OpenMP

Veljko Petrović
Novembar, 2022

OpenMP

Model deljene memorije

Šta je OpenMP

- Ovo bi sve trebali da već znate.
- No, dovoljno je bitno da pređemo ponovo.
- OpenMP je ‘Open MultiProcessing’
- Centralna ideja je deljena memorija i više niti izvršavanja
- Nekada davno, svo paralelno programiranje je bilo ovako.
- Ovo je i dalje stil programiranja koji je najčešći u korisničkim aplikacijama zato što se izvršavaju na jednoj mašini, bez obzira koliko procesora ima.
- Sam jezik je C/C++ mada %OpenM%P postoji i za Fortran.

Arhitektura OpenMP

- Glavna jedinica podele izvršavanja u OpenMP je nit (thread)
- Niti se mogu podeliti u:
  - Glavnu
  - Niti radilice (worker threads)
- Svaka nit je nezavisna traka izvršavanja koja prolazi kroz program
- Ono što odlikuje glavnu nit jeste što počinje prva i što njeno završavanje završava program. Glavna nit takođe pokreće, zaustavlja i kontroliše niti-radilice.
- Način na koji se ovakvo ponašanje postiže jeste kroz OpenMP direktive.

Koliko niti?

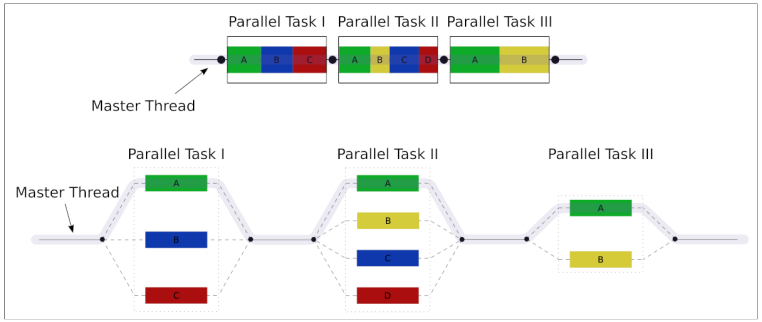
- Koliko mi hoćemo.
- U praksi, broj niti jako zavisi od naše svrhe. Ako imamo više niti da bi u stvari efektno čekali na više stvari istovremeno zato što smo u sistemu koji je I/O ograničen onda je broj niti neograničen, odnosno, samo ograničen sposobnošću sistema da to istrpi.
- Ovako rade sistemi, npr. web servera ili sistemi koji rade sa velikim brojem operacija nad fajlovima.
- Ovako, takođe, rade sistemi koji hoće da pruže glatko interaktivno iskustvo dok nešto rade u pozadini: onda obično postoji jedna GUI nit i više niti-radilica.
- Neki visoko interaktivni sistemi dodeljuju niti zadacima. Tipičan primer su video igre gde obično imate podelu tipa:

nit za iscrtavanje, nit za kontrolu, nit za zvuk i nit za veštačku inteligenciju, u zavisnosti od prirode engine-a koji se koristi.

Koliko niti?

- Ovde, naš cilj je maksimum brzine stoga je pravilo mnogo jednostavnije.
- Broj OpenMP niti treba da bude manji ili jednak ukupnom broju sistemskih niti na raspolaganju.
- Sistemske niti su ravne broju ‘procesora’ u kontekstu SLURM-a, tj. broju stvari koje procesor(i) u sistemu mogu da rade istovremeno.
- Što ne više? Zato što se paralelizam preko hardverske granice ostvaruje koristeći preemptivno izvršavanje što znači da umesto da dobijamo na ukupnoj ostvarenoj brzini mi je gubimo na context-switching overhead.

Fork-join metod izvršavanja



Fork-join metod izvršavanja

- Program počinje izvršavajući samo jednu stvar, sekvencijalno.
- U nekim trenucima, stvara se veći broj dodatnih niti izvršavanja (račvanje).
- Te niti izvršavanja se izvode nezavisno sve dok se ne priključe ponovo glavnoj niti kroz formu implicitne sinhronizacije barijerom.

