Laboratoriniai darbai

Domantas Keturakis

Lapkritis 2024

Turinys

Užduotis #1
Pirma dalis

Lygiagretinimo galimybių analizė

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

```c
for (int i=0; i<numX; i++) {
    X[i] = i;
    bestX[i] = i;
}
u = evaluateSolution(X);
bestU = u;
```

Pav. 1: Pradinės naujo ir geriausio sprendinių reikšmių apskaičiavimas

Didžiąją dalį laiko užima šis ciklas:

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

Pav. 2: Visų galimų sprendinių perrinkimas

Šio ciklo iš esmės "aklai" parelilizuoti 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 reikėtų laukti, kol praeita gija baigs savo darbą, visiškai nustelbiant parelelizavimo 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.

```c
int increaseX(int *X, int index, int maxindex) {
    if (X[index]+1 < maxindex-(numX-index-1)) {
        X[index]++;
    }
    else {
        if ((index == 0) && (X[index]+1 == maxindex-(numX-index-1))) {
            return 0;
        }
        else {
            if (increaseX(X, index-1, maxindex)) X[index] = X[index-1]+1;
            else return 0;
        }
    }
    return 1;
}
```

Pav. 3: Funkcija `increseX`

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 $O(\texttt{numDP} \cdot \texttt{numX})$, tuo tarpu `increseX` rekursyviai save gali iškviesti daugiausiai `numX` kartų.

```c
double evaluateSolution(int *X) {
    double U = 0; double totalU = 0;
    int bestPF, bestX;
    double d;

    for (int i=0; i<numDP; i++) {
        totalU += demandPoints[i][2];
        bestPF = 1e5;
        for (int j=0; j<numPF; j++) {
            d = HaversineDistance(i, j);
            if (d < bestPF) bestPF = d;
        }
        bestX = 1e5;
        for (int j=0; j<numX; j++) {
            d = HaversineDistance(i, X[j]);
            if (d < bestX) bestX = d;
        }

        if (bestX < bestPF) U += demandPoints[i][2];
        else if (bestX == bestPF) U += 0.3*demandPoints[i][2];
    }
    return U/totalU*100;
}
```

Pav. 4: Funkcija `evaluateSolution`

Sprendimo lygiagretinimas

Dėl anksčiau išvardintų priežąsčių `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 jinai 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.

```c
bool increased = true;
int *manyXs = new int[NUM_THREADS * numX];

#pragma omp parallel private(u)
{
    while (increased) {
        int thread_id = omp_get_thread_num();
        int *localX = manyXs + (thread_id * numX);

        #pragma omp critical(increaseX)
        {
            increased = increaseX(X, numX-1, numCL);
            memcpy(localX, X, sizeof(int) * numX);
        }

        u = evaluateSolution(localX);

        #pragma omp critical(best)
        {
            if (u > bestU) {
                bestU = u;
                memcpy(bestX, localX, sizeof(int) * numX);
            }
        }
    }
}
```

