

Multiprocesorski sistemi

OpenMP

Matija Dodović, Marko Mišić

13S114MUPS, 13E114MUPS, 13M114MUPS

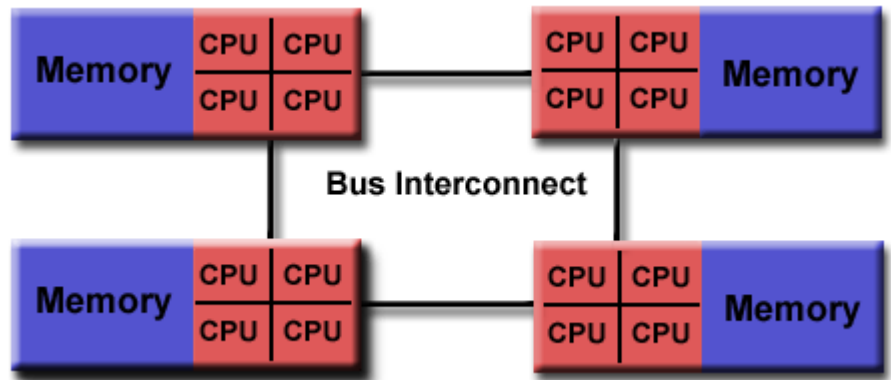
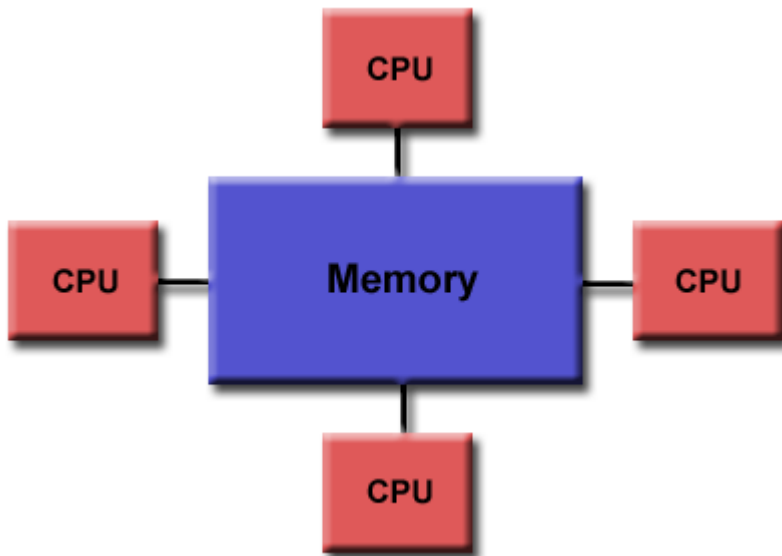
2023/2024.

Koncepti deljene memorije

Sistemi sa deljenom memorijom (1)

- Višenitno programiranje se najčešće koristi na paralelnim sistemima sa deljenom memorijom
 - Tipično, računarski sistem se sastoji od više procesorskih jedinica i zajedničke memorije
- Ključna karakteristika ovih sistema je *jedinstven adresni prostor* u celom memorijskom sistemu
 - Postoji jedan logički memorijski prostor
 - Svaki procesor može da ravnopravno pristupa svim memorijskim lokacijama u sistemu
 - Svi procesori pristupaju memorijskoj lokaciji koristeći istu adresu

Sistemi sa deljenom memorijom (2)



Realni hardver

- Realni hardver sistema sa deljenom memorijom je komplikovaniji od ovoga...
 - Memorija može manje biti podeljena u manje jedinice
 - Može postojati više nivoa keš memorije
 - Neki od ovih nivoa mogu biti deljeni između podskupova procesora
 - Interkonekciona mreža može imati složenu topologiju
- ...ali jedinstveni adresni prostor je i dalje podržan
 - Hardverska kompleksnost *može uticati na performanse programa, ali ne i na njihovu korektnost*

Koncept niti

- Programski model deljene memorije je zasnovan na pojmu niti
 - Niti su kao procesi, osim što niti mogu deliti memoriju međusobno (mogu i imati privatnu memoriju)
- Sve niti mogu pristupiti deljenim podacima
- Samo nit-vlasnik može pristupiti privatnim podacima
- Različite niti mogu pratiti različite tokove kontrole kroz isti program
 - Svaka nit ima svoj programski brojač
- Obično jedna nit po procesoru/jezgru
 - Može ih biti i više
 - Moguća je hardverska podrška za više niti po jezgru
 - *Simultaneous Multithreading* (SMT) kod Intel procesora

Komunikacija između niti

- Radi upotrebljivosti paralelnih programa, neophodna je razmena podataka između niti
- Niti komuniciraju preko čitanja i upisivanja u deljene podatke
 - Na primer:
 - Nit 1 upisuje vrednost u deljenu promenljivu A
 - Nit 2 zatim može da čita vrednost iz A
- U ovom programskom modelu ne postoji pojam poruke

Sinhronizacija

- Niti se podrazumevano izvršavaju asinhrono
- Svaka nit nastavlja da izvršava programske instrukcije nezavisno od ostalih niti
- To znači da moramo da obezbedimo korektan poredak akcija nad deljenim promenljivama
 - Izmene deljenih promenljivih ($a = a + 1$) nisu atomične
 - Može se dogoditi *race condition* između dve niti prilikom istovremenog pristupa radi izmene

Poslovi

- Posao (*task*) je deo izračunavanja koji se može izvršiti nezavisno od drugih poslova
- U principu, možemo kreirati novu nit za izvršavanje svakog posla
 - U praksi ovo može biti suviše skupo, naročito ako imamo veliki broj malih poslova
- Umesto toga, niti se mogu izvršavati pomoću statički kreiranog bazena niti (*thread pool*)
 - Poslovi se predaju bazenu
 - Neka nit iz bazena izvršava posao
 - U nekom trenutku u budućnosti je zagarantovano da će se posao završiti
- Poslovi mogu, ali ne moraju imati uspostavljen međusobni poredak

Paralelne petlje

- Petlje su glavni izvor paralelizma u mnogim aplikacijama
 - Ako iteracije petlje nemaju međusobnu zavisnost po podacima onda možemo raspodeliti iteracije različitim nitima
 - Iteracije se mogu se izvršavati u bilo kom redosledu

- Na primer, ako imamo dve niti i petlju

```
for (i=0; i<100; i++) {  
    a[i] += b[i];  
}
```

možemo da iteriramo od 0-49 u jednoj niti i od 50-99 u drugoj

- Jednu iteraciju, ili skup iteracija, možemo smatrati poslom

Redukcije

- Redukcija proizvodi jednu vrednost pomoću asocijativnih operacija
 - Sabiranje, množenje, maksimum, minimum, logičko i/ili
- Na primer:

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

- Dopuštanje samo jednoj niti da menja b bi uklonilo sav paralelizam
- Umesto toga, svaka nit akumulira rezultat u svoju privatnu kopiju, pa se onda ove kopije redukuju za dobijanje konačnog rezultata
- Ako je broj operacija mnogo veći od broja niti, većina operacija može teći u paraleli

Uvod u OpenMP

Šta je OpenMP?

