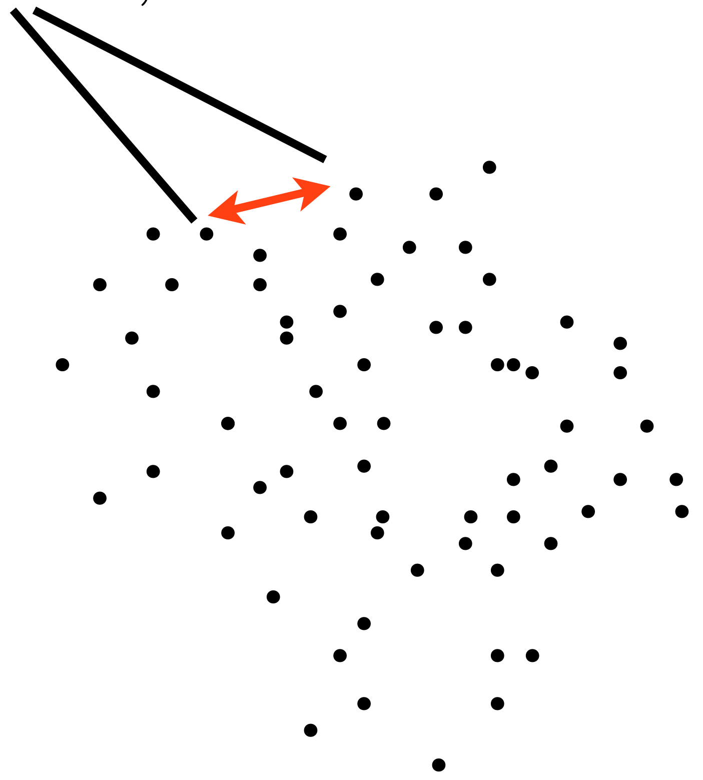
N-Body Dynamics

$$F_{1,2} = -\frac{G m_1 m_2}{r_{1,2}^2}\hat{\mathbf{r}}$$

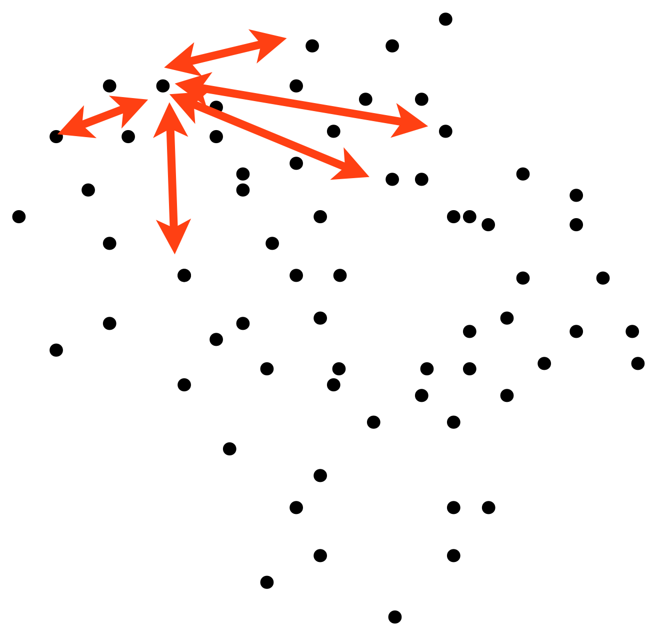


N-Body dynamics

- N interacting bodies
- Pairwise forces; here, Gravity.
- (here, stars in a cluster; could be molecular dynamics, economic agents...)

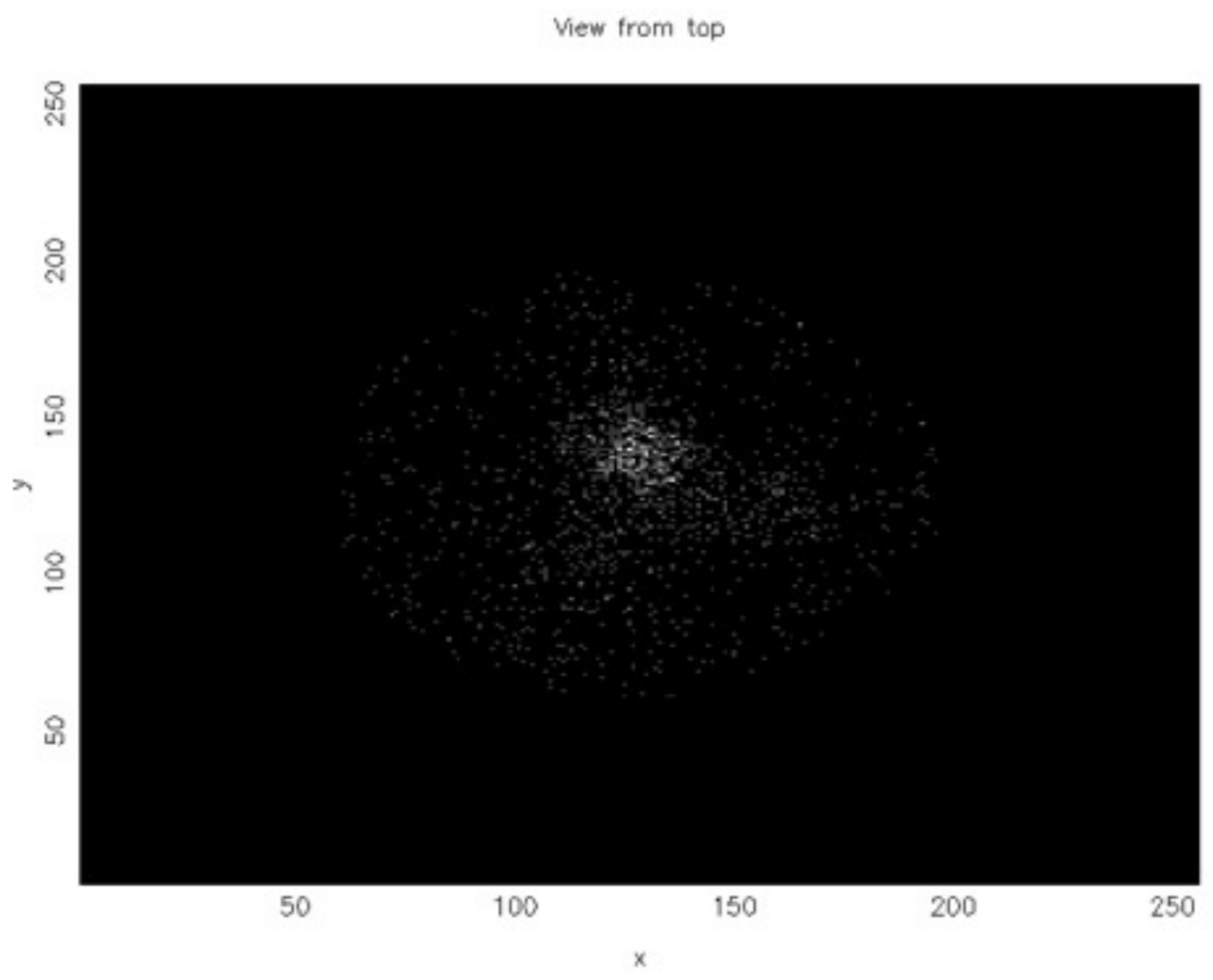
N-Body dynamics

- N interacting bodies
- Pairwise forces; here, Gravity.
- (here, stars in a cluster; could be molecular dynamics, economic agents...)



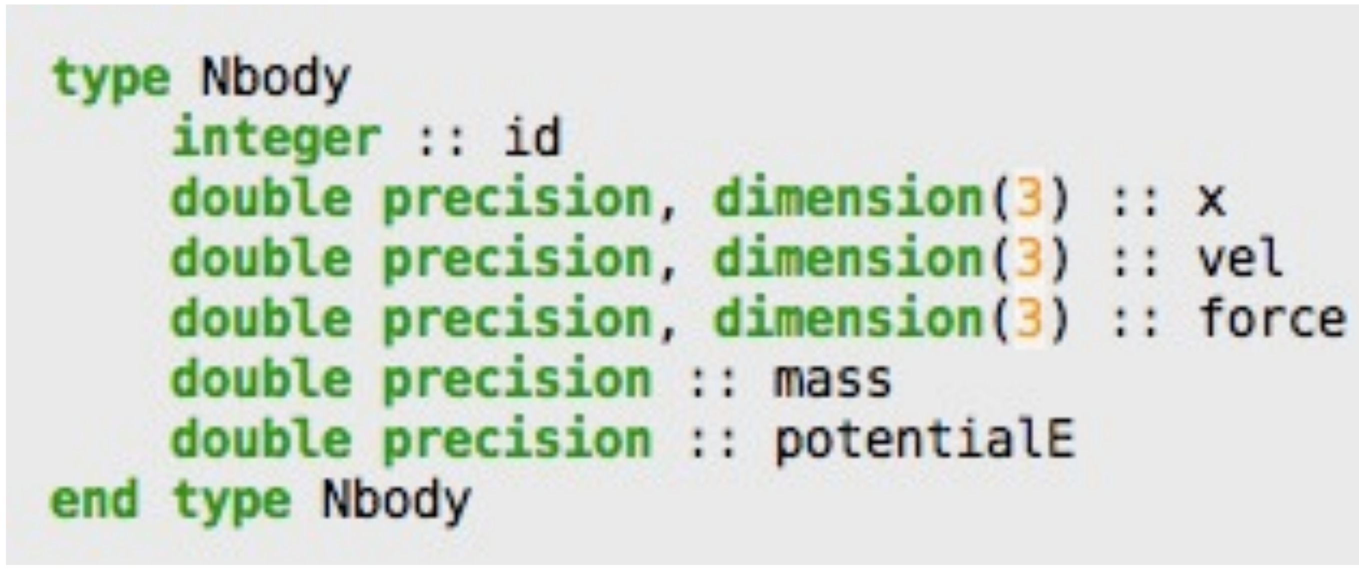
nbody

- `cd ~/ppp/nbodyc`
- `make`
- `./nbodyc`



A Particle type

- Everything based on a array of structures ('derived data types')

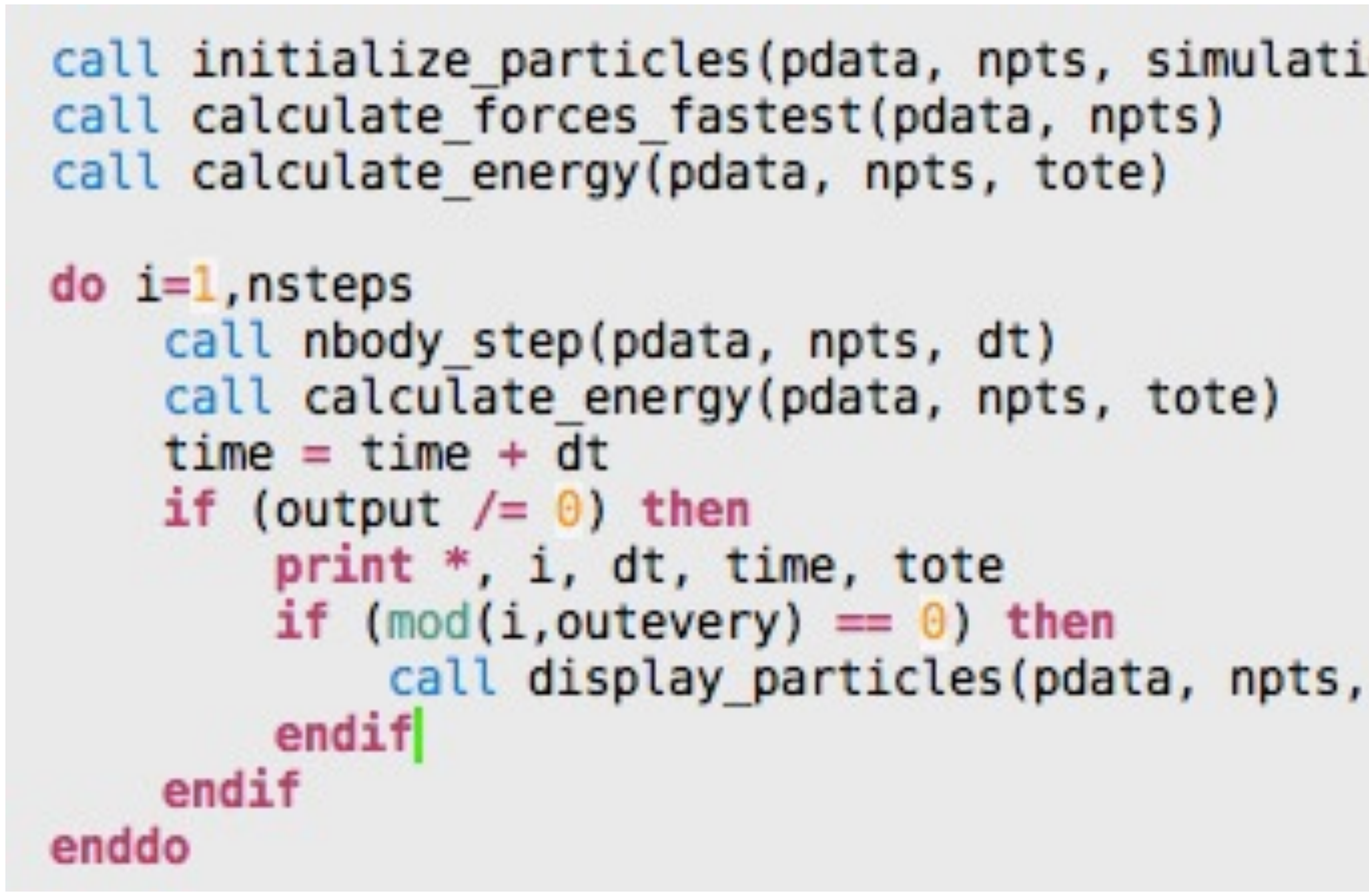


```
type Nbody
    integer :: id
    double precision, dimension(3) :: x
    double precision, dimension(3) :: vel
    double precision, dimension(3) :: force
    double precision :: mass
    double precision :: potentialE
end type Nbody
```

`nbody.f90`, line 5

Main loop

- nbody_step - calls calculate forces, updates positions.
- calculate energy (diagnostic)
- display particles.

