HW6-Solutions1

March 5, 2024

1 PHYS 6260: HW6 Solutions

[10]:
```
import time
import multiprocessing as multi
import numpy as np
import matplotlib.pyplot as plt

# change the start method (only issue for OS)
multi.set_start_method('fork')
```

1.1 Problem 1: Parallel MC solver

(a) Create a function that simulates radioactive decay

[2]:
```
def decay_sim(thalf, N0, tgrid):

    """Simulate the radioactive decay of N0 nuclei.

    thalf - half-life in some units of time.
    N0 - initial number of particles.
    tgrid - sequence of evenly-spaced time to run simulation.

    """

    # calculate the lifetime from the half-life.
    tau = thalf / np.log(2)

    # deduce Nt and the time step, dt.
    Nt = len(tgrid)
    dt = tgrid[1] - tgrid[0]

    # create an empty array to save results
    N = np.empty(Nt, dtype=int)

    # define intial number of particles
    N[0] = N0

    # probability that a given nucleus will decay in time dt.
    p = dt / tau

    # loop over all time iterations
    for i in range(1, Nt):

        # at each time step, start with the undecayed nuclei from the previous.
        N[i] = N[i-1]

        # consider each nucleus in turn and decide whether it decays or not.
        for j in range(N[i-1]):

            # choose random number to decide if it decays
            r = np.random.random()

            # if probability is greater than random #, decay
            if r < p:
                # This nucleus decays.
                N[i] -= 1

    return N
```

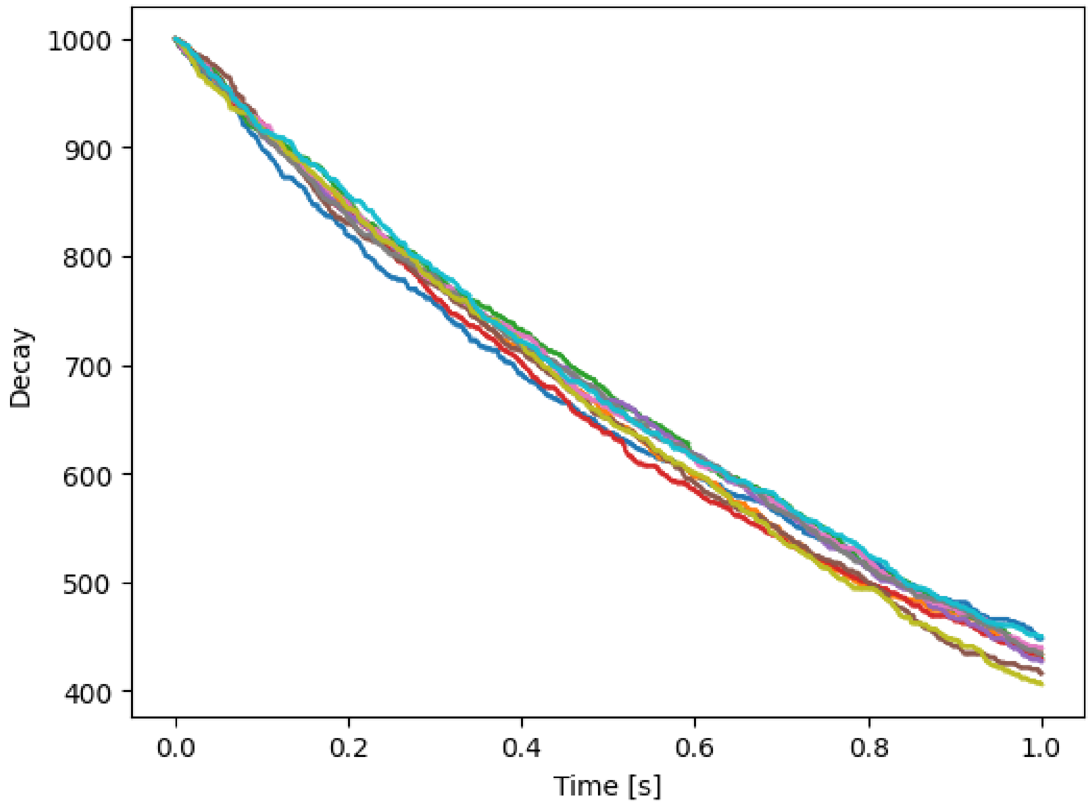
(b) solve it serially, time, and plot result

