UNIVERZITET U BEOGRADU - ELEKTROTEHNIČKI FAKULTET

MULTIPROCESORKI SISTEMI (13S114MUPS, 13E114MUPS)

DOMAĆI ZADATAK 1 – OPENMP

Izveštaj o urađenom domaćem zadatku

Predmetni saradnici:

dipl. ing. Matija Dodović

Studenti:

Ljubica Majstorović 2020/0253

Pavle Šarenac 2020/0359

Beograd, novembar 2023.

SADRŽAJ

SADRŽAJ **2**

1. PROBLEM 1 – IZRAČUNAVANJE ARITMETIČKIH BROJEVA KORIŠĆENJEM DIREKTIVE ZA PODELU POSLA (FOR DIREKTIVE) **4**

1.1. TEKST PROBLEMA 4
1.2. DELOVI KOJE TREBA PARALELIZOVATI 4
1.2.1. Diskusija *4*
1.2.2. Način paralelizacije *6*
1.3. REZULTATI 7
1.3.1. Logovi izvršavanja *7*
1.3.2. Grafici ubrzanja *9*
1.3.3. Diskusija dobijenih rezultata *10*

2. PROBLEM 2 – IZRAČUNAVANJE ARITMETIČKIH BROJEVA KORIŠĆENJEM KONCEPTA POSLOVA (TASKS) **12**

2.1. TEKST PROBLEMA 12
2.2. DELOVI KOJE TREBA PARALELIZOVATI 12
2.2.1. Diskusija *12*
2.2.2. Način paralelizacije *12*
2.3. REZULTATI 13
2.3.1. Logovi izvršavanja *13*
2.3.2. Grafici ubrzanja *15*
2.3.3. Diskusija dobijenih rezultata *16*

3. PROBLEM 3 – GENERISANJE ELEMENATA HALTON QUASI MONTE CARLO SEKVENCE BEZ DIREKTIVA ZA PODELU POSLA (WORKSHARING DIREKTIVA) **18**

3.1. TEKST PROBLEMA 18
3.2. DELOVI KOJE TREBA PARALELIZOVATI 18
3.2.1. Diskusija *18*
3.2.2. Način paralelizacije *22*
3.3. REZULTATI 22
3.3.1. Logovi izvršavanja *22*
3.3.2. Grafici ubrzanja *23*
3.3.3. Diskusija dobijenih rezultata *25*

4. PROBLEM 4 – GENERISANJE ELEMENATA HALTON QUASI MONTE CARLO SEKVENCE SA DIREKTIVAMA ZA PODELU POSLA (WORKSHARING DIREKTIVAMA) **26**

4.1. TEKST PROBLEMA, ... 26
4.2. DELOVI KOJE TREBA PARALELIZOVATI 26
4.2.1. Diskusija *26*
4.2.2. Način paralelizacije *26*
4.3. REZULTATI 27
4.3.1. Logovi izvršavanja *27*
4.3.2. Grafici ubrzanja *27*
4.3.3. Diskusija dobijenih rezultata *29*

5. PROBLEM 5 – SIMULACIJA KRETANJA N TELA (N-BODY PROBLEM) **30**

5.1. TEKST PROBLEMA 30
5.2. DELOVI KOJE TREBA PARALELIZOVATI 30

5.2.1. Diskusija *30*
5.2.2. Način paralelizacije *33*
5.3. REZULTATI 33
5.3.1. Logovi izvršavanja *33*
5.3.2. Grafici ubrzanja *35*
5.3.3. Diskusija dobijenih rezultata *36*

1. Problem 1 – Izračunavanje aritmetičkih brojeva korišćenjem direktive za podelu posla (for direktive)

1.1. Tekst problema

Paralelizovati program koji vrši izračunavanje aritmetičkih brojeva. Pozitivan ceo broj je aritmetički ako je prosek njegovih pozitivnih delilaca takođe ceo broj. Program se nalazi u datoteci aritmetic.c u arhivi koja je priložena uz ovaj dokument. Obratiti pažnju na ispravno deklarisanje svih promenljivih prilikom paralelizacije. Program testirati sa parametrima koji su dati u run skripti.

1.2. Delovi koje treba paralelizovati

1.2.1. Diskusija

Segment koda nad kojim je izvršena paralelizacija u ovom problemu je sledeći:

```
for (n = 1; arithmetic_count <= num; ++n)
{
    unsigned int divisor_count;
    unsigned int divisor_sum;
    divisor_count_and_sum(n, &divisor_count, &divisor_sum);
    if (divisor_sum % divisor_count != 0)
        continue;
    ++arithmetic_count;
    if (divisor_count > 2)
        ++composite_count;
}
```

Ova petlja je vrlo pogodna za paralelizaciju zato što ne postoji nikakva zavisnost po podacima između njenih iteracija, pa one mogu da se izvršavaju potpuno nezavisno.

Jedina nezgodna stvar je što petlja nije u zadovoljavajućoj formi koju for direktiva zahteva, pa je potrebno malo preurediti ovaj kod kako bi bila moguća takva paralelizacija.

Još jedan deo koda koji troši značajan deo procesorskog vremena:

```
for (unsigned int p = 3; p * p <= n; p += 2)
{
    unsigned int count = 1, sum = 1;
    for (power = p; n % p == 0; power *= p, n /= p)
    {
        ++count;
        sum += power;
    }
    divisor_count *= count;
    divisor_sum *= sum;
}
```

Zaključili smo da nije dobro pokušati paralelizaciju ove petlje iz više razloga:

1) Ona se izvršava u telu funkcije divisor_count_and_sum koja se poziva u svakoj iteraciji for petlje u okviru koje se poziva ova funkcija, pri čemu ova petlja može imati veoma veliki broj iteracija i zato bismo u tom slučaju isto toliko puta pravili paralelni region, što bi bili ogromni režijski troškovi i diskutabilno je da li bi se uopšte postiglo ubrzanje u odnosu na sekvencijalni kod u tom slučaju.
2) Postoji zavisnost po podacima između iteracija spoljašnje petlje zato što se „n“ koje je u uslovu spoljašnje petlje menja u unutrašnjoj petlji, pa zato ovako napisan algoritam nije pogodan za paralelizaciju. Da bi paralelizacija potencijalno bila moguća, bilo bi potrebno kompletno restrukturiranje ovog dela algoritma.

1.2.2. Način paralelizacije

```
    while (arithmetic_count <= num)
    {
        number_of_iterations = num + 1 - arithmetic_count;
        n += number_of_iterations;
#pragma omp parallel for default(none) private(i) shared(start, number_of_iterations)             \
    reduction(+ : arithmetic_count) reduction(+ : composite_count) num_threads(number_of_threads) \
    schedule(guided)
        for (i = start; i < start + number_of_iterations; i++)
        {
            unsigned int divisor_count;
            unsigned int divisor_sum;
            divisor_count_and_sum(i, &divisor_count, &divisor_sum);
            if (divisor_sum % divisor_count != 0)
                continue;
            ++arithmetic_count;
            if (divisor_count > 2)
                ++composite_count;
        }
        start += number_of_iterations;
    }
```

Zaključili smo da nije moguće našu glavnu for petlju paralelizovati pomoću for direktive zato što forma petlje nije odgovarajuća pošto nije poznat broj iteracija unapred.

Međutim, smislili smo način da ipak upotrebimo for direktivu. Rešenje leži u činjenici da ako želimo da nađemo npr. prvih 10 000 aritmetičkih brojeva, mi zasigurno moramo da imamo najmanje 10 000 iteracija petlje. Ne znamo unapred za koliko će broj iteracija petlje biti veći od 10 000, ali znamo da ih mora biti najmanje 10 000. Ovo možemo iskoristiti onda tako što ćemu napisati for petlju koja će imati 10 000 iteracija, i u njoj je onda poznat broj iteracija unapred pa se može paralelizovati pomoću for direktive. Uslov prvobitne for petlje ćemo izmestiti u novu spoljašnju while petlju. Recimo da se u 10 000 iteracija for petlje našlo 8 000 aritmetičkih brojeva. To znači da for petlja u narednoj iteraciji while petlje mora da se izvrši još najmanje 2 000 puta jer je toliko aritmetičkih brojeva preostalo da se pronađe. Ovaj postupak ponavljamo sve dok se ne pronađe željeni broj aritmetičkih brojeva. Na ovaj način smo u svakoj iteraciji spoljašnje while petlje menjali broj iteracija for petlje tako da se uvek znao unapred taj broj, što nam je omogućilo paralelizaciju for direktivom.

Naravno, uočili smo da promenljive arithmetic_count i composite_count modifikuju sve niti i da se one samo inkrementiraju. Zato smo ove promenljive deklarisali kao redukcione kako bi svaka nit imala svoju kopiju ovih promenljivih i menjala nju, a zatim da se na kraju sve te lokalne kopije saberu i dodaju na inicijalne vrednosti promenljivih.

