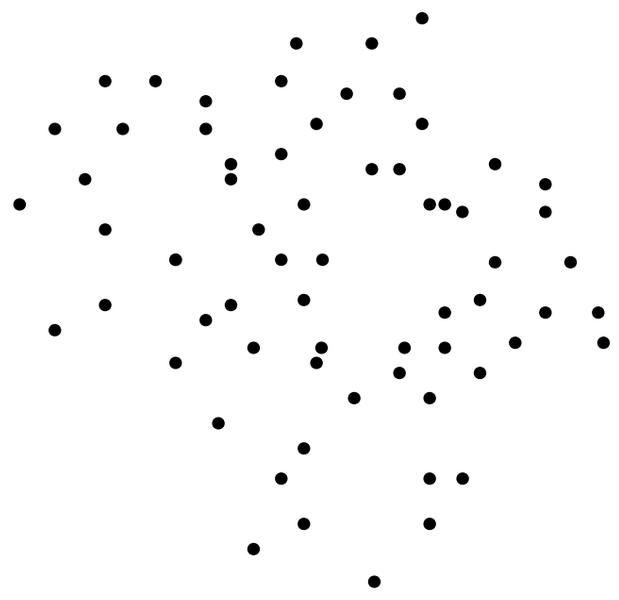
N-Body II: MPI

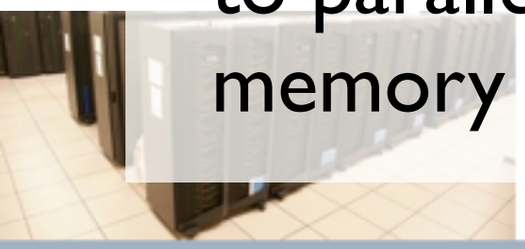
SciNet Parallel Scientific Computing Course

Aug 31 - Sept 4, 2009

Decomposing onto different processors

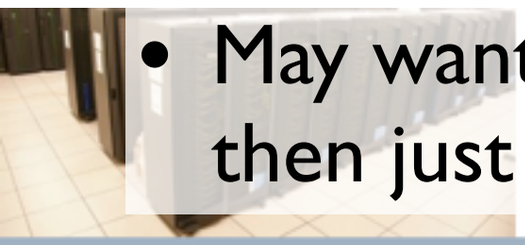


- Direct summation (N^2) - each particle needs to know about all other particles
- *No* locality possible
- Inherently a difficult problem to parallelize in distributed memory



Make a particle MPI type

- We're going to be passing particle information back and forth quite a bit
- Make an MPI type at start so things are easier
- May want to adjust this later; then just change type



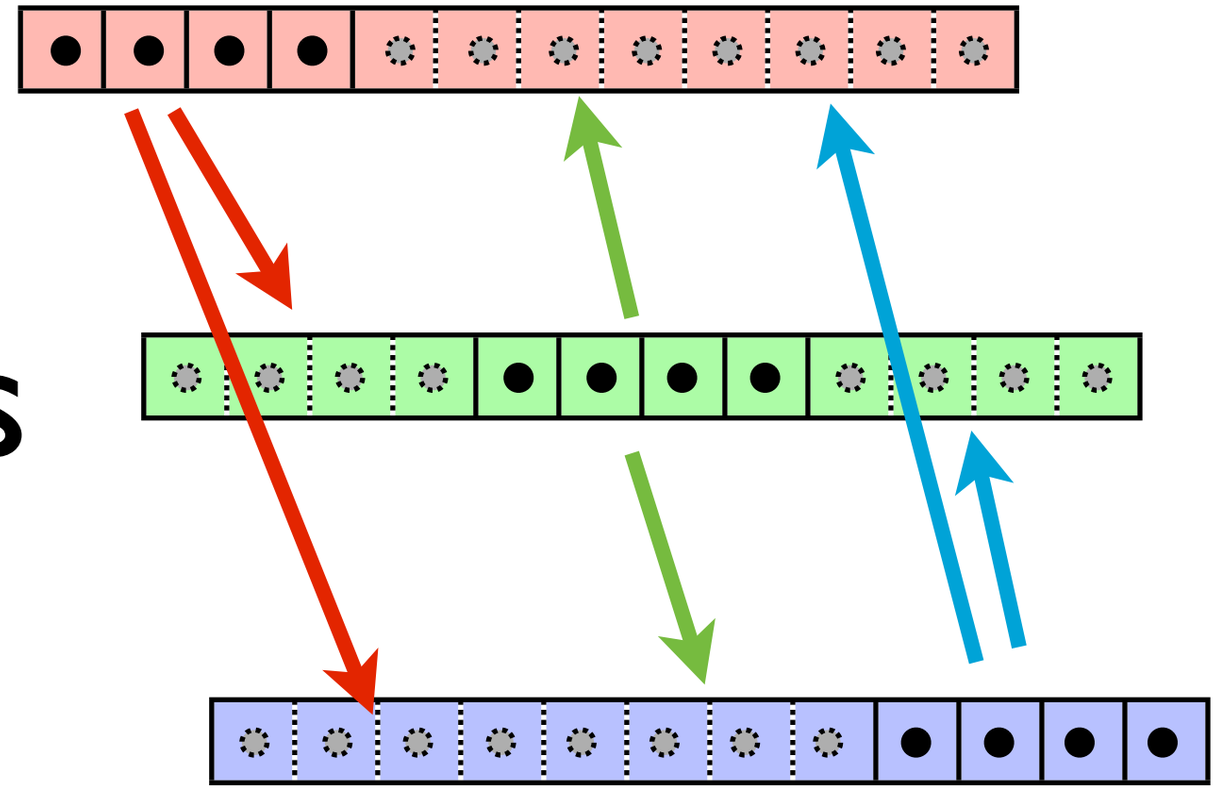
```
#ifdef USEFLOAT
        typedef float NType;
        #define MPI_NType MPI_FLOAT
#else
        typedef double NType;
        #define MPI_NType MPI_DOUBLE_PRECISION
#endif
```

pca_utils.h

```
typedef struct nbody_struct_s {
  NType x[NDIM];    /*the particle positions*/
  NType v[NDIM];    /*the particle velocities*/
  NType f[NDIM];    /*the forces on the particles*/
  NType mass;
  NType PE;         /* potential energy */
} NBody;

MPI_Datatype MPI_Particle;  /* derived data type for above */
```

```
  /* Create Particle Type */

  MPI_Type_contiguous( 3*NDIM+2, MPI_NType, &MPI_Particle );
  MPI_Type_commit ( &MPI_Particle );
```

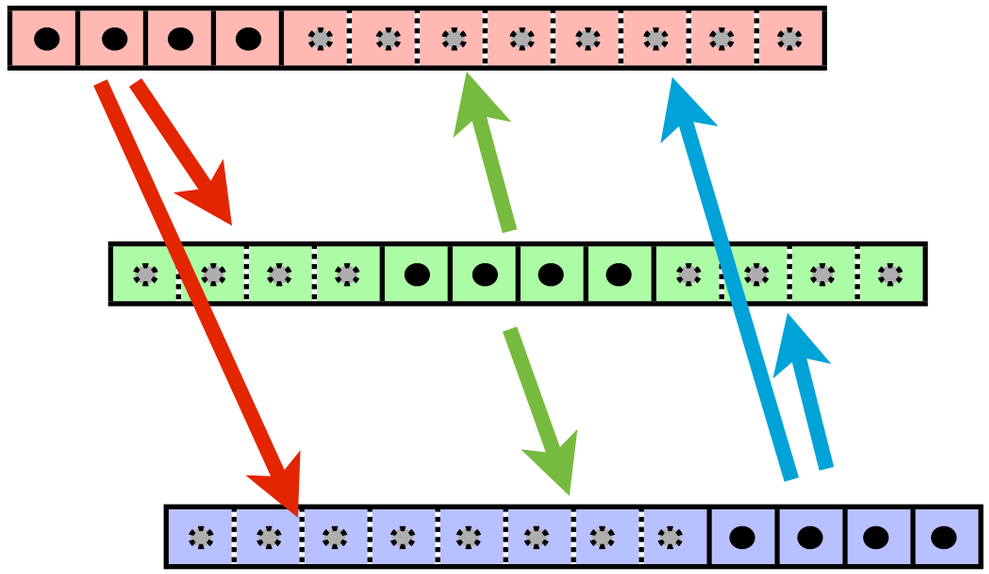
First go: Everyone sees everything

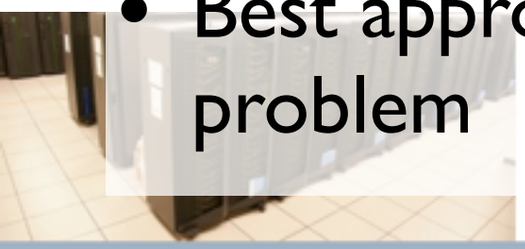


- Directly analogous to OpenMP approach
- Just work on our own particles
- Send everyone our particles afterwards

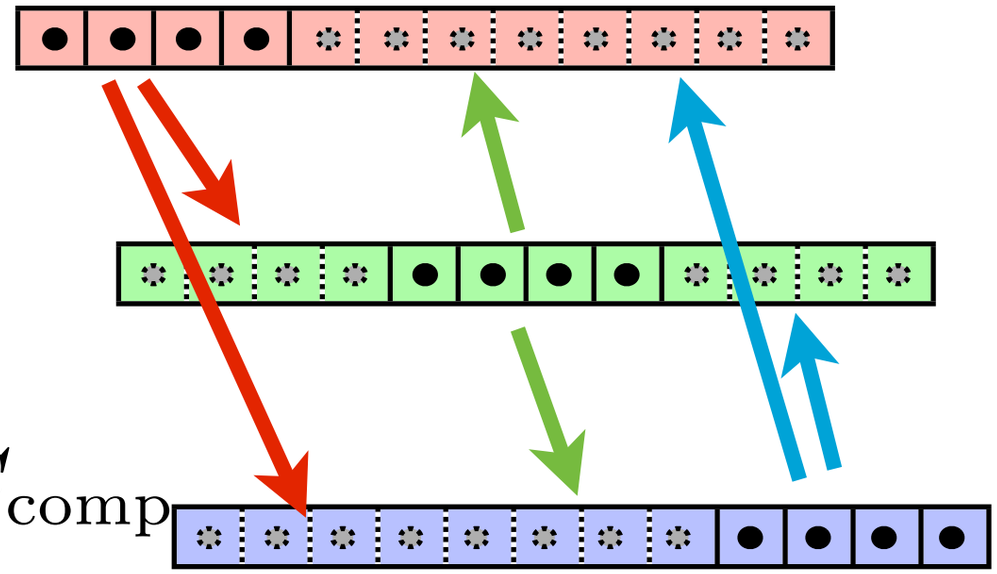
Terrible Idea (I)

- Requires the entire problem to fit in the memory of each node.
- In general, you can't do that ($10^{10\text{-}11}$ particle simulation; Pen)
- No good for MD, astrophysics but could be useful in other areas (few bodies, complicated interactions) - agent-based simulation
- Best approach depends on your problem





Terrible Idea (II)



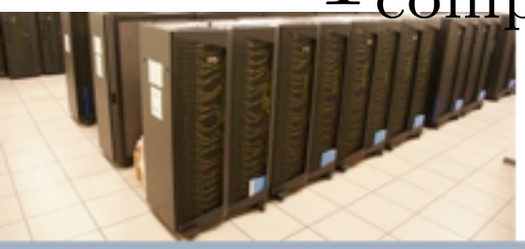
$$T_{\text{comp}} \sim c_{\text{grav}} \left(\frac{N}{P}\right) N C_{\text{comp}}$$

$$= c_{\text{grav}} \frac{N^2}{P} C_{\text{comp}}$$

$$T_{\text{comm}} \sim c_{\text{particle}} \frac{N}{P} (P-1) C_{\text{comm}}$$

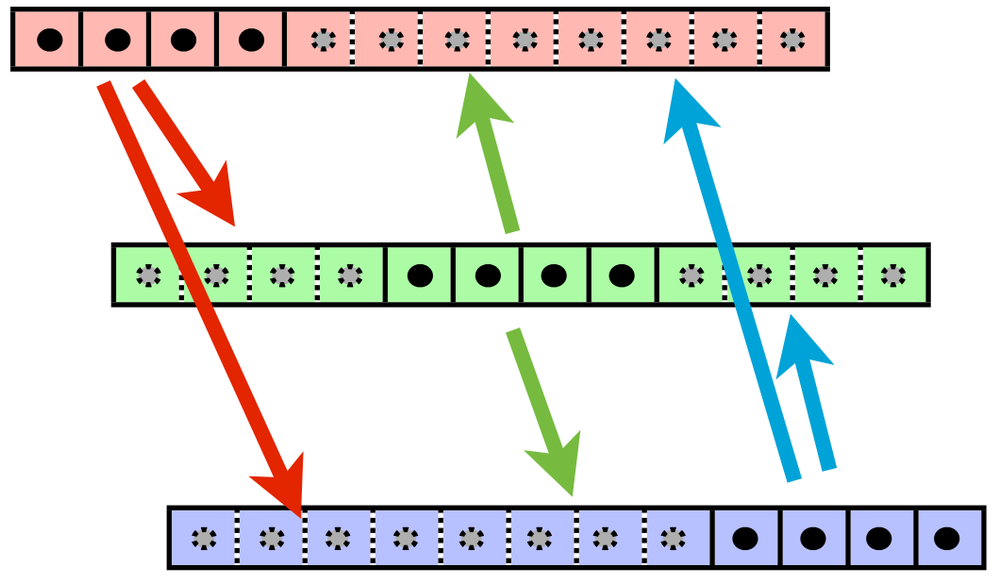
$$\approx c_{\text{particle}} N C_{\text{comm}}$$

$$\frac{T_{\text{comm}}}{T_{\text{comp}}} \approx \frac{c_{\text{particle}}}{c_{\text{grav}}} \frac{1}{N} P \frac{C_{\text{comm}}}{C_{\text{comp}}}$$



