

Računarski sistemi visokih performansi

Nikola Vukić, Petar Trifunović, Veljko Petrović

Računarske vežbe
Zimski semestar 2024/25.

OpenMP 2

Sadržaj

- Sinhronizacija zaključavanjem.
- Klauzule za rad sa podacima.
- Paralelizacija nepoznatog broja iteracija.
- *Task* konstrukcija + konstrukcije za kreiranje i sinhronizaciju zadataka.
- Zadaci.

Lock sinhronizaciori mehanizam

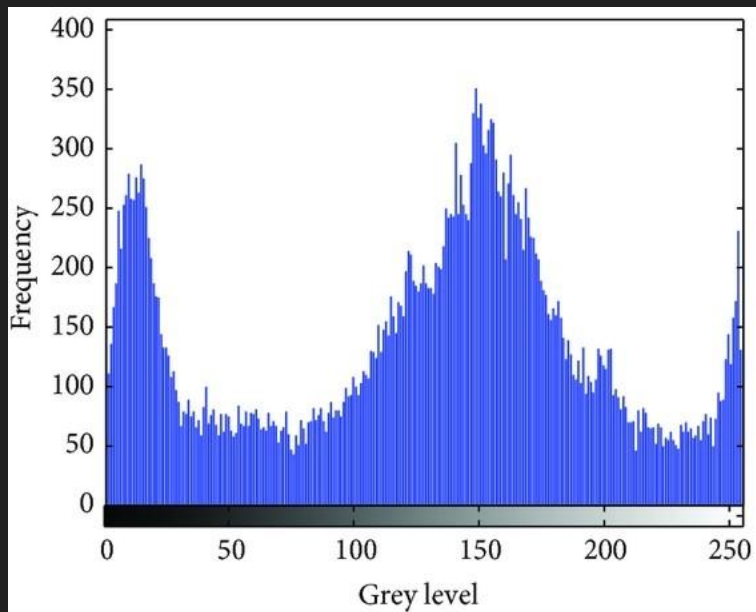
Lock sinhronizacioni mehanizam

- Mehanizam sinhronizacije niskog nivoa.
- Implementira se funkcijama koje se pozivaju nad promenljivama tipa *omp_lock_t*.
- Spisak funkcija za implementaciju zaključavanja:
 - *void omp_init_lock(omp_lock_t* lock)* — inicijalizuje *omp_lock_t* promenljivu
 - *void omp_set_lock(omp_lock_t* lock)* — čeka dok prosleđena promenljiva ne bude slobodna i nakon toga je zaključava
 - *void omp_unset_lock(omp_lock_t* lock)* — otključava prosleđenu promenljivu
 - *void omp_destroy_lock(omp_lock_t* lock)* — uništava *omp_lock_t* promenljivu

Lock sinhronizacioni mehanizam

- *Critical* direktiva u pozadini koristi ovaj mehanizam.
- Zašto nam je onda ikada potreban ovaj mehanizam direktno?
- Ukoliko se obezbeđuje isključiv pristup jednoj promenljivoj, *critical* je dovoljno.
- Šta ako nam je potreban isključiv pristup pojedinačnim elementima niza?

Histogram — primer *lock* mehanizma



```
int histogram[255];
```

Histogram — primer *lock* mehanizma

```
#define NBUCKETS 255
```

broj mogućih različitih vrednosti u nizu
(veličina histograma)

```
#define NVALS 1 << 19
```

