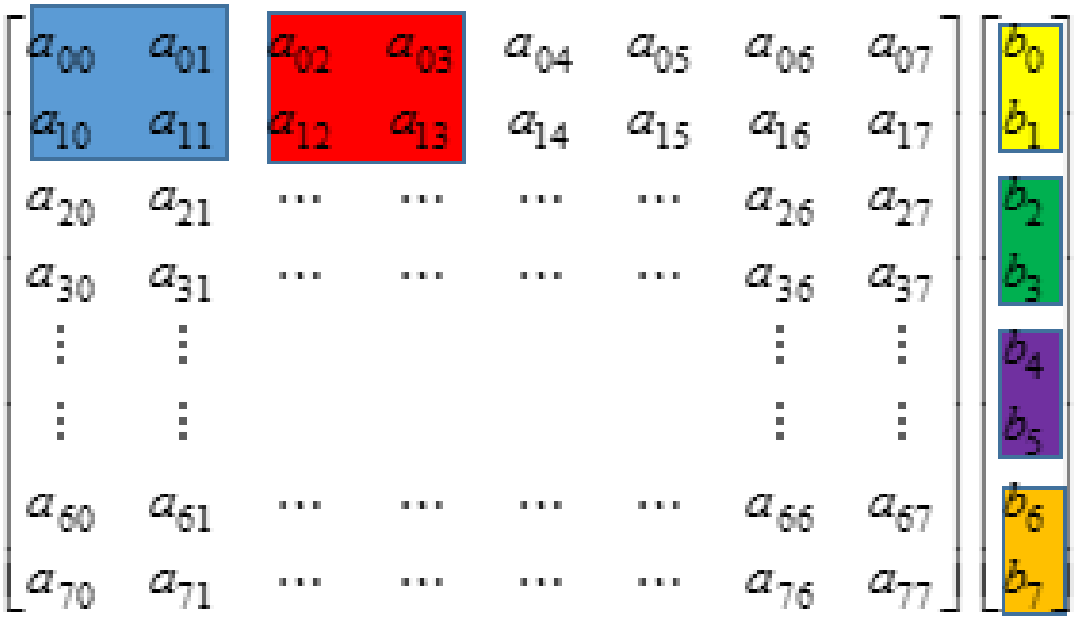
Napisati MPI program koji realizuje množenje matrice A_{nxn} i vektora b_n, čime se dobija rezultujući vektor c_n. Matrica A i vektor b se inicijalizuju u master procesu. Broj procesa je p i uređeni su kao matrica qxq (q^2=p). Matrica A je podeljena u podmatrice dimenzika kxk (k=n/q) i master proces distribuira odgovarajuće blokove matrica A po procesima kao što je prikazano na Slici 1. za n=8 i p=16. Vektor b se distribuira u delovima od po n/q elemenata, tako da nakon slanja procesi u prvoj koloni matrice procesa sadrže prvih n/q elemenata, u 2. koloni matrice procesa sledećih n/q elemenata itd..Na osnovu primljenih podataka svaki proces obavlja odgovarajuća izračunavanja i učestvuje u generisanju rezultata koji se prikazuje u master procesu. Predvideti da se slanje podmatrica matrice A svakom procesu obavlja sa po jednom naredbom MPI_Send kojom se šalje samo 1 izvedeni tip podatka. Slanje blokova vektora b i generisanje rezultata implementirati korišćenjem grupnih operacija i funkcija za kreiranje novih komunikatora.

(0,0) (0,1) **P0** (1,0) (1,1)	(0,2) (0,3) **P1** (1,2) (1,3)	(0,4) (0,5) **P2** (1,4) (1,5)	(0,6) (0,7) **P3** (1,6) (1,7)
P4	**P5**	**P6**	**P7**
P8	**P9**	**P10**	**P11**
(6,0) (6,1) **P12** (7,0) (7.1)	(6,2) (6,3) **P13** (7,2) (7,3)	(6,4) (6,5) **P14** (7,4) (7,5)	(6,6) (6,7) **P15** (7,6) (7,7)



