

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:

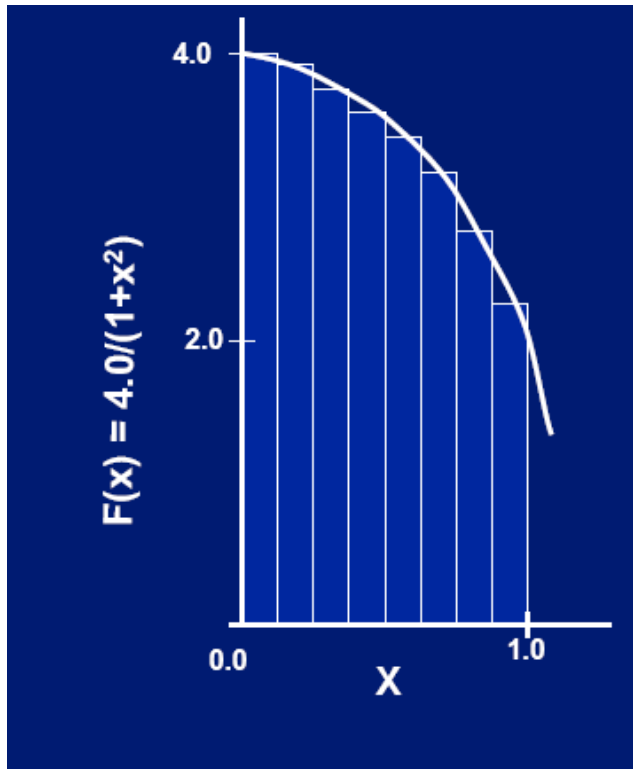
j=0:a[3] = b[6]c[0];

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; sum=0.0;
    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;      double step;
#define NUM_THREADS 10
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);
}
```


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=0; i < N-1; i++)
```

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

```
i=1:
```

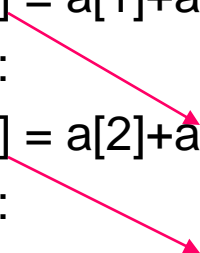
```
a[1] = a[1] + a[0];
```

```
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

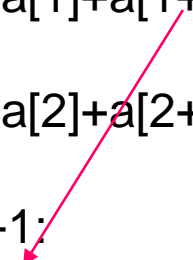
for(i=2;i<=N; i+=2)	i=2:	
a[i] = a[i] + a[i-1];	a[2] = a[2]+a[1];	
	i=4:	
	a[4] = a[4]+a[3];	=> ne postoje zavisnosti
	i=6:	
	a[6] = a[6]+a[5];	
for(i=1;i<=N/2; i++)	i=1:	
a[i] = a[i] + a[i+N/2];	a[1] = a[1]+a[1+N/2];	
	i=2:	
	a[2] = a[2]+a[2+N/2];	=> ne postoje zavisnosti
	...	
	i=N/2:	
	a[N/2] = a[N/2]+a[N];	

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];
```



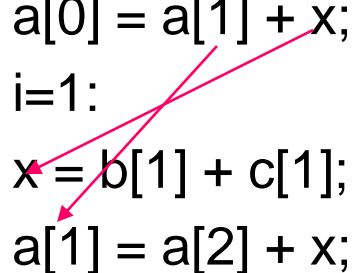
Klasifikacija zavisnosti

- 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 flowdependence 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.

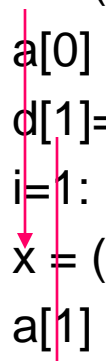
Ako je vrednost koju čita S1 inicijalizovana pre (u primeru vrednost a[i+1]) anti-dependence se otklanja tako što se pravi kopija vrednosti (niza a) pre petlje, a u petlji se čita kopija a ne originalna vrednost.

```
#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

//d2 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 petlje (prikazane 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 u 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 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.

```
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:

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

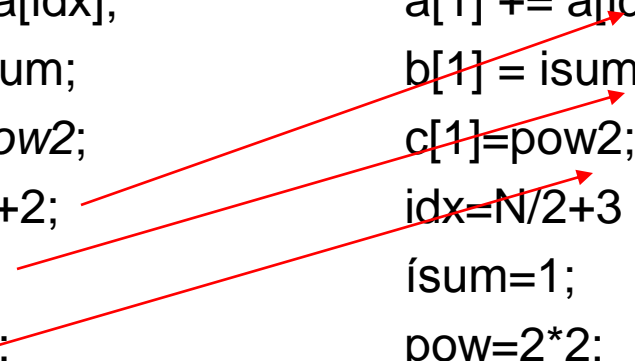
$b[2] = isum$;

$c[2] = pow2$;

$idx = N/2+4$;

$isum = 1+2$;

$pow = 2*2*2$;



Otklanjanje flow-dependence

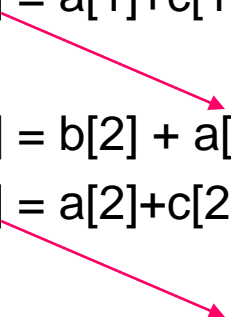
3. Krivljenje (Loop skewing). Ovom tehnikom LC zavisnosti se transformišu u NLC zav.

```
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 petlje (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 petlje.

=> 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 petlje 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 pocepiti 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];
```