Paralelni sistemi

Rešeni blanketi sa usmenog dela ispita

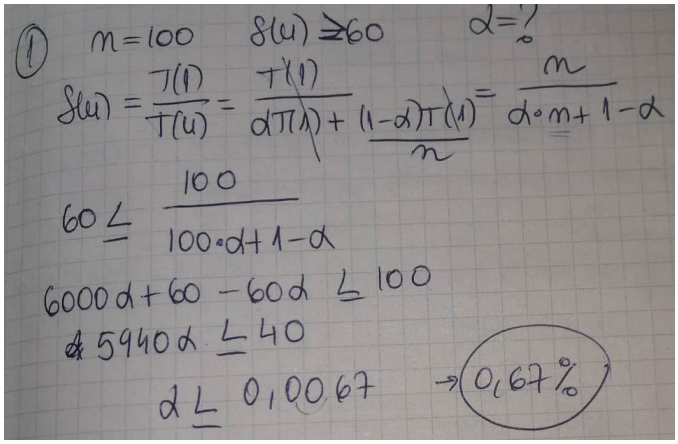
Akreditacija 2013

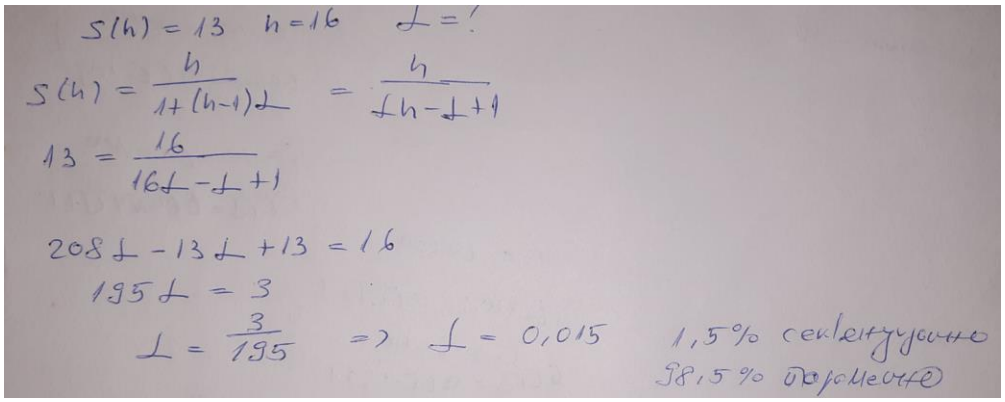
1. (2) Program P se izvršava za 100 sec na jedno-procesorskom sistemu. Kada se program paralelizuje, 20 sec traje izvršenje dela programa koji se ne može paralelizovati, a ostali deo programa se može idealno paralelizovati.
   a. Ako se program izvršava na 16 procesorskom sistemu, koliko vremena će trajati izvršenje programa?
      T(n) = Ts + Tp / n, T(16) = 20s + 80s / 16 = **25s**
   b. Ako imamo beskonačno mnogo procesora na raspolaganju, koliko vremena će trajati izvršenje programa?
      α = Ts / T(1) = 0.2
      lim n->∞S(n) = 1 / α = 1 / 0.2 = 5
      T(∞) = T(1) / S(∞) = 100 / 5 = **20s**

2. (2) Sekvencijalna verzija programa se izvršava za 10000 sati. Izvršenje programa na višeprocesorskom sistemu sa 1000 procesora traje 20 sati.
   a. Koliko je ubrzanje a kolika efikasnost višeprocesorkog sistema?
      ubrzanje: **S(n) = T(1) / T(n)** S(1000) = 10000h / 20h = 500
      efikasnost: **E(n) = S(n) / n** E(1000) = 500 / 1000 = 0.5 * 100% = 50%
   b. Koliko bi trajalo izvršenje programa ako bi efikasnost 1000 procesorskog sistema bila 100%
      E(1000) = 100% = 1
      1 = S(1000) / 1000 => S(n) = 1000
      1000 = 10000h / T(n) => T(n) = 10000h / 1000 = **10h**

3. Pretpostavimo da program koji se izvršava na višeprocesorkom sistemu ima na početku deo koji se izvršava sekvencijalno i zahteva 3ms vremena, i deo na kraju koji zahteva sekvencijalno izvršenje i traje 4ms. Između ova dva dela, program može biti podeljen na 5 jednakih delova koji se mogu izvršavati paralelno i svaki deo zahteva 16ms vremena. Koliko iznosi maksimalno ubrzanje po Amdahl-ovom zakonu?
   n = 5
   T(1) = 3ms + 4ms + 5 * 16ms = 87ms
   T(n) = 3ms + 16ms + 4ms = 23ms
   S(n) = 87ms / 23ms = **3.78**

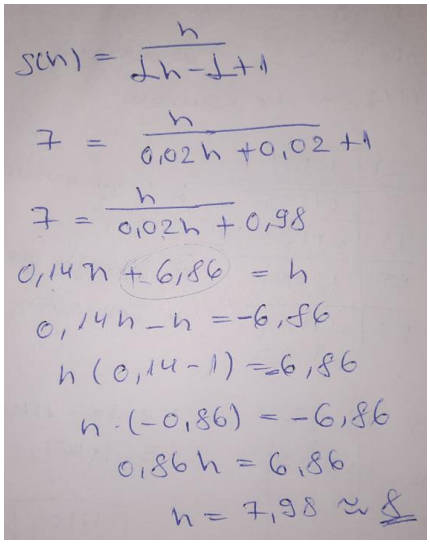
4. Pretpostavimo da imamo multiprocesor sa 100 procesora i želimo da postignemo ubrzanje od najmanje 60. Koliko programa se može izvršavati sekvencijalno?



5. (2) Definisati ubrzanje i efikasnost. Eksperimentalno je utvrđeno da paralelna verzija izvesne aplikacije postiše ubrzanje od 13 kada se izvršava na 16-procesorskom sistemu. Pod pretpostavkom da se program sastoji iz delova koji su potpuno sekvencijalni i potpuno paralelni (sa zanemarljivim vremenom komunikacije i sinhronizacije) odrediti koliki je deo (frakcija) programa koji je paralelizovan. Koliko se maksimalno ubrzanje ovog programa može postići ako bi imali neograničeni broj procesa na raspolaganju?
   Ubrzanje je odnos vremena izvršenja programa bez poboljšanja i vremena sa poboljšanjem.
   Efikasnost je odnos ubrzanja i broja procesora, sa porastom broja procesora opada efikasnost.
   

   za n->∞ lim S(n) $_{n->\infty}$ = 1 / α = 1 / 0.015 = 66.67.

6. Sistem se sastoji od 10 procesora. 98% koda se može paralelizovati. Da li je moguće na ovom sistemu postići ubrzanje 7? Ako da, koliko je minimalno procesora potrebno da bi se to postigo?



7. (2) Koje od sledećih zavisnosti mogu dovesti do problema kod protočnih procesora kod kojih je moguće izvršenje van redosleda, ali ne mogu nastupiti kod protočnih procesora kod koga je izvršenje po redosledu.
   a. write-after-write
   b. read-after-write
   c. write-after-read
   d. read-after-read

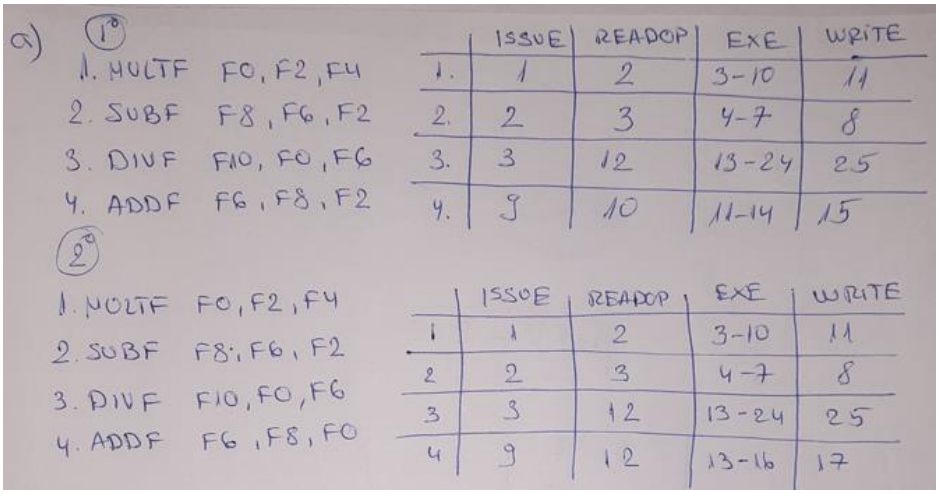
8. (2) Procesor ima sledeće funkcionalne jedinice: jednu za sabiranje/oduzimanje u pokretnom zarezu sa latencijom od 4 clk ciklusa, jednu za množenje u FP sa latentnošću od 8 clk ciklusa i jednu za deljenje u FP sa latentnošću od 12 clk ciklusa. Za koliko klok ciklusa će se izvršiti sledeći niz instrukcija ako se koristi a) Scoreboard b) Tomasulov algoritam za planiranje izvršenja instrukcija. Odgovore dati popunjavanjem odgovarajućih tabela statusa instrukcija.

   MULTF F0, F2, F4
   SUBF F8, F6, F2
   DIVF F10, F0, F6
   ADDF F6, F8, F0

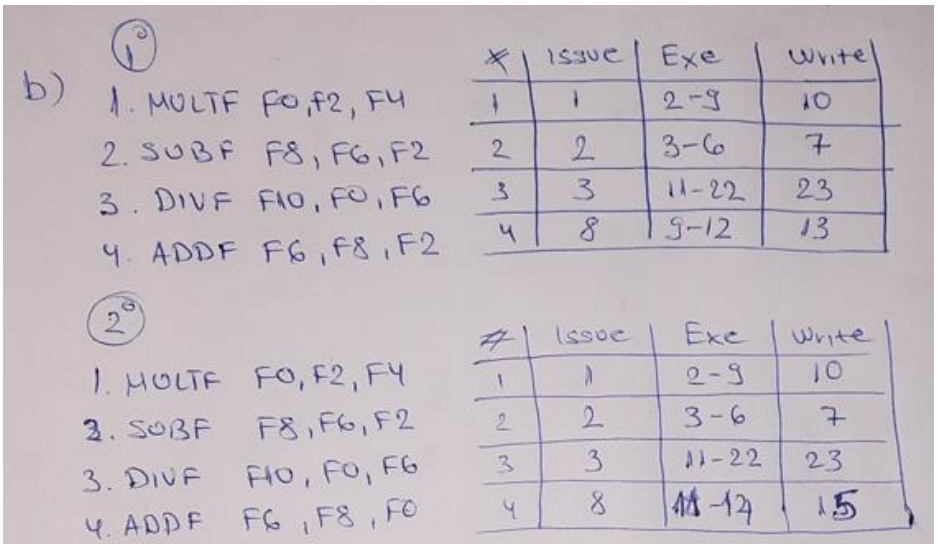
   (Isti zadatak sa drugog blanketa razlika je samo: ADDF F6, F8, F2)

   a.

a)



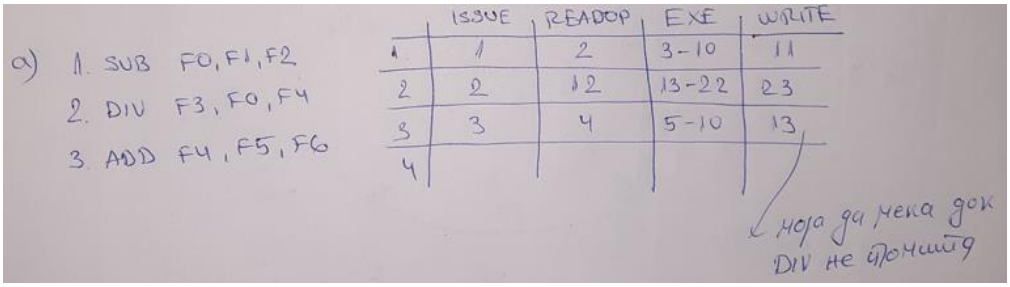
b.



