

Računarski sistemi visokih performansi

Nikola Vukić, Petar Trifunović, Veljko Petrović

Računarske vežbe
Zimski semestar 2024/25.

OpenMP 1

Sadržaj

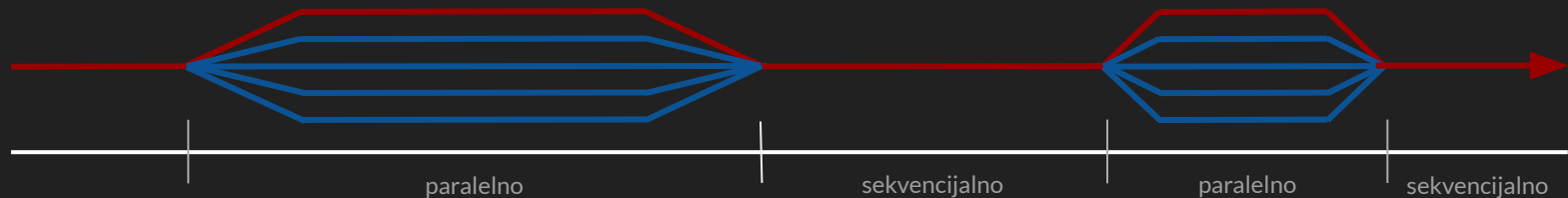
- Šta je *OpenMP*?
- *Fork-join* model.
- Kreiranje paralelnog regiona.
- Kompajliranje.
- Opseg vidljivosti promenljivih.
- Lažno deljenje.
- Sinhronizacija (eksplicitna i implicitna).
- Paralelna *for* petlja i redukcija.
- Ostale konstrukcije za podelu posla.

OpenMP

- API za programiranje paralelnih aplikacija na višeprocesnim (engl. *multiprocessing*) mašinama.
- Zasnovan na konceptu **deljene memorije**.
- Obuhvata skup kompajlerskih direktiva, bibliotečkih rutina i promenljivih okruženja.
- Postoji podrška za programske jezike **C**, **C++** i *Fortran*.

Fork-join model

Fork-join model



Ilustracija preuzeta iz knjige [Parallel Programming in MPI and OpenMP, Victor Eijkhout \(2017.\)](#)

- Paralelni region — deo programa koji izvršava tim niti.
- **Master nit:**
 - nit koja je postojala pre paralelnog regiona, a postoji i u njemu
 - uvek ima identifikator 0
- **Novokreirane niti:**
 - identifikatori od 1 do N-1
 - ne postoje nakon izlaska iz paralelnog regiona u kom su kreirane
- Sve niti u paralelnom regionu čine **tim niti**.

Kreiranje paralelnog regiona

Kreiranje paralelnog regiona

- Tim niti se kreira korišćenjem *omp parallel* direktive.

```
#pragma omp parallel [klauzule]
{
    ...
}
```

- U jednom programu može postojati proizvoljan broj paralelnih regiona.
- Blok koji sledi nakon *omp parallel* direktive:
 - ima tačno jednu ulaznu tačku,
 - ima tačno jednu izlaznu tačku (ne može sadržati *break* i *goto*; može sadržati *exit*).

Klauzule paralelnog regiona

- *num_threads(željeni_broj_niti):*
 - način za postavljanje broja niti u paralelnom regionu
 - druga dva načina za postavljanje broja niti:
 - funkcijom *omp_set_num_threads(int num_of_threads)*
 - postavljanjem promenljive okruženja *OMP_NUM_THREADS*
 - ako se koristi više od jednog načina, najviši prioritet ima klauzula, zatim funkcija, a na poslednjem mestu po prioritetu je promenljiva okruženja
 - klauzula utiče samo na broj niti u paralelnom regionu nad kojim je iskorišćena
 - ako se broj niti ne postavi, podrazumevano će se pokrenuti broj niti jednak broju **logičkih jezgara** računara

Klauzule paralelnog regiona

- *num_threads(željeni_broj_niti)*:
 - fizička jezgra:
 - hardverska jezgra računara
 - mogu imati više logičkih jezgara
 - logička jezgra:
 - proizvod sposobnosti fizičkih jezgara procesora da izvršavaju više stvari konkurentno
 - hardverski kontrolisane niti
 - na fizičkim jezgrima postoji realan paralelizam — zadaci se mogu izvršavati na svim fizičkim jezgrima istovremeno
 - logička jezgra omogućavaju konkurentno izvršavanje unutar fizičkog

Klauzule paralelnog regiona

- `num_threads(željeni_broj_niti)`

```
nikola@nikola-ThinkPad-L580:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Address sizes:         39 bits physical, 48 bits virtual
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Vendor ID:             GenuineIntel
Model name:            Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
CPU family:            6
Model:                 142
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
```

broj logičkih jezgara

broj logičkih po fizičkom jezgru

broj fizičkih jezgara

Klauzule paralelnog regiona

- *num_threads(željeni_broj_niti)*:
 - moguće je kreirati veći broj niti od broja logičkih procesora
 - *OpenMP* i operativni sistem će raspoređivati niti za izvršenje

Klauzule paralelnog regiona

- Klauzule za određivanje vidljivosti promenljivih:
 - *private*
 - *firstprivate*
 - *lastprivate*
 - *shared*
- Detaljnije o njima kasnije.

Hello, world!

