Vilniaus universitetas
Matematikos ir informatikos fakultetas
Programų sistemų katedra

Lygiagrečiojo programavimo
Laboratorinio darbo #2 ataskaita

Autorius(-iai): 3 kursas, 3 grupė
Greta Mameniškytė

Vilnius, 2017

Užduotis

1. Parsisiųskite programos kodą (lab_03_src.cpp). Programa generuoja 256 eilučių atsitiktinių skaičių matricą, kurios eilutės suskirstytos blokais po 64. Pirmojo bloko eilutėse generuojama po 2500, antrojo bloko eilutėse – po 5000, trečiojo bloko eilutės – po 7500 ir paskutiniojo bloko eilutėse – po 10000 atsitiktinių skaičių. Kiekvienos eilutės elementai surikiuojami didėjimo tvarka.
2. Papildykite programą kiekvienos surikiuotos eilutės medianos skaičiavimu.
3. Sudarykite lygiagrečią paskirstytos atminties(MPI) programos versiją. Pagrindinis procesas(master) turi padalinti kiekvieną eilučių bloką visiems procesams (įskaitant ir save) ir surinkti paskaičiuotas kiekvienos eilutės medianas. Procesai-darbininkai (slaves) priima eilučių bloką, surikiuoja kiekvieną eilutę, paskaičiuoja jos elementų medianą ir siunčia medianą pagrindiniam procesui. Procedūra taikoma kiekvienam eilučių blokui (4 kartus). Pagal galimybes, naudokite MPI kolektyvinės komunikacijos šablonus MPI_Scatter, MPI_Gather ir kt.
4. Vykdykite skaičiavimus naudodami 1, 2 ir 4 procesus, fiksuodami lygiagrečiojo algoritmo pagreitėjimą.
5. Ataskaitoje pateikite:
   - užduoties sąlygą;
   - skaičiavimų laiko vidurkių priklausomybės nuo procesų skaičiaus grafiką (naudojant 1, 2 ir 4 procesus);
   - lygiagrečiojo algoritmo pagreitėjimo priklausomybės nuo procesų skaičiaus grafiką (naudojant 1, 2 ir 4 procesus);
   - MPI programos kodą.

Skaičiuodami algoritmo pagreitėjimą laikykite kad $T_0 = T_1$, t.y., kad nuosekliojo algoritmo vykdymo laikas sutampa su lygiagrečiojo algoritmo vykdymo laiku naudojant vieną procesą.

Gauti rezultatai

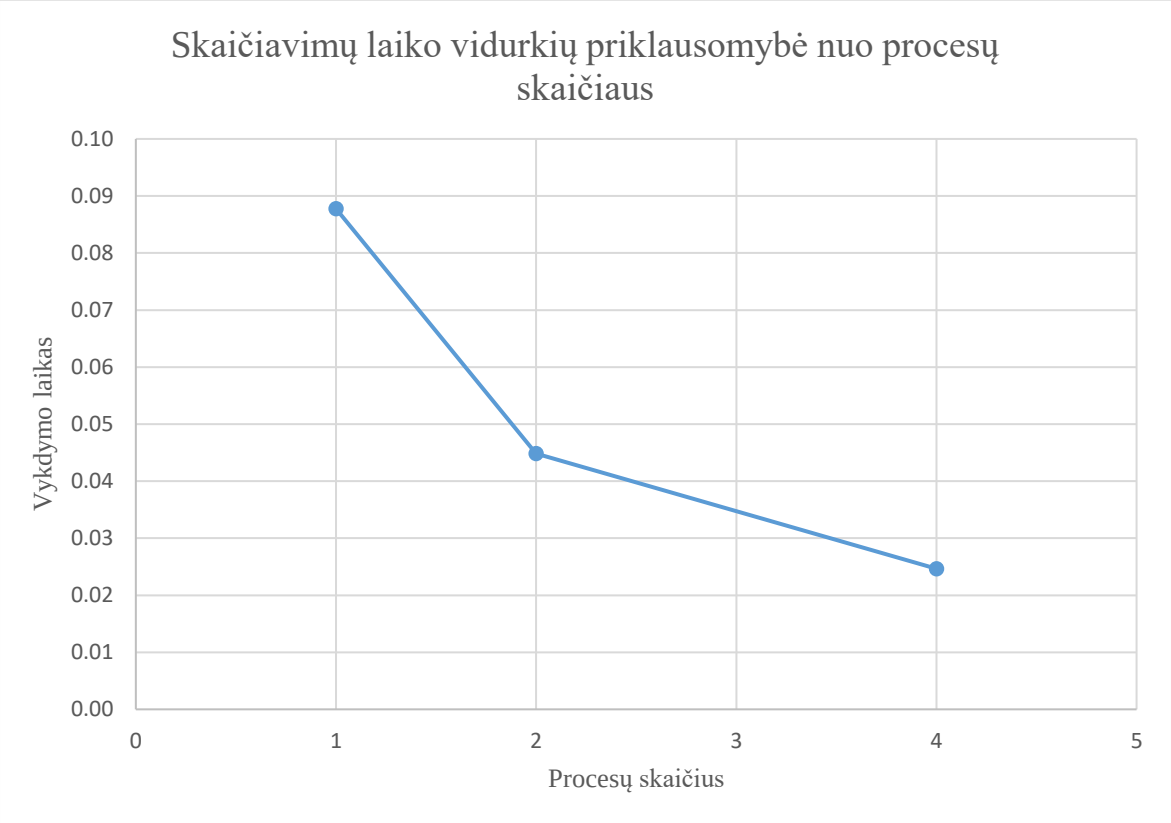
Originalus programos kodas buvo modifikuotas NxM matricos matmenis pakeitus iš 16x20 į 256x200 matmenų matricą, norint pastebėti reikšmingesnį skirtumą vykdant programą su skirtingu procesų skaičiumi.