Since N is fixed, as P goes up, this fraction gets worse and worse

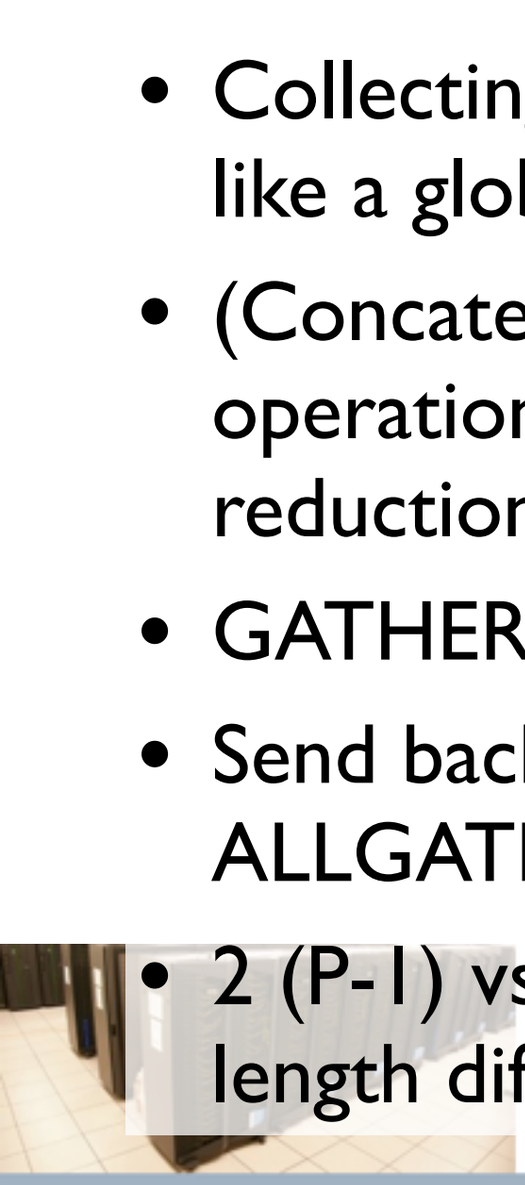
Terrible Idea (III)

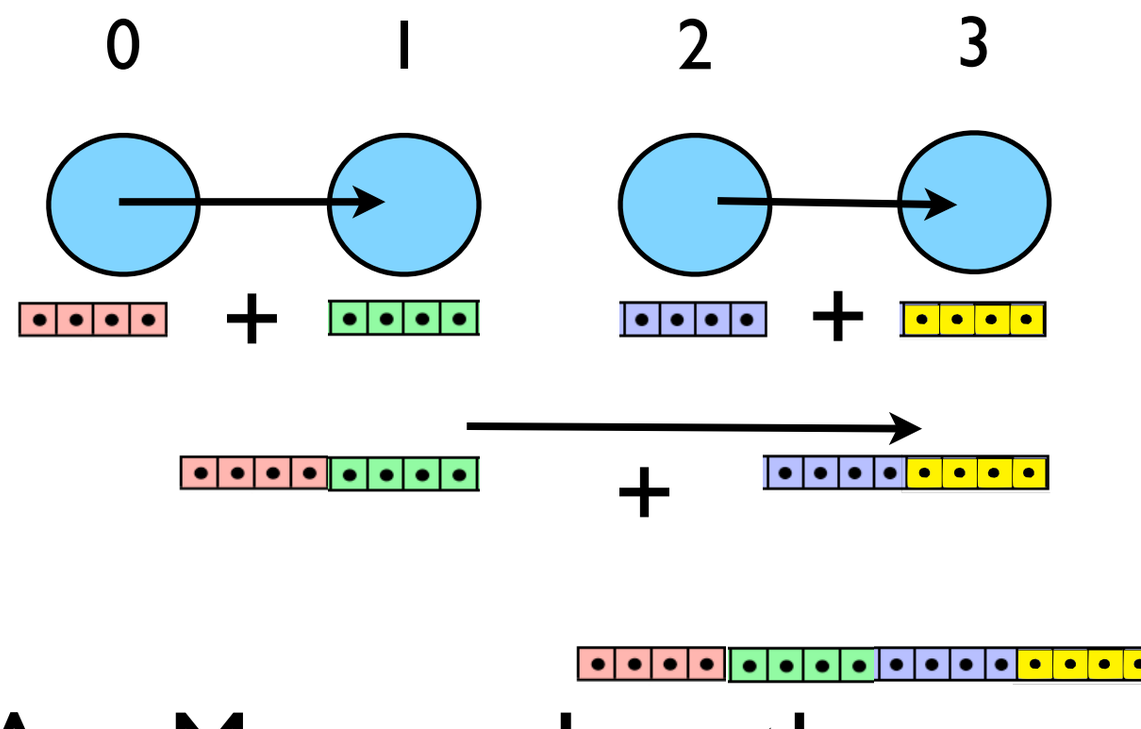


- Wastes computation.
- Proc 0 and Proc 2 both calculate the force between particle 1 and particle 11.

Can address (II) a little

- Collecting everyone's data is like a global sum
- (Concatenation is the sort of operation that allows reduction)
- GATHER operation
- Send back the results: ALLGATHER
- 2 (P-1) vs P^2 messages, but length differs

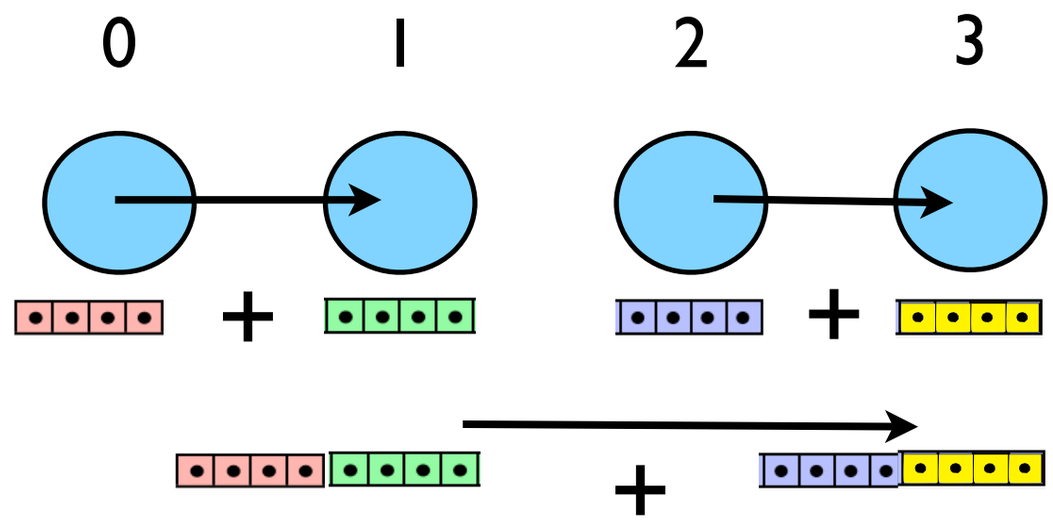




Avg Message Length =
$(N/2 \log_2 P)/(P-1)$
$\sim N + N/P \log_2(P)$

Total sent ~
$2 N \log_2(P)$ vs $N P$

Can address (l) a little

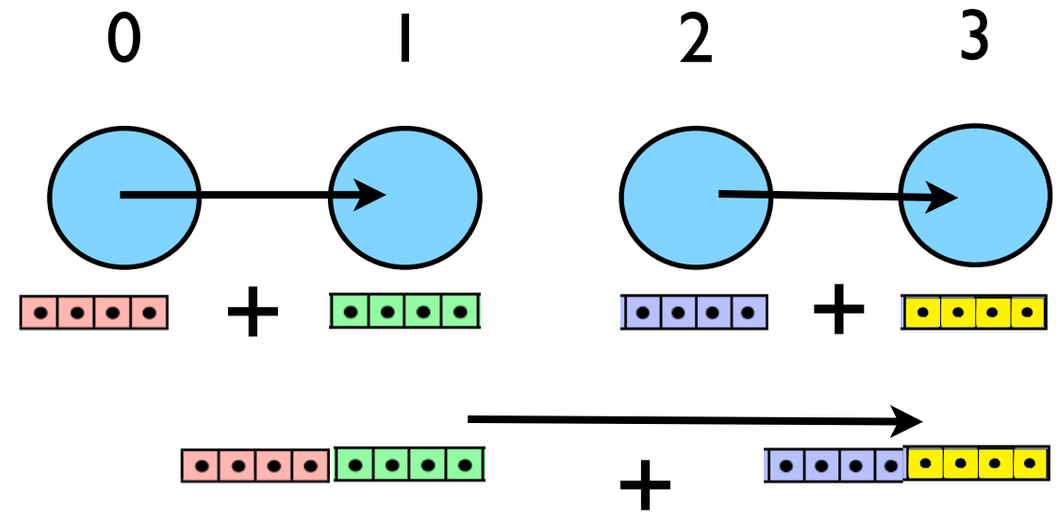


$$T_{\text{comp}} = c_{\text{grav}} \frac{N^2}{P} C_{\text{comp}}$$

$$T_{\text{comm}} \sim c_{\text{particle}} 2N \frac{\log_2 P}{P} C_{\text{comm}}$$

$$\frac{T_{\text{comm}}}{T_{\text{comp}}} \approx \frac{c_{\text{particle}}}{c_{\text{grav}}} \frac{2}{N} \log_2(P) \frac{C_{\text{comm}}}{C_{\text{comp}}}$$

Another collective operation



Stuff you're sending

How Much

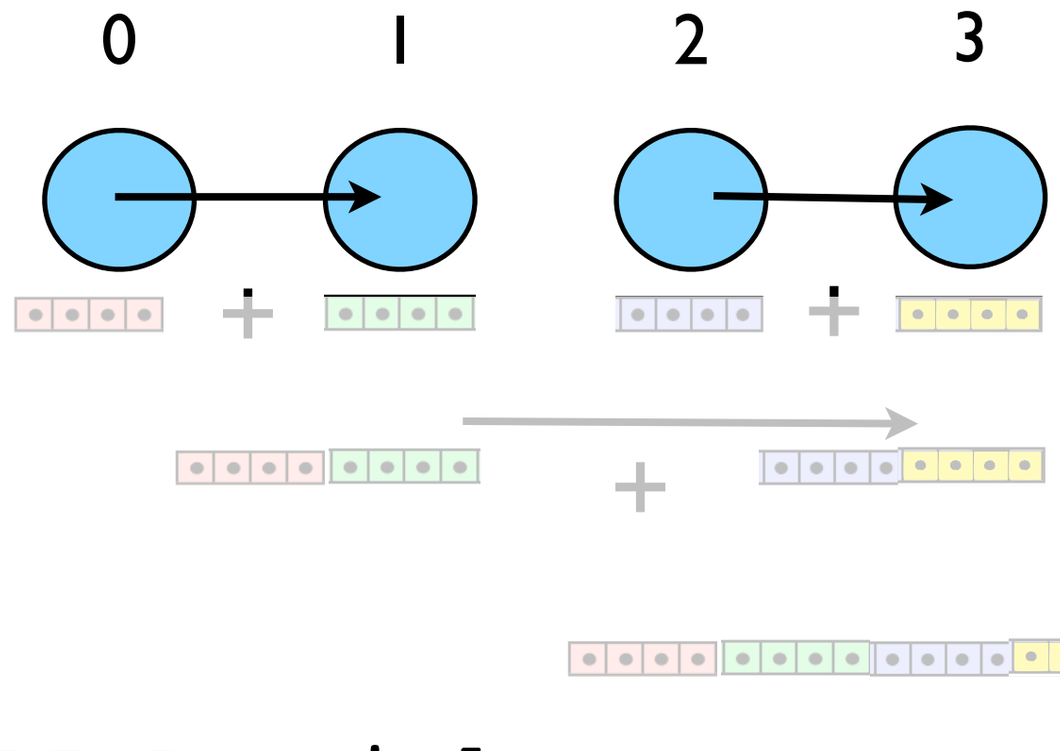
What Type

```
int MPI_Gather ( void *sendbuf, int sendcnt, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 int root, MPI_Comm comm )
```

