Homework Set #6 – PHYS 6260
Prof. John Wise
Due Friday, February 28th, 11:59pm (Submit github URL to Canvas; all code on github)

- Your assignment should be uploaded as a **Jupyter notebook** for Problem 1 and a **Python script** for Problem 2 because `mpi4py` is largely incompatible[1] with notebooks.
- Please use the template notebook uploaded on Canvas and github as a starter.
- Comment your code through inline comments with # or markdown blocks, where the latter option is preferred.
- In the problem descriptions, "programs" are referring to single or multiple code blocks in a notebook.
- The materials that you are required to include are indicated at the end of each problem, next to the check symbol: ☑

---

1. **Parallel Monte Carlo Solver (45 points total):** Here you will write a Monte Carlo solver that simulates radioactive decay. Your code should be able to run in serial (1 core) and in parallel, using Python's thread pool in the `multiprocessing` module.

Consider the radioactive decay of ^{8}Li. This isotope of Li is unstable and has a relatively long half-life ($\tau_{1/2}$) of 838.7(3) ms. ^{8}Li undergoes a β^- decay and produces daughter isotope of ^{8}Be. The decay rate can be described using a simple differential equation:

$$\frac{dN}{dt} = -\lambda N, \tag{1}$$

where λ is the decay rate. By integrating Equation 1, we obtain:

$$N = N_0 e^{-\lambda t}, \tag{2}$$

where N_0 is the initial number of particles. The decay rate, λ, can be rewritten in terms of the half-life: $\lambda = \ln(2)/\tau_{1/2}$.

(a) (15 points) Write a function to model the radioactive decay of ^{8}Li particles using a Monte Carlo algorithm. Consider a system with $N_0 =$ 1,000 and model the decay up to a maximum time of 1 second. You first want to create an appropriately spaced time grid where we can see the decay occur. Within each timestep, you should start with the undecayed nuclei from the previous timestep and decide whether each decays within the proceeding timestep given by the probability shown in Equation 2.

(b) (15 points) Use the function that you wrote in part (a) to model the decay in serial. Repeat the decay process for all 1,000 initial particles 10 times and create a plot of the number of nuclei as a function of time for all 10 simulations. Record the time it takes for your computer to execute this code.

[1] Recent versions of IPython Parallel supports `mpi4py` running in notebooks.

(c) (15 points) Use the `multiprocessing` module, specifically `Process()`, to parallelize the code that you wrote in part (b). Change the random seed and repeat the decay process 10 times, plot the number of nuclei versus time to ensure that you get a similar result to part (b), and record the time it takes for your computer to execute the parallelized code. Hint: To parallelize your code, it may be helpful to define a "worker" function that will save the output from your function in part (a). You can then add the data to a `Manager()` to create a dictionary.

☑ **For full credit**, you should include your program with comments, plots of the decay versus time, and the time taken for your computer to solve the problem in serial and in parallel.

2. **Parallel N-body Solver (45 points total):** Here you will parallelize an N-body solver that follows the 3D dynamics of a purely gravitational system. You can find the serial code, called `HW6-Nbody0.py`, in the github repository that you clone. It has comments (saying `MODIFY HERE`) specifying where you need to add parallelism with MPI collective calls. Your code should be able to run in serial (1 core) and in parallel. First we will describe the system setup and underlying physics.

Consider a set of $N = 1000$ point masses each with a mass $m_i = 1$ kg. They are organized in a sphere of radius $r_0 = 1$ m with a uniform density. The system is rotating with tangential velocities that are not sufficient to keep it from collapsing. The force $\vec{F}_{ij}$ and thus acceleration $\vec{a}_{ij}$ on the i^{th} particle from the j^{th} particle is

$$\vec{F}_{ij} = \frac{G m_i m_j \hat{r}_{ij}}{|\vec{r}_{ij}|^2}, \tag{3}$$

where G is the gravitational constant and $\vec{r}_{ij} = \vec{r}_j - \vec{r}_i$. The total force $\vec{F}_i$ experienced by the i^{th} particle is the sum of all other particles

$$\vec{F}_i = \sum_{j,\, j\neq i}^{N} \vec{F}_{ij} = m_i \vec{a}_i \tag{4}$$

We initialize the uniform sphere with solid body rotation. The particle positions are uniformly distributed within a sphere of radius $r_0 = 1$ m. The circular (θ-component) velocity V_c in a uniform sphere of the i^{th} particle is given by

$$V_{c,i} = \sqrt{\frac{GM_{\rm enc}}{r_i}} = \sqrt{\frac{G}{r} \times M_{\rm total}\left(\frac{r_i}{r_0}\right)^3} = \sqrt{\frac{GM_{\rm total} r_i^2}{r_0^3}}, \tag{5}$$

where $M_{\rm enc}$ is the enclosed mass at radius r_i, $M_{\rm total}$ is the total mass of the sphere, and the second expression comes from the mass scaling as volume. Our system is partially rotationally supported with $v_{\theta,i} = V_{c,i}/2$. All other velocity components are zero.

The routine `calc_accel` computes the force and thus acceleration felt by each particle using Equations (3) and (4). The routine `leapfrog` evolves the system with the Leapfrog

(Verlet) Method with a timestep $\Delta t = 10$ s for a total of 5,000 s. The sphere should contract and form an ellipsoid. The code contains timers so you can monitor the performance as you develop your parallelism and test on different numbers of cores.

Pro-tip: Prototype your code first with a small N (e.g. 10) and 2 cores to test and debug before running at the final $N = 1000$ value.

(a) (10 points) The initialization routine is only run on the root (rank-0) MPI process. Use an MPI collective call to share those data with the other MPI processes.

(b) (10 points) We will parallelize the force calculation by splitting the particle position list and sending them to each core. Modify the routine `subrange` to calculate the starting and ending indices of the portion associated with each MPI process (rank).

(c) (15 points) In the `calc_accel` routine, each core only works on a subset of the particles, effectively parallelizing it. However, the new acceleration arrays are local to the MPI process on which they were calculated. Use an MPI collective call to share these results with all MPI processes and then combine these results into a single array with a shape = (N,3). Run your code with 2, 4, and 8 cores and report on the time elapsed and performance metrics reported. Plot the parallel efficiency as a function of number of cores in the Jupyter notebook with Problems 1 and 3. Also include the last particle position plot in the notebook.

(d) (10 points) We only parallelized the for-loop in the force calculation but left most of the other code untouched. From inspecting the performance increases in part (c) and the nature of the algorithm(s), is it worthwhile to parallelize the other parts of the code? Justify your answer in a few sentences.

☑ **For full credit**, include your code with additional comments. For parts (a)-(c), you should make a fully-functional code that produces the same results as the provided serial version. Your code should correctly use MPI and demonstrate a speed-up.

3. **Application question (10 points):** In a couple of paragraphs (about 250 words), describe a system that interests you, whether it is time-dependent or time-independent, and which quantities are being computed / evolved. In particular, which numerical method (covered earlier in the semester) would you use to explore the system's structure and evolution (if time-dependent). Which parallel programming model would you choose to parallelize this code? How would you divide up the work among the cores? What are some performance bottlenecks that you could expect? You do not have to provide any code.