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

Glava 1

Uvod

Glava 2

Karakteristike programskog jezika Go

U ovom poglavlju, opisane su karakteristike i osnovni koncepti programiranja jezika Go. Konkurentnost jezika je detaljnije opisana i njoj je posvećeno naredno poglavlje.

2.1 Projekat Go

Programski jezik Go je projekat koji razvija kompanija Google od 2007. godine. Postao je javno dostupan kao projekat otvorenog koda 2009. godine i u stalnom je razvoju. Cilj projekta je bio kreirati novi statički tipiziran jezik koji se kompilira, koji bi omogućavao jednostavno programiranje kao kod interpretiranih, dinamički tipizarnih programskih jezika.

Smatara se da pripada C familiji programskih jezika, ali pozajmljuje i adaptira ideje raznih drugih jezika, izbegavajući karakteristike koje dovode do komplikovanog i nepouzdanog koda. Iako nije objektno-orijentisan, Go podržava određene koncepte kao što su metodi i interfejsi koji pružaju feleksibilnu abstrakciju podataka. Omogućena je efikasna konkurentnost koja je ugrađena u sam jezik, kao i automatsko upravljanje memorijom, odnosno garbage collection [2].

Zbog svojih osobina kao što je konkurentnost, Go je posebno dobar za izradu različitih vrsta serverskih aplikacija, ali je pre svega jezik opšte namene pa se može koristiti za rešavanje svih vrsta problema. Ima primenu u raznim oblastima kao što su grafika, mobilne aplikacije, mašinsko učenje i mnoge druge. Osim kompanije Google, koristi se u velikom broju drugih kompanija širom sveta kao alternativa

Slika 2.1: Neke od kompanija koje koriste programski jezik Go

jezicima poput Python-a i Javascript-a, jer pruža znatno viši stepen efikasnosti i bezbednosti. Neke od većih kompanija koje koriste programski jezik Go, prikazane su na slici 2.1 [1].

2.2 Hello World

Hello World program u jeziku Go, prikazan je u listingu 2.1. Na početku programa navodi se naziv paketa. Svaki paket koji sadrži `main` funkciju, mora nositi naziv `main`. Sa `import` naredbom, navode se nazivi svih paketa koji se koriste u programu. Go ne dozvoljava importovanje suvišnih paketa, već se mogu navesti samo paketi koji se upotrebljavaju u programu. Ukoliko se navede paket koji se ne koristi, biće prijavljena greška tokom kompilacije. Nakon svake naredbe, nije obavezno navođenje ';', osim ako je potrebno navesti više naredbi u istoj liniji.

Listing 2.1: Hello World program u jeziku Go

```
package main

import "fmt"

func main() {
    fmt.Println("Hello World!")
}
```

2.3 Tipovi podataka

Go je statički tipiziran jezik što znači da se promenljivoj 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 prijavljuje se greška prilikom kompilacije. Pravilo koje važi za pakete važi i za promenljive, svaka deklarisana promenljiva mora biti upotrebljena.

Primer prikazan u listingu 2.2 pokazuje definiciju i deklaraciju različitih vrsta promenljivih. Tip promenljive se ne mora ekspilicitno navesti, već se može zaključiti na osnovu dodele operatorom ':=' kada se promenljiva uvodi. Iako nije definisana, promenljiva `a` ima podrazumevanu vrednost koja je za numeričke tipove 0.

Tipovi podataka koji su definisani u programskom jeziku Go, mogu se klasifikovati u četiri kategorije [2]:

1. Bazični tipovi (numerički, bulovski, teksturalni)
2. Složeni tipovi (nizovi i strukture)
3. Referentni tipovi (pokazivači, isečci, mape, kanali, funkcije)
4. Interfejsni tipovi (interfejsi)

Listing 2.2: Primer programa koji ilustruje rad sa promenljivama

```
package main

import "fmt"

func main() {
    var a float32
    b := 5
    var c int

    c = int(a) + b
    fmt.Println(c)    // "5"
}
```

Bazični tipovi podataka

Bazični tipovi koji postoje u programskom jeziku Go, podeljeni su na:

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

Konstante (`const`) u Go-u, predstavljaju izraze čija vrednost je unapred poznata kompilatoru i čija evaluacija se izvršava tokom kompilacije, a ne tokom izvršavanja programa. U pozadaini, svaka konstanta predstavlja jedan od osnovnih tipova i ne može se definisati za neku drugu vrstu podataka.

Paket strings pruža veliki broj funkcija za manipulaciju tipom `string`. Iako u okviru paketa postoji funkcija `Compare`, Go dozvoljava poređenje stringova operatorom '==', kao i ostalim relacionimi operatorima.

Operatori koji su definisani u Go-u, podeljeni su u sledeće kategorije:

- Aritmetički operatori (`+` , `-` , `*` , `/` , `%` , `++` , `- -`)
- Relacioni operatori (`==` , `!=` , `>` , `<` , `>=` , `<=`)
- Logički operatori (`&&` , `||` , `!`)
- Bitski operatori (`&` , `|` , `^` , `>>` , `<<`)
- Operatori dodele (`=` , `+=` , `-=` , `*=` , `/=` , `%=` , `>>=` , `<<=` , `&=` , `|=` , `^=`)

Složeni tipovi podataka

Složenim tipovima podataka pripadaju **nizovi** i **strukture**. **Niz** predstavlja sekvencu elemenata fiksne dužine, određenog tipa podataka. Za razliku od nizova, isečci, koji su opisani u sledećem segmentu, su promenljive dužine. Elementima niza se pristupa standardnom '`[indeks]`' notacijom, gde prvi element ima indeks 0. Ugrađena funkcija `len`, koristi se za dobijanje podatka o dužini niza. Dužina niza se mora navesti prilikom deklaracije ili, ukoliko se izvršava i inicijalizacija, umesto dužine, može se navesti '`...`', a dužina će biti zaključena na osnovu incijalizacije.

