

OpenMP

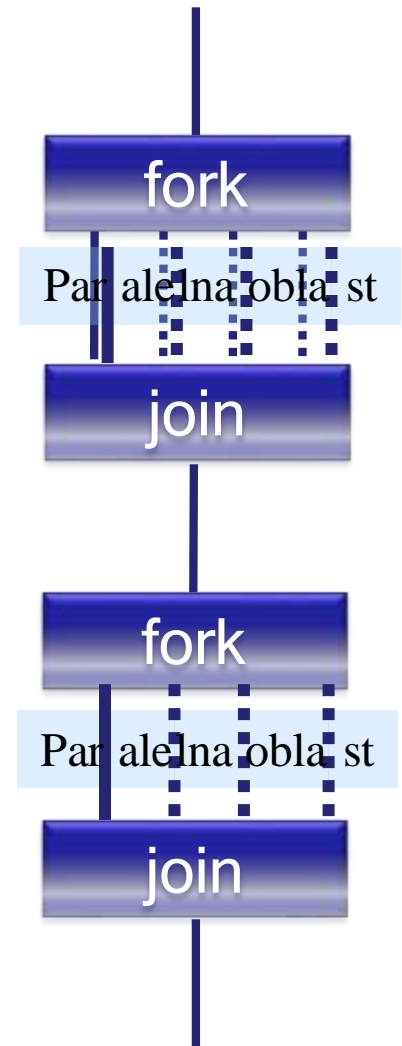
Open MultiProcessing

OpenMP

- OpenMP (Open MultiProcessing) je API koji nudi podršku za razvoj softvera na multicore procesorima sa deljivom memorijom u programskim jezicima C i C++ u okviru Visual Studio IDE.
- OpenMP je pre svega namenjen paralelizaciji for petlji

Fork-join model izvršenja

- Program počinje izvršenje kao jedna nit
- Svaki put kad naiđe na OpenMP paralelnu direktivu, kreira se tim niti i program postaje roditeljska nit i saraduje sa ostalim nitima na izvršenju programa
- Na kraju OpenMP paralelne direktive samo glavna nit nastavlja sa izvršenjem, dok ostale prekidaju izvršenje



OpenMP API

- OpenMP se sastoji iz skupa kompajlerskih direktiva, bibliotečkih rutina i promenljivih okruženja koji obezbeđuju pisanje paralelnih programa za multicore procesore sa deljivom memorijom
- OpenMP direktiva je posebno formatiran komentar ili pragma, koja se uglavnom primenjuje na kod koji sledi u nastavku programa. OpenMP obezbeđuje korisniku skup direktiva koje omogućavaju:
 - kreiranje tima niti za paralelno izvršavanje,
 - specifikaciju načina podele posla između članova tima,
 - deklaraciju zajedničkih i privatnih promenljivih,
 - sinhronizaciju niti

Kreiranje timova niti

- Tim niti kreira se da bi izvršio deo koda u paralelnoj oblasti OpenMP programa
- U tu svrhu koristi se direktiva **parallel**:
#pragma omp parallel [odredba [[,] odredba]...] novi_red
{
blok_naredbi
}
- Odredba može biti:
 - **if(logički izraz)**
 - **private(lista_promenljivih)**
 - **firstprivate(lista_promenljivih)**
 - **default(shared | none)**
 - **shared(lista_promenljivih)**
 - **reduction(operator: lista_promenljivih)**
 - **num_threads(celobrojni izraz)**
- Na kraju paralelnog regiona je implicitna *barijera* koja uslovljava da niti čekaju dok sve niti iz tima ne obave posao unutar paralelnog regiona.

Primer

```
#include <omp.h>
#include <stdio.h>
void main()
{
#pragma omp parallel
{
    int ID = omp_get_thread_num();/*funkcija koja vraća identifikator niti za koje se izvršava par.blok*/
    printf(" hello(%d) ", ID);
    printf(" world(%d) \n", ID);
}
}
```

Jedan od mogućih izlaza za 4 niti:

```
hello(1) hello(0) world(1)
world(0)
hello (3) hello(2) world(3)
world(2)
```

Runtime library funkcije

- `omp_set_num_threads(num)`, postavlja broj niti na *num*
- `omp_get_num_threads()`, vraća broj niti
- `omp_get_thread_num()`, vraća identifikator niti u timu
- `omp_get_wtime()`, vraća double precision vrednost jednaku broju sekundi proteklih od inicijalne vrednosti realtime časovnika operativnog sistema

Primer

```
double start;  
double end;  
start = omp_get_wtime();  
... work to be timed ...  
end = omp_get_wtime();  
printf("Work took %lf sec. time.\n", end-  
start);
```


Odredbe parallel direktive

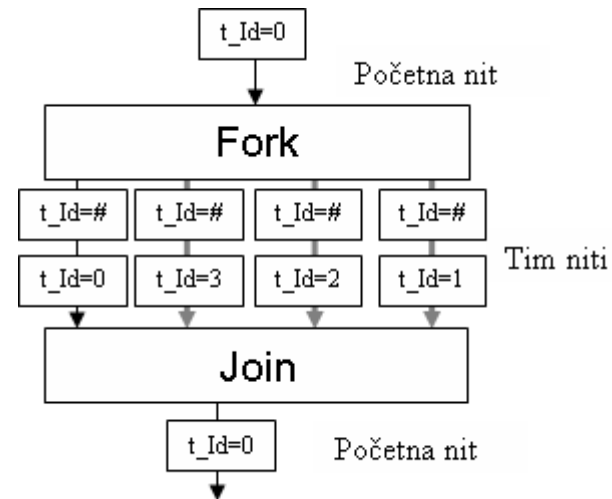
- **If odredba-** Ako uslov u if odredbi nije ispunjen kod se izvršava sekvencijalno
- **Private** odredba-služi za navođenje privatnih promenljivih za svaku nit(svaka nit ima svoju kopiju privatne promenljive koja nije inicijalizovana na ulasku u paralelni region).

