UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Miloš Mitrović

KONKURENTNOST U PROGRAMSKOM JEZIKU GO

master rad

Beograd, 2018.

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: Situacija u računarstvu u današnje vreme višejezgarnih procesora, mrežnih sistema i velikih serverskih projekata koji se mogu sastojati od nekoliko desetina miliona linija koda, značajno se razlikuje od sredine u kojoj su kreirani jezici kao što su C, C++ i Python. Programski jezik Go je upravo dizajniran sa ciljem postizanja maksimalne produktivnosti u ovakvoj sredini, na takav način da programeru maksimalno olakša sam proces razvoja softvera.

Go je statički tipizirani programski jezik koji se kompilira, sa jednostavnom sintaksom koja omogućava razvoj kvalitetnog softvera sa malim brojem linija koda. Jedna od glavnih osobina koje Go čini posebnim je njegova konkurentnost koja je ugrađena u sam jezik i na intuitivan način obezbeđuje programeru pisanje memorijski i vremenski efikasnih programa sa veoma čitljivim kodom.

Rad opisuje karakteristike jezika Go, posebno razmatra njegovu konkurentnost i daje rezultate uporednih testova konkurentnog izvršavanja implementacija izabranih algoritima u jezicima Go, C/C++ i Python. Go predstavlja jezik opšte namene, ali zbog svoje dobro realizovane konkurentnosti, pogodan je za razvoj serverskih aplikacija, pa je iz tog razloga kao primer upotrebe razvijena serverska aplikacija koja demonstrira način na koji se konkurentnost može iskoristiti, kao i pojedine druge aspekte programskog jezika.

Ključne reči: programski jezik Go, konkurentno programiranje, serverska aplikacija

Sadržaj

Glava 1

Uvod

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, a koji bi omogućavao jednostavno programiranje kao kod interpretiranih, dinamički tipizarnih programskih jezika.

Smatra se da Go 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 fleksibilnu apstrakciju podataka. Go omogućava efikasnu konkurentnost koja je ugrađena u sam jezik, kao i automatsko upravljanje memorijom, odnosno sakupljanje smeća.

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 još mnoge druge. Osim kompanije Google, koristi se u velikom broju drugih kompanija kao alternativa jezicima poput Python-a i Javascript-a, jer pruža znatno viši stepen efikasnosti i bezbednosti.

U radu su istražene i opisane karakteristike programskog jezika Go (glava 2), sa akcentom na njegove mogućnosti za konkurentno programiranje (glava 3). Implementirana su konkurentna rešenja izabranih algoritama u programskim jezicima Go, C/C++, Python, i urađena je uporedna analiza vremenskih i memorijskih zahteva u zavisnosti od dimenzije problema i broja jezgara na kojima se softver izvršava (glava 4). Kao primer upotrebe, razvijena je serverska aplikacija koja demonstrira korišćenje konkurentnosti kao i drugih aspekata jezika Go (glava 5).

Glava 2

Karakteristike programskog jezika Go

U ovoj glavi opisane su karakteristike i osnovni koncepti programiranja jezika Go. Konkurentnost jezika je detaljnije opisana i njoj je posvećena glava 3.

2.1 Go alat

Alat `go` predstavalja osnovni alat za upravljanje Go kodovima. Neke od definisanih komandi u okviru alata su: `build` za kompilaciju paketa, `run` za kompilaciju i izvršavanje Go programa, `test` za testiranje paketa, `get` za preuzimanje i instaliranje paketa, `fmt` za formatiranje koda paketa. Alat se upotrebljava `go komanda [argumenti]` notacijom.

2.2 Početni primer

Početni program *Hello World* u jeziku Go prikazan je u primeru 2.1. Na početku programa navodi se naziv paketa. Svaki paket koji sadrži `main` funkciju mora nositi naziv `main`. Sa naredbom `import` navode se nazivi svih paketa koji se koriste u programu. Go ne dozvoljava uključivanje 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 znaka `;` , osim ako je potrebno navesti više naredbi u istoj liniji. Komentari se navode na isti način kao u jeziku C.

Primer 2.1: Program *Hello World* u jeziku Go

```
package main

import "fmt"

func main() {
    fmt.Println("Hello World!") // ispisuje 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 vrednosti već se konverzija vrednosti 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.

Program prikazan u primeru 2.2 prikazuje definiciju i deklaraciju različitih vrsta promenljivih. Tip promenljive se ne mora eksplicitno 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.

Primer 2.2: Program 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
}
```

Tipovi podataka koji su definisani u programskom jeziku Go mogu se klasifikovati u četiri kategorije [3]:

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

Konstante (`const`) u Go-u predstavljaju izraze čija je vrednost unapred poznata kompilatoru i čija se evaluacija izvršava tokom kompilacije, a ne tokom izvršavanja programa. Konstante se mogu definisati samo za osnovne tipove podataka.

Osnovni tipovi podataka

Osnovni 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`)

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

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

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

Upotreba nad određenim tipovima podataka, uloga i arnost Go operatora, definisani su na isti način kao u jeziku C.

Složeni tipovi podataka

U složene tipove podatka u programskom jeziku Go spadaju **nizovi** i **strukture**.

Niz predstavlja sekvencu elemenata istog tipa, fiksne dužine. Elementima niza se pristupa standardnom `[indeks]` notacijom gde prvi element ima indeks 0. Ugrađena funkcija `len(niz)` 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 znak `...` , 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. Primer 2.3 prikazuje definisanje i upotrebu niza.

Strukture su složeni podaci koji grupišu nula ili više elemenata ne obavezno istih tipova. Svaki element mora biti jedinstveno imenovan i predstavlja jedno polje strukture. Definisanjem strukture definiše se novi tip podataka korišćenjem ključne reči `type` ispred same definicije strukture. Poljima strukture pristupa se pomoću znaka `.` i naziva polja. Prazne strukture `struct{}` mogu se koristiti za komunikaciju preko kanala u slučajevima kada nije bitan sadržaj poruke već samo davanje signala. U implementaciji serverske aplikacije u glavi 5 u potpoglavlju 5.1 prikazan je primer upotrebe praznih struktura za realizaciju semafora.

Ugnježdavanje struktura je dozvoljeno, odnosno definisanje struktura 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 poglavlju 2.6. Primer 2.3 demonstrira rad sa strukturama.

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. U tom slučaju, pristupanje vrednostima niza unutar funkcije postiže se korišćenjem operatora `*` odnosno notacijom `(*naziv)[indeks]`. Ukoliko se funkciji prosledi struktura preko reference, poljima se može pristupiti uobičajenom notacijom upotrebom znaka `.` i naziva polja.

Primer 2.3: Rad sa nizovima i strukturama

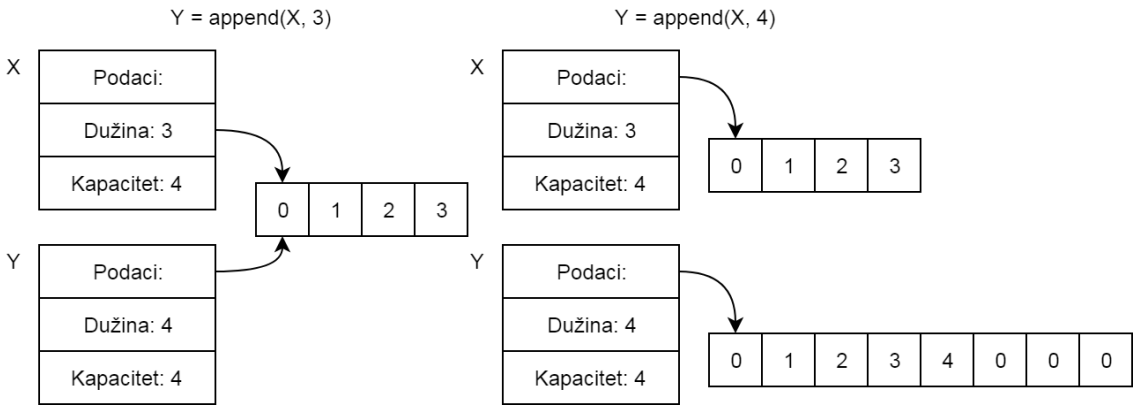
```
var a [3]int
b := [...]int{1, 2, 3}
a[0] = 1
fmt.Println(a == b) // false

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
```

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 posvećeno potpoglavlje 2.5, i kanala, koji su opisani u glavi 3 čija je tema konkurentnost, u poglavlju 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 (engl. „slice") u programskom jeziku Go, predstavlja pogled na elemente niza. Za razliku od nizova, isečci su promenljive dužine. 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([]tip, dužina, kapacitet)`. Funkcija kreira isečak koji sadrži onoliko elemenata kolika mu je zadata dužina, koji su postavljeni na podrazumevane vrednosti. Kapacitet isečka nije fiksan, a za podatak o njegovoj trenutnoj vrednosti koristi se ugrađena



Slika 2.1: Poziv funkcije `append` kada postoji i kada ne postoji dovoljno prostora za dodavanje

funkcija `cap(isečak)`.

Primer 2.4: Rad sa isečcima

```
a := [...]int{1, 2, 3, 4, 5}

s1 := a[:3]
fmt.Println(s1, cap(s1))      // [1 2 3] 5

s1 = append(s1, 6)            // [1 2 3 6]
a[0] = 10                     // deluje i na s1
s2 := make([]int, 4, 4)       // [0 0 0 0]
copy(s2, s1)
s1[1] = 11                    // ne deluje na s2

fmt.Println(s1, cap(s1))      // [10 11 3 6] 5
fmt.Println(s2, cap(s2))      // [10 2 3 6] 4

s1 = append(s1, 7, 8)         // realocira s1
a[2] = 12                     // ne deluje na s1
fmt.Println(s1, cap(s1))      // [10 11 3 6 7 8] 12
```

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

Za rad sa isečcima postoji ugrađena funkcija `copy(destinacija, izvor)` 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(isečak, elementi)` koristi se za dodavanje elemenata 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, pokazuju na različite nizove, što je i prikazano na slici 2.1 [3]. Primer 2.4 ilustruje rad sa isečcima.

Mapa je referentni tip podataka koji referiše na heš tabelu. U tabeli se čuvaju parovi ključ - vrednost, pri čemu je vrednost svakog ključa jedinstvena. Svi ključevi su istog tipa i sve vrednosti su istog tipa. Jedini uslov je da se tip ključa može porediti operatorom `==` , dok vrednosti mogu biti proizvoljnog tipa, uključujući i druge mape.

Funkcija `make(map[tip_ključa] tip_vrednosti)` se koristi prilikom alociranja prostora za novu mapu. Upis u mapu postiže se `naziv[ključ] = vrednost` notacijom. Ugrađena funkcija `delete(mapa, ključ)` koristi se za brisanje parova iz mape. Funkcija kao parametre ima naziv mape i ključ para koji se briše. Ukoliko ključ ne postoji, funkcija nema nikakav efekat. Dodeljivanje vrednosti promenljivoj na osnovu ključa iz mape postiže se `v := naziv[ključ]` notacijom, pri čemu ukoliko ključ ne postoji, vrednost `v` će biti postavljena na podrazumevanu. Moguće je izvršiti proveru da li je određeni ključ definisan u mapi upotrebom `v, ok := naziv[ključ]` ili `_, ok := naziv[ključ]` notacije, gde je `ok` promenljiva tipa `bool` koja nosi informaciju da li ključ postoji ili ne.

Primer 2.5: Rad sa mapama

```
m1 := make(map[string]int)
m1["a"] = 1
m2 := map[string]float32{"m": 1.23, "a": 2.34}

_, ok := m2["b"]
delete(m2, "p")       // nema efekta
delete(m2, "a")
fmt.Println(m2, ok) // [m:1.23] false
```

U programskom jeziku Go, operacije nad mapama nisu atomične i prilikom konkurentnog korišćenja potrebno je koristiti odgovarajuće mere sigurnosti koje su opisane u glavi 3. Primer 2.5 demonstrira rad sa mapama.

2.4 Naredbe za kontrolu toka

U programskom jeziku Go, naredbe za kontrolu toka izvršavanja su `if`, `switch`, `for` i `defer`. Sintaksa naredbe `if` prikazana je u primeru 2.6. U naredbi `if` pre uslova moguće je navesti jednu naredbu koja je razdvojena znakom `;` od uslova kao kod `for` petlje.

Primer 2.6: Sintaksa naredbe `if`

```
if inicijalizacija; uslov1 {
    // naredbe
} else if uslov2 {
    // naredbe
} else {
    // naredbe
}
```

Naredba `switch` omogućava ispitivanje vrednosti izraza po slučajevima i njena sintaksa je prikazana u primeru 2.7. Ukoliko je vrednost izraza jednaka nekom od slučajeva, izvršiće se deo koda naveden nakon znaka `:` , do definicije sledećeg slučaja ili kraja naredbe. U suprotnom izvršava se slučaj `default`, ako je naveden. Ukoliko se na kraju slučaja navede `fallthrough` naredba, izvršava se i kôd slučaja direktno ispod. Za pojedinačni slučaj moguće je navoditi više vrednosti razdvojenih zarezima.

Primer 2.7: Sintaksa naredbe `switch`

```
switch izraz {
case vrednost1:
    // naredbe
case vrednost2, vrednost3:
    // naredbe
default:
    // naredbe
}
```

Jedina petlja koja postoji u programskom jeziku Go jeste `for` i njena sintaksa je prikazana u primeru 2.8. Za izlazak iz petlje koristi se naredba `break`, a za prelazak na sledeću iteraciju naredba `continue`. Petlja `for` se može koristiti zajedno sa naredbom `range` za iteriranje kroz sekvencijalne strukture kao što su nizovi, isečci i mape.

Primer 2.8: Sintaksa `for` petlje

```
for inicijalizacija; uslov; inkrementacija; {
    // naredbe
}

for indeks, vrednost := range sekvencijalna_struktura {
    // naredbe
}
```

Naredba `defer` odlaže izvršavanje navedene funkcije do trenutka završetka funkcije u kojoj se nalazi. Argumenti funkcije se određuju u trenutku navođenja naredbe `defer`, samo je izvršavanje funkcije odloženo. Naredba garantuje da će se funkcija izvršiti i u slučaju dolaska do greške.

Primer 2.9: Primer koji demonstrira upotrebu kontrolnih struktura

```
a := [...]uint{1, 2, 3}
if len(a) > 1 {
    for i,v := range a {
        fmt.Println(i,v) // indeks, vrednost
    }
}

switch a[2] {
case 3:
    fmt.Println("bigger than two")
    fallthrough
case 2:
    fmt.Println("bigger than one")
default:
    fmt.Println("one")
}
```

Najčešća upotreba naredbe `defer` je za operacije koje se obavljaju u paru, kao što je otvaranje i zatvaranje datoteka. Odmah nakon otvaranja datoteke moguće je naredbom `defer` pozvati funkciju za zatvaranje datoteke, čime se garantuje da će se datoteka zatvoriti i doprinosi čitljivijem kodu.

