Kauno technologijos universitetas

Informatikos fakultetas

P170B328 Lygiagretusis programavimas

Inžinerinis projektas – duomenų lygiagretumo priemonių taikymas

Nedas Liaudanskis

Studentas

doc. Prakt. Ryselis Karolis

Dėstytojas

KAUNAS, 2023

Turinys

Užduotis

Pagal pateiktą uždavinio sąlygą sudarykite tikslo funkciją ir išspręskite jį vienu iš gradientinių metodų (gradientiniu, greičiausio nusileidimo).

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:

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

Ši uždavinio sprendimui gauti naudojama, C# programavimo kalba, kartu su parallel LINQ priemone, kuri padeda valdyti gijas.

Užduoties analizė ir sprendimas

Visas uždavinys susiėda iš esamų ir naujų parduotuvių. Kiekviena parduotuvė turi savo koordinates: (x_1, y_2). Panaudodami šias koordinates galime rasti kainą tarp parduotuvių. Uždavinio tikslas yra gauti ko mažesnę kainą parduotuvių išsidėstėme.

Atstumo tarp dviejų parduotuvių, kurių koordinatės $(\boldsymbol{x1}, \boldsymbol{y1})$ ir $(\boldsymbol{x2}, \boldsymbol{y2})$, kaina apskaičiuojama pagal formulę:

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

Parduotuvės, kurios koordinatės $(\boldsymbol{x1}, \boldsymbol{y1})$, vietos kaina apskaičiuojama pagal formulę:

$$C^p(x_1, y_1) = \frac{x_1^4 + y_1^4}{1000} + \frac{\sin(x_1) + \cos(y_1)}{5} + 0{,}4$$

Pats uždavinio sprendimo principas gan paprastas. Naudojamas gradientinio nusileidimo algoritmas optimizuojant naujų parduotuvių vietas. Ši optimizacija vyksta per iteracijas, kur kiekviena naujoji parduotuvė yra atnaujinama pagal gradientą ir apribojama koordinates, kad jos nepersikristų su miesto ribomis ($-10 \leq \boldsymbol{x} \leq 10$, $-10 \leq \boldsymbol{y} \leq 10$).

Lygiagretumą naudojame apskaičiuoti atstumus tarp naujų ir esamų parduotuvių, naujų parduotuvių vietos kainas ir atnaujinant naujų parduotuvių vietas, naudojant gradientinį nusileidimą. Taip padeda greičiau atlikti iteracijas, kurios be paralelizmo užtruktu labai ilgai.

Gradientams apskaičiuoti naudojame, baigtinių skirtinių metodą:

$$f'(x) \approx \frac{f(x+h) - f(h)}{h}$$

h – nedidelis dydis, naudojamas gauti mažą keitinį, kuris nustato funkcijos augimo kryptį.

Kodas

- **Store klasė**: Aprašoma parduotuvės pozicija (x, y).

```
public class Store
{
    public double X { get; set; }
    public double Y { get; set; }
}
```

- **DistanceCost funkcija**: Skaičiuoja atstumo kainą tarp dviejų parduotuvių pagal pateiktą formulę.

```
static double DistanceCost(Store store1, Store store2)
    {
        return Math.Exp(-0.3 * ((store1.X - store2.X) * (store1.X - store2.X) +
(store1.Y - store2.Y) * (store1.Y - store2.Y)));
    }
```

- **PlacementCost funkcija**: Skaičiuoja parduotuvės vietos kainą pagal pateiktą formulę.

```
static double PlacementCost(Store newStore)
    {
        return (Math.Pow(newStore.X, 4) + Math.Pow(newStore.Y, 4)) / 1000 +
(Math.Sin(newStore.X) + Math.Cos(newStore.Y)) / 5 + 0.4;
    }
```

- **TotalCost funkcija**: Skaičiuoja visą pastatymo kainą, apimantį atstumus tarp naujų ir esamų parduotuvių bei vietos kainas.

```
static double TotalCost(List<Store> existingStores, List<Store> newStores)
    {
        double totalCost = 0;

        totalCost += newStores.AsParallel().WithDegreeOfParallelism(parallelThreads)
            .SelectMany(newStore => existingStores, (newStore, existingStore) =>
DistanceCost(newStore, existingStore)).Sum();

        totalCost += newStores.AsParallel().WithDegreeOfParallelism(parallelThre-
ads).Select(newStore => PlacementCost(newStore)).Sum();

        return totalCost;
}
```

