Увод

U savremenom računarskom dobu, oblast paralelnog računarstva a posebno paralelnog programiranja postale su veoma važne iz nekoliko razloga. Kao prvo, naši računari su već paralelni. Drugo, paralelizam može biti od velike praktičnog značaja kada je u pitanju rešavanje računarski zahtevnih problema iz različitih oblasti. I konačno, postoji inercija u projektovanju savremenih računara koja održava paralelizam kao neophodan deo budućih računara.

Svaki današnji komercijalni računar, tablet i pametni telefon sadrže procesor sa više jezgara, od kojih je svako sposobno da pokreće sopstveni tok instrukcija. Ako su tokovi dizajnirani tako da jezgra sarađuju u pokretanju aplikacije, aplikacija se pokreće paralelno i može biti znatno ubrzana. Paralelizam omogućava da se iskoriste prednosti visokoperformansnih računarskih sistema. Ova potreba proizlazi iz sve veće složenosti problema koje želimo rešiti i ubrzanja koje možemo postići istovremenim izvršavanjem različitih delova zadatka.

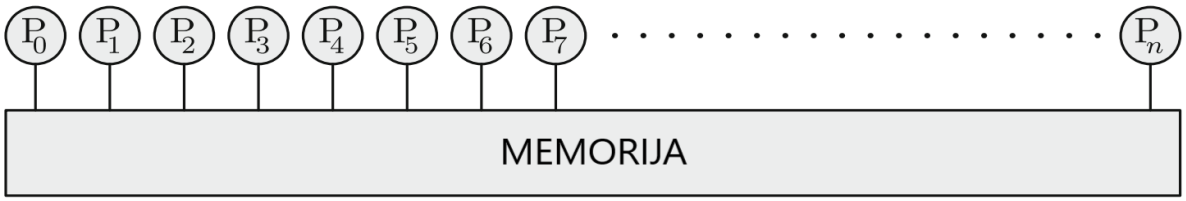
Jedna od efikasnih tehnika za implementaciju paralelizacije je OpenMP (*Open Multi-Processing*). OpenMP predstavlja otvoreni standard za paralelno programiranje, dizajniran sa ciljem da olakša razvoj efikasnih i prenosivih paralelnih programa. Ovaj standard pruža jednostavan i pristupačan model programiranja baziran na deljenju resursa između niti (threads), omogućavajući programerima da istovremeno izvršavaju određene delove koda. Kroz praktična rešenja, rad se fokusira na istraživanje osnovnih principa i mogućnosti OpenMP-a, u cilju temeljnog razumevanja i pronalaženja načina za povećanje performansi sistema upotrebom paralelne obrade.

U narednim poglavljima, razmatraće se osnovni koncepti paralelnog programiranja korišćenjem OpenMP-a, biće istražene važne funkcionalnosti i predstaviće se praktični primeri koji ilustruju primenu ovog standarda u različitim kontekstima. Takođe, pokazaće se kako kombinacija direktiva kompajlera i bibliotečkih funkcija može da obezbedi okruženje za implementaciju programa sa više niti. Kroz analizu osnovnih principa i demonstraciju konkretnih primera, cilj rada je da omogući da se stekne solidno razumevanje OpenMP-a kao alata za efikasno paralelno programiranje.

1. Model za programiranje sa zajedničkom memorijom

Model multiprocesora sa deljivom memorijom, kako se vidi iz perspektive programera, sastoji se od više odvojenih procesora koji dele jednu glavnu memoriju, kao što je prikazano na Slici 1.1. Pošto svaki procesor upravlja svojom sopstvenom kontrolnom jedinicom, svi oni mogu direktno da pristupe bilo kojoj tački u glavnoj memoriji i izvršavaju različite instrukcije za različite podatke u bilo kom trenutku. Ovaj pristup je poznat kao MIMD, ili višestruki tok podataka višestruki tok instrukcija, prema Flinovoj taksonomiji paralelnih sistema.

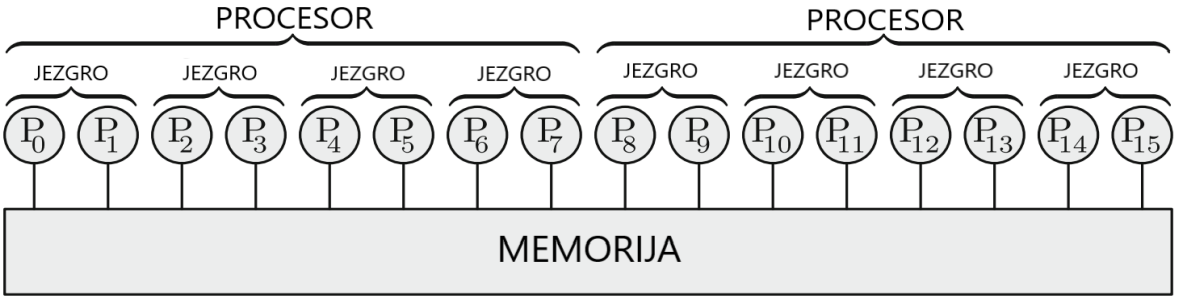
Mnogi elementi poput keš memorije ili interne strukture glavne memorije su izostavljeni iz ovog modela kako bi ga učinili jednostavnim i univerzalnim, uprkos činjenici da su od najveće važnosti kada se paralelni sistem u potpunosti koristi za postizanje maksimalne moguće brzine. Kada se shvate specifikacije paralelnog sistema, korišćenje jednostavnog i opšteg modela olakšava kreiranje i implementaciju prenosivih programa koji se mogu optimizovati za određeni sistem.



Slika 1.1 Model multiprocesora sa zajedničkom memorijom

Većina savremenih CPU-a uključuje više odvojenih procesorskih jedinica ili jezgara, što ih čini višejezgarnim procesorima. Štaviše, moderni CPU-ovi imaju simultani višenitni režim rada (SMT), što omogućava svakom jezgru da izvršava brojne nezavisne tokove instrukcija, poznate kao niti, praktično istovremeno. Svako jezgro svakog procesora se programeru pojavljuju kao brojna logička jezgra, od kojih je svako sposobno da nezavisno pokreće program ili nit unutar programa.

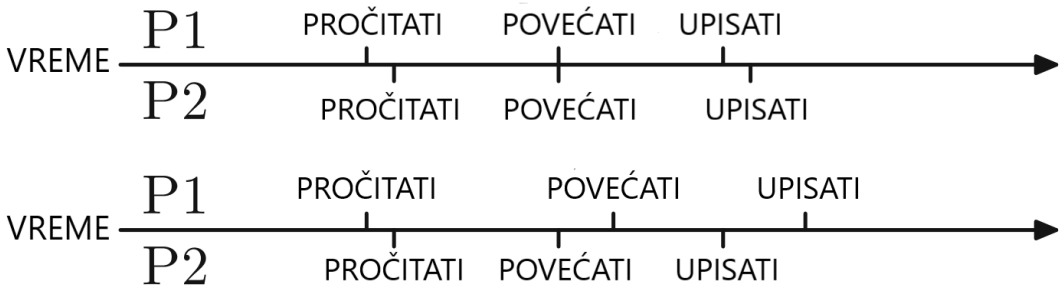
Današnji procesori za mobilne uređaje, desktop računare i servere obično uključuju 2 do 24 jezgra i, sa mogućnošću višenitnog rada, sposobni su da podrže do 48 niti odjednom. Na primer, mobilni Intel i7 dvojezgarni CPU sa hiper-nitnošću (Intelov SMT) ima 2 (fizička) jezgra, što omogućava 4 logička jezgra. Slično ovome, kao što je ilustrovano na Slici 1.2 sistem sa dva ovakva CPU-a nudi 16 logičkih jezgara, dok četvorojezgarni Intelov Xeon Phi procesor sa hiper-nitnošću nudi 8 logičkih jezgara. Svako logičko jezgro može nezavisno da pokreće sopstvenu nit ako se zanemari zajednička upotreba resursa poput magistrale ili keša. Bez obzira na to kako je sistem fizički implementiran, programer može pretpostaviti da ima 16 logičkih jezgara, od kojih svako funkcioniše kao poseban procesor, kao što je prikazano na Slici 1.1, gde je n = 16.



Slika 1.2 Paralelni sistem sa dva četvorojezgarna CPU-a koji podržavaju istovremeni višenitni rad sadrži 16 logičkih jezgara koja su sva povezana na istu memoriju

Mnogojezgarni procesori, koji imaju desetine ili stotine fizičkih jezgara, nude još jednu opciju za višejezgarne procesore. Na primer, Intelov Xeon Phi ima 60 do 72 fizička jezgra sa 240 do 288 niti koje se izvršavaju simultano.

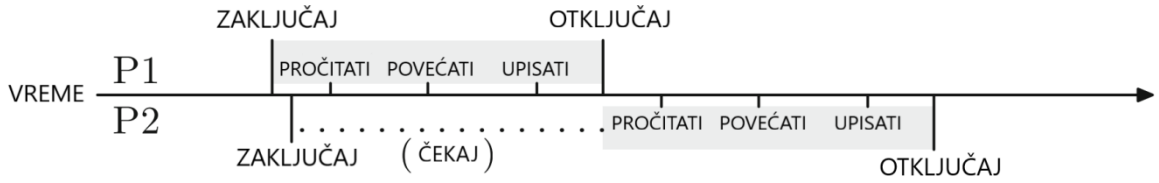
Postoji problem koji je povezan sa kapacitetom savremenih sistema da pokreću nekoliko niti istovremeno na različitim procesorima ili (logičkim) jezgrima. Uslov „trke" (race-condition), ili scenario u kojem ishod zavisi od preciznog vremena pristupa kod čitanja i upisa na istu adresu u glavnoj memoriji, može se desiti ako je pojedinačnim nitima dozvoljen pristup bilo kojoj memorijskoj lokaciji u glavnoj memoriji i dozvoljeno im je da izvršavaju instrukcije nezavisno. Recimo, na primer, da dve niti moraju povećati vrednost uskladištenu na istoj memorijskoj lokaciji, jedna za 1, a druga za 2, tako da se na kraju poveća za 3. Vrednost se čita od strane svake niti, povećava se, a zatim se upisuje ponovo. Pravi ishod se dobija ako jedna nit prvo izvrši ove tri instrukcije, a zatim druga. Međutim, pošto niti izvršavaju instrukcije odvojeno, redosled kojim se ove tri instrukcije izvršavaju od strane svake niti može se ponekad preklapati, kao što je prikazano na Slici 1.3. U takvim slučajevima, ishod je i pogrešan i nedefinisan: u svakom scenariju, vrednost memorije će biti povecćana za 1 ili 2, ali ne i za 1 i za 2.



Slika 1.3 Dva primera uslova trke kada dve niti pokušavaju da povećaju vrednost na istoj lokaciji u glavnoj memoriji

Ekskluzivni pristup deljenoj adresi u glavnoj memoriji mora biti obezbeđen korišćenjem tehnike kao što je zaključavanje pomoću semafora ili atomski pristup korišćenjem instrukcija čitanje-modifikovanje-upisivanje, kako bi se sprečio uslov trke. Ako se koristi zaključavanje, kao što je prikazano na Slici 1.4, svaka nit mora zaključati pristup deljenoj memorijskoj lokaciji pre nego što je promeni i otključati je nakon toga.

Ako nit pokuša da zaključa nešto što je druga nit već zaključala, mora sačekati dok druga nit to ne otključa. Ova strategija čini da jedna nit čeka, čime se obezbeđuje tačnost.



Slika 1.4 Sprečavanje uslova trke kao što je ilustrovano na slici 3.3 pomoću zaključavanja

Na Slici 1.1 i 1.2, periferni uređaji nisu prikazani. Pretpostavlja se da sve niti mogu pristupiti svim perifernim uređajima, iako je softver na kraju odgovoran za određivanje koja nit može pristupiti svakom uređaju u bilo kom trenutku.

2. Pisanje višenitnih programa s OpenMP-om

Na multiprocesoru s deljenom memorijom, paralelni program obično se sastoji od određenog broja niti. Tokom izvođenja programa, broj niti se može promeniti, ali svaka nit uvek radi na jednom logičkom jezgru. Sistem nije u potpunosti iskorišćen ako ima manje niti od logičkih jezgara jer su neka logička jezgra neaktivna. Operativni sistem koristi multitasking među nitima koje se izvršavaju na istim logičkim jezgrima, ako ima više niti nego logičkih jezgara. Operativni sistem može da izvrši balansiranje opterećenja tokom izvršavanja programa, što uključuje premeštanje niti iz jednog logičkog jezgra u drugo u nastojanju da održi jednako korišćenje svih logičkih jezgara.

Mnoge biblioteke i strukture se mogu koristiti za pisanje višenitnih programa na različitim programskim jezicima. Na UNIX-u, na primer, praktično svaki respektabilan programski jezik može da koristi pthreads, ali rezultujući program je pun detalja niskog nivoa koje kompajler može da obradi i nije prenosiv. Zbog toga je poželjno koristiti alat posebno namenjen za kreiranje paralelnih programa. Jedan primer ovakvog alata je OpenMP, okruženje za paralelno programiranje koje dobro funkcioniše za kreiranje programa koji će raditi paralelno na platformama sa deljivom memorijom. OpenMP-ov aplikciono programski interfejs (API - application programming interface) čine:

- Shell promenljive,
- funkcije podrške i
- direktive kompajlera.

Direktive kompajlera OpenMP-a pružaju kompajleru informacije o paralelizmu u izvornom kodu i daju uputstva za generisanje paralelnog koda odnosno prevođenje izvornog koda u višenitni. Direktive su uvek navedene kao #pragmas u C/C++. Programeri mogu da iskoriste i upravljaju paralelizmom u toku izvršenja programa zahvaljujući funkcijama podrške. Shell promenljive (ili varijable) omogućavaju prilagođavanje kompajliranih programa određenom paralelnom sistemu.

2.1 Izrada i korištenje OpenMP aplikacije

Listing 2.1 sadrži osnovni program koji će se koristiti za demonstraciju nekoliko tipova OpenMP API elemenata.

U početku se ovaj program izvršava kao jedna nit, ispisujući pozdravnu poruku. Trenutnoj niti se pridružuje nekoliko dodatno kreiranih niti kada izvršenje dostigne *omp parallel* direktivu. Sve niti, početna i novostvorene niti, zajedno čine skup odnosno tim niti. Svaka nit u novouspostavljenom timu niti izvršava naredbu odmah nakon direktive: u ovom primeru ona samo ispisuje svoj jedinstveni broj niti dobijen pozivanjem OpenMP funkcije *omp_get_thread_num*. Kada sve niti to urade, niti kreirane omp parallel direktivom se prekidaju i program nastavlja kao jedna nit koja štampa jedan znak novog reda i završava program izvršavanjem *return* 0.