[7]:
```
# define intial number of particles
N0 = 1000

# half life of Li8 in seconds
thalf = 838.7*10**-3

# define the maximum time in seconds
tmax = 1

# define number of timesteps
Nt = 1000

# create grid of times
tgrid = np.linspace(0, tmax, Nt)

# repeat the simulation "experiment" nsims times.
nsims = 10

# create empty matrix to save results
Nsim = np.empty((nsims,Nt))

# start timer
start_time = time.perf_counter()
```

```
# loop over all "experiments"
for i in range(nsims):

    # calculate decay
    Nsim[i,:] = decay_sim(thalf, N0, tgrid)

# finish timer
finish_time = time.perf_counter()

# record time
print(f"Program finished in {finish_time-start_time} seconds")
```

```
Program finished in 2.471665923949331 seconds
```

[8]:
```
# create plot
for i in range(nsims):
    plt.plot(tgrid,Nsim[i,:], '.-',markersize=0.5)
plt.xlabel("Time [s]")
plt.ylabel("Decay")
plt.show()
```

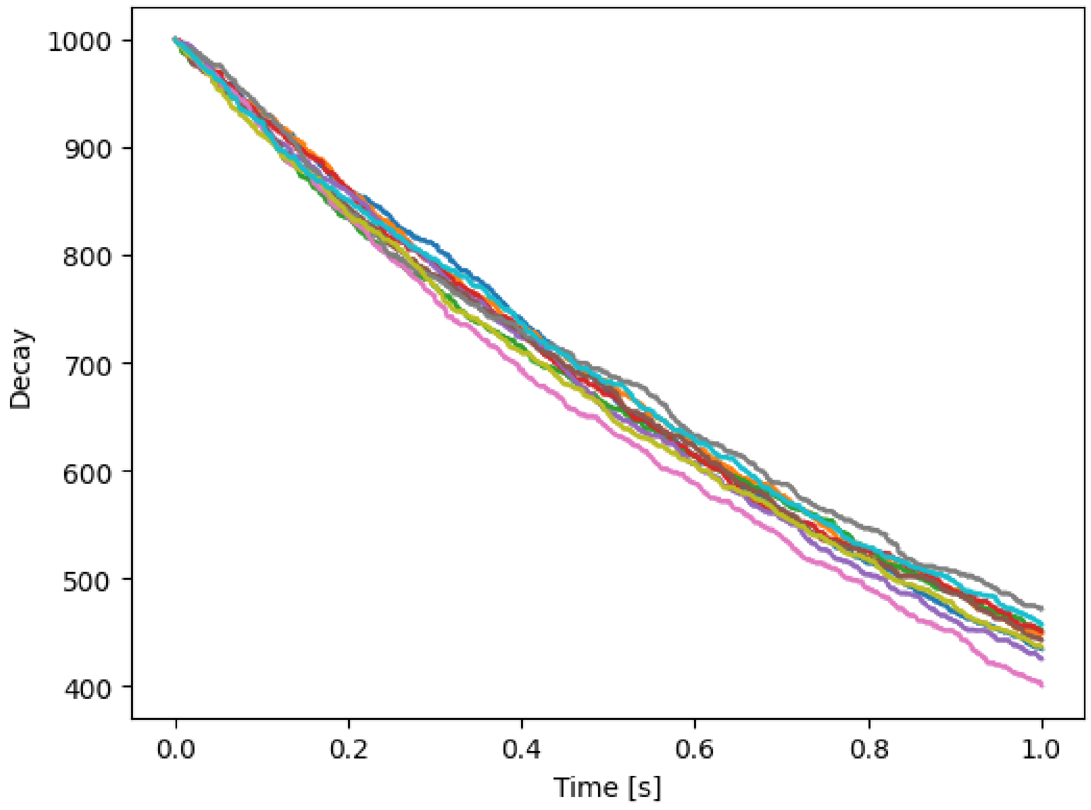


(c) use multiprocessing and compare the time difference

[9]:
```python
def worker(i, return_dict):

    """worker function"""

    return_dict[i] = decay_sim(thalf, N0, tgrid)


if __name__ == "__main__":

    # start timer
    start_time = time.perf_counter()

    # define manager and dict to save values
    manager = multi.Manager()
    return_dict = manager.dict()

    # create empty array
    processes =[]

    # creates 10 processes then starts them
    for i in range(nsims):

        # change random seed
        np.random.seed()

        # create multiprocess
        p = multi.Process(target=worker, args=(i, return_dict))
        p.start()
        processes.append(p)


    # finish time
    finish_time = time.perf_counter()

    print(f"Program finished in {finish_time-start_time} seconds")
```

```
Program finished in 0.048384873021859676 seconds
```