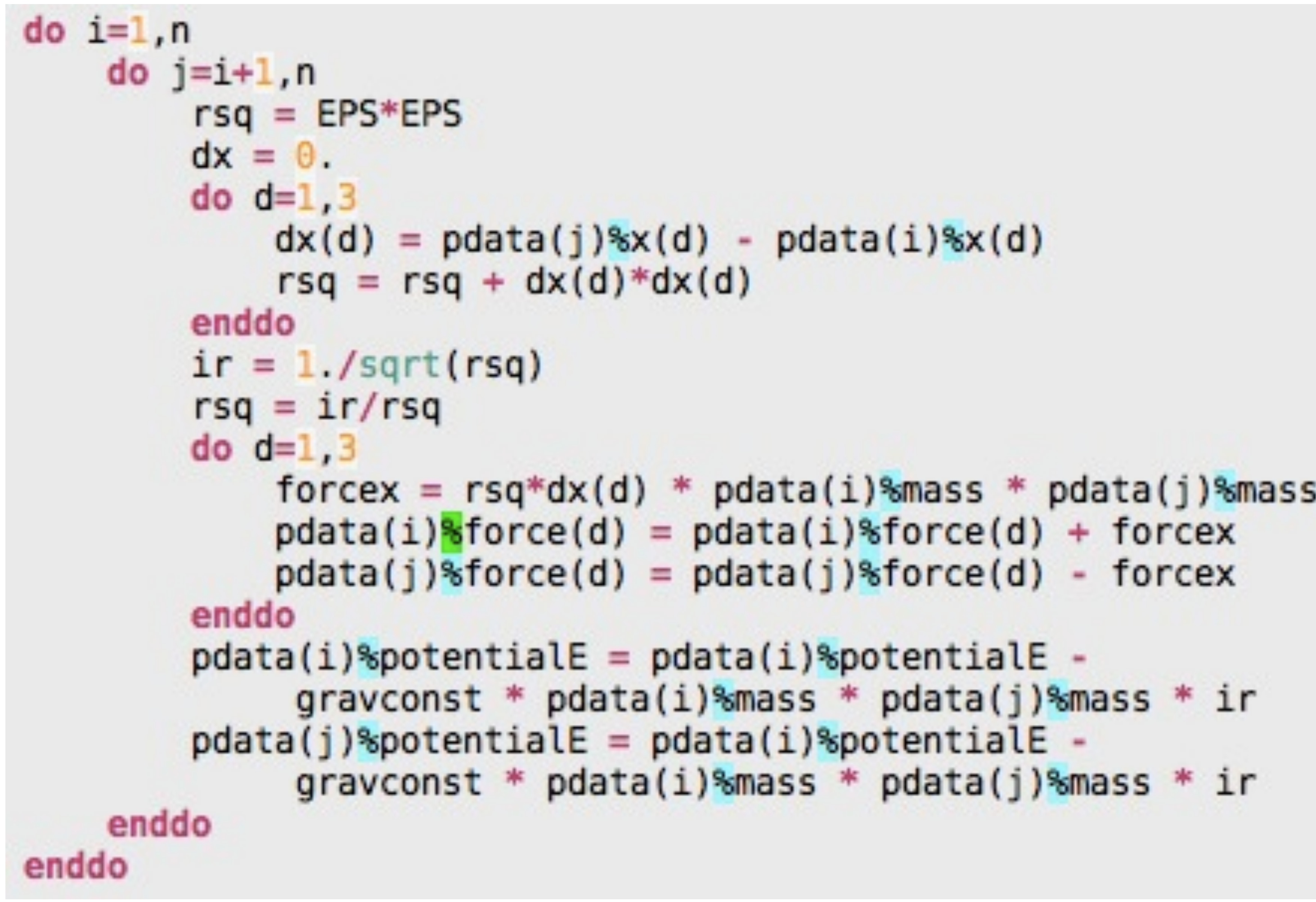


```
call initialize_particles(pdata, npts, simulati
call calculate_forces_fastest(pdata, npts)
call calculate_energy(pdata, npts, tote)

do i=1,nsteps
    call nbody_step(pdata, npts, dt)
    call calculate_energy(pdata, npts, tote)
    time = time + dt
    if (output /= 0) then
        print *, i, dt, time, tote
        if (mod(i,outevery) == 0) then
            call display_particles(pdata, npts,
        endif
    endif
enddo
```

`nbody.f90`, line 35

Calculate Forces

- For each particle i
- Foreach other particle j>i
- Calculate distance (most expensive!)
- Increment force
- Increment potential energy

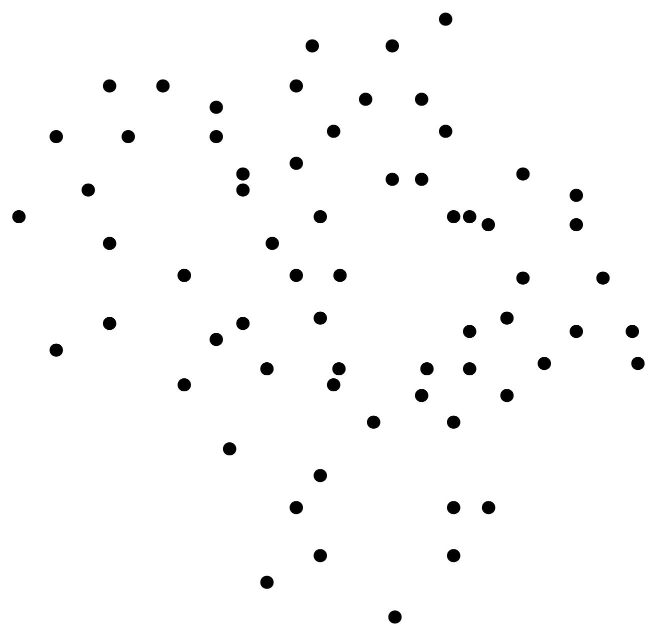


```
do i=1,n
    do j=i+1,n
        rsq = EPS*EPS
        dx = 0.
        do d=1,3
            dx(d) = pdata(j)%x(d) - pdata(i)%x(d)
            rsq = rsq + dx(d)*dx(d)
        enddo
        ir = 1./sqrt(rsq)
        rsq = ir/rsq
        do d=1,3
            forcex = rsq*dx(d) * pdata(i)%mass * pdata(j)%mass
            pdata(i)%force(d) = pdata(i)%force(d) + forcex
            pdata(j)%force(d) = pdata(j)%force(d) - forcex
        enddo
        pdata(i)%potentialE = pdata(i)%potentialE -
             gravconst * pdata(i)%mass * pdata(j)%mass * ir
        pdata(j)%potentialE = pdata(i)%potentialE -
             gravconst * pdata(i)%mass * pdata(j)%mass * ir
    enddo
enddo
```

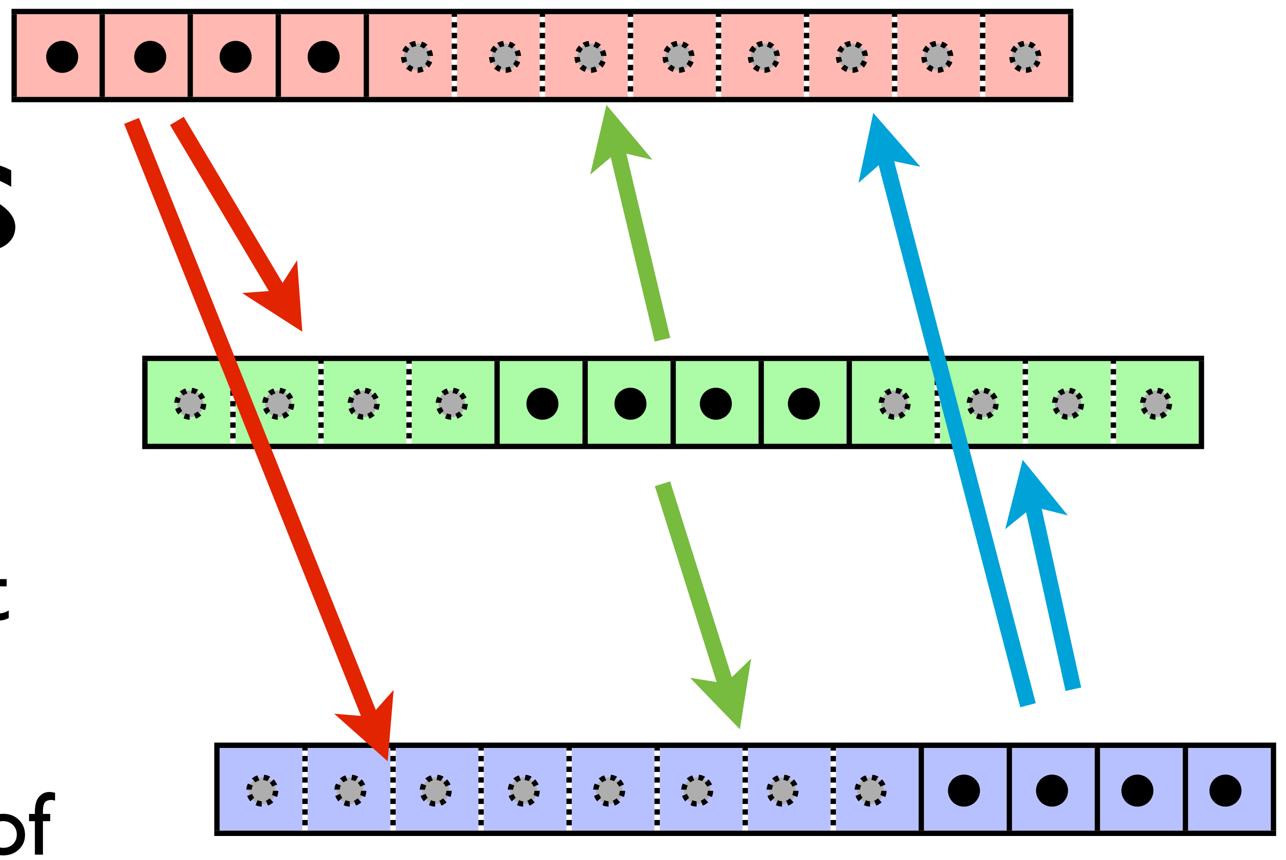
`nbody.f90`, line 100

Decomposing onto different processors

- Direct summation (N^2) - each particle needs to know about all other particles
- Limited locality possible
- Inherently a difficult problem to parallelize in distributed memory

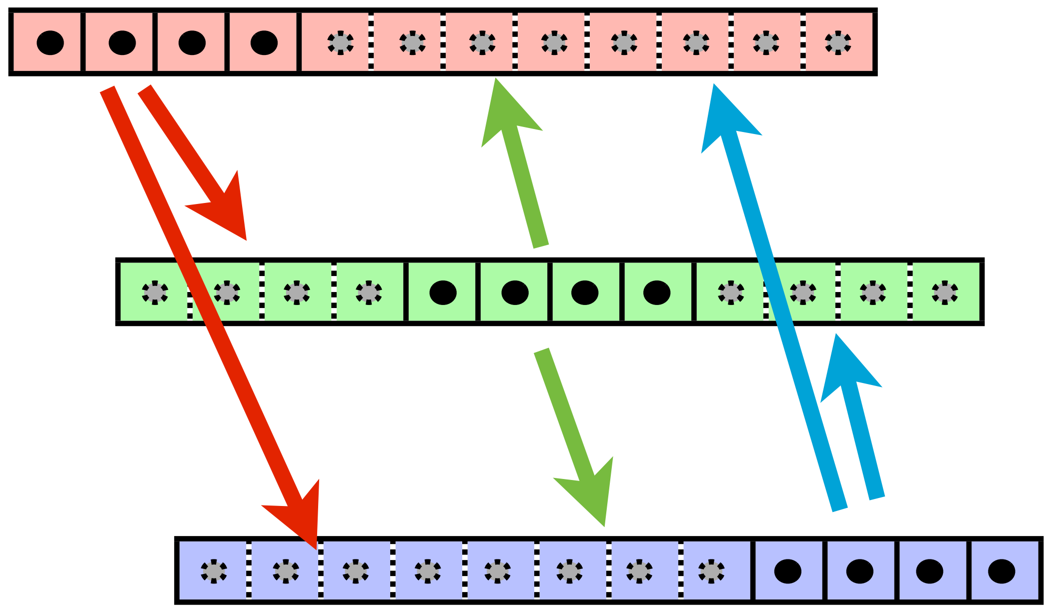


First go: Everyone sees everything

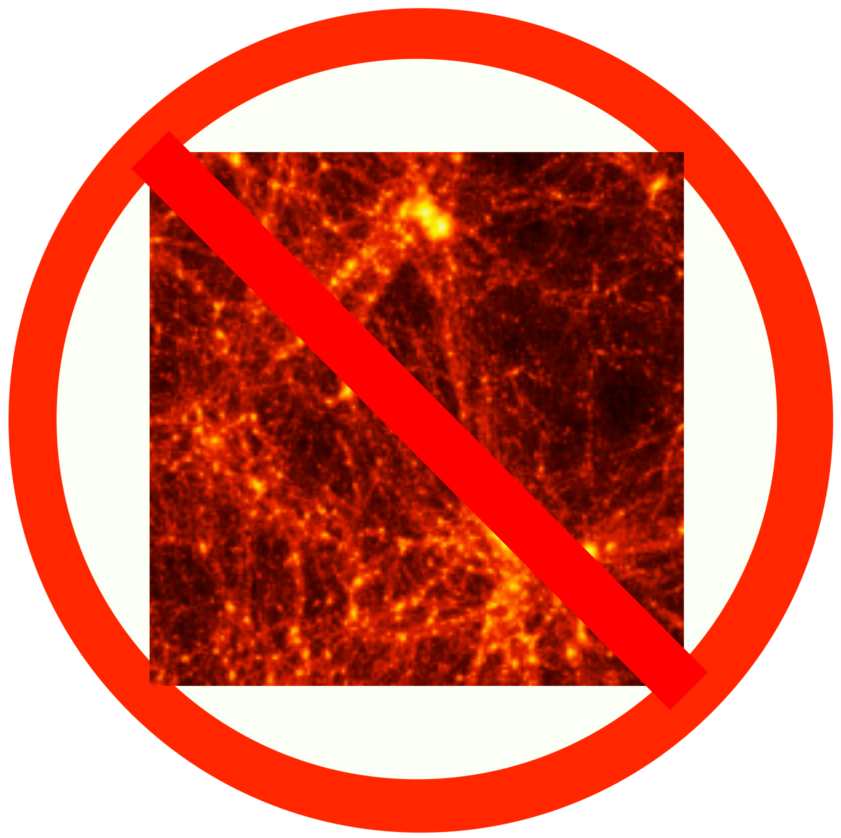


- Distribute the work, but not the data
- Everyone has complete set of particle data
- Just work on our own particles
- Send everyone our particles' data afterwards