Place you're receiving

Who's getting all the data

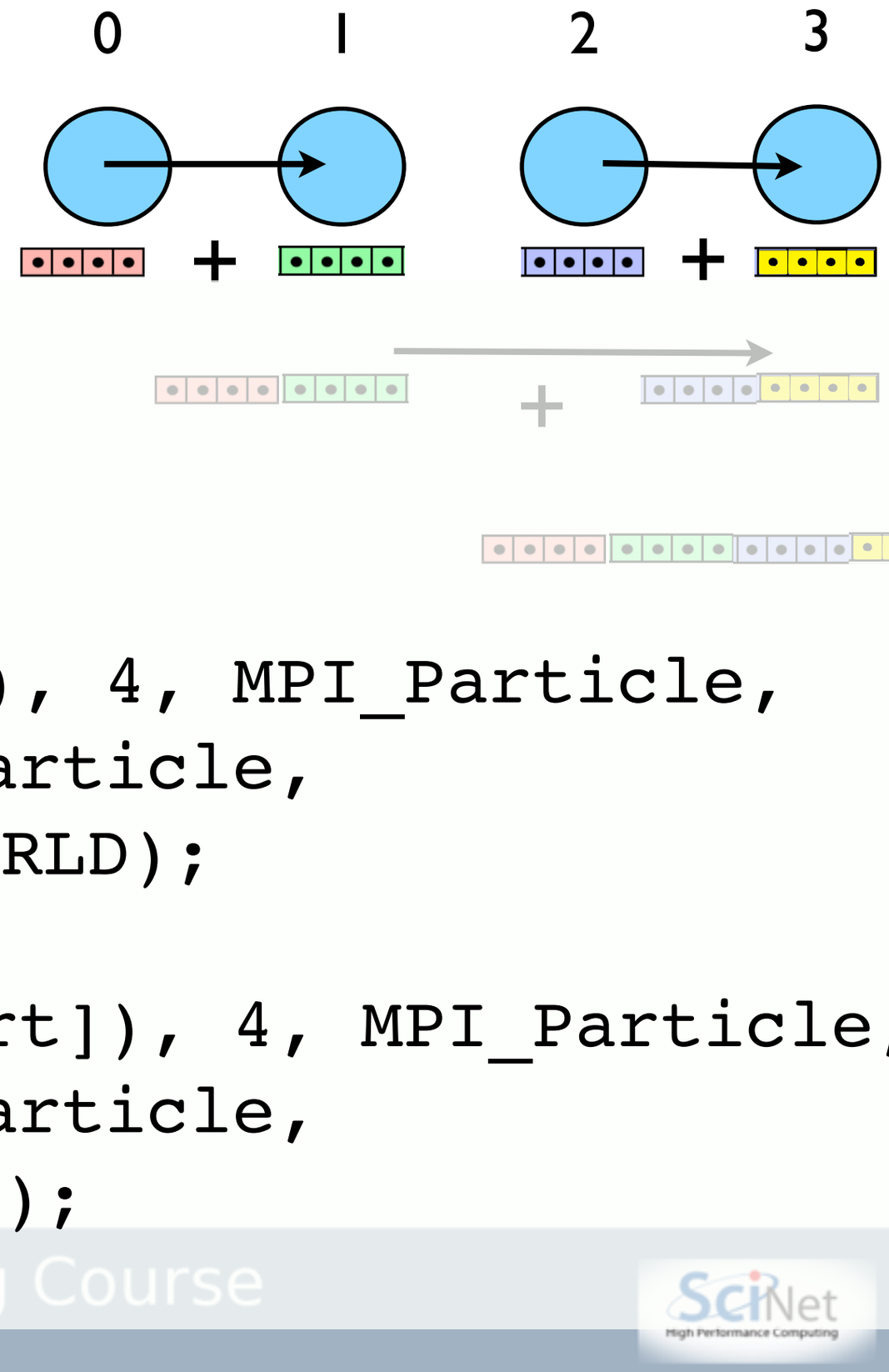
Another collective operation




```
NBody justmydata[4];
NBody globaldata[16];
MPI_Datatype MPI_Particle;
```

```
MPI_Gather(justmydata, 4, MPI_Particle,
           globaldata, 4, MPI_Particle,
            3, MPI_COMM_WORLD);

MPI_Allgather(justmydata, 4, MPI_Particle,
           globaldata, 4, MPI_Particle,
            MPI_COMM_WORLD);
```

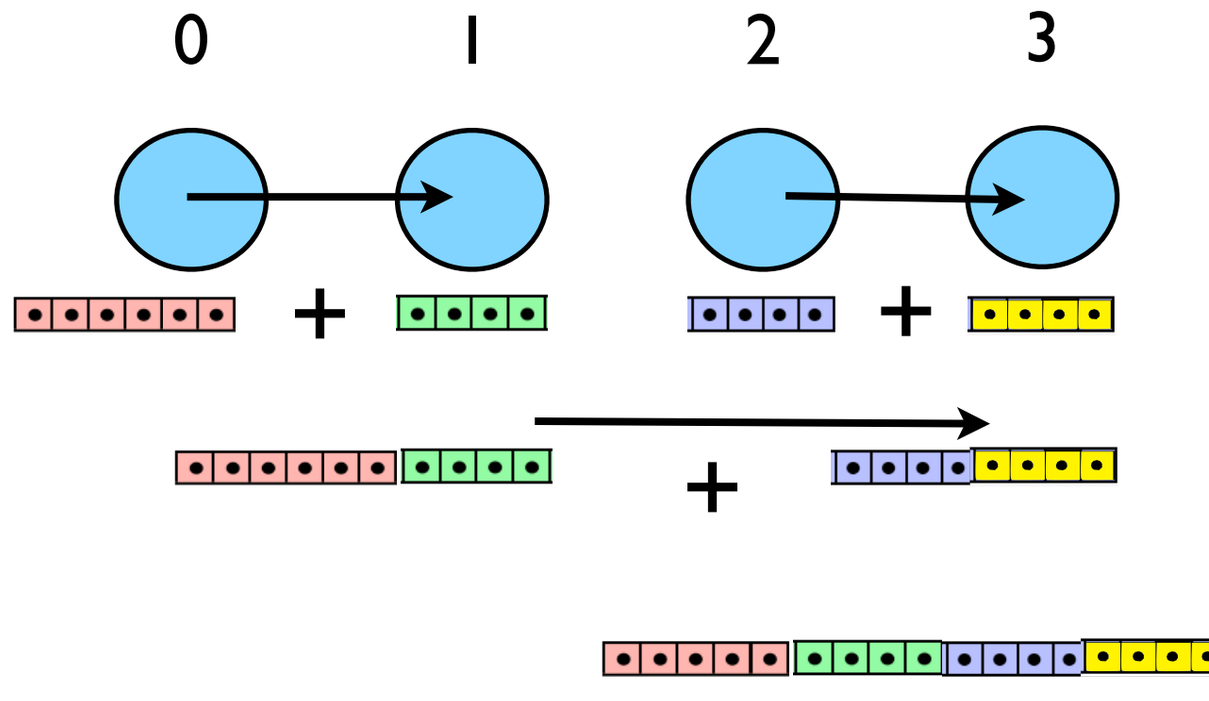
Another collective operation




```
NBody data[4*size];
int mystart=4*rank;
MPI_Datatype MPI_Particle;
```

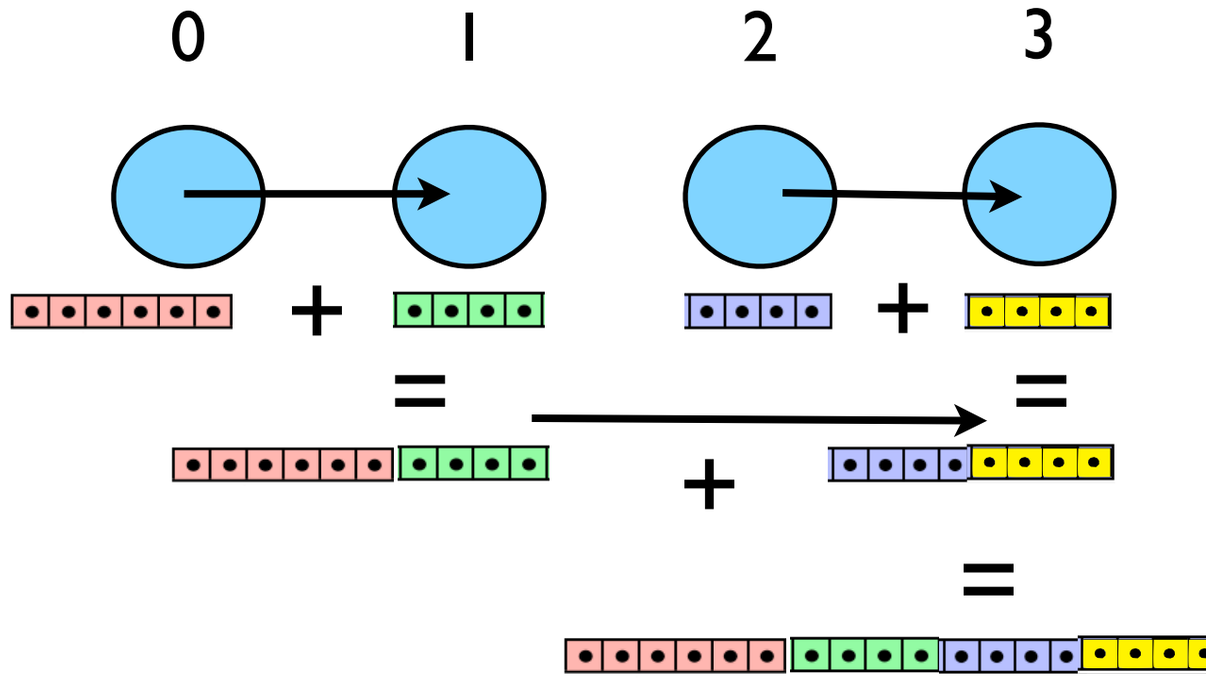
```
MPI_Gather(&(data[mystart]), 4, MPI_Particle,
           data, 4, MPI_Particle,
            3, MPI_COMM_WORLD);

MPI_Allgather(&(data[mystart]), 4, MPI_Particle,
           data, 4, MPI_Particle,
            MPI_COMM_WORLD);
```

What if not same # of particles?



- When everyone has same # of particles, easy to figure out where one processor's piece goes in the global array
- Otherwise, need to know how many each has and where their chunk should go in the global array

What if not same # of particles?