- OpenMP je aplikativni programski interfejs (API) dizajniran za programiranje paralelnih računarskih sistema sa deljenom memorijom
- OpenMP koristi koncepte *niti* i *poslova*
- OpenMP je skup nadogradnji za Fortran, C i C++
- Nadogradnje se sastoje od:
 - Prevodilačkih direktiva
 - Rutina iz izvršne biblioteke (*runtime*)
 - Promenljivih okruženja

Kratak istorijat OpenMP-a

- Motivacija – nedostatak standardizacije za paralelizaciju direktivama u sistemima sa deljenom memorijom
- Prva verzija 1997, najnovija 4.0 (jul 2013.)
 - Verzija 3.0 je podržala *task* paralelizam
 - Verzija 4.0 donosi podršku za akceleratore
- Rukovodeće telo OpenMP Consortium
 - Preko 25 kompanija i akademskih institucija saraduje (IBM, Intel, AMD, HP...)
- Podržan od strane glavnih prevodilaca
 - gcc, Microsoft, Intel, PGI

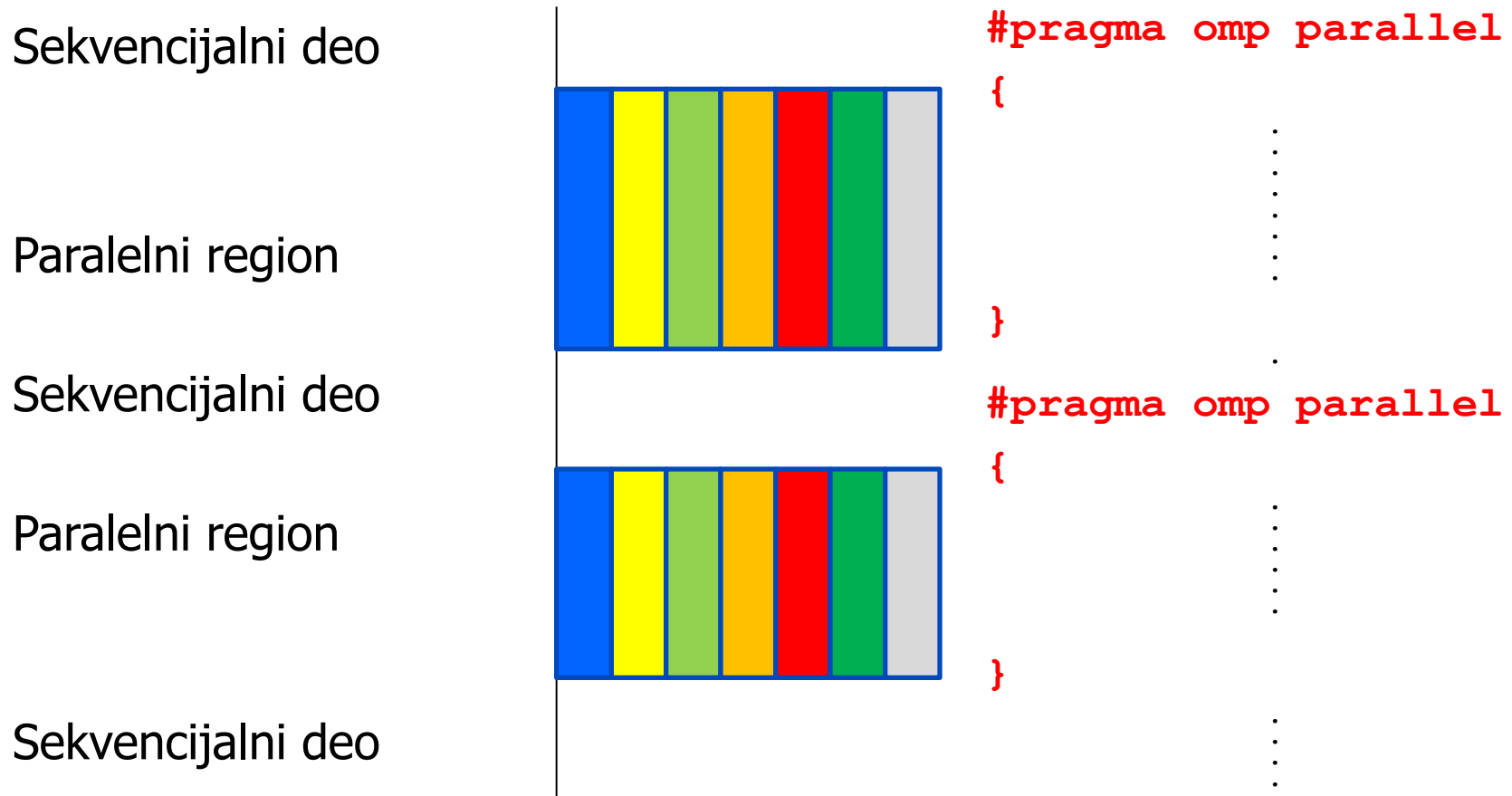
Direktive i oznake

- Direktiva je posebna linija izvornog koda koja ima značenje samo određenim prevodiocima
 - Onima koji podržavaju odgovarajući OpenMP standard
 - Ostali ih jednostavno ignorišu
- Direktiva se prepoznaje pomoću oznake (*sentinel*) na početku linije
 - Na C/C++ su to **#pragma** direktive
- OpenMP oznaka za C/C++ je:
#pragma omp
- OpenMP direktive se ignorišu ako se kod prevodi kao običan sekvencijalni kod

Paralelni region (1)

- Paralelni region je osnovna paralelna konstrukcija u OpenMP
 - Paralelni region definiše sekciju jednog programa
- Program počinje izvršavanje na jednoj, glavnoj (*master*) niti
- Kada se naiđe na prvi paralelni region, glavna nit kreira tim niti (*team of threads*)
 - Poznati fork/join model
- Svaka nit izvršava naredbe zadate unutar paralelnog regiona
 - Posao se replicira
 - Glavna nit ravnopravno učestvuje u poslu
- Na kraju paralelnog regiona, glavna nit čeka na ostale niti da završe i potom nastavlja sa izvršavanjem ostalih naredbi

Paralelni region (2)



Deljeni i privatni podaci

- Unutar paralelnog regiona, promenljive mogu biti deljene (*shared*) ili privatne (*private*)
- Sve niti vide istu kopiju neke deljene promenljive
- Sve niti mogu da čitaju ili pišu u deljene promenljive
- Svaka nit ima svoju kopiju privatne promenljive, koja je nevidljiva za ostale niti
 - Iz privatne promenljive može da čita ili u nju upisuje samo nit koja je njen vlasnik

Paralelne petlje

- Sve niti izvršavaju isti kod u paralelnom regionu
- U OpenMP postoje i direktive koje naznačuju da se određeni posao deli među nitima, a ne replicira
 - To su *worksharing* direktive
- Kako su niti glavni izvor paralelizma u mnogim aplikacijama, OpenMP ima opsežnu podršku za paralelizaciju petlji
 - Postoji veliki broj opcija kojima se kontrolišu koje niti izvršavaju koje iteracije petlje
- Odgovornost programera je da obezbedi da su iteracije paralelne petlje međusobno nezavisne
- Samo one petlje kod kojih se broj iteracija može unapred izračunati (pre izvršavanja) mogu da se na ovaj način paralelizuju

Osnovni sinhronizacioni koncepti

- Barijera
 - Sve niti moraju stići do barijere pre nego što bilo koja može da nastavi
 - Primer: razgraničavanje faza izračunavanja
- Kritični region
 - Sekcija koda u kojoj samo jedna nit može boraviti u datom trenutku
- Atomično ažuriranje
 - Samo jedna nit u datom trenutku može da ažurira datu promenljivu
 - Primer: izmena deljene promenljive
- Glavni (*master*) i *single* regioni
 - Sekcije koda koju može da izvršava samo jedna nit
 - Primer: inicijalizacija, upis u fajl...