Pav. 5: Sulygiagretintas visų galimų sprendinių perrinkimas

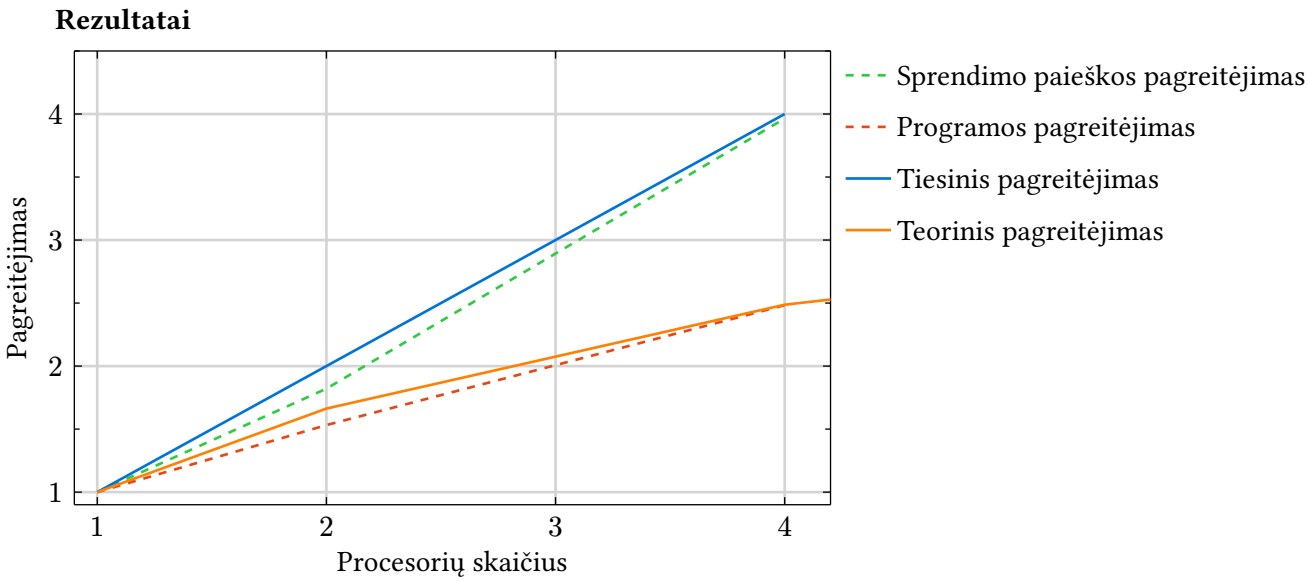

Fig. 1: Pagreitėjimo ir Procesorių skaičiaus santykis, kai matricos skaičiavimas nuoseklus

Iš Fig. 1 matoma, kad lygiagretinamos dalies pagreitėjimas seka tiesinį pagreitėjimą. O visos programos pagreitėmijas 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 eilutėms reikia žymiau mažiau laiko negu paskutinesnėms, todėl pritaikyta `guided` paskirstymo (angl. *scheduling*) direktyva. Tai leidžia efektyviau paskirstyti ciklų darbą per skirtingas gijas.

```c
distanceMatrix = new double*[numDP];
#pragma omp parallel
{
    #pragma omp for schedule(guided)
    for (int i=0; i<numDP; i++) {
        distanceMatrix[i] = new double[i+1];
        for (int j=0; j<=i; j++) {
            distanceMatrix[i][j] = HaversineDistance(
              demandPoints[i][0],
              demandPoints[i][1],
              demandPoints[j][0],
              demandPoints[j][1]
            );
        }
    }
}
```