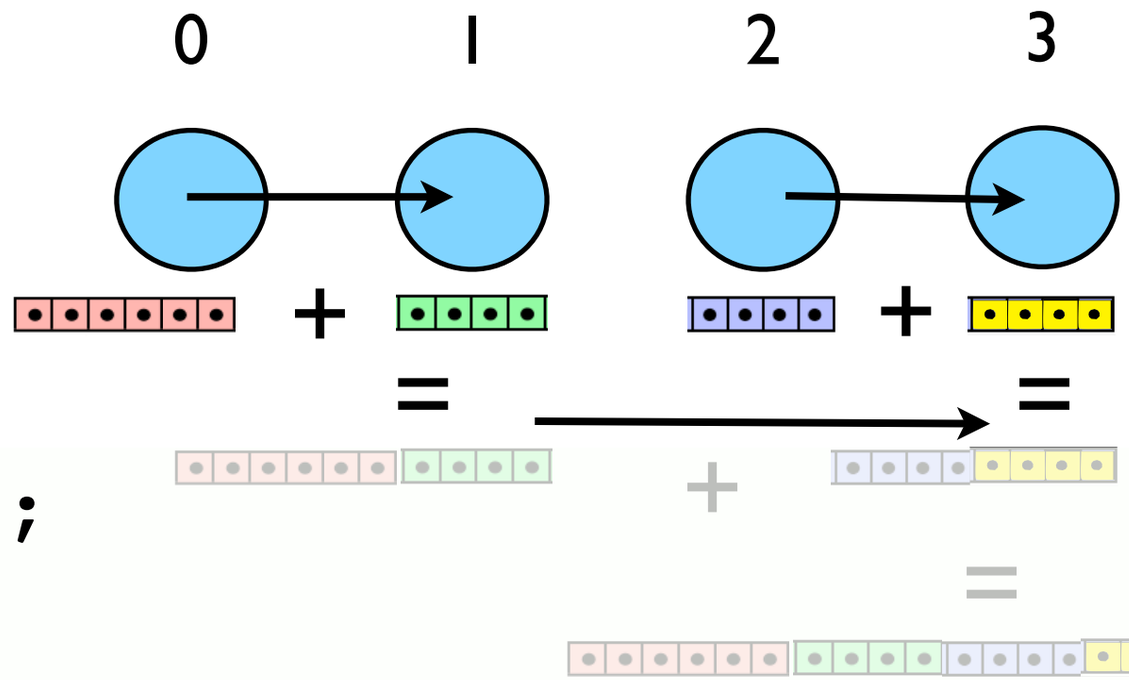



```
int MPI_Allgatherv ( void *sendbuf, int sendcount, MPI_Datatype sendtype,
                     void *recvbuf, int *recvcounts, int *displs,
                     MPI_Datatype recvtype, MPI_Comm comm )
```

Array of counts; eg {6,4,4,4}

Where they should go; eg {0,6,10,14}

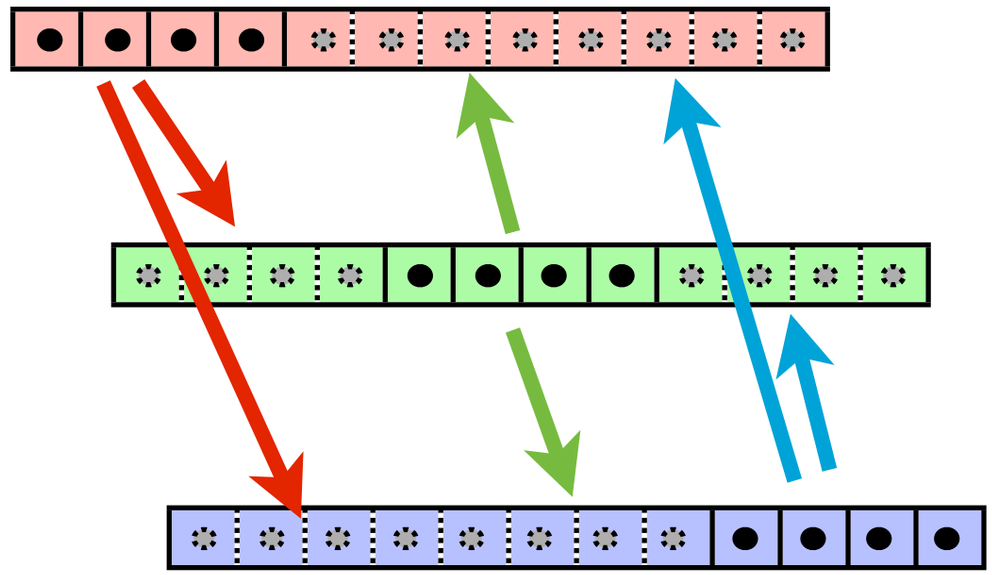
How would we get this data? Allgather!




```
int counts[size], disp[size];
int mystart=..., mynump=...;
```

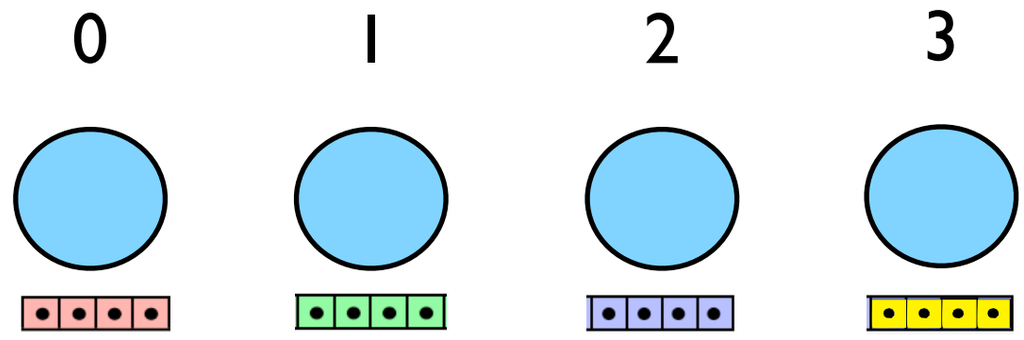
```
MPI_Allgather(&mynump, 1, MPI_INT,
              counts, 1, MPI_INT, MPI_COMM_WORLD);
disp[i]=0;
for (i=1;i<size;i++) disp[i]=disp[i-1]+counts[i];

MPI_Allgatherv(&(data[mystart]), mynump, MPI_Particle,
               data, counts, disp,
                MPI_COMM_WORLD);
```

Other stuff about the nbody code



- At least plotting remains easy.
- Generally n-body codes keep track of things like global energy as a diagnostic
- We have a local energy we calculate on our particles;
- Should communicate that to sum up over all processors.

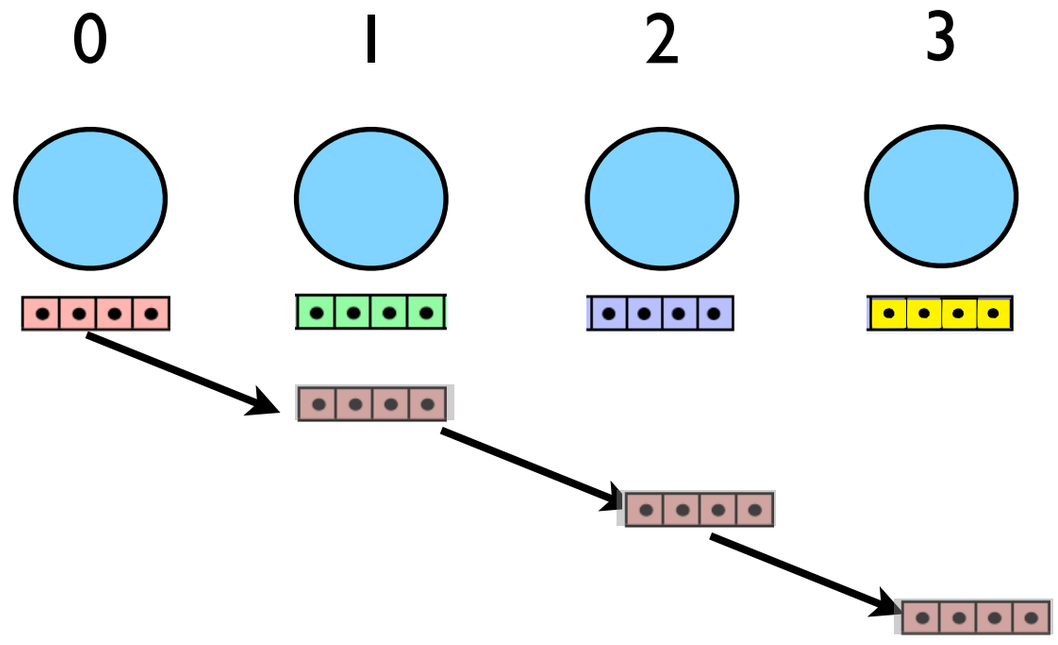
Problem (I) remains -- memory



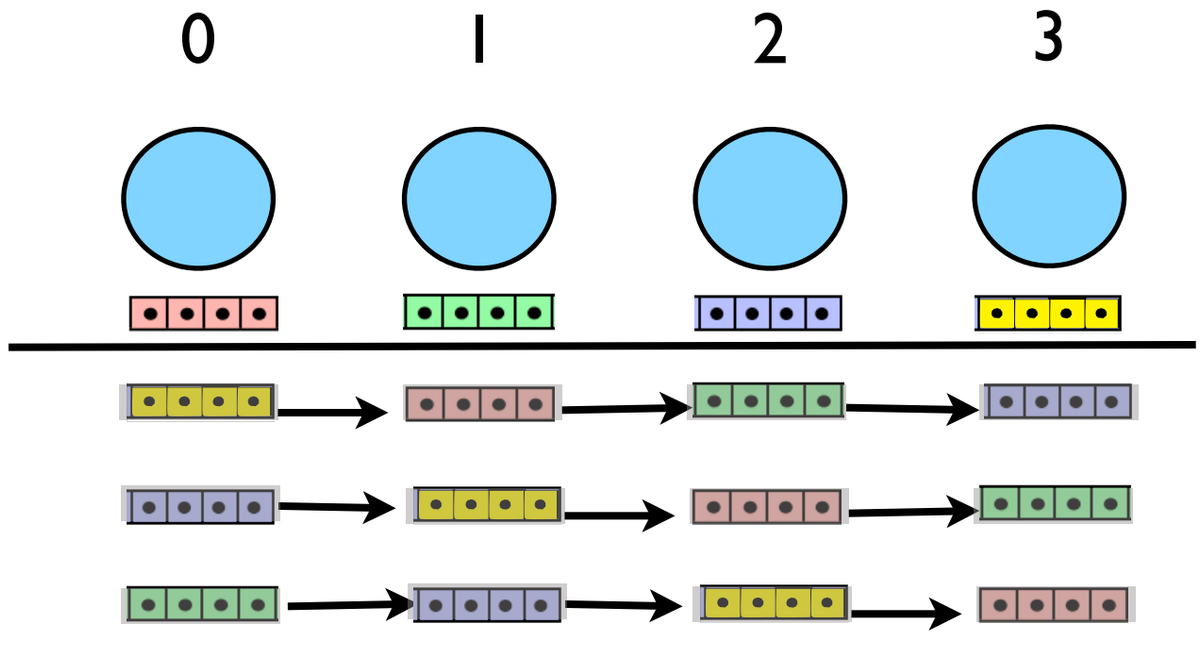
- How do we avoid this?
- For direct summation, we need to be able to see all particles;
- But not necessarily at once.

Pipeline

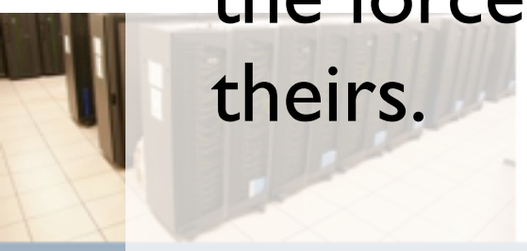
- 0 sends chunk of its particles to 1, which computes on it, then 2, then 3
- Then 1 does the same thing, etc.
- Size of chunk: tradeoff - memory usage vs. number of messages
- Let's just assume all particles go at once, and all have same # of particles (bookkeeping)



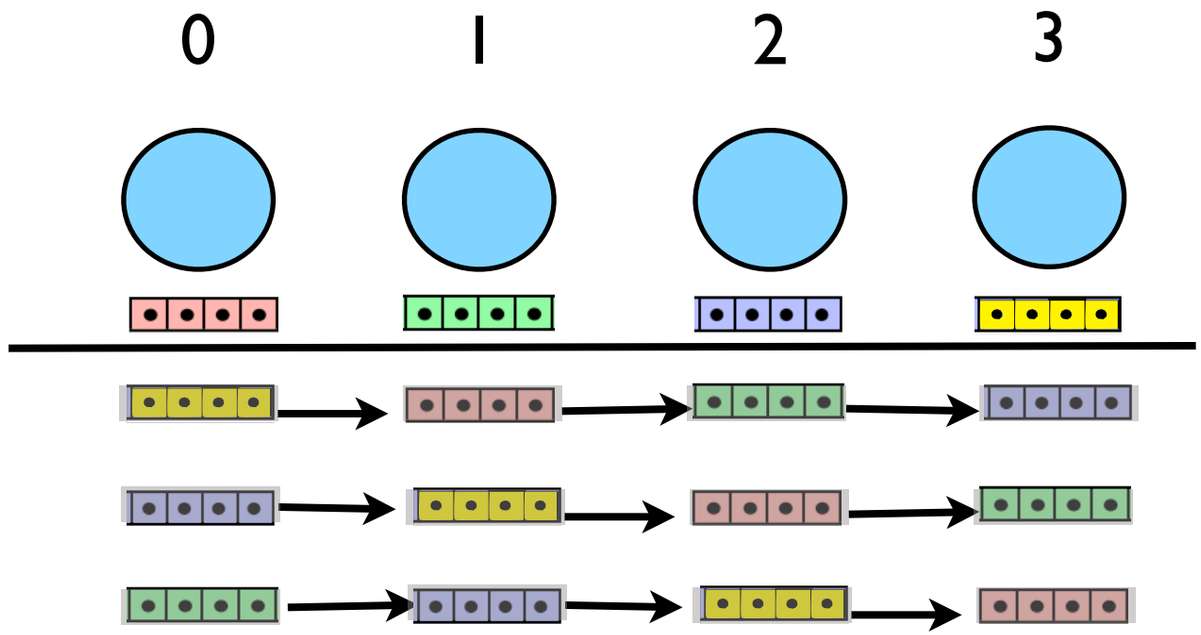
Pipeline



- No need to wait for 0s chunk to be done!
- Everyone sends their chunk forward, and keeps getting passed along.
- Compute local forces first, then start pipeline, and foreach (P-1) chunks compute the forces on your particles by theirs.