SPMD

- Najčešća forma koji fork/join paralelizam ima u okviru OpenMP-a jeste Single Program Multiple Data, praktični rođak SIMD arhitekture koji smo pominjali pre par časova.
- To znači da svaka od niti izvršava isti kod koga samo razlikuju za nit specifične privatne promenljive (više o ovome kasnije).
- Ovo je onda način da podelimo posao na iste komade.

Alternative SPMD

- OpenMP je jednako lako konfigurisati da paralelno izvršava različite komade koda.
- Ponekad algoritam ovo zahteva, mada to nije nužno dobra ideja.
- Heterogen paralelni kod se ne uklapa tako glatko u fork/join arhitekturu zbog toga što je vreme izvršavanja nepredvidivo što znači da join može da ima nepredvidivu količinu čekanja da se izvršavanje sinhronizuje.
- Najbolje je ovo prilagoditi kontekstu algoritma.

Ugnježden paralelizam

- Moguće je račvati nit izvršavanja unutar već račvane niti izvršavanja.
- Tj. fork unutar fork-a.
- Kako se ovo izvrši zavisi od toga kako je OpenMP implementacija koja se koristi implementirana.
- Neke ignorišu dublje slojeve paralelizacije i tretiraju ih kao sekvencijalan kod, a neki izvršavaju kako je napisano dok god ima neiskorišćenih sistemskih niti.
- Ovakva forma ugnježdavanja je korisna ili u specijalizovanim algoritmima ili kada imamo računar sa jako puno paralelnih niti, možda neki čvor sa četiri EPYC

Promenljive niti

- OpenMP je baziran na modelu deljene memorije.
- Podrazumevano je da sve promenljive budu deljene tj. da svaka nit može da im pristupi.
- Određene promenljive, sa druge strane, mogu da budu podeljene tako da svaka nit ima svoj primerak.
- Ovo bi moglo da se uradi i ručno, tako što imamo nekakvu strukturu podatka gde se promenljive indeksiraju kroz broj niti, ali OpenMP to omogućava automatski.
- Neke promenljive nisu ni privatne ni globalne, doduše, no mešaju ta dva pristupa. Ovo je od koristi kada se rade operacije redukcije, o čemu više kasnije.

Sistemske promenljive i OpenMP

- OpenMP gleda vrednosti sistemskih promenljivih ne bi li odredio svoje ponašanje.
- Sistemske promenljive mogu biti nameštene:
  - U profilu korisnika.
  - U skripti koja pokreće aplikaciju.
  - Na komandnoj liniji.

Sistemske promenljive i OpenMP

Promenljiva	Tip vrednosti	Značenje
OMP_NUM_THREADS	Broj	Broj istovremenih niti
OMP_DYNAMIC	Bulova	Dinamički menja broj istovremenih niti. Može povećati efikasnost kroz adaptaciju, ali ima cenu u performansama.
OMP_SCHEDULE	Reč.Broj	Tip rasporeda izvršavanja praćen dimenzijom particije izvršavanja. Više o tome kasnije.
OMP_NESTED	Bulova	Da li imamo ugnježdeni paralelizam ili ne. Može da nam pomogne

Promenljiva	Tip vrednosti	Značenje
		da upravljamo distribucijom niti u komplikovanim situacijama, ali ima fiksnu cenu u overhead-u.

Promenljiva	Tip vrednosti	Značenje
OMP_MAX_TASK_PRIORITY	Broj	Najveći sistemski prioritet koji se dodeljuje zadacima.
OMP_STACKSIZE	Broj praćen sa B, K, M, ili G	Veličina sistemskog steka za jednu nit izvršavanja

Sistemske promenljive i OpenMP

Promenljiva	Tip vrednosti	Značenje
OMP_CANCELLATION	Bulova	Da li ‘cancel’ direktiva radi ili ne.
OMP_MAX_ACTIVE_LEVELS	Broj	Koliko ugnježdenih regiona je dozvoljeno. Podrazumevano je da je neograničeno.