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

int main() {
    printf("Hello world :");
    #pragma omp parallel
        printf(" %d", omp_get_thread_num());
    printf("\n");
    return 0;

}
```

Листинг 2.1 Програм „Hello world“, OpenMP стил.

Program je pisan u Microsoft Visual Studio 2022 koji koristi Microsoft Visual C++ kompajler. Program je pokrenut tri puta kako bi se videlo šta se dobija na izlazu:

```
1. Hello world : 0 1 2 3
2. Hello world : 0 3 2 1
3. Hello world : 1 3 0 2
```

OpenMP : paralelni regioni

Paralelni region u programu se definiše kao:

`#pragma omp parallel` *[clause [[,] clause] ...]*
structured-block

Formira se tim niti i nit koja je naišla na omp parallel direktivu postaje glavna nit unutar ovog tima. Strukturirani blok izvršava svaka nit u timu. To je ili jedan iskaz (deklaracija), moguće složen, sa jednim ulazom na vrhu i jednim izlazom na dnu ili druga OpenMP konstrukcija. Na kraju, postoji implicitna prepreka (barijera), tj. tek nakon što se sve niti završe, niti kreirane ovom direktivom se prekidaju i samo master nastavlja sa izvršenjem. Paralelni region bi mogao biti preciziran listom klauzula, na primer:

- `num_threads`(*integer*) specificira broj niti koje treba da izvrše strukturirani blok paralelno.

Za kompajliranje i pokretanje programa prikazanog u Listingu 2.1 i snimljenog kao *Source.c*, koristi se Command Prompt za Visual Studio 2022 gde programi mogu biti kompajlirani i pokrenuti na sledeći način:

```
> cl /openmp /O2 Source.c
> set OMP_NUM_THREADS=8
> Source.exe
```

Broj niti u ovom programu nije jasno naveden. Dakle, vrednost shell varijable OMP_NUM_THREADS odgovara broju niti. Ako je OMP_NUM_THREADS podešeno na 8, softver može da štampa:

```
Hello, world: 2 5 1 7 6 0 3 4
```

Softver bi podesio broj niti da odgovara broju logičkih jezgara na kojima niti mogu da se izvrše, ako OMP_NUM_THREADS nije podešen. Na primer, procesor sa dva jezgra sa hiper-threading-om može da koristi četiri niti za štampanje permutacije cifara između 0 i 3.

Jednom kada se nit pokrene, planiranje i rešavanje sporova za pojedinačni standardni izlaz na kome se štampa permutacija prepušta se specifičnoj OpenMP implementaciji i, u ovom slučaju, osnovnom operativnom sistemu. Dakle, verovatno je da će svaki put kada se program izvrši, drugačija kombinacija brojeva niti biti odštampana. U našem slučaju posle pokretanja programa tri puta dobili smo tri različita rešenja koja su prikazana na Slici 2.1.

OpenMP: upravljanje brojem niti

Sledeće shell variable mogu se koristiti za kontrolu broja niti u programu nakon što je kompajliran:

- OMP_NUM_THREADS lista pozitivnih brojeva razdvojenih zarezima.
- OMP_THREAD_LIMIT pozitivan ceo broj

Prva varijabla određuje koliko niti program treba da koristi ukupno, ili koliko niti treba da koristi svaki ugnežđeni nivo paralelne obrade. Druga (koja ima prednost nad OMP_NUM_THREADS) ograničava maksimalan broj niti koje program može da koristi.

Brojem niti u programu može se upravljati pomoću sledećih funkcija:

- void omp_set_num_threads() postavlja broj niti koje se koriste u pratećim paralelnim regionima bez eksplicitne specifikacije broja niti;
- int omp_get_num_threads() vraća broj niti u trenutnom timu koji se odnosi na najdublji paralelni region;
- int omp_get_mak_threads() vraća maksimalan broj niti dostupnih za sledeće paralelne regione;
- int omp_get_thread_num() vraća broj niti pozivajuće niti unutar trenutnog tima niti.

```
C:\Users\Milos\Desktop\Faks\Diplomski openMP\COsnove>Source.exe
Hello world : 5 4 1 6 3 2 0 7

C:\Users\Milos\Desktop\Faks\Diplomski openMP\COsnove>Source.exe
Hello world : 0 4 2 5 1 3 6 7

C:\Users\Milos\Desktop\Faks\Diplomski openMP\COsnove>Source.exe
Hello world : 0 3 7 1 5 4 2 6
```

Slika 2.1 Rezultat pokretanja programa sa Listinga 2.1, uz pomoć 8 niti

2.2 Praćenje OpenMP programa – Fibonačijev niz

Retko je dovoljno koristiti razmišljanje o paralelnim algoritmima i kako ih efikasnije kodirati prilikom projektovanja, razvoja i otklanjanja grešaka u paralelnim programima. Najjednostavniji način da se razume kako OpenMP program zaista funkcioniše na sistemu sa više jezgara je da se nadgledaju i kvantifikuju performanse programa. Štaviše, ovo je najjednostavniji i najpouzdaniji metod za određivanje pravog broja jezgara koje softver koristi. Kao primer, pogledaćemo program u Listingu 2.2. Program pokreće nekoliko niti, od kojih svaka štampa jedan Fibonačijev broj izračunat pomoću dugotrajnog i naivnog rekurzivnog algoritma.

Većina operativnih sistema nam omogućava da lako izračunamo koliko dugo će program raditi. Na primer, kompajliranje i pokretanje pomenutog programa na Linux-u pomoću uslužnog programa za vreme (time) proizvodi neke Fibonačijeve brojeve, a zatim kao poslednji red izlaza, daje informacije o vremenu rada programa.

```
1  #include <stdio.h>
2  #include <omp.h>
3
4   long fib(int n) { return (n < 2 ? 1 : fib(n - 1) + fib(n - 2)); }
5
6  int main() {
7      int n = 37;
8      #pragma omp parallel
9          {
10         int t = omp_get_thread_num();
11         printf("%d: %ld\n", t, fib(n + t));
12         }
13     return 0;
14
15 }
```

Listing 2.2 Izračunavanje nekih Fibonačijevih brojeva.

Sa PowerShell-om pokrenutim u x64 Native Tools Command Prompt za Visual Studio 2022, programi mogu biti kompajlirani i pokrenuti na sledeći način:

```
> powershell
> cl /openmp /O2 fibonaciBrojevi.c
> $env:OMP_NUM_THREADS=8
> ./fibonaciBrojevi.exe
```

U PowerShell-u, vreme trajanja programa može biti izmereno korišćenjem komande Measure-Command:

```
> Measure-Command {./FibonaciBrojevi.exe}
```

Nakon pokretanja programa, dobijeni su rezultati prikazani na Slici 2.2.

```
PS C:\Users\Milos\Desktop\Faks\Diplomski openMP\Fibonaci> $env:OMP_NUM_THREADS=8
PS C:\Users\Milos\Desktop\Faks\Diplomski openMP\Fibonaci> ./FibonaciBrojevi.exe
0: 39088169
1: 63245986
2: 102334155
3: 165580141
4: 267914296
5: 433494437
6: 701408733
7: 1134903170
```

Slika 2.2 Fibonačijevi brojevi od 37. do 44.

Vreme izvršenja programa se prijavljuje u završnoj liniji izlaza (Slika 2.3).

```
PS C:\Users\Milos\Desktop\Faks\Diplomski openMP\Fibonaci> Measure-Command{./FibonaciBrojevi.exe}


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 5
Milliseconds      : 702
Ticks             : 57023914
TotalDays         : 6.5999900462963E-05
TotalHours        : 0.00158399761111111
TotalMinutes      : 0.0950398566666667
TotalSeconds      : 5.7023914
TotalMilliseconds : 5702.3914
```

Slika 2.3 Vreme izvršenja programa

Ukupno vreme koje sva logička jezgra zajedno provedu u radu softvera predstavlja se kao korisničko i sistemsko vreme. Korisničko i sistemsko vreme sabrano u gornjem primeru premašuje stvarno vreme, često poznato kao proteklo vreme ili vreme na zidnom satu. Dakle, različiti delovi programa morali su da rade istovremeno na nekoliko logičkih jezgara.

Većina operativnih sistema dolazi sa sistemskim monitorima koji pokazuju količinu posla koju svako jezgro obavlja, pored ostalih metrika. Savetuje se oprez jer većina sistemskih monitora pokazuje ukupno opterećenje svakog logičkog jezgra ili opterećenje

svih programa koji se trenutno pokreću na tom logičkom jezgru, što može da zavara prilikom razvoja OpenMP programa.

Korišćenjem sistemskog monitora, opterećenje na određenim logičkim jezgrama se može videti kada softver u Listingu 2.2 radi na sistemu koji inače ne radi. Može se videti da se opterećenje pojedinačnih logičkih jezgara smanjuje kako izvršavanje napreduje i niti se završavaju jedna za drugom. Takođe se može videti da operativni sistem periodično pomera završnu nit iz jednog logičkog jezgra u drugo kako se izvršavanje bliži kraju i ostaje samo jedna nit.

3. Paralelizacija petlji

Preporučljivo je početi sa nekim primerima koji pokazuju šta OpenMP nudi za efikasnu i prenosivu implementaciju paralelnih petlji, pošto većina CPU-intenzivnih programa za rešavanje naučnih ili tehničkih problema većinu svog vremena provodi u petlji.

3.1 Nezavisne iteracije u paralelizovanim petljama

Počinjemo sa jednostavnim primerom paralelizacije petlje kako bismo izbegli preterano komplikovanje diskusije o paralelnim petljama u OpenMP-u: razmatramo primer o štampanju svih celih brojeva od 1 do vrednosti koju odredi korisnik, recimo *max*, bilo kojim redosledom. Listing 3.1 prikazuje program koji radi paralelno.

```
1   #include <stdio.h>
2   #include <omp.h>
3
4   int main(int argc, char* argv[]) {
5       int max;
6       sscanf_s(argv[1], "%d", &max);
7       printf("Vrednost max je: \n", max);
8
9       #pragma omp parallel for
10          for (int i = 1; i <= max; i++)
11          printf("%d: %d\n", omp_get_thread_num(), i);
12      return 0;
13
14  }
```

Listing 3.1 Štampanje svih celih brojeva od 1 do *max* bez određenog redosleda

Posle pokretanje programa, rezultat programa je prikazan na Slici 3.1.

```
Unesite vrednost za max: 10
Thread 0: 1
Thread 0: 2
Thread 0: 3
Thread 2: 7
Thread 2: 8
Thread 3: 9
Thread 3: 10
Thread 1: 4
Thread 1: 5
Thread 1: 6
```

Slika 3.1 Rezultat programa iz Listinga 3.1

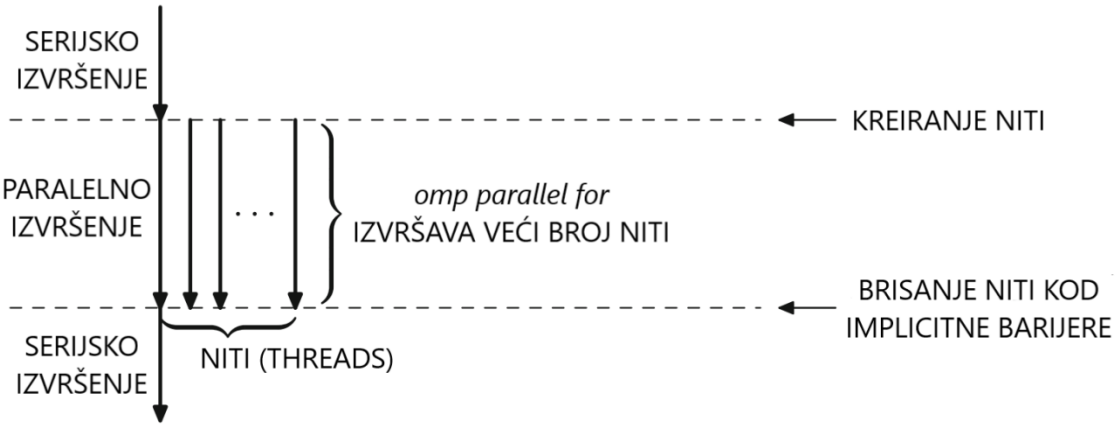
Listing 3.1 prikazuje program koji počinje kao jedna inicijalna nit. Nakon čitanja, vrednost *max* se čuva u promenljivoj max. Najvažniji deo programa, *for* petlja, koja štampa brojeve (svakom prethodi broj niti koja ga štampa) se zatim dostiže tokom izvršenja. Međutim, *omp parallel for* direktiva reda 9 zahteva da se for petlja izvršava paralelno, tj. da deli i izvršava svoje iteracije preko nekoliko niti koje se izvršavaju na svim dostupnim procesorskim jedinicama. Kao rezultat, generiše se mnoštvo slejv niti, po jedna za svaku procesorsku jedinicu koja je dostupna ili prema eksplicitnom specificiranju (osim one na kojoj radi glavna (master) nit). Originalna nit postaje glavna odnosno master nit, i zajedno sa novokreiranim slejv nitima formira se tim niti. Onda:

• iteracije paralelne *for* petlje se zatim dele između niti, pri čemu svaku iteraciju izvodi nit kojoj je dodeljena, i

• kada se sve iteracije izvrše, sve niti u timu se sinhronizuju na implicitnoj barijeri na kraju paralelne *for* petlje i sve slejv niti se prekidaju.

Na kraju, program se izvršava na sekvencijalni način, pri čemu ga glavna nit završava izvršavanjem *return* 0. Slika 3.2 pokazuje kako se izvršava program u Listingu 3.1.

Neophodno je napomenuti nekoliko stvari o programu Listinga 3.1 (i izvršavanju paralelnih for petlji uopšte). Prvo, program u Listingu 3.1 ne definiše kako iteracije treba da budu podeljene između niti. Kada se to dogodi, većina OpenMP implementacija deli prostor iteracije u delove, od kojih se svaki obrađuje jednom niti i sadrži podinterval svih iteracija. Ipak to ne mora uvek biti slučaj jer, u nedostatku bilo kakve specifikacije, OpenMP implementacija može da radi šta god želi.



Slika 3.2 Izvođenje programa za štampanje celih brojeva kao što je implementirano u Listingu 3.1.

OpenMP: paralelne petlje

Paralele *for* petlje su deklarisane kao:

```
#pragma omp for [clause [[,] clause] ...]
    for-loops
```

Ova direktiva, koja se mora koristiti u okviru paralelnog regiona, precizira da će iteracije jedne ili više ugnežđenih *for* petlji biti izvršene od strane tima niti unutar paralelnog regiona (*omp parallel for* je skraćenica za pisanje *for* petlje koja sama obuhvata čitavu paralelnu oblast). Svaka *for* petlja među *for*-petljama koje su povezane sa direktivom *omp for* moraju biti u kanonskom obliku.
U C programu, to znači da:
• promenljiva petlje je privatna za svaku nit u timu i mora biti ili (neoznačen) ceo broj ili pokazivač,
• promenljiva petlje ne treba da se menja tokom izvršavanja bilo koje iteracije;
• uslov u *for* petlji mora biti jednostavan relacioni izraz,
• povećanje u *for* petlji mora specifikovati promenu konstantnim aditivom izraza;
• broj iteracija svih pridruženih petlji mora biti poznat pre početka od najudaljenije *for* petlje.

Klauzula je specifikacija koja dalje opisuje paralelnu petlju, na primer,
• *collapse* (ceo broj) određuje koliko je najudaljenijih *for* petlji od *for*-petlji povezano sa direktivom, pa su tako paralelizovane zajedno;
• *nowait* eliminiše implicitnu barijeru i time sinhronizaciju na kraju *for*-petlji.

Drugo, nakon podele prostora iteracije na delove, iteracije svakog pojedinačnog dela se izvode uzastopno, jedna za drugom. Treće, pošto je svakoj niti potrebna kopija promenljive *i*, promenljiva paralelne *for* petlje *i* postaje privatna u svakoj niti koja završava određeni broj iteracija. Međutim, pošto je promenljiva *max* unapred definisana i čita se samo u paralelnom odeljku, mogu je deliti sve niti.

Najvažnija stvar koju treba zapamtiti je da glavni zadatak štampanja svakog celog broja od 1 do maksimuma u bilo kojoj sekvenci može biti podeljen na N potpuno odvojenih podzadataka skoro identične veličine. U ovim situacijama, paralelizacija je zanemarljiva.

Štampanje celih brojeva se ne dešava paralelno kao što izgleda pošto je standardni izlaz serijalizovan. Dakle, sledeća je ilustracija potpuno paralelnog računanja.

OpenMP: deljenje podataka

Različite klauzule za deljenje podataka mogu se koristiti u omp parallel direktivi za specifikovanje da li i kako se podaci dele među nitima:

• *shared* (lista) specifikuje da svaku promenljivu u listi dele sve niti u timu, tj. sve niti dele istu kopiju promenljive;

• *private* (lista) specifikuje da je svaka promenljiva u listi privatna za svaku nit u timu, tj. svaka nit ima sopstvenu lokalnu kopiju promenljive;

• *firstprivate* (list) je kao private, ali svaka navedena promenljiva je inicijalizovana vrednošću koju je sadržala kada je naišla na paralelni region;

• *lastprivate* (list) je kao private, ali kada se završi paralelni region, svaka navedena promenljiva se ažurira svojom konačnom vrednošću unutar paralelnog regiona.

Ove klauzule zabranjuju navođenje bilo koje promenljive kao komponente druge promenljive. Ako nije rečeno drugačije, onda:

• automatske promenljive deklarisane izvan paralelne konstrukcije se dele,

• automatske promenljive deklarisane u okviru paralelne konstrukcije su privatne,

• statički i dinamički dodeljene promenjive se dele.

Da bi se eksplicitno sprečili uslovi trke, kao što su oni izazvani ažuriranjem deljenih promenljivih ili variranjem životnog veka *lastprivate* promenljive, OpenMP strukture moraju se posebno koristiti.

Primer: Sabiranje vektora

Razmotrimo sabiranje vektora. Listing 3.2 prikazuje funkciju odgovornu za njegovu implementaciju. Napravićemo program za testiranje. Koristićemo dugačke vektore i izvršiti mnogo vektorskih sabiranja da bismo ga izmerili i pratili jer sabiranje vektora uopšte nije kompleksan proces. Struktura funkcije *vectAdd* slična je programu za celobrojno štampanje prikazanom u Listingu 3.1: jednostavna paralelna for petlja u kojoj je izlaz jedne iteracije potpuno nepovezan sa izlazom narednih iteracija. Štaviše, različite iteracije čitaju iz i upisuju na potpuno odvojene memorijske lokacije, čime se pristupa različitim elementima niza. Zbog toga ne može biti uslova za trku.

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
double* vectAdd(double* c, double* a, double* b, int n) {
    int i;
#pragma omp parallel for
    for (i = 0; i < n; i++)
        c[i] = a[i] + b[i];
    return c;
}
int main() {
    int n = 10000;  // velicina vektora
    int num_additions = 10000;  // broj koliko puta se sabira

    double* a = (double*)malloc(n * sizeof(double));
    double* b = (double*)malloc(n * sizeof(double));
    double* c = (double*)malloc(n * sizeof(double));

    int i;     // Inicijalizacija vektora a and b i dodela neke vrednosti
    for (i = 0; i < n; i++) {
        a[i] = i;
        b[i] = 2 * i;
    }
    double start_time = omp_get_wtime();
    for (i = 0; i < num_additions; i++) {
        vectAdd(c, a, b, n);
    }
    double end_time = omp_get_wtime();
    double elapsed_time = end_time - start_time;

    printf("Vektorsko sabiranje se izvrsilo %d puta ", num_additions);
    printf("sa vektorima velicine %d in %.6f sekundi.\n", n, elapsed_time);
    free(a); // oslobadjanje memory
    free(b);
    free(c);

    return 0;
}
```

Listing 3.2 Paralelno sabiranje vektora

Sada razmotrimo štampanje svakog para brojeva od 1 do maksimuma bilo kojim redosledom, što će zahtevati dve ugnežđene *for* petlje. Jedna od dve ugnežđene petlje može biti paralelizovana, dok druga ne može jer su iteracije svake petlje nezavisne. *Omp parallel for* direktiva je pozicionirana ispred petlje koja treba da bude paralelizovana da bi se ovo uradilo. Listing 3.3, na primer, prikazuje program sa paralelnom spoljnom *for* petljom.

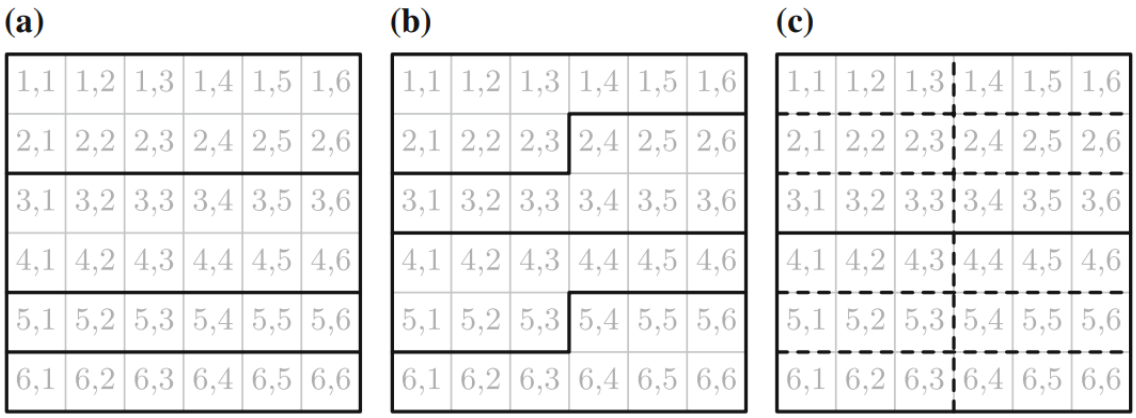
Pretpostavimo da kvadratna tabela sadrži sve parove brojeva od 1 do maksimalnog. Svaka iteracija paralelne spoljne for petlje štampa nekoliko redova iz tabele, kao što je prikazano na Slici 3.3a, ako se koristi *max* = 6 i 4 niti. Obratimo pažnju na to da prve dve niti dobijaju dvostruko više posla od druge dve, što znači da ako postoje četiri logička jezgra, poslednje dve će morati da sačekaju dok prve dve ne završe.

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

int main(int argc, char* argv[]) {
    int max, i, j;
    sscanf_s(argv[1], "%d", &max);
    printf("Vrednost max je: \n", max);
    #pragma omp parallel for
        for (i = 1; i <= max; i++)
        for (j = 1; j <= max; j++)
        printf("Thread %d: (%d,%d)\n", omp_get_thread_num(), i, j);

    return 0;

}
```

Листинг 3.3 Штампање свих парова целих бројева од 1 до *max* без одређеног редоследа паралелизацијом само најудаљеније *for* петља

Међутим, постоје још два приступа паралелизаciji угнежђенih petlji. Da bi se dve ugnežđene *for* petlje paralelizovale, prvo se mora koristiti klauzula *collapse(2)*, kao što ilustruje Listing 3.4.

Kompajler spaja tj. kombinuje dve ugnežđene *for* petlje u jednu petlju i paralelizuje je kao rezultat klauzule *collapse(2)* u redu 8. Jedna petlja koja se kreće od 1 do max^2 zamenjuje spoljnu *for* petlju koja se kreće od 1 do max i maksimalnu unutrašnju *for* petlju koje se kreće od 1 do *max*. Svaka max^2 iteracija je podeljena na sve dostupne niti.



Slika 3.3 Podela domena problema kada svi parovi celih brojeva od 1 do 6 moraju biti odštampani koristeći 4 niti: **a)** ako je samo spoljna for petlja paralelna, **b)** ako su obe for petlje paralelne zajedno, i **c)** ako su obe for petlje paralelne odvojeno

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

int main(int argc, char* argv[]) {
    int max, i, j;
    sscanf_s(argv[1], "%d", &max);
    printf("Vrednost max je: \n", max);
    #pragma omp parallel for collapse(2)
        for (i = 1; i <= max; i++)
        for (j = 1; j <= max; j++)
        printf("Thread %d: (%d,%d)\n", omp_get_thread_num(), i, j);

    return 0;

}
```

Listing 3.4 Štampanje svih parova celih brojeva od 1 do *max* bez određenog redosleda paralelizacijom obe *for* petlje zajedno.

Pošto je samo jedna petlja ona koja se sastoji od iteracija obe ugnežđene *for* petlje paralelna, izvršavanje programa Listinga 3.4 se ipak pridržava šeme sa Slike 3.2. Na primer, ako je *max* = 6, svih 36 iteracija kolapsirane pojedinačne petlje se deli na 4 niti kao što je prikazano na Slici 3.3b. Rad je podeljen među nitima ravnomernije nego što je to u programu Listinga 3.3. Listing 3.5 ilustruje alternativni pristup paralelizovanju ugnežđenih petlji: paralelizovanje svake *for* petlje nezavisno.

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

int main(int argc, char* argv[]) {
    int max, i, j;
    sscanf_s(argv[1], "%d", &max);
    printf("Vrednost max je: \n", max);
    #pragma omp parallel for
        for (i = 1; i <= max; i++) {
        #pragma omp parallel for
            for (j = 1; j <= max; j++) {
            printf("Thread %d: (%d,%d)\n", omp_get_thread_num(), i, j);

        }

    }
    return 0;

}
```

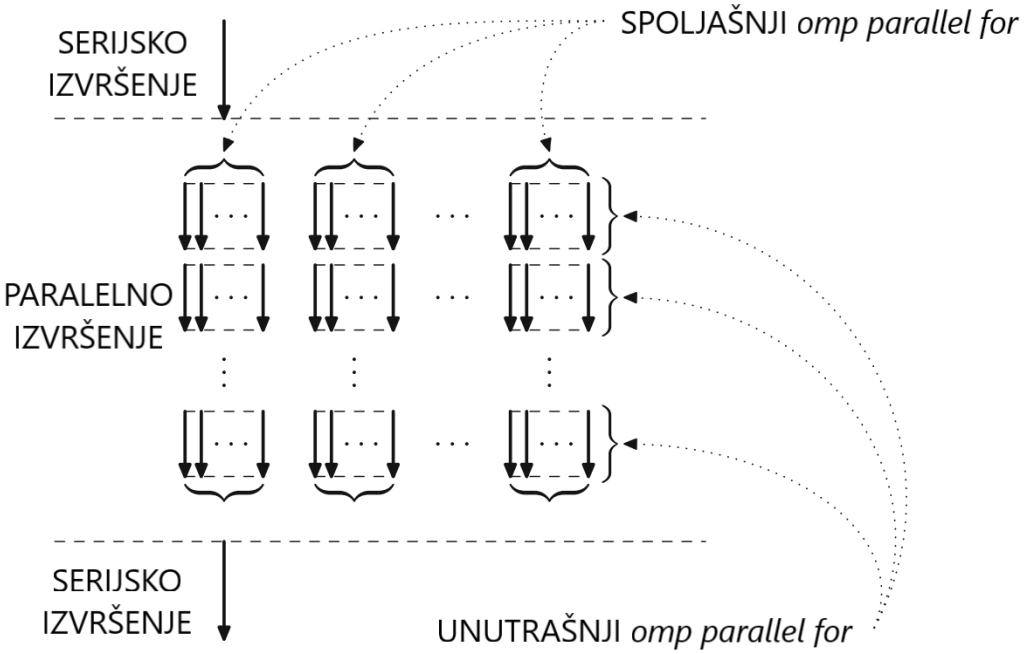
Listing 3.5 Štampanje svih parova celih brojeva od 1 do *max* bez određenog redosleda paralelizacijom svaka je ugnežđena *for* petlju posebno

OpenMP: ugnežđeni paralelizam

Ugnežđeni paralelizam je omogućen ili onemogućen podešavanjem shell promenljive
• *OMP_NESTED nested*
gde je ugnežđeno ili *true* ili *false*. U okviru programa, to se može postići korišćenjem sledeće dve funkcije:
• *void omp_set_nested(int nested)* omogućava ili onemogućava ugnežđeni paralelizam;
• *int omp_get_nested()* govori da li je ugnežđeni paralelizam omogućen ili onemogućen.

Broj niti na svakom ugnežđenom nivou može se podesiti pozivanjem funkcije *omp_set_num_threads* ili podešavanjem *OMP_NUM_THREADS*. U poslednjem slučaju, ako je data lista celih brojeva, svaki ceo broj specificira broj niti na uzastopnom nivou gnežđenja.

Da bi jedan paralelni region u okviru drugog bio aktivan kao što je prikazano u Listingu 3.5 istovremeno, prvo mora biti omogućeno ugnežđenje paralelnih regiona. Da bi se ovo postiglo, ili se postavlja *OMP_NESTED* na *true* ili poziva *omp_set_nested(1)* pre *mtxMul*. Kao što se vidi na Slici 3.4, iteracije obe petlje se izvode paralelno, odvojeno, kada se aktivira gnežđenje. Ispitajmo Sliku 3.2 i 3.4 i primetimo broj dodatnih niti koje su pokrenute i zaustavljene u sledećem primeru, pokazujući da li su ugnežđene petlje paralelne nezavisno ili ne.



Slika 3.4 Izvršenje programa za štampanje svih parova celih brojeva koristeći odvojeno paralelizovane ugnežđene petlje kao što je implementirano u Listingu 3.5

Program u Listingu 3.5 se može izvršiti postavljanjem *OMP_NUM_THREADS = 2*, 2 i uspostavljanjem tima od dve (spoljne) niti. Svaka od ovih niti će izvesti po tri iteracije

spoljne petlje, baš kao što bi to uradile da je ugnežđenje onemogućeno. Tim od dve (unutrašnje) niti je kreiran da izvrši po tri iteracije unutrašnje petlje tokom svake iteracije spoljašnje petlje, koja je potrebna za izračunavanje jedne linije tabele. Kao što se vidi na Slici 3.3 (c), tabela parova koje treba odštampati je podeljena između niti. Međutim, treba biti oprezan kada tumačimo izlaz, pošto funkcija *omp_get_thread_num* uvek vraća broj niti u odnosu na njen tim, tako da se niti svake iteracije unutrašnje petlje broje od 0 nadalje.

Primer: Množenje matrica

Množenje matrica matricama je još jedna ilustracija iz linearne algebre. Prema definiciji, tradicionalni algoritam se sastoji od dve ugnežđene *for* petlje koje izračunavaju n^2 odvojenih tačkastih proizvoda (svaki tačkasti proizvod se izračunava korišćenjem dodatne, najdublje *for* petlje). Kao rezultat toga, struktura koda za množenje matrice je slična onoj iz Listinga 3.3, 3.6 i 3.7, sa izuzetkom što se druga *for* petlja koristi za izračunavanje tačkastog proizvoda umesto originalnog, jednostavnijeg koda za izlaz para brojeva. Listing 3.6 prikazuje funkciju koja obavlja množenje dve kvadratne matrice u kojima su dve najudaljenije *for* petlje zajedno kolapsirane i paralelizovane. Listing 3.7 ilustruje alternativni metod paralelizovanja množenja matrice.

```
1  double **mtxMul (double **c, double **a, double **b, int n) {
2      #pragma omp parallel for collapse(2)
3          for (int i = 0; i < n; i++)
4              for (int j = 0; j < n; j++) {
5                  c[i][j] = 0.0;
6                  for (int k = 0; k < n; k++)
7                      c[i][j] = c[i][j] + a[i][k] * b[k][j];
8              }
9      return c;
10 }
```

Listing 3.6 Množenje matrice gde su dve najudaljenije petlje paralelne zajedno.

