## Porazdeljeni sistemi

# 4. Nitenje z OpenMP

Predavatelj: izr. prof. Uroš Lotrič
Asistent: Davor Sluga

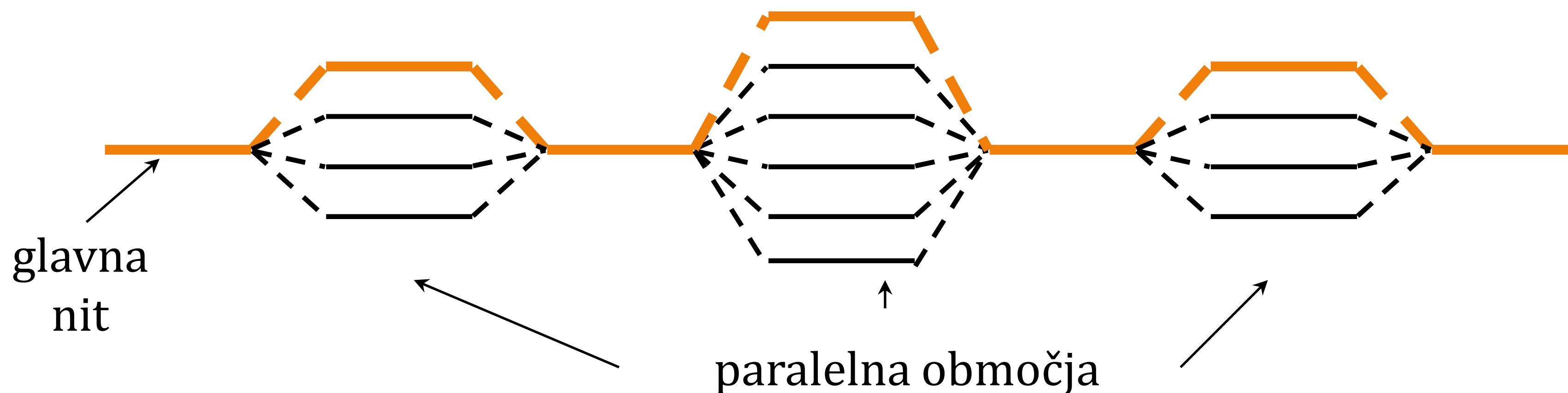
# OpenMP

- Niti malo drugače.
- OpenMP je programski vmesnik za pisanje večnitnih programov, ki vključuje:
  - množico ukazov za prevajalnike, s katerimi opišemo paralelizem v programih,
  - manjši nabor funkcij in
  - nekaj spremenljivk.
- OpenMP zelo poenostavi pisanje paralelnih programov v jezikih C in Fortran
- Uvaja standardiziran pristop k pisanju večnitnih programov

OpenMP™

# Pregled – delo z nitmi

- Pristop s cepitvijo in združevanjem niti (ang. fork – join model)
  - Glavna nit ustvari množico niti
  - Te se na koncu pridružijo glavni niti



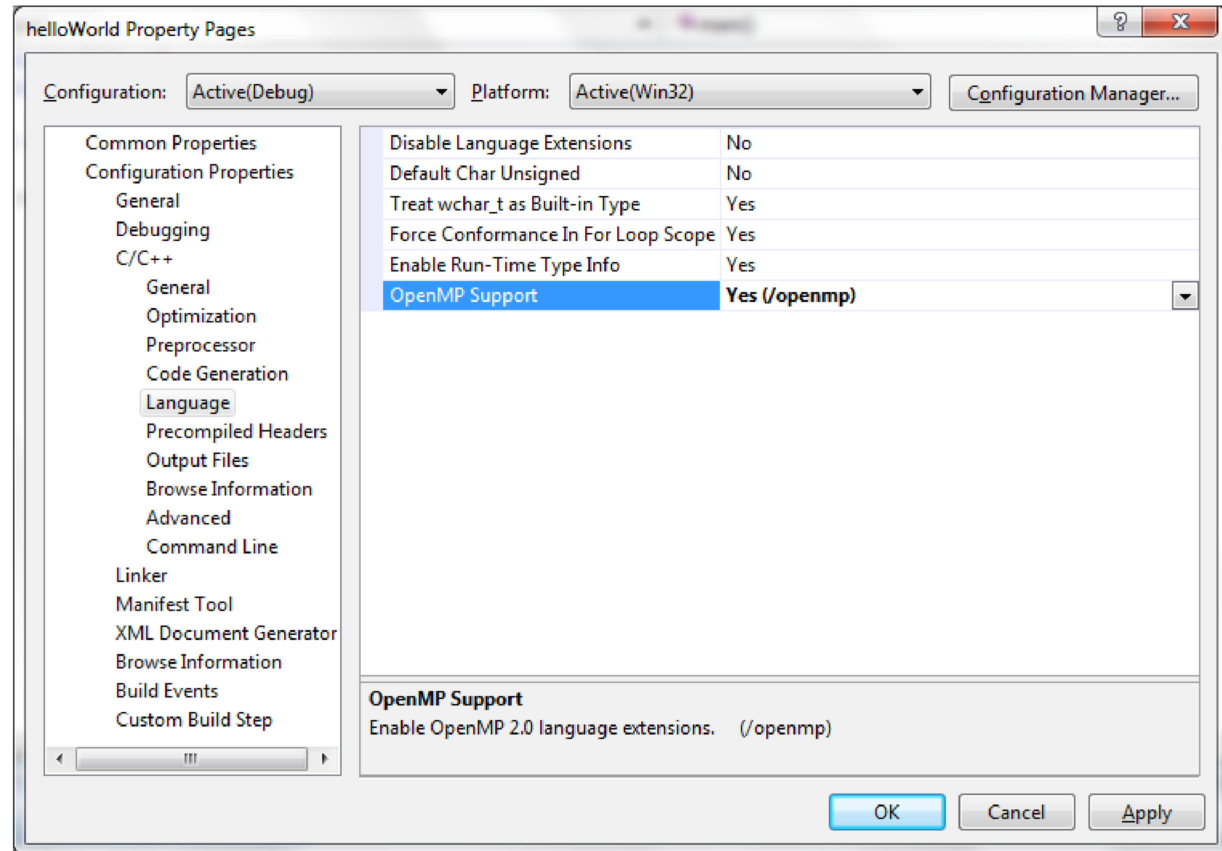

  - Z OpenMP lahko program paraleliziramo postopno
    - Program korak po korak iz sekvenčnega spreminjamo v paralelnega