Bibliotečke rutine

- OpenMP se sastoji od: sistemskog okruženja izvršavanja, biblioteka, i direktiva.
- Biblioteke su klasične C biblioteke i nude funkcionalnost aplikacije kroz funkcije.
- Fajl zaglavlja za C biblioteku je `omp.h`

Bibliotečke rutine

```
int n = omp_get_num_threads(); // koliko ima
        niti
int k = omp_get_thread_num(); // koja je tekuća
        nit koja ovo izvršava
```

Primetite omp_ prefiks i snake case imena umesto camel case

Čemu direktive?

- OpenMP ima nezavidan zadatak
- C je fundamentalno napravljen sa idejom jednostrukog izvršavanja
- To je ugrađeno u sam jezik
- Većina sistema za paralelnog izvršavanje radi tako što nas tera da eksplicitno pravimo strukture podataka koje predstavljaju jedinice paralelnog izvršavanja, te ih pokrećemo sami.
- Ovo čini takav kod izuzetno nezgodnim za praćenje.
- Razmišljajte o, recimo, Pthread tehnologiji

Pthreads

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

void* inc_x(void* void_ptr){
  int* x_ptr = (int*)void_ptr;
  while(++(*x_ptr) < 100);
  printf("Inkrementacija gotova\n");
  return NULL;
}
```

Užas

- Ovo je užasavajuće sa tačke gledišta dobrog programiranja
- Prvo, koristimo void pokazivače za sve: to je užasno samo po sebi
- Drugo, imamo funkciju, nešto što karakteriše ulaz i izlaz, koja nema nikakav izlaz nego samo modifikuje stanje preko nekontrolisanog pokazivača

Pthreads

```
int main(){
  int x = 0, y = 0;
  pthread_t inc_x_thread;
  if(pthread_create(&inc_x_thread, NULL,
        inc_x, &x)){
    fprintf(stderr, "Ne mogu da napravim
        nit\n");
    return 1;
  }

  while(++y < 100);
  printf("Inkrementacija Y gotova.\n");
  if(pthread_join(inx_x_thread, NULL)){
    fprintf(stderr, "Ne mogu da sačekam
        nit\n");
```

Užas 2: Užas uzvraća udarac

- Primetite kako smo potpuno invertovali tok programa: sve vreme moramo sa nitima da radimo kao sa rukavicama, sve vreme moramo da rukom proveravamo ishod niti, dve niti koje bi trebale da rade istu stvar u stvari imaju drugi tok koda, sve u svemu ovo se teško piše, teško čita, i može da krije bagove vrlo lako.
- Šta je problem? Nekompatibilne paradigme.

Šta je alternativa?

- Pa i nema je baš.
- Problem je u tome što C nije napravljen da bude proširiv.
- Ako konstrukti ugrađeni u jezik ne zadovoljavaju naš obrazac programiranja nema mnogo toga što možemo da uradimo.
- Možemo da koristimo pre-procesor da modifikujemo kod pre nego stigne kod kompajlera.
- Ovakav pristup vrlo uspešno koristi, npr. Qt koji dodaje konstrukte u C++ koji prvobitno nisu bili tu.
- Problem sa tim jeste što kod koji se zaista izvršava (i kod koji posle moramo da debagujemo) nema baš puno veze sa onim što smo napisali. Ovo je generator glavobolja.

FOSS rešenje

- GCC je open source.
- `git clone git@github.com:gcc-mirror/gcc.git` i već se bavimo razvojem kompajlera.
- Zašto jednostavno ne proširiti C funkcionalnošću koja nam treba?
- Zato što to onda nije više standardni C.
- Suptilne nekompatibilnosti između standarda i implementacije su odgovorne za neverovatne komplikacije. Plus, naš kod je sada zauvek vezan za tu, modifikovanu verziju GCC-a.
- Može li bolje?

`#pragma`

- `#pragma` je jednostavna ideja
- To je način da se direktno obratimo kompajleru i damo nekakve instrukcije.
- `#pragma` direktive su eksplicitno tu da budu nestandardne: svaki kompajler je potpuno slobodan da doda bilo koji broj svojih pragmi koje rade šta god taj kompajler hoće.
- *...sve dok ignorišu sve pragme koje ne znaju šta rade.*
- Ovo je tajni sastojak koji omogućava proširivost. Naš kod je i dalje legalan čak i ako je pun pragmi: kompajler koji ih ne podržava će samo da ih ignoriše i sve će i dalje da radi.

GCC podrška po kompajlerima

Od GCC verzije	Podržan OpenMP standard	Na jezicima
4.2.0	2.5	C/C++/Fortran
4.4.0	3	C/C++/Fortran
4.7.0	3.1	C/C++/Fortran
4.9.0	4	C/C++
4.9.1	4	C/C++/Fortran
6.1	4.5	C/C++
7	4.5	C/C++/Fortran
11	5.0	C/C++

gcc i openMP podrška

- Da li moramo onda da modifikujemo GCC?
- Naravno da ne, neko je već bio fin i to uradio umesto nas.
- Sve što je neophodno jeste da kompajliramo naš kod sa opcijom -fopenmp i dobijemo svu OpenMP podršku koja nam treba, a bibliotečke funkcije pruža libgomp

Najosnovniji OpenMP program

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

int main(){
  #pragma omp parallel
  {
    printf("Hello World\n");
  }
  return 0;
}
```