-P0, P4,P8,P12

-P1, P5,P9,P13

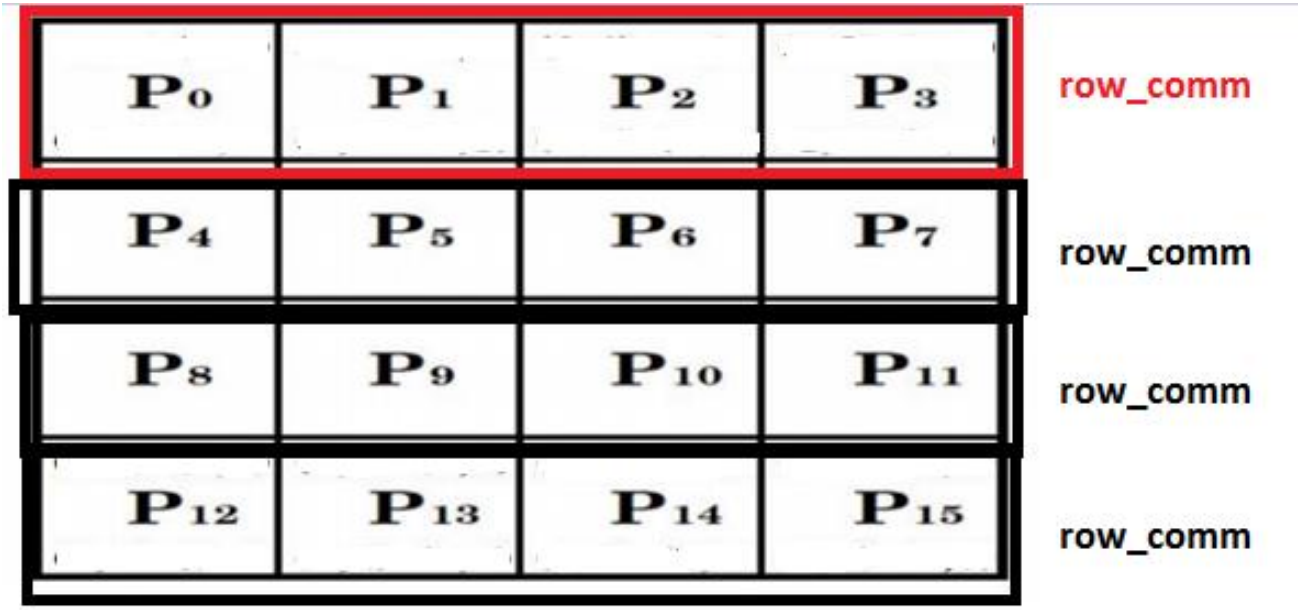
-P2, P6,P10,P14

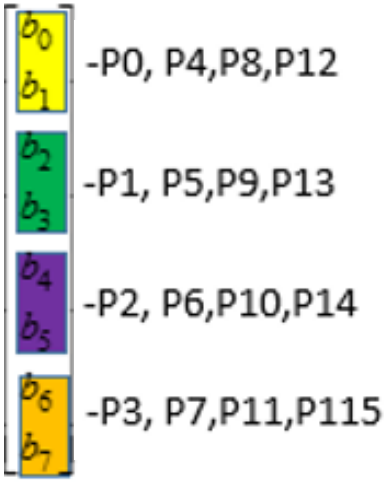
-P3, P7,P11,P115

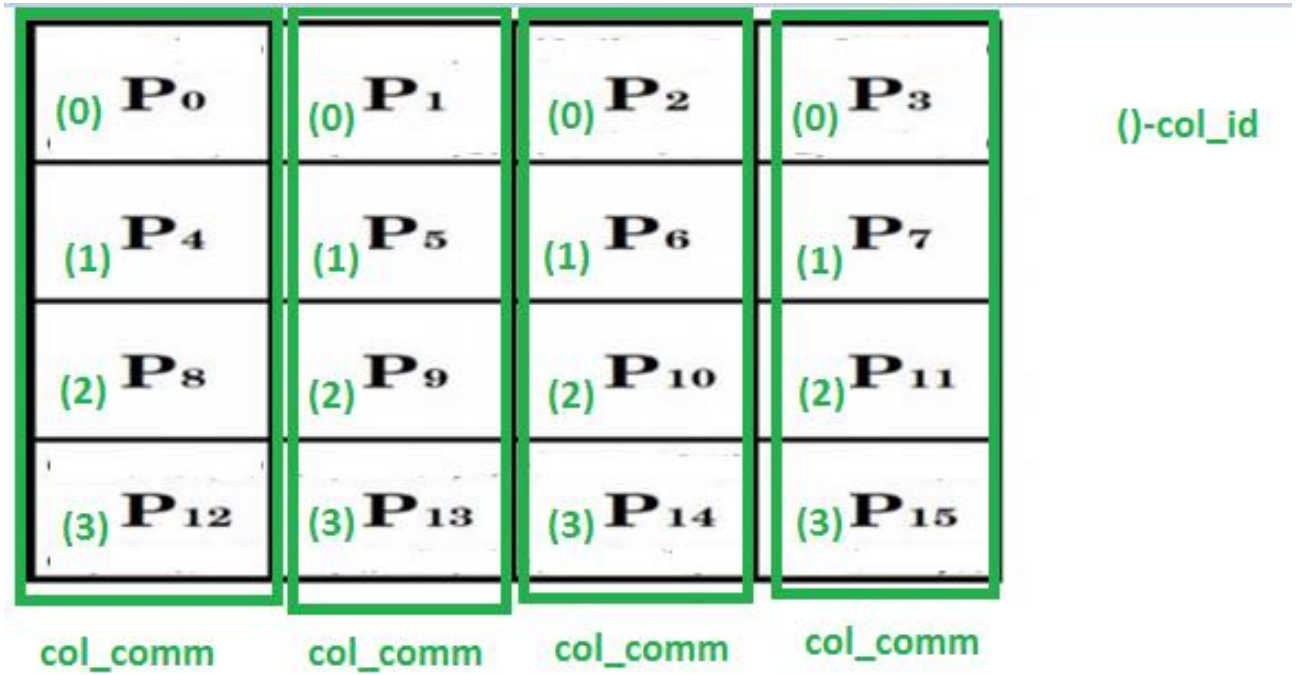
```
#define n 6
void main(int argc, char *argv[])
{
        int irow, jcol, p,i,j,k,q,l,y=0;
        MPI_Comm row_comm, col_comm, com;
        int rank, row_id, col_id;
        int a[n][n],b[n],c[n];
        MPI_Datatype  vrblok;
        MPI_Status status;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &p);
        q=(int)sqrt((double)p);
        k=n/q;
        int* local_a=(int*)calloc(k*k,sizeof(int));
        int* local_b=(int*)calloc(k,sizeof(int));
        MPI_Type_vector(k,k,n,MPI_INT,&vrblok);
        MPI_Type_commit(&vrblok);
```

```
if (rank == 0)
        {
       for(i = 0; i < n; i++)
       {
           for(j = 0; j < n; j++)
           {
               a[i][j] = i+j;

           }
       }

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


               b[i] = 1;


        }
```

