KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS

Lygiagretusis programavimas (P170B328)

Inžinerinio projekto ataskaita

Atliko:
IFF-1/4 gr. studentas
Mildaras Karvelis
2023 m. gruodžio 18 d.
Priėmė:
Lekt. Barisas Dominykas

KAUNAS 2023

TURINYS

1. Užduoties analizė ir sprendimo metodas

Uždavinys 7-10 variantams

Miestas išsidėstęs kvadrate, kurio koordinatės $(-10 \leq \boldsymbol{x} \leq 10,\ -10 \leq \boldsymbol{y} \leq 10)$. Mieste yra **n** $(\boldsymbol{n} \geq \mathbf{3})$ vieno tinklo parduotuvių, kurių koordinatės yra žinomos (*Koordinatės gali būti generuojamos atsitiktinai, negali būti kelios parduotuvės toje pačioje vietoje*). Planuojama pastatyti dar **m** $(\boldsymbol{m} \geq \mathbf{3})$ šio tinklo parduotuvių. Parduotuvės pastatymo kaina (vietos netinkamumas) vertinama pagal atstumus iki kitų parduotuvių ir poziciją (koordinates). Reikia parinkti naujų parduotuvių vietas (koordinates) taip, kad parduotuvių pastatymo kainų suma būtų kuo mažesnė (naujos parduotuvės gali būti statomos ir už miesto ribos).

Atstumo tarp dviejų parduotuvių, kurių koordinatės $(\boldsymbol{x_1}, \boldsymbol{y_1})$ ir $(\boldsymbol{x_2}, \boldsymbol{y_2})$, kaina apskaičiuojama pagal formulę:

$$C(x_1, y_1, x_2, y_2) = \exp(-0.3 \cdot ((x_1 - x_2)^2 + (y_1 - y_2)^2))$$

Parduotuvės, kurios koordinatės $(\boldsymbol{x_1}, \boldsymbol{y_1})$, vietos kaina apskaičiuojama pagal formulę:

$$C^P(x_1, y_1) = \frac{x_1^4 + y_1^4}{1000} + \frac{\sin(x_1) + \cos(y_1)}{5} + 0.4$$

Sprendžiama problema: Ši programa sprendžia optimizavimo problemą, kurioje ieškoma optimalios X ir Y koordinačių reikšmės, taip, kad sumažėtų tam tikra funkcija, sudaryta iš skirtingų mateminių išraiškų. Optimizavimo procesas vyksta iteratyviai, keičiant koordinates ir tikslingai mažinant funkcijos reikšmę.

Pasirinktas sprendimo būdas: Programoje naudojama lygiagretumo priemonė – **multiprocessing** modulis su **Pool** objektu, kad būtų galima efektyviau atlikti skaičiavimus. Lygiagretumo privalumas pasireiškia tokiu būdu, kad dalis skaičiavimų gali būti vykdoma paraleliai keliuose procesuose, kai kiekvienas procesas atlieka dalinį darbą.

Duomenų lygiagretumo priemonės: Programoje yra naudojamas lygiagretus skaičiavimų vykdymas, kuris yra pagrįstas padalijimu į kelis procesus. **Pool** objektas naudojamas kiekvieno proceso įgyvendinimui, o **starmap** funkcija padeda paskirstyti skaičiavimus tarp skirtingų procesų. Taip galima pagreitinti skaičiavimus, ypač kai problema yra apie procesorių skaičiavimų ribas.

2. Testavimas ir programos vykdymo instrukcija

2.1. Target(args) - Tikslinė funkcija:

Priima argumentą args, kuriame yra X ir Y koordinatės bei kita informacija.
Skaičiuoja funkcijos reikšmę, kurią programa stengiasi minimizuoti.

2.2. NumericalGradient(args) - Skaitinė gradientų funkcija:

Priima argumentą args, kuriame yra X ir Y koordinatės bei kita informacija.
Skaičiuoja funkcijos gradientus pagal X ir Y.

2.3. compute_partial_gradient(i, args) - Skaitinė dalinė gradiento funkcija:

Priima indeksą i ir argumentą args, kuriame yra X ir Y koordinatės bei kita informacija.
Skaičiuoja dalinį gradientą tik pagal vieną koordinatę (X arba Y).

2.4. optimize(args, processes) - Optimizavimo funkcija:

Priima argumentą args, kuriame yra pradinės sąlygos ir optimizavimo parametrai (X, Y, dist, step, h, fff).