Pav. 6: Sulygiagretintas atstumų matricos skaičiavimas

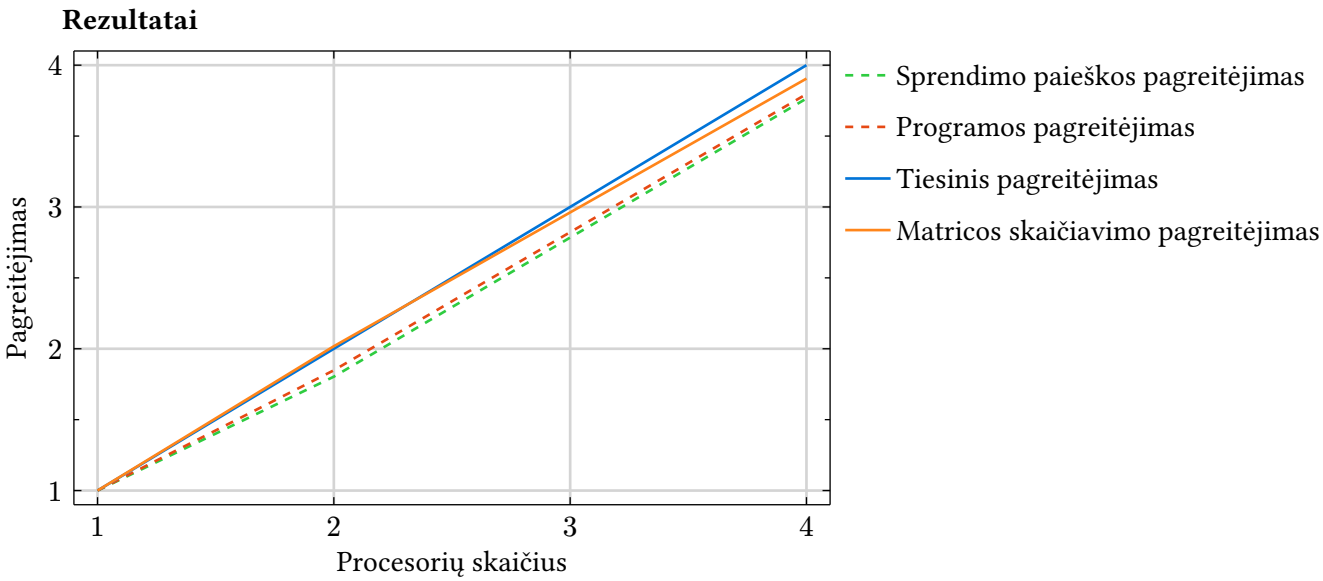

Fig. 2: Pagreitėjimo ir Procesorių skaičiaus santykis, kai matricos skaičiavimas sulygiagretintas

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.

Uždotis #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 kitiem procesam.

```c
int world_size; MPI_Comm_size(MPI_COMM_WORLD, &world_size);
int world_rank; MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

// Iškirpta: loadDemandPoints(), distanceMatrix skaičiavimas, MPI_Buffer_attach, t.t.
MPI_Barrier(MPI_COMM_WORLD);

bool increased = true;
while (increased) {
   int sends = 1;
   if (world_rank == 0) {
      for(int ix = 1; ix < world_size; ++ix) {
         increased = increaseX(X, numX - 1, numCL);

         MPI_Send(X, numX, MPI_INT, ix, DATA_X, MPI_COMM_WORLD);
         sends = ix + 1;

         if (!increased) {
            for (int ix = 1; ix < world_size; ++ix) {
               MPI_Bsend(X, 0, MPI_BYTE, ix, SIGNAL_DONE, MPI_COMM_WORLD);
            }
            break;
         }
      }
   }
   // ...
}
```

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

Tuo tarpu procesai darbuotojai laukia naujos `X` reiškės, 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` reiškmių ir todėl gali baigti savo darbą.

```c
while(increased) {
  // ...
  if (world_rank != 0) {
    MPI_Status status;
    MPI_Probe(0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

    if (status.MPI_TAG == SIGNAL_DONE) {
     MPI_Recv(NULL, 0, MPI_BYTE, 0, SIGNAL_DONE, MPI_COMM_WORLD, MPI_STATUS_IGNOR
     break;
    }

    MPI_Recv(X, numX, MPI_INT, 0, DATA_X, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    u = evaluateSolution(X);

    MPI_Send(&u, 1, MPI_DOUBLE, 0, DATA_U, MPI_COMM_WORLD);
    MPI_Send(X, numX, MPI_INT,  0, DATA_X, MPI_COMM_WORLD);
  }
  // ...
}
```

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

```c
while(increased) {
   // ...
   if (world_rank == 0) {
      for (int ix = 1; ix < sends; ++ix) {
         MPI_Recv(&u, 1, MPI_DOUBLE, ix, DATA_U, MPI_COMM_WORLD, MPI_STATUS_IGNOR
         MPI_Recv(X, numX, MPI_INT, ix, DATA_X, MPI_COMM_WORLD, MPI_STATUS_IGNORE

         if (u > bestU) {
            bestU = u;
            memcpy(bestX, X, sizeof(int) * numX);
         }
      }
   }
}
```

