

Laboratoriniai darbai

Domantas Keturakis

Lapkritis 2024

TURINYS

Užduotis #1	2
Pirma dalis	2
Lygiagretinimo galimybių analizė	2
Sprendimo lygiagretinimas	4
Rezultatai	5
Antra dalis	5
Rezultatai	6
Užduotis #2	7
Pirma dalis	7
Pirmas bandymas	7
Rezultatai	9
Antras bandymas	9

UŽDUOTIS #1

Pirma dalis

Lygiagretinimo galimybių analizė

Pirmasis skaičiavimo skaičiavimo ciklas (Pav. 1) palyginus užtrunka nedaug laiko (apie 0.002 sekundės). Praktiniams tikslams, jį galima ignoruoti.

```
1 for (int i=0; i<numX; i++) {  
2     X[i] = i;  
3     bestX[i] = i;  
4 }  
5 u = evaluateSolution(X);  
6 bestU = u;
```

Pav. 1: Pradinės naujo ir geriausio sprendinių reikšmių apskaičiavimas

Didžiąją dalį laiko užima šis ciklas:

```
1 while (increaseX(X, numX-1, numCL) == true) {  
2     u = evaluateSolution(X);  
3     if (u > bestU) {  
4         bestU = u;  
5         for (int i=0; i<numX; i++) bestX[i] = X[i];  
6     }  
7 }
```

Pav. 2: Visų galimų sprendinių perrinkimas

Šio ciklo iš esmės “aklai” parelizuoti negalima, nes reikia atsižvelgti į tai kad:

- `increaseX` - keičia masyvo `X` reikšmę (Pav. 3), ir ne tik `index`-ąjį elementą, rekursyviai kviesdamas save sumažina `index` reikšmę vienu, t.y. iškvietus `increaseX` visos masyvos reikšmės yra keičiamos. To pasekmė, kad `index`-ojo elemento skaičiavimo negalima paskirstyti skirtingoms gijoms, kitaip vėlesnėms gijoms reiktų laukti, kol praeita gija baigs savo darbą, visiškai nustelbiant paralelizavimo naudą.
- `increaseX` skaičiavimai priklauso vienas nuo kito, t.y. norint apskaičiuoti `X` reikšmę n -ame ciklo vykdyme, reikia pirma apskaičiuoti `X` reikšmę $(n - 1)$ -ame ciklo vykdyme. Analogiškai negalima lygiagretinti nes kitos gijos lauktų, kol praeita gija baigs savo darbą.
- `if (u > bestU) { ... }` irgi gali tik vienas ciklas vienu metu, nes `bestU` ir `X` pakeitimas turi būti atliekamas “žingsniu” - t.y. atomiškai.

Iš esmės neperrašius `increaseX`, `X` skaičiavimų lygiagretinti nėra praktiška.

```

1  int increaseX(int *X, int index, int maxindex) {
2      if (X[index]+1 < maxindex-(numX-index-1)) {
3          X[index]++;
4      }
5      else {
6          if ((index == 0) && (X[index]+1 == maxindex-(numX-index-1))) {
7              return 0;
8          }
9          else {
10             if (increaseX(X, index-1, maxindex)) X[index] = X[index-1]+1;
11             else return 0;
12          }
13      }
14      return 1;
15 }

```

Pav. 3: Funkcija increaseX

Tuo tarpu funkcija evaluateSolution (Pav. 4) nekeičia jokių globalių kintamųjų ar savo argumentų. Analitiškai žiūrint galima spėti, kad čia ir didžioji dalis skaičiavimo laiko yra sugaištama. Teorinis šios funkcijos *big O* yra $O(\text{numDP} \cdot \text{numX})$, tuo tarpu increaseX rekursyviai save gali iškviesti daugiausiai numX kartų.

```

1  double evaluateSolution(int *X) {
2      double U = 0; double totalU = 0;
3      int bestPF, bestX;
4      double d;
5
6      for (int i=0; i<numDP; i++) {
7          totalU += demandPoints[i][2];
8          bestPF = 1e5;
9          for (int j=0; j<numPF; j++) {
10             d = HaversineDistance(i, j);
11             if (d < bestPF) bestPF = d;
12          }
13          bestX = 1e5;
14          for (int j=0; j<numX; j++) {
15             d = HaversineDistance(i, X[j]);
16             if (d < bestX) bestX = d;
17          }
18
19          if (bestX < bestPF) U += demandPoints[i][2];
20          else if (bestX == bestPF) U += 0.3*demandPoints[i][2];
21      }
22      return U/totalU*100;
23 }