Go dozvoljava poređenje nizova iste dužine definisanih nad istim tipom podataka, relacionim operatorima ' `==`' i ' `!=`'. Dva niza su jednaka ako imaju jednake vrednosti na istim pozicijama. U listingu 2.3 je prikazan primer definisanja i upotrebe niza.

Strukture su složeni podaci kuji grupišu nula ili više elemenata proizvoljnog tipa. Svaki element mora biti jedinstveno imenovan i prestavlja jedno polje strukture. Definisanjem strukture se definiše novi tip podataka korišćenjem ključne reči `type` ispred definicije strukture. Poljima strukture se pristupa sa '`.naziv_polja`' notacijom.

Ugnježdene strukture su dozvoljene, odnosno strukture koje sadrže druge strukture kao polja. Strukture istog tipa, mogu se porediti relacionim operatorima ' `==`' i ' `!=`'. Dve strukture su jednake ako imaju jednake vrednosti na istim poljima. Nad strukturama se mogu definisati metodi koji su opisani u segementu 2.5. Primer koji demonstrira rad sa strukturama, prikazan je u listingu 2.3.

Listing 2.3: Primer koji demonstrira rad sa nizovima i strukturama

```
package main

import "fmt"

func main() {
    var a [3]int
    b := [...]int{1, 2, 3}
    a[0] = 1
    fmt.Println(a==b) // "false"

    type Point struct{ X, Y int }
    p1 := Point{1, 2}
    p2 := Point{Y:2}
    p2.X = 1
    fmt.Println(p1==p2) // "true"

    c := new([3]int)
    c[0] = 1
    fmt.Println(a == *c) // "true"
}
```

Prilikom alociranja memorije za nizove i strukture umesto deklarisanja promenljivih, može se koristiti ugrađena funkcija `new` koja vraća pokazivač na alocirani niz ili

strukturu. Nizovi se funkcijama prosleđuju kao kopija, ne preko reference kao što je slučaj u drugim jezicima poput C-a. Prenošenje niza preko reference, mora se eksplicitno naglasiti korišćenjem pokazivača. Pristupanje vrednostima niza unutar funkcije, postiže se korišćenjem operatora '`*`' odnosno notacijom '`(*naziv)[indeks]`'. Ukoliko se struktura prosledi funkciji preko reference, poljima se može pristupiti običnom '`.`' notacijom.

Referentni tipovi podataka

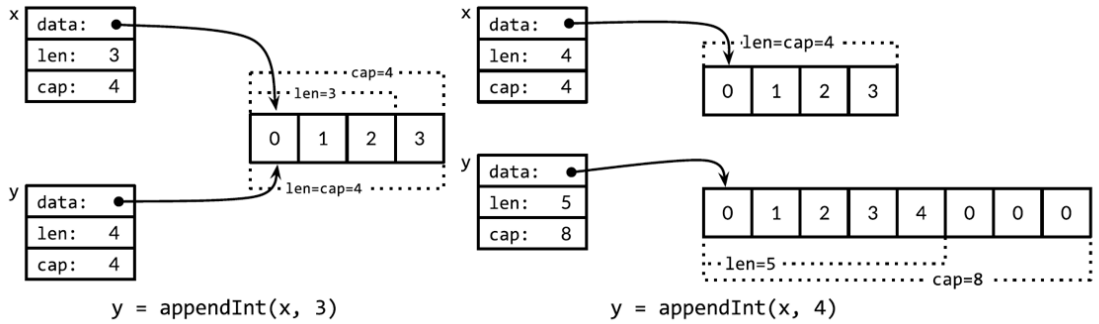
U referentne tipove podataka spadaju **pokazivači**, **isečci**, **mape**, **kanali** i **funkcije**. U ovom segementu, opisani su svi referentni tipovi podataka osim funkcija, kojima je posvećen segment 2.5, i kanala, koji su opisani u narednom poglavlju čija je tema konkurentnost, u segementu 3.3.

Pokazivač referiše na vrednost neke promenljive koja se trenutno čuva u memoriji. Pokazivači kao vrednost čuvaju lokaciju promenljive u memoriji na koju referišu. Za dobijanje lokacije promenljive, koristi se operator '`&`'. Za dereferenciranje pokazivača, odnosno dobijanje vrednosti na koju pokazivač referiše, koristi se operator '`*`'.

Isečak u programskom jeziku Go, predstavlja fleksibilan pogled na elemente niza. Svaki isečak se sastoji od tri podatka: pokazivača na niz, dužine i kapaciteta. Sintaksa za rad sa isečcima je ista kao i za rad sa nizovima, osim što prilikom deklaracije nije potrebno navoditi dužinu isečka. Za alociranje isečaka, koristiti se ugrađena funkcija `make`, koja kao parametre ima tip, dužinu i kapacitet isečka. Funkcija kreira isečak koji sadrži onoliko elemenata kolika mu je zadata dužina, koji su postavljeni na podrazumevane vrednosti. Kapacitet isečka nije fiksan, a za trenuntan podatak o njegovoj vrednosti koristi se ugrađena funkcija `cap`.