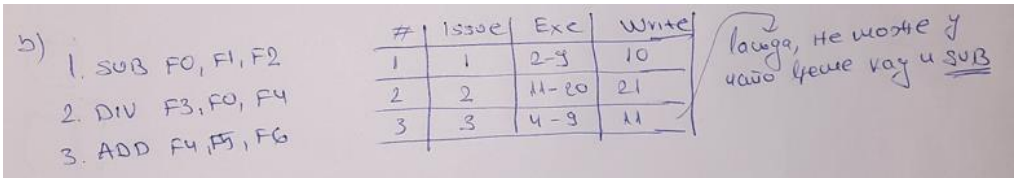
9. Za koliko ciklusa će se izvršiti sledeći niz instrukcija ako se koristi a) Scoreboard b)Tomasulov algoritam za planiranje izvršenja instrukcija. Odgovore dati popunjavanjem odgovarajućih tabela statusa instrukcija.

SUB F0, F1, F2 (8 cycles)
DIV F3, F0, F4 (10 cycles)
ADD F4, F5, F6 (6 cycles)

a.



b.



10. Za upravljanje izvršenjem instrukcija Scoreboard koristi nekoliko tabela. Koje su to tabele? Ukratko objasniti značenje pojedinih polja u svakoj tabeli.
    - **Status instrukcije (Instruction status):**
        - Govori u kojoj od 4 faze izvršenja se instrukcija nalazi.
    - **Status funkcionalne jedinice (Functional unit status):**
        - Ukazuje na stanje FU. Za svaku FU postoji 9 polja u tabeli statusa:
        - *Busy* Govori da li je FU slobodna ili ne
        - *Op* Operacija koju obavlja FU (npr., + ili –)
        - *Fi* Odredišni registar
        - *Fj, Fk* Brojevi izvornih registara
        - *Qj, Qk* FU koja proizvodi podatke za izvorne registre Fj, Fk
        - *Rj, Rk* Flegovi koji ukazuju kada su Fj, Fk spremni (postavljaju se na yes kada su operandi dostupni, brišu se – postavljaju na no kada se pročitaju operandi)
    - **Tabela statusa registra rezultata (Register result status):**
        - Ukazuje koja će FU izvršiti upis u koji regstar, ako aktivna instrukcija ima registar kao odredište.
        - Ako ne postoji FU koja vrši upis u dati registar, odgovrajuće polje je prazno.

11. U kojoj od faza izvršenje kod Scoreboard i Tomasulovog algoritma se razrešavaju
    a. WAR hazardi
       SC: Write result, T: Issue (preimenovanje)
    b. WAW hazardi
       SC: Issue, T: Issue (preimenovanje)
    c. RAW hazardi
       SC: Read operands, T: Execution

12. U kojoj fazi izvršenja se kod Tomasulovog algoritma obavlja preimenovanje registara?
    Obavlja se u **Issue** fazi.

13. U kojoj fazi protočnog izvršenja instrukcije se razrešava grananje?
    Ishod uslova grananja i adresa odredišta grananja se određuje u trećoj (EX) fazi, programski brojač PC se puni u četvrtoj fazi adresom odredišta grananja, dok će se u petoj fazi pribaviti instrukcija koja sledi instrukciju grananja.

14. U kojoj fazi protočnog izvršenja instrukcija se detektuju svi hazardi?

Svi hazardi se detektuju u **ID fazi** (u toku dekodiranja instrukcije).

15.

a. Koje faze u izršenju instrukcija postoje kod
   - Scoreboard tehnike
     Issue, Read operands, Execution, Write result
   - Tomasulovog algoritima
     Issue, Execution, Write result

b. Koje aktivnosti se obavljaju u ISSUE fazi kod
   - Scoreboard tehnike
     Ako je funkcionalna jedinica slobodna i ni jedna druga aktivna instrukcija nema isti odredišni registar (ne postoji WAW hazard), Scoreboard propušta instrukciju u funkcionalnu jedinicu i ažurira internu strukturu podataka.
     Ovde se detektuju **strukurni** i **WAW** hazardi. Ako strukturni ili WAW hazard postoji, onda se izdavanje svih instrukcija **zaustavlja** dok se hazard ne obriše.
   - Tomasulovog algoritma
     instrukcija se pribavlja iz reda čekanja.
     - Ako je u pitanju FP operacija, instrukcija se izdaje ako postoji slobodna rezervaciona stanica i šalju se operandi, ako su u registrima.
     - Ako je u pitanju load ili store instrukcija, ona se izdaje ako postoji slobodan load ili store bafer.
     - Ako nema slobodne rezervacione stanice ili load/store bafera onda postoji strukturni hazard i instrukcija se zaustavlja.
     - U ovom koraku se vrši i proces preimenovanja registara

16. (4) Pored svake od sledećih tvrdnji staviti tačno (T) ili netačno (N)

a. Scoreboard vrši preimenovanje registara da bi izbegao WAW i WAR hazarde. **N**
b. Scoreboard ne može forwardovati rezultat, Tomasulov može. **T**
c. Tomasulov izbegava WAR i WAW hazarde. **T**
d. Kod Scoreboard instrukcija se zaustavlja u Read Operands stepenu dok svi izvorni operandi ne postanu dostupni. **T**
e. Kod Tomasulovog algoritma instrukcija čita sve operande iz registarskog fajla. **N**
f. Jedina informacija koja se čuva u rezervacionoj stanici je kod operacije (operation), raspoloživost (busy) funkcionalne jedinicie, i oznake FU koje će generisati izvorne operande. **N**
g. Scoreboard vrši statičko planiranje izvršenja instrukcija a Tomasulov algoritam dinamičko. **N**
h. Scoreboard zaustavlja izvršenje da bi razrešio WAR i WAW hazarde, a Tomasulov algoritam ne. **T**
i. Scoreboard koristi rezervacione stanice da upravlja statusom funkcionalnih jedinica i registrar, a Tomasulov algoritam ne. **N**
j. Scoreboard koristi distribuirano upravljanje a Tomasulov algoritam centralizovano. **N**

17. Ukratko objasniti razlike izmedju VLIW i superskalarne organizacije procesora.
    - **Superskalarni** mogu da izdaju različit broj instrukcija po clk clickusu, dok VLIW izdaje fiksni broj instrukcija u vidu jedne velike instrukcije ili fiksnog instrukcionog paketa.
    - **VLIW** procesori su po definiciji sa statičkim planiranjem izvršenja instrukcija, dok superskalarni mogu da koriste i statičko i dinamičko planiranje.

18. Zaokružiti šta od navedenog važi za Superskalarne (SP), a šta za VLIW procesore.
    Planiranje izvršenja instrukcija se obavlja
    a. statički uz pomoć kompajera (**SP, VLIW**)
    b. dinamički u fazi izvršenja programa (**SP**, VLIW)
    c. moguće su obe varijante (**SP**, VLIW)

19. (4) Objasniti GCD test za detekciju loop-carry zavisnosti
    NZD (Najveći Zajednički Delilac), tj. **GCD**(Greatest Common Divisor) je jednostavan i dovoljan test koji se koristi za detekciju zavisnosti.
    Ako loop-carry zavisnost postoji tada **NZD(c, a)** mora deliti **(d – b)** bez ostatka, baš (d – b) jer je j < k.
    Pri čemu su:
    - j i k iterativni indeksi petlji
    - a i c množe j i k pri indeksiranju
    - b i d su konstante koje se dodaju na iterativne indekse.

    Primer: a * j + b = c * k + d

20. (2) Pomoću GCD testa utvrditi da li u sledećem kodu postoji loop-carry zavisnost.
    for p = 1, P
    for q = 1, Q
    for r = 1, R
    for s = 1, S
    A(2p + 2q - r + 3s) = A(p - 6q + 5r +3) * B(r + s - 4q)
    GCD(2, 1, 3, 6, 5) = 1
    1 deli 3

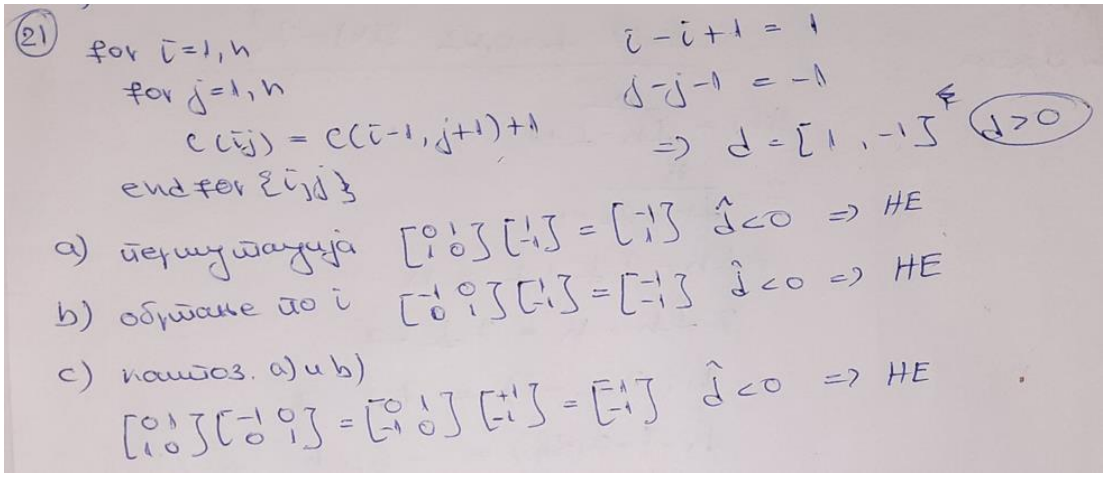
21. Pomoću GCD testa utvrditi da li u sledećem kodu postoji loop-carry zavisnost.
    for i := 1, n1
    for j := 1, n2
    for k := 1, n3
    a(2 * i + 5 * j + 4 * k) = a(6 * i + 2 * j - 7 * k +4) + …
    GCD(2, 5, 4, 6, 7) = 1
    1 deli 4

22. (3) Pomoću GCD testa utvrditi da li u sledećem kodu postoji loop-carry zavisnost.
    for i = 1, n1

for j = 1, n2
for k = 1, n3
a(2i + 2j - 3 * k) = a(4i - 6j + 3)
GCD(2, 3, 4, 6) = 1
1 deli 3
(varijanta 2: a(2i + 2j) = a(4i - 6j + 3))
GCD(2, 4, 6) = 2
2 ne deli 3

23. Objasniti osnovne transformacije koje se mogu primeniti na gnezdo petlji.
Permutacija - omogućava zamenu mesta dvemu petljama. Ako želimo da zamenimo mesta petljama po indeksima i i j, ta transformacija se opisuje pomoću matrice T koja se dobija kada se u jediničnoj matrici zamene mesta i-toj i j-toj vrsti.
Obrtanje - omogućava promenu redosleda izračunavanja po odredjenoj indeksnoj promenljivoj
Transformacija se opisuje jediničnom matricom u kojoj je u i - toj vrsti dijagonalni element jednak -1.
Krivljenje - ovom transformacijom se obavlja krivljenje jednog iterativnog indeksa u odnosu na drugi za faktor f. Transformacija se opisuje jediničnom matricom, pri čemu se faktor f dodaje u preseku kolone i dijagonale.
Primer matrice: krivljenje k u odnosu na j za faktor f.
1 0 0 i
0 1 0 j
0 f 1 k

24. Da li je na sledećem gnezdu petlji dozvoljeno primeniti
for i = 1, n
for j = 1, n
c(i, j) = c(i - 1, j +1) + 1
endfor {i, j}
   a. permutaciju petlji i i j
   b. obrtanje po indeksnoj promenljivoj i
   c. kompoziciju transformacija pod a i b
Tamo gde je moguće izvršiti transformaciju prikazati kako izgleda transformisano gnezdo petlji.

