

UNIVERZA V LJUBLJANI  
FAKULTETA ZA MATEMATIKO IN FIZIKO

Matematika – 1. stopnja

Tjaž Eržen

**PARALELIZACIJA GRAFOVSKIH  
ALGORITMOV V FUNKCIJSKIH  
PROGRAMSKIH JEZIKIH**

Delo diplomskega seminarja

Mentorica: izr. prof. dr. Matija Pretnar

Ljubljana, 2023



# Kazalo

<b>1</b>	<b>Uvod</b>	<b>7</b>
<b>2</b>	<b>Kratek uvod v funkcijsko programiranje</b>	<b>8</b>
2.1	Zakaj funkcijski programski jeziki . . . . .	8
2.2	Funkcijske podatkovne strukture . . . . .	9
2.3	Pisanje algoritmov . . . . .	9
<b>3</b>	<b>Konec dela</b>	<b>10</b>



# **Paralelizacija grafovskih algoritmov v funkcijskih programskih jezikih**

POVZETEK

TODO

# **Parallelisation of Graph Algorithms in Functional Programming Languages**

ABSTRACT

TODO

**Math. Subj. Class. (2020):** 68R10, 68W10, 68N18, 68N19, 05C85

**Ključne besede:** TODO

**Keywords:** TODO



# 1 Uvod

Grafovski algoritmi so dan danes ključni pri modeliranju širokega nabora vsakdanjih problemov: Igrajo pomembno vlogo pri modeliranju družbenih ter računalniških omrežij, z njimi pa se prav tako razrešujejo problemi na področjih računalniške komunikacije ter podatkovne analitike. Splošno, z grafi lahko abstrahiramo kakršno koli množico odnosov med danimi entitetami, vse od računanja razdalj med mesti, pa do računanje vodnega pretoka od enega kraja do drugega.

Ker kompleksnost grafov raste nelinearno z večanjem podatkov, z njihovim računanjem ter procesiranjem pogosto trčimo ob računsko prezahtevno oviro. Načinov za reševanje tega je več, pogosto pa je ena izmed najpogostejših ozkih grl to, da našega programa nismo napisali paralelno ter s tem izkoristili celotne kapacitete računalnika, temveč smo naš program poganjali zgolj na enem jedru. Posledično je paralelizacija tovrstnih algoritmov v zadnjih letih postala zanimiva in aplikativna raziskovalna tema. V svoji diplomski nalogi se bom tako ukvarjal z paralelizacijo grafovskih algoritmov.

Funkcijski programski jeziki zagotavljajo matematične abstrakcije na višjem nivoju, kot so jih to sposobni navadni imperativni programski jeziki, kot so na primer Python, C++ in Java. Ena izmed prednosti funkcijskih programskih jezikov je, da omogočajo deklarativno programiranje, kar jih naredi bolj paralelizabilne (več o tem v nadaljevanju). Tako me je zanimanje za matematične grafe, algoritme in funkcijske jezike pripeljalo v združevanje vseh teh tem hkrati: V svoji diplomski nalogi se bom zato osredotočil na *paralelizacijo grafovskih algoritmov v funkcijskih programskih jezikih*.

**Definicija 1.1. Deklarativni programski jezik** je vrsta programskega jezika, kjer razvijalec opiše kaj naj program stori, namesto kako naj to stori. Programi so tipično strukturirani kot nizi deklaracij, ki določajo razmerja in omejitve med med različnimi entitetami znotraj problema. Nasprotno, **imperativni programski jezik** je vrsta programskega jezika, kjer programer navede zaporedje ukazov, ki naj jih računalnik izvrši za rešitev problema.

Vzporednost na nivoju pomnilnika je moč doseči na dva načina: Tako, da je pomnilnik vsem računalniškim jedrom skupen, ali pa da je pomnilnik *porazdeljen*. V tej diplomski nalogi se bom osredotočal na sisteme s skupnim pomnilnikom. V tej diplomski nalogi se bom osredotočal zgolj na sisteme s skupnim pomnilnikom ter porazdeljeno procesorsko močjo.

Vse več funkcijskih programskih jezikov, kot so na primer Scala, F# ter Haskell ima implementirane knjižnice in ogrodja za paralelizacijo grafovskih algoritmov:

- knjižnica *Graphalyze* v Haskellu zagotavlja vzporedne algoritme za najkrajšo pot ter iskanje krepko povezanih komponent.
- *GraphX* v Scali ponuja implementacijo paralelnega Googlevega PageRank algoritma.
- *Alea.cuBase* nam ponuja osnovne paralelne algoritme v jeziku F#.