Naudoja gradientinio nusileidimo metodą ieškant optimalių X ir Y.
Lygiagrečiai skaičiuoja dalinius gradientus kiekvienam taškui naudodama Pool objektą.

2.5. experiment(processes) - Eksperimento funkcija:

Priima argumentą processes, kuris nurodo, kiek procesų naudoti eksperimentui su optimizavimu.
Įvykdo optimizavimą su nurodytu procesų skaičiumi ir grąžina vykdymo laiką.

2.6. Pagrindinė skripto dalis:

Generuoja pradines koordinates X ir Y.
Pateikia pradines koordinates grafiškai.
Vykdymo metu keičia X ir Y, siekiant minimizuoti funkciją.
Parodo galutinį rezultatą grafiškai.

2.7. Programos paleidimas

Programai paleisti yra naudojamas Jupiter ipynb arba Jupyter lab, kurį suteikia universitetas, adresu: https://ai-notebook.ktu.edu , kad paleisti koda, tiesgio užtenka perkelti kodo tekstą į python.notebook aplinką ir jį paleisti.

2.8. Programos kodas

```
import numpy as np
import matplotlib.pyplot as plt
import random
from multiprocessing import Pool
import time

def Target(args):
    X, Y, dist = args
    n = len(X)
    full_distance = 0

    for i in range(n):
        for j in range(i + 1, n):
            computation_distance = np.exp(-0.3 * ((X[i] - X[j]) ** 2 + (Y[i] - Y[j]) ** 2))
            compuation_price = ((X[i] ** 4 + Y[i] ** 4) / 1000) + ((np.sin(X[i]) + np.cos(X[i])) / 5) + 0.4
            full_distance += computation_distance + compuation_price

    full_distance = full_distance + np.average(X) ** 2 + np.average(Y) ** 2
    return full_distance

def NumericalGradient(args):
    X, Y, dist, h = args
    n = len(X)
    xx = np.array(X)
    yy = np.array(Y)
    Gx = np.zeros(n)
    Gy = np.zeros(n)

    for i in range(n):
        xx[i] = xx[i] + h
        Gx[i] = (Target((xx, Y, dist)) - Target((X, Y, dist))) / h
        xx[i] = X[i]

        yy[i] = yy[i] + h
        Gy[i] = (Target((X, yy, dist)) - Target((X, Y, dist))) / h
        yy[i] = Y[i]
```

```
    aa = np.linalg.norm(np.hstack((Gx, Gy)))
    Gx0 = Gx / aa
    Gy0 = Gy / aa

    return Gx0, Gy0

def compute_partial_gradient(i, args):
    X, Y, dist, h = args
    xx = np.array(X)
    yy = np.array(Y)

    xx[i] = xx[i] + h
    Gx_i = (Target((xx, Y, dist)) - Target((X, Y, dist))) / h
    xx[i] = X[i]

    yy[i] = yy[i] + h
    Gy_i = (Target((X, yy, dist)) - Target((X, Y, dist))) / h
    yy[i] = Y[i]

    return Gx_i, Gy_i

def optimize(args, processes=10):
    X, Y, dist, step, h, fff = args
    pool = Pool(processes=processes)

    for iii in range(itmax):
        results = pool.starmap(compute_partial_gradient, [(i, (X, Y, dist, h)) for i in range(n)])
        Gx0, Gy0 = zip(*results)
        Gx0 = np.sum(Gx0, axis=0)
        Gy0 = np.sum(Gy0, axis=0)
        X = X - step * Gx0
        Y = Y - step * Gy0
        fff1 = Target((X, Y, dist))

        if fff1 > fff:
            X = X + step * Gx0
            Y = Y + step * Gy0
            step = step / 3
            print('step=', step)
        else:
            fff = fff1

        if step < 1e-16:
            print('optimizavimas baigtas fff=', fff, "iteracijų skaičius=", iii)
            break

    pool.close()
    pool.join()
    return X, Y, fff

def experiment(processes):
    start_time = time.time()

    # Call your optimization function with the specified number of processes
    optimized_result = optimize((X, Y, dist, step, h, fff), processes=processes)
```

```
    end_time = time.time()
    elapsed_time = end_time - start_time

    return elapsed_time

if __name__ == '__main__':
    start_time = time.time()  # Initialize the start time

    plt.ion()  # Enable interactive mode

    n = 5
    X = np.zeros(n)
    Y = np.zeros(n)

    for i in range(n): X[i] = random.uniform(-10., 10.)
    for i in range(n): Y[i] = random.uniform(-10., 10.)

    plt.subplot(2, 1, 1)  # Create subplot for optimization result
    plt.plot(X, Y, 'rp')
    plt.grid()
    plt.axis('equal')

    m = 5
    X = np.zeros(m)
    Y = np.zeros(m)

    for i in range(m): X[i] = random.uniform(-20., 20.)
    for i in range(m): Y[i] = random.uniform(-20., 20.)

    itmax = 1000
    step = 0.2
    h = 0.00001
    dist = 3
    fff = Target((X, Y, dist))

    # Run the optimization
    optimized_result = optimize((X, Y, dist, step, h, fff), processes=1)

    plt.subplot(2, 1, 2)  # Create subplot for performance vs processes
    # Define the number of processes to experiment with
    process_counts = [1, 2, 4, 8, 10, 12]

    # Store execution times for each process count
    execution_times = []

    for processes in process_counts:
        elapsed_time = experiment(processes)
        execution_times.append(elapsed_time)

    # Plot the results
    plt.subplot(2, 1, 2)
    plt.plot(process_counts, execution_times, marker='o')
    plt.xlabel('Number of Processes')
    plt.ylabel('Execution Time (seconds)')
    plt.title('Performance vs Number of Processes, Iterations = {itmax}')

    plt.tight_layout()  # Adjust layout to prevent overlap
```

```
plt.show()

plt.ioff()  # Turn off interactive mode

end_time = time.time()
total_elapsed_time = end_time - start_time
print(f"Total execution time: {total_elapsed_time} seconds")
```