veličina niza

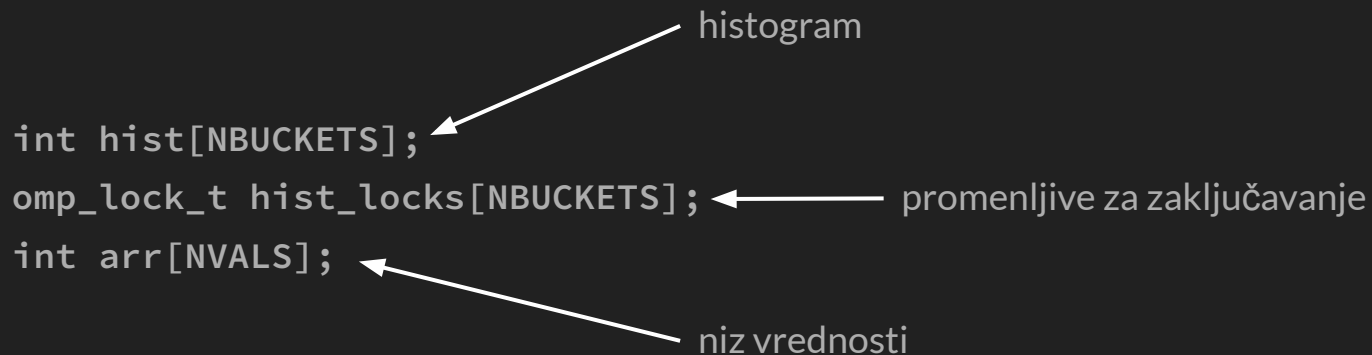
Histogram — primer *lock* mehanizma

```
int hist[NBUCKETS];  
omp_lock_t hist_locks[NBUCKETS];  
int arr[NVALS];
```

histogram

promenljive za zaključavanje

niz vrednosti

The diagram consists of three lines of C code. The first line is 'int hist[NBUCKETS];'. An arrow points from the label 'histogram' to this line. The second line is 'omp_lock_t hist_locks[NBUCKETS];'. An arrow points from the label 'promenljive za zaključavanje' to this line. The third line is 'int arr[NVALS];'. An arrow points from the label 'niz vrednosti' to this line.

Histogram — primer *lock* mehanizma

- Inicijalizacija histograma i promenljivih za zaključavanje.

```
#pragma omp parallel for
for (int i = 0; i < NBUCKETS; i++)
{
    hist[i] = 0;
    omp_init_lock(&hist_locks[i]);
}
```

Histogram — primer *lock* mehanizma

- Isključiv pristup elementima histograma pri ažuriranju.

```
#pragma omp parallel for
for (int i = 0; i < NVALS; i++)
{
    int val = arr[i];

    omp_set_lock(&hist_locks[val]);
    hist[val]++;
    omp_unset_lock(&hist_locks[val]);
}
```

Histogram — primer *lock* mehanizma

- Isključiv pristup elementima histograma pri ažuriranju.

```
#pragma omp parallel for  
for (int i = 0; i < NVALS; i++)  
{  
    int val = arr[i];
```

```
    omp_set_lock(&hist_locks[val]);  
    hist[val]++;  
    omp_unset_lock(&hist_locks[val]);
```

```
}
```

← kritična sekcija

Histogram — primer *lock* mehanizma

- Isključiv pristup elementima histograma pri ažuriranju.

```
#pragma omp parallel for
for (int i = 0; i < NVALS; i++)
{
    int val = arr[i];

    omp_set_lock(&hist_locks[val]);
    hist[val]++;
    omp_unset_lock(&hist_locks[val]);
}
```

ako više niti ima istu vrednosti *val*
promenljive, samo jedna će
zaključati element histograma i
proći do sledeće naredbe

Histogram — primer *lock* mehanizma

- Isključiv pristup elementima histograma pri ažuriranju.

```
#pragma omp parallel for  
for (int i = 0; i < NVALS; i++)  
{
```

```
    int val = arr[i];
```

```
    omp_set_lock(&hist_locks[val]);
```

```
    hist[val]++;
```

```
    omp_unset_lock(&hist_locks[val]);
```

```
}
```

ostale niti će čekati da
se vrednost otključa


Histogram — primer *lock* mehanizma

- Isključiv pristup elementima histograma pri ažuriranju.

```
#pragma omp parallel for
for (int i = 0; i < NVALS; i++)
{
    int val = arr[i];

    omp_set_lock(&hist_locks[val]);
    hist[val]++;
    omp_unset_lock(&hist_locks[val]);
}
```

nakon što nit otključa promenljivu,
neka druga nit će moći da uđe u
kritičnu sekciju



