

UNIVERZITET U BEOGRADU  
MATEMATIČKI FAKULTET

Miloš Mitrović

# **KONKURENTNOST U PROGRAMSKOM JEZIKU GO**

master rad

Beograd, 2017.

**Mentor:**

dr Milena VUJOŠEVIĆ JANIČIĆ

Univerzitet u Beogradu, Matematički fakultet

**Članovi komisije:**

dr Vesna MARINKOVIĆ

Univerzitet u Beogradu, Matematički fakultet

dr Milan BANKOVIĆ

Univerzitet u Beogradu, Matematički fakultet

**Datum odbrane:** \_\_\_\_\_

*Mojoj sestri Ivoni*

**Naslov master rada:** Konkurentnost u programskom jeziku Go

**Rezime:** text

**Ključne reči:** programski jezik Go, konkurentno programiranje

# Sadržaj

<b>1</b>	<b>Uvod</b>	<b>1</b>
<b>2</b>	<b>Karakteristike programskog jezika Go</b>	<b>2</b>
2.1	Tipovi podataka . . . . .	2
2.2	Funkcije i metodi . . . . .	3
2.3	Interfejsi . . . . .	3
2.4	Refleksija . . . . .	3
2.5	Garbage collection . . . . .	3
2.6	Testiranje . . . . .	3
2.7	Paketi . . . . .	3
<b>3</b>	<b>Konkurentno programiranje u jeziku Go</b>	<b>4</b>
3.1	Osnovni pojmovi konkurentnog programiranja . . . . .	4
3.2	Mogućnosti u programskom jeziku Go . . . . .	4
<b>4</b>	<b>Poređenje sa drugim programskim jezicima</b>	<b>6</b>
4.1	C . . . . .	6
4.2	C++ . . . . .	6
4.3	Python . . . . .	6
4.4	Poređenje na primerima jednostavnih algoritama . . . . .	6
4.5	Quicksort . . . . .	7
4.6	Množenje matrica . . . . .	13
4.7	Eratostenovo sito . . . . .	17
<b>5</b>	<b>Primer upotrebe programskog jezika Go</b>	<b>24</b>
5.1	Serverska aplikacija . . . . .	24
<b>6</b>	<b>Zaključak</b>	<b>36</b>



# 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 koristiti 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.

## 2.2 Funkcije i metodi

## 2.3 Interfejsi

## 2.4 Refleksija

## 2.5 Garbage collection

## 2.6 Testiranje

## 2.7 Paketi

## Glava 3

# Konkurentno programiranje u 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

# Poređenje sa drugim programskim 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 Interpreter Lock) is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes at once.

### 4.4 Poređenje na primerima jednostavnih algoritama

Kao kriterijumi poređenja koriste se prosečna brzina izvršavanja, maksimalna upotreba memorije i broj linija kôda. Primeri su testirani na hardveru sa 48 jezgara

```
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

pod Linux-om Ubuntu 16.04 ukoliko nije naglašeno suprotno.

## 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 rekursivni poziv, u suprotnom, ukoliko nema slobodnog mesta u baferu, rekursivni 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 rekursivna 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 funkcije 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

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()
}
```

postavlja se maksimalni broj aktivnih niti. U drugoj verziji je iskorišćen Parallel Python paket sa već pomenutim konceptom dubine rekurzije za kontrolu broja niti.

## 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 postoji za nizove veće dužine međutim ono je manje u odnosu na C i C++ OpenMP verziju.

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. Napomena da rezultati dobijeni testiranjem na drugom računaru služe samo kao relativan odnos sekvencijalnog i konkurentnog izvršavanja, a ne za poređenje sa drugim implementacijama, usled različite brzine izvršavanja.

U tabeli 4.2 je prikazano prosečno vreme izvršavanja implementacija sa različitim brojem niti/rutina. Python implementacija nije testirana iz razloga što se sa većim brojem niti izvršava sve sporije. Najbolje vreme se postiže sa 100 niti/rutina za Go i C++ OpenMP verziju i sa 1000 niti za C i C++ verziju sa standardnom bibliotekom. Prilikom ostalih testiranja je korišćen onaj broj niti/rutina za koji implementacija pokazuje najbolje rezultate. Svi prikazani rezultati Python implementacije su testirani sa 30 niti.

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. Ubrzanje sa povećanjem broja jezgara je najzastupljenije kod C++ verzije sa OpenMP bibliotekom.



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

Verzija \ n [10 <sup>6</sup> ]	Konkurentno izvršavanje			Sekvencijalno izvršavanje		
	1	10	100	1	10	100
C	<b>0.33</b>	2.68	<b>14.29</b>	0.63	2.90	58.86
C++ omp	0.87	<b>2.58</b>	15.22	0.68	<b>2.64</b>	50.66
Go	0.49	2.71	29.36	<b>0.48</b>	4.22	<b>43.70</b>
C++ std	1.58	3.67	31.68	1.72	17.02	194.13
Python thr	28.98	340.82	-	9.71	135.32	-
Python pp*	4.35	58.87	-	8.46	118.33	-

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

Tabela 4.2: Prosečno vreme izvršavanja [s] quicksort implementacija sa različitim brojem niti, testirano sa 48 jezgara za niz dužine 10 miliona

Br. niti/dubina rek.	10/3	30/5	100/7	1000/10	4000/12
Go	3.27	3.10	<b>2.71</b>	3.41	3.46
C	3.82	3.53	2.97	<b>2.40</b>	4.12
C++ omp	3.85	3.36	<b>2.58</b>	2.61	2.68
C++ std	6.44	5.39	4.37	<b>3.77</b>	4.82

