KAUNO TECHNOLOGIJOS UNIVERSITETAS
Informatikos fakultetas

P170B328 Lygiagretusis programavimas

Individualus Projektas

Data: 2020-12-07

Dėstytojas:
lekt. Vasiljevas Mindaugas
Studentas:
Mykolas Paulauskas
Grupė: IFF-8/13

KAUNAS, 2020

Turinys

1. Užduoties analizė ir sprendimo metodas

Tikslas: panaudoti lygiagretų programavimą sprendžiant uždavinį ir ištirti kaip keičiasi programos vykdymo laikas su skirtingais procesų kiekiais.

Užduotis: Plokštumoje ($-10 \leq x \leq 10, -10 \leq y \leq 10$) išsidėstę n taškų ($3 \leq n$), vienas jų fiksuotas koordinačių pradžioje (0; 0). Kiekvienas taškas su visais kitais yra sujungtas tiesiomis linijomis (stygomis). Raskite tokias taškų koordinates, kad atstumas tarp taškų būtų kuo artimesnis vidutiniam atstumui, o stygų ilgių suma kuo geriau atitiktų nurodytą reikšmę S ($10 \leq S$).

Šio uždavinio sprendimas:

1. Pradinių taškų susigeneravimas
2. Gradiento paskaičiavimas kiekvienam taškui kiekvienoje iteracijoje
3. Stygų ilgių sumos paskaičiavimas naudojant pradinius taškus
4. Stygų ilgių sumos skaičiavimas naudojant pakeistus taškus pagal gradientą
5. Jeigu suma pagerėja pradiniams taškams priskiriamia pakeistus taškus
6. Programa vykdoma kol pasiekiamas norimas tikslumas arba pasiekiamas nurodytas iteracijų skaičius

Lygiagretinimas buvo panaudotas gradientų skaičiavimui. Kiekvienoje vykdymo iteracijoje reikia apskaičiuoti gradientą kiekvienam taškui, vykdant nuosekliai tai užtrunka ilgai ir šią vietą galime išlygiagretinti.

Šio uždavinio lygiagretinimui naudojamas Python bei procesų telkinys (pool)

2. Programos aprašymas

- **optimize_points(points)** – pagrindinis metodas, kuris kviečia visus kitus metodus. Turi ciklą kur kiekvienoje iteracijoje paskaičuojamas taškų gradientas. Taškai perstumiami pagal gradientą, lyginamas stygų ilgių vidurkių palyginimas. Vyksta tol kol pasieka nustatytą iteracijų limitą arba pasiekia nurodyta tikslumą
- **move_by_gradient(gradient_vector, points)** – perstumia taškus pagal gradiento vektorių
- **points_gradient_vector(points, current_sum)** – paskaičiuoja kiekvieno taško gradientą iš kurių susidaro vektorius. Šitame metode padarytas lygiagretinimas
- **point_gradient(i, points, current_sum)** – paskaičiuoja taško pagl duotą indeksą iš taškų masyvo
- **connect_each_point(points)** – sudaro masyvą, kuriame sujungiami visi taškai. Naudojas rezultato grafike norint atvaizduoti stygas jungiančias taškus
- **generate_points(n)** - sugeneruoja taškus priklausomai nuo globaus kintamojo n
- **distance(point1, point2)** – paskaičiuoja atstumą tarp dviejų taškų
- **distance_sum(points)** - paskaičiuoja visų stygų ilgių vidurkį

-

3. Programos pagrindinių dalių tekstai su komentarais.

```
def optimize_points(points):
    """
    Optimizes points location so that strings between points would be
near similar distance
    :param points: points array
    :return: optimized points array, current sum of strings distances,
iteration count
    """
    global alpha
    points = copy.deepcopy(points)

    max_iterations = 1000
    current_sum = distance_sum(points)
    primary_sum = current_sum.copy()
    counter = 0
    not_improving_counter = 0
    while counter < max_iterations and alpha >= eps and not_improv-
ing_counter <= 10:
        counter += 1

        points_gradient = points_gradient_vector(points, current_sum)
        gradient_norm = [item / np.linalg.norm(points_gradient) for
item in points_gradient]
        moved_points = move_by_gradient(gradient_norm, points)

        next_sum = distance_sum(moved_points)
        if next_sum < current_sum:
            points = moved_points
            current_sum = next_sum
            not_improving_counter = 0
        else:
            alpha /= 2
            not_improving_counter += 1

    return points, current_sum, counter + 1, primary_sum
```

