks of source nodes
outdegree	number of outgoing edges
destinations	ranks of destination nodes
weights	weights of outgoing edges
info	hints
reorder	`true` if can reorder processes, `false` otherwise
newcomm	new communicator with graph topology.

Create new communicators

```
...
MPI_Dist_graph_create_adjacent(comm, 4, neighbors[NW], 4,
    &neighbors[SE], MPI_UNWEIGHTED, NULL, false, &outcomm);

MPI_Dist_graph_create_adjacent(comm, 4, neighbors[SE], 4,
&neighbors[NW], MPI_UNWEIGHTED, NULL, false, &outcomm);]
...
```

Neighborhood collectives

```
MPI_Neighbor_allgather( sendbuf, sendcount, sendtype, recvbuf, recvcount,
recvtype, comm )
```

Neighborhood collectives

```
MPI_Neighbor_allgather( sendbuf, sendcount, sendtype, recvbuf, recvcount,
recvtype, comm )
```

sendbuf send buffer

Neighborhood collectives

```
MPI_Neighbor_allgather( sendbuf, sendcount, sendtype, recvbuf, recvcount,
recvtype, comm )
```

sendbuf send buffer

sendcount number of elements sent to each destination

Neighborhood collectives

```
MPI_Neighbor_allgather( sendbuf, sendcount, sendtype, recvbuf, recvcount,
recvtype, comm )
```

- sendbuf send buffer
- sendcount number of elements sent to each destination
- sendtype data type of sent elements

Neighborhood collectives

```
MPI_Neighbor_allgather( sendbuf, sendcount, sendtype, recvbuf, recvcount,
recvtype, comm )
```

- sendbuf send buffer
- sendcount number of elements sent to each destination
- sendtype data type of sent elements
- recvbuf receive buffer

Neighborhood collectives

```
MPI_Neighbor_allgather( sendbuf, sendcount, sendtype, recvbuf, recvcount,
recvtype, comm )
```

- sendbuf: send buffer
- sendcount: number of elements sent to each destination
- sendtype: data type of sent elements
- recvbuf: receive buffer
- recvcount: number of elements received from each source

Neighborhood collectives

```
MPI_Neighbor_allgather( sendbuf, sendcount, sendtype, recvbuf, recvcount,
recvtype, comm )
```

sendbuf	send buffer
sendcount	number of elements sent to each destination
sendtype	data type of sent elements
recvbuf	receive buffer
recvcount	number of elements received from each source
recvtype	data type of received elements

Neighborhood collectives

```
MPI_Neighbor_allgather( sendbuf, sendcount, sendtype, recvbuf, recvcount,
recvtype, comm )
```

sendbuf	send buffer
sendcount	number of elements sent to each destination
sendtype	data type of sent elements
recvbuf	receive buffer
recvcount	number of elements received from each source
recvtype	data type of received elements
comm	communicator with topology structure

Neighborhood collectives

`MPI_Neighbor_allgather( sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm )`

sendbuf	send buffer
sendcount	number of elements sent to each destination
sendtype	data type of sent elements
recvbuf	receive buffer
recvcount	number of elements received from each source
recvtype	data type of received elements
comm	communicator with topology structure

Same data sent to all destination processes; different data received from source processes

`MPI_Neighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm)`

Neighborhood collectives

Different data sent to different destinations

```
MPI_Neighbor_alltoall(sendbuf, sendcount,sendtype, recvbuf, recvcount,
recvtype, comm)
MPI_neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
recvcounts, rdispls, recvtype, comm)
```

Neighborhood collectives

Different data sent to different destinations

```
MPI_Neighbor_alltoall(sendbuf, sendcount,sendtype, recvbuf, recvcount,
recvtype, comm)
MPI_neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
recvcounts, rdispls, recvtype, comm)
```

sendbuf send buffer

Neighborhood collectives

Different data sent to different destinations

```
MPI_Neighbor_alltoall(sendbuf, sendcount,sendtype, recvbuf, recvcount,
recvtype, comm)
MPI_neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
recvcounts, rdispls, recvtype, comm)
```

sendbuf send buffer

sendcounts number of elements sent to each destination

Neighborhood collectives

Different data sent to different destinations

```
MPI_Neighbor_alltoall(sendbuf, sendcount,sendtype, recvbuf, recvcount,
recvtype, comm)
MPI_neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
recvcounts, rdispls, recvtype, comm)
```

- sendbuf send buffer
- sendcounts number of elements sent to each destination
- sdispls displacement to 1st element sent to each destination