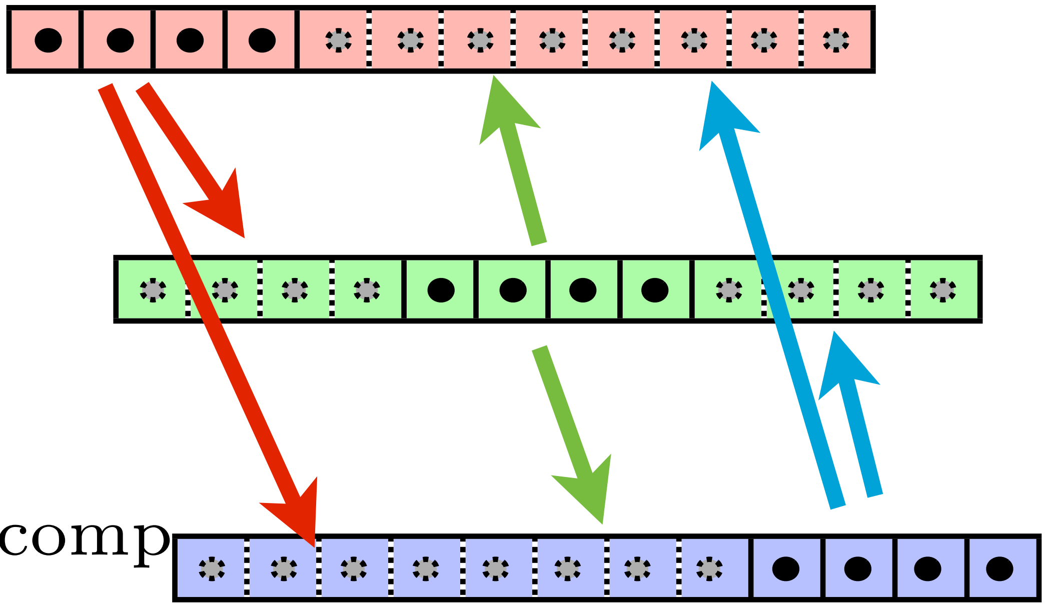
Terrible Idea (I)



- Requires the entire problem to fit in the memory of each node.
- In general, you can't do that (10^{10-11} particle simulation)
- 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)



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

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

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

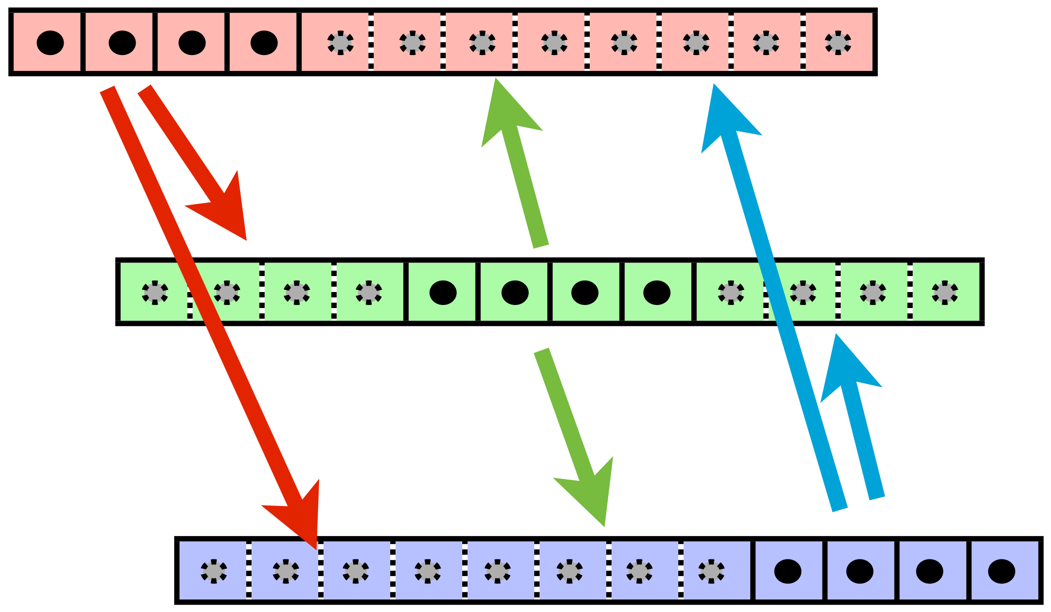
$$\approx c_{\mathrm{particle}} N C_{\mathrm{comm}}$$

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

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

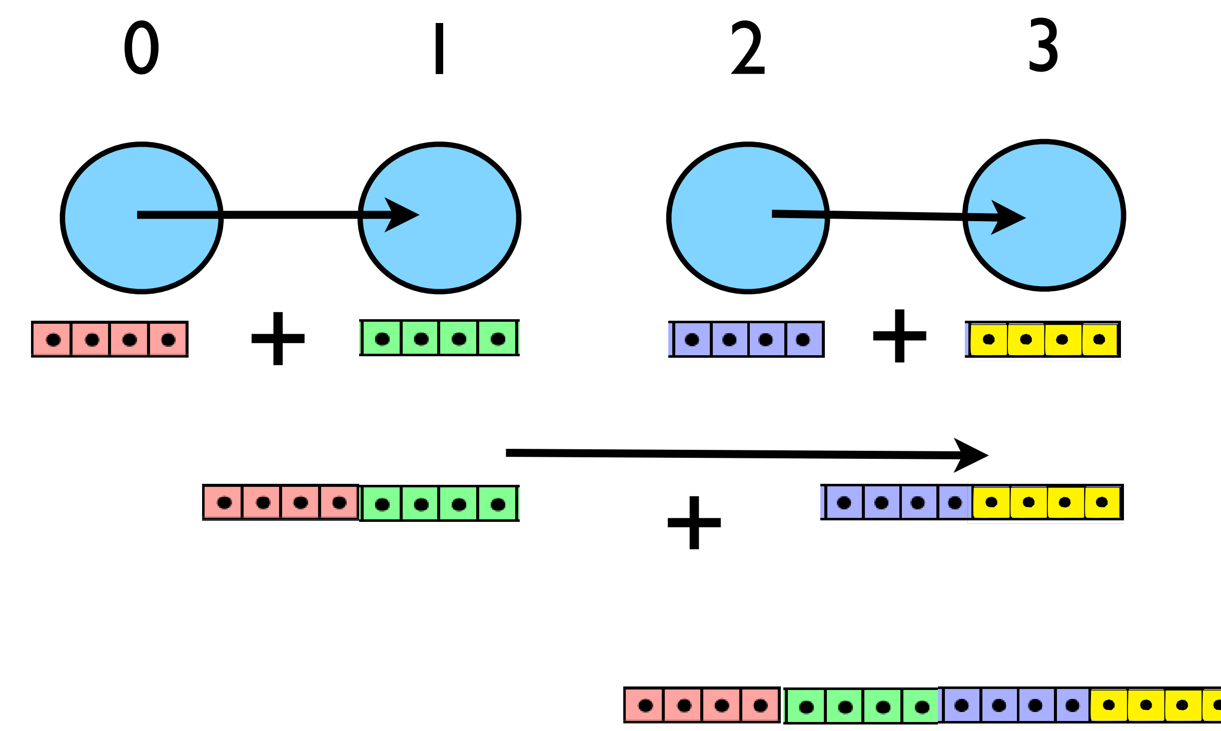
Terrible Idea (III)

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



Can address (II) a little

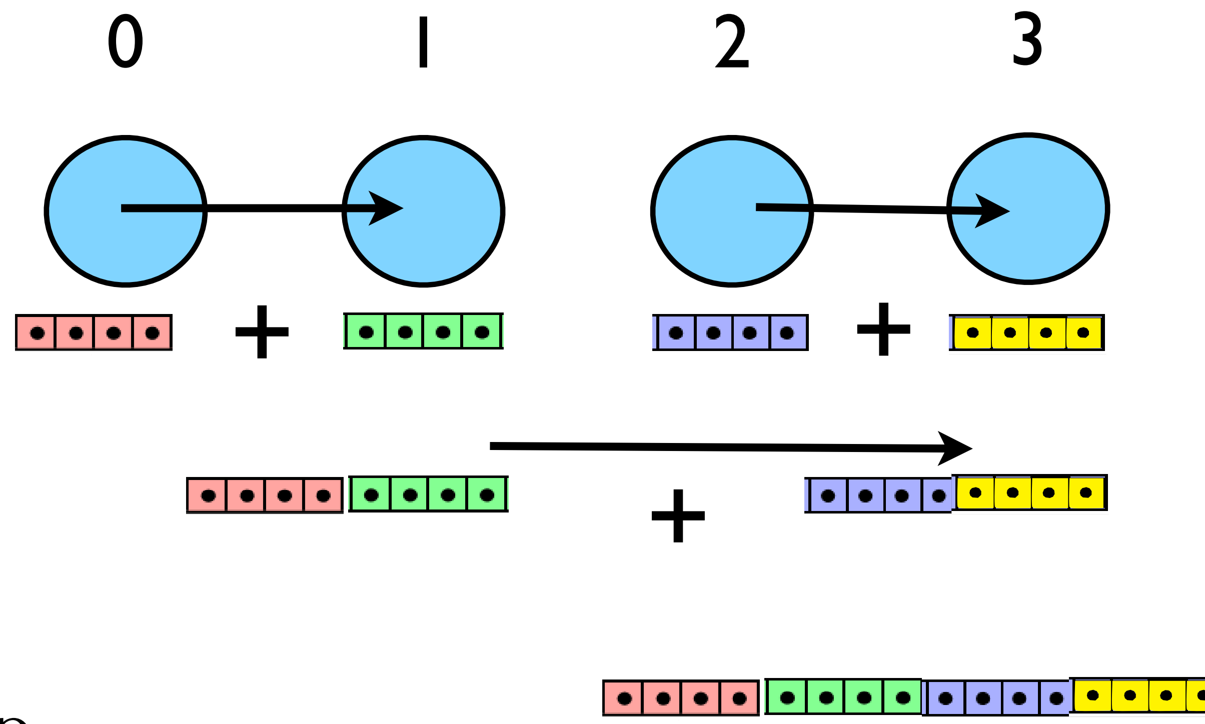
- Collecting everyone's data is like a global sum
- (Concatenation is the sort of operation that allows reduction)
- GATHER operation
- Send back the results: ALLGATHER
- 2 (P-1) vs P^2 messages, but length differs