Što se tiče raspodele posla između niti, zaključili smo da posao koji niti izvršavaju postaje sve teži kako se odmiče ka kasnijim iteracijama naše paralelizovane for petlje zato što je onda vrednost parametra n koji se prosleđuje funkciji divisor_count_and_sum sve veća, a samim tim će i

najzahtevnija for petlja u toj funkciji imati više iteracija pošto se to isto n nalazi u uslovu te petlje. Zato je očigledno da nije pogodna podrazumevana schedule(static) raspodela pošto balans opterećenja nije idealan. Najlogičnije nam je bilo da je za ovu situaciju najbolja schedule(guided) raspodela zato što ona upravo početnim nitima dodeljuje veći broj iteracija (što potpuno ima smisla raditi pošto su ranije iteracije manje zahtevne od kasnijih), a onda se broj iteracija koji se dodeljuje kasnijim nitima eksponencijalno smanjuje. Na taj način se ostvaruje bolji balans opterećenja. Isprobavanjem static, dynamic i guided raspodela sa raznim parametrima raspodele smo primetili da su rezultati praktično isti, tako da smo se odlučili za podrazumevanu schedule(guided) raspodelu bez ikakvog parametra jer nema značajne razlike u performansama.

1.3. Rezultati

U okviru ove sekcije su izloženi rezultati paralelizacije problema 1.

1.3.1. Logovi izvršavanja

```
Sequential implementation execution time: 0.002867s
Parallel implementation (one thread) execution time: 0.001502s
Test PASSED
Parallel implementation (two threads) execution time: 0.000306s
Test PASSED
Parallel implementation (four threads) execution time: 0.000207s
Test PASSED
Parallel implementation (eight threads) execution time: 0.000180s
Test PASSED
```

Izvršavanje komande ./arithmetic 10000

```
Sequential implementation execution time: 0.010243s
Parallel implementation (one thread) execution time: 0.008363s
Test PASSED
Parallel implementation (two threads) execution time: 0.004207s
Test PASSED
Parallel implementation (four threads) execution time: 0.002238s
Test PASSED
Parallel implementation (eight threads) execution time: 0.001501s
Test PASSED
```

Izvršavanje komande ./arithmetic 100000

```
Sequential implementation execution time: 0.172909s
Parallel implementation (one thread) execution time: 0.171841s
Test PASSED
Parallel implementation (two threads) execution time: 0.086988s
Test PASSED
Parallel implementation (four threads) execution time: 0.044931s
Test PASSED
Parallel implementation (eight threads) execution time: 0.028311s
Test PASSED
```

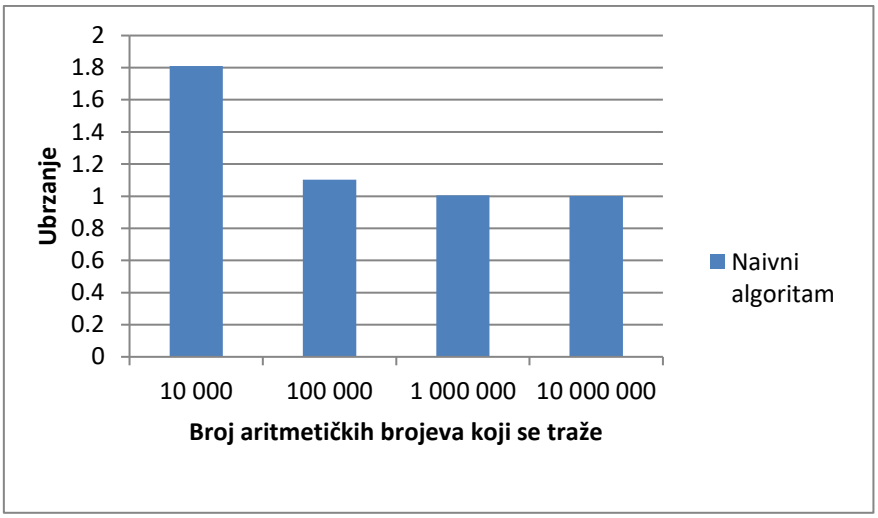
Izvršavanje komande ./arithmetic 1000000

```
Sequential implementation execution time: 4.199454s
Parallel implementation (one thread) execution time: 4.193153s
Test PASSED
Parallel implementation (two threads) execution time: 2.085609s
Test PASSED
Parallel implementation (four threads) execution time: 1.071954s
Test PASSED
Parallel implementation (eight threads) execution time: 0.571603s
Test PASSED
```

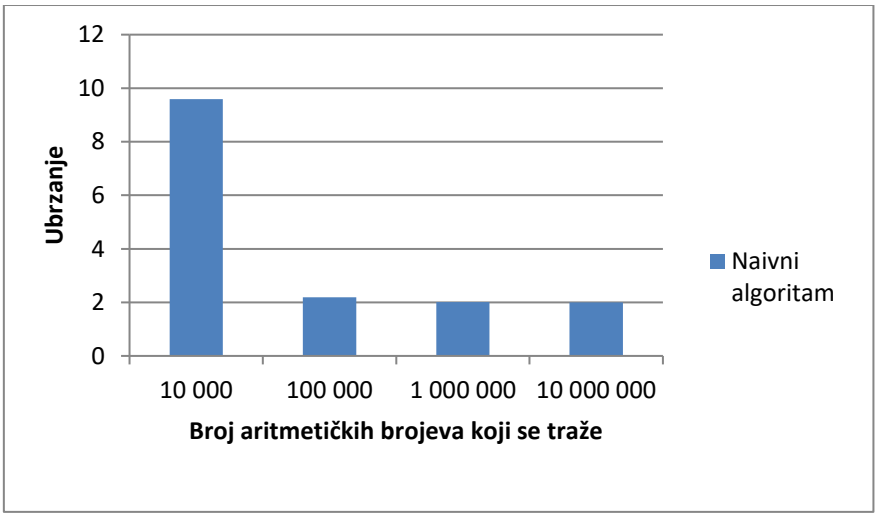
Izvršavanje komande ./arithmetic 10000000

1.3.2. Grafici ubrzanja

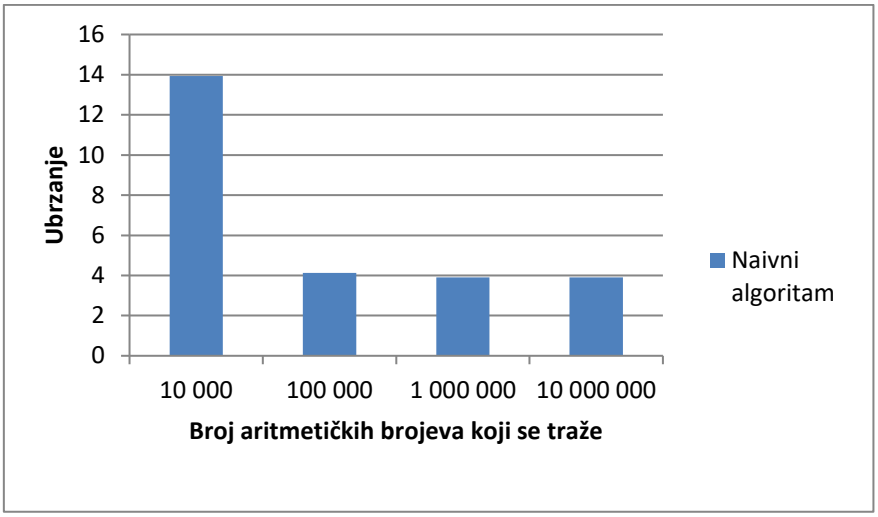
U okviru ove sekcije su dati grafici ubrzanja u odnosu na sekvencijalnu implementaciju.



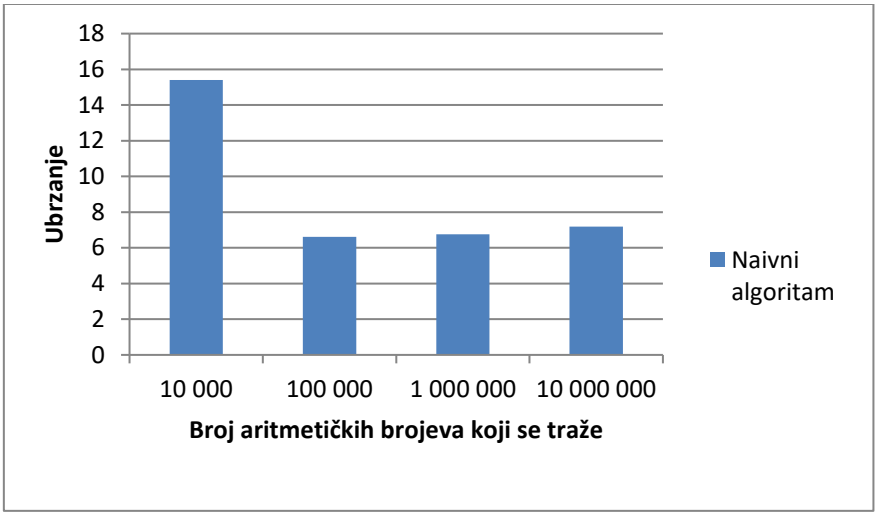
Slika 1. Grafik zavisnosti ubrzanja naivnog algoritma od broja aritmetičkih brojeva koji se traže za N = 1 niti



Slika 1. Grafik zavisnosti ubrzanja naivnog algoritma od broja aritmetičkih brojeva koji se traže za N = 2 niti



Slika 1. Grafik zavisnosti ubrzanja naivnog algoritma od broja aritmetičkih brojeva koji se traže za N = 4 niti



Slika 1. Grafik zavisnosti ubrzanja naivnog algoritma od broja aritmetičkih brojeva koji se traže za N = 8 niti

1.3.3. Diskusija dobijenih rezultata

Kako bi grafici i naši rezultati bili precizniji, zabeležili smo rezultate kako sekvencijalnog, tako i paralelnog izvršavanja programa za sve parametre i za svaki broj niti 5 puta i uzimali prosečna vremena.

