DTS202TC Fundamentals of Parallel Computing

Tutorial 4: N-Body Problem

Xi'an Jiaotong-Liverpool University

西交利物浦大學

N-Body Problem

We try to calculate the position and velocity of some interacting particles after a period of time. For example, astronomers may want to know the positions and velocities of certain stars, while chemists want to know the positions and velocities of molecules or atoms.

The inputs to the problem are the **mass** of the particle, the **position** and **velocity** at the beginning, and the outputs are generally the **velocity** and **position** of the particle in a specified time series, or just the **velocity** and **position** of the particle after a specified time period

Force of particle k on particle q

$$f_{qk}(t) = -\frac{Gm_q m_k}{|s_q(t) - s_k(t)|^3}[s_q(t) - s_k t]$$

$s_q(t)$ Is the location of particle q at time t

$s_k(t)$ Is the location of particle k and time t

G is the gravitational constant

m_q,m_k is the mass of particles q and k, respectively

Force of particle k on particle q

$$f_{qk}(t) = -\frac{Gm_q m_k}{\left|s_q(t) - s_k(t)\right|^3}\left[s_q(t) - s_k t\right]$$

We set we have 0,1,…..n-1 particle. The total force acting on particle q

$$F_q(t) = \sum_{k=0,k!=q}^{n-1} f_{qk}(t) = -Gm_q \sum_{k=0,k!=q}^{n-1} \frac{m_k}{\left|s_q(t) - s_k(t)\right|^3}\left[s_q(t) - s_k t\right]$$

With Newton's second law we can know that $$s = v_0 t + \frac{a}{2} t^2$$

Acceleration is the second derivative of displacement $$a = s''$$

So we can get the total force $$F_q(t) = m_q a_q(t) = m_q s_q''(t)$$

With this function we will have acceleration

$$s_q''(t) = -G \sum_{j=0, j!=q}^{n-1} \frac{m_j}{|s_q(t) - s_j(t)|^3} [s_q(t) - s_j(t)]$$

What we want to do:

We want to get the displacement $\mathbf{S_q(t)}$ at time **t** and the speed $v_q(t) = s'(t)$

Suppose you need to calculate in a time series: t=0, Δt, 2Δt, ... , TΔt

The input to the program is **n**(the number of particle), Δt

T and the weight of every particle, initial
location and initial speed

In general we choose three dimension to represent location and speed because they move in 3D space. However here we assume they move on the **2D space** so we use 2D dimension represent them

Implement with openMP

Serial way

```
1     Get input data;
2     for each timestep {
3        if (timestep output) Print positions and velocities of
             particles;
4        for each particle q
5           Compute total force on q;
6        for each particle q
7           Compute position and velocity of q;
8     }
9     Print positions and velocities of particles;
```

The two inner loops are both iterating over particles. So, in principle, parallelizing the two inner for loops will map tasks/particles to cores.

Before parallel **it we should** check whether it has race condition

The first loop

```
#  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.

```
#  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];
   }
```

The second loop

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.

Final Implement with openMP

```
#  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;
   }
```

OpenMP provides a ***single*** instruction: a group of threads executes a block of code, but a portion of that block can only be executed by one of those threads, thereby preventing each thread from printing location and speed

Threads in the same group are used by both iterations of the inner loop and the outer loop.

Implement with openMP

```
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;
```

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()

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

Treating the computation associated with a single particle as a complex task makes it intuitive to parallelize the basic algorithm with **MPI**. Communication between tasks only occurs when calculating the forces, and each task/particle needs to obtain the positions and masses of the other particles, in which case you need the **MPI_Allgather** function, which collects the same information for each thread from the other threads.

In the shared-memory implementations, we collected most of the data associated with a single particle (mass, position, and velocity) into a single struct. However, if we use this data structure in the MPI implementation, we'll need to use a derived datatype in the call to **MPI_Allgather**, and communications with derived datatypes tend to be slower than communications with basic MPI types because it will make more sense to use individual arrays for the masses, positions, and velocities

Comparison and Discussion

Table 6.6 Run-Times for OpenMP and MPI *n*-Body Solvers (times in seconds)

Processes/ Threads	OpenMP		MPI	
	Basic	Reduced	Basic	Reduced
1	15.13	8.77	17.30	8.68
2	7.62	4.42	8.65	4.45
4	3.85	2.26	4.35	2.30

We see that the basic OpenMP solver is a good deal faster than the basic MPI solver. This isn't surprising since MPI Allgather is such an expensive operation.

Comparison and Discussion

We set **n** particles and **p** processes or threads

The MPI solver allocates *n* **double**s per process for the masses. It also allocates 4*n*/*p* **double**s for the tmp pos and tmp forces arrays, so in addition to the local velocities and positions, the MPI solver stores $n + 4n/p$

For openMP

The OpenMP solver allocates a total of 2*pn* + 2*n* **double**s for the forces and *n* **double**s for the masses, so in addition to the local velocities and positions, the OpenMP solver stores $3n/p + 2n$

Comparison and Discussion

Thus, the difference in the local storage required for the OpenMP version and the MPI version is $n - n/p$

In other words, if *n* is large, the local storage required for the MPI version is substantially less than the local storage required for the OpenMP version. So, for a fixed number of processes or threads, we should be able to run much larger simulations with the MPI version than the OpenMP version.

For more detail: Please refer to the chacter 6 of textbook.
Pacheco, P., & Malensek, M. (2021). An Introduction to Parallel Programming (**2nd ed.**).