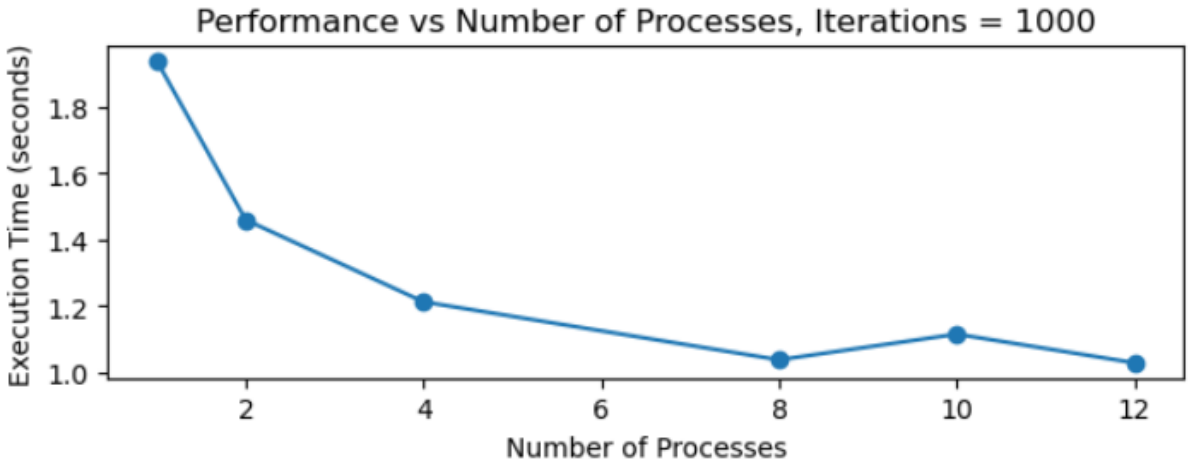
3. Vykdymo laiko kitimo tyrimas

3.1. a

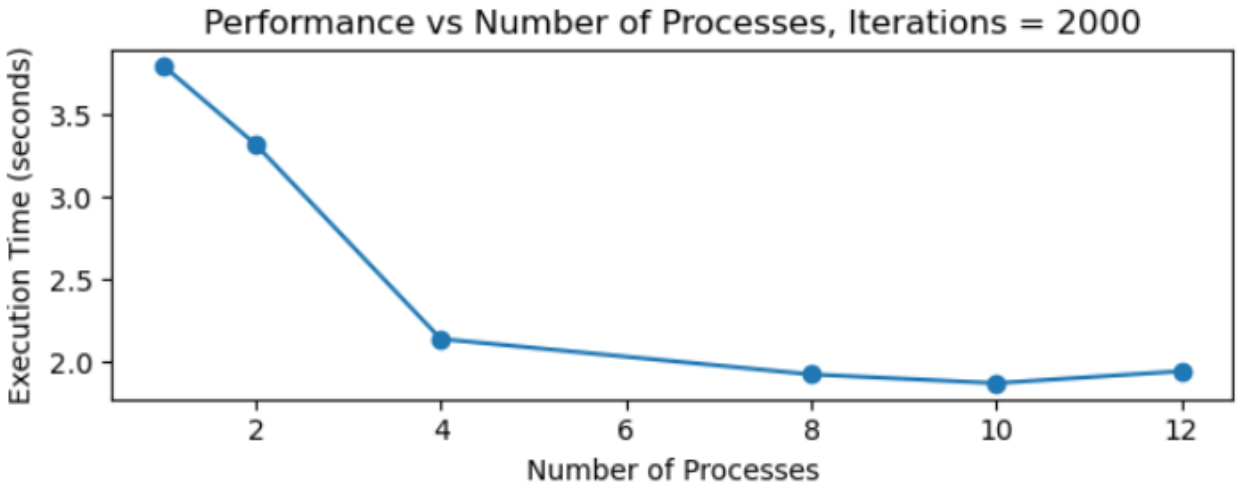
Buvo paruošti 8 pagrindiniai užduoties variantai su kintačiu itmax kintamuoju.