Odredbe parallel direktive

- Po defaultu su **private**:
 - indeksna promenljiva for petlje koja se paralelizuje
 - promenljive koje su deklarisanе unutar strukturnog bloka paralelnog regiona
 - promenljive koje su deklarisanе unutar funkcije koja se poziva u okviru paralelnog regiona kao i parametri te funkcije
- **Firstprivate** odredba-ima isto značenje kao private samo što se promenljive inicijalizuju vrednostima koje su imale pre ulaska u paralelnu oblast

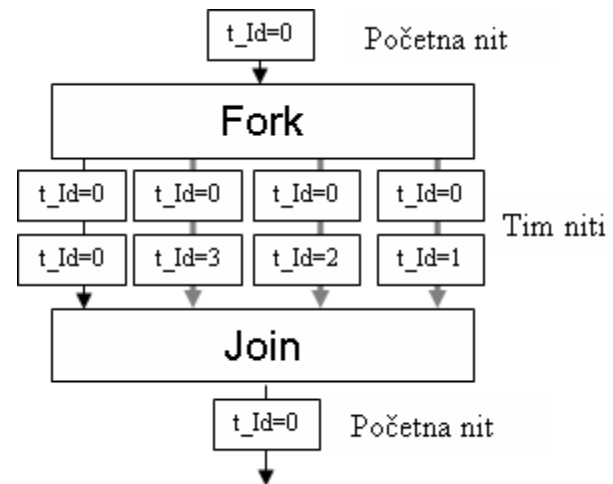
Primer. Private odredba

```
...  
int t_Id=0;  
#pragma omp parallel private (t_Id)  
{  
    tid = omp_get_thread_num();  
    ...  
}
```



Primer. Firstprivate odredba

```
int t_Id=0;  
  
#pragma omp parallel firstprivate (t_Id)  
{  
    tid = omp_get_thread_num();  
    ...  
}
```



Odredbe shared i default

- **Shared** odredba -Promenljive koje se navedu kao argumenti ove odredbe, zajedničke su za sve niti u timu, koje rade na izvršenju paralelne oblasti. Sve niti pristupaju istoj oblasti u memoriji. Sve promenljive deklarisanе van paralelnog regiona su ***shared***, po defaultu
- **Default** odredba-može biti u obliku default(shared) ili default(none).
 - Default(shared) ako želimo da većina promenljivih bude shared u paralelnom bloku onda u nastavku navodimo samo one koje odstupaju od toga
 - #pragma omp parallel default(shared) private(a,b,c)
 - Ako je default(none) onda se za svaku promenljivu mora specificirati posebno oblast važenja

Odredbe **num_threads** i **reduction**

- Odredba **num_threads** ima oblik **num_threads** (broj niti) postavlja broj niti za izvršenje u paralelnom bloku. Ovo može da se uradi i funkcijom *omp_set_num_threads(broj niti)*
- Odredba **reduction** vrši redukciju skalarne promenljive koja se javlja u listi promenljivih operatorom *op*. Ima oblik:
 - **reduction**(*op: lista_promenljivih*)

Odredba reduction

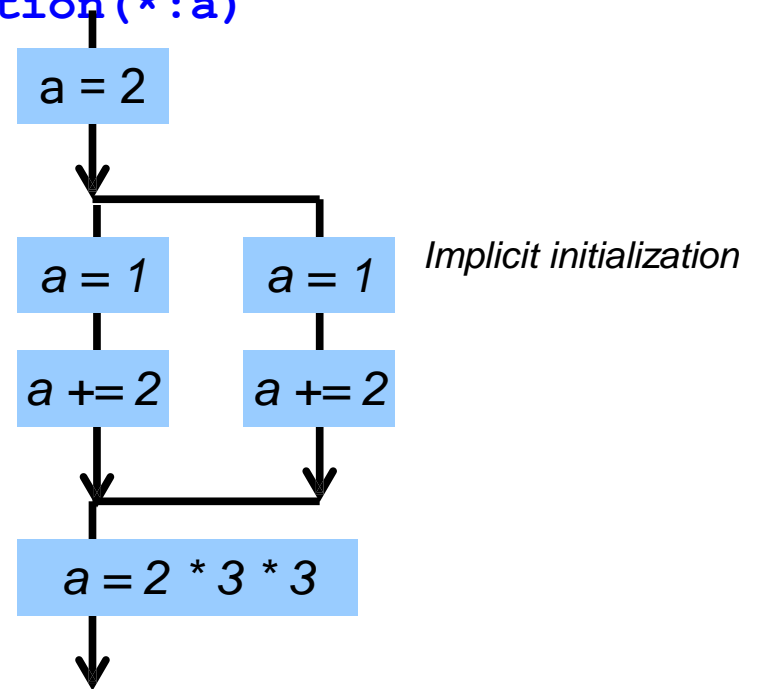
- Sintaksa ove odredbe je sledeća:
reduction(*op:lista_promenljivih*)
- Kreira se privatna kopija za svaku promenljivu koja se nalazi u *listi_promenljivih*, po jedna za svaku nit, i inicijalizuje u zavisnosti od operatora *op*(npr. 0 for “+”).
- Na kraju oblasti, originalna promenljiva se ažurira kombinacijom svoje originalne vrednosti sa krajnjim vrednostima svake privatne kopije, korišćenjem navedenog operatora.

Operator redukcije

- + suma
- * proizvod
- & bitsko I
- | bitsko ILI
- ^ bitsko isključivo ILI
- && logičko I
- || logičko ILI

Primer reduction

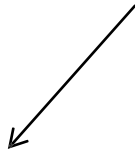
```
void main()  
{  
    int a = 2;  
    omp_set_num_threads(2);  
    #pragma omp parallel reduction(*:a)  
    {  
        a += 2;  
    }  
    printf("%d\n", a);  
}
```



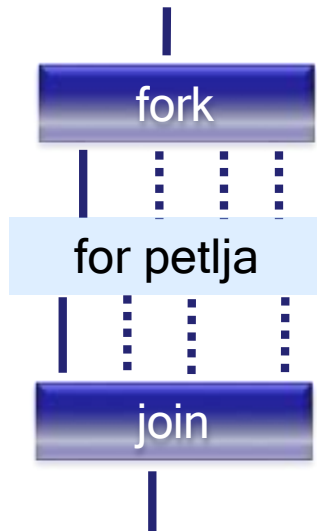
Direktive za podelu poslova

- Direktive za podelu poslova raspoređuju izvršenje određenog dela koda između niti u timu. One ne kreiraju nove niti i ne podrazumevaju postojanje naredbe **barrier** pre izvršenja direktive, ali podrazumevaju na kraju. Open MP definiše sledeće direktive za podelu poslova.
 - **for** direktiva (za distribuciju iteracija između niti)
 - **sections** direktiva (za distribuciju nezavisnih radnih jedinica između niti)
 - **single** direktiva (označava da će taj deo koda izvršavati samo jedna nit u timu)
- Svaka direktiva za podelu poslova mora da se nađe u aktivnom paralelnom bloku da bi imala efekta