- **Main funkcija**: Pagrindinė programa, kurioje vartotojas įveda esamų ir naujų parduotuvių kiekius. Sugeneruojamos pradinės parduotuvių vietos ir vykdomas gradientinio nusileidimo

algoritmas per tam tikrą iteracijų skaičių. Rezultatai išvedami į konsolę, kartu su vykdymo laiko informacija.

```
static void Main()
    {
        Console.Write("Enter the number of existing stores: ");
        int n = Convert.ToInt32(Console.ReadLine());
        Console.Write("Enter the number of new stores to add: ");
        int m = Convert.ToInt32(Console.ReadLine());

        Console.Write("Enter the number of parallel threads: ");
        SetThreads(Convert.ToInt32(Console.ReadLine()));

        Console.WriteLine("-----------------------------------------------");

        List<Store> existingStores = Enumerable.Range(1, n).Select(_ =>
            new Store
            {
                X = random.NextDouble() * 20 - 10,
                Y = random.NextDouble() * 20 - 10
            }).ToList();

        List<Store> newStores = Enumerable.Range(1, m).Select(_ =>
            new Store
            {
                X = random.NextDouble() * 20 - 10,
                Y = random.NextDouble() * 20 - 10
            }).ToList();

        int maxIterations = 100;
        double learningRate = 0.1;

        Stopwatch stopwatch = new Stopwatch();
        stopwatch.Start();

        for (int iteration = 0; iteration < maxIterations; iteration++)
        {
            newStores = newStores.AsParallel().WithDegreeOfParallelism(paral-
lelThreads)
                .Select(newStore =>
                {
                    (double gradientX, double gradientY) = CalculateGra-
dient(existingStores, newStores, newStore);

                    newStore.X -= learningRate * gradientX;
                    newStore.Y -= learningRate * gradientY;

                    newStore.X = Math.Max(-10, Math.Min(10, newStore.X));
                    newStore.Y = Math.Max(-10, Math.Min(10, newStore.Y));

                    return newStore;
                }).ToList();

            Console.WriteLine($"Iteration {iteration + 1}, Total Cost: {Total-
Cost(existingStores, newStores)}");
        }

        stopwatch.Stop();
        Console.WriteLine($"Total execution time: {stopwatch.ElapsedMilli-
seconds} milliseconds");

        Console.WriteLine("-----------------------------------------------");

        Console.WriteLine("Initial store locations:");
```

```
        foreach (var store in existingStores)
        {
            Console.WriteLine($"X: {store.X}, Y: {store.Y}");
            Console.WriteLine("Store cost: " + PlacementCost(store));
        }

        Console.WriteLine("-----------------------------------------------");

        Console.WriteLine("New store locations:");
        foreach (var store in newStores)
        {
            Console.WriteLine($"X: {store.X}, Y: {store.Y}");
            Console.WriteLine("Store cost: " + PlacementCost(store));
        }
          }
```

- **CalculateGradient funkcija**: Skaičiuoja gradientą, reikalingą naujos parduotuvės pozicijos atnaujinimui gradientinio nusileidimo metodu.

```
static (double, double) CalculateGradient(List<Store> existingStores, List<Store>
newStores, Store currentStore, double epsilon = 1e-6)
    {
        double originalX = currentStore.X;
        double originalY = currentStore.Y;

        double originalCost = TotalCost(existingStores, newStores);

        currentStore.X = originalX + epsilon;
        double perturbedXCost = TotalCost(existingStores, newStores);

        currentStore.X = originalX;

        currentStore.Y = originalY + epsilon;
        double perturbedYCost = TotalCost(existingStores, newStores);

        currentStore.Y = originalY;

        double gradientX = (perturbedXCost - originalCost) / epsilon;
        double gradientY = (perturbedYCost - originalCost) / epsilon;

        return (gradientX, gradientY);
    }
```

Testavimas ir programos vykdymo instrukcija

Programą pasileidus, galima pasirinkti norimą gijų skaičių, pradinių(jau esamų) parduotuvių kiekį ir pridedamų parduotuvių skaičių.

```
Enter the number of existing stores: 9
Enter the number of new stores to add: 9
Enter the number of parallel threads: 9
```