Neighborhood collectives

Different data sent to different destinations

```
MPI_Neighbor_alltoall(sendbuf, sendcount,sendtype, recvbuf, recvcount,
recvtype, comm)
MPI_neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
recvcounts, rdispls, recvtype, comm)
```

- sendbuf: send buffer
- sendcounts: number of elements sent to each destination
- sdispls: displacement to 1st element sent to each destination
- sendtype: type of sent elements

Neighborhood collectives

Different data sent to different destinations

```
MPI_Neighbor_alltoall(sendbuf, sendcount,sendtype, recvbuf, recvcount,
recvtype, comm)
MPI_neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
recvcounts, rdispls, recvtype, comm)
```

- sendbuf: send buffer
- sendcounts: number of elements sent to each destination
- sdispls: displacement to 1st element sent to each destination
- sendtype: type of sent elements
- recvbuf: receive buffer

Neighborhood collectives

Different data sent to different destinations

```
MPI_Neighbor_alltoall(sendbuf, sendcount,sendtype, recvbuf, recvcount,
recvtype, comm)
MPI_neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
recvcounts, rdispls, recvtype, comm)
```

- sendbuf: send buffer
- sendcounts: number of elements sent to each destination
- sdispls: displacement to 1st element sent to each destination
- sendtype: type of sent elements
- recvbuf: receive buffer
- recvcounts: number of elements received from each destination

Neighborhood collectives

Different data sent to different destinations

```
MPI_Neighbor_alltoall(sendbuf, sendcount,sendtype, recvbuf, recvcount,
recvtype, comm)
MPI_neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
recvcounts, rdispls, recvtype, comm)
```

sendbuf	send buffer
sendcounts	number of elements sent to each destination
sdispls	displacement to 1st element sent to each destination
sendtype	type of sent elements
recvbuf	receive buffer
recvcounts	number of elements received from each destination
rdispls	displacement to 1st element received form each source

Neighborhood collectives

Different data sent to different destinations

```
MPI_Neighbor_alltoall(sendbuf, sendcount,sendtype, recvbuf, recvcount,
recvtype, comm)
MPI_neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
recvcounts, rdispls, recvtype, comm)
```

sendbuf	send buffer
sendcounts	number of elements sent to each destination
sdispls	displacement to 1st element sent to each destination
sendtype	type of sent elements
recvbuf	receive buffer
recvcounts	number of elements received from each destination
rdispls	displacement to 1st element received form each source
recvtype	tyoe of received elements

Neighborhood collectives

Different data sent to different destinations

```
MPI_Neighbor_alltoall(sendbuf, sendcount,sendtype, recvbuf, recvcount,
recvtype, comm)
MPI_neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
recvcounts, rdispls, recvtype, comm)
```

sendbuf	send buffer
sendcounts	number of elements sent to each destination
sdispls	displacement to 1st element sent to each destination
sendtype	type of sent elements
recvbuf	receive buffer
recvcounts	number of elements received from each destination
rdispls	displacement to 1st element received form each source
recvtype	tyoe of received elements
comm	communicator

Neighborhood collectives

Different data sent to different destinations

```
MPI_Neighbor_alltoall(sendbuf, sendcount,sendtype, recvbuf, recvcount,
recvtype, comm)
MPI_neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
recvcounts, rdispls, recvtype, comm)
```

sendbuf	send buffer
sendcounts	number of elements sent to each destination
sdispls	displacement to 1st element sent to each destination
sendtype	type of sent elements
recvbuf	receive buffer
recvcounts	number of elements received from each destination
rdispls	displacement to 1st element received form each source
recvtype	tyoe of received elements
comm	communicator

Neighborhood collectives

Different data sent to different destinations

```
MPI_Neighbor_alltoall(sendbuf, sendcount,sendtype, recvbuf, recvcount,
recvtype, comm)
MPI_neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
recvcounts, rdispls, recvtype, comm)
```

sendbuf	send buffer
sendcounts	number of elements sent to each destination
sdispls	displacement to 1st element sent to each destination
sendtype	type of sent elements
recvbuf	receive buffer
recvcounts	number of elements received from each destination
rdispls	displacement to 1st element received form each source
recvtype	tyoe of received elements
comm	communicator

