UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Miloš Mitrović

KONKURENTNO PROGRAMIRANJE U PROGRAMSKOM JEZIKU GO

master rad

Beograd, 2017.

Mentor:

dr Milena Vujošević Janičić
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Ana Anić
Univerzitet u Beogradu, Matematički fakultet

dr Laza Lazić
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Mojoj sestri Ivoni

Naslov master rada: Konkurentno programiranje u programskom jeziku Go

Rezime: text

Ključne reči: programski jezik Go, konkurentno programiranje

Sadržaj

Glava 1

Uvod

Glava 2

Karakteristike programskog jezika Go

Go je imperativni programski jezik otvorenog koda koji razvija kompanija Google od 2007. godine. Napravljen je kao kompilirani jezik opšte namene sa statičkim tipovima koji podseća na interpretirane jezike sa dinamičkim tipovima. Podržava konkurentno programiranje, automatsko upravljanje memorijom kao i refleksiju tokom izvršavanja programa. Pogodan je za rešavanje svih vrsta problema a najviše se koristi za izgradnju servera, skriptove i samostalne aplikacije za komandnu liniju a može se korititi i za grafičke i mobilne aplikacije.

2.1 Tipovi podataka

Go je statički tipiziran jezik što znači da se varijabli dodeljuje tip prilikom njene deklaracije i on se ne može menjati tokom izvršavanja programa. Za razliku od C-a Go ne podržava automatsku konverziju tipova već se konverzija mora navesti eksplicitno, u suprotnom se prijavljuje greška.

Od osnovnih ugrađenih tipova podataka Go podržava:

- Numeričke - celobrojne označene (`int8`, `int16`, `int32`, `int64`) i neoznačene (`uint8`, `uint16`, `uint32`, `uint64`), u pokretnom zarezu (`float32`, `float64`) i kompleksne (`complex64`, `complex128`)
- Bulovske (`bool`)

- Tekstualne (`string`)

Od drugih vrsta podataka, Go podržava nizove, mape i slajseve - nizove sa promenljivom dužinom. Postoje i paketi koji omogućavaju rad sa listama.

2.2 Funkcije i metodi

2.3 Interfejsi

2.4 Refleksija

2.5 Testiranje

2.6 Paketi

Glava 3

Konkurentno programiranje u programskom jeziku Go

3.1 Osnovni pojmovi konkurentnog programiranja

3.2 Mogućnosti u programskom jeziku Go

Go-rutine

Kanali

Sinhronizacija

Data race detektor

Select naredba

```
WARNING: DATA RACE
Write at 0x00c420072006 by goroutine 18:
  main.mark_prime()
      /home/cg/root/7238354/main.go:53 +0x1d8

Previous read at 0x00c420072006 by goroutine 83:
  [failed to restore the stack]

Goroutine 18 (running) created at:
  main.Prime()
      /home/cg/root/7238354/main.go:21 +0x1a0
  main.main()
      /home/cg/root/7238354/main.go:72 +0x126

Goroutine 83 (running) created at:
  main.Prime()
      /home/cg/root/7238354/main.go:21 +0x1a0
  main.main()
      /home/cg/root/7238354/main.go:72 +0x126
```

Slika 3.1: Primer upozorenja koje „data race” detektor daje za implementaciju Eratostenovog sita

Glava 4

Primeri i poređenje sa drugim jezicima

U ovom poglavlju su izloženi primeri konkurentnih Go programa i poređenje implementacija nekoliko jednostavnih algoritama u jezicima C, C++ i Python. Primeri ilustruju na koji način se može koristiti konkurentnost u jeziku Go i kakva je njegova efikasnost u odnosu na druge pomenute jezike. Prvo sledi kratak pregled programskih jezika C, C++ i Python i njihove konkurentnosti a zatim poređenje implementacija algoritama quicksort, množenje matrica i Eratostenovo sito.

4.1 C

4.2 C++

4.3 Python

GIL (Global Iinterpreter Lock) is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes at once.

4.4 Osnovni primeri sa poređenjem

Kao kriterijumi poređenja koriste se brzina izvršavanja, maksimalna upotreba memorije i broj linija kôda. Primeri su testirani na hardveru sa 48 jezgara pod Linux-om Ubuntu 16.04 ukoliko nije naglašeno suprotno.

```
algorithm quicksort(A, lo, hi):
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1 )
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) :
    pivot := A[hi]
    i := lo - 1
    for j := lo to hi - 1 do
        if A[j] < pivot then
            i := i + 1
            swap A[i] with A[j]
    if A[hi] < A[i + 1] then
        swap A[i + 1] with A[hi]
    return i + 1
```

Slika 4.1: Pseudokod algoritma quicksort

4.5 Quicksort

Quicksort je algoritam za sortiranje brojeva u mestu koji spada u grupu algoritama podeli i vladaj. U svakom koraku, jedan element - pivot se postavlja na svoju poziciju u sortiranom nizu i deli se na dva podniza particionisanjem, jedan u kome su svi brojevi veći od pivota i drugi u kome su svi brojevi manji od pivota, koji se zatim sortiraju rekurzivno. Pseudo kod algoritma je prikazan na slici 4.1.

Paralelizacija algoritma je izvršena tako što se za svaki rekurzivni poziv pokreće po jedna nit/rutina do određene granice kada se prelazi u sekvencijalni režim rada. Za testiranje su korišćeni pseudoslučajno generisani nizovi različitih dužina.

