Vilniaus universitetas
Matematikos ir informatikos fakultetas
Programų sistemų katedra

Lygiagrečiojo programavimo

Laboratorinio darbo #2 ataskaita

Autoriai: Monika Gasparaitė, 3k. 4 gr.
Mantas Petrikas, 3k. 5gr.

Vilnius, 2017

Užduotis

1. Sudaryti nuoseklųjį algoritmą, kuris įvertintų apytikslę π reikšmę pagal aukščiau aprašytą teorinį modelį. Siekiant išvengti didelių skaičių, naudokite ciklo cikle struktūrą. Pavyzdžiui, norint sugeneruoti 1 mln. taškų, išorinis ciklas vykdomas 100 kartų, o vidinis – 10000. Generuojamų taškų skaičių parinkite taip, kad nuosekliojo algoritmo vykdymo laikas būtų daugiau 5 sekundžių.
2. Išlygiagretinkite sudarytą algoritmą vidinio ciklo iteracijas paskirstydami gijoms. Po kiekvienos išorinio ciklo iteracijos, pagrindinis procesorius (master) turi atspausdinti turimą π skaičiaus aproksimaciją.
3. Skaičių generavimui naudokite funkciją rand() ir stebėkite algoritmo pagreitėjimą didinant gijų skaičių.
4. Funkciją rand() pakeiskite funkcija randr(unignedint*seed), kur seed – pradinė atsitiktinių skaičių sekos reikšmė, kuri kiekvienai gijai turi būti skirtinga.

Gauti rezultatai

Norint įvertinti pagreitėjimą nuoseklūs ir lygiagretūs algoritmai buvo po tris kartus leidžiami prisijungus prie MIF Linux serverio kompiuterio, turinčio 12 branduolių. Paleidžiant lygiagrečius algoritmus buvo keičiamas naudojamų branduolių skaičius.

Algoritmas vykdymo metu sugeneruoja 200 000 000 taškų (išorinis ciklas vykdomas 10 kartų, vidinis - 20 000 000 kartų).

1 lentelėje pateikiamos nuoseklaus algoritmo (žr. 1 priedą), naudojančio funkciją rand, vykdymo trukmės.

1 lentelė. Nuoseklaus algoritmo vykdymo laiko įverčiai naudojant rand funkciją

Vykdymo laikas sekundėmis			
1 bandymas	**2 bandymas**	**3 bandymas**	**Vidutinis**
13,131	12,285	12,127	12,551

2 lentelėje pateikiamos išlygiagretinto algoritmo (žr. 2 priedą), naudojančio funkciją rand, vykdymo trukmės ir praktiniai pagreitėjimai, apskaičiuoti pagal formulę:

$$S_p = \frac{T_0}{T_p} \ ,$$

kur T_0 – nuosekliojo algoritmo vykdymo laikas, T_p – lygiagretaus algoritmo vykdymo laikas naudojant p procesorių.

2 lentelė. Lygiagretaus algoritmo vykdymo laiko įverčiai naudojant rand funkciją

	Vykdymo laikas sekundėmis				
Procesorių skaičius	**1 bandymas**	**2 bandymas**	**3 bandymas**	**Vidutinis**	**Pagreitėjimas**
1	14,418	19,972	13,019	13,469	0,931
2	92,577	88,206	73,534	84,772	0,148
4	331,604	367,49	365,555	354,883	0,035

Pakeitus funkciją rand į rand_r ir kiekvienai gijai suteikus skirtingą pradinę kintamojo seed reikšmę, buvo atliekami pakartotiniai bandymai.

Nuoseklaus algoritmo (žr. 3 priedą), naudojančio funkciją rand_r, vykdymo trukmės pateikiamos 3 lentėje.

3 lentelė. Nuoseklaus algoritmo vykdymo laiko įverčiai naudojant rand_r funkciją

Vykdymo laikas sekundėmis			
1 bandymas	**2 bandymas**	**3 bandymas**	**Vidutinis**
10,126	10,823	11,088	10,679