Iz priloženih grafika je očigledno da za svaki ulazni parametar, tj. broj aritmetičkih brojeva koji se traže, dobijamo veće ubrzanje ukoliko povećamo broj niti koje rade u paraleli. Ovo je i očekivano upravo zato što su iteracije naše for petlje koju paralelizujemo potpuno nezavisne i nemamo gotovo nikakvu komunikaciju, a ni sinhronizaciju između niti pri paralelizaciji ove petlje. Jedina međusobna interakcija između niti jeste na kraju paralelnog regiona kada se redukcione

promenljive od svih niti saberu, ali osim toga, nikakve interakcije između niti nema. Zato se slobodno možemo usuditi da nazovemo ovaj problem *„embarrasingly parallel”*.

Međutim, vidi se da, iako se ubrzanje povećava kako povećavamo broj niti od jedan do osam za sve ulazne parametre, stepen povećanja ubrzanja se blago smanjuje – sa povećanjem sa jedne na dve niti, ubrzanje za parametar 10 000 se upetostručilo, dok se za ostale parametre udvostručilo. Nakon toga vidimo da sa povećanjem sa četiri na osam niti, ubrzanje za parametar 10 000 se povećalo sa 13.93 na 15.4 puta, dok se za ostale ulazne parametre ono povećalo ali manje od 2 puta kao što je to bio slučaj pri prelascima sa jedne na dve, i sa dve na četiri niti. Ovo nam govori da definitivno iako povećanje broja niti poboljšava performanse, ta ubrzanja nisu neograničena. Uverili smo se u ovo tako što smo izvršili naš program čak i sa šesnaest niti (pošto računar koji smo koristili ima šesnaest jezgara pa je moguć paralelizam sa šesnaest niti), i zaista, ubrzanje se vrlo malo povećalo u odnosu na ubrzanje sa osam niti.

Interesantan trend koji se takođe odmah primećuje za svaki broj niti u programu jeste da je ubrzanje pri ulaznom parametru od 10 000 drastično veće od ubrzanja za sve ostale ulazne parametre od 100 000, 1 000 000 i 10 000 000. Ubrzanja za ova preostala tri ulazna parametra su poprilično slična za svaki broj niti.

Još jedna vrlo interesantna pojava koja se može primetiti jeste da pri izvršenju paralelne implementacije programa sa jednom niti imamo ubrzanje koje je skoro duplo za ulazni parametar 10 000, za parametar 100 000 je ubrzanje dosta manje, oko 1.1, dok za parametre 1 000 000 i 10 000 000 praktično nema ubrzanja jer je ono nešto malo preko 1.0. Zaključili smo da je ovo slučaj zato što smo for petlju iz sekvencijalne implementacije izmenili u paralelnoj implementaciji tako da se zna broj iteracija petlje unapred svaki put kada se ona izvršava. Smatramo da je to onda omogućilo gcc prevodiocu koji smo koristili za prevođenje našeg C programa da izvrši određene optimizacije pri prevođenju te petlje, pa zato dobijamo ubrzanje čak i kada se paralelni program izvršava sa jednom niti. Sa grafika se može zaključiti da ove optimizacije više utiču na ubrzanje programa kada je broj iteracija petlje manji, tj. za manje vrednosti ulaznog parametra.

2. PROBLEM 2 – IZRAČUNAVANJE ARITMETIČKIH BROJEVA KORIŠĆENJEM KONCEPTA POSLOVA (TASKS)

2.1. Tekst problema

Rešiti prethodni problem korišćenjem koncepta poslova (tasks). Obratiti pažnju na eventualnu potrebu za sinhronizacijom i granularnost poslova. Program testirati sa parametrima koji su dati u run skripti.

2.2. Delovi koje treba paralelizovati

2.2.1. Diskusija

Važi sve isto kao u poglavlju 1.2.1, osim što sada želimo da isti problem uradimo na drugačiji način, koristeći koncept poslova (task-ova).

2.2.2. Način paralelizacije

```
    while (arithmetic_count <= num)
    {
        number_of_iterations = num + 1 - arithmetic_count;
        n += number_of_iterations;
        chunk_size = (number_of_iterations + number_of_threads - 1) / number_of_threads;
#pragma omp parallel default(none) private(i, current_chunk, task_start, task_end, number_of_chunks)      \
    shared(start, number_of_iterations, arithmetic_count, composite_count, chunk_size, number_of_threads) \
    num_threads(number_of_threads)
        {
#pragma omp single
            {
                number_of_chunks = number_of_iterations < number_of_threads ? number_of_iterations : number_of_threads;
                for (current_chunk = 0; current_chunk < number_of_chunks; current_chunk++)
                {
#pragma omp task
                    {
                        task_start = start + current_chunk * chunk_size;
                        task_end =
                            (task_start + chunk_size < start + number_of_iterations ? task_start + chunk_size : start + number_of_iterations);
                        for (i = task_start; i < task_end; i++)
                        {

                            unsigned int divisor_count;
                            unsigned int divisor_sum;
                            divisor_count_and_sum(i, &divisor_count, &divisor_sum);
                            if (divisor_sum % divisor_count == 0)
                            {
#pragma omp atomic
                                ++arithmetic_count;
                                if (divisor_count > 2)
                                {
#pragma omp atomic
                                    ++composite_count;
                                }
                            }
                        }
                    }
                }
                start += number_of_iterations;
            }
        }
    }
```

U ovom problemu je paralelizacija urađena pomoću koncepta poslova (task-ova) pošto je to eksplicitno traženo. Ovde smo se uverili na praktičnom primeru koliko je važno paziti da se ne generiše previše task-ova pošto se program drastično usporio kada smo pokušali da napravimo po task za svaku iteraciju for petlje iz prošlog primera. Zato smo se odlučili da ukupan broj iteracija podelimo na jednake delove (samo će jedan deo biti drugačije veličine u odnosu na ostale u slučaju kada ukupan broj iteracija nije deljiv sa brojem niti koje se koriste), i da takve delove koji se sastoje iz više iteracija upakujemo u task-ove. Na ovaj način će uvek biti generisan broj task-ova koji je manji ili jednak broju niti, i na taj način su smanjeni režijski troškovi koje stvara proizvodnja task-ova, a i ovako će uvek moći svi napravljeni task-ovi da se zaista izvršavaju paralelno. Naravno, ovakva implementacija se pokazala odličnom jer su postignuti rezultati koji su lošiji u odnosu na one kada je problem bio rešen pomoću ugrađene for direktive za raspodelu posla – ovo je skroz u redu pošto je logično da će for direktiva efikasnije uraditi raspodelu posla pravilne for petlje u odnosu na ručnu raspodelu posla pomoću taskova.

2.3. Rezultati

2.3.1. Logovi izvršavanja

```
Sequential implementation execution time: 0.002867s
Parallel implementation (one thread) execution time: 0.001425s
Test PASSED
Parallel implementation (two threads) execution time: 0.000395s
Test PASSED
Parallel implementation (four threads) execution time: 0.000401s
Test PASSED
Parallel implementation (eight threads) execution time: 0.000427s
Test PASSED
```

Izvršavanje komande ./arithmetic 10000

```
Sequential implementation execution time: 0.008377s
Parallel implementation (one thread) execution time: 0.008601s
Test PASSED
Parallel implementation (two threads) execution time: 0.005047s
Test PASSED
Parallel implementation (four threads) execution time: 0.003480s
Test PASSED
Parallel implementation (eight threads) execution time: 0.003172s
Test PASSED
```

Izvršavanje komande ./arithmetic 100000

```
Sequential implementation execution time: 0.176871s
Parallel implementation (one thread) execution time: 0.177571s
Test PASSED
Parallel implementation (two threads) execution time: 0.104079s
Test PASSED
Parallel implementation (four threads) execution time: 0.057059s
Test PASSED
Parallel implementation (eight threads) execution time: 0.037382s
Test PASSED
```

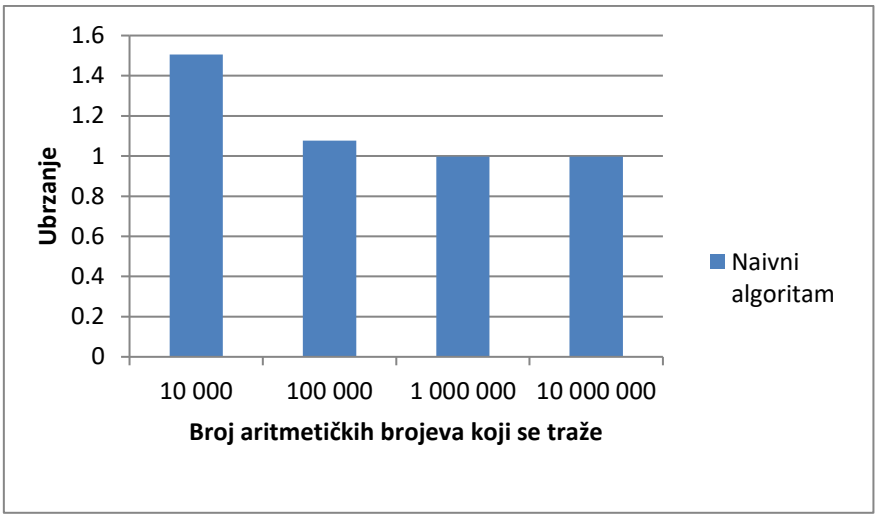
Izvršavanje komande ./arithmetic 1000000

```
Sequential implementation execution time: 4.200572s
Parallel implementation (one thread) execution time: 4.212053s
Test PASSED
Parallel implementation (two threads) execution time: 2.507355s
Test PASSED
Parallel implementation (four threads) execution time: 1.346216s
Test PASSED
Parallel implementation (eight threads) execution time: 0.777307s
Test PASSED
```