Go

Implementacija 4.1 koristi koncept višestrukog semafora za ograničavanje broja aktivnih go-rutina. Semafor je realizovan pomoću kanala sa baferom i select naredbe gde kapacitet kanala označava maksimalan broj aktivnih go-rutina. U select naredbi se pokušava „dobijanje tokena" odnosno slanje poruke kroz kanal sa baferom. Kanal je definisan nad tipom prazne strukture jer nam nije bitna sama poruka već samo trenutno zauzeće kanala. U slučaju da možemo da dobijemo token odnosno uspešno pošaljemo praznu strukturu kroz kanal pokrećemo go-rutinu za rekurzivni poziv,

u suprotnom, ukoliko nema slodbnog mesta u baferu, rekurzivni poziv se izvršava sekvencijalno. Na kraju svake go-rutine je potrebno pročitati poruku iz kanala odnosno osloboditi jedno mesto. Da bismo bili sigurni da su sve go-rutine završile sa svojim radom, za sinhronizaciju koristimo wait grupe. Svaki konkurentni poziv funkcije kreira svoju wait grupu kojoj postavlja brojač na dva, a zatim, na kraju, čeka da oba rekurzivna poziva završe sa radom. S obzirom da unapred nije poznato kada će moći da se izvrši konkurentni a kada sekvencijalni poziv funkcije, potrebno je u oba slučaja signalizirati wait grupi da je završeno sa radom. Niz se prenosi preko reference i nije potrebno nikakvo zaključavanje jer svaki poziv fununkcije menja samo svoj deo niza.

C

Za razliku od Go implementacije, ovde je upotrebljena dubina rekurzije za ograničavanje broja niti koje program kreira. Kada se dostigne zadata dubina rekurzije program više ne kreira nove niti već prelazi u sekvencijalni režim rada. Svaki konkurentni poziv funkcije kreira po dve nove niti ukoliko maksimalna dubina nije dostignuta, nakon čega se join funkcijom čeka na njihov završetak sa radom.

C++

Za C++ su razmatrane dve implementacije: prva, u kojoj je niz reprezentovan strukturom vektor i koristi standardnu biblioteku, i druga, koja koristi običan niz int-ova i OpenMP biblioteku. Prva implementacija ima koncept dubine rekurzije za restrikciju broja niti na isti način kao što je realizovano u C-u. Druga, u kojoj je upotrebljena OpenMP biblioteka, ima mogućnost da postavi maksimalni broj aktivnih niti u jednom trenutku.

Python

Ovde se takođe razmatraju dve implementacije koje koriste različite pakete za realizaciju konkurentnosti. Kod prve implementacije koristi se threading paket i postavlja se maksimalni broj aktivnih niti. U drugoj verziji je iskorišćen Parallel Python paket sa već pomenutim konceptom dubine rekurzije za kontrolu broja niti.

Listing 4.1: Go implementacija konkurentne quicksort funkcije

```
var semaphore = make(chan struct{}, 100)

func QuickSortConcurrent(a *[]int, low, hi int) {
    if hi < low {
        return
    }

    p := partition(a, low, hi)

    wg := sync.WaitGroup{}
    wg.Add(2)

    select{
    case semaphore <- struct{}{}:
        go func(){
            QuickSortConcurrent(a, low, p-1)
            <- semaphore
            wg.Done()
        }()
    default:
        QuickSortSequential(a,low, p-1)
        wg.Done()
    }

    select{
    case semaphore <- struct{}{}:
        go func(){
            QuickSortConcurrent(a, p+1, hi)
            <- semaphore
            wg.Done()
        }()
    default:
        QuickSortSequential(a, p+1, hi)
        wg.Done()
    }

    wg.Wait()
}
```

Rezultati

Vremenska efikasnost implementacija u zavisnosti od veličine niza je prikazana u tabeli 4.1. C i C++ OpenMP implementacija su se pokazale kao najefikasnije. Vreme izvršavanja Go implementacije je uporedivo za nizove od milion i 10 miliona dok je za niz od 100 miliona brojeva potrebno dva puta više vremena u odnosu na C i C++ OpenMP implementacije. C++ verzija sa standardnom bibliotekom je značajno sporija od pomenutih implementacija, ali je Python-u potrebno najviše vremena i nije bilo mogućnosti testirati ga za niz od 100 miliona brojeva.

U odnosu na sekvencijalno izvršavanje, najveće ubrzanje ima C. Sekvencijalno se najbrže izvršava Go i za niz od milion brojeva mu je potrebno isto vremena kao i pri konkurentnom izvršavanju. Ubrzanje se vidi za nizove veće dužine međutim ono je manje u odnosu na C i C++ OpenMP verziju. C++ implementacija sa standardnom bibliotekom nema nikakvo ubrzanje kada je testirano na već pomenutom hardveru sa 48 jezgara. Međutim, testirano na drugom računaru pokazalo se da postoji ubrzanje od približno 50% kada se izvršava konkurentno. Napomena da rezultati dobijeni testiranjem na drugom računaru služe samo kao relativan odnos sekvencijalnog i konkurentog izvršavanja, a ne za poređenje sa drugim implementacijama, usled različite brzine izvršavanja.

