

UNIVERZITET U BEOGRADU - ELEKTROTEHNIČKI FAKULTET
MULTIPROCESORSKI SISTEMI (13S114MUPS, 13E114MUPS)



DOMAĆI ZADATAK 2 – MPI

Izveštaj o urađenom domaćem zadatku

Predmetni saradnici:

dipl. ing. Matija Dodović

Studenti:

Ljubica Majstorović 2020/0253

Pavle Šarenac 2020/0359

Beograd, decembar 2023.

SADRŽAJ

SADRŽAJ.....	2
1. PROBLEM 1 – IZRAČUNAVANJE ARITMETIČKIH BROJEVA.....	3
1.1. TEKST PROBLEMA.....	3
1.2. DELOVI KOJE TREBA PARALELIZOVATI.....	3
1.2.1. <i>Diskusija</i>	3
1.2.2. <i>Način paralelizacije</i>	4
1.3. REZULTATI.....	5
1.3.1. <i>Logovi izvršavanja</i>	5
1.3.2. <i>Grafici ubrzanja</i>	7
1.3.3. <i>Diskusija dobijenih rezultata</i>	8
2. PROBLEM 2 – GENERISANJE ELEMENATA HALTON QUASI MONTE CARLO SEKVENCE.....	9
2.1. TEKST PROBLEMA.....	9
2.2. DELOVI KOJE TREBA PARALELIZOVATI.....	9
2.2.1. <i>Diskusija</i>	9
2.2.2. <i>Način paralelizacije</i>	11
2.3. REZULTATI.....	12
2.3.1. <i>Logovi izvršavanja</i>	12
2.3.2. <i>Grafici ubrzanja</i>	13
2.3.3. <i>Diskusija dobijenih rezultata</i>	14
3. PROBLEM 3 – SIMULACIJA KRETANJA N TELA (N-BODY PROBLEM).....	16
3.1. TEKST PROBLEMA.....	16
3.2. DELOVI KOJE TREBA PARALELIZOVATI.....	16
3.2.1. <i>Diskusija</i>	16
3.2.2. <i>Način paralelizacije</i>	18
3.3. REZULTATI.....	19
3.3.1. <i>Logovi izvršavanja</i>	19
3.3.2. <i>Grafici ubrzanja</i>	20
3.3.3. <i>Diskusija dobijenih rezultata</i>	22
4. PROBLEM 4 – SIMULACIJA KRETANJA N TELA (N-BODY PROBLEM).....	23
4.1. TEKST PROBLEMA.....	23
4.2. DELOVI KOJE TREBA PARALELIZOVATI.....	23
4.2.1. <i>Diskusija</i>	23
4.2.2. <i>Način paralelizacije</i>	25
4.3. REZULTATI.....	26
4.3.1. <i>Logovi izvršavanja</i>	26
4.3.2. <i>Grafici ubrzanja</i>	27
4.3.3. <i>Diskusija dobijenih rezultata</i>	28

1.PROBLEM 1 – IZRAČUNAVANJE ARITMETIČKIH BROJEVA

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 `arithmetic.c` u arhivi koja je priložena uz ovaj dokument. Program testirati sa parametrima koji su dati u run skripti.

Proces sa rangom 0 treba da učitava ulazne podatke, raspodeli posao ostalim procesima, na kraju prikupi dobijene rezultate i ravnopravno učestvuje u obradi. Za razmenu podataka, koristiti rutine za kolektivnu komunikaciju.

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 broj iteracija ovako napisane petlje nije unapred poznat, 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 jer 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

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

Međutim, smislili smo način da ipak upotrebimo napišemo petlju kod koje znamo broj iteracija unapred. 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 ćemo napisati for petlju koja će imati 10 000 iteracija, i u njoj je onda poznat broj iteracija unapred pa se može paralelizovati. 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.

Naravno, uočili smo da promenljive `arithmetic_count` i `composite_count` modifikuju svi procesi i da se one samo inkrementiraju. Zato ove promenljive nakon glavne for petlje redukujemo, pri čemu samo MASTER proces treba da dobije redukovanu promenljivu `composite_count`, jer ostalim procesima ona nije potrebna, dok svi procesi treba da dobiju redukovanu promenljivu `arithmetic_count` jer im je svima potrebna zbog preračunavanja svog dela posla u narednoj iteraciji while petlje. Naravno, svaki proces lokalno preračunava svoj deo posla u zavisnosti od svog ranga u komunikatoru.