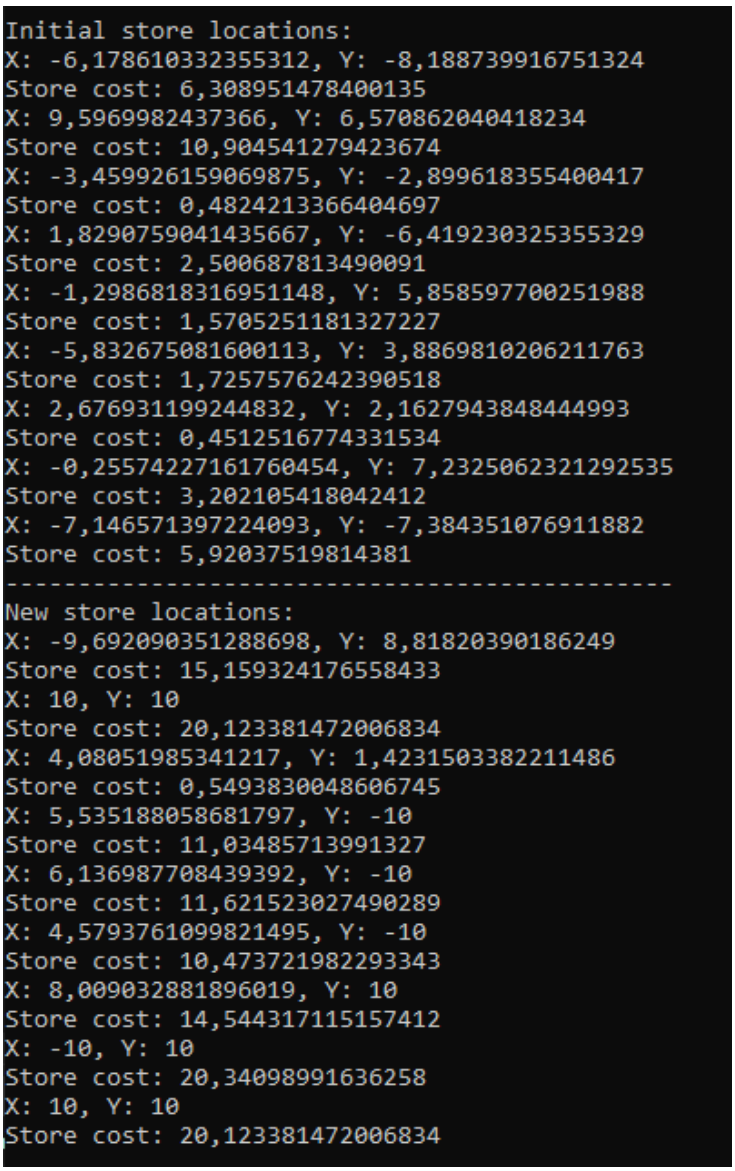
Viską įvedus vyksta skaičiavimai. Pateikiamos visos iteracijos ir kiekvienoje iteracijoje esančios kainos suma.

```
Iteration 1, Total Cost: 72,99368081089233
Iteration 2, Total Cost: 98,51852074256368
Iteration 3, Total Cost: 163,49084973352205
Iteration 4, Total Cost: 130,65015456359018
Iteration 5, Total Cost: 119,90207247445885
Iteration 6, Total Cost: 147,82124504530827
Iteration 7, Total Cost: 150,37261181505662
Iteration 8, Total Cost: 135,19147311770047
Iteration 9, Total Cost: 142,12908863965228
Iteration 10, Total Cost: 136,03838274446554
Iteration 11, Total Cost: 136,40505161569567
Iteration 12, Total Cost: 107,21295454944156
Iteration 13, Total Cost: 118,01579806416413
Iteration 14, Total Cost: 114,09985908634432
Iteration 15, Total Cost: 117,9777651593297
Iteration 16, Total Cost: 106,94042193377555
Iteration 17, Total Cost: 161,11737235800967
Iteration 18, Total Cost: 146,00302475632296
Iteration 19, Total Cost: 129,09114093760607
Iteration 20, Total Cost: 117,92271127088574
Iteration 21, Total Cost: 102,59579854036905
Iteration 22, Total Cost: 107,20197811940749
Iteration 23, Total Cost: 96,69928930229008
```

Užbaigus skaičiavimus yra pateikiamas veikimo laikas, kuris bus naudojamas sudaryti grafikus.

```
Total execution time: 209 milliseconds
```

Po skaičiavimo laiko yra parodoma pradinių parduotuvių koordinatės ir naujų parduotuvių koordinatės, su parduotuvės kaina.