Kod Python threading implementacije, konkurentno izvršavanje je sporije u odnosu na sekvencijalno kao direktna posledica GIL-a što je objašnjeno u odeljku 4.3. Verzija koja koristi Parallel Python paket pokazuje da je moguće napraviti konkurentan program sa nitima u Pythonu koji je efikasniji u odnosu na sekvencijalno izvršavanje. Međutim i ova verzija je višestruko sporija u odnosu na implementacije u drugim jezicima.

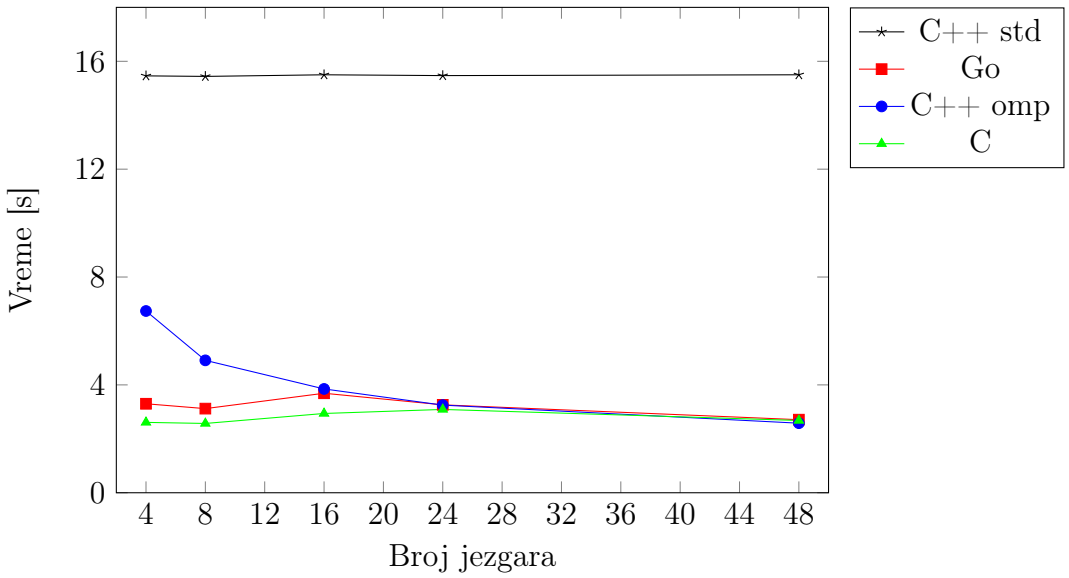
Vremenska efikasnost implementacija u zavisnosti od broja jezgara koji se koristi za izvršavanje, za niz od 10 miliona brojeva je prikazana na slici 4.2. Python nije testiran jer broj jezgara ne utiče na njegovu brzinu izvršavanja. C++ verzija sa OpenMP bibliotekom jedina ima strogo opadajuću krivu, odnosno sa većim brojem jezgara potrebno joj je manje vremena. Go i C ne osciluju značajno, osim što rade nešto sporije sa 16 i 24 jezgara, nema velike razlike u brzini sa 4 i 48 jezgara. Broj jezgara ne utiče na C++ verziju sa standardnom bibliotekom jer kao što je i prethodno testiranje pokazalo na ovom računaru konkurentno izvršavanje ne donosi nikakvo ubrzanje.

Maksimalna upotreba memorije za nizove različitih dužina je prikazana u tabeli 4.2. Obe verzije u C++-u su podjednako efikasne i potrebno im je najmanje me-

Tabela 4.1: Vreme izvršavanja [s] quicksort implementacija za različito n, testirano sa 48 jezgara

	Konkurentno izvršavanje			Sekvencijalno izvršavanje		
n [10^6] / Verzija	1	10	100	1	10	100
C	**0.33**	2.68	**14.29**	0.63	2.90	58.86
C++ omp	0.87	**2.58**	15.22	0.68	**2.64**	50.66
Go	0.49	2.71	29.36	**0.48**	4.22	**43.70**
C++ std	1.37	15.50	172.23	1.47	15.53	173.68
Python thr	28.98	340.82	-	9.71	135.32	-
C++ std*	0.35	3.52	60.96	0.68	7.80	98.74
Python pp*	4.35	58.87	-	8.46	118.33	-

*Testirano na dva jezgra/četiri niti pod Linux-om Ubuntu 17.04



Slika 4.2: Grafik brzine izvršavanja različitih quicksort implementacija u zavisnosti od broja jezgara, testirano za niz od 10 miliona brojeva

Tabela 4.2: Maksimalna upotreba memorije [MB] quicksort implementacija za različite dužine niza

n [10^6] / Verzija	1	10	100 0
C++ omp	**6.3**	42.3	**393.1**
C++ std	6.7	**42.0**	393.5
C	58.1	68.8	408.5
Go	13.4	87.5	816.3
Python thr	40.4	396.3	-
Python pp	49.6	411.8	-

Tabela 4.3: Dužine kôda quicksort implementacija

	C	Go	C++ std	C++ omp	Python pp	Python thr
Br. linija koda	119	98	84	79	55	43

morije u odnosu na druge implementacije. C je približno efikasan kao i C++ osim što i za nizove manjih dužina zahteva veliku količinu memorije. Go koristi dva puta više memorije nego ostale implementacije dok je Pythonu potrebna višestruko veća količina memorije.