```

Pav. 4: Funkcija evaluateSolution

Sprendimo lygiagretinimas

Dėl anksčiau išvardintų priežasčių `increaseX` apskaičiavimas išskiriamas į `critical` bloką, tam, kad tik viena gija galėtų modifikuoti `X` reikšmę vienu metu. Apskaičiavus ir atnaujinus `X`, kiekviena gija susikuria savo `X` kopiją - `localX`. Šią kopiją galima naudoti `evaluateSolution` nes jinais ne bus keičiama. Kiekviena gija taip pat gauna `u` kopiją į kurią įrašo `evaluateSolution` apskaičiuotą reikšmę. Ciklo gale vėl naudojamas `critical`, tam kad tik viena gija vienu metu galėtų įvertinti `u > bestU` ir pakeisti `bestU` ir `bestX` reikšmes.

```
1  bool increased = true;
2  int *manyXs = new int[NUM_THREADS * numX];
3
4  #pragma omp parallel private(u)
5  {
6      while (increased) {
7          int thread_id = omp_get_thread_num();
8          int *localX = manyXs + (thread_id * numX);
9
10         #pragma omp critical(increaseX)
11         {
12             increased = increaseX(X, numX-1, numCL);
13             memcpy(localX, X, sizeof(int) * numX);
14         }
15
16         u = evaluateSolution(localX);
17
18         #pragma omp critical(best)
19         {
20             if (u > bestU) {
21                 bestU = u;
22                 memcpy(bestX, localX, sizeof(int) * numX);
23             }
24         }
25     }
26 }
```

Pav. 5: Sulygiagretintas visų galimų sprendinių perrinkimas

Rezultatai

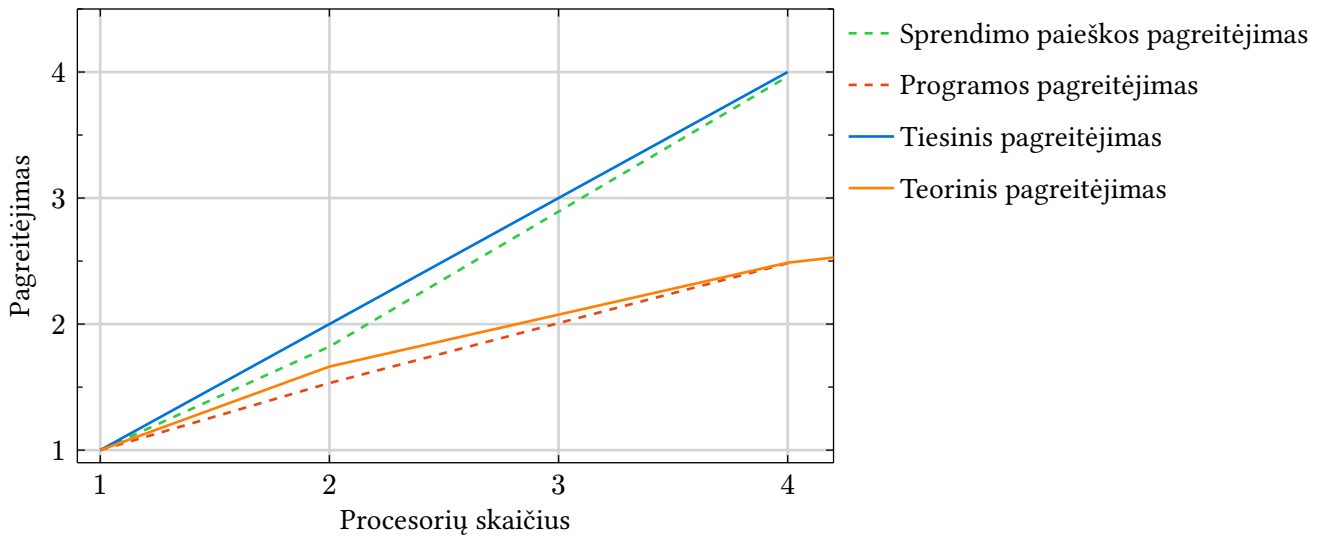


Fig. 1: Pagreitėjimo ir Procesorių skaičiaus santykis, kai matricos skaičiavimas nuoseklus

Iš Fig. 1 matoma, kad lygiagretinamos dalies pagreitėjimas seka tiesinį pagreitėjimą. O visos programos pagreitėjimas seka teorinį pagreitėjimą nusakytą pagal Amdalo dėsnį.