# Pregled – kako niti sodelujejo?

- OpenMP je namenjen za sisteme z deljenim pomnilnikom – niti komunicirajo preko deljenih spremenljivk
- Pozor: nepravilna uporaba deljenih spremenljivk lahko pripelje do okoliščin, v katerih je rezultat odvisen od vrstnega reda izvajanja niti!!!
- Zgornje težave v OpenMP lahko rešimo z medsebojnim izključevanjem ali sinhronizacijo, ki pa sta časovno potratni operaciji.

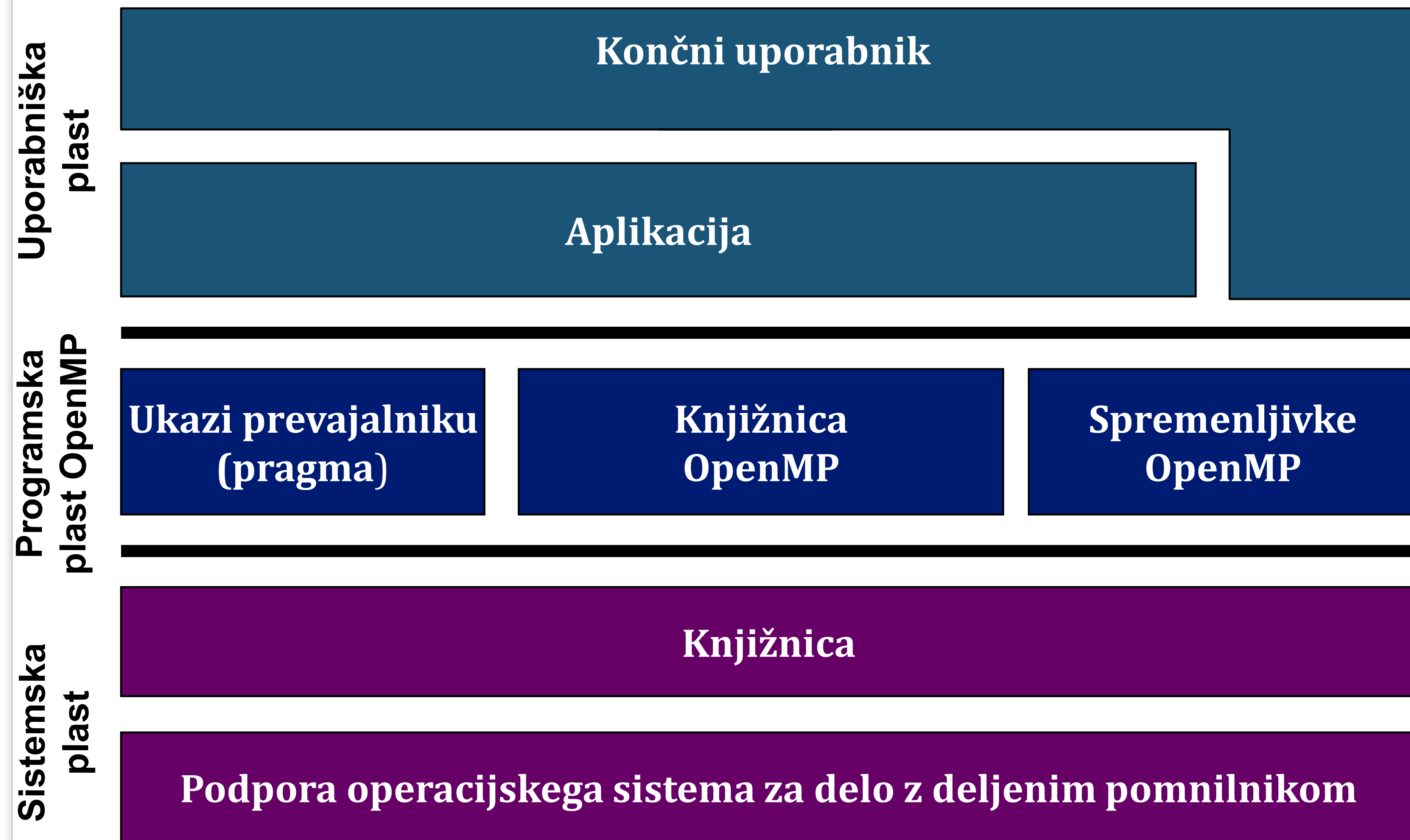
# Pregled - prevajanje

- Nastavitve
  - MS Visual Studio 2010:




  - GCC
    - gcc –fopenmp –g –o prg prg.c

# Arhitektura



# Sintaksa

- Večino programskih konstruktov v OpenMP predstavljajo ukazi prevajalniku
  **#pragma omp** …

- Ukazi prevajalniku
  (ang. pragmatic information, pragma) se lahko raztezajo čez več datotek izvorne kode.