itmax
1000
2000
5000
10000
20000
50000
100000
200000

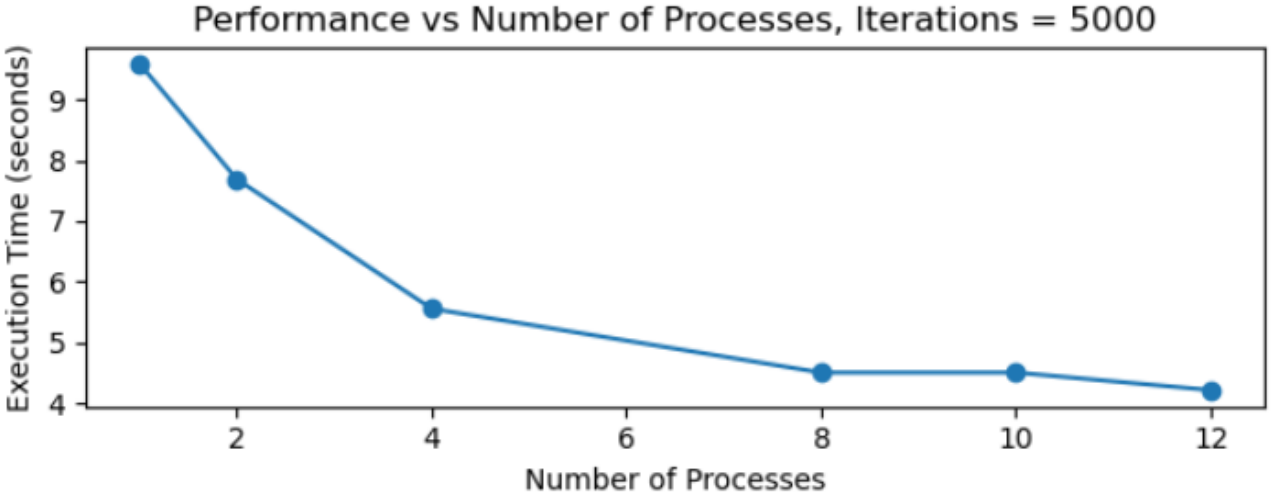
3.2. b



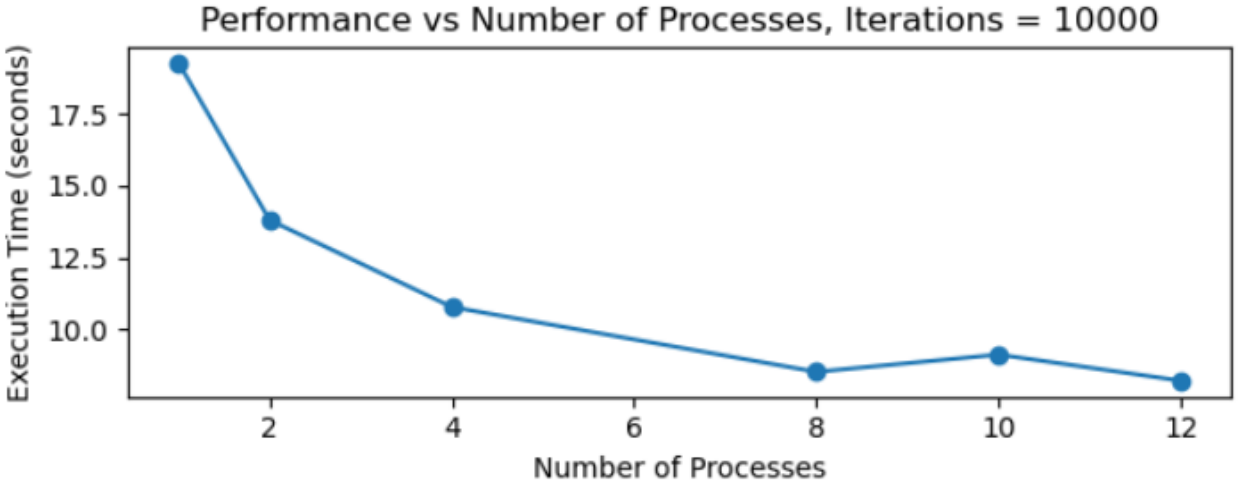
Optimalus gijų skaičius = 8, darbo laikas = 1.1000541 sek. Mano manymu iteracijų skaičius yra per mažas todėl, bet koks gijų skaičius didesnis