Definisanje isečka nad nizom se postiže korišćenjem '`niz[od:do]` notacije pri čemu se gornja granica ne uključuje. Ukoliko se donja granica izostavi, donja granica je počtak niza, ukoliko se gornja granica izostavi, gornja granica je kraj niza, a ukoliko se navede samo `:`, isečak će sadržati sve elemente niza.

Za rad sa isečcima postoji ugrađena funkcija `copy` koja kopira elemente jednog isečka u drugi i kao rezultat vraća broj kopiranih elemenata. Maksimalan broj elemenata koji će biti kopiran, jednak je dužini kraćeg isečka. Dozvoljeno je da isečci pre kopiranja dele elemente. Funkcija `append`, koristi se za dodavanje elemenata



Slika 2.2: Poziv funkcije `append` kada postoji i kada ne postoji dovoljno prostora za dodavanje

isečku. Tokom rada, ukoliko ne postoji dovoljno mesta, funkcija može imati potrebu da realocira niz i zbog toga kao rezultat vraća ažuriranu referencu na isečak. Nakon poziva funkcije `append`, moguće je da isečci koji su delili elemente niza [2], pokazuju na različite nizove, što je i prikazano na slici 2.2. Primer koji ilustruje rad sa isečcima, prikazan je u listingu 2.4

Listing 2.4: Primer koji demonstrira rad sa isečcima

```
package main

import "fmt"

func main() {
    var a [3]int
    b := [...]int{1, 2, 3}
    a[0] = 1
    fmt.Println(a==b) // "false"

    type Point struct{ X, Y int }
    p1 := Point{1, 2}
    p2 := Point{Y:2}
    p2.X = 1
    fmt.Println(p1==p2) // "true"

    c := new([3]int)
    c[0] = 1
    fmt.Println(a == *c) // "true"
}
```

Mapa

2.4 Kontrole toka

2.5 Funkcije i metodi

2.6 Interfejsi

2.7 Refleksija

2.8 Garbage collection

2.9 Upravljanje greškama

2.10 Testiranje

2.11 Paketi

Glava 3

Konkurentno programiranje u jeziku Go

3.1 Osnovni pojmovi konkurentnog programiranja

3.2 Go-rutine

3.3 Kanali

3.4 Sinhronizacija

3.5 Select naredba

3.6 Dizajn pattern-i

3.7 Data race detektor

```
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 Iinterpreter 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. Kodovi svih implementacija su dostupni na repozitorijumu[1].

Go

Implementacija 4.1 koristi koncept višestrukog semafora za ograničavanje broja aktivnih go-rutina. Semafor je realizovan pomoću kanala sa baferom i select naredbe

[1] `https://github.com/MitrovicMilosh/Go-Concurrency/tree/master/quicksort`

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 je moguće dobiti token odnosno uspešno poslati praznu strukturu kroz kanal pokreće se go-rutina 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.

Kako bi se sačekalo da sve go-rutine završile sa svojim radom, za sinhronizaciju se koriste 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.

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()
}
```

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.

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 segmentu 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 konkurentog 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

Tabela 4.1: Prosečno 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	1.58	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

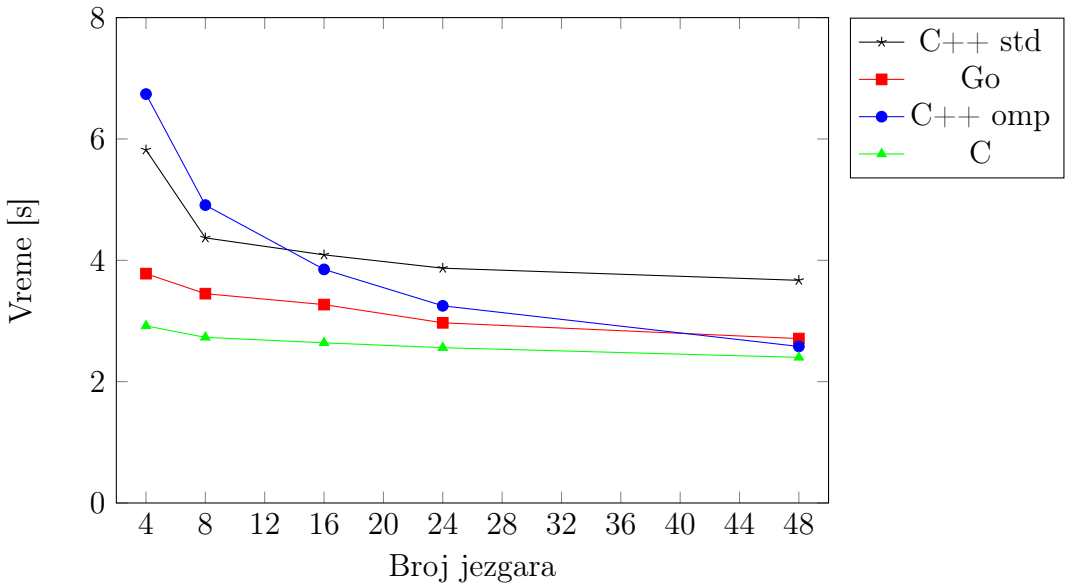
Tabela 4.2: Prosečno vreme izvršavanja [s] quicksort implementacija sa različitim brojem niti, testirano sa 48 jezgara za niz dužine 10 miliona

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

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. 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.



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

n [10^6] / Verzija	1	10	100 0
C++ omp	**6.3**	**42.3**	**393.1**
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	-

Dužine implementacija, prikazane su u tabeli 4.4. Za implementaciju algoritma, u C-u je potrebno najviše linija kôda, 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

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

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. Kodovi svih implementacija su dostupni na repozitorijumu[2].