Dužine programskih kodova se mogu pogledati u tabeli 4.3. U C-u je potrebno najviše linija kôda za implementaciju algoritma dok je u Pythonu potrebno najmanje.

Zaključak

Na ovom primeru Go se pokazao vremenski efikasan isto koliko i C i C++ za nizove dužine do 10 miliona dok mu je potrebno dva puta više vremena za nizove dužine od 100 miliona, ali se prilkom sekvencijalnog izvršavanja ispostavio kao najefikasniji. Potrebno mu je dva puta više memorije nego C-u i C++-u ali je i memorijski i vremenski višestruko efikasniji od Pythona. Dužina koda je uporediva sa C-om i C++-om.

4.6 Množenje matrica

Implementiran je standardni algoritam za množenje matrica. Vrednost na poziciji ij proizvoda matrica A i B se izračunava kao:

$$(AB)_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

gde je n dužina matrice A.

Algoritam je paralelizovan tako što svaka nit/rutina računa po jedan red matrice, odnosno jedan red prve matrice množi sa svim kolonama druge matrice. Za testiranje su korišćene pseudoslučajno generisane kvadratne matrice različitih dimenzija.

Go

Restrikcija broja go-rutina se ostvaruje pomoću semafora na isti način kao što je urađeno u prethodnom primeru 4.5. Može se primetiti u implementaciji 4.2 da je neophodno go-rutinama proselditi i kao argument anonimne funkcije kako bi svaka imala svoju kopiju. U suprotnom, u svakoj sledećoj iterciji for petlje vrednost i bi bila ažurirana u svim go-rutinama. Za razliku od prethodnog primera gde se niz koji se sortira prenosi pomoću reference, ovde su rezultujuća i početne matrice definisane kao globalne. Ni u ovom slučaju nije potrebno zaključavanje jer se početne matrice koriste samo za čitanje, a kod rezultujuće matrice svaka go-rutina popunjava samo svoj red.

Ostale implementacije

Implementacije u ostalim jezicima su realizovane na sličan način. Maksimalan broj niti koji se kreira u toku izvršavanja programa se unapred zadaje. Za C++ razmatrane su dve implementacije. Jedna je ostvarena pomoću OpenMP biblioteke i koristi običan niz za reprezntaciju matrice, dok druga koristi strukturu vektor i standardnu biblioteku.

Rezultati

Grafik brzine izvršavanja u zavisnosti od veličine matrice je prikazan na slici 4.3. Python se izvršava znatno sporije od ostalih implementacija tako da nije bilo

Listing 4.2: Go implementacija konkurentne funkcije za množenje matrica

```
func multiply(){
    wg := sync.WaitGroup{}
    wg.Add(n)

    for i := 0; i < n; i++ {
        select{
        case semaphore <- struct{}{}:
            go func(row int){
                multiply_row(row)
                <- semaphore
                wg.Done()
            }(i)
        default:
            multiply_row(i)
            wg.Done()
        }
    }

    wg.Wait()
}
```

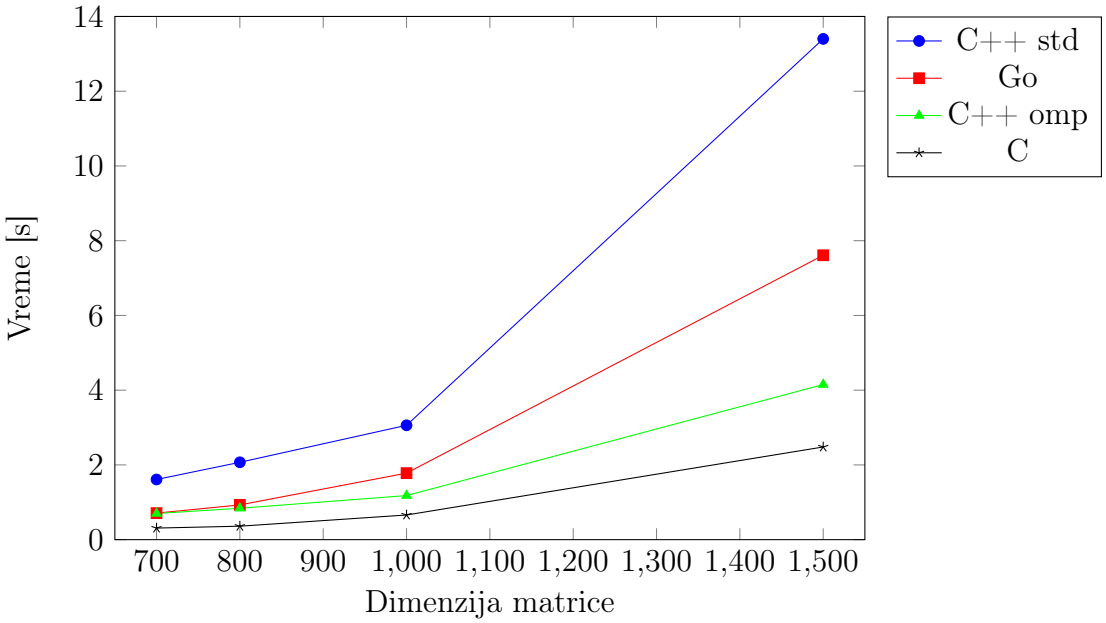
mogućnosti testirati ga na ulazima iste veličine, a rezultati testiranja Python implementacije su prikazani u tabeli 4.4. Na grafiku se vidi da je C implementacija najefikasnija, a zatim C++ verzija sa OpenMP bibliotekom, dok verzija sa standardnom bibliotekom radi najsporije. Za matrice manje veličine Go radi podjednako efikasno kao i OpenMP verzija C++ implementacije, ali za matricu veličine 1500 mu je potrebno dva puta više vremena, što je približno četiri puta više nego C-u.