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

int main()
{

#pragma omp parallel
{
    int id = omp_get_thread_num();
    printf("Hello(%d),", id);
    printf(" world!(%d)\n", id);
}

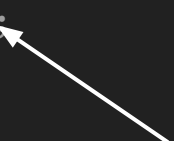
return 0;
}
```

Hello, world!

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

int main()
{
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        printf("Hello(%d),", id);
        printf(" world!(%d)\n", id);
    }

    return 0;
}
```



Funkcija koja vraća redni broj,
odnosno ID trenutne niti

Hello, world! — primer izvršavanja

Hello(0),Hello(1),Hello(6), world!(0)

Hello(3), world!(3)

world!(6)

Hello(2), world!(2)

Hello(4), world!(4)

world!(1)

Hello(5),Hello(7), world!(7)

world!(5)

- Primer je pokrenut na procesoru sa 8 logičkih jezgara.

Korisne funkcije

- `void omp_set_num_threads(int num_threads)` — postavlja broj niti za izvršavanje
- `int omp_get_thread_num(void)` — vraća ID trenutne niti
- `int omp_get_num_threads(void)` — vraća broj niti u trenutnom paralelnom regionu

Kompajliranje

Kompajliranje

- Pozicionirati se u direktorijum sa izvornom datotekom (izvornad.c).
- `$ gcc [-g] [-o izvorsnad] izvornad.c -fopenmp [-O2 | -O3]`
- Opcije:
 - `-g` — omogućava prikupljanje podataka za debugiranje
 - `-o` — specificira naziv izvršne datoteke
 - `-fopenmp` — omogućava obradu *omp* pragmi i generisanje *OpenMP* paralelnog okruženja
 - `-O2` ili `-O3` — naglašava kompajleru da primeni određeni skup optimizacija (`O3` je veći skup od `O2`); ne preporučuje se u kombinaciji sa `-g`

`_OPENMP` makro

- `#pragma` direktive nisu problem — kompajler ih ignoriše ukoliko ih ne prepozna.
- Pozivi funkcija iz `omp.h` biblioteke nisu poznate kompajleru i doći će do greške ako ne budu prepoznati.
- Kompajliranje sa `-fopenmp` opcijom, ukoliko bude uspešno, obezbediće da C prekompajler definiše `_OPENMP` makro.
- Svi kompajleri prilagođeni za rad sa *OpenMP*-jem moraju da definišu ovaj makro da bi ispoštovali standard.

`_OPENMP` makro

- Ispitivanjem ovog makroa mogu se izbeći greške ukoliko se kod izvršava u okruženju koje ne podržava *OpenMP*.

```
#ifdef _OPENMP  
... pozivi funkcija iz OpenMP API-ja...  
#endif
```

Pokretanje izvršne datoteke

- Kao i bilo koji običan C program.
- `$./izvrsnad <lista argumenata>`
- Ako nije korišćena `-o` opcija pri kompajliranju, naziv izvršne datoteke biće *a.out*.

Zadatak 1.0 — Računanje vrednosti broja π

- Korišćenjem samo *parallel* konstrukcije, paralelizovati program koji računa vrednost integrala:

$$\int_0^1 \frac{4}{(1+x^2)} dx$$

- Rezultat ovog integrala je jednak broju π . Šta se dešava sa rezultatom nakon paralelizacije?
- Primer rešenja: funkcija *parallel_code_incorrect*, datoteka *resenja/01_omp_pi.c*.

Opseg vidljivosti promenljivih

Opseg vidljivosti promenljivih

- Deljene:
 - deklarisanе izvan paralelnog regiona (npr. *sum* promenljiva iz rešenja prvog zadatka)
 - sve niti imaju pristup istoj promenljivoj — **trka do podataka**
- Privatne:
 - brojačka promenljiva *for* petlje **prvog nivoa**
 - promenljive deklarisanе unutar paralelnog regiona (uključujući i promenljive deklarisanе u funkcijama pozvanim iz paralelnog regiona)
 - svaka *OpenMP* nit ima svoju instancu privatne promenljive

Problem sa rešenjem zadatka 1.0

- Postoji štetno preplitanje u radu niti.
- Trka do deljene *sum* promenljive narušava ispravno računanje rezultata.

Zadatak 1.1 — Računanje vrednosti broja π

- Modifikovati rešenje prethodnog zadatka tako da se ukloni štetno preplitanje.
- Primer rešenja: funkcija *parallel_code*, datoteka *resenja/01_omp_pi.c*.