Histogram — primer *lock* mehanizma

- Uništavanje promenljivih za zaključavanje.

```
#pragma omp parallel for
for (int i = 0; i < NBUCKETS; i++)
{
    omp_destroy_lock(&hist_locks[i]);
}
```


Klauzule za rad sa podacima

Klauzule za rad sa podacima — *shared*

- Eksplicitno navodi koje promenljive su deljene.
- U C/C++ implementaciji *OpenMP* standarda, ovo je podrazumevano ponašanje za promenljive deklarisanе pre ulaska u blok *omp* konstrukcije.
- Može biti deo mnogih konstrukcija, ali za nas je najvažnije da se može dodati uz *parallel* konstrukciju.

Klauzule za rad sa podacima — *private*

- Navodi koje promenljive deklarisanе van bloka *omp* konstrukcije su privatne unutar bloka.
- Inicijalna vrednost navedenih promenljivih je nedefinisana unutar bloka, makar bila i eksplicitno postavljena pre njega.
- Po završetku bloka konstrukcije, privatne promenljive nestaju, a promenljiva ima vrednost koju je imala pre ulaska u blok.
- Može stajati uz većinu već viđenih konstrukcija (*parallel, loop, single, sections*).

Klauzula *private* — primer

```
int x = 5;
#pragma omp parallel num_threads(4) private(x)
{
    // vrednost x je ovde nedefinisana
    printf("Nit %d: %d\n", omp_get_thread_num(), x);
}
printf("Vrednost nakon paralelnog bloka: %d\n", x);
```

- Primer izvršavanja:

```
Nit 0: 32710
Nit 3: 0
Nit 2: 0
Nit 1: 0
Vrednost nakon paralelnog bloka: 5
```

Klauzule za rad sa podacima — *firstprivate* i *lastprivate*

- Proširenja u odnosu na *private* klauzulu.
- Klauzula *firstprivate* obezbeđuje da inicijalna vrednost privatnih promenljivih bude jednaka vrednosti koju je promenljiva imala pre ulaska u blok konstrukcije; može stajati uz *parallel*, *loop*, *sections*, *single*.
- Klauzula *lastprivate* obezbeđuje da se, po izlasku iz bloka, sačuva vrednost iz logički poslednje iteracije petlje (ako se navede uz *loop* konstrukciju), ili vrednost iz sekcije koja se poslednja javlja u kodu (ako se navede uz *sections*).
- Ovo su jedine dve klauzule u kojima se jedna promenljiva može istovremeno naći u jednoj istoj konstrukciji.

Klauzula *firstprivate* — primer

```
int x = 5;
#pragma omp parallel num_threads(4) firstprivate(x)
{
    // x će zadržati vrednost od pre paralelnog regiona
    printf("Nit %d: %d\n", omp_get_thread_num(), x);
}
```

- Primer izvršavanja:

```
Nit 3: 5
Nit 0: 5
Nit 1: 5
Nit 2: 5
```

Klauzula *lastprivate* — primer

```
int x = 5;
#pragma omp parallel for num_threads(4) lastprivate(x)
for (int i = 0; i < 4; i++)
{
    x = i;
}
printf("x = %d\n", x);
```

- Primer izvršavanja:

x = 3

Kombinacija *firstprivate* i *lastprivate* — primer

```
int x = 5;
#pragma omp parallel for num_threads(4) firstprivate(x) lastprivate(x)
for (int i = 0; i < 4; i++)
{
    x += i;
}
printf("x = %d\n", x);
```

- Primer izvršavanja:

x = 8

- Bez *firstprivate*, rezultat bi bio nedefinisan.