Prevođenje programa

- OpenMP je ugrađen u većinu prevodilaca u uobičajenoj upotrebi
- Za prevođenje je potrebno dodati određene opcije komandama za prevođenje i povezivanje programa:
 - `-fopenmp` za gcc
 - `-openmp` za Intel i Sun (Oracle) prevodioce
 - `/openmp` za Microsoft prevodioce
- Broj niti koji će se koristiti se određuje u toku izvršavanja pomoću `OMP_NUM_THREADS` globalne promenljive
 - Može se postaviti ili iz operativnog sistema, ili odgovarajućim funkcijama iz zaglavlja `<omp.h>`
- Pokretanje kao i za običan sekvencijalni program



Paralelni regioni

Paralelni region

- Kod u okviru paralelnog regiona se izvršava od strane svih niti
- C/C++ (*) direktiva

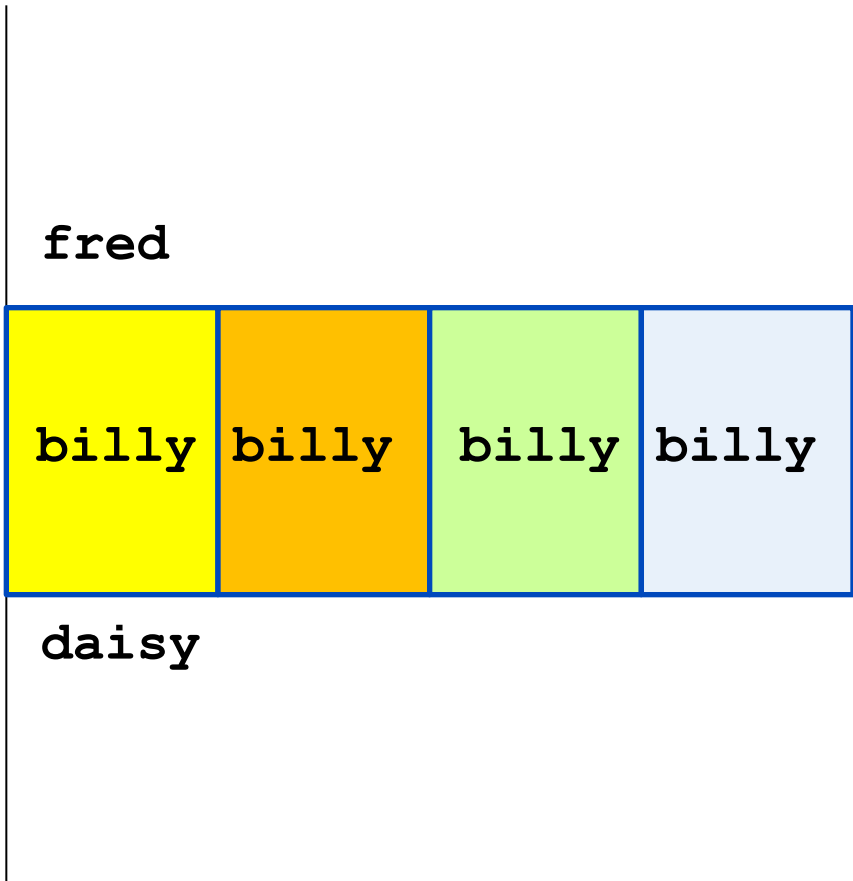
```
#pragma omp parallel (**)  
{  
    blok koda  
}  
ili
```

```
#pragma omp parallel  
jedan iskaz (do sledećeg `;`)
```

* - ubuduće svi iseći koda su za C/C++ podrazumevano

** - podsećanje za C/C++ - novi red obavezno na kraju direktive

Direktiva za paralelni region - primer



```
fred();  
#pragma omp parallel  
{  
    billy();  
}  
daisy();
```


Korisne funkcije i odredbe direktiva

- Korisne funkcije:

- Zaglavlje `<omp.h>`

- Određivanje broja korišćenih niti:

- `int omp_get_num_threads(void);`

- Vraća 1 ako se zove izvan paralelnog regiona

- Određivanje rednog broja niti koja se trenutno izvršava:

- `int omp_get_thread_num(void);`

- Uzima vrednosti od 0 do `omp_get_num_threads() - 1`

- Odredbe direktiva:

- Za specificiranje dodatnih informacija u direktivi za paralelni region se koriste *odredbe* (clauses):

- `#pragma omp parallel [clauses]`

- Odredbe se odvajaju se blanko znakom

Deljene i privatne promenljive

- Unutar paralelnog regiona, promenljive mogu biti *deljene* (sve niti vide istu kopiju) ili *privatne* (svaka nit ima svoju kopiju)
- Odgovarajuće odredbe:

shared (var_list) - podrazumevano

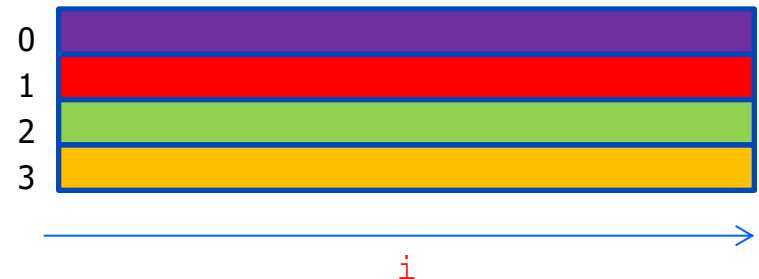
private (var_list)

default (shared|none)

- Primer – svaka nit inicijalizuje svoju vrstu matrice:

- Operator '\ ' se koristi za konkatenciju ako direktiva prelazi u novi red

```
#pragma omp parallel default(none) \  
private(i,myid) shared(a,n)  
{  
    myid = omp_get_thread_num();  
    for (i=0; i<n; i++)  
        a[myid][i] = 1;  
}
```



Inicijalizacija privatnih promenljivih

- Privatne promenljive su neinicijalizovane na početku paralelnog regiona
- Ako želimo da ih inicijalizujemo, koristimo sledeću odredbu:

`firstprivate(var_list)`

- Primer:

```
b = 23.0;
. . . . .
#pragma omp parallel firstprivate(b), private(i,myid)
{
    myid = omp_get_thread_num();
    for (i=0; i<n; i++){
        b += c[myid][i];
    }
    c[myid][n] = b;
}
```

Redukcije

- *Redukcija* proizvodi jednu vrednost pomoću asocijativnih operacija kao što su sabiranje, množenje, maksimum, minimum, logičko i/ili
- Svaka nit redukuje svoj deo posla u privatnu kopiju, pa zatim sve njih redukujemo za dobijanje konačnog rezultata
- Koristimo *reduction* odredbu

reduction(operation: var_list)

- Primer:

```
b = 10;
#pragma omp parallel \
    reduction(+:b), private(i,myid)
{
    myid = omp_get_thread_num() + 1;
    for (i = 0; i < n; i++) {
        b = b + c[i][myid];
    }
}
a = b;
```

Vrednost iz originalne promenljive je sačuvana

Svaka nit dobija privatnu kopiju **b**, inicijalizovanu na 0

Svi pristupi promenljivoj **b** unutar paralelnog regiona se odnose na privatne kopije

Na kraju paralelnog regiona, sve privatne kopije se dodaju na originalnu promenljivu

Direktive za podelu posla

Paralelne for petlje (1)

- Petlje su najčešći izvor paralelizma u većini programa
 - Paralelne petlje su veoma važne!
- Paralelna `for` petlja vrši raspodelu iteracija između niti
- Na kraju bloka niti postoji sinhronizaciona tačka
 - Sve niti moraju da završe svoje iteracije pre nego što bilo koja od njih može da nastavi
- Sintaksa:

```
#pragma omp for [clauses]  
for loop
```

- Ograničenja u C/C++:
 - Postoje ograničenja forme koju petlja može uzeti
 - Petlja mora da ima odrediv broj ponavljanja, odnosno da bude u formi:
for (var = a; var logical_op b; inc_expr)
 - gde je **logical_op** jedna od relacija `<`, `<=`, `>`, `>=` i **inc_expr** oblika **var = var +/- inc** ili semantički ekvivalent poput **var++**.
 - Ne može se menjati **var** u telu petlje