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 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.

[2] `https://github.com/MitrovicMilosh/Go-Concurrency/tree/master/matrix_multiplication`

Listing 4.2: Go implementacija konkurentne funkcije za množenje matrica

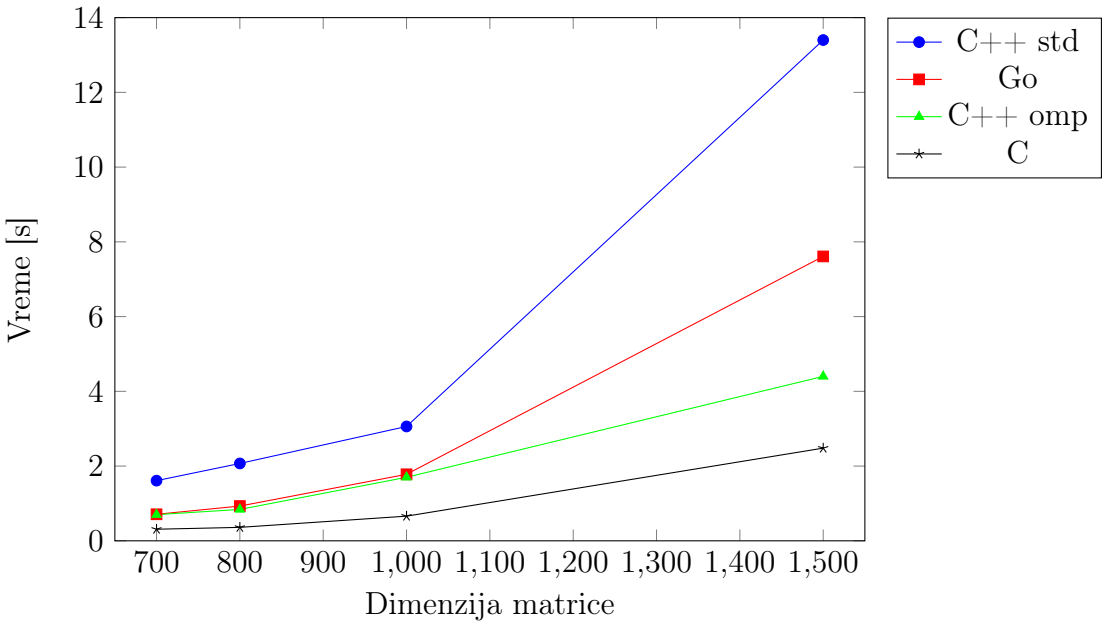
```
func multiply(){
    wg := sync.WaitGroup{}
    wg.Add(n)

    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()
}
```

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 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, potrebno je dva puta više vremena, što je približno četiri puta više nego C-u.



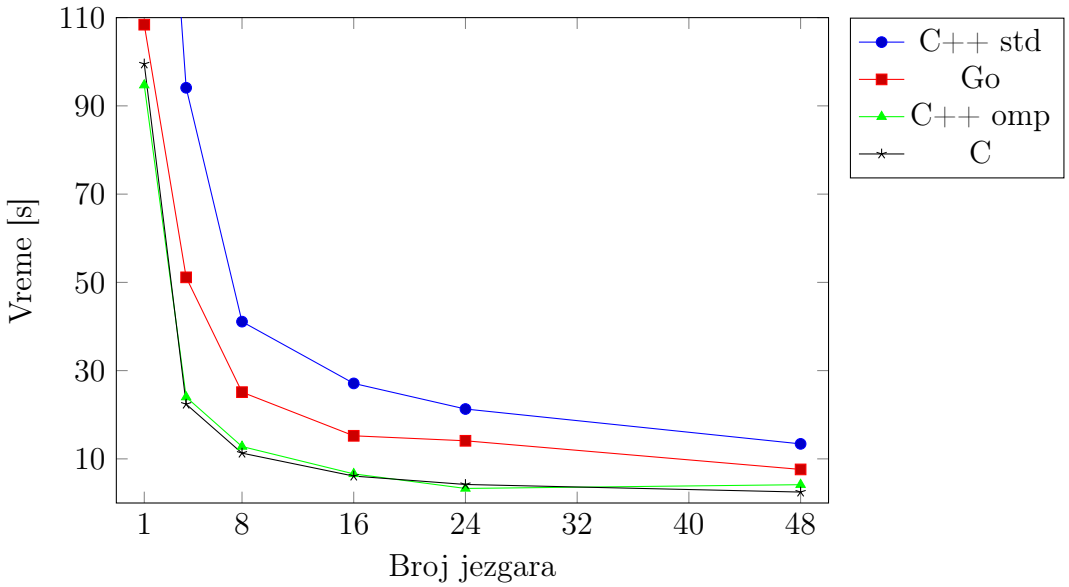
Slika 4.3: Grafik brzine izvršavanja različitih implementacija množenja matrica u zavisnosti od veličine matrice, testirano na 48 jezgara

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	**1.78**	1.82
C	2.74	1.39	0.92	**0.81**
C++ omp	3.03	1.95	**1.70**	1.97
C++ std	15.31	8.71	3.21	**3.06**

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.