Od konca lanskega leta pa imamo nov funkcijski programski jezik, ki je postal “multicore” - to je OCaml, ki se med drugim poučuje na naši fakulteti v sklopu

računalniško-orientiranih predmetov. Zato sem to priložnost izoristil, da svoje računalniške programe pišem v tem funkcijskem jeziku. V tej diplomski naloge se bom tako spustil v tovrstne knjižnice v funkcijskih jezikih, ki že obstajajo, pregledal novejša raziskave kar se tiče grafovskih algoritmov v imperativnih jezikih, to prevedel v deklarativni jezik ter vse to skupaj povezav v zaključeno celovito knjižnico v OCamlu.

## 2 Kratek uvod v funkcijsko programiranje

### 2.1 Zakaj funkcijski programski jeziki

**Definicija 2.1. Stranski učinki** so v programiranju kakršen koli odklon med čistimi matematičnimi funkcijami ter našim programom.

**Definicija 2.2. Funkcijsko programiranje** je računalniški koncept, znotraj katerega programe pišemo komponiranjem in apliciranjem matematičnih funkcij. .

Že po definiciji je moč čutiti, da bo taka vrsta programov v splošnem bolj podobna pisanju čistih matematičnih funkcij. Posledično programiranje take vrste uporabnika preko zasnove samega jezika spodbudi, piše programe z manj stranskimi učinki.

Programiranje take vrste uporabnika preko zasnove samega jezika spodbudi, da piše t.i. programe brez stranskih učinkov oz. jih omeji kakor se le da. Primer stranskega učinka bi bilo spreminjanje uporabniškega pomnilnika, ali pa branje tekstovnih datotek.

Kot rečeno, v splošnem se stranskim učinkom želimo izogniti, saj pogosto zbegajo tako razvijalca kot tudi uporabnika: Sorazvijalcu zmanjšajo preglednost naše kode ter privedejo do bolj pogostih napak, iz uporabniškega stališča pa se kakšen podatek v ozadju nehoti spremeni, kar uporabnika pogosto zbega. Tiste dele kode, ki pa stranske učinke imajo, pa izoliramo v smislu da dodatno ne počnejo še drugih čistih matematičnih operacij.

Zadnjih par let funkcijski programski jeziki zaradi svojih lastnosti pridobivajo na popularnosti. Zaradi zasnove funkcijskih programskih jezikov so tovrstni programi tipično lepše razdeljeni v manjše kose kode, znotraj ima vsak kot kode natanko eno funkcijo, kar omogoči lažjo uporabo iste programske kode na več različnih problemih. Zaradi takih učinkov ter drugih razlik, ki ločijo funkcijske ter imperativne jezike (npr. preverjanje tipov pred izvajanjem programa) pa je naš program prav tako precej lažje testirati, razhroščevati in vzdrževati. Funkcijski programer se pogosto pošali, da mu program njegovo napako sporoči že preden je program sploh zagnal. Tako je uporabnik "prisiljen" pisati bolj robustne ter zanesljive programe, manj izpostavljene potencialnim napakam ter sesutjem.

Zadnja prednost, ki bi jo izpostavil, je razlika v hitrosti funkcijskih jezikov - le-ti so po eni strani precej bolj ekspresivni od nižjenivojskih imperativnih jezikov (C++, C) ter le malo počasnejši kar se tiče izvajanja, po drugi strani pa precej hitrejši od visokonivojskih programskih jezikov (Python, Ruby, Java), s čimer imajo dobro uporabno nišo v industriji, kjer sta važna tako hitrost kot ekspresivnost jezika (npr. uporaba takih jezikov za visoko-frekvenčno trgovanje s finančnimi instrumenti).



Kot že omenjeno v uvodu, pa funkcijski jeziki prav tako omogočajo lažjo sočasnost ter paralelizabilnost programa. Vse to funkcijske jezike naredi primerne za t.i. *visoko performančne sisteme z večnitinimi programi* (ang. high performance computing).

## 2.2 Funkcijske podatkovne strukture

Ko potrebujem npr. implementacijo vrste s prednostjo v Pythonu, je pogosto dovolj, da v splošnem učbeniku najdem njeno implementacijo ter to prepisem. Razvijalci, ki pa svojo kodo pišejo v funkcijskih programskih jezikih, pa pogosto te sreče nimajo. Razloga sta dva: Prvi bi bil, da funkcijski jeziki niso tako razširjeni kot imperativni, drugi pa je, da se funkcijski jeziki med sabo bolj razlikujejo kot se med sabo razlikujejo imperativni. Poljubno psevdokodo, napisano v imperativnem smislu, je lažje prirediti v Python, kot pa je npr. poljubno psevdokodo, napisano v funkcijskem smislu prirediti v na primer OCaml.