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 (I) a little

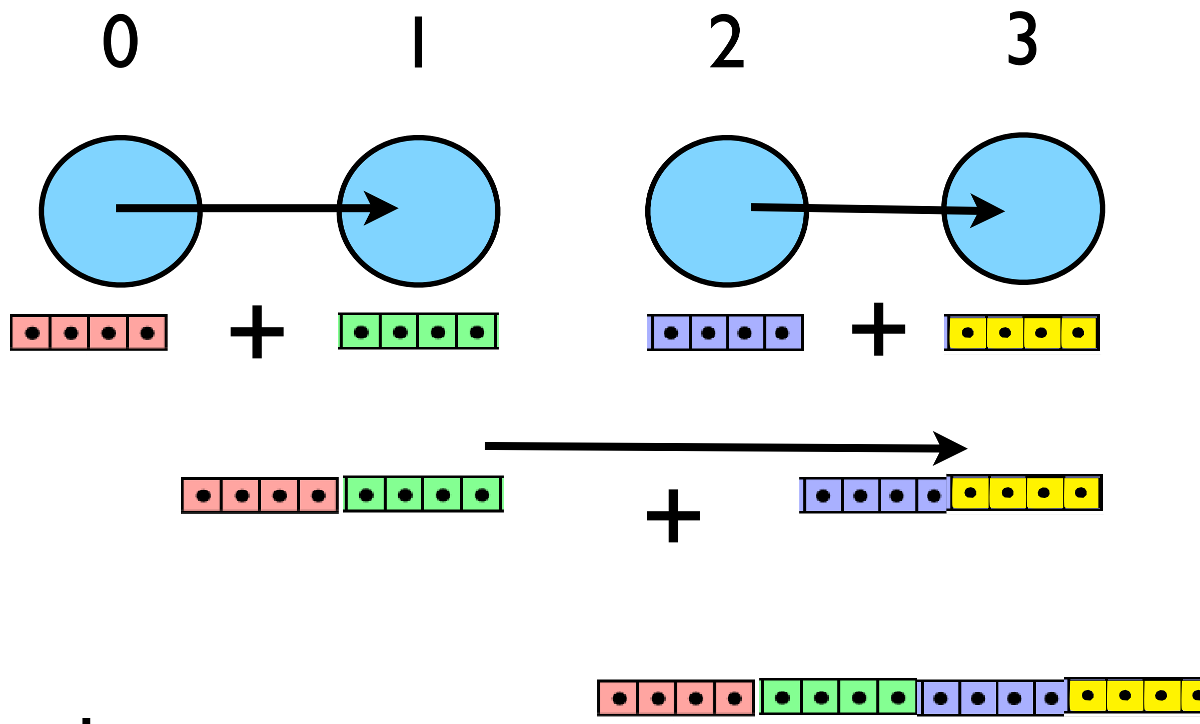


$$T_{\text{comp}} = c_{\text{grav}} \frac{N^2}{P} C_{\text{comp}}$$

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

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

Another collective operation



Stuff you're sending

How Much

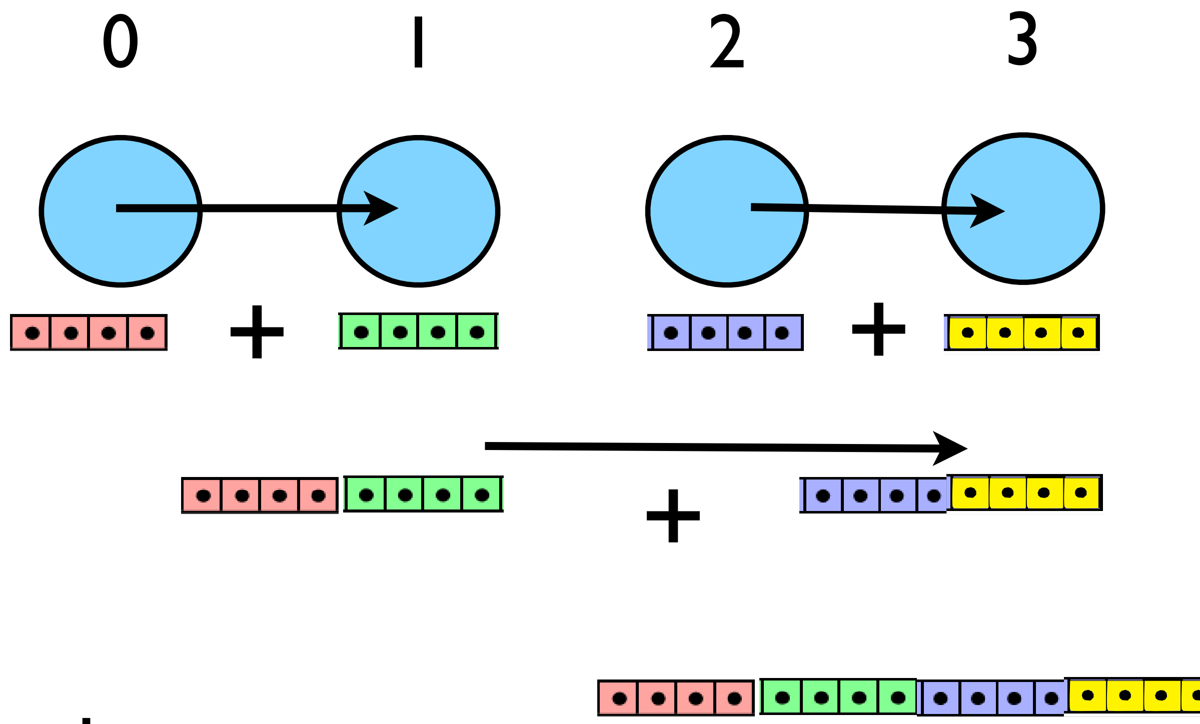
What Type

```
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



Stuff you're sending

How Much

What Type

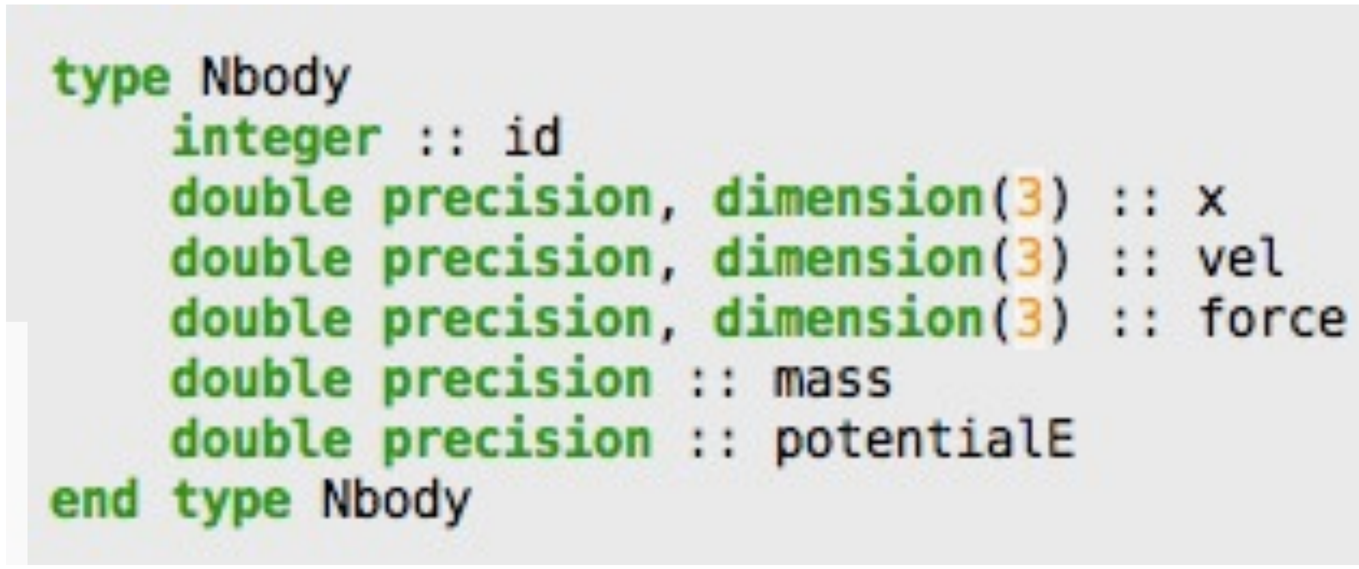
```
MPI_GATHER (sendbuf, INTEGER sendcnt, INTEGER sendtype,
               recvbuf, INTEGER recvcount, INTEGER recvtype,
               INTEGER root, INTEGER comm, INTEGER ierr);
```

Place you're receiving

Who's getting all the data

But what data type should we use?

- Not just a multiple of a single data type
- Contiguous, vector, subarray types won't do it.



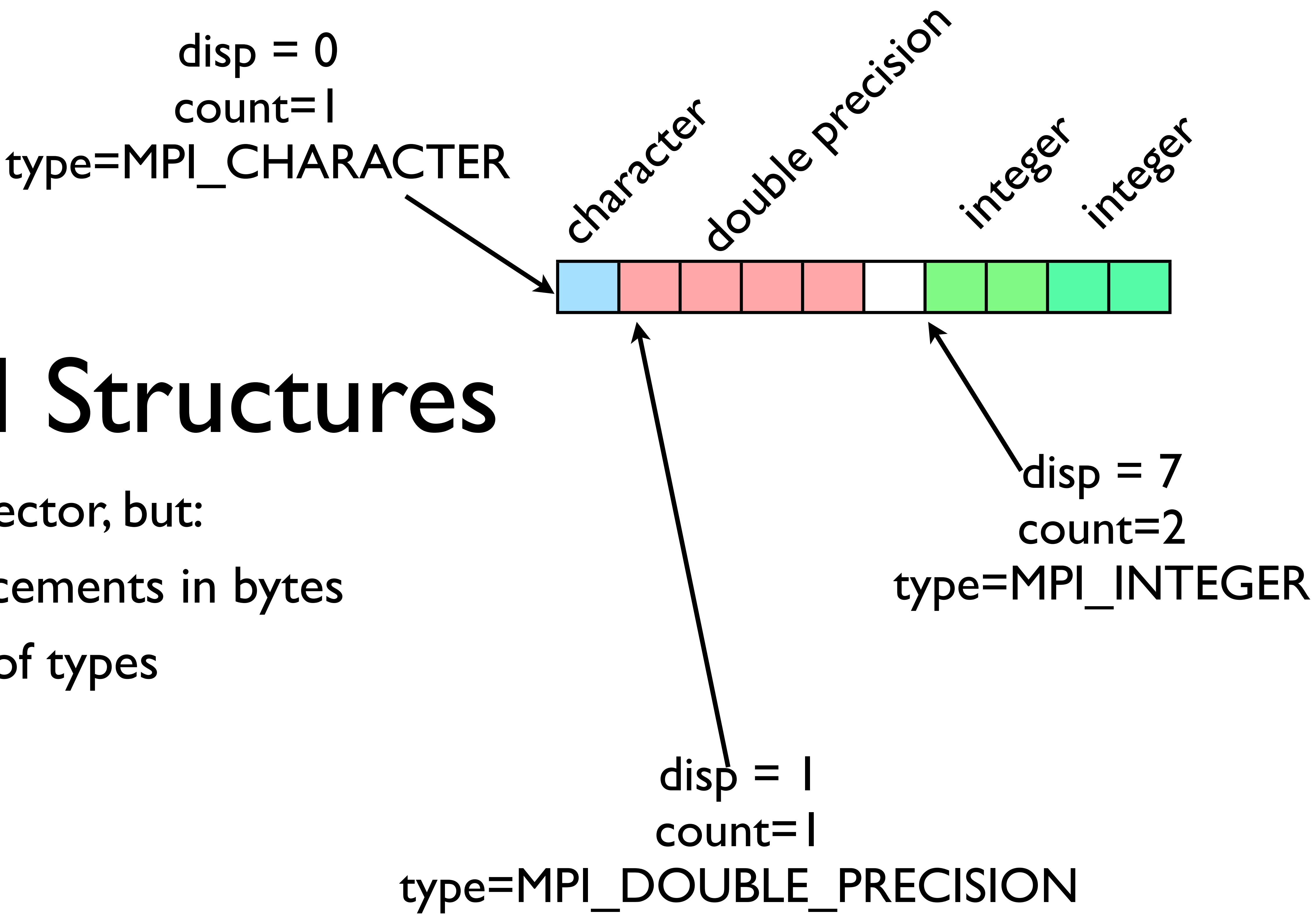
```
type Nbody
    integer :: id
    double precision, dimension(3) :: x
    double precision, dimension(3) :: vel
    double precision, dimension(3) :: force
    double precision :: mass
    double precision :: potentialE
end type Nbody
```

```
MPI_TYPE_CREATE_STRUCT(INTEGER COUNT, INTEGER ARRAY_OF_BLOCKLENGTHS(*),
            INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF DISPLACEMENTS(*),
            INTEGER ARRAY_OF_TYPES(*), INTEGER NEWTYPE, INTEGER IERROR)
```

```
int MPI_Type_create_struct(int count, int array_of_blocklengths[],
    MPI_Aint array_of_displacements[], MPI_Datatype array_of_types[],
    MPI_datatype *newtype);
```

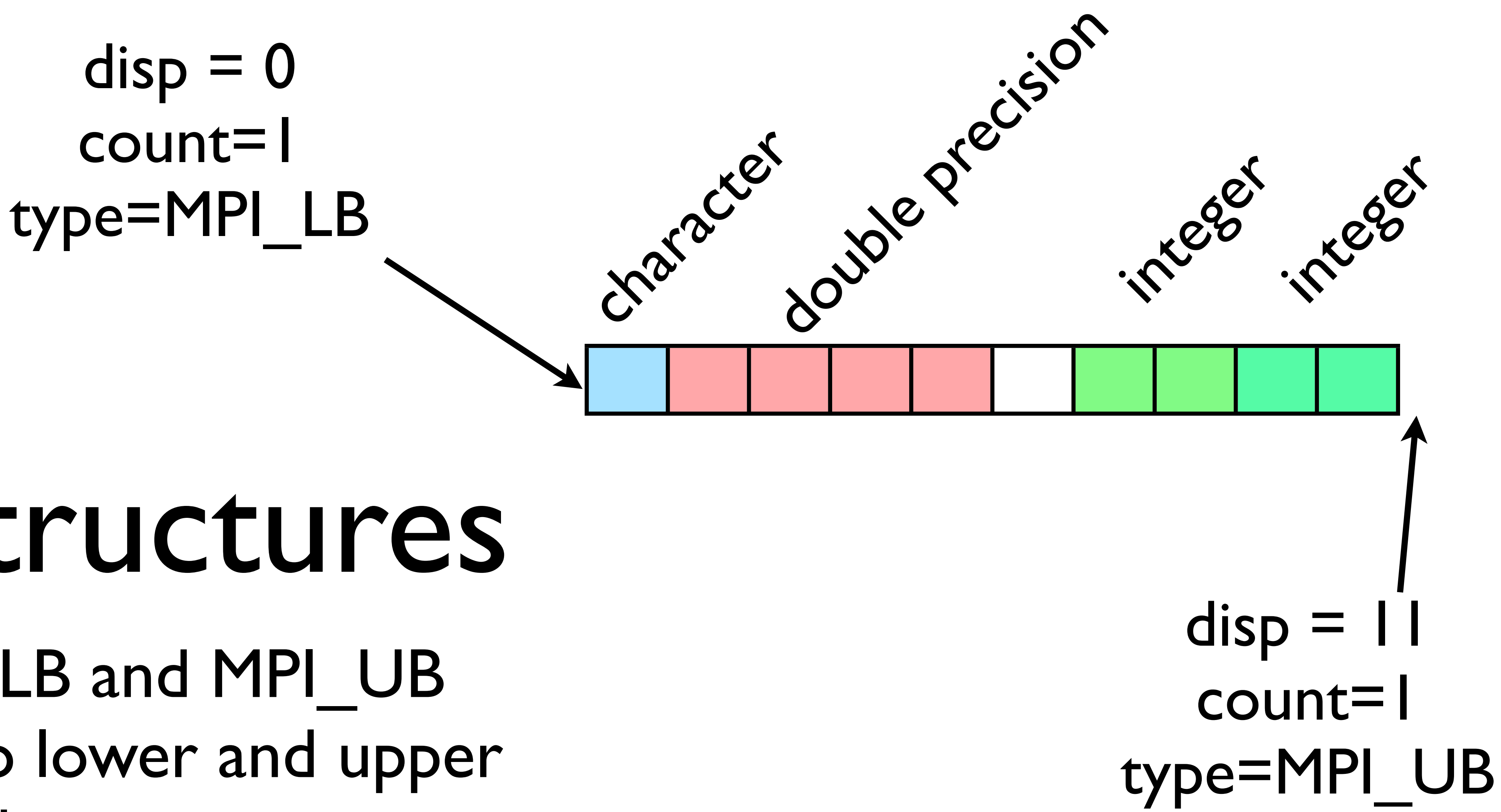
MPI Structures

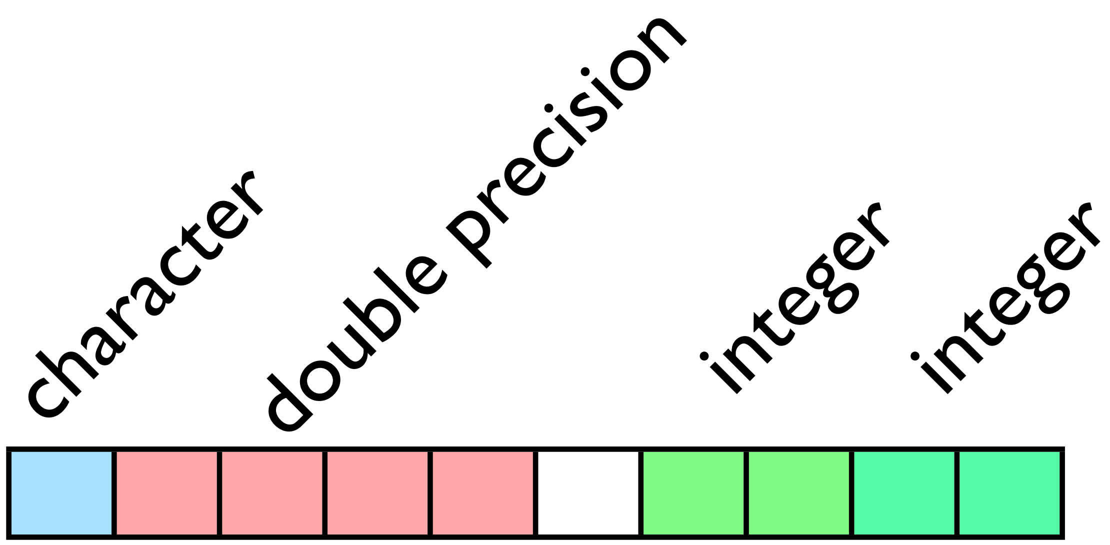
- Like vector, but:
- displacements in bytes
- array of types



MPI Structures

- Types MPI_LB and MPI_UB can point to lower and upper bounds of the structure, as well





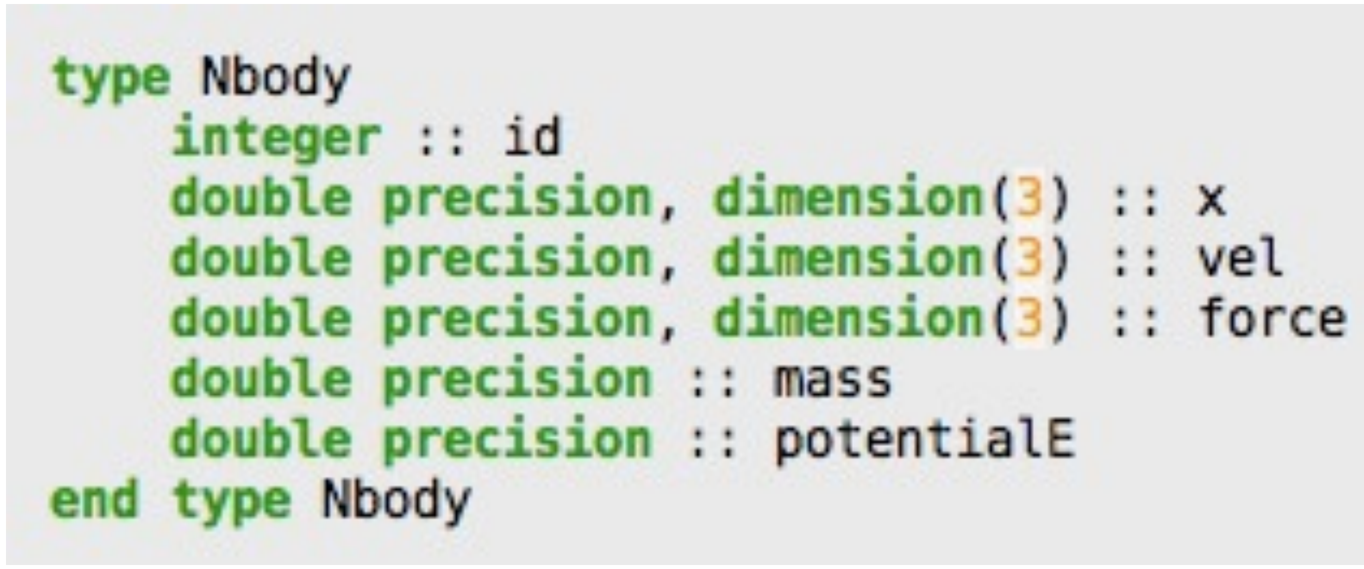
MPI Type Maps

- Complete description of this structure looks like:
  blocklens = (1,1,1,2,1)
  displacements = (0,0,1,6,10)
  types = (MPI_LB, MPI_CHARACTER, MPI_DOUBLE_PRECISION, MPI_INTEGER, MPI_UB)

- Note typemaps not unique; could write the integers out as two single integers with displacements 6, 8.

MPI Type Maps

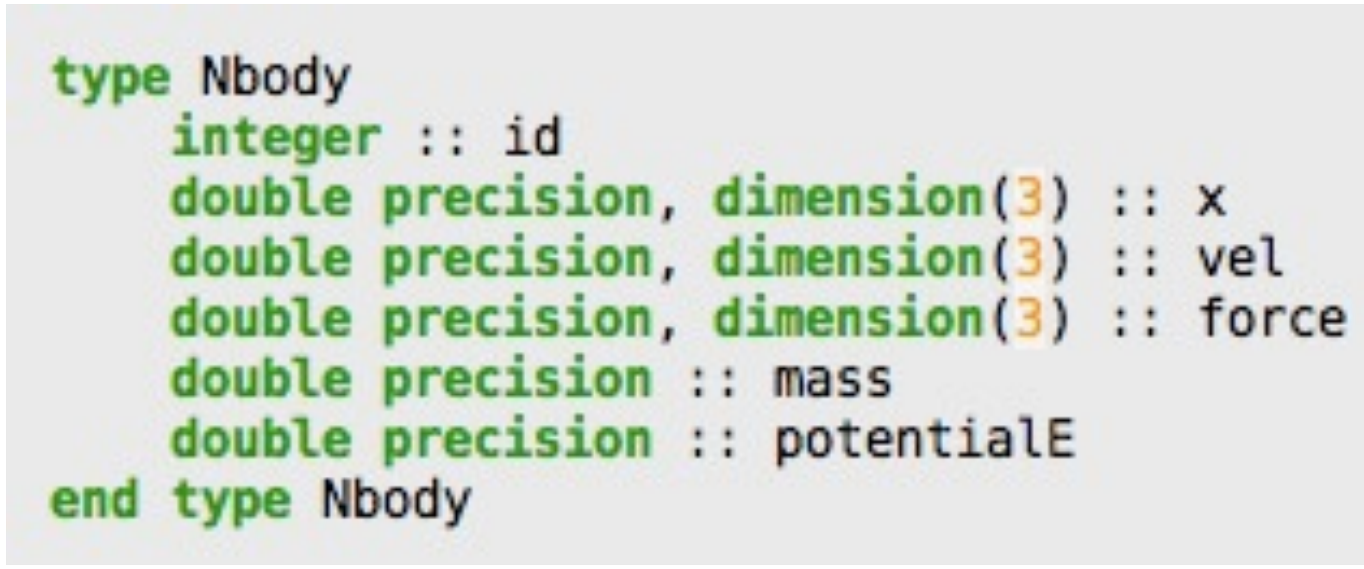
- What does type map look like for Nbody?