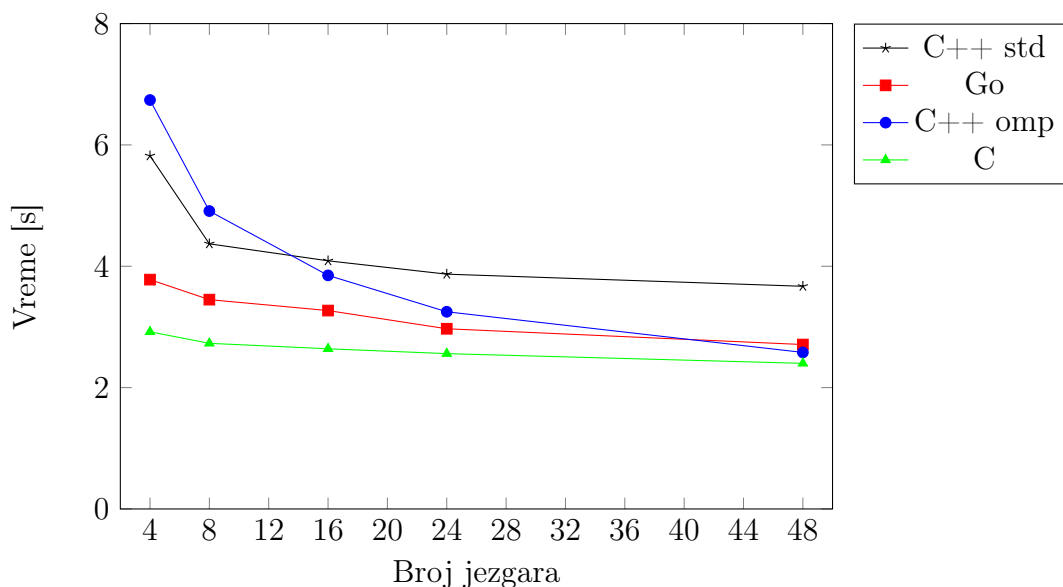
Kod ostalih implementacija, najveća razlika u brzini se vidi između izvršavanja sa 4 i 8 jezgara dok je razlika između 24 i 48 jezgara minimalna.

Maksimalna upotreba memorije za nizove različitih dužina je prikazana u tabeli 4.3. Obe verzije u C++-u su podjednako efikasne i potrebno im je najmanje memorije 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 4.4. 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 ni-



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

Tabela 4.3: Maksimalna upotreba memorije [MB] quicksort implementacija za različite dužine niza

Verzija \ n [10 <sup>6</sup> ]	n [10 <sup>6</sup> ]		
	1	10	100 0
C++ omp	<b>6.3</b>	<b>42.3</b>	<b>393.1</b>
C++ std	25.2	55.0	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 4.4: Dužine kôda quicksort implementacija

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

zove dužine od 100 miliona, ali se prilikom 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 proslediti  $i$  kao argument anonimne funkcije kako bi svaka imala svoju kopiju. U suprotnom, u svakoj sledećoj iteraciji 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 reprezentaciju matrice, dok druga koristi strukturu vektor i standardnu biblioteku.

