UNIVERZITET U BEOGRADU - ELEKTROTEHNIČKI FAKULTET

MULTIPROCESORKI SISTEMI (13S114MUPS, 13E114MUPS)

DOMAĆI ZADATAK 2 – MPI

Izveštaj o urađenom domaćem zadatku

Predmetni asistent:

doc. dr Marko Mišić

Kandidati:

Marija Kostić 2015/0096

Stefan Milanović 2015/0361

Beograd, decembar 2018.

SADRŽAJ

SADRŽAJ **2**

1. PROBLEM 1 - SGEMM **3**

1.1. TEKST PROBLEMA 3
1.2. DELOVI KOJE TREBA PARALELIZOVATI 3
1.2.1. Diskusija *3*
1.2.2. Način paralelizacije *4*
1.3. REZULTATI 5
1.3.1. Logovi izvršavanja *6*
1.3.2. Tabela trajanja programa *9*
1.3.3. Grafici ubrzanja *9*
1.3.4. Diskusija dobijenih rezultata *12*

2. PROBLEM 2 - JACOBI **13**

2.1. TEKST PROBLEMA 13
2.2. DELOVI KOJE TREBA PARALELIZOVATI 13
2.2.1. Diskusija *13*
2.2.2. Način paralelizacije *14*
2.3. REZULTATI 15
2.3.1. Logovi izvršavanja *15*
2.3.2. Tabela trajanja programa *17*
2.3.3. Grafici ubrzanja *17*
2.3.4. Diskusija dobijenih rezultata *18*

3. PROBLEM 3 - KMEANS **19**

3.1. TEKST PROBLEMA 19
3.2. DELOVI KOJE TREBA PARALELIZOVATI 19
3.2.1. Diskusija *19*
3.2.2. Način paralelizacije *20*
3.3. REZULTATI 23
3.3.1. Logovi izvršavanja *23*
3.3.2. Tabela trajanja programa *27*
3.3.3. Grafici ubrzanja *27*
3.3.4. Diskusija dobijenih rezultata *28*

1. PROBLEM 1 - SGEMM

U okviru ovog poglavlja je dat kratak izveštaj u vezi rešenja zadatog problema 1.

1.1. Tekst problema

Paralelizovati program koji vrši jednostavno generalizovano množenje matrica u jednostrukoj preciznosti *Single precision floating General Matrix Multiply* (SGEMM). SGEMM operacije je definisana sledećom formom:

$$C \leftarrow \alpha \cdot A \cdot B + \beta \cdot C$$

Program se nalazi u datoteci sgemm.c u arhivi koja je priložena uz ovaj dokument. Proces sa rangom 0 treba da učita ulazne podatke, raspodeli posao ostalim procesima, na kraju prikupi dobijene rezultate i ravnopravno učestvuje u obradi. Za razmenu podataka, koristiti rutine za kolektivnu komunikaciju. Program testirati sa parametrima koji su dati u datoteci run. [1, N]

1.2. Delovi koje treba paralelizovati

1.2.1. Diskusija

Kod koji rešava ovaj problem se može razbiti na tri velike celine – najpre se vrši učitavanje ulaznih matrica iz dva tekstualna fajla, vrši se obrada ulaznih podataka u vidu kreiranja izlazne matrice koja predstavlja proizvod unetih matrica, i na samom kraju se vrši ispis izlazne matrice u tekstualni fajl.

Prvu i treću celinu nije moguće paralelizovati – ovi delovi koda predstavljaju rad sa I/O sistemom koji se mora obaviti isključivo unutar jedne niti.

Jedini deo koda nad kojim se može vršiti paralelizacija jeste deo koda koji radi obradu, tj. funkcija *void basicSgemm(args)*. U skladu sa postavkom zadatka, paralelizacija će se vršiti na dva načina – prvi način deli posao ravnopravno između procesa (uključujući i master proces, koji vrši učitavanje i prikupljanje rezultata, a takođe vrši i obradu), dok drugi način podrazumeva korišćenje *manager-worker* modela, pri čemu master predstavlja proces gospodara (menadžera).

1.2.2. Način paralelizacije

Zadatak 1

Rešenje ovog zadatka nalazi se u datoteci *dz2z1.cpp*. U ovom zadatku svaki proces obrađuje *chunk* redova izlazne matrice koristeći deo matrice A koji njemu pripada i kopiju matrice B koja mu je neophodna za samu obradu podataka.

Ovo je postignuto tako što svi procesi pozivaju funkciju *basicSgemm_par().* Unutar ove funkcije svaki proces najpre odredi svoj rank unutar komunikatora i broj procesa u komunikatoru i nakon toga pozivaju *broadcast* funkciju za sve neophodne podatke. Ove podatke jedino poseduje master proces koji ih šalje svim ostalim procesima. U ovom trenutku izvršena je raspodela samo neophodnih prostih promenljivih i sada se alociraju tri bafera (ovo radi svaki proces, uključujući i master proces). Bafer A sadrži odgovarajuće redove originalne matrice A koju ovaj proces treba da obradi, bafer B sadrži kopiju matrice B, i bafer C služi za lokalno čuvanje rezultata koji će kasnije biti poslat master procesu na sakupljanje. Matrica A se od mastera šalje svim procesima koristeći *Scatter* funkciju, a nakon ove raspodele svaki proces odradi svoj deo obrade i zatim pozivom funkcije *Gather* vraća deo rezultata masteru koji to smešta u odgovarajući deo matrice C koja predstavlja rezultat paralelne obrade.

Bitno je primetiti da je matrica A linearizovana *po kolonama.* Kako bi se omogućilo slanje samo određenih redova ove matrice svim procesima korišćen je korisnički definisan vektorski tip *row_matA*. Ovaj tip uzima *k* blokova dužine *1* sa *stride-om m* između blokova čime se postiže dohvatanje reda matrice A. Ukoliko se uzme *chunk* za dužinu bloka umesto 1, dobija se dohvatanje *chunk* redova matrice A, i upravo ovo se koristi za raspodelu matrice na delove procesima.

Zadatak 2

Rešenje ovog zadatka nalazi se u datoteci *dz2z2.cpp*. U ovom zadatku neophodno je koristiti *manager*-worker model – master čita neophodne podatke i generiše poslove koje će radnici obrađivati. Po prijemu podataka radnik vrši obradu, vraća rezultat gospodaru i šalje signal da je spreman da primi sledeći posao. Sada master ne vrši obradu kao u prethodnom zadatku, već samo radi podelu posla i sakupljanje rezultata sve dok problem nije u celosti rešen.

Main funkcija izgleda identično kao u prethodnom zadatku. Svi pozivaju funkciju *basicSgemm_par()* unutar koje će biti izvršena raspodela posla. Opet se prvo radi *broadcast* svih neophodnih prostih promenljivih, kao i matrice B koja je neophodna svim procesima za izračunavanje rezultata. Za razliku od prethodnog zadatka, sada se ne šalju odmah delovi matrice A ostalim procesima, već se matrica A šalje red po red od mastera ka radnicima i to samo kada je radnik spreman da krene nov ciklus obrade. Takođe u ovom zadatku se sam kod procesa sa rankom master i kod drugih procesa dosta razlikuje.

Master sada prati koliko je taskova poslato radnicima i koliko rezultata je primljeno, a zatim u odgovarajućim petljama radi slanje podataka za obradu i primanje rezultata. Jedan task predstavlja izračunavanje jednog reda matrice C (detaljnija diskusija o ovome će se naći u poglavlju *1.3.4.*). Kada se šalje task radniku, najpre se šalje redni broj reda matrice A koji će radnik obraditi, a nakon toga i sam red matrice. Ovo se radi pomoću poziva asinhrone *Isend* funkcije Kada se prima rezultat od radnika (ovo radi samo master), on poziva sinhronu *Recv* funkciju i to sa parametrom *MPI_ANY_SOURCE* što omogućava prijem od bilo kog radnika koji je završio. Nakon ovog prijema povećava se brojač primljenih rezultata od svih radnika, a rezultat koji radnik šalje se smešta na odgovarajuće mesto u matrici C. Ukoliko ima još posla, master radniku koji je upravo završio šalje novi red matrice A na obradu. Ukoliko više nema posla, master šalje *end* poruku radnicima.