Naredbe `if` i `for` zahtevaju upotrebu vitičastih zagrada i u slučajevima kada se u bloku nalazi samo jedna naredba, a uslove nije potrebno navoditi unutar zagrada. Primer 2.9 ilustruje upotrebu standardnih kontrolnih struktura.

Osim navedenih kontrolnih struktura Go podržava i `go to` naredbe u kombinaciji sa labelama. Generalno, upotreba `go to` naredbi nije preporučena jer dovodi do loše strukture i nečitljivog koda.

2.5 Funkcije

Funkcije u programskom jeziku Go predstavljaju referentni tip podataka. Sintaksa za definisanje funkcije prikazana je u primeru 2.10. Lista parametara se navodi u formatu `naziv tip`, a ukoliko postoji više parametara istog tipa za redom dovoljno je navesti njihov tip samo jedanput. Ukoliko je lista rezultata jednočlana, nije potrebno navoditi zagrade.

Primer 2.10: Sintaksa za definisanje funkcije

```
func naziv(lista parametara) (lista rezultata) {
    telo funkcije
}
```

Osim što mogu imati proizvoljan broj argumenata, funkcije mogu imati više povratnih vrednosti. Postoje i *varijadičke funkcije*, odnosno funkcije sa promenljivim brojem argumenata. Varijadičke funkcije definišu se upotrebom znaka `...` ispred tipa argumenta koji se može pojaviti nula ili više puta.

Omogućeno je definisanje funkcija unutar funkcija gde unutrašnja funkcija ima pristup svim promenljivama spoljašnje funkcije, kao i definisanje *anonimnih funkcija*, odnosno funkcija bez imena koje se izvršavaju na mestu gde su definisane. Funkcije predstavljaju tipove prvog reda i mogu se prosleđivati drugim funkcijama i dodeljivati promenljivama. Primer 2.11 demonstrira rad sa funkcijama.

Primer 2.11: Rad sa funkcijama

```
func add(args ...int) int {
    sum := 0
    for _, v := range args {
        sum += v
    }
    return sum
}
func calc(x, y int) (int, int) {
    add := func() int { return x + y }
    sub := func() int { return x - y }

    return add(), sub()
}
func power(x int) func(int)int {
    res := 1
    return func(a int)int {
        for i:=0; i<a; i++ {
            res *= x
        }
        return res
    }
}

func main() {
    fmt.Println(add(1,2,3))      // 6
    fmt.Println(calc(1,2))       // 3, -1
    fmt.Println(power(2)(4))     //16
}
```

2.6 Metodi

U programskom jeziku Go ne postoji pojam klase, ali je omogućeno definisanje metoda nad korisnički definisanim tipovima. Metod predstavlja običnu funkciju koja u sebi sadrži specijalni *prijemnik* (engl. „receiver") za tip podataka nad kojim je metod definisan. Metod se definiše na isti način kao i funkcija, osim što se prijemnik navodi između ključne reči `func` i naziva metoda.

Pozivanje metoda postiže se navođenjem znaka `.` i naziva metoda nakon promenljive za koju je potrebno pozvati metod. Ukoliko metod menja podatke promenljive

nad kojom je pozvan, potrebno je koristiti prijemnik sa referencom, odnosno navesti znak * ispred tipa prijemnika.

Go nema definisane nivoe vidljivosti pomoću ključnih reči kao što su `public`, `private` i `protected`. Jedini mehanizam za kontrolu vidljivosti je sledeći: svi identifikatori koji počinju velikom slovom, biće eksportovani van paketa, svi identifikatori koji počinju malim slovom, neće biti eksportovani van paketa [3]. Isto pravilo važi i za polja strukture. Na taj način struktrure nad kojima su definisani metodi mogu da oponašaju klase i enkapsulaciju podataka. Primer 2.12 demonstrira rad sa metodima.

Primer 2.12: Rad sa metodima

```
type Point struct{ X, Y float64 }

func (p Point) Distance(q Point) float64 {
    return math.Sqrt((p.X-q.X)*(p.X-q.X)+(p.Y-q.Y)*(p.Y-q.Y))
}

func (p *Point) Translate(x, y float64) {
    p.X += x
    p.Y += y
}

func main() {
    p1 := Point{0,0}
    p2 := Point{1,1}

    fmt.Println(p1.Distance(p2)) // 1.41421...
    p2.Translate(5,2)
    fmt.Println(p2)              // {6 3}
}
```

2.7 Interfejsi

Interfejsi se koriste kao dodatno sredstvo apstrakcije i enkapsulacije podataka. Tip interfejs definisan je kao skup potpisa metoda. Svaki interfejs se sastoji od vrednosti i konkretnog tipa. Vrednost tipa interfejs može biti bilo koja vrednost čiji tip implentira zadate metode. Konkretni tipovi su svi tipovi podataka osim interfejsa.

Tip zadovoljava neki interfejs ukoliko implementira sve metode koje taj interfejs zahteva. Unutar interfejsa moguće je navoditi i nazive drugih interfejsa koji takođe moraju biti zadovoljeni. Svi tipovi zadovoljavaju prazan interfejs (`interface{}`). Mnogi paketi i funkcije koriste interfejse kao što je već viđeni paket `fmt` sa interfejsom `Stringer` koji se koristi za definisanje ispisa određenog tipa podataka. Greške, koje su opisane poglavlju 2.9, takođe predstavljaju interfejs. Primer 2.13 demonstrira rad sa interfejsima.

Primer 2.13: Rad sa interfejsima

```
type geometry interface {
    area() float64
}
type rect struct {width, height float64}
type circle struct {radius float64}

func (r rect) area() float64 {
    return r.width * r.height
}
func (c circle) area() float64 {
    return math.Pi * c.radius * c.radius
}
func measure(g geometry) {
    fmt.Println(g, g.area())
}

func main(){
    r := rect{width: 3, height: 4}
    c := circle{radius: 5}
    measure(r)  //{3 4} 12
    measure(c)  //{5} 78.5398
}
```

Pretpostavljanje tipova (engl. „type assertion") omogućava pristup konkretnom tipu u okviru interfejsa što se postiže sa `i.(T)`, gde je `i` promenljiva tipa interfejs, a `T` naziv konkretnog tipa. Ukoliko se pretpostavi pogrešan tip dolazi do greške. Za proveru da li interfejs sadrži određeni tip prilikom pretpostavljanja mogu se vratiti dve vrednosti, vrednost tipa i vrednost `bool` koja sadrži informaciju da li je pretpostavljanje uspešno. Za obradu pojedinačnih slučajeva za svaki od mogućih

tipova može se koristiti naredba `switch`. Primer 2.14 demonstrira pretpostavljanje tipova.

Primer 2.14: Pretpostavljanje tipova kod interfejsa

```
var i interface{} = "hello"

s := i.(string)
fmt.Println(s)      // hello

s, ok := i.(string)
fmt.Println(s, ok)  // hello true

f, ok := i.(float64)
fmt.Println(f, ok)  // 0 false
```

2.8 Refleksija

U računarstvu, refleksija predstavlja sposobnost programa da izvrši ispitivanje svoje strukture i ponašanja u toku izvršavanja. Refleksija se može koristiti za ispitivanje tipova promenljivih tokom izvršavanja u zavisnosti od čega program može da preduzme odgovarajuće akcije. Go podržava refleksiju tipova koja je omogućena upotrebom paketa `reflect`.

Unutar paketa, definisani su tipovi `Type` i `Value`, funkcija `TypeOf(promenljiva)` za dobijanje informacije o tipu promenljive i funkcija `ValueOf(promenljiva)` za dobijanje vrednosti promenljive. Funkcije kao argument prihvataju tip `interface{}`.

Za postavljanje vrednosti postoje metodi `Set(vrednost)` za svaki tip podataka koji se poziva nad tipom `Value`, kao i metod `CanSet()` za proveru da li je moguće postaviti vrednost datoj promenljivoj. Vrednost je moguće postaviti samo referentnim tipovima, odnosno promeniti vrednost na koju oni referišu. Metodi `Set` se ne mogu pozivati nad promenljivom `Value` bez pripreme, već je nad njom prvo potrebno pozvati metod `Elem()` koji omogućava pristup vrednosti na koju promenljiva referiše.

Primer 2.15 demonstrira osnovnu upotrebu refleksije. Paket nudi još veliki broj drugih funkcija za refleksiju tipova i njihovu manipulaciju [14].

Refleksija predstavlja jak alat ukoliko se upotrebljava na pravi način, ali trebalo bi je koristiti sa oprezom. Postoji nekoliko razloga za opreznost: mogućnost dolaska

do greške u toku izvršavanja programa koja bi mogla biti prijavljena tokom kompilacije ukoliko program ne bi koristio refleksiju; smanjena čitljivost koda ukoliko program previše koristi refleksiju; sporije izvršavanje funkcija koje koriste refleksiju naspram funkcija specijalizovanih za pojedinačne tipove [3].

Primer 2.15: Osnovna upotreba refleksije

```
var x float64 = 3.14
v := reflect.ValueOf(&x)

fmt.Println(v.Type())    // *float64
fmt.Println(v.CanSet()) // false

v = v.Elem()
fmt.Println(v.CanSet()) // true

v.SetFloat(3.15)
fmt.Println(x)           // 3.15
```

2.9 Upravljanje greškama

Stanje greške u programskom jeziku Go označava se vrednostima `error`. Tip `error` predstavlja ugrađeni interfejs koji zahteva implementiranje metoda `Error()` koji kao rezultat vraća `string` sa informacijom o grešci. Ako neka funkcija kao rezultat vraća tip `error`, njegova vrednost će biti `nil` ukoliko nije došlo do greške.

Implementacijom metoda `Error` za neki tip moguće je definisati novi tip greške. Ukoliko se upotrebljava već postojeći tip `error`, za definisanje poruke greške koristi se funkcija `New` iz paketa `errors`. Primer 2.16 demonstrira upravljanje greškama.

Na ovaj način omogućeno je elegantno upravljanje greškama. Za prekidanje toka izvršavanja u slučaju ozbiljnijih grešaka i nepredviđenih situacija koristi se ugrađena funkcija `panic(string)`. Ukoliko funkcija `F` pozove funkciju `panic` njeno izvršavanje se prekida, a zatim se izvršavaju `defer` pozivi funkcija nakon čega se funkcija `F` vraća svom pozivaocu. Tada se za pozivaoca funkcija `F` ponaša kao poziv funkcije `panic` i proces se ponavlja dokle god sve funkcije u tekućoj gorutini ne prestanu sa radom i dok sam program ne prekine sa izvršavanjem. Gorutine su osnovne jedinice izvršavanja u Go-u i predstavljaju niti niže kategorije. Gorutinama je posvećeno poglavlje 3.2 u glavi 3 koja se bavi konkurentnim programiranjem. Ugrađene funkcije obično pozivaju funkciju `panic` ukoliko dođe do neke nepredviđene greške ili

situacije koja se nije mogla otkriti tokom faze kompilacije, ali moguće je pozvati funkciju i manuelno.

Primer 2.16: Upravljanje greškama

```
//definisanje novog tipa
type MyError struct {
    When time.Time
    What string
}
func (e MyError) Error() string {
    return fmt.Sprintf("%v: %v", e.When, e.What)
}
//funkcija koja koristi novi tip
func divide1(a,b float64) (float64, error) {
    if b == 0 {
        return 0, &MyError{time.Now(),"division by zero"}
    }else {
        return a/b, nil
    }
}
//funkcija koja koristi error tip
func divide2(a,b float64) (float64, error) {
    if b == 0 {
        return 0, errors.New("Division by zero")
    }else {
        return a/b, nil
    }
}
//provera greske u funkciji main
func main() {
    if v1, err := divide1(3.14,0); err != nil {
        fmt.Println(err)
    }
    if v2, err := divide2(3.14,0); err != nil {
        fmt.Println(err)
    }
}
```

Pored funkcije `panic` postoji i ugrađena funkcija `recover()` koji služi za preuzimanje kontrole gorutine koje je u stanju `panic`. Funkciju `recover` ima jedino smisla koristiti unutar naredbe `defer`. Ukoliko je tekuća gorutina u stanju `panic`,

`recover` funkcija će preuzeti vrednost koju je dobila funkcija `panic` i nastaviti normalno izvršavanje programa.

2.10 Sakupljanje smeća

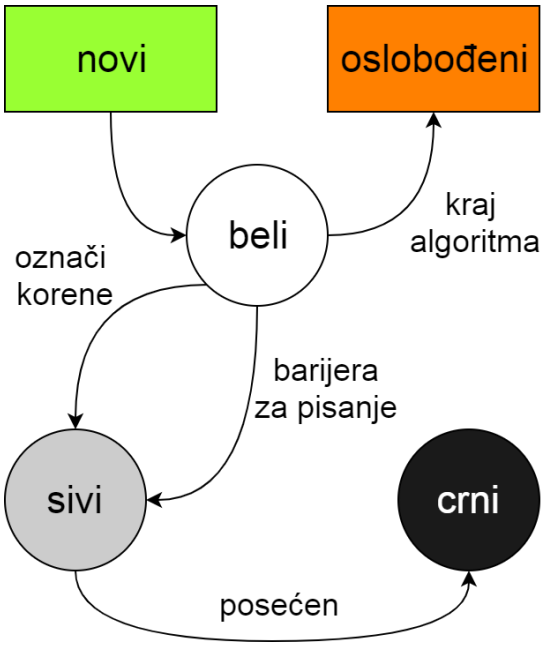
Programski jezik Go ima automatsko upravljanje memorijom odnosno sakupljanje smeća (engl. „garbage collection"). Sakupljač skenira memoriju, pronalazi objekte na koje više ne postoji ni jedna referenca i nakon toga ih briše i oslobađa memoriju. Upotrebom paketa `runtime`, sakupljač se može eksplicitno pozvati korišćenjem funkcije `GC()`.

Od verzije Go 1.5, sakupljač smeća [4] kreiran je kao konkurentni, trobojni, označi-počisti (engl. „mark-sweep") sakupljač. Kod trobojnih sakupljača svaki objekat je bele, sive ili crne boje i hip segement se posmatra kao jedan povezani graf. Hip segment predstavlja segement memorije u kome se dinamčki alociraju podaci tokom izvršavanja programa.

Na početku ciklusa, svi objekti su beli. Sakupljač prvo posećuje sve korene grafa koji predstavljaju objekte kojima se može pristupiti direktno iz aplikacije, kao što su globalne promenljive i promenljive na steku, koje boji u sivo. Nakon toga, sakupljač bira neki od sivih objekata, boji ga u crno i proverava da li poseduje pokazivače ka drugim objektima. Ukoliko pronađe pokazivač ka belom objektu, označava ga sivom bojom. Proces se ponavlja dokle god postoje sivi objekti. Na kraju, belom bojom su označeni objekti do kojih nije moguće doći, odnosno objekti na koje niko ne referiše.