- Pred prevajanjem v zaglavje dodamo
  **#include "omp.h"**

# Kaj lahko paraleliziramo?

- OpenMP je namenjen za paralelizacijo programskih blokov
- **Programski blok** je en ali več stavkov med zavitimi oklepaji { }

```
#pragma omp parallel
{
   res[id] = lots_of_work(id);
}
```

```
#pragma omp for
  for(i=0; i<N; i++)
  {
       res[i] = big_calc(i);
       a[i] = b[i] + res[i];
  }
```

# Kaj lahko paraleliziramo?

- Programski bloki morajo biti **strukturirani** - imajo eno vstopno točko na začetku in eno izstopno točko na koncu.
- Edina dovoljena vejitev je klic funkcije exit().

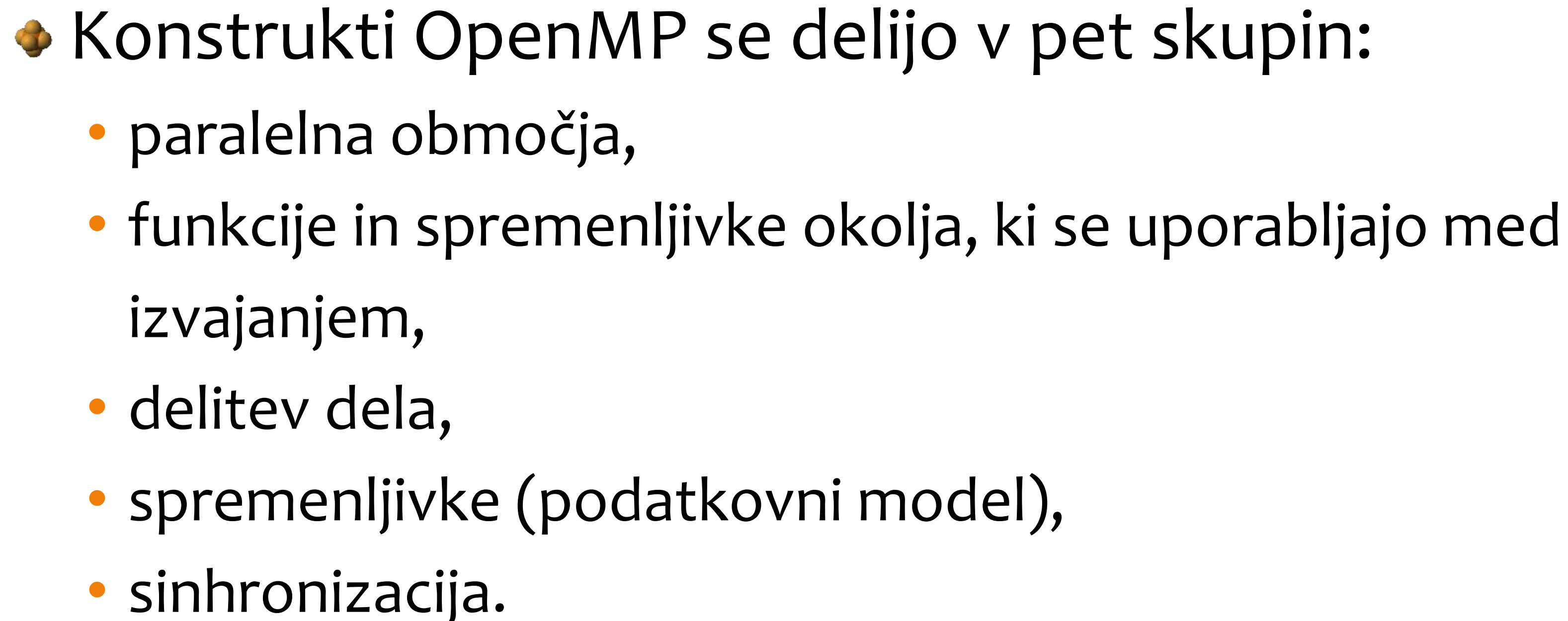
```
#pragma omp parallel
{
        int id = omp_get_thread_num();
more:  res(id) = do_big_job(id);
        if(conv(res(id)) goto more;
}
printf(" All done \n");
```

**strukturirani blok**

```
if(go_now()) goto more;
#pragma omp parallel
{
        int id = omp_get_thread_num();
more:  res(id) = do_big_job(id);
        if(conv(res(id)) goto done;
        go to more;
}
done:   if(!really_done()) goto more;
```

**nestrukturirani blok**

# Konstrukti OpenMP



- Konstrukti OpenMP se delijo v pet skupin:
  - paralelna območja,
  - funkcije in spremenljivke okolja, ki se uporabljajo med izvajanjem,
  - delitev dela,
  - spremenljivke (podatkovni model),
  - sinhronizacija.

# Funkcije

- Funkcije, s katerimi med izvajanjem programa izvemo kaj o okolju
  - Število niti
    - **omp_set_num_threads(),**
    - **omp_get_num_threads(),**
    - **omp_get_thread_num(),**
    - **omp_get_max_threads()**
  - Smo v paralelnem območju?
    - **omp_in_parallel()**
  - Število procesorjev v sistemu
    - **omp_get_num_procs()**
  - Merjenje časa
    - **omp_get_wtime()**
    - **omp_get_wtick()**

# Paralelna območja

- Ukaz **omp parallel**
  - Niti ustvarjamo z ukazom prevajalniku
    **#pragma omp parallel**
  - To je navodilo prevajalniku, da morajo programski blok, ki ga označuje ta ukaz prevajalniku, izvesti vse niti

# Paralelna območja

- Ukaz **omp parallel**
  - Primer programa
    - Sekvenčno

```
#include “omp.h”
void main()
{

    int ID = omp_get_thread_num();
    printf(“hello(%d) ”, ID);
    printf(“world(%d) \n”, ID);
}
```

**Primer izpisa:**

hello(0) world(0)