Lažno deljenje

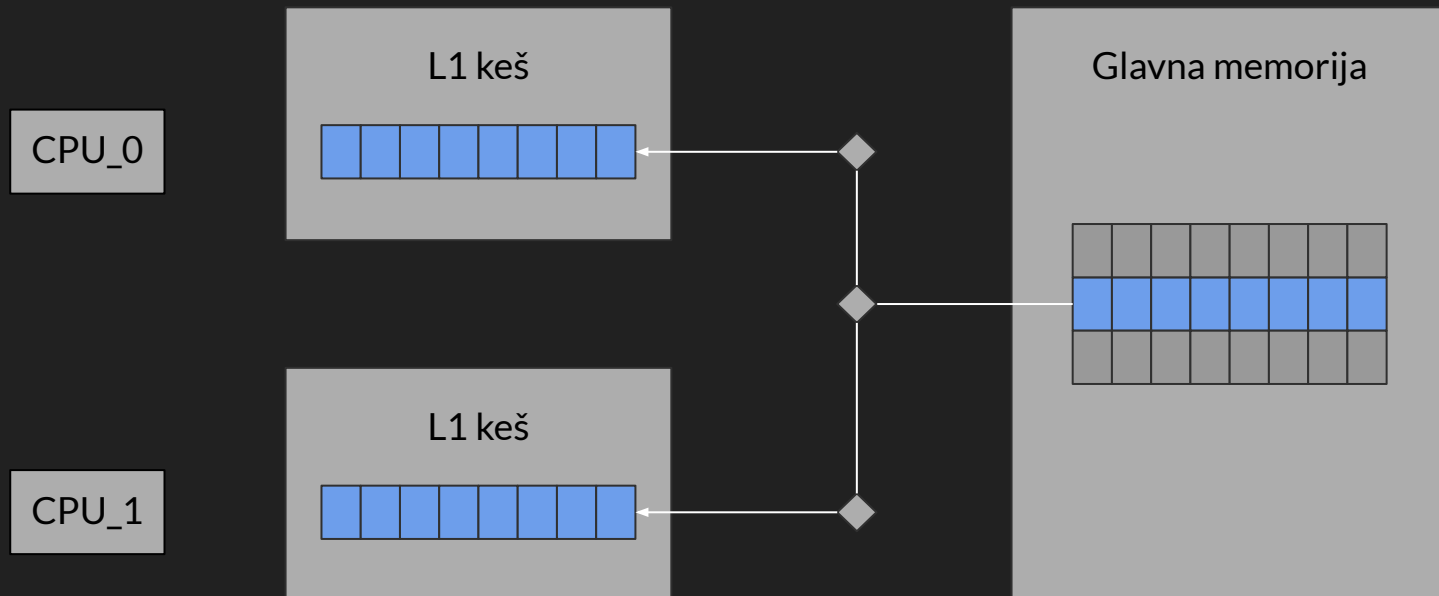
Problem sa rešenjem zadatka 1.1

- Pri ažuriranju parcijalnih suma, dolazi do problema *lažnog deljenja*.
- Lažno deljenje nastupa kada dve niti pristupaju različitim memorijskim lokacijama, ali su te lokacije dovoljno blizu u memoriji da se nađu u istoj keš liniji.

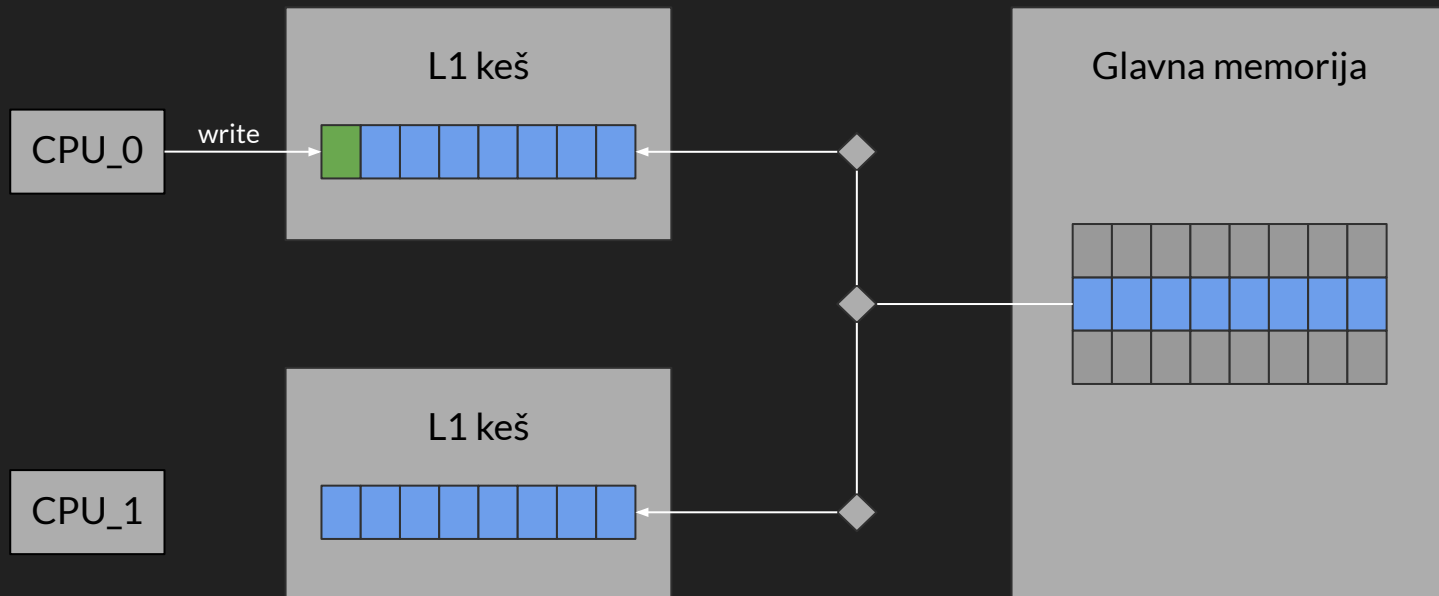
Problem sa rešenjem zadatka 1.1 — lažno deljenje

- Kada su podaci potrebni procesoru, pribavljaju se iz glavne memorije i smeštaju u keš (ukoliko već nisu u kešu).
- Iz memorije se ne pribavlja samo potreban podatak, već skup podataka veličine jednake veličini keš linije (engl. *cache line*). Veličina linije u bajtovima je neki stepen dvojke (\$ cat /proc/cpuinfo | grep cache_align).
- Kada se radi sa nizovima, nekoliko elemenata istog niza naći će se u istoj keš liniji.
- Kada nit izmeni svoj element, to može čitavu liniju u drugim kopijama keša označiti kao nevalidnu.
- Druge niti moraju da ažuriraju svoju kopiju keša tražeći drugu validnu kopiju ili pribavljajući novu iz glavne memorije, iako element sa kojim one rade nije promenjen. Ovo bespotrebno usporava rad.