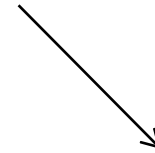
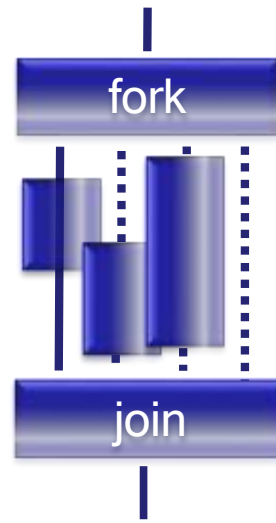
Podela poslova - direktive



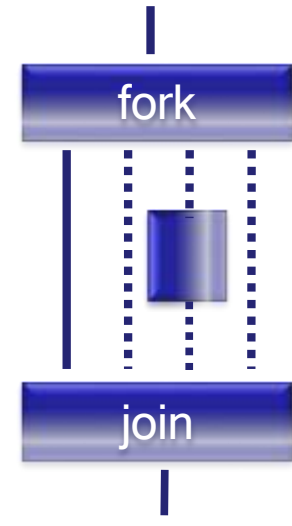
**Korišćenje
direktive `for`**



**Korišćenje direktive
`sections`**



**Korišćenjem
direktive `single`**



Direktiva for

- Raspodeljuje iteracije for petlje nitima u timu
- Najkorišćenija direktiva za podelu poslova
- **#pragma omp for** [*odredba* [[,] *odredba*]...] *novi_red*
for_petlja
- Odredba može biti:
 - **private**(*lista_promenljivih*)
 - **firstprivate**(*lista_promenljivih*)
 - **lastprivate**(*lista_promenljivih*)
 - **reduction**(*operator: lista_promenljivih*)
 - **ordered**
 - **schedule** (*kind*[, *chunk_size*])
 - **nowait**

Primer-Zbir dva niza

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

Primer-Zbir dva niza

Pr.n=4, p=2

T0(i=0,1)

$c[0] = a[0] + b[0];$

$c[1] = a[1] + b[1];$

T1(i=2,3)

$c[2] = a[2] + b[2];$

$c[3] = a[3] + b[3];$

Kombinovana parallel-for direktiva

```
#pragma omp parallel
```

```
{
```

```
#pragma omp for
```

```
for-loop
```

```
}
```

se može napisati kao:

```
#pragma omp parallel for
```

```
for-loop
```

Odredba lastprivate

- **Lastprivate** odredba-po izlasku iz paralelne oblasti izvršenja programa, vrednost koju promenljiva ima na kraju poslednje iteracije petlje dodeljuje originalnoj promenljivoj

Primer. Lastprivate odredba

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

Za n=5 i brojniti=3 izlaz:

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 `schedule`

- Odredba **`schedule`** specificira kako se iteracije u petlji dele između niti u timu. Ispravnost programa ne sme zavisiti od redosleda izvršavanja niti
- Vrednost parametra *`chunk_size`*, ukoliko je naveden, mora biti pozitivna, celobrojna konstantna vrednost (nepromenljiva u petlji)
- Parametar *`kind`* može imati sledeće vrednosti:
 - `static`
 - `dynamic`
 - `guided`
 - `runtime`

Odredba schedule (static)

- *static*-iteracije su podeljene na delove čija je veličina određena sa *chunk_size*. Delovi su dodeljeni nitima u round-robin redosledu. Svaka nit izvršava samo iteracije koje su joj dodeljene i poznate na početku izvršenja petlje
 - Ako *chunk-size* nije naveden onda se iteracije dele na približno jednake delove (n/p , n -broj iteracija petlje, p -broj niti) koji se dodeljuju nitima
- *Pr. $n=16, p=4$*

TID	0	1	2	3
Bez chunk-a	1-4	5-8	9-12	13-16
chunk = 2	1-2 9-10	3-4 11-12	5-6 13-14	7-8 15-16

Odredba schedule (*static*)

- Iako je broj iteracija izbalansiran po nitima korišćenjem *static* odredbe može se desiti da iteracije nisu podjednako opterećene poslom (neke izvršavaju više posla, neke manje) pa se i u slučaju static tada može javiti load imbalance koji se odražava na performanse jer brže niti čekaju sporije niti da završe

Odredba schedule (*dynamic*)