[10]:
```python
# create plot
for i in range(nsims):
    plt.plot(tgrid,np.array(return_dict.values())[i,:], '.-',markersize=0.5)
plt.xlabel("Time [s]")
plt.ylabel("Decay")
plt.show()
```



1.2 Problem 2: Parallel N-body solver

[1]:
```
%%file HW6-nbody-parallel.py

import time
import numpy as np
from mpi4py import MPI
import matplotlib.pyplot as plt

Mtotal = 1000  # system mass [kg]
Npart = 1000   # number of particles
Mpart = Mtotal / Npart
radius = 1     # sphere radius [m]
Vcf = 0.5      # fraction of circular velocity
G = 6.673e-11  # gravitational constant

tstop = 5000  # stop time (seconds)
dt = 10.0

np.random.seed(12345)
```

```
# I initialize a uniform distribution in a cube centered on the origin and then␣
 ↪restrict to sphere.  A cube that bounds a sphere has about twice the volume.␣
 ↪ So to be on the safe side (because randomness), I generate `3*Npart`␣
 ↪particle positions and then select the first `Npart` particles that lie␣
 ↪within the sphere.
#
#
def initialize(N):
    # Initialize a uniform sphere with N particles
    ninside = 0
    while ninside < N:
        cubepos = 2*np.random.random([3*N,3]) - 1
        r = (cubepos**2).sum(1)**0.5
        inside = (r<1)
        ninside = inside.sum()
    pos = cubepos[inside][:N,:]
    r = (pos**2).sum(1)**0.5

    # Circular radius of a cross-section (xy-plane) of the sphere at coordinate␣
 ↪z of the particle.
    # This is the particle's initial orbit for a solid body rotator.
    # We need this to calculate the x- and y-components of the initial velocity.
    rcirc = (pos[:,:2]**2).sum(1)**0.5
    vc = np.sqrt(G * Mtotal * r**2 / radius**3)

    vel = np.empty([N,3])
    vel[:,0] = -Vcf*vc * (pos[:,1] / rcirc)  # ratio is (y/r), which is␣
 ↪sin(theta)
    vel[:,1] = +Vcf*vc * (pos[:,0] / rcirc)  # ratio is (x/r), which is␣
 ↪cos(theta)
    vel[:,2] = 0.0

    return pos, vel

def subrange(comm):
    divisions = np.linspace(0, Npart, comm.size+1).astype('int')
    start = divisions[comm.rank]
    end = divisions[comm.rank+1]
    return start, end, end-start

def calc_accel(comm, pos):
    start, end, Nsub = subrange(comm)
    idx = np.arange(Npart)  # include forces from all particles
    Nsub = end - start
    # acceleration for particles being calculated on this MPI process (core)
    accel = np.empty([Nsub,3])
    # only calculate acceleration for start -> end in the particle list
    for i in range(start,end):
        rij = pos[idx != i, :] - pos[i,:]  # exclude self (idx != i)
        rij_mag = (rij**2).sum(1)**0.5
        accel[i-start,:] = (rij / rij_mag[:,np.newaxis]**1.5).sum(0)
    # multiply by G*M at the very end
    accel *= G*Mpart
    accel_list = comm.allgather(accel)
    return np.concatenate(accel_list)

def leapfrog_initialstep(comm, pos, vel, dt):
    accel = calc_accel(comm, pos)
    vhalf = vel + 0.5*dt * accel
    return vhalf

def leapfrog(comm, pos, vel, vhalf, dt):
    pos = pos + dt * vhalf
    anext = calc_accel(comm, pos)
    k = dt * anext
    # Only need v(t+dt) if we need to calculate associated quantities (e.g.␣
 ↪energy) or plot it
    vel = vhalf + 0.5*k
    vhalf = vhalf + k
    return pos, vel, vhalf

#######################
#######################
#######################

comm = MPI.COMM_WORLD
if comm.rank == 0:
    pos, vel = initialize(Npart)
else:
    pos, vel = None, None
pos, vel = comm.bcast((pos,vel))
vhalf = leapfrog_initialstep(comm, pos, vel, dt)
istep = 0
_time = 0.0
nstep = tstop/dt
time0 = time.perf_counter()
while _time < tstop:
    pos, vel, vhalf = leapfrog(comm, pos, vel, vhalf, dt)
    istep += 1
    _time += dt
    if (comm.rank == 0) and (istep % (nstep/10) == 0 or istep == 1):
        #plt.quiver(pos[:,0], pos[:,1], vel[:,0], vel[:,1], time/tstop)
        plt.clf()
        plt.scatter(pos[:,1], pos[:,2], s=10, c=np.linspace(0,1,Npart),␣
↪cmap='coolwarm')
        plt.title(f"Time = {_time:.0f} s")
        plt.xlim(-1,1)
        plt.ylim(-1,1)
        plt.savefig(f"positions-{istep:04d}.png")
        print(f"time = {_time:.0f}/{tstop:.0f} seconds")
time1 = time.perf_counter()
telapsed = time1-time0
if comm.rank == 0:
    print(f"Compute time elapsed = {telapsed:.3f} seconds ({nstep/telapsed:.1f}␣
↪steps/sec // {Npart*nstep/telapsed:.3g} particle-evolve/sec)")
```

```
Writing HW6-nbody-parallel.py
```