```
1  double **mtxMul (double **c, double **a, double **b, int n) {
2      #pragma omp parallel for
3          for (int i = 0; i < n; i++)
4              #pragma omp parallel for
5                  for (int j = 0; j < n; j++) {
6                      c[i][j] = 0.0;
7                      for (int k = 0; k < n; k++)
8                          c[i][j] = c[i][j] + a[i][k] * b[k][j];
9                  }
10     return c;
11 }
```

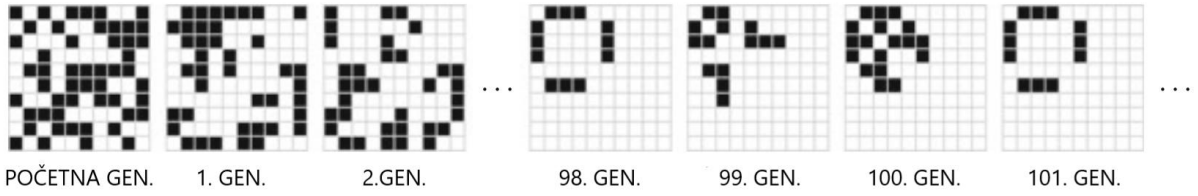
Listing 3.7 Množenje matrice gde su dve najudaljenije petlje paralelne odvojeno

Primer: Konvejeva Igra života

Konačna ravan kvadratnih ćelija, od kojih je svaka ili "živa" ili "mrtva", koristi se za igranje ove igre bez igrača. Svaki naredni korak stvara novu generaciju ćelija, gde

• svaka živa ćelija sa manje od dva suseda umire od nedovoljne populacije,
• svaka živa ćelija sa dva ili tri suseda živi dalje,
• svaka živa ćelija sa više od tri suseda umire od prenaseljenosti, i
• svaka mrtva ćelija sa tri suseda postaje živa ćelija.

Smatra se da postoji osam suseda za svaku ćeliju, četiri na svakom njenom uglu i četiri duž njenih strana. Sve naredne generacije se mogu izračunati kada se postavi prva generacija. Populacija živih ćelija može povremeno izumreti, povremeno postati statična kolonija ili naizmenično fluktuirati neograničeno. Postoje još složeniji obrasci koji se mogu pojaviti, kao što su pokretne kolonije i generatori kolonija. Evolucija oscilirajuće kolonije na ravni 10 × 10 je prikazana na Slici 3.5.



Slika 3.5 Konvejeva igra života: određena početna populacija pretvara se u oscilirajuću

Listing 3.8 prikazuje glavni deo dugog softvera koji se koristi za izračunavanje Konvejeve igre života. Da bismo to razumeli, obratimo pažnju na sledeće:
• Broj generacija koje treba izračunati je sadržan u promenljivoj *gens*.
• *size* × *size* ćelije u *plane* dvodimenzionalnom nizu sadrže trenutnu generaciju;
• sledeća generacija se izračunava kao dvodimenzionalni niz nazvan *aux_plane* koji u sebi ima ćelije *size* × *size*;
• oba dvodimenzionalna niza, tj. *plane* i *aux_plane,* se dodeljuju kao jednodimenzionalni niz pokazivača na redove dvodimenzionalne ravni;
• funkcija *neighbors* vraća broj suseda ćelije u ravni specificirano prvim argumentom veličine specificiranom drugim argumentom na poziciji precizirano trećim i četvrtim argumentom.

Listing 3.8 sadrži isti kod kao da je napisan za sekvencijalno izvršavanje, sa izuzetkom *omp parallel for* direktive. Najudaljenija *while* petlja izračunava sve generacije koje treba izračunati, dok unutrašnje dve petlje izračunavaju sledeću generaciju i čuvaju je u *aux_plane* datoj trenutnoj generaciji u ravni. Tačnije, pravila igre su implementirana u redovima 6–11 naredbe *switch*: pravila za mrtve ćelije su implementirana u slučaju *plane [i][j]==0*, dok su pravila za žive ćelije implementirana u slučaj za *plane [i][j]==1*. Nizovi se menjaju tako da generacija koja je upravo izračunata postaje tekuća generacija nakon što je nova generacija izračunata.

```
1  while (gens -- > 0) {
2      #pragma omp parallel for collapse(2)
3          for (int i = 0; i < size; i++)
4              for (int j = 0; j < size; j++) {
5                  int neighs = neighbors (plane , size , i, j);
6                  switch (plane[i][j]) {
7                      case 0: aux_plane[i][j] = (neighs == 3);
8                          break;
9                      case 1: aux_plane[i][j] = (neighs == 2) || (neighs == 3);
10                         break;
11                 }
12             }
13 char ** tmp_plane = aux_plane; aux_plane = plane; plane = tmp_plane;
14 }
```

Listing 3.8 Računanje generacija Konvejeve igre života

Omp parallel for direktiva u redu 2 ukazuje da se dve iteracije petlje *for* u redovima 3–12 mogu izvršavati istovremeno. Ispitivanje koda otkriva da svaka iteracija spoljašnje for petlje izračunava jedan red ravni koja predstavlja sledeću generaciju, baš kao kod množenja matrice, a svaka iteracija unutrašnje petlje izračunava jednu ćeliju sledeće generacije. Ne može postojati uslov trke ili zavisnosti između iteracija jer se niz *plane* samo čita, i (i, j)-ta iteracija kolapsirane petlje je jedina koja upisuje u (i, j)-tu ćeliju niza *aux_plane*.

Najvažnije, implicitna sinhronizacija se dešava na kraju paralelizovanog gnezda petlje. Bez sinhronizacije, sve druge niti bi ozbiljno poremetile izračunavanje ako bi glavna nit završila zamenu u redu 13 pre nego što druge niti unutar obe *for* petlje dovrše izračunavanje.

Na kraju, baš kao i kod množenja matrice, takođe je moguće paralelizirati dve *for* petlje nezavisno, a ne istovremeno. Međutim, najudaljenija petlja ili petlja *while* ne može se paralelizirati jer svaka iteracija – osim prve – zavisi od ishoda prethodne.

3.2 Kombinovanje rezultata paralelnih iteracija

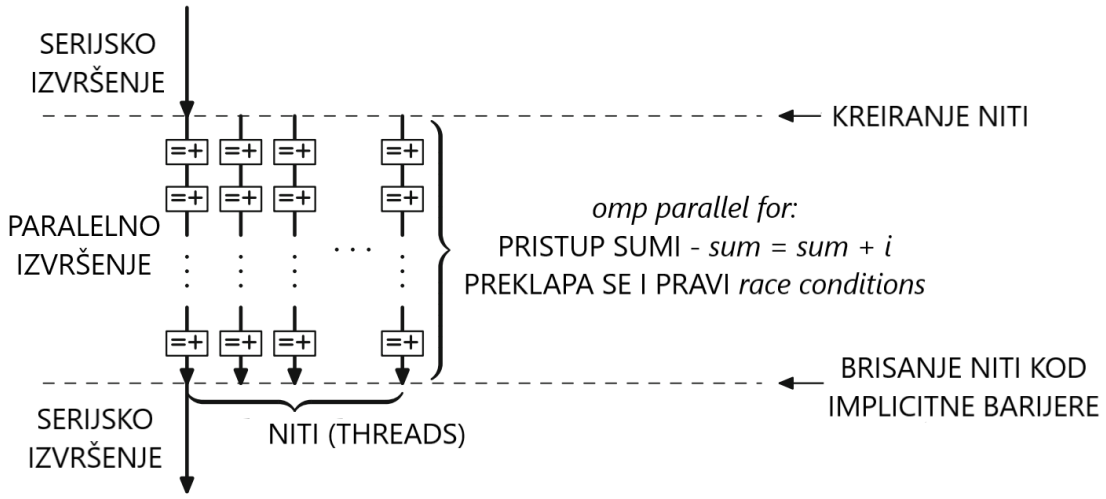
U većini slučajeva, međutim, pojedinačne iteracije petlje nisu potpuno nezavisne jer se koriste za zajedničko rešavanje jednog problema i tako svaka iteracija doprinosi svom delu. U većini slučajeva, delovi ishoda iz nekoliko različitih iteracija moraju biti integrisani zajedno.

Ako je cilj dodavanje celih brojeva iz navedenog intervala, a ne njihovo štampanje, onda svi podzadaci moraju da rade zajedno na neki način da bi generisali pravi zbir. Listing 3.9 prikazuje prvo paralelno rešenje koje vam padne na pamet. Metoda koristi jednu promenljivu *sum* za akumuliranje ishoda.

```
1  #include <stdio.h>
2
3  int main (int argc , char *argv[]) {
4      int max, i;
5      sscanf_s(argv[1], "%d", &max);
6      int sum = 0;
7      #pragma omp parallel for
8          for (i = 1; i <= max; i++)
9              sum = sum + i;
10     printf ("Suma je: %d\n", sum);
11     return 0;
12
13 }
```

Listing 3.9 Sumiranje celih brojeva iz datog intervala koristeći jednu zajedničku promenljivu — pogrešno.

Još jednom, iteracije paralelne *for* petlje su podeljene među više niti. Niti čitaju i upisuju u isti memorijski region u redu 9 koristeći istu zajedničku promenljivu *sum* na obe strane dodele. Program će verovatno zadovoljiti *"race conditions"*, kao što je prikazano na Slici 1.3, a pristupi promenljivoj *sum* se preklapaju, kao što je prikazano na slici 3.6, gde svako polje koje sadrži =+ označava zbir *sum = sum + i*.



Slika 3.6 Izvršavanje sabiranja celih brojeva kao što je implementirano u Listingu 3.9.

U stvari, postoji velika šansa da ponavljanje ovog programa sa većem brojem nitima, neće uvek rezultirati istim ishodom. Drugačije rečeno, ponekad će dovesti do netačnog ishoda. Kako bi proverili da li će program uvek rezultirati istim ishodom ili ne pokrenućemo program, koristićemo 50 niti i staviti da program sabere prvih 100 brojeva, odnosno postavićemo *max* promenljivu na 100. Program pokrećemo 10 puta kako bi uvideli devijacije u rešenju slike 3.7. Komande za pokretanje programa su:

```
>powershell
>cl /openmp /O2 SabiranjeBr.c
>$env:OMP_NUM_THREADS=50
>./sabiranjeBr.exe 100
```

```
PS C:\Users\Milos\Desktop\Faks\Diplomski openMP\Sabiranje> ./SabiranjeBr.exe 100
4880
PS C:\Users\Milos\Desktop\Faks\Diplomski openMP\Sabiranje> ./SabiranjeBr.exe 100
4935
PS C:\Users\Milos\Desktop\Faks\Diplomski openMP\Sabiranje> ./SabiranjeBr.exe 100
4963
PS C:\Users\Milos\Desktop\Faks\Diplomski openMP\Sabiranje> ./SabiranjeBr.exe 100
5050
PS C:\Users\Milos\Desktop\Faks\Diplomski openMP\Sabiranje> ./SabiranjeBr.exe 100
4776
PS C:\Users\Milos\Desktop\Faks\Diplomski openMP\Sabiranje> ./SabiranjeBr.exe 100
5050
PS C:\Users\Milos\Desktop\Faks\Diplomski openMP\Sabiranje> ./SabiranjeBr.exe 100
5050
PS C:\Users\Milos\Desktop\Faks\Diplomski openMP\Sabiranje> ./SabiranjeBr.exe 100
4919
PS C:\Users\Milos\Desktop\Faks\Diplomski openMP\Sabiranje> ./SabiranjeBr.exe 100
4923
PS C:\Users\Milos\Desktop\Faks\Diplomski openMP\Sabiranje> ./SabiranjeBr.exe 100
5050
```

Slika 3.7 Rezultati programa sa Listinga 3.9

Kao što vidimo na slici 3.7 program je pokrenut 10 puta od čega je četiri puta odštampao tačan zbir, dok je u ostalih šest slučajeva pogrešio.

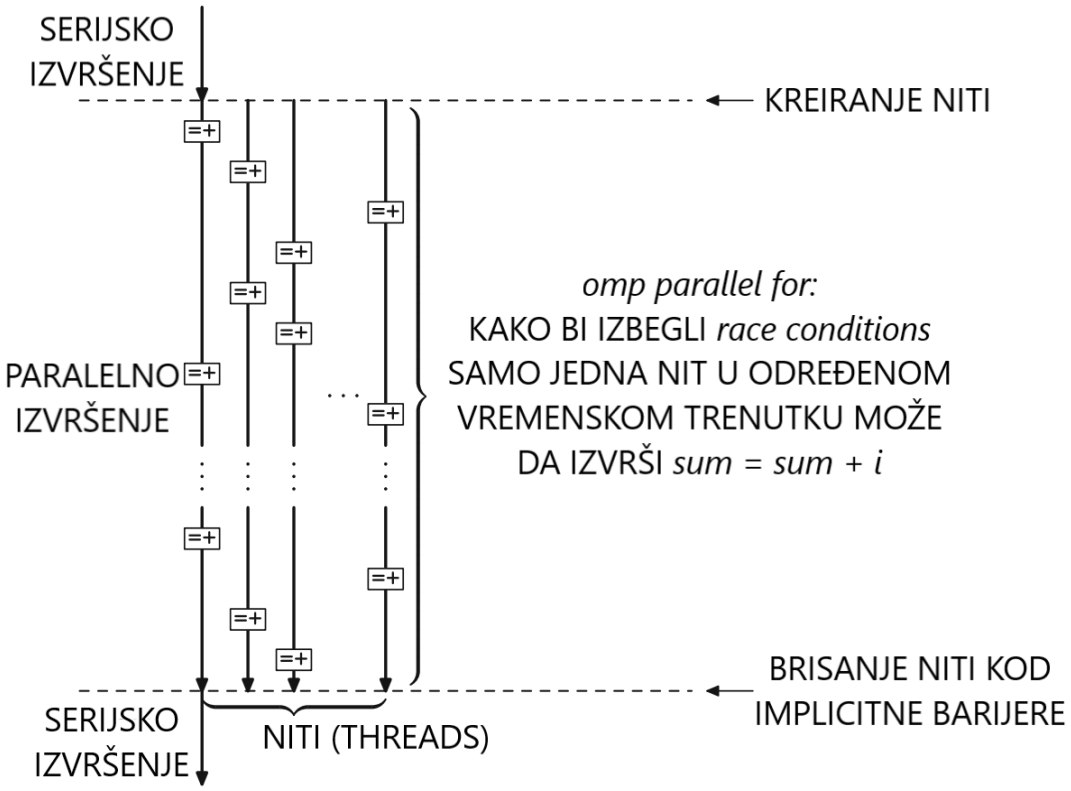
Kako bi se sprečili uslovi trke, naredba *sum = sum + i* može se postaviti unutar kritične sekcije (critical section), koji je deo programa koji se izvršava od strane jedne po jedne niti. Direktiva *omp critical*, koja se primenjuje na naredbu ili blok odmah posle nje, postiže ovo. Listing 3.10 prikazuje program koji koristi kritične sekcije.