Paralelne for petlje (2)

- Primer:

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

- Konstrukcija koja kombinuje paralelni region i `for` direktivu je toliko česta da postoji skraćeni zapis:

```
#pragma omp parallel for [clauses]
    for loop
```

Odredbe for direktive

- **for** direktive mogu koristiti **private**, **firstprivate** i **reduction** odredbe koje se odnose na opseg petlje
 - promenljiva koja sadrži indeks paralelne petlje je **private** podrazumevano, ali ostali indeksi petlji nisu **private** u C
 - U Fortranu, recimo, jesu
- **parallel for** direktiva može koristiti sve odredbe **parallel** direktive
- Bez dodatnih odredbi, **for** direktiva će izdeliti particije što ujednačenije po nitima
- Ipak, ovo zavisi od implementacije, i u opštem slučaju postoji neodređenost u podeli:
 - 7 iteracija se mogu podeliti na 3 niti kao 3+3+1, ili 3+2+2

Schedule odredba

- *Schedule* direktiva daje razne mogućnosti za specificiranje niti koja će izvršiti određenu naredbu
- Sintaksa:

```
schedule(kind [, chunksize])
```
- gde je *kind* iz skupa {**STATIC, DYNAMIC, GUIDED, AUTO, RUNTIME**}, a *chunksize* celobrojni pozitivan izraz

Static raspored

- Ako *chunksize* nije specificiran, iteracije se približno ravnomerno dele u jednake pakete (*chunks*)
 - Broj je jednak broju niti
 - Svaki paket se redom dodeljuje odgovarajućoj niti
 - 0-toj niti 0-ti paket, itd. (blokovski raspored)
- Ako *chunksize* jeste specificiran, iteracije se dele u pakete veličine *chunksize* iteracija
 - Ciklično se dodeljuju nitima redom po *id* niti
 - Ciklični blokovski raspored



`schedule (static)`



`schedule (static, 4)`

Dynamic i guided rasporedi (1)

- *Dynamic* raspored vrši podelu iteracija u pakete veličine *chunksize*, a zatim ih dodeljuje nitima po principu *first-come first-served*
 - Na primer, kada nit završi jedan paket, dodeljuje joj se sledeći paket iz liste paketa
- Ako *chunksize* nije specificiran, podrazumevana vrednost je 1
- *Guided* raspored je sličan *dynamic*, ali paketi su na početku veliki, a sa vremenom se eksponencijalno smanjuju
 - Veličina sledećeg paketa je proporcionalna broju iteracija podeljenim sa brojem niti
- *Chunksize* određuje minimalnu veličinu paketa
- Ako *chunksize* nije specificiran, podrazumevana vrednost je 1

Dynamic i guided rasporedi (2)



Auto raspored

- Pomera svo odlučivanje o rasporedu u vreme izvršavanja
- Ako se paralelna petlja izvršava mnogo puta, u toku izvršavanja se može razviti dobar raspored koji ima dobar balans opterećenja i malo režijsko vreme
- Podržavaju ga samo neke implementacije
 - Ostale ovo najčešće obrade kao *static* raspored

Izbor rasporeda

- Kada koristiti koji raspored?
 - *Static* je najbolji za petlje sa dobrim balansom opterećenja (najmanje režijsko vreme)
 - *Static, n* je dobar za petlje sa blagim ili ujednačenim disbalansom opterećenja
 - Javlja se malo režijsko vreme
 - *Dynamic* je koristan ako iteracije imaju veoma varirajuće opterećenje, ali kviri lokalnost podataka
 - *Guided* obično manje košta od *dynamic*, ali treba se paziti petlji gde su početne iteracije najskuplje
 - *Auto* može biti koristan ako se petlja izvršava nanovo mnogo puta

Ugneždene petlje

- Koristi se za savršeno ugneždene pravougaone petlje
- Višestruke petlje možemo paralelizovati *collapse* odredbom, pomoću sažimanja:

```
#pragma omp parallel for collapse(2)
  for (int i=0; i<N; i++) {
    for (int j=0; j<M; j++) {
      .....
    }
  }
```

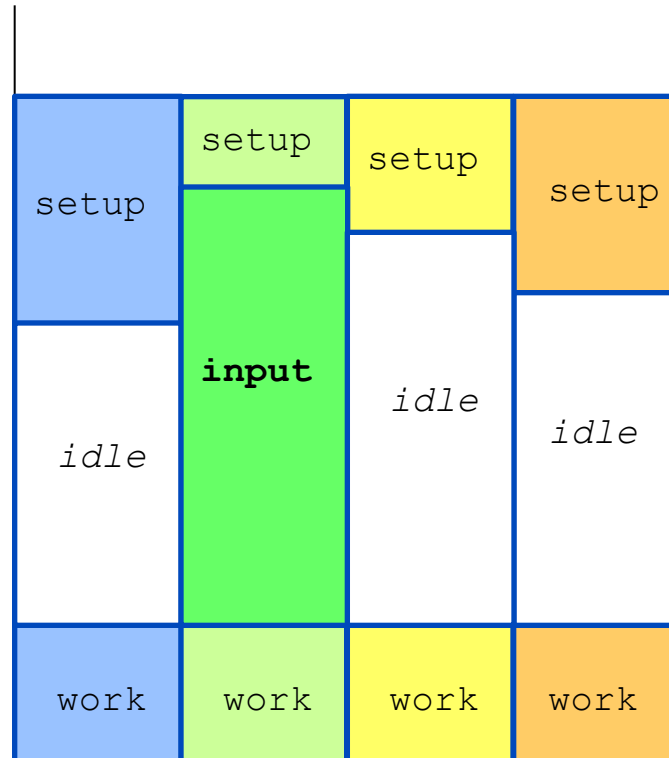
- Argument odredbe je broj petlji koje treba sažeti, počev od spoljne
 - Napraviće jednu petlju dužine $N \times M$ i onda je paralelizovati
 - Korisno ako je N jednako broju niti, u kom slučaju paralelizacija spoljne petlje može imati dobar balans opterećenja

Single direktiva

- Naznačuje da blok koda treba izvršiti samo pomoću jedne niti
- Prva nit koja dosegne single direktivu će izvršiti taj blok
- Postoji sinhronizaciona tačka na kraju bloka
 - Sve niti čekaju dok se ceo blok ne izvrši
- Sintaksa:
#pragma omp single [clauses]
structured block
- **single** direktiva može koristiti **private** i **firstprivate** odredbe

Single direktiva (nastavak)

```
#pragma omp parallel
{
    setup(x);
    #pragma omp single
    {
        input(y);
    }
    work(x,y);
}
```



Master direktiva

- Naznačuje da blok koda treba izvršiti samo pomoću glavne (master) niti
 - To je niti sa *id 0*
- Ne postoji sinhronizacija na kraju bloka
 - Ostale niti preskaču blok i nastavljaju sa izvršavanjem
 - Glavna razlika u odnosu na `single` direktivu
 - Loša dizajnerska odluka prilikom definisanja standarda!
- Sintaksa:

```
#pragma omp master  
structured block
```

Sinhronizacija

Zašto je potrebna sinhronizacija?