Rezultatai

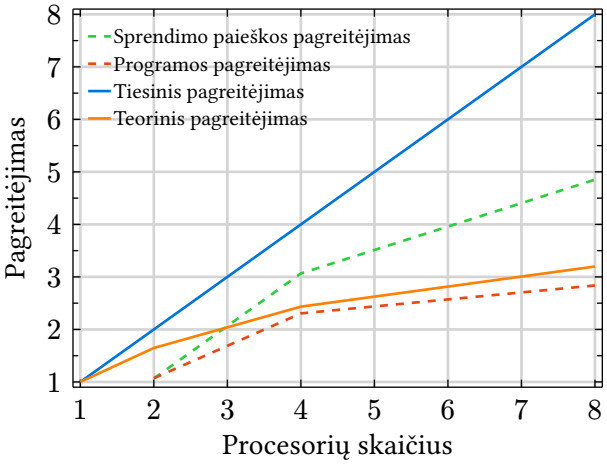

Fig. 3: Pagreitėjimo ir Procesorių skaičiaus santykis

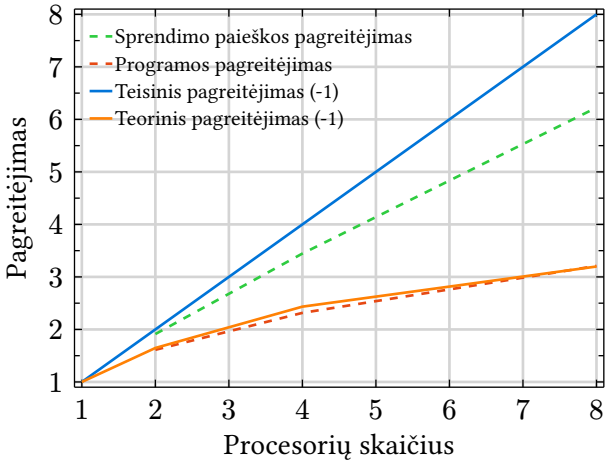

Fig. 4: Pagreitėjimo ir Procesorių skaičiaus santykis, neskaičiuojant pagrindinio proceso

Šis sprendimas nėra prastas (Fig. 3), bet pilnai neišnaudoja pagrindinio proceso, galima daryti išvadas, kad jis dažnai neturi darbo ir laukio kol galės kitiem procesam išsiųsti atnaujintas `X` reikšmes. Neskaičiuojant pagrindinio proceso (Fig. 4), t.y. skaičiuojant pagreitėjimą su 8 procesais ištikrųjų paleidžiami 9 procesai, praktiškai pasiekiamas teorinis pagreitėjimas.

Antras bandymas

Visgi galima padaryti taip, kad pagrindinis procesas irgi skaičiuotu `X` reikšmes. Kad procesai darbuotojai nelauktų, kol pagrindinis procesas baigs skaičiuoti savo dalį, nebelaukiama atsakymo iš procesų darbuotojų, `X` siuntimui pasitelkiamas `MPI_Bsend`.

```c
while (true) {
   if (world_rank == 0 && increased) {
      for(int ix = 1; ix < world_size; ++ix) {
         increased = increaseX(X, numX - 1, numCL);
         MPI_Bsend(X, numX, MPI_INT, ix, DATA_X, MPI_COMM_WORLD);
         sends += 1;

         if (!increased) {
            for (int ix = 1; ix < world_size; ++ix) {
               MPI_Bsend(&dummy_load, 1, MPI_INT, ix, SIGNAL_DONE, MPI_COMM_WORLD
            }
            break;
         }
      }
   }
// ...
}
```

Toliau `while` cikle pridedamas paskaičiavimas pagrindiniam procesui ir `SIGNAL_DONE` išsiuntimas, jeigu šiame cikle baigtusi `X` skaičiavimas:

```c
while(true) {
   // ...
   if (world_rank == 0 && increased) {
      increased = increaseX(X, numX - 1, numCL);

      u = evaluateSolution(X);
      if (u > bestU) {
         bestU = u;
         memcpy(bestX, X, sizeof(int) * numX);
      }

      if (!increased) {
         for (int ix = 1; ix < world_size; ++ix) {
            MPI_Bsend(&dummy_load, 1, MPI_INT, ix, SIGNAL_DONE, MPI_COMM_WORLD);
         }
      }
   }
   // ...
}
```