Čitav ovaj proces izvršava se konkurentno sa aplikacijom koja se još naziva i mutator jer menja pokazivače dok je sakupljač aktivan. Mutator ne sme da narušava pravilo da nijedan crni objekat ne pokazuje na beli kako sakupljač ne bi izgubio evidenciju objekata koji su ubačeni u deo hipa koji je već posetio. Za očuvanje ovog pravila zadužena je barijera za pisanje (engl. „write barrier") koja predstavlja funkciju koja se aktivira prilikom promene pokazivača na hipu. Barijera označava beli objekat kao sivi ukoliko u tom trenutku postoji neki pokazivač koji pokazuje na njega, čime se osigurava da će ga pre ili kasnije sakupljač obraditi, odnosno proveriti da li poseduje pokazivače ka drugim objektima. Šematski prikaz trobojnog algoritma prikazan je na slici 2.2.

Sakupljač obavlja što veći deo posla konkurentno, međutim potrebno je na kratko da zaustavi sve (engl. „stop the world") kako bi proverio potencijalne izvore sivih objekata. Pronalaženje pravog trenutka za zaustavljanje i njegovo samo trajanje,



Slika 2.2: Prikaz označavanja objekata kod trobojnog algoritma

poboljšava se svakom novom distribucijom jezika Go. Od verzije Go 1.8, pauze uobičajeno traju manje od 100 mikrosekundi, a često se dešava da je njihovo trajanje samo 10 mikrosekundi, što je značajno unapređenje od verzije Go 1.5 gde su pauze trajale nekoliko milisekundi.

2.11 Testiranje

Go ima ugrađenu podršku za automatsko testiranje paketa. Testiranje se izvršava pomoću alata `go test` i paketa `testing`. Alat se koristi za testiranje funkcija i kao rezultat daje vrednosti `PASS` ili `FAIL`. Benčmark funkcije za testiranje performansi su takođe podržane. Sa alatom dolazi veliki broj flegova koji omogućava različite opcije kao što su broj ponavljanja benčmark testova ili broj korišćenih procesora prilikom testiranja. Testovi se izdvajaju u posebnu test datoteku.

Funkcije za testiranje u nazivu imaju prefiks `Test`, a zatim naziv funkcije koju testiraju. Kao argument funkcije imaju parametar `T` koji se koristi za obaveštavanje o neuspešnim testovima i logovanje dodatnih informacija. Benčmark funkcije imaju prefiks `Benchmark`, a kao argument imaju parametar `B` koji poseduje veliki broj istih metoda kao i parametar `T`, sa dodatnim stavkama koje su potrebne za merenje

performansi kao što je polje `N` koje označava broj ponavljanja operacije koja se testira.

Alat `go test` takođe podržava i funkcije primere (engl. „example functions") čija je primarna uloga dokumentacija koda. Funkcije primeri imaju prefiks `Example`, a kao argument nemaju nikakve parametre i ne vraćaju nikakav rezultat. Alat prevodi i izvršava ove funkcije, a ukoliko imaju neki ispis, on se ispisuje na standardnom izlazu. Pod komentarom koji počinje sa `Output:` moguće je navesti ispis funkcije koji će biti upoređen sa ispisom funkcije na standardnom izlazu. Primer 2.17 prikazuje različite tipove funkcija za testiranje.

Primer 2.17: Različiti tipovi funkcija za testiranje

```
// funkcija koja se testira
func Sum(x int, y int) int {
    return x + y
}
// funkcija za testiranje
func TestSum(t *testing.T) {
    total := Sum(1, 2)
    if total != 3 {
        t.Errorf("Sum was incorrect, got: %d, want: %d.", total, 3)
    }
}
// benchmark funkcija
func BenchmarkSum(b *testing.B) {
    for i := 0; i < b.N; i++ {
        Sum(2,2)
    }
}
// example funkcija
func ExampleSum() {
    fmt.Println(Sum(5,0))
    fmt.Println(Sum(1,3))
    // Output:
    // 5
    // 4
}
```

2.12 Paketi

U okviru same distribucije programskog jezika Go postoji veliki broj paketa standardne biblioteke koji obezbeđuju različite usluge. Neki od osnovnih paketa su upotrebljeni u primerima: `fmt` za formatirani ulaz/izlaz; `strings` za rad sa stringovima; `math` sa implementacijama matematičkih funkcija. Još jedan paket koji se često upotrebljava je paket `os` koji predstavlja interfejs ka operativnom sistemu i omogućava funkcionalnosti kao što su rad sa sistemom datoteka, pristup argumentima komandne linije i pokretanje procesa.

U standardnoj biblioteci takođe postoje paketi za rad sa SQL bazama podataka, kriptografskim funkcijama, kompresiju podataka, rad sa slikama, podršku za mrežno programiranje i još mnogi drugi. Pored paketa standardne biblioteke postoje dodatni i prošireni paketi u okviru Go projekta koji imaju labavije uslove kompatibilnosti Go verzija. Ovde se mogu pronaći paketi za razvoj aplikacija za mobilne platforme, eksperimentalni debager, prošireni kriptografski i mrežni paketi.

Paketi van distribucije samog jezika mogu se preuzeti korišćenjem komande `go get` i navođenjem repozitorijuma u kome se paket nalazi. Na veb sajtu `https://godoc.org/` mogu se naći paketi i njihove dokumentacije koje razvijaju članovi Go zajednice koja vremenom postaje sve veća. Paketi su odobreni od strane Go projekta i mogu se naći paketi koji obezbeđuju najrazličitije alate za programiranje kao i API-je mnogih postojećih servisa.

Glava 3

Konkurentno programiranje u jeziku Go

U ovoj glavi objašnjeni su osnovni pojmovi i problemi koji se javljaju kod konkurentnog programiranja. Nakon toga, opisano je kako je konkurentnost realizovana u programskom jeziku Go, kako se ona upotrebljava i načini na koje se rešavaju određeni problemi.

3.1 Osnovni pojmovi konkurentnog programiranja

U računarstvu, *konkurentnost* označava mogućnost procesa da određene delove izvršava nezavisno jedne od drugih, bez uticanja na sam rezultat programa. Pojam konkurentnosti se često poistovećuje sa *paralelizmom*. Za razliku od konkurentnosti, paralelizam podrazumeva da se delovi jednog procesa izvršavaju istovremeno, dok konkurentnost samo definiše celine koje se mogu izvršavati u preklapajućim vremenskim intervalima, ali ne obavezno istovremeno.

Niti

Osnovni pojam u konkurentnom programiranju predstavlja *nit*. Niti su osnovne jedinice izvršavanja u okviru procesa. Jedan proces se može sastojati od više niti koje se izvršavaju konkurentno. Niti jednog procesa dele određene resurse kao što su kôd segement, segement podataka i hip segment, ali poseduju i sopstvene resurse kao što je stek.

Korišćenje niti donosi mnoge prednosti kao što su ušteda memorijskog prostora i vremena. Niti zauzimaju manje prostora nego pojedinačni procesi jer dele određene resurse i kao rezultat se kreiraju znatno brže od procesa. Takođe, niti omogućavaju aplikacijma da obavljaju druge, nezavisne zadatke dok su neki delovi blokirani ili zauzeti. Serverske aplikacije često koriste kreiranje niti umesto kreiranja procesa za svakog novog korisnika jer se niti brže kreiraju i zauzimaju manje prostora, pa je omogućeno da se brže odgovori na zahteve i opsluži veći broj korisnika [5].

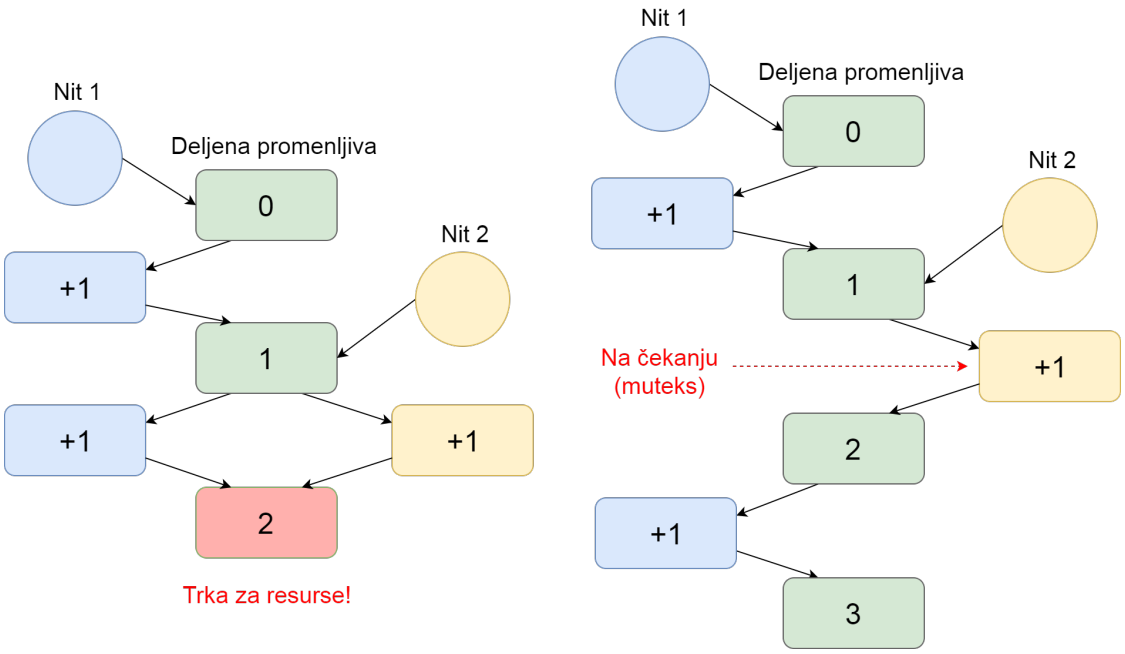
Mogući problemi pri konkurentnom izvršavanju

Osim prednosti koje konkurentnost donosi, postoje problemi koji se mogu javiti pri manipulaciji zajedničkim podacima i resursima. Nekontrolisano pristupanje zajedničkim podacima može dovesti do neispravnih rezultata. Ukoliko su neki resursi deljeni, kao na primer standardni izlaz, nesinhronizovani ispis može proizvesti isprepletane, nečitljive poruke. Situacija u kojoj krajnji rezultat zavisi od redosleda izvršavanja koraka različitih procesa ili niti koji manipulišu zajedničkim podacima ili resursima naziva se *trka za resursima* (engl. „race condition") [5].

Na slici 3.1 levo, prikazan je jednostavan primer u kome se javlja problem trke za resursima. Problem se javlja jer uvećanje vrednosti deljene promenljive ne predstavlja atomičnu operaciju. Promena vrednosti podrazumeva, čitanje tekuće vrednosti i pravljenje lokalne kopije, uvećanje vrednosti lokalne kopije i na kraju izmenu originalne vrednosti. U primeru, nit 1 čita vrednost deljene promenljive, kopira je, uvećava vrednost lokalne kopije i zatim ažurira tekuću vrednost deljene promenljive. Nakon toga, nit 2 čita tekuću vrednost deljene promenljive, pravi lokalnu kopiju i uvećava njenu vrednost. Pre nego što je nit 2 imala priliku da ažurira vrednost, nit 1 ponovo pristupa deljenoj promenljivoj, čita i kopira njenu vrednost. Na kraju, kada obe niti ažuriraju vrednost deljene promenljive, ona će biti uvećana samo jedanput, što nije bio željeni rezultat. Problem koji se javio u primeru, kao i drugi problemi koje može izazvati trka za resursima, rešavaju se različitim načinima sinhronizacije toka izvršavanja niti.

Sinhronizacija

Sinhronizacija niti predstavlja mehanizam koji osigurava da dve ili više niti neće istovremeno izvršavati deo koda u kojem se pristupa zajedničkim podacima. Ovaj



Slika 3.1: Primer problema u kome se javlja trka za resursima (levo) i primer rešavanja istog problema upotrebom muteksa (desno)

deo koda se obično naziva *kritična sekcija*. Neki od mehanizama koji se koriste su *muteksi*, *semafori* i *barijere*.

Muteksi (engl. „MUTual EXclusion", međusobno isključivanje) ili katanci predstavljaju najjednostavniji mehanizam za sinhronizaciju. Ukoliko nit počinje da izvršava kritičnu sekciju, prvo zaključava muteks što onemogućava drugim nitima da pristupe kritičnoj sekciji. Nakon izlaska iz kritične sekcije nit otključava muteks, a nit koja je bila blokirana može da pristupi kritičnoj sekciji. U primeru prikazanom na slici 3.1, kritična sekcija predstavlja uvećanje deljene promenljive i postavljanjem muteksa onemogućeno je da joj niti istovremeno pristupe, čime je problem izbegnut.

Semafor predstavlja strukturu koja u zavisnosti od zadovoljenosti uslova blokira ili propušta niti. *Barijera* za grupu niti označava mesto u kodu gde sve niti moraju da se zaustave i sačekaju dok sve niti iz grupe ne stignu do tog mesta. U zavisnosti od programskog jezika i sistema postoje i drugi mehanizmi sinhronizacije, kao i razni algoritmi i projektni uzorci koji olakšavaju rešavanje različitih problema.

Upotrebljavanje sinhronizacije na neispravan način može dovesti do problema tokom izvršavanja kao što su *mrtva petlja* ili *izgladnjivanje*. *Mrtva petlja* se dešava u situacijima kada više niti čeka na resurse koje drže druge niti koje takođe čekaju

na resurse, tako da postoji zatvoreni lanac čekanja bez napredovanja. *Izgladnjivanje* predstavlja situaciju u kojoj neke niti imaju veću prednost i time onemogućavaju drugim nitima da dođu do resursa. Probleme koji se javljaju kod konkurentnosti je ponekad teško uočiti jer se ne pojavljuju prilikom svakog izvršavanja, ali mogu dovesti do smanjenih performansi ili neupotrebljivog rešenja.

3.2 Gorutine

U programskom jeziku Go kada je u pitanju konkurentnost, osnovni pojam predstavljaju *gorutine*. Reč gorutina je nastala spajanjem reči go i reči *korutina*, koja označava funkciju koja se nezavisno izvršava. Gorutine su nešto što je karakteristično samo za jezik Go i predstavljaju niti niže kategorije. Kada se program pokrene, jedina gorutina koja postoji je glavna gorutina koja poziva funkciju `main`. Nove gorutine, kreiraju se upotrebom ključne reči `go` i navođenjem funkcije koja će se izvršavati u novoj gorutini konkurentno.