už 8 turėtu gražinti panašius rezultatus.



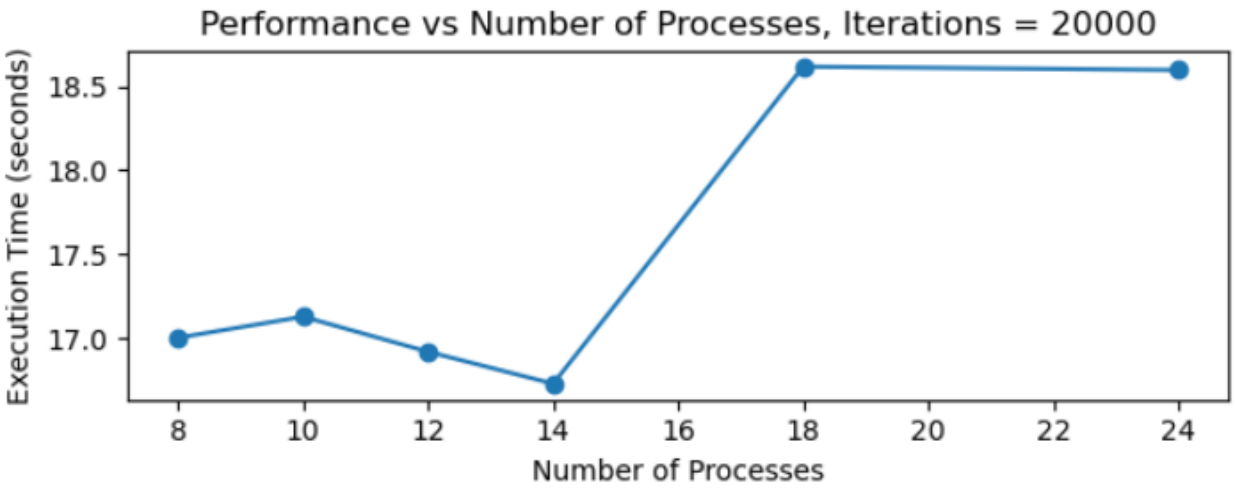
Optimalus gijų skaičius = 10, darbo laikas = 1.8693246841430664 sek. Mano manymu iteracijų skaičius yra per mažas todėl, bet koks gijų skaičius didesnis už 10 turėtu gražinti panašius rezultatus.