Rezultati testiranja implementacija na različitom broju jezgara prikazani su na slici 4.4. Na grafiku se vidi velika razlika u brzini između sekvencijalnog (na jednom jezgru) i konkurentnog izvršavanja. Brzina raste do 16 i 24 jezgara, dok povećanje na 48 jezgara, ne donosi značajno ubrzanje.

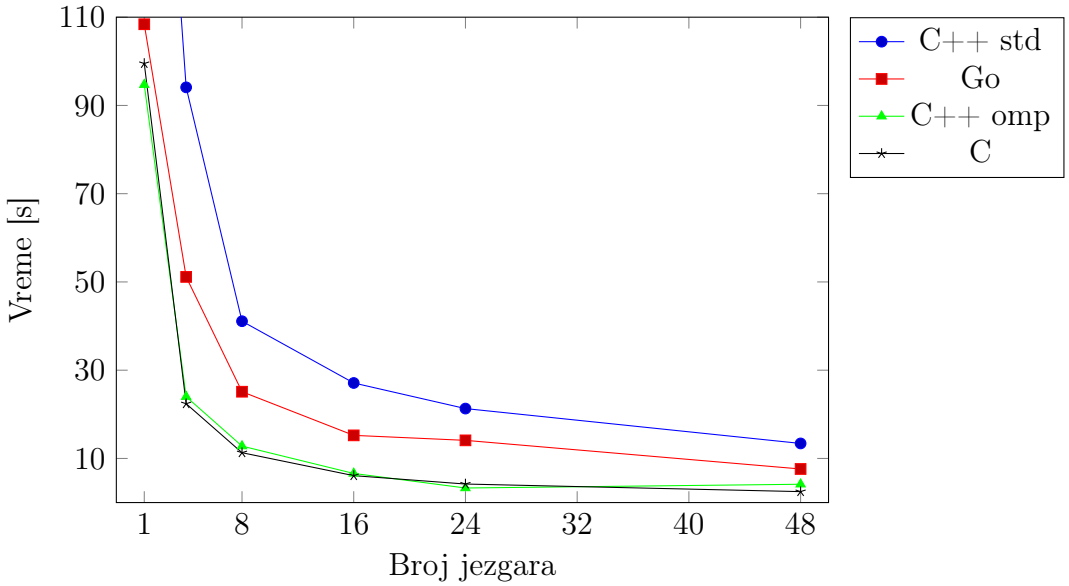
Python se izvršava višestruko sporije i potrebno mu je više od 90 sekundi za matricu veličine 500, dok je ostalim implementacijama potrebno manje od jedne sekunde. Konkurentno izvršavanje je ponovo sporije od sekvencijalnog zbog već pomenutog problema sa GIL-om u odeljku 4.3. Rezultati su prikazani u tabeli 4.4.

Grafik maksimalne upotrebe memorije u zavisnosti od dimenzije matrica je prikazan na slici 4.5. Memorijska efikasnost se u ovom primeru poklapa sa vremenskom. C i C++ OpenMP vezija su najefikasnije dok Go koristi dva puta više memorije. C++ verzija sa standardnom bibliotekom zahteva najveću količinu memorije.

Broj linija kôda svih implementacija je prikazan u tabeli 4.5. Python ima naj-



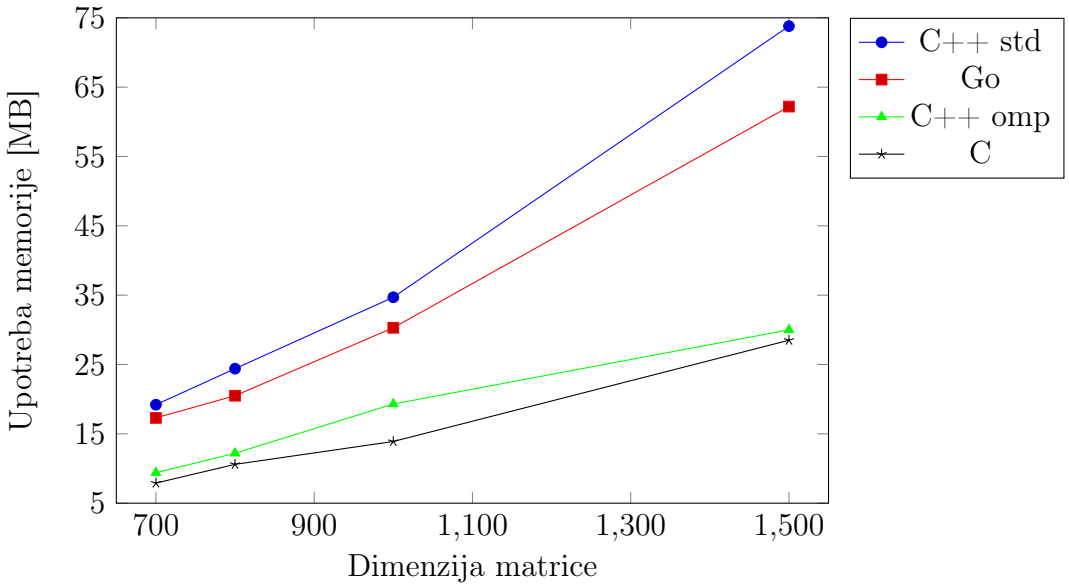
Slika 4.3: Grafik brzine izvršavanja različitih implementacija množenja matrica u zavisnosti od veličine matrice, testirano na 48 jezgara



