

# OpenMP

---

- \* Odredbe (klauzule)
- \* Direktive za sinhronizaciju

# Odredbe (klauzule)

\* To opcije kojima se na jednostavn i moćan način može upravljati ponašanjem direktive na koju su primenjene.

- Shared
- private
- lastprivate
- firstprivate
- default
- nowait
- schedule
- if
- num\_threads
- ordered
- reduction
- copyin
- copyprivate

# Odredba `shared`

- \* Koristi se da definiše koje će promenljive biti zajedničke (deljive) za tim niti u paralelnom regionu.
  - To ukazuje da postoji samo jedna jedinstvena instanca promenljive i da je svaka nit može čitati ili modifikovati.
- \* Sintaksa
  - *shared*(lista\_promenljivih)
    - Sve promenljive navedene u listi će biti deljive (zajedničke)

# Primer

```
#pragma omp parallel for shared(a)
for (i=0; i<n; i++)
{
    a[i] += i;
} /*-- End of parallel for --*/
```

- Sve niti mogu da čitaju i upisuju u vektor **a**.
- Unutar paralelne petlje svaka nit će pristupiti postojećoj vrednosti elementa **a[i]** i izračunati novu vrednost.
- Nakon okončanja paralelnog regiona sve nove vrednosti elemenata vektora **a** će biti zapamćene u glavnoj memoriji gde im master nit može pristupiti.

\* Kao posledica deljenja pristupa istoj promenljivoj, može se desiti da se ažurira ista memorijska lokacija ili da jedna nit pokušava da pročita sadržaj lokacije dok druga pokušava da upiše u nju.

- Zbog toga je potrebno voditi računa da ne dođe do ovakvih situacija i da se pristup deljivim promenljivim odvija kako se zahteva algoritmom.

# Primer: nedeterminisanost zbog trke

```
#pragma omp parallel for shared(a)
```

```
{
```

```
    for (i=0, i<=omp_get_num_threads(), i++)
```

```
        a+=omp_get_thread_num();
```

```
}
```

```
printf("a=%d",a)
```

- Nakon okončanja paralelnog regiona vrednost promenljive a je nedeterminisana!
  - Neophodno je sinhronizovati pristup deljivim promenljivim !

# Odredba private

\* Omogućava da se definiše koje će promenljive biti privatne za svaku nit u timu

\* Sintaksa

- *private* (lista\_promenljivih)

- Svaka promenljiva u listi se replicira tako da svaka nit u timu ima ekskluzivno pravo pristupa svojoj kopiji promenljive.
- Promene koje učini jedna nit nisu vidljive drugoj niti.
- U OpenMP, promenljive koje su deklarisanе lokalno unutar strukturnog bloka ili rutine (funkcije) koja se poziva u okviru paralelnog regiona, su privatne po definiciji (by default)
- Po definiciji, indeksna promenljiva u *for* petlji je privatna za svaku nit, tj. svaka nit ima svoju kopiju indeksne promenljive
- Vrednosti privatnih promenljivih nisu definisane na ulasku i izlasku iz paralelnog regiona, čak i ako je promenljiva bila definisana pre ulaska u paralelni region.

# Primer

```
#pragma omp parallel for private(i,a)
for (i=0; i<n; i++)
{
    a = i+1;
    printf("Thread %d has a value of a = %d for i = %d\n",
        omp_get_thread_num(),a,i);
} /*-- End of parallel for --*/
```

\* U ovom primeru i iterativna (indeksna) promenljiva *i* i *a* su privatne.

- Da je promenljiva *a* bila deklarirana kao deljiva (shared) više niti bi pokušalo da ažurira istu promenljivu različitim vrednostima na nekontrolisani način.
  - Konačna vrednost bi zavisila od toga koja nit je poslednja upisala vrednost

Mogući izlaz iz gornjeg programa za n=5 i tri niti bi moga biti

```
Thread 0 has a value of a = 1 for i = 0
Thread 0 has a value of a = 2 for i = 1
Thread 2 has a value of a = 5 for i = 4
Thread 1 has a value of a = 3 for i = 2
Thread 1 has a value of a = 4 for i = 3
```

# Odredba *lastprivate*

\* Šta ako je promenljiva *a* iz prethodnog primera potrebna nakon okončanja paralelnog regiona?