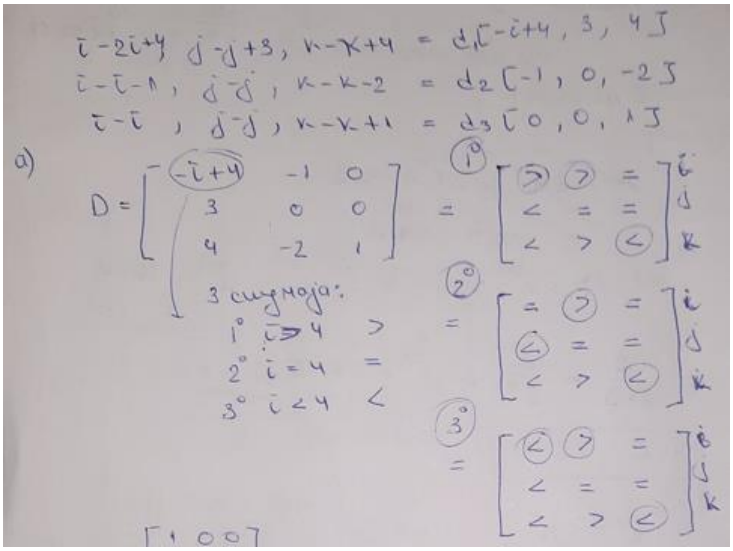


25.

a. Odrediti sve vektore zavisnosti i pravce zavisnosti za zadato gnezdo petlji.
b. Za svaku od petlji odgovoriti da li se može paralelizovati/vektorizovati, i ako ne može zbog kojih zavisnosti ne može
c. Primeniti odgovarajuću transformaciju tako da se unutrašnja petlja može vektorizovati. Pokazati ceo postupak kako se dolazi do transformisane petlje.

```
for(i = 2; i < N - 1; i++) {
        for(j = 3; j < N; j++) {
                for(k = 4; k < N - 3; k++) {
                   A[i][j][k] = A[2i - 4][j - 3][k - 4] + A[i + 1][j][k + 2] + A[i][j][k - 1];
}}}
```

a.



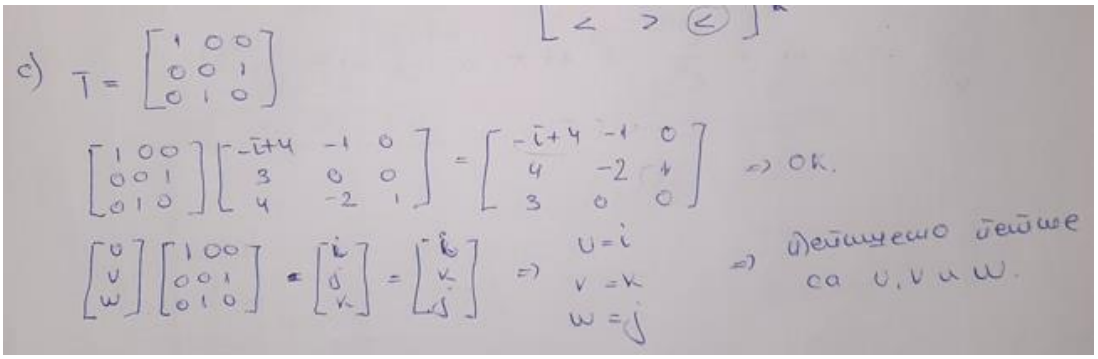
b.

a. случај **j > 4**:
   i - P(ne), V(ne), WAR zavisnosti
   j - P(**da**), V(**da**) nema zavisnosti
   k - P(ne), V(ne) loop-carry zavisnost
b. случај **j = 4**:

i - P(ne), V(ne), WAR zavisnosti
j - P(ne), V(ne), loop-carry zavisnost
k - P(ne), V(ne), loop-carry zavisnost

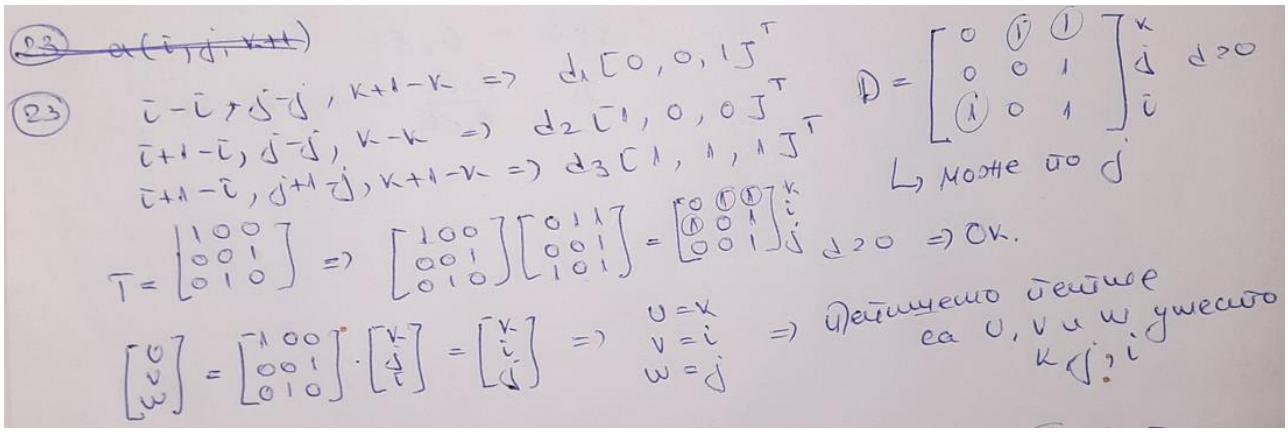
c. slučaj **j < 4**:
i - P(ne), V(ne), loop-carry i WAR zavisnosti
j - P(**da**), V(**da**), nema zavisnost
k - P(ne), V(ne), loop-carry zavisnost

c.



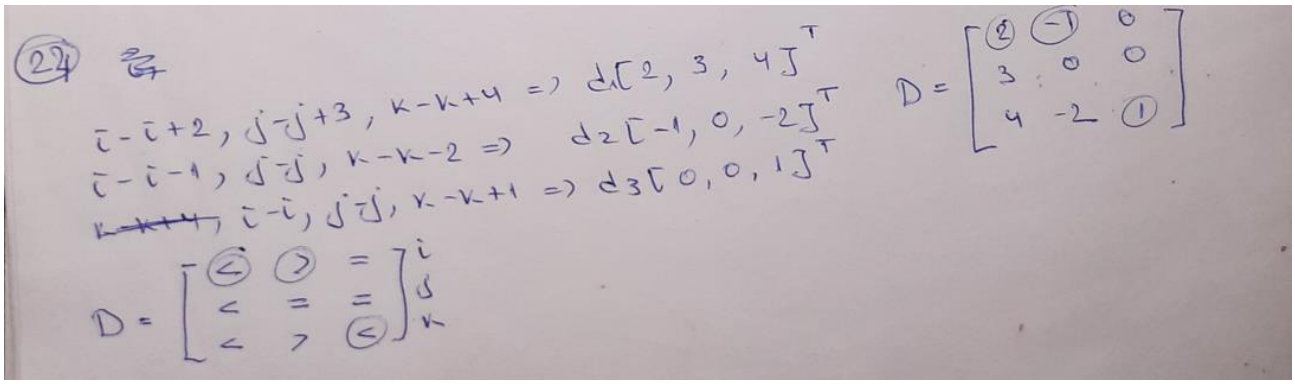
26. Po kojoj indeksnoj promenljivoj je moguće izvršiti vektorizaciju sledećeg gnezda petlji? Pokazati kako izgleda transformisano gnezdo petlji u kome se unutrašnja petlja može vektorizovati.

```
for(k = 0; k < L; ++k) {
        for(j = 0; j < M; ++j) {
                for(i = 0; i < N; i++) {
                        a[i][j][k + 1] = a[i][j][k] + X1;
                        b[i +1][j][k] = b[i][j][k] + X2;
                        c[i + 1][j + 1][k + 1] = c[i][j][k] + X3
                }
        }
}
```



27. Za svaku petlju u sledećem kodu navesti da li se može paralelizovati ili ne, i u slučaju da nije navesti koje zavisnosti sprečavaju paralelizaciju (navesti tip zavisnosti i članove koji su uključeni c1, c2, c3, c4)

```
for(i = 2; i < N; i++) {
        for(j = 3; j < N; j++) {
                for(k = 4; k < N - 3; k++) {
                        A[i][j][k](c1) = A[i - 2][j - 3][k - 4](c2) + A[i + 1][j][k + 2](c3) +
A[i][j][k - 1];(c4)
                }
        }
}
```



Napomena: Paralelizacija petlje je moguća na bilo kojoj dubini. Petlja ne mora da bude spoljašnja da bi se mogla paralelizovati, to je samo poželjno jer nema n-1 fork-join-a. zavisnosti: < (loop-carry), > (WAR), = (RAW koji ne sprečava ni paralelizaciju ni vektorizaciju)

petlja i paralelizacija nije moguća
zavisnosti: loop-carry(c2), WAR(c3)
petlja j paralelizacija je moguća
nema zavisnosti i može da se paralelizuje (zato što nije spoljna petlja pa ne nosi zavisnosti, u slučaju da je spoljna ceo red mora da bude =).
petlja k paralelizacija nije moguća
zavisnosti: loop-carry(c4)

28. Mnoge zavisnosti u programu mogu biti eliminisane preuređenjem koda. Zadata je sledeća petlja

```
for(i = 0; i < N; ++i)
{
        a[i] = b[i] + C;        //S1
        d[i] = a[i] + E;        //S2
}
```

Da li je korektno izvršiti preuređenje ove petlje na sledeći način (obavezno obrazložiti)

```
d[0] = a[0] + E
for(i = 1; i < N; ++i)
{
        a[i - 1] = b[i - 1] + C; //S1
        d[i] = a[i] + E;        //S2
```

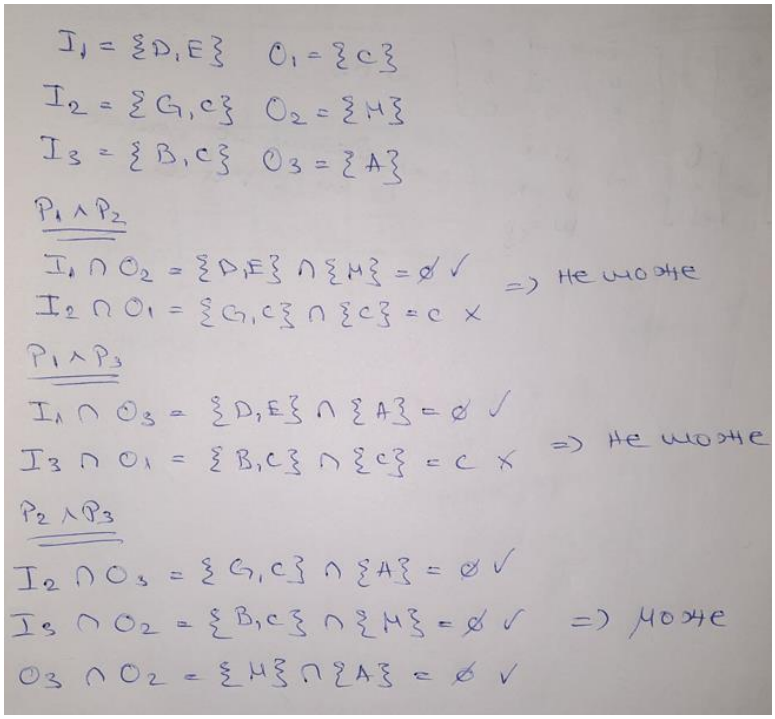
}
a[n] = b[n] + C
Petlja pre preuređenja imala je samo RAW hazard kod a[i], nakon uređenja to se pretvorilo u loop-carry zavisnost, tako da preuređenje nije korektno.