Pipeline



- Work unchanged

$$T_{\text{comp}} = c_{\text{grav}} \frac{N^2}{P} C_{\text{comp}}$$

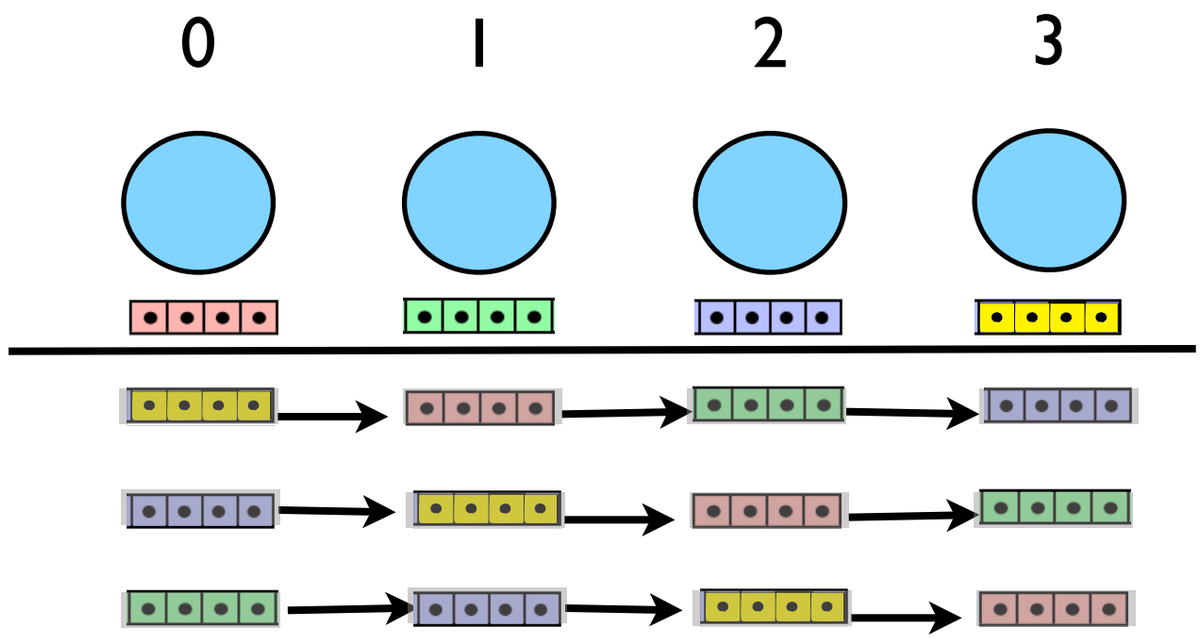
- Communication - each process sends (P-1) messages of length (N/P)

$$T_{\text{comm}} = c_{\text{particle}}(P-1)\frac{N}{P}C_{\text{comm}} \rightarrow c_{\text{particle}} N C_{\text{comm}}$$

$$\frac{T_{\text{comm}}}{T_{\text{comp}}} \approx \frac{c_{\text{particle}}}{c_{\text{grav}}} \frac{1}{N} P \frac{C_{\text{comm}}}{C_{\text{comp}}}$$

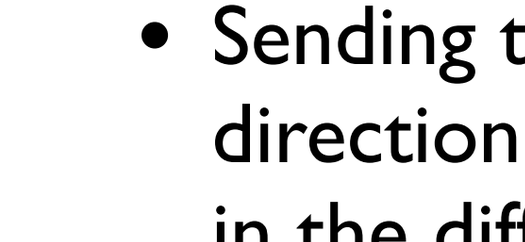
Pipeline

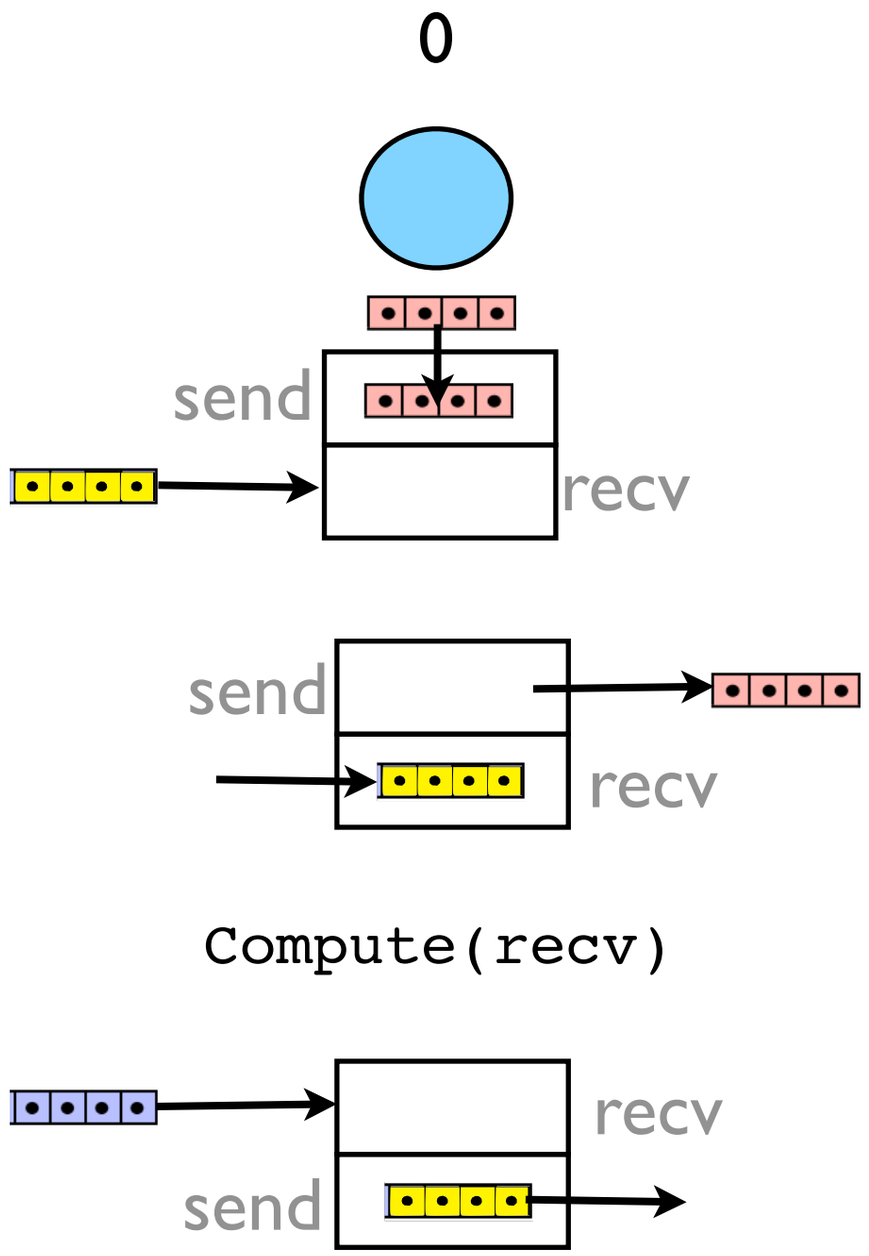
- Back to the first approach.
- *But* can do much bigger problems
- If we're filling memory, then N ~ P, and T_{comm}/T_{comp} is constant (yay!)
- With previous approach, maximum problem size is fixed by one processor's memory.



Pipeline

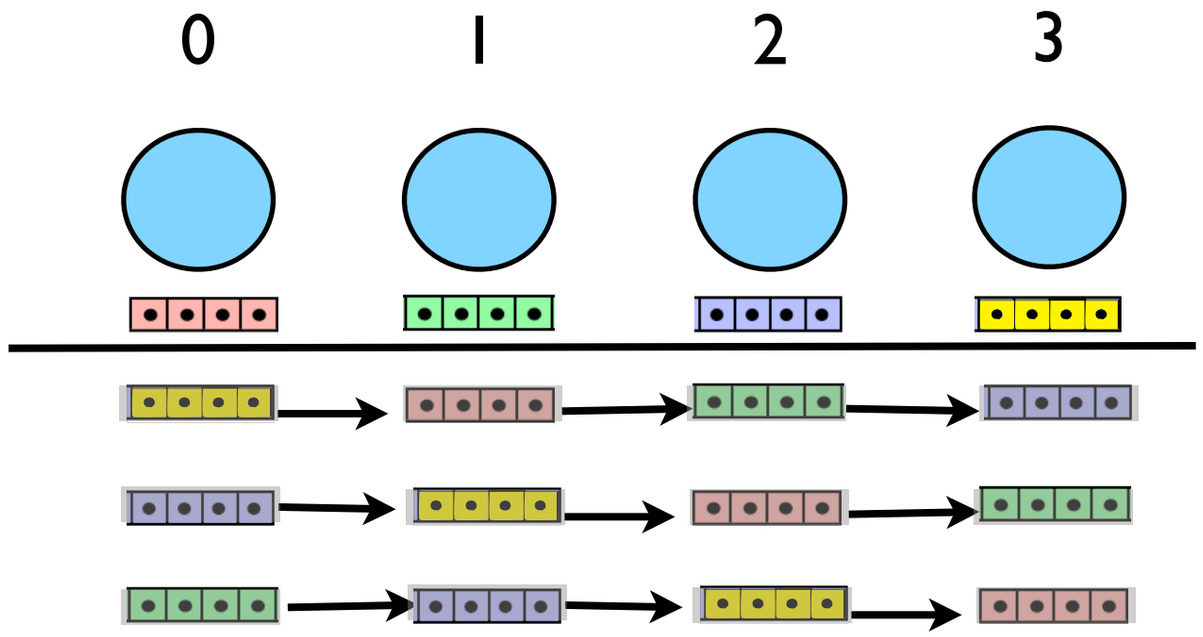
- Sending the messages: like one direction of the guardcell fills in the diffusion eqn; everyone sendrecv's.
- Periodic or else 0 would never see anyone elses particles!
- Copy your data into a buffer; send it, receive into another one.
- Compute on received data
- Swap send/recv and continue.





Pipeline

- Good: can do bigger problems!
- Bad: High communication costs, not fixable
- Bad x 2: still doing double work.

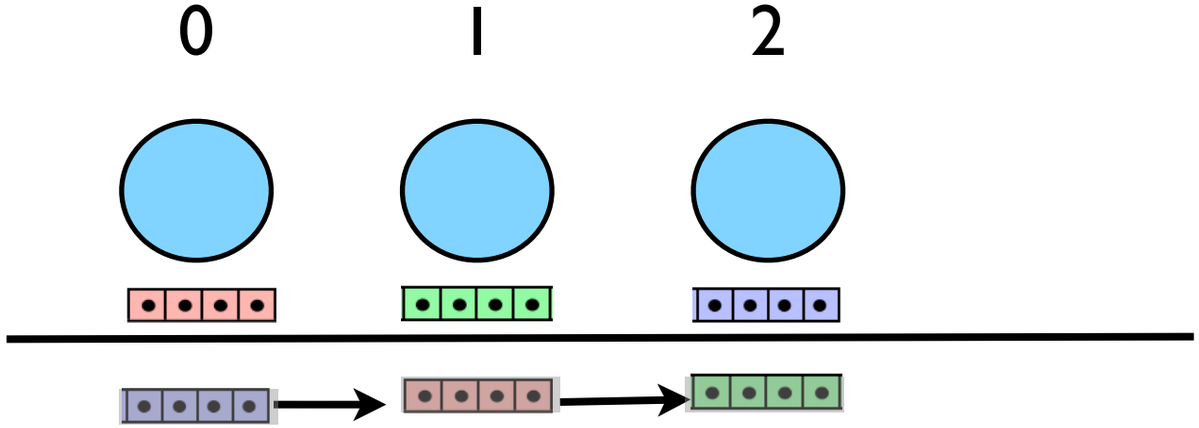


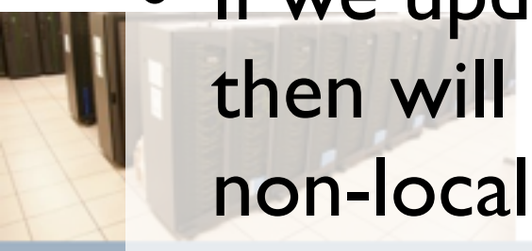
Pipeline

- Double work might be fixable
- We are sending whole particle structure when nodes only need x[NDIMS], mass.
- Option 1: we could only send chunk half way (for odd # procs); then every particle has seen every other
- If we update forces in both, then will have computed all non-local forces...)

```
typedef struct nbody_struct_s {
  NType x[NDIM];    /*the particle positions*/
  NType v[NDIM];    /*the particle velocities*/
  NType f[NDIM];    /*the forces on the particles*/
  NType mass;
  NType PE;        /* potential energy */
} NBody;

MPI_Datatype MPI_Particle;  /* derived data type for above */
```

```
  /* Create Particle Type */

  MPI_Type_contiguous( 3*NDIM+2, MPI_NType, &MPI_Particle );
  MPI_Type_commit ( &MPI_Particle );
```



