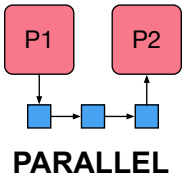
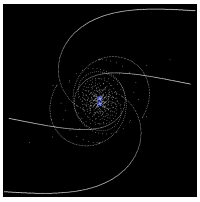


CISC 372: Parallel Computing

University of Delaware, Spring 2022

HW 5



This assignment is due at noon on ***Wednesday, Apr. 13****. Topics: Pthreads, n-body problems.*

1. Instructions

You should already have a directory named `hw` in your personal repository. Add a subdirectory `hw05` to that directory.

Please follow instructions on naming of files and directories, and formatting, precisely, since some aspects of the grading are done programmatically. If your solution is mis-graded because of a minor formatting issue, you will have an opportunity to correct it, but it will save everyone time and effort if we can get it right the first time.

2. Background: gravitational n-body problems

2.1. **Newton's universal law of gravitation.** The universal law of gravitation describes the gravitational force that one object (or "body") exerts on another. If m_0 and m_1 are the masses of the two bodies, and r is the distance between the center of mass of one and the center of mass of the other, then the force exerted by body 1 on body 0 has magnitude

$$F = \frac{Gm_0m_1}{r^2}.$$

Note:

- G is the universal gravitational constant, approximately 6.67430×10^{-11} $\mathrm{m}^3 \cdot \mathrm{kg}^{-1} \cdot \mathrm{s}^{-2}$. This is a very small number—masses have to get pretty large to induce a nontrivial force. We are usually interested in things like planets, stars, and black holes, i.e., astronomical bodies.
- The force falls off with the square of the distance. For example, if you multiply the distance by 3, the force is divided by 9. Hence objects that are sufficiently far away generally have very little gravitational impact compared to closer objects.
- The direction of the force imposed by body 1 on body 0 is along the line leading from the center of mass of body 0 to the center of mass of body 1.
- Body 0 also imposes a force on body 1. The magnitude of the two forces are the same, but they point in opposite directions. See Figure 1.
- The standard unit of force is the newton, $N = \mathrm{kg} \cdot m/s^2$.

Force is actually a vector, i.e., it has an x, y, and z-component. In this exercise we will assume all bodies lie in the plane $z = 0$, so vectors can be described using only an x and a y-component.

FIGURE 1. Two bodies (objects) exert gravitational forces on each other. The distance between the centers is r. The masses are m_0 and m_1. The magnitude of each force is Gm_0m_1/r^2.

Given the positions (x_0, y_0) and (x_1, y_1) of the two bodies, we can find the x and y-components of the force exerted by body 1 on body 0 as follows:

$$r = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$$
$$F_x = \frac{x_1 - x_0}{r} \cdot F = \frac{Gm_0m_1(x_1 - x_0)}{r^3}$$
$$F_y = \frac{y_1 - y_0}{r} \cdot F = \frac{Gm_0m_1(y_1 - y_0)}{r^3}$$

The force exerted by body 1 on body 0 is then the vector $\mathbf{F} = (F_x, F_y)$.

When multiple objects exert forces on body 0, the total force is the *vector sum* of the individual forces. The vector sum is just the component-wise sum. For example, the sum of the two forces $(4.0, 0.0)$ and $(4.0, 3.0)$ is $(8.0, 3.0)$. See Figure 2.

Once you know the total force $\mathbf{F}$ acting on an object, the motion of that object is determined by Newton's second law of motion

$$\mathbf{F} = m\mathbf{a},$$

where m is the mass of the object, and $\mathbf{a}$ is its acceleration. Acceleration is the rate of change of velocity $\mathbf{v}$ with respect to time, and velocity is the rate of change of position $\mathbf{p}$ with respect to time. $\mathbf{F}$, $\mathbf{a}$, $\mathbf{v}$, and $\mathbf{p}$ are all vectors—each has an x and a y-component.

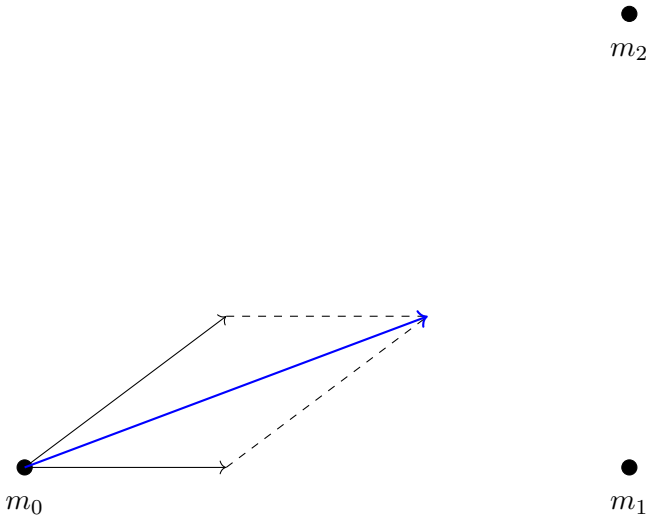


FIGURE 2. Bodies 1 and 2 exert a gravitational force on body 0. The total force exerted on body 0 (blue arrow) is the vector sum of those forces.