Zadatak 3

- Datoteka *zadaci/03_omp_mandelbrot.c* u direktorijumu *zadaci* sadrži paralelno *OpenMP* rešenje za određivanje Mandelbrotovog skupa.
- Postoje problemi u rešenju vezani za:
 - štetno preplitanje, i
 - korišćenje klauzula za rad sa podacima.
- Potrebno je pronaći probleme i ukloniti ih.
- Ispravljeno rešenje može se naći u datoteci *resenja/03_omp_mandelbrot.c*.

Paralelizacija nepoznatog broja iteracija

Paralelizacija nepoznatog broja iteracija

- *OpenMP* je zamišljen za paralelizaciju petlji kod kojih se unapred zna broj iteracija.
- Problem:
 - Kako paralelizovati *while* petlju?
 - Kako paralelizovati rekurziju?

Paralelizacija nepoznatog broja iteracija

- *OpenMP* je zamišljen za paralelizaciju petlji kod kojih se unapred zna broj iteracija.
- Problem:
 - Kako paralelizovati *while* petlju?
 - Kako paralelizovati rekurziju?
- Rešenja:
 - Transformisati problem u *for* petlju ako je moguće.
 - Upotrebiti *task* konstrukciju (uvedena u *OpenMP* 3.0).

Paralelizacija nepoznatog broja iteracija — primer

- Potrebno je paralelizovati sledeći kôd za obradu čvorova liste:

```
while (p != NULL)
{
    processwork(p);
    p = p->next;
}
```

Paralelizacija nepoznatog broja iteracija — primer

- Prebrojati elemente u listi:

```
int nelems = 0;
while (p != NULL)
{
    nelems++;
    p = p->next;
}
```

Paralelizacija nepoznatog broja iteracija — primer

- Sačuvati sve pokazivače:

```
p = head;
for (int i = 0; i < nelems; i++)
{
    ptrs[i] = p;
    p = p->next;
}
```

Paralelizacija nepoznatog broja iteracija — primer

- Paralelizovati obradu:

```
#pragma omp parallel for
for (int i = 0; i < nelems; i++)
{
    processwork(ptrs[i]);
}
```


Task konstrukcija

Task konstrukcija

- Nezavisna jedinica posla.
- Čine je:
 - kôd koji zadatak izvršava,
 - podaci (privatne i deljene promenljive), i
 - *Internal Control Variables* (ICV) (indikator da li zadatak može da bude dodeljen različitim nitima, vrsta raspoređivanja, broj niti u paralelnom regionu...).

```
#pragma omp task [klauzule]
```

Blok koda

Task konstrukcija — kombinacija sa single

- *task* i *single* konstrukcije se često koriste zajedno.
- Jedna nit pravi zadatke koji se smeštaju u red zadataka odakle su dostupni svim nitima.
- Niti, po internom mehanizmu raspoređivanja, *uzimaju zadatke iz reda* i izvršavaju ih.
- Moguće je da jedna nit izvrši više zadataka, ali će svaki zadatak biti izvršen samo jednom.

Task konstrukcija — kombinacija sa single

```
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        foo();

        #pragma omp task
        bar();
    }
}
```

- Bez *single*, svaka nit bi *napravila* po dva zadatka.
- Sve niti koje su u paralelnom regionu ih *izvršavaju*.

Task konstrukcija — kombinacija sa single

```
5 void foo() {
6     printf("Thread: %d - picked up `Foo` from task queue.\n",
7           omp_get_thread_num());
8 }
9
10 void bar() {
11     printf("Thread: %d - picked `Bar` from task queue.\n",
12           omp_get_thread_num());
13 }
14
15 int main(void) {
16
17     #pragma omp parallel
18     {
19         #pragma omp single
20         {
21             int32_t id = omp_get_thread_num();
22             for (size_t i = 0; i < 4; i++) {
23                 printf("Thread: %d, adding `Foo` to task queue... \n", id);
24             #pragma omp task
25             {
26                 foo();
27             }
28
29             for (size_t i = 0; i < 4; i++) {
30                 printf("Thread: %d, adding `Bar` to task queue... \n", id);
31             #pragma omp task
32             {
33                 bar();
34             }
35         }
36     }
37 }
```

