nBody solver using OpenMP

- The pseudocode for the serial program:

```
for each timestep
 {
        if (timestep output) Print positions and velocities of particles;
        for each particle q
        Compute total force on q;
        for each particle q
        Compute position and velocity of q;
  }
```

- The two inner loops are both iterating over particles. So, in principle, parallelizing
  the two inner for loops will map tasks/particles to cores, and we might try something like this:

```
        for each timestep
        {
                if (timestep output) Print positions and velocities of particles;
                # pragma omp parallel for
                for each particle q
                Compute total force on q;
                # pragma omp parallel for
                for each particle q
                Compute position and velocity of q;
        }
```

- In the basic version the first loop has the following form:

```
        # pragma omp parallel for
                for each particle q
                 {
                        forces[q][X] = forces[q][Y] = 0;
                for each particle k != q
                        {
                        x_diff = pos[q][X] - pos[k][X];
                        y_diff = pos[q][Y] - pos[k][Y];
                dist = sqrt(x_diff*x_diff + y_diff*y_diff);
                dist cubed = dist_dist_dist;
                forces[q][X] -= G*masses[q]*masses[k] / dist_cubed * x_diff;
                forces[q][Y] -= G*masses[q]*masses[k] / dist_cubed * y_diff;
                        }
                }
```

- Since the iterations of the **for each particle q** loop are partitioned among the threads, only one thread will access **forces[q]** for any q. Different threads do access the same elements of the pos array and the **masses** array. However, these arrays are only **read** in the loop. The remaining variables are used for temporary storage in a single iteration of the inner loop, and they can be

private. Thus, the parallelization of the first loop in the basic algorithm won't introduce any race conditions.

- The second loop has the form:

```
# pragma omp parallel for
        for each particle q
         {
        pos[q][X] += delta_t * vel[q][X];
        pos[q][Y] += delta_t * vel[q][Y];
        vel[q][X] += delta_t/masses[q] * forces[q][X];
        vel[q][Y] += delta_t/masses[q] * forces[q][Y];
        }
```

- Here, a single thread accesses pos[q], vel[q], masses[q], and forces[q] for any particle q, and the scalar variables are only read, so parallelizing this loop also won't introduce any race conditions.

- Let's return to the issue of repeated forking and joining of threads. In our pseudocode, we have

```
for each timestep
 {
if (timestep output) Print positions and velocities of particles;
# pragma omp parallel for
for each particle q
        Compute total force on q;
# pragma omp parallel for
for each particle q
        Compute position and velocity of q;
}
```

- OpenMP provides the single directive for exactly this situation:
we have a team of threads executing a block of code, but a part of the code should only be executed by one of the threads. Adding the single directive gives us the following pseudocode:

```
# pragma omp parallel
        for each timestep
{
        if (timestep output)
{
# pragma omp single
        Print positions and velocities of particles;
}
# pragma omp for
        for each particle q
        Compute total force on q;
# pragma omp for
        for each particle q
        Compute position and velocity of q;
}
```

Parallelizing the reduced solver using OpenMP

- The reduced solver has an additional inner loop: the initialization of the forces array to 0. If we try to use the same parallelization for the reduced solver, we should also parallelize this loop with a for directive.

- The pseudocode for the parallelized version of reduced solver:

```
# pragma omp parallel
        for each timestep
        {
        If (timestep output)
        {
# pragma omp single
        Print positions and velocities of particles;
        }
# pragma omp for
        for each particle q
        forces[q] = 0.0;
# pragma omp for
        for each particle q
        Compute total force on q;
# pragma omp for
        for each particle q
        Compute position and velocity of q;
        }
```

- Parallelization of the initialization of the forces should be fine, as there's no dependence among the iterations. The updating of the positions and velocities is the same in both the basic and reduced solvers.
- In the reduced version, this loop has the following form:

```
# pragma omp for    /* Can be faster than memset */
for each particle q
{
force_qk[X] = force_qk[Y] = 0;
for each particle k > q
{
x_diff = pos[q][X] - pos[k][X];
y_diff = pos[q][Y] - pos[k][Y];
dist = sqrt(x diff_x diff + y diff_y diff);
dist cubed = dist_dist_dist;
force_qk[X] = G*masses[q]*masses[k]/dist_cubed* x_diff;
force_qk[Y]= G*masses[q]*masses[k]/dist _cubed * y_diff;
forces[q][X] += force_qk[X];
forces[q][Y] += force_qk[Y];
forces[k][X] -= force_qk[X];
forces[k][Y] -= force_qk[Y];
```

```
}
}
```

- The variables of interest are ***pos, masses***, and ***forces***, since the values in the remaining variables are only used in a single iteration, and hence, can be private. Also, as before, elements of the pos and masses arrays are only read, not updated. We therefore need to look at the elements of the forces array. In this version, unlike the basic version, a thread may update elements of the forces array other than those corresponding to its assigned particles. For example, suppose we have two threads and four particles and we're using a block partition of the particles. Then the total force on particle 3 is given by

$$\mathbf{F}_3 = -\mathbf{f}_{03} - \mathbf{f}_{13} - \mathbf{f}_{23}.$$

- Thread 0 will compute f03 and f13, while thread 1 will compute f23. Thus, the updates to forces[3] do create a race condition. In general, then, the updates to the elements of the forces array introduce race conditions into the code.
- Solution to this problem is to use a ***critical*** directive to limit access to the elements of the forces array. There are at least a couple of

ways to do this. The simplest is to put a ***critical*** directive before all the updates to forces

```
# pragma omp critical
{
forces[q][X] += force qk[X];
forces[q][Y] += force qk[Y];
forces[k][X] □= force qk[X];
forces[k][Y] □= force qk[Y];
}
```

- However, with this approach access to the elements of the forces array will be effectively serialized. Only one element of forces can be updated at a time,

and

contention for access to the critical section is actually likely to seriously degrade the performance of the program.

