Cluster architectures and Calculations

Assignment 1

N-body simulation

Tzemis Evangelos

khb107@alumni.ku.dk

Vectorized Solution

First I needed to change the representation model of the galaxy data. What I did was simply make a vector with all the mazes, another one with all x's positions and so on. Then I combined them into a numpy array. The relevant code is this:

```
m = numpy.random.random(n) * numpy.random.randint(1, max_mass,n)/ (4 * numpy.pi ** 2)
x = numpy.random.randint(-x_max, x_max,n)
y = numpy.random.randint(-y_max, y_max,n)
z = numpy.random.randint(-z_max, z_max,n)
vx= numpy.ones(n)
return numpy.array([[m],[x],[y],[z],[vx],[vx],[vx]]
```

In this part the performance was influenced since we avoid to have the for-loop by inserting a third argument in the random functions.

Then I had to change the for-loop, which I did it in the following way. First I transposed the galaxy matrix, so that I will have in each row a specific atom. Then for every element of this matrix I am calling calc_force function with the galaxy matrix and one element of the transposed Galaxy matrix as input arguments. In this way we will find all forces for this specific atom. The relevant code is here.

```
for i in a:
  calc_force(i[0], galaxy, dt)
```

In this part performance increased cause we avoided the second nested loop. Trasposed matrux is calculated in nbodyVec file and is passed as an argument in the move function.

In the last part of the move function, we need to make the calculation of all new X, Y and Z which is now just a simple vector calculation. In the following we first add to x the vx value, and then I do the same for y and z.

```
galaxy[1] += galaxy[4]
```

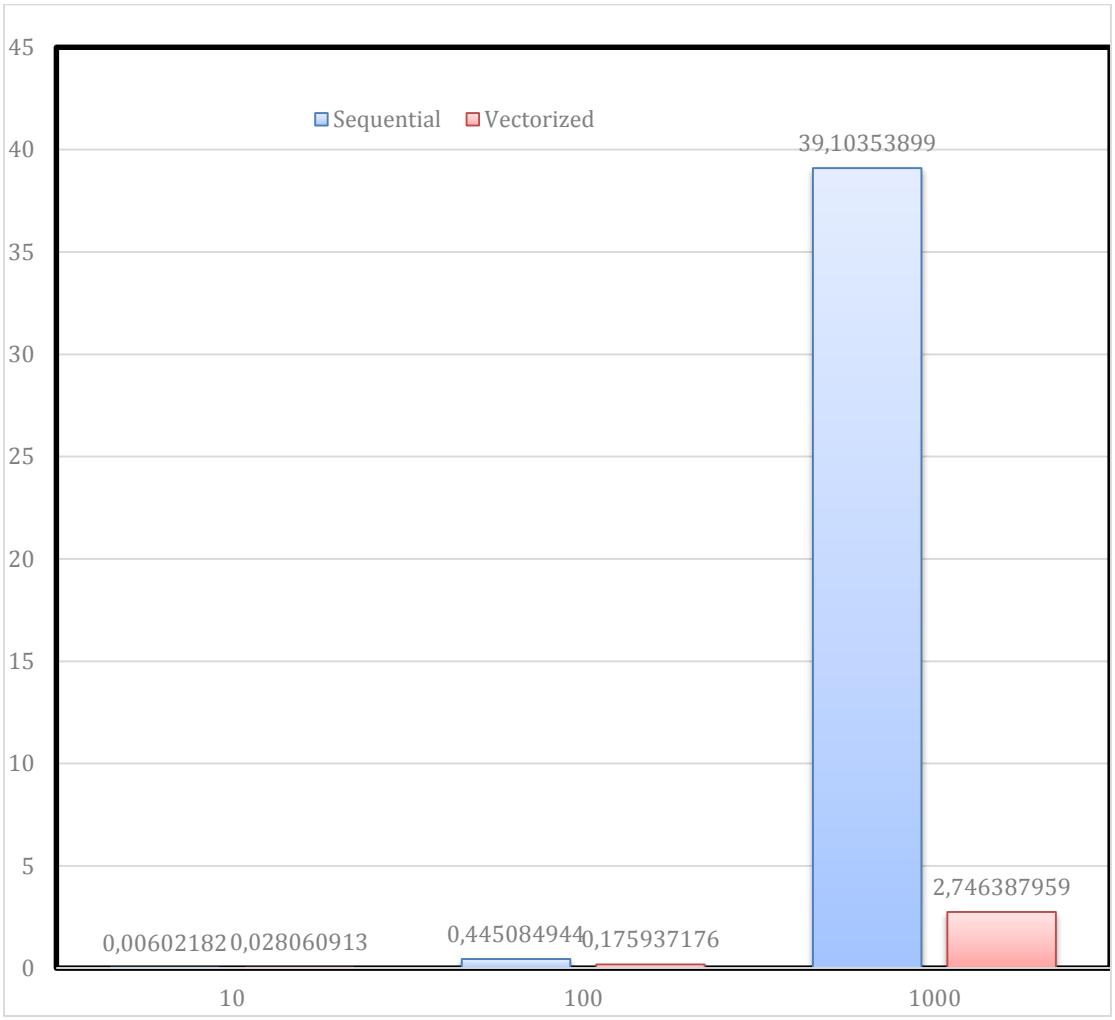
```
galaxy[2] += galaxy[5]
galaxy[3] += galaxy[6]
```