Šis metodas bando optimizuoti taškus tol kol pasiekia iteracijų limitą(šiuo atveju 1000) arba taško pakeitimo žingsnis(alpha) tampa mažesniu nei nurodytas tikslumas(eps=1e-4). Kiekvienoje iteracijoje paskaičiuoja taškų gradientą, perstumia taškus pagal gradientą, paskaičiuoja stygų ilgių vidurkį su pakeistais taškais. Galiausiai palygina tą vidurkį su prieš tai buvusių taškų vidurkiu. Jeigu naujas vidurkis mažesnis tai jis tampa dabartiniu, pakeisti taškai tampa pradiniais taškais. Jeigu vidurkis didesnis tada mažinamas žingsnius(alpha) ir skaičiuojama iš naujo.

```
def points_gradient_vector(points, current_sum):
    """
    Calculates gradient for each point's x and y
    :param points: points array
    :param current_sum: current sum of strings distances
    :return:
    """
    gradients = [0.0, 0.0]
    # Gives work to workers pool
    args = partial(point_gradient, points=points, current_sum=cur-
rent_sum)
    result = pool.map(args, range(1, len(points)))

    # Converts 2d array to 1d
    [gradients.extend(point) for point in result]

    return gradients
```

Šis metodas paskaičiuoja gradientą kiekvienam taškui. Šioje vietoja panaudotas lygiagretumas. Naudojamas iš anksto sukurtas procesų telkinys. Su metodu pool.map darbas paskirstomas kiekvienam darbuotojui tolygiai. Rezultatai iš kiekvieno darbuotojai sustatomi į result masyvą eilės tvarka. Galiausiai tie rezultatai pridedami į gradients masyvą ir grąžinami.

```
def point_gradient(i, points, current_sum):
    """
    Calculates point given by i and j gradient
    :param i: point index in array
    :param j: x or y of the point
    :param points: point array
    :param current_sum: current value between each strings
    :return: gradient for given point
    """
    # Point's x
    changed_points_x = copy.deepcopy(points)
    changed_points_x[i][0] += h

    # Point's y
    changed_points_y = copy.deepcopy(points)
    changed_points_y[i][1] += h

    return [(distance_sum(changed_points_x) - current_sum) / h, (dis-
tance_sum(changed_points_y) - current_sum) / h]
```

Šį metodą vykdo darbuotojai. Grąžina paskaičiuota gradiento reiškmę taške.

```
pool = Pool(processes=processes)
```

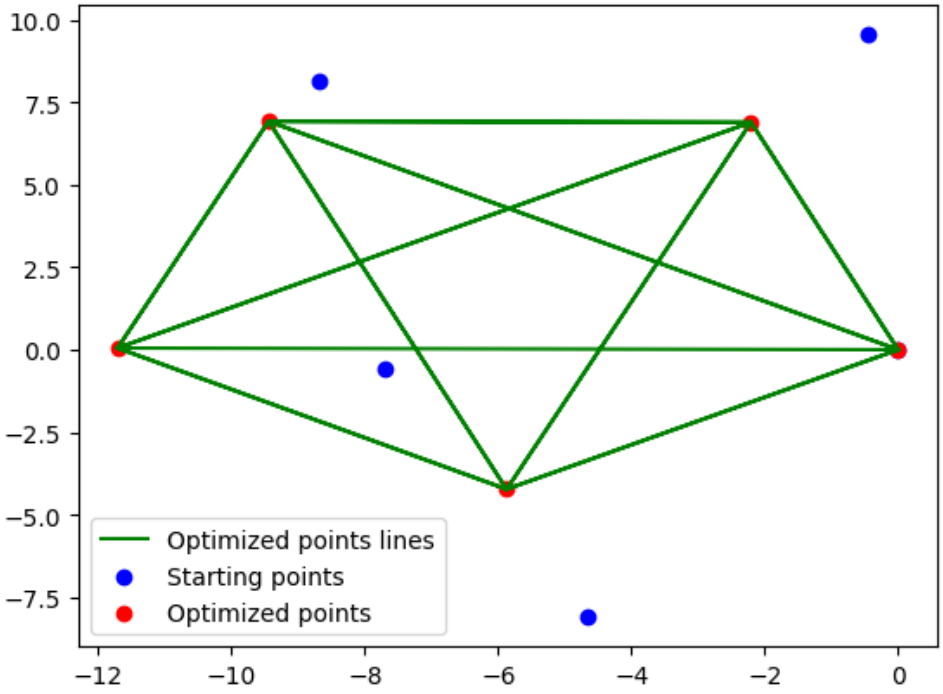
Taip pat, main metode sukuriame procesų telkinį su aukščiau parodytu kodu (skliausteliuose nurodome kiek darbuotojų sukurti)