# Paralelna območja

- Ukaz **omp parallel**
  - Primer programa
    - Paralelno

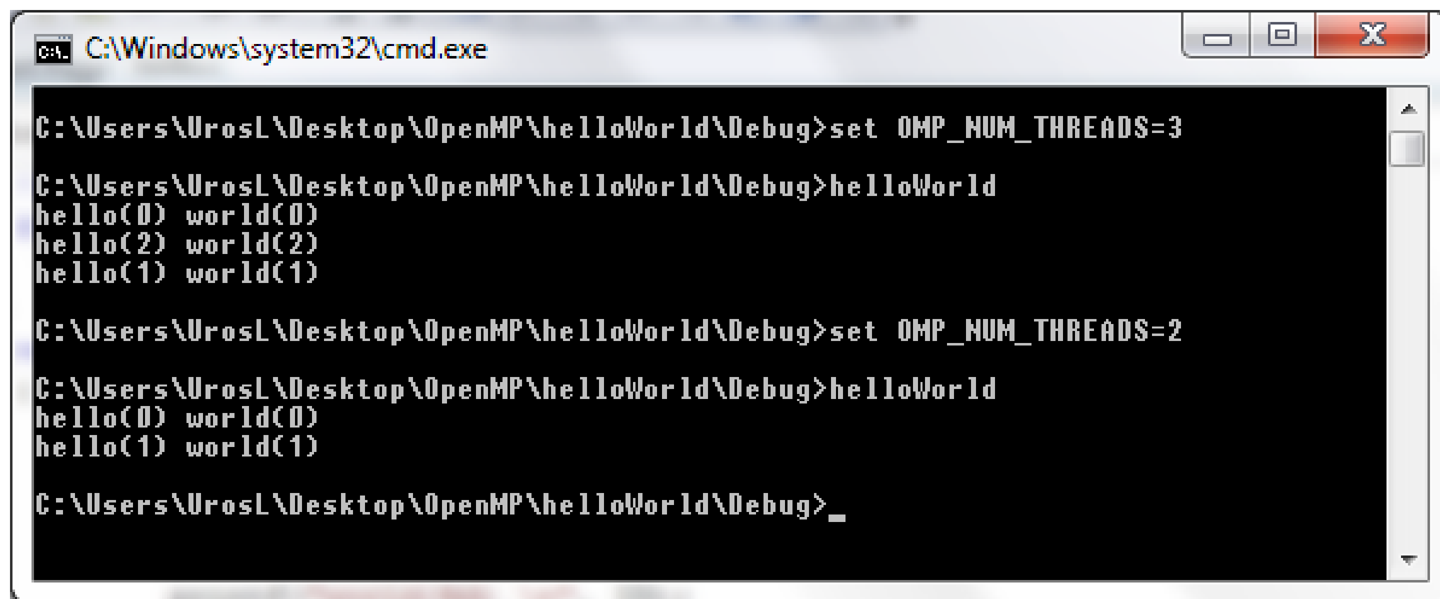
```
#include "omp.h"
void main()
{
    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        printf("hello(%d) ", ID);
        printf("world(%d) \n", ID);
    }
}
```

**Primer izpisa:**

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

# Spremenljivke okolja

- Spremenljivke, s katerimi lahko pred zagonom programa vplivamo na način izvajanja
  - Primer:
    - število niti: **OMP_NUM_THREADS**




    - način razvrščanja dela na niti: **OMP_SCHEDULE**

# Delitev dela

- Ukaza **omp sections** in **omp section**
  - Omogočata nam, da paralelno izvajamo več različnih blokov
  - Bloki se lahko razlikujejo tudi po funkciji
  - Na koncu **omp sections** je postavljena prepreka
  - Namesto ukaza **omp sections** lahko uporabimo tudi ukaz **omp parallel sections**

```
#pragma omp parallel
#pragma omp sections
{
   #pragma omp section
   v = alpha();
   #pragma omp section
   w = beta();
   #pragma omp section
   y = delta();
}
```

# Delitev dela

- Ukaza **omp sections** in **omp section**
  - Primer "hello world" malo drugače
    - Ena nit izpiše hello, druga pa world
    - Primer izpisa:

```
hello(0) world(1)
```

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

void main(void)
{
#pragma omp parallel
    {
#pragma omp sections
        {
#pragma omp section
            {
            int ID = omp_get_thread_num();
            printf("hello(%d) ", ID);
            }
#pragma omp section
            {
            int ID = omp_get_thread_num();
            printf("world(%d) ", ID);
            }
        }
    }
}
```

# Delitev dela

- Dopolnilo **nowait**
  - Prevajalnik za vsakim programskim blokom označenim z ukazom **sections** doda prepreko.
  - Niti pot nadaljujejo šele potem, ko do prepreke pride tudi zadnja med njimi.
  - Z dodatkom **nowait** dovolimo, da program nadaljuje izvajanje še preden se v paralelnem bloku končajo vse niti.

# Delitev dela

- Dopolnilo **nowait**
  - Primer
    - brez nowait

```
hello world HELLO WORLD
```

    - z nowait

```
hello HELLO WORLD world
```

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

void main(void)
{
    #pragma omp parallel
    {
        #pragma omp sections nowait
        {
            #pragma omp section
            {
                printf("hello ");
            }
            #pragma omp section
            {
                printf("world ");
            }
        }
        #pragma omp sections nowait
        {
            #pragma omp section
            {
                printf("HELLO ");
            }
            #pragma omp section
            {
                printf("WORLD ");
            }
        }
    }
}
```

# Delitev dela

- Ukaz **omp for**
  - razdeli iteracije v zanki for na več niti
  - Zaradi pogoste uporabe obstaja tudi ukaz **omp parallel for**

```
#pragma omp parallel
#pragma omp for
for (i=0; i<N; i++)
{
    hardWork(i);
}
```

```
#pragma omp parallel for
for (i=0; i<N; i++)
{
    hardWork(i);
}
```

  - Ob koncu bloka označenega z **omp for** je prepreka.
  - Izvajanje programa se nadaljuje za prepreko šele potem, ko vse niti pridejo do prepreke.