Što se koda procesa radnika tiče, on ima odgovarajuće bafere za matricu A i C i jedan flag *needed*, koji je inicijalno postavljen na *true* i označava to da li radnik uopšte treba da čeka novi task ili treba da završi rad. Radnik se vrti u ovoj petlji dokle god je *needed* ispunjeno i, analogno slanju koje se vrši u masteru, prvo prima redni broj reda matrice A a zatim i sam red, vrši obradu koja je identična kao u prvom zadatku, i zatim masteru šalje rezultat. Kada se šalje rezultat, prvo se šalje redni broj reda (kako bi master znao gde da smesti rezultat radnika), a zatim i sam rezultat (koji je radnik kod sebe lokalno čuvao u baferu C).

1.3. Rezultati

U okviru ove sekcije su izloženi rezultati paralelizacije problema 1 koristeći rešenja prethodne dve implementacije.

1.3.1. Logovi izvršavanja

Ovde su dati logovi izvršavanja za definisane test primere i različit broj niti.

Listing 1. Zadatak 1 – 1 nit

```
Opening file:data/small/input/matrix1.txt
Matrix dimension: 128x96
Opening file:data/small/input/matrix2t.txt
Matrix dimension: 160x96
Opening file:result_small_par.txt for write.
Matrix dimension: 128x160
**********DZ2Z1**********
Elapsed time - SEQ: 0.00781145.
Elapsed time - PAR(1): 0.00804341.
TEST PASSED
**********
Opening file:data/medium/input/matrix1.txt
Matrix dimension: 1024x992
Opening file:data/medium/input/matrix2t.txt
Matrix dimension: 1056x992
Opening file:result_medium_par.txt for write.
Matrix dimension: 1024x1056
**********DZ2Z1**********
Elapsed time - SEQ: 12.9546
Elapsed time - PAR(1): 11.9514.
TEST PASSED
**********
```

Listing 2. Zadatak 1 – 2 niti

```
Opening file:data/small/input/matrix1.txt
Matrix dimension: 128x96
Opening file:data/small/input/matrix2t.txt
Matrix dimension: 160x96
Opening file:result_small_par.txt for write.
Matrix dimension: 128x160
**********DZ2Z1**********
Elapsed time - SEQ: 0.0084999.
Elapsed time - PAR(2): 0.00456744.
TEST PASSED
**********
Opening file:data/medium/input/matrix1.txt
Matrix dimension: 1024x992
```

```
Opening file:data/medium/input/matrix2t.txt
Matrix dimension: 1056x992
Opening file:result_medium_par.txt for write.
Matrix dimension: 1024x1056
**********DZ2Z1**********
Elapsed time - SEQ: 12.106.
Elapsed time - PAR(2): 3.442.
TEST PASSED
**********
```

Listing 3. Zadatak 1 – 4 niti

```
Opening file:data/small/input/matrix1.txt
Matrix dimension: 128x96
Opening file:data/small/input/matrix2t.txt
Matrix dimension: 160x96
Opening file:result_small_par.txt for write.
Matrix dimension: 128x160
**********DZ2Z1**********
Elapsed time - SEQ: 0.126872.
Elapsed time - PAR(4): 0.0044802.
TEST PASSED
**********
Opening file:data/medium/input/matrix1.txt
Matrix dimension: 1024x992
Opening file:data/medium/input/matrix2t.txt
Matrix dimension: 1056x992
Opening file:result_medium_par.txt for write.
Matrix dimension: 1024x1056
**********DZ2Z1**********
Elapsed time - SEQ: 11.427.
Elapsed time - PAR(4): 1.91111.
TEST PASSED
**********
```

Listing 4. Zadatak 2 – 1 nit

```
Inadequate number of processes
--------------------------------------------------------------------------
MPI_ABORT was invoked on rank 0 in communicator MPI_COMM_WORLD
with errorcode -10.

NOTE: invoking MPI_ABORT causes Open MPI to kill all MPI processes.
```

```
You may or may not see output from other processes, depending on
exactly when Open MPI kills them.
--------------------------------------------------------------------------
Inadequate number of processes
--------------------------------------------------------------------------
MPI_ABORT was invoked on rank 0 in communicator MPI_COMM_WORLD
with errorcode -10.

NOTE: invoking MPI_ABORT causes Open MPI to kill all MPI processes.
You may or may not see output from other processes, depending on
exactly when Open MPI kills them.
--------------------------------------------------------------------------
```

Listing 5. Zadatak 2 – 2 niti

```
Opening file:data/small/input/matrix1.txt
Matrix dimension: 128x96
Opening file:data/small/input/matrix2t.txt
Matrix dimension: 160x96
Opening file:result_small_par.txt for write.
Matrix dimension: 128x160
**********DZ2Z2**********
Elapsed time - SEQ: 0.0125579.
Elapsed time - PAR(2): 0.00879433.
TEST PASSED
**********
Opening file:data/medium/input/matrix1.txt
Matrix dimension: 1024x992
Opening file:data/medium/input/matrix2t.txt
Matrix dimension: 1056x992
Opening file:result_medium_par.txt for write.
Matrix dimension: 1024x1056
**********DZ2Z2**********
Elapsed time - SEQ: 12.0136.
Elapsed time - PAR(2): 3.66735.
TEST PASSED
**********
```

Listing 6. Zadatak 2 – 4 niti

```
Opening file:data/small/input/matrix1.txt
Matrix dimension: 128x96
Opening file:data/small/input/matrix2t.txt
```

Matrix dimension: 160x96

Opening file:result_small_par.txt for write.

Matrix dimension: 128x160

**********DZ2Z2**********

Elapsed time - SEQ: 0.00975984.

Elapsed time - PAR(4): 0.00312711.

TEST PASSED

****

Opening file:data/medium/input/matrix1.txt

Matrix dimension: 1024x992

Opening file:data/medium/input/matrix2t.txt

Matrix dimension: 1056x992

Opening file:result_medium_par.txt for write.

Matrix dimension: 1024x1056

**********DZ2Z2**********

Elapsed time - SEQ: 13.3526.

Elapsed time - PAR(4): 1.40961.