29. (3) Ako je latentnost memorije (8, 6) clk ciklusa a vektorski korak (10, 5), koliko je najmanje memorijskih banaka potrebno da bi se elementima vektora pristupilo bez konfilkata ako broj banaka mora biti stepen dvojke?
Konflikt kod pristupa memorijskoj banci nastupa ako važi sledeći uslov:
NZS(vektorski_korak, broj_mem_banaka) / vektorski korak < latentnost_mem_sistema

NZS(10, n) / 10 < 8
NZS(10, n) < 80
n = 16
provera: NZS(10, 16) < 80
80 < 80 **=> nema konflikta 80 == 80, nije manje**

NZS(5, n) / 5 < 6
NZS(5, n) < 30
n = 8
provera: NZS(5, 8) < 30
40 < 30 **=> nema konflikta**

30. Korišćenjem Bernštajnovih uslova utvrditi koje naredbe se mogu izvršavati paralelno
P1: C = D x E
P2: M = G + C
P3: A = B + C



31. Sledeća pitanja odnose se na OpenMP
    a. Na koje se sve načine može definisati broj niti koji se kreira u paralelnom regionu i koji su njihovi prioriteti?
        - **if** odredba - određuje da li će parallel direktiva dovesti do kreiranja tima niti. Ako je vrednost logičkog izraza false, onda se paralelni region izvršava samo od strane jedne niti.
        - **num_threads** odredba
        - **omp_set_num_threads()** biblitečka funkcija - nakon početka izvršenja korišćenjem izvršne bibliotečke funkcije
        - **OMP_NUM_THREADS** promenljiva okruženja -
        - **Default** - zavisno od implementacija - ako broj niti nije eksplicitno naveden, OpenMP će automatski odrediti broj niti na osnovu dostupnih resursa sistema.
    b. U čemu je razlika između private i firstprivate odredbi?
    Privatne promenljive paralelnog regiona nisu inicijalizovane na ulasku u paralelni region. Firstprivate promenljiva će na početku paralelnog bloka biti inicijalizovana na vrednost koju je imala istoimena promenljiva pre ulaska u paralelni region.
    c. U čemu je razlika imeđu #pragma omp single i #pragma omp master direktiva?
    Single direktivom se postiže da se strukturni blok izvršava od strane samo jedne niti, ali pre izvršenja nije poznato koja će nit izvršiti blok. Master direktiva označava da će samo glavna (master) nit izvršiti stukturni blok. Ova direktiva nema implicitnu barijeru na ulasku ili nakon okončanja strukturnog bloka.
    d. Koje odredbe se mogu koristiti uz direktivu #pragma omp for?
    private, firstprivate, lastprivate, reduction, ordered, schedule, nowait

e. Potrebno je paralelno izvršiti 100 iteracija petlje. Ako su kreirane 4 niti, prikazati kako će izgledati distribucija iteracija po nitima ako se koriste odredbe.

- schedule(static, 10)
  1. (0-9) … (40-49) … (80-89)
  2. (10-19) … (50-59) … (90-99)
  3. (20-29) … (60-69)
  4. (30-39) … (70-79)
- schedule(dynamic, 10)
  1. (0-9) … (60-69) … (90-99)
  2. (20-29) … (50-59) … (80-89)
  3. (10-19) … (40-49) - da ga jebem nisam procesor
  4. (30-39) … (70-79)
- schedule(guided, 10)
  1. (0-24) … (67-76)
  2. (25-42) … (77-86)
  3. (43-56) … (87-96)
  4. (57-66) … (97-99)

32. Uz OpenMP direktive mogu se koristiti određenje odredbe (klauzule). Popuniti donju tabeli navodeći oznaku x tamo gde se određena klauzula može koristiti navedenu direktivu.

	#pragma omp parallel	#pragma omp for	#pragma omp sections
shared	x		
private	x	x	x
lastprivate		x	x
firstprivate	x	x	x
nowait		x	x
schedule		x	
if	x		
num_threads	x		
ordered		x	
reduction	x	x	x

33. Sinhronizacija ili koordinacija akcija niti (thread) je nekad neophodna da bi se obezbedio korektan pristup deljivim prodacima i da bi se sprečilo narušavanje podataka. OpenMP nudi relativno mali skup jednostavih sinhronizacionih

mehanizama. Koje su direktive na raspolaganju u OpenMP za sinhronizaciju između niti? Ukratko objasniti sintaksu i dejstvo svake od njih.

- **#pragma omp barrier** - Kod ovog vida sinhronizacije zahteva se da sve niti stignu do barijere pre nego što nastave dalje sa izvršenjem.
  - Sintaksa:
    #pragma omp barrier
- **#pragma omp ordered** - Omogućava da se strukturni blok **u okviru paralelne petlje** izvrši sekvencijalno. Kada nit koja izvršava prvu iteraciju petlje naiđe na ordered direktivu, ona ulazi u strukturni blok bez čekanja. Da bi neka nit koja izvršva neku drugu iteraciju petlje izvršila naredbe koje se nalaze u strukturnom bloku mora da sačeka da sve niti koje izvršavaju prethodne iteracije okončaju izvršenje strukturnog bloka u okviru ordered konstrukcije Ako se koristi ova direktiva onda se mora iskoristiti i klauzula (odredba) ordered uz paralelni region u kojem se koristi direktiva ordered.
  - Sintaksa:
    #pragma omp ordered
    strukturni blok
- **#pragma omp critical** - Omogućava da niti pristupaju kritičnoj sekciji uzajamno isključivo. Kada neka nit naiđe na direktivu critical ona čeka da druga nit okonča pristup kritičnoj sekciji sa datim imenom (bilo gde u programu) pre nego što ona uđe u kritičnu sekciju.
  - Sintaksa:
    #pragma omp critical [(ime)]
    strukturni blok
- **#pragma omp atomic** - Atomic direktiva omogućava efikasno ažuriranje deljivih promenljivih od strane više niti na hardverskoj platformi koja podržava atomične operacije.Za razliku od ostalih direktiva, ona se primenjuje samo na jednu naredbu dodeljivanja koja neposredno sledi iza konstrukcije atomic.
  - Sintaksa:
    #pragma omp atomic
    naredba
- **#pragma omp master** - definiše blok naredbi koji će izvršiti samo master nit. Slična je single direktivi. Ova direktiva nema implicitnu barijeru na ulasku ili nakon okončanja strukturnog bloka.
  - Sintaksa:
    #pragma omp master
    strukturni blok
- **bibliotečke funkcje – lock** - nisu direktive

34.

a. Na raspolaganju je mutliprocesorki sistem sa 4 jezgra. Za paralelizaciju for petlje koja inicijalizuje na nulu gornji trougao kvadratne matrice dimenzije 100x100 koristi se OpenMP
   #pragma openmp parallel for private(j) shedule(...)
   for(i = 0; i < 99; i++)

```
for(j = i + 1; j < 100; j++)
        a[i][j] = 0.0;
```

Poređati sledeće odredbe počev od one koja će dati najbrže izvršenje do najsporijeg

- schedule(static) 6
- schedule(static, 10) 5
- schedule(static, 1) 2
- schedule(dynamic, 1) 1
- schedule(dynamic, 10) 3
- schedule(dynamic, 20) 4

b. Koja vrednost promenljive q će biti odštampana u sledećem programu. Obrazložiti.

```
void main() {
    int q = 2;
        #pragma omp parallel num_threads(4) reduction(* : q)
        {
            q += 2;
        }
        printf("%d\n", q)
}
```

q = 2 * (1 + 2) * (1 + 2) * (1 + 2) * (1 + 2) = 162

35. Sledeći sekvencijalni program

```
void ccode(float a[], float b[], float c[], int n) {
        float x, y;
        int i;
        for(i = 0; i < n; i++) {
                x = a[i] - b[i];
                y = b[i] + a[i];
                c[i] = x * y;
        }
}
```

Je paralelizovan pomoću OpenMP na sledeći način

```
void ccode(float a[], float b[], float c[], int n) {
        float x, y;
        int i;
        #pragma omp parallel for shared(a, b, c, n, x, y) private(i)
        for(i = 0; i < n; i++) {
                x = a[i] - b[i];
                y = b[i] + a[i];
                c[i] = x * y;
        }
}
```

Šta je pogrešno u ovom OpenMP programu što će uzrokovati da se ne generiše isti rezultat kao kod sekvencijalnog izvršenja? Napisati kako bi izgledala korektna verzija OpenMP programa.

```
void ccode(float a[], float b[], float c[], int n) {
        float x, y;
        int i;
        #pragma omp parallel for shared(a, b, c, n) private(i) lastprivate(x, y)
        for(i = 0; i < n; i++) {
                x = a[i] - b[i];
                y = b[i] + a[i];
                c[i] = x * y;
        }
}
```

36. Paralelizovati sledeći kod korišćenjem OpenMP

```
C[0] = 1;
for(i = 1; i < N; i++) {
        C[i] = C[i - 1];
        for(j = 0; j < N; j++) {
                C[i] *= A[i, j] + B[i, j];
        }
}
```

```
C[0] = 1;
for(i = 1; i < N; i++) {
        prod = C[i - 1];
        #pragma omp parallel for reduction(* : prod)
        for(j = 0; j < N; j++) {
                prod *= A[i, j] + B[i, j];
        }
        C[i] = prod;
}
```

ILI (bolje valjda, nema N-1 fork join-a)

```
C[0] = 1;
#pragma omp parallel for ordered schedule(dynamic)
for(i = 1; i < N; i++) {
        int prod = 1;
        for(j = 0; j < N; j++) {
                prod *= A[i, j] + B[i, j];
        }
        #pragma omp ordered
        C[i] = C[i - 1] * prod;
}
```

37. Da li se sledeći programski kod na neki način može paralelizovati pomoću OpenMP

```
for(i = 0; i < 100; i++)
        a[i + 1] = a[i] + 1;
```

Ako je moguće, dati kako izgleda paralelna verzija programa. Ako nije moguće, objasniti zašto.

Program nije moguće paralelizovati zbog loop-carry zavisnosti. Odnosno zavisnosti koje su posledica korišćenja vrednosti izračunatih u nekoj od prethodnih iteracija. Da bi se petlja mogla paralelizovati potrebno je da ne nosi zavisnosti, ni loop-carry, ni WAR.

Petlja se može transformisati na sledeći način gde se jasno uočava zavisnosti između iteracija.

```
for(i = 1; i < 100; i++)
        a[i] = a[i - 1] + 1;
```

38. Šta je pogrešno u sledećem OpenMP kodu (obavezno obrazložiti)

```
np = omp_get_num_threads();
#pragma omp parallel for schedule(static)
for(I = 0; I < np; I++)
        work(I);
```

omp_get_num_threads() - ova funkcija može da se pozove i izvan paralelnog regiona ali vratiće 1 zato što je aktivna samo jedna nit, master.