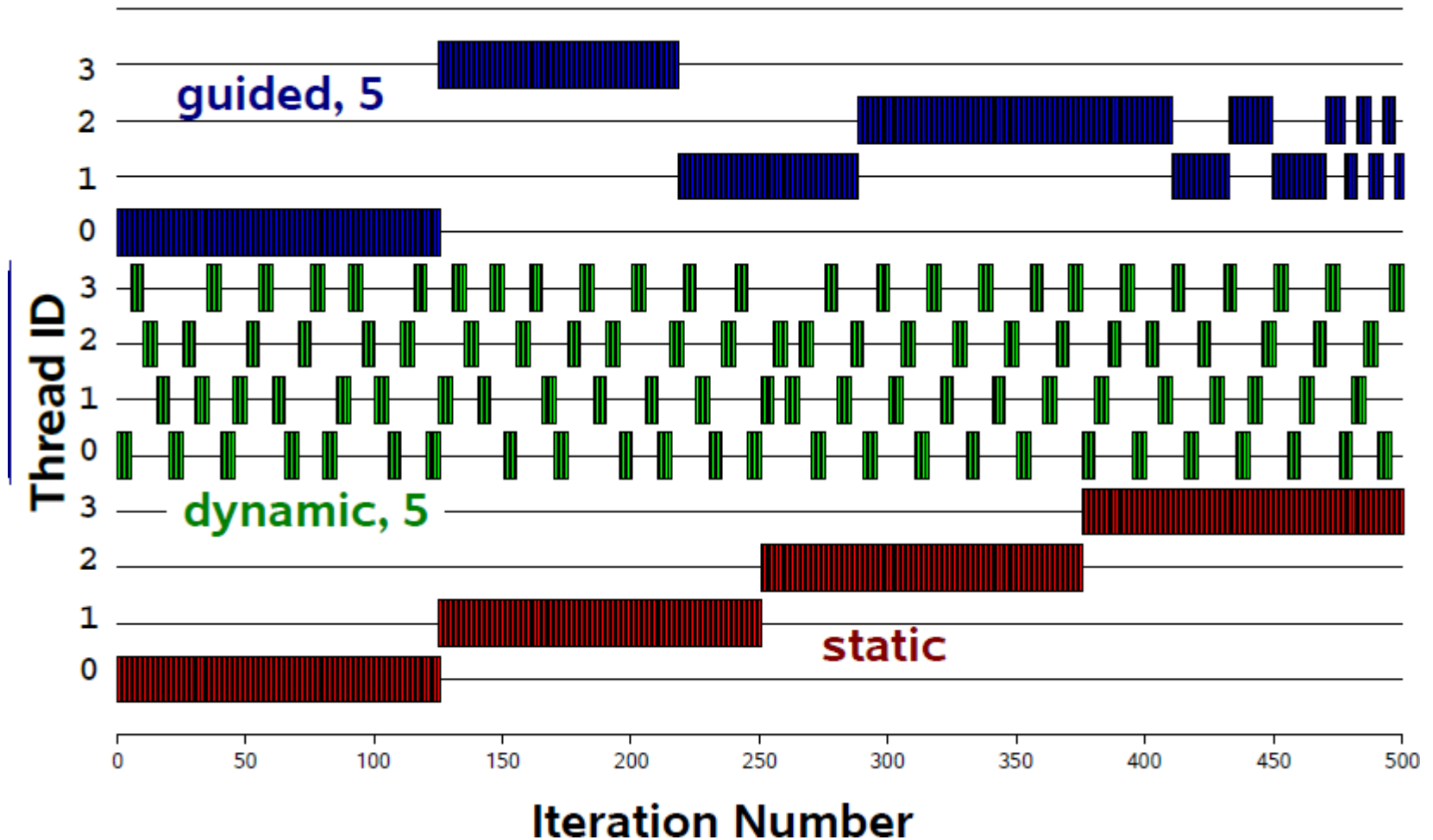
- *dynamic*-iteracije se dodeljuju nitima u toku izvršenja petlje i ne može se pretpostaviti redosled kojim će se iteracije dodeliti nitima.
 - iteracije su podeljene na delove, od kojih svaki sadrži *chunk_size* iteracija. Svaki od njih dodeljuje se niti, koja čeka na izvršenje. Nit izvršava iteracije i čeka sledeće, sve dok one postoje.
 - Ukoliko se *chunk_size* nije naveden, podrazumeva se 1
 - OpenMP runtime sistem mora da koordiniše dodeljivanje iteracija nitima koje su slobodne i zahtevaju nove i da obezbedi da se svaka iteracija izvrši samo jednom. Ovo unosi troškove sinhronizacije, koji ne postoje kod static odredbe
 - Što je manja veličina chunk-a kod dynamic bolji je load balance, ali su troškovi sinhronizacije veći

Odredba schedule

- *guided*-iteracije se dinamički dodeljuju nitima u delovima na osnovu veličine u opadajućem redosledu.
 - Kada nit završi sa izvršavanjem jednog dela, automatski zahteva i dobija drugi, i tako dok se svi ne izvrše
 - Inicijalno je veličina chunk-a $k_0 = n/p$, a onda se najčešće smanjuje po formuli $k_i = (1 - 1/p)^i k_{i-1}$
 - Prednost *guided* u odnosu na *dynamic* je ta što zbog manjeg broja chunk-ova imamo manje troškove sinhronizacije, ali komplikovanu funkciju za izračunavanje veličine chunk-a
- *runtime* - odluka o raspoređivanju se odlaže do početka izvršenja.
 - Način raspoređivanja i veličina poslova određuje se za vreme izvršavanja postavljanjem promenljive okruženja ***OMP_SCHEDULE***.

Odredba schedule

500 iterations on 4 threads



Odredba nowait

- Ukida podrazumevanu barijeru na kraju for direktive
- Treba je koristiti pažljivo

Single direktiva

```
#pragma omp single [odredba [ [,] odredba]...] novi_red  
    blok_naredbi
```

Odredba može biti:

private(lista_promenljivih)

firstprivate(lista_promenljivih)

lastprivate(lista_promenljivih)

nowait

Single direktiva

```
#pragma omp parallel
{
    printf("Hello from thread %d\n",
omp_get_thread_num());
    #pragma omp single
        printf("There are %d thread[s] \n ",
omp_get_num_threads());
}
```

Direktive za sinhronizaciju

- Critical direktiva

```
#pragma omp critical novi_red  
    blok_naredbi
```

- Direktiva critical omogućuje da samo jedna nit u jednom trenutku izvrši deo koda (kritičnu oblast) koji sledi nakon ove direktive. Kada nit koja se izvršava naiđe na direktivu critical operativni sistem proverava da neka druga nit nije već započela izvršenje kritične oblasti. Ukoliko to nije slučaj, nit izvršava kritičnu oblast. Ako je neka druga nit započela izvršenje kritične oblasti, nit koja naiđe na direktivu critical se blokira i čeka da prethodna nit izađe iz kritične oblasti.

Direktive za sinhronizaciju

- Barrier direktiva
`#pragma omp barrier`
- Direktiva `barrier` sinhronizuje sve niti u timu. Kada dođe do ove direktive, svaka nit “čeka” dok sve ostale niti u timu ne dođu do barrier direktive. Nakon što niti stignu do direktive `barrier` svaka nit nastavlja sa izvršenjem svog dela koda

Direktive za sinhronizaciju

```
#pragma omp parallel
```

```
{
```

```
/* All threads execute this. */
```

```
SomeCode();
```

```
#pragma omp barrier
```

```
/* All threads execute this, but not before
```

```
* all threads have finished executing SomeCode().
```

```
*/
```

```
SomeMoreCode();
```

```
}
```

Faktorijel sa critical direktivom

```
void main()
{
    int fac = 1, number;
    scanf("%d",&number);
    #pragma omp parallel for
    for(int n=2; n<=number; ++n)
    {
        #pragma omp critical
        fac *= n;
    }
}
```

Faktorijel sa critical direktivom

```
void main()
{  int fac = 1, number;
   scanf("%d",&number);
   #pragma omp parallel for
   for(int n=2; n<=number; ++n)
   {
       #pragma omp critical
       fac *= n;
   }
}
```

Primer-faktorijel critical

Pr.number=5, p=2

fac=1;

T0(n=2,3)

fac=fac*2=2;

fac=fac*3=2*4*3;

T1(n=4,5)

fac=fac*4=2*4;

fac=fac*5=2*4*3*5;

Primer-faktorijel reduction

Pr.number=5, p=2

fac=1;

T0(n=2,3)

fac=1;

fac=fac*2=2;

fac=fac*3=2*3;

T1(n=4,5)

fac=1;

fac=fac*4=4;

fac=fac*5=4*5;

=>fac=1*2*3*4*5(=5!)