- Podsećanje:
 - Potrebno je sinhronizovati akcije nad deljenim promenljivama
 - Potrebno je osigurati ispravan redosled čitanja i pisanja
 - Potrebno je zaštititi ažuriranja deljenih promenljivih
 - Ažuriranja nisu podrazumevano atomična
- Sinhronizacija se u OpenMP implementira kroz barijere, kritične sekcije, brave i atomične operacije

Barrier direktiva (1)

- Nijedna nit ne može proći barijeru dok ostale niti još nisu pristigle
- Implicitne barijere postoje na kraju `for`, `sections` i `single` direktiva
- Sintaksa:
 - `#pragma omp barrier`
- Ili će sve niti doći do barijere, ili nijedna od njih
 - U suprotnom nastaje **deadlock!!!**

Barrier direktiva - primer (2)

- Barijera je potrebna da se forsira sinhronizacija nad **a**

```
#pragma omp parallel private(i,myid,neighb)
{
    myid = omp_get_thread_num();
    neighb = myid - 1;
    if (myid=0) neighb = omp_get_num_threads() - 1;
    ...
    a[myid] = a[myid]*3.5;
    #pragma omp barrier
    b[myid] = a[neighb] + c;
    ...
}
```

Kritične sekcije (1)

- Kritična sekcija je blok koda koji može izvršavati samo jedna nit u jednom trenutku
- Može se koristiti za zaštitu prilikom ažuriranja deljenih promenljivih
- Sintaksa:

```
#pragma omp critical [(name)]  
structured block
```

- Direktiva dozvoljava da se kritične sekcije imenuju
 - Ako je jedna nit u kritičnoj sekciji sa datim imenom, nijedna druga nit ne može biti u kritičnoj sekciji sa istim imenom
 - Niti mogu biti u kritičnim sekcijama sa različitim imenima
- Ako se ime izostavi, podrazumevano ime je **null**
 - Sve neimenovane kritične sekcije imaju efektivno isto ime

Kritične sekcije (2)

- Primer implementacije steka

```
#pragma omp parallel shared(stack), private(inext, inew)
{
#pragma omp critical (stackprot)
{
    inext = getnext(stack);
}
work(inext, inew);
#pragma omp critical (stackprot)
{
    if (inew > 0) putnew(inew, stack);
}
...
}
```


Rad sa bravama (1)

- Povremeno je potrebna veća fleksibilnost nego što nam omogućava **critical** direktiva
- Brava je specijalna promenljiva koja može biti zaključana od strane neke niti
 - Nijedna druga nit ne može da je zaključa dok je ona nit koja je drži zaključanom ne otključa
 - Zaključavanje može biti blokirajuće i neblokirajuće
- Brava mora biti inicijalizovana pre upotrebe, a može biti uništena kada više nije potrebna
- Brave ne bi trebalo koristiti za druge namene
 - Kao obične promenljive ili za bilo šta drugo što nije opisano njihovom semantikom

Rad sa bravama (2)

- Sintaksa:

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

```
void omp_init_lock(omp_lock_t *lock);
```

```
void omp_set_lock(omp_lock_t *lock);
```

```
int omp_test_lock(omp_lock_t *lock);
```

```
void omp_unset_lock(omp_lock_t *lock);
```

```
void omp_destroy_lock(omp_lock_t *lock);
```

- Postoje i rutine za brave čiji se pozivi mogu ugneždavati

- One dozvoljavaju da ista nit zaključa bravu više puta pre nego što je otključa isti broj puta
- Podrška za implementaciju rekurzije

Rad sa bravama (3)

- Primer – izračunavanje stepena svakog čvora u grafu

```
for (i=0; i<nvertexes; i++) {
    omp_init_lock(lockvar[i]);
}
#pragma omp parallel for
for (j=0; j<nedges; j++) {
    omp_set_lock(lockvar[edge[j].vertex1]);
    degree[edge[j].vertex1]++;
    omp_unset_lock(lockvar[edge[j].vertex1]);
    omp_set_lock(lockvar[edge[j].vertex2]);
    degree[edge[j].vertex2]++;
    omp_unset_lock(lockvar[edge[j].vertex2]);
}
```

Atomic direktiva

- *Atomic* direktiva specificira da se određena memorijska lokacija mora ažurirati atomično
 - Ne dozvoljava se upis od strane više niti istovremeno
 - Esencijalno, omogućava definisanje kratke kritične sekcije
 - Može biti podržana hardverski na nekim platformama
- Sintaksa:

```
#pragma omp atomic
    statement_expression
```
- Direktiva se primenjuje samo na naredbu koja je neposredno prati

Poslovi

Task direktiva

- Konstrukcija *posla* (*task*) definiše sekciju koda koju može izvršiti nit koja prva naiđe na nju ili može biti upakovana za kasnije izvršavanje
 - Unutar paralelnog regiona, nit koja naiđe na `task` direktivu će upakovati posao za izvršavanje
 - Neka nit u paralelnom regionu će izvršiti posao u nekom trenutku u budućnosti
 - Posao će biti stavljen u bafer za kasnije izvršavanje
- Sintaksa:

`#pragma omp task [clauses]`
`structured-block`

Prosleđivanje podataka poslovima

- Promenljive se u poslove podrazumevano prosleđuju kao **firstprivate**
 - Posao se može izvršiti u nekom kasnijem trenutku
 - Originalna promenljiva ne mora postojati u trenutku izvršavanja posla, već može biti van doseg
- Promenljive koje se dele u svim konstrukcijama počev od one najugneždenije okružujuće **parallel** konstrukcije su **shared**
- U sledećem primeru, A je **shared**, B je **firstprivate**, C je **private**:

```
#pragma omp parallel shared(A) private(B)
{
    . . .
    #pragma omp task
    {
        int C;
        compute(A, B, C);
    }
}
```

Gde i kada se poslovi završavaju?

- Na barijeri za niti (eksplicitnoj ili implicitnoj):
 - Važi za sve poslove generisane u trenutnom paralelnom regionu sve do barijere
- Na **taskwait** direktivi:
 - Nit koja naiđe na ovu direktivu će čekati na završetak poslova koji su generisani od početka izvršavanja tekućeg posla
 - Važi samo za poslove generisane u trenutnom poslu, ne i za „naslednike”
- Sintaksa:

#pragma omp taskwait

Obilazak ulančane liste (1)

- Primer obilaska ulančane liste
 - Klasičan obilazak povezane liste
 - Izvršava se neka akcija nad svakim elementom liste
 - Podrazumeva se da elementi mogu biti obrađeni nezavisno
 - Ne može se koristiti OpenMP `for` direktiva

```
p = listhead ;  
while (p) {  
    process (p) ;  
    p=next (p) ;  
}
```

Obilazak ulančane liste (2)

```
#pragma omp parallel
{
    #pragma omp single private(p)
    {
        p = listhead ;
        while (p) {
            #pragma omp task
                process (p) ;
            p=next (p) ;
        }
    }
}
```

Samo jedna nit pakuje poslove

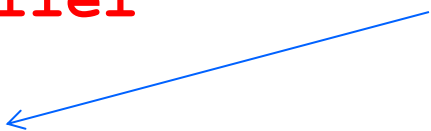
p je podrazumevano **firstprivate** unutar ovog posla

Implicitni **taskwait**

Obilazak ulančane liste (3)

```
#pragma omp parallel
{
    #pragma omp for private(p)
    for ( int i =0; i < numlists; i++) {
        p = listheads[i];
        while (p) {
            #pragma omp task
                process (p);
            p=next (p) ;
        }
    }
}
```

Sve niti pakuju poslove



Postorder obilazak stabla

- Implementira se kroz binarno stablo poslova
- Obilazi se pomoću rekurzivne funkcije
- Posao ne može da se završi dok se svi poslovi ispod njega u stablu ne završe:

```
void postorder(node *p) {  
    if (p->left)  
        #pragma omp task  
            postorder(p->left);  
    if (p->right)  
        #pragma omp task  
            postorder(p->right);  
    #pragma omp taskwait  
    process(p->data);  
}
```

Roditeljski posao suspendovan dok se poslovi-deca ne završe

Promena posla za izvršavanje (1)

- Određene konstrukcije imaju tačke za raspoređivanje poslova na definisanim lokacijama unutar njih
 - *Task scheduling point*
- Kada nit naiđe na tačku raspoređivanja, dozvoljava joj se da suspenduje trenutni posao i izvrši drugi
 - To se zove *promena posla (task switching)*
- Nakon toga može da se vrati početnom poslu i nastavi dalje