TEST PASSED

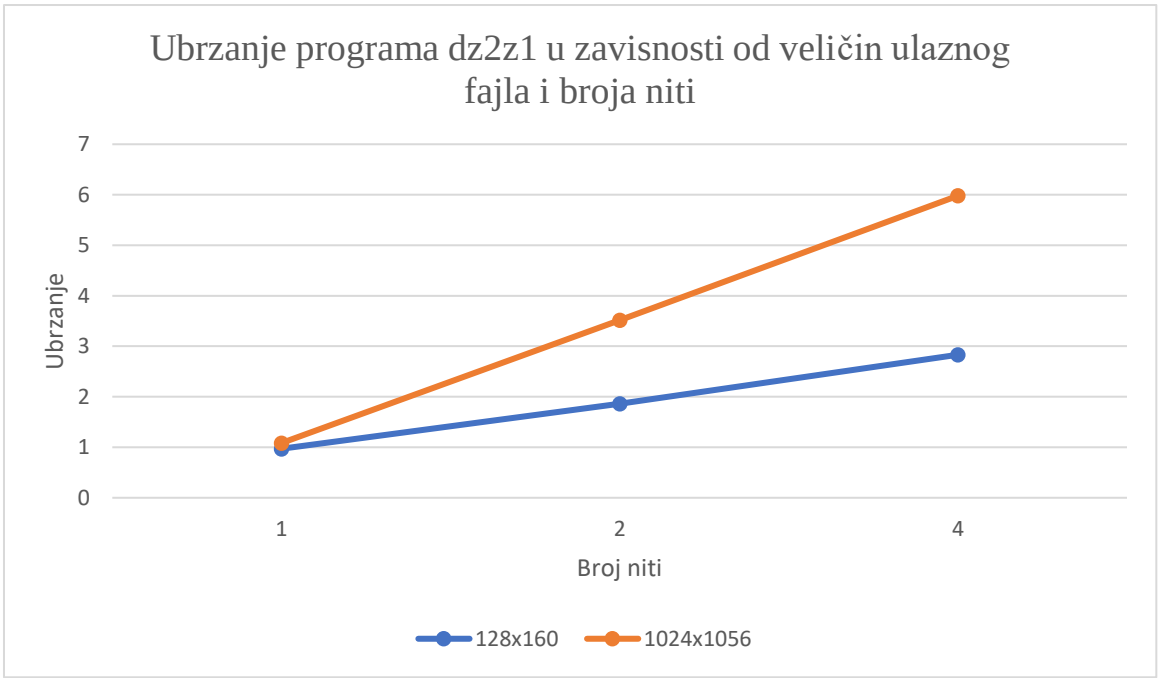
****

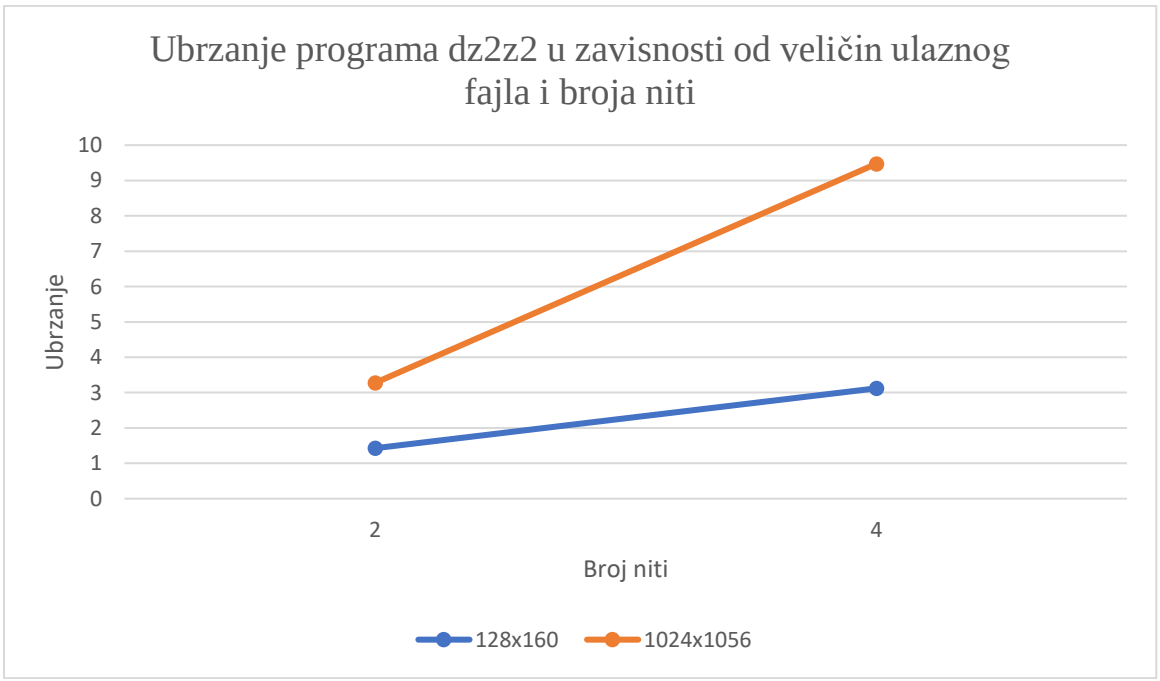
1.3.2. Tabela trajanja programa

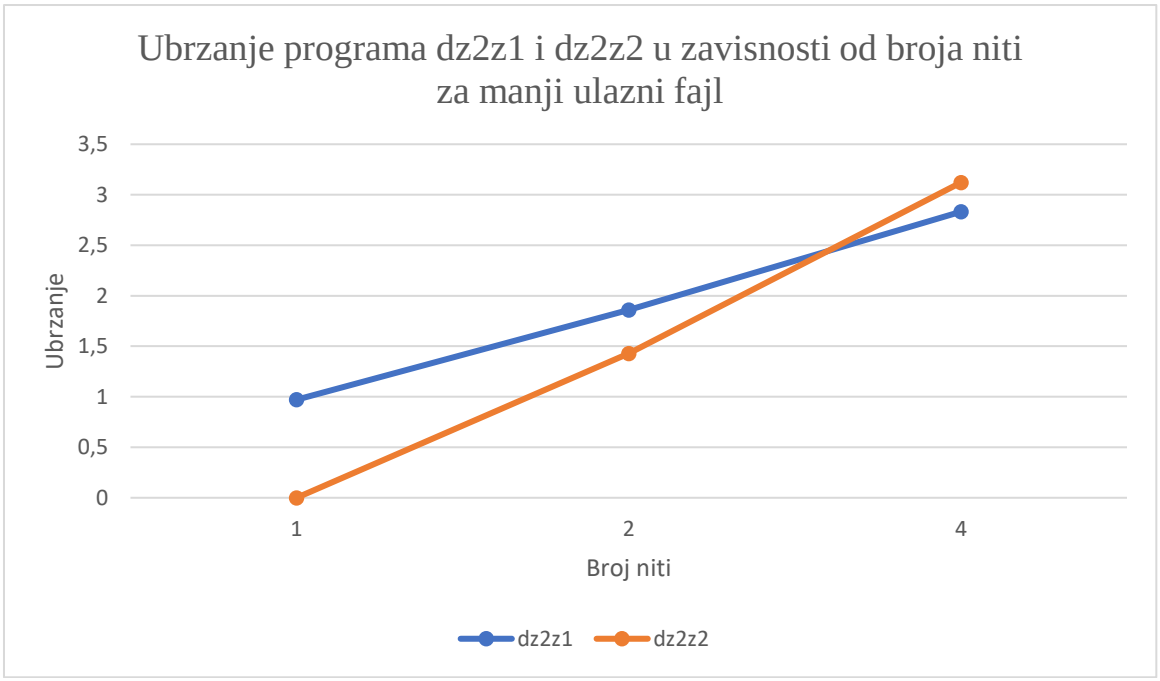
	small input			medium input		
	Sekvencijalno vreme	Paralelno vreme	Ubrzanje	Sekvencijalno vreme	Parelelno vreme	Ubrzanje
Zadatak 1 – 1 nit	0,00781145	0,00804341	0,97116148	12,9546000	11,9514000	1,08393996
Zadatak 1 – 2 niti	0,00849990	0,00456744	1,86097683	12,1060000	3,44200000	3,51714120
Zadatak 1 – 4 niti	0,12687200	0,04480200	2,83183786	11,4270000	1,91111100	5,97924453
Zadatak 2 – 2 niti	0,01255790	0,00879433	1,42795415	12,0136000	3,66735000	3,27582587
Zadatak 2 – 4 niti	0,00975984	0,00312710	3,12105145	13,3526000	1,40961000	9,47254914