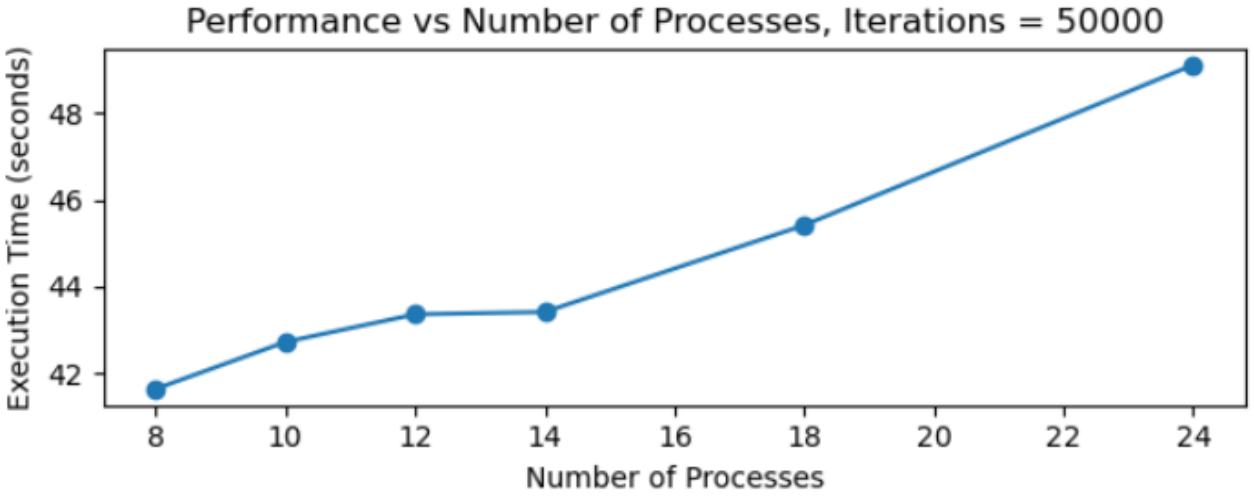
Optimalus gijų skaičius = 12, darbo laikas = 4.213501930236816 sek. Mano manymu iteracijų skaičius yra per mažas todėl, bet koks gijų skaičius didesnis už 12 turėtu gražinti panašius rezultatus.



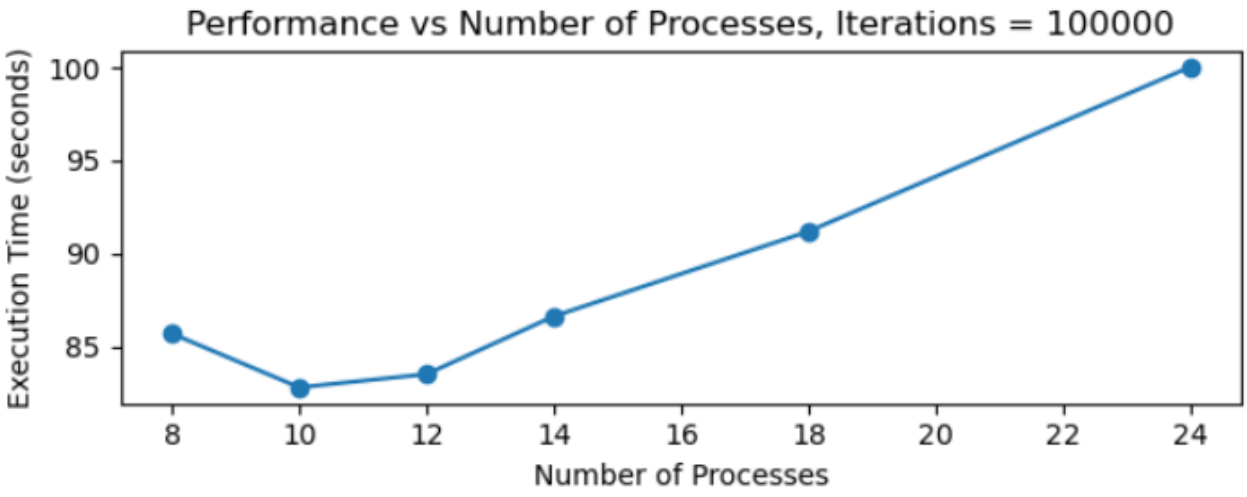
Optimalus gijų skaičius = 12, darbo laikas = 8.222364902496338 sek. Mano manymu iteracijų skaičius yra per mažas todėl, bet koks gijų skaičius didesnis už 12 turėtu gražinti panašius rezultatus.