Faktorijel sa reduction odredbom

```
void main()
{  int fac = 1,number;
   scanf("%d",&number);
   #pragma omp parallel for reduction(*:fac)
   for(int n=2; n<=number; ++n)
       fac *= n;
   printf("%d",fac);
}
```

Množenje matrice i vektora

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

```
#include <stdlib.h>
```

```
void main(int argc, char *argv[])
```

```
{
```

```
    double *a,*b,*c;
```

```
    int i, j, m, n;
```

```
    scanf("%d %d",&m,&n);
```

```
    a=(double *)malloc(m*sizeof(double))
```

```
    b=(double *)malloc(m*n*sizeof(double))
```

```
    c=(double *)malloc(n*sizeof(double))
```

```
for (j=0; j<n; j++)
    c[j] = 2.0;
for (i=0; i<m; i++)
    for (j=0; j<n; j++)
        b[i*n+j] = i;
#pragma omp parallel for private(j)
for (i=0; i<m; i++)
{
    a[i] = 0.0;
    for (j=0; j<n; j++)
        a[i] += b[i*n+j]*c[j];
}
free(a);free(b);free(c);
}
```

Primer

Pr.m=4,n=2, p=2

T0(i=0,1)

i=0:

j=0:a[0] = b[0]c[0];

j=1:a[0] = b[0]c[0]++ b[1]c[1]

i=1:

j=0:a[1] = b[2]c[0];

j=1:a[1] = b[2]c[0]++ b[3]c[1]

T1(i=2,3)

i=2:

j=0:a[2] = b[4]c[0];

j=1:a[2] = b[4]c[0]+ b[5]c[1]

i=3:

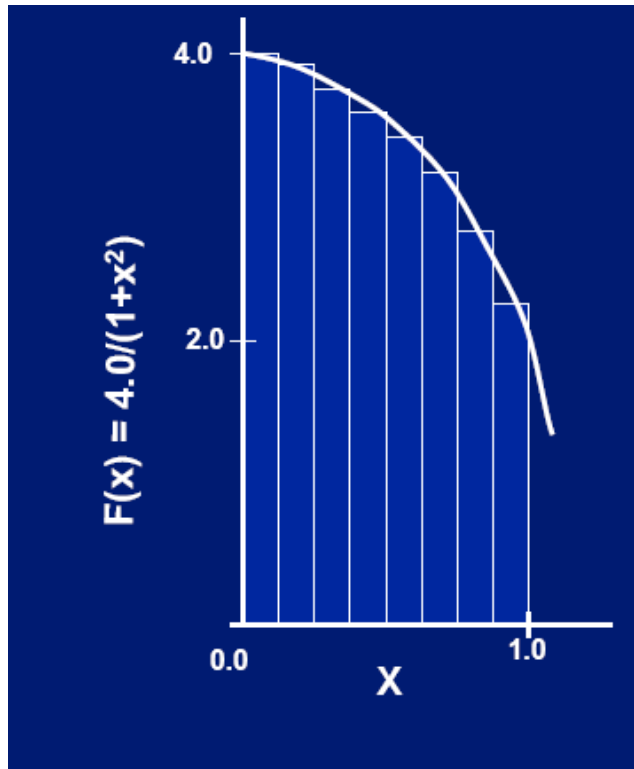
j=0:a[3] = b[6]c[0];

j=1:a[3] = b[6]c[0]++ b[7]c[1]

Da li je paralelizacija moguća po indeksu j?

Numerička integracija

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$



$$\sum_{i=0}^N F(x_i) \Delta x \approx \pi$$

Sekvencijalno rešenje

```
#include <stdio.h>
static long num_steps = 1000000000;      double step;
void main ()
{   int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    for (i=1;i<= num_steps; i++){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
    printf("pi sekvencijalno=%lf\n",pi);
}
```

1. OpenMP rešenje sa critical direktivom

```
static long num_steps = 100000;
#define NUM_THREADS 10          double step;
void main ()
{   int i;  double x, pi, sum =0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel private (x)
{   #pragma omp for
    for (i=1;i<=num_steps;i++){
        x = (i-0.5)*step;
        #pragma omp critical
            sum += 4.0/(1.0+x*x);
    }
}
pi = sum*step;
printf("%lf",pi);
}
```


2. OpenMP rešenje sa critical direktivom

```
static long num_steps = 100000;      double step;
#define NUM_THREADS 10
void main ()
{
    int i;  double x, sum, pi=0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel private (x, sum)
    {
        sum=0.0;
        #pragma omp for
        for (i=1;i<=num_steps;i++)
            { x = (i-0.5)*step;
              sum += 4.0/(1.0+x*x); }
        #pragma omp critical
            pi += sum*step;
    }
    printf("%lf",pi);
}
```

OpenMP rešenje korišćenjem odredbe reduction

```
#include <stdio.h>
#include <omp.h>
static long num_steps = 100000000;      double step;
#define NUM_THREADS 2
void main ()
{
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel for reduction(+:sum) private(x)
        for (i=1;i<= num_steps; i++){
            x = (i-0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
        }
    pi = step * sum;
    printf("pi sa redukcijom=%lf\n",pi);
}
```

Direktiva task

- Dostupna od OpenMP verzije 3.0 - doprinosi paralelizaciji beskonačnih i while petlji i rekurzivnih problema
- Task - nezavisna jedinica izvršenja

#pragma omp task [odredba/e]

Odredbe:

- if
- untied
- private
- firstprivate
- shared
- default
- depend (u novijim verzijama)

Direktiva task

```
omp_set_num_threads( 2 );  
#pragma omp parallel default(none)  
{  
    #pragma omp task  
        printf( "A\n" );  
  
    #pragma omp task  
        printf( "B\n" );  
}
```

- Redosled izvršenja taskova je defaultno nedefinisan
- Svaka nit kreira taskove!