- Privatnim promenljivima se ne može pristupiti nakon okončanja paralelnog regiona (petlje ili sekcije).

\* Odredba *lastprivate* (lista\_promenljivih)

- obezbeđuje da vrednost promenljivih navedenih u listi budu definisane nakon izlaska iz paralelnog regiona.
- Poslednja vrednost (*lastprivate*) kod sekvencijalnog izvršenja je nedvosmislena.
  - Međutim, kod paralelnog izvršenja je potrebno objasniti šta znači „poslednja” vrednost.
    - U slučaju da se radi o paralelnoj for petlji, promenljiva će nakon okončanja paralelne petlje imati vrednost koju bi imala nakon poslednje iteracije sekvencijalne petlje.
  - Ako se odredba *lastprivate* koristi sa *section* direktivom onda promenljiva dobija vrednost koju ima na kraju u leksikografski poslednjoj *section* konstrukciji.



# Primer

```
#pragma omp parallel for private(i) lastprivate(a)
  for (i=0; i<n; i++)
  {
    a = i+1;
    printf("Thread %d has a value of a = %d for i = %d\n",
          omp_get_thread_num(),a,i);
  } /*-- End of parallel for --*/

printf("Value of a after parallel for: a = %d\n",a);
```

➤ Izlaz iz programa za n=5 bi mogao da izgleda ovako

```
Thread 0 has a value of a = 1 for i = 0
Thread 0 has a value of a = 2 for i = 1
Thread 2 has a value of a = 5 for i = 4
Thread 1 has a value of a = 3 for i = 2
Thread 1 has a value of a = 4 for i = 3
Value of a after parallel for: a = 5
```

# Odredba firstprivate

\* Privatne promenljive paralelnog regiona nisu inicijalizovane na ulasku u paralelni region.

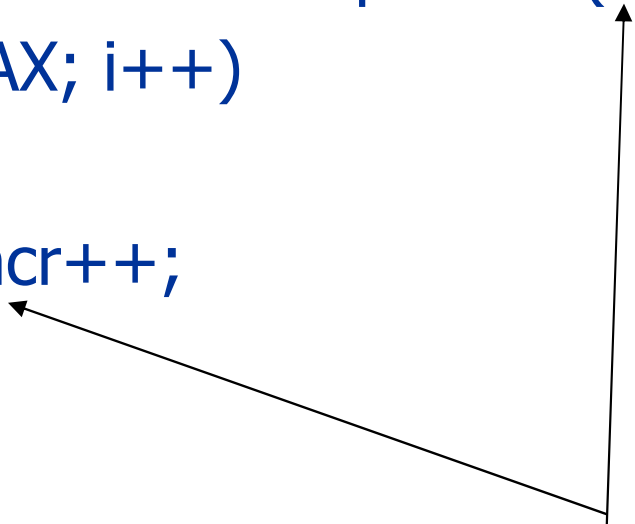
\* Odredbom

- *firstprivate* (lista\_promenljivih)

- se promenljive navedene u listi promenljivih deklariraju kao privatne, ali se inicijalizuju na vrednost promenljive sa istim imenom koja se nalazi van paralelnog regiona.
- Inicijalizaciju obavlja polazna (master) nit pre izvršenja paralelne konstrukcije

# Primer

```
incr = 0;  
#pragma omp parallel for firstprivate (incr)  
for (i = 0; i <= MAX; i++)  
{  
    if ((i%2)==0) incr++;  
    A[i] = incr;  
}
```



Svaka nit dobija svoju  
kopiju promenljive incr  
inicijalizovanu na 0

# Primer

## \* Razlika između private i firstprivate odredbe

$A = 1, B = 1, C = 1$

#pragma omp parallel private(B) firstprivate(C)