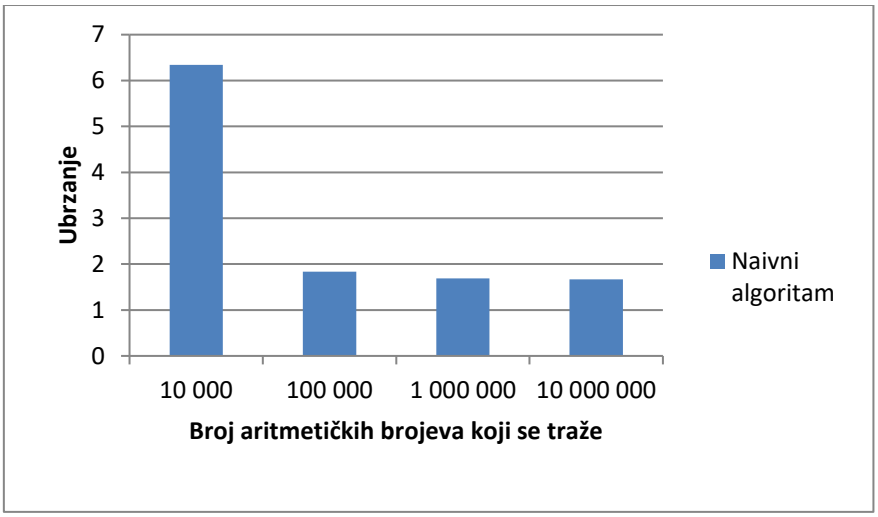
Izvršavanje komande ./arithmetic 10000000

2.3.2. Grafici ubrzanja

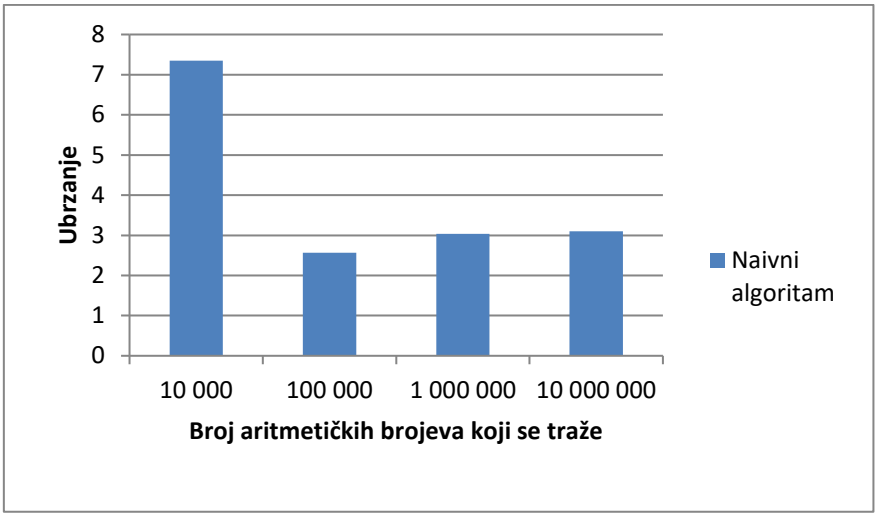
U okviru ove sekcije su dati grafici ubrzanja u odnosu na sekvencijalnu implementaciju.



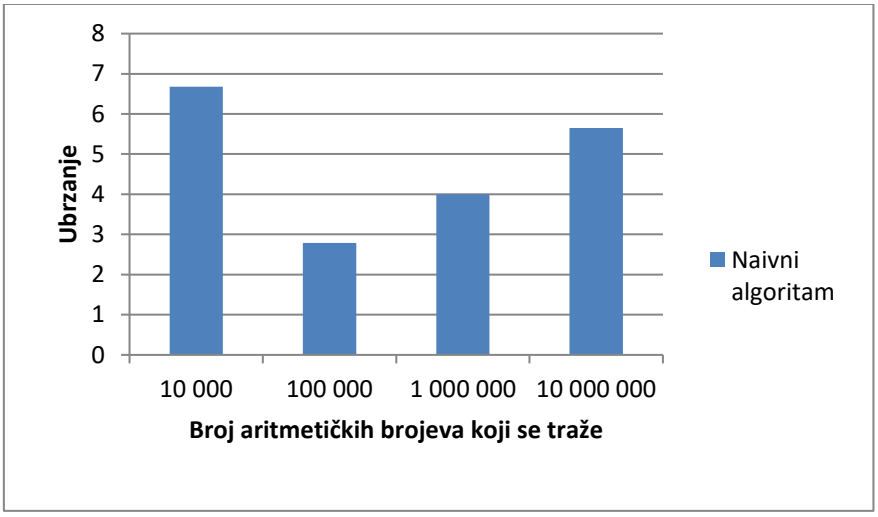
Slika 1. Grafik zavisnosti ubrzanja naivnog algoritma od broja aritmetičkih brojeva koji se traže za N = 1 niti



Slika 1. Grafik zavisnosti ubrzanja naivnog algoritma od broja aritmetičkih brojeva koji se traže za N = 2 niti



Slika 1. Grafik zavisnosti ubrzanja naivnog algoritma od broja aritmetičkih brojeva koji se traže za N = 4 niti



Slika 1. Grafik zavisnosti ubrzanja naivnog algoritma od broja aritmetičkih brojeva koji se traže za N = 8 niti

2.3.3. Diskusija dobijenih rezultata

Kako bi grafici i naši rezultati bili precizniji, zabeležili smo rezultate kako sekvencijalnog, tako i paralelnog izvršavanja programa za sve parametre i za svaki broj niti pet puta i uzimali prosečna vremena.

Na ovim graficima se jasno ponovo vidi da se primetno veće ubrzanje postiže za najmanji ulazni parametar u odnosu na ostale ulazne parametre, što je opet zbog optimizacija gcc prevodioca koje odradi za pravilnu for petlju kod koje se zna broj iteracija unapred. Te optimizacije objašnjavaju i naizgled neočekivano ubrzanje kod paralelne implementacije sa jednom niti.

Očigledno je da su sva ubrzanja kod rešenja pomoću task-ova u odnosu na ona kod rešenje preko for direktive manja zbog režijskih troškova koje donose task-ovi. Takođe, ovakvi rezultati imaju smisla pošto je logično da će for direktiva efikasnije raspodeliti posao između niti nego mi ručno pomoću task-ova.

Takođe vidimo da ovde ti režijski troškovi posebno dolaze do izražaja pri izvršenju paralelne implementacije problema sa osam niti – za ulazni parametar 10 000 čak imamo i usporenje u odnosu na izvršenje sa četiri niti, dok je za ostale ulazne parametre ubrzanje više poraslo za veće ulazne parametre u odnosu na ubrzanje pri izvršenju sa četiri niti. To je i očekivano pošto u ovom rešenju imamo povećane režijske troškove, pa ako se još koristi više niti imamo i dodatne troškove koji se više osete pri manjim ulaznim parametrima.

3. Problem 3 – Generisanje elemenata Halton Quasi Monte Carlo sekvence bez direktiva za podelu posla (worksharing direktiva)

3.1. Tekst problema

Paralelizovati program koji vrši generisanje elemenata Halton Quasi Monte Carlo (QMC) sekvence. Program se nalazi u datoteci halton.c u arhivi koja je priložena uz ovaj dokument. Prilikom paralelizacije nije dozvoljeno koristiti direktive za podelu posla (worksharing direktive), već je iteracije petlje koja se paralelizuje potrebno raspodeliti ručno. Obratiti pažnju na ispravno deklarisanje svih promenljivih prilikom paralelizacije. Program testirati sa parametrima koji su dati u run skripti.

3.2. Delovi koje treba paralelizovati

3.2.1. Diskusija

Ova petlja bi mogla da se paralelizuje:

```
for (m = 1; m <= iter; m++)
{
    r = halton_sequence_sequential(0, 10, m);
    free(r);
}
```

Bili smo svesni da je bilo potrebno odlučiti da li je bolje paralelizovati ovu petlju ili paralelizovati petlje unutar funkcije halton_sequence_sequential. Očigledno je da sa svakom iteracijom raste i količina posla koju je potrebno obaviti, pa samim tim nije efikasno raspodeliti posao između niti tako da svaka dobije približno podjednak broj iteracija (to je statička raspodela posla). Kako se u ovom problemu eksplicitno tražilo da se raspodela posla obavi ručno, a tada bi raspodela posla upravo bila statička, zaključili smo (a i uverili se isprobavanjem) da je bolje opredeliti se za paralelizaciju petlji unutar funkcije halton_sequence_sequential.

Sledeći deo koda je pogodan za paralelizaciju:

```
for (j = 0; j < m; j++)
{
    t[j] = i;
}
for (j = 0; j < m; j++)
{
    prime_inv[j] = 1.0 / (double)(prime(j + 1));
}
```

Pre paralelizacije, moguća je i optimizacija ovog koda jer nema potrebe da se ovaj posao odradi pomoću dve for petlje – dovoljna je jedna. Kada se to uradi, potrebno je samo uraditi najobičniju paralelizaciju pravilne for petlje.