4. Testavimas ir programos instaliavimo bei vykdymo instrukcija

Testavimas bus atliekamas keičiant globalų taškų skaičiaus kintamąjį n. Galimas taškų optimizavimas labai priklauso nuo pasirinkto taškų generavimo seed.

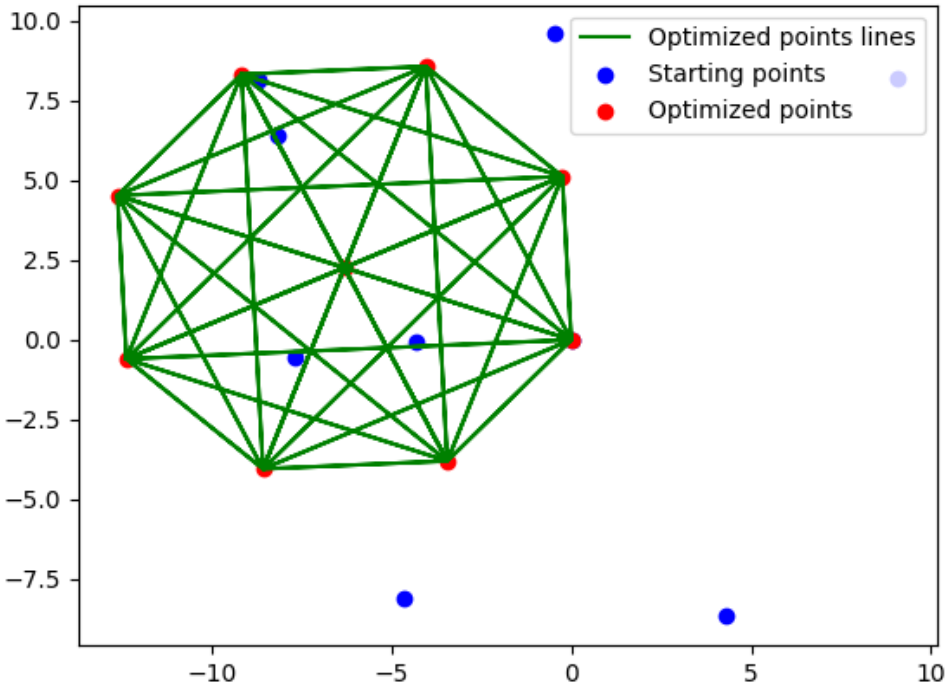
Skaičiavimams naudojamas 1 procesas, duomenų generavimo seed 1000101

n = **4**
Pradinė kaina: 135.44897036135106
Optimizuota kaina: 52.78640451136721
Iteracijų skaičius: 95
Vykdymo laikas: 0.5510332584381104s