# Delitev dela

- Ukaz **omp for**
  - Prevajalnik mora iz zapisa zanke določiti število iteracij.
  - Zapis zanke mora biti zato podan v enostavni (kanonični) obliki

```
                                        index++
                                        ++index
                            <           index--
                            <=          --index
for (index = start; index   >=   end;   index += inc             )
                            >           index -= inc
                                        index = index + inc
                                        index = inc + index
                                        index = index - inc
```

# Delitev dela

- Ukaz **omp for**
  - Poenostavitev programa z ukazom **omp parallel for**

Sekvenčni program

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

Program s konstruktom parallel

```
#pragma omp parallel
{
   id = omp_get_thread_num();
   Nthrds = omp_get_num_threads();
   istart = id * N / Nthrds;
   iend = (id+1) * N / Nthrds;
   for(i=istart; i<iend; i++)   {a[i] = a[i] + b[i];}
}
```

Program s konstruktom parallel for

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

# Delitev dela

- Ukaz **omp for**
  - Primer: števec
  - Z eno nitjo dobimo pravilen rezultat.
  - Z vsaj dvema nitma dobimo napačen rezultat

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

#define N 10000000
int counter = 0;

void main(void)
{
    int i;

    #pragma omp parallel for
    for(i=0; i<N; i++)
        counter++;

    printf("Counter = %d\n", counter);
}
```

# Delitev dela

- Dopolnilo **schedule**
  - Z njim povemo, kako naj se iteracije v zanki for razvrstijo med različne niti
  - **schedule(static [, število])**
    - Vsaka nit dobi *število* zaporednih iteracij
    - Smiselno uporabiti, kadar je količina dela predvidljiva in je delo na posamezno niti časovno približno enako zahtevno
    - Privzet način
  - **schedule(dynamic[, število])**
    - Vsaka nit vzame v svojo vrsto *število* zaporednih iteracij. Ko jih izvede, vzame število novih iteracij. Postopek se nadaljuje dokler se zanka ne izvede v celoti.
    - Primerno, kadar je količino dela težko oceniti, čas izvajanja niti pa se močno razlikuje

# Delitev dela

- Dopolnilo **schedule**
  - **schedule(guided[, število])**
    - Podobno kot dynamic, le da so bloki na začetku večji, nato pa se zmanjšujejo do velikosti število.
    - Smiselno uporabiti namesto dynamic, da zmanjšamo delo, potrebno za razvrščanje opravil
  - **schedule(runtime)**
    - Uporabimo, če uporabniku dovolimo da preko spremenljivke okolja **OMP_SCHEDULE** določa način dodeljevanja niti

# Delitev dela

- Dopolnilo **schedule**
  - Primer: razlika med dinamičnim in statičnim prirejanjem

static, 1

```
Iteracija 1 koncana (1).
Iteracija 3 koncana (1).
Iteracija 5 koncana (1).
Iteracija 7 koncana (1).
Iteracija 9 koncana (1).
Iteracija 0 koncana (0).
Iteracija 2 koncana (0).
Iteracija 4 koncana (0).
Iteracija 6 koncana (0).
Iteracija 8 koncana (0).
```

static, 2

```
Iteracija 2 koncana (1).
Iteracija 3 koncana (1).
Iteracija 6 koncana (1).
Iteracija 7 koncana (1).
Iteracija 0 koncana (0).
Iteracija 1 koncana (0).
Iteracija 4 koncana (0).
Iteracija 5 koncana (0).
Iteracija 8 koncana (0).
Iteracija 9 koncana (0).
```

```
Iteracija 1 koncana (1).
Iteracija 2 koncana (1).
Iteracija 3 koncana (1).
Iteracija 4 koncana (1).
Iteracija 5 koncana (1).
Iteracija 6 koncana (1).
Iteracija 7 koncana (1).
Iteracija 8 koncana (1).
Iteracija 9 koncana (1).
Iteracija 0 koncana (0).
```

dynamic, 1

```
Iteracija 2 koncana (1).
Iteracija 3 koncana (1).
Iteracija 4 koncana (1).
Iteracija 5 koncana (1).
Iteracija 6 koncana (1).
Iteracija 7 koncana (1).
Iteracija 8 koncana (1).
Iteracija 9 koncana (1).
Iteracija 0 koncana (0).
Iteracija 1 koncana (0).
```

dynamic, 2

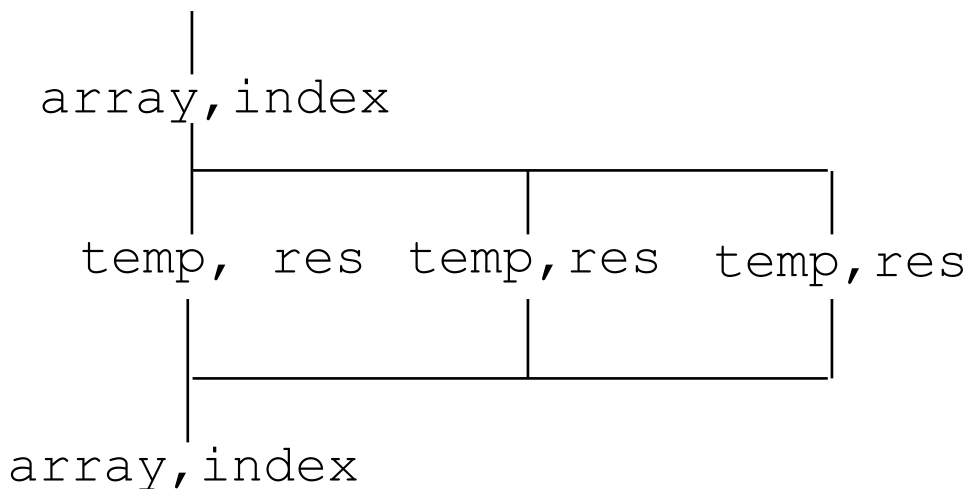
```
#include "omp.h"
#include <stdio.h>
#include <windows.h>

void main(void)
{
    int i;