Listing 4.2: Go implementacija konkurentne funkcije za množenje matrica

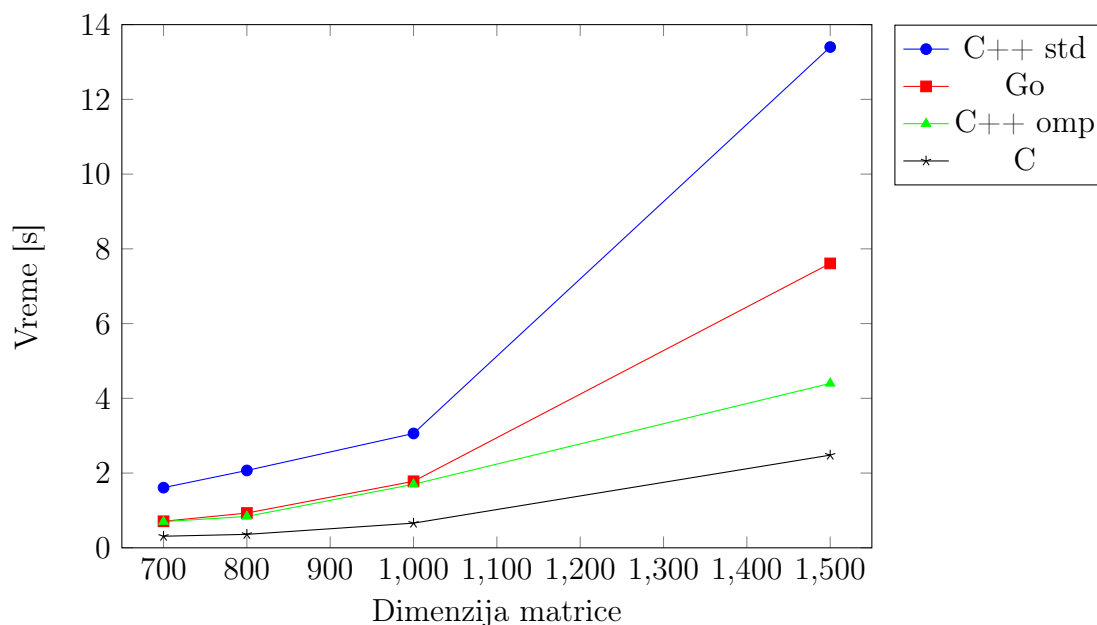
```
func multiply() {  
    wg := sync.WaitGroup{}  
    wg.Add(n)  
  
    for i := 0; i < n; i++ {  
        select {  
        case semaphore <- struct{}{}:  
            go func(row int) {  
                multiply_row(row)  
                <- semaphore  
                wg.Done()  
          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

## 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 mogućnosti testirati ga na ulazima iste veličine, a rezultati testiranja Python implementacije su prikazani u tabeli 4.6. 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.

Prosečno vreme izvršavanja sa različitim brojem niti za matricu veličine 1000 je prikazano u tabeli 4.5. Najbolje vreme se ponovo postiže sa 100 niti/rutina za Go i C++ OpenMP verziju i sa 1000 niti za C i C++ verziju sa standardnom bibliotekom. Kod ostalih testiranja je korišćen onaj broj niti/rutina za koji implementacija pokazuje najbolje rezultate.

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.



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

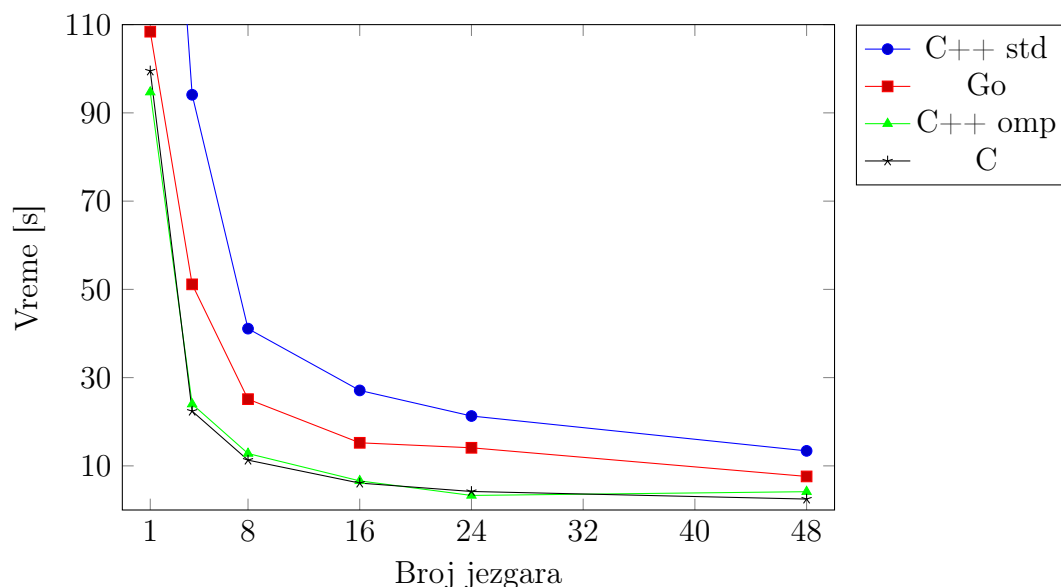
Tabela 4.5: Prosečno vreme izvršavanja [s] implementacija množenja matrica sa različitim brojem niti, testirano sa 48 jezgara za matrice veličine 1000

Br. niti	10	30	100	1000
Go	6.39	3.01	<b>1.78</b>	1.82
C	2.74	1.39	0.92	<b>0.81</b>
C++ omp	3.03	1.95	<b>1.70</b>	1.97
C++ std	15.31	8.71	3.21	<b>3.06</b>

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 4.6.

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 verzija 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 4.7. Python ima najkraću implementaciju sa samo 28 linija kôda. Najduža implementacija je C++



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

Tabela 4.6: Prosečno 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

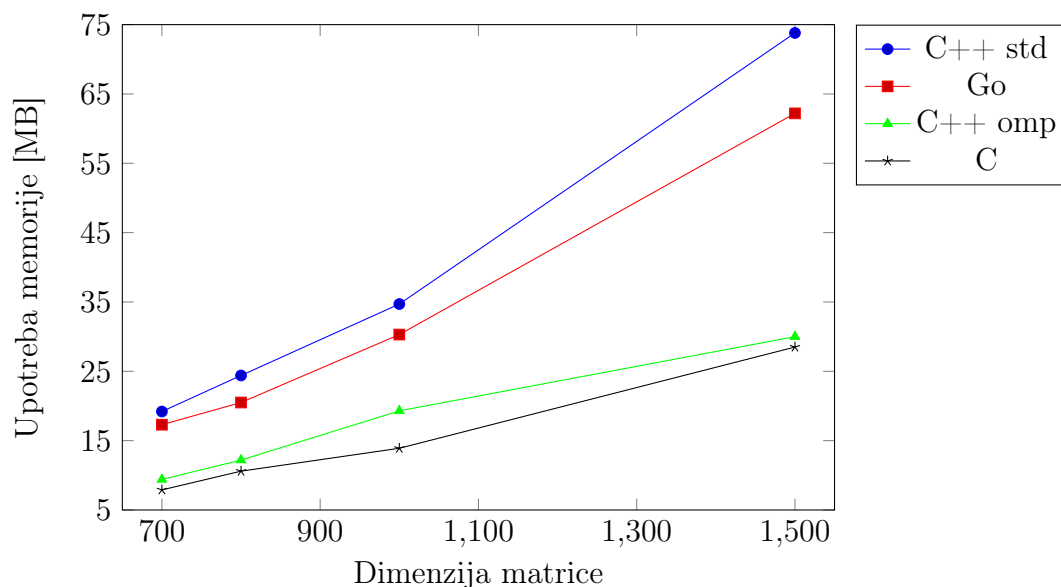
Tabela 4.7: Dužine kôda implementacija množenja matrica

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

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 dva puta veća količina memorije ali je neuporedivo vremenski i memorijski efikasniji



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

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 eliminišu 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

```
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

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.

## Go

Koristi se globalni niz od  $n$  bulovskih promenljivih postavljenih na podrazumevanu 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 4.3: 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)

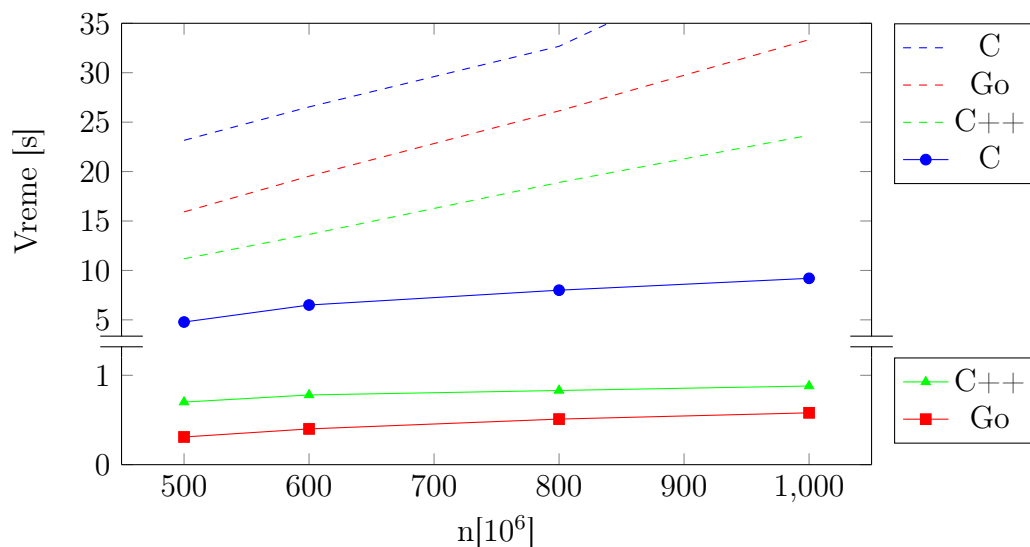
    for i:=0; i < num_goroutines-1; i++{
        go mark_prime(list, first, last, sqrt, &wg, true)
        first = last + 1
        last += step
    }