- Da li su  $A, B, C$  lokalne za svaku nit ili su deljive unutar paralelnog regiona?
- Koje su inicijalne vrednosti ovih promenljivih na početku, a koje na kraju paralelnog regiona?
- Unutar paralelnog regiona ...
  - "A" je deljiva za sve niti i njena vrednost na ulasku u paralelni region je 1.
  - "B" i "C" su lokalne promenljive za svaku nit.
  - vrednost promenljive B na ulasku u paralelni region nije definisana
  - Inicijalna vrednost promenljive C u paralelnom regionu je 1
- Nakon izlaska iz paralelnog regiona...
  - B i C će se vratiti na vrednosti pre ulaska u paralelni region, tj. imaće vrednost 1
  - A će imati vrednost koju je dobio unutar paralelnog regiona (ili 1 ako joj vrednost nije promenjena u paralelnom regionu)

# Odredba default

- \* Koristi se da dodeli promenljivima podrazumevani atribut deljenja. .
  - Sintaksa u C/C++
    - **default** (none | shared)
      - *default*(private) nije dostupna u C/C++, ali jeste u FORTRANu, i deklariše sve promenljive kao privatne
  - Ova odredba se koristi da definiše atribut deljenja za većinu promenljivih u paralelnoj konstrukciji, a samo izuzetci se eksplicitno navode u okviru drugih odredbi (private, firstprivate ili lastprivate).
  - Npr

```
#pragma omp for default (shared) private (a,b,c)
```
  - deklariše da su sve promenljive deljive izuzev promenljivih a, b i c.
  - Ako se koristi *default*(none) onda programer mora za svaku promenljivu koja se koristi u paralelnoj konstrukciji da navede atribut deljenja.

# Odredba nowait

- \* Ova odredba omogućava programeru da fino podesi performanse.
  - Kod svih direktiva za podelu posla postoji implicitna barijera koja uslovljava da sve niti iz tima moraju da se sinhronizuju na kraju tj. da sačekaju da sve niti okončaju sa izvršenjem paralelne konstrukcije.
  - Ovom odredbom se to pravilo poništava.
    - Ako se ova odredba koristi, kada nit dođe do kraja konstrukcije odmah nastavlja dalje sa izvršenjem, tj. ne čeka se da sve niti okončaju.
- \* **Ipak, barijera na kraju paralelnog *regiona* se ne može poništiti**

```
#pragma omp for nowait
for (i=0; i<n; i++)
{
    .....
}
```

- Kada nit okonča posao u paralelnoj for petlji može da nastavi dalje bez čekanja na ostale niti.

# Primer

✱ Treba biti oprezan sa korišćenjem ove odredbe!

```
#pragma omp parallel shared(n,a,b) private(i)
{
    #pragma omp for
    for (i=0; i<n; i++)
        a[i] = i;

    #pragma omp for
    for (i=0; i<n; i++)
        b[i] = 2 * a[i];
} /*-- End of parallel region --*/
```

✱ ako bi postavili nowait odredbu uz prvu paralelnu for petlju, moglo bi se doći u situaciju da vrednost a[i] nije izračunata a da je potrebna za b[i].

# Odredba schedule

\* Ova odredba se može koristiti **samo sa paralelnom for** direktivom.

- Koristi se da se upravlja načinom raspoređivanja (distribucije) iteracija petlje između niti što može imati značajnog uticaja na performanse programa
- Sintaksa
  - **schedule** (kind [, chunk size]).
- Odredba schedule definiše kako se iteracije petlje dodeljuju nitima u timu.
- Granularnost distribucije je chunk (parče) kontinualnog, nepraznog podskupa iterativnog prostora.
- Parametar *chunk\_size* ne mora biti konstanta;
  - bilo koji celobrojni izraz sa pozitivnom vrednošću može biti iskorišćen za definisanje *chunk\_size*.
- Postoje četiri načina distribucije:
  - *static*, *dynamic*, *guided*, *runtime*



# Distribucija *static*.

## \* Najjednostavnija distribucija je *static*.

- Iteracije se dodeljuju nitima u blokovima veličine `chunk_size`.
- Blokovi se dodeljuju nitima statički kružno (na round robin način) po redosledu brojeva niti.
- Poslednji blok koji se dodeljuje niti može biti manji, tj. imati manji broj iteracija.
- Ako nije definisan parametar `chunk_size`, iterativni prostor se deli na blokove približno jednake veličine.
  - npr ako imamo 200 iteracija i 4 niti, svakoj niti će biti dodeljeno po 50 iteracija petlje (nit 0 iteracije 0-49, nit 1 iteracije 50-99, itd)
- Svakoj niti se dodeljuje bar jedan blok.