39. Šta je pogrešno u sledećem OpenMP kodu (obavezno obrazložiti)

```
void ccode(int n)
{
        #pragma omp parallel
        {
                #pragma omp critical
                {
                        work1();
                        #pragma omp barrier
                        work2();
                }
        }
}
```

#pragma omp barrier se ne sme biti unutar kritične sekcije. Samo jedna nit može jednovremeno ući u kritičnu sekciju, dok barijera zahteva da sve niti dođu do nje.

40. Sledeći OpenMP program se izvršava na 4-jezgarnom procesoru. Koliko se ubrzanje može postići za veliko N? Vreme potrebno za testiranje u if naredbi zanemariti. Usvojiti da ostale operacije traju 1 vremensku jedinicu.

```
float total = 0.0;
#pragma omp parallel for
for (int i = 0; i < N; i++) {
```

```
        if(i % 16 < 8) // usvojiti da testiranje zahteva 0 operacija
                out[i] = 1 * in[i]; // 1op
        else
                out[i] = 2 + in[i]; //1op
}
for(int i = 0; i < N; i++)
        total += out[i]; //1op
```

Sekvencijalno: N + N = 2N operacija
Paralelno (4 niti): N / 4 + N = 5N / 4 operacija
Ubrzanje: S(n) = T(1) / T(n) = 2N / (5N / 4) = 8 / 5=1.6

```
float total = 0.0;
#pragma omp parallel for reduction(+ : total)
for (int i = 0; i < N; i++) {
        total += i % 16 < 8 ? 1 * in[i] : 2 + in[i] //1op
}
```

S(n) = 2N / (N / 4) = 8

41. Koliko je maksimalno ubrzanje moguće ostvariti izvršenjem sledećeg OpenMP programa? Objasniti, rečima, kako se program može drugacije napisati da bi se postiglo veće ubrzanje.

```
void sub_a (float a[], float b[], float c[], float d[], int n, int m)
{
        int i, j;
        #pragma omp parallel shared (a, b, c, d, n, m) private(i, j)
        {
                #pragma omp sections nowait
                {
                        #prgma omp section
                        for(i = 1; i < n; i++) {
                                for(j = 0; j <= i; j++)
                                        b[j][i] = (a[j][i] + a[j][i-1]) / 2;
                        }

                        #pragma omp section
                        for(i = 1; i < m; i++) {
                                for(j = 0; j <= i; j++)
                                        d[j][i] = (c[j][i] + c[j][i - 1]) / 2;
                        }
                }
        }
}
```

```
void sub_a (float a[], float b[], float c[], float d[], int n, int m)
{
        int i, j;
        #pragma omp parallel shared (a, b, c, d, n, m) private(i, j)
        {
                #pragma omp for schedule(dynamic) nowait
                        for(i = 1; i < n; i++) {
                                for(j = 0; j <= i; j++)
                                        b[j][i] = (a[j][i] + a[j][i-1]) / 2;
                        }

                #pragma omp for schedule(dynamic) nowait
                        for(i = 1; i < m; i++) {
                                for(j = 0; j <= i; j++)
                                        d[j][i] = (c[j][i] + c[j][i - 1]) / 2;
                        }
        }
}
```

42.

a. Zadat je sledeći OpenMP program.

```
int i, j, n = 10, t = 0;
omp_set_num_threads(2);
#pragma omp parallel private(j)
{
   #pragma omp for
   for(i = 0; i < n; i++) {
          printf("a\n");
      }
      for(j = 0; j < n; j++) {
          printf("b\n");
      }
}
```

Koliko puta će biti odštampano a? Koliko puta će biti odštampano b?

a - 10

b - 20 - nema direktive za podelu poslova, obe niti izvršavaju celu for

b. Zadat je sledeći OpenMP program

```
int i, j, n = 100000, t = 0;
#pragma omp parallel for
   for(i = 0; i < n; i++) {
          t = t + 1;
   }
   printf("t = %d\n", t);
```

}

Koja od sledećih tvrdnji je tačna, kratko obrazložiti

a. Uvek će biti odštampano 100000
b. **Odštampana vrednost će biti 100000, a moguće su i vrednosti manje od 100000**
c. Bilo koja vrednost između -MAXINT i MAXINT je moguća

Može doći do trke jer t = t + 1 nije atomična operacija i nije u kritičnoj sekciji. Više niti mogu da pročitaju istu vrednost promenljive t, inkrementiraju je i upišu isti rezultat.

43. (2) Koja vrednost promenljive j će biti odštampana nakon izvršenja sledećeg koda. Šta bi bilo odštampano ako bi umesto master bila iskorišćena single direktiva?

```
int j;
#pragma omp threadprivate(j)
int main() {
        j = 1;
        #pragma omp parallel copyin(j) {
                #pragma omp master
                j = 2;
        }
        printf("j = %d\n", j);
}
```

Vrednost j nakon izvršenja biće 2, u slučaju single 1. Zato što je j deklarisana kao threadprivate i biće inicijalizovana na 2 u master niti zbog odredbe copyin. U slučaju single vrednost dodeljena u paralelnom regionu neće se održati izvan njega.

44. Prikazati moguće rezultate izvršenja sledećeg OpenMP programa ako se kreiraju dve niti

```
#pragma omp parallel
{
        printf("A ");
        printf("race ");
        printf("car ");
}
printf("n");
```

A A race race car car n
A race car A race car n
A race A race car car n
A race A car race car n
…

45.

1. a) Prikazati šta je izlaz iz sledećeg OpenMP programa, ako postoje 4 niti.

```
int i;
double sum = 0.0;
#pragma omp parallel for reduction(+:sum)
for (i=1; i <= 4; i++)
sum = sum + i;
printf("The sum is %lf\n", sum);
```

b) Prikazati šta je izlaz iz sledećeg OpenMP programa, ako postoje 4 niti.

```
int i;
double sum;
#pragma omp parallel for private (sum)
{
      sum = 0.0;
      for (i=1; i <= 4; i++)
      sum = sum + i;
      printf("The sum is %lf\n", sum);
}
```

c) Prikazati šta je izlaz iz sledećeg OpenMP programa, ako postoje 4 niti.

```
int i;
double sum = 0.0;
#pragma omp parallel for shared (sum)
{
      for (i=1; i <= 4; i++)
      sum = sum + i;
}
printf(„The sum is %lf\n“, sum);
```

d) Prikazati šta je izlaz iz sledećeg OpenMP programa, ako postoje 4 niti.

```
int i;
double sum = 0.0;
#pragma omp parallel for private (sum)
{
      for (i=1; i <= 4; i++)
      sum = sum + 1;
}
printf("The sum is %lf\n", sum);
```

a) **The sum is 10**
b) **The sum is 0** zato što je sum private (neće se ni pokrene jer je sum = 0.0 ispod for direktive, ali u slučaju da je direktiva odmah iznad petlje pokrenuće se)
c) **The sum is 10 ili manje od 10** jer se utrkuju ko će da upiše
d) **The sum is 0** zato što je sum private (u slučaju da da je direktiva odmah iznad petlje)

46. Napisati program na OpenMP kojim se vrši paralelno izračunavanje proizvoda bez korišćenja odredbe reduction i sa korišćenjem ove odredbe. Predvideti štampanje rezultata.

$$P = \prod_{i=0}^{99} x_i$$

```
#include <omp.h>
#include <stdio.h>

#define N 100

int main()
{
```

```
	double X[N];
	int i = 0;

	#pragma omp parallel for
	for (i = 0; i < N; i++)
		X[i] = i + 1;

	double productReduction = 1;

	#pragma omp parallel for reduction(*:productReduction)
	for (i = 0; i < N; i++)
		productReduction *= X[i];

	printf("The product with reduction is\n%lf\n", productReduction);

	double product = 1;
	double productLocal = 1;

	#pragma omp parallel shared(product, X) firstprivate(productLocal)
	{
		#pragma omp for
		for (i = 0; i < N; i++)
		{
			productLocal *= X[i];
		}
		#pragma omp critical
		{
			product *= productLocal;
		}
	}

	printf("The product without reduction is\n%lf\n", product);
	return 0;
}
```

47. Kako se može obaviti maskiranje procesnih elemenata u SIMD?
    - **Direktno maskiranje**
        - Statički određeno kompajlerom i kontrolnom jedinicom.
        - Po 1 bit za svaki procesni element.
        - Omogućava maskiranje proizvoljnog skupa.
    - **Maskiranje pomoću adrese PE**
        - Za N = 2^m PE, koristi se m-bitna maska.
        - Svaka pozicija u maski može imati vrednost 0, 1, X.
        - Aktivni su oni PE čija se adresa poklapa sa maskom.
        - Nije moguće maskirati proizvoljan podskup PE. (pr. nije moguće maskirati 000 i 111)
    - **Dinamičko maskiranje** (maskiranje podacima)
        - Svi procesni elementi izvrše test uslov zatim postavljaju svoj status na 1 ili 0, i podele se na dva disjunktna skupa, oni koji će izvršiti naredbe

ako je uslov zadovoljen i oni koji će izvršiti naredbe kad uslov nije zadovoljen.

48. Kojim sprežnim funkcijama su definisane sledeće sprežne mreže:
    a. mešanje-zamena (shuffle-exchange)

$$S(b_{m-1}, b_{m-2}, \ldots, b_1, b_0) = b_{m-2}, \ldots, b_1, b_0, b_{m-1}$$

$$E(b_{m-1}, b_{m-2}, \ldots, b_1, b_0) = b_{m-1}, b_{m-2}, \ldots, b_1, \bar{b}_0$$

    b. kub

Definisana sa $m=\log_2 N$ sprežnih f-ja

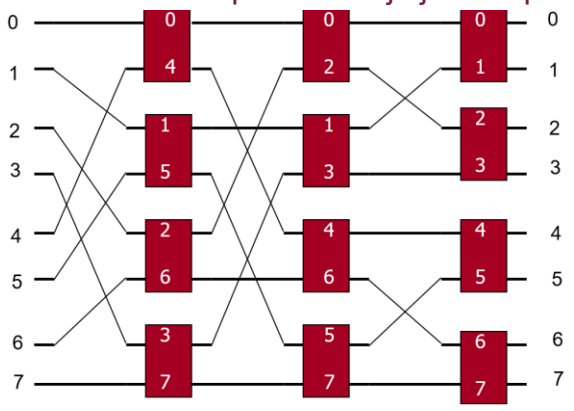
$$C_i(b_{m-1}, \ldots, b_i, \ldots . b_1, b_0) = b_{m-1}, \ldots, \bar{b}_i, \ldots . b_1, b_0,$$
$$i = 0,1,\ldots,m-1$$

    c. rešetka

Rešetka (mash): (dijametar $O(\sqrt{n})$, $r = \sqrt{N}$, N – broj PE)
Definisana sa 4 sprežne f-je:

- $M_{+1}(x)=(x+1) \bmod N$
- $M_{-1}(x)=(x-1) \bmod N$
- $M_{+r}(x)=(x+r) \bmod N$
- $M_{-r}(x)=(x-r) \bmod N$, $r = \sqrt{N}$

49. Višestepena sprežna mreža cub.
    - Ili ti generalizovani kub
    - Sastoji se od $\log_2 N = m$ stepena i N / 2 komutacionih elemenata
    - Svaki stepen realizuje jednu sprežnu funkciju kub mreže.