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

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)[j]; j+=i {
                (*list)[j] = true
            }
        }
    }
    wg.Done()
}
```



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

## Ostale implementacije

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

## Rezultati

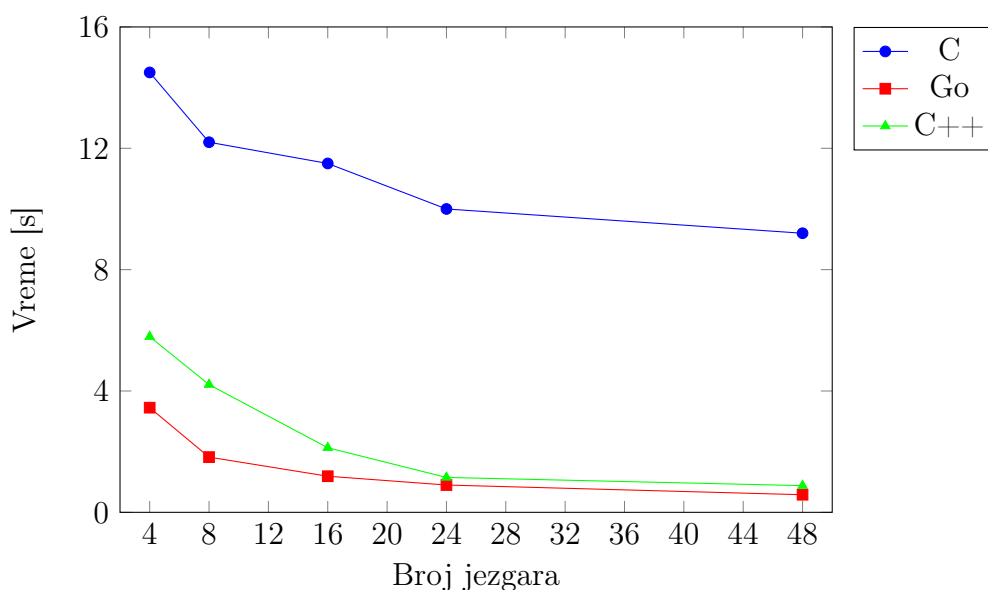
Prosečno vreme konkurentnog i sekvencijalnog izvršavanja implementacija je predstavljeno grafikom na slici 4.7. Python ponovo nije mogao da bude uključen usled znatno sporijeg izvršavanja, a rezultati su prikazani u tabeli 4.9. Go implementacija se pokazala kao najefikasnija a zatim implementacija u 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.

U tabeli 4.8 je prikazano vreme izvršavanja sa različitim brojem niti/rutina kada je  $n = 500$  miliona. Sve tri implementacije postižu najbolje performanse sa 1000 niti/rutina što je iskorišćeno prilikom svih ostalih testiranja.

Grafik brzine izvršavanja implementacija u zavisnosti od broja jezgara je pri-

Tabela 4.8: Prosečno vreme izvršavanja [s] implementacija Eratostenovog sita sa različitim brojem niti, testirano sa 48 jezgara za  $n = 500$  miliona

Br. niti	10	100	1000	10000
Go	2.14	0.40	<b>0.31</b>	0.49
C	6.17	4.81	<b>4.74</b>	7.85
C++	4.53	2.84	<b>0.73</b>	0.76



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

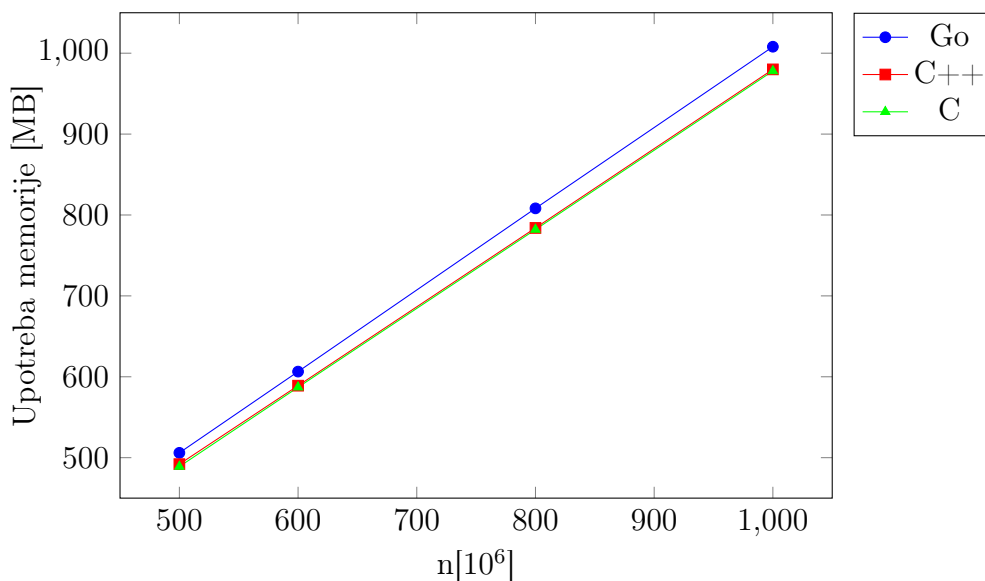
kazan na slici 4.8. Kod svih implementacija postoji ubrzanje sa povećanjem broja jezgara, ali ne postoji značajna razlika u brzini prilikom izvršavanja sa 24 i 48 jezgara.

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  $n$ .