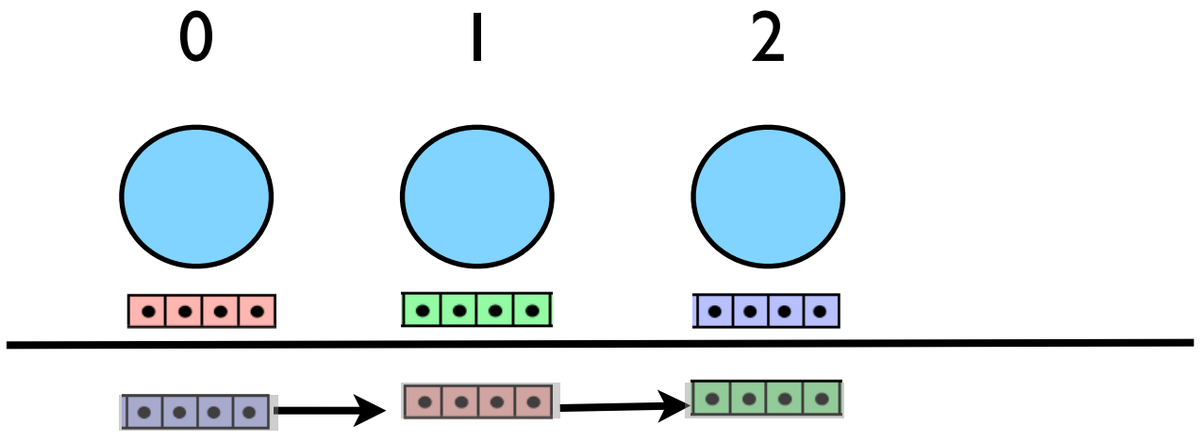


Pipeline

- Option 2: we could proceed as before, but only send the essential information
- Cut down size of message by a factor of 4/11
- Which is better?

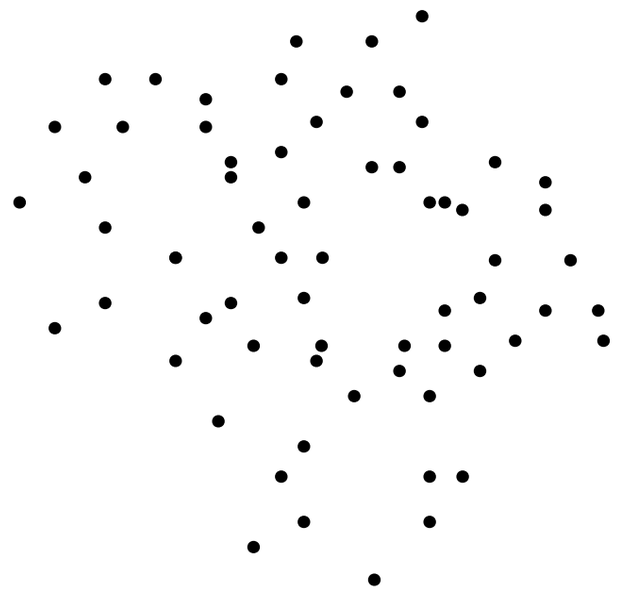
```
typedef struct nbody_struct_s {
  NType x[NDIM];    /*the particle positions*/
  NType v[NDIM];    /*the particle velocities*/
  NType f[NDIM];    /*the forces on the particles*/
  NType mass;
  NType PE;        /* potential energy */
} NBody;

MPI_Datatype MPI_Particle;  /* derived data type for above */
```

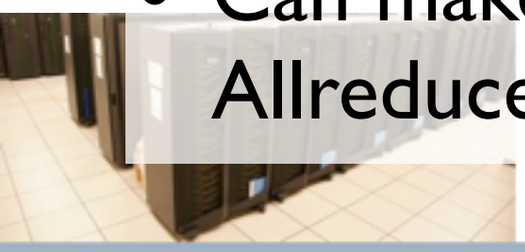
```
  /* Create Particle Type */

  MPI_Type_contiguous( 3*NDIM+2, MPI_NType, &MPI_Particle );
  MPI_Type_commit ( &MPI_Particle );
```



Displaying Data

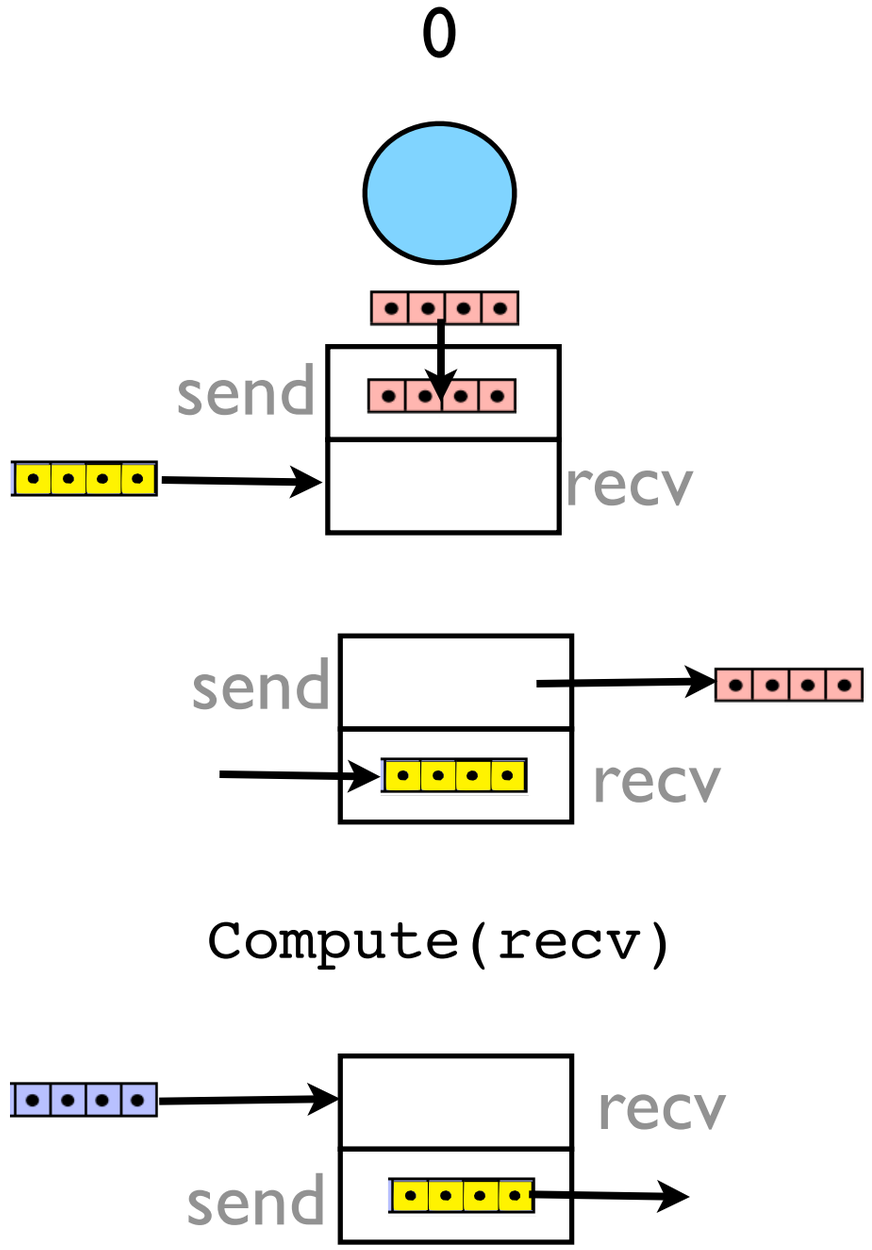
- Now that no processor owns all of the data, can't make plots any more
- But the plot is small; it's a projection onto a 2d grid of the 3d data set.
- In general it's only data-sized arrays which are 'big'
- Can make it as before and Allreduce it (like map!)





Overlapping Communication & Computation

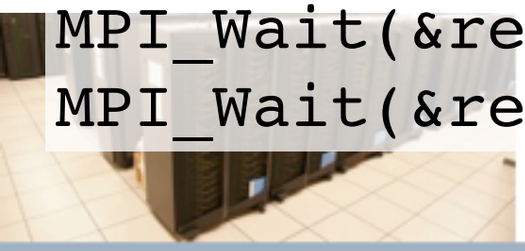
- If only updating local forces, aren't changing the data in the pipeline at all.
- What we receive is what we send.
- Could issue send right away, but need to compute...



Non-blocking Sends!

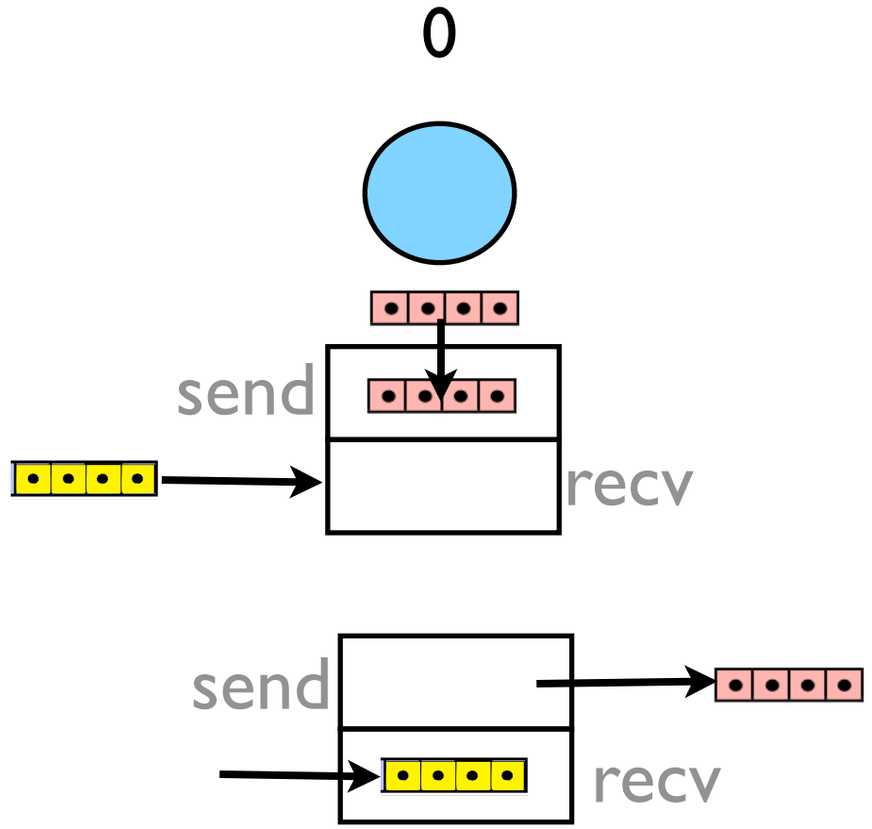
```
MPI_Request request, request2;
MPI_Status status;
int tag;

...
```

```
MPI_Irecv(buffer, 10, MPI_INT, leftneigh, tag, MPI_COMM_WORLD,
          &request);
MPI_Isend(buffer2, 10, MPI_INT, rightneigh, tag, MPI_COMM_WORLD,
          &request2);

/* do stuff.... */
MPI_Wait(&request, &status);
MPI_Wait(&request2, &status);
```



