**CS 267 - Applications of Parallel Computers - Homework 3**
**Spring 96 - Due March 8**

For this programming assignment, we will do a different kind of particle simulation, which is in the category of a "cellular automaton" simulation of fluid flow. The particles are hard, frictionless spheres of radius 1, moving in a two-dimensional box, which exert forces only when they collide with one another, or with the walls of the box. We assume collisions are perfectly elastic, i.e. they conserve kinetic energy as well as momentum, and since the balls are frictionless, they only exert force normal to one another at the collision point, not tangent. The same applies to collisions with the bounding box.

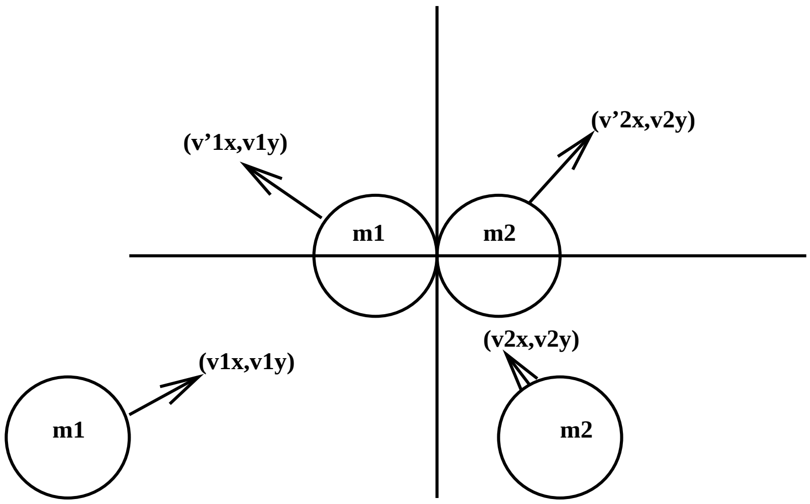
Thus, until a collision, a particle moves at constant velocity. To figure out their velocities after a collision, assume without loss of generality that particle one has mass $m_1$ and is moving (generally right) with velocity $(v_{1x}, v_{1y})$, and that particle two has mass $m_2$ is moving (generally left) with velocity $(v_{2x}, v_{2y})$. Suppose that when they collide, their common tangent at the collision point is vertical (in the general case, you have to rotate the coordinate system to apply the following computation). Then only the $x$-components of their velocities change, since we assume there is no friction. To compute the new $x$-velocities $v'_{1x}$ and $v'_{2x}$, we write down the law of conservation of momentum in the $x$-direction:

$$m_1 v_{1x} + m_2 v_{2x} = m_1 v'_{1x} + m_2 v'_{2x}$$

and conservation of kinetic energy:

$$m_1(v_{1x}^2 + v_{1y}^2) + m_2(v_{2x}^2 + v_{2y}^2) = m_1({v'}_{1x}^2 + v_{1y}^2) + m_2({v'}_{2x}^2 + v_{2y}^2) \quad .$$

We can solve these two equations in two unknowns for the new velocities $v'_{1x}$ and $v'_{2x}$ (the answers are simple linear combinations of $v_{1x}$ and $v_{2x}$, with coefficients depending on $m_1$ and $m_2$).




When a particle hits the wall, since we assume the walls are rigid, the particle just bounces back with the negative of its normal component velocity. You should confirm that if you take $v_{2x} = v_{2y} = 0$ and $m_2 \to \infty$, you get a rigid wall as the limiting case.

One interesting physical quantity such a collection of bouncing balls exhibits is *pressure*. One computes the pressure on the walls as follows. When a particle with mass $m_i$ hits a wall with incident normal velocity component $v_i$ and bounces back with normal velocity $-v_i$, the wall absorbs a momentum of $2m_iv_i$. If we sum over all particle-wall collisions occuring during one second, this is the total force exerted on the wall during that second. Dividing by the area of the wall yields the pressure.

So the basic loop of the simulation will be as follows.

```
input the particle masses, and their initial positions and velocities
for each time step
    for each particle
        determine whether a collision will occur in this time step
        if a collision will occur, then
            update the particle's position and velocity appropriately
            if the collision is with a wall, accumulate the force on the wall
        else
            update the particle's position and velocity appropriately
        endif
    endfor
    compute the pressure
    display the positions of (some) particles
endfor
```

In principle, there are at least two ways to divide the work among parallel processors. The first is to have each processor responsible for updating the position of a fixed set of $n/p$ particles, where $n$ is the total number of particles and $p$ is the number of processors. The second is to geometrically divide the box into $p$ subboxes, and have one processor responsible for the particles inside one subbox. The first approach will require all particles to visit all processors in each time step (as in gravity), since particles in distant processors may or may not collide. In the second approach, the only communication necessary is for particles in a small region near the boundary of the subbox, since only these have a chance of colliding with particles in neighboring processors in any time step (provided the time step is small enough). In other words, the second algorithm exploits the *physical locality* of the particles as they move to assign them to processors to achieve *locality of computation*. When a particle's center crosses a box boundary, it becomes the responsibility of the next processor to simulate.