Tabela 4.9: Prosečno 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



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

Tabela 4.10: Dužine kôda implementacija Eratostenovog sita

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

Dužine implementacija su prikazane u tabeli 4.10. 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.

## Glava 5

# Primer upotrebe programskog jezika Go

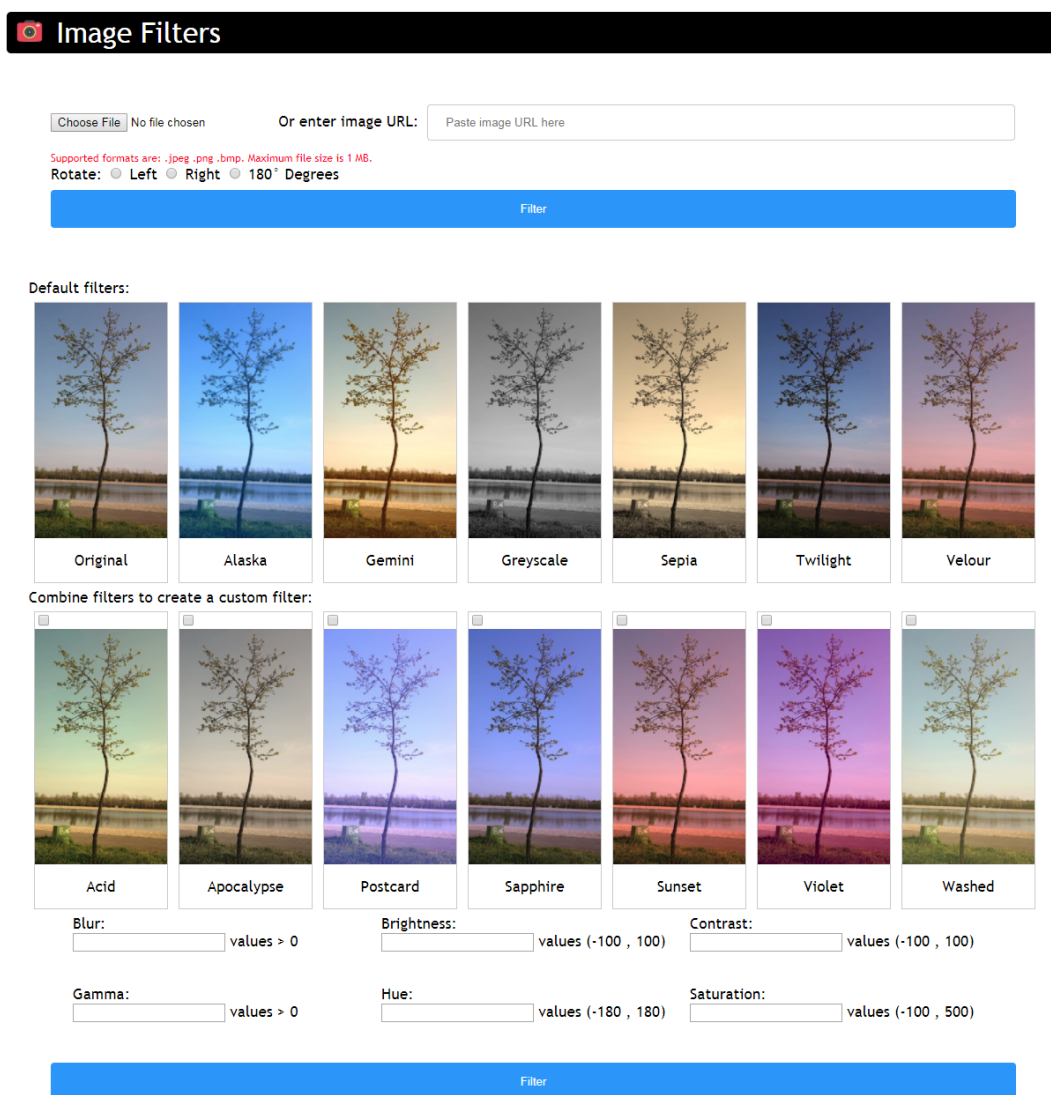
Kao primer upotrebe programskog jezika Go, razvijena je serverska aplikacija koja demonstrira korišćenje konkurentnosti kao i drugih aspekata jezika. U ovom poglavlju je predstavljena struktura aplikacije i opisani su pojedinačni delovi koji ilustruju različite karakteristike programskog jezika.

### 5.1 Serverska aplikacija

Razvijena serverska aplikacija se bavi primenom različitih filtera na slikama. Omogućava primenu postojećih predefisanih filtera kao i kreiranje sopstvenog, kombinovanjem različitih ponuđenih filtera i unošenjem željenih vrednosti za pojedine karakteristike. Slika koje se obrađuje se može upload-ovati sa računara ili se može proslediti njen URL. Svi filteri se primenjuju paralelno nakon čega se mogu videti pojedinačni rezultati obrade koji su dostupni za preuzimanje. Grafički interfejs aplikacije je prikazan na slici 5.1.

#### Kreiranje servera

Za kreiranje servera je korišćen `net/http` paket koji obezbeđuje skup funkcija za jednostavnu implementaciju i upravljanje serverom. Primer jednog jednostavnog servera je prikazan na listingu 5.2.



Slika 5.1: Grafički interfejs serverske aplikacije za primenu filtera

Listing 5.1: Main funkcija, kreiranje servera