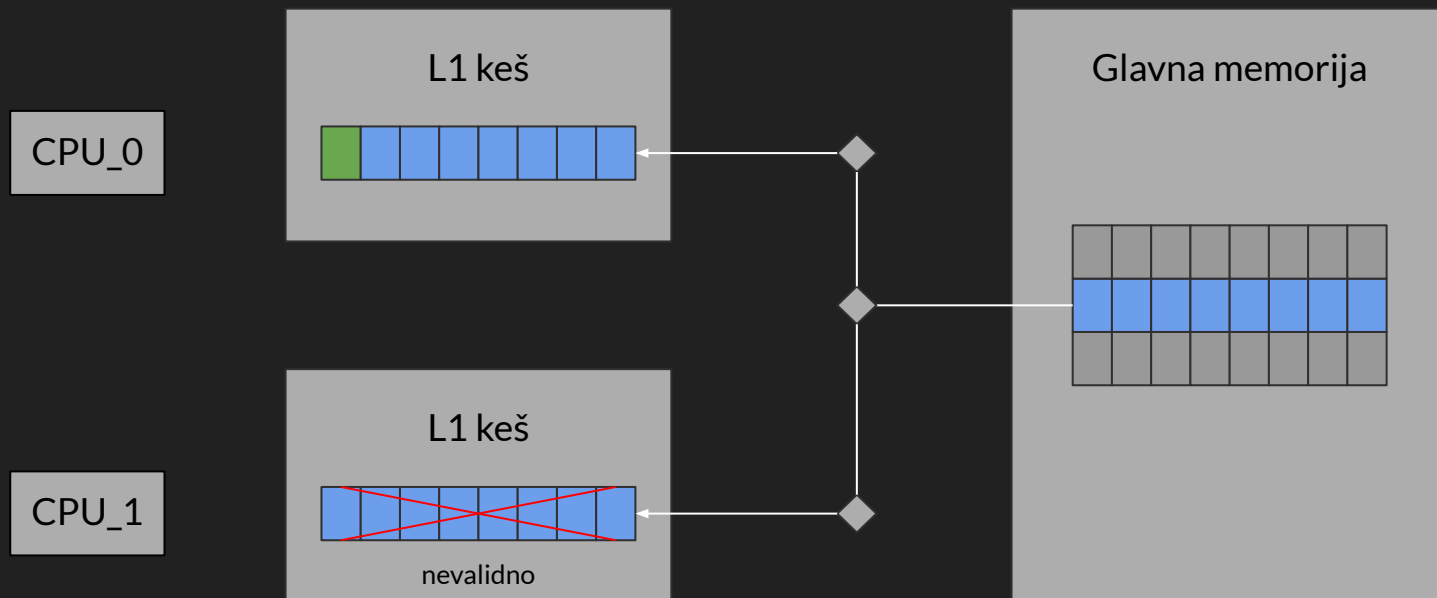
Problem sa rešenjem zadatka 1.1 — lažno deljenje



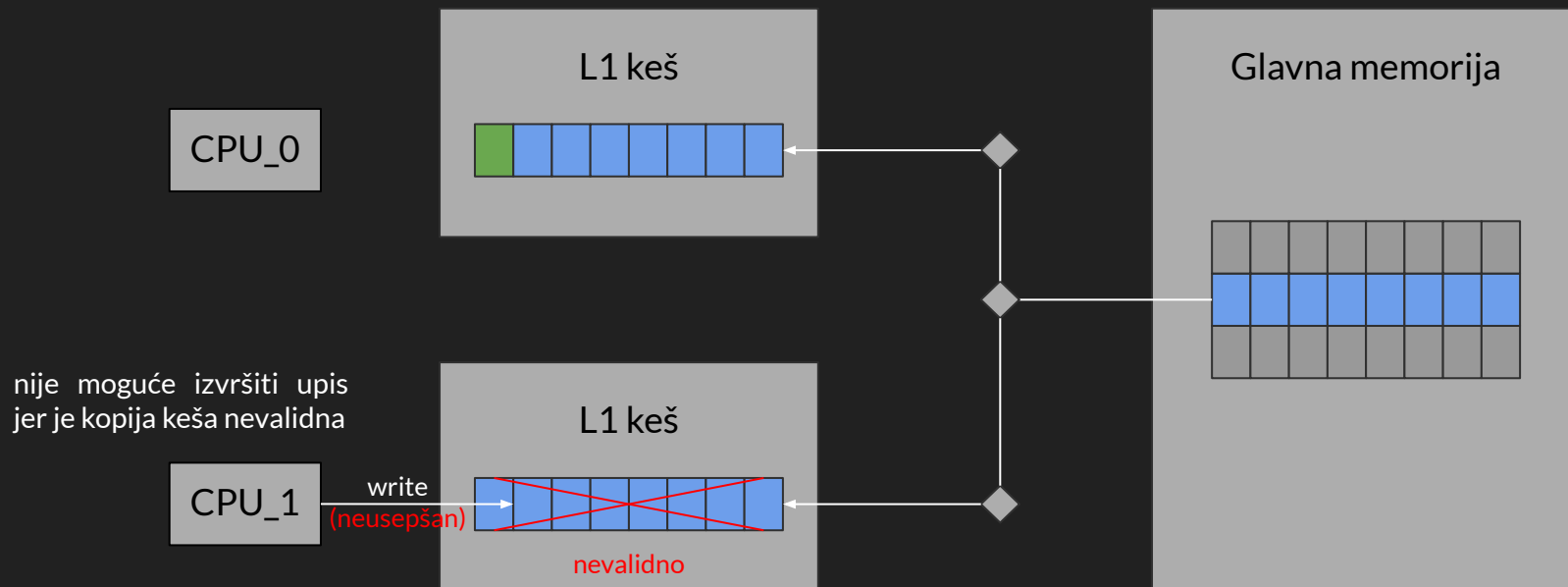
Problem sa rešenjem zadatka 1.1 — lažno deljenje