To make your program go fast, you will find it attractive to similarly store the balls local to a processor in a spatial data structure which limits the number of collision detection comparisons one has to make. For both the subbox owned by a processor, as well as the sub-subboxes into which a processor divides its local balls, you should think about whether the boxes should be long rectangles, squares, or some other shape, in order to minimize the number of comparisons and/or communications needed.

You will discover that there is a tradeoff between the speed and the accuracy of the simulation. For example, accounting for all possible multiple ball collisions (more than 2 at a time) is complicated and time consuming. Ignoring possible multiple collisions degrades the accuracy slightly, but multiple collisions are physically very unlikely (compared to 2-ball collisions) and turns out not to affect statistics like pressure at all, so they can be safely ignored. Another unlikely possibility is a sequence of related collisions (like a multiple car pile-up on the freeway) all occuring in a short period of time. So it is physically accurate enough to assume that each ball can participate in at most one collision within a time step.

You should write a program to do this simulation in either Split-C or CMMD, as you prefer. Your program should take as input 1) the dimension of a square box in which to do the simulation; 2) a list of balls, with their (common) radius, and individual masses, initial positions, and initial velocities, 3) a timestep and a final time.

Interesting quantities you should measure or display include:

- Speedup $S_p$ and Efficiency $E_p$ for $p$ equals 1, 2, 4, 8, ... processors. Your program for 1 processor should not do any communication, so that this speedup graph is "fair".
- Scaled speedup $S_p^s$ and Scaled Efficiency $E_p^s$. Let the number of particles grow proportionally to the number of processors.
- Pressure. This is computed as described above. Recalling the formula $PV = nRT$ from high-school chemistry, double the number of particles and see if the pressure doubles, and halve the volume and see if the pressure doubles. (We may or may not have enough computer time to simulate enough particles to reproduce these relationships very accurately.)
- Graphics. Try displaying the positions of just a few particles; otherwise you won't be able to see anything. Try having particles of two masses (light and heavy), and color the two kinds differently to see what happens. For example, you might have the box full of light particles initially and then inject heavy particles.

Later, we will discuss how to merge the two kinds of simulations you have done so far, i.e. involving both force fields like gravity and collisions. To make your progarm modular, write a subroutine collide(p1,p2) which determines if two particles are going to collide, and a subroutine interaction(p1,p2) which determines what happens during a collision. Thus, more complicated (say chemical) interactions could be hidden inside these two subroutines.

Here are some general questions you should also answer.

1. How many messages (number of messages and number of bytes), and how many calls to "collide" are needed per time step by the first algorithm (particles assigned to processors by number), for $n$ processors, and $p$ particles?
2. Assuming $n$ particles are uniformly distributed on $p$ processors, how many messages, bytes sent, and calls to "collide", are used by the second algorithm (particles assigned to processors by location)? This is a more interesting modeling question, because it will depend on the geometry of the subboxes, the time step (and so how many "boundary" particles need to be communicated on average), and how cleverly your

data structure can avoid unnecessary calls to "collide" for particles local to a processor. Is is better to have square subboxes, rectangular subboxes, or something else? Instrument your code to count these quantities, and compare with your model. Your goal should be a model which can predict the resources needed by your program with some accuracy (within 10% would be very good).

3. It is very likely that the bouncing balls in a closed box will be uniformly distributed, making the assumptions of the last question appropriate. Suppose the the "interaction" function, or gravity or a current, made the particles much less uniformly distributed. How could this effect the performance of your system? What algorithmic change would you propose to compensate for this?

Here is how one might use such a program in a simple physical application. Suppose one has two pipes containing different gases (say fuel and oxygen) that one wants to mix well in a small space. How should the walls of the mixing areas be shaped to guarantee good mixing? For example, in the following figure the gray obstacles have been placed to try to mix the two gases before they exit at the bottom. The shapes of these obstacles can be optimized by running the simulation with a variety of obstacle shapes, and measuring the distribution of gases 1 and 2 across the mouth of the exit at the bottom; if particles of both gases are evenly distributed across the mouth, the mixing has succeeded.