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 process) execution time: 0.000675s
Test PASSED
Parallel implementation (two processes) execution time: 0.000489s
Test PASSED
Parallel implementation (four processes) execution time: 0.000381s
Test PASSED
Parallel implementation (eight processes) execution time: 0.000432s
Test PASSED
```

Izvršavanje komande `./arithmetic 10000`

```
Sequential implementation execution time: 0.010243s
Parallel implementation (one process) execution time: 0.010813s
Test PASSED
Parallel implementation (two processes) execution time: 0.006225s
Test PASSED
Parallel implementation (four processes) execution time: 0.003692s
Test PASSED
Parallel implementation (eight processes) execution time: 0.003007s
Test PASSED
```

Izvršavanje komande `./arithmetic 100000`

```
Sequential implementation execution time: 0.172909s
Parallel implementation (one process) execution time: 0.206654s
Test PASSED
Parallel implementation (two processes) execution time: 0.140386s
Test PASSED
Parallel implementation (four processes) execution time: 0.068218s
Test PASSED
Parallel implementation (eight processes) execution time: 0.059723s
Test PASSED
```

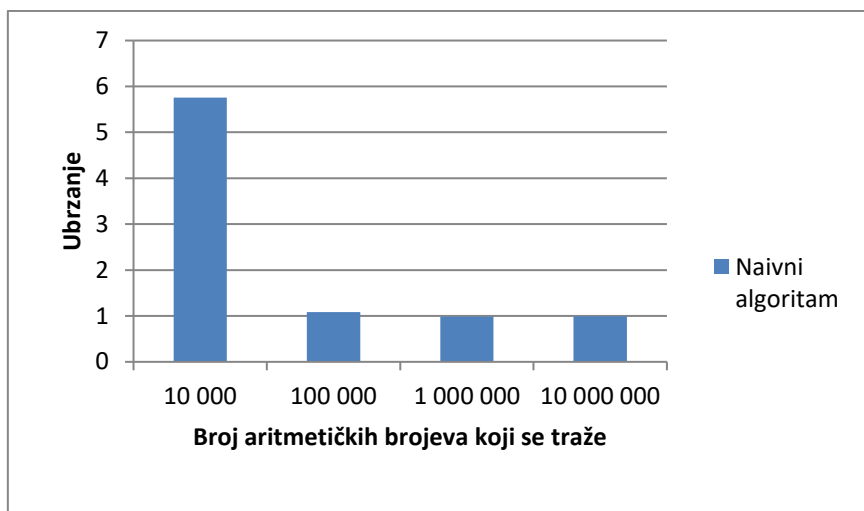
Izvršavanje komande ./arithmetic 1000000