Slika 4.4: Grafik brzine izvršavanja različitih implementacija množenja matrica u zavisnosti od broja jezgara za matricu veličine 1500

Tabela 4.4: Vreme izvršavanja i maksimalna upotreba memorije Python implementacije množenja matrica za različito n

n	Konkurentno [s]	Sekvencijalno [s]	Memorija [MB]
100	0.78	0.32	7.5
300	21.11	7.59	10.0
500	97.73	38.27	17.4



Slika 4.5: Grafik maksimalne upotrebe memorije različitih implementacija množenja matrica u zavisnosti od dimenzije matrica

Tabela 4.5: Dužine kôda implementacija množenja matrica

	C++ std	Go	C	C++ omp	Python
Br. linija koda	170	75	70	50	28

kraću implementaciju sa samo 28 linija kôda. Najduža implementacija je C++ verzija sa standardnom bibliotekom jer je razdvojena na klase i nekoliko fajlova radi jednostavnijeg korišćenja.

Zaključak

Go se ponovo pokazao vremenski efikasan koliko C i C++ za ulaze manjih dimenzija, dok za velike ulaze zahteva dva puta više vremena. Takođe, potrebna mu je

```
for i:=2 to n do
    A[i]:=true

ErathostenesSieve(n):
    for i:=2 to floor(sqrt(n)) do
         if A[i] = true:
             j := i * i
             while j < n do
                 A[j] := false
                 j := j + i
```

Slika 4.6: Pseudokod algoritma Eratostenovo sito

dva puta veća količina memorije ali je neuporedivo vremenski i memorijski efikasniji od Python-a. Potreban je približno isti broj linija kôda za implementaciju algoritma koliko i u C-u.

4.7 Eratostenovo sito

Eratostenovo sito je algoritam za određivanje prostih brojeva manjih od n. Ideja algoritma je da se elimnišu svi brojevi koji nisu prosti između 2 i n. Na početku se pretpostavlja da su svi brojevi prosti odnosno definiše se niz od n bulovskih vrednosti postavljenih na true. Kreće se od prvog prostog broja što je 2 tako što se eliminiše svaki drugi broj počevši od 2^2, a zatim se prelazi na sledeći prost broj i postupak se ponavlja. Uopšteno, za i-ti prost broj eliminiše se svaki i-ti broj počevši od i^2. Postupak je dovoljno ponoviti za proste brojeve koji su manji od $\sqrt{n}$. Pseudokod algoritma je prikazan na slici 4.6.

Paralelizacija algoritma se postiže deljenjem opsega od 2 do n na jednake delove. Svaka nit/rutina dobija svoj deo opsega u okviru kojeg eliminiše brojeve koji nisu prosti. Za svaki prost broj je prvo potrebno odrediti njegov prvi umnožak unutar opsega. Iako svaka nit/rutina ima svoj opseg, ona mora da pristupa članovima niza drugih niti/rutina jer su joj potrebni svi prosti brojevi manji od $\sqrt{n}$. To kao posledicu dovodi do mogućnosti da se u nekim slučajevima bespotrebno eliminišu umnošci brojeva koji nisu prosti ukoliko ih druga nit/rutina još uvek nije eliminisala. Problem je rešen tako što se proverava dodatni uslov prilikom eliminacije: da li je neka druga nit/rutina u međuvremenu označila da taj broj nije prost.

Listing 4.3: Go implementacija konkurentne funkcije za određivanje prostih brojeva manjih od n

```
func Prime(list *[]bool, n int, is_concurrent bool){
    sqrt := int(math.Sqrt(float64(n)))
    first := 0
    step := int(n/ num_goroutines)
    last := step
    wg := sync.WaitGroup{}
    wg.Add(num_goroutines)

    for i:=0; i < num_goroutines-1; i++{
        go mark_prime(list,first,last,sqrt,&wg,true)
        first = last + 1
        last += step
    }

    mark_prime(list,first,n-1,sqrt,&wg)
    wg.Wait()
}
```

Go

Koristi se globalni niz od n bulovskih promenljivih postavljenih na podrzumevanu vrednost - false umesto na true radi jednostavnosti. Funkcija koja kreira go-rutine je prikazana u listingu 4.3. Za svaki broj koji je trenutno označen kao prost, najpre je potrebno je odrediti njegov prvi umnožak, a zatim, označiti sve njegove umnoške unutar opsega, što je i prikazano u listingu 4.4. Kao što je već pomenuto, ako svaka go-rutina ima svoj opseg, ona mora da pristupa i članovima niza drugih go-rutina jer su joj potrebni svi prosti brojevi manji od $\sqrt{n}$. To kao posledecu dovodi do pojave data race-a, međutim, u ovom slučaju je to dopustivo i nisu potrebni muteksi upravo zato što proveravamo dodatni uslov da li je pročitana vrednost u međuvremenu bila menjana. Ako se data race detektor pozove, dobija se izveštaj koji upozorava da postoji data race. Primer izveštaja se može videti na slici 3.1.