Visi žemiau aprašyti bandymai buvo atlikti nuotoliniu būdu prisijungus prie VU MIF Linux kompiuterio, turinčio 12 branduolių. Kiekvienas bandymas buvo atliekamas 3 kartus ir po to buvo skaičiuojami gautų rezultatų vidurkiai.

Žemiau esančioje 1 lentelėje ir 1 grafike pavaizduota bandymų skaičiavimo laikų rezultatų bei vidurkių priklausomybė nuo procesų skaičiaus.

1 lentelė **Programos vykdymo su skirtingu procesorių skaičiumi bandymų rezultatai**

Procesų skaičius	1 bandymas	2 bandymas	3 bandymas	Vidurkis
1	0.087634	0.087777	0.087885	0.087765
2	0.045025	0.044804	0.044606	0.044812
4	0.024342	0.024807	0.024808	0.024652

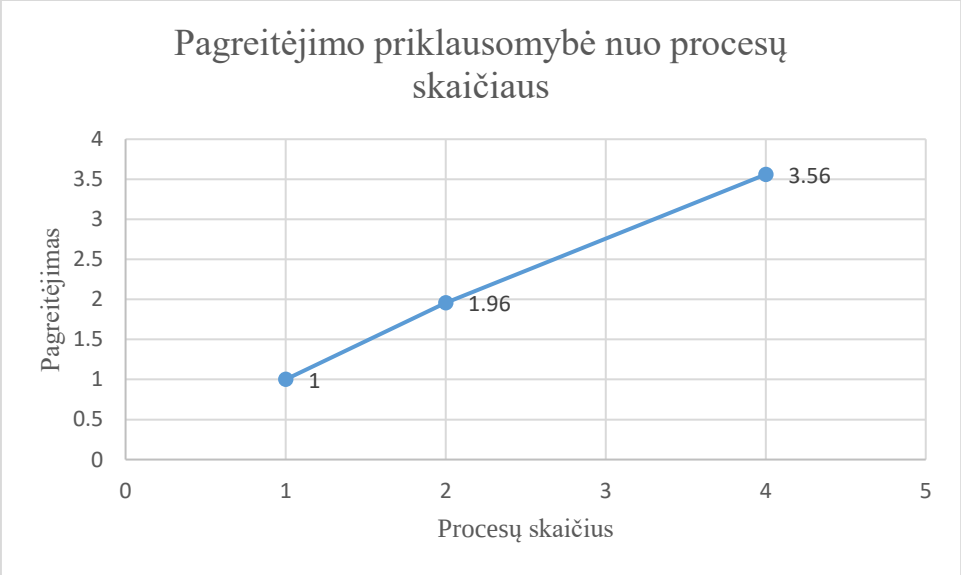


1 grafikas **Skaičiavimų laiko vidurkių priklausomybė nuo procesų skaičiaus**

2 grafike bei 2 lentelėje vaizduojamas algoritmo pagreitėjimas, kuris skaičiuojamas pagal formulę: $S_p = \frac{T_0}{T_p}$, kur T_0 - nuosekliojo algoritmo vykdymo trukmė, T_p – lygiagrečiojo algoritmo vykdymo trukmė, naudojant p procesorių. Šiame uždavinyje T_0 laikomas T_1.