Napomena: Pragme nisu, u stvari, magične.

- Lako je zaboraviti da, koliko god da su zgodne za programiranje, pragme i konstrukti paralelnog programiranja koje donose, sve se to i dalje izvršava na istom procesoru kao i sav naš drugi kod.
- Pre ili kasnije to postanu pozivi nad funkcijama, komande kontrole toka itd.

Pre pragmi

```c
#pragma omp parallel;
{
  body;
}
```

Posle pragmi

```c
void subfunction(void* data){
  use data;
  body;
}

setup data;
GOMP_parallel_start(subfunction, &data,
        num_threads);
subfunction(&data);
GOMP_parallel_end();
```

Napomena: Pragme nisu, u stvari, magične.

- Ipak, ako mi sve dobro programiramo i ako je FSF sve dobro programirao trebalo bi da ne moramo puno da mislimo o tome šta to OpenMP radi iza kulisa.
- Ne puno, ali pomalo.
- Ne valja da zaboravimo da su naši alati *alati.*

Osnovna sintaksa

- Svaka pragma za OpenMP počinje sa `#pragma omp`
- Zatim ide ključna reč pragme koja definiše šta ta pragma radi praćena parametrima u zagradi (ako ih ima)
- Moguće je pragme slagati u jednoj direktivi radi uštede prostora.

`#pragma parallel`

- Maksimalno račva izvršavanje pred ulazak u predstojeći izraz/blok i izvršava ga u onoliko niti koliko je specificirano.
- Nit se implicitno račva na početku, a sinhronizuje na kraju. `

`#pragma private`

- Ova pragma definiše niz promenljivih (definisanih u nizu u zagradama) kao privatne za nit koja ih koristi.
- Navodi se posle ‘parallel’ direktive i odnosi se na niti tako stvorene.

Paralelna for petlja

- Najčešća forma paralellizacije jeste podela iteracija for petlje među nitima. Ako je svaki ciklus petlje nezavisan, onda stepen paralelizma zavisi samo od ograničenja našeg hardvera.
- Prirodno, OpenMP ima metode koje olakšavaju ovako nešto.

Sekvencijalno zbrajanje nizova

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

int main(){
  const int N = 20;
  int nthreads, threadid, i;
  double a[N], b[N], result[N];
  for(int i = 0; i < N; i++) {a[i] = 1.0 * i;
        b[i] = 2.0 * i;}
  for(int i = 0; i < N; i++){
    result[i] = a[i] + b[i];
  }
}
```

Paralelno zbrajanje nizova

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

int main(){
  const int N = 20;
  int nthreads, threadid, i;
  double a[N], b[N], result[N];
  for(int i = 0; i < N; i++) {a[i] = 1.0 * i;
        b[i] = 2.0 * i;}
  #pragma omp parallel
  {//fork
    #pragma omp for
    for(int i = 0; i < N; i++){
      result[i] = a[i] + b[i];
    }
```