I narednu spoljašnju petlju treba paralelizovati:

```
for (j = 0; j < n; j++)
{
    for (i = 0; i < m; i++)
    {
        r[i + j * m] = 0.0;
    }
}
```

Primećeno je i da se isplati paralelizovati i sledeću petlju:

```
for (j = 0; j < m; j++)
{
    d = (t[j] % prime(j + 1));
    r[j + k * m] = r[j + k * m] + (double)(d)*prime_inv[j];
    prime_inv[j] = prime_inv[j] / (double)(prime(j + 1));
    t[j] = (t[j] / prime(j + 1));
}
```

Sve ove petlje su pogodne za paralelizaciju jer su im iteracije međusobno nezavisne.

Bila je razmatrana i paralelizacija naredne spoljašnje petlje:

```
for (k = 0; k < n; k++)
{
    for (j = 0; j < m; j++)
    {
        t[j] = i;
    }
    for (j = 0; j < m; j++)
    {
        prime_inv[j] = 1.0 / (double)(prime(j + 1));
    }

    while (0 < i4vec_sum(m, t))
    {
        for (j = 0; j < m; j++)
        {
            d = (t[j] % prime(j + 1));
            r[j + k * m] = r[j + k * m] + (double)(d)*prime_inv[j];
            prime_inv[j] = prime_inv[j] / (double)(prime(j + 1));
            t[j] = (t[j] / prime(j + 1));
        }
    }
    i = i + i3;
}
```

Međutim, na taj način se ne bi dobili isti rezultati kao kod sekvencijalne implementacije problema pošto postoji zavisnost po podacima između iteracija ove petlje – u svakoj iteraciji se promenljiva „i“ menja, a takođe se i koristi pri izračunavanjima.

Naredna petlja takođe deluje kao da bi trebalo da se paralelizuje:

```
for (i = 0; i < n; i++)
{
    sum = sum + a[i];
}
```

Ipak, poređenjem rezultata bilo je očigledno da paralelizacija ove petlje uspori program zato što su u našim test primerima vrednosti promenljive „n“ brojevi 10, 100 i 1000. Očigledno se za tako male vrednosti ne isplati paralelizovati ovu petlju.

3.2.2. Način paralelizacije

Raspodela posla je odrađena ručno zato što je to eksplicitno traženo u ovom problemu.

3.3. Rezultati

3.3.1. Logovi izvršavanja

```
Sequential implementation execution time: 0.001752s
Parallel implementation (one thread) execution time: 0.001892s
Test PASSED
Parallel implementation (two threads) execution time: 0.000929s
Test PASSED
Parallel implementation (four threads) execution time: 0.000517s
Test PASSED
Parallel implementation (eight threads) execution time: 0.000655s
Test PASSED
```

Izvršavanje komande ./halton 10

```
Sequential implementation execution time: 0.019810s
Parallel implementation (one thread) execution time: 0.022293s
Test PASSED
Parallel implementation (two threads) execution time: 0.019171s
Test PASSED
Parallel implementation (four threads) execution time: 0.012504s
Test PASSED
Parallel implementation (eight threads) execution time: 0.010541s
Test PASSED
```

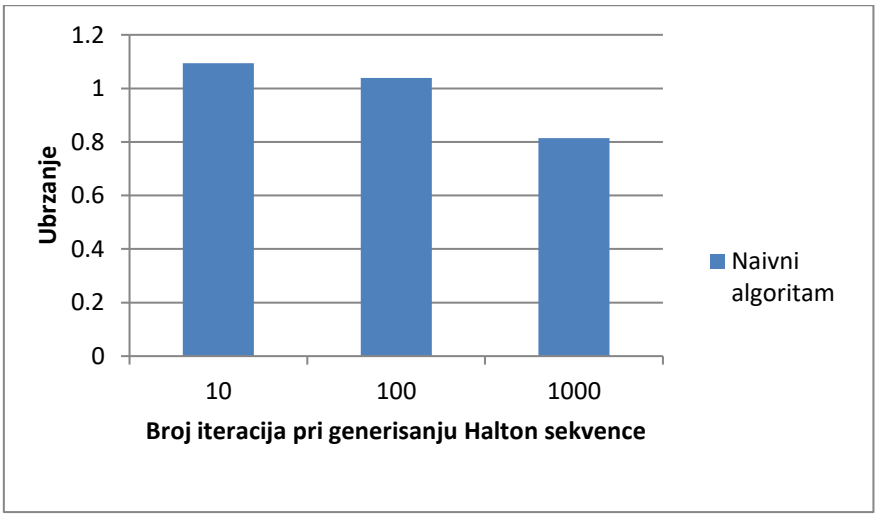
Izvršavanje komande ./halton 100

```
Sequential implementation execution time: 2.468872s
Parallel implementation (one thread) execution time: 2.440273s
Test PASSED
Parallel implementation (two threads) execution time: 1.998612s
Test PASSED
Parallel implementation (four threads) execution time: 1.013140s
Test PASSED
Parallel implementation (eight threads) execution time: 0.641103s
Test PASSED
```

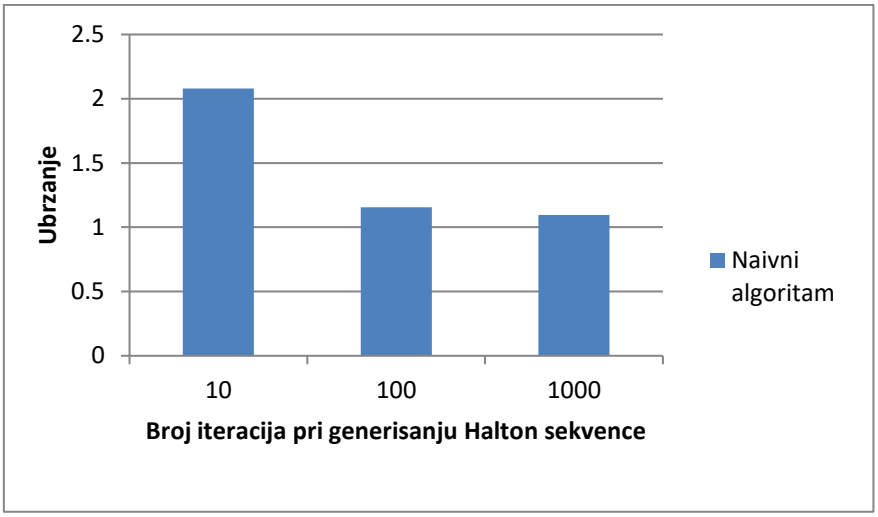
Izvršavanje komande ./halton 1000

3.3.2. Grafici ubrzanja

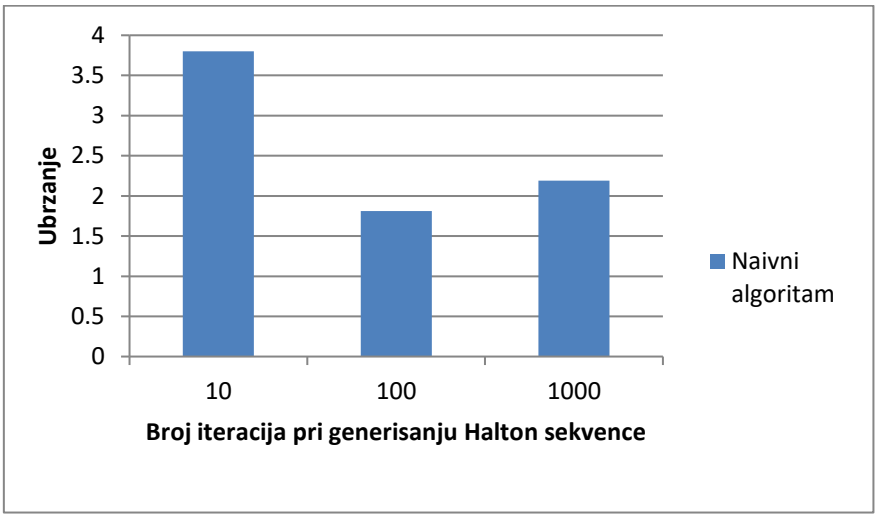
U okviru ove sekcije su dati grafici ubrzanja u odnosu na sekvencijalnu implementaciju.