50. U statičkoj hiperkub mreži potrebno je poslati poruku iz čvora (0, 0, 1, 0, 0, 0) u čvor (1, 0, 1, 0, 1, 0). Prikazati preko kojih čvorova će poruka biti preneta do odredišta.
    001000 => 101000 => 101010

51. Kako se obavlja rutiranje poruka u sprežnoj mreži ako je ona realizovana kao
    a. Statička sprežna mreža
    b. Jednostepena dinamička sprežna mreža

c. Višestepena dinamička sprežna mreža

52. Navesti osnovne karakteristike i prednosti i mane multiprocesorskih i multiračunarskih sistema.

Multiprocesor

- Sadrži dva ili više homogenih procesora podjednakih mogućnosti
- Svi procesori dele pristup zajedničkoj memoriji ali svaki može imati i malu lokalnu memoriju.
- Procesor je sa zajedničkom memorijom povezan preko procesor-memorija sprežne mreže.
- Komunikacija između procesora ostvaruje se preko deljive memorije.
- Razmena upravljačkih signala preko procesor-procesor sprežne mreže.

Multiračunar

- Skup procesora može biti heterogen
- Svaki procesor ima svoju lokalnu memoriju i skup U/I uređaja
- Ne postoji zajednička memorija
- Komunikacija se ostvaruje slanjem poruka kroz sprežnu mrežu
- Sprežne mreže su statičke

53. (2) Objasniti problem keš koherencije i kako se rešava.

Kada više procesora pristupa zajedničkoj memoriji može doći do situacije gde jedan procesor ažurira vrednost podataka u svom lokalnom kešu dok drugi procesor čita isti podatak iz svoje verzije lokalnog keša. To stvara nekoherenciju između keševa što znači različiti procesori imaju različite vrednosti za isti podatak.

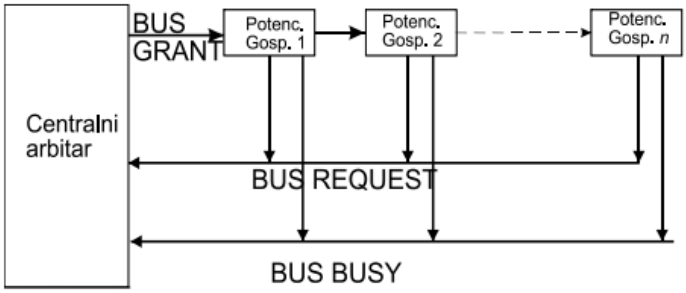
Postoji više prilaza za rešavanje ovog problema

- Hardverski implementirani protokoli za postizanje keš koherencije
- Softverske tehnike

54. (2) Objasniti arbitražu na magistrali koja se zasniva na deljivim zahtevima i lančanom zahvatanju.

- Svaki potencijalni gospodar magistrale izdaje zahtev za dodelu magistrale preko linije BUS REQUEST. Svi zahtevi su povezani žičanom ILI logikom.
- Kada centralni arbitar primi zahtev on predaje signal BUS GRANT potencijalnom gospodaru magistrale.
    - Linija BUS GRANT povezuje sve potencijalne gospodare u lanac, tako da gospodar1 predaje signal gospodaru2, itd…
    - Gospodar koji je izdao zahtev ne prenosi signal dalje i aktivira liniju BUS BUSY čime ukazuje da je zahvatio magistralu.
    - Kada tekući gospodar završi sa magistralom on oslobađa liniju BUS BUSY i naredni ciklus arbitraže može da počne.
- Politika dodele je zasnovana na fiksnim prioritetima, gospodar koji je bliži centralnom arbitru ima viši prioritet.
- Prednost je jednostavnost i mali broj linija za povezivanje.

- Nedostatak je to što je prioritet određen fizičkom pozicijom. Može doći do izgladnjivanja gospodara sa najnižim prioritetom u slučaju da gospodar bliži arbitru često zahteva magistralu.



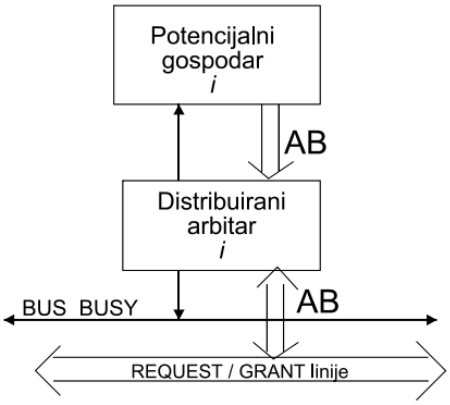
55. Navesti politike dodele magistrale kod multiprocesorskih sistema.

Politike mogu biti zasnovane na:

- **Prioritetu** - svakom potencijalnom gospodaru dodeljen je fiksni prioritet.
- **Nepristrasnosti** - potencijalni gospodari imaju jednak prioritet. Svakom gospodaru koji je izdao zahtev mora se garantovati dodela magistrale pre nego što se bilo kom drugom gospodaru po drugi put dodeli magistrala.
- **Kombinovana** - politika zasnovana na prioritetu i nepristrasnosti. U/I zahtevima se vrši dodela zasnovana na prioritetima, a procesorskim zahtevima na politici nepristrasnosti.

56. Objasniti distribuiranu arbitražu na magistrali kod multiprocesorskih sistema?

- Hardver za arbitražu je raspodeljen po potencijalnim gospodarima magistrale
- Svaki potencijalni gospodar ima sopstveni arbitar i jedinstveni **arbitracioni broj**.
- Kada dva ili više gospodara zahtevaju korišćenje magistrale, **pobednik je onaj koji ima viši arbitracioni broj**.
- Svaki potencijalni gospodar može poslati svoj arbitracioni broj na deljive REQUEST/GRANT linije.
- Od svih zahteva se formira **zbirni arbitracioni broj.**
- Svaki arbitar poredi svoj arbitracioni broj sa zbirnim, počev od bita najveće težine
    - Ako je njegov broj niži to znači da je njegov prioritet niži i uklanja svoj zahtev.
    - Na kraju na linijama ostaje arbitracioni broj onog gospodara koji ima najviši prioritet i njemu se dodeljuje magistrala.



57. (2) Ako nastupi promašaj pri upisu kod obraćanja lokalnom kešu koje se akcije preduzimaju i koje je rezultujuće stanje lokalnih udaljenih kopija ako se koristi
    a. Write-once protokol
       Kada nastupi promašaj pri upisu procesor mora prvo da pribavi kopiju iz glavne memorije ili iz keša koji ima dirty kopiju. Ovo se postiže slanjem Read_Inv komande koja će pribaviti i ažurirati blok i invalidirati sve ostale keš kopije. Lokalna kopija se zatim nalazi u stanju dirty
    b. Firefly protokol
       - **nema kopije u keševima**
         - pribavljanje iz glavne memorije Read_Blk
         - ažuriranje P_Write
         - stanje je Dirty
       - **postoji kopija u nekom kešu**
         - pribavljanje iz nečijeg keša Read_Blk
         - ažuriranje P_Write
         - ažuriranje svih kopija Write_Update
         - stanje je Shared

    Objasniti koje se lokalne procesorke komande i komande konzistencije tom prilikom emituju.

58. (3) (32, 64) PE povezana u hiperkub višestepenom sprežnom mrežom.
    a. Koliko stepena ima mreža i koliko komutacionih elemenata?
       N = 32 => $\log_2 N$ = **5** stepena
       N / 2 = **16** komutacionih elemenata

       N = 64 => $\log_2 N$ = **6** stepena
       N / 2 = **32** komutacionih elemenata
    b. Iz PEa sa adresom 21 potrebno je poslati poruku PE-ovima sa adresama 9, 13, 25, i 29. Da li je moguće obaviti emisiju? Ako jeste kako izgleda zaglavlje koje se dodaje poruci?
       Da li je moguće obaviti emisiju?
       - Broj odredišta je stepen dvojke? **DA**

- Odredišne adrese se razlikuju na j = 2 pozicija, a poklapaju na m - j = 3 pozicija? **DA**

=> Emisiju je moguće obaviti

21: 10101

9: 01001 T_9 = 10101 ⊕01001 = 1**1**1**0**0

13: 01101 T_{13} = 10101 ⊕01101 = 1**1**0**0**0

25: 11001 T_{25} = 10101 ⊕11001 = 0**1**1**0**0

29: 11101 T_{29} = 10101 ⊕11101 = 0**1**0**0**0

R: T9 = 111000

B: T_9 ⊕T_{29} ili T_{13} ⊕T_{25}

{R, B} = {111000, 10100}

c. Iz PEa sa adresom 60 potrebno je poslati poruku PE-ovima sa adresama 37, 39, 53, 55. Da li je moguće obaviti emisiju? Ako jeste kako izgleda zaglavlje koje se dodaje poruci?

Da li je moguće obaviti emisiju?

- Broj odredišta je stepen dvojke? **DA**
- Odredišne adrese se razlikuju na j = 2 pozicija, a poklapaju na m - j = 4 pozicija? **DA**

=> Emisiju je moguće obaviti

60: 111100

37: 100101 T_{37} = 111100 ⊕100101 = **0**1**1**0**0**1

39: 100111 T_{39} = 111100 ⊕100111 = **0**1**1**0**1**1

53: 110101 T_{53} = 111100 ⊕110101 = **0**0**1**0**0**1

55: 110111 T_{55} = 111100 ⊕110111 = **0**0**1**0**1**1

R: T_{37} = 011001

B: T_{37} ⊕T_{55} = 010010

{R, B} = {011001, 010010}

59.

a. U ESC (Extra Stage Cube) višestepenoj sprežnoj mreži koja povezuje 16 PE potrebno je poslati poruku iz čvora sa adresom 5 u čvor sa adresom 11. Kako izgleda zaglavlje poruke za primarni i sekundarni put?

S = 0101, D = 1011 => T = S⊕D = 1110

Primarni put: 0t(m-1)t(m-2)...t0 = 01110

Sekundarni put: 1t(m-1)t(m-2)...~t0 = 11111

b. U ESC (Extra Stage Cube) višestepenoj sprežnoj mreži koja povezuje 16 PE potrebno je poslati poruku iz čvora sa adresom 3 u čvorove sa adresama 4 i 6 (obaviti emisiju). Kako izgleda zaglavlje poruke za primarni i sekundarni put?

N = 16 = 2^m => m = 4

Emisija prema 2^j elementa, j = 1 => Adrese moraju da se razlikuju na najvise j bit-pozicija, i iste treba da budu na m-j bit pozicija:

S = 0101

E = 0100
F = 0110
TE = S ⊕E = 0001
TF = S ⊕F = 0011

Zaglavlje je kod emisije odredjeno skupom {R, B}
R = TE (moze i TF) = 0001
B = TE ⊕TF = E ⊕F = 0010

Primarni put: R_novo = 0r(m-1)r(m-2)...r0 = 00001
Sekundarni put: R_novo = 1r(m-2)r(m-2)...~r0 = 10011
B = 0 + 0010 = 00010

c. U hiperkub statičkoj sprežnoj mreži sa 16 procesora potrebo je poslati poruku iz čvora sa adresom 1001 u čvor 0110. Preko koji čvorova će poruka biti prosleđena do odredišnog čvora? Prikazati kako se došlo do rešenja.
S = 1001 -> 0001 -> 0101 -> 0111 -> 0110 = D

60. Potrebno je povezati 625 procesora. Na raspolaganju su vam sledeće sprežne mreže: a) magistrala, b) potpuno povezana mreža i c) 25x25 rešetka (mesh). Za svaku od sprežnih mreža navesti po jednu prednost i nedostatak. Koju sprežnu mrežu biste odabrali. Obrazložiti.
    a) **Magistrala**
    Prednost: najmanja cena
    Mana: najmanja propusnost (lako dolazi do zagušenja i nije pouzdana)
    b) **Potpuno povezana**
    Prednost: rutiranje
    Mana: najveća cena, veliki broj komutacionih elemenata
    c) **Mesh 25x25**
    Prednost: manje komutacionih elemenata od crossbar mreže, a brža je od magistrale
    Mana: rutiranje

    Biramo **Mesh** pokemona - (obrazloženje) - dimenzije rešetke odgovaraju broju procesora, brži je prenos od magistrale i nema previše komutacionih elemenata.