```
1  #include <stdio.h>
2
3  int main (int argc , char *argv[]) {
4      int max, i;
5      sscanf_s(argv[1], "%d", &max);
6      int sum = 0;
7      #pragma omp parallel for
8          for (i = 1; i <= max; i++)
9              #pragma omp critical
10                 sum = sum + i;
11     printf ("Suma je: %d\n", sum);
12     return 0;
13
14 }
```

Listing 3.10 Sumiranje celih brojeva iz datog intervala pomoću kritične sekcije — sporo

Kao što je prikazano na Slici 1.4, *omp critical* direktiva eliminiše probleme trke izvođenjem zaključavanja oko dela koda koji sadrži, tj. dela koda koji pristupa

promenljivoj *sum*. Ovo omogućava programu da funkcioniše kako je predviđeno. Ali upotreba kritičnih sekcija usporava program jer, kao što pokazuje Slika 3.8, najviše jedna nit vrši dodavanje i dodeljivanje u bilo kom trenutku, dok druge niti čekaju.



Slika 3.8 Izvršavanje sabiranja celih brojeva kao što je implementirano u Listingu 3.10

Poređeno je vreme rada programa između koda na Listingu 3.9 i 3.10. Da bismo videli razliku na brzom višejezgarnom CPU-u, potreban je veliki broj za *max* koji bi mogao da izazove prelivanje.

Kako bi poredili vreme izvršenja koda sa listinga 3.11 i 3.12 moramo ih modifikovati i dodati funkciju za merenje vremena: *omp_get_wtime().*Modifikovani programi prikazani su na listingu 3.11 i 3.12 respektivno. Oba programa su dobila isti zadatak, da izračunaju sumu brojeva od 1 do 123456789. Broj niti je bio 50 tj. *OMP_NUM_THREADS=50.*

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

int main(int argc, char* argv[]) {
    int max, i;
    sscanf_s(argv[1], "%d", &max);
    int sum = 0;

    double start_time = omp_get_wtime(); // Uzima početno vreme

    #pragma omp parallel for
    for (i = 1; i <= max; i++) {
        sum = sum + i;
    }

    double end_time = omp_get_wtime(); // uzima završno vreme

    printf("Suma brojeva od 1 do %d je: %d\n", max, sum);
    printf("Vreme izvrsavanja: %f sekundi\n", end_time - start_time);
    return 0;
}
```

Listing 3.11 Merenje vremena potrebnog za sumiranje celih brojeva iz datog intervala koristeći jednu zajedničku promenljivu — pogrešno

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

int main(int argc, char* argv[]) {
    int max, i;
    sscanf_s(argv[1], "%d", &max);
    int sum = 0;

    double start_time = omp_get_wtime(); // Uzima početno vreme

    #pragma omp parallel for
    for (i = 1; i <= max; i++) {
        #pragma omp critical
        sum = sum + i;
    }

    double end_time = omp_get_wtime(); // uzima završno vreme

    printf("Suma brojeva od 1 do %d je: %d\n", max, sum);
    printf("Vreme izvrsavanja: %f sekundi\n", end_time - start_time);
    return 0;
}
```

Listing 3.12 Merenje vremena potrebnog za sumiranje celih brojeva iz datog intervala pomoću kritične sekcije — sporo

Posle izračunavanja, program sa Listinga 3.11 je izvrsio računicu za 0.251075 sekundi, dok je suma brojeva bila 1326108201, što takođe možemo videti na Slici 3.9.

```
PS C:\Users\Milos\Desktop\Faks\Diplomski openMP\Sabiranje> ./SabiranjeBr.exe 123456789
Suma brojeva od 1 do 123456789 je: 1326108201
Vreme izvrsavanja: 0.251075 sekundi
```

Slika 3.9 Rezultat programa sa Listinga 3.11

Iako je program izvršio sabiranje jako brzo, rezultat nije tačan. Na Slici 3.10 možemo videti rešenje programa sa Listinga 3.12.

```
PS C:\Users\Milos\Desktop\Faks\Diplomski openMP\Sabiranje> ./SabiranjeBr.exe 123456789
Suma brojeva od 1 do 123456789 je: 1330264167
Vreme izvrsavanja: 7.840118 sekundi
```

Slika 3.10 rezultat programa sa Listinga 3.12

Ovaj program je izvršio sabiranje brojeva za 7.840118 sekundi što je dosta sporije u odnosu na program sa listinga 3.11, ali zato se za rezultat dobija tačno rešenje.

Listing 3.13 ilustruje kako se koristi atomski pristup promenljivama kao dodatni metod za sprečavanje uslova trke.

```
1  #include <stdio.h>
2
3  int main(int argc, char* argv[]) {
4      int max, i;
5      sscanf_s(argv[1], "%d", &max);
6      int sum = 0;
7      #pragma omp parallel for
8          for (i = 1; i <= max; i++)
9          #pragma omp atomic
10         sum = sum + i;
11     printf("Suma je: %d\n", sum);
12     return 0;
13
14 }
```

Listing 3.13 Sumiranje celih brojeva iz datog intervala koristeći pristup atomskoj promenljivoj — brže.

Iako je promenljiva *sum* jedna zajednička varijabla tima niti, program izračunava tačan odgovor jer *omp atomic* direktiva govori kompajleru da proizvede kod koji ažurira *sum = sum + i* kao jednu atomsku operaciju (možda uz pomoć hardverom podržanim uputstvom za čitanje-modifikovanje-upisivanje).

Kritična sekcija (*critical section*) i atomski pristup (*atomic accesses*) promenljivoj su konceptualno veoma slični, ipak, atomski pristup je tipično brži od kritičnog odeljka jer je mnogo jednostavniji i može da sadrži mnogo složenije izračunavanje. Stoga se program Listinga 3.13 u osnovi pokreće kao što prikazuje Slika 3.8.

OpenMP: kritične sekcije

Kritična sekcija se deklariše kao
`#pragma omp critical` *[(name) [hint(hint)]]*
structured-block

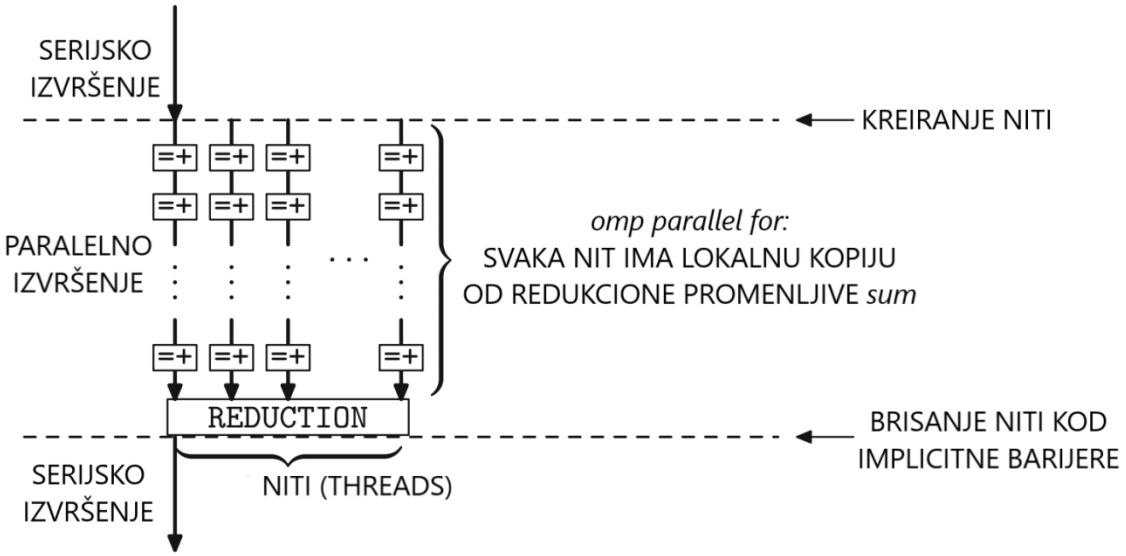
Garantovano je da jedna po jedna nit izvršava strukturirani blok. Tako da nekoliko poslova može nezavisno da implementira istu kritičnu sekciju, kritičnoj sekciji se može dati ime, identifikator i spoljna veza. Da bi se uspostavila detaljna kontrola koja leži u osnovi zaključavanja, nagoveštaj konstantnog celobrojnog izraza može biti dodeljen imenovanoj ključnoj sekciji.

OpenMP nudi jedinstvenu operaciju koja se zove redukcija (*reduction*) kako bi se izbegle situacije trke, zaključavanje ili eksplicitni atomski pristup promenljivim istovremeno. Program Listinga 3.9 je ponovo napisan a pomoću njega nastajes program Listinga 3.14.

```
1  #include <stdio.h>
2
3  int main(int argc, char* argv[]) {
4      int max, i;
5      sscanf_s(argv[1], "%d", &max);
6      int sum = 0;
7      #pragma omp parallel for reduction(+:sum)
8          for (i = 1; i <= max; i++)
9          sum = sum + i;
10     printf("Suma je: %d\n", sum);
11     return 0;
12
13 }
```

Listing 3.14 Sumiranje celih brojeva iz datog intervala korišćenjem redukcije — brzo.

Dodatna klauzula *reduction (+:sum)* navodi da se kreiraju T privatne promenljive *sum*, jedna promenljiva po niti. Privatna promenljiva *sum* se koristi za izračunavanje unutar svake niti. Ove promenljive se dodaju promenljivoj *sum* definisanoj u redu 6 i štampa se u redu 10 tek nakon što se završi paralelna *for* petlja. Kompajler i sistem koji izvršava OpemMP obavljaju konačno sumiranje lokalnih *sum* promenljivih na način pogodan za stvarnu ciljnu arhitekturu.



Slika 3.11 Izvršavanje sabiranja celih brojeva kao što je implementirano u Listingu 3.14

OpenMP: atomski pristup

Atomski pristup promenljivoj unutar *expression-stmt* je deklarisan kao

`#pragma omp atomic` *[seq_cst [,]] atomic-clause [[,] seq_cst]*
expression-stmt
ili
`#pragma omp atomic` *[seq_cst]*
expression-stmt

kada se pretpostavlja *update*. *Omp atomic* direktiva primenjuje ekskluzivni pristup lokaciji za skladištenje među svim nitima u skupu niti za povezivanje bez obzira na timove kojima niti pripadaju.
Tri najvažnije atomske klauzule su:
• *read* izaziva atomsko čitanje x u iskazima oblika *expr = x*;
• *write* izaziva atomsko upisivanje u x u iskazima oblika *x = expr*;
• *update* izaziva atomsko ažuriranje x u izjavama oblika *++x, x++, –x, x–, x=x binop exspr, x = expr binop x, x binop= exspr.*

Ako se koristi *seq_cst*, implicitna operacija *flush* promenljive kojoj se pristupa, atomski se izvodi nakon izraza *statement-expr*.

OpenMP: redukcija

Redukcija je dodatni atribut deljenja podataka koji je tehnički izražen klauzulom

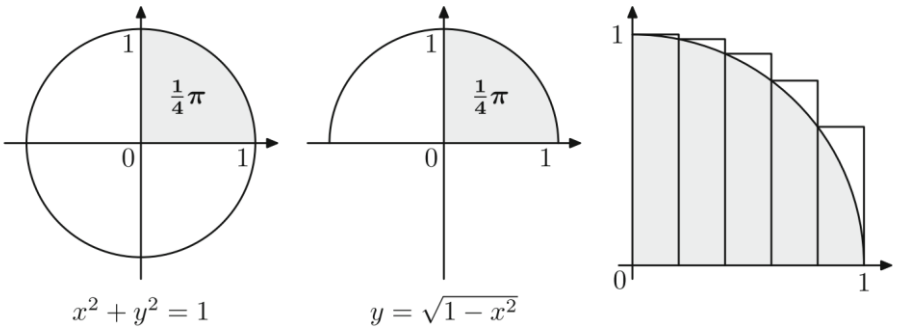
`reduction` *(reduction-identifier:list)*

Za svaku promenljivu na listi, privatna kopija se kreira u svakoj niti paralelnog regiona, i inicijalizovana je na vrednost specificiranu pomoću *reduction-identifier*.
Nakon što se paralelni region završi, operacija specificirana za *reduction-identifier* ažurira originalnu promenljivu vrednostima svih privatnih kopija. Mogu se koristiti +, -, *&*, |, ^, *&&*, ||, *min* i *max* kao *reduction-identifier*. Početna vrednost za * i && je 1, za *min* i *max* su najveća i najmanja vrednost tipa varijable, respektivno, dok je za ostale operacije početna vrednost 0.

Program Listinga 3.14 se pokreće na način prikazan na Slici 3.11. Iako liči na pogrešan program koji radi u Listingu 3.9, red 9 dozvoljava svakoj niti da koristi sopstvenu privatnu promenljivu *sum*, stoga nema trkačkih situacija. Međutim, ove privatne varijable se dodaju ukupnoj globalnoj promenljivoj *sum* na kraju paralelne *for* petlje.

Primer: Korišćenje numeričke integracije za izračunavanje π

Kombinovanje rezultata iteracije petlje je neophodno u mnogim situacijama. Počnimo sa jednodimenzionalnom numeričkom integracijom. Recimo da želimo da pronađemo vrednost π tako što ćemo izračunati površinu jediničnog kruga koja je data formulom $x^2 + y^2 = 1$. Nakon što je pretvorimo u njen eksplicitni oblik, $y = \sqrt{1 - x^2}$ želimo da koristimo jednačinu za izračunavanje vrednosti π, kao što je ilustrovano na Slici 3.12.



Slika 3.12 Integraljenje y = √1 − x2 numerički od 0 do 1

$$\pi = 4 \int_0^1 \sqrt{1 - x^2} dx \tag{3.1}$$

Numerički proračun se koristi za određivanje integrala na desnoj strani prethodne jednačine. Kao rezultat, za dati broj intervala N, interval [0, 1] je podeljen na N intervala

$[\frac{1}{N}i, \frac{1}{N}(i+1)]$ gde je $0 \leq i \leq (N-1)$. Levi Rimanov zbir je izabran da bi se pojednostavio program: površina svakog pravougaonika se izračunava kao širina pravougaonika, ili 1/N, pomnožena sa vrednošću funkcije izračunatom na levoj krajnjoj tački intervala, ili $\sqrt{1-(i/N)^2}$. Stoga,

$$\int_0^1 \sqrt{1-x^2}dx \approx \sum_{i=0}^{N-1}\left(\frac{1}{N}\sqrt{1-\left(\frac{1}{N}i\right)^2}\right) \tag{3.2}$$

za dovoljno veliko N.

Listing 3.15 prikazuje program za određivanje π koristeći zbir na desnoj strani aproksimacije.