**Definicija 2.3.** V splošnem podatkovne strukture delimo na dve skupini:

- Podatkovna struktura je **vztrajna**, če lahko vidimo njeno zgodovino spreminjanja. Taka podatkovna struktura hrani svoje prejšnje verzije.
- Podatkovna struktura je **minljiva**, je podatkovna struktura, ki ne pomni svojih prejšnjih verzij.

Funkcijski programski jeziki imajo zanimivo lastnost, da so vse podatkovne strukture avtomatsko *vztrajne*, medtem ko je v imperativnih programskih jezikih vztrajne podatkovne strukture tipično težje implementirati, prav tako pa so asimptotsko počasnejše kot “minljive” podatkovne strukture. Prednost vztrajnih podatkovnih struktur očitno ta, da lahko dostopamo do svojih prejšnjih različic, prednost minljivih struktur pa je, da so hitrejša na pram vztrajnih. Posledično so v splošnem podatkovne strukture v funkcijskih jezikih asimptotsko počasnejše kot pri imperativnih jezikih.

Še ena slabost podatkovnih struktur v funkcijskih jezikih je, da se nekaterih podatkovnih struktur preprosto ne da učinkovito implementirati na funkcijski način.

## 2.3 Pisanje algoritmov

Za pisanje algoritmov sta na voljo okolji `algorithm` in `algorithmic` iz paketov `algorithm` in `algorithmix`, ki sodelujeta podobno kot `table` in `tabular`. Algoritmi plavajo med tekstom, enako kot slike in tabele, nanje se lahko tudi sklicujemo, kot prikazano v izvorni kodi in v algoritmu

`ref alg:metoda`. Sklicujemo se lahko tudi na pomembne vrstice, npr. na vrstico `ref alg:pomembna-vrstica`, ki predstavlja glavni del algoritma. Za primer pisanja algoritma se posvetujte s primerom v tem dokumentu, za bolj napredne primere uporabe, kot na primer razbijanje algoritma na več kosov, pa z (precej razumljivo) uradno dokumentacijo<sup>1</sup>. Če želite vključiti izvorno kodo nekega programa, priporočamo paket `minted`<sup>2</sup>.

<sup>1</sup><http://tug.ctan.org/macros/latex/contrib/algorithmicx/algorithmicx.pdf>

<sup>2</sup><https://github.com/gpoore/minted>

---

**Algoritem 1** Opis, ki ima enako funkcionalnost kot opis pod sliko.

---

**Vhod:** Števili  $n, m \in \mathbb{N}, n > m$ .

**Izhod:** Decimalno število  $x$ , ki aproksimira rešitev enačbe  $nx = m$ .

```
1: function REŠI( $n, m$ )                                ▷ Vsi vhodni parametri morajo biti opisani.
2:    $a \leftarrow []$                                        ▷ Spremenljivka  $a$  naj postane prazna kopica.
3:   for  $i \leftarrow 1$  to  $n$  do
4:     if  $i \bmod 7 = 5$  then
5:       HEAPOPOP( $a$ )
6:     else if  $i < 5$  then
7:       HEAPPUSH( $a, \frac{i+12}{7} + \pi$ )                    ▷ Lahko uporabljamo matematiko.
8:     else
9:       HEAPPUSH( $a, i$ )
10:    end if
11:  end for
12:   $x \leftarrow 0$                                        ▷ Prazna vrstica
13:  for each  $e$  in  $a$  do                                ▷ To je primer komentarja.
14:     $x \leftarrow 1 + \sqrt[e]{x}$ 
15:  end for
16:  while  $|x| > \varepsilon$  do
17:     $x \leftarrow x/2$ 
18:  end while
19:   $x \leftarrow m/n$ 
20:  return  $x$       ▷ Vsi izhodni parametri morajo biti opisani nad algoritmom.
21: end function
```

---

### 3 Konec dela

#### Slovar strokovnih izrazov

**funkcijsko programiranje** računalniški koncept, znotraj katerega programe pišemo s komponiranjem in apliciranjem matematičnih funkcij

**stranski učinki programov** odklon med čistimi matematičnimi funkcijami ter našim programom.

**Vztrajna podatkovna struktura**

**Minljiva podatkovna struktura**