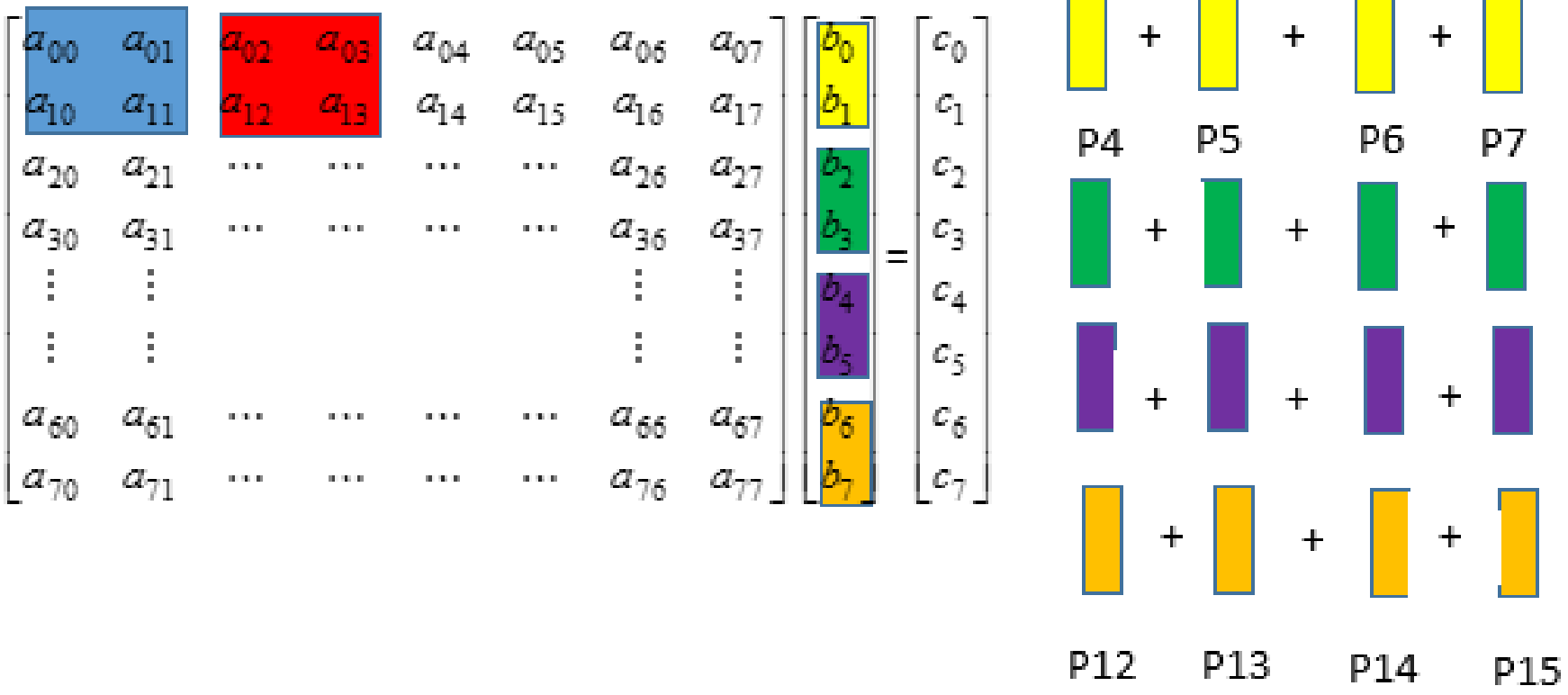
```
if (rank==0)
{
        for (i = 0; i< k; i++)

          for (j = 0; j< k; j++)

                    local_a[y++]=a[i][j];
        l=1;
        for (j=0;j<q;j++)
                    for (i=0;i<q;i++)
                         if ((i+j)!=0)

                          MPI_Send(&a[j*k][k*i],1,vrblok,l,2,MPI_COMM_WORLD);
                                        l++;
}
else

        MPI_Recv(local_a,k*k,MPI_INT,0,2,MPI_COMM_WORLD, &status);
```


P0 P1 P2 P3
P4 P5 P6 P7
P8 P9 P10 P11
P12 P13 P14 P15
row_comm
row_comm
row_comm
row_comm


b0 b1
-P0, P4,P8,P12
b2 b3
-P1, P5,P9,P13
b4 b5
-P2, P6,P10,P14
b6 b7
-P3, P7,P11,P115


(0) P0 (0) P1 (0) P2 (0) P3
(1) P4 (1) P5 (1) P6 (1) P7
(2) P8 (2) P9 (2) P10 (2) P11
(3) P12 (3) P13 (3) P14 (3) P15
col_comm col_comm col_comm col_comm
()-col_id

```
irow = rank/q;
 jcol = rank%q;
 com =MPI_COMM_WORLD;

 MPI_Comm_split(com, irow, jcol, &row_comm);
 MPI_Comm_split(com, jcol, irow, &col_comm);
 MPI_Comm_rank(row_comm, &row_id);
 MPI_Comm_rank(col_comm, &col_id);

 if (col_id==0)
            MPI_Scatter(b, k, MPI_INT, local_b,k, MPI_INT, 0,row_comm);

 MPI_Bcast(local_b,k, MPI_INT, 0, col_comm);
 int* MyResult   = (int*) malloc(k * sizeof(int));
 int* Result   = (int*) malloc(n * sizeof(int));
 int index = 0;
```