Procesams darbuotojams pagrinde ne daug kas keičiasi. Jie irgi pakeičia `MPI_Send` į `MPI_Bsend`.

```c
   if (world_rank != 0) {
      int master_done = WRP_Check_for(0, SIGNAL_DONE, MPI_COMM_WORLD);
      int master_sent_X = WRP_Check_for(0, DATA_X, MPI_COMM_WORLD);

      if (master_sent_X) {
         MPI_Recv(X, numX, MPI_INT, 0, DATA_X, MPI_COMM_WORLD, MPI_STATUS_IGNORE)

         u = evaluateSolution(X);
         if (bestU < u) { bestU = u; bestX = memcpy(bestX, X, sizeof(int) * numX)

         MPI_Bsend(&u, 1, MPI_DOUBLE, 0, DATA_U, MPI_COMM_WORLD);
         MPI_Bsend(X, numX, MPI_INT,  0, DATA_X, MPI_COMM_WORLD);
      }

      if (master_done && !master_sent_X) { break; }
   }
```

Čia `WRP_Check_for(int source, int tag, MPI_Comm comm)` (apibrėžimas Pav. 7) viduje naudoja `MPI_Iprobe(int source, int tag, MPI_Comm communicator, int* flag, MPI_Status* status)` ir gražina `flag` dalį.

```c
    if (world_rank == 0) {
       if (!increased && receives == sends) { break; }

       MPI_Status status;
       int worker_sent_X;
       MPI_Iprobe(MPI_ANY_SOURCE, DATA_X, MPI_COMM_WORLD, &worker_sent_X, &status)
       while(worker_sent_X) {
         MPI_Recv(&copy_u, 1, MPI_DOUBLE, status.MPI_STATUS, DATA_U, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
          MPI_Recv(copy_X, numX, MPI_INT, status.MPI_STATUS, DATA_X, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
          receives += 1;

          if (copy_u > bestU) {
             bestU = copy_u;
             memcpy(bestX, copy_X, sizeof(int) * numX);
          }

          worker_sent_X = WRP_Check_for(MPI_ANY_SOURCE, DATA_X, MPI_COMM_WORLD);
       }
    }
```

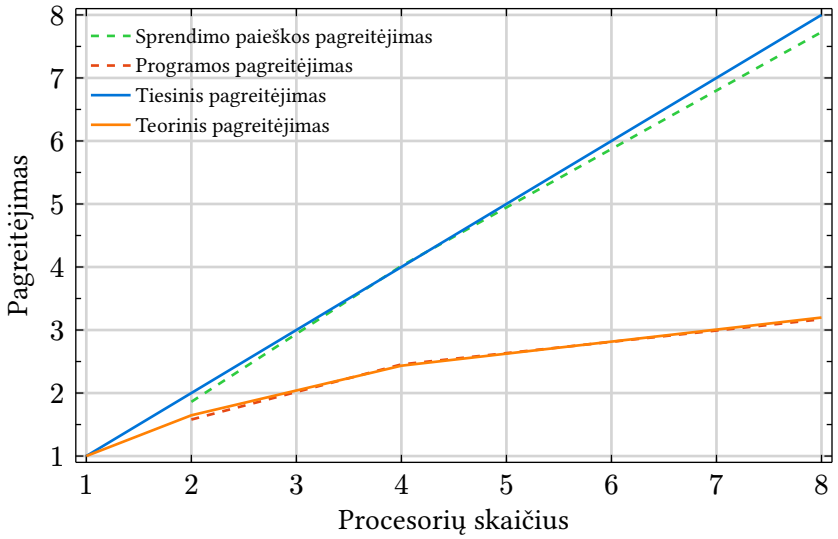

Fig. 5: Pagreitėjimo ir Procesorių skaičiaus santykis

Šis metodas pasieka teorinį pagreitėjimą be papildomo proceso.

Antra dalis

Užduotis

Sulygiagretinti matricos skaičiavimus.