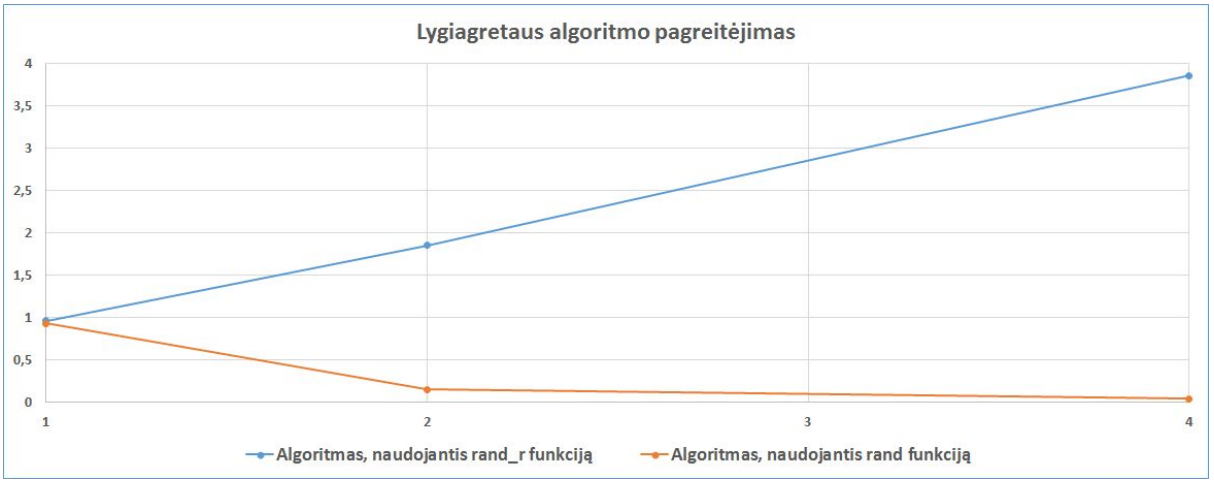
Išlygiagretinto algoritmo (žr. 4 priedą) naudojančio funkciją rand_r vykdymo trukmės ir praktiniai pagreitėjimai pateikimi 4 lentelėje.

4 lentelė. Lygiagretaus algoritmo vykdymo laiko įverčiai naudojant rand_r funkciją

	Vykdymo laikas sekundėmis				
Procesorių skaičius	**1 bandymas**	**2 bandymas**	**3 bandymas**	**Vidutinis**	**Pagreitėjimas**
1	10,448	11,119	11,805	11,124	0,959
2	5,972	6,015	5,313	5,766	1,851
4	2,713	2,664	2,93	2,769	3,856

Lygiagrečių algoritmų, naudojančių rand ir rand_r funkcijas bei vykdomų su skirtingais procesorių kiekiais, pagreitėjimo palyginimas pateikiamas 1 grafike.

1 grafikas. Lygiagretaus algoritmo pagreitėjimas naudojant rand ir rand_r funkcijas



Išvados

Nuoseklusis algoritmas, naudojantis funkciją rand, yra vykdomas ilgiau nei nuoseklusis algoritmas, naudojantis funkciją rand_r. Beje, nuosekliųjų algoritmų atitinkamos vykdymo trukmės yra mažesnės už lygiagrečiųjų algoritmų vykdymo trukmes naudojant 1 procesorių.
Išlygiagretinto algoritmo, naudojančio rand funkciją, praktinis pagreitėjimas mažėja didinant procesorių skaičių, tuo tarpu naudojant rand_r funkciją - didėja. Palyginus išlygiagretintų algoritmų, naudojančių rand bei rand_r funkcijas, vykdymo trukmes ir pagreitėjimus, galima daryti išvadą, jog funkcija rand nėra skirta naudoti lygiagrečiam kode, nes ji lėtina algoritmo darbą didinant procesorių skaičių, o pakeitus rand funkciją į rand_r ir kiekvienai gijai suteikus skirtingą parametro *seed* pradinę reikšmę - algoritmo vykdymo trukmė mažėja.

Priedai

1 Priedas. Nuosekliojo algoritmo kodas naudojantis funkciją rand().

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

double getRandomDoubleWithoutSeed() {
  return (double)rand()/RAND_MAX;
}

double GetTime() {
  struct timeval laikas;
  gettimeofday(&laikas, NULL);
  double rez = (double)laikas.tv_sec+(double)laikas.tv_usec/1000000;
  return rez;
}

int calculatePi (int outerLoop, int innerLoop) {
  int inCircle = 0;
  int total = 0;
  for (int fr1 = 0; fr1 < outerLoop; fr1++) {
    for (int fr2 = 0; fr2 < innerLoop; fr2++) {
      double x = getRandomDoubleWithoutSeed();
      double y = getRandomDoubleWithoutSeed();
        if (pow(x,2) + pow(y,2) < 1) {
            inCircle++;
        }
        total++;
      }
    printf("pi %f\n", ((double)inCircle / total) * 4);
  }
}
```