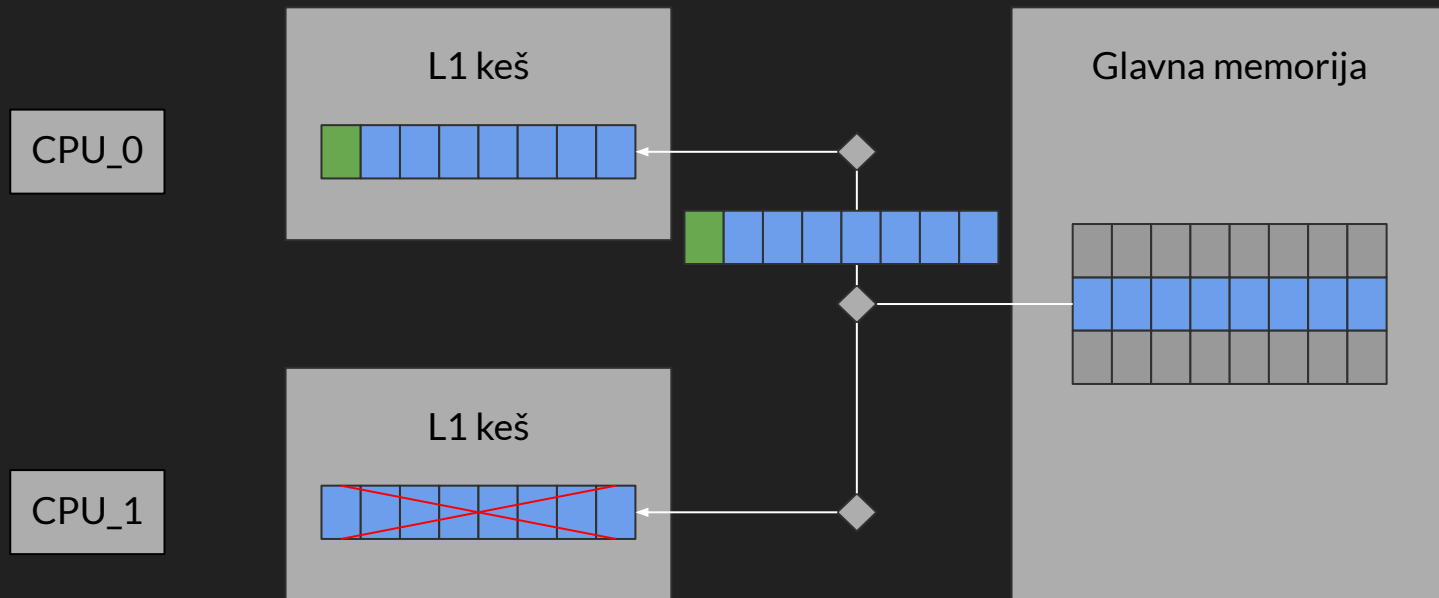
Problem sa rešenjem zadatka 1.1 — lažno deljenje



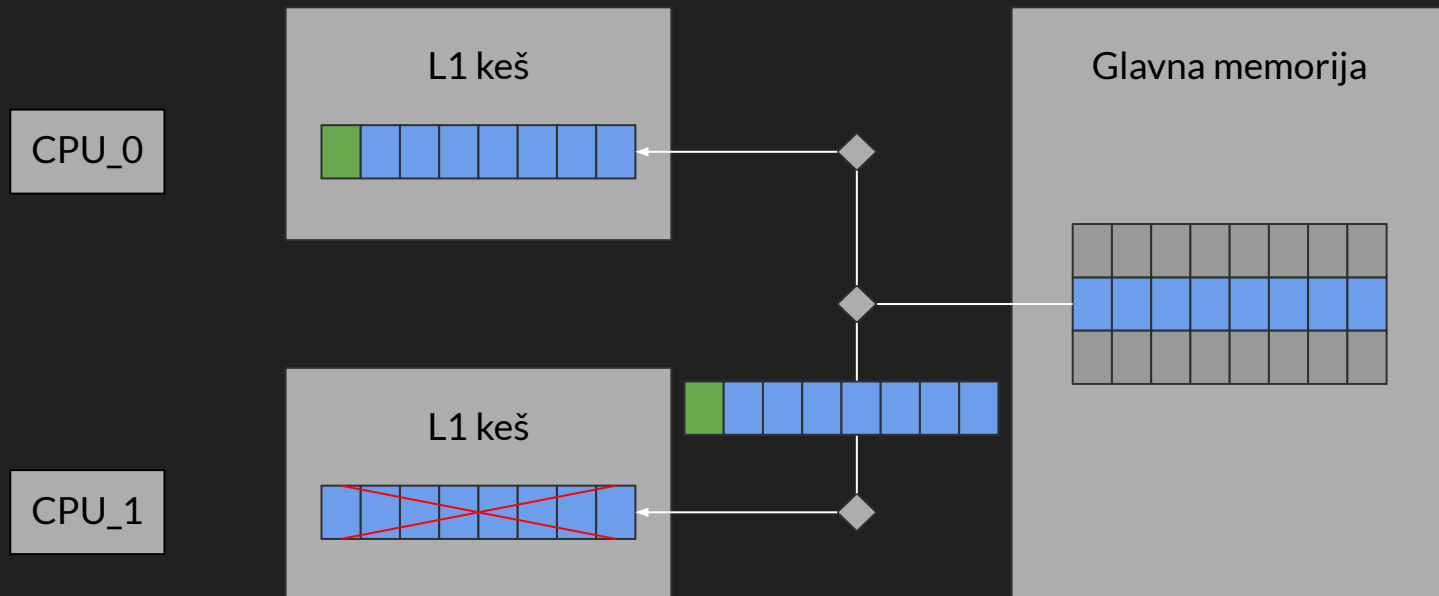
Problem sa rešenjem zadatka 1.1 — lažno deljenje