Vykdymo laiko kitimo tyrimas

Visi tyrimai buvo atlikti naudojant **Lenova legion 5** kompiuterį:

Processor: AMD Ryzen 7 5800H with Radeon Graphics	3.20 GHz
Installed RAM: 16,0 GB (15,9 GB usable)	
VRAM: NVIDIA GeForce RTX 3060 laptop	

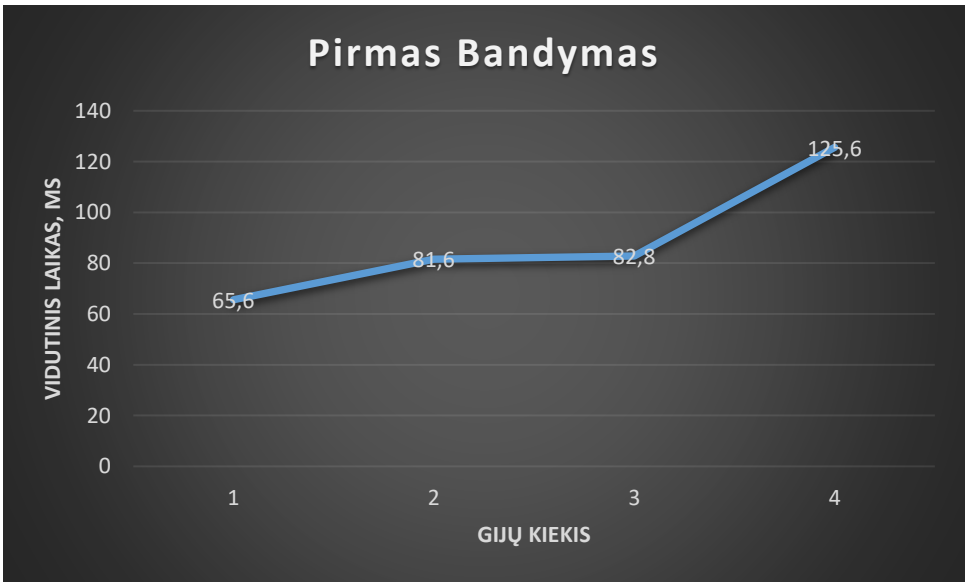
Testavimas buvo daromas 8 kartus, su skirtingais duomenų kiekiais ir gijomis. 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, Lentelėse yra pavaizduoti visi gauti testavimo atvejai, jų duomenys ir rezultatai. Vidutinis programos veikimo laiko kitimas, priklausantis nuo gijų skaičiaus buvo atvaizduotas grafuose: 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8.

1. Pirmas bandymas, Esamos parduotuvės: 1, Naujos parduotuvės: 1.

Šiame bandyme buvo testuojamas duomenų minimumas. Kai turime tik vieną parduotuvę ir reikia pridėti tik vieną naują parduotuvę. Rezultatai gavosi tokie, kokių buvo tikėtina. Kai turime tik vieną parduotuvę, daugiau nei vienos gijos naudoti nereikia, todėl didinant gijas veikimo vidutinis laikas kyla, nes reikia tas gijas aktyvuoti ir uždaryti, kai baigiame darbą, tai duoda papildomų skaičiavimų, kurie prailgina laiką.

4.1 lentelė, Pirmas bandymas

Pirmas Bandymas				
Esamų parduotuvių skaičius=1, Naujų parduotuvių skaičius=1				
	Gijų kiekis			
Bandymo numeris	1	2	4	8
1	68	82	75	114
2	71	95	90	110
3	63	87	81	153
4	60	75	82	116
5	66	69	86	135
Vidurkis	65,6	81,6	82,8	125,6