# *Distribucija dynamic*

\* Iteracije se dodeljuju nitima po zahtevu niti.

- Nit izvršava blok iteracija čija je veličina definisana vrednošću parametra *chunk\_size*, zatim zahteva sledeći blok sve dok ima blokova koji treba da se obrade.
- Poslednji blok može imati manje iteracija od *chunk\_size*.

\* Ako nije definisan *chunk\_size*, podrazumevana veličina bloka je 1.

\* broj paralelnih niti može da varira od jednog do drugog paralelnog regiona

- broj niti koji se definiše promenljivom okruženja predstavlja gornju granicu, tj. broj niti u paralelnom regionu može biti i manji

# *Distribucija guided.*

\* To je varijacija distribucije dynamic.

- Iteracije se dodeljuju nitima po zahtevu niti.
- Veličina bloka se menja.
  - Parametar *chunk\_size* definiše najmanju veličinu bloka
- Kada okonča izvršenje bloka, nit zahteva sledeći blok, itd.
- Veličina bloka se smanjuje eksponencijalno dok ne dostigne vrednost *chunk\_size*:
  - Prvi blok koji se dodeljuje je veličine  $broj\_iteracija / broj\_niti$ , sledeći je jednak  $broj\_preostalih\_iteracija / broj\_niti$ , itd. dok se ne dostigne veličina bloka definisana sa *chunk\_size*

\* Ako parametar *chunk\_size* nije naveden podrazumeva se vrednost 1.