Opservacije

- Primetite da je paralelno izvršavanje i for petlja različita stvar
- Paralelno izvršavanje je baš to.
- For petlja je konstrukt koji deli rad između niti.
- Takođe, valja primetiti da je brojačka promenljiva automatski privatna, što je zgodno.
- Uprkos tome što su paralelizacija i for-deljenje odvojene operacije moguće je (i poželjno je!) da ih pišemo zajedno

Paralelno zbrajanje, kraće

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

int main(){
  const int N = 20;
  int nthreads, threadid, i;
  double a[N], b[N], result[N];
  for(int i = 0; i < N; i++) {a[i] = 1.0 * i;
        b[i] = 2.0 * i;}
  #pragma omp parallel for
  for(int i = 0; i < N; i++){
    result[i] = a[i] + b[i];
  }
}
```

Koliko ima niti? I koja radi na čemu?

- Ovde se valja podsetiti `OMP_SCHEDULE` promenljive.
- Ona definiše podrazumevano ponašanje `for` direktive.
- `SCHEDULE` mehanizam ima za cilj, jednostavno, da sve indekse for petlje od 0 do N-1 podeli na neki broj regiona koji se ne preklapaju.
- Moguće je za svaku for petlju kontrolisati kako će to da uradi kroz schedule direktivu koja u zagradama ima prvo ključnu reč a zatim veličinu regiona.

Ključne reči schedule direktive

Ključna reč	Značenje
static	Iteracije se dele u komade veličine koja je specificirana, ako je specificirana, u suprotnom se veličina računa tako što se ukupan broj iteracija podeli brojem niti. Zatim se tako definisani regioni dele među nitima koristeći “round robin” pristup.
dynamic	Kao prethodno, ali osim na početku, niti se dele u first-come first-served pristupu gde niti traže još posla kada završe rad.
guided	Kao prethodno, ali veličina regiona je fleksibilna i proporcionalna je broju nedodeljenih iteracija podeljenih sa brojem dostupnih niti uz minimum ravan podešenoj veličini regiona.
auto	Kompajler/runtime bira šta se izvršava i kada.
runtime	Omogućava da se schedule podesi iz koda koristeći void omp_set_schedule(omp_sched_t kind, int chunk_size);

Monotono i nemonotono izvršavanje

- Specifikaciju `schedule` direktive može pratiti ključna reč `monotonic` i ključna reč `nonmonotonic`.
- `Monotonic` znači da svaka nit izvršava dodeljene iteracije u redosledu strogo povećavajuće vrednosti brojača
- `Nonmonotonic` znači da svaka nit izvršava dodeljene iteracije u proizvoljnom, nedeterminističkom redosledu.

Alternativa `SPMD` modelu: sekcije

- Umesto da niti izvršavaju fundamentalno isti kod (nad različitim podacima) kroz deljenje posla u nitima alternativa su sekcije.
- Ideja je jednostavna: u okviru paralelnog regiona se napravi više blokova anotiranih sa 'section' pragmom.
- Među nitima se onda ravnomerno raspoređuju one koje rade različite sekcije istovremeno i paralelni region se join-uje kada su sve sekcije obavljene.
- Očigledno, redosled izvršavanja apsolutno nije garantovan.

Sekcije

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

int main(){
  #pragma omp parallel
  {
    #pragma omp sections
    {
      //nulti paralelni blok
      #pragma omp section
      {
        //prvi paralelni blok
      }

      #pragma omp section
```

Sinhronizacija

- OpenMP ima vrlo labavu komunikaciju između niti
- To je potencijalno odlično budući da omogućava da kod koji je zamalo isti kao sekvencijalni ima sve prednosti paralelizacije, i zaista, ako samo koristimo paralelizaciju operacija u petljama nema šta puno da se brinemo.
- Dosta programa radi ovako. Numpy na primer: cela tajna je multiprocesorsko jezgro za rad sa vektorima pisano u C-u koje se onda koristi u fundamentalno sekvencijlanim algoritmima.
- Ovo je maksimum koristi za minimum glavobolje.
- Ali šta ako hoćemo još brzine i fleksibilnije algoritme?