```
type Nbody
    integer :: id
    double precision, dimension(3) :: x
    double precision, dimension(3) :: vel
    double precision, dimension(3) :: force
    double precision :: mass
    double precision :: potentialE
end type Nbody
```

MPI Type Maps

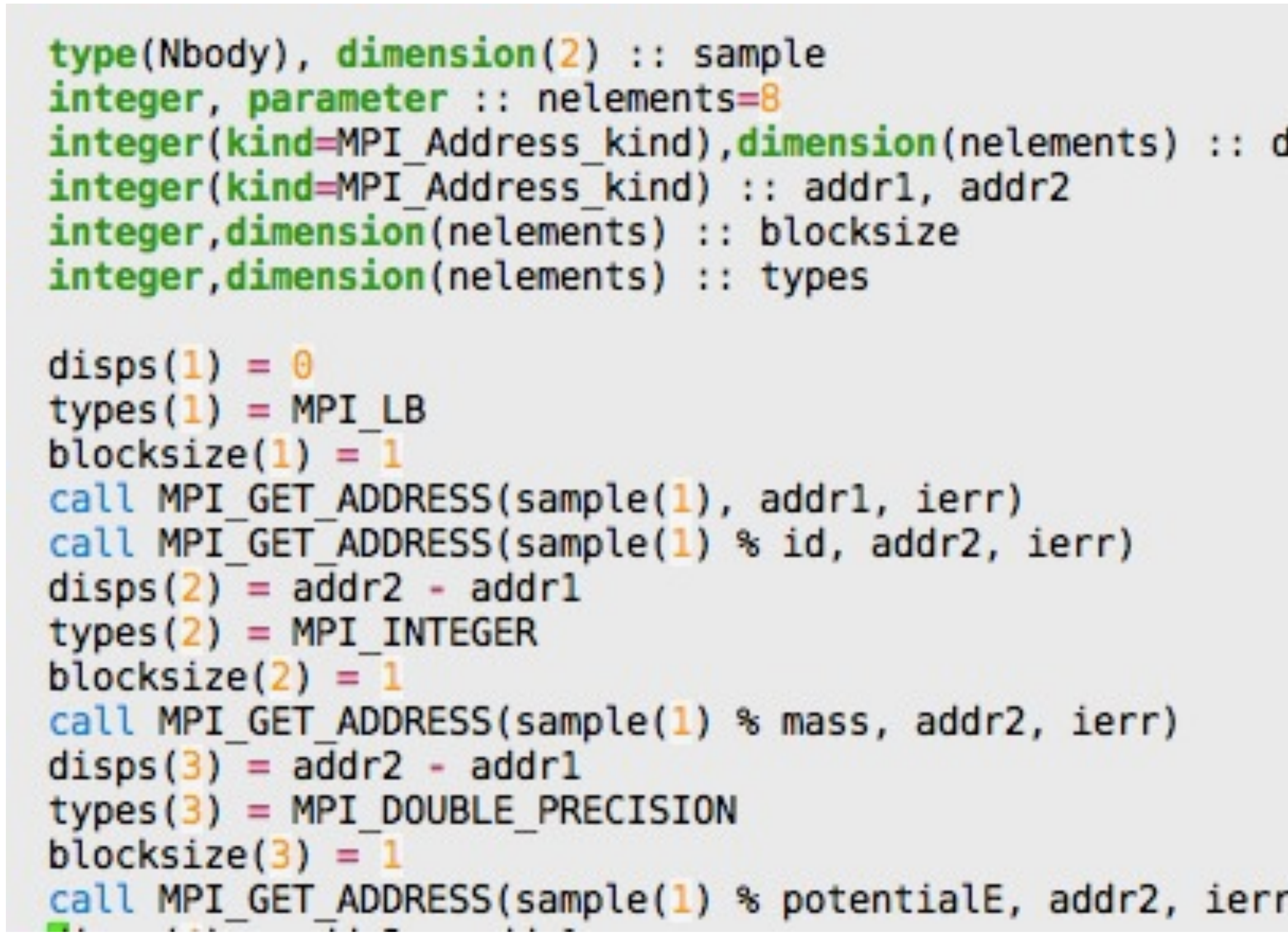
- What does type map look like for Nbody?
- How laid out in memory depends entirely on compiler, compiler options.
- alignment, padding...



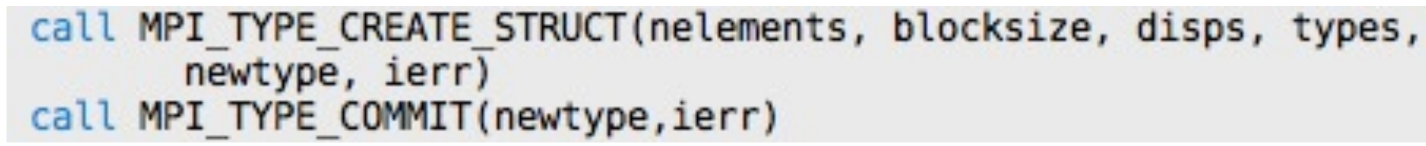
```
type Nbody
    integer :: id
    double precision, dimension(3) :: x
    double precision, dimension(3) :: vel
    double precision, dimension(3) :: force
    double precision :: mass
    double precision :: potentialE
end type Nbody
```

MPI Type Maps

- Use MPI_GET_ADDRESS to find addresses of different objects, and subtract the two to get displacements
- Build structure piece by piece.

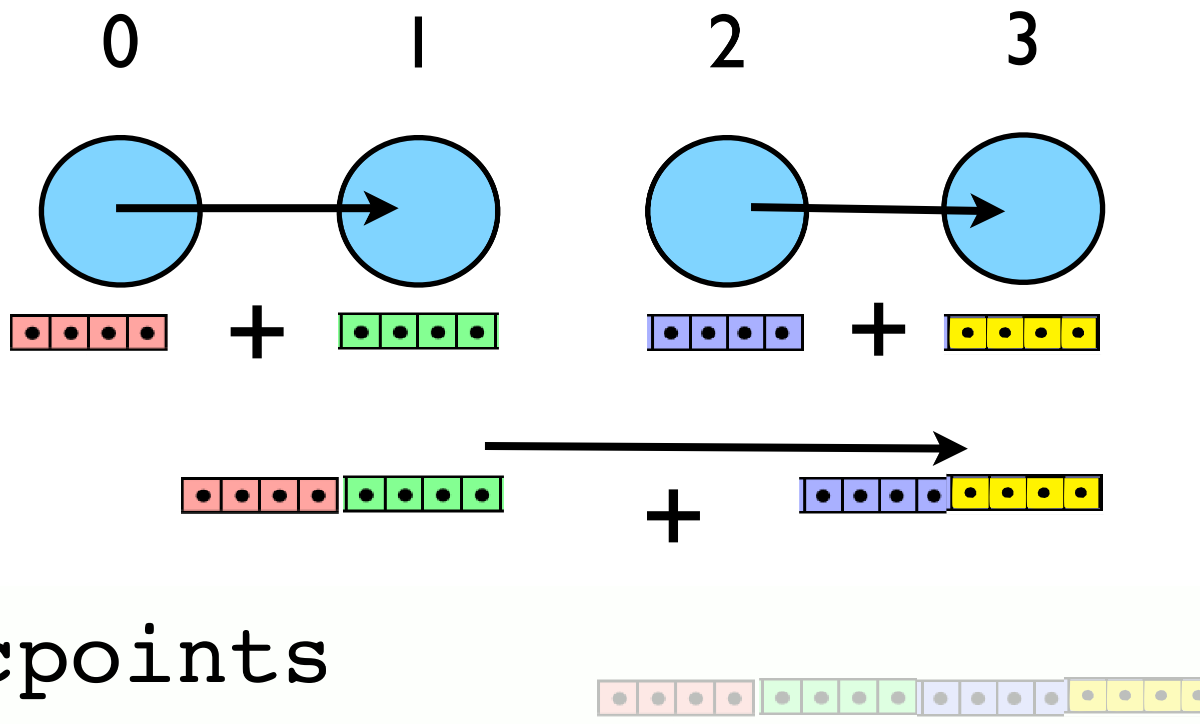


```
type(Nbody), dimension(2) :: sample
integer, parameter :: nelements=8
integer(kind=MPI_Address_kind),dimension(nelements) :: d
integer(kind=MPI_Address_kind) :: addr1, addr2
integer,dimension(nelements) :: blocksize
integer,dimension(nelements) :: types

disps(1) = 0
types(1) = MPI_LB
blocksize(1) = 1
call MPI_GET_ADDRESS(sample(1), addr1, ierr)
call MPI_GET_ADDRESS(sample(1) % id, addr2, ierr)
disps(2) = addr2 - addr1
types(2) = MPI_INTEGER
blocksize(2) = 1
call MPI_GET_ADDRESS(sample(1) % mass, addr2, ierr)
disps(3) = addr2 - addr1
types(3) = MPI_DOUBLE_PRECISION
blocksize(3) = 1
call MPI_GET_ADDRESS(sample(1) % potentialE, addr2, ierr
```