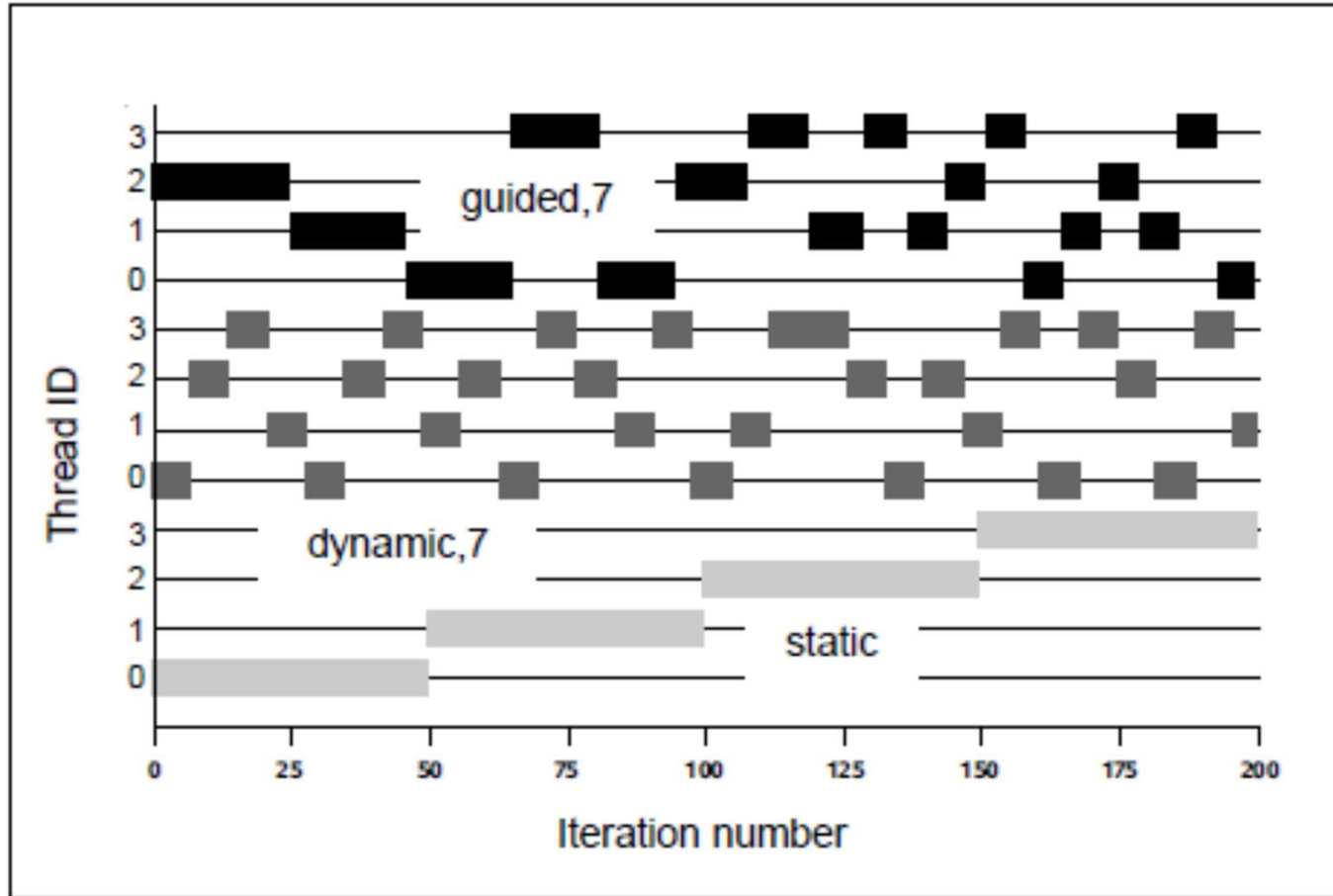
# *Distribucija runtime.*

- Ako je ova distribucija odabrana, odluka o načinu dodeljivanja iteracija petlje nitima donosi se u toku izvršenja programa.
  - Način distribucije kao i veličina bloka (`chunk_size`) se definišu preko `OMP_SCHEDULE` promenljive okruženja (environment variable).
- 

\* Ako nije navedena odredba `schedule`, podrazumevana distribucija je zavsina od implementacije.

- Dva najčešće korišćena načina distribucije iteracija petlje po nitima su statičko i dinamičko.

# Primer distribucije static, dynamic i guided



za static distribuciju nije navedena vrednost `chunk_size` pa je iterativni prostor od 200 iteracija podeljen na 4 niti od po 50 iteracija  
dynamic i guided koriste `chunk_size=7`  
kod dynamic prvoj niti se dodeljuje 25 iteracija, drugoj  $(200-25)/4$ , itd dok se ne dodje do veličine `chunk_size=7`

# primer

Sledeći kod se može paralelizovati pomoću OpenMP

```
for(int i = 0; i < N; i++){  
  for(int j = i; j < N; j++){  
    array[i*N + j] = sin(i) + cos(j);  
  }  
}
```

## Verzija A

**#pragma omp parallel for schedule(static)**

```
for(int i = 0; i < N; i++){  
  for(int j = i; j < N; j++){  
    array[i*N + j] = sin(i) + cos(j);  
  }  
}
```

## Verzija B:

**#pragma omp parallel for schedule(dynamic)**

```
for(int i = 0; i < N; i++){  
  for(int j = i; j < N; j++){  
    array[i*N + j] = sin(i) + cos(j);  
  }  
}
```

Koja varijanta  
će dati veće  
ubrzanje, npr.  
na 2 proc.sist.?

# Sinhronizacija direktive u OpenMP

\* Ove direktive omogućavaju da se sinhronizuje pristup deljivim promenljivim od strane više niti.

- Algoritam može zahtevati da se orkestrira (upravlja) redosledom pristupa deljivim promenljivim da bi se obezbedilo da više niti ne pristupa jednovremeno deljivoj promenljivoj radi upisa.