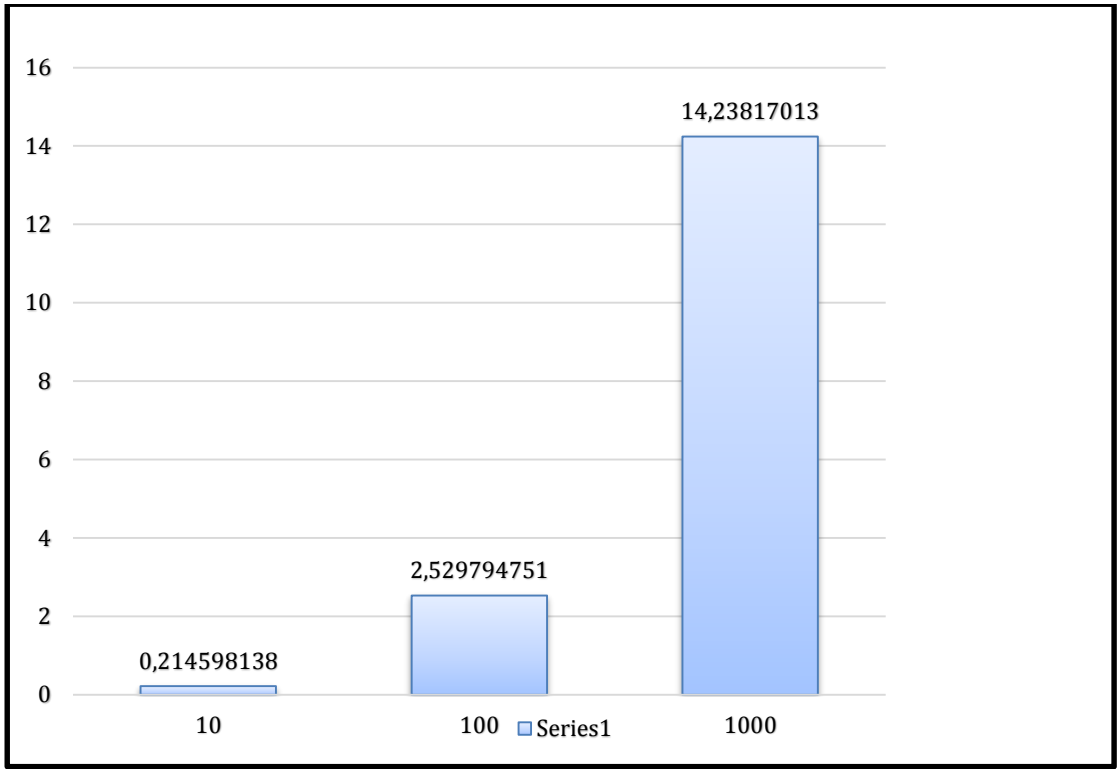
Finally I had to make changes also to the calc_force function but it only was the I have to sum all elements and make sure to avoid or maybe handle division by zero. The code is this:

```
r = ((b[1] - a[1]) ** 2 + (b[2] - a[2]) ** 2 + (b[3] - a[3]) ** 2) ** 0.5
a[4] += numpy.sum(numpy.nan_to_num(G * a[0] * b[0] / r ** 2 * ((b[1] - a[1]) / r) / a[0] * dt ))
 a[5] += numpy.sum(numpy.nan_to_num(G * a[0] * b[0] / r ** 2 * ((b[2] - a[2]) / r) / a[0] * dt ))
 a[6] += numpy.sum(numpy.nan_to_num(G * a[0] * b[0] / r ** 2 * ((b[3] - a[3]) / r) / a[0] * dt ))
```

In this part since we work with vector and we call this function only once for every atom, performance is increased. Moreover calculations are being done with a vectorized way.

Result and Performance Graphs





The correspondence tables for those graphs are this.

Bodies	*Sequential*	*Vectorized*	*SpeedUp*
10	0,00602182	0,028060913	0,214598138
100	0,445084944	0,175937176	2,529794751
1000	39,10353899	2,746387959	14,23817013