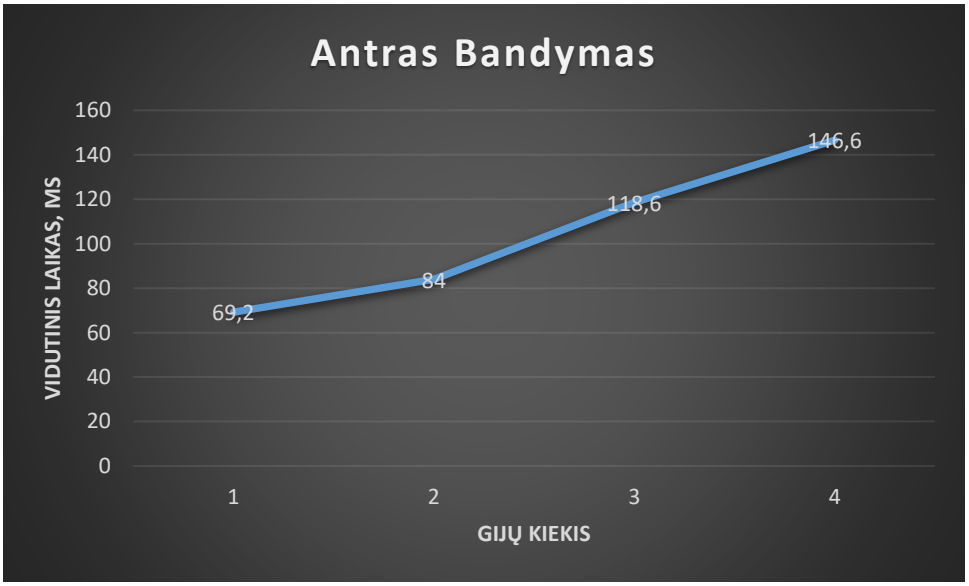
4.1 Grafas, Pirmas bandymas

2. Antras bandymas, Esamos parduotuvės: 2, Naujos parduotuvės: 2.

Antras testavimo bandymas, gavosi labai panašus į pirmą, kadangi duomenų kiekis vis dar labai mažas, gijų aktyvavimas ir uždarymas kainuoja daugiau laiko, negu jų panaudojimas.

4.2 lentelė, Antras bandymas

Antras bandymas				
Esamų parduotuvių skaičius=2, Naujų parduotuvių skaičius=2				
	Gijų kiekis			
Bandymo numeris	1	2	4	8
1	63	81	113	166
2	60	84	110	146
3	70	85	122	131
4	81	81	148	152
5	72	89	100	138
Vidurkis	69,2	84	118,6	146,6



4.2 Grafas, Antras bandymas

3. Trečias bandymas, Esamos parduotuvės: 4, Naujos parduotuvės: 4.

Trečiame bandymo rezultatai labai panašūs į antro bandymo, tai komentarų daug neturiu,

4.3 lentelė, Trečias bandymas

Trečias bandymas				
Esamų parduotuvių skaičius=4, Naujų parduotuvių skaičius=4				
	Gijų kiekis			
Bandymo numeris	1	2	4	8
1	77	94	124	129
2	98	80	125	150
3	76	100	120	145
4	91	123	196	136
5	72	107	145	147
Vidurkis	82,8	100,8	142	141,4