\* Za sinhronizaciju su na raspolaganju sledeće direktive

- barrier
- ordered
- critical
- atomic
- master
- bibliotečke funkcije – lock

# Barrier direktiva

- \* Barijera je tačka u izvršenju programa u kojoj niti čekaju jedna drugu tako da ni jedna nit ne može da produži sa izvršenjem dok sve niti iz tima niti ne stignu u barijeru.
  - Mnoge OMP direktive u sebi već sadrže implicitno barijere, tj. kompajler automatski umeće barijere na kraj direktive tako da sve niti tu čekaju dok se ne obavi ceo posao specificiran direktivom (npr. for petljom).
  - Zbog toga često nije neophodno da programer eksplicitno ubacuje barijere u program.
  - Međutim ako je to potrebno OMP to omogućava.
  - Sintaksa u C/C++
    - #pragma omp barrier**
    - Dve stvari o kojima se mora voditi računa kod korišćenja barrier direktive :
      - Do svake barijere moraju stići sve niti ili ni jedna nit
      - redosled barijera na koje nailaze niti mora biti isti za sve niti u timu
    - Bez ovih ograničenja moglo bi se desiti da se napiše program gde neke niti čekaju beskonačno dugo (zauvek) da ostale niti stignu u barijeru



# Primer1- Barrier sinhronizacija

```
#pragma omp parallel
{
    int id=omp_get_thread_num();
    A[id] = big_calc1(id);
    #pragma omp barrier

    B[id] = big_calc2(id, A);
}
```

- Barijere se najčešće koriste da bi se izbegle trke podacima.
- Ubacivanje barijere između upisa i čitanja deljive promenljive garantuje da se pristup obavlja u korektnom redosledu, npr. da se upis okončao pre nego što neka druga nit traži da pročita deljivu promenljivu

# Primer2 – barrier sinhronizacije

```
#pragma omp parallel shared (A, B, C) private(id)
{
    id=omp_get_thread_num();
    A[id] = big_calc1(id);
    #pragma omp barrier
    #pragma omp for
        for(i=0;i<N;i++){C[i]=big_calc3(i,A);}
    #pragma omp for nowait
        for(i=0;i<N;i++){ B[i]=big_calc2(C, i); }
    A[id] = big_calc4(id);
}
```

implicit barrier at the end of a  
for worksharing construct

implicit barrier at the end  
of a parallel region

no implicit barrier  
due to nowait

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

Šta je rezultat štampanja Print 1, 2 i 3 ?

```
int main(){
    int x;

    x = 2;
    #pragma omp parallel num_threads(2) shared(x)
    {

        if (omp_get_thread_num() == 0) {
            x = 5;
        } else {
            /* Print 1: the following read of x has a race */
            printf("1: Thread# %d: x = %d\n", omp_get_thread_num(), x );
        }

        #pragma omp barrier

        if (omp_get_thread_num() == 0) {
            /* Print 2 */
            printf("2: Thread# %d: x = %d\n", omp_get_thread_num(), x );
        } else {
            /* Print 3 */
            printf("3: Thread# %d: x = %d\n", omp_get_thread_num(), x );
        }
    }
    return 0;
}
```

# Direktiva ordered

✱ Omogućava da se strukturni blok u okviru paralelne petlje izvrši sekvencijalno.

- Sintaksa direktive u C/C++

**#pragma omp ordered**

*strukturni blok*

- Kod van ovog bloka se izvršava paralelno.

- Kada nit koja izvršava prvu iteraciju petlje naiđe na ordered direktivu, ona ulazi u strukturni blok bez čekanja.
- Da bi neka nit koja izvršava 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.
  - Korišćenje klauzule informiše kompajler da se koristi konstrukcija ordered

# Odredba ordered

Ova odredba nema nikakve argumente i podržana je samo sa petljama.

- Ova *odredba* mora da se iskoristi ako se **koristi direktiva `ordered`** paralelnom regionu
  - svrha ove odredbe je da informiše kompajler o prisustvu **ordered direktive**.