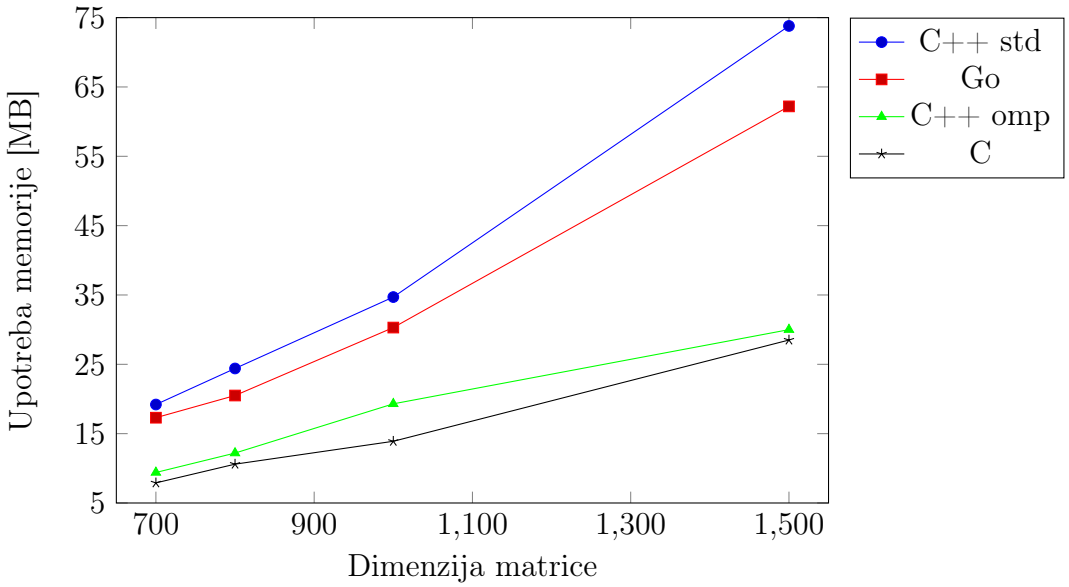
Slika 4.4: Grafik brzine izvršavanja različitih implementacija množenja matrica u zavisnosti od broja jezgara za matricu veličine 1500

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

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 vezija su najefikasnije dok Go koristi dva puta više memorije. C++ verzija sa standardnom bibliotekom zahteva najveću količinu memorije.



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

Tabela 4.7: Dužine kôda implementacija množenja matrica

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

Broj linija kôda svih implementacija, prikazan je u tabeli 4.7. Python ima najkraću 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 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.

```
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

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. Kodovi svih implementacija su dostupni na repozitorijumu[3].

[3] https://github.com/MitrovicMilosh/Go-Concurrency/tree/master/prime_sieve

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.

Ostale implementacije

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

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

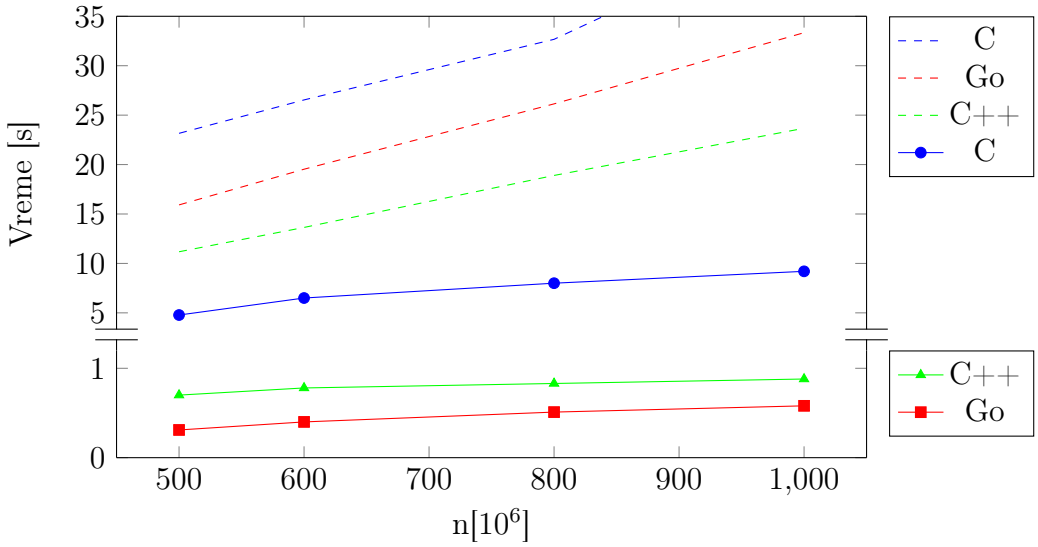
```
func mark_prime(list *[]bool,first,last,sqrt int,wg *sync.WaitGroup){
    for i:=2;  i<= sqrt && i*i<= last; i++{
        if !(*list)[i] {
            var j int
            if i*i < first {
                if (first - i*i)%i == 0 {
                    j = i*i + ((first-i*i)/i)*i
                }else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

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

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 posebnoj tabeli. 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 prikazan 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.