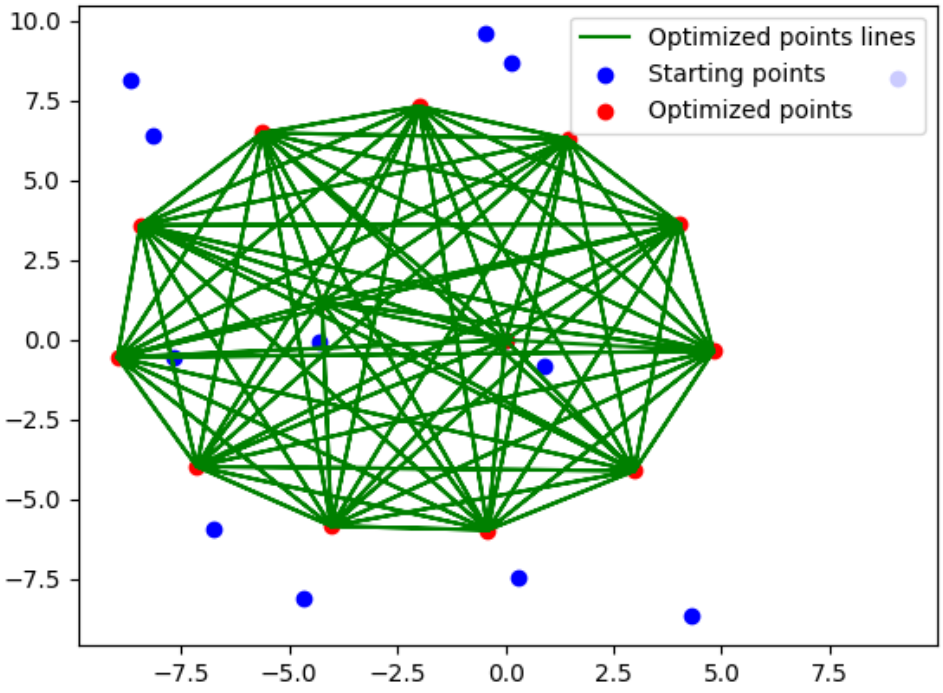
Paveikslėlis 1. Rezultatas, kai n yra 4

n = **8**
Pradinė kaina: 1063.8328743743955
Optimizuota kaina: 370.7714352639593
Iteracijų skaičius: 387
Vykdymo laikas: 1.8078646659851074s


Optimized points lines
Starting points
Optimized points
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10
−5
0
5
10

Paveikslėlis 2. Rezultatas, kai n yra 8

n = **12**
Pradinė kaina: 2050.187256266456
Optimizuota kaina: 972.5107487868653
Iteracijų skaičius: 420
Vykdymo laikas: 3.7837564945220947s