```
Sequential implementation execution time: 4.199454s
Parallel implementation (one process) execution time: 4.565289s
Test PASSED
Parallel implementation (two processes) execution time: 2.786152s
Test PASSED
Parallel implementation (four processes) execution time: 1.536255s
Test PASSED
Parallel implementation (eight processes) execution time: 1.108353s
Test PASSED
```

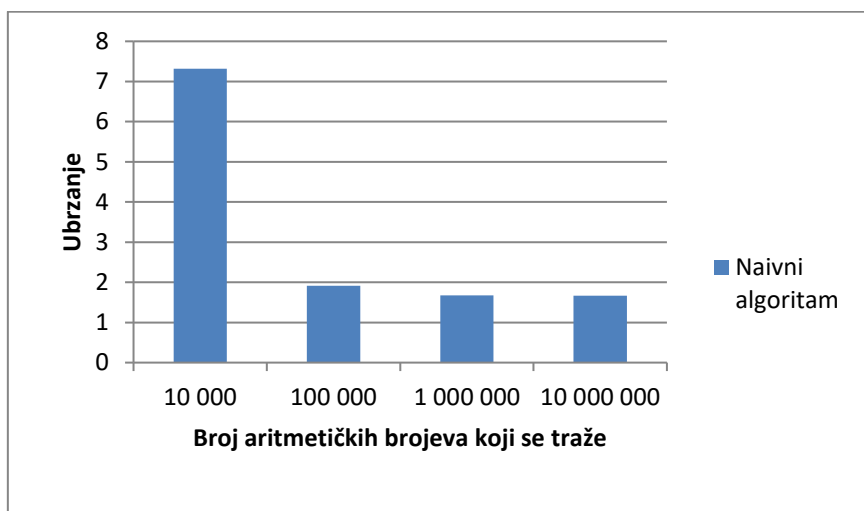
Izvršavanje komande ./arithmetic 10000000

1.3.2. Grafici ubrzanja

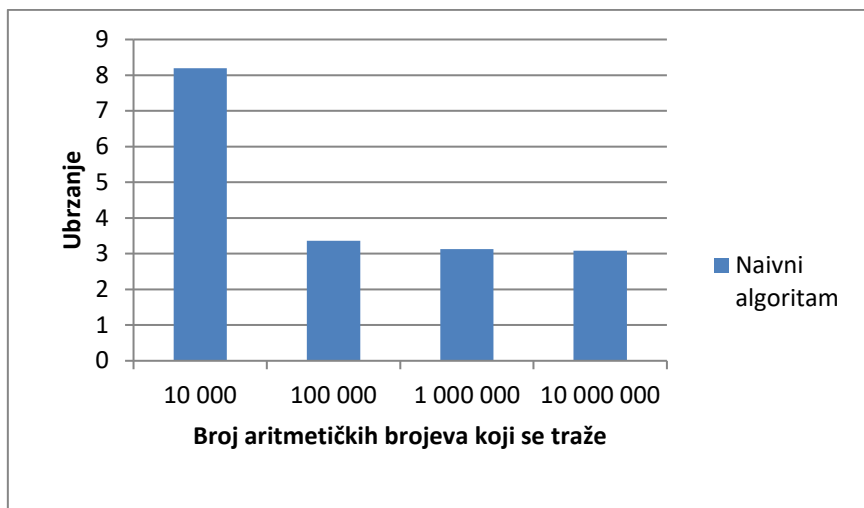
U okviru ove sekcije su dati grafici ubrzanja u odnosu na sekvencijalnu implementaciju.



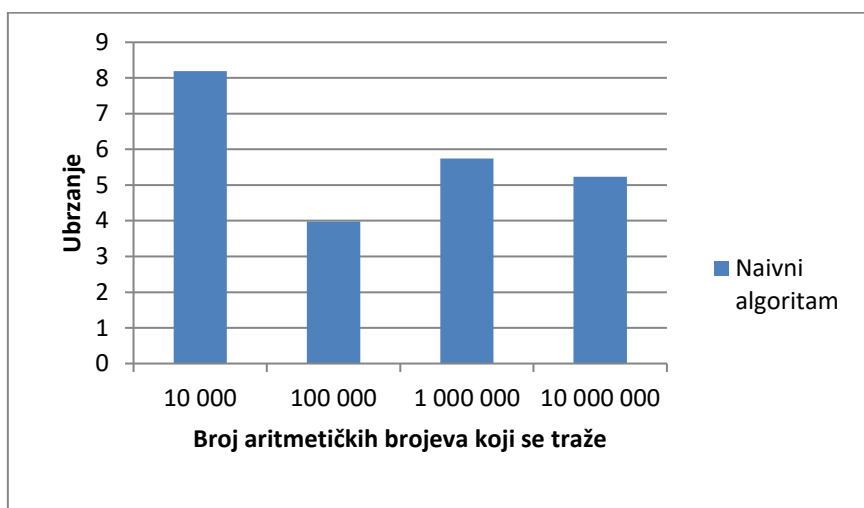
Slika 1. Grafik zavisnosti ubrzanja naivnog algoritma od broja aritmetičkih brojeva koji se traže za N = 1 procesa



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



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



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

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.

Vidimo da je trend ubrzanja vrlo sličan onom koji smo imali kod OpenMP, samo što su naravno tada same vrednosti ubrzanja bile veće jer kod MPI imamo dosta više overhead-a zbog veće komunikacije i sinhronizacije između procesa.

2. PROBLEM 2 – GENERISANJE ELEMENATA HALTON QUASI MONTE CARLO SEKVENCE

2.1. 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. Program testirati sa parametrima koji su dati u run skripti.

Ukoliko je moguće, koristiti rutine za neblokirajuću komunikaciju za razmenu poruka..

2.2. Delovi koje treba paralelizovati

2.2.1. 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. Tako smo zaključili (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++)
{
    prime_inv[j] = 1.0 / (double)(prime(j + 1));
}
```

I narednu unutrašnju 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.

2.2.2. Način paralelizacije

Paralelizacija je odrađena ručnom raspodelom posla (što ujednačenijom) između procesa, pri čemu su korišćene standardne funkcije iz MPI kako za kolektivnu, tako i za pojedinačnu komunikaciju između procesa.

2.3. Rezultati

2.3.1. Logovi izvršavanja

```
Sequential implementation execution time: 0.001752s
Parallel implementation (one process) execution time: 0.000647s
Test PASSED
Parallel implementation (two processes) execution time: 0.000967s
Test PASSED
Parallel implementation (four processes) execution time: 0.001479s
Test PASSED
Parallel implementation (eight processes) execution time: 0.002248s
Test PASSED
```

Izvršavanje komande ./halton 10

```
Sequential implementation execution time: 0.019810s
Parallel implementation (one process) execution time: 0.033814s
Test PASSED
Parallel implementation (two processes) execution time: 0.026647s
Test PASSED
Parallel implementation (four processes) execution time: 0.021350s
Test PASSED
Parallel implementation (eight processes) execution time: 0.017296s
Test PASSED
```