```
call MPI_TYPE_CREATE_STRUCT(nelements, blocksize, disps, types,
        newtype, ierr)
call MPI_TYPE_COMMIT(newtype,ierr)
```

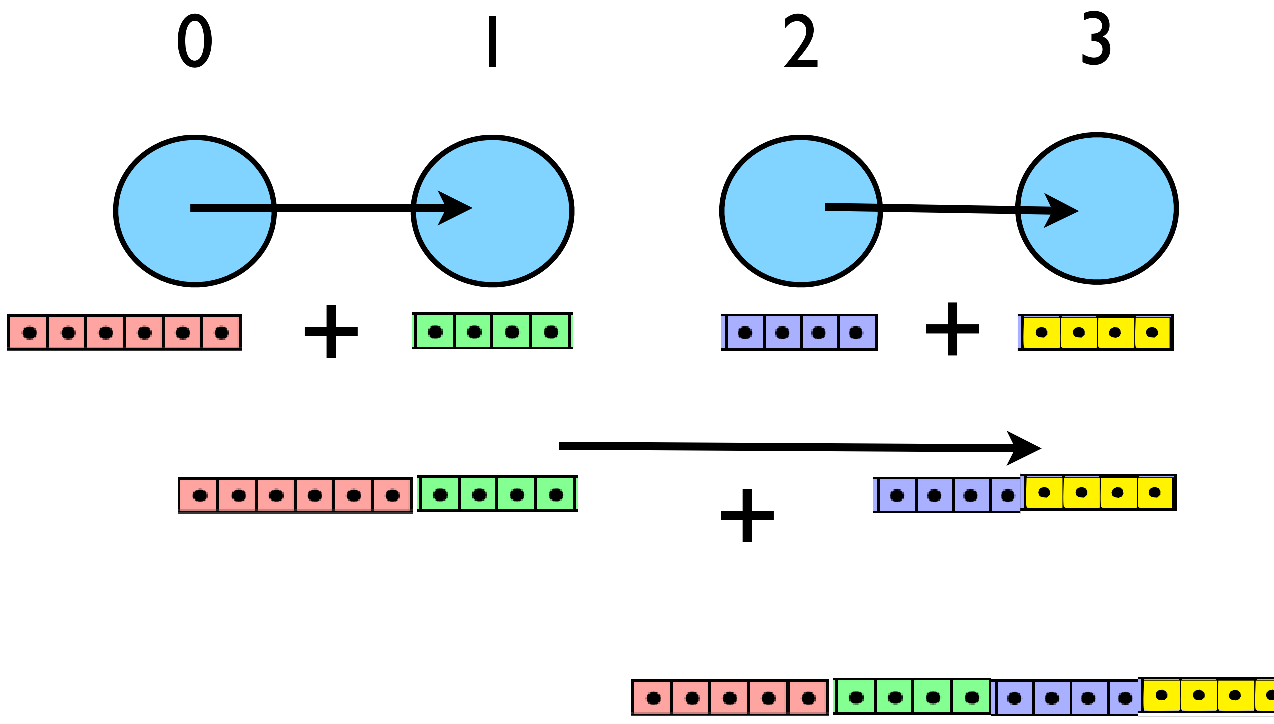
Another collective operation



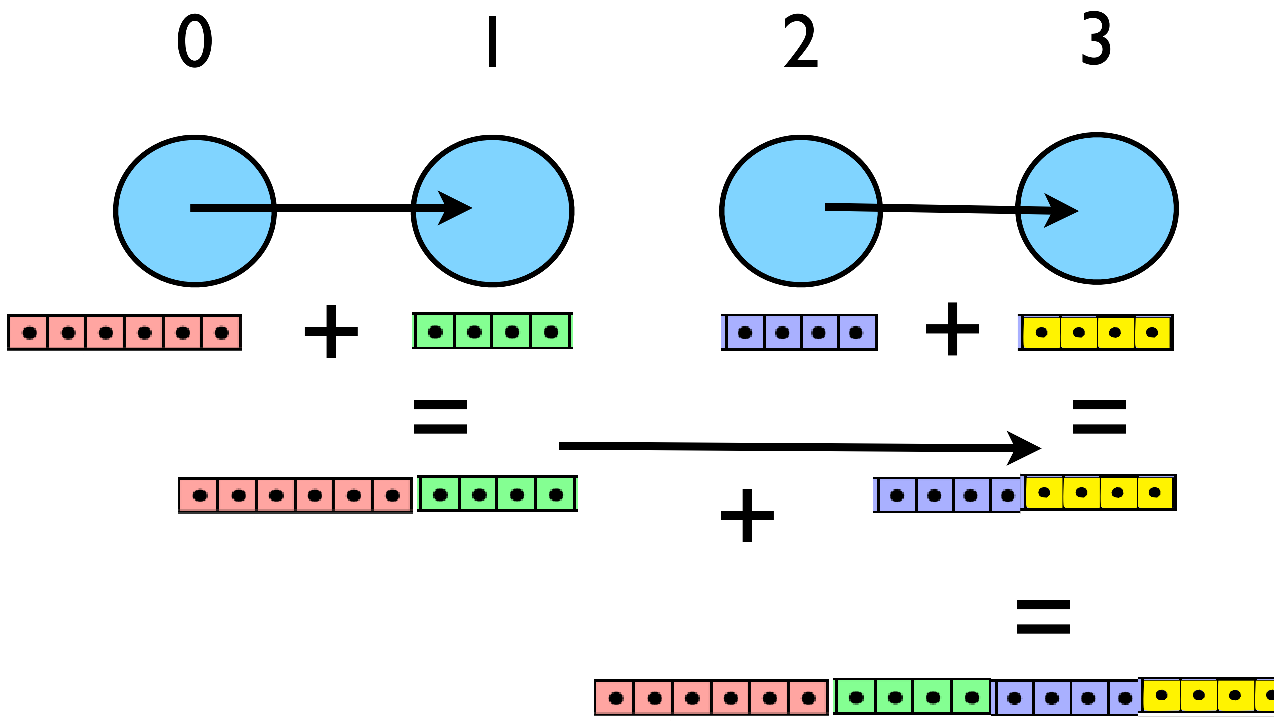
```
integer :: startp, endp, locpoints
integer :: ptype
type(Nbody), dimension(N) :: pdata

call MPI_Allgather(pdata(startp), locpoints, ptype,
            pdata, locpoints, ptype,
            MPI_COMM_WORLD, ierr)
```

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 <u>how many</u> each has and <u>where their chunk should go</u> in the global array



What if not same # of particles?

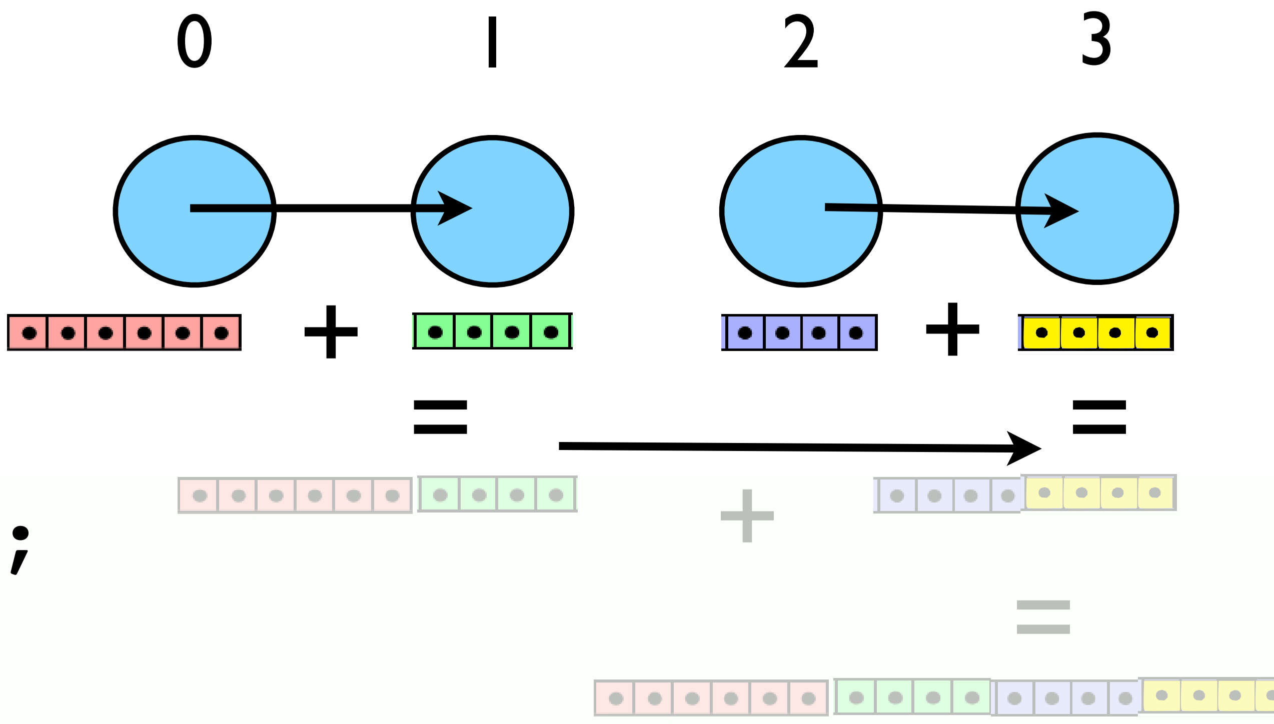


```
int MPI_Allgatherv ( void *sendbuf, int sendcount, MPI_Datatype sendtype,
                     void *recvbuf, int *recvcounts, int *displs,
                    MPI_Datatype recvtype, MPI_Comm comm )
```

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

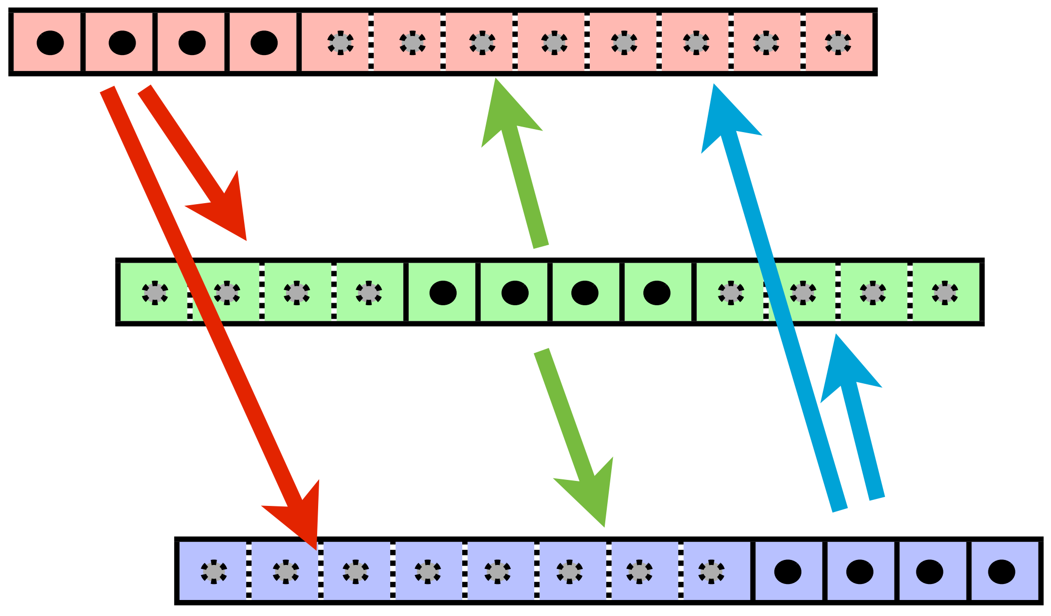
Where they should go; eg {0,6,10,14}

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_Particle,
              MPI_COMM_WORLD);
```

Other stuff about the nbody code

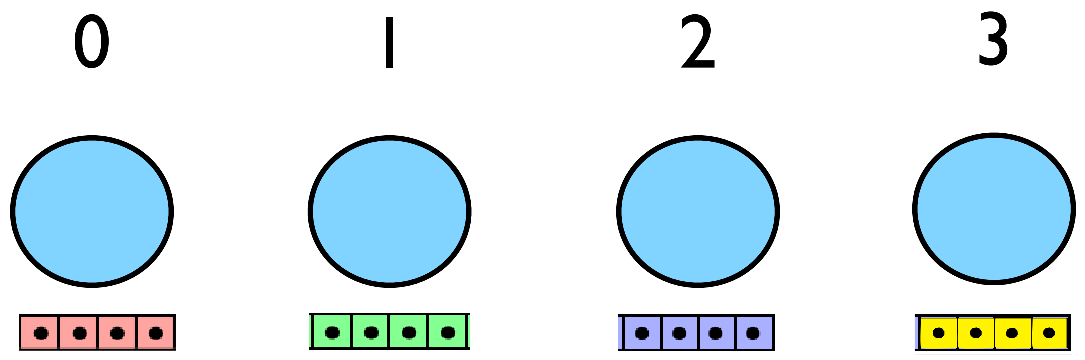


- At least plotting remains easy.
- Generally n-body codes keep track of things like global energy as a diagnostic
- We have a local energy we calculate on our particles;
- Should communicate that to sum up over all processors.
- Let's do this together