Lygiagretinimas

Tam, kad gerai sulygiagretinti, reikia paskirstyti skaičiavimus taip, kad kiekvienas procesas tūrėtų po tiek pat darbo. Tam panaudojama funkcija `lengths` (Jos apibrėžimas - Pav. 8), kuri parenka tinkamus intervalus, taip, kad kiekvienas procesas paskaičiuotu apytiksliai tiek pat matricos elementų.

```c
int main() {
   // ...
   int *lens = lengths(numDP, world_size);
   int *counts = calloc(sizeof(int), world_size);
   for (int ix = 0; ix < world_size; ++ix) {
      counts[ix] = (lens[ix + 1] - lens[ix]) * numDP;
   }

   int *disps = calloc(sizeof(int), world_size);
   for (int ix = 0; ix < world_size; ++ix) {
      disps[ix] = lens[ix] * numDP;
   }

   distanceMatrix = calloc(sizeof(double), numDP * numDP);
   for (int i = lens[world_rank]; i < lens[world_rank + 1]; i++) {
      for (int j = 0; j <= i; j++) {
         distanceMatrix[numDP * i + j] =
                     HaversineDistance4(demandPoints[i][0], demandPoints[i][1],
demandPoints[j][0], demandPoints[j][1]);
      }
   }
   // ...
```

Čia svarbu paminėti, kad `distanceMatrix` tipas buvo pakeistas iš `double **` į `double *` ir atmintis visai matricai priskiriamia iškarto `calloc(sizeof(double), numDP * numDP)`, prieeiga prie patricos irgi atitinkamai pakeista iš `distanceMatrix[i][j] = v` į `distanceMatrix[i * numDP + j] = v`. Kadangi šis pakeitimas iš esmės pakeičia greitaveikos savybės (eksperimentiškai visos programos skaičiavimas sumažėja nuo 22s iki 17s), tai pradinė programa, su kuria lygininama lygiagretinta programa, irgi buvo pakeista, tam kad palyginimai būtų teisingi.

Kai procesas baigia savo dalį, jis nusiunčia nusiunčia kitiem procesam savo dalį ir laukia kitų dalių naudojant `MPI_Allgatherv`.

```c
    MPI_Allgatherv(
        distanceMatrix + disps[world_rank],
        counts[world_rank],
        MPI_DOUBLE,
        distanceMatrix,
        counts,
        disps,
        MPI_DOUBLE,
        MPI_COMM_WORLD
    )
```

Galima pakeisti skaičiuoti ir naudojant `MPI_Iallgatherv`, pridedant prieš kiekvieną `evaluateSolution` (nes tik ten ir naudojami matricos duomenys) šią eilutę, tam, kad duomenys pirma būtų pilnai surinkti.

```c
if (first_run) { MPI_Wait(&req, MPI_STATUS_IGNORE); first_run = false; }
```