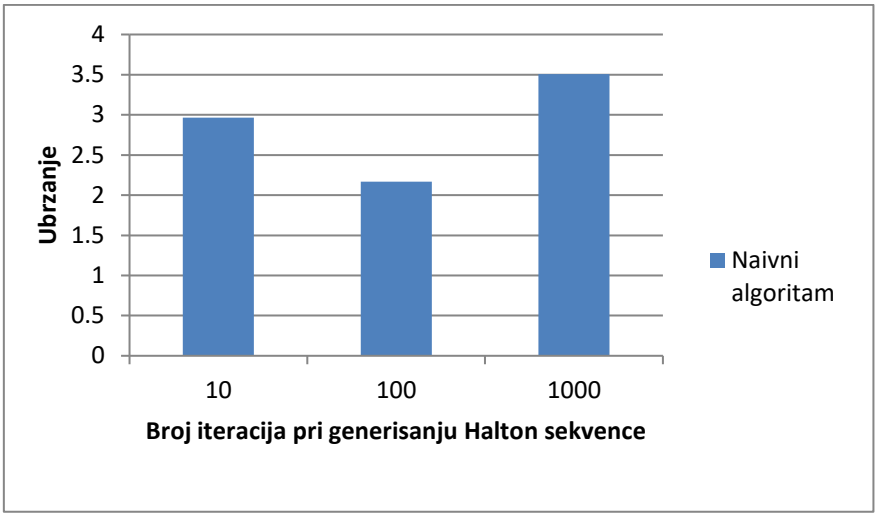
Slika 1. Grafik zavisnosti ubrzanja naivnog algoritma od broja iteracija pri generisanju Halton sekvence za N = 1 niti



Slika 1. Grafik zavisnosti ubrzanja naivnog algoritma od broja iteracija pri generisanju Halton sekvence za N = 2 niti



Slika 1. Grafik zavisnosti ubrzanja naivnog algoritma od broja iteracija pri generisanju Halton sekvence za N = 4 niti



Slika 1. Grafik zavisnosti ubrzanja naivnog algoritma od broja iteracija pri generisanju Halton sekvence za N = 8 niti

3.3.3. Diskusija dobijenih rezultata

Prokomentarisaćemo svaki ulazni parametar zasebno:

1) ulazni parametar 10:

   Pri izvršavanju paralelne implementacije programa sa dve i četiri niti, najveće ubrzanje se postiže za ovaj ulaz i takođe se vidi veliki napredak u ubrzanju pri prelasku sa dve na četiri niti. Međutim, već za osam niti vidimo primećujemo čak i usporenje u odnosu na vreme izvršenja programa sa četiri niti, što ima smisla – za ovako mali broj iteracija u tom slučaju režijski troškovi prosto postaju preveliki.

2) ulazni parametar 100:

   Što se ovog ulaza tiče, sa povećanjem broja niti sa dve na četiri, pa onda i sa četiri na osam se primećuje povećanje ubrzanja. Ipak, i ovde se primećuje uticaj režijskih troškova zbog relativno malog broja iteracija, pa je napredak ubrzanja pri prelasku sa četiri na osam niti primetno manji od onog pri prelasku sa dve na četiri niti.

3) ulazni parametar 1000:

   Ovo je jedini ulaz za koji vidimo ne samo povećanje ubrzanja, nego i veći napredak u ubrzanju pri prelasku sa četiri na osam niti u odnosu na onaj pri prelasku sa dve na četiri niti. Ovo ima smisla zato što sada ipak imamo veći broj iteracija, pa je uticaj režijskih troškova manji.

4. Problem 4 – Generisanje elemenata Halton Quasi Monte Carlo sekvence sa direktivama za podelu posla (worksharing direktivama)

4.1. Tekst problema,

Prethodni program paralelizovati korišćenjem direktiva za podelu posla (worksharing direktive). Program testirati sa parametrima koji su dati u run skripti.

4.2. Delovi koje treba paralelizovati

4.2.1. Diskusija

Istu odluku je bilo potrebno opet doneti kao u prošlom problemu – da li paralelizovati ovu petlju ili petlje unutar funkcije halton_sequence_sequential. Pošto nam je u ovom problemu dozvoljeno da koristimo for direktivu, imamo mogućnost da dinamički raspodelimo posao između niti pomoću schedule odredbe, što nam upravo odgovara kod ovakve petlje kod koje veličina posla koji se obavlja raste sa brojem iteracija. Isprobavanjem smo utvrdili da se dobijaju slična ubrzanja kao kada paralelizaciju vršimo unutar funkcije halton_sequence_sequential, pa smo se odlučili da na istim mestima vršimo paralelizacije kao u prošlom problemu, samo sa for direktivom.

```
for (m = 1; m <= iter; m++)
{
    r = halton_sequence_sequential(0, 10, m);
    free(r);
}
```

Sve ostalo isto važi kao u poglavlju 3.2.1., samo što je ovde raspodela posla urađena for direktivom umesto ručno.

4.2.2. Način paralelizacije

Sve petlje za koje je uočeno da ih treba paralelizovati su pravilne for petlje, pa su zato one sve paralelizovane upotrebom for direktive zato što je u ovom slučaju to najefikasniji način. Kako je balans opterećenja u svim petljama gotovo idealan, nema potrebe koristiti drugačiju raspodelu posla od podrazumevane schedule(static), gde se svakoj niti unapred dodeli određen broj uzastopnih iteracija, tj. koristi se blokovska raspodela.

4.3. Rezultati

4.3.1. Logovi izvršavanja

```
Sequential implementation execution time: 0.001438s
Parallel implementation (one thread) execution time: 0.001973s
Test PASSED
Parallel implementation (two threads) execution time: 0.001196s
Test PASSED
Parallel implementation (four threads) execution time: 0.000477s
Test PASSED
Parallel implementation (eight threads) execution time: 0.000572s
Test PASSED
```

Izvršavanje komande ./halton 10

```
Sequential implementation execution time: 0.021944s
Parallel implementation (one thread) execution time: 0.021488s
Test PASSED
Parallel implementation (two threads) execution time: 0.019701s
Test PASSED
Parallel implementation (four threads) execution time: 0.011934s
Test PASSED
Parallel implementation (eight threads) execution time: 0.009279s
Test PASSED
```

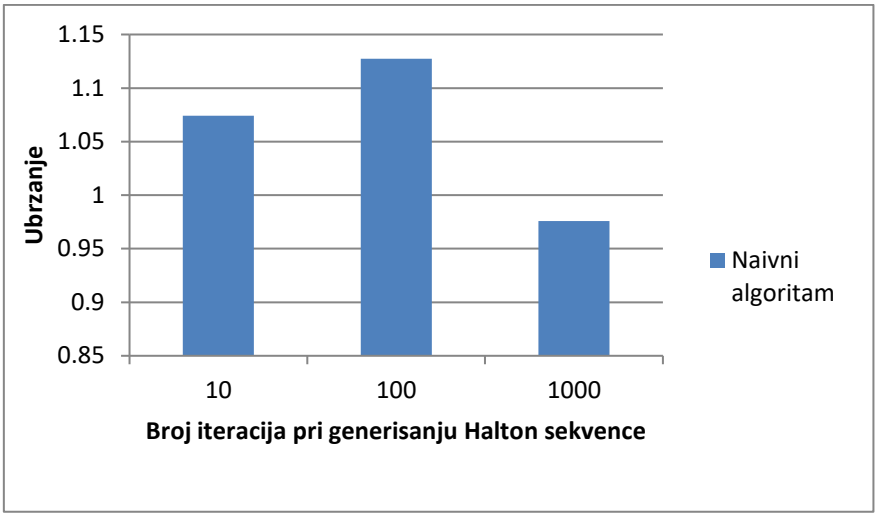
Izvršavanje komande ./halton 100

```
Sequential implementation execution time: 2.426729s
Parallel implementation (one thread) execution time: 2.411400s
Test PASSED
Parallel implementation (two threads) execution time: 2.040469s
Test PASSED
Parallel implementation (four threads) execution time: 1.010487s
Test PASSED
Parallel implementation (eight threads) execution time: 0.565278s
Test PASSED
```

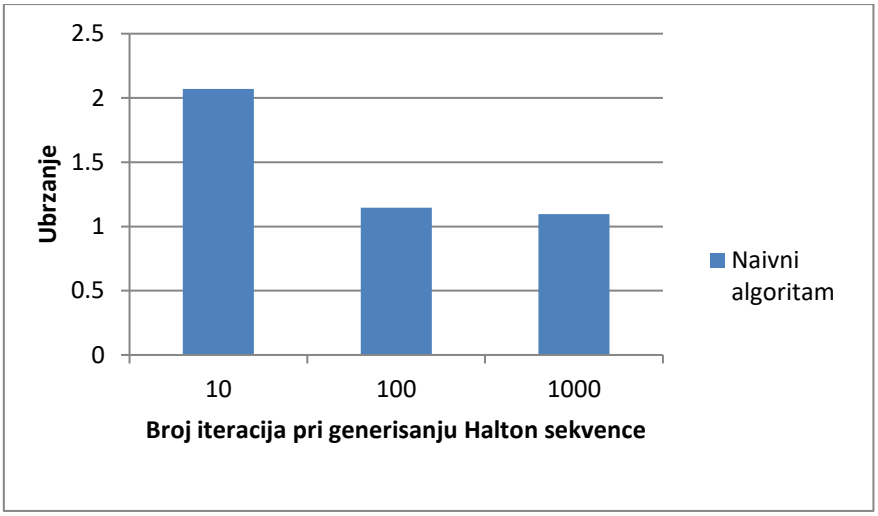
Izvršavanje komande ./halton 1000

4.3.2. Grafici ubrzanja

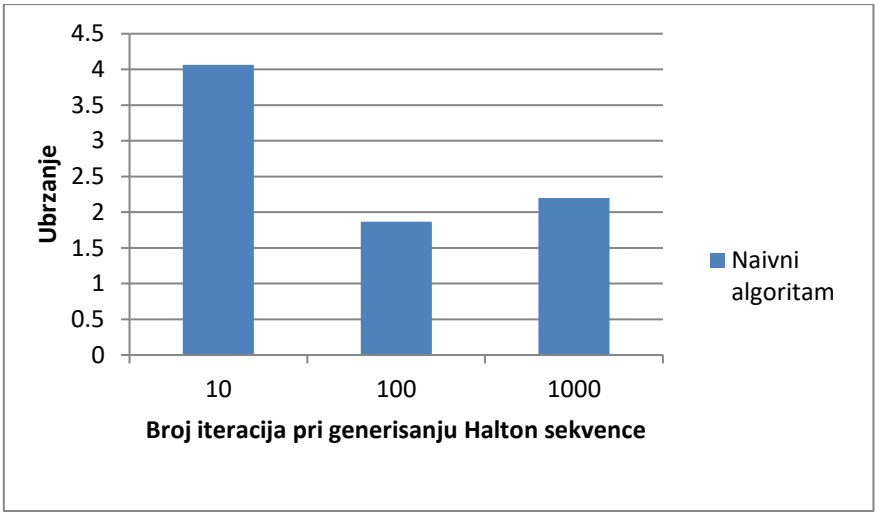
U okviru ove sekcije su dati grafici ubrzanja u odnosu na sekvencijalnu implementaciju.