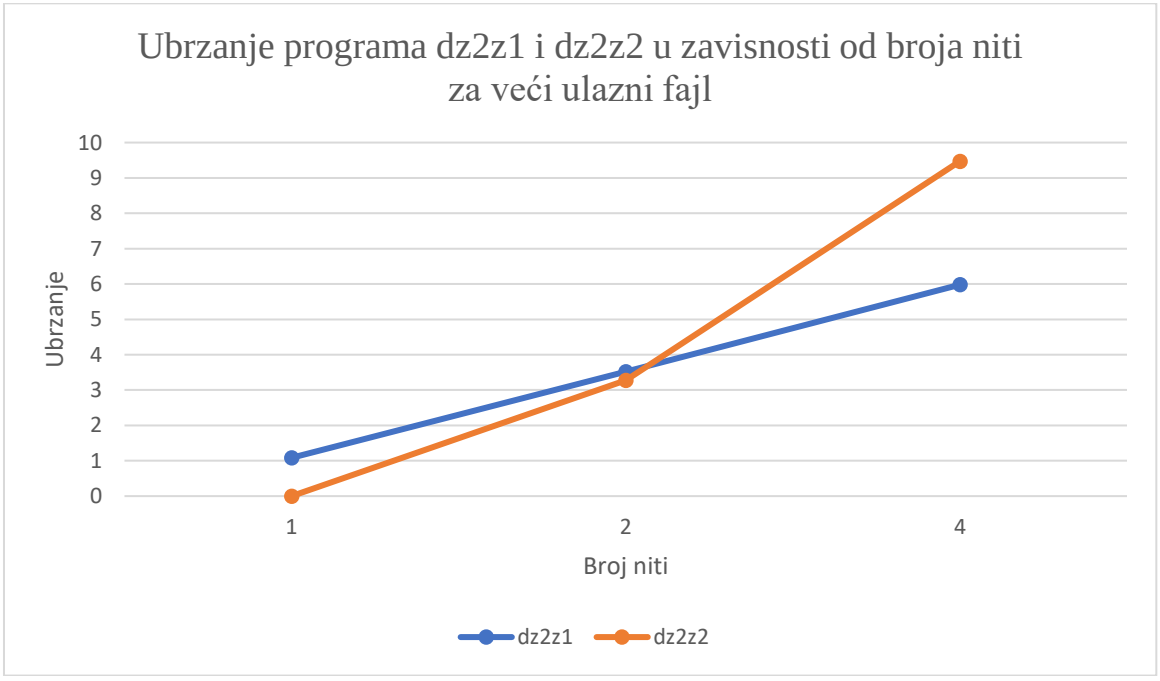
1.3.3. Grafici ubrzanja

U okviru ove sekcije su dati grafici ubrzanja paralelnih implementacija u odnosu na sekvencijalnu implementaciju. Ubrzanje na svim narednim graficima predstavljeno je kao odnos trajanja sekvencijalne implementacije programa i trajanja paralelizovane implementacije.


Ubrzanje programa dz2z1 u zavisnosti od veličin ulaznog fajla i broja niti
Ubrzanje
0
1
2
3
4
5
6
7
1
2
4
Broj niti
128x160
1024x1056


Ubrzanje programa dz2z2 u zavisnosti od veličin ulaznog fajla i broja niti
Ubrzanje
0
1
2
3
4
5
6
7
8
9
10
2
4
Broj niti
128x160
1024x1056


Ubrzanje programa dz2z1 i dz2z2 u zavisnosti od broja niti za manji ulazni fajl
3,5
3
2,5
2
1,5
1
0,5
0
Ubrzanje
1
2
4
Broj niti
dz2z1
dz2z2


Ubrzanje programa dz2z1 i dz2z2 u zavisnosti od broja niti za veći ulazni fajl
10
9
8
7
6
5
4
3
2
1
0
Ubrzanje
1
2
4
Broj niti
dz2z1
dz2z2

1.3.4. Diskusija dobijenih rezultata

Rezultati se mogu posmatrati tako da se zasebno gledaju izmerene performanse kada se na ulaz sistema dovedu matrice malih dimenzije i kada se na ulaz sistema dovedu matrice većih dimenzija.

Što se manjeg ulaznog fajla tiče, iz dobijenih rezultata se može videti da se najveće ubrzanje prilikom paralelizacije dobija za 4 procesa. Za veći broj procesa bi performanse paralelizovanog koda verovatno opale zbog toga što bi *overhead*-i paralelizacije, kreiranja i pokretanja tolikog broja procesa bili dosta veliki i ne bi bili opravdani nad ovako malim skupom ulaznih podataka.

Za veće ulazne matrice performanse paralelnih programa su mnogo bolje nego kada su ulazi manje matrice. To je zato što je vreme izračunavanja računskih operacija verovatno mnogo veće od vremena koje je potrebno za kreiranje procesa i komunikaciju.

U prvom zadatku su korišćene rutine za kolektivnu komunikaciju uz podjednaku podelu posla svim procesima, dok se u drugom zadatku koristi *manager-worker* model. Za manje ulazne podatke prvi zadatak ima bolje performanse zato što ima mnogo manje komunikacije između procesa – jednom na početku i jednom na kraju. U drugom zadatku je *overhead* oko slanja i prijema *task*-ova za ovako male ulazne podatke preveliki. Suprotno tome, kod većih matrica komunikacija u prvom zadatku je spora zato što zahteva prenos veoma velikih poruka (*Broadcast*, *Scatter* i *Gather* celih matrica). U ovom slučaju drugi zadatak postiže mnogo bolje performanse koje se povećavaju sa porastom broja procesa, zato što je prenos manjih poruka mnogo brži iako ih ima više.

Za drugi zadatak nisu prikazani rezultati izvršavanja samo sa jednim procesom zato što u tom slučaju nije moguće iskoristiti *manager-worker* model kod koga MASTER proces samo deli zadatke i prikuplja rezultate. Ako postoji samo jedan proces on je MASTER koji nema kome da pošalje zadatke.

Što se tiče veličine *task*-a u drugom zadatku eksperimentalno je utvrđeno da program sa *task*-om veličine jednog reda izlazne matrice ima najbolje performanse. U obzir su uzeti i *task*-ovi veličine jednog polja izlazne matrice i više redova izlazne matrice.

2. PROBLEM 2 - JACOBI

U okviru ovog poglavlja je dat kratak izveštaj u vezi rešenja zadatog problema 2.

2.1. Tekst problema

Paralelizovati program koji vrši jednostavno generalizovano množenje matrica u jednostrukoj Paralelizovati program koji rešava sistem linearnih jednačina *A * x = b* Jakobijevim metodom. Kod koji treba paralelizovati se nalazi u datoteci *jacobi.c* u arhivi koja je priložena uz ovaj dokument. Ukoliko je moguće, koristiti rutine za neblokirajuću komunikaciju za razmenu poruka. Program testirati sa parametrima koji su dati u datoteci run. [1, N]

2.2. Delovi koje treba paralelizovati

2.2.1. Diskusija

Za razliku od paralelne implementacije koristeći OpenMP, ovde je neophodno više izmeniti način obrade kako bi se omogućila najveća moguća paralelizacija. Moguće je izbeći korišćenje niza *b* u potpunosti – ovaj niz se nikad ne menja (iz njega se samo čita) i od početka programa ima vrednosti 0 za sve elemente niza osim za poslednji, koji sadrži vrednost *n+1*. Samim tim moguće je ovaj niz ukloniti, a zatim koristiti vrednost *n+1* umesto *b[i]* samo u procesu koji obrađuje poslednji element poslednjeg dela niza *x*. Ovim se izbegava *scatter* niza *b* od mastera ka ostalim procesima ili njegova inicijalizacija u svim procesima.

Pored ovoga, moguće je izbeći dosta komunikacije oko prepisivanja niza *xnew* u niz *x* tako što se kod paralelizuje tako da proces koji nije master svoj rezultat šalje samo kada se sve iteracije petlje *for (it = 0; it < m; i++)* završe. To se omogućava tako što deo obrade koji generiše vrednost promenljive *r* lokalno (dakle, u okviru jednog procesa) koristi svoj niz *xnew* (u kodu je ovo *xnew-buffer*) umesto niza *x* (što je urađeno u sekvencijalnoj verziji gde se prepisivanje iz niza *xnew* u niz *x* radi pre računanja vrednosti r). Nakon ovog koraka, proces lokalno vrši prepisivanje iz svog niza *xnew* u svoj niz *x* (u kodu su ovo baferi). Ovo omogućava odlaganje komunikacije između drugih procesa i mastera do samog kraja obrade.