```
#include <stdio.h>
#include <math.h>

int main(int argc, char* argv[]) {
    int intervals, i; sscanf_s(argv[1], "%d", &intervals);
    double integral = 0.0;
    double dx = 1.0 / intervals;
    #pragma omp parallel for reduction(+:integral)
        for (i = 0; i < intervals; i++) {
        double x = i * dx;
        double fx = sqrt(1.0 - x * x);
        integral = integral + fx * dx;

    }
    double pi = 4 * integral;
    printf("%20.18lf\n", pi);
    return 0;

}
```

Listing 3.15 Izračunavanje π integraljenjem $\sqrt{1-x^2}$ od 0 do 1

Posle pokretanja programa u našem slučaju, dodeljujemo vrednost 100, 1000 i 10000 za promenljivu, što možemo videti na Slici 3.13

```
PS C:\Users\Milos\Desktop\Faks\Diplomski openMP\Racunanje pi> ./pi.exe 100
3.160417031779046759
PS C:\Users\Milos\Desktop\Faks\Diplomski openMP\Racunanje pi> ./pi.exe 1000
3.143555466911027718
PS C:\Users\Milos\Desktop\Faks\Diplomski openMP\Racunanje pi> ./pi.exe 10000
3.141791477611322936
```

Slika 3.13 Rezultat programa sa Listinga 3.15.

Kao što je prikazano na slici, što je broj intervala veći, preciznije se izračunava vrednost π.

Pošto numerička integracija nije ništa drugo do zbir pravougaonih površina, programi u Listingu 3.15 i Listingu 3.14 su upadljivo identični uprkos komponentama numeričke integracije. Međutim, ne treba zanemariti jednu ključnu tačku: za razliku od *intervals*, *integral* i promenljive *dx*, *x* i *fx*, moraju biti privatne niti.

U ovom trenutku, bilo bi korisno još jednom pokazati da se svaka petlja ne može paralelizirati. Množenje unutar petlje je promenjeno u sabiranje u kodu prikazan u Listingu 3.16.

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main (int argc, char* argv[]){
5      int intervals, i;
6      sscanf_s(argv[1], "%d", &intervals);
7      double integral = 0.0;
8      double x = 0.0;
9      double dx = 1.0 / intervals;
10     #pragma omp parallel for reduction(+:integral)
11         for (i = 0; i < intervals; i++)
12         {
13             double fx = sqrt(1.0 - x * x);
14             integral = integral + fx * dx;
15             x = x + dx;
16
17         }
18
19     double pi = 4 * integral;
20     printf("Vrednost pi je: %20.18lf\n", pi);
21     return 0;
22
23 }
```

Listing 3.16 Računanje π integrisanjem √1 − x2 od 0 do 1 koristeći petlju koja se ne može paralelizovati

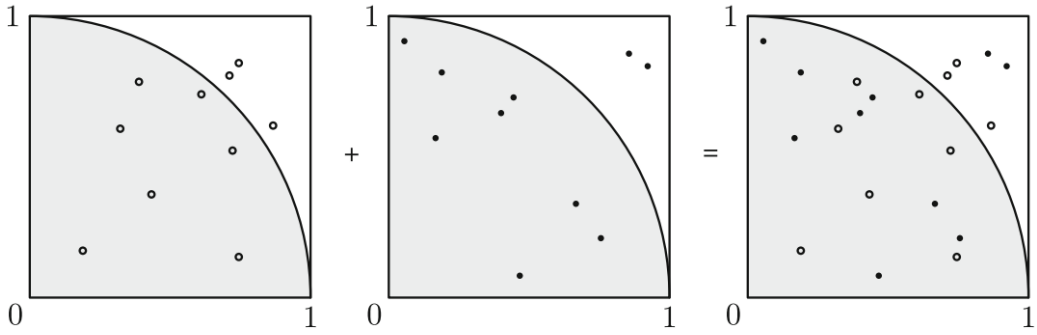
Ovo daje netačan rezultat ako se koristi više niti, ali savršeno funkcioniše ako program pokreće jedna nit (podesite *OMP_NUM_THREADS* na 1). Razlog za to je što iteracije više nisu nezavisne pošto se vrednost x prenosi sa jedne iteracije na drugu, sprečavajući izvršenje sledeće iteracije dok se prethodna ne završi.

Sa druge strane, petlja može i treba da bude paralelna, prema *omp parallel for* direktivi na liniji 10 iz Listinga 3.16.

Primer: Korišćenje slučajnog gađanja za izračunavanje π

Još jedan metod izračunavanja π je nasumično gađanje u kvadrat [0, 1]×[0, 1] i brojanje koliko metaka padne unutar jediničnog kruga, a koliko ne. Površina jediničnog

kruga unutar [0, 1]×[0, 1] je aproksimirana odnosom pogodaka prema svim gađanjima. Pošto je svaki pogodak odvojen od ostalih, pogoci se mogu podeliti na nekoliko niti. Ovaj koncept je prikazan na slici 3.14 ako se koriste dve niti. Listing 3.17 prikazuje implementaciju za bilo koji broj niti.



Slika 3.14 Računanje π slučajnim pogocima: različite niti pucaju nezavisno, ali konačno rezultat je kombinacija svih udaraca

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5  double rnd(unsigned int* seed) {
6      * seed = (1140671485 * (*seed) + 12820163) % (1 << 24);
7      return ((double)(*seed)) / (1 << 24);
8
9  }
10
11 int main(int argc, char* argv[]) {
12     int num_shots, shot;
13     sscanf_s(argv[1], "%d", &num_shots);
14     unsigned int* seeds = (unsigned int*)malloc(omp_get_max_threads() * sizeof(unsigned int));
15     for (int thread = 0; thread < omp_get_max_threads(); thread++)
16         seeds[thread] = thread;
17     int num_hits = 0;
18     #pragma omp parallel for reduction(+:num_hits)
19         for (shot = 0; shot < num_shots; shot++) {
20         int thread = omp_get_thread_num();
21         double x = rnd(&seeds[thread]);
22         double y = rnd(&seeds[thread]);
23         if (x * x + y * y <= 1) num_hits = num_hits + 1;
24
25     }
26     double pi = 4.0 * (double)num_hits / (double)num_shots;
27     printf("%20.18lf\n", pi);
28     free(seeds);
29     return 0;
30
31 }
```

Listing 3.17 Izračunavanje π slučajnim gađanjem pomoću paralelne petlje

Program u Listingu 3.17 je u suštini jednostavan kada se posmatra iz perspektive paralelnog programiranja: broj pogodaka se akumulira u promenljivoj *num_hits*, a *num_shots* su gađanja unutar paralelne *for* petlje u redovima 19–26. Štaviše, sličan je programu u Listingu 3.14 po tome što je krajnji rezultat (Slika 3.15) posledica kombinovanja rezultata nezavisnih iteracija. Proces pravljenja nasumičnih gađanja je najkompleksniji aspekt programa. Pošto koriste jedno skriveno stanje koje se modifikuje pri svakom pozivu, bez obzira na nit u kojoj je poziv napravljen, standardni generatori slučajnih vrednost, kao što su *rand* ili *random*, nemaju mogućnost ponovnog ulaza (nisu

reentrant) niti su bezbedne za niti (nisu *thread-safe*). Kao takvi, ne bi trebalo da se pozivaju u više niti. Funkcija *rnd* je kreirana da zaobiđe ovaj problem; prihvata *seed* (u daljem tekstu seme) menja ga i vraća nasumičnu vrednost iz opsega [0,1). Da bi se utvrdilo koliko će budućih niti biti korišćeno u paralelnoj for petlji kasnije, OpenMP funkcija *omp_get_max_threads* se koristi u redovima 14–16 da se uspostavi jedinstveno seme za svaku nit. Program ima jedan jedinstveni generator slučajnih vrednosti za svaku nit, koji se koristi sa ovim semenima.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5  double rnd(unsigned int* seed) {
6      * seed = (1140671485 * (*seed) + 12820163) % (1 << 24);
7      return ((double)(*seed)) / (1 << 24);
8
9  }
10
11 int main(int argc, char* argv[]) {
12     int num_shots, shot; sscanf_s(argv[1], "%d", &num_shots);
13     int num_hits = 0;
14     #pragma omp parallel
15         {
16         unsigned int seed = omp_get_thread_num();
17         #pragma omp for reduction(+:num_hits)
18             for (shot = 0; shot < num_shots; shot++) {
19             double x = rnd(&seed);
20             double y = rnd(&seed);
21             if (x * x + y * y <= 1) num_hits = num_hits + 1;
22
23         }
24         }
25     double pi = 4.0 * (double)num_hits / (double)num_shots;
26     printf("%20.18lf\n", pi);
27     return 0;
28
29 }
```

Listing 3.18 Izračunavanje π slučajnim pogocima pomoću paralelne petlje

Kada se numerička integracija zameni slučajnim pogocima, stopa konvergencije prema π je znatno niža. Ali ako se generator slučajnih vrednosti koristi na odgovarajući način, ovaj primer pokazuje koliko je lako primeniti širok spektar Monte Karlo metoda: sve što treba da se uradi jeste pokrenuti sve pojedinačne eksperimente na osnovu slučajnosti, kao što su gađanja u [0, 1]×[0, 1] u redovima 21–23 i objediniti rezultate, kao što je brojanje broja pogodaka unutar jediničnog kruga.

Metoda predstavljena u Listingu 3.17 je dovoljna, pod uslovom da je broj pojedinačnih eksperimenata poznat unapred i da je kompleksnost svakog eksperimenta približno ista. Ako ne, potrebna je naprednija strategija.

Možemo da pojednostavimo program u Listingu 3.17 pre nego što nastavimo dalje. Listing 3.18 ilustruje kako možemo da definišemo seme lokalno u niti (*thread-local seed*) unutar paralelnog regiona tako što ćemo podeliti *omp parallel* i *omp for*.

Iako se može drugačije kodirati kao što je ilustrovano u Listingu 3.19, izračunavanje π brojanjem hitaca unutar jediničnog kruga i nasumično pucanjem u [0,1]×[0,1] ostaje u osnovi isto.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5  double rnd(unsigned int* seed) {
6      * seed = (1140671485 * (*seed) + 12820163) % (1 << 24);
7      return ((double)(*seed)) / (1 << 24);
8
9  }
10
11 int main(int argc, char* argv[]) {
12     int num_shots; sscanf_s(argv[1], "%d", &num_shots);
13     int num_hits = 0;
14     #pragma omp parallel reduction(+:num_hits)
15         {
16         unsigned int seed = omp_get_thread_num();
17         int loc_shots = (num_shots / omp_get_num_threads()) +
18             ((num_shots % omp_get_num_threads() > omp_get_thread_num())? 1 : 0);
19         while (loc_shots-- > 0) {
20             double x = rnd(&seed);
21             double y = rnd(&seed);
22             if (x * x + y * y <= 1) num_hits = num_hits + 1;
23
24         }
25         }
26     double pi = 4.0 * (double)num_hits / (double)num_shots;
27     printf("Vrednost pi je: %lf\n", pi);
28     return 0;
29
30 }
```

Listing 3.19 Izračunavanje π slučajnim pogocima korišćenjem paralelnih sekcija

Konkretno, paralelna *for* petlja je zamenjena paralelnim regionima – po jedan za svaku dostupnu nit – specificiranom *omp parallel* direktivom u redu 14. (videti i Listing 2.1 i 2.2). Linija 16 generiše seme generatora slučajnih vrednosti lokalno u niti unutar svake paralelne zone. Redovi 18–19 zatim izračunavaju koliko hitaca nit treba da izvede, a na kraju se svi hici izvode u sekvencijalnoj *while* petlji lokalno u niti u linijama 20-24. Iteracije *while* petlje ne uključuju poziv metode *omp_get_thread_num*, za razliku od iteracija paralelne *par* petlje u Listingu 3.18. S druge strane, redukcija se koristi na isti način kao u Listingu 3.18 za sakupljanje, tj, agregaciju broja pogodaka – rezultata dobijenih po paralelnim oblastima.

3.3 Raspodela iteracija među nitima

Načinu na koji su iteracije paralelne petlje, ili mnogih sažetih paralelnih petlji, raspodeljene između različitih niti unutar jednog tima niti, do sada nije posvećeno mnogo pažnje. Bez obzira na to, programer može definisati više različitih algoritama za planiranje iteracija koristeći OpenMP.

Korišćenjem softvera prikazanog u Listingu 3.14 još jednom razmatramo izračunavanje zbira brojeva iz određenog intervala. Ali ovaj put, program će biti izmenjen

na način kako to ilustruje Listing 3.20. Prvo, *omp for* direktiva u redu 9 se ažurira klauzulom *schedule* (*runtime*). Ovo omogućava da se shell promenljiva OMP_SCHEDULE koristi za definisanje pristupa rasporedu iteracija kada se program pokrene. Drugo, red 11 ispisuje broj niti koje se koriste za izvršavanje svake iteracije. Treće, kao što je naznačeno argumentom funkcije *sleep* u redu 12, različite iteracije se izvršavaju u različito vreme. To jest, dok je za iteracije 1, 2 i 3 potrebno 2, 3 i 4 sekunde da se završe, preostale iteracije traju samo jednu sekundu.