- An alternative would be to have one critical section for each particle. However,

as we've seen, OpenMP doesn't readily support varying numbers of critical sections, so we would need to use one lock for each particle instead and our updates would look like this:

```
omp_set_ lock(&locks[q]);
forces[q][X] += force_qk[X];
forces[q][Y] += force_qk[Y];
omp_unset_lock(&locks[q]);

omp set lock(&locks[k]);
forces[k][X] -= force_qk[X];
forces[k][Y] -= force_qk[Y];
```

```
omp_unset_lock(&locks[k]);
```

- This assumes that the master thread will create a shared array of locks, one for
  each particle, and when we update an element of the forces array, we first set the lock corresponding to that particle.

- Another possible solution is to carry out the computation of the forces in two phases. In the first phase, each thread carries out exactly the same calculations it carried out in the erroneous parallelization. However, now the calculations are stored in its own array of forces. Then, in the second phase, the thread that has been assigned particle q will add the contributions that have been computed by the different threads.

- In our example above, thread 0 would compute -f03 -f13, while thread 1 would compute -23. After each thread was done computing its contributions to the forces, thread 1, which has been assigned particle 3, would find the total force on particle 3 by adding these two values.

- Ex: Suppose we have three threads and six particles. If we're using a block partition of the particles, then the computations in the first phase are shown in Table 1. The last three columns of the table show each thread's contribution to the computation of the total forces. In phase 2 of the computation, the thread specified in the first column of the table will add the contents of each of its assigned rows—that is, each of its assigned particles.

Thread	Particle	Thread		
		0	1	2
0	0	$\mathbf{f}_{01}+\mathbf{f}_{02}+\mathbf{f}_{03}+\mathbf{f}_{04}+\mathbf{f}_{05}$	0	0
	1	$-\mathbf{f}_{01}+\mathbf{f}_{12}+\mathbf{f}_{13}+\mathbf{f}_{14}+\mathbf{f}_{15}$	0	0
1	2	$-\mathbf{f}_{02}-\mathbf{f}_{12}$	$\mathbf{f}_{23}+\mathbf{f}_{24}+\mathbf{f}_{25}$	0
	3	$-\mathbf{f}_{03}-\mathbf{f}_{13}$	$-\mathbf{f}_{23}+\mathbf{f}_{34}+\mathbf{f}_{35}$	0
2	4	$-\mathbf{f}_{04}-\mathbf{f}_{14}$	$-\mathbf{f}_{24}-\mathbf{f}_{34}$	$\mathbf{f}_{45}$
	5	$-\mathbf{f}_{05}-\mathbf{f}_{15}$	$-\mathbf{f}_{25}-\mathbf{f}_{35}$	$-\mathbf{f}_{45}$

Table 1: First-Phase Computations for a Reduced Algorithm with Block Partition.

- Table 2 shows the same computations if we use a cyclic partition of the particles. if we compare this table with the table that shows the block partition, it's clear that the cyclic partition does a better job of balancing the load.

Thread	Particle	Thread		
		0	1	2
0	0	$f_{01}+f_{02}+f_{03}+f_{04}+f_{05}$	0	0
1	1	$-f_{01}$	$f_{12}+f_{13}+f_{14}+f_{15}$	0
2	2	$-f_{02}$	$-f_{12}$	$f_{23}+f_{24}+f_{25}$
0	3	$-f_{03}+f_{34}+f_{35}$	$-f_{13}$	$-f_{23}$
1	4	$-f_{04}-f_{34}$	$-f_{14}+f_{45}$	$-f_{24}$
2	5	$-f_{05}-f_{35}$	$-f_{15}-f_{45}$	$-f_{25}$

Table 2: First-Phase Computations for a Reduced Algorithm with Cyclic Partition

- To implement this, during the first phase our revised algorithm proceeds as before, except that each thread adds the forces it computes into its own subarray
of ***loc_forces***:

```
#  pragma omp for
   for each particle q {
      force_qk[X] = force_qk[Y] = 0;
      for each particle k > q {
         x_diff = pos[q][X] - pos[k][X];
         y_diff = pos[q][Y] - pos[k][Y];
         dist = sqrt(x_diff*x_diff + y_diff*y_diff);
         dist_cubed = dist*dist*dist;
         force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
         force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff;

         loc_forces[my_rank][q][X] += force_qk[X];
         loc_forces[my_rank][q][Y] += force_qk[Y];
         loc_forces[my_rank][k][X] -= force_qk[X];
         loc_forces[my_rank][k][Y] -= force_qk[Y];
      }
   }
```

- During the second phase, each thread adds the forces computed by all the threads for its assigned particles:

```
#  pragma omp for
   for (q = 0; q < n; q++) {
      forces[q][X] = forces[q][Y] = 0;
      for (thread = 0; thread < thread_count; thread++) {
         forces[q][X] += loc_forces[thread][q][X];
         forces[q][Y] += loc_forces[thread][q][Y];
      }
   }
```