# Primer

```
#pragma omp parallel for default(none) ordered schedule(runtime) \  
    private(i,TID) shared(n,a,b)  
for (i=0; i<n; i++)  
{  
    TID = omp_get_thread_num();  
  
    printf("Thread %d updates a[%d]\n",TID,i);  
  
    a[i] += i;  
  
    #pragma omp ordered  
    {printf("Thread %d prints value of a[%d] = %d\n",TID,i,a[i]);}  
  
} /*-- End of parallel for --*/
```

- ordered direktiva se koristi da bi se obezbedilo da se elementi vektora **a**, **a[i]**, štampaju po redu  $i=0,1,2,\dots,n-1$ .
- Ažuriranja elemenata vektora **a** se mogu obavljati u bilo kom redosledu



```

#pragma omp parallel for default(none) ordered schedule(runtime) \
    private(i,TID) shared(n,a,b)
for (i=0; i<n; i++)
{
    TID = omp_get_thread_num();

    printf("Thread %d updates a[%d]\n",TID,i);

    a[i] += i;

    #pragma omp ordered
    {printf("Thread %d prints value of a[%d] = %d\n",TID,i,a[i]);}

} /*-- End of parallel for --*/

```

Mogući izlaz iz programa za n=9 i 4 niti

```

Thread 0 updates a[3]
Thread 2 updates a[0]
Thread 2 prints value of a[0] = 0
Thread 3 updates a[2]
Thread 2 updates a[4]
Thread 1 updates a[1]
Thread 1 prints value of a[1] = 2
Thread 3 prints value of a[2] = 4
Thread 0 prints value of a[3] = 6
Thread 2 prints value of a[4] = 8
Thread 2 updates a[8]
Thread 0 updates a[7]
Thread 3 updates a[6]
Thread 1 updates a[5]
Thread 1 prints value of a[5] = 10
Thread 3 prints value of a[6] = 12
Thread 0 prints value of a[7] = 14
Thread 2 prints value of a[8] = 16

```

# Direktiva critical

- \* Omogućava da niti pristupaju kritičnoj sekciji uzajamno isključivo.
  - Kritičnoj sekciji se opcionalno može dati ime.
    - Ovo ime je globalno i mora biti jedinstveno.
      - U protivnom bi ponašanje programa bilo nepredvidivo.
- \* 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.
  - Sve neimenovane kritične sekcije se preslikavaju u isto (nedefinisano) ime
- \* Sintaksa
  - #pragma omp critical** [(ime)]  
*strukturni blok*



# Primer

```
sum = 0;  
for (i=0; i<n; i++)  
    sum += a[i];
```

Sekvencijalna petlja

\* U okviru for petlje vrši se sumiranje elemenata vektora **a**.

- Ova operacija se može paralelizovati tako što će svaka nit nezavisno izvršiti sumiranje podskupa elemenata vektora **a** i zapamtiti rezultat u privatnoj promenljivoj.
- Kada sve niti okončaju sumiranje, one dodaju izračunatu vrednost deljivoj promenljivoj (sum) da bi se dobila ukupna suma

# primer korišćenje direktive critical

```
sum = 0;
#pragma omp parallel shared(n,a,sum) private(TID,sumLocal)
{
    TID = omp_get_thread_num();
    sumLocal = 0;
    #pragma omp for
        for (i=0; i<n; i++)
            sumLocal += a[i];
    #pragma omp critical (update_sum)
    {
        sum += sumLocal;
        printf("TID=%d: sumLocal=%d sum = %d\n",TID,sumLocal,sum);
    }
} /*-- End of parallel region --*/
printf("Value of sum after parallel region: %d\n",sum);
```

- Ubačena je imenovana kritična sekcija *update\_sum* u kojoj se pristupa deljivoj promenljivoj *sum* i štampa vrednost identifikatora niti (TID), lokalne sume (sumLocal) i vrednosti trenutno dobijene sume *sum* nakon dodavanja lokalne sume.
- nakon izlaska iz paralelnog regiona štampa se vrednost dobijene sume (sum)

# Mogući izlaz iz programa ako se koriste tri niti

```
TID=0: sumLocal=36 sum = 36  
TID=2: sumLocal=164 sum = 200  
TID=1: sumLocal=100 sum = 300  
Value of sum after parallel region: 300
```