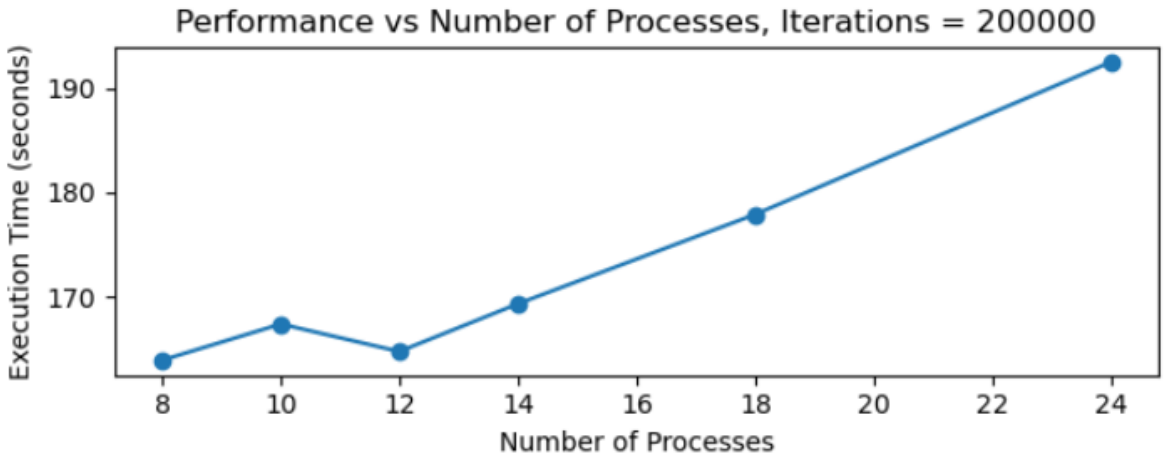
Optimalus gijų skaičius = 14, darbo laikas = 16.72975754737854 sek. Mano manymu iteracijų skaičius yra pakankamai didelis, kad būtų galima pamatyti , kaip tiksliai gijų skaičius priklauso nuo iteracijų skaičiaus, 18-24 gijų skaičiuose matome, jog programa vykdo daug ilgiau negu su mažesniu gijų skaičiumi, mano manymu, taip vyksta, nes vienos gijos turi laukti, kol kitos baigs darbą, to pasekoje laikas didėja.



Optimalus gijų skaičius = 8, darbo laikas = 41.64823603630066 sek. Mano manymu iteracijų skaičius yra pakankamai didelis, kad būtų galima pamatyti , kaip tiksliai gijų skaičius priklauso nuo iteracijų skaičiaus, 14-24 gijų skaičiuose matome, jog programa vykdo daug ilgiau negu su mažesniu gijų skaičiumi, mano manymu, taip vyksta, nes vienos gijos turi laukti, kol kitos baigs darbą, to pasekoje laikas didėja.



Optimalus gijų skaičius = 10, darbo laikas = 82.82362794876099 sek. Mano manymu iteracijų skaičius yra pakankamai didelis, kad būtų galima pamatyti , kaip tiksliai gijų skaičius priklauso nuo iteracijų skaičiaus, 14-24 gijų skaičiuose matome, jog programa vykdo daug ilgiau negu su mažesniu gijų skaičiumi, mano manymu, taip vyksta, nes vienos gijos turi laukti, kol kitos baigs darbą, to pasekoje laikas didėja.



Optimalus gijų skaičius = 8, darbo laikas = 163.9320845603943 sek. Mano manymu iteracijų skaičius yra pakankamai didelis, kad būtų galima pamatyti , kaip tiksliai gijų skaičius priklauso nuo iteracijų skaičiaus, 14-24 gijų skaičiuose matome, jog programa vykdo daug ilgiau negu su mažesniu gijų skaičiumi, mano manymu, taip vyksta, nes vienos gijos turi laukti, kol kitos baigs darbą, to pasekoje laikas didėja.

4. Išvados

Apibendrindamas gautus rezultatus, galiu teigti, jog didesnis gijų kiekis nevisada reiškia geresnius rezultatus. Taip pat, programos darbo laikas eksponentiškai kyla. Dirbdamas ties šiuo projektu išmokau dirbti su gijomis python aplinkoje ir maždaug supratau kaip veikia multiprocesing.pool objektas.

5. Literatūra

Programai paleisti ir rašyti naudotas: https://ai-notebook.ktu.edu
https://docs.python.org/3/library/multiprocessing.html
https://stackoverflow.com/questions/31711378/python-multiprocessing-how-to-know-to-use-pool-or-process
https://stackoverflow.com/questions/5442910/how-to-use-multiprocessing-pool-map-with-multiple-arguments