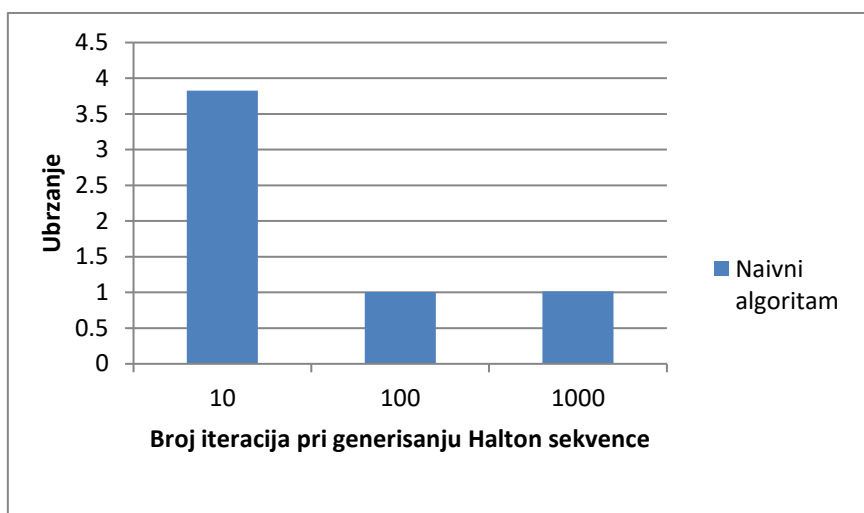
Izvršavanje komande ./halton 100

```
Sequential implementation execution time: 2.468872s
Parallel implementation (one process) execution time: 1.946138s
Test PASSED
Parallel implementation (two processes) execution time: 1.515994s
Test PASSED
Parallel implementation (four processes) execution time: 0.929994s
Test PASSED
Parallel implementation (eight processes) execution time: 0.877080s
Test PASSED
```

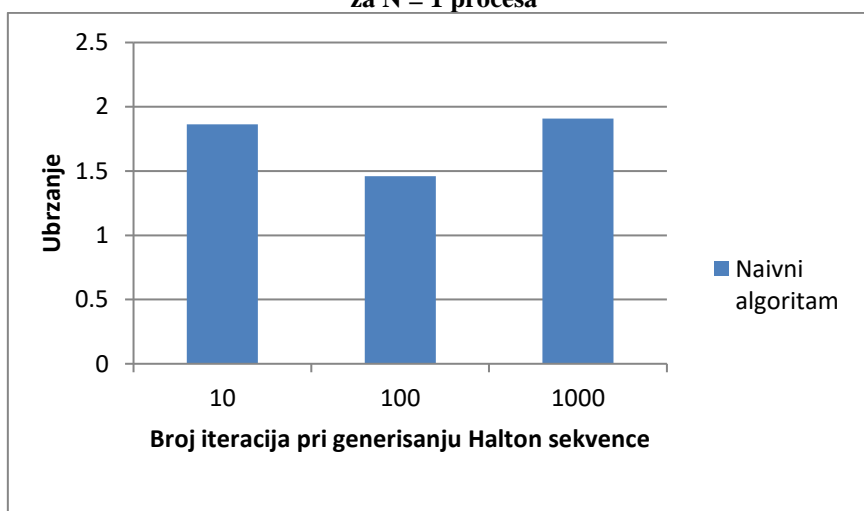
Izvršavanje komande ./halton 1000

2.3.2. Grafici ubrzanja

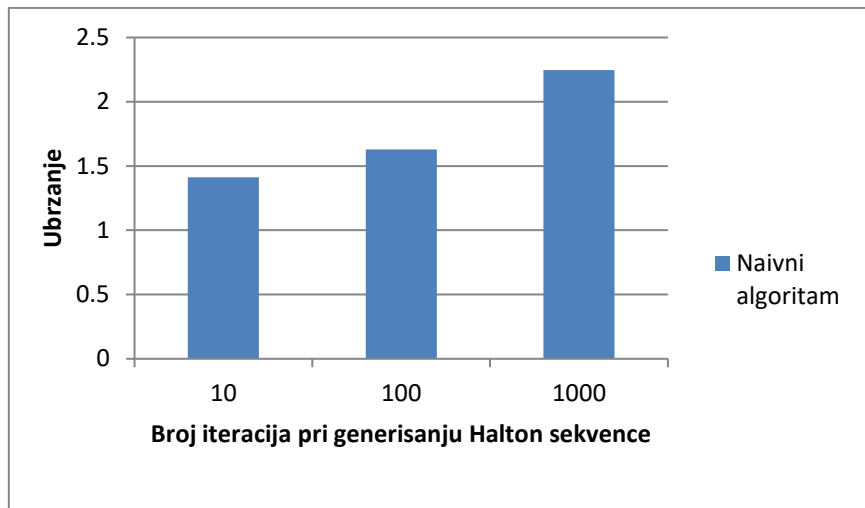
U okviru ove sekcije su dati grafici ubrzanja u odnosu na sekvencijalnu implementaciju.