```
func main() {
    fmt.Println("Starting server...")
    rand.Seed(time.Now().UTC().UnixNano())
    http.HandleFunc("/", DefaultHandler)
    http.HandleFunc("/results", ImageHandler)
    http.Handle("/data/", http.HandlerFunc(file_server))
    http.ListenAndServe(address + ":http", nil)
}
```

Listing 5.2: Primer jednostavnog servera

```
package main
import ("net/http"; "fmt")

func hello(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Hello!")
}
func main() {
    http.HandleFunc("/", hello)
    http.ListenAndServe(":12345", nil)
}
```

Funkcija `HandleFunc` se koristi za registrovanje handler funkcija za određeni šablon. Kao parametre, handler funkcija ima `ResponseWriter` koji se koristi za konstrukciju http odgovora i `Request` koji sadrži sve podatke http zahteva. `ListenAndServe` funkcija osluškuje na zadatoj adresi (prvi parametar) TCP mreže i poziva `Serve` funkciju kojoj prosleđuje `Handler` (drugi parametar). U pozadini, funkcija `Serve` kreira novu gorutinu za svaku http konekciju i poziva `Handler`. Ukoliko je `Handler` `nil`, koristi se `DefaultServeMux` multiplekser koji uparuje URL zahteve sa registrovanim šablonima i poziva odgovarajuću handler funkciju.[2]

Server se može implementirati u Go-u koristeći samo 2 paketa i manje od 10 linija koda što je karakteristično za dinamički tipizirane, interpretirane jezike kao što su Python, PHP, Ruby i drugi. U C/C++ -u, koji su statički tipizirani jezici koji se kompiliraju, kao što je i Go, je potrebno 9 biblioteka i više od 50 linija koda.<sup>1</sup>

Deo koda koji se odnosi na kreiranje servera u aplikaciji se nalazi u `main` funkciji koja je prikazana u listingu 5.1. Registrovane su dve handler funkcije: `DefaultHandler` koja se poziva kada se pristupa početnoj strani i `ImageHandler` koja se poziva prilikom obrade slike i prikazivanja rezultata. `Handle` funkcija se koristi za registrovanje `Handler`-a za fajl sistem i šablona koji se odnosi na zahteve resursa. U `ListenAndServe` funkciji je postavljeno da se osluškuje na http portu zadate adrese servera koja je u ovom slučaju postavljena na `localhost`.

`Handler` za fajl sistem je prikazan u listingu 5.3. U ovoj funkciji se definišu restrikcije nad URL zahtevima za resurse i definiše se root direktorijum svih resursa. URL se deli na segmente funkcijom `split` i zatim se ispituje poslednji segment. Ukoliko je poslednji segment prazan odnosno ako se URL završava sa `'/'` to znači da

---

<sup>1</sup>[https://rosettacode.org/wiki/Hello\\_world/Web\\_server#C](https://rosettacode.org/wiki/Hello_world/Web_server#C)



u zahtevu nije tražen određeni fajl već je u pitanju samo deo putanje. `DefaultServeMux` funkcioniše tako što dodaje `'/'` na kraju svakog URL zahteva koji postoji u podstablu root direktorijuma, što znači da ako korisnik unese putanju do nekog direktorijuma, poslednji deo URL zahteva će biti prazan.[2] U tom slučaju korisniku će biti onemogućen pristup direktorijumu i neće moći da pročita njegov sadržaj.

Listing 5.3: Handler za fajl sistem

```
func file_server(w http.ResponseWriter, r *http.Request) {
    parts := strings.Split(r.URL.Path, "/")
    last := parts[len(parts)-1]
    if last == "" {
        http.NotFound(w, r)
        return
    }
    fileServer := http.StripPrefix("/data/",
        http.FileServer(http.Dir("data")))
    fileServer.ServeHTTP(w, r)
}
```

## Generisanje HTML stranice

Paket `html/template` se koristi za generisanje HTML izlaza sa datim parametrima koji ima zaštitu protiv umetanja koda. Paket nudi veliki broj različitih escape funkcija koje kodiraju specijalne karaktere ali u ovoj aplikaciji nije bilo potrebe za njihovom upotrebom.

U `DefaultHandler` funkciji se parsira i izvršava template koji se prikazuje kada se pristupi početnoj strani. Deo funkcije koji se odnosi na izvršavanje template-a je prikazan u listingu 5.4. Ovde kao i na drugim mestima, greške će biti ignorisane jer se podrazumeva da su parametri funkcije ispravni i da neće doći do greške. Funkcija `ExecuteTemplate` kao argumente prima `ResponseWriter`, naziv template-a koji se izvršava i podatke koji se koriste za njegovo popunjavanje. Prosleđena promenljiva `data` je definisana struktura koja sadrži podatke o filterima u obliku mapa.

Listing 5.4: Izvršavanje HTML šablona

```
func DefaultHandler(w http.ResponseWriter, r *http.Request) {
    ...
    tmpl, _ := template.ParseFiles("index.html")
    tmpl.ExecuteTemplate(w, "index", data)
}
```

U template-u sve instrukcije, podaci i kontrole toka se navode između dvostrukih vitičarstih zagrada. Primer template-a je prikazan u listingu 5.5. Na početku svakog template-a se pomoću `define` definiše naziv, a kraj template-a je potrebno naznačiti sa `end`. Omogućeno je iteriranje nad prosleđenim podacima, `if-else` naredbe i izvršavanje template-a unutar template-a. U ovom slučaju se iterira nad mapom `Filters` koja sadrži nazive filtera i putanje do njihovih slika. Prosleđeni podaci počinju sa znakom `'.'`, dok promenljive počinju sa znakom `'$'`. [5]

Listing 5.5: Izvršavanje HTML šablona

```
{{ define "index" }}
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Image Filters</title>
  </head>
  <body>
    ...
    {{ range $name, $image := .Filters }}
    <div class="gallery">
      
      <div class="desc">{{ $name }}</div>
    </div>
    {{ end }}
    ...
  </body>
</html>
{{ end }}
```

## Preuzimanje i procesiranje fajla

Funkcija `ImageHandler` se poziva kada korisnik klikne na dugme filter i bavi se preuzimanjem fajla, proverom zadovoljenosti svih uslova i primenom filtera. Struktura funkcije je prikazana u listingu 5.6. U funkciji se nalazi beskonačna petlja sa `select` naredbom. Semafor se koristi za kontrolu maksimalnog broja konekcija koji server može da opsluži. Ukoliko ima slobodnih mesta, izvršiće se glavni deo funkcije nakon čega se izlazi iz petlje. U slučaju da nema slobodnih mesta, proverava se da li korisnik čeka duže od dozvoljenog vremena za čekanje - `timeout`. Ako je vreme čekanja isteklo, izlazi se iz petlje i korisniku ispisuje poruka da je server trenutno zauzet, u suprotnom se ponovo proverava da li ima slobodnih mesta.

Listing 5.6: Struktura ImageHandler funkcije

```
func ImageHandler(w http.ResponseWriter, r *http.Request) {
    start := time.Now()
    done := false