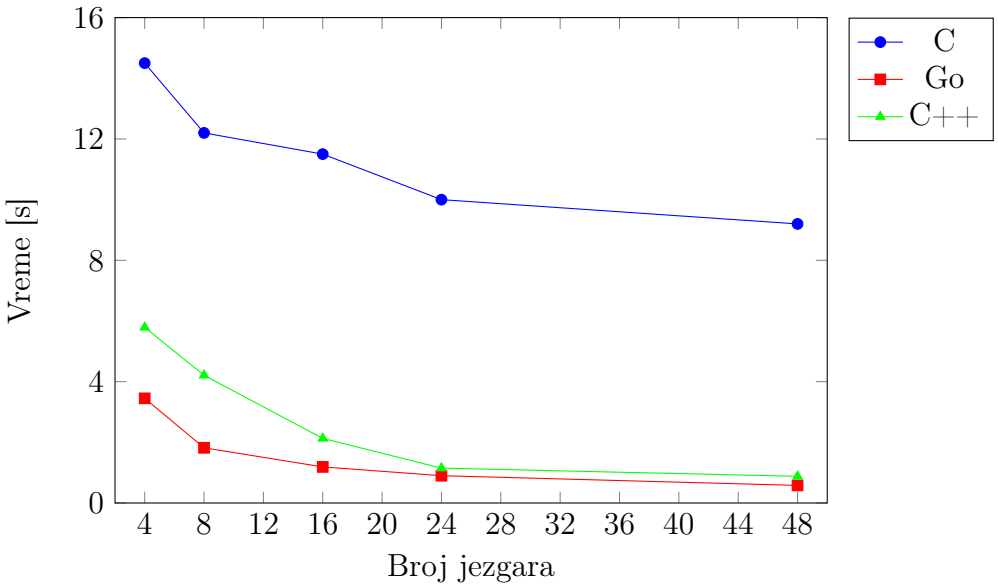
Slika 4.7: Grafik brzine izvršavanja različitih implementacija Eratostenovog sita za različito n, testirano na 48 jezgara; isprekidanom linjom je prikazano sekvencijalno izvršavanje dok je konkurentno prikazano punom linijom

Tabela 4.8: 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	**0.31**	0.49
C	6.17	4.81	**4.74**	7.85
C++	4.53	2.84	**0.73**	0.76

Python je testiran za deset puta manju vrednost n u odnosu na ostale implementacije i rezultati testiranja su prikazani u tabeli 4.9. Python-u je potrebno više od jednog minuta, dok Go za deset puta veću vrednost radi ispod jedne sekunde. Upotreba memorije je četiri 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

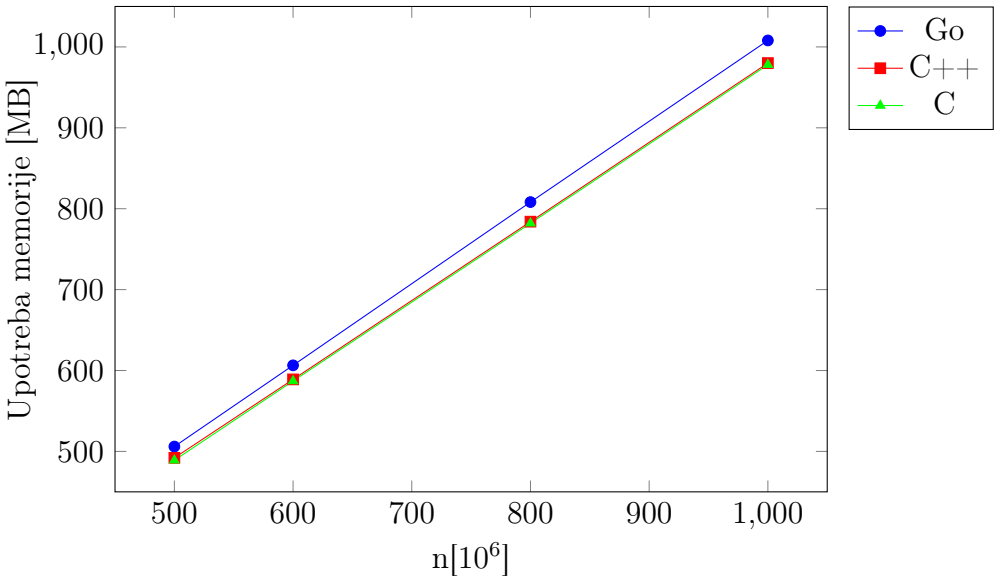
Tabela 4.9: Prosečno vreme izvršavanja i maksimalna upotreba memorije Python implementacije Eratostenovog sita za različito n

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

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.



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

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.