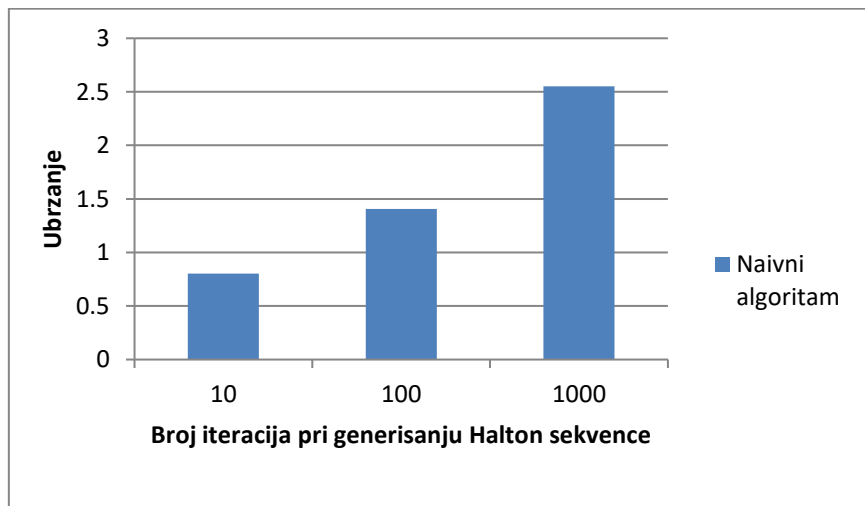
Slika 1. Grafik zavisnosti ubrzanja naivnog algoritma od broja iteracija pri generisanju Halton sekvence za N = 1 procesa



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



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



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

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.

Ulazni parametri 10 i 100 nisu preterano relevantni za diskusiju jer su vrlo mali, i nikakvo značajno ubrzanje ne donosi paralelizacija zbog overhead-a koji MPI donosi. Detektovano ubrzanje za parametar 10 pripisujemo prosto različitim trenucima merenja tog vremena i vremena sekvencijalne implementacije (pri različitom opterećenju rtidev5 računara). Vidimo da ubrzanje za

parametar 1000 uvek raste sa povećanjem broja procesa, što je odlično. Opet, bolje je bilo kod OpenMP.

3. PROBLEM 3 – SIMULACIJA KRETANJA N TELA (N-BODY PROBLEM)

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

Skripta run pokreće simulaciju za različite parametre, i nakon toga, za određene simulacije poziva python kod koji kreira gifove.

3.2. Delovi koje treba paralelizovati

3.2.1. 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.

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.

3.2.2. Način paralelizacije

Paralelizacija je određena ravnomernom ručnom podelom posla i koristeći MPI rutine za kolektivnu komunikaciju. Svi procesi učestvuju u obradi, uključujući i MASTER proces.

3.3. Rezultati

3.3.1. Logovi izvršavanja

```
Running ./nbody with 30 particles, 100 iterations, and saving to simulation_1
folder
Sequential implementation execution time: 0.007780s
Parallel implementation (one process) execution time: 0.004377s
Test PASSED
Parallel implementation (two processes) execution time: 0.004900s
Test PASSED
Parallel implementation (four processes) execution time: 0.004370s
Test PASSED
Parallel implementation (eight processes) execution time: 0.004600s
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 process) execution time: 0.068007s
Test PASSED
Parallel implementation (two processes) execution time: 0.056577s
Test PASSED
Parallel implementation (four processes) execution time: 0.068677s
Test PASSED
Parallel implementation (eight processes) execution time: 0.043350s
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 process) execution time: 6.623248s
Test PASSED
Parallel implementation (two processes) execution time: 3.488469s
Test PASSED
Parallel implementation (four processes) execution time: 1.963410s
Test PASSED
Parallel implementation (eight processes) execution time: 1.668833s
Test PASSED
```