Izbegnuta je inicijalizacija i raspodela početnog niza *x* zato što je on inicijalno popunjen nulama. Nema potrebe zagušiti mrežu rađenjem *scatter*-a ovog niza, već je moguće da svaki

proces inicijalizuje početnu vrednost svog bafera niza *x* nulama, a zatim radi odgovarajuću obradu.

Takođe, unutar funkcija koje vrše obradu, kod je izmenjen tako da svaki proces obrađuje svoj niz *x_buffer* koji je dužine *chunk* (*chunk* se dobija kao rezultat deljenja vrednosti *n* (dužine originalnog niza *x*) sa brojem procesa koji rade obradu). Pritom se prvo vrši slanje graničnih elemenata niza (ukoliko je neophodno), zatim se vrši obrada za *i=1..chunk -2,* pa se na kraju čeka na dobijanje graničnih elemenata od okolnih procesa (objašnjeno u narednom poglavlju) i onda se vrši obrada za vrednosti *i=0* i *i=chunk-1*.

2.2.2. Način paralelizacije

Paralelizacija ovog problema funkcioniše na sličan način kao i u zadatku 1 – naime, i master radi obradu zajedno sa ostalim procesima i rezultat čuva u okviru svog lokalnog bafera x*_buffer*. Za računanje novih vrednosti tog niza koristi se pomoćni niz koji je takođe lokalan za sve procese *xnew_buffer,* a za računanje vrednosti *d* i *r* svaki proces poseduje svoje promenljive *d_local* i *r_local*. Ove dve promenljive se redukuju koristeći *Ireduce* zato što procesi mogu nastaviti ostatak obrade i ne moraju da čekaju da i ostali procesi završe odgovarajuću računicu. Master kada pozove ovu funkciju takođe poziva i *Wait* funkciju koristeći *MPIRequest* objekat koji mu je vraćen pozivom *Ireduce* funkcije kako bi sačekao da svi procesi završe računanje svojih *d_local* i *r_local* vrednosti, i zatim radi ispis redukovanih vrednosti.

Ovo predstavlja jedan deo obrade. Za sve procese obrada se radi *m* puta i sastoji se od sledećih koraka: najpre se radi *jacobi update* i računanje vrednosti *d_local,* zatim se radi računanje vrednosti *r_local,* a nakon toga se vrši prepisivanje niza *xnew_buffer* u *xnew* (ovo je izmenjeno u odnosu na sekvencijalnu implementaciju tako da se ne radi plitko kopiranje, već se samo obrću vrednosti pokazivača na ova dva niza). Posle ovoga radi se redukcija vrednosti *r_local* i *d_local* po potrebi. Ova obrada se ponavlja *m* puta, pri čemu vrednosti *m* i *n* poseduju svi procesi jer su to vrednosti prosleđene kao argumenti komandne linije.

Ovim je već postignut najveći deo paralelizacije – svaki proces obrađuje svoj deo niza i na kraju se rezultat šalje masteru na sakupljanje, ispis, i proveru tačnosti. Jedini problem koji se ovde javlja jeste da postoji zavisnost po podacima prilikom računanja vrednosti *xnew_buffer[i]* i *r_local* za graničnu vrednosti brojača *i,* a to su 0 i *chunk-1.* Zavisnost po

podacima se javlja zato što je za računanje *i=0* i *i=chunk-1* elementa neophodno dobiti vrednost niza *x* za *i=chunk-1* onog procesa čiji je rang za 1 manji od trenutnog ranga (ukoliko postoji) i vrednost niza *x* za *i=0* onog procesa čiji je rang za 1 veći od trenutnog ranga (ukoliko postoji), respektivno.

Kao rešenje ovog problema se u funkcijama koje rade *jacobi update*, računanje *d_local* i *r_local* vrednosti najpre vrši slanje odgovarajućih graničnih elemenata svojim procesima susedima pomoću funkcije *Isend* (nulti element se šalje procesu sa rangom niži za 1, a *chunk-1* element se šalje procesu sa rangom viši za 1). Pored ovog slanja, takođe se i šalju zahtevi za primanje odgovarajućeg podatka od suseda pomoću *Irecv* funkcije. Obe ove funkcije su asinhrone zato što svaki proces može da nastavi da računa rezultate za sve elemente kod kojih ne postoji zavisnost po podacima, a to su elementi za vrednost brojača *i=1..chunk-2.* Ovim se dobija ubrzanje paralelizacije jer ne postoji nepotrebno blokirajuće čekanje na neophodne vrednosti koje poseduju susedi, već se radi korisna obrada sve dok ti podaci nisu neophodni. Kada se izvrši obrada svih elemenata osim graničnih, tada se mora čekati da se završi prijem od suseda pozivom funkcije *MPI_Wait*, a onda se i granični elementi obrađuju pošto je neophodna informacija od suseda pristigla.

Time se postiže najveći mogući nivo paralelizacije koristeći rutine za neblokirajuću komunikaciju.

2.3. Rezultati

U okviru ove sekcije su izloženi rezultati paralelizacije problema 2 koristeći rutine za neblokirajuću komunikaciju za razmenu poruka.

2.3.1. Logovi izvršavanja

Ovde su dati logovi izvršavanja za definisane test primere i različit broj niti.

Listing 1. Zadatak 4 – 1 nit

```
**********DZ2Z3**********
Elapsed time - SEQ: 0.0875039.
Elapsed time - PAR(1): 0.0618948.
TEST PASSED
**********
**********DZ2Z3**********
Elapsed time - SEQ: 0.262239.
```

Elapsed time - PAR(1): 0.227064.

TEST PASSED

****

**********DZ2Z3**********

Elapsed time - SEQ: 6.78218.

Elapsed time - PAR(1): 6.40606.

TEST PASSED

****

**********DZ2Z3**********

Elapsed time - SEQ: 26.7743.

Elapsed time - PAR(1): 26.1202.

TEST PASSED

****

Listing 2. Zadatak 4 – 2 niti

**********DZ2Z3**********

Elapsed time - SEQ: 0.0855327.

Elapsed time - PAR(2): 0.0367261.

TEST PASSED

****

**********DZ2Z3**********

Elapsed time - SEQ: 0.255073.

Elapsed time - PAR(2): 0.134064.

TEST PASSED

****

**********DZ2Z3**********

Elapsed time - SEQ: 6.64812.

Elapsed time - PAR(2): 3.42375.

TEST PASSED

****

**********DZ2Z3**********

Elapsed time - SEQ: 28.503.

Elapsed time - PAR(2): 15.6521.

TEST PASSED

****

Listing 3. Zadatak 4 – 4 niti

**********DZ2Z3**********

Elapsed time - SEQ: 0.112622.

Elapsed time - PAR(4): 0.0248323.

TEST PASSED

****

**********DZ2Z3**********

Elapsed time - SEQ: 0.28755.

Elapsed time - PAR(4): 0.0928026.

TEST PASSED

****

**********DZ2Z3**********

Elapsed time - SEQ: 7.18861.

Elapsed time - PAR(4): 2.03025.

TEST PASSED

****

**********DZ2Z3**********

Elapsed time - SEQ: 29.3591.

Elapsed time - PAR(4): 8.69126.