Antra dalis

Šioje vietoje for direktyva atrodo lengvai pritaikoma, kadangi atstumų matricos kiekvieną eilutę galima apskaičiuoti nepriklausomai nuo to ar praeitos eilutės yra apskaičiuotos. Tačiau, pirmos eilutės reikia žymiai mažiau laiko negu paskutines, todėl pritaikyta *guided* paskirstymo (angl. *scheduling*) direktyva. Tai leidžia efektyviau paskirstyti ciklą darbą per skirtingas gijas.

```

1  distanceMatrix = new double*[numDP];
2  #pragma omp parallel
3  {
4      #pragma omp for schedule(guided)
5      for (int i=0; i<numDP; i++) {
6          distanceMatrix[i] = new double[i+1];
7          for (int j=0; j<=i; j++) {
8              distanceMatrix[i][j] = HaversineDistance(
9                  demandPoints[i][0],
10                 demandPoints[i][1],
11                 demandPoints[j][0],
12                 demandPoints[j][1]
13             );
14         }
15     }
16 }
```

Pav. 6: Sulygiagretintas atstumų matricos skaičiavimas

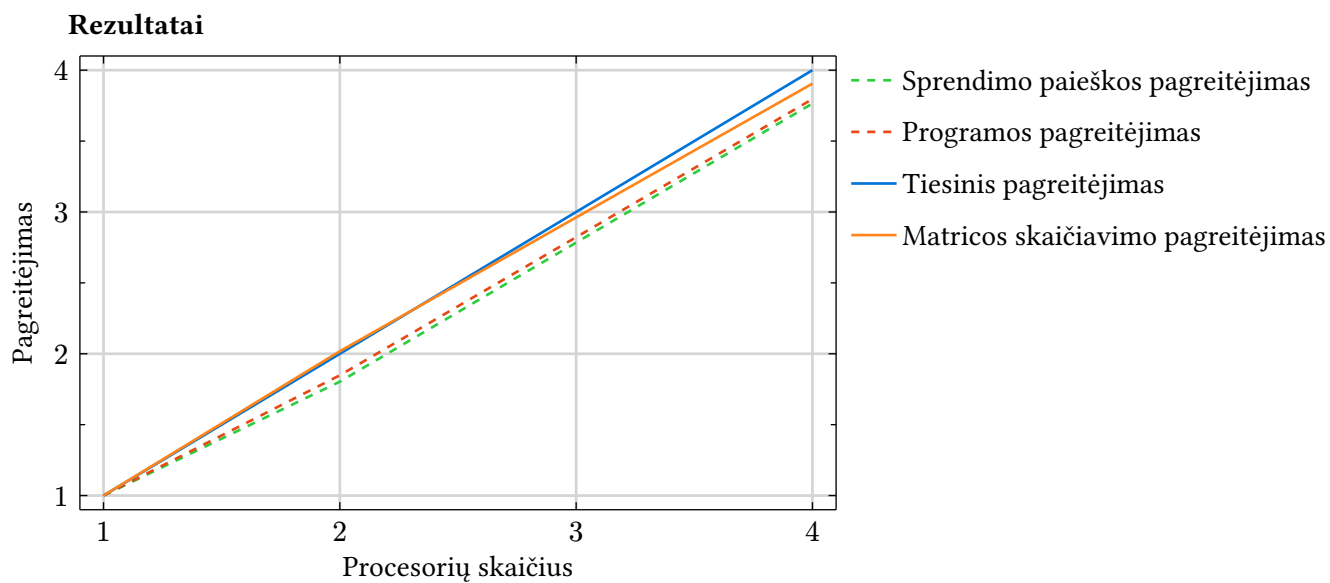


Fig. 2: Pagreitėjimo ir Procesorių skaičiaus santykis, kai matricos skaičiavimas sulygiagretintas

Iš Fig. 2 matoma, kad matricos skaičiavimo ir sprendimo ieškojimo pagreitėjimas seka tiesinę kreivę, kas matosi ir visos programos pagreitėjime, kuri irgi seka tiesinę kreivę.

UŽDUOTIS #2

Sulygiagretinti skaičiavimus naudojantis MPI biblioteka.

Pirma dalis

Sulygiagretinti sprendinio skaičiavimus.

Pirmas bandymas

Procesus galima paskirstyti taip, kad yra vienas pagrindinis (*main*) procesas ir likę - darbuotojai (*workers*). Kur pagrindinis procesas skaičiuoja atnaujintas X reikšmes ir jas išsiunčia kitiems procesams.

```
1  int world_size; MPI_Comm_size(MPI_COMM_WORLD, &world_size);
2  int world_rank; MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
3
4  // Iškirpta: loadDemandPoints(), distanceMatrix skaičiavimas, MPI_Buffer_attach,
   // t.t.
5  MPI_Barrier(MPI_COMM_WORLD);
6
7  bool increased = true;
8  while (increased) {
9      int sends = 1;
10     if (world_rank == 0) {
11         for(int ix = 1; ix < world_size; ++ix) {
12             increased = increaseX(X, numX - 1, numCL);
13
14             MPI_Send(X, numX, MPI_INT, ix, DATA_X, MPI_COMM_WORLD);
15             sends = ix + 1;
16
17             if (!increased) {
18                 for (int ix = 1; ix < world_size; ++ix) {
19                     MPI_Bsend(X, 0, MPI_BYTE, ix, SIGNAL_DONE, MPI_COMM_WORLD);
20                 }
21                 break;
22             }
23         }
24     }
25     // ...
26 }
```

Radus galutinę X reikšmę pagrindinis procesas išsiučia žinutę su žyme SIGNAL_DONE, kad pranešti procesams darbuotojams, kad jie gali baigti savo darbą.