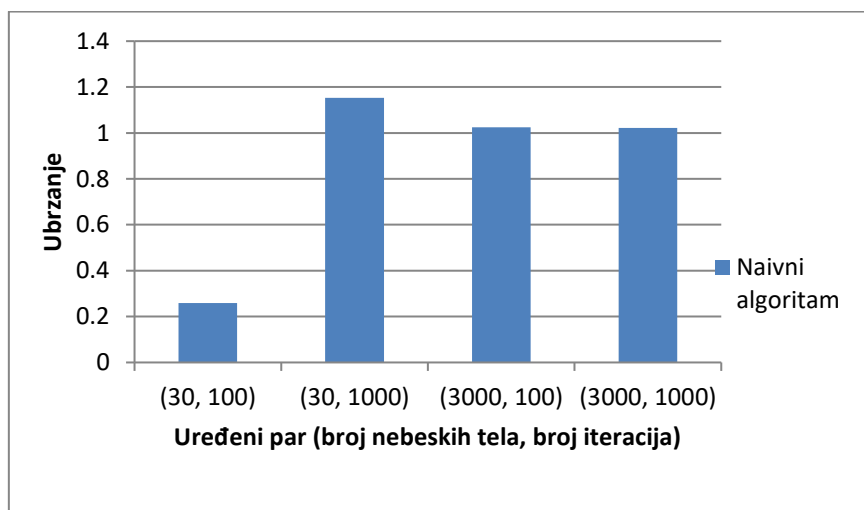
Izvršavanje komande ./nbody 3000 100 simulation_3

```
Running ./nbody with 3000 particles, 1000 iterations, and saving to simulation_4
folder
Sequential implementation execution time: 23.769746s
Parallel implementation (one process) execution time: 89.042483s
Test PASSED
Parallel implementation (two processes) execution time: 40.363270s
Test PASSED
Parallel implementation (four processes) execution time: 22.983543s
Test PASSED
Parallel implementation (eight processes) execution time: 21.320739s
Test PASSED
```

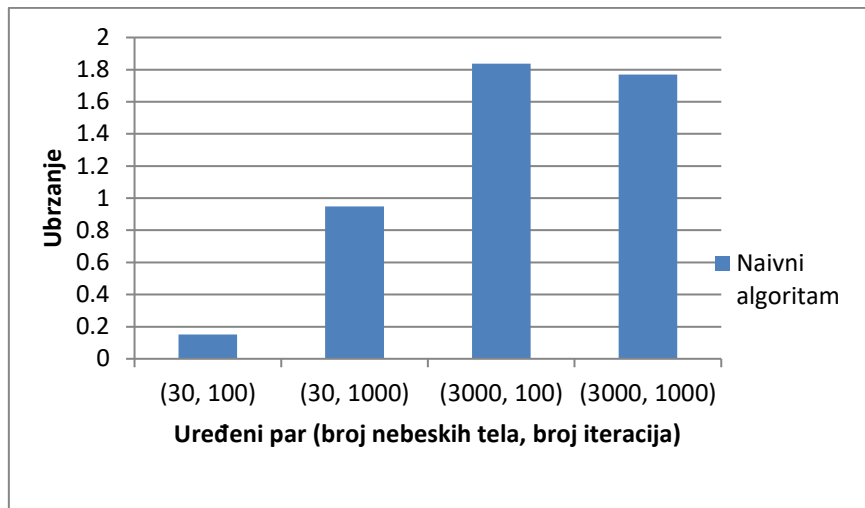
Izvršavanje komande ./nbody 3000 1000 simulation_4

3.3.2. Grafici ubrzanja

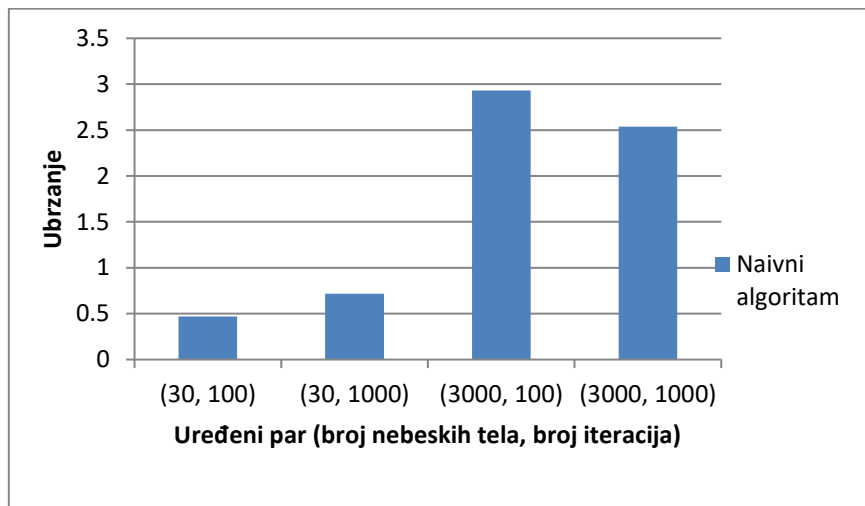
U okviru ove sekcije su dati grafici ubrzanja u odnosu na sekvencijalnu implementaciju.