TEST PASSED

****

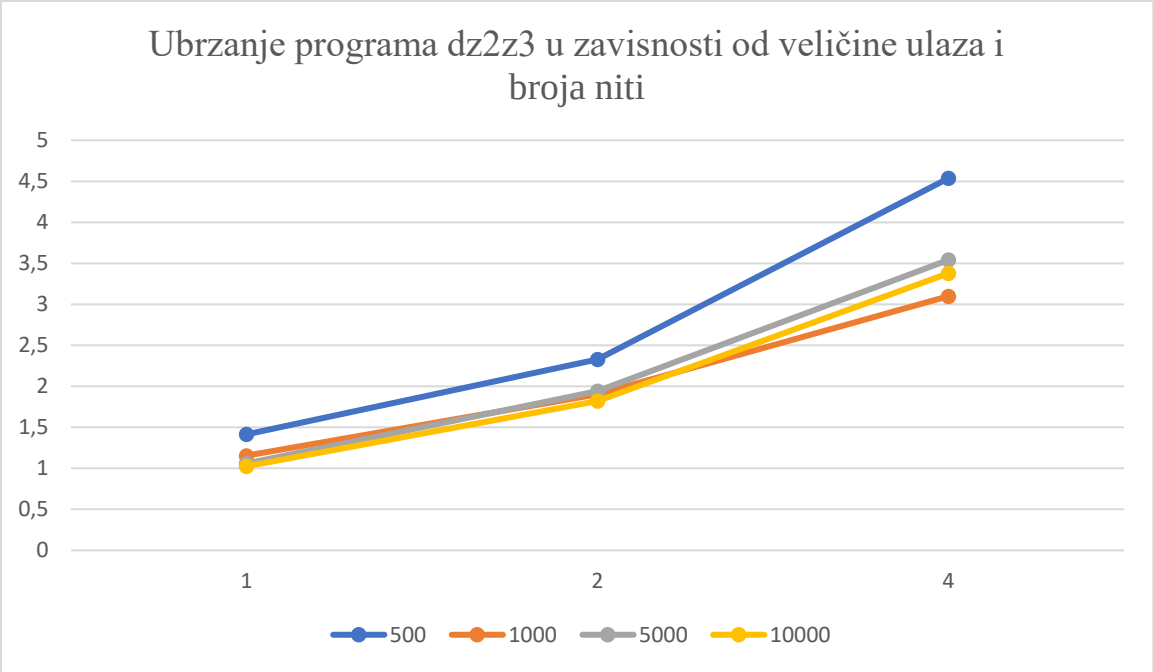
2.3.2. Tabela trajanja programa

	input 500			input 1000		
	Sekvencijalno vreme	Paralelno vreme	Ubrzanje	Sekvencijalno vreme	Parelelno vreme	Ubrzanje
Zadatak 3 – 1 nit	0,0875039	0,0618948	1,41375	0,262239	0,227064	1,1549123
Zadatak 3 – 2 niti	0,085527	0,0367261	2,32878	0,255073	0,134064	1,9026211
Zadatak 3 – 4 niti	0,112622	0,024823	4,537	0,28755	0,0928026	3,0985123

	input 500			input 1000		
	Sekvencijalno vreme	Paralelno vreme	Ubrzanje	Sekvencijalno vreme	Parelelno vreme	Ubrzanje
Zadatak 3 – 1 nit	6,78218	6,40606	1,058713	26,7743	26,1202	1,025042
Zadatak 3 – 2 niti	6,64812	3,42375	1,941766	28,503	15,6521	1,821034
Zadatak 3 – 4 niti	7,18861	2,03025	3,540751	29,3591	8,69126	3,378003

2.3.3. Grafici ubrzanja

U okviru ove sekcije su dati grafici ubrzanja paralelnih implementacija u odnosu na sekvencijalnu implementaciju. Ubrzanje na svim narednim graficima predstavljeno je kao odnos trajanja sekvencijalne implementacije programa i trajanja paralelizovane implementacije.



2.3.4. Diskusija dobijenih rezultata

Rešenje ovog zadatka zahteva slanje velikog broja poruka i relativno čestu sinhronizaciju. Očekivano se performanse i ubrzanje povećavaju sa veličinom ulaznih podataka i brojem procesa zbog toga što je odnos *overhead*-a vezanog za komunikaciju i količine izračunavanja povoljniji.

3. PROBLEM 3 - KMEANS

U okviru ovog poglavlja je dat kratak izveštaj u vezi rešenja zadatog problema 3.

3.1. Tekst problema

Paralelizovati program koji vrši k-means klasterizaciju podataka. Klasterizacija metodom k-srednjih vrednosti (eng. k-means clustering) je metod koji particioniše n objekata u k klastera u kojem svaki objekat pripada klasteru sa najbližom srednjom vrednošću. Objekat se sastoji od niza vrednosti - osobina (eng. features). Podelom objekata u potklastere, algoritam predstavlja sve objekte pomoću njihovi srednjih vrednosti (tzv. centroida potklastera). Inicijalni centroid za svaki potklaster se bira ili nasumično ili pomoću odgovarajuće heuristike. U svakoj iteratiji, algoritam pridružuje svaki objekat najbližem centroidu na osnovu definisane metrike. Novi centroidi za sledeću iteraciju se izračunavaju usrednjavanjem svih objekata unutar potklastera. Algoritam se izvršava sve dok se makar jedan objekat pomera iz jednog u drugi potklaster. Program se nalazi u direktorijumu kmeans u arhivi koja je priložena uz ovaj dokument. Program se sastoji od više datoteka, od kojih su od interesa datoteke kmeans.c, cluster.c i kmeans_clustering.c. Analizirati dati kod i obratiti pažnju na različite mogućnosti i nivoe na kojima se može obaviti paralelizacija koda, kao i na deo koda za generisanje novih centroida u svakoj iteraciji unutar datoteke kmeans_clustering.c. Ulazni test primeri se nalaze u direktorijumu data, a način pokretanja programa u datoteci run. [1, N]

3.2. Delovi koje treba paralelizovati

3.2.1. Diskusija

Kod koji sekvencijalno rešava ovaj problem razdvojen je u tri fajla – *kmeans.c, cluster.c,* i *kmeans_clustering.*c. Sama *main* funkcija koja se nalazi u *kmeans.c* fajlu samo vrši učitavanje svih neophodnih argumenata iz komandne linije i iz odgovarajućih fajlova i alokaciju prostora, a zatim kroz određeni broj iteracija (trenutno postavljeno na 1) poziva funkciju *cluster* iz *cluster.c* fajla. Pored toga, kod iz *main* funkcije samo vrši ispis i dealokaciju prostora. U paralelnoj verziji, većinu ovog posla radi proces sa rangom master. On će pokrenuti sekvencijalnu implementaciju programa i zapamtiti vreme izvršavanja, a zatim će odgovarajuće vrednosti pomoću funkcije *broadcast* proslediti i ostalim procesima koji rade

obradu. Na kraju, proces master vrši ispis dobijenih rezultata i upoređuje validnost dobijenih vrednosti sa onim dobijenih iz sekvencijalne implementacije, a i ispisuje vremena izvršavanja obe implementacije.

Funkcija *cluster* iz *cluster.c* fajla je samo *wrapper* funkcija koja alocira niz *membership* koji se koristi u *kmeans_clustering* funkciji, generiše *seed* random generatora, i generiše rezultat tako što poziva funkciju *kmeans_clustering*. Povratna vrednost funkcije *kmeans_clustering* se zatim vraća u *main* funkciju preko promene argumenta *cluster_centres* koji je prosleđen po referenci. U slučaju paralelizacije koristeći MPI svi procesi će pozvati funkciju *cluster_par*, samo će u njoj alocirati *membership* koji odgovara samo objektima koji su dodeljeni tom procesu, a zatim se unutar *kmeans_clustering_par* funkcije vrši komunikacija između mastera i ostalih procesa.

U ovom zadataku najveće ubrzanje dobija se paralelizacijom funkcije koja se u sekvencijalnoj implementaciji nalazi u fajlu *kmeans_clustering.c*. Ovo uključuje paralelizaciju funkcije *float** kmeans_clustering(args).* Funkcije *int find_nearest_point(args), float euclid_dist_2(args)* su pomoćne funkcije koje ne treba paralelizovati. Način na koji je izvršena paralelizacija za navedenu funkciju opisan je u narednom poglavlju.