Listing 4.4: Go implementacija konkurentne funkcije za označavanje prostih brojeva

```
func mark_prime(list *[]bool,first,last,sqrt int,wg *sync.WaitGroup){
    for i:=2; i<= sqrt && i*i<= last; i++{
        if !(*list)[i] {
            var j int
            if i*i < first {
                if (first - i*i)%i == 0 {
                    j = i*i + ((first-i*i)/i)*i
                }else {
                    j = i*i + ((first-i*i)/i + 1)*i
                }
            }else {
                j = i*i
            }

            for ; j <= last && !(*list)[i]; j+=i {
                (*list)[j] = true
            }
        }
    }
    wg.Done()
}
```

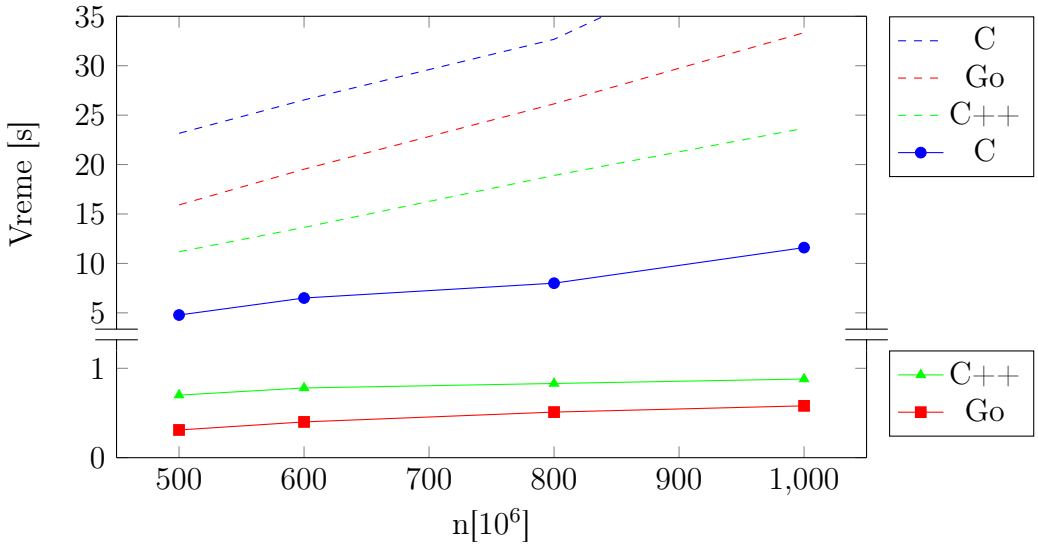
Ostale implementacije

Implementacije u ostalim jezicima su realizovane na isti načini. Za C++ je razmatrana samo jedna implementacija koja koristi OpenMP biblioteku.

Rezultati

Vreme konkurentnog i sekvencijalnog izvršavanja implementacija je predstavljeno grafikom na slici 4.7. Python ponovo nije mogao da bude uključen usled znatno sporijeg izvršavanja, a rezultati su prikazani u tabeli 4.6. Go implementacija se pokazala kao najefikasnija a zatim implementacija u C++-u. C je znatno sporiji i njegovo izvršavanje se meri u sekundama dok Go i C rade ispod jedne sekunde. U odnosu na sekvencijano izvršavanje, Go ima najveće ubrzanje. Pri konkurentnom izvršavanju, veličina ulaza nema značajan uticaj tako da za Go i C++ nema velike razlike u brzini kada n iznosi 500 miliona ili 1 bilion.

Grafik brzine izvršavanja implementacija u zavisnosti od broja jezgara je prikazan na slici 4.8. Za Go i C++ je potrebno manje vremena za izvršavanje sa većim brojem jezgara, međutim povećanje sa 24 na 48 jezgara ne donosi značajno ubrzanje. Pokazalo se da C radi sporije sa 8 i 16 jezgara nego sa 4, ali je ipak najefikasniji



Slika 4.7: Grafik brzine izvršavanja različitih implementacija Eratostenovog sita za različito n, testirano na 48 jezgara; isprekidanom linjom je prikazano sekvencijalno izvršavanje dok je konkurentno prikazano punom linijom