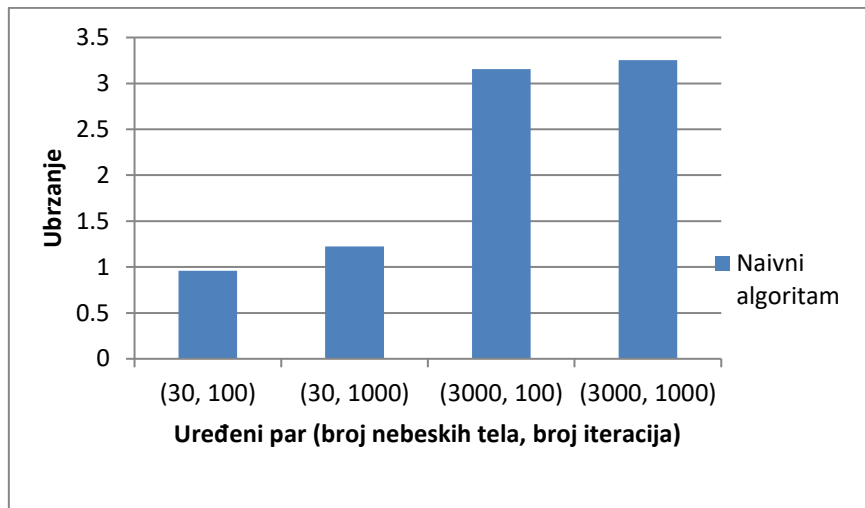
Slika 1. Grafik zavisnosti ubrzanja naivnog algoritma od broja nebeskih tela i broja iteracija za $N = 1$ procesa



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



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



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

3.3.3. Diskusija dobijenih rezultata

Primećujemo da nikakvo značajno ubrzanje nije postignuto u ovom zadatku u odnosu na istinsku sekvencijalnu implementaciju (izvršenu van MPI sveta). To je i logično – ograničavajući faktor pri MPI paralelizaciji ovog zadatka jeste sama priroda problema koja zahteva da u svakoj iteraciji svi procesi moraju da imaju kod sebe ažuran niz nebeskih tela sa svim ažurnim koordinatama (da bi mogli da preračunaju nove koordinate brzina, trebaju im sve koordinate položaja). Ova činjenica je uzrok velikog overhead-a kod MPI jer u svakoj iteraciji petlje mora da se upravo šalje ovaj veliki niz svima, što nije naravno bio slučaj kod OpenMP gde su sve niti radile nad istim jednim nizom, dok kod MPI svaki proces radi nad svojom lokalnom kopijom niza.

4. PROBLEM 4 – SIMULACIJA KRETANJA N TELA (N-BODY PROBLEM)

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

Skripta run pokreće simulaciju za različite parametre, i nakon toga, za određene simulacije poziva python kod koji kreira gifove.

4.2. Delovi koje treba paralelizovati

4.2.1. 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.

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.

4.2.2. Način paralelizacije

Paralelizacija je odrađena koristeći manager – worker model u kom je MASTER proces zadužen za raspodelu posla i signalizaciju da li je još posla potrebno da se uradi, a ostali procesi (radnici) su zaduženi za samu obradu.

4.3. Rezultati

4.3.1. Logovi izvršavanja

```
Running ./nbody with 30 particles, 100 iterations, and saving to simulation_1
folder
Sequential implementation execution time: 0.007780s
Parallel implementation (two processes) execution time: 0.009032s
Test PASSED
Parallel implementation (four processes) execution time: 0.005065s
Test PASSED
Parallel implementation (eight processes) execution time: 0.008151s
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 (two processes) execution time: 0.060754s
Test PASSED
Parallel implementation (four processes) execution time: 0.070935s
Test PASSED
Parallel implementation (eight processes) execution time: 0.060809s
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 (two processes) execution time: 6.683932s
Test PASSED
Parallel implementation (four processes) execution time: 2.459006s
Test PASSED
Parallel implementation (eight processes) execution time: 2.307185s
Test PASSED
```

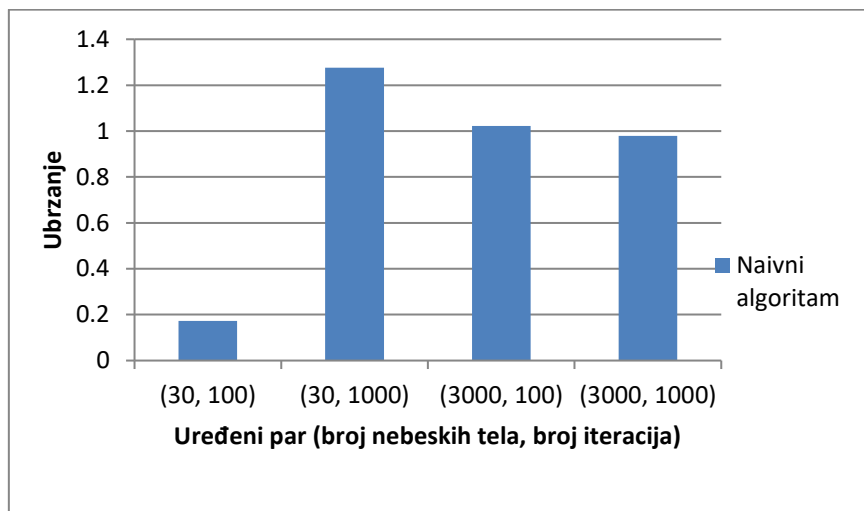
Izvršavanje komande ./nbody 3000 100 simulation_3

```
Running ./nbody with 3000 particles, 1000 iterations, and saving to simulation_4
folder
Sequential implementation execution time: 23.769746s
Parallel implementation (two processes) execution time: 70.209837s
Test PASSED
Parallel implementation (four processes) execution time: 28.602495s
Test PASSED
Parallel implementation (eight processes) execution time: 24.621491s
Test PASSED
```