- Primer ispisa:

```
Thread: 20, adding `Foo` to task queue...
Thread: 20, adding `Foo` to task queue...
Thread: 18 - picked up `Foo` from task queue.
Thread: 20, adding `Foo` to task queue...
Thread: 20, adding `Bar` to task queue...
Thread: 20, adding `Bar` to task queue...
Thread: 20, adding `Bar` to task queue...
Thread: 20, adding `Bar` to task queue...
Thread: 22 - picked `Bar` from task queue.
Thread: 11 - picked `Bar` from task queue.
Thread: 14 - picked up `Foo` from task queue.
Thread: 18 - picked up `Foo` from task queue.
Thread: 4 - picked `Bar` from task queue.
Thread: 0 - picked `Bar` from task queue.
Thread: 7 - picked up `Foo` from task queue.
```

Task konstrukcija — razlika u odnosu na *sections*

- *Task* konstrukcija nema implicitnu sinhronizaciju (u našem primeru, sinhronizaciju uvodi *single*).
- *Task* konstrukcija je pogodna kada je potreban dinamički broj zadataka.
- Skuplja, zbog reda zadataka, dinamičke alokacije resursa, upravljanja zavisnostima, imaju veći *overhead*.
- Potrebni kompleksni, nezavisni, i veliki problemi da bi bilo vremenski isplativo (*coarse-grained parallelism*).
- Postoji implicitna sinhronizacija, može se ukloniti sa *nowait* klauzulom
- *Sections* konstrukcija je pogodna kada je broj potrebnih sekcija poznat u vremenu kompajliranja, odnosno statičan.
- Kako se sekcije podele na početku paralelnog regiona, nije potreban red zadataka.

Zadatak 4 — modifikacija liste

- Data je sekvencijalna implementacija liste (datoteka *zadaci/04_omp_linked_list.c*) u kojoj svaki element sadrži po jedan Fibonačijev broj dobijen funkcijom *processwork*.
- Napraviti paralelni *OpenMP* program korišćenjem *task* konstrukcije.
- Primer rešenja: datoteka *resenja/04_omp_linked_list.c*.

Task konstrukcija — sinhronizacija

- *taskwait* — sinhronizuje samo zadatke istog nivoa

```
#pragma omp taskwait
```

- *taskgroup* — sinhronizuje i podzadatke

```
#pragma omp taskgroup
```

Blok koda

Task konstrukcija — *taskwait* sinhronizacija

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp task
    {
        printf("Task 1\n");

        #pragma omp task
        {
            printf("Task 2\n");
        }
    }

    #pragma omp taskwait

    #pragma omp task
    {
        printf("Task 3\n");
    }
}
```

- Zadatak 3 će se sigurno izvršiti nakon zadatka 1, jer *taskwait* obezbeđuje sinhronizaciju između njih.
- Zadatak 2 će se **možda** izvršiti pre zadatka 3, ali za to ne postoji garancija, jer *taskwait* ne sinhronizuje zadatke u podnivou.

Task konstrukcija — *taskgroup* sinhronizacija

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp taskgroup
    {
        #pragma omp task
        {
            printf("Task 1\n");

            #pragma omp task
            {
                sleep(1);
                printf("Task 2\n");
            }
        }
    }

    #pragma omp task
    {
        printf("Task 3\n");
    }
}
```

- Svi zadaci iz *taskgroup* grupe moraju okončati svoje izvršavanje pre nego što se izvrši naredni zadatak.
- Ovime se garantuje da će se i zadatak 1 i zadatak 2 izvršiti pre zadatka 3.

Zadatak 5 — Fibonačijev niz

- U datoteci *zadaci/05_omp_fibonacci.c* data je sekvencijalna implementacija rekurzivnog algoritma koji računa n -ti element Fibonačijevog niza.
- Paralelizovati implementaciju korišćenjem *task* konstrukcija i sinhronizacija.
- Uporediti vreme izvršavanja paralelne i sekvencijalne implementacije.
- Primer rešenja: datoteka *resenja/05_omp_fibonacci.c*.

