UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO

Matematika – 1. stopnja

Tjaž Eržen

Paralelizacija grafovskih algoritmov v funkcijskih programskih jezikih

Delo diplomskega seminarja

Mentor: prof. dr./doc. dr. Matija Pretnar

Ljubljana, 2023

Kazalo

Paralelizacija grafovskih algoritmov v funkcijskih programskih jezikih

Povzetek

TODO

Paralellisation of Graph Algorithms in Functional Programming Languages

Abstract

TODO.

Math. Subj. Class. (2010): navedite vsaj eno klasifikacijsko oznako – dostopne so na `www.ams.org/mathscinet/msc/msc2010.html`

Ključne besede: Grafovski algoritmi, paralelizacija, funkcijski programski jeziki, večjedrno procesorsko računanje, OCaml 5

Keywords: Graph algorithms, paralellisation, functional programming languages, OCaml 5

1. Uvod

Grafovski algoritmi so dan danes ključni pri modeliranju širokega nabora vsakdanjih problemov: Družbena omrežja, računalniška omrežja in računalniške komunikacije ter podatkovna analitika. 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 velikost grafov raste nelinearno z večanjem podatkov, z računanjem 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 moji diplomski nalogi se bom tako ukvarjal z paralelizacijo grafovksih 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 tako ukvarjal s *paralelizacijo grafovskih algoritmov v funkcijskih programskih jezikih.*

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.

Vse več funkcijskih programskih jezikov, kot so 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 Googlovega 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 novejše 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.

Funkcijsko programiranje je računalniški koncept, znotraj katerega programe pišemo komponiranjem in apliciranjem matematičnih funkcij. 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 kakor se le da izogniti, saj zmanjšajo preglednost naše kode, prav tako pa lahko uporabnika zmede, če večkrat požene isto.

Kot rečeno, v splošnem se stranskim učinkov 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 nehote 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 katerih vsak kos počne natanko eno vlogo, 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 visokonivojskih imperativnih (Python, Ruby, Java) ter funkcijskih programskih jezikov.

Zadnja prednost, ki bi jo izpostavil, je razlika v hitrosti funkcijskih jezikov - le-ti so po eni strani precej bolj ekpresivni 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 poglavju 1, 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*.

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 prepišem. 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 2.2. Podatkovna struktura je **vztrajna**, če lahko vidimo njeno zgodovino spreminjanja. Taka podatkovna struktura hrani svoje prejšne verzije. Podatkovna