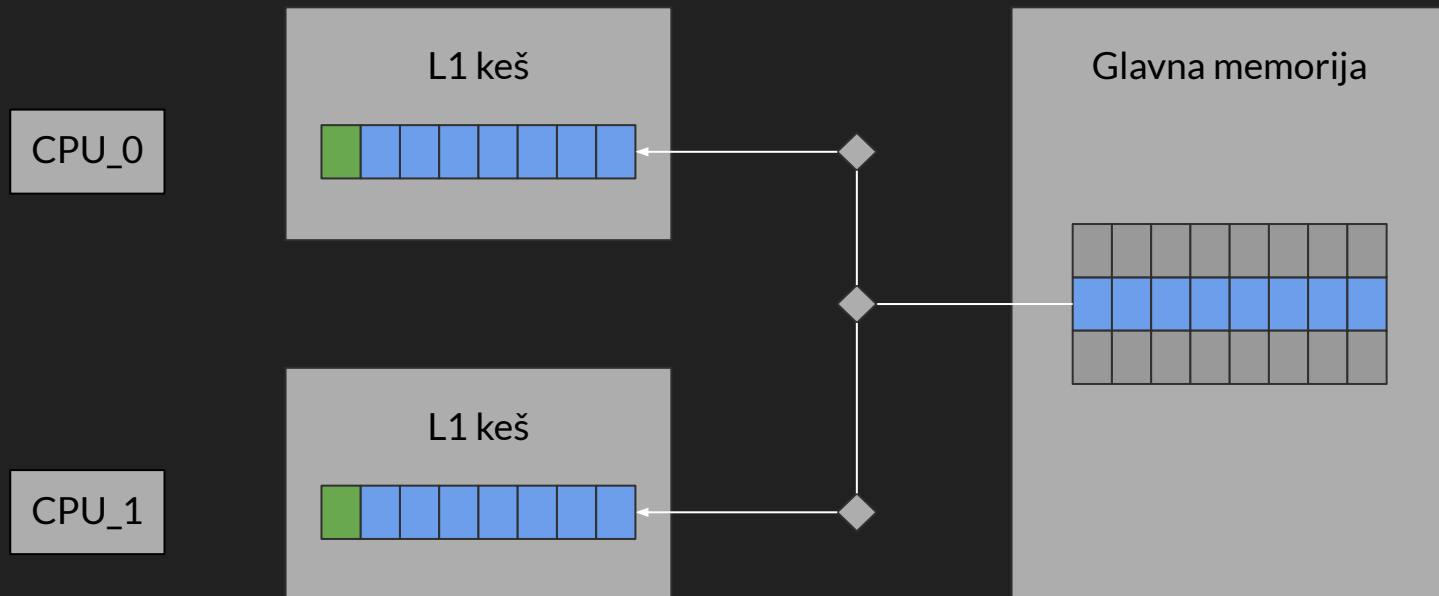
Problem sa rešenjem zadatka 1.1 — lažno deljenje



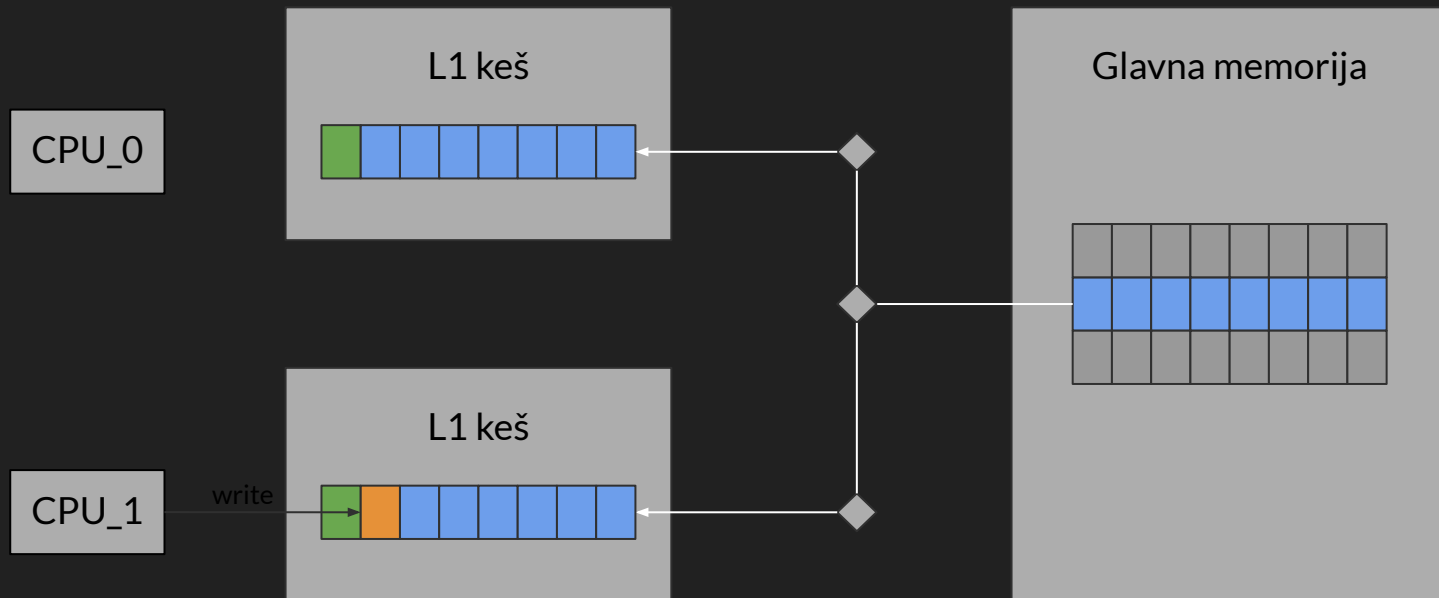
Problem sa rešenjem zadatka 1.1 — lažno deljenje



Problem sa rešenjem zadatka 1.1 — lažno deljenje



Problem sa rešenjem zadatka 1.1 — lažno deljenje



Zadatak 1.2 — Računanje vrednosti broja π

- Modifikovati rešenje prethodnog zadatka tako da se ukloni lažno deljenje *padding*-om.
- Primer rešenja: funkcija *parallel_code_no_false_sharing*, datoteka *resenja/01_omp_pi.c*.