    #pragma omp parallel for schedule(dynamic, 1)
    for(i=0; i<10; i++)
    {
        int id = omp_get_thread_num();
        if(i==0)
            Sleep(1000);
        printf("Iteracija %i koncana (%d).\n", i, id);
    }
}
```

# Spremenljivke

- Spremenljivke si delijo skupni pomnilnik
  - večina spremenljivk je deljenih
- Niti si delijo globalne spremenljivke
- Niti si ne delijo
  - spremenljivk, ki so deklarirane v programskih blokih paralelnih območij
  - spremenljivk podprogramov, ki so klicani iz paralelnih območij; te so lokalne za vsako nit

# Spremenljivke

- Globalne in lokalne spremenljivke
  - Spremenljivki *array* in *index* si delijo vse niti
  - Spremenljivk *res* in *temp* si niti ne delijo




```
int do_some_work(int i)
{
   double temp;
   ...
}

double array[10];

void main(void)
{
   int index;

   #pragma omp parallel
   {
      int res = do_some_work(index);
   }
}
```

# Spremenljivke

- Običajne lastnosti spremenljivk lahko spremenimo z dopolnili
  - **shared**
  - **private**
  - **firstprivate**
  - **lastprivate**
  - **threadprivate**

# Spremenljivke

- Dopolnilo **shared**(*vars*)
  - Z njim poudarimo, da si niti delijo globalne spremenljivke *vars*
  - To je privzeta nastavitev, zato dopolnila **shared** ni potrebno uporabljati

# Spremenljivke

- Dopolnilo **private**(*vars*) naredi za vsako nit svojo kopijo spremenljivk *vars*
  - Spremenljivke *vars* v nitih niso inicializirane
  - Kopije v pomnilniku niso povezane z originalnimi spremenljivkami

```
var = 0;
#pragma omp parallel for private(var)
for(i=0; i<10; i++)
{
   var = var + i;
}
fprintf("%d", var);
```

ne glede na dogajanje v paralelnem območju je vrednost spremenljivke *var* enaka nič

spremenljivka *var* ni inicializirana

# Spremenljivke

- Dopolnilo **private**(vars)
  - Primer: vgnezdena zanka
  - Izpis:

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

void main(void)
{
    int i, j;

    #pragma omp parallel for private(j)
    for( i=0; i<2; i++)
        for(j=0; j<10; j++)
            printf("%d-%d\n", i, j);
}
```

brez private(j)

```
1-0
1-1
1-2
1-3
0-0
0-5
1-4
1-7
1-8
1-9
0-6
```

s private(j)

```
0-0
0-1
0-2
1-0
1-1
1-2
0-3
0-4
1-3
1-4
1-5
1-6
0-5
0-6
1-7
1-8
1-9
0-7
0-8
0-9
```

# Spremenljivke

- Dopolnilo **firstprivate**(*vars*) naredi za vsako nit svojo kopijo spremenljivke *vars*
  - Spremenljivke *vars* v nitih dobijo enako vrednost kot globalne spremenljivke *vars* pred vstopom v paralelno območje
  - Kopije v pomnilniku niso povezane z originali

```
var = 0;
#pragma omp parallel for firstprivate(var)
for(i=0; i<10; i++)
{
   var = var + i;
}
fprintf("%d", var);
```

ne glede na dogajanje v paralelnem območju je vrednost spremenljivke *var* enaka nič

# Spremenljivke

- Dopolnilo **lastprivate**(*vars*) prenese vrednosti spremenljivk *vars* iz zadnje iteracije v globalne spremenljivke *vars*

```
var = 0;
#pragma omp parallel for firstprivate(var) lastprivate(var)
for(i=0; i<10; i++)
{
   var = var + i;
}
fprintf("%d", var);
```

- Pri dveh nitih program izpiše 5+6+7+8+9=35

# Spremenljivke

- Primer

```
a = b = c = 1;
#pragma omp parallel private(b) firstprivate(c)
{
    ....
}
...
```

- spremenljivka *a* je vidna vsem nitim, njena vrednost je 1
- spremenljivki *b* in *c* sta lokalni za vsako nit
  - vrednost spremenljivke *b* v paralelnem območju ni definirana
  - spremenljivka *c* ima ob vstopu v paralelno območje vrednost 1
- za paralelnim območjem
  - Spremenljivki *b* in *c* imata vrednost 1

# Spremenljivke

- Dopolnilo **threadprivate**(*var*) naredi spremenljivke lokalne za vsako nit
  - spremenljivke, označene z dopolnilom threadprivate, se po zaključku paralelnega območja ne sprostijo
  - v primeru, da z enakim številom niti ponovno vstopimo v paralelno območje spremenljivk, označenih s threadprivate, jih ni potrebno še enkrat inicializirati
  - Spremenljivke iniciliziramo z ukazom **copyin**
  - Primerno za uporabo lokalnih kopij večjih struktur