```
1  #include <stdio.h>
2  #include <omp.h>
3  #include <windows.h>
4
5  int main(int argc, char* argv[]) {
6      int max, i;
7      sscanf_s(argv[1], "%d", &max);
8      long int sum = 0;
9  #pragma omp parallel for reduction(+:sum) schedule(runtime)
10     for (i = 1; i <= max; i++) {
11         printf("%2d @ %d\n", i, omp_get_thread_num());
12         Sleep(i < 4 ? (i + 1) * 1000 : 1000); // Sleep for seconds
13         sum = sum + i;
14     }
15     printf("%ld\n", sum);
16     return 0;
17 }
```

Listing 3.20 Sumiranje celih brojeva iz datog intervala gde je strategija planiranja iteracije određena u vremenu izvođenja.

Pretpostavimo da se koriste 4 niti i *max* = 14:

- U slučaju kada je OMP_SCHEDULE = *static*, iteracije se dele na delove od kojih svaki sadrži približno isti broj iteracija i svakoj niti je dat najviše jedan deo. Jedna potencijalna statička raspodela od 14 iteracija između četiri niti je:

$$T_0: \underbrace{\{1, 2, 3, 4\}}_{10\ \text{secs}} \quad T_1: \underbrace{\{5, 6, 7, 8\}}_{4\ \text{secs}} \quad T_2: \underbrace{\{9, 10, 11\}}_{3\ \text{secs}} \quad T_3: \underbrace{\{12, 13, 14\}}_{3\ \text{secs}}$$

Najduže iteracije su dodeljene niti T0: iako iteracije sa i = 4 traju 1 sekundu, ali iteracije sa i od 1, 2 ili 3 traju 2 s, 3 s, odnosno 4 s. Svim ostalim nitima je potrebna samo jedna sekunda za svaku iteraciju. Zbog toga se nit T0 završava mnogo kasnije od bilo koje druge niti, kao što se vidi na Slici 3.16.

OpenMP: planiranje iteracija paralelne petlje

Raspodelu iteracija paralelnih petlji među timskim nitima kontroliše klauzula *schedule*. Najvažnije opcije su sledeće:

• *schedule (static)*: Iteracije su podeljene na delove od kojih svaki sadrži približno isti broj iteracija i svakoj niti je dat najviše jedan deo.

• *schedule (static,chunk_size)*: Iteracije su podeljene u delove gde svaki deo sadrži chunk_size iteracija. Zatim se delovi dodeljuju nitima kružnom metodom (jedan za drugim).

• *schedule (dynamic,chunk_size)*: Iteracije su podeljene u delove gde svaki deo sadrži chunk_size iteracija. Delovi su dodeljeni nitima dinamički: svaka nit uzima jedan po jedan deo iz zajedničkog skupa delova, izvršava ga i zahteva novi deo dok se skup ne isprazni.

• *schedule (auto)*: Izbor strategije planiranja je prepušten kompajleru i sistemu u vreme izvršenja.

• *schedule (runtime)*: Strategija planiranja je specificirana u vreme izvršenja koristeći shell promenljivu OMP_SCHEDULE.

Ako klauzula schedule nije prisutna u direktivi *omp for*, kompajleru i sistemu u vreme izvršenja je dozvoljeno da sami izaberu strategiju planiranja.

- Iteracije se dele na delove koji imaju 1 ili 2 iteracije, respektivno, ako je *OMP_SCHEDULE=static, 1* ili *OMP_SCHEDULE=static, 2*. Nakon toga, delovi se raspoređuju između niti kružnom metodom, sa tačkom i zarezom između delova koji pripadaju istoj niti.

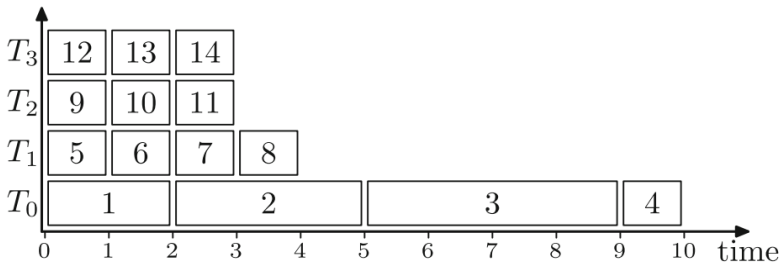
$$T_0: \underbrace{\{1; 5; 9; 13\}}_{5\text{ secs}} \quad T_1: \underbrace{\{2; 6; 10; 14\}}_{6\text{ secs}} \quad T_2: \underbrace{\{3; 7; 11\}}_{6\text{ secs}} \quad T_3: \underbrace{\{4; 8; 12\}}_{3\text{ secs}}$$

ili

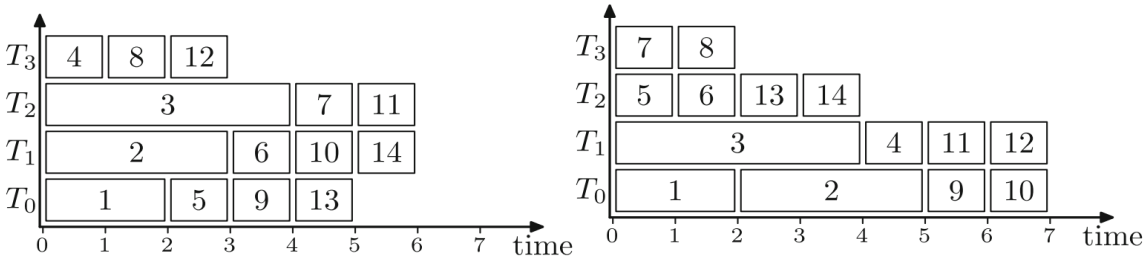
$$T_0: \underbrace{\{1, 2; 9, 10\}}_{7\text{ secs}} \quad T_1: \underbrace{\{3, 4; 11, 12\}}_{7\text{ secs}} \quad T_2: \underbrace{\{5, 6; 13, 14\}}_{4\text{ secs}} \quad T_3: \underbrace{\{7, 8\}}_{2\text{ secs}}$$

Vreme izvršenja različitih niti se razlikuje manje nego ako se koristi jednostavno statičko raspoređivanje, kao što je ilustrovano na Slici 3.15. Pored toga, u zavisnosti od veličine delova, ukupno vreme izvršenja se skraćuje sa 10 na 6 ili 7 sekundi.

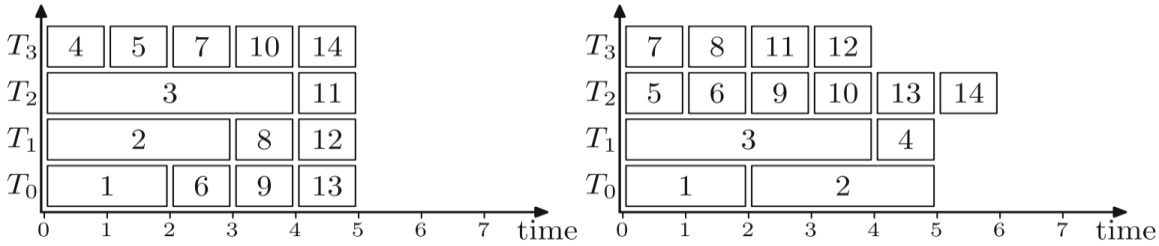
- U slučajevima kada je *OMP_SCHEDULE = dynamic, 1* ili *OMP_SCHEDULE = dynamic, 2*, iteracije se razdvajaju u delove koji sadrže jednu ili dve iteracije, respektivno. Niti se dinamički dodeljuju delovima: svaka nit preuzima deo iz zajedničkog skupa delova, izvršava ga, a zatim zahteva drugi deo dok se skup ne iscrpi.



Slika 3.15 Raspodela od 14 iteracija između 4 niti gde iteracije 1, 2 i 3 zahtevaju više vremena u odnosu na druge iteracije, koristeći statičku strategiju planiranja iteracija



Slika 3.16 Raspodela od 14 iteracija između 4 niti gde iteracije 1, 2 i 3 zahtevaju više vremena od drugih iteracija, koristeći *static*,1 (levo) i *static*,2 (desno) strategije planiranja iteracija



Slika 3.17 Raspodela od 14 iteracija između 4 niti gde iteracije 1, 2 i 3 zahtevaju više vreme od drugih iteracija, koristeći *dynamic*,1 (levo) i *dynamic*,2 (desno) strategije planiranja iteracija

Stoga, za delove koji se sastoje od 1 i 2 iteracije, respektivno, dva moguća dinamička dodeljivanja su

$$T_0: \underbrace{\{1; 6; 9; 13\}}_{5 \text{ secs}} \quad T_1: \underbrace{\{2; 8; 12\}}_{5 \text{ secs}} \quad T_2: \underbrace{\{3; 11\}}_{5 \text{ secs}} \quad T_3: \underbrace{\{4; 5; 7; 10; 14\}}_{5 \text{ secs}}$$

ili

$$T_0: \underbrace{\{1, 2\}}_{5 \text{ secs}} \quad T_1: \underbrace{\{3, 4\}}_{5 \text{ secs}} \quad T_2: \underbrace{\{5, 6; 9, 10; 13, 14\}}_{6 \text{ secs}} \quad T_3: \underbrace{\{7, 8; 11, 12\}}_{4 \text{ secs}}$$

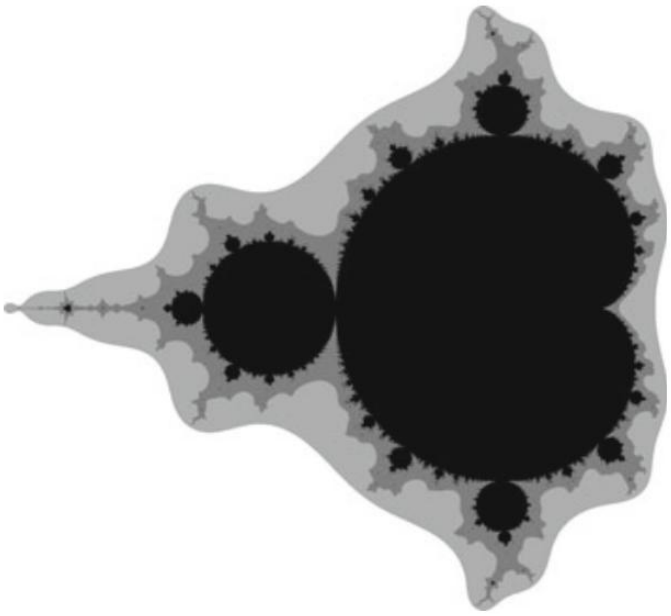
Slika 3.16 pokazuje kako su iteracije raspoređene: na osnovu veličine komada ukupno vreme rada se dalje smanjuje na 5 ili 6 s. Pošto svaka nit obavlja istu količinu posla, ukupno vreme rada od 5 s je najmanje koje se može postići.

Primer: Mandelbrotov skup

Problem koji se rešava je odlučujući faktor u izboru najboljeg metoda raspoređivanja iteracija. Razumevanje izračunavanja Mandelbrotovog skupa je važan metod planiranja iteracije. Definisan je kao

$$M = \{c;\ \lim_{n\to\infty}|z_n| \leq 2\ where\ z_0 = 0\ \wedge\ z_{n+1} = z_n^2 + c\} \qquad (3.2)$$

kao što je i prikazano na Slici 3.18.



Slika 3.18 Mandelbrotov skup u kompleksnoj ravni [−2,6, +1,0] × [−1.2i, +1.2i] (crna) i njegov komplement (bela, svetlo i tamno siva). Zatamnjenost svake tačke označava vreme potrebno za utvrđivanje da li tačka pripada Mandelbrotovom skupu ili ne

Koristeći program prikazan u Listingu 3.21, koji kreira sliku sastavljenu od *i_size* x *j_size* piksela, Mandelbrotov skup se može izračunati. Niz $z_{n+1} = z_n^2 + c$ gde *c* označava koordinate piksela u kompleksnoj ravni, ponavlja se za svaki piksel sve dok počne da divergira ($|z_{n+1}| > 2$) ili dostigne unapred određen maksimalni broj iteracija (*max_iters*) koji je unapred definisan.

```
1  #pragma omp parallel for collapse(2) schedule(runtime)
2  for (int i = 0; i < i_size; i++) {
3      for (int j = 0; j < j_size; j++) {
4          // printf ("# (%d,%d) t=%d\n", i, j, omp_get_thread_num());
5              double c_re = min_re + i * d_re;
6          double c_im = min_im + j * d_im;
7
8              double z_re = 0.0;
9          double z_im = 0.0;
10         int iters = 0;
11         while ((z_re * z_re + z_im * z_im <= 4.0) &&
12             (iters < max_iters)) {
13             double new_z_re = z_re * z_re - z_im * z_im + c_re;
14             double new_z_im = 2 * z_re * z_im + c_im;
15             z_re = new_z_re; z_im = new_z_im;
16             iters = iters + 1;
17
18         }
19         picture[i][j] = iters;
20
21     }
22
23 }
```

Listing 3.21 Računanje Mandelbrotovog skupa.

Za generisanje Slike 3.18 *max_iters* je podešeno na 100. Potrebno je 100 iteracija da se izračuna svaka tačka unutar Mandelbrotovog skupa, ili crnog regiona. Ali za svaku tačku u tamno sivoj zoni potrebno je više od deset, ali ne više od stotinu ponavljanja. Slično tome, svakoj tački u svetlosivoj zoni potrebno je između pet i deset iteracija, ali svakoj drugoj tački – to jest, belim tačkama – nije potrebno više od pet iteracija. Radi se o tome da li se koristi pristup planiranju iteracija jer različite lokacije, a samim tim i različite iteracije kolapsiranih *for* petlji u redovima 2 i 3, zahtevaju dramatično različite količine izračunavanja. Konkretno, korišćenje *static, 100* za razliku od samo *static* smanjuje vreme rada za oko trideset procenata; koristeći *dinamic, 100* dodatno skraćuje vreme rada.

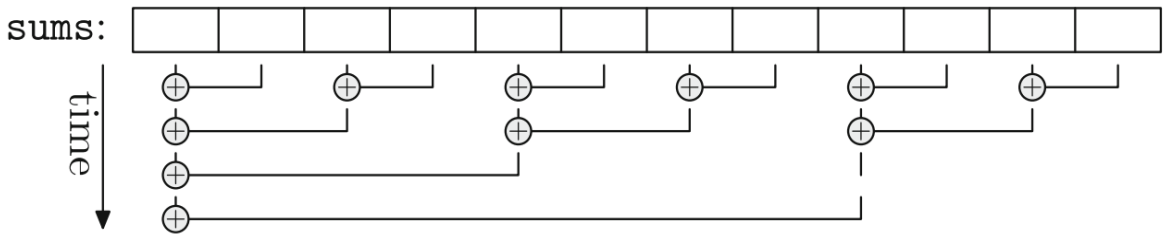
3.4 Detalji paralelnih petlji i redukcija

U OpenMP programiranju, paralelna *for* petlja i operacija redukcije su veoma važne da zahtevaju pažljivo proučavanje i razumevanje. Vratimo se na Listing 3.14, gde je prikazan program za sabiranje svih celih brojeva od 1 do *max*. Program se može prepisati kao Listing 3.22 ako se pretpostavi da T, broj niti, deli *max* i da se koristi *static* iterativna strategija raspoređivanja.

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main(int argc, char* argv[]) {
5      int max; sscanf_s(argv[1], "%d", &max);
6      int ts = 10;
7      if (max % ts != 0) return 1;
8      int sums[10];
9  #pragma omp parallel
10     {
11         int t = omp_get_thread_num();
12         int lo = (max / ts) * (t + 0) + 1;
13         int hi = (max / ts) * (t + 1) + 0;
14         sums[t] = 0;
15         for (int i = lo; i <= hi; i++)
16             sums[t] = sums[t] + i;
17     }
18     int sum = 0;
19     for (int t = 0; t < ts; t++) sum = sum + sums[t];
20     printf("Suma je: %d\n", sum);
21     return 0;
22
23 }
```

Listing 3.22 Implementacija efikasnog sabiranja celih brojeva ručno korišćenjem jednostavne redukcije

Koristeći OpenMP funkciju *omp_get_max_threads*, prva nit prvo određuje T, ukupan broj dostupnih niti, a zatim gradi niz koji sadrži *sums* promenljivih koje se koriste za sumiranje unutar svake niti. Svaka nit će pristupiti samo jednom od T elemenata niza *sums*, iako će ovaj niz deliti sve niti.



Slika 3 .19 Izračunavanje smanjenja vremena $O(\log_2 T)$ koristeći $T/2$ niti kada je $T = 12$

Kada glavna nit dostigne *omp prallel* region, generiše (T − 1) podređenih niti koje rade paralelno sa njom. Pre nego što izvrši svoju *thread-local* sekvencijalnu *for* petlju (redovi 14–15), svaka nit—master ili slejv—izračunava svoj podinterval (redovi 11–12), inicijalizuje svoju lokalnu promenljivu sumiranja na nulu (red 13). Glavna nit je jedina koja ostaje aktivna nakon što sve niti završe izračunavanje lokalnih *sums*. Lokalne promenljive sumiranja se dodaju, a rezultat se štampa. Generalno izvođenje se izvodi kao

što ilustruje slika 3.6. T-ta nit koristi t-ti element *sums[t]* sums niza, tako da nema problema sa trkom jer svaka nit koristi sopstvenu promenljivu sumiranja.

Iz perspektive implementacije, program Listinga 3.22 sprovodi redukciju samo pomoću glavne niti, i umesto da koristi promenljive za sumiranje lokalnih niti, koristi *sums* niza. Eksplicitno smanjenje se vrši u redu 19 nakon što su slejv niti prekinute i njihove lokalne varijable (*t, lo, hi* i *n*) su izgubljene. Ovo je omogućeno tako što glavna nit kreira sume niza pre kreiranja podređenih niti.

Štaviše, smanjenje se postiže sukcesivnim dodavanjem lokalnih promenljivih sumiranja – to jest, elemenata *sums* – ukupnoj promenljivoj *sum*. Ovo dobro funkcioniše sa manjim brojem niti, kao što je T = 4 ili T = 8, i zahteva O(T) vremena. Rešenje iz Listinga 3.23, koje se izvršava u vremenu $O(\log_2 T)$ i vraća rezultat u *sums[0]*, obično se bira kada se radi sa nekoliko stotina niti, osim ako dizajn ciljnog sistema zahteva složeniji pristup.

```
1  for (int d = 1; d < ts; d = d * 2)
2      #pragma omp parallel for
3          for (int t = 0; t < ts; t = t + 2 * d)
4              if (t + d < ts) sums[t] = sums[t] + sums[t + d];
```

Listing 3.23 Implementacija efikasnog sabiranja celih brojeva ručno korišćenjem jednostavne redukcije.

Slika 3.19 ilustruje koncept koji leži u osnovi koda predstavljenog u Listingu 3.23.U Listingu 3.23 promenljiva *d* sadrži rastojanje između elemenata niza *sums*, i kako se udvostručuje u svakoj iteraciji, postoji $\lceil \log_2 T \rceil$ iteracije spoljašnje petlje.