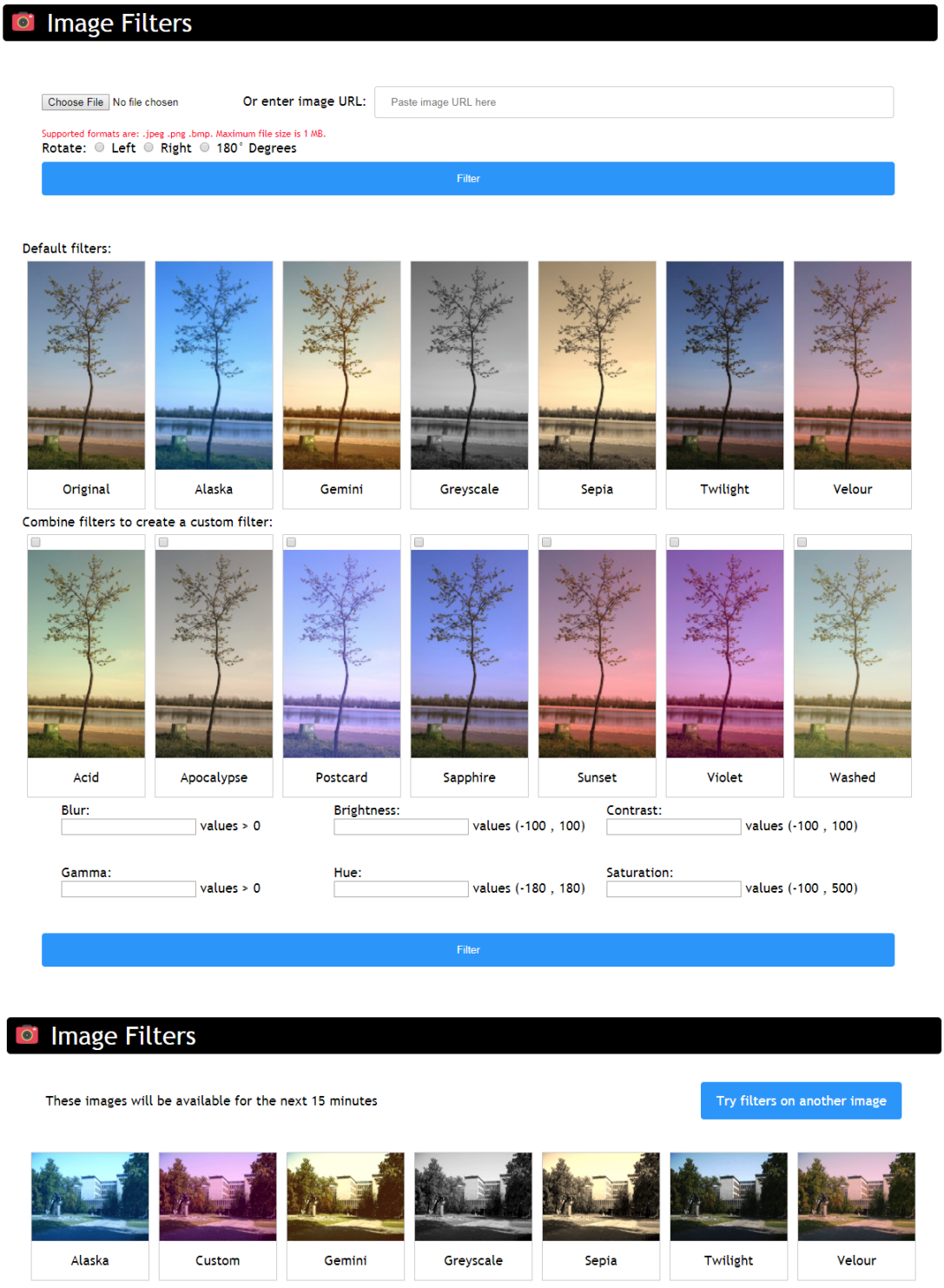
5.1 Serverska aplikacija

Razvijena serverska aplikacija se bavi primenom različitih filtera na slikama. Omogućava primenu postojećih predefinisanih 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. Kompletan kod aplikacije je dostupan na repozitorijumu[1]. Grafički interfejs aplikacije je prikazan na slici 5.1.

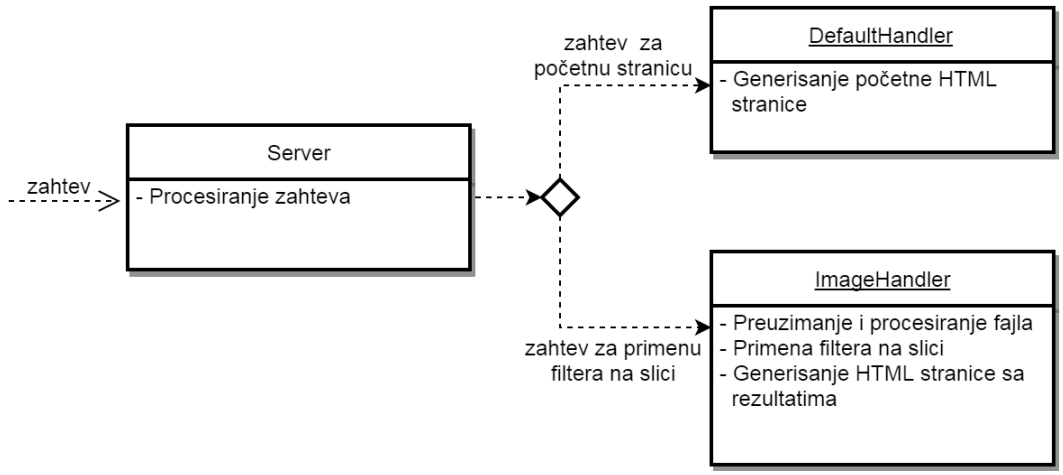
Struktura aplikacije

Struktura aplikacije predtsavljena je dijagramom na slici 5.2. Server, na osnovu zahteva koji dobija od korisnika, poziva odgovarajuću hendler funkciju. Definisane su dve hendler funkcije: `DefuaultHandle`, koja ima zadatak da generiše početnu

[1] `https://github.com/MitrovicMilosh/Go-Concurrency/blob/master/server/server.go`



Slika 5.1: Grafički interfejs početne stranice i stranice za prikaz rezultata serverske aplikacije za primenu filtera



Slika 5.2: Dijagram koji prikazuje strukturu aplikacije

HTML stranicu, i `ImageHandler`, koja se bavi preuzimanjem i procesiranjem fajla, primenom filtera na slici i generisanjem HTML stranice za prikaz rezultata.

Kreiranje servera i procesiranje zahteva

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 5.1.

Listing 5.1: 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 hendler funkcija za određeni šablon. Kao parametre, hendler funkcija ima `ResponseWriter` koji se koristi za konstrukciju HTTP odgovora i `Request` koji sadrži sve podatke HTTP zahteva.

Funkcija `ListenAndServe` 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 hendler funkciju [5].

Server se može implementirati u Go-u koristeći samo dva paketa i manje od deset linija koda što je karakateristično 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 devet bibiloteka i više od pedeset linija koda.[3]

Deo koda koji se odnosi na kreiranje servera u aplikaciji, nalazi se u `main` funkciji koja je prikazana u listingu 5.2. `HandleFunc` funkcijom, registrovane su dve hendler 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.

Listing 5.2: Funkcija `main`, 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)
}
```

Hendler za fajl sistem je prikazan u listingu 5.3. U ovoj funkciji definišu se 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 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 [5]. U tom slučaju korisniku će biti onemogućen pristup direktorijumu i neće moći da pročita njegov sadržaj.

Listing 5.3: Hendler 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"))).ServeHTTP(w, r)
}
```

Generisanje HTML stranice

Paket `htmp/template` koristi se 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, parsira se i izvršava šablon koji se prikazuje kada se pristupi početnoj strani. Deo funkcije koji se odnosi na izvršavanje šablona je prikazan u listingu 5.4. Ovde, kao i na drugim mestima, moguće greške će biti ignorisane jer se podrazumeva da su parametri funkcije ispravni i da neće doći do greške. Fiksni parametri su provereni tokom faze razvijanja i debagovanja aplikacije, dok parametri koji zavise od unosa korisnika, prethodno prolaze kroz odgovarajuće provere. Funkcija `ExecuteTemplate` kao argumente prima `ResponseWriter`, naziv šablona 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 šablonu se sve instrukcije, podaci i kontrole toka navode između dvostrukih vitičarstih zagrada. Primer šablona je prikazan u listingu 5.5. Na početku svakog

šablona se pomoću `define` definiše naziv, a kraj šablona je potrebno naznačiti sa `end`. Omogućeno je iteriranje nad prosleđenim podacima, `if-else` naredbe i izvršavanje šablona unutar šablona. U ovom slučaju, iterira se nad mapom `Filters` koja sadrži nazive filtera i putanje do njihovih slika. Prosleđeni podaci počinju sa znakom '.', dok promenljive počinju sa znakom '$' [8].

Listing 5.5: Izvršavanje HTML šablona

```
{{define "index"}}
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>Image Filters</title>
    </head>
    <body>
        ...
        {{range $name, $image := .Filters}}
        <div class="gallery">
            <img src="{{$image}}" >
            <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, prikazan je 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 demonstrirano je 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 korišćem `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 optereti server. Iz tog razloga se koristi funkcija `MaxBytesReader` koja prekida konekciju sa klijentom ukoliko je pređena dozvoljena veličina fajla [5]. 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 funcijom `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 provera 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[4] koji ne ulazi u originalnu Go distribuciju.

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[6]. `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 [4].

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 slika 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 orginalnom slikom, nakon čega se funkcijom `Encode` filterovana slika kodira u `dst_file` [4].

Listing 5.8: Definisanje i primena filtera

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

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, prikazan je u listingu 5.9. Funkcija `decode_image` dekodira funkciju u zavsinosti od formata slike nakon čega metod `rotate` rotira sliku ukoliko je to korisnik izabrao.

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)
```

Listing 5.10: Funkcija za kreiranje zadatog 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
}
```

Listing 5.11: Mape koje se koriste za definisanje različitih vrsta filtera

```
    var filters = map[string] *gift.GIFT{
        "Sepia": gift.New(
            gift.Sepia(100),
            gift.Contrast(10),
        ),
        ...
    }
    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)},
        ...
    }
```

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 5.11. Nakon toga, zadatom filteru, dodaju se 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 zadatim 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 5.11. Funkcija kao rezultat vraća filter koji je definisan pomoću numeričkog parametra `x` koji je korisnik uneo. U slučaju da korinsik 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 5.12: 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 5.12 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 (mape sa predefinisanim filterima, definisane su kao globalne, a svaka konekcija se obrađuje u zasebnoj gorutini). Napomena da od verzije Go 1.9 postoje mape sa konkurentnim pristupom u okviru `sync` paketa [7]. 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 ne bi došlo do data race-a prilikom upisa u mapu.

Nakon primene filtera, izvršava se šablon 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 priveremnog direktorijuma koja je prikazana u listingu 5.13. Unutar gorutine, kreira se tajmer koji se aktivira nakon zadatog vremena. Tajmer funkcioniše tako što je izlaz njegovog kanala blokiran do trenutka isteka zadatog vremena kada se kanal oslobođa [9]. Po isteku vremena, briše se privremeni direktorijum sa svim korisnikovim slikama.

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 komletne 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 onemogući 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] *Companies currently using Go throughout the world*. on-line at:`https://github.com/golang/go/wiki/GoUsers`. 2017.

[2] Alan A.A. Donovan i Brian W. Kernighan. *The Go Programming Language*. 1st. Addison-Wesley Professional, 2015. ISBN: 0134190440, 978-0134190440.

[3] *Rosetta Code, Hello World web server*. on-line at:`https://rosettacode.org/wiki/Hello_world/Web_server`. 2017.

[4] *The Go Programming Language, package gift*. on-line at: `https://godoc.org/github.com/disintegration/gift`. 2017.

[5] *The Go Programming Language, package http*. on-line at: `https://golang.org/pkg/net/http/`. 2017.

[6] *The Go Programming Language, package image*. on-line at: `https://golang.org/pkg/image/`. 2017.

[7] *The Go Programming Language, package sync*. on-line at: `https://golang.org/pkg/sync/`. 2017.

[8] *The Go Programming Language, package template*. on-line at: `https://golang.org/pkg/html/template/`. 2017.

[9] *The Go Programming Language, package time*. on-line at: `https://golang.org/pkg/time/`. 2017.