# Spremenljivke

- Primer: dopolnili threadprivate in firstprivate
  - Dve niti
  - Izpis

```
K0:: N0: FP=  0, TP=  0 | N1: FP=  0, TP=  0 |
K1:: N0: FP=  1, TP=  1 | N1: FP=  2, TP=  2 |
K2:: N0: FP=  0, TP=  1 | N1: FP=  0, TP=  0 |
K3:: N0: FP= 10, TP= 11 | N1: FP=  0, TP=  0 |
K4:: N0: FP=110, TP=111 | N1: FP=210, TP=202 |
K5:: N0: FP= 10, TP=111 | N1: FP=  0, TP=  0 |
```

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

int log[6][2][2] = {{0}};
int varFP = 0;
int varTP = 0;
#pragma omp threadprivate(varTP)

void main(void)
{
    int i,j, tn;

    int Tn = omp_get_thread_num();
    log[0][Tn][0] = varFP; log[0][Tn][1] = varTP;
    #pragma omp parallel copyin(varTP) firstprivate(varFP)
    {
        int tn = omp_get_thread_num();
        varFP = varFP + (tn+1); varTP = varTP + (tn+1);
        log[1][tn][0] = varFP; log[1][tn][1] = varTP;
    }
    log[2][Tn][0] = varFP; log[2][Tn][1] = varTP;
    varFP = varFP + 10*(Tn+1); varTP = varTP + 10*(Tn+1);
    log[3][Tn][0] = varFP; log[3][Tn][1] = varTP;
    #pragma omp parallel firstprivate(varFP)
    {
        int tn = omp_get_thread_num();
        varFP = varFP + 100*(tn+1); varTP = varTP + 100*(tn+1);
        log[4][tn][0] = varFP; log[4][tn][1] = varTP;
    }
    log[5][Tn][0] = varFP; log[5][Tn][1] = varTP;

    for(i=0; i<6; i++)
    {
        printf("K%d:: ",i);
        for(j=0; j<2; j++)
            printf("N%d: FP=%3d, TP=%3d | ", j, log[i][j][0], log[i][j][1]);
        printf("\n");
    }
}
```

# Spremenljivke

- Dopolnilo **reduction**(*op:var*) je poseben način deljenja spremenljivk med niti
  - vsaka nit dobi svojo lokalno kopijo globalne spremenljivke *var*
  - lokalne kopije so inicializirane glede na tip redukcije *op*
  - nazadnje so lokalne kopije zbrane v skupno globalno spremenljivko
  - operacije redukcije

| Operand | + | * | & | \| | ^ | && | \|\| |
|---|---|---|---|---|---|---|---|
| Opis | sešte vanje | množe nje | bit in | bit ali | bit XOR | logični in | logični ali |
| Začetna vrednost | 0 | 1 | vsi biti 1 | vsi biti 0 | vsi biti 0 | 1 (true) | 0 (false) |

# Spremenljivke

- Dopolnilo **reduction**(*op*:*var*)
  - Primer
    - Rezultat je zdaj pravilen!

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

#define N 10000000
int counter = 0;

void main(void)
{
    int i;

    #pragma omp parallel for reduction(+:counter)
    for(i=0; i<N; i++)
        counter++;

    printf("Counter = %d\n", counter);
}
```

# Sinhronizacija

- Konstrukti OpenMP, s katerimi lahko sinhroniziramo niti
  - critical
  - atomic
  - barrier
  - ordered
  - single
  - master
  - flush

# Sinhronizacija

- Ukaz **critical**(*name*)
  - Samo ena nit je lahko hkrati v območju, ki je označeno s **critical**
  - Območje po potrebi lahko poimenujemo (*name*)
  - Dve različni niti lahko izvajata kodo v drugače poimenovanih kritičnih območjih
  - Primer: štetje
    - samo ena nit na enkrat je v območju critical
    - štetje pravilno, vendar počasi

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

#define N 10000000
int counter = 0;

void main(void)
{
    int i;

    #pragma omp parallel for
    for(i=0; i<N; i++)
        #pragma omp critical
        counter++;

    printf("Counter = %d\n", counter);
}
```

# Sinhronizacija

- Ukaz **atomic**
  - Gre za posebno obliko označevanja kritičnega območja, ki ga lahko uporabljamo samo za določene enostavne stavke
  - Enostavni stavki so tisti, ki z eno samo prireditvijo spremenijo skalarno spremenljivko
    - x++, x- -, ++x, - -x
    - x <op>= izraz
      <op> = +, -, *, /, &, |, <<, >>
  - Primer: štetje
    - hitreje kot s critical,
    - počasneje kot reduction

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

#define N 10000000
int counter = 0;

void main(void)
{
    int i;

    #pragma omp parallel for
    for(i=0; i<N; i++)
        #pragma omp atomic
        counter++;

    printf("Counter = %d\n", counter);
}
```

# Sinhronizacija

- Ukaz **barrier** poskrbi, da se na določenem mestu v kodi vse niti počakajo
  - Izvajanje kode se nadaljuje šele takrat, ko vse niti dosežejo območje označeno z barrier
- Z ukazom **ordered** poskrbimo, da se vsa koda v nitih izvaja v enakem vrstnem redu kot v sekvenčnem programu

# Sinhronizacija

- Ukaz **single** poskrbi, da označeno območje kode izvrši ena sama nit
- Ukaz **master** poskrbi, da označeno območje kode izvrši samo glavna nit

# Sinhronizacija

- Ukaz **flush**(*vars*)
  - z njim označimo točko v programu, kjer hočemo, da se vrednosti globalnih spremenljivk *vars* v nitih sinhronizirajo
  - pred ukazom flush se morajo zaključiti vsa pisanja v pomnilnik in branja iz njega, hkrati pa se ne sme začeti nobeno novo
  - če je spremenljivka spremenjena pred ukazom flush, mora nit poskrbeti, da njeno vrednost iz predpomnilnika ali registra zapiše v pomnilnik
  - po ukazu flush, mora nit poskrbeti, da spremenljivko prebere iz pomnilnika in ne iz predpomnilnika ali registra