Gorutine se kreiraju sa stekom male veličine od nekoliko kilobajta, koji u zavisnosti od potreba može da se proširuje. Ova karakteristika omogućava kreiranje i izvršavanje velikog broja gorutina. Go poseduje sopstveni M:N planer izvršavanja gorutina koji mapira proizvoljan broj gorutina M u proizvoljan broj niti operativnog sistema N, što omogućava da se više gorutina može mapirati u jednu nit operativnog sistema.

Rantajm (engl. „runtime") programskog jezika Go automatski suspenduje gorutine koje su postale blokirane i nastavlja njihovo izvršavanje kada postanu odblokirane. Takođe, ukoliko se više gorutina izvršava na jednoj niti operativnog sistema, ako jedna od njih postane blokirana rantajm premešta ostale gorutine na drugu nit operativnog sistema kako i one ne bi bile blokirane [2].

3.3 Kanali

Kanali `chan` pripadaju referentnom tipu podataka koji omogućava dvosmernu komunikaciju između dve gorutine. Tip kanala definiše tip vrednosti koju je moguće poslati kroz kanal. Operator strelice `<-` , koristi se za prosleđivanje i preuzimanje vrednosti u, odnosno iz kanala u zavisnosti sa koje strane strelice se kanal nalazi. Kanal se kreira upotrebom funkcije `make(chan tip)` u kojoj se navodi tip kanala.

U kanalu bez bafera u jednom trenutku može se nalaziti samo jedna vrednost. Ukoliko neka gorutina pokuša da preuzme vrednost iz kanala, ona postaje blokirana do trenutka dok druga gorutina ne pošalje vrednost kroz isti kanal. Slično, ukoliko vrednost iz kanala nije preuzeta, gorutina postaje blokirana ako pokuša da pošalje novu vrednost kroz kanal do trenutka dok druga gorutina ne preuzme vrednost iz tog kanala. Na ovaj način kanalima može da se postigne sinhronizacija izršavanja gorutina.

Upotreba kanala prikazana je u primeru 3.1. U funkciji `main` definiše se isečak `s` i kreira se kanal `c` tipa `int`. Funkcija `sum` poziva se u jednoj gorutini za prvu polovinu, a u drugoj gorutini za drugu polovinu isečka korišćenjem ključne reči `go`. Funkcijama je kao argument prosleđen kanal `c`. Nakon poziva funkcija preuzimaju se izračunate vrednosti iz kanala. Glavna gorutina je u ovom trenutku blokirana sve dok obe gorutine ne pošalju svoje izračunate vrednosti. U funkciji se prvo izračunava suma isečka koje se zatim šalje kroz kanal `c`. Gorutina će biti blokirana ukoliko je druga gorutina već poslala svoju vrednost kroz kanal, a funkcija `main` u glavnoj gorutini je još uvek nije preuzela.

Primer 3.1: Upotreba kanala

```
func sum(s []int, c chan int) {
    res := 0
    for _, v := range s {
        res += v
    }
    c <- res
}

func main() {
    s := []int{1, 2, 3, 4, 5, 6}

    c := make(chan int)
    go sum(s[:len(s)/2], c)
    go sum(s[len(s)/2:], c)
    x, y := <-c, <-c

    fmt.Println(x, y, x+y)  // 15 6 21
}                           // redosled ispisa brojeva 15 i 6 varira
```

Pošiljalac može zatvoriti kanal upotrebom funkcije `close(kanal)`. Za razliku od datoteka kanale nije neophodno zatvoriti. Zatvaranje kanala se koristi kako bi primalac znao da se nove vrednosti više neće slati. Slanje kroz zatvoreni kanal dovodi do greške. Primalac može proveriti da li je kanal zatvoren korišćenjem dodatne promenljive prilikom naredbe preuzimanja: `v, ok := <-ch`. Petlja `for` u kombinaciji sa naredbom `range` može se koristiti za preuzimanje vrednosti iz kanala do trenutka zatvaranja.

Moguće je definisati kanale sa baferom kroz koji se mogu poslati više od jedne vrednosti bez blokiranja prilikom slanja. Veličina bafera definiše se prilikom kreiranja kanala u funkciji `make`. Podrazumevana vrednost bafera je 1. Funkcija `cap(kanal)` koristi se za dobijanje veličine bafera kanala. Osim dvosmernih mogu se definisati i jednosmerni kanali upotrebom operatora strelice. Smer kanala označen je pozicijom operatora strelice u odnosu na ključnu reč `chan`. Konverzija dvosmernog kanala u jednosmerni je dozvoljena dok suprotna situacija nije.

Upotreba jednosmernih kanala i kanala sa baferom prikazana je u primeru 3.2. U funkciji `main` kreira se dvosmerni kanal `c` sa baferom veličine 10, a zatim se funkcija `fibonacci` poziva u novoj gorutini. U `for` petlji ispisuju se vrednosti koje su poslate kroz kanal do njegovog zatvaranja. Funkcija `fibonacci` kao argumente ima broj `n` Fibonačijevih brojeva koji je potrebno da generiše i usmereni kanal `c`. Prilikom poziva ove funkcije dolazi do konverzije dvosmernog kanala u jednosmerni kanal za slanje. U petlji se kroz kanal šalju vrednosti nakon čega se kanal zatvara.

Primer 3.2: Upotreba jednosmernog kanala i kanala sa baferom

```
func fibonacci(n int, c chan<- int) {
    x, y := 0, 1
    for i := 0; i < n; i++ {
        c <- x
        x, y = y, x+y
    }
    close(c)
}
func main() {
    c := make(chan int, 10)
    go fibonacci(cap(c), c)
    for i := range c {
        fmt.Println(i) // 0 1 1 2 3 5 8 13 21 34
    }
}
```

3.4 Sinhronizacija

Pomenuto je da kanali svojim mehanizmom blokiranja mogu da se koriste kao sredstvo za sinhronizaciju izvršavanja gorutina. Ipak, pored ugrađenih mehanizama Go obezbeđuje paket `sync` kao dodatno sredstvo za sinhronizaciju. U okviru paketa `sync` nalaze se strukture za sinhronizaciju kao što su `Mutex`, `WaitGroup` i `Once`.

Tip `WaitGroup` predstavlja vid barijere koji čeka na završavanje grupe gorutina. Pre nego što se gorutine kreiraju potrebno je pozvati metod `Add(broj)` nad promenljivom `wg` tipa `WaitGroup`, koji kao argument ima broj gorutina koji je potrebno sačekati. Svakoj gorutini potrebno je proslediti referencu na `wg` koja na kraju svog izvršavanja poziva metod `Done()`. Pozivanjem metoda `Wait()`, tekuća gorutina se blokira i čeka na završetak rada svih gorutina. Metod `Done` ima efekat dekrementiranja brojača definisanog metodom `Add`, a kada se brojač spusti na nulu, gorutina koja je pozvala `Wait` prestaje da bude blokirana. Suvišni poziv metoda `Done`, odnosno spuštanje brojača ispod nule, dovodi do greške.

Primer 3.3: Upotreba paketa `sync` za sinhronizaciju

```
var wg sync.WaitGroup
var once sync.Once
var mutex sync.Mutex
var count=0

func f(x int){
    once.Do(func(){fmt.Println("The numbers are:")})

    mutex.Lock()
    count++
    mutex.Unlock()

    fmt.Println(x)
    wg.Done()
}

func main() {
    wg.Add(3)
    go f(21); go f(4); go f(9);
    wg.Wait()
    fmt.Println("Count =",count)     //The numbers are: 9 21 4 Count = 3
}                                    // redosled ispisa brojeva 9, 21 i 4 varira
```

Struktura `Once` ima implementiran metod `Do(funkcija)` koji garantaje da će funkcija koja mu je prosleđena biti izvršena samo jedanput. Tip `Mutex` predstavlja jednostavni katanac sa dva metoda: za zaključavanje `Lock()` i otključavanje `Unlock()`. Takođe postoji i muteks za čitanje i pisanje `RWMutex`, koji nudi dodatne metode `RLock()` i `RUnlock()` koji dopuštaju labaviji stepen ekskluzivnosti i dozvoljavaju istovremeno čitanje podataka u kritičnoj sekciji.

Primer 3.3 demonstrira upotrebu paketa `sync` za sinhronizaciju. Muteks je upotrebljen prilikom inkrementiranja globalne promenljive `count`, tip `once` je upotrebljen za početni ispis, a grupa za čekanje je upotrebljena prilikom čekanja na završetak rada svih gorutina.

U paketu `sync` definisane su i druge strukture za sinhronizaciju osim tipova `WaitGroup` i `Once`, ali one su uglavnom namenjene za upotrebu bibliotečkih funkcija niskog nivoa. Za sinhronizaciju na višem nivou bolje je koristiti kanale za komunikaciju između gorutina [15].

3.5 Naredba select

U programskom jeziku Go postoji specijalna naredba `select` koja funkcioniše kao naredba `switch` i upotrebljava se samo za kanale. Kao slučajevi se navode naredbe slanja ili preuzimanja, u odnosno iz kanala. Naredba je blokirana do trenutka kada neki od slučajeva može da se izvrši. Ukoliko je u jednom trenutku moguće izvršiti više slučajeva, jedan se bira na slučajan način. Upotreba naredbe `select` za implementaciju isteka vremena konekcije prikazana je u primeru 3.4. Funkcija `After(vreme)` kreira kanal kroz koji nakon zadatog vremena šalje podatak o trenutnom vremenu.

Primer 3.4: Upotreba naredbe `select`

```
select {
case msg1 := <- c1:
    fmt.Println( msg1)
case msg2 := <- c2:
    fmt.Println( msg2)
case <- time.After(10 * time.Second):
    fmt.Println("timeout")
default:
    fmt.Println("nothing ready")
}
```

```
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 3.2: Primer upozorenja koje detektor trke za resursima daje za implementaciju *Eratostenovog sita*

3.6 Detektor trke za resursima

Detektor `race` je koristan alat za automatsko detektovanje postojanja trke za resursima koji dolazi sa osnovnom distribucijom jezika Go. Koristi se dodavanjem flega `-race` nakon komande `run`, `test`, `build` ili `install` u okviru alata `go`. U izlazu koji detektor proizvodi navedeni su pozivi gorutina sa funkcijama koje pokušavaju da izvrše nesinhronizovano čitanje i pisanje određenog dela memorije. Primer upozorenja koje detektor trke za resursima daje za implementaciju *Eratostenovog sita* iz poglavlja 4.6, prikazan je na slici 3.2.

Glava 4

Poređenje sa drugim programskim jezicima

U ovoj glavi, izložene su konkurentne implementacije tri jednostavna algoritma u jezicima Go, C/C++ i Python. Implementacije demonstriraju upotrebu konkurentnosti u jeziku Go, kao i kakvi su njegovi vremenski i memorijski zahtevi u odnosu na druge pomenute jezike. Prvo sledi kratak pregled programskih jezika C/C++ i Python, a zatim poređenje i rezultati testiranja implementacija algoritama *quicksort*, *množenje matrica* i *Eratostenovo sito*.

4.1 C/C++

Programski jezik C je statički tipizirani, imperativni jezik opšte namene. Kreiran je sa svrhom ponovnog implementiranja Unix operativnog sistema početkom 70-ih godina prošlog veka. Naredbe u jeziku C efikasno se mapiraju u standardne mašinske instrukcije što ga čini idealnim za implementaciju sistemskih aplikacija. C predstavlja jedan od najkorišćenijih programskih jezika i uticao je na razvoj mnogih drugih jezika, uključujući i jezik Go.

C++ je imperativni jezik opšte namene i podržava više programski paradigmi (objektno-orijentisana, funkcionalna, generička). Nastao je kao nadogradnja jezika C sa svrhom kreiranja efikasnog jezika sa visokim stepenom apstrakcije podataka koji ima mogućnost upravljanja memorijom na niskom nivou. Kompatibilan je sa C-om tako da se biblioteke i programi koji su implementirani za C uglavnom mogu koristiti i u C++-u, osim nekoliko izuzetaka.

Postoji veliki broj biblioteka koje obezbeđuju podršku za konkurentno izvrša-

vanje u ovim jezicima, a za potrebe ovog rada izabrane su biblioteka `pthreads` i `OpenMP`.

Biblioteka `pthreads` [9] obezbeđuje sredstva za kreiranje i izvršavanje niti, kao i za njihovu sinhronizaciju. Za sinhronizaciju niti u okviru biblioteke definisani su muteksi i uslovne promenljive (promenljive koje dozvoljavaju pristup kada je uslov zadovoljen).

Biblioteka `OpenMP` [6] na efikasan način obezbeđuje jednostavno programiranje konkurentnih programa upotrebom paralelnih blokova. U okviru direktive paralelnog bloka navode se podešavanja, kao i podaci koji mogu biti deljeni između niti i podaci koji su specifični za svaku nit pojedinačno. Takođe postoje strukture kao što su paralelna `for` petlja, muteksi i barijere.

4.2 Python

Python je dinamički tipizirani, interpretirani jezik opšte namene. Jedan od osnovnih ciljeva jezika je jednostavno pisanje čitljivog koda u što manjem broju linija. Python podržava više programskih paradigmi uključujući funkcionalnu i objektno-orijentisanu i ima široku upotrebu od razvoja za veb do naučnog izračunavanja.

U Python-u je omogućeno konkurentno programiranje, međutim ono nije realizovano na najefikasniji način. Probleme prilikom konkurentnog izvršavanja proizvodi GIL (engl. „Global Interpreter Lock", katanac globalnog interpretera). GIL predstavlja mehanizam koji se koristi za sinhronizovano izvršavanje niti tako da se samo jedna nit istog procesa može izvršavati u jednom trenutku. Na ovaj način je onemogućeno paralelno izvršavanje dve ili više niti istog procesa. Ipak, neki zadaci kao što su ulazno/izlazne operacije mogu se izvršavati nezavisno od GIL-a, kao i pojedine biblioteke (npr. biblioteka `numpy`).

Za realizaciju konkurentnosti korišćena je biblioteka `Parallel Python` [8]. Biblioteka funkcioniše tako što kreira paralelni server poslova (engl. „parallel job server") na kome se može izvršavati zadati broj radnika (engl. „worker"). Svaki radnik zapravo predstavlja poseban proces koji obavlja jedan od zadatih poslova. Kako su u pitanju procesi a ne niti, nije omogućeno deljenje memorije. Procesi zahtevaju više memorijskog prostora, a i neophodno prosleđivanje potrebnih podataka dodatno utiče na memorijske zahteve. U nastavku, radi jednostavnosti prilikom poređenja, procesi su oslovljavani kao niti. Efikasna paralelizacija nije karakteristika Python-a,