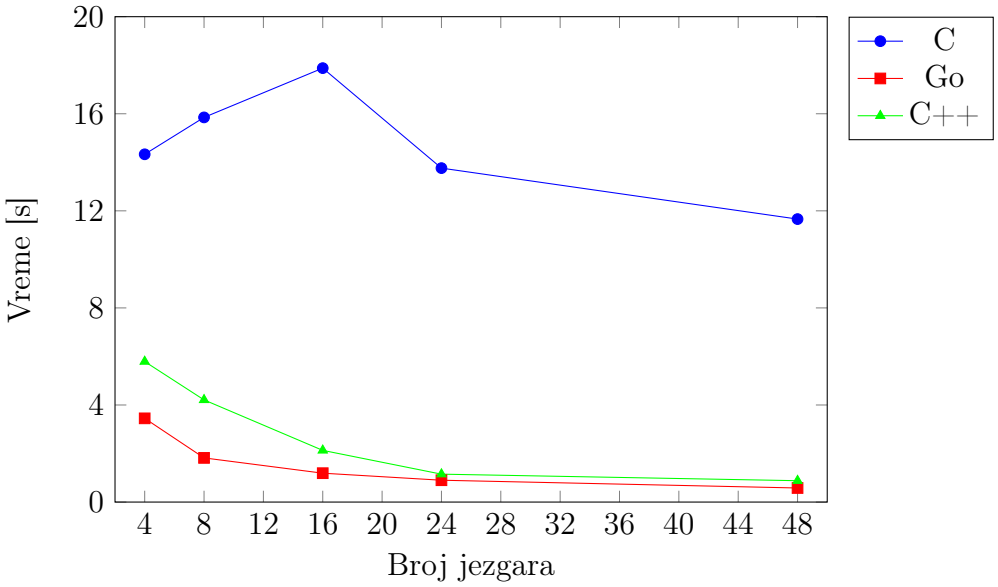
Tabela 4.6: Vreme izvršavanja i maksimalna upotreba memorije Python implementacije Eratostenovog sita za različito n

$n[10^6]$	Konkurentno [s]	Sekvencijalno [s]	Memorija [MB]
1	0.57	0.32	13.5
10	9.92	3.53	51.0
30	32.18	12.04	129.9
50	52.76	19.42	201.2
100	113.80	57.58	397.1

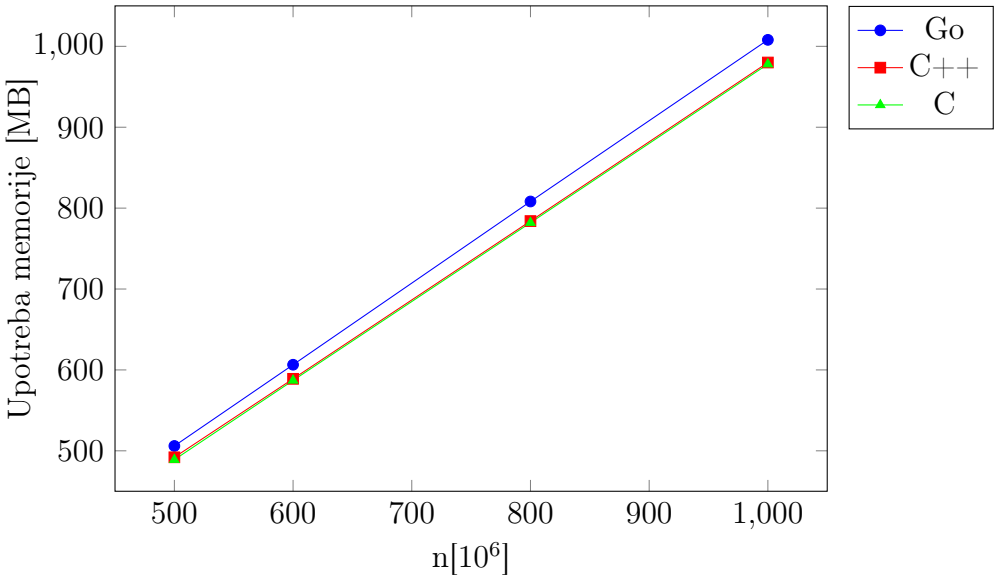
sa 48 jezgara.

Python je testiran za 10 puta manju vrednost n u odnosu na ostale implementacije i potrebno mu je više od jednog minuta, dok Go za 10 puta veću vrednost radi ispod jedne sekunde. Upotreba memorije je 4 puta veća nego kod ostalih implementacija i potrebno mu je više vremena pri konkurentnom izvršavanju nego pri sekvencijalnom usled već pomenutog problema sa GIL-om u odeljku 4.3.

Maksimalna upotreba memorije u zavisnosti od n prikazana je na slici 4.9. Sve tri implementacije imaju približno istu upotrebu memorije. Na grafiku se vidi približno linearna zavisnost y=x, što znači da je potrebno onoliko MB memorije koliko miliona iznosi vrednost n.



Slika 4.8: Grafik brzine izvršavanja različitih implementacija Eratostenovog sita u zavisnosti od broja jezgara



Slika 4.9: Grafik maksimalne upotrebe memorije različitih implementacija Eratostenovog sita za različito n

Tabela 4.7: Dužine kôda implementacija Eratostenovog sita

	C	Go	C++	Python
Br. linija koda	89	78	61	42

Dužine implementacija su prikazane u tabeli 4.7. I u ovom primeru najkraća implementacija je u Python-u dok je u C-u najduža. Go se ponovo nalazi između C-a i C++-a po broju linija kôda.

Zaključak

Za ovaj algoritam Go implementacija se ispostavila kao vremenski najefikasnija sa značajnim ubrzanjem u odnosu na sekvencijalno izvršavanje. Upotreba memorije je približno ista kolika je i kod C-a i C++-a kao i dužina kôda implementacije.

4.8 Složeniji primer u programskom jeziku Go

Glava 5

Zaključak