edit nbody-allgather.f90

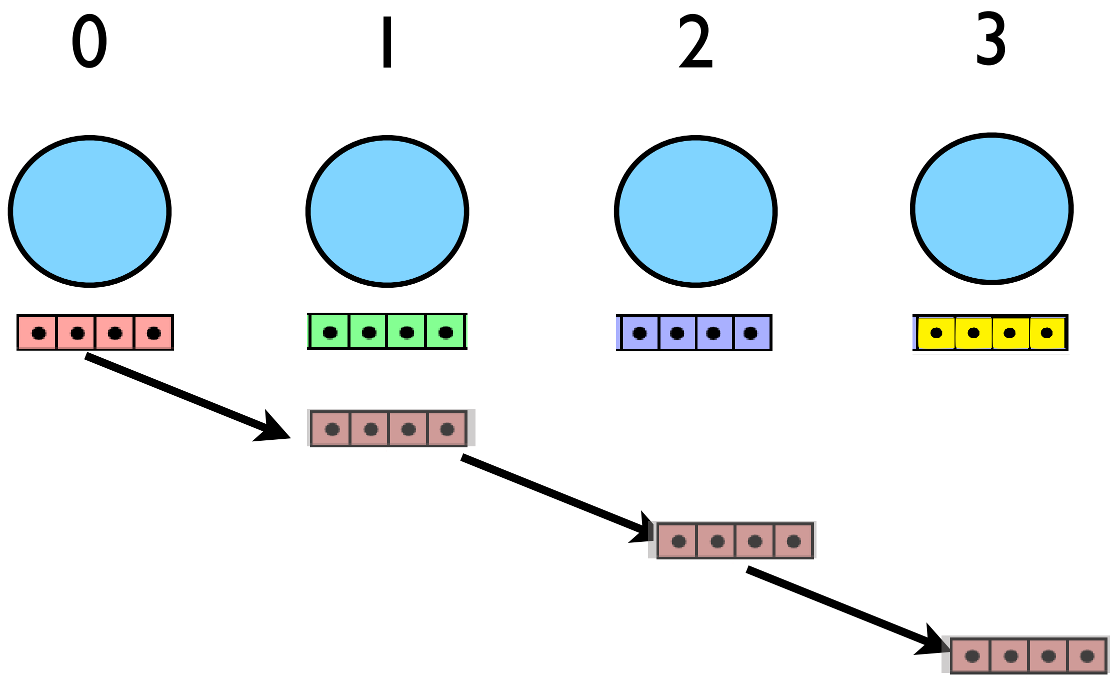
Problem (I) remains -- memory

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



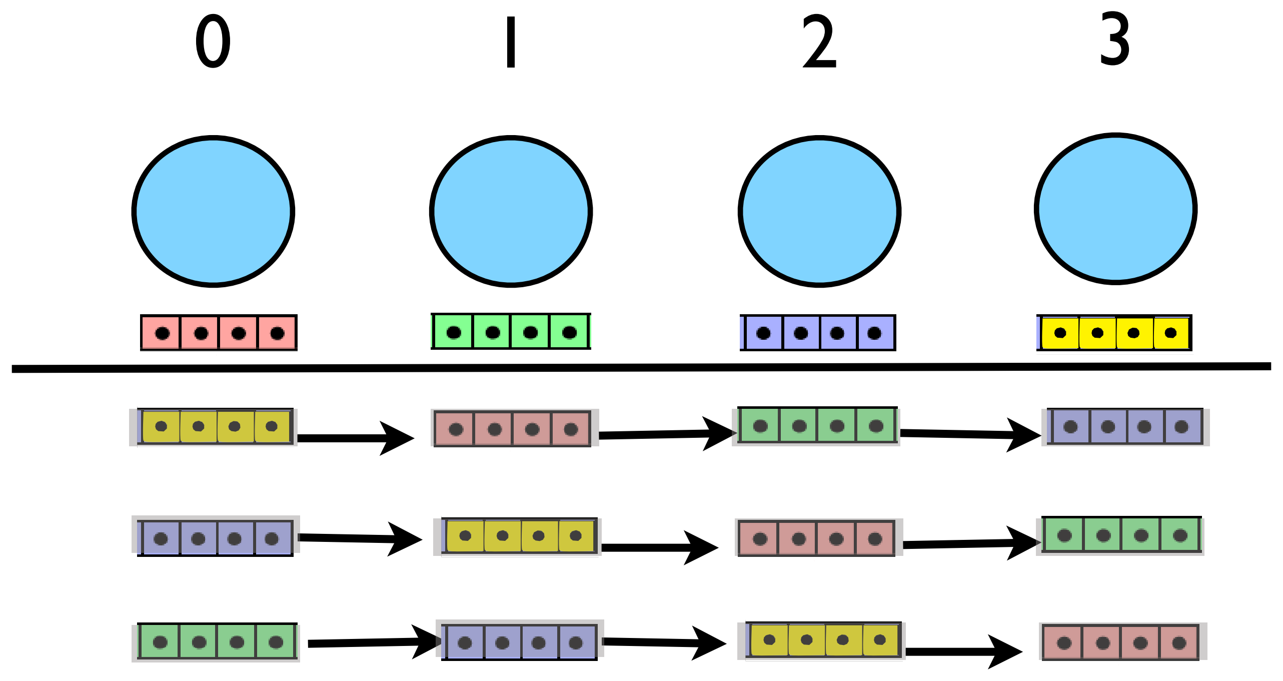
Pipeline

- 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)

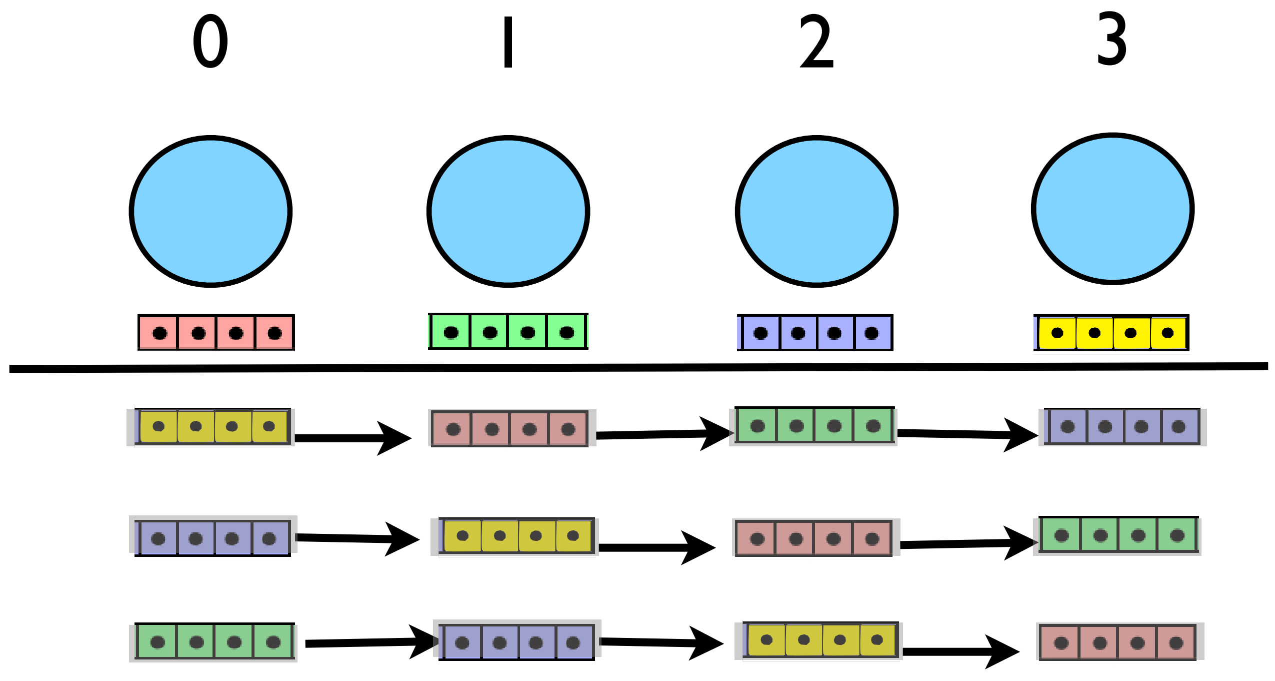


Pipeline

- 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

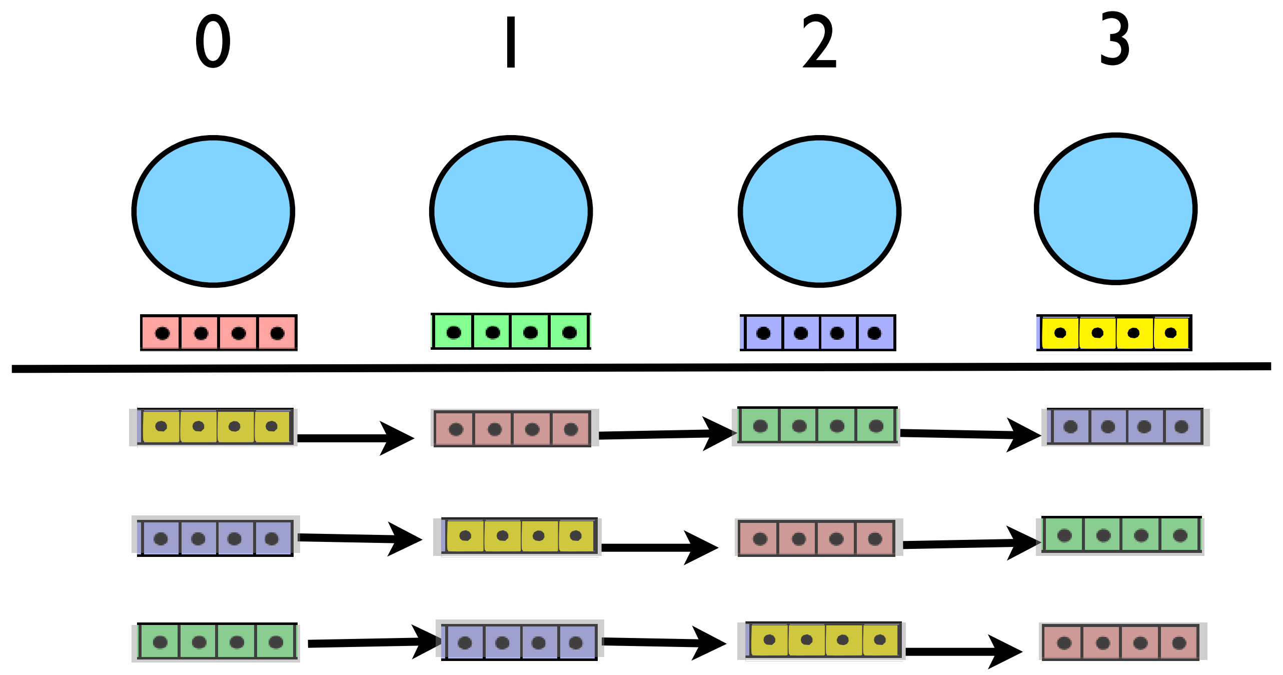
$$T_{\mathrm{comp}} = c_{\mathrm{grav}} \frac{N^2}{P} C_{\mathrm{comp}}$$

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

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

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

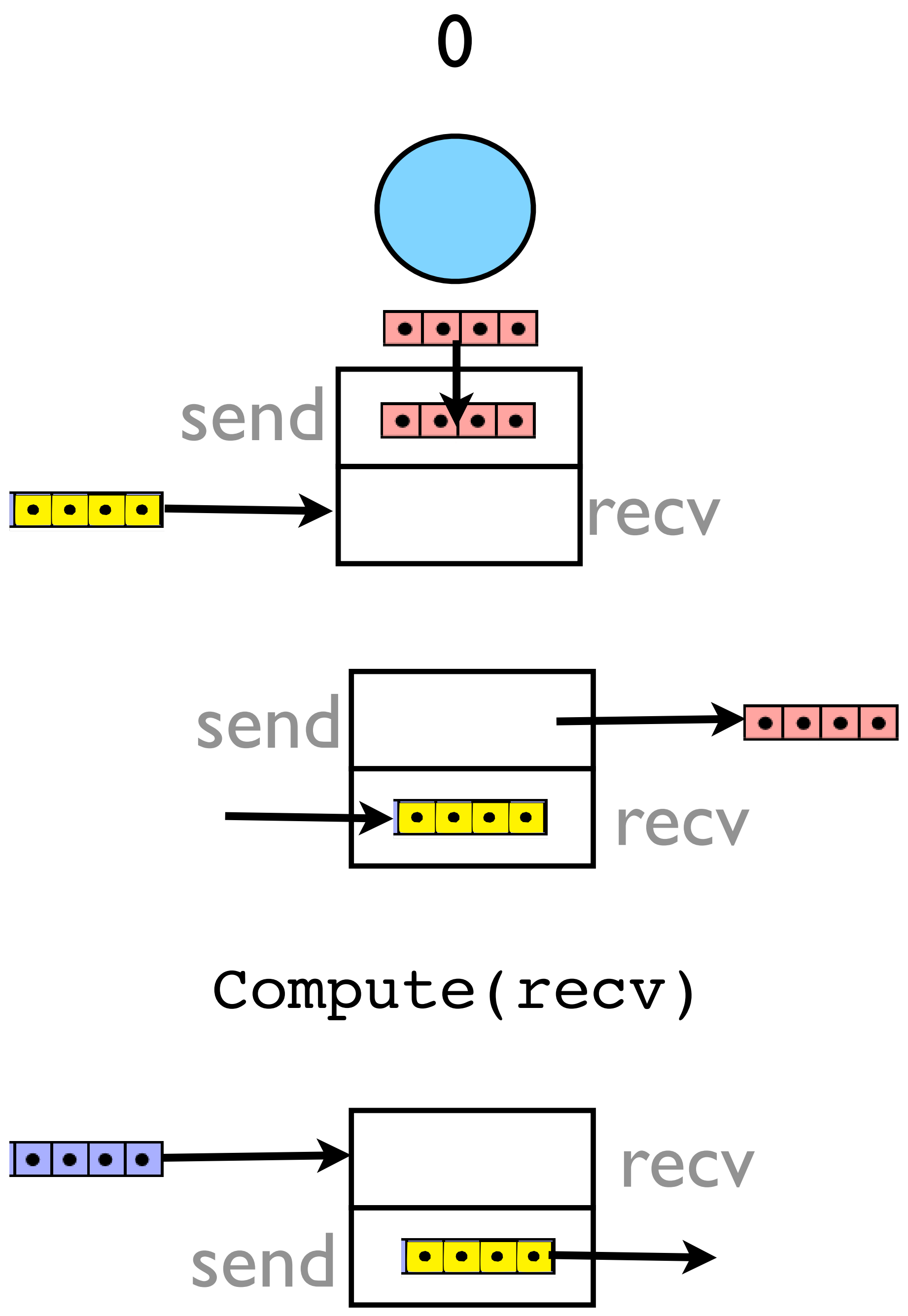
Pipeline



- 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.