2.2. **The n-body problem.** In an n-body problem, you are given the masses of n bodies, as well as the initial position and velocity of each. The goal is to determine the motion of the bodies over time.

For $n = 2$, a general analytical solution is known, i.e., there are explicit mathematical formulas for the positions of the two bodies over time (`https://en.wikipedia.org/wiki/Gravitational_two-body_problem`.) No general solution is known for any $n \geq 3$ (though solutions for some special cases are known). This is unfortunate, because astro-physicists, for example, would like to study phenomena such as galaxy collisions, involving millions of bodies.

When we can't solve problems using math, we turn to computation. An n-body system can be simulated by iterating over time, incrementing the current time by some small Δt at each step. The program maintains the current position $\mathbf{p}_i = (x_i, y_i)$ and velocity $\mathbf{v}_i = (v_{x,i}, v_{y,i})$ of each body. At each time step:

(1) For each i $(0 \leq i < n)$, compute the total force $\mathbf{F}_i$ on body i from all $n-1$ bodies other than i, using the universal law of gravitation and taking the vector sum:

$$\mathbf{F}_i = \sum_{j \neq i} \left(\frac{G m_i m_j (x_j - x_i)}{r_{i,j}^3}, \frac{G m_i m_j (y_j - y_i)}{r_{i,j}^3} \right),$$

where $r_{i,j} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ is the distance between bodies i and j.

(2) Use this to compute the total acceleration $\mathbf{a}_i$ of each body:
- $\mathbf{a}_i = \mathbf{F}_i / m_i$

(3) For each i, compute the new velocity $\mathbf{v}_i'$ and the new position $\mathbf{p}_i'$ of body i:
- $\mathbf{v}_i' = \mathbf{v}_i + \mathbf{a}_i \cdot \Delta t$
- $\mathbf{p}_i' = \mathbf{p}_i + \mathbf{v}_i' \cdot \Delta t$

You can see in (1) above that the asymptotic time to do an update is $O(n^2)$ — there is an outer loop over i with an inner loop over j, both with n or $n-1$ iterations. So n-body simulations are computationally demanding in the extreme.

A simple optimization: we can combine (1) and (2) and simplify as follows:

$$\mathbf{a}_i = \frac{\mathbf{F}_i}{m_i} = \sum_{j \neq i} \left(\frac{G m_j (x_j - x_i)}{r_{i,j}^3}, \frac{G m_j (y_j - y_i)}{r_{i,j}^3} \right).$$

You can probably think of other optimizations and refactorings to reduce the number of operations in this update formula.

3. Problem 1: parallelize `nbody.c`

The given program performs a simulation of an n-body gravitational system. The masses and initial positions and velocities of the bodies are given. All measurements are in standard metric units.

A problem also specifies a color and radius for each body. These are used to create an animation in which each body is depicted as a circle of the given color with radius the specified number of pixels. The color and radius are not otherwise used in the simulation.

The source code is in `372-2022S/code/src/seq/nbody`. The main source code is `nbody.c`. There is also a header file `nbody.h`, and a separate file for each specific problem: `planets-elliptical.c`,

figure8.c, galaxy.c, and galaxy2.c. A complete program is constructed by linking nbody.c with one of the specific problem files.

Two structure types are defined in nbody.h. The Body struct records the radius, color, and mass of a body—these are the static (unchanging) data. It also records the *state* of the body. The state is a State struct, and maintains the position and velocity vectors for the body—this is the dynamic (changing) part of the data.

The program uses the ANIM library. It creates an animation of kind NBODY. See the documentation for function ANIM_Create_nbody in anim.h. For an NBODY animation, ANIM_Write_frame expects a buffer of $2n$ doubles specifying the coordinates of each body: x_0, y_0, x_1, y_1, ..., x_{n-1}, y_{n-1}.

Create the videos for the first 3 animations using the provided Makefile. (You may want to hold off on galaxy2 for now; it consists of 2002 bodies and will take some time.) For example, type make planets-elliptical.mp4. Study the code and the resulting animations and make sure you understand what it is doing.

Now for your job: speed up this program by using Pthreads to parallelize the update procedure. Keep it simple: at each time step, spawn the threads and distribute the bodies to the threads using your favorite distribution scheme. Each thread computes the new state for its bodies. Since the updates of two different bodies are independent of each other, the threads do not require any synchronization or communication. Wait for the threads to complete and then resume sequential execution for the remainder of the time step.

You are provided a Makefile which will help you compile and test your program. The tests work by comparing the output of the sequential and parallel versions, as usual.