Paveikslėlis 3. Rezultatas, kai n yra 12

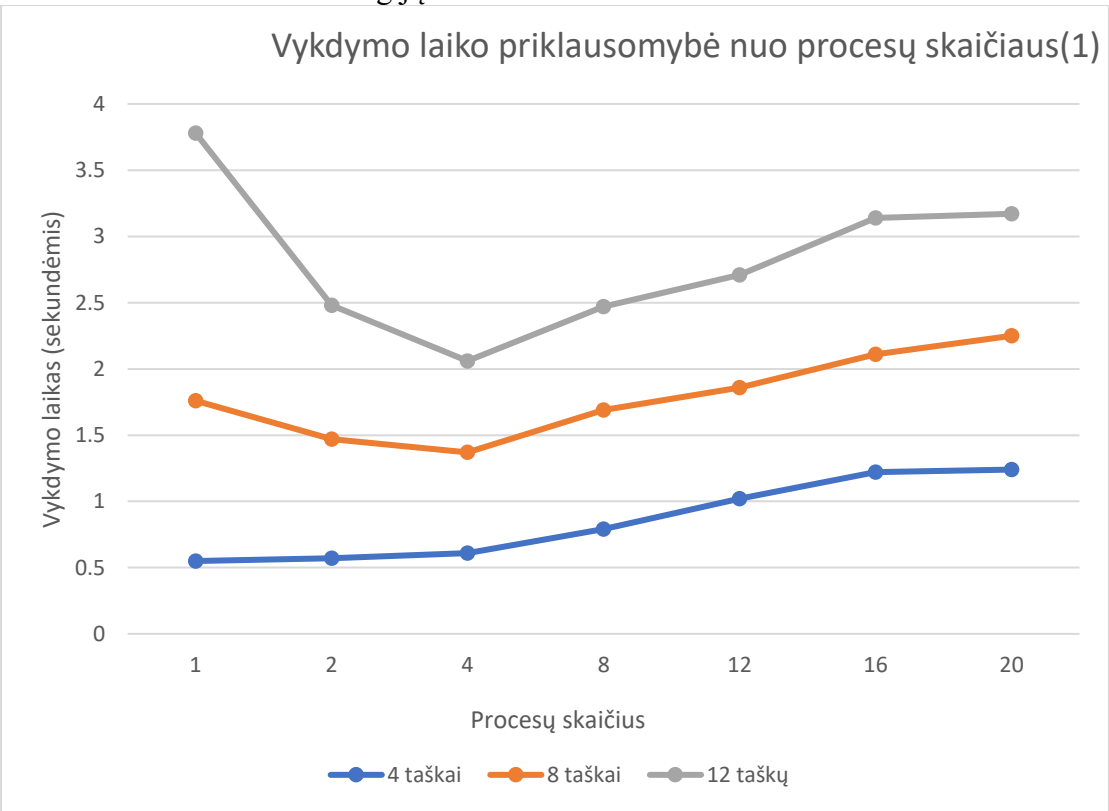
Pagal pateiktus rezultatus ir nubaižytus grafikus galima matyti, kad programa veikia

Programos paleidimas

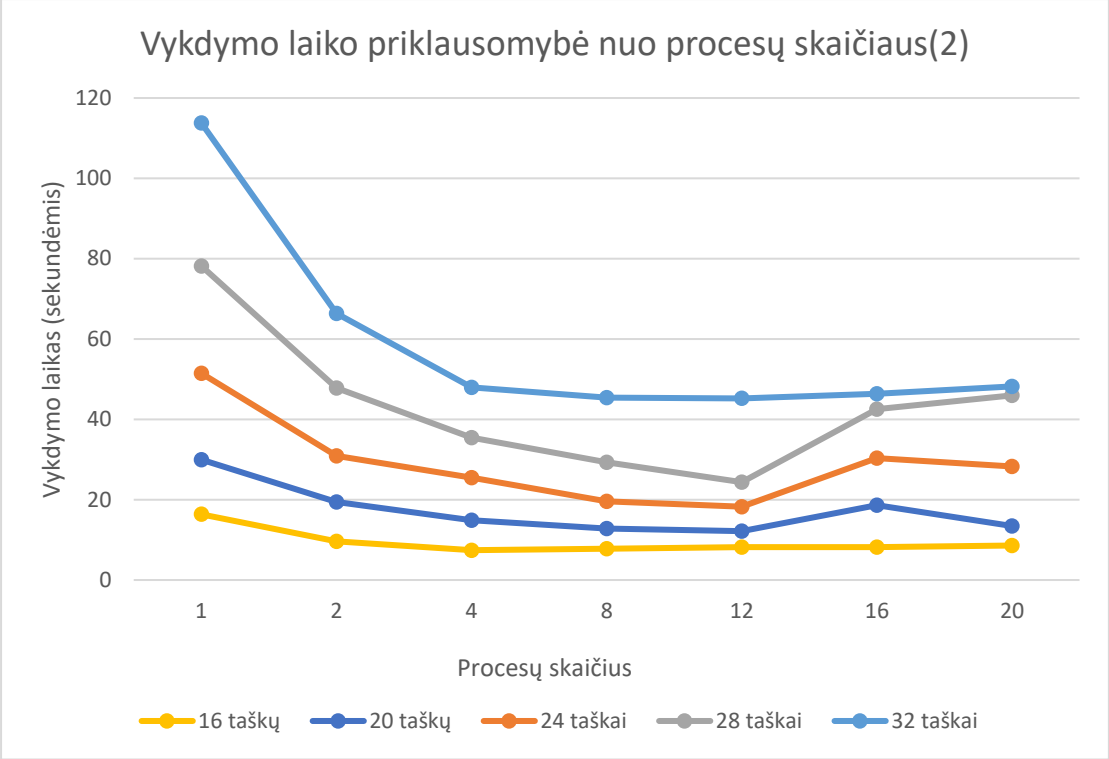
Norint pasileisti šią programą reikia parsisiųsti main.py failą iš moodle. Naudojau VS Code IDE, ir tiesiog galima atsidaryti failą per ten ir paleisti, parodys kurias bibliotekas reikia suinstaliuoti ir su instaliavus galima paleisti programą. Taip pat galima paleisti ir iš terminalo, reikia suinstaliuoti reikalinas bibliotekas (numpy ir matplotlib) **pip install matplotlib ir pip3 install numpy**. Tada nueiti į katalogą kur patalpintas parsisiųstas main.py failas ir paleisti su komanda **python main.py**