    for ;; {
        select {
        case semaphore <- struct{}{}:
            defer func() { <-semaphore }()
            ...
            done = true
        default:
            if time.Since(start) > timeout {
                ErrorHandler(w, "Server too busy, try again later... ")
                done = true
            }
        }
        if done { break }
    }
}
```

Deo funkcije koji se odnosi na otvaranje i proveru fajla je prikazan u listingu 5.7. U funkciji se koriste dve korisnički definisane strukture: `file_info`, koja sadrži neophodne informacije o fajlu (naziv, veličina, referenca na otvoreni fajl, čitač fajla), i `processor`, tip nad kojim su definisani metodi koji koriste `ResponseWriter` i `Request` kako se ne bi prosleđivali pojedinačno svakoj funkciji koja ih koristi. Ovi metodi tokom svog rada proveravaju da li je došlo do neke greške pa kao rezultat vraćaju bool promenljivu kako bi funkcija `ImageHandler` prestala sa daljim radom. Generalno, validaciju fajla je prirodnije raditi na strani klijenta u javascript-u kako ne bi opterećivali server lošim zahtevima, ali u ovom slučaju je demonstrirano kako se validacija izvršava u Go-u.

Metod `open_file` otvara fajl i popunjava strukturu `file_info` na odgovarajući način u zavisnosti da li je prosleđen URL fajla ili je on upload-ovan. Fajlu i svim ostalim elementima forme koje je korisnik uneo se pristupa pomoću `ResponseWriter`-a pozivanjem metoda `Form` ili `FormValue`. Nakon završetka sa radom svakog fajla, neophodno ga je zatvoriti. To se postiže na najbolji način korišćenjem `defer` naredbe odmah nakon otvaranja jer će se u tom slučaju zatvaranje sigurno izvršiti čak i ukoliko dođe do neke greške. U zavisnosti kako je prosleđen, tip fajla se razlikuje pa se u strukturi nalaze polja za obe vrste fajla.

Listing 5.7: Otvaranje i provera fajla u funkciji ImageHandler

```
var f_info file_info
p := processor{w,r}

r.Body = http.MaxBytesReader(w, r.Body, safety_max_file_size)
if !p.open_file(&f_info) {return}

if f_info.Url_file != nil {
    defer f_info.Url_file.Close()
} else if f_info.Source_file != nil {
    defer f_info.Source_file.Close()
}

if !p.check_file_size(f_info.File_size){return}

dir_path := create_user_directory()
extension, original := create_and_copy_file(dir_path, f_info)
defer original.Close()

if !p.check_file_type(dir_path, original) {return}
```

Provera veličine fajla se izvršava u metodu `check_file_size`. Međutim, korisnik je i dalje u mogućnosti da upload-uje fajl nedozvoljene veličine pre nego što dođe do ove provere i time opteretiti server. Iz tog razloga se koristi funkcija `MaxBytesReader` koja prekida konekciju sa klijentom ukoliko je pređena dozvoljena veličina fajla.[2] Ovom funkcijom je postavljena gornja bezbednosna granica za veličinu (u ovom slučaju 10 MB), kako bi funkcija `check_file_size` mogla da obavesti korisnika ako je slučajno prosledio fajl nešto iznad dozvoljene granice (u ovom slučaju 1 MB).

Funkcija `create_user_directory` kreira privremeni direktorijum u kome smešta sve korisnikove slike. Naziv direktorijuma je pseudoslučajni string dužine 20 kako bi naziv bio jedinstven i direktorijum uvek mogao da se kreira. Nakon toga se fajl kopira funkcijom `create_and_copy_file` koja kao rezultat vraća pokazivač na otvoreni fajl i string sa njegovom ekstenzijom. `Defer` naredbom se osigurava zatvaranje samog fajla. Tip fajla se proverava metodom `check_file_type` koji koristi `DetectContentType` funkciju iz `http` paketa zbog sigurnosti jer sama ekstenzija fajla nije dovoljna.

## Primena filtera

Za rad sa slikama koristi se osnovni image paket a za primenu filtera je iskorišćen GIFT (Go Image Filtering Toolkit) paket. Osnovni image paket obezbeđuje funkcije

za rad sa slikama u formatima JPEG, PNG i GIF, a postoje i dodatni paketi za rad sa formatima kao što su BMP, TIFF i drugi.[3] GIFT paket sadrži skup filtera za obradu slika kao što su kontrast, blur, sepia i slični. Osim filtera, paket omogućava i transformacije slika poput promene veličine, crop-a, rotiranja i drugih.[1]

Listing 5.8: Definisanje i primena filtera

```
src , _ = jpeg.Decode(src_file)
filter := gift.New(
    gift.Grayscale(),
    gift.Contrast(10),
)
dst := image.NewRGBA(filter.Bounds(src.Bounds()))
filter.Draw(dst, src)
jpeg.Encode(dst_file, dst, &jpeg.Options{Quality:100})
```

Primer definisanja i primene filtera na JPEG slici je prikazan u listingu 5.8. Promenljiva `filter` je tipa `GIFT` i predstavlja niz filtera `Filter` kojise primenjuju na slici. Funkcijom `Decode` se dekodira JPEG fajl i kao rezultat vraća promenljiva tipa `Image`. `NewRGBA` funkcija kreira praznu sliku iste veličine kao što je i originalna slika. U `image` paketu postoje funkcije za rad i sa drugim modelima boja pored `RGBA`. Funkcijom `Draw` se primenjuju filteri nad originalnom slikom, nakon čega se funkcijom `Encode` filterovana slika kodira u `dst_file`. [1]

Listing 5.9: Primena filtera u funkciji `ImageHandler`

```
img := decode_image(extension, original)
p.rotate(&img)
custom := p.create_custom_filter()
img_paths := apply_filters(&img, custom, dir_path, extension)
```

Nakon svih provera kada je sigurno da se radi sa slikom dozvoljene veličine i formata, prelazi se na obradu slike. Deo funkcije `ImageHandler` koji se odnosi na primenu filtera je prikazan u listingu 5.9. Funkcija `decode_image` dekodira sliku u zavisnosti od formata slike nakon čega metod `rotate` rotira sliku ukoliko je to korisnik izabrao.

Listing 5.10: Funkcija za kreiranje zadanog filtera