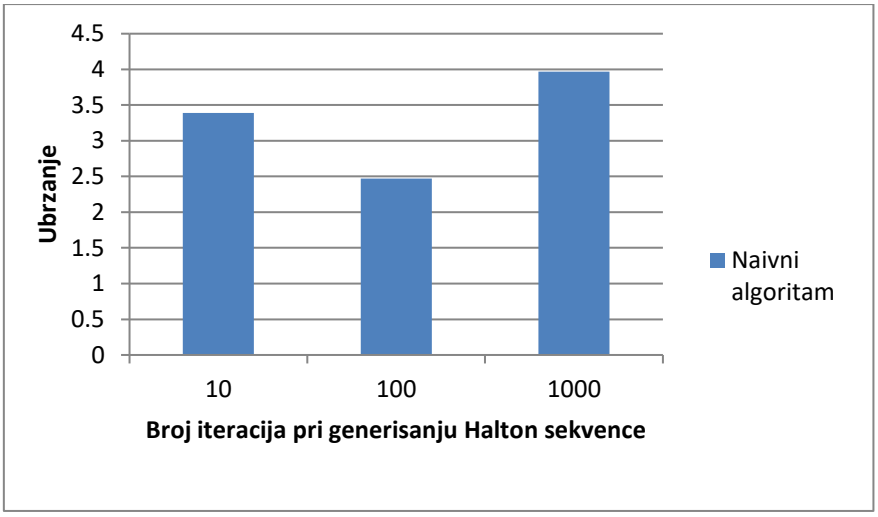
Slika 1. Grafik zavisnosti ubrzanja naivnog algoritma od broja iteracija pri generisanju Halton sekvence za N = 1 niti



Slika 1. Grafik zavisnosti ubrzanja naivnog algoritma od broja iteracija pri generisanju Halton sekvence za N = 2 niti



Slika 1. Grafik zavisnosti ubrzanja naivnog algoritma od broja iteracija pri generisanju Halton sekvence za N = 4 niti



Slika 1. Grafik zavisnosti ubrzanja naivnog algoritma od broja iteracija pri generisanju Halton sekvence za N = 8 niti

4.3.3. Diskusija dobijenih rezultata

Tumačenja dobijenih rezultata su poprilično slična onim iz poglavlja 3.3.3., s tim što su sva ubrzanja malo bolja sada pošto se raspodela posla ovde radi pomoću for direktive umesto ručno.

5. PROBLEM 5 – SIMULACIJA KRETANJA N TELA (N-BODY PROBLEM)

5.1. Tekst problema

Paralelizovati program koji se bavi problemom n tela (n-body problem). Sva tela imaju jediničnu masu, trokomponentni vektor položaja (x, y, z) i trokomponentni vektor brzine (vx, vy, vz). Simulaciju n tela se odvija u iteracijama, pri čemu se u svakoj iteraciji izračunava sila kojom sva tela deluju na sva ostala, a zatim se brzine i koordinate tela ažuriraju prema II Njutnovom zakonu. Brzine i položaji su slučajno generisani na početku simulacije. Zbog same prirode numeričke simulacije uveden je parametar SOFTENING, koji predstavlja korektivni faktor prilikom izračunavanja rastojanja između čestica (kako je gravitaciona sila obrnuto proporcionalna rastojanju između čestica, za nulta rastojanja i rastojanja bliska nuli, izračunata gravitaciona sila postaje izuzetno velika – teži beskonačnosti).

Program se nalazi u datoteci direktorijumu nbodymini u arhivi koja je priložena uz ovaj dokument. Program koji treba paralelizovati nalazi se u datoteci nbody.c. Pored samog izračunavanja, program čuva rezultate svake iteracije u zasebnim datotekama (za svako telo se čuvaju pozicije i brzine), dok kod show_nbody.py kreira gif same simulacije.

Ukoliko je potrebno međusobno isključenje prilikom paralelizacije programa, koristiti kritične sekcije ili atomske operacije. Skripta run pokreće simulaciju za različite parametre, i nakon toga, za određene simulacije poziva python kod koji kreira gifove.

5.2. Delovi koje treba paralelizovati

5.2.1. Diskusija

Vrlo je pogodno paralelizovati ovde spoljašnju petlju pošto su iteracije međusobno nezavisne.

```
void bodyForce(Body *p, float dt, int n)
{
    for (int i = 0; i < n; i++)
    {
        float Fx = 0.0f;
        float Fy = 0.0f;
        float Fz = 0.0f;

        for (int j = 0; j < n; j++)
        {
            float dx = p[j].x - p[i].x;
            float dy = p[j].y - p[i].y;
            float dz = p[j].z - p[i].z;
            float distSqr = dx * dx + dy * dy + dz * dz + SOFTENING;
            float invDist = 1.0f / sqrtf(distSqr);
            float invDist3 = invDist * invDist * invDist;

            Fx += dx * invDist3;
            Fy += dy * invDist3;
            Fz += dz * invDist3;
        }

        p[i].vx += dt * Fx;
        p[i].vy += dt * Fy;
        p[i].vz += dt * Fz;
    }
}
```

Ovo je takođe jedina paralelizacija petlje koja nam je donela ubrzanje programa za veći broj nebeskih tela.

Ova petlja bi mogla da se paralelizuje:

```
void randomizeBodies(float *data, int n)
{
    for (int i = 0; i < n; i++)
    {
        data[i] = 2.0f * (rand() / (float)RAND_MAX) - 1.0f;
    }
}
```

Međutim, isprobali smo i ne isplati se jer se program tada uspori zbog dodatnih režijskih troškova od paralelizacije koji su veći od benefita te paralelizacije pošto se ova funkcija poziva samo jednom na početku našeg programa i nikada više.

Evo još jedne petlje koja bi mogla da se paralelizuje:

```
for (int i = 0; i < n; i++)
{
    fprintf(file, "%f,%f,%f,%f,%f,%f\n", p[i].x, p[i].y, p[i].z, p[i].vx, p[i].vy, p[i].vz);
}
```

Isprobali smo i ovo, i program se uspori dodatno, tako da smo odustali od ove paralelizacije.

Razmatrali smo i paralelizaciju ove petlje:

```
for (int i = 0; i < nBodies; i++)
{
    p[i].x += p[i].vx * dt;
    p[i].y += p[i].vy * dt;
    p[i].z += p[i].vz * dt;
}
```

Ali i tu smo dobili usporenje, pa nećemo ni ovde raditi paralelizaciju.

Pomislili smo i da paralelizujemo ovde spoljašnju petlju.

```
for (int iter = 0; iter < nIters; iter++)
{
    bodyForce(p, dt, nBodies);

    saveToCSV(p, nBodies, iter, folder);

    for (int i = 0; i < nBodies; i++)
    {
        p[i].x += p[i].vx * dt;
        p[i].y += p[i].vy * dt;
        p[i].z += p[i].vz * dt;
    }
}
```

Međutim shvatili smo da tako ne bismo dobili korektne rezultate zato što je važno da se uvek prvo pozove funkcija bodyForce kako bi se ažurirale koordinate brzine svakog nebeskog tela, pa onda nakon toga da se ažuriraju koordinate položaja svih nebeskih tela. Jasno je da, kada bismo paralelizovali ovu spoljašnju petlju, ne bi bio garantovan ovakav sled izvršavanja jer bi bilo omogućeno da se dogodi da više niti istovremeno pozove funkciju bodyForce i pre nego što su ikakva ažuriranja koordinata položaja urađena od strane bilo koje niti, pa se tada uopšte ne bi dobilo isto pomeranje nebeskih tela kao u sekvencijalnoj implementaciji problema, već bismo imali nedeterminističko kretanje nebeskih tela. Ovo naravno ne želimo jer je neophodno da paralelna implementacija problema ima iste rezultate kao i sekvencijalna kako bi paralelizacija uopšte imala smisla.

5.2.2. Način paralelizacije

Na kraju smo zaključili da je najbolje paralelizovati samo spoljašnju for petlju u funkciji bodyForce, pa pošto je ona pravilna najbolje je tu paralelizaciju uraditi for direktivom.