Sinhronizacija

- Onda moramo kontrolisati “labavu” komunikaciju između niti.
- Treba da bude onoliko otvorena koliko može, naravno, ali nikako ne preko toga.
- U suprotnom imamo grozomorne probleme gde sav kod udara po istoj memoriji i ona završi u nedefinisanom stanju.
- Nedefinisana stanja valja izbeći.

Nedefinisana stanja

```
ERROR: System attempted to parse HTML with
regular expression; system returned Cthulhu.
```

Uvećavanje promenljive kao primer haosa

- Mi napišemo
  - `Thread A: i++`
  - `Thread B: i++`
- Mi dobijemo
  - A očita i sa vrednošću 7
  - B očita i sa vrednošću 7
  - B poveća 7 na 8 i smesti ga u i
  - A poveća 7 na 8 i smesti ga u i
- Komplikacija: Zbog toga kako procesori rade, ovo ne može da se desi baš ovako, *možda,* ali je zgodan primer.

Implicitna sinhronziacija

- Svaki paralelni blok je implicitno sinhronizovan, budući da sve niti moraju da sačekaju da se sve ostale niti završe pre nego se izvršavanje nastavi.
- Ovo omogućava da delimo izvršavanje na sekcije koje zavise jedna od druge.
- Centralni problem paralelizacije jeste, na kraju krajeva, to što u sekvencijalnom izvršavanju relacija sekvence izvršavanja i relacija zavisnosti su ista stvar.
- Paralelizacija je razdvajanje te dve relacije gde je to moguće.
- Negde, nešto mora da se završi da bi se kasniji deo koda izvršio (nije moguće koristiti neki podataka pre nego što je izračunat osim jako blizu površini horizonta Košija rotirajućih crnih rupa sa Kerovom metrikom. Možda.)

Eksplicitna sinhronizacija

- Moguće je naterati niti da se sinhronizuju kroz direktive i to:
  - Direktiva kritičnog regiona
  - Direktiva glavne niti
  - Direktiva barijere
  - Direktiva jednostrukog izvršavanja

Direktiva kritičnog regiona

- Direktiva kritičnog regiona (`#pragma omp critical`) definiše blok koda kao kritičan.
- Kritičan kod je takav da se u njemu u jednom trenutku može naći samo jedna jedina nit.
- Ovo nas štiti baš od onog problema sa inkrementacijom promenljive, budući da sve što treba da uradimo jeste da se postaramo da je I++ u kritičnom regionu i znamo da će cela operacija biti završena odjednom.

Direktiva glavne niti

- Direktiva glavne niti (`#pragma omp master`) definiše blok koda koji je takav da ga može izvršiti samo i isključivo glavna nit.
- Svaka nit-radilica koja naleti na ovaj blok će ga ignorisati kao da nije tu.
- To znači da ovakav kod za razliku od većine sinhronizacija ne usporava izvršavanje.

Direktiva barijere

- Direktiva barijere (`#pragma omp barrier`) služi da natera niti da se sinhronizuju. Gde god da se stavi u kodu definiše graničnu tačku.
- Kada bilo koja nit stigne to granične tačke pauzira dok sve druge niti nisu, takođe, stigle do granične tačke. Tek onda se izvršavanje nastavlja.

Direktiva jednostrukog izvršavanja

- Direktiva jednostrukog izvršavanja (`#pragma omp single`) definiše blok koda koji se izvršava u samo jednoj niti.
- Radi kao direktiva glavne niti osim što:
  - Važi za prvu nit koja stigne do nje.
  - Zahteva sinhronizaciju na kraju bloka, tj. druge niti će čekati dok se ne završi izvršavanje bloka pod direktivnom jednostrukog izvršavanja.
- Ova druga razlika se može isključiti kroz upotrebu `nowait` dodatka iza single direktive (`#pragma omp single nowait`)