[3]:
```
!mpirun -n 1 python HW6-nbody-parallel.py
```

```
[phys44145:1048379] pmix_mca_base_component_repository_open: unable to open
mca_pnet_tcp: perhaps a missing symbol, or compiled for a different version of
OpenPMIx (ignored)
[phys44145:1048379] pmix_mca_base_component_repository_open: unable to open
mca_pnet_test: perhaps a missing symbol, or compiled for a different version of
OpenPMIx (ignored)
time = 10/5000 seconds
time = 500/5000 seconds
time = 1000/5000 seconds
time = 1500/5000 seconds
time = 2000/5000 seconds
time = 2500/5000 seconds
time = 3000/5000 seconds
time = 3500/5000 seconds
time = 4000/5000 seconds
time = 4500/5000 seconds
time = 5000/5000 seconds
Compute time elapsed = 47.562 seconds (10.5 steps/sec // 1.05e+04 particle-
evolve/sec)
```

[4]:
```
!mpirun -n 2 python HW6-nbody-parallel.py
```

```
[phys44145:1050083] pmix_mca_base_component_repository_open: unable to open
mca_pnet_tcp: perhaps a missing symbol, or compiled for a different version of
OpenPMIx (ignored)
[phys44145:1050083] pmix_mca_base_component_repository_open: unable to open
mca_pnet_test: perhaps a missing symbol, or compiled for a different version of
OpenPMIx (ignored)
time = 10/5000 seconds
time = 500/5000 seconds
time = 1000/5000 seconds
time = 1500/5000 seconds
time = 2000/5000 seconds
time = 2500/5000 seconds
time = 3000/5000 seconds
time = 3500/5000 seconds
time = 4000/5000 seconds
time = 4500/5000 seconds
time = 5000/5000 seconds
Compute time elapsed = 22.588 seconds (22.1 steps/sec // 2.21e+04 particle-
evolve/sec)
```

[2]:
```
!mpirun -n 4 python HW6-nbody-parallel.py
```

```
[phys44145:1048260] pmix_mca_base_component_repository_open: unable to open
mca_pnet_tcp: perhaps a missing symbol, or compiled for a different version of
OpenPMIx (ignored)
[phys44145:1048260] pmix_mca_base_component_repository_open: unable to open
mca_pnet_test: perhaps a missing symbol, or compiled for a different version of
OpenPMIx (ignored)
time = 10/5000 seconds
time = 500/5000 seconds
time = 1000/5000 seconds
time = 1500/5000 seconds
time = 2000/5000 seconds
time = 2500/5000 seconds
time = 3000/5000 seconds
time = 3500/5000 seconds
time = 4000/5000 seconds
time = 4500/5000 seconds
time = 5000/5000 seconds
Compute time elapsed = 12.855 seconds (38.9 steps/sec // 3.89e+04 particle-
evolve/sec)
```