Zadatak 1.2 — Računanje vrednosti broja π

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]

keš bez *padding*-a

A[0]							
A[1]							
A[2]							
...							

keš sa *padding*-om

Eksplicitna sinhronizacija

Konstrukcije za eksplicitnu sinhronizaciju

- Barijera — tačka u kodu do koje moraju stići sve niti pre nego što se nastavi izvršavanje.

```
...  
#pragma omp barrier  
...
```

Konstrukcije za eksplicitnu sinhronizaciju

- Kritična sekcija:
 - označava kritičnu sekciju koda u kojoj se u trenutku nalazi samo jedna nit
 - nakon što nit izađe iz kritične sekcije, u nju ulazi neka od preostalih

```
...  
#pragma omp critical  
{  
    Blok koda  
}
```

Konstrukcije za eksplicitnu sinhronizaciju

- Atomične operacije:
 - operacija se izvršava na atomičan, hardverski implementiran način, ukoliko postoji podrška u hardveru
 - ako nema hardverske podrške, izvršiće se kao da je u pitanju *critical*

...

```
#pragma omp atomic  
operacija
```

Zadatak 1.3 — Računanje vrednosti broja π

- Modifikovati rešenje zadatka 1.1 tako da se lažno deljenje eliminiše sinhronizacijom.
- Primer rešenja: funkcija *parallel_code_synchronization*, datoteka *resenja/01_omp_pi.c*.

Paralelna *for* petlja

Konstrukcije za podelu posla (engl. *worksharing constructs*) — *Loop konstrukcija*

```
#pragma omp for [klauzule]  
for-petlja
```

- Problemi u računarstvu visokih performansi se često svode na iterativnu obradu velikih nizova.
- Petlje sa nezavisnim iteracijama su pogodne za paralelizaciju; svaka nit može dobiti parče niza za obradu.
- Ovakva obrada je već implementirana u zadatku 1.1, ali je podela iteracija vršena ručno.

Loop konstrukcija

```
#pragma omp for [klauzule]  
for-petlja
```

- Klauzule:
 - schedule, collapse, private, shared, reduction, nowait...
 - više detalja o posebnim klauzulama u nastavku...

Loop konstrukcija — primer

- Primer upotrebe *loop* konstrukcije.

```
#pragma omp parallel
{
    int sum = 0;
    #pragma omp for
    for (int i = 0; i < N; i++)
        sum += A[i];
}
```

Loop konstrukcija — primer

- Ukoliko čitav kôd treba biti u petlji, moguće je spojiti *parallel* i *for*.

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
{
    int sum = 0;
    sum += A[i];
}
```

- Odmah nakon *parallel for* mora da sledi petlja.

Loop konstrukcija — raspoređivanje

- Strategije raspoređivanja mogu se specificirati *schedule* klauzulom.
- Raspoređivanje se odnosi na način na koji će iteracije biti podeljene nitima.

```
#pragma omp for schedule (tip[, veličina bloka])
```

- Veličina bloka meri se u broju iteracija koje čine blok.

Loop konstrukcija — raspoređivanje

- Strategije raspoređivanja:
 - *static* — blokovi iteracija se dodeljuju nitima u vreme kompajliranja po *round-robin* principu
 - *dynamic* — blokovi iteracija se dodeljuju nitima u vreme izvršavanja tako da izvršavanje bude što je bolje moguće, odnosno da niti retko budu nezaposlene
 - *guided* — modifikacija dinamičkog gde je svaki naredni blok iteracija manji od prethodnog
 - *auto* — kompajler određuje koja strategija će se iskoristiti
 - *runtime* — strategija se bira na osnovu *OMP_SCHEDULE* promenljive okruženja

Redukcija

Loop konstrukcija — redukcija

- U već viđenom primeru sa *loop* konstrukcijom, svaka nit računa svoju parcijalnu sumu.

```
#pragma omp parallel
{
    int sum = 0;
    #pragma omp for
    for (int i = 0; i < N; i++)
        sum += A[i];
}
```

- Sada treba sabrati sve parcijalne sume u jednu, konačnu.

Loop konstrukcija — redukcija

- Može da se izvede ovako...

```
int sum = 0;
#pragma omp parallel for
for (int i = 0; i < N; i++)
{
    #pragma omp critical
    sum += A[i];
}
```

- ... ali je ovo vrlo neefikasno.

Loop konstrukcija — redukcija

- Bolji način — korišćenjem redukcije.

```
#pragma omp [parallel] for reduction (redukcioni_operator:lista_promenljivih)
```

- Lista promenljivih se sastoji od **deljenih promenljivih** prethodno **deklarisanih van paralelnog regiona**.
- Svaka nit dobija svoju privatnu kopiju promenljivih navedenih u listi, inicijalizovanu na neutralni element za redukcioni operator (0 za +, 1 za *...).
- Niti u svojim iteracijama rade sa privatnim kopijama promenljivih.
- Na kraju petlje, konačna vrednost jedne promenljive se dobija primenom redukcionog operatora nad **privatnim kopijama** promenljive i nad **vrednošću koju je promenljiva imala pre ulaska u paralelni region**.

Loop konstrukcija — redukcija


- Kombinovanje parcijalnih suma u našem primeru može se izvršiti na sledeći način:

```
int sum = 0;
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < N; i++)
{
    sum += A[i];
}
```

Loop konstrukcija — redukcija

- Kombinovanje parcijalnih suma u našem primeru može se izvršiti na sledeći način:

```
int sum = 0;
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < N; i++)
{
    sum += A[i];
}
```

 Ovde se koriste privatne kopije

Loop konstrukcija — redukcija

- Kombinovanje parcijalnih suma u našem primeru može se izvršiti na sledeći način:

```
int sum = 0;
```

Ova vrednost će se na kraju sabrati sa privatnim kopijama. Ne mora biti 0.