Promena posla za izvršavanje (2)

- Primer:

```
#pragma omp single
{
  for (i=0; i < ONEZILLION; i++)
    #pragma omp task
      process(item[i]);
}
```

- Rizik od generisanja previše poslova
- Generisani posao mora biti suspendovan neko vreme
- Sa promenom posla, nit koja se izvršava može da:
 - Izvrši već generisani posao, iscrpljujući time rezervu poslova (*task pool*)
 - Izvrši posao na koji naiđe

Pravilna upotreba poslova

- Pravilno dohvaćanje opsega atributa podataka može biti poprilično zahtevno
 - Podrazumevana pravila opsega su različita nego kod drugih konstrukcija
 - Kao i obično, korišćenje **default (none)** može biti dobra ideja
- Ne koristiti poslove za stvari koje su već dobro podržane u OpenMP kao što su *worksharing* direktive
 - Za standardne for petlje
- Režijsko vreme korišćenja poslova je veće
- Ne treba očekivati čuda od izvršnog okruženja
 - Najbolji rezultati se dobijaju tamo gde korisnik kontroliše broj i granularnost poslova

Ostale teme u OpenMP

Ugneždeni paralelizam (1)

- OpenMP dozvoljava ugneždeni (*nested*) paralelizam
 - Omogućeno je pomoću **OMP_NESTED** promenljive okruženja ili **omp_set_nested()** rutine
- Ako se *parallel* direktiva nađe unutar druge *parallel* direktive, novi *tim niti* (*team of threads*) će biti kreiran
 - Ako ugnježđeni paralelizam nije omogućen, novi tim će sadržati samo jednu nit
- Ugneždeni paralelizam nije podržan u nekim implementacijama OpenMP
 - Kod će se izvršiti, ali kao da je **OMP_NESTED** bilo postavljeno na **false**

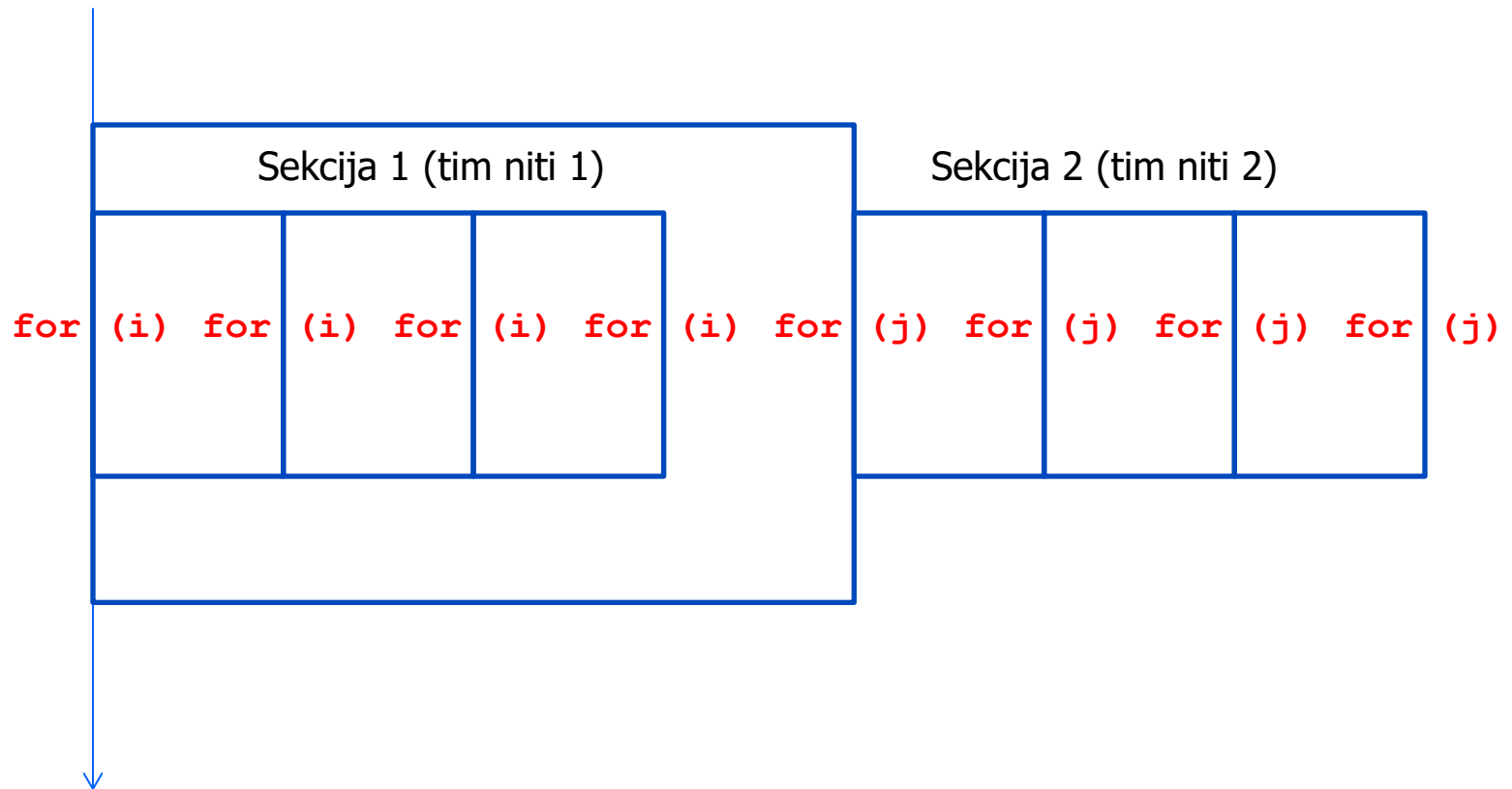
Ugneždeni paralelizam (2)

- Ponekad, može biti zgodno da se iskoristi neskalabilni paralelizam
 - Koristi se **sections** direktiva

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            #pragma omp parallel for
            for (int i = 0; i < n; i++) x[i] = 1;
        }
        #pragma omp section
        {
            #pragma omp parallel for
            for (int j = 0; j < n; j++) y[j] = 2;
        }
    }
}
```

Ugneždeni paralelizam (3)

- Različiti timovi niti rade nad različitim petljama



Numthreads direktiva

- Jedan način da se kontroliše broj niti koji se koristi na svakom nivou je korišćenje `numthreads` odredbe

```
#pragma omp parallel for num_threads(4)
{
    for (int i = 0; i < n; i++)
    {
        #pragma omp parallel for num_threads(totalthreads/4)
        {
            for (int j = 0; j < n; j++) a[i][j] = b[i][j];
        }
    }
}
```

- Vrednost postavljena u odredbi ima veći prioritet od vrednosti promenljive okruženja `OMP_NUM_THREADS` ili vrednosti postavljene korišćenjem `omp_set_num_threads()`

Orphaned direktive (1)

- Direktive su aktivne u *dinamičkom* opsegu paralelnog regiona, ne samo u njegovom *leksičkom* opsegu
 - One koje se nalaze samo u dinamičkom opsegu se zovu *orphaned* direktive
- Ovo je veoma korisno, jer dozvoljava modularan stil programiranja
 - Može biti i veoma zbunjujuće, ako je stablo poziva komplikovano
- Postoje dodatna pravila u vezi sa atributima opsega podataka

Orphaned direktive (2)

- Primer:

```
#pragma omp parallel  
    fred();
```

...