lentelė 2 **Algoritmo pagreitėjimo ir procesų skaičiaus sąryšis**

Procesų skaičius	**Pagreitėjimas**
1	1
2	1.958538
4	3.560123



2 grafikas **Pagreitėjimo priklausomybė nuo procesų skaičiaus**

Išvados

Didinant procesų skaičių lygiagrečiosios algoritmo dalies vykdymo laikas sparčiai mažėja, o pagreitėjimas auga beveik tiesiškai.

Priedai

1. MPI programos kodas

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <mpi.h>
#include <time.h>

void genMatrix(int *A, int N, int M) {
 // Clean matrix
    for (int i=0; i<N*M; i++) {
       A[i] = 0;
 }
    // Generate matrix
    int m = M/4;
    for (int i=0; i<N; i++) {
       for (int j=0; j<m; j++) A[i*M+j] =
(int)((double)rand()/RAND_MAX*99) + 1;
       if (i > 0 && (i+1) % (N/4) == 0) m += M/4;
    }
}

void sortAndFindMedian(int M, int N,int *A,float *medians){
 int t, n;
 for (int k=0; k<N; k++) {
       n = 0;
       while (A[k*M+n] != 0 && n < M) n++;
       //sort row
       for (int i=0; i<n-1; i++) {
             for (int j=0; j<n-1; j++) {
                   if (A[k*M+j] > A[k*M+j+1]) {
                         t = A[k*M+j];
                         A[k*M+j] = A[k*M+j+1];
                         A[k*M+j+1] = t;
                   }
             }
       }
       //find row median
       if (n%2 != 0) {
```

```
                medians[k]=static_cast<float>(A[k*M +n/2]);
            }
            else {
                medians[k] =(static_cast<float>(A[k*M+(n/2-
1)]+A[k*M+(n/2)]))/static_cast<float>(2);
            }
      }
     }

     int main(int argc, char** argv) {
         srand(time(NULL));
         int N = 256; //eiluciu_sk
         int M = 200; //stulpeliu_sk
      int nBlocks = 4;
      int *globalA = new int[N*M];
         float *globalMedians = new float[N];
      int nProcs, rank;
      double mpi_startTime, mpi_endTime;
      for(int i=0;i<N;i++){
            globalMedians[i]=0;
      }
         genMatrix(globalA, N, M);
      MPI_Init(&argc, &argv);
      MPI_Comm_size(MPI_COMM_WORLD, &nProcs);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Barrier(MPI_COMM_WORLD);
      mpi_startTime = MPI_Wtime();
      int *localA = new int[N*M/nBlocks/nProcs];
      float *localMedians = new float[N/nBlocks/nProcs];
      for(int i = 0; i< nBlocks; i++){
            MPI_Scatter(globalA+i*M*N/nBlocks, N*M/nBlocks/nProcs, MPI_INT,
localA, N*M/nBlocks/nProcs, MPI_INT, 0, MPI_COMM_WORLD);
            sortAndFindMedian(M, N/nBlocks/nProcs, localA, localMedians);
            MPI_Gather(localMedians, N/nBlocks/nProcs, MPI_FLOAT,
globalMedians+i*N/nBlocks, N/nBlocks/nProcs, MPI_FLOAT, 0, MPI_COMM_WORLD);
            MPI_Gather(localA, N*M/nBlocks/nProcs, MPI_INT,
globalA+i*N*M/nBlocks, N*M/nBlocks/nProcs, MPI_INT, 0, MPI_COMM_WORLD);
      }
      MPI_Barrier(MPI_COMM_WORLD);
      mpi_endTime = MPI_Wtime();
      if (rank == 0) {
```

```
            //print matrix
            printf("Array after sorting rows\n");
            for (int i=0; i<N; i++) {
                  for (int j=0; j<M; j++) {
                        printf("%3d", globalA[i*M+j]);
                  }
                  printf("\n");
            }

            for(int i; i<N; i++){
                  printf("Row %d median value is %.1f\n", i,
globalMedians[i]);
            }

            printf("MPI time: %f\n", mpi_endTime-mpi_startTime);
      }
      MPI_Finalize();
     }
```