```
int main(int argc, char *argv[]){
   srand(time(NULL));
   double startTime = GetTime();
   calculatePi(10, 20000000);
   double endTime = GetTime();
   printf("time: %.3f\n", endTime - startTime);
}
```

2 priedas. Lygiagrečiojo algoritmo kodas naudojantis funkcija rand().

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <omp.h>
#include <math.h>
#include <sstream>

double getRandomDoubleWithoutSeed() {
  return (double)rand()/RAND_MAX;
}

double GetTime() {
  struct timeval laikas;
  gettimeofday(&laikas, NULL);
  double rez = (double)laikas.tv_sec+(double)laikas.tv_usec/1000000;
  return rez;
}

int calculatePi (int outerLoop, int innerLoop) {
  int inCircle = 0;
  int total = 0;
  #pragma omp parallel
  {
    unsigned int seed = (unsigned int) omp_get_thread_num();
    for (int fr1 = 0; fr1 < outerLoop; fr1++) {
      #pragma omp parallel for reduction (+:inCircle, total)
      for (int fr2 = 0; fr2 < innerLoop; fr2++) {

        double x = getRandomDoubleWithoutSeed();
        double y = getRandomDoubleWithoutSeed();
        if (pow(x,2) + pow(y,2) < 1) {
            inCircle++;
```

```
            }
            total++;
        }
        #pragma omp master
        {
            printf("pi %f\n", ((double)inCircle / total) * 4);
        }
    }
  }
}
int main(int argc, char *argv[]){
  if (argc > 1) {
    // thread count value;
    int threadarg;
    std::istringstream ss(argv[1]);
    if (!(ss >> threadarg)) {
      printf("Invalid argument: Invalid number %s\n", argv[1]);
      return -1;
    }
    omp_set_num_threads(threadarg);
  }
  srand(time(NULL));
  double startTime = GetTime();
  calculatePi(10, 20000000);
  double endTime = GetTime();
  printf("time: %.3f\n", endTime - startTime);
}
```

3 priedas. Nuosekliojo algoritmo kodas naudojantis funkcija rand_r(unsigned int *seed).

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <omp.h>
#include <math.h>
#include <sstream>

double getRandomDouble(unsigned int *seed) {
  return (double)rand_r(seed)/RAND_MAX;
}

double GetTime() {
  struct timeval laikas;
  gettimeofday(&laikas, NULL);
  double rez = (double)laikas.tv_sec+(double)laikas.tv_usec/1000000;
  return rez;
}

int calculatePi (int outerLoop, int innerLoop, int seedRandom) {
  int inCircle = 0;
  int total = 0;
  #pragma omp parallel
  {
    unsigned int seed = (unsigned int) omp_get_thread_num() * seedRandom;
    for (int fr1 = 0; fr1 < outerLoop; fr1++) {
      #pragma omp for reduction (+:inCircle, total)
      for (int fr2 = 0; fr2 < innerLoop; fr2++) {
        double x = getRandomDouble(&seed);
        double y = getRandomDouble(&seed);
        if (pow(x,2) + pow(y,2) < 1) {
            inCircle++;
```

```
            }
            total++;
        }
        #pragma omp master
        {
            printf("pi %f\n", ((double)inCircle / total) * 4);
        }
    }
  }
}
int main(int argc, char *argv[]){
  if (argc > 1) {
    // thread count value;
    int threadarg;
    std::istringstream ss(argv[1]);
    if (!(ss >> threadarg)) {
      printf("Invalid argument: Invalid number %s\n", argv[1]);
      return -1;
    }
    omp_set_num_threads(threadarg);
  }
  struct timeval laikas;
  gettimeofday(&laikas, NULL);
  double startTime = GetTime();
  calculatePi(10, 20000000, (int) laikas.tv_usec);
  double endTime = GetTime();
  printf("time: %.3f\n", endTime - startTime);
}
```

4 priedas. Lygiagrečiojo algoritmo kodas naudojantis funkcija rand_r(*seed).

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <omp.h>
#include <math.h>
#include <sstream>

double getRandomDouble(unsigned int *seed) {
  return (double)rand_r(seed)/RAND_MAX;
}

double GetTime() {
  struct timeval laikas;
  gettimeofday(&laikas, NULL);
  double rez = (double)laikas.tv_sec+(double)laikas.tv_usec/1000000;
  return rez;
}

int calculatePi (int outerLoop, int innerLoop, int seedRandom) {
  int inCircle = 0;
  int total = 0;
  #pragma omp parallel
  {
    unsigned int seed = (unsigned int) omp_get_thread_num() * seedRandom;
    for (int fr1 = 0; fr1 < outerLoop; fr1++) {
      #pragma omp for reduction (+:inCircle, total)
      for (int fr2 = 0; fr2 < innerLoop; fr2++) {
        double x = getRandomDouble(&seed);
        double y = getRandomDouble(&seed);
        if (pow(x,2) + pow(y,2) < 1) {
            inCircle++;
        }
```

```
            total++;
        }
        #pragma omp master
        {
            printf("pi %f\n", ((double)inCircle / total) * 4);
        }
    }
  }
}
int main(int argc, char *argv[]){
  if (argc > 1) {
    // thread count value;
    int threadarg;
    std::istringstream ss(argv[1]);
    if (!(ss >> threadarg)) {
      printf("Invalid argument: Invalid number %s\n", argv[1]);
      return -1;
    }
    omp_set_num_threads(threadarg);
  }
  struct timeval laikas;
  gettimeofday(&laikas, NULL);
  double startTime = GetTime();
  calculatePi(10, 20000000, (int) laikas.tv_usec);
  double endTime = GetTime();
  printf("time: %.3f\n", endTime - startTime);
}
```