```
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < N; i++)
{
    sum += A[i];
}
```

Loop konstrukcija — redukcija

- Redukcioni operatori za C/C++:

Operator	Inicijalna vrednost
+	0
*	1
-	0
&	~0
	0
^	0
&&	1
	0
max	Najmanji negativan broj
min	Najveći pozitivan broj

Zadatak 1.4 — Računanje vrednosti broja π

- Implementirati računanje vrednosti broja π korišćenjem redukcije.
- Primer rešenja: funkcija *parallel_code_for_construct*, datoteka *resenja/01_omp_pi.c*.

Implicitna i eksplicitna barijera

- Eksplicitna barijera, kao što je već viđeno, postavlja se pomoću *omp barrier* konstrukcije.
- Implicitna barijera dolazi zajedno sa nekom drugom konstrukcijom.
- Na primer, implicitna barijera postoji na kraju petlje u *loop* konstrukciji.
- Implicitna barijera može se eksplicitno ukloniti pomoću *nowait* klauzule.

Implicitna barijera

Implicitna barijera

```
#pragma omp parallel
{
    #pragma omp for
    for(int i = 0; i < N; i++)
    {
        // prvi blok naredbi
    }
    //drugi blok naredbi
}
```

- Postoji implicitna barijera na kraju prvog bloka naredbi.
- Dok sve niti ne završe svoje iteracije, nijedna nit neće započeti drugi blok naredbi.

Implicitna barijera

```
#pragma omp parallel
{
    #pragma omp for nowait
    for(int i = 0; i < N; i++)
    {
        // prvi blok naredbi
    }
    //drugi blok naredbi
}
```

- Klauzula *nowait* uklanja implicitnu barijeru.
- Sada niti mogu nastaviti sa drugim blokom i pre nego što sve završe sa prvim.

Implicitna barijera

- Implicitna barijera postoji i na samom kraju paralelnog regiona, kao deo *omp parallel* konstrukcije.
- Ovu implicitnu barijeru nije moguće ukloniti *nowait* klauzulom.

Sections/section konstrukcija

Konstrukcije za podelu posla — *sections/section* konstrukcija

- Svaku sekciju izvršiće samo jedna nit, ali jedna nit može izvršiti više sekcija.
- Prva sekcija ne zahteva *omp section* direktivu; naredne sekcije, ukoliko ih ima, zahtevaju.
- Na kraju *omp sections* konstrukcije postoji implicitna barijera i može se ukloniti *nowait* klauzulom.

```
#pragma omp sections [klauzule]
{
    [#pragma omp section]
    Blok koda
    [#pragma omp section]
    Blok koda]
    ...
}
```

Sections/section konstrukcija — primer

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        x_calculation();

        #pragma omp section
        y_calculation();

        #pragma omp section
        z_calculation();
    }
}
```

Sections/section konstrukcija — primer

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section ← nije neophodno
        x_calculation();

        #pragma omp section
        y_calculation();

        #pragma omp section
        z_calculation();
    }
}
```

Single konstrukcija

Konstrukcije za podelu posla — *single* konstrukcija

- Označava blok koda koji treba da izvrši samo jedna, i to bilo koja nit.
- Na kraju konstrukcije postoji implicitna barijera i može se ukloniti *nowait* klauzulom.

```
#pragma omp single [klauzule]
```

Blok koda

Single konstrukcija — primer

```
#pragma omp parallel
{
    #pragma omp single
    {
        // prvi blok naredbi
    }
    // drugi blok naredbi
}
```

Single konstrukcija — primer

```
#pragma omp parallel
{
    #pragma omp single
    {
        // prvi blok naredbi
    }
    // -----
    // drugi blok naredbi
}
```

implicitna barijera

Master konstrukcija

Master konstrukcija

- Slično kao *single*, uz sledeće razlike:
 - blok koda izvršava *isključivo* glavna nit (nit sa indeksom 0),
 - nema implicitne sinhronizacije.

```
#pragma omp master  
Blok koda
```

Šta bi trebalo da sada znamo?

- Kada se koristi *OpenMP* i po kom modelu radi.
- Kako se kreira paralen region i kako se podešava broj niti u njemu.
- Koje niti vide koje promenljive.
- Kako izbeći lažno deljenje.
- Kako kreirati paralelnu petlju i kako izvršiti redukciju u njoj.
- Kako kreirati sekcije sa proizvoljnim zadacima.
- Kako dodeliti posao samo jednoj niti.

Zadatak 2 — parcijalna redukcija

- Dat je kôd koji na specifičan način implementira parcijalnu redukciju — računaju se sume podnizova čitavog niza.
- U datoteci *zadaci/02_omp_parallel_recution.c* nalaze se sekvencijalna i parcijalna implementacija redukcije.
- Potrebno je:
 - rastumačiti na koji način se računaju parcijalne sume,
 - obratiti pažnju na vreme izvršenja datih rešenja (koristi se posebna *omp_get_wtime* funkcija),
 - videti kako promena parametra N utiče na vreme izvršavanja (N mora biti deljivo sa *NUM_THREADS*), i
 - ustanoviti problem sa paralelnim rešenjem i ukloniti isti.
- Datoteka *resenja/02_omp_parallel_reduction.c* sadrži rešenje zadatka.