[7]:
```
!mpirun -n 6 python HW6-nbody-parallel.py
```

```
[phys44145:1050531] pmix_mca_base_component_repository_open: unable to open
mca_pnet_tcp: perhaps a missing symbol, or compiled for a different version of
OpenPMIx (ignored)
[phys44145:1050531] pmix_mca_base_component_repository_open: unable to open
mca_pnet_test: perhaps a missing symbol, or compiled for a different version of
OpenPMIx (ignored)
time = 10/5000 seconds
time = 500/5000 seconds
time = 1000/5000 seconds
time = 1500/5000 seconds
time = 2000/5000 seconds
time = 2500/5000 seconds
time = 3000/5000 seconds
time = 3500/5000 seconds
time = 4000/5000 seconds
time = 4500/5000 seconds
time = 5000/5000 seconds
Compute time elapsed = 10.659 seconds (46.9 steps/sec // 4.69e+04 particle-
evolve/sec)
```

I'm running this on my desktop, which has 6 physical cores, so I have to provide the `--oversubscribe` option to `mpirun`.

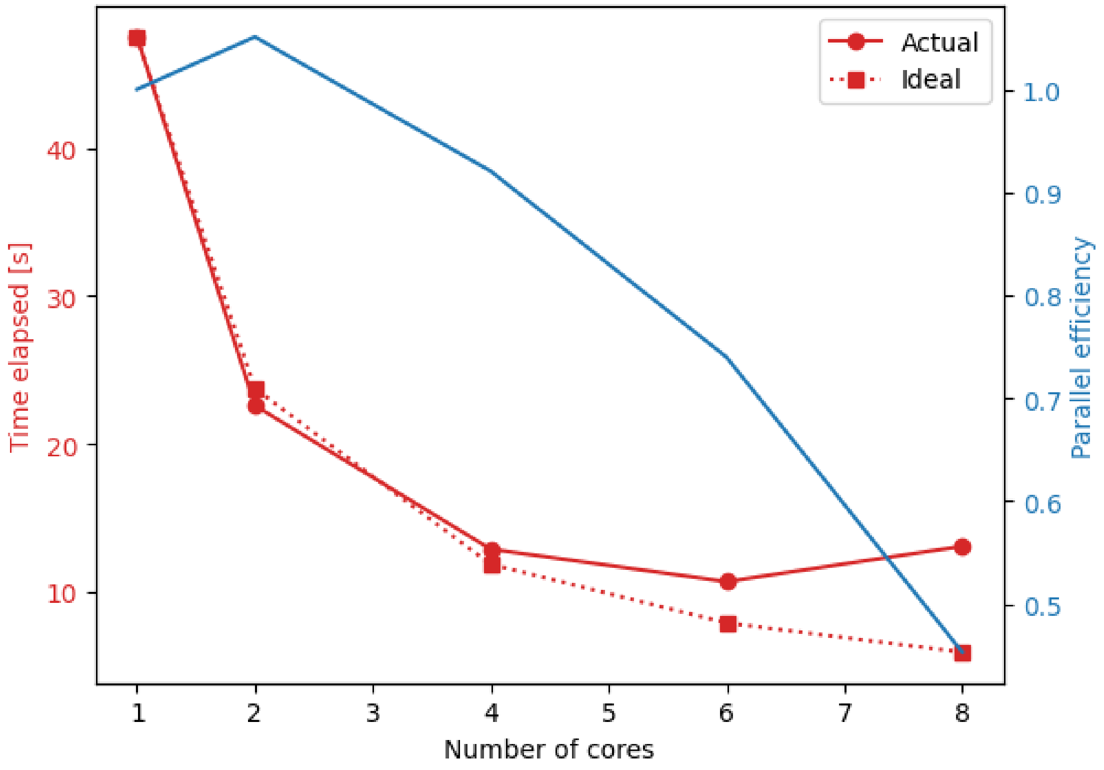
[6]:
```
!mpirun -n 8 --oversubscribe python HW6-nbody-parallel.py
```

```
[phys44145:1050252] pmix_mca_base_component_repository_open: unable to open
mca_pnet_tcp: perhaps a missing symbol, or compiled for a different version of
OpenPMIx (ignored)
[phys44145:1050252] pmix_mca_base_component_repository_open: unable to open
mca_pnet_test: perhaps a missing symbol, or compiled for a different version of
OpenPMIx (ignored)
time = 10/5000 seconds
time = 500/5000 seconds
time = 1000/5000 seconds
time = 1500/5000 seconds
time = 2000/5000 seconds
time = 2500/5000 seconds
time = 3000/5000 seconds
time = 3500/5000 seconds
time = 4000/5000 seconds
time = 4500/5000 seconds
time = 5000/5000 seconds
Compute time elapsed = 13.088 seconds (38.2 steps/sec // 3.82e+04 particle-
evolve/sec)
```