P0
P1
P2
P3
P4
P5
P6
P7
P12
P13
P14
P15

```
 for(i=0; i < k; i++){

      MyResult[i]=0;

      for(j=0;j<k; j++)

          MyResult[i] += local_a[index++] * local_b[j];
}
MPI_Gather (MyResult,k, MPI_INT, Result, k, MPI_INT, 0, col_comm);
if (col_id==0)
        MPI_Reduce(Result,c,n,MPI_INT, MPI_SUM,0, row_comm);

if (rank==0)
        for (i = 0; i< n; i++)
       {
          printf("c[%d]=%d ",i,c[i]);
         }
        MPI_Finalize();
}
```

OpenMP

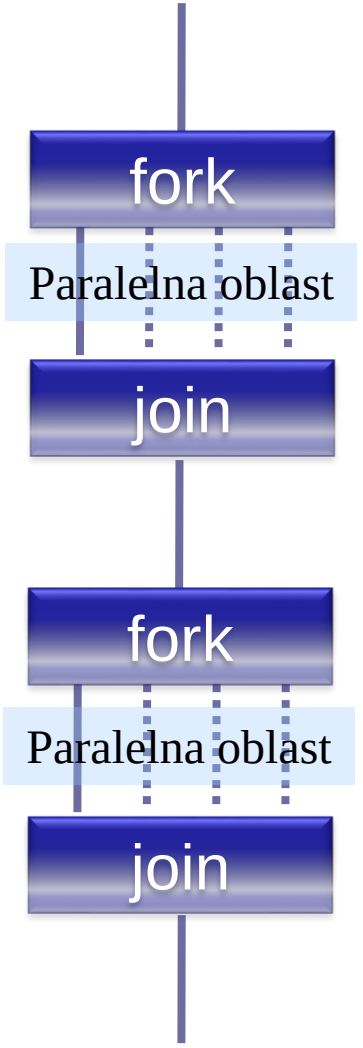
Open MultiProcessing

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 sarađuje 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 parallelnog 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

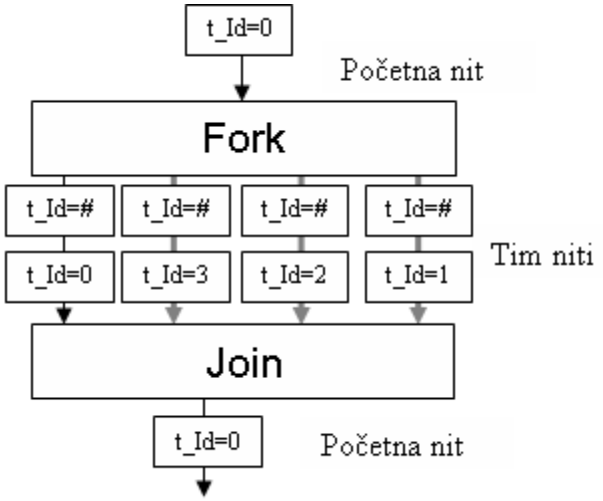
- **If odredba-** Ako uslov u if odredbi nije ispunjen kod se izvršava sekvencijalno
- **Private** odredba-služi za navođenje privatnih promenljivih za svaku nit(svaka nit ima svoju kopiju privatne promenljive koja nije inicijalizovana na ulasku u paralelni region).

Odredbe parallel direktive

- Po defaultu su **private**:
  - indeksna promenljiva for petlje koja se paralelizuje
  - promenljive koje su deklarisane unutar strukturnog bloka paralelnog regiona
  - promenljive koje su deklarisane 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