```
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]
    swap A[i + 1] with A[hi]
    return i + 1
```

Slika 4.1: Pseudokod algoritma *quicksort*

ali ovde se koristi prilikom poređenja jer se jezici često upotrebljavaju u slične svrhe.

4.3 Poređenje konkurentnih implementacija

Kao kriterijumi poređenja koriste se prosečna brzina izvršavanja, maksimalna upotreba memorije i broj linija koda. Primeri su testirani na hardveru sa 48 jezgara pod Linux-om Ubuntu 16.04 ukoliko nije naglašeno suprotno. Informacije o prosečnom vremenu izvršavanja, odnose se samo na realizaciju algoritma, ne na kompletno trajanje programa. Za dobijanje informacije o prosečnoj brzini izvršavanja, primeri su pokretani za n različitih pseudoslučajno generisanih ulaza, a vrednost n isnosi 10 osim u slučajevima kada izvršavanje traje više od jednog minuta. Za ograničavanje broja jezgara na kojima se primeri izvršavaju korišćena je sitemska komanda `taskset`. Verzija jezika korišćena za implementacije u jeziku Go je 1.8.

4.4 Quicksort

Quicksort je algoritam za sortiranje brojeva u mestu koji spada u grupu algoritama podeli i vladaj. U svakom koraku, jedan element - pivot postavlja se na svoju poziciju u sortiranom nizu i niz se deli 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. Pseudokod algoritma, prikazan je na slici 4.1.

Paralelizacija algoritma je izvršena tako što se za svaki rekurzivni poziv pokreće po jedna nit/gorutina do određene dubine rekurzije kada se prelazi u sekvencijalni režim rada. Za testiranje su korišćeni pseudoslučajno generisani nizovi različitih dužina. Kodovi svih implementacija dostupni su na repozitorijumu[1].

Primer 4.1: Implementacija konkurentne funkcije *quicksort* u jeziku Go

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

    p := partition(a, low, hi)

    if depth > 0 {
        wg := sync.WaitGroup{}
        wg.Add(2)
        go func(){
            QuickSortConcurrent(a, low, p-1, depth-1)
            wg.Done()
        }()
        go func(){
            QuickSortConcurrent(a, p+1, hi, depth-1)
            wg.Done()
        }()
        wg.Wait()
    }else{
        QuickSortSequential(a,low, p-1)
        QuickSortSequential(a, p+1, hi)
    }
}
```

Implementacija u jeziku Go

U primeru 4.1 prikazana je implementacija konkurentne *quicksort* funkcije. Funkciji se prosleđuje promenljiva `depth` sa informacijom o dubini rekurzije koja se dekrementira svakim novim pozivom. Ukoliko zadata dubina rekurzije nije dostignuta,

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

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

	Konkurentno izvršavanje				Sekvencijalno izvršavanje			
n [10^6] / Verzija	1	10	30	50	1	10	30	50
Go	0.17	0.84	2.92	4.83	0.27	3.13	10.02	17.04
C/C++ omp	0.24	1.23	3.25	5.12	0.54	3.81	11.98	20.78
C/C++ pthr	0.22	2.08	4.42	10.32	0.55	4.86	15.13	26.79
Python	4.19	38.61	106.06	201.44	11.41	161.23	617.23	-

odnosno promenljiva `depth` je veća od nule, kreira se po jedna gorutina za svaki rekurzivni poziv, u suprotnom, rekurzivni pozivi se izvršavaju sekvencijalno. Kako bi se sačekalo da obe gorutine završe sa svojim radom, za sinhronizaciju se koriste grupe za čekanje. Svaki konkurentni poziv funkcije kreira svoju grupu kojoj postavlja brojač na dva, a zatim, na kraju, čeka da oba rekurzivna poziva završe sa radom. Niz se prenosi preko reference i nije potrebno nikakvo zaključavanje jer svaki poziv funkcije menja samo svoj deo niza.

Implementacije u drugim jezicima

Implementacija u Python-u razlikuje se od ostalih implementacija jer nije moguće proslediti podatke preko reference i globalne promenljive nisu deljene, tako da svaki rekurzivni poziv kreira novi niz koji vraća kao rezultat, što dodatno utiče na memorijske zahteve. Paralelizacija implementacija u drugim jezicima realizovana je na isti način.

Rezultati

Vremenska efikasnost implementacija u zavisnosti od veličine niza prikazana je u tabeli 4.1. Za nizove manje dužine, Go i C/C++ rade približno istom brzinom. Za nizove veće dužine, Go i C/C++ omp rade najbrže, dok C/C++ pthreads zahteva dva puta više vremena. Python radi najsporije i zahteva oko dvadeset puta više vremena od C/C++ pthreads verzije.

U odnosu na sekvencijalno izvršavanje, najveće ubrzanje imaju Go i C/C++ omp. Python takođe dobija veliko ubrzanje konkurentnim izvršavanjem, ali kao što je već pomenuto, radi značajno sporije od ostalih jezika.

Tabela 4.2: Prosečno vreme izvršavanja [s] sa različitom dubinom rekurzije, implementacija algoritma *quicksort*, testirano sa 48 jezgara za niz dužine $10 \cdot 10^6$

Dubina rek.	3	5	7	10	14
Go	1.69	1.21	**0.84**	1.17	1.26
C/C++ omp	2.31	1.73	**1.23**	2.20	2.68
C/C++ pthr	3.27	2.93	**2.08**	2.41	9.69
Python	5.07	**4.19**	7.13	-	-

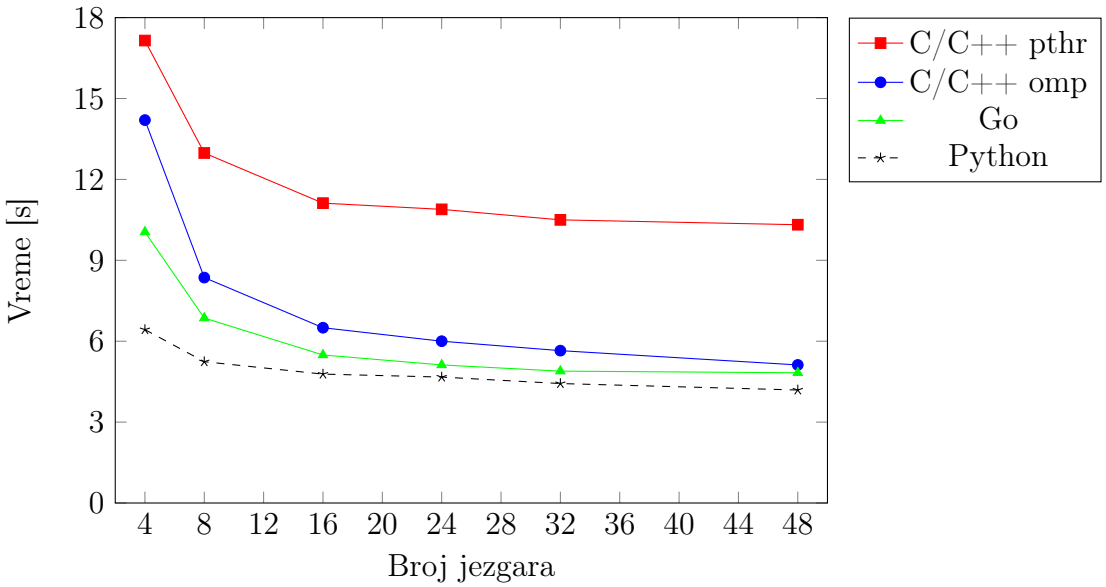
Tabela 4.3: Maksimalna upotreba memorije [MB] implementacija algoritma *quicksort* za različite dužine niza

n [10^6] / Verzija	1	10	30	50
C/C++ omp	9.2	42.7	121.5	199.1
C/C++ pthr	32.2	76.9	128.8	229.3
Go	11.3	84.7	245.5	406.7
Python	91.4	827.9	2324.5	3872.4

U tabeli 4.2 je prikazano prosečno vreme konkurentnog izvršavanja do zadate dubine rekurzije. Implementacije najefikasnije rade kada je dubina rekurzije 7. Izuzetak je Python, kome je najmanje vremena potrebno za dubinu 5. Prevelika dubina, najlošije utiče na C/C++ pthreads. Prilikom ostalih testiranja, korišćena je ona dubina rekurzije za koju implementacija pokazuje najbolje rezultate.

Vremenska efikasnost implementacija u zavisnosti od broja jezgara koji se koristi pri konkurentnom izvršavanju za niz od $50 \cdot 10^6$ brojeva, prikazana je grafikom na slici 4.2. Kao posledice samog algoritma i rekurzije, postoje granice do kojih paralelizacija i povećanje broja jezgara donosi ubrzanje. Sam izbor pivota, takođe može uticati na ubrzanje, jer može izazvati neuravnotežene rekurzivne pozive (složenost algoritma u najgorem slučaju je kvadratna) [1]. Implementacije dobijaju veliko ubrzanje do 16 jezgara, kada dodatno povećanje jezgara ne donosi značajno ubrzanje. Rezultati Python-a prikazani su za niz od 10^6 brojeva, kako bi sve implementacije mogle da se predstave na istom grafiku.

Maksimalna upotreba memorije za nizove različitih dužina prikazana je u tabeli 4.3. Memorijski najefikasniji su C/C++ osim što pthreads verzija zahteva veliku količinu memorije za nizove manje dužine. Go koristi dva puta više memorije nego C/C++ omp, dok Python zahteva najviše memorije što je posledica upotrebljavanja procesa umesto niti.



Slika 4.2: Grafik brzine izvršavanja različitih implementacija algoritma *quicksort* u zavisnosti od broja jezgara, testirano za niz od $50 \cdot 10^6$ brojeva (osim Python-a koji je testiran za niz od 10^6 brojeva)

Tabela 4.4: Dužine kodova implementacija algoritma *quicksort*

	C/C++ pthr	C/C++ omp	Go	Python
Br. linija koda	129	92	84	38

Dužine implementacija prikazane su u tabeli 4.4. Za implementaciju algoritma C/C++ pthreads verziji potreban je najveći broj linija koda, dok je u Python-u potreban najmanji.

Rezime

Za konkurentnu implementaciju algoritma *quicksort*, Go se pokazao vremenski najefikasniji. Što se tiče memorijskih zahteva, Go koristi dva puta više memorije nego C/C++, ali je memorijski i vremenski višestruko efikasniji od Python-a. Dužina koda Go implementacije je manja od C/C++-a, ali znatno veća od Python-a.

4.5 Množenje matrica

Za izradu implementacija upotrebljen je standardni algoritam za *množenje matrica*. Vrednost na poziciji *ij* proizvoda matrica A i B, izračunava se kao:

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

gde su matrice A i B kvadratne matrice dimenzije `n`.

Algoritam je paralelizovan tako što svaka nit/gorutina računa po jedan red rezultujuće matrice, odnosno, jedan red prve matrice množi sa svim kolonama druge matrice. Za testiranje, korišćene su pseudoslučajno generisane kvadratne matrice različitih dimenzija. Kodovi svih implementacija dostupni su na repozitorijumu[2].

Implementacija u jeziku Go

Implementacija konkurentne funkcije za *množenje matrica* prikazana je u primeru 4.2. Broj gorutina koji se koristi prilikom izvršavanja algoritma zadaje se kao parametar funkcije. Gorutinama koje se kreiraju unutar `for` petlje neophodno je proslediti vrednost `i` kao parametar anonimne funkcije, kako bi svaka gorutina imala svoju kopiju. U suprotnom, u svakoj sledećoj iteraciji `for` petlje, vrednost `i` bi bila ažurirana u svim gorutinama. Za razliku od implementacije 4.1, gde se niz koji se sortira prenosi pomoću reference, ovde su matrice definisane kao globalne. Nije potrebno nikakvo zaključavanje jer se ulazne matrice koriste samo za čitanje, a kod rezultujuće matrice svaka gorutina popunjava samo svoj red.

Primer 4.2: Implementacija konkurentne funkcije za *množenje matrica* u jeziku Go

```
func multiply_conc(num_routines int) {
    wg := sync.WaitGroup{}
    wg.Add(num_routines)
    for i := 0; i < num_routines; i++ {
        go func(row int){
            multiply_row(row, num_routines)
            wg.Done()
        }(i)
    }
    wg.Wait()
}
```

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

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

Br. niti	40	100	200	500	1000
C/C++ pthr	1.48	0.95	0.85	0.70	**0.65**
C/C++ omp	1.98	1.81	1.78	1.71	**1.67**
Go	2.27	2.16	2.01	1.84	**1.75**

Implementacije u drugim jezicima

Implementacija u Python-u razlikuje se zbog nemogućnosti deljenja globalnih promenljivih, pa je matrice potrebno proslediti kao parametar funkcije, što ima negativan uticaj na memorijske zahteve. Paralelizacija implementacija u drugim jezicima, realizovana je na isti način.

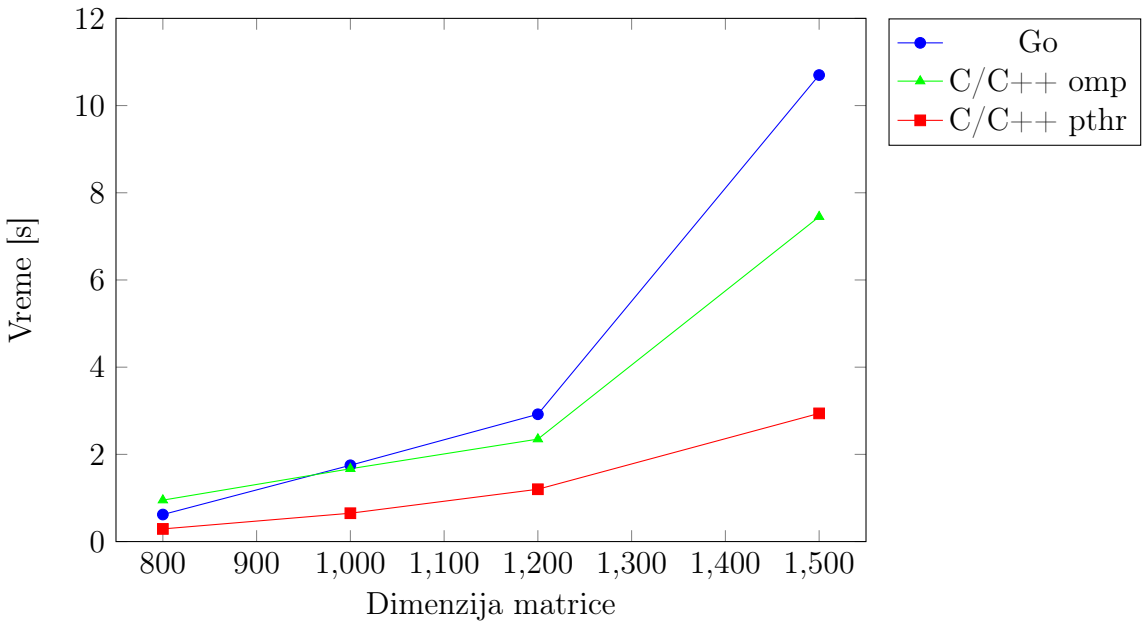
Rezultati

Grafik brzine konkurentnog izvršavanja u zavisnosti od veličine matrice, prikazan je na slici 4.3. Python se izvršava znatno sporije od ostalih implementacija, tako da nije bilo mogućnosti testirati ga za matrice iste veličine. Rezultati testiranja Python implementacije prikazani su u posebnim tabelama. Od prikazanih implementacija, C/C++ pthreads je vremenski najefikasnija. C/C++ omp i Go se izvršavaju istom brzinom za manje ulaze, ali za ulaze veće dimenzije Go radi dosta sporije.