# Sinhronizacija: ključavnice

- Funkcije
  - void **omp_init_lock**(omp_lock_t *)
  - void **omp_set_lock**(omp_lock_t *)
  - void **omp_unset_lock**(omp_lock_t *)
  - void **omp_destroy_lock**(omp_lock_t *)
  - int **omp_test_lock**(omp_lock_t *)
- Ključavnica je tipa **omp_lock_t**

- Ponujajo večjo fleksibilnost kot ukaz critical
  - Postavimo jih na poljubno mesto v kodi
  - Ni potrebe, da z njimi označujemo cele bloke
  - **Ključavnico lahko odklene samo nit, ki jo je zaklenila!!!**

# Sinhronizacija: ključavnice

- Primer
  - Zaklepanje in odklepanje
  - Izpis

```
Nit 1 - zaklenjeno
Nit 1 - odklenjeno
Nit 1 - zaklenjeno
Nit 1 - odklenjeno
Nit 1 - zaklenjeno
Nit 1 - odklenjeno
Nit 1 - zaklenjeno
Nit 1 - odklenjeno
Nit 1 - zaklenjeno
Nit 1 - odklenjeno
Nit 0 - zaklenjeno
Nit 0 - odklenjeno
Nit 0 - zaklenjeno
Nit 0 - odklenjeno
Nit 0 - zaklenjeno
Nit 0 - odklenjeno
Nit 0 - zaklenjeno
Nit 0 - odklenjeno
Nit 0 - zaklenjeno
Nit 0 - odklenjeno
```

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

omp_lock_t my_lock;

int main()
{
    omp_init_lock(&my_lock);
    #pragma omp parallel
    {
        int tid = omp_get_thread_num( );
        int i;

        for (i = 0; i < 5; ++i)
        {
            omp_set_lock(&my_lock);
            printf("Nit %d - zaklenjeno\n", tid);
            Sleep(10);
            printf("Nit %d - odklenjeno\n", tid);
            omp_unset_lock(&my_lock);
        }
    }
    omp_destroy_lock(&my_lock);
}
```

# Dopolnilo collapse

- Hkratno paralelno izvajanje več zank
  - collapse(n), n je število zank, ki naj se izvajajo paralelno

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

int main(void)
{
    int i = 3, j = 7;

    #pragma omp parallel for collapse(2)
    for( i=0; i<2; i++)
        for(j=0; j<10; j++)
            printf("%d-%d\n", i, j);

    return 0;
}
```

```
0-0
0-1
0-2
0-3
1-6
1-9
1-7
1-2
1-3
1-4
1-5
1-0
1-1
0-8
0-9
1-8
0-6
0-7
0-4
0-5
```

# Naloge (tasks) v OpenMP

- Novost v OpenMP3.0
- Brez njih je nemogoče učinkovito sprogramirati določne tipe paralelizma
- Naloge morajo biti pripravljene tako, da jih je mogoče izvajati neodvisno
- Koncept
  - Izbrana nit generira naloge in jih pošilja v bazen nalog
  - Ostale niti izvajajo naloge iz bazena
- Uporabniški vmesni
  - S posebnim ukazom #pragma omp task definiramo nalogo

# Naloge v OpenMP

- Izvajalno okolje
  - Ko nit zazna #pragma omp task, pripravi novo nalogo
  - Čas izvedbe naloge je stvar operacijskega sistema
- Sinhronizacija nalog #pragma omp taskwait
  - Naloga čaka na zaključek vseh nalog, ki jih je ustvarila

# Naloge v OpenMP

- Enostaven primer
  - Naloge lahko uporabljamo samo v paralelnih blokih
  - Običajno naloge začne kreirati ena sama nit (master / single)
  - Vsak blok, ki stoji za #pragma omp task se izvede ločeno
  - Izpisi
    - Vsaka vrstica je rezultat enega zagona programa

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

void main()
{
    #pragma omp parallel
    #pragma omp master
    {
        printf("... ");
        #pragma omp task
        printf("hello ");
        #pragma omp task
        printf("world ");
        printf("!!! ");
    }
}
```

```
... hello !!! world
... hello world !!!
... hello !!! world
... hello !!! world
... hello !!! world
... hello !!! world
... hello !!! world
... hello !!! world
... hello !!! world
... hello !!! world
... hello world !!!
```

# Naloge v OpenMP

- Algoritmi deli in vladaj
  - Naloga, ki računa problem, ustvari dodatno nalogo za reševanje subproblem1, sama pa reši subproblem 2
  - Ko se dodatna naloga zaključi, se iz obeh delov zgradi rešitev

```
function A(problem)
    subproblem1 = f(problem, 1);
    subproblem2 = f(problem, 2);
    #pragma omp task
    solution1 = A(subproblem1);
    solution2 = A(subproblem2);
    #pragma omp taskwait
    solution = g(solution1, solution2);

#pragma omp parallel
#pragma omp master
A(problem)
```

# Naloge v OpenMP

- Algoritmi deli in vladaj
  - deluje, vendar neučinkovito
    - preveč enostavna funkcija
      - večina časa gre za ustvarjanje in razporejanje nalog
      - omejitev globine generiranja novih nalog

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

#define N 30

int fib (int n)
{
    int x, y;

    if (n < 2)
        return n;

    #pragma omp task shared(x)
    x = fib (n-1);
    y = fib (n-2);

    #pragma omp taskwait
    return x + y;
}

int main()
{
    int num;

    #pragma omp parallel
    #pragma omp single
    num = fib(N);

    printf("fib(%d) = %d\n", N, num);
}
```