Redukcija

- Neke operacije su lake za paralelizaciju pošto uzimaju n ulaza a proizvode n ili više izlaza koji su nezavisni.
- Onda ih je lako iscepati na delove.
- Recimo da hoćemo da izračunamo sinus svake vrednosti u nekom ogromnom nizu: vrednost elementa 400494 ne zavisi od vrednosti elementa 403222 i nije bitno da li ih računa jedna nit ili više.
- Šta kada imamo zavisnost?
- Pa, ponekad ništa. Neki problemi jednostavno ne mogu da se paralelizuju ili zahtevaju lukavstvo (kako paralelizovati računanje Fibonačijevih brojeva?)
- Ali ponekad je zavisnost malo pravilnija i moguće je koristiti za to specijalizovane konstrukte.

`#pragma reduction`

- Postoji `reduction` pragma sa sintaksom `#pragma reduction(op : var)`
- Tu je op operator koji može biti: `+, *, -, /, &, ^ |`
- `var` je promenljiva za rezultat
- Ima samo smisla u paralelnom kontekstu
- Mora se odnositi na blok u kome se pojavljuje komad koda koji izgleda ovako: `var = var op izraz`

Trivijalan primer

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

int main(){
  const int N = 1024;
  int i;
  double a[N], s = 0;
  for(i = 0; i < N; i++) a[i] = 1.0 * i;
  #pragma omp parallel for default(shared)
          private(i) schedule(static, 4)
          reduction(+ : s)
  for(i = 0; i < N; i++){
    s = s + a[i];
  }
```

Fibonačijevi brojevi?

- Fibonačijevi brojevi su divan primer situacije gde tradicionalne tehnike paralelizacije nisu osobito korisne.
- Konvencionalni algoritam za njih je fundamentalno serijski.
- Šta znači fundamentalno ovde?
  - N-ti korak zavisi od n-1 i n-2 koraka.
  - N-1 korak zavisi od n-2 i n-3 koraka
  - N-2 korak zavisi od n-3 i n-4 koraka
  - itd.
- Kako onda?
  - Varanje!

Varanje

- Reformulišemo problem tako da odgovara našim potrebama.
- Postoji rekurzivna implementacija računanja fibonačijevog broja, ali takođe postoji direktna formula

$$F_n = \sum_{k=0}^{\lfloor \frac{n-1}{2} \rfloor} \binom{n-k-1}{k}$$

Varanje

- Da li ovo pomaže? Ne skroz. Računanje binomijalnih koeficijenata je bazirano na računanju faktorijela što opet stvara umeren problem. Može se paralelizovati (to je proizvod niza umesto sume), ali zahteva ugnježdavanje paralelizama što može da bude nepotpuno podržano na našoj arhitekturi.
- Možemo da budemo direktniji, možda, naročito ako hoćemo da računamo fibonačijeve brojeve zaredom.

Varanje

$F(n+2) = F(n+1) + F(n)$ i naravno
$F(n+3) = F(n+2) + F(n+1) = 2 \cdot F(n+1) + F(n)$
$F(n+4) = F(n+3) + F(n+2) = 3 \cdot F(n+1) + 2 \cdot F(n)$
$F(n+5) = F(n+4) + F(n+3) = 5 \cdot F(n+1) + 3 \cdot F(n)$
$F(n+6) = F(n+3) + F(n+2) = 8 \cdot F(n+1) + 5 \cdot F(n)$

Varanje

Stoga, uopšteno:

$$F(n+k) = F(n+1) \cdot F(k) + F(n) \cdot F(k-1)$$

Varanje

- Sada je naš zadatak jednostavan: izračunamo m Fibonačijevih brojeva i čuvamo ih u memoriji
- Zatim sve brojevo od $m+1$ do $2m$ računamo u m-tostrukoj paraleli uzimajući $n = m$ a $k = 1..m$.
- Onda samo pomerimo da je $n = 2m+1$ i ponovimo proces
- Drugim rečima, naš algoritam je serijski korak prekomputacije, a zatim serijsko ponavljanje m-tostruko paralelizovanog računanja bloka vrednosti
- Stepen paralelizacije je, onda, praktično neograničen.
- Pobeda!
- Više o ovakvim egzibicijama za čas-dva