Direktiva task

```
omp_set_num_threads( 2 );  
#pragma omp parallel  
{  
    #pragma omp single  
    {  
        #pragma omp task  
        printf( "A\n" );  
  
        #pragma omp task  
        printf( "B\n" );  
    }  
}
```

Task - obrada lančane liste

```
#pragma omp parallel
{
    #pragma omp single nowait
    {
        element *p = listHead;
        while( p != NULL )
        {
            #pragma omp task firstprivate(p)
            Process( p );
            p = p->next;
        }
    }
    #pragma omp taskwait
}
```

Task - Fibonačijev niz

```
#pragma omp parallel
{
    #pragma omp single
    {
        fib(input);
    }
}
```

```
int fib( int n)
{
    int fn1, fn2;
    if (n < 2 ) return (n);
    #pragma omp task shared(fn1) firstprivate (n)
        fn1 = fib (n-1);

    #pragma omp task shared(fn2) firstprivate (n)
        fn2 = fib (n-2);

    #pragma omp taskwait
    return(fn1 + fn2);
}
```

Odredba depend

```
#pragma omp task depend(in: x) depend (out: y)  
                                depend (inout: z)
```

- Koristi se za definisanje redosleda izvršenja taskova, ukoliko između njih postoje određene zavisnosti
- Korišćenjem opcije IN, task postaje zavistan od taska koji je prethodno navedenu promenljivu koristio kao OUT ili INOUT, i mora sačekati da se taj task završi pre nego što krene sa svojim izvršenjem

Odredba depend

```
#pragma omp parallel
{
    #pragma omp single
    {
        int x, y, z ;
        #pragma omp task depend (out: x )
            x = init( );
        #pragma omp task depend (in: x) depend (out: y)
            y = f(x) ;
        #pragma omp task depend (in: x) depend (out: z)
            z = g(x);
        #pragma omp task depend (in: y, z)
            finalize(y, z) ;
    }
}
```

Noviji koncepti

- taskgroup [task_reduction]
- taskloop
- taskyield

Zavisnosti po podacima

- U slučaju paralelizacije petlje neophodno je zadržati tačnost programa. Program koji je paralelizovan je beskoristan ako se izvršava brzo a generiše netačne rezultate.
- Ako program ne sadrži zavisnosti po podacima njega je lako paralelizovati, u suprotom treba koristiti tehnike koje otklanjaju mogućnost da paralelizacijom te zavisnosti dovedu do netačnih rezultata
- Kad god neka naredba programa čita/upisuje u memorijsku lokaciju a druga naredba čita/upisuje u istu memorijsku lokaciju i najmanje jedna od njih upisuje u tu lokaciju kažemo da postoji zavisnost po podacima između te dve naredbe.

Zavisnosti po podacima

- Npr. Petlja sa zavisnostima koja onemogućava paralelizaciju (može se desiti da se pročita vrednost za $a[1]$ pre nego što se u $a[1]$ upiše vrednost)

```
for(i=1; i < N-1; i++)
```

```
  a[i] = a[i] + a[i-1];
```

```
i=1:
```

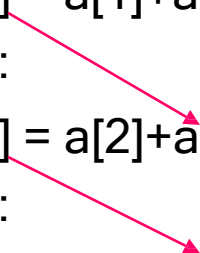
```
a[1] = a[1] + a[0];
```

```
i=2:
```

```
a[2] = a[2] + a[1];
```

```
i=3:
```

```
a[3] = a[3] + a[2];
```



Zavisnosti po podacima

- Npr. Odrediti da li u navedenim primerima postoje zavisnosti po podacima između iteracija petlje

<pre>for(i=2;i<=N; i+=2) a[i] = a[i] + a[i-1];</pre>	<pre>i=2: a[2] = a[2]+a[1]; i=4: a[4] = a[4]+a[3]; i=6: a[6] = a[6]+a[5];</pre>	=> ne postoje zavisnosti
<pre>for(i=1;i<=N/2; i++) a[i] = a[i] + a[i+N/2];</pre>	<pre>i=1: a[1] = a[1]+a[1+N/2]; i=2: a[2] = a[2]+a[2+N/2]; ... i=N/2: a[N/2] = a[N/2]+a[N];</pre>	=> ne postoje zavisnosti

Zavisnosti po podacima

- Npr. Odrediti da li u navedenim primerima postoje zavisnosti po podacima između iteracija petlje

```
for(i=1;i<=N/2+1;i++)  
    a[i] = a[i] + a[i+N/2];
```

i=1:

a[1] = a[1]+a[1+N/2];

i=2:

a[2] = a[2]+a[2+N/2]; => postoje zavisnosti

...

i=N/2+1:

a[N/2+1] = a[N/2+1]+a[N+1];



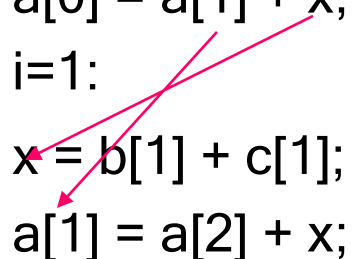
Klasifikacija zavisnosti

- Zavisnosti se mogu klasifikovati na osnovu toga da li su loop-carried(LC) ili nisu (non-loop-carried-NLC). LC su one koje se javljaju između različitih iteracija petlji, a NLC u pojedninačnoj iteraciji. Ono što može sprečiti korektnu paralelizaciju su LC zavisnosti, dok se NLC ignorišu.
- Neka je S1 naredba koja se ranije izvršava u sekvencijalnom izvršenju petlje, a S2 kasnije. Moguće su sledeće zavisnosti između ovih naredbi:
 - S1 upisuje vrednost u mem. lokaciju, a S2 čita vrednost sa iste mem. lokacije (flow-dependence)
 - S1 čita vrednost sa mem. lokacije, a S2 upisuje vrednost u istu mem. lokaciju (anti-dependence)
 - S1 i S2 upisuju u istu mem. lokaciju (output-dependence)
- Petlje sa anti-dependence i output-dependence je uvek moguće paralelizovati dok one sa flow dependence nekad jeste a nekad nije moguće paralelizovati

Otklanjanje anti-dependence

```
for(i=0; i < N-1; i++)  
{  
  x = b[i] + c[i];  
  a[i] = a[i+1] + x;  
}
```

```
i=0:  
x = b[0] + c[0];  
a[0] = a[1] + x;  
i=1:  
x = b[1] + c[1];  
a[1] = a[2] + x;
```



Ovu petlju je nemoguće paralelizovati u ovom obliku jer postoje zavisnosti između iteracija petlje (prikazane na slici)

Ako svaka iteracija inicijalizuje mem. lokaciju pre nego što je S1 pročita tu lokaciju (u primeru vrednost x) anti-dependence se otklanja privatizacijom promenljive. Da bi vrednost bila dostupna i posle petlje, treba lastprivate.