# direktiva atomic

- \* Takođe omogućava da više niti ažurira deljivu promenljivu bez interferencije.
  - Za razliku od ostalih direktiva, ona se primenjuje samo na jednu naredbu dodeljivanja koja neposredno sledi iza konstrukcije atomic. Sintaksa u C/C++  
**#pragma omp atomic**  
*naredba*
- \* Atomic direktiva omogućava efikasno ažuriranje deljivih promenljivih od strane više niti na hardverskoj platformi koja podržava atomične operacije.
  - Razlog zašto se ova direktiva primenjuje samo na jednu naredbu je što atomične operacije realizovane u hardveru štite ažuriranja jedne memorijske lokacije (one koja se nalazi na levoj strani naredbe dodeljivanja).
    - Naime, ako hardver pruža podršku da se čitanje, modifikacija i upis u memorijsku lokaciju obave kao jedna nedeljiva (atomična) operacija, onda direktiva atomic nalaže kompajleru da iskoristi takvu instrukciju

```
int ic, i, n;  
ic = 0;  
#pragma omp parallel shared(n,ic) private(i)  
    for (i=0; i++, i<n)  
    {  
        #pragma omp atomic  
        ic = ic + 1;  
    }  
printf("counter = %d\n", ic);
```

# Direktiva master

\* definiše blok naredbi koji će izvršiti samo master nit.  
Slična je *single* direktivi.

\* Sintaksa

**#pragma omp master**

*strukturni blok*

\* Ova direktiva nema implicitnu barijeru na ulasku ili nakon okončanja strukturnog bloka.

- Zbog toga, ako je neophodno, treba eksplicitno ubaciti barrier direktivu da bi se postigla željena sinhronizacija.

- Npr. ako se master direktiva koristi da inicijalizuje podatke, mora se voditi računa da se inicijalizacija okonča pre nego što ostale niti u timu krenu da koriste podatke.

# Primer korišćenja master direktive

```
#pragma omp parallel shared(a,b) private(i)
{
```

```
    #pragma omp master
    {
```

```
        a = 10;
```

```
        printf("Master construct is executed by thread %d\n",
               omp_get_thread_num());
    }
```

```
#pragma omp barrier
```

```
#pragma omp for
```

```
    for (i=0; i<n; i++)
```

```
        b[i] = a;
```

```
} /*-- End of parallel region --*/
```

```
printf("After the parallel region:\n");
```

```
for (i=0; i<n; i++)
```

```
    printf("b[%d] = %d\n",i,b[i]);
```

Master construct is executed by thread 0

After the parallel region:

b[0] = 10

b[1] = 10

b[2] = 10

b[3] = 10

b[4] = 10

b[5] = 10

b[6] = 10

b[7] = 10

b[8] = 10

primer izlaza za n=9

# funkcije za sinhronizaciju

- \* pored navedenih direktiva OpenMP ima i skup run time funkcija koje su po načinu rada slične semaforima
- \* Opšti oblik ovih funkcija je

```
void omp_func_lock (omp_lock_t *lck)
```

- pri čemu *func* može biti *init*, *destroy*, *set*, *unset*, *test*, ili *init\_nest*, *destroy\_nest*, *set\_nest*, *test\_nest*
- Ove funkcije kao argument imaju promenljivu posebne namene, *lock* promenljivu kojoj se može pristupati samo preko ovih funkcija.
  - Postoje dva tipa brava (locks):
    - simple locks i
    - nestable locks (analogno binarnim i brojnim semaforima).
  - Simple lock promenljive se deklarišu kao promenljive tipa `omp_lock_t`, a nestable kao `omp_nest_lock_t`.

# Primer

```
omp_lock_t lck;  
omp_init_lock(&lck);  
#pragma omp parallel private (tmp, id)  
{  
    id = omp_get_thread_num();  
    tmp = do_lots_of_work(id);  
    omp_set_lock(&lck);  
    printf("%d %d", id, tmp);  
    omp_unset_lock(&lck);  
}
```

čekaj ovde da dođeš  
na red



oslobodi lock tako da  
sledeća nit može da  
pristupi

