nBody solver using MPI

- The only communication among the tasks occurs when we're computing the forces, and, in order to compute the forces, each task/particle needs the position and mass of every other particle. ***MPI Allgather*** is expressly designed for this situation, since it collects on each process the same information from every other process.

- We use individual arrays for the masses, positions, and velocities. We'll also need an array for storing the positions of all the particles. If each process has sufficient memory, then each of these can be a separate array. In fact, if memory isn't a problem, each process can store the entire array of masses, since these will never be updated and their values only need to be communicated during the initial setup.
- Suppose that the array ***pos*** can store the positions of all n particles. Further suppose that ***vect _mpi_t*** is an MPI datatype that stores two contiguous doubles. Also suppose that ***n*** is evenly divisible by ***comm._sz*** and ***loc_n = n/comm._sz***. Then, if we store the local positions in a separate array, loc_pos, we can use the following call to collect all of the positions on each process:

```
MPI_Allgather(loc_pos, loc_n, vect_mpi_t,
    pos, loc_n, vect_mpi_t, comm);
```

- If we can't afford the extra storage for ***loc_pos***, then we can have each process ***q*** store its local positions in the ***qth*** block of ***pos***. That is, the local positions of each process should be stored in the appropriate block of each process' pos array:

  Process 0: pos[0], pos[1], . . . , pos[loc_n-1]
  Process 1: pos[loc_n], pos[loc_n+1], . . . , pos[loc_n + loc_n-1]
  . . .
  Process q: pos[q* loc_n], pos[q* loc_n +1], . . . , pos[q* loc_n + Loc_n-1]
  . . .

- With the pos array initialized this way on each process, we can use the following
  call to MPI_Allgather:
  MPI_Allgather(MPI_IN_PLACE, loc_n, vect_mpi_t, pos, loc_n, vect_mpi_t, comm);

- The first loc_n and vect_mpi_t arguments are ignored. However, it's not a bad idea to use arguments whose values correspond to the values that will be used, just to increase the readability of the program.

- In the program we've written, we made the following choices with respect to the
  data structures: .

- Each process stores the entire global array of particle masses.
- Each process only uses a single n-element array for the positions.
- Each process uses a pointer loc pos that refers to the start of its block of pos. Thus, on process, 0 local_pos = pos, on process 1 local_pos = pos + loc_n, and, so on.

- With these choices we can implement the basic algorithm with the pseudocode
  shown in Program.2 below.

```
1  Get input data;
2  for each timestep {
3     if (timestep output)
4        Print positions and velocities of particles;
5     for each local particle loc_q
6        Compute total force on loc_q;
7     for each local particle loc_q
8        Compute position and velocity of loc_q;
9     Allgather local positions into global pos array;
10 }
11 Print positions and velocities of particles;
```

Program 2: Pseudocode for the MPI version of the basic n-body solver

- Process 0 will read and broadcast the command line arguments.
  It will also read the input and print the results. In Line 1, it will need to distribute the input data. Therefore, Get input data might be implemented as
  follows:

```
if (my_rank == 0) {
   for each particle
      Read masses[particle], pos[particle], vel[particle];
}
MPI_Bcast(masses, n, MPI_DOUBLE, 0, comm);
MPI_Bcast(pos, n, vect_mpi_t, 0, comm);
MPI_Scatter(vel, loc_n, vect_mpi_t, loc_vel, loc_n, vect_mpi_t, 0,
    comm);
```

- So process 0 reads all the initial conditions into three n-element arrays. Since we're storing all the masses on each process, we broadcast masses. Also, since each process will need the global array of positions for the first computation of forces in the main for loop, we just broadcast ***pos***. However, velocities are only used locally for the updates to positions and velocities, so we scatter ***vel***.

- we gather the updated positions in Line 9 at the end of the body of the outer for loop of Program. This insures that the positions will be available for output in both Line 4 and Line 11.

- If we're printing the results for each timestep, this placement allows us to eliminate an expensive collective communication call: if we simply gathered

the positions onto process 0 before output, we'd have to call MPI_Allgather before the computation of the forces.

- we can implement the output with the following pseudocode:

```
Gather velocities onto process 0;
    if (my rank == 0) f
    Print timestep;
    for each particle
    Print pos[particle] and vel[particle]
    }
```

9.Parallelizing the reduced solver using MPI

- Before computing the forces, each process will need to gather a subset of the positions, and after the computation of the forces, each process will need to scatter some of the individual forces it has computed and add the forces it receives. Fig 6 shows the communications that would take place if we had three processes, six particles, and used a block partitioning of the particles among the processes.