[20]:
```
cores = [1, 2, 4, 6, 8]
times = [47.5, 22.6, 12.9, 10.7, 13.1]
ideal = times[0] / np.array(cores)
peff = ideal / times
```

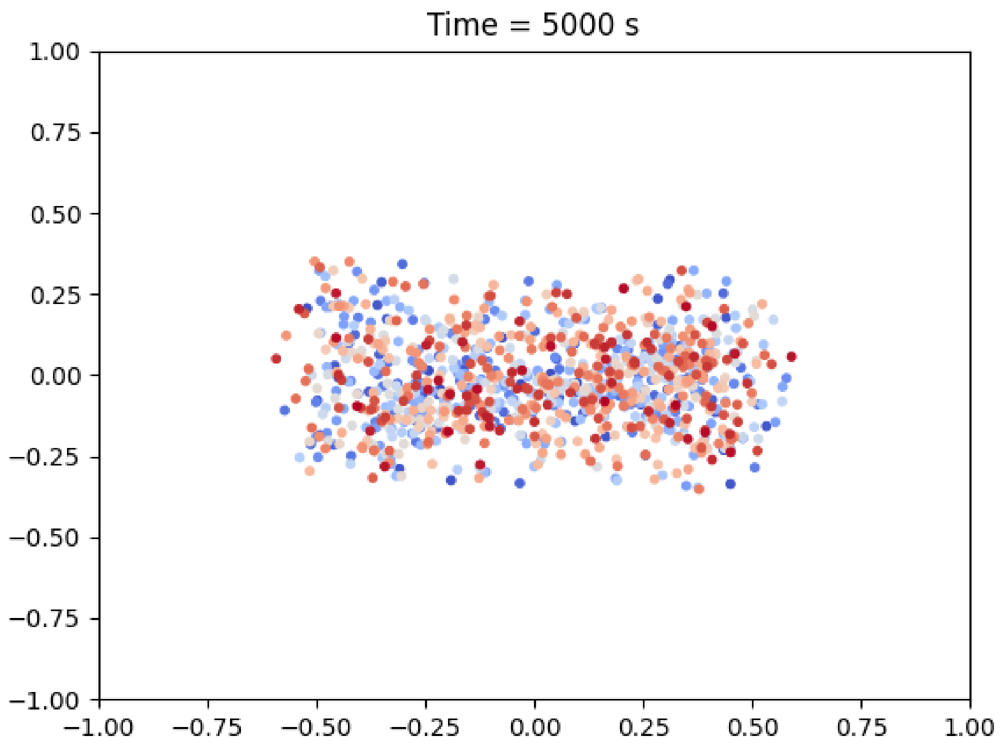
[23]:
```
fig, ax1 = plt.subplots()

color = 'tab:red'
ax1.plot(cores, times, color=color, label='Actual', marker='o')
ax1.plot(cores, ideal, color=color, ls=':', label='Ideal', marker='s')
ax1.set_xlabel('Number of cores')
ax1.set_ylabel('Time elapsed [s]', color=color)
ax1.tick_params(axis='y', labelcolor=color)
ax1.legend(loc='best')

ax2 = ax1.twinx()
```

```
color = 'tab:blue'
ax2.set_ylabel('Parallel efficiency', color=color)
ax2.plot(cores, peff, color=color)
ax2.tick_params(axis='y', labelcolor=color)

plt.show()
```



[24]:
```
from IPython.display import Image
Image(filename='positions-0500.png')
```

[24]:



1.2.1 Problem 2d

> 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.

The most expensive part of the calculation is the distance calculation between each particle pair, which is an N^2 operation. Thus all of the other operations are only a fraction of the total computational time. If we add more parallelism, we have to also consider the communication time. Given the low particle count and low core count, we would not see much speedup parallelizing the other functions, but it is possible that it would be beneficial in larger N-body calculations on more cores.

```
[ ]:
```