5.3. Rezultati

5.3.1. Logovi izvršavanja

```
Running ./nbody with 30 particles, 100 iterations, and saving to simulation_1
folder
```

```
Sequential implementation execution time: 0.007780s
Parallel implementation (one thread) execution time: 0.003356s
Test PASSED
Parallel implementation (two threads) execution time: 0.066015s
Test PASSED
Parallel implementation (four threads) execution time: 0.022392s
Test PASSED
Parallel implementation (eight threads) execution time: 0.031040s
Test PASSED
```

Izvršavanje komande ./nbody 30 100 simulation_1

```
Running ./nbody with 30 particles, 1000 iterations, and saving to simulation_2 folder
Sequential implementation execution time: 0.034302s
Parallel implementation (one thread) execution time: 0.036580s
Test PASSED
Parallel implementation (two threads) execution time: 0.058436s
Test PASSED
Parallel implementation (four threads) execution time: 0.057357s
Test PASSED
Parallel implementation (eight threads) execution time: 0.104761s
Test PASSED
```

Izvršavanje komande ./nbody 30 1000 simulation_2

```
Running ./nbody with 3000 particles, 100 iterations, and saving to simulation_3 folder
Sequential implementation execution time: 2.410705s
Parallel implementation (one thread) execution time: 2.367187s
Test PASSED
Parallel implementation (two threads) execution time: 1.339817s
Test PASSED
Parallel implementation (four threads) execution time: 0.877911s
Test PASSED
Parallel implementation (eight threads) execution time: 0.567523s
Test PASSED
```

Izvršavanje komande ./nbody 3000 100 simulation_3

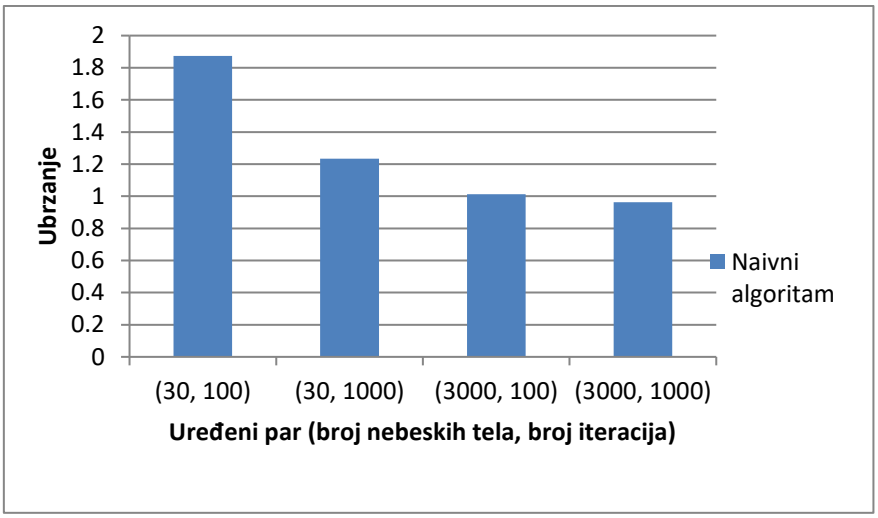
```
Running ./nbody with 3000 particles, 1000 iterations, and saving to simulation_4 folder
Sequential implementation execution time: 23.769746s
Parallel implementation (one thread) execution time: 23.672586s
Test PASSED
```

```
Parallel implementation (two threads) execution time: 13.691755s
Test PASSED
Parallel implementation (four threads) execution time: 9.099547s
Test PASSED
Parallel implementation (eight threads) execution time: 6.949680s
Test PASSED
```

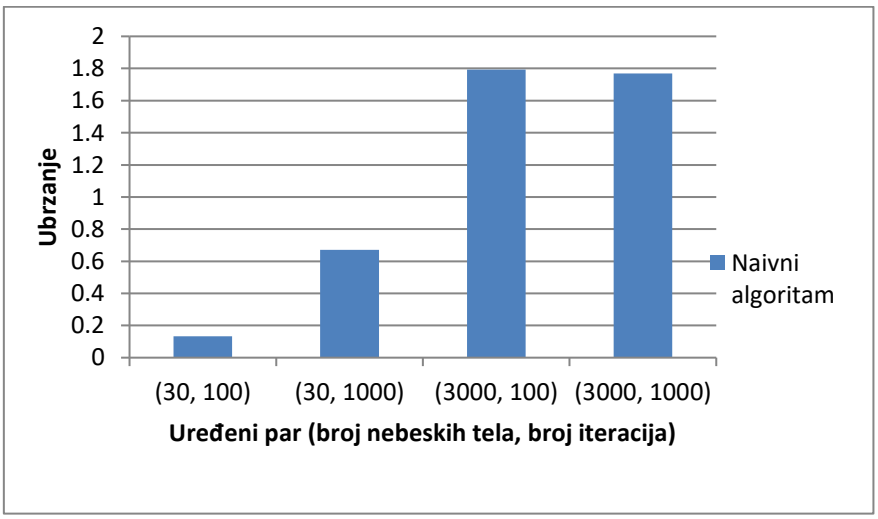
Izvršavanje komande ./nbody 3000 1000 simulation_4

5.3.2. Grafici ubrzanja

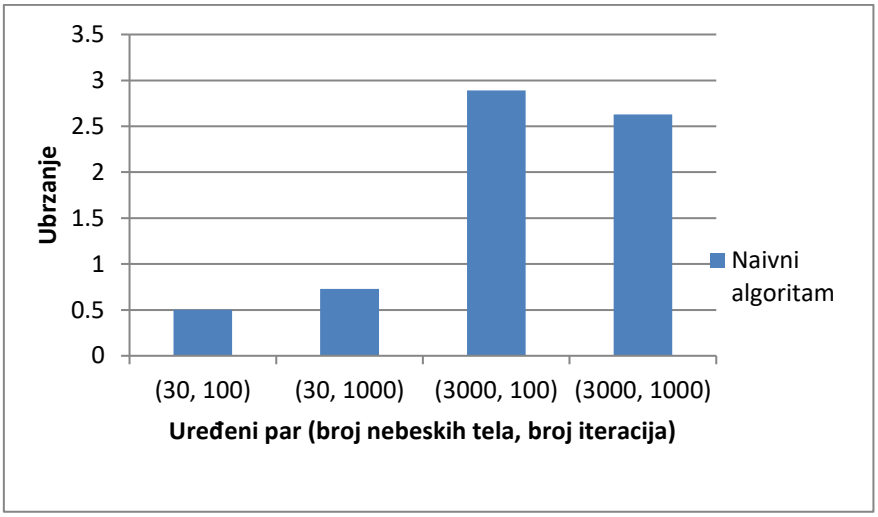
U okviru ove sekcije su dati grafici ubrzanja u odnosu na sekvencijalnu implementaciju.



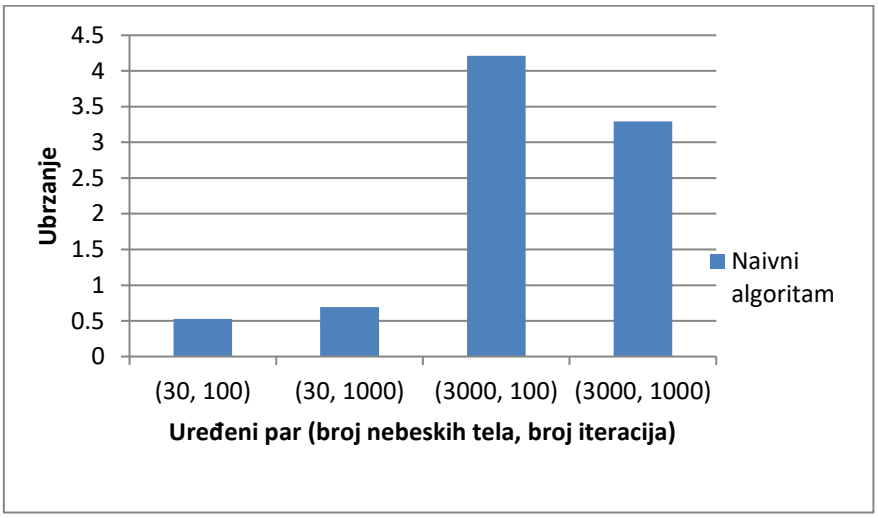
Slika 1. Grafik zavisnosti ubrzanja naivnog algoritma od broja nebeskih tela i broja iteracija za N = 1 niti



Slika 1. Grafik zavisnosti ubrzanja naivnog algoritma od broja nebeskih tela i broja iteracija za N = 2 niti



Slika 1. Grafik zavisnosti ubrzanja naivnog algoritma od broja nebeskih tela i broja iteracija za N = 4 niti



Slika 1. Grafik zavisnosti ubrzanja naivnog algoritma od broja nebeskih tela i broja iteracija za N = 8 niti

5.3.3. Diskusija dobijenih rezultata

Kako broj iteracija petlje koju paralelizujemo direktno zavisi od broja nebeskih tela, vrlo su logični rezultati koje pokazuju grafikoni zato što očigledno za mali broj nebeskih tela (30) režijski troškovi paralelizacije nadjačavaju potencijalnu dobit od iste. Sa druge strane, kada je broj nebeskih tela 3000, naša paralelizacija najviše ima smisla jer tada imamo dobra ubrzanja koja se konstantno povećavaju sa povećanjem broja niti.

Ubrzanje od oko 1.8 koje je prisutno pri izvršenju paralelne implementacije programa sa jednom niti za ulazne parametre (30, 100) objašnjavamo mogućim optimizacijama sistema koje se događaju

implicitno u pozadini. Pretpostavljamo da imaju veze sa radom sa fajlovima pošto svako izvršenje našeg programa kreira i upisuje u dosta fajlova. Smatramo da se te optimizacije osete samo za ovako male ulaze, jer vidimo da za ostale ulaze kod paralelne implementacije programa sa jednom niti nemamo toliko ubrzanje – kod ulaza (30, 1000) je ono oko 1.2, dok već za ostale veće ulaze (3000, 100) i (3000, 1000) tog ubrzanja ni nema jer je oko 1 kao što je i očekivano.