3.2.2. Način paralelizacije

Ideja paralelizacije ovog problema jeste da se svakom procesu dodeli određeni deo niza *clusters* na izračunavanje, i da se zatim u master procesu izvrši sakupljanje rezultata obrade, pri čemu bi i master radio obradu zajedno sa ostalim procesima. Pošto se određivanje niza *clusters* radi pomoću dve velike *for* petlje koje se ponavljaju sve dok važi *delta > threshold* cilj je njih paralelizovati na najoptimalniji mogući način. Implementacija koja ovo postiže detaljno je opisana u nastavku ovog poglavlja:

Slično kao i u prethodnom zadatku, sama *main* funkcija je osmišljena tako da master prikuplja sve argumente sa komandne linije, vrši čitanje ulaznih podataka iz dostavljenih fajlova, pokreće sekvencijalnu verziju i meri vreme izvršavanja obe implementacije. Nakon ovoga, on vrši raspodelu neophodnih promenljivih, i na kraju paralelne obrade sakuplja rezultat i vrši ispis i upoređivanje vremena između paralelne i sekvencijalne verzije i proverava tačnost dobijenog rezultata.

Na početku *main* funkcije svaki proces za sebe nalazi *rank* i *size* globalnog komunikatora, a od procesa mastera dobija vrednosti za *numObjects, numAttributes, nclusters i threshold* pozivom funkcije *Broadcast.* Pored ovoga, svi procesi koji nisu master imaju bafer koji sadrži kopiju matrice atributa. Nakon ove raspodele, svi procesi pozivaju funkciju *cluster_par*, pri čemu se u sekvencijalnoj verziji rezultat vraćao putem parametra *cluster_centres_par.* U ovoj implementaciji ovaj parametar se posmatra jedino u master procesu, pa će samo master i u funkciji *cluster_par* vraćati rezultat, dok se za ostale procese ta povratna vrednost ne posmatra i nigde ne koristi. Samo sakupljanje rezultata se radi unutar *kmeans_clustering_par* funkcije, tako da se ništa ne radi na kraju poziva *kmeans_clustering_par* i *cluster_par* funkcija osim vraćanja dobijenih vrednosti.

Funkciju *cluster_par* pozivaju svi procesi. Svaki proces za sebe izračuna dve vrednosti – *chunkClusters* i *chunkPoints*. Pošto se obrada sastoji iz dve velike petlje od kojih jedna ide do *npoints*, a druga do *nclusters*, idealno je podeliti ovu petlju tako da svaki proces radi deo ove obrade, a to se upravo omogućava podelom određenih struktura na *chunkClusters* ili *chunkPoints* delova i zatim te delove dodeliti određenim procesima. Samim tim, unutar ove funkcije svaki proces generiše svoj niz *membership* (koji je sada veličine *chunkPoints*) koji pomaže u izradi rezultata u *kmeans_clustering_par* funkciji. Jedino proces master u ovoj funkciji radi vraćanje rezultata u glavni program.

Funkcija *kmeans_clustering_par* sadrži najveći deo paralelizovanog koda. Kao što je diskutovano u prethodnom izveštaju za OpenMP, problem u paralelizaciji ovog koda jeste što se mora izvršiti redukcija niza *new_centers_len* i matrice *new_centers* u sredini same obrade. U toku prve petlje, upis u *new_centers_len* niz i *new_centers* matricu zavisi od vrednosti *index* koja se dobija pozivom funkcije *find_nearest_point*. Ova funkcija uvodi dva ograničenja prilikom paralelizacije:

1. Pošto funkcija kao parametar prihvata ceo niz *clusters* kao argument, zahteva se prisustvo ažurne kopije celog ovog niza pri svakoj iteraciji velike *do-while* petlje. Kako bi se ovo rešilo, na kraju *do-while* petlje vrši se prikupljanje rezultata trenutne iteracije u masteru pomoću *MPI_Gather* poziva, dok na početku petlje master ovu ažurnu kopiju *clusters* niza daje i ostalim procesima pomoću *MPI_Bcast* poziva.

2. Pošto povratna vrednost ove funkcije (tj. *index*) može predstavljati bilo koji broj iz opsega *0..numObjects*, a jedan proces radi samo sa *chunkPoints* delom ulaznih podataka, ovo zahteva da svaki proces mora pamtiti svoje bafere *local_new_centers_len* i *local_new_centers* koji su veličine *nclusters* i *nclusters * nfeatures*, a ne *chunkClusters* i *chunkCsluters * nfeatures.* Tada se pri redukciji ovih lokalnih bafera dobijaju korektne vrednosti za sve objekte.

Takođe, unutar ove prve petlje svaki proces za sebe računa lokalnu vrednost parametra *delta* (u kodu nazvana *localDelta*) koju je neophodno redukovati pre same provere uslova *do-while* petlje. Pošto svi procesi rade nezavisno, a svi treba da završe u istom trenutku, poziva se funkcija *MPI_Allreduce* za parametar *delta*. Nakon ovog poziva svi procesi u promenljivoj *delta* imaju ažurnu vrednost ovog podatka.

Između prve i druge *for* petlje obrade radi se redukcija dobijenih rezultata u svim procesima. Najpre se radi redukcija local_new_centers_len niza u niz new_centers_len, a zatim redukcija matrice local_new_centers u new_centers. Pošto i dalje svaki proces treba da radi nad svojim delom podataka, odmah se nakon ove redukcije vrši raspodela odgovarajućeg dela podataka odgovarajućem procesu. Iz ovog razloga, koristi se poziv funkcije *MPI_Reduce_scatter* za oba bafera u svim procesima. Nakon ovoga, izvršena je redukcija međurezultata, i oni su raspodeljeni nazad u odgovarajuće procese.

Nakon ovoga vrši se paralelizacija druge velike *for* petlje obrade, pri čemu sada svaki proces kreira svoj rezultat u nizu *local_clusters*. Pošto svaki proces kreira deo rezultata, ova petlja je podeljena i svaki proces vrši *chunkClusters* iteracija ove petlje. Nakon izvršene obrade, vrši se prikupljanje svih rezultata koji se nalaze u *local_clusters* matrici u *clusters* matrici koja se nalazi u master procesu. Ukoliko je ovo poslednja iteracija, master u *clusters* matrici ima celokupan rezultat i on se vraća kao rezultat obrade pomoću *return clusters;* naredbe i ova matrica propagira do glavnog programa kao rešenje problema. Ukoliko ovo nije poslednja iteracija, ponovo se radi *broadcast* sada ovog međurezultata koji se nalazi u matrici *clusters* i obrada se ponovo izvršava sve dok je broj premeštanja objekata iz jednog klastera u drugi veći od zadate vrednosti parametra *threshold.*

3.3. Rezultati

U okviru ove sekcije su izloženi rezultati paralelizacije problema 3.

3.3.1. Logovi izvršavanja

Ovde su dati logovi izvršavanja za definisane test primere i različit broj niti. Obavezno uključiti u ispis i vremena izvršavanja. Logove pojedinačno uokviriti i obeležiti.