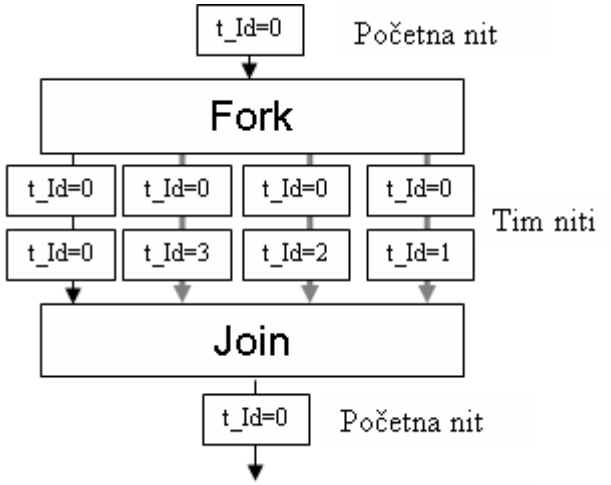
Primer. Private odredba

```
…
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 deklarisane 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:

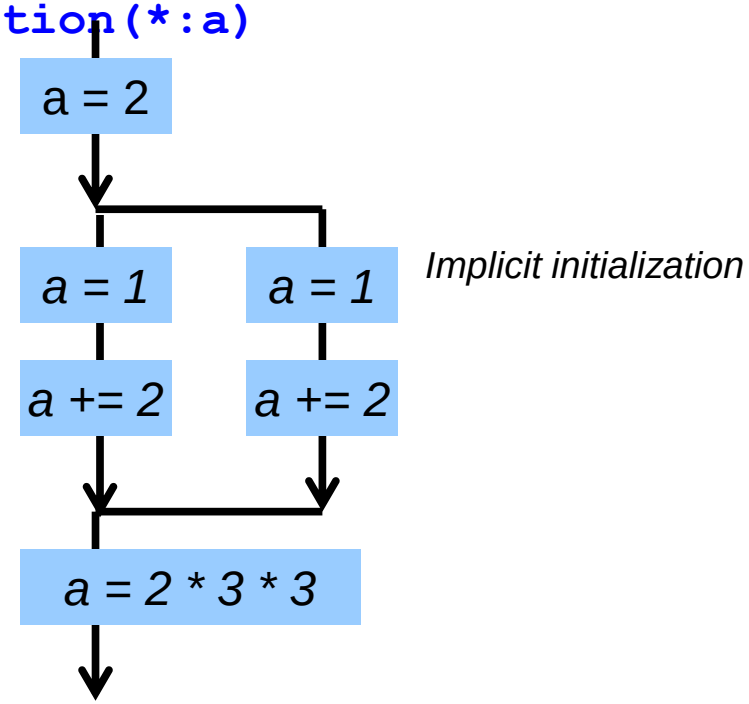
  reduction(*op*:*lista_promenljivih*)
- Kreira se privatna kopija za svaku promenljivu koja se nalazi u *listi_promenljivih*, po jedna za svaku nit, i inicijalizuje u zavisnosti od operatora *op(npr. 0 for “+”).*
- Na kraju oblasti, originalna promenljiva se ažurira kombinacijom svoje originalne vrednosti 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);

}
```

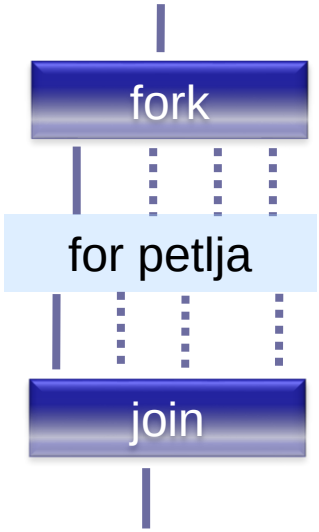


Direktive za podelu poslova

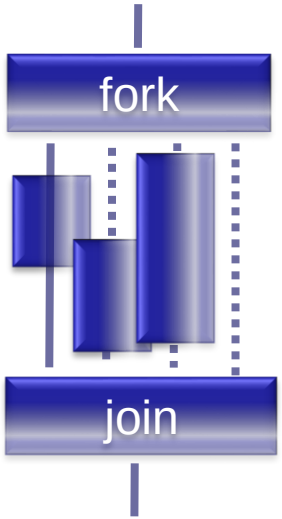
- Direktive za podelu poslova raspoređuju izvršenje određenog dela koda između niti u timu. One ne kreiraju nove niti i ne podrazumevaju postojanje naredbe ***barrier*** pre izvršenja direktive, ali podrazumevaju na kraju. Open MP definiše sledeće direktive za podelu poslova.
  - ***for*** direktiva (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