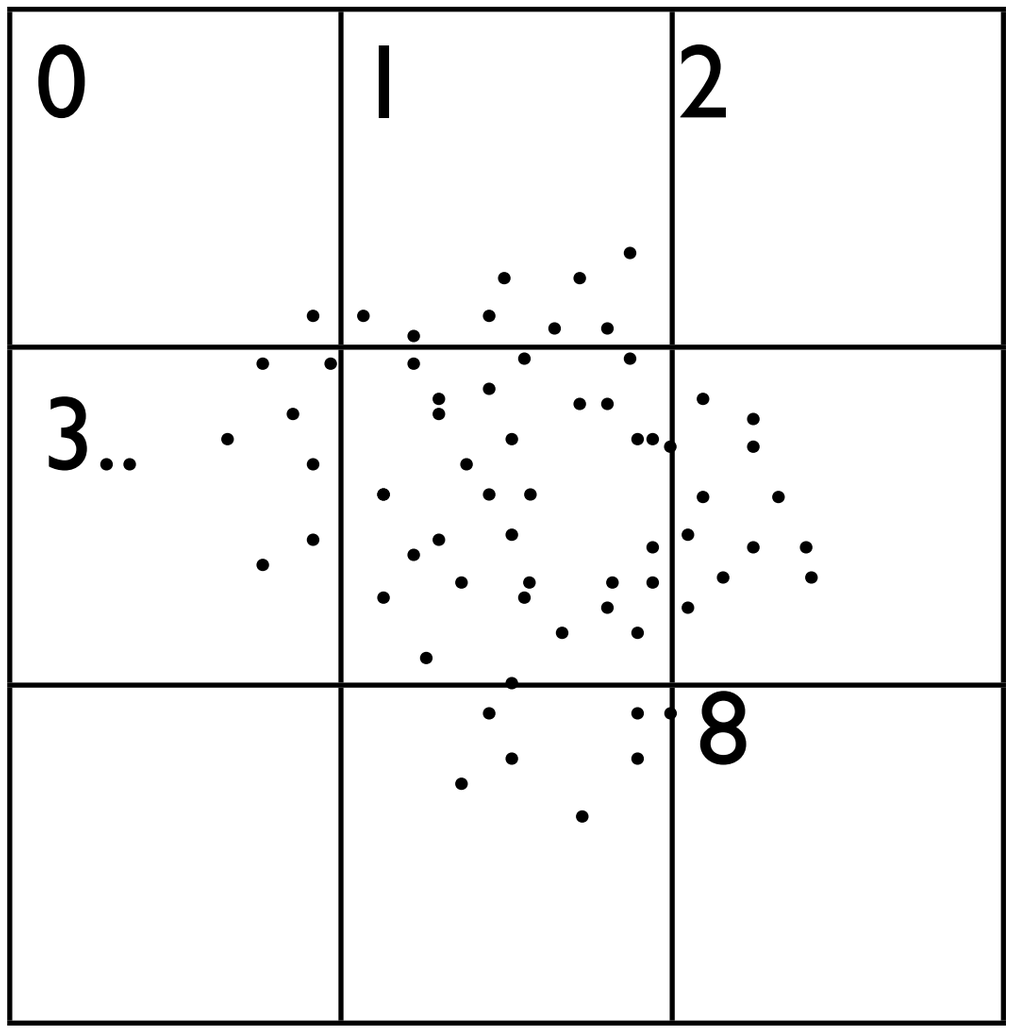
Overlapping Communication & Computation

- Now the communications will happen while we are computing
- Significant time savings! (~30% with 4 process)



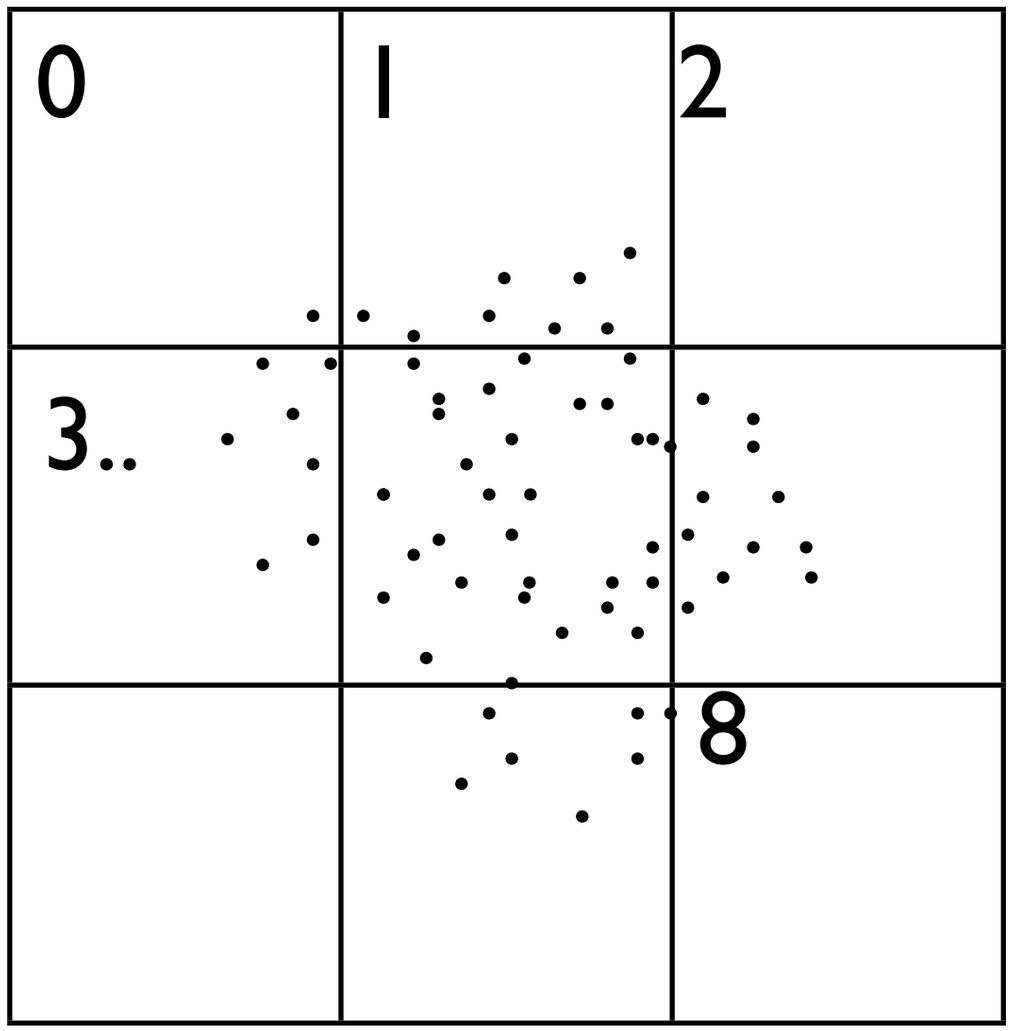
Grid-Particle codes

- For some purposes (FFT, multigrid gravity) a grid is imposed on the particle distribution, and the processors 'own' the particles



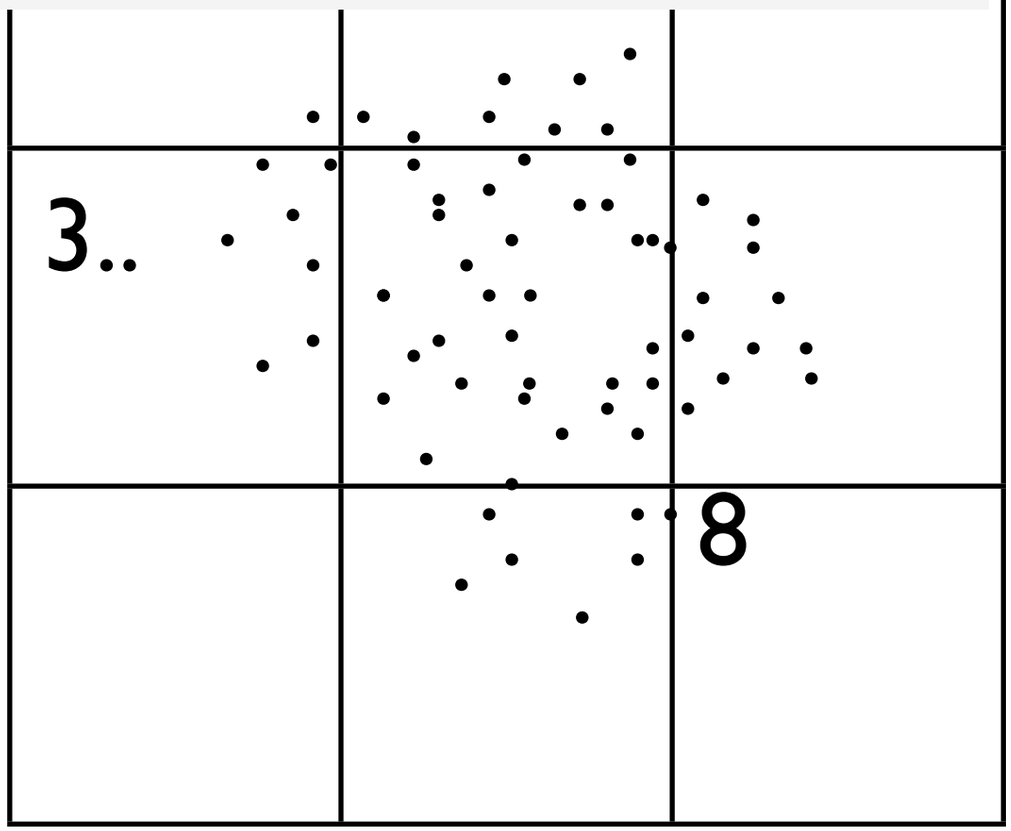
Grid-Particle codes

- Up to now, we have decided ourselves which processor gets which piece of the domain; but MPI actually has some routines for this.



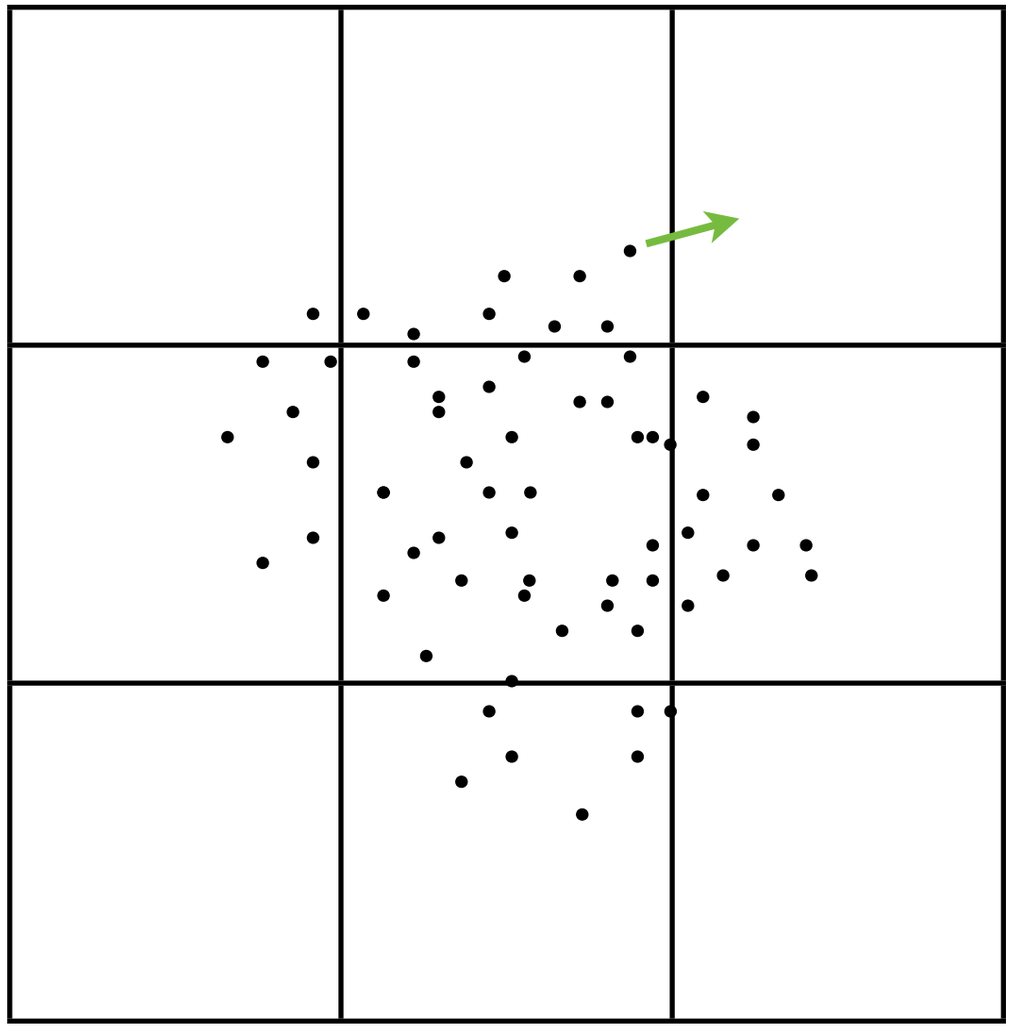
```
MPI_Cart_create(MPI_COMM_WORLD, NDIM, dims, periodic, 1, &GRID_COMM);
MPI_Cart_get(GRID_COMM, NDIM, dims, periodic, gridcoords);

for (i=0; i<NDIM; i++) {
  MPI_Cart_shift(GRID_COMM, i, +1, &neighs[i][0], &neighs[i][1]);
```

- Calculates neighbors, etc for you
- And calculates where you are in the grid of processes
- Saves some bookkeeping, and might do a better job...