Listing 1. Zadatak 4 – 1 nit

```
I/O completed
numObjects=100, numclusters=8
==================================SEQ=====================================
number of Clusters 8
number of Attributes 34
Time for process: 0.001155
==================================PAR=====================================
number of threads: 1
number of Clusters 8
number of Attributes 34
Time for process: 0.001093
TEST PASSED

I/O completed
numObjects=494016, numclusters=8
==================================SEQ=====================================
number of Clusters 8
number of Attributes 34
Time for process: 24.416080
==================================PAR=====================================
number of threads: 1
number of Clusters 8
number of Attributes 34
Time for process: 25.038107
TEST PASSED

I/O completed
numObjects=204796, numclusters=8
==================================SEQ=====================================
number of Clusters 8
number of Attributes 34
```

Time for process: 5.807440

==========PAR==========

number of threads: 1

number of Clusters 8

number of Attributes 34

Time for process: 5.907594

TEST PASSED

I/O completed

numObjects=819196, numclusters=8

==========SEQ==========

number of Clusters 8

number of Attributes 34

Time for process: 41.560881

==========PAR==========

number of threads: 1

number of Clusters 8

number of Attributes 34

Time for process: 41.783075

TEST PASSED

Listing 2. Zadatak 4 – 2 niti

I/O completed

numObjects=100, numclusters=8

==========SEQ==========

number of Clusters 8

number of Attributes 34

Time for process: 0.000939

==========PAR==========

number of threads: 2

number of Clusters 8

number of Attributes 34

Time for process: 0.000734

TEST PASSED

I/O completed

numObjects=494016, numclusters=8

==========SEQ==========

number of Clusters 8

number of Attributes 34

Time for process: 25.310123

==========PAR==========

number of threads: 2

number of Clusters 8

number of Attributes 34

Time for process: 13.162480

TEST FAILED

I/O completed

numObjects=204796, numclusters=8

==========SEQ==========

number of Clusters 8

number of Attributes 34

Time for process: 5.906832

==========PAR==========

number of threads: 2

number of Clusters 8

number of Attributes 34

Time for process: 3.022614

TEST PASSED

I/O completed

numObjects=819196, numclusters=8

==========SEQ==========

number of Clusters 8

number of Attributes 34

Time for process: 44.133237

==========PAR==========

number of threads: 2

number of Clusters 8

number of Attributes 34

Time for process: 21.836951

TEST FAILED

Listing 3. Zadatak 4 – 4 niti

I/O completed

numObjects=100, numclusters=8

==========SEQ==========

number of Clusters 8

number of Attributes 34

Time for process: 0.004375

==========PAR==========

number of threads: 4

number of Clusters 8

number of Attributes 34

Time for process: 0.002913

TEST PASSED

I/O completed

numObjects=494016, numclusters=8

==========SEQ==========

number of Clusters 8

number of Attributes 34

Time for process: 26.797842

==========PAR==========

number of threads: 4

number of Clusters 8

number of Attributes 34

Time for process: 6.978688

TEST FAILED

I/O completed

numObjects=204796, numclusters=8

==========SEQ==========

number of Clusters 8

number of Attributes 34

Time for process: 6.140115

==========PAR==========

number of threads: 4

number of Clusters 8

number of Attributes 34

Time for process: 1.601039

TEST PASSED

I/O completed

numObjects=819196, numclusters=8

==========SEQ==========

number of Clusters 8

number of Attributes 34

Time for process: 46.224978

==========PAR==========

number of threads: 4

number of Clusters 8

number of Attributes 34

Time for process: 11.541173

TEST FAILED

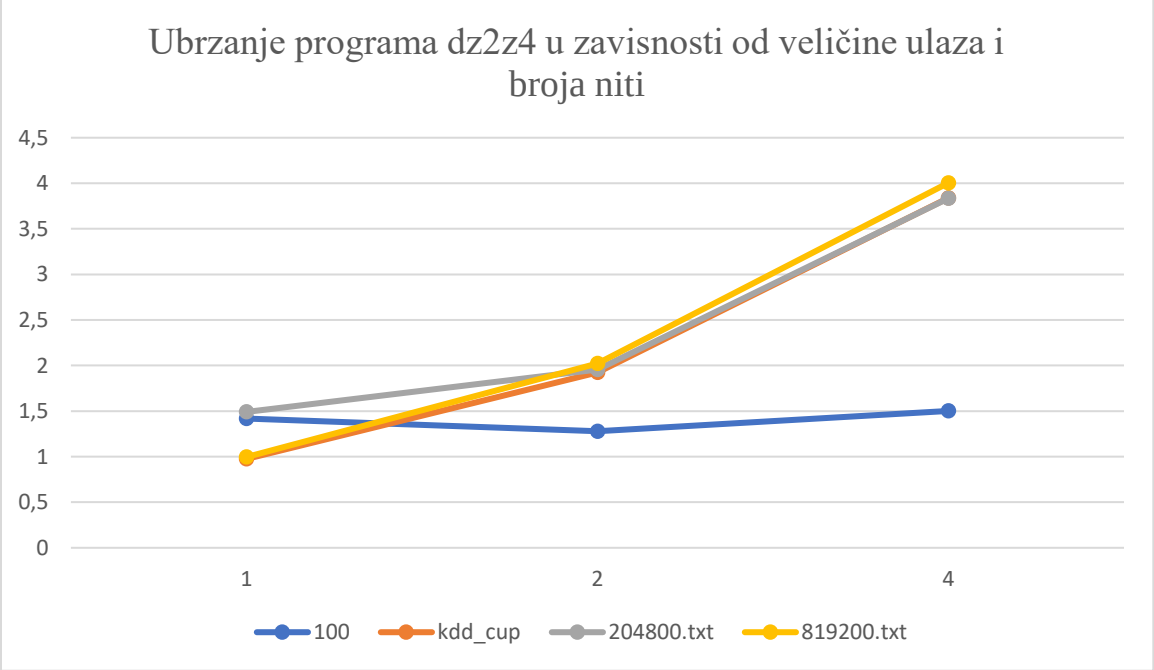
3.3.2. Tabela trajanja programa

	100			kdd_cup		
	Sekvencijalno vreme	Paralelno vreme	Ubrzanje	Sekvencijalno vreme	Parelelno vreme	Ubrzanje
Zadatak 4 – 1 nit	0,00155	0,001093	1,41812	24,41608	25,038107	0,9751568
Zadatak 4 – 2 niti	0,000939	0,000734	1,27929	25,310123	13,16248	1,9228993
Zadatak 4 – 4 niti	0,004375	0,002913	1,50189	26,797842	6,978688	3,8399542

	204800.txt			819200.txt		
	Sekvencijalno vreme	Paralelno vreme	Ubrzanje	Sekvencijalno vreme	Parelelno vreme	Ubrzanje
Zadatak 4 – 1 nit	8,80744	5,907594	1,490868	41,56088	41,73075	0,995929
Zadatak 4 – 2 niti	5,906832	3,022614	1,954213	44,13324	21,83695	2,021035
Zadatak 4 – 4 niti	6,140115	1,60103	3,835103	46,22498	11,54117	4,005224

3.3.3. Grafici ubrzanja

U okviru ove sekcije su dati grafici ubrzanja paralelnih implementacija u odnosu na sekvencijalnu implementaciju. Ubrzanje na svim narednim graficima predstavljeno je kao odnos trajanja sekvencijalne implementacije programa i trajanja paralelizovane implementacije.



3.3.4. Diskusija dobijenih rezultata

U ovom zadatku dobijena je velika paralelizacija za date ulazne primere, pri čemu je ubrzanje direktno zavisilo od broja niti. Povećanje broja niti dovelo do značajnog ubrzanja izvršavanja programa, pogotovo za velike ulazne fajlove. Dobijeni rezultati ukazuju na to da je uspešno izvršena paralelizacija i da je pokretanje mehanizma kreiranja niti i njihovog puštanja u rad opravdano jer su dobici u brzini izvršavanja dosta veliki. Testovi ispisuju TEST_FAILED zbog male vrednosti konstante *ACCURACY* koja je zadata u zadatku.

Na datom grafiku u sekciji *3.3.3.* performanse kod najmanjeg ulaznog fajla se ne povećavaju značajno sa povećanjem broja procesa, što ima smisla zbog velikog *overhead*-a prilikom paralelizacije. Za ulazne primere koji imaju značajno više podataka značaj režijskih troškova nema preveliki uticaj.