Neighborhood collectives

Different data sent to different destinations

```
MPI_Neighbor_alltoall(sendbuf, sendcount,sendtype, recvbuf, recvcount,
recvtype, comm)
MPI_neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
recvcounts, rdispls, recvtype, comm)
```

sendbuf	send buffer
sendcounts	number of elements sent to each destination
sdispls	displacement to 1st element sent to each destination
sendtype	type of sent elements
recvbuf	receive buffer
recvcounts	number of elements received from each destination
rdispls	displacement to 1st element received form each source
recvtype	tyoe of received elements
comm	communicator

Neighborhood collectives

Different data sent to different destinations

```
MPI_Neighbor_alltoall(sendbuf, sendcount,sendtype, recvbuf, recvcount,
recvtype, comm)
MPI_neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
recvcounts, rdispls, recvtype, comm)
```

sendbuf	send buffer
sendcounts	number of elements sent to each destination
sdispls	displacement to 1st element sent to each destination
sendtype	type of sent elements
recvbuf	receive buffer
recvcounts	number of elements received from each destination
rdispls	displacement to 1st element received form each source
recvtype	tyoe of received elements
comm	communicator

Neighborhood collectives

Different data sent to different destinations

```
MPI_Neighbor_alltoall(sendbuf, sendcount,sendtype, recvbuf, recvcount,
recvtype, comm)
MPI_neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
recvcounts, rdispls, recvtype, comm)
```

sendbuf	send buffer
sendcounts	number of elements sent to each destination
sdispls	displacement to 1st element sent to each destination
sendtype	type of sent elements
recvbuf	receive buffer
recvcounts	number of elements received from each destination
rdispls	displacement to 1st element received form each source
recvtype	tyoe of received elements
comm	communicator

Neighborhood collectives

Different data sent to different destinations

```
MPI_Neighbor_alltoall(sendbuf, sendcount,sendtype, recvbuf, recvcount,
recvtype, comm)
MPI_neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
recvcounts, rdispls, recvtype, comm)
```

sendbuf	send buffer
sendcounts	number of elements sent to each destination
sdispls	displacement to 1st element sent to each destination
sendtype	type of sent elements
recvbuf	receive buffer
recvcounts	number of elements received from each destination
rdispls	displacement to 1st element received form each source
recvtype	tyoe of received elements
comm	communicator

Neighborhood collectives

Different data sent to different destinations

```
MPI_Neighbor_alltoall(sendbuf, sendcount,sendtype, recvbuf, recvcount,
recvtype, comm)
MPI_neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
recvcounts, rdispls, recvtype, comm)
```

sendbuf	send buffer
sendcounts	number of elements sent to each destination
sdispls	displacement to 1st element sent to each destination
sendtype	type of sent elements
recvbuf	receive buffer
recvcounts	number of elements received from each destination
rdispls	displacement to 1st element received form each source
recvtype	tyoe of received elements
comm	communicator

Neighborhood collectives

Different data sent to different destinations

```
MPI_Neighbor_alltoall(sendbuf, sendcount,sendtype, recvbuf, recvcount,
recvtype, comm)
MPI_neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
recvcounts, rdispls, recvtype, comm)
```

sendbuf	send buffer
sendcounts	number of elements sent to each destination
sdispls	displacement to 1st element sent to each destination
sendtype	type of sent elements
recvbuf	receive buffer
recvcounts	number of elements received from each destination
rdispls	displacement to 1st element received form each source
recvtype	tyoe of received elements
comm	communicator

Neighborhood collectives

Different data sent to different destinations

```
MPI_Neighbor_alltoall(sendbuf, sendcount,sendtype, recvbuf, recvcount,
recvtype, comm)
MPI_neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
recvcounts, rdispls, recvtype, comm)
```

sendbuf	send buffer
sendcounts	number of elements sent to each destination
sdispls	displacement to 1st element sent to each destination
sendtype	type of sent elements
recvbuf	receive buffer
recvcounts	number of elements received from each destination
rdispls	displacement to 1st element received form each source
recvtype	tyoe of received elements
comm	communicator

Neighborhood collectives

Different data sent to different destinations

```
MPI_Neighbor_alltoall(sendbuf, sendcount,sendtype, recvbuf, recvcount,
recvtype, comm)
MPI_neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
recvcounts, rdispls, recvtype, comm)
```

sendbuf	send buffer
sendcounts	number of elements sent to each destination
sdispls	displacement to 1st element sent to each destination
sendtype	type of sent elements
recvbuf	receive buffer
recvcounts	number of elements received from each destination
rdispls	displacement to 1st element received form each source
recvtype	tyoe of received elements
comm	communicator