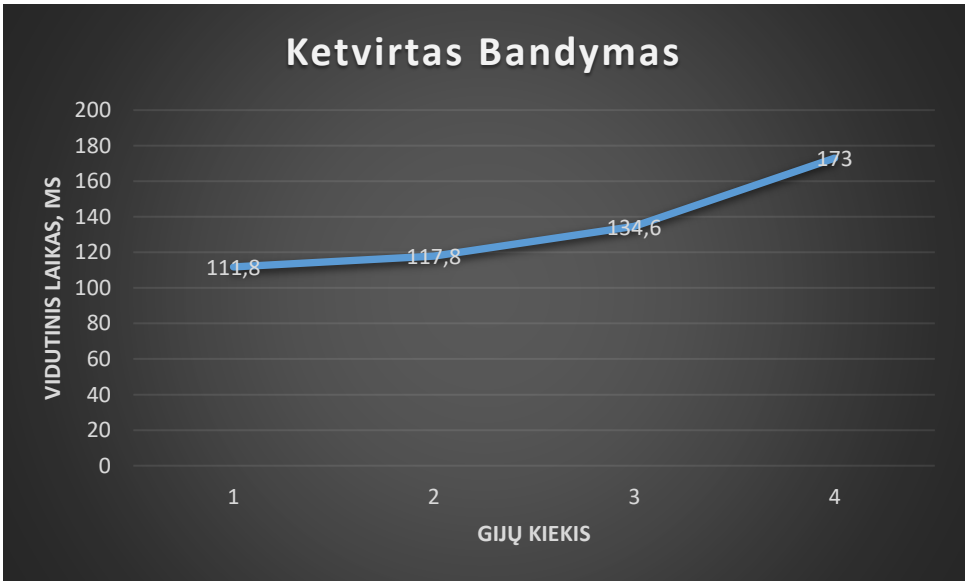
3.3 Grafas, Trečias bandymas

4. Ketvirtas bandymas, Esamos parduotuvės: 8, Naujos parduotuvės: 8.

Ketvirtame bandyme, galime pastebėti, jog naudojant vieną ar dvi gijas, vidutinis veikimo laikas yra labai panašus. Čia yra pakytis lyginant su pieštai darytais bandymais kai tarp gijų kiekių buvo labai dideli skirtumai, dabar jie mažėja, nes didėja duomenų kiekis ir skaičiavimai tampa sudėtingesni, todėl didesnis gijų skaičius atsiperka labiau, negu naudojant vieną giją.

4.4 lentelė, Ketvirtas bandymas

Ketvirtas bandymas				
Esamų parduotuvių skaičius=8, Naujų parduotuvių skaičius=8				
	Gijų kiekis			
Bandymo numeris	1	2	4	8
1	110	103	151	166
2	102	115	132	231
3	99	153	131	161
4	129	117	127	159
5	119	101	132	148
Vidurkis	111,8	117,8	134,6	173



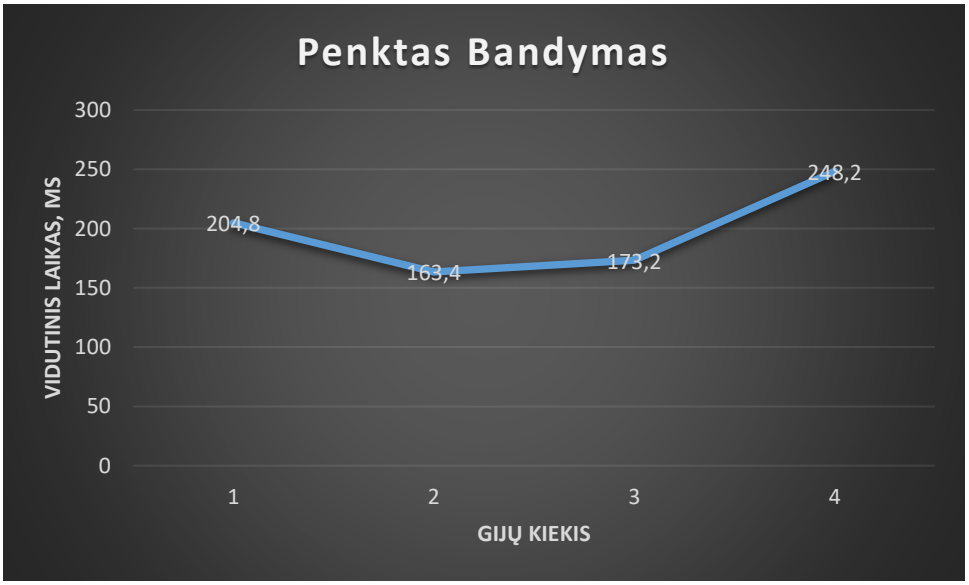
4.4 Grafas, Ketvirtas bandymas

5. Penktas bandymas, Esamos parduotuvės: 16, Naujos parduotuvės: 16.

Penktas bandymas, kai parduotuvių kiekis yra 16 ir naujų norimų parduotuvių yra 16. Galime pastebėti, jog 2 ir 4 gijų rezultatai yra daug greičiau gaunami nei naudojant 1 ar 8 gijas. Didesnis duomenų kiekis tampa per didelis vienai gijai, o 8 gijų startavimas ir baigimas kainuoja daugiau negu panaudojimas, todėl tokiam duomenų kiekiui puikiai tinka 2, 4 gijos.