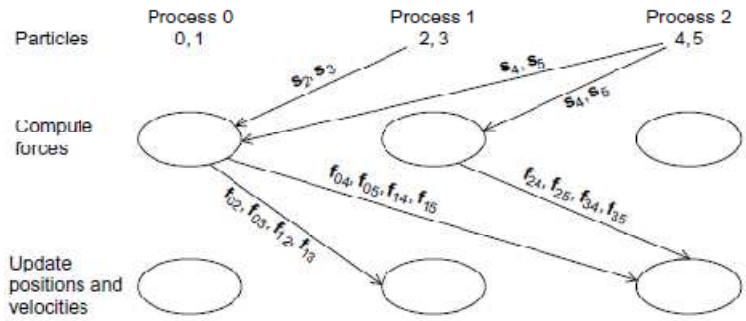

Fig 6:Communication in a possible MPI implementation of the reduced n-body solver

- Fortunately, there's a much simpler alternative that uses a communication structure that is sometimes called a ring pass. In a ring pass, we imagine the processes as being interconnected in a ring Fig.7

- Process 0 communicates directly with processes 1 and comm_sz-1, process 1 communicates with processes 0 and 2, and so on. The communication in a ring pass takes place in phases, and during each phase each process sends data to its "lower-ranked" neighbor, and receives data from its "higher-ranked" neighbor. Thus, 0 will send to comm._sz-1 and receive from 1. 1 will send to 0 and receive from 2, and so on. In general, process q will send to process . ***(q-1+comm_sz)%comm_sz*** and receive from process (***q+1)%comm_sz***.

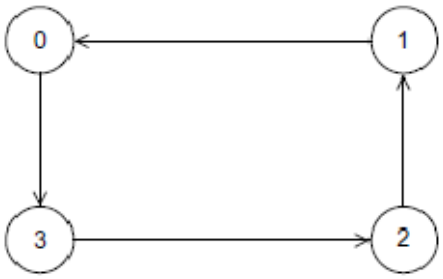

Fig 7 : A Ring of processes

- By repeatedly sending and receiving data using this ring structure, we can arrange that each process has access to the positions of all the particles.

- During the first phase, each process will send the positions of its assigned particles to its “lower-ranked” neighbor and receive the positions of the particles assigned to its higher-ranked neighbor.

- During the next phase, each process will forward the positions it received in the
  first phase. This process continues through comm._sz-1 phases until each process has received the positions of all of the particles. Figure 8 shows the three phases if there are four processes and eight particles that have been cyclically distributed.

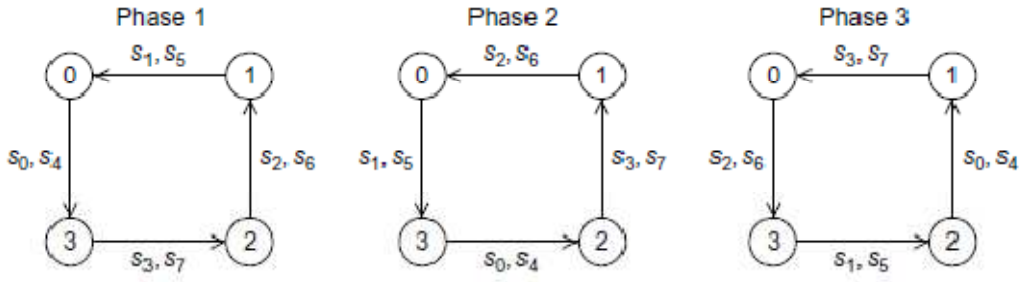

Fig 8: Ring pass of positions

- The virtue of the reduced algorithm is that we don’t need to compute all of the inter-particle forces since $f_{kq} = -f_{qk}$, for every pair of particles q and k. Using the reduced algorithm, the inter particle forces can be divided into those that are added into and those that are subtracted from the total forces on the particle. For example, if we have six particles, then the reduced algorithm will compute the force on particle 3 as

  F3 = -f03-f13-f23+f34+f35.

- The key to understanding the ring pass computation of the forces is to observe that the interparticle forces that are subtracted are computed by another task/particle, while the forces that are added are computed by the

owning task/particle. Thus, the computations of the interparticle forces on particle 3 are assigned as follows:

Force	f_{03}	f_{13}	f_{23}	f_{34}	f_{35}
Task/Particle	0	1	2	3	3

- Instead of simply passing loc_n = n/comm._sz positions, we also pass loc n forces. Then in each phase, a process can
    - compute interparticle forces resulting from interaction between its assigned particles and the particles whose positions it has received, and
    - once an interparticle force has been computed, the process can add the force into a local array of forces corresponding to its particles, and it can subtract the interparticle force from the received array of forces.

```
1  source = (my_rank + 1) % comm_sz;
2  dest = (my_rank - 1 + comm_sz) % comm_sz;
3  Copy loc_pos into tmp_pos;
4  loc_forces = tmp_forces = 0;
5
6  Compute forces due to interactions among local particles;
7  for (phase = 1; phase < comm_sz; phase++) {
8     Send current tmp_pos and tmp_forces to dest;
9     Receive new tmp_pos and tmp_forces from source;
10    /* Owner of the positions and forces we're receiving */
11    owner = (my_rank + phase) % comm_sz;
12    Compute forces due to interactions among my particles
13       and owner's particles;
14 }
15 Send current tmp_pos and tmp_forces to dest;
16 Receive new tmp_pos and tmp_forces from source;
```

Program 3: Pseudocode for the MPI implementation of the reduced n-body solver