Neighborhood collectives

Different data sent to different destinations

```
MPI_Neighbor_alltoall(sendbuf, sendcount,sendtype, recvbuf, recvcount,
recvtype, comm)
MPI_neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
recvcounts, rdispls, recvtype, comm)
```

sendbuf	send buffer
sendcounts	number of elements sent to each destination
sdispls	displacement to 1st element sent to each destination
sendtype	type of sent elements
recvbuf	receive buffer
recvcounts	number of elements received from each destination
rdispls	displacement to 1st element received form each source
recvtype	tyoe of received elements
comm	communicator

Neighborhood collectives

Different data sent to different destinations

```
MPI_Neighbor_alltoall(sendbuf, sendcount,sendtype, recvbuf, recvcount,
recvtype, comm)
MPI_neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
recvcounts, rdispls, recvtype, comm)
```

sendbuf	send buffer
sendcounts	number of elements sent to each destination
sdispls	displacement to 1st element sent to each destination
sendtype	type of sent elements
recvbuf	receive buffer
recvcounts	number of elements received from each destination
rdispls	displacement to 1st element received form each source
recvtype	tyoe of received elements
comm	communicator

Neighborhood collectives

Different data sent to different destinations

```
MPI_Neighbor_alltoall(sendbuf, sendcount,sendtype, recvbuf, recvcount,
recvtype, comm)
MPI_neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
recvcounts, rdispls, recvtype, comm)
```

sendbuf	send buffer
sendcounts	number of elements sent to each destination
sdispls	displacement to 1st element sent to each destination
sendtype	type of sent elements
recvbuf	receive buffer
recvcounts	number of elements received from each destination
rdispls	displacement to 1st element received form each source
recvtype	tyoe of received elements
comm	communicator

Neighborhood collectives

Different data sent to different destinations

```
MPI_Neighbor_alltoall(sendbuf, sendcount,sendtype, recvbuf, recvcount,
recvtype, comm)
MPI_neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
recvcounts, rdispls, recvtype, comm)
```

sendbuf	send buffer
sendcounts	number of elements sent to each destination
sdispls	displacement to 1st element sent to each destination
sendtype	type of sent elements
recvbuf	receive buffer
recvcounts	number of elements received from each destination
rdispls	displacement to 1st element received form each source
recvtype	tyoe of received elements
comm	communicator

Neighborhood collectives

Different data sent to different destinations

```
MPI_Neighbor_alltoall(sendbuf, sendcount,sendtype, recvbuf, recvcount,
recvtype, comm)
MPI_neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
recvcounts, rdispls, recvtype, comm)
```

sendbuf	send buffer
sendcounts	number of elements sent to each destination
sdispls	displacement to 1st element sent to each destination
sendtype	type of sent elements
recvbuf	receive buffer
recvcounts	number of elements received from each destination
rdispls	displacement to 1st element received form each source
recvtype	tyoe of received elements
comm	communicator

Communication

```
\* numatoms -- number of atoms *\
\* loc[k][i] -- coordinates of local atom i *\
\*  (k=0) or i-th atom received from k-th neighbor (k>0) *\
\* mass, charge, force -- ibid *\

int atomcounts[4]; \* number of atoms coords received *\
int displs[4]; \* displacements *\
for(i=0; i<4;i++)
 displs[i] = 2*i*MAXATOMS
\* communicate atom counts *\
MPI_Neighbor_allgather(&numatoms, 1, MPI_INT,
    atomcounts, 1, MPI_INT, incomm);

\* communicate atom locations *\
for(i=0;i<4;i++)
 atomcounts[i]*=2;
MPI_Neighbor_allgatherv(loc, 2*numatoms, MPI_DOUBLE,
    loc[MAXATOMS], atomcounts, MPI_DOUBLE, incomm);
... \* communicate masses and charges *\\
```

Communication

```
... \* compute forces *\
\* communicate back forces *\
double outforces[5][MAXATOMS][2] \* arrray of locally computed forces *\
double inforces[5][MAXATOMS][2] \* array for received forces *\
MPI_Neighbors_alltoallv(outforces[1], atomcounts, displs, MPI_DOUBLE,
    inforces[1], 2*numatoms, displs, MPI_DOUBLE, outcomm);
\* sum forces *\
for(i=0; i<numatoms, i++) {
 v_assign_v(outforce[0][i],inforce[0][i]);
 for(j=1,j<5;j++)
   v_plus_v(inforce[j][i],inforce[0][i],inforce[0][i]);
```