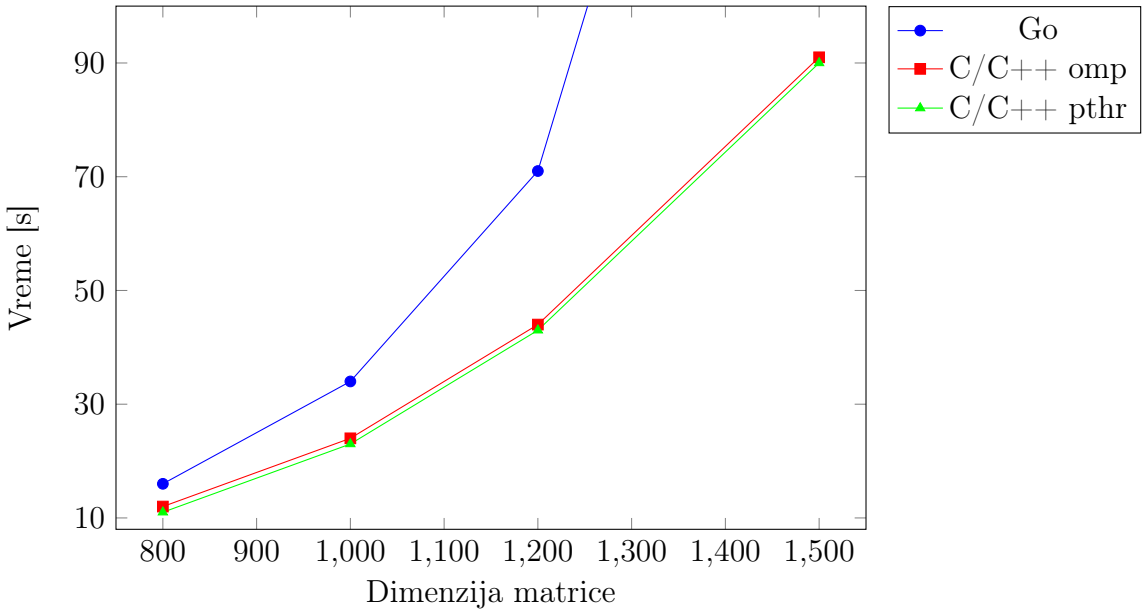
Grafik brzine sekvencijalnog izvršavanja u zavisnosti od veličine matrice prikazan je na slici 4.4. Sve implementacije dobijaju veliko ubrzanje konkurentnim izvršavanjem. Za matricu veličine 1500 implementacije rade duže od jednog minuta, dok im je pri konkurentnom izvršavanju potrebno od 3 do 10 sekundi.

Prosečno vreme izvršavanja sa različitim brojem niti za matricu veličine 1000, prikazano je u tabeli 4.5. Implementacije postižu najbolje vreme sa 1000 niti/gorutina. Prilikom ostalih testiranja korišćen je onaj broj niti/gorutina za koji implementacija pokazuje najbolje rezultate, što je u ovom slučaju n niti/gorutina za matricu dimenzije n kod svih implementacija .

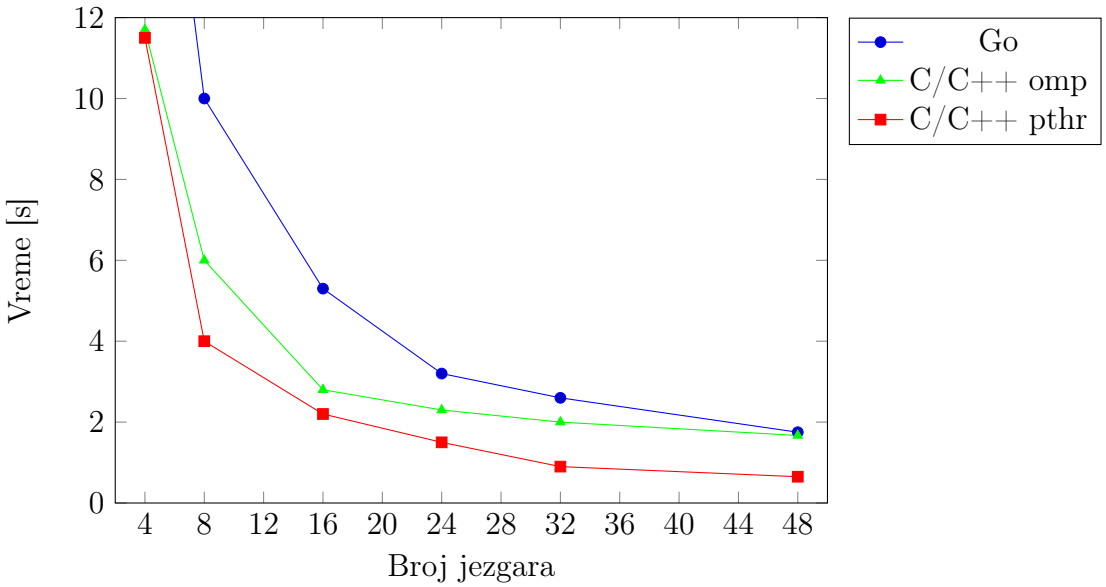
Rezultati testiranja implementacija na različitom broju jezgara prikazani su grafikom na slici 4.5. Implementacije dobijaju veliko ubrzanje povećanjem broja korišćenih jezgara. Ipak, ne postoji značajna razlika u brzini pri upotrebi 32 i 48 jezgara, što je posledica samog algoritma koji ne koristi dodatne optimizacije koje omogućavaju maksimalnu iskorišćenost procesora [7].



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



Slika 4.4: Grafik brzine sekvencijalnog izvršavanja različitih implementacija algoritma *množenja matrica* u zavisnosti od veličine matrice



Slika 4.5: Grafik brzine izvršavanja različitih implementacija algoritma *množenja matrica* u zavisnosti od broja jezgara za matricu veličine 1000

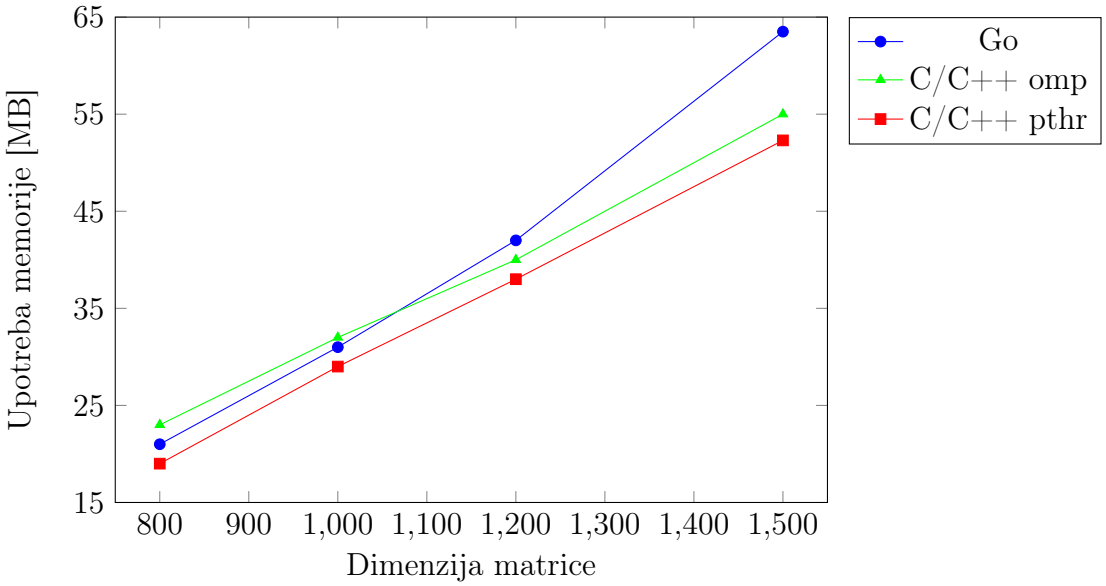
Grafik maksimalne upotrebe memorije u zavisnosti od dimenzije matrice prikazan je na slici 4.6. Implementacije imaju slične memorijske zahteve, osim što za veće ulaze Go zahteva nešto više memorije od ostalih.

Rezultati Python implementacije prikazani su u tabeli 4.6. Python radi višestruko sporije od ostalih implementacija. Za matricu veličine 800, potrebno mu je više od 3 minuta, dok ostale implementacije rade ispod jedne sekunde. U odnosu na sekvencijalno izvršavanje Python se izvršava oko tri puta brže. Kada je reč o memorijskim zahtrevima pri konkurentnom izvršavanju, Python-u je potrebna pet puta veća količina memorije nego ostalim implementacijama, što je posledica upotrebe procesa umesto niti.

U tabeli 4.7 prikazano je vreme izvršavanja u zavisnosti od broja jezgara Python implementacije, za matricu veličine 300. Kod Python-a, sa povećanjem broja jezgara ne dobija se značajno ubrzanje za razliku od drugih jezika.

U tabeli 4.8 prikazano je vreme izvršavanja sa različitim brojem niti u zavisnosti od broja jezgara Python implementacije, za matricu veličine 300. Broj niti, kao i broj jezgara nema značajan uticaj na brzinu izvršavanja kod Python-a.

Broj linija koda svih implementacija prikazan je u tabeli 4.9. Najdužu implementaciju algoritma ima C/C++ pthreads, dok najkraću ima Python.



Slika 4.6: Grafik maksimalne upotrebe memorije u zavisnosti od dimenzije matrica, različitih implementacija algortima *množenja matrica*

Tabela 4.6: Prosečno vreme izvršavanja i maksimalna upotreba memorije pri konkurentnom izvršavanju za različito n, implementacije algoritma *množenja matrica* u jeziku Python

n	Konkurentno [s]	Sekvencijalno [s]	Memorija [MB]
100	0.55	1.41	15.5
300	10.78	25.54	22.7
500	51.19	144.52	47.8
800	195.293	-	115.3

Tabela 4.7: Prosečno vreme izvršavanja u zavisnosti od broja jezgara za matricu veličine 300, implementacije algoritma *množenja matrica* u jeziku Python

Br. jezgara	4	8	16	24	32	48
Vreme [s]	14.70	12.84	11.66	10.91	10.85	10.78

Tabela 4.8: Prosečno vreme izvršavanja sa različitim brojem niti za matricu veličine 300, implementacije algoritma *množenja matrica* u jeziku Python

Br. niti	4	10	30	50	100	300
Vreme [s]	12.57	11.60	10.91	10.82	10.85	10.78

Tabela 4.9: Dužine kodova implementacija množenja matrica

	C/C++ pthr	Go	C/C++ omp	Python
Br. linija koda	78	71	55	28

```
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 4.7: Pseudokod algoritma *Eratostenovo sito*

Rezime

Za konkurentnu implementaciju algoritma *množenja matrica*, Go se pokazao vremenski efikasan koliko i C/C++ za ulaze manjih dimenzija, dok za ulaze većih dimenzija zahteva više vremena. Go upotrebljava nešto više memorije nego C/C++ za matrice većih dimenzija, ali je neuporedivo vremenski i memorijski efikasniji od Python-a. Potreban je približno isti broj linija koda za implementaciju algoritma u Go-u kao i za C/C++ pthreads verziju.

4.6 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`. Polazi se od prvog prostog broja što je 2, i eliminišu se brojevi deljivi sa 2, odnosno za svaki drugi broj odgovarajuća bulovska promenljiva se postavlja na `false` počevši od 2^2, a zatim, prelazi se na sledeći prost broj i postupak se ponavlja. Uopšteno, za prost broj `i` eliminiše se svaki i-ti broj počevši od $\texttt{i}^2$. Postupak je dovoljno ponoviti za proste brojeve koji su manji od $\sqrt{\texttt{n}}$. Pseudokod algoritma je prikazan na slici 4.7.

Paralelizacija algoritma se postiže deljenjem opsega od 2 do `n` na jednake delove. Svaka nit/gorutina dobija svoj deo opsega u okviru kojeg eliminiše brojeve koji nisu prosti. Za svaki prost broj prvo je potrebno odrediti njegov prvi umnožak unutar opsega. Iako svaka nit/gorutina ima svoj opseg, ona mora da pristupa članovima niza drugih niti/gorutina 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/gorutina još uvek nije eliminisala. Problem je rešen tako što se proverava dodatni uslov prilikom eliminacije: da li je neka druga nit/gorutina u međuvremenu označila da taj broj nije prost. Kodovi svih implementacija, dostupni su na repozitorijumu[3].

Primer 4.3: Implementacija konkurentne funkcije za određivanje prostih brojeva manjih od n u jeziku Go