Levi element svakog para koji se dodaje u unutrašnju petlju označen je promenljivom *t*. Međutim, pošto unutrašnja petlja radi na različitim članovima zbira niza paralelno koristeći najmanje T/2 niti, svi dodaci unutar unutrašnje petlje se završavaju istovremeno, ili u vremenu O(1).

U oba slučaja za izračunavanje smanjenja su potrebni dodaci (T − 1). Koristeći prvi metod, dodaci se izvršavaju jedan za drugim dok u drugom metodu neki dodaci mogu biti izvršeni simultano.

4. Paralelni zadaci

Nije uvek slučaj da paralelni programi troše najveći deo svog vremena na izvršenje paralelnih petlji. Stoga je vredno istražiti mogućnost paralelizacije programa koji se sastoji od različitih zadataka.

Počinjemo sa opisom zadataka gde saradnja nije potrebna, kao i ranije, gde je prvo objašnjena paralelizacija petlji koje ne moraju da kombinuju rezultate svojih iteracija. Razmotrićemo još jednom izračunavanje zbira celih brojeva od 1 do *max*. Raspodela iteracija jedne paralelne *for* petlje između niti je demonstrirana na kraju prethodnog odeljka. Ovaj put, međutim, interval od 1 do *max* je podeljen na nekoliko podintervala koji se međusobno isključuju. Svaki podinterval ima zadatak koji dodaje zbir podintervala globalnoj sumi nakon što izračuna zbir svih celih brojeva za taj podinterval.

Program u Listingu 4.1 predstavlja implementaciju ideje. Pretpostavlja se da T, koji predstavlja broj zadataka i koji se nalazi u promenljivoj *tasks*, deli *max* radi jednostavnosti.

Izračunavanje zbira se izvršava u bloku *parallel* na linijama 9-25. Kod u redovima 13–23 definiše svaki zadatak, a *for* petlja u redu 12 generiše svih T zadataka. Planiranje i izvršavanje zadataka je uglavnom prepušteno OpenMP sistemu u vremenu izvršenja nakon što se oni kreiraju.

Međutim, *for* petlja u redu 12 mora biti izvršena od strane jedne niti, inače bi svaka nit generisala poseban skup od T zadataka. Da bi se ovo postiglo petlja *for* je pozicionirana u liniji 11 pod OpenMP direktivom *single*.

Kod u redovima 14–23 treba da se izvede kao jedan zadatak, prema OpenMP direktivi task u redu 13. Lokalni zbir je inicijalizovan na nulu, a granice podintervala se izračunavaju korišćenjem broja zadatka, ili t. Da bi se izbegao uslov trke između dva različita zadatka, celi brojevi podintervala se sabiraju, a lokalni zbir se dodaje globalnom zbiru korišćenjem atomske sekcije.

Treba napomenuti da kada se kreira novi zadatak, onaj zadatak koji ga je generisao nastavlja da se izvršava bez kašnjenja; novi kreirani zadatak tada ima samostalnu egzistenciju. Preciznije, kada glavna nit Listinga 4.1 izvršava *for* petlju, ona kreira novi zadatak za svaku iteraciju. Tako, iteracije, i shodno tome, kreiranje novih zadataka, izvode se uzastopno bez čekanja da se novokreirani zadaci završe; u stvari, bilo bi nelogično to učiniti. Međutim, *parallel* region se ne može završiti dok se ne završe svi zadaci. Dakle, svi zadaci su već završeni kada red 27 Listinga 4.1 ispisuje globalni zbir.

Drugi način da se objasne razlike između metoda korišćenih u prethodnom i ovom odeljku je sledeći. Naime, zadaci se implicitno kreiraju, po jedan za svaku nit, kada se iteracije jedne paralelne *for* petlje raspodeljuju među nitima. Međutim, kada se koristi veliki broj eksplicitnih zadataka, sama petlja se deli na više zadataka, koji se zatim raspodeljuju među nitima.

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

int main(int argc, char* argv[]) {
    int max; sscanf_s(argv[1], "%d", &max);
    int tasks; sscanf_s(argv[2], "%d", &tasks);
    if (max % tasks != 0) return 1;
    int sum = 0;
    #pragma omp parallel
        {
        #pragma omp single
            for (int t = 0; t < tasks; t++) {
            #pragma omp task
                {
                int local_sum = 0;
                int lo = (max / tasks) * (t + 0) + 1;
                int hi = (max / tasks) * (t + 1) + 0;
                // printf ("%d: %d..%d\n", omp_get_thread_num(), lo, hi);
                    for (int i = lo; i <= hi; i++)
                    local_sum = local_sum + i;
                #pragma omp atomic
                    sum = sum + local_sum;
                }

        }
        }
    printf("%d\n", sum);
    return 0;

}
```

Listing 4.1 Implementacija sabiranja celih brojeva korišćenjem fiksnog broja zadataka.

Primer: Fibonačijevi brojevi

Može biti prilično naivno izračunati prvih *max* Fibonačijevih brojeva koristeći vremenski zahtevnu rekurzivnu proceduru, posebno ako se rekurzivna funkcija poziva nezavisno za svaki od njih. Međutim, to pokazuje kako se koriste prednosti zadataka. Program Listinga 4.2 ilustruje kako se to može postići. Sve zadatke pokreće jedna nit unutar paralelnog regiona, po jedan za svaki broj. Kada se program napiše, najmanji Fibonačijevi brojevi se prvo izračunavaju npr. za n = 1, 2, ..., a najveći se ostavljaju da se kasnije izračunaju.

Koristeći funkciju *fib* u liniji 4 Listinga 4.2, *n*- Fibonačijev broj se može izračunati za vreme koje je reda $O(1.6^n)$ vremena. Zbog toga, vremenska složenost pojedinačnih zadataka raste eksponencijalno sa *n*. Prema tome, možda bi bilo bolje kreirati (i tako izvršiti) zadatke obrnutim redosledom, pri čemu najteži zadaci dolaze prvi, a najlakši poslednji, kao što je prikazano u Listingu 4.3.

OpenMP: zadaci

Zadatak je deklarisan korišćenjem direktive

#pragma omp task [clause [[,] clause] ...]
structured-block

Direktiva *task* kreira novi zadatak koji izvršava strukturirani blok. Novi zadatak se može izvršiti odmah ili se može odložiti. Odloženi zadatak može kasnije da izvrši bilo koja nit u timu.
Direktiva *task* može se dalje precizirati brojnim klauzulama, od kojih su najvažnije sledeće:
• *final* (*scalar-logical-expression*) uzrokuje, ako se skalarno-logički-izraz proceni na istinito (true), da kreirani zadatak više ne generiše nove zadatke, tj. uključen je kod potencijalnih novih podzadataka i na taj način izvršeni u okviru ovog zadatka;
• *if ([task:]scalar-logical-expression*) uzrokuje, ako skalarno-logički-izraz procenjuje na netačno (*false*), da je kreiran neodloženi zadatak, tj. kreirani zadatak suspenduje zadatak kreiranja dok se kreirani zadatak ne završi.

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  long fib(int n) { return (n < 2 ? 1 : fib(n - 1) + fib(n - 2)); }
5
6  int main(int argc, char* argv[]) {
7      int max; sscanf_s(argv[1], "%d", &max);
8      #pragma omp parallel
9          #pragma omp single
10         for (int n = 1; n <= max; n++)
11         #pragma omp task
12         printf("Thread %d: %d  %ld\n", omp_get_thread_num(), n, fib(n));
13     return 0;
14
15 }
```

Listing 4.2 Računanje Fibonačijevih brojeva korišćenjem zadataka (*tasks*) OpenMP-a: prvo se izračunavaju manji Fibonačijevi brojevi.

Retko je korisno i nije posebno fascinantno pretvoriti paralelnu *for* petlju u skup zadataka. Međutim, stvarna moć zadataka se može ceniti kada se broj i veličina pojedinačnih zadataka ne mogu unapred znati. Drugačije rečeno, kada problem ili algoritam tako zahtevaju, zadaci se kreiraju dinamično.

OpenMP: ograničavanje izvršenja na jednu nit

Unutar odeljka *parallel*, direktiva

#pragma omp single [clause [[,] clause] ...]

structured-block

uzrokuje da strukturirani blok izvršava tačno jedna nit u timu (ne nužno glavna nit). Ako nije drugačije navedeno, sve ostale niti čekaju neaktivne na implicitnoj barijeri na kraju direktive *single*.

Najvažnije klauzule su sledeće:

• *private* (*list*) specifikuje da je svaka promenljiva u listi privatna za kod koji se izvršava unutar direktive *single*.;

• *nowait* uklanja implicitnu barijeru na kraju pojedinačne direktive i na taj način omogućava drugim nitima u timu da nastave bez čekanja da se završi kod pod *single* direktivom.

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  long fib(int n) { return (n < 2 ? 1 : fib(n - 1) + fib(n - 2)); }
5
6  int main(int argc, char* argv[]) {
7      int max; sscanf_s(argv[1], "%d", &max);
8      #pragma omp parallel
9          #pragma omp single
10         for (int n = max; n >= 1; n--)
11             #pragma omp task
12             printf("%d: %d %ld\n", omp_get_thread_num(), n, fib(n));
13     return 0;
14
15 }
```

Listing 4.3 Računanje Fibonačijevih brojeva korišćenjem zadataka (*tasks*) OpenMP-a: prvo se izračunavaju veći Fibonačijevi brojevi

Primer: Brzo sortiranje

Jednostavan, ali dobro poznat primer sortiranja je korišćenje algoritma brzog sortiranja (*Quicksort*). Paralelna verzija koja koristi zadatke je prikazana u Listingu 4.4.

Particioni deo algoritma, implementiran u linijama 4–14 Listinga 4.4, je isti kao u sekvencijalnoj verziji. Rekurzivni pozivi su, međutim, modifikovani jer se mogu izvoditi nezavisno, odnosno istovremeno. Svaki od dva rekurzivna poziva se stoga izvršava kao sopstveni zadatak.

Međutim, koliko god efikasno bilo kreiranje novih zadataka, potrebno je vreme. Kreiranje novog zadatka ima smisla samo ako je deo tabele koji se mora sortirati pomoću rekurzivnog poziva dovoljno velik. U Listingu 4.4, klauzula *final* u redovima 15 i 17 se koristi da spreči kreiranje novih zadataka za delove tabele koji sadrže manje od 1000 elemenata. Prag 1000 je izabran iskustvom; izbor najboljeg praga zavisi od mnogih

faktora (broj elemenata, vreme potrebno za poređenje dva elementa, implementacija OpenMP zadataka, …).

```
1  void par_qsort(char** data, int lo, int hi,
2      int (*compare)(const char*, const char*)) {
3      if (lo > hi) return;
4      int l = lo;
5      int h = hi;
6      char* p = data[(hi + lo) / 2];
7      while (l <= h) {
8          while (compare(data[l], p) < 0) l++;
9          while (compare(data[h], p) > 0) h--;
10         if (l <= h) {
11             char* tmp = data[l]; data[l] = data[h]; data[h] = tmp;
12             l++; h--;
13
14         }
15
16     }
17     #pragma omp task final(h - lo < 1000)
18         par_qsort(data, lo, h, compare);
19     #pragma omp task final(hi - l < 1000)
20         par_qsort(data, l, hi, compare);
21
22 }
```

Listing 4.4 Paralelna implementacija algoritma brzog sortiranja gde se svaki rekurzivni poziv izvodi kao novi zadatak

Postoji analogija sa sekvencijalnim algoritmom: rekurzija takođe zahteva vreme i da bi se ubrzao sekvencijalni algoritam brzog sortiranja, sortiranje umetanjem se koristi kada broj elemenata padne ispod određenog praga.

Ne bi trebalo da bude zabune oko argumenata za funkciju *par_qsort*. Međutim, funkcija *par_qsort* mora biti pozvana unutar paralelnog (*parallel*) regiona od strane tačno jedne niti kao što je prikazano u Listingu 4.5.

```
1  #pragma omp parallel
2      #pragma omp single
3          par_qsort(strings, 0, num_strings - 1, compare);
```

Listing 4.5 Poziv paralelne implementacije algoritma brzog sortiranja

Pošto je algoritam brzog sortiranja sam po sebi prilično efikasan, tj. izvršava se za vreme O(n log n), mora se koristiti dovoljan broj elemenata da bi se videlo da paralelna verzija zapravo nadmašuje sekvencijalnu. Poređenje vremena izvršenja je prikazano u Tabeli 4.1. Upoređivanjem vremena izvršenja sekvencijalne verzije sa paralelnom verzijom koja radi unutar jedne niti, može se proceniti vreme potrebno za kreiranje i uništavanje OpenMP niti.

Korišćenjem 4 ili 8 niti, paralelna verzija je definitivno brža, iako brzina koju razmatramo kod algoritma brzog sortiranja nije proporcionalna broju korišćenih niti. Treba imati na umu da se podela tabele u redovima 4–14 Listinga 4.4 vrši sekvencijalno i podsetite se Amdalovog zakona.

n	sekvencijalno	paralelno		
		1 nit	4 niti	8 niti
10^5	0.05 s	0.07 s	0.04 s	0.04 s
10^6	0.79 s	0.99 s	0.44 s	0.32 s
10^7	11.82 s	12.47 s	4.27 s	3.57 s
10^8	201.13 s	218.14 s	71.90 s	61.81 s

Tabela 4.1 Poređenje vremena izvršenja sekvencijalne i paralelne verzije algoritma brzog sortiranja prilikom sortiranja n nasumično odabranih nizova maksimalne dužine 64 elementa korišćenjem četvorojezgarnog procesora sa višenitnim izvršenjem

Закључак

OpenMP се, без сумње, показао као снажан савезник у оптимизацији перформанси програмских решења, посебно у контексту проблема који се лако паралелизују. Кроз његов једноставан и интуитиван модел програмирања, програмери могу ефикасно искористити потенцијал модерних вишепроцесорских система и убрзати извршавање својих програма.

Упркос томе, важно је напоменути да успех паралелног програмирања не лежи само у алату који се користи, већ и у пажљивом разматрању проблема, правилном идентификовању делова кода који се могу паралелизовати и оптимизацији ресурса. OpenMP пружа средство за постизање ових циљева, али квалитетна имплементација захтева дубоко разумевање принципа паралелног програмирања.

Надаље, будући рад у овој области може се фокусирати на напредније аспекте OpenMP-а, истраживање напредних техника паралелизације, решавање проблема синхронизације и ефикасну употребу различитих врста меморије. Све у свему, OpenMP остаје кључни актер у домену паралелног програмирања, пружајући програмерима моћан инструмент за постизање оптималних перформанси у свету све комплекснијих рачунарских проблема.

Literatura

[1] Roman Trobec, Bostjan Slivnik, Patricio Bulic, Borut Robic: Introduction to Parallel Computing - From Algorithms to Programming on State-of-the-Art Platforms. Undergraduate Topics in Computer Science, Springer 2018, ISBN 978-3-319-98832-0, pp. 3-242

[2] Pacheco, Peter S.: An introduction to parallel programming. San Francisco 2011, ISBN 978-0-12-374260-5, pp. 1-350

[3] Blaise Barney: OpenMP, Lawrence Livermore National Laboratory, 2023, https://hpc-tutorials.llnl.gov/openmp/

[4] Michael Womack, Anjia Wang, Patrick Flynn, Xinyao Yi, Yonghong Yan: OpenMP Programming, 2022, https://passlab.github.io/OpenMPProgrammingBook/cover.html