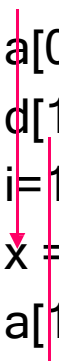
Ako je vrednost koju čita S1 inicijalizovana pre (u primeru vrednost a[i+1]) anti-dependence se otklanja tako što se pravi kopija vrednosti (niza a) pre petlje, a u petlji se čita kopija a ne originalna vrednost.

```
#pragma omp parallel for shared (a, a2)  
for(i=0; i < N-1; i++)  
  a2[i] = a[i+1];  
#pragma omp parallel for shared (a, a2) lastprivate(x)  
for(i=0; i < N-1; i++)  
{  
  x = b[i] + c[i];  
  a[i] = a2[i] + x;  
}
```


Otklanjanje output-dependence

//d[2] i niz a,b, i c su inicijalizovani
pre petlje

```
for(i=0;i< N; i++)  
{  
  x = (b[i] + c[i])/2;  
  a[i] = a[i] + x;  
  d[1]=2*x;  
}  
y=x+d[1]+d[2];  
i=0;  
x = (b[0] + c[0])/2;  
a[0] = a[0] + x;  
d[1]=2*x;  
i=1;  
x = (b[1] + c[1])/2;  
a[1] = a[1] + x;  
d[1]=2*x;  
y=x+d[1]+d[2];
```



Ovu petlju je nemoguće paralelizovati u ovom obliku jer postoje zavisnosti između iteracija petlji(prika zane na slici). Ove zavisnosti se otklanjaju privatizacijom promenljive, a kako se vrednosti za x, d[1] i d[2] potrebne van petlje, rešenje je lastprivate. Međutim, postoji problem kada se niz (ili struktura) nadje u okviru lastprivate odredbe i ako se samo neki elementi niza menjaju u poslednjoj iteraciji, ostali elementi niza (strukture) posle petlje imaju nedefinisano vrednost.(ovde bi d[2] bilo nedefinisano). Zato ćemo uvesti u okviru petlje vrednost d1 čiju će vrednost nakon petlje da preuzme d[1]

```
#pragma omp parallel for shared (a) lastprivate(x,d1)  
for(i=0;i< N; i++)  
{  
  x = (b[i] + c[i])/2;  
  a[i] = a[i] + x;  
  d1=2*x;  
}  
d[1]=d1;y=x+d[1]+d[2];
```

Otklanjanje flow-dependence

Uglavnom je ove zavisnosti teško otkloniti, ali postoje slučajevi gde je to moguće postići

1.Redukciona izračunavanja

```
x = 0.0;
```

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

```
{
```

```
x = x + a[i];
```

```
}
```

```
i=0:
```

```
x = x + a[0];
```

```
i=1:
```

```
x = x + a[1];
```

```
i=2:
```

```
x = x + a[2];
```

Paralelizacija je moguća uvođenjem redukcije:

```
x = 0.0;
```

```
#pragma omp parallel for reduction(+:x)
```

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

```
{
```

```
    x = x + a[i];
```

```
}
```

Otklanjanje flow-dependence

2.Otklanjanje eliminacijom indukcionih promenljivih

Ako petlja ažurira vrednost promenljive na isti način kao redukcija, ali i koristi vrednost te promenljive u nekom izrazu koji nije onaj koji ažurira vrednost, onda ne možemo da otklonimo zavisnost odredbom redukcije. Međutim, postoje izračunavanja kod kojih je vrednost redukcione promenljive za vreme svake iteracije jednostavna funkcija indeksa petlje (množenje konstantom, sabiranje konstantom, sabiranje sa indeksom petlje...) i tada promenljivu možemo zameniti izrazom koji sadrži indeks petlje.

```
x=0;
for(i=0;i<=N;i++)
{
a[i] = x;
x=x+1;
}
```

Sekvencijalno:

i=0:a[0]=0;x=1

i=1:a[1]=1;x=2

i=2:a[2]=2;x=3

```
x=0;
#pragma omp parallel for reduction(+:x)
for(i=0;i<=N;i++)
{
a[i] = x;
x=x+1;
}
```

Paralelno:

i=0:	i=1:	i=2:
x=0;	x=0;	x=0;
a[0]=0;	a[1]=0;	a[2]=0;
x=1;	x=1;	x=1;

Otklanjanje flow-dependence

2.Otklanjanje eliminacijom indukcionih promenljivih

Ako petlja ažurira vrednost promenljive na isti način kao redukcija, ali i koristi vrednost te promenljive u nekom izrazu koji nije onaj koji ažurira vrednost onda ne možemo da otklonimo ovu FD sa odredbom redukcije. Međutim postoje izračunavanja kod kojih je vrednost redukcione promenljive za vreme svake iteracije jednostavna funkcija indeksa petlje (množenje konstantom, sabiranje konstantom, sabiranje sa indeksom petlje...) i tada promenljivu možemo zameniti izrazom koji sadrži indeks petlje.

```
idx = N/2+1;  
isum = 0; pow2 = 1;  
for(i=0;i<=N/2; i++)  
{  
  a[i] = a[i] + a[idx];  
  b[i] = isum;  
  c[i] = pow2;  
  idx=idx+1;  
  isum=isum + i;  
  pow2=pow2 *2;  
}
```

Postoje flow-dependence po idx, isum i pow2 čijim otklanjanjem dobijamo:

```
#pragma omp parallel for  
for(i=0;i<=N/2; i++)  
{  
  a[i]+= a[i+1+N/2];  
  b[i] = (i*(i-1))/2;  
  c[i] = (int)pow((float)2,i);  
}
```

Otklanjanje flow-dependence

2.Otklanjanje eliminacijom indukcionih promenljivih

Ako petlja ažurira vrednost promenljive na isti način kao redukcija, ali i koristi vrednost te promenljive u nekom izrazu koji nije onaj koji ažurira vrednost onda ne možemo da otklonimo ovu FD sa odredbom redukcije. Međutim postoje izračunavanja kod kojih je vrednost redukcione promenljive za vreme svake iteracije jednostavna funkcija indeksa petlje (množenje konstantom, sabiranje konstantom, sabiranje sa indeksom petlje...) i tada promenljivu možemo zameniti izrazom koji sadrži indeks petlje.