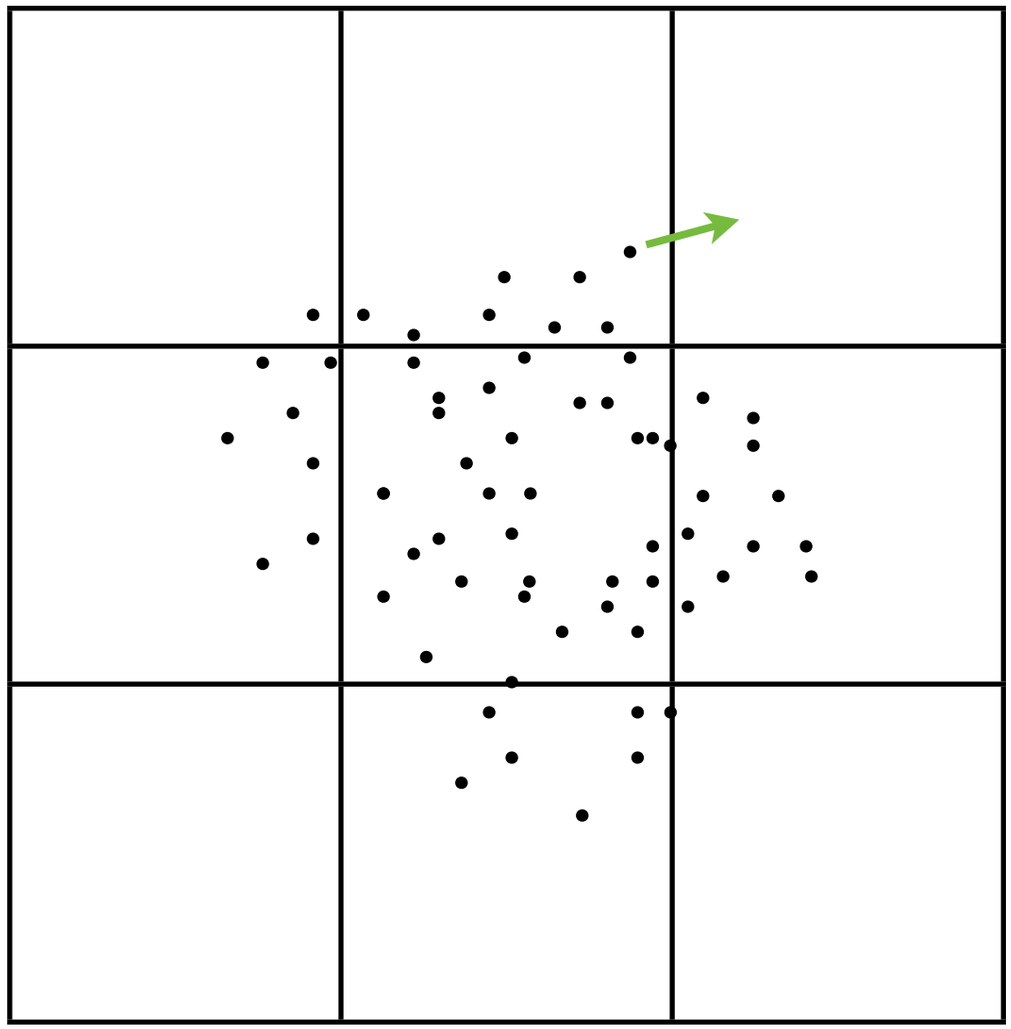
Grid-Particle codes

- But what happens when a particle moves?
- Has to be a mechanism for moving the particle to the appropriate processor.
- Tricky. Can't just tell your neighbor; how do they know to listen for you?



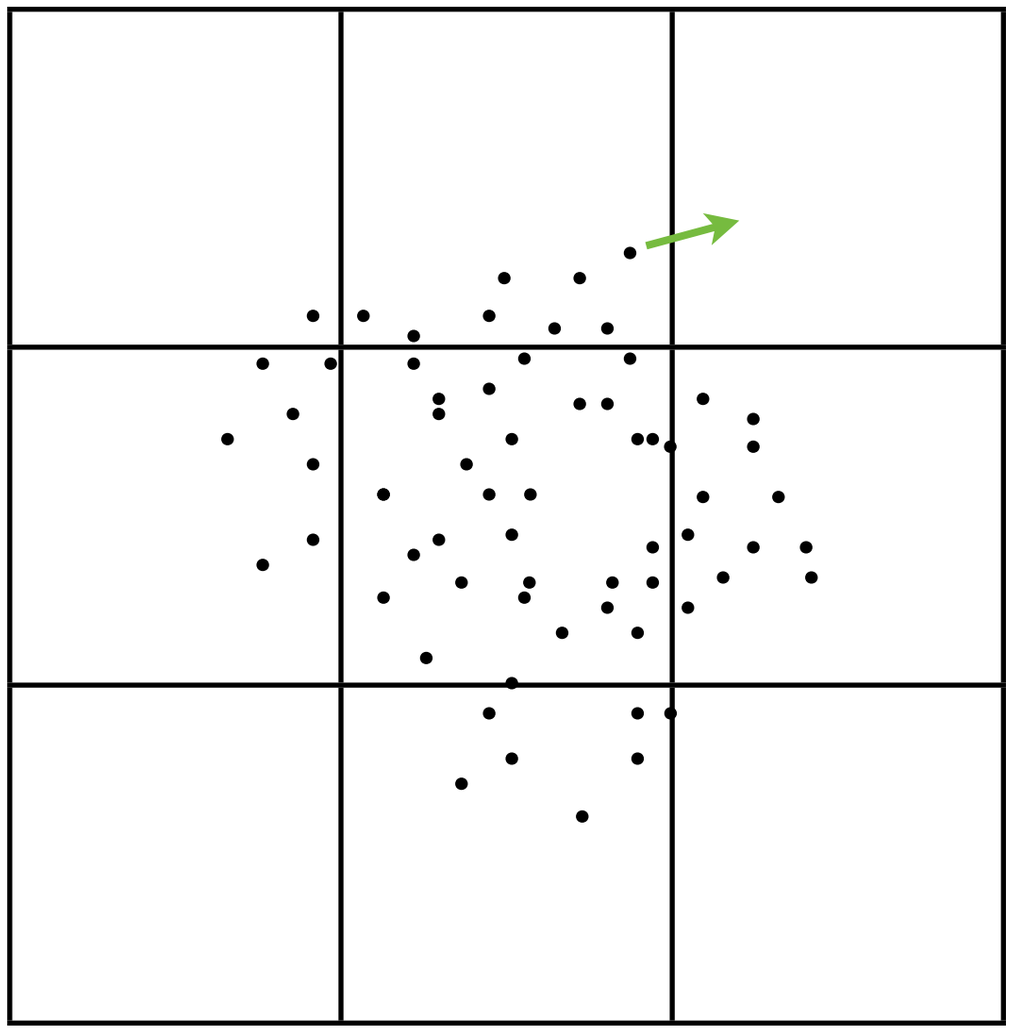
Grid-Particle codes

- Could create list -- number of processors who has particles for processor i
- Allreduce sum it
- Then i knows to wait for that many messages



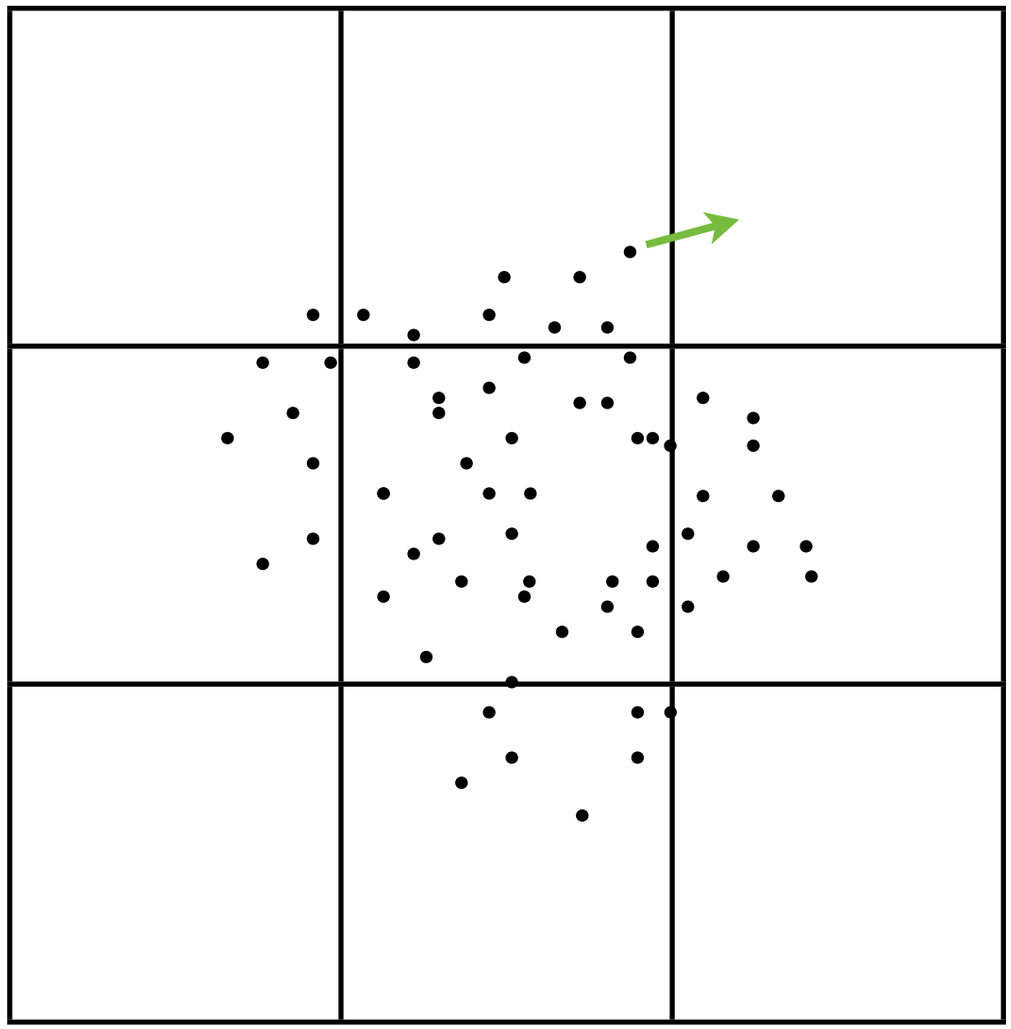
Grid-Particle codes

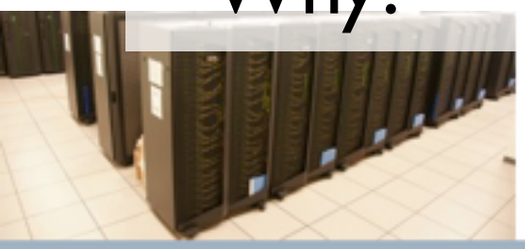
- But particles probably don't move much
- Do 'shifting'. If anyone has particles that need to be moved in X direction, shift all particles to be moved in X; pull of right ones
- Then Y, then Z.



Grid-Particle codes

- This is implemented in nbody-gridparticles
- Executable in completedexcutables
- Try running it...
- Fairly quickly gets very slow. Why?





Homework (hw7)

- Code skeleton for these parallelizations exist in sourcecode/nbody.
- Parallelize allgather, and blocking pipeline
- Run some timing tests
- Figure out which of two optimizations to do for blocking pipeline

Allgather

- Need to figure out your start/end particle,
- Sum total energies,
- And make the allgatherv call.
- (Look for HW in the source code).

Pipeline

- Get the plot all data working
- Implement pipeline
- Do off-process force calculation
- (Again, look for HW in the source code nbody-pipeline.c)

Timings

- On cluster, qsub some batch scripts and make timing comparisions. Could be anything - scaling test (how does it perform w/ different # of procs?), algorithm comparision (pipeline vs. non-blocking pipeline?) Can use executables in `completedexecutables`

Timings

- Finally, of the two discussed optimizations for the (blocking pipeline) how much does each effect communication cost? Computation cost?
- Which is more likely to be useful? Why?
- blocking-optimizations.txt

C syntax

```
MPI_Status status;



ierr = MPI_Allgather ( sendptr, sendcount, MPI_TYPE,
                       recvptr, recvcount, MPI_TYPE, Communicator);
ierr = MPI_Allgatherv( sendptr, sendcount, MPI_TYPE,
                       recvptr, recvcounts, displacements,
                       MPI_TYPE, Commuicator);
int MPI_Cart_create ( MPI_Comm comm_old, int ndims, int *dims,
                       int *periods, int reorder,
                       MPI_Comm *comm_cart );
int MPI_Cart_shift ( MPI_Comm comm, int direction, int displ,
                       int *source, int *dest );


Communicator -> MPI_COMM_WORLD
MPI_Type -> MPI_FLOAT, MPI_DOUBLE, MPI_INT, MPI_CHAR...
MPI_OP -> MPI_SUM, MPI_MIN, MPI_MAX,...
```