61. (2) Multiprocesorski sistem se sastoji od tri procesora P1, P2, P3. Deljiva memorija se sastoji od 4 bloka x, y, z i w. Svaki procesor ima keš u kojime se može naći samo jedan blok u datom trenutku. Za postizanje keš koherencije korisi se Write-once protokol. Pretpostaviti da su keš memorije inicijalno prazne a da su sadržaji memorijskih blokova
    x 10, y 20, z 30, w 40
    (verzija 2: x 10, y 30, z 80, w 20)
    Prikazati sadržaje keševa i glavne memorije i stanje keš blokova i GM nakon svake operacije.

akcija	keš P1	keš P2	keš P3	glavna memorija x y z w
P1 read(x)	10 (Valid)			x: 10, y: 20, z: 30, w: 40
P2 read(x)	10 (Valid)	10 (Valid)		x: 10, y: 20, z: 30, w: 40
P3 read(x)	10 (Valid)	10 (Valid)	10 (Valid)	x: 10, y: 20, z: 30, w: 40
P1 x += 25	35 (Reserved)	10 (Invalid)	10 (Invalid)	x: 35, y: 20, z: 30, w: 40
P1 read(z)	30 (Valid)	10 (Invalid)	10 (Invalid)	x: 35, y: 20, z: 30, w: 40
P2 read(x)	30 (Valid)	35 (Valid)	10 (Invalid)	x: 35, y: 20, z: 30, w: 40
P3 x = 15	30 (Valid)	35 (Invalid)	15 (Dirty)	x: 35, y: 20, z: 30, w: 40
P1 z += 1	31 (Reserved)	35 (Invalid)	15 (Dirty)	x: 35, y: 20, z: 31, w: 40

62. U dvo-procesorskom sistemu koristi se write once protokol za postizanje keš koherencije. Incijalno su keš memorije procesora prazne. Promenljiva X koja se nalazi u glavnoj memoriji ima vrednost 0.

	Događaj u P1	Komanda konzistencije	Sadržaj keša u P1	Događaj u P2	Komanda konzistencije	Sadržaj keša u P2	Sadr mem lokac
	promašaj pri čitanju	Read_Blk	0 (Valid)				
			0 (Valid)	promašaj pri čitanju	Read_Blk	0 (Valid)	
a	pogodak pri upisu	Write_inv	1 (Reserved)			0 (Invalid)	
a			1(Invalid)	promašaj pri upisu	Read_Inv	2 (dirty)	

63. Sledeći program izvršava se paralelno na dvo procesorskom sistemu

I1: LOAD R1, [A] // čita sadržaj memorijske lokacije na adresi A i pamti u registar R1.
I2: LOAD R2, [B] // čita sadržaj memorijske lokacije na adresi B i pamti u registar R2.
I3: ADD R3, R1, R2 //sabira sadržaje registrar R1 i R2 i pamti registar R3.
I4: STORE [A], R3 //pamti sadržaj registra R3 u memoriju na adresu A.
I5: SUB R4, R4,1 //oduzima 1 od R4.
I6: BNZ R4, I1 //grananje na I1 ako R4 nije 0.

Ako procesori P1 i P2 izvršavaju instrukcije u sledećem redosledu:

I ako se promenljive A i B nalaze u dva razlilčita keš bloka, popuniti sledeću tabelu ako se koristi write-once protokol za postizanje keš koherencije. Inicijalno su keševi prazni.

Instrukcija	Stanje u kešu P1 za blok A	Stanje u kešu P1 za blok B	Stanje u kešu P2 za blok A	Stanje u kešu P2 za blok B
P1: I1	Valid			
P2: I1	Valid		Valid	
P1: I2	Valid	Valid	Valid	
P1: I3	Valid	Valid	Valid	
P1: I4	Reserved	Valid	Invalid	
P2: I2	Reserved	Valid	Invalid	Valid
P2: I3	Reserved	Valid	Valid	Valid
P2: I4	Invalid	Valid	Reserved	Valid

64. Prikazati sadržaje keš memorija i glavne memorije i stanje keš blokova nakon svake operacije navedene u tabeli za slučaj da se koristi write-once protokol.

		Glavna memorija	P keš		Q keš	
korak	događaj	lokacija X	lokacija X	stanje bloka	lokacija X	stanje bloka
0	Početna vrednost	5				
1	P čita X	5	5	Valid		
2	Q čita X	5	5	Valid	5	Valid
3	Q X += 5	10	5	Invalid	10	Reserved
4	Q čita X	10	5	Invalid	10	Reserved
5	Q X += 5	10	5	Invalid	15	Dirty
6	P X += 5	10	20	Dirty	15	Invalid
7	Q čita X	20	20	Valid	20	Valid

65. Pretpostavimo da imamo multiprocesorski sistem sa (1024, 512) procesora. Svaki procesor ima lokalnu keš memoriju kapaciteta 1 MB. Kapacitet glavne memorije je 1GB. Veličina keš bloka je 64B.

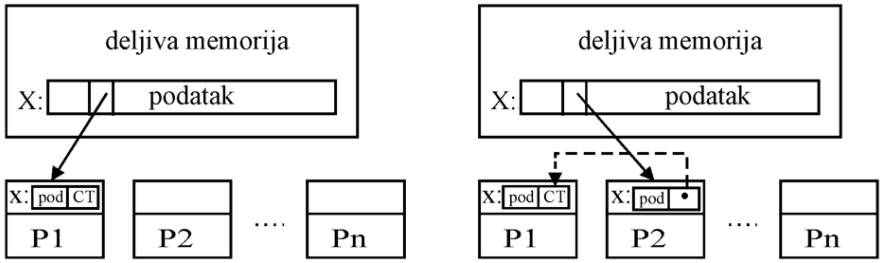
a. Ako se koristi write-once protokol za postizanje keš koherencije koliko bitova stanja zahteva implementacija ovog protokola.
   Keš blok ima 4 stanja: Valid, Invalid, Reserved i Dirty, koje kodiramo sa 2 bita.
   Broj keš blokova u lokalnoj keš memoriji procesora: 1MB / 64B = 2^{20} / 2^{6} = **2^{14}**
   Ukupan broj bitova stanja dobijamo kada pomnožimo broj procesora sa brojem keš blokova po procesoru i sve to sa 2b kojim kodiramo stanje keš bloka
   2b * 1024 * 2^{14} =2b * 2^{10} * 2^{14} = **2^{25} b**
b. Ako se koristi direktorijumska šema sa potpuno preslikanim adresarima koliko bitova stanja je potrebno za implementaciju protokola.
   Jedan dirty bit kao i po 1 bit za svaki keš blok
   Glavna memorija: (1024 + 1) * 1GB / 64B
   Lokalni keševi: 1024 * 2b * 1MB / 64B
   ukupno = (1024 + 1) * 1GB / 64B + 1024 * 2b * 1MB / 64B
c. Koju od ove dve šeme biste iskoristili za postizanje keš koherencije? Obrazložiti.

66. Multiprocesorski sistem od 256 procesora koristi direktorijumsku šemu sa ulančanim adresarima za održavanje keš koherencije. Svaki procesor ima privatnu keš memoriju kapaciteta 1MB, a veličina keš bloka je 64 byte. Veličina glavne memorije je 4 GB. Koliko ukupno bitova stanja zahteva implementacija ovog protokola?
    U direktorijumu jednog memorijskog bloka sa ulancanim adresarima, jedan valid bit i m bitova za adresu prvog procesora u lancu, gde je N=2^m, u ovom slucaju N=256 => m = 8.
    Kod kes bloka u svakom procesoru se koristi m bitova za adresu sledeceg ulancanog procesora.
    S obzirom da je velicina glavne memorije 4GB a velicina bloka 64B, to znaci da u memoriji ima 2^32 / 2^6 = 2^26 memorijskih blokova.
    U svakom procesoru ima 1 MB / 64B = 2^20 / 2^6 = 2^14 kes bloka, sto znaci da ima N * 2^14 = 2^8 * 2^14 = 2^22 kes blokova u sistemu.

    Ukupno bitova je Nb = (1+m)*br_mem_blokova + m*br_kes_blokova = 9*2^26 + 8*2^22 = 9*2^26 + 2^25

67. (2) Objasniti kako izgleda formiranje direktorijumskog lanca ako se koristi direktorijumska šema sa ulančanim adresarima.
    - Pretpostavimo da u sistemu ne postoji kopija bloka sa lokacijom X.
    - Ako P1 čita lokaciju X, glavna memorija šalje kopiju bloka kešu procesora P1 i ujedno pokazivač na kraj lanca CT (Chain Termination).
    - U u adresaru glavne memorije čuva se pokazivač na procesor P1
    - Ako P2 zahteva čitanje lokacije X, glavna memorija šalje kopiju bloka kešu procesora P2 i pokazivac na P1.
    - Sada se u adresaru glavne memorije čuva pokazivač na procesor P2
    - Ponavljanjem ovih koraka svi keševi mogu dobiti kopiju bloka.
    - Neka jedan od procesora, recimo P3, zahteva da modikuje sadržaj lokacije X.

- Da bi se održala konzistentnost potrebno je da se komanda invalidacije prosledi kroz lanac.
- Kontroler glavne memorije ne daje dozvolu modifikacije procesoru P3 sve dok ne primi potvrdu o invalidaciji od procesora koji sadrži CT pointer.



68. Objasniti kako se održava konzistentnost podataka kod direktorijumskih šema

Svaki memorijski modul sadrži poseban direktorijum koji beleži stanje i prisustvo memorijskog bloka u određenom kešu.
Svaki direktorijumski ulaz u glavnoj memoriji sadrži po jedan bit za svaki procesor u sistemu i jedan "dirty" bit.
Svaki bit pridružen procesoru ukazuje na status bloka u odgovarajućem procesorskom kešu (prisutan ili ne).
ako je "dirty" bit postavljen, tada je jedan i samo jedan procesorski bit postavljen i taj procesor ima pravo da vrši upis u kešovani blok (tj. da ga modifikuje).
Svaki keš direktorijum sadrži dva bita stanja za svaki kešovani blok.

- Jedan bit ukazuje da li je blok važeći, a
- drugi da li se u važeći blok može vršiti upis.

69.

a. U čemu je razlika između čvrsto spregnutih i slabo spregnutih MIMD sistema?
Multiprocesor sastoji se od homogenih procesora, multiračunar može od heterogenih.
Multiprocesor imaj zajedničku memoriju i skup U/I uređaja, multiračunar nema, svaki procesor ima svoju lokalnu.
Kod multiprocesora komunikacija se obavlja putem deljive memorije, dok kod multiračunara slanjem poruka kroz sprežnu mrežu.
…

b. Pretpostavimo da projektujete slabo spregnuti MIMD sistem. Koju vrstu keš koherentih protokola biste koristili? Zašto?
Slabo spregnuti MIMD sistem nema deljivu memoriju.

c. U multiprocesorskom sistemu sa privatnim keš memorijama deljivi podatak se čita od strane svih procesora u svakom klok ciklusu a modifikuje jednom na svakih 1000 ciklusa. Koju vrstu keš koherentog protokola biste koristili? Zašto?
   - Write-invalidate zato što program vrši mnogo više upisa od čitanja i pošto će uglavnom procesori raditi sa svojim lokalnim keš kopijama, manji će saobraćaj biti na magistrali.

d. Pretpostavimo da se koristi keš koherentni protkol baziran na direktorijumskim šemama. Pretpostavimo da za neki blok A, direktorijumski ulaz u glavnoj memoriji ima sve nule. Šta ovo govori o bloku?
Da ne postoji nijedna kopija tog bloka.