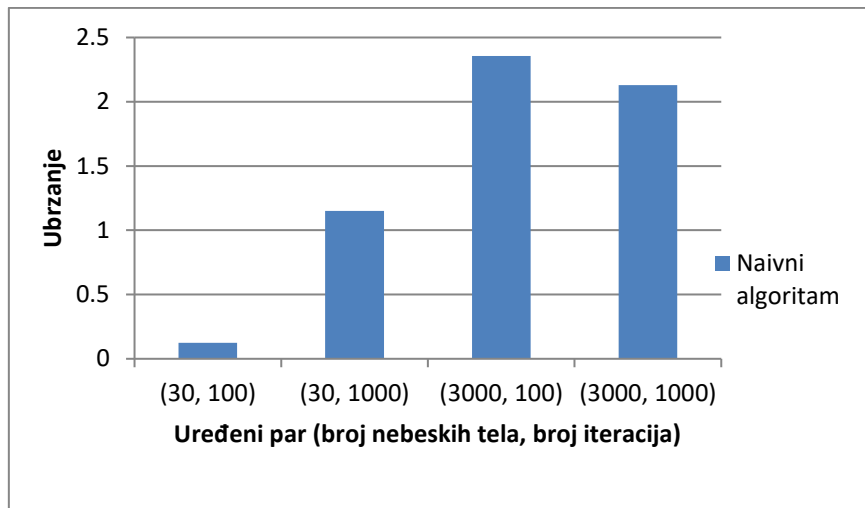
Izvršavanje komande ./nbody 3000 1000 simulation_4

4.3.2. Grafici ubrzanja

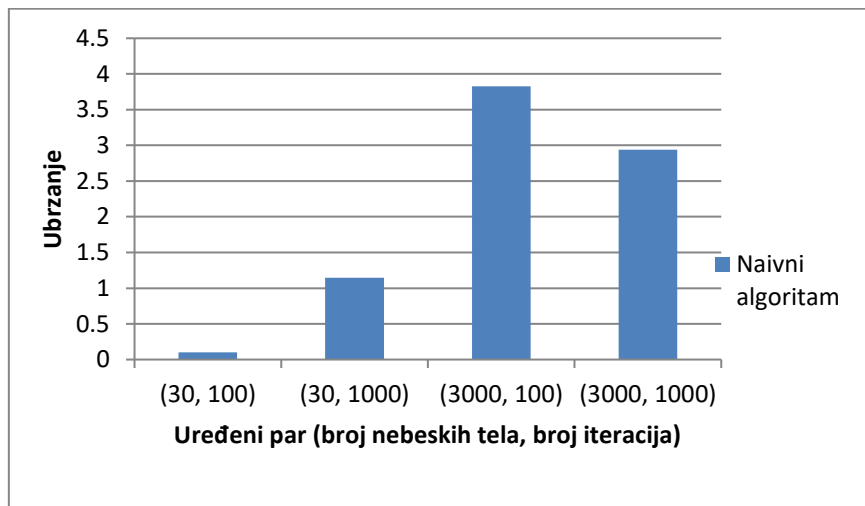
U okviru ove sekcije su dati grafici ubrzanja u odnosu na sekvencijalnu implementaciju.



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



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



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

4.3.3. Diskusija dobijenih rezultata

Primećujemo da nikakvo značajno ubrzanje nije postignuto u ovom zadatku u odnosu na istinsku sekvencijalnu implementaciju (izvršenu van MPI sveta). To je i logično – ograničavajući faktor pri MPI paralelizaciji ovog zadatka jeste sama priroda problema koja zahteva da u svakoj iteraciji svi procesi moraju da imaju kod sebe ažuran niz nebeskih tela sa svim ažurnim koordinatama (da bi mogli da preračunaju nove koordinate brzina, trebaju im sve koordinate položaja). Ova činjenica je uzrok velikog overhead-a kod MPI jer u svakoj iteraciji petlje mora da se upravo šalje ovaj veliki niz svima, što nije naravno bio slučaj kod OpenMP gde su sve niti radile nad istim jednim nizom, dok kod MPI svaki proces radi nad svojom lokalnom kopijom niza.