4.5 lentelė, Penktas bandymas

Penktas bandymas				
Esamų parduotuvių skaičius=16, Naujų parduotuvių skaičius=16				
	Gijų kiekis			
Bandymo numeris	1	2	4	8
1	202	166	173	204
2	200	160	180	289
3	201	165	166	271
4	214	169	178	243
5	207	157	169	234
Vidurkis	204,8	163,4	173,2	248,2



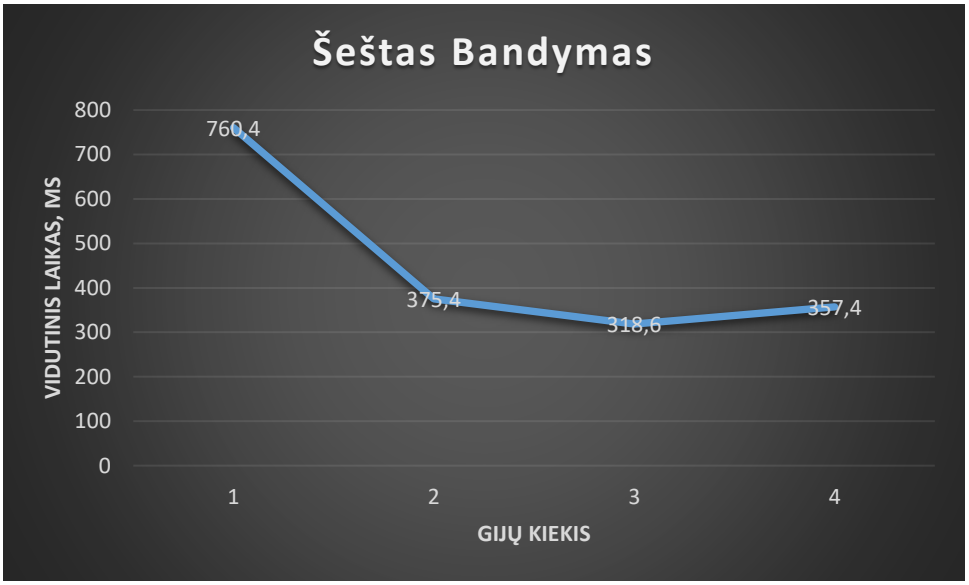
4.5 Grafas, Penktas grafikas

6. Šeštas bandymas, Esamos parduotuvės: 32, Naujos parduotuvės: 32.

Šeštame bandyme, pastebime, jog naudojant vieną giją norint apskaičiuoti šiuos duomenis užtrunka labai ilgą laiką 760 ms. O naudojant 2, 4, galime pamatyti, jog skaičiavimų laikai krinta taip kaip ir turėtų, 2 gijos užtrunka ilgiau negu 4 gijos. 8 gijos dirba ilgiau, nes duomenų kiekis per mažas.

4.6 lentelė, Šešta lentelė

Šeštas bandymas				
Esamų parduotuvių skaičius=32, Naujų parduotuvių skaičius=32				
	Gijų kiekis			
Bandymo numeris	1	2	4	8
1	754	378	305	354
2	768	389	331	362
3	744	369	324	366
4	781	377	317	347
5	755	364	316	358
Vidurkis	760,4	375,4	318,6	357,4