Rezultatai

TODO: actual data

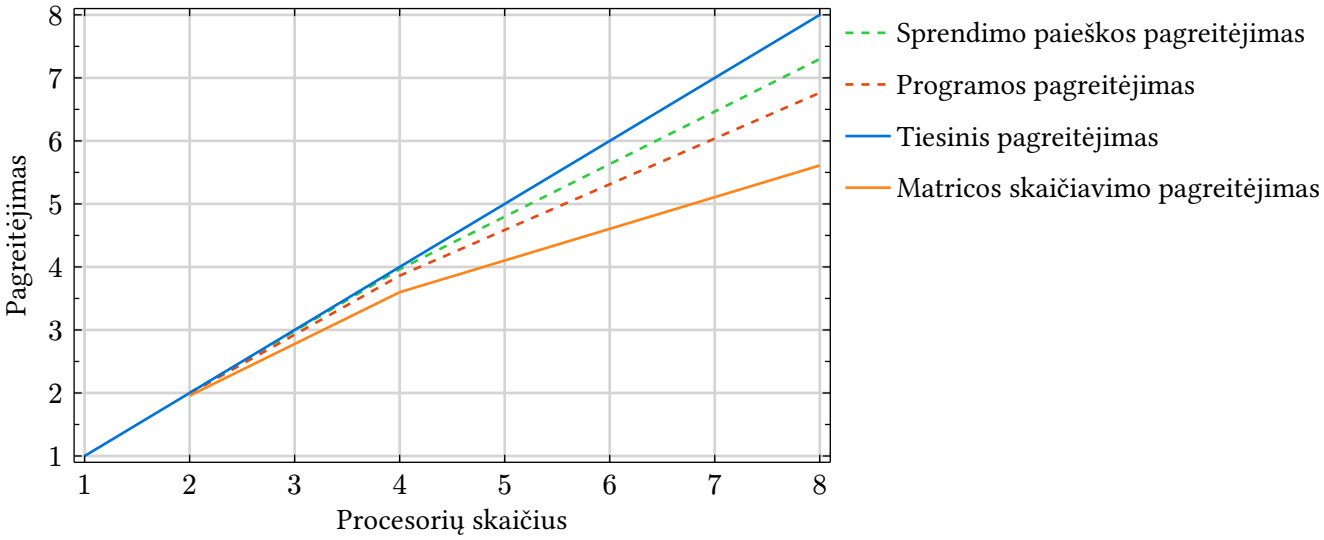

Fig. 6: Pagreitėjimo ir Procesorių skaičiaus santykis, kai matricos skaičiavimas sulygiagretintas (`MPI_Allgather`)

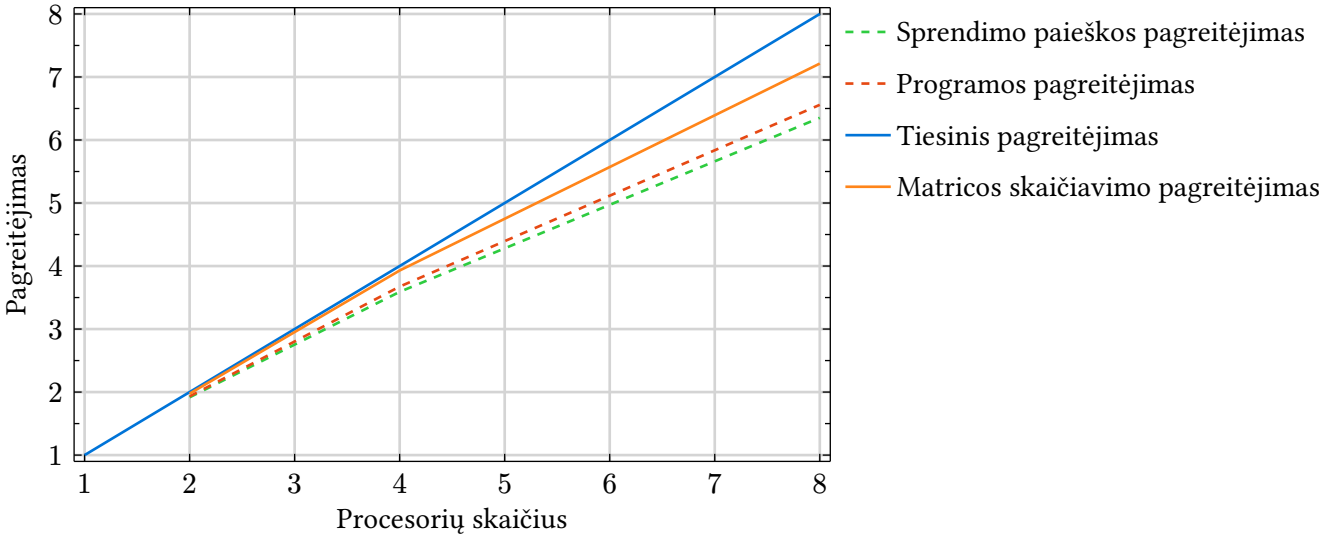

Fig. 7: Pagreitėjimo ir Procesorių skaičiaus santykis, kai matricos skaičiavimas sulygiagretintas (`MPI_Iallgather`)

Priedai

```c
int WRP_Check_for(int source, int tag, MPI_Comm comm) {
   MPI_Status status;
   int flag;
   MPI_Iprobe(source, tag, comm, &flag, &status);

   return flag;
}
```

Pav. 7: Optimalus intervalus parinkimas

```c
 int *lengths(int leg_length, int process_count) {
     int area = leg_length * leg_length / 2;

     int small_area = area / process_count;
     int *lenghts = calloc(sizeof(int), process_count + 1);
     for (int ix = 1; ix < process_count; ++ix) {
         lenghts[ix] = (int) sqrt(2 * ( ix ) * small_area);
     }
     lenghts[process_count] = leg_length;

     return lenghts;
 }
```

Pav. 8: Optimalus intervalus parinkimas