Work in a directory hw05/nbody. Call your program nbody_pthread.c. It should take two command line arguments: first, the number of threads to use, then the file name for the output. Copy over the given Makefile; it should work without alteration. There is no need to copy nbody.h or the specific problem files, since the Makefile will find these in the 372-2022S repository.

Note: You may have to do a wee bit of refactoring to get this to work. Pthreads requires a function of a certain signature. Each thread will need its ID number to figure out which bodies it must update. Should the pointer swap be performed by all threads, or just the main thread?

After you believe your program is correct, evaluate it on the big problem, galaxy2, using at most one full Bridges2 node. (You can't use more than one node, since you are using Pthreads, and there is no shared memory between nodes.) Create a subdirectory bridges and put in it a batch file galaxy2.sh. The stdout output (which should include the time) should go to a file galaxy2.out, also in the bridges directory.

As a baseline, the sequential version of galaxy2 took 234s on one core of Bridges2 RM partition. (For best times, I recommend using RM, not RM-shared, for this problem.) A time of 23s for a correct solution will get full credit.

Be sure to look at galaxy2.mp4 so you can see what happens when two spiral galaxies form and then collide.

4. Problem 2: A Producer-Consumer System

In program pc0.c, two threads are created: a producer and consumer. The producer produces a sequence of random integers in the interval $[-1, 1000]$. These integers are inserted into a *bounded buffer* (aka "circular buffer"). The consumer pulls items out of the buffer and adds them up. The run ends after the producer produces -1 and the consumer consumes the -1. Just before terminating, the program prints the final sum. Some busy work has been inserted into the producer and consumer, just to mix up the interleavings and make the runs more interesting.

Let us describe the bounded buffer, which acts as a FIFO queue with a fixed capacity n; see `https://en.wikipedia.org/wiki/Circular_buffer`. The data are stored in an array b (called `buf` in the code) of length n. Two integers are also needed to keep track of the state of the buffer: i (`first`) is the index of the first (oldest) element currently in the buffer, and k (`size`) is the number of elements currently in the buffer. The indexes of the k elements, from oldest to newest, are:

$$i\%n,\ (i+1)\%n,\ \ldots,\ (i+k-1)\%n.$$

Initially, $i = k = 0$. To enqueue a new value x, x is assigned to $b[(i+k)\%n]$ and k is incremented. To dequeue a value, $b[i]$ is read, i is replaced by $(i+1)\%n$, and k is decremented.

Some implementations store the indexes of the first and last elements, rather than the index of the first and the size. Other implementations use pointers instead of indexes. The essential ideas are the same.

Now for a bounded buffer, or any FIFO queue, to work correctly, the following rules must be enforced: one can enqueue only when the buffer is not full, i.e., when $k < n$. And one can dequeue only when the buffer is not empty, i.e., when $k > 0$. The given program gets this wrong—it does not use proper synchronization techniques to control when an enqueue or dequeue occurs. It also has data races. Run this program multiple times and you may see different results. Try `make bigtest0` to see a tabulation of the final results from 1000 different runs. In a correct version, the result should always be the same, since the program does not set a random seed.

Your job is to fix the program, using mutexes and condition variables. Accesses to shared variables should be protected by mutex lock(s). The producer should wait until the buffer is not full before enqueing; the consumer should wait until the buffer is not empty before dequeing. Signals must be sent at appropriate times to "wake up" a thread that may be waiting. Call the corrected program pc.c.

Here is an excerpt of the output from a correct run, with buffer capacity of 5, with some debugging printfs uncommented:

```
$ ./pc.exec 5
...
Producer: inserting into buffer... 519
Producer: inserting into buffer... 979
Producer: inserting into buffer... 727
Producer: inserting into buffer... 946
Producer: inserting into buffer...  47
Consumer: removing from buffer.......... 519
Consumer: removing from buffer.......... 979
Consumer: removing from buffer.......... 727
```

```
Producer: inserting into buffer... 448
Producer: inserting into buffer... 489
Producer: inserting into buffer... 349
Consumer: removing from buffer.......... 946
Consumer: removing from buffer..........  47
Consumer: removing from buffer.......... 448
Consumer: removing from buffer.......... 489
Consumer: removing from buffer.......... 349
Producer: inserting into buffer... 168
Producer: inserting into buffer... 163
Producer: inserting into buffer... 481
Producer: inserting into buffer... 593
...
Producer: inserting into buffer... 673
Producer: inserting into buffer... 585
Producer: inserting into buffer... 787
Consumer: removing from buffer.......... 520
Consumer: removing from buffer.......... 706
Consumer: removing from buffer.......... 673
Producer: inserting into buffer...  -1
Consumer: removing from buffer.......... 585
Consumer: removing from buffer.......... 787
Consumer: removing from buffer..........  -1
Sum: 227338
```

The provided Makefile can be used to compile and run the two versions.