```
func (p processor) create_custom_filter() *gift.GIFT{

    selected_custom := p.r.Form["custom"]
    custom := gift.New()
    for _, name := range selected_custom {
        custom.Add(base_filters[name])
    }

    for name := range input_filter_descriptions {
        if val := p.r.FormValue(name); val != "" {
            x, _ := strconv.ParseFloat(val, 32)
            f := input_filters[name](float32(x))
            custom.Add(f)
        }
    }

    return custom
}
```

U funkciji `create_custom_filter`, koja je prikazana u listingu 5.10, kreira se zadati filter na osnovu korisnikovog izbora. Prvo se sakupljaju informacije o svim izabranim ponuđenim filterima koji se na osnovu imena dodaju novom filteru `custom`. Definicije mapa sa predefinisanim filterima su prikazane u listingu 5.12. Nakon toga, zatom filteru se dodaju filteri za izabrane vrednosti karakteristika koje je korisnik uneo. Za svaku karakteristiku postoji input polje u formi za koje je potrebno proveriti da li je korisnik uneo vrednost. Ukoliko korisnik jeste uneo vrednost za odgovarajuće polje, na osnovu imena, poziva se njegova odgovarajuća funkcija sa zatom parametrom. Funkcije u Go-u su validan tip tako da je moguće definisati mapu koja će na osnovu stringa vratiti funkciju, što je i prikazano u listingu 5.12. Funkcija kao rezultat vraća filter koji je definisan pomoću numeričkog parametra `x` koji je korisnik uneo. U slučaju da korisnik nije izabrao ni jedan ponuđen filter i nije uneo nijedan parametar, vraća se prazan filter koji kada se primeni kao rezultat ima originalnu sliku.

Listing 5.11: Funkcija za paralelnu primenu filtera

```
func apply_filters(img *image.Image, custom *gift.GIFT,
    dir_path string, extension string) map[string]string {

    img_paths := make(map[string]string)
    wg := sync.WaitGroup{}
    mutex := &sync.Mutex{}

    for name := range filters {
        wg.Add(1)
        go func(name string){
            tmp := apply_filter(name, nil, img, dir_path, extension)
            mutex.Lock()
            img_paths[name] = tmp
            mutex.Unlock()
            wg.Done()
        }(name)
    }

    tmp := apply_filter("Custom", custom, img, dir_path, extension)

    wg.Wait()
    img_paths["Custom"] = tmp

    return img_paths
}
```

Kada je zadati filter definisan, potrebno je primeniti filtere na slici. U listingu 5.11 je prikazana funkcija `apply_filters` koja primenjuje filtere i kao rezultat vraća mapu sa putanjama do rezultujuće slike za svaki filter. Primena svakog filtera se izvršava u zasebnoj gorutini, konkurentno. U Go-u nije dozvoljeno konkurentno pisanje u mapu pa se u ovom slučaju koristi muteks za kontrolu pristupa mapi. Konkurentno čitanje mape bez pisanja je dozvoljeno i korišćeno je u svim ostalim slučajevima, iz razloga što su mape sa predefinisanim filterima globalne a svaka konekcija se obrađuje u posebnoj gorutini. Napomena da od verzije Go 1.9 postoje mape sa konkurentnim pristupom u okviru `sync` paketa.[4] Nakon kreiranja gorutine za svaki definisani filter, u tekućoj gorutini se primenjuje zadati filter. Tek nakon završetka svih gorutina, upisuje se putanja do slike zadatog filtera kako nebi došlo do data race-a prilikom upisa u mapu.

Listing 5.12: Mape koje se koriste za definisanje različitih vrsta filtera

```
var filters = map[string] *gift.GIFT{
    "Twilight": gift.New(
        gift.Brightness(5),
        gift.Gamma(0.55),
        gift.Colorize(240, 50, 15),
    ),
    ...
}

var base_filters = map[string] gift.Filter{
    "Sunset":      gift.ColorBalance(30, -10, -10),
    ...
}

var input_filters = map[string] func(float32) gift.Filter{
    "Brightness": func(val float32) gift.Filter
                  { return gift.Brightness(val)},
    ...
}
```

Nakon primene filtera, izvršava se template za prikaz rezultata kome se prosleđuje mapa sa putanjama do rezultujuće slike svakog filtera. Pre izlaska iz beskonačne petlje i funkcije `ImageHandler`, pokreće se gorutina za brisanje privremenog direktorijuma koja je prikazana u listingu 5.13. Unutar gorutine, kreira se tajmer koji se aktivira nakon zadatog vremena. Timer funkcioniše tako što je izlaz njegovog kanala blokiran do trenutka isteka zadatog vremena kada se kanal oslobodja.[6] Po isteku vremena, privremeni direktorijum sa svim korisnikovim slikama se briše.

Listing 5.13: Gorutina za brisanje privremenih direktorijuma

```
go func() {
    timer := time.NewTimer(time_available)
    <-timer.C
    os.RemoveAll(dir_path)
}()
```



## Bezbednost

Kada je reč o bezbednosti, aplikacija ne poseduje mnogo tačaka koje bi bile meta zloupotrebe ili napada. Korisnici nemaju naloge i ne ostavljaju osetljive informacije koje mogu biti zloupotrebljene ukoliko se dospe do njih. Jedina meta napada mogu biti slike korisnika koje se trenutno čuvaju na serveru. Kao što je već pomenuto u segmentu 5.1, fajl server ne dopušta korisniku da pristupi samim direktorijumima kako bi pročitao njihov sadržaj. Jedini način da se pristupi slici drugog korisnika jeste nagađanjem kompletne putanje do same slike. Kako deo putanje predstavlja privremeni direktorijum sa pseudoslučajnim stringom dužine 20, možemo biti sigurni sa velikom verovatnoćom da napadač neće nasumičnim nagađanjem doći do slika drugih korisnika.

Korisnik nije u mogućnosti da preoptereti server prevelikim fajlom ali je u mogućnosti da šalje veliki broj zahteva i onemogućiti pristup aplikaciji drugim korisnicima. Generalno, ova vrsta aplikacije koja pruža jednostavnu uslugu, pri čemu korisnici ne ostavljaju osetljive informacije, nije tipična meta napada. Za potrebe razvijanja složenijih aplikacija koje zahtevaju viši nivo bezbednosti, postoje paketi koji pružaju autentikaciju korisnika, rutiranje i dodatne bezbednosne provere.

Glava 6

Zaključak

# Literatura

- [1] *The Go Programming Language, package gift*. on-line at: <https://godoc.org/github.com/disintegration/gift>. 2017.
- [2] *The Go Programming Language, package http*. on-line at: <https://golang.org/pkg/net/http/>. 2017.
- [3] *The Go Programming Language, package image*. on-line at: <https://golang.org/pkg/image/>. 2017.
- [4] *The Go Programming Language, package sync*. on-line at: <https://golang.org/pkg/sync/>. 2017.
- [5] *The Go Programming Language, package template*. on-line at: <https://golang.org/pkg/html/template/>. 2017.
- [6] *The Go Programming Language, package time*. on-line at: <https://golang.org/pkg/time/>. 2017.