70. (2) Multiprocesorki sistem se sastoji od 3 procesora. Svaki procesor ima svoju keš memoriju. Prikazati kako se vrši promena bitova stanja u keš memorijama procesora i u direktorijumu glavne memorije ako se za održavanje keš koherencije koriste ograničeni adresari a maksimalni broj jednovremenih kopija je 2. Procesori pristupaju deljivoj promenljivoj X. Akcije procesora navedene su u prvoj koloni tabele. Popuniti ostale ćelije tabele.

Akcija procesora	Direktorijum u GM	Bitovi stanja u P1	Bitovi stanja u P2	Bitovi stanja u P3
P1 čita X				
P3 čita X				
P3 upisuje u X				
P1 čita X				
P2 čita X				

Akcija procesora	Direktorijum u GM			Bitovi stanja u P1		Bitovi stanja u P2		Bitovi stanja u P3	
	Dirty	kopija1	kopija2	Valid	Write	Valid	Write	Valid	Write
P1 čita X	0	P1		1	0				
P3 čita X	0	P1	P3	1	0			1	0
P3 upisuje u X	1	P1	P3	0	0			1	1
P1 čita X	0	P1	P3	1	0			1	0
P2 čita X	0	P2	P3			1	0	1	0

71. **(bonus)** Kako se sledeći kod može modifikovati tako da se postigne maksimalni paralelizam, a da konačni rezultat bude kao da se program sekvencijalno izvršio, tj. da na kraju x, a, b i c imaju iste vrednosti koje bi imale kada se kod sekvencijalnog izvršava. Prikazati kako bi izgledala paralelna verzija takvog koda u OpenMP.

```
x++;
a = x + 2;
b = a + 3;
c++;
```

```
x++;
#pragma omp parallel sections
```

```
{
        #pragma omp section
        a=x+2;
        #pragma omp section
        b=x+5;
        #pragma omp section
        c++;
}
```

CUDA Pitanja

1. Neka je dat CUDA kernel i odgovarajuća funkcija koja ga poziva.

```
__global__ void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
        int i = threadIdx.x + blockDim.x * blockIdx.x;
        if(i < n)
                C_d[i] = A_d[i] + B_d[i];
}

int vectAdd(float* A, float* B, float* C, int n) //podrazumevati da su nizovi A, B i C dužine n
{
        int size = n * sizeof(float);
        cudaMalloc((void **) &A_d, size);
        cudaMalloc((void **) &B_d, size);
        cudaMalloc((void **) &C_d, size);
        cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
        cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);
        vecAddKernel<<<ceil(n/256), 256>>>(A_d, B_d, C_d, n);
        cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
}
```

   a. Ako su A, B, C nizovi od 1000 elemenata, koliko će blokova niti biti pokrenuto?
      ceil(1000 / 256) = 4 bloka
   b. Ako su A, B, C nizovi od 1000 elemenata, koliko će warp-ova biti u svakom od blokova?
      Pod pretpostavkom da jedan warp čine 32 niti, biće 256 / 32 = 8 warpa.
   c. Ako su A, B, C nizovi od 1000 elemenata, koliko će niti biti u gridu?
      4 * 256 = 1024 niti
   d. Ako su A, B, C nizovi od 1000 elemenata, da li će postojati divergentnost u izvršenju kernela? Ako da, u kojoj liniji. Objasniti zašto.
      Postojaće divergencija kod if (i < n) jer će samo niti sa vrednošću i manjom od n vršiti izračunavanja, dok će ostale preskočiti ovaj blok (pokreće se više niti od onog što je potrebno jer 1000 nije deljivo sa 256, pa se pokreće ceo nov blok sa 256 niti od kojih se više od pola ne koristi).
   e. Ako su A, B, C, nizovi od 768 elemenata, da li će postojati divergentnost u izvršavanju kernela? Ako da, u kojoj liniji? Objasniti zašto?

U ovom slučaju neće postojati divergencija jer će sve niti biti upošljene i izvršavaće identičan kod (768 je deljivo sa 256, ne pokreće se više niti od onog što je potrebno).

f. Šta dati kernel radi?
Vrši sabiranje dva vektora, A i B, i upisuje rezultat u vektor C. Svi vektori su na globalnoj memoriji uređaja.

g. Modifikovati kernel tako da svaki 256. element rezultujućeg niza sadrži srednju vrednost prethodnih 255 elemenata

```
__global__ void vecAddKernel(float* A_d, float* B_d, float* C_d, int
n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    float C_loc;
    __shared__ float avg = 0;

    if (i < n)
    {
        if (i == 0 || i % 256 != 0)
        {
            C_loc = A_d[i] + B_d[i];
            C_d[i] = C_loc;
            atomicAdd(&avg, C_loc / 255.f);
        }

        __syncthreads();

        if (i > 0 && i % 256 == 0)
            C_d[i] = avg;
    }
}
```

2.

a. Koji sve tipovi memorije postoje na GPU?
Postoje registri i keš memorija, globalna memorija uređaja i deljena memorija unutar Streaming Multiprocessora.

b. Koja je razlika između shared memorije i registara na GPU?
Registri su brži od deljive memorije. Svaka nit može da pristupa svojim lokalnim promenljivama u registrima. Deljenu memoriju dele niti unutar bloka.

c. Gde se specificira broj blokova i niti koje će izvršavati zadati kernel u CUDA programu? Specifira se prilikom poziva kernela, uz pomoć odgovarajuće izvršne konfiguracije (myKernel<<< n, m >>(args…), gde je n organizacija blokova na nivou grida, a m organizacija niti na nivou bloka).

d. Da li postoje ograničenja za tip povratne vrednosti kernel funkcije u CUDA programu?
   Da, *__global__* funkcije ne mogu imati povratnu vrednost (void su). One dalje mogu da pozovu *__device__* funkcije koje nemaju ovo ograničenje.

3.

a. Zašto se u CUDA kernelima izračunavanja dele dva puta, jednom na nivou rešetke, drugi put na nivou bloka?
   Zbog strukture GPU-a. On se sastoji od mnoštva Streaming Multiprocessora koji mogu da izvršavaju blokove niti.
b. Zašto niti iz različitih blokova ne mogu da koriste istu deljivu memoriju?
   Različiti blokovi se u većini slučajeva izvršavaju na različitim Streaming Multiprocessorima koji ne dele istu deljivu memoriju. Čak i kada se oni izvršavaju na istom SM-u, oni ne mogu da dele shared memoriju. Razlog je to što se unapred ne može garantovati redosled izvršavanja blokova niti, pa ne bi imalo smisla koristiti deljenu memoriju.
c. Pretpostaviti da je neki CUDA kernel pokrenut sa 1000 blokova niti, svaki sa 256 niti. Ako je neka promenljiva deklarisana kao lokalna promenljiva u kernel funkciji, koliko verzija te promenljive će biti kreirano za vreme izvršenja kernela?
   256 000
d. Potrebno je napisati CUDA kernel za sabiranje dva vektora. Ako želimo da svaka pojedinačna nit računa jedan element rezultujućeg vektora, koji bi izraz bio najpogodniji za mapiranje indeksa niti i bloka na indeks rezultujućeg vektora:
   i. i = threadIdx.x + threadIdx.y;
   ii. i = blockIdx.x + threadIdx.x;
   iii. i = blockIdx.x * blockDim.x + threadIdx.x; - sve niti unutar jednog warp-a će adresirati susedne memorijske lokacije, biće više keš pogodaka pošto niti u warpu dele keš memoriju
   iv. i = threadIdx.x * blockDim.x + blockIdx.x;
   v. i = blockIdx.x * threadIdx.x;
e. Za prethodno navedeni kernel, neka je dužina vektora 5000, neka svaka nit računa samo jedan element rezultujućeg vektora, i neka je veličina bloka 512 niti. Koliko će ukupno niti biti u gridu? 5120 (mora postojati najmanje 10 blokova niti)

4.

a. Da li niti koji izvršavaju CUDA kernel mogu međusobno da komuniciraju? Kako? Da li postoje ograničenja?
   Niti unutar istog bloka mogu da komuniciraju koristeći sinhronizaciju, atomske operacije i deljenu memoriju. Niti iz različitih blokova ne mogu da sarađuju.
b. Da li blokovi niti koji izvršavaju isti CUDA kernel mogu međusobno da komuniciraju? Kako? Da li postoje ograničenja?
   Ne.

5.

a. Kojim memorijama mogu da pristupe sve niti u jednom bloku?

Sve niti u jednom bloku na GPU mogu da pristupe globalnog memoriji uređaja i deljenoj memoriji SM koji izvršava taj blok.

b. Koja je prednost deljive memorije u CUDA-i?
Prednost deljive memoriju u odnosu na globalnu je drastično manje vreme pristupa, pa treba težiti smanjenju pristupa globalnoj memoriji korišćenjem deljene.

c. Šta je CUDA kernel?
CUDA kernel je program koji se izvršava na (nvidia) GPU.

d. Napisati C/C++ funkciju *float operatorX(int a, int b, float c)* koja vraća vrednost a*c-b

```
float operatorX(int a, int b, float c)
{
    return a*c-b;
}
```

e. Napisati C/C++ funkciju *float* calculateVector(float* A, int dim, int c)* koja najpre popunjava kvadratnu matricu A tako da je A[i][j] = operatorX(i,j,c), a vraća vektor u kome je i-ti element dobijen kao zbir elemenata i-te vrste matrice A.

```
float* calculateVector(float* A, int dim, int c)
{
    float* vec = (float*) malloc(dim * sizeof(float));

    for (int i = 0; i < dim, i++)
    {
        float sum = 0;
        for (int j = 0; j < dim, j++)
        {
            A[i][j] = operatorX(i, j, c);
            sum += A[i][j];
        }
        vec[i] = sum;
    }
    return vec;
}
```

f. Korišćenjem CUDA tehnologije, u programskom jeziku C/C++ napisati kernel funkciju koja odgovara funkciji *calculateVector*. Obratiti pažnju na efikasnost paralelizacije. Omogućiti pozivanje kernela za matrice proizvoljne veličine.

g. Za prethodno napisani kernel, napisati i C/C++ host kod kojim se omogućava njegovo pozivanje. Pretpostaviti veličinu bloka od 256 niti i broj blokova ne veći od 256.

6.

a. Šta označavaju kvalifikatori __*global*__, __*host*__, i __*device*__ u CUDA programu?
   U kontekstu funkcija:
   i. __global__ označava kernel (poziva se na hostu, izvršava na device)
   ii. __host__ označava funkciju koja se poziva i izvršava na hostu
   iii. __device__ označava funkciju koja se poziva i izvršava na device-u

   Takođe, kvalifikator __device__ može da označava globalnu promenljivu na GPU.
b. Da li sve niti koje se izvršavaju na GPU imaju pristup istoj deljivoj memoriji? Zašto?
   Ne, niti unutar jednog bloka mogu da pristupe samo deljivoj memoriji Streaming Multiprocessora na kom se taj blok izvršava.
c. Koji je tip povratne vrednosti kernel funkcije u CUDA programu?
   void

трећа срећа