5. Vykdymo laiko kitimo tyrimas

Vykdymo laiko kitimo tyrimui bus naudojami šie taškai : 4, 8, 12, 16, 20, 24, 28, 32. Kiekvienam iš šių takų skaičių bus vykdomas optimizavimas su kiekvienu procesų skaičiumi. Procesai: 1, 2, 4, 8, 12, 16, 20. Rezultatai bus vaizduojami vykdymo laiko priklausomybe nuo procesų skaičiaus. Grafiko x ašis rodys procesų skaičių, y ašis – vykdymo laiką sekundėmis. Šiam tyrimui buvo naudotas CPU i7-8750H modelis, kuris turi 6 branduolius ir 12 gijų.



Paveikslėlis 4. Vykdymo laiko priklausomybė nuo procesų skaičiaus su 4, 8, 12 taškų



Paveikslėlis 5. Vykdymo laiko priklausomybė nuo procesų skaičiaus su 16,20, 24, 28, 32 taškams

6. Išvados

Individualaus projekto metu buvo lygiagretinama skaitinių algoritmų modulio antro laboratorinio darbo optimizavimo uždavinio programa. Tikslas buvo pagreitinti programos darbą. Daugiausiai laiko programa užtrunka ir paprasčiausia yra sulygiagretinti gradiento kiekvienam taškui skaičiavimą, dėl to ši vieta ir buvo lygiagretinama

Išskyriau į du skirtingus grafikus, kad būtų geriau matyti skirtumai tarp vykdymo laikų (kad nebūtų per daug didelis skaičius tarp ilgiausio ir trumpiausio laiko, ir geriau matytųsi). Pagal tai galima įžvelgti

- Pirmame grafike (4 paveikslėlis) yra naudojami gana nedideli taškų skaičiai (nuo 4 iki 12) ir programą daug efektyviau yra vykdyti su mažiau paleistų procesų, taip yra dėl to, kad gradientų skaičiavimas užtrunka sąlyginai trumpai ir daugiau laiko užtrunka pats programos lygiagretinimas bei konteksto perjunginėjimas
- Antrame grafike (5 paveikslėlis) yra naudojami didesni taškų skaičiai, dėl to gradiento skaičiavimas užtrunka ilgiau ir matoma, jog kuo daugiau procesų paduodama, tuo greičiau atliekamas darbas, tai pastebima iki 12 procesų, nuo tada darbas nepagreitėja, dažnai net sulėtėja, nes procesorius turi tik 12 gijų ir vienu metu gali atlikti tiek procesų, visi papildomai procesai tiesiog kainuoja daugiau konteksto perjungimų ir tai lėtina veikimo laiką.
- Galiausiai, antrame grafike matome, kad su pačiu didžiausiu paduotu taškų skaičiumi (32) programa logaritmiškai greitėja. Tokio elgesio ir buvo tikėtasi

7. Literatūra

1. https://moodle.ktu.edu/pluginfile.php/45753/mod_resource/content/5/13-python.pdf
2. Stack Overflow - Where Developers Learn, Share, & Build Careers
3. multiprocessing — Process-based parallelism — Python 3.9.1rc1 documentation