Dodatni zadaci

Zadatak 6 — Traženje korena funkcije

- U direktorijumu *zadaci/06_omp_bisection*, datoteka *main.c*, data je sekvencijalna implementacija metode za određivanje korena funkcije nad zadatim intervalom metodom bisekcije.
- Ispratiti uputstvo za pokretanje iz datoteke *README.md*.
- Implementirati paralelno rešenje.
- Uporediti vreme izvršavanja paralelne i sekvencijalne implementacije.
- Primer rešenja: direktorijum *resenja/06_omp_bisection*, datoteka *main.c*.

Zadatak 7 — Genetski algoritam

- U direktorijumu *zadaci/07_omp_genetic_algorithm*, datoteka *main.c*, data je sekvencijalna implementacija jednostavnog genetskog algoritma.
- Ispratiti uputstvo za pokretanje iz datoteke *README.md*.
- Nakon izvršenja programa biće ispisano koji deo algoritma troši koji procenat ukupnog vremena izvršenja.
- Obratiti pažnju koji delovi algoritma su vremenski najzahtevniji, razmotriti mogućnosti paralelizacije, i paralelizovati delove gde za to postoji prostor.
- Obratiti pažnju kako promena broja iteracija i jedinki utiče na rezultat
- Primer rešenja: direktorijum *resenja/07_omp_genetic_algorithm*, datoteka *main.c*.
- Zakomentarisati neke od uvedenih paralelizacija i obratiti pažnju na njihov uticaj na vreme izvršavanja.

Zadatak 8 — Množenje matrica

- U direktorijumu *zadaci/08_omp_matrix_mul*, datoteka *main.c*, dat je kostur zadatka za množenje matrica.
- Ispratiti uputstvo za pokretanje iz datoteke *README.md*.
- Implementirati sekvencijalno rešenje (funkcija *matrix_multiply*).
- Implementirati paralelno rešenje (funkcija *matrix_multiply_openmp*).
- Uporediti vreme izvršavanja paralelne i sekvencijalne implementacije.
- Primer rešenja: direktorijum *resenja/08_omp_matrix_mul*, datoteka *main.c*.
- Videti uputstvo za generisanje matrica i proveru rešenja na **sledećem slajdu**.

Zadatak 8 — Množenje matrica

- Kao pomoć za generisanje matrica i proveru rešenja treba koristiti *python* skripte za rad sa *hdf5* formatom podataka.
- Skripte se nalaze na *acs-u*, u repozitorijumu predmeta, na putanji *vezbe/hdf5utils*.
- Obe skripte sadrže uputstvo za upotrebu u vidu komentara.
- Za rad sa ovim skriptama najbolje je:
 - kreirati i aktivirati [python virtuelno okruženje](#),
 - instalirati *h5py* biblioteku komandom *pip install h5py*.

Zadatak 9 — Transponovanje matrice

- U direktorijumu *zadaci/09_omp_matrix_transp*, datoteka *main.c*, data je sekvencijalna implementacija transponovanja matrice (funkcija *transpose*).
- Implementirati paralelno transponovanje matrica (funkcija *transpose_openmp*).
- Da li bi bilo moguće transponovati matricu na isti način, ukoliko se podaci ne prepisuju u novu matricu, već se samo menja stara (**mat[i, j]** postaje **mat[j, i]**)?
- Videti uputstvo za generisanje matrica i proveru rešenja na **sledećem slajdu**.

Zadatak 9 — Transponovanje matrice

- Za generisanje matrica može se iskoristiti skripta kao i za zadatak sa množenjem matrica.
- Za proveru rešenja koristiti skriptu *hdf5_matrix_transpose_comparator.py*, koja sadrži uputstvo za korišćenje u vidu komentara.