Tuo tarpu procesai darbuotojai laukia naujos X reiškęs, jos sulaukę, apskaičiuoja u reikšmę ir išsiunčia ją, kartu su X, pagrindiniam procesui. Taip pat jie patikrinina ar negavo žinutės su SIGNAL_DONE žyma, kurios pagalba jie sužino, kad daugiau negaus X reiškių ir todėl gali baigti savo darbą.

```
1  while(increased) {
2      // ...
3      if (world_rank != 0) {
4          MPI_Status status;
5          MPI_Probe(0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
6
7          if (status.MPI_TAG == SIGNAL_DONE) {
8              MPI_Recv(NULL, 0, MPI_BYTE, 0, SIGNAL_DONE, MPI_COMM_WORLD,
9                  MPI_STATUS_IGNORE);
10             break;
11         }
12         MPI_Recv(X, numX, MPI_INT, 0, DATA_X, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
13         u = evaluateSolution(X);
14
15         MPI_Send(&u, 1, MPI_DOUBLE, 0, DATA_U, MPI_COMM_WORLD);
16         MPI_Send(X, numX, MPI_INT, 0, DATA_X, MPI_COMM_WORLD);
17     }
18     // ...
19 }
```

Tuo tarpu pagrindinis laukia tiek u ir X reiškių kiek išsiuntė procesams darbuotojams, gavęs reikšmę su didesne u reikšme atnauja savo bestU ir bestX kintamuosius.

```
1  while(increased) {
2      // ...
3      if (world_rank == 0) {
4          for (int ix = 1; ix < sends; ++ix) {
5              MPI_Recv(&u, 1, MPI_DOUBLE, ix, DATA_U, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
6              MPI_Recv(X, numX, MPI_INT, ix, DATA_X, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
7
8              if (u > bestU) {
9                  bestU = u;
10                 memcpy(bestX, X, sizeof(int) * numX);
11             }
12         }
13     }
14 }
```


Rezultatai

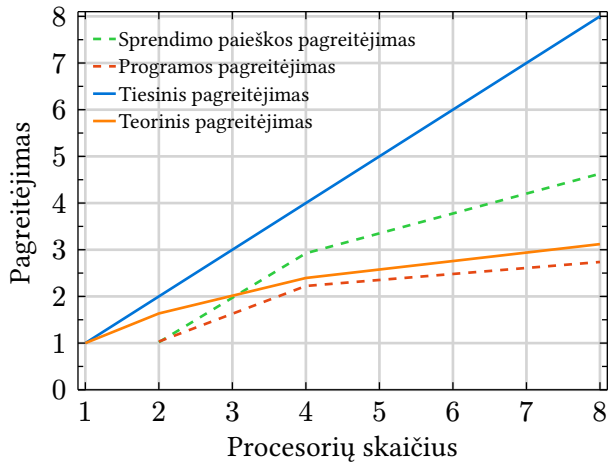


Fig. 3: Pagreitėjimo ir Procesorių skaičiaus santykis

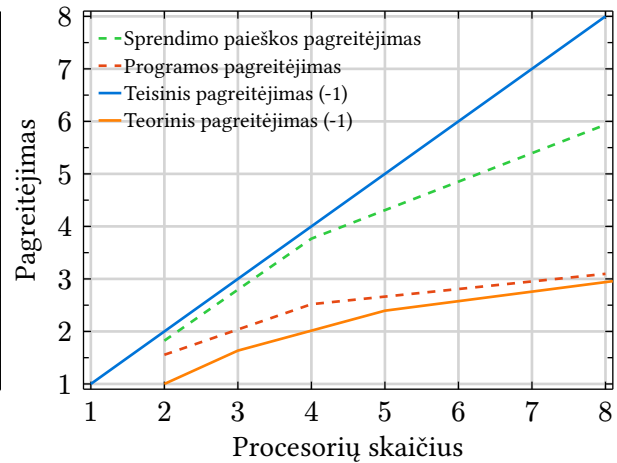


Fig. 4: Pagreitėjimo ir Procesorių skaičiaus santykis, neskaičiuojant pagrindinio proceso

Šis sprendimas iš ties yra prastas, net su 8 procesais sprendinio ieškojimas net nepasiekia puse teorinio (tiesinio) pagreitėjimo (Fig. 3). Net neskaičiuojant pagrindinio proceso (Fig. 4), t.y. skaičiuojant pagreitėjimą su 8 procesais ištikrųjų paleidžiami 9 procesai.

Antras bandymas