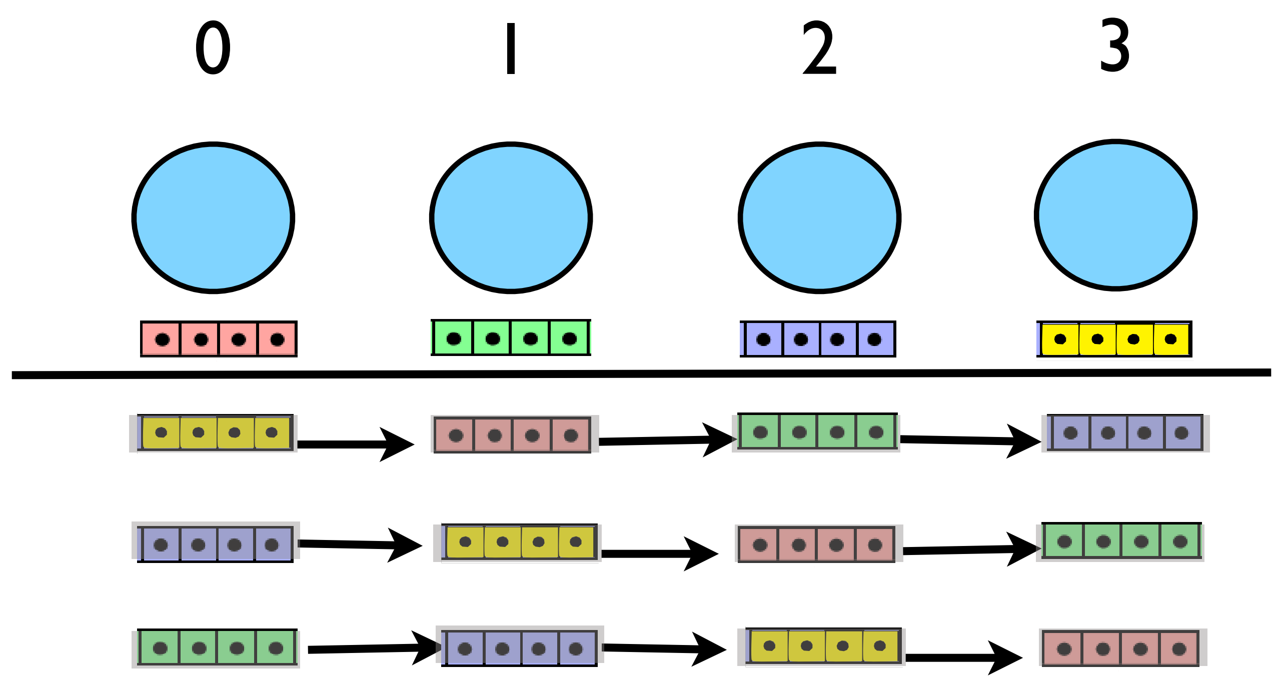
Pipeline

- 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.



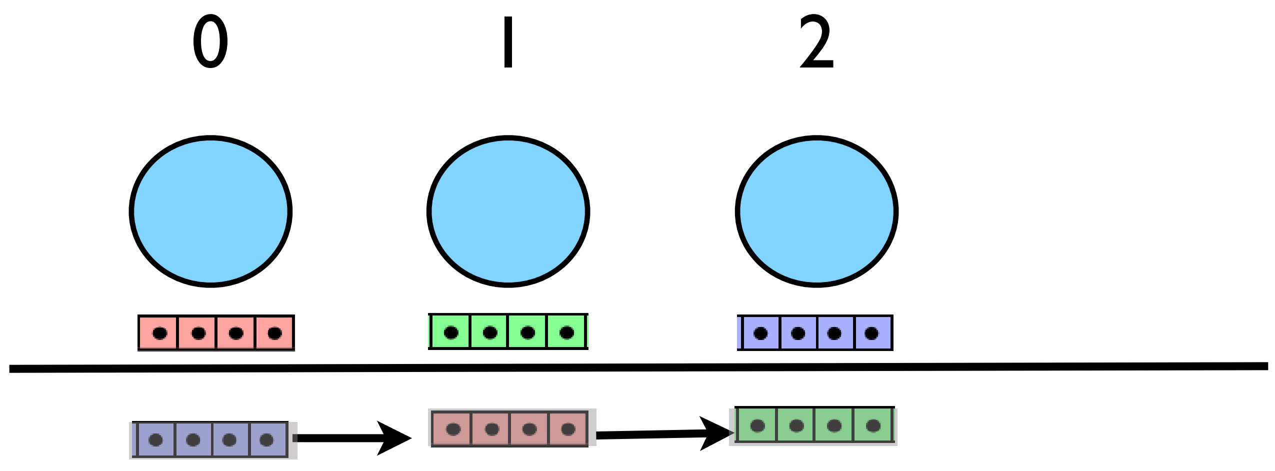
Pipeline

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



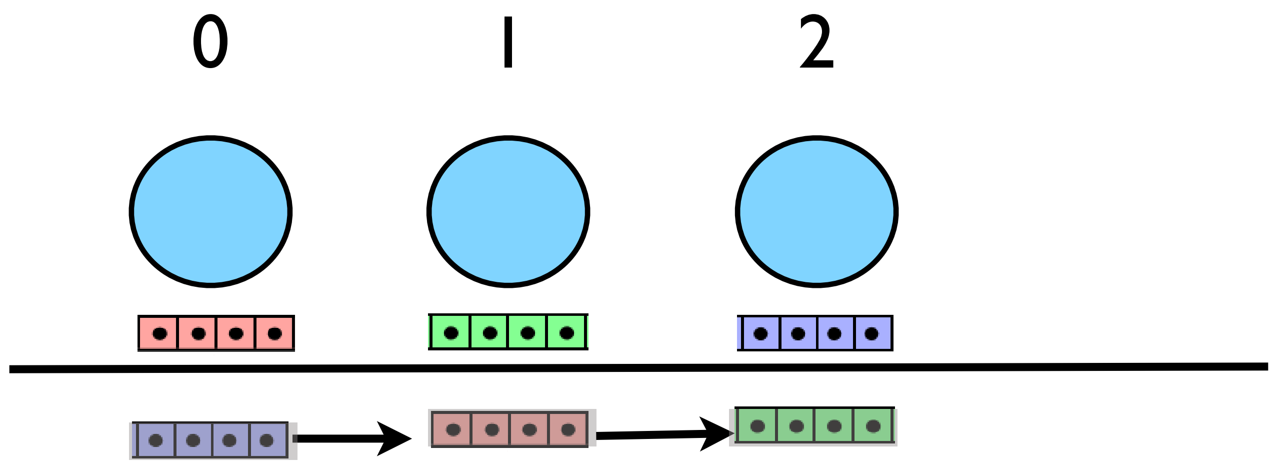
Pipeline

- 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...)



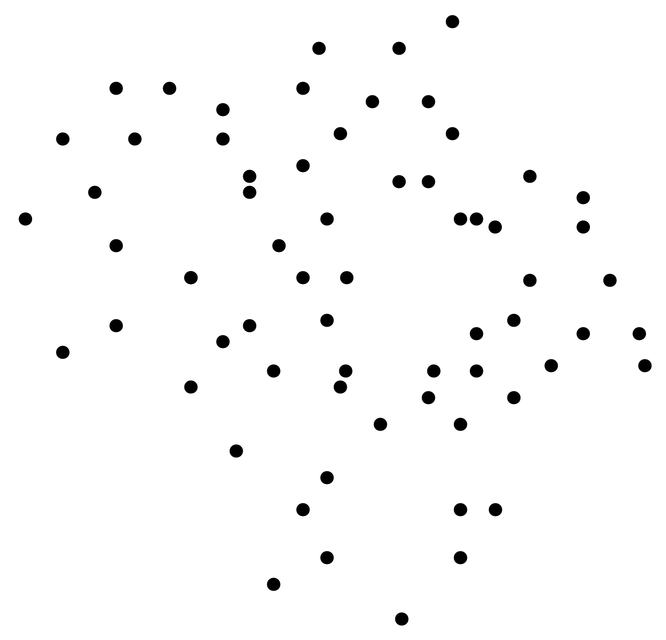
Pipeline

- 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?



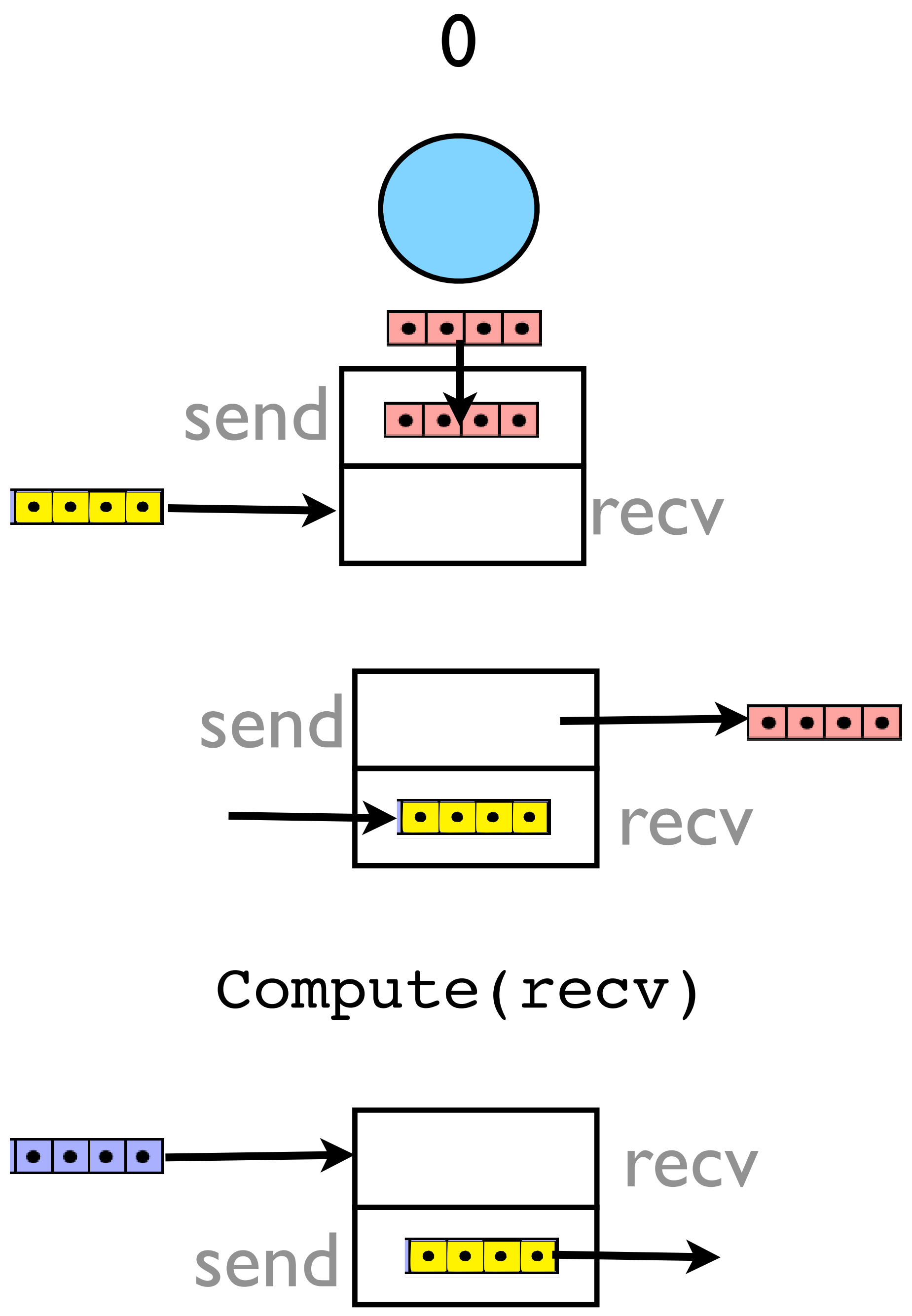
Displaying Data

- 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



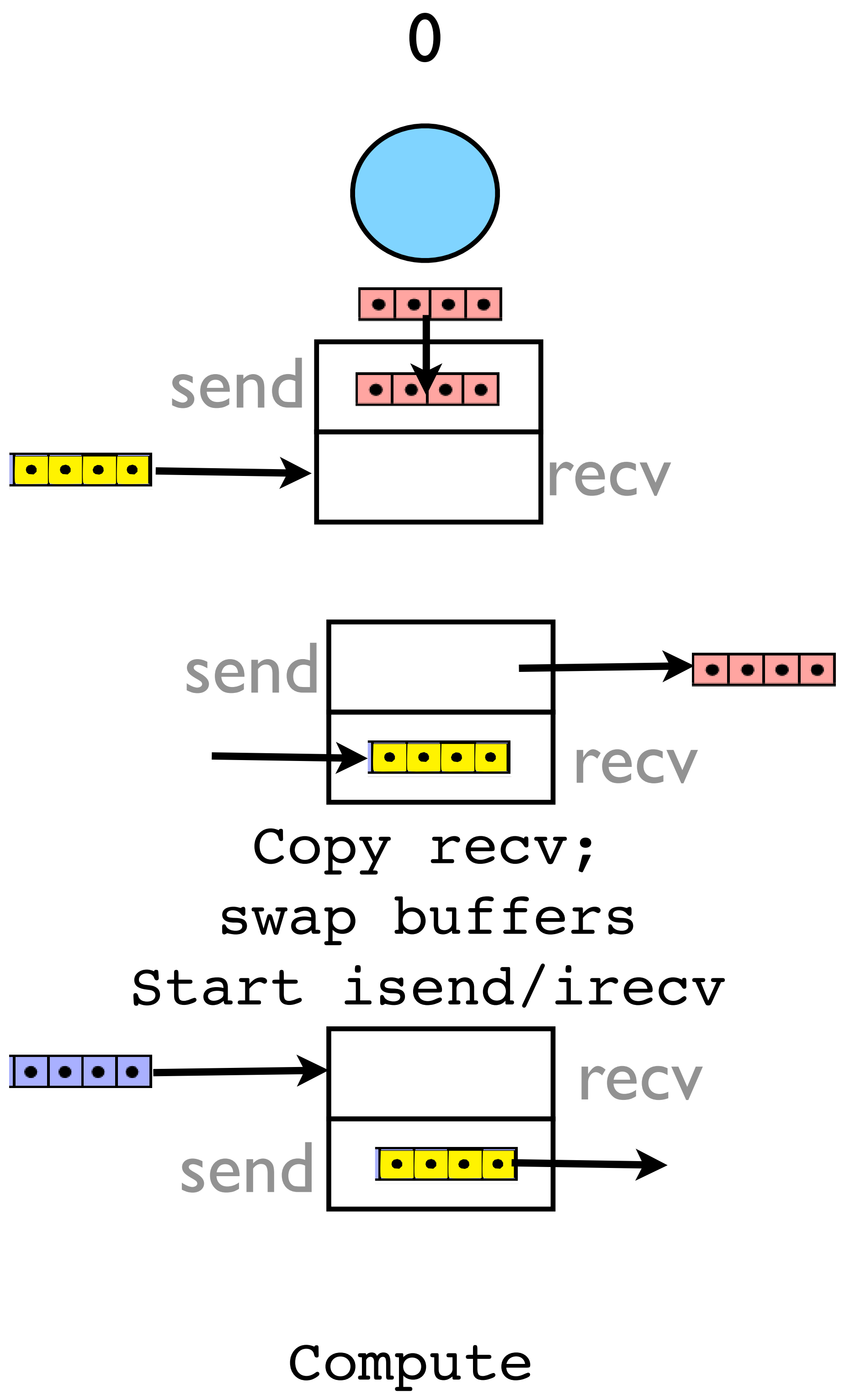
Overlapping Communication & Computation

- 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...



Overlapping Communication & Computation

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



Hands on

- Implement simplest pipeline (blocking)
- Try just doing one timestep, but calculating forces one block at a time
- Then sending blocks around
- Then non-blocking/double buffering