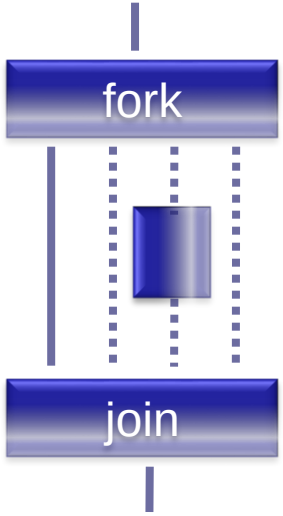
Korišćenje direktive `for`



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

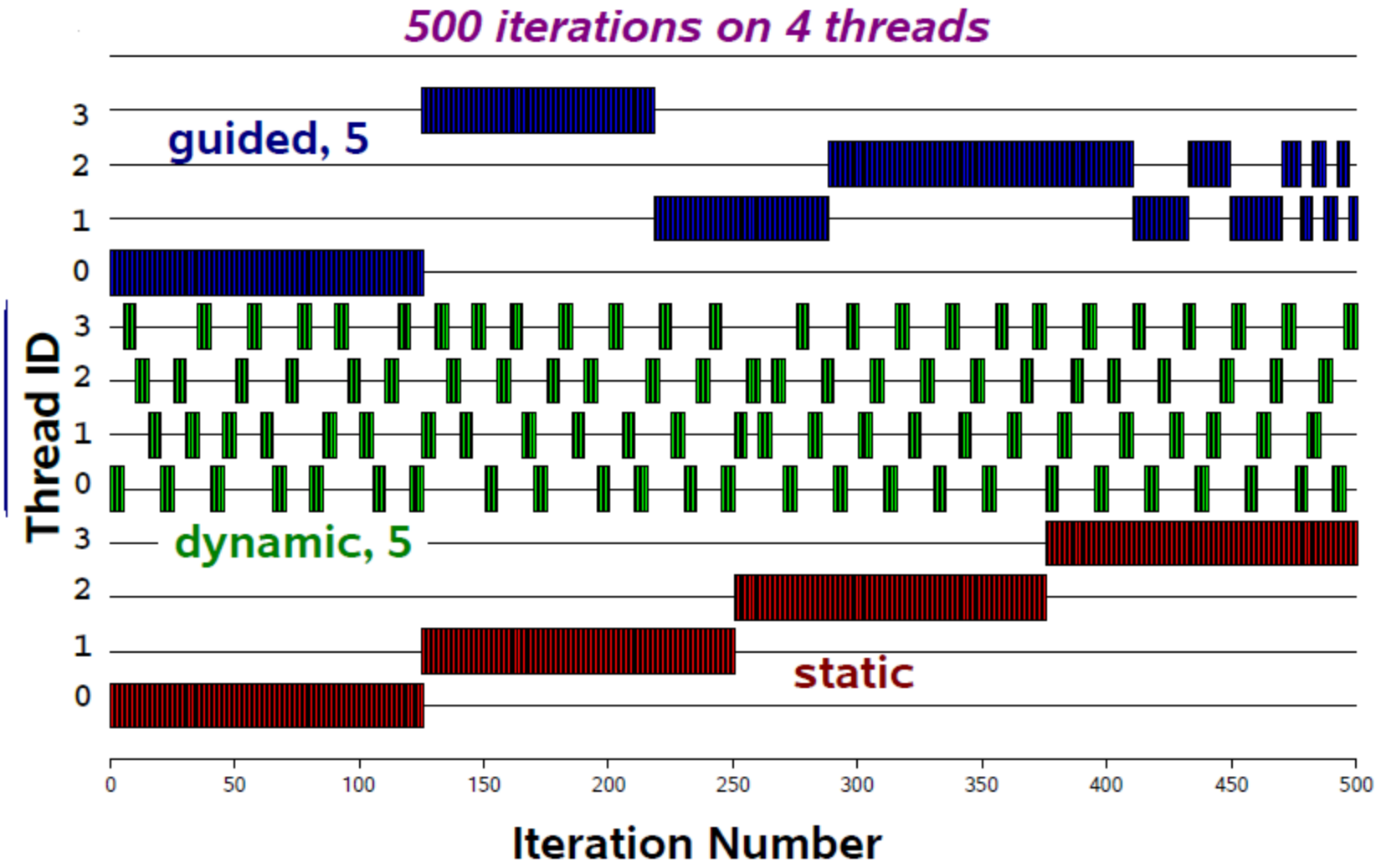
Odredba schedule (*dynamic*)

- *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 $k_i=(1-1/p)* k_{i-1}$
  - 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



Odredba nowait

- Ukida podrazumevanu barijeru na kraju for direktive

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 (kritčnu 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;

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);
}
```

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!)