```
void fred() {  
#pragma omp for  
{  
    for (int i = 0; i < n; i++) {  
        a[i] += 23.5;  
    }  
}  
}
```

Orphaned direktive (3)

- Pravila opsega podataka – kada pozivamo neku funkciju iz paralelnog regiona:
 - Promenljive u listi argumenata nasleđuju atribut opsega podataka od pozivajuće rutine
 - Globalne promenljive u C++ su deljene, osim ako se ne označe sa **threadprivate**
 - **static** lokalne promenljive u C/C++ su deljene
 - Sve ostale lokalne promenljive su privatne
- Pravila vezivanja – otklanjaju višeznačnost vezanu za paralelni region na koji direktive referišu:
 - **for**, **sections**, **single**, **master** i **barrier** direktive se uvek vezuju za najbližu okružujuću **parallel** direktivu

Globalne promenljive privatne za nit (1)

- Može biti zgodno da svaka nit ima svoju kopiju promenljive sa globalnim dosegom
 - Promenljive datotečnog dosega ili prostora imena u C/C++
- Izvan paralelih regiona i *master* direktiva, pristupi ovakvim promenljivima referišu na kopiju koja pripada glavnoj niti
- Sintaksa:
#pragma omp threadprivate (var_list)
- Ova direktiva mora biti u datotečnom dosegu ili prostoru imena, posle deklaracija svih promenljivih iz **var_list** i pre bilo koje reference na promenljive iz **var_list**
 - Pogledati dokumentaciju standarda za druga ograničenja

Globalne promenljive privatne za nit (2)

- Promenljive označene sa `threadprivate` imaju nedefinisanu vrednost prilikom prvog nailaska na paralelni region
 - Ove promenljive zadržavaju vrednost između dva paralelna regiona u kodu
 - Ukoliko broj niti ostane isti
- `Copyin` odredba omogućava inicijalizaciju `threadprivate` promenljivih početnim vrednostima na početku paralelnog regiona
- Sintaksa:
`copyin(var_list)`

Merenje vremena (1)

- OpenMP ima podršku za prenosivi tajmer
 - Vraća ukupno proteklo vreme do trenutka poziva (u odnosu na proizvoljni početak) pomoću:

```
double omp_get_wtime(void);
```
 - Vraća preciznost tajmera:

```
double omp_get_wtick(void);
```
- Skalabilno i prenosivo rešenje za merenje vremena na različitim paralelnim sistemima

Merenje vremena (2)

- Primer:

```
double starttime, endtime;  
starttime = omp_get_wtime();  
...// rad čije vreme izvršavanja se meri  
endtime = omp_get_wtime() - starttime;
```

- Tajmeri su lokalni za niti
 - Stoga oba poziva moraju biti napravljena u istoj niti
- Ne postoji nikakva garancija o rezoluciji tajmera

Memorijski model

Zašto nam treba memorijski model? (1)

- Na modernim računarima kod se retko izvršava u redosledu u kojem je naveden u izvornom kodu
 - Out-of-order izvršavanje
- Prevodioci, procesori i memorijski sistemi preuređuju kod da izvuku maksimum performansi
 - Spekulativno izvršavanje
- Pojedinačne niti, kada se posmatraju izolovano, ispoljavaju *as-if-serial* semantiku
- Pretpostavke programera bazirane na memorijskom modelu važe čak i u slučaju preuređivanja koda izvršenog od strane prevodioca, procesora i memorije

Zašto nam treba memorijski model? (2)

- Rezonovanje u vezi sa višenitnim izvršavanjem nije sasvim jednostavno

T1 **T2**

```
x=1; int r1=y;
```

```
y=1; int r2=x;
```

- Ako nema preuređivanja i nit **T2** vidi da je vrednost **y** pri čitanju jednaka 1, onda sledeće čitanje **x** bi trebalo da takođe vrati vrednost 1
- Ako je kod niti **T1** preuređen, onda ne možemo više da pravimo ovakvu pretpostavku

OpenMP memorijski model

- OpenMP održava *relaxed-consistency* model deljene memorije
- Niti mogu da održavaju privremeni pogled (*temporary view*) na deljenu memoriju koji nije konzistentan sa pogledom drugih niti
- Ovi privremeni pogledi postaju konzistentni samo u određenim tačkama u programu
- Operacija koja sprovodi konzistenciju se zove **flush** operacija

Flush operacija (1)

- Definiše određenu tačku u sekvenci operacija
 - U toj tački nit ima garantovano konzistentan pogled na memoriju
- Sva prethodna čitanja ili pisanja koje je proizvela tekuća nit su završena i vidljiva drugim nitima
 - Nikakva čitanja niti upisi od strane ove niti se nisu dogodili nakon te tačke
- *Flush* operacija je analogna operaciji *fence* u drugim API za deljenu memoriju

Flush operacija (2)

- *Flush* operacija je implicitna u OpenMP sinhronizacionim tačkama:
 - Na ulazu/izlazu paralelnog regiona
 - Na implicitnim i eksplicitnim barijerama
 - Na ulazu/izlazu kritičnih regiona
 - Kad god se brava zaključa ili otključa
- Međutim, *flush* operacija ne događa:
 - Na ulazu u *worksharing* regione ili ulazu/izlazu *master* regiona

Producer-consumer obrazac (1)

- Kvalifikator `volatile` u C/C++ ne daje dovoljne garancije u vezi sa višenitnim izvršavanjem
- Ovaj kod nije ispravan:

```
Nit 0           Nit 1  
a = foo();      while (!flag);  
flag = 1;      b = a;
```

- Prevodilac ili hardver mogu preurediti čitanja/pisanja iz/u promenljive `a` i `flag`
- Promenljiva `flag` se može nalaziti u registru

Producer-consumer obrazac (2)

- OpenMP ima `flush` direktivu koja specificira eksplicitnu operaciju
- Ovaj kod koji koristi flush direktivu je ispravan:

Nit 0

```
a = foo();
```

```
#pragma omp flush
```

```
flag = 1;
```

```
#pragma omp flush
```

Prvi `flush` osigurava da se `flag` piše posle `a`
Drugi `flush` osigurava da se `flag` piše u memoriju

Nit 1

```
#pragma omp flush
```

```
while (!flag){
```

```
    #pragma omp flush
```

```
}
```

```
#pragma omp flush
```

```
b = a;
```

Prvi i drugi `flush` osiguravaju da se `flag` čita iz memorije
Treći `flush` osigurava korektan poredak svih `flush` direktiva

Korišćenje flush direktive

- Da bi upis iz niti A u neku promenljivu garantovano bio validan i garantovano vidljiv u niti B, sledeće operacije se moraju desiti tačno u navedenom redosledu:
 1. nit A piše u promenljivu
 2. nit A izvršava *flush* operaciju
 3. nit B izvršava *flush* operaciju
 4. nit B čita promenljivu
- Ispravno korišćenje `flush` je teško i podložno greškama
 - Izuzetno je teško za proveru ispravnosti koda
 - Može da se izvršava korektno na jednoj prevodiocu, ali ne i na drugoj
 - Greške se mogu izazvati promenom nivoa optimizacije u prevodiocu
- **Ne koristite osim ako niste 100% sigurni šta radite**
 - Pa čak i onda budite oprezni...

Podešavanje performansi

Generatori režijskog vremena

- Postoji 6 glavnih uzroka slabih performansi u paralelnim programima sa deljenom memorijom:
 - Sekvencijalan kod
 - Komunikacija
 - Disbalans opterećenja
 - Sinhronizacija
 - Zasićenje hardverskih resursa
 - (Ne)optimizacije prevodioca

Minimizacija režijskog vremena

- Problem – kod daje loše ubrzanje i nije poznato zašto
- Postupak rešavanja:
 1. Odustati
 - Ako je mašina/jezik „gomila smeća”
 - Opcije koje slede vam ne mogu pomoći, odustanite
 2. Pokušati sa klasifikacijom i lokalizacijom overhead-a
 - Koja je vrsta problema, gde u kodu se ispoljava?
 - Koristiti sve dostupne alate – tajmere, hardverske brojače, profajlere...
 - Prvo popraviti probleme koji više doprinose sa overhead-om
 - Iterirati

Optimizacija sekvencijalnog koda

- Količina sekvencijalnog koda u programu ograničava performanse
 - U skladu sa Amdalovim zakonom
 - Potrebni su načini za njegovu paralelizaciju ili lokalizaciju
- U OpenMP-u, sav kod izvan `parallel` regiona i unutar `master`, `single` i `critical` direktiva se izvršava sekvencijalno
 - Količina ovog koda mora biti minimalna

Optimizacija sinhronizacije

- Barijere predstavljaju veliki vremenski trošak
 - Tipično od 1000 do 10000 ciklusa procesora
 - Mogu se ukloniti `nowait` odredbama
- Izbor između `critical` / `atomic` / `lock` direktiva može imati uticaj na performanse
 - Pokušati sa drugačijom implementacijom
- Paralelizovati uvek spoljni nivo
 - Može zahtevati preuređivanje petlji i/ili pristupa nizovima

Nowait odredba (1)

- `Nowait` odredba se može koristiti za poništavanje implicitnih barijera na kraju `for`, `sections` i `single` direktiva
- Sintaksa:

```
#pragma omp for nowait  
    for loop
```
- Analogno za `sections` i `single`

Nowait odredba (2)

- Primer: dve petlje bez međuzavisnosti:

```
#pragma omp parallel
{
    #pragma omp for nowait
        for (j = 0; j < n; j++)
            a[j] = c * b[j];
    #pragma omp for
        for (i = 0; i < m; i++)
            x[i] = sqrt(y[i]) * 2;
}
```

Nowait odredba (3)

- Koristiti sa **izuzetnom pažnjom !!!**
- Vrlo lako se može ukloniti potrebna barijera
- To rezultuje najgorom vrstom greške - nedeterminističkim ponašanjem:
 - Ponekad je ispravno, ponekad ne, u debageru se opet promeni ponašanje, itd.
- Jedan dobar stil kodiranja je onaj gde se gde se sve implicitne barijere isključuju sa `nowait`, a onda svuda gde je potrebno postavljaju eksplicitne barijere

Nowait odredba (4)

- Može se ukloniti *//* prva *//* druga barijera, ali ne i obe, zbog zavisnosti od **a**:

```
#pragma omp for schedule(static, 1)
  for (j = 0; j < n; j++)
    a[j] = b[j] * c[j];
#pragma omp for schedule(static, 1)
  for (j = 0; j < n; j++)
    d[j] = e[j] * f[j];
#pragma omp for schedule(static, 1)
  for (j = 0; j < n; j++)
    z[j] = a[j] * a[j+1];
```

Komunikacija

- Na sistemima sa deljenom memorijom, komunikacija je sakrivena iza povećane cene pristupa memoriji
 - Više vremena je potrebno da bi se podatak dobavio iz operativne memorije ili keš memorije drugog procesora, nego iz lokalne keš memorije
- Pristupi memoriji su skupi
 - Oko ~300 ciklusa za operativnu memoriju, poredeći sa 1-3 ciklusa za registre i keš memoriju
- Komunikacija između procesora se vrši posredstvom mehanizma za održavanje koherencije keš memorije
- Za razliku od modela razmene poruka, komunikacija je prisutna tokom celokupnog izvršavanja programa
 - Komunikacija nije lokalizovana u tačno određenim tačkama
 - Zbog toga je znatno teže pratiti i analizirati

Razmeštanje podataka (1)

- Podaci će biti keširani na procesorima koji im pristupaju
 - Stoga treba koristiti keširane podatke što je više moguće
- Kod treba pisati sa što većim afinitetom prema istim podacima
 - Treba obezbediti da ista nit pristupa istom podskupu podataka što je više moguće
- Takođe, podskupovi podataka koje obrađuje jedna nit treba da budu veliki, kontinualni blokovi memorije
 - Izbegava se *false sharing* efekat

Razmeštanje podataka (2)

- Na sistemima sa distribuiranom deljenom memorijom, lokacija podataka u operativnoj memoriji je važna
 - To je slučaj sa cc-NUMA sistemima, kakvi su svi *multi-socket* x86 sistemi
 - OpenMP ne može da kontroliše razmeštanje podataka po procesorima na takvim sistemima
- Podrazumevana politika većine operativnih sistema je da podatke alokira bliže procesoru koji im prvi pristupa (*first touch policy*)
 - Za OpenMP programe ovo može biti najgora moguća opcija
 - Podaci se često inicijalizuju u master niti i prema tome alociraju u memoriji jednog čvora u sistemu
 - Kako sve niti pristupaju podaima na istom čvoru, on postaje ozbiljno usko grlo sistema!

Razmeštanje podataka (3)

- Pojedini operativni sistemi imaju opcije za kontrolu razmeštanja podataka
 - Na novijim Linux jezgrima se *first touch policy* politika može zameniti sa *round robin* politikom
- *First touch policy* se može koristiti za kontrolu razmeštanja podataka indirektno, paralelizacijom inicijalizacije podataka
 - Iako sam postupak možda nije vredan paralelizacije sa stanovišta vremena izvršenja
 - Paralelizacija ne mora biti potpuno uniformna, bitno je samo da se izbegne usko grlo
- Operativni sistemi rade alokaciju memorije na bazi stranica
 - Tipično od 4KB do 16KB
 - Treba biti obazriv sa velikim stranicama, zbog *false sharing* efekta

Balansiranje opterećenja

- Disbalans u opterećenju može nastati i zbog disbalansa u komunikaciji i disbalansa u izračunavanju
 - Rešenje može biti u primeni različitih opcija za raspoređivanje u zavisnosti od promenljive okruženja **OMP_SCHEDULE**
 - Tada se za raspoređivanje koristi **schedule(runtime)** odredba
- Za neke probleme je raspoređivanje pogodno uraditi ručno
 - Neregularno, blokovsko raspoređivanje je pogodno za neke tipove ugneženih, trougaonih petlji
- Za izračunavanja sa izraženom neregularnošću u smislu opterećenja, najbolje je koristiti OpenMP poslove (*tasks*)
 - Tada se izvršno okruženje brine za balansiranje opterećenja

Zasićenje hardverskih resursa

- Kod sistema sa deljenom memorijom, postoje resursi kojima sva procesorska jezgra pokušavaju da pristupe:
 - Memorijski propusni opseg
 - Keš memorija
 - Funkcionalne jedinice
- Neki programi mogu zauzeti više resursa nego što im zapravo pripada
- Treba iskoristiti lokalnost podataka kako bi se smanjio potreban propusni opseg i poboljšalo korišćenje keš memorija

Optimizacije prevodioca

- Ponekad dodavanje paralelnih direktiva sprečava prevodioca u obavljanju optimizacija sekvencijalnog koda
 - Klasičan primer predstavlja 1-nitni paralelni kod koji se izvršava duže nego sekvencijalni kod
 - Takođe, mašinski kod tada ima veći broj instrukcija
- Ponekad se ovi problemi mogu razrešiti proglašavanjem deljenih podataka privatnim, tamo gde je to moguće
 - Početna vrednost im se može preneti pomoću `firstprivate` odredbe

Dodatni izvori

Literatura

- Dodatni izvori:

- Zvanični sajt: <http://www.openmp.org>

- Specifikacije jezika, linkovi ka prevodiocima i alatima, forumi...

- Knjiga:

- Chapman , Jost and Van der Pas,
Using OpenMP:
Portable Shared Memory Parallel Programming, MIT Press

- Tutorijal:

- <https://computing.llnl.gov/tutorials/openMP/>