```
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()
}
```

Implementacija u jeziku Go

Funkcija koja kreira gorutine prikazana je u primeru 4.3. Za svaki broj koji je trenutno označen kao prost najpre je potrebno odrediti njegov prvi umnožak, a zatim označiti sve njegove umnoške unutar opsega što je i prikazano u primeru 4.4. Koristi se globalni niz od `n` bulovskih promenljivih postavljenih na podrazumevanu

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

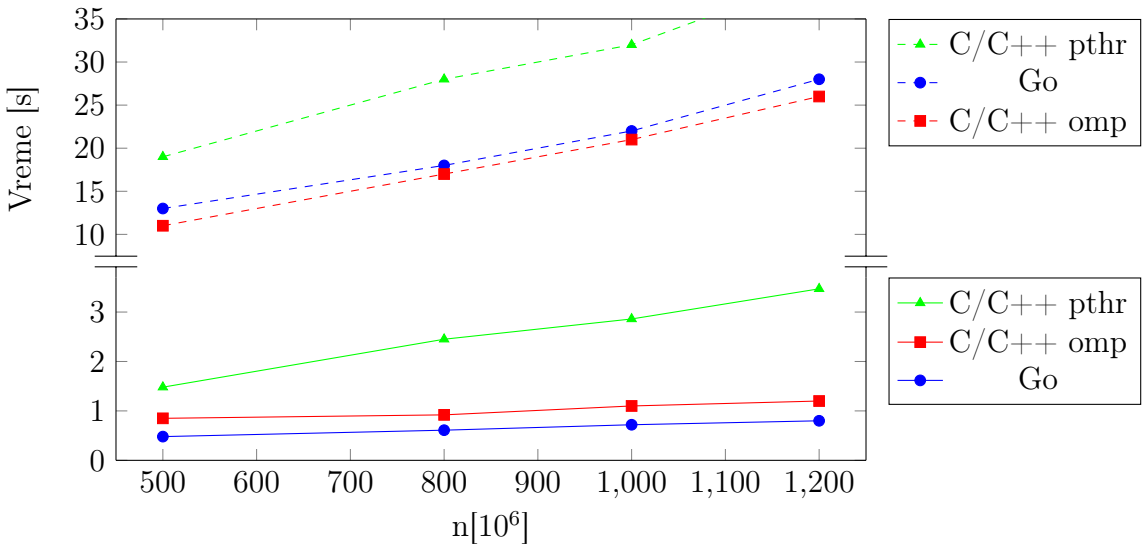
vrednost `false` umesto na `true` radi jednostavnosti. Kao što je već pomenuto, ako svaka gorutina ima svoj opseg, ona mora da pristupa i članovima niza drugih gorutina jer su joj potrebni svi prosti brojevi manji od $\sqrt{n}$. To kao posledicu dovodi do pojave trke za resursima, međutim u ovom slučaju je to dopustivo i nisu potrebni muteksi, upravo zato što se proverava dodatni uslov da li je pročitana vrednost u međuvremenu bila menjana. Ako se detektor trke za resursima pozove, dobija se izveštaj koji upozorava da je ona prisutna. Primer izveštaja se može videti u poglavlju 3.6 na slici 3.2.

Primer 4.4: Implementacija konkurentne funkcije za označavanje prostih brojeva u jeziku Go

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

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

Ostale implementacije

Implementacija u Python-u razlikuje se zbog nemogućnosti deljenja globalnih promenljivih, pa je odgovarajući deo niza potrebno proslediti kao parametar funkcije, što ima uticaj na memorijske zahteve. Na ovaj način je onemogućena provera toga da li se eliminišu umnošci broja koji je već označen da nije prost, što ima uticaj



Slika 4.8: Grafik brzine izvršavanja različitih implementacija algoritma *Eratostenovo sito* za različito n, testirano na 48 jezgara; isprekidanom linjom, prikazano je sekvencijalno izvršavanje dok je konkurentno prikazano punom linijom

na vremensku efikasnost. Implementacije u ostalim jezicima realizovane su na isti način.

Rezultati

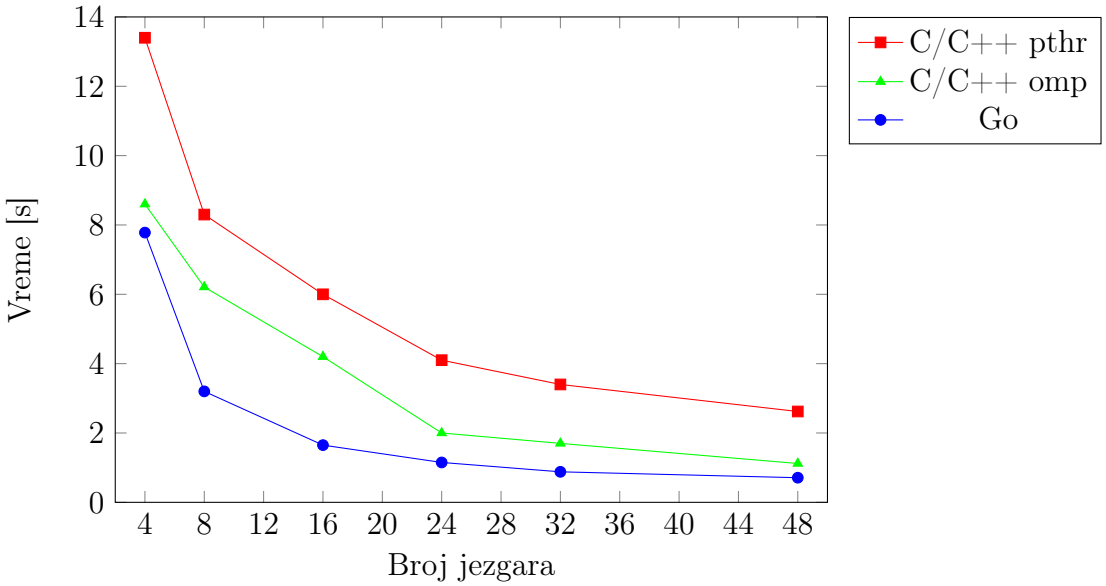
Prosečno vreme konkurentnog i sekvencijalnog izvršavanja implementacija predstavljeno je grafikom na slici 4.8. Python nije mogao da bude uključen na ovom grafiku usled znatno sporijeg izvršavanja, a rezultati Python implementacije prikazani su u posebnim tabelama. Go i C/C++ omp su vremenski najefikasniji i povećanje vrednosti n ne utiče značajno na njihovo vreme izvršavanja. U odnosu na sekvencijalno izvršavanje sve implementacije dobijaju višestruko ubrzanje.

U tabeli 4.10 prikazano je vreme izvršavanja sa različitim brojem niti/gorutina kada n iznosi $1000 \cdot 10^6$. Sve tri implementacije postižu najbolje performanse sa 10000 niti/gorutina. Broj niti/gorutina utiče na opseg pretraživanja i veličinu posla koju jedna nit/gorutina obavlja, što ima velike posledice na vremensku efikasnost.

Grafik brzine izvršavanja implementacija u zavisnosti od broja jezgara prikazan je na slici 4.9. Kod svih implementacija postoji ubrzanje sa povećanjem broja jezgara, ali ne postoji značajna razlika u brzini prilikom izvršavanja sa 32 i 48 jezgara,

Tabela 4.10: Prosečno vreme izvršavanja [s] implementacija algoritma *Eratostenovo sito* sa različitim brojem niti, testirano sa 48 jezgara kada n iznosi $1000 \cdot 10^6$

Br. niti	50	100	1000	10000	100000
Go	4.15	3.21	0.78	**0.71**	1.31
C/C++ omp	14.53	9.84	5.31	**1.12**	1.41
C/C++ pthr	7.34	6.13	5.02	**2.62**	4.72

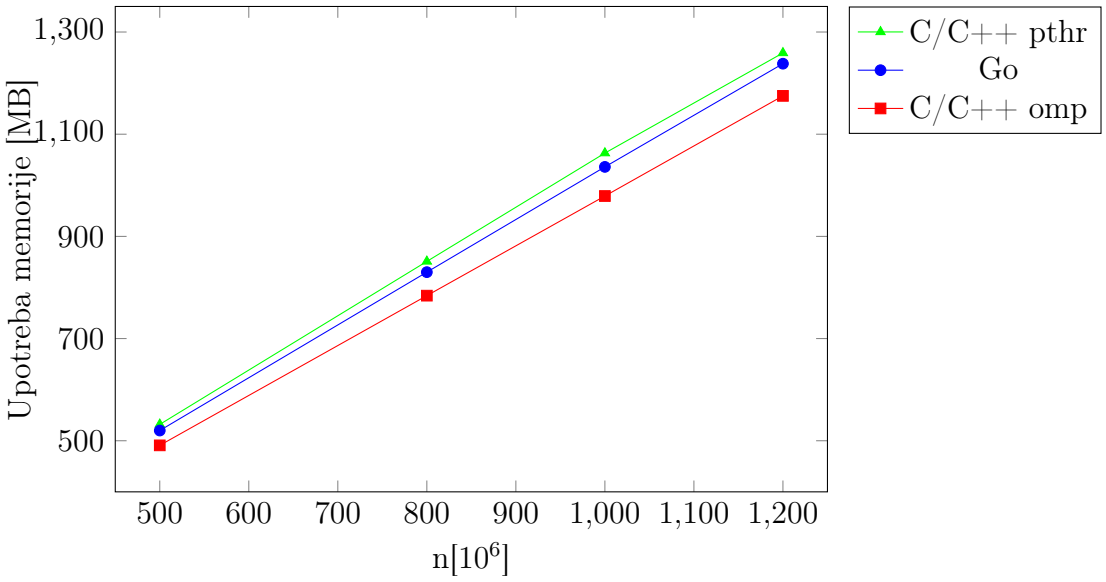


Slika 4.9: Grafik brzine izvršavanja različitih implementacija algoritma *Eratostenovo sito* u zavisnosti od broja jezgara kada n iznosi 1 milijaradu

što je posledica ograničenja samog algoritma bez optimizacije.

Maksimalna upotreba memorije u zavisnosti od n prikazana je grafikom na slici 4.10. Sve tri implementacije imaju slične memorijske zahteve. Na grafiku se vidi približno linearna zavisnost y=x, što znači da je potrebna količina memorije direktno proporcijalna vrednosti n.

Prosečno vreme izvršavanja i upotreba memorije Python implementacije pri konkurentnom izvršavanju, prikazani su u tabeli 4.11. Python radi najsporije od svih implementacija i potrebno mu je više od 20 sekundi za ulaz veličine $10 \cdot 10^6$, dok ostale implementacije rade oko jedne sekunde za ulaz veličine $500 \cdot 10^6$. U odnosu na sekvencijalno izvršavanje Python dobija ubrzanje koje je neuporedivo manje u odnosu na druge jezike. Memorijski zahtevi su takođe znatno veći u odnosu na druge implementacije usled upotrebe procesa umesto niti.



Slika 4.10: Grafik maksimalne upotrebe memorije različitih implementacija algoritma *Eratostenovo sito* za različito n

Tabela 4.11: Prosečno vreme izvršavanja i maksimalna upotreba memorije pri konkurentnom izvršavanju za različito n, implementacije algoritma *Eratostenovo sito* u jeziku Python

n	Konkurentno [s]	Sekvencijalno [s]	Memorija [MB]
100,000	0.12	0.19	17.2
1,000,000	1.95	3.15	102.4
10,000,000	23.88	29.63	888.7

Tabela 4.12: Prosečno vreme izvršavanja u zavisnosti od broja jezgara za n = 10^6, implementacije algoritma *Eratostenovo sito* u jeziku Python

Br. jezgara	4	8	16	24	32	48
Vreme [s]	2.93	2.42	2.13	2.01	1.96	1.95

Tabela 4.13: Prosečno vreme izvršavanja sa različitim brojem niti za n = 10^6, implementacije algoritma *Eratostenovo sito* u jeziku Python

Br. niti	4	10	30	50	100	500
Vreme [s]	2.38	2.03	2.05	1.98	1.95	2.07

U tabeli 4.12 prikazano je vreme izvršavanja u zavisnosti od broja jezgara, a u tabeli 4.13 vreme izvršavanja sa različitim brojem niti, za Python implementaciju kada n iznosi 10^6. Povećanje broja jezgara nema značajan uticaj na vremensku efikasnost Pythona, kao ni različit broj niti koji se koristi.

Dužine implementacija su prikazane u tabeli 4.14. Najkraća implementacija algoritma je u Python-u, dok je najduža C/C++ pthreads verzija.

Tabela 4.14: Dužine kodova implementacija algoritma *Eratostenovo sito*

	C/C++ pthr	Go	C/C++ omp	Python
Br. linija koda	98	82	70	42

Rezime

Za implementaciju algoritma *Eratostenovo sito*, Go se ispostavio kao vremenski najefikasniji sa značajnim ubrzanjem u odnosu na sekvencijalno izvršavanje. Go ima približno istu upotrebu memorije kao C/C++, kao i približno istu dužinu 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 ovoj glavi predstavljena je struktura aplikacije i opisani su pojedinačni delovi koji ilustruju različite karakteristike programskog jezika.

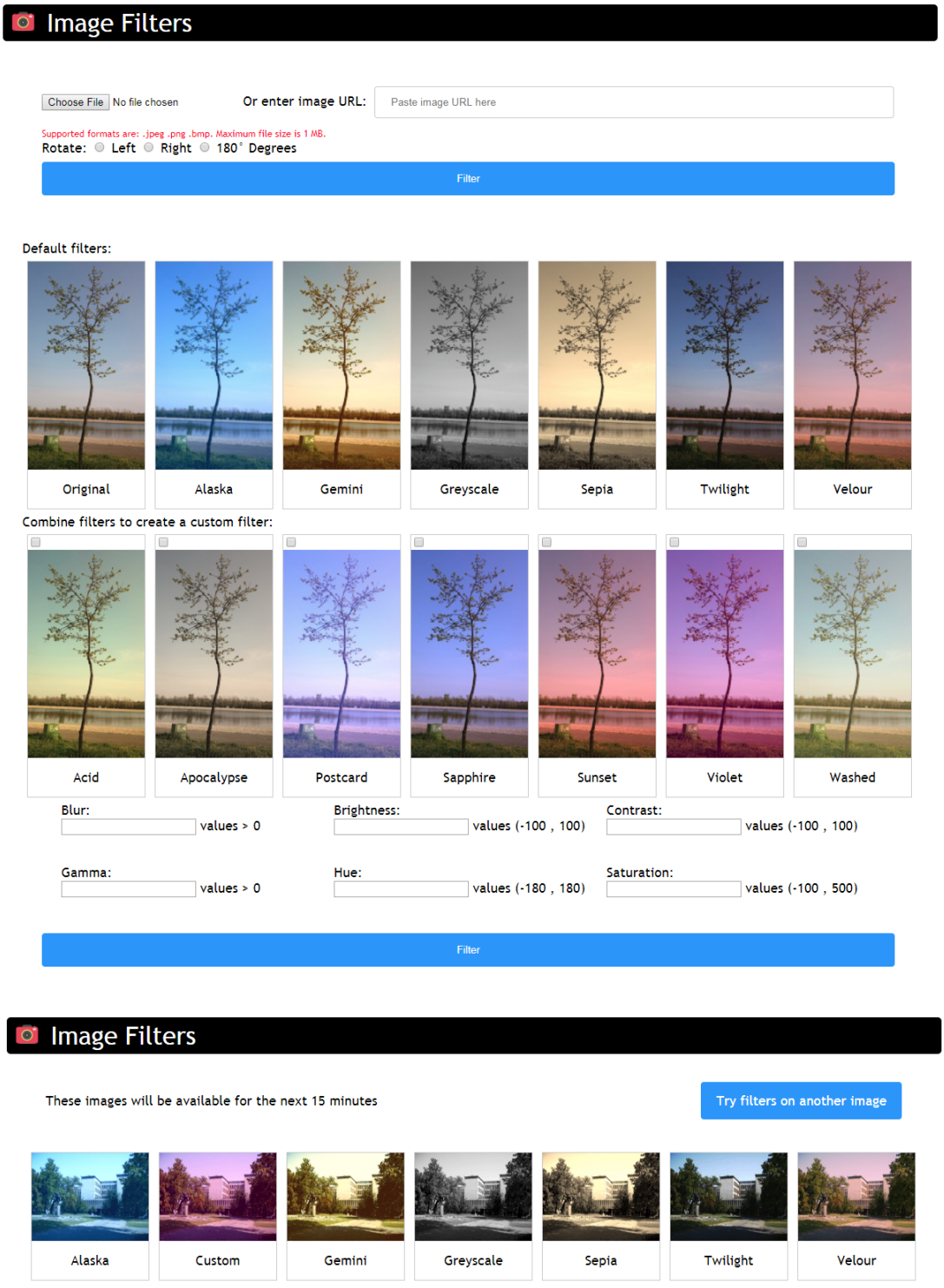
5.1 Serverska aplikacija

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 učitati 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]. Za razvoj korišćena je Go 1.8 verzija programskog jezika. Grafički interfejs aplikacije je prikazan na slici 5.1.

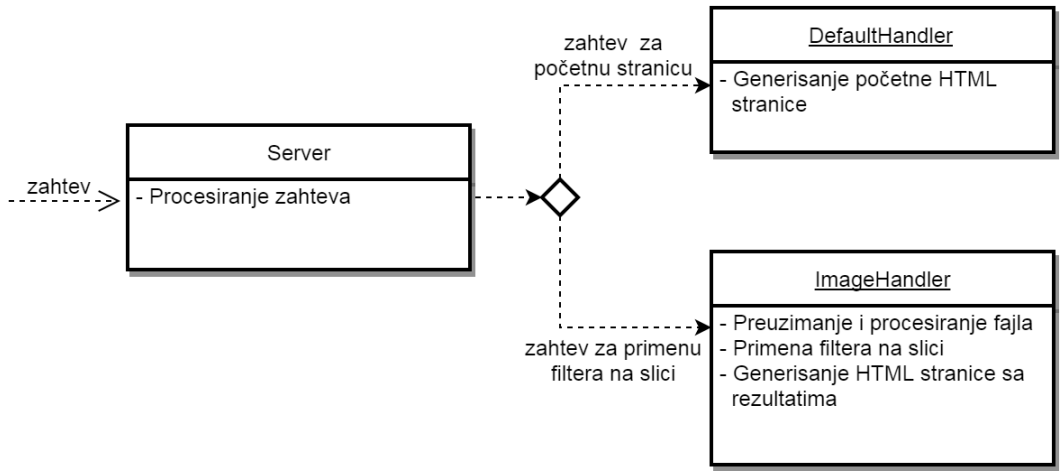
Struktura aplikacije

Struktura aplikacije predstavljena je dijagramom na slici 5.2. Server, na osnovu zahteva koji dobija od korisnika, poziva odgovarajuću funkciju za rukovanje (engl. „handler"). Definisane su dve funkcije za rukovanje: `DefaultHandle` koja ima zada-

[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

tak da generiše početnu HTML stranicu, i `ImageHandler` koja se bavi preuzimanjem i obradom datoteka, primenom filtera na slici i generisanjem HTML stranice za prikaz rezultata.

Kreiranje servera i obrada zahteva

Za kreiranje servera je korišćen paket `net/http` koji obezbeđuje skup funkcija za jednostavnu implementaciju i upravljanje serverom. Jednostavni server prikazan je u primeru 5.1.

Primer 5.1: Jednostavni server