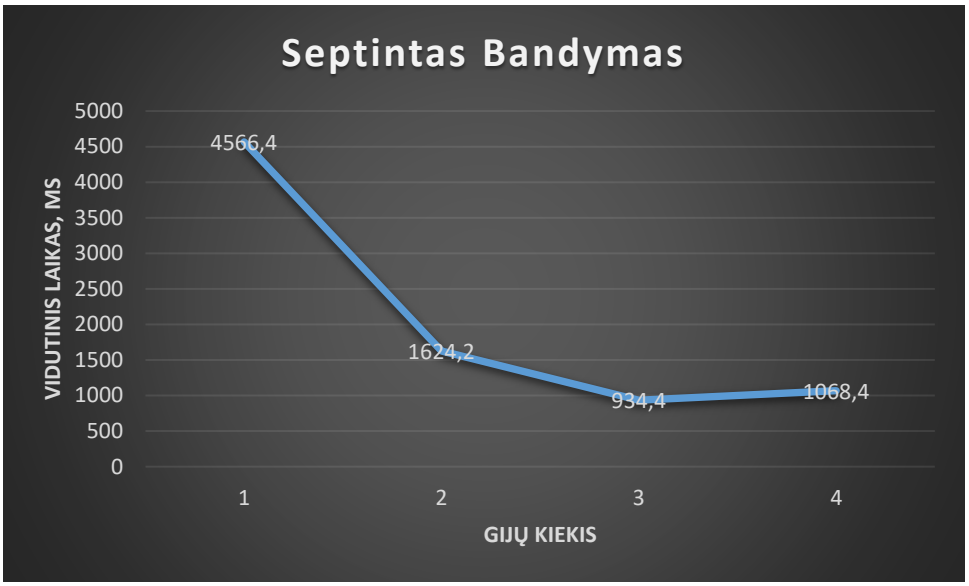


4.6 Grafas, Šeštas bandymas

7. Septintas bandymas, Esamos parduotuvės: 64, Naujos parduotuvės: 64.

4.7 lentelė, Septintas bandymas

Septintas bandymas				
Esamų parduotuvių skaičius=64, Naujų parduotuvių skaičius=64				
	Gijų kiekis			
Bandymo numeris	1	2	4	8
1	4552	1607	964	1091
2	4561	1659	932	1102
3	4588	1630	907	1121
4	4572	1583	947	974
5	4559	1642	922	1054
Vidurkis	4566,4	1624,2	934,4	1068,4



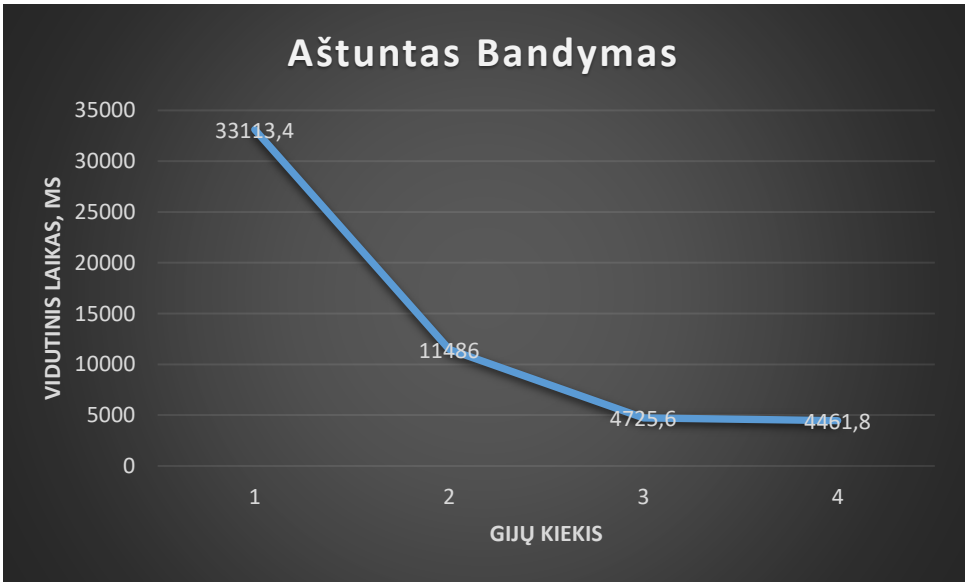
4.7 Grafas, Septintas bandymas

8. Aštuntas bandymas, Esamos parduotuvės: 128, Naujos parduotuvės: 128.

Aštuntas bandymas, parodo kaip nuo gijų skaičiaus priklauso greitis kai turime daug duomenų. Kuo daugiau gijų, tuo greitesnis gaunamas skaičiavimas, tok kol gijų paleidimas ir užbaigimas, nekainuoja daugiau negu skaičiavimai.

4.8 lentelė, Aštuntas bandymas

Aštuntas bandymas				
Esamų parduotuvių skaičius=128, Naujų parduotuvių skaičius=128				
	Gijų kiekis			
Bandymo numeris	1	2	4	8
1	32990	10257	4521	4550
2	33241	10391	4690	4439
3	32974	10421	4757	4453
4	33114	12890	4789	4478
5	33248	13471	4871	4389
Vidurkis	33113,4	11486	4725,6	4461,8



4.8 Grafas, Aštuntas bandymas

Išvados

Projektas pavyko puikiai. Išmokau naudotis C# parallel LINQ priemone. Su ja lengvai pritaikiau lygiagretumą, savo uždaviniui. Atsakymai iš testavimų gavosi tokie kokie buvo tikėtini. Norint naudoti gijas efektyviai reikia žinoti, proporcingumą tarp duomenų ir gijų skaičiaus. Nes norint prižiūrėti didelį skaičių gijų kainuoja papildomų laiko išlaidų, kurių nereikia skaičiuojant mažus duomenų kiekius.