$idx = N/2 + 1$; $isum = 0$; $pow2 = 1$;

$i=0$:

$a[0] += a[idx]$;

$b[0] = isum$;

$c[0] = pow2$;

$idx = N/2 + 2$;

$isum = 0$;

$pow2 = 2$;

$i=1$:

$a[1] += a[idx]$;

$b[1] = isum$;

$c[1] = pow2$;

$idx = N/2 + 3$

$isum = 1$;

$pow = 2 * 2$;

$i=2$:

$a[2] += a[idx]$;

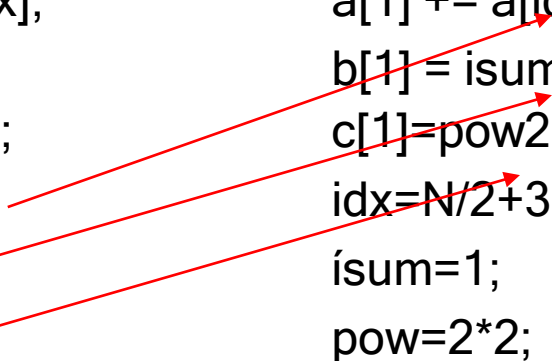
$b[2] = isum$;

$c[2] = pow2$;

$idx = N/2 + 4$;

$isum = 1 + 2$;

$pow = 2 * 2 * 2$;



Otklanjanje flow-dependence

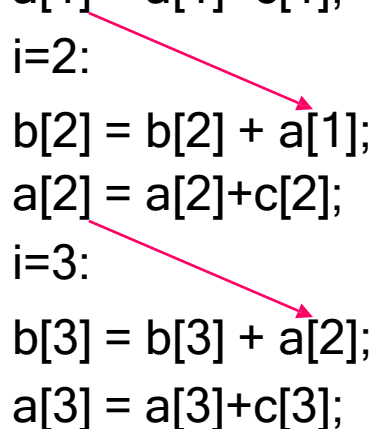
3. Krivljenje(Loop skewing). Ovom tehnikom LC zavisnosti se transformišu u NLC zav.

```
for(i=1; i< N; i++)  
{  
  b[i] = b[i] + a[i-1];  
  a[i] = a[i]+c[i];  
}
```

Ovu petlju je nemoguće paralelizovati u ovom obliku jer postoje zavisnosti između iteracija petlji(prikazane na slici) tj. $a[i]$ u koji se upisuje u i -toj iteraciji i $a[i-1]$ iz koga čita istu vrednost u $i+1$ iteraciji i pritom $a[i]$ ne zavisi ni od jednog drugog elementa niza a .

Paralelizacija je moguća sa sledećom transformacijom:

```
i=1:  
b[1] = b[1] + a[0];  
a[1] = a[1]+c[1];  
i=2:  
b[2] = b[2] + a[1];  
a[2] = a[2]+c[2];  
i=3:  
b[3] = b[3] + a[2];  
a[3] = a[3]+c[3];
```



```
b[1]=b[1]+a[0];  
#pragma omp parallel for shared (a,b,c)  
for(i=1; i< N-1; i++)  
{  
  a[i] = a[i] + c[i];  
  b[i+1] = b[i+1]+a[i];  
}  
a[N-1] = a[N-1]+c[N-1];
```

Otklanjanje flow-dependence

Postoje zavisnosti koje je teško ili nemoguće otkloniti, ali je nekad moguće paralelizovati neki drugi deo koda koji sadrži zavisnosti. Postoje različite tehnike da se to uradi, ali je bitno da se pritom ne promene druge zavisnosti ili ne uvedu nove.

```
for(i=1;i< m; i++)
  for(j=0;j<n;j++)
  {
    a[i][j] = 2.0*a[i-1][j];
  }
```

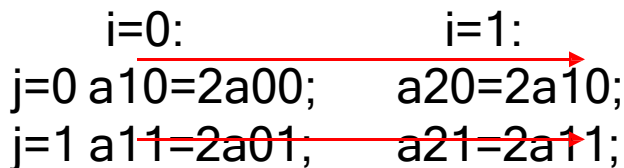
Ovu petlju je nemoguće paralelizovati u ovom obliku jer postoje zavisnosti po indeksu i između iteracija petlji.
=> Pošto ne postoje zavisnosti po j , korektno je da se paralelizuje j petlja:

```
for(i=1;i< m; i++)
  #pragma omp parallel for
  for(j=0;j<n;j++)
  {
    a[i][j] = 2.0*a[i-1][j];
  }
```

Dobijena petlja je paralelizovana ali je može biti loša sa stanovišta performansi, jer imamo $m-1$ fork-join koraka.
=> Pošto je moguće zameniti indekse petlji dobićemo najbolje rešenje:

```
#pragma omp parallel for private (i)
for(j=0;j< n; j++)
  for(i=1;i<m;i++)
  {
    a[i][j] = 2.0*a[i-1][j];
  }
```

$i=0:$ $j=0$ $a_{10}=2a_{00};$ $a_{20}=2a_{10};$
 $j=1$ $a_{11}=2a_{01};$ $a_{21}=2a_{11};$



Otklanjanje flow-dependence

Paralelizacija dela petlje cepanjem

```
for (i=1; i<n; i++)  
{  
    a[i]+=a[i-1];  
    y=y+c[i];  
}
```

=>

Ovu petlju je nemoguće paralelizovati u ovom obliku jer postoje zavisnosti po indeksu i po promenljivoj a između iteracija petlji. Pošto se naredba $y=y+c[i]$ može paralelizovati uvođenjem odredbe redukcije. Petlju možemo pocepati na dva dela i taj drugi deo paralelizovati:

```
for (i=1; i<n; i++)  
    a[i]+=a[i-1];  
#pragma omp parallel for reduction(+:y)  
for (i=1; i<n; i++)  
    y=y+c[i];
```