```
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 funkcija za rukovanje za određeni šablon. Funkcija za rukovanje kao parametre 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 funkciju `Serve` kojoj prosleđuje promenljivu tipa `Handler` (drugi parametar). U pozadini, funkcija `Serve` kreira novu gorutinu za svaku HTTP konekciju i poziva funkciju za rukovanje. Ukoliko je promenljiva tipa `Handler nil`, koristi se multiplekser `DefaultServeMux` koji uparuje URL zahteve sa registrovanim šablonima i poziva odgovarajuću funkciju za rukovanje[12].

Server se može implementirati u Go-u koristeći samo dva paketa i manje od deset 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 devet bibiloteka i više od pedeset linija koda [10].

Deo koda koji se odnosi na kreiranje servera u aplikaciji nalazi se u funkciji `main` koja je prikazana u primeru 5.2. Funkcijom `HandleFunc` registrovane su dve funkcije za rukovanje: `DefaultHandler` koja se poziva kada se pristupa početnoj strani i `ImageHandler` koja se poziva prilikom obrade slike i prikazivanja rezultata. Funkcija `Handle` koristi se za registrovanje funkcije za rukovanje sa sistemom datoteka i šablona koji se odnosi na zahteve resursa. U funkciji `ListenAndServe` je postavljeno da se osluškuje na http portu zadate adrese servera koja je u ovom slučaju postavljena na `localhost`.

Primer 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)
}
```

Funkcija za rukovanje sa sistemom datoteka prikazana je u primeru 5.3. U ovoj funkciji definišu se restrikcije nad URL zahtevima za resurse i definiše se korenog 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 znakom `/`, to znači da u zahtevu nije tražena određena datoteka već

je u pitanju samo deo putanje. `DefaultServeMux` funkcioniše tako što dodaje znak / 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 [12]. U tom slučaju korisniku će biti onemogućen pristup direktorijumu i neće moći da pročita njegov sadržaj.

Primer 5.3: Funkcija za rukovanje sa sistemom datoteka

```
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 `html/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.

Primer 5.4: Izvršavanje HTML šablona

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

U funkciji `DefaultHandler` parsira se i izvršava šablon koji se prikazuje kada se pristupi početnoj strani. Deo funkcije koji se odnosi na izvršavanje šablona je prikazan u primeru 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.

U šablonu se sve instrukcije, podaci i kontrole toka navode između dvostrukih vitičastih zagrada. Izgled šablona je prikazan u primeru 5.5. Na početku svakog šablona pomoću `define` definiše se naziv, a kraj šablona je potrebno naznačiti sa `end`. Omogućeno je iteriranje nad prosleđenim podacima, naredba `if-else` 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 znakom `.` , dok promenljive počinju znakom `$` [16].

Primer 5.5: Izgled 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 obrada datoteke

Funkcija `ImageHandler` se poziva kada korisnik klikne na dugme filter i bavi se preuzimanjem datoteke, proverom zadovoljenosti svih uslova i primenom filtera. Struktura funkcije je prikazana u primeru 5.6. Unutar funkcije nalazi se jedna

naredba `select`. Za kontrolu maksimalnog broja konekcija koji server može da opsluži, koristi se semafor. Ukoliko ima slobodnih mesta izvršiće se glavni deo funkcije. Ukoliko nema slobodnih mesta, nakon isteka dozvoljenog vremena čekanja `timeout` korisniku se ispisuje poruka da je server trenutno zauzet.

Primer 5.6: Struktura funkcije ImageHandler

```
func ImageHandler(w http.ResponseWriter, r *http.Request) {
    select {
    case semaphore <- struct{}{}:
        ...
    case <- time.After(timeout):
        ErrorHandler(w,"Server too busy, try again later... ")
    }
}
```

Deo funkcije koji se odnosi na otvaranje i proveru datoteke prikazan je u primeru 5.7. U funkciji se koriste dve korisnički definisane strukture: `file_info` koja sadrži neophodne informacije o datoteci (naziv, veličina, referenca na otvorenu datoteku, čitač datoteke) 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 ukoliko je došlo do greške. Generalno, validaciju datoteke 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 datoteku i popunjava strukturu `file_info` na odgovarajući način u zavisnosti da li je prosleđen URL datoteke ili je ona učitana. Datoteci i svim ostalim elementima forme koje je korisnik uneo se pristupa pomoću `ResponseWriter`-a pozivanjem metoda `Form` ili `FormValue`. Nakon završetka sa radom sa svakom datotekom, neophodno je zatvoriti datoteku. To se postiže na najbolji način korišćem naredbe `defer` 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 datoteke se razlikuje pa se u strukturi nalaze polja za obe vrste datoteka.

Provera veličine datoteke se izvršava u metodu `check_file_size`. Međutim, korisnik je i dalje u mogućnosti da učita datoteku 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 datoteke [12]. 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 datoteku nešto iznad dozvoljene granice (u ovom slučaju 1 MB).

Funkcija `create_user_directory` kreira privremeni direktorijum u koji se smeštaju sve korisnikove slike. Naziv direktorijuma je pseudoslučajni string dužine 20 karaktera kako bi naziv bio jedinstven i direktorijum uvek mogao da se kreira. Nakon toga, datoteka kopira funkcijom `create_and_copy_file`, koja kao rezultat vraća pokazivač na otvorenu datoteku i string sa njegovom ekstenzijom. Naredbom `defer`, osigurava se zatvaranje same datoteke. Tip datoteke se proverava metodom `check_file_type` koji koristi funkciju `DetectContentType` iz paketa `http` radi sigurnosti jer sama ekstenzija datoteke nije dovoljna.

Primer 5.7: Otvaranje i provera datoteke 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}
```

Primena filtera

Za rad sa slikama koristi se paket `image`, a za primenu filtera je iskorišćen paket `gift` (Go Image Filtering Toolkit) [11] koji ne ulazi u originalnu Go distribuciju. Osnovni paket `image` 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 [13]. Paket `gift` sadrži skup filtera za obradu slika kao što su kontrast, blur, sepia i slični. Osim filtera, paket omogućava i transformacije slika poput promene veličine, isecanja dela slike, rotiranja i drugih [11].

Definisanje i primena filtera na JPEG slici prikazani su u primeru 5.8. Promenljiva `filter` je tipa `GIFT` i predstavlja niz filtera `Filter` koji se primenjuju na slici. Funkcijom `Decode` dekodira se JPEG datoteka i kao rezultat vraća promenljiva tipa `Image`. Funkcija `NewRGBA` kreira praznu sliku iste veličine kao što je i originalna slika. U paketu `image` postoje funkcije za rad i sa drugim modelima boja pored RGBA. Funkcijom `Draw` primenjuju se filteri nad orginalnom slikom, nakon čega se funkcijom `Encode` filterovana slika kodira u `dst_file` [11].

Primer 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 primeru 5.9. Funkcija `decode_image` dekodira funkciju u zavisnosti od formata slike nakon čega metod `rotate` rotira sliku ukoliko je to korisnik izabrao.

Primer 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)
```

U funkciji `create_custom_filter`, koja je prikazana u primeru 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 primeru 5.11.

Primer 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
}
```

Primer 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)},
        ...
    }
```

Nakon toga, zadatom filteru dodaju se filteri za izabrane vrednosti karakteristika koje je korisnik uneo. Za svaku karakteristiku postoji polje `input` 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 primeru 5.11. 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.

Primer 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 primeru 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.[2] 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). 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 trke za resursima 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 privremenog direktorijuma koja je prikazana u primeru 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 [17]. Po isteku vremena briše se privremeni direktorijum sa svim korisnikovim slikama.

Primer 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 poglavlju 5.1, sistem datoteka 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

[2] Napomena da od verzije Go 1.9 postoje mape sa konkurentnim pristupom u okviru paketa `sync` [15]

sigurni sa velikom verovatnoćom da napadač neće nasumičnim nagađanjem doći do slika drugih korisnika.

Korisnik nije u mogućnosti da preoptereti server prevelikom datotekom 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

Programski jezik Go je relativno nov jezik koji je za kratko vreme uspeo da se ustali u programerskoj zajednici zahvaljući svojim pogodnim karakteristikama. Jezik je u konstantnom razvoju i svakom novom verzijom značajno poboljšava svoje performanse i dobija nove karakteristike koje olakašavaju programiranje. Jednostavna sintaksa, kao i stabilnost i efikasnost koju donose statički tipizirani jezici koji se kompiliraju, čine Go idealnu alternativu dinamičkim, interpretiranim jezicima kao što je Python.

Kokurentnost koja je ugrađena u sam jezik realizovana je na takav način da omogućava intuitivno i jednostavno programiranje koje dovodi do izuzetne čitljivosti koda. Koncept gorutina jezika Go obezbeđuje vremenski i memorijski efikasno konkurentno izvršavanje koje se može uporediti sa jezicima kao što su C/C++, što je pokazano testiranjem implementacija različitih algoritama u pomenutim jezicima.

Kao jezik opšte namene, Go se upotrebljava u raznim oblastima i koristi se u velikom broju kompanija širom sveta. Paketi standardne biblioteke jezika Go u kombinaciji sa dobro realizovanom konkurentnošću, čine jezik pogodan za brz razvoj stabilnih i bezbednih serverskih aplikacija. Na primeru serverske aplikacije koja je razvijena, prikazana je jednostavnost izgradnje servera i njegovog manipulisanja, kao i mogućnosti i upotreba pojedinih paketa.

Vremenom, Go zajednica sve više raste što doprinosi razvoju samog jezika stvarnjem velikog broja paketa koji omogućavaju različite usluge. U vreme pisanja ovog rada, aktuelna verzija jezika je Go 1.9. Verzija Go 2 koja je trenutno u razvoju, trebalo bi da donese velike promene koje bi uticale na poboljšanje mnogih aspekata jezika. Činjenica je da je programski jezik Go dobar alat za razvoj softvera koji je pronašao svoju svrhu, i koji se vremenom može samo dodatno poboljšavati.

Literatura

[1] *A Parallel Stable Sort Using C++11 for TBB, Cilk Plus, and OpenMP*. on-line at:`https://software.intel.com/en-us/articles/a-parallel-stable-sort-using-c11-for-tbb-cilk-plus-and-openmp`. 2017.

[2] Katherine Cox-Buday. *Concurrency in Go: Tools and Techniques for Developers*. 1st. O'Reilly, 2017. ISBN: 978-1491941195.

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

[4] *Go GC: Prioritizing low latency and simplicity*. on-line at: `https://blog.golang.org/go15gc`. 2017.

[5] Miroslav Marić. *Operativni sistemi*. 1st. Univerzitet u Beogradu - Matematički fakultet, 2015. ISBN: 978-86-7589-101-7.

[6] *OpenMP 4.5 Summary Card - C/C++*. on-line at:`http://www.openmp.org/wp-content/uploads/OpenMP-4.5-1115-CPP-web.pdf`. 2017.

[7] *Parallel Matrix Multiplication*. on-line at:`https://medium.com/tech-vision/parallel-matrix-multiplication-c-parallel-processing-5e3aadb36f27`. 2017.

[8] *Parallel Python module API*. on-line at:`https://www.parallelpython.com/content/view/15/30/`. 2017.

[9] *POSIX Threads Programming*. on-line at:`https://computing.llnl.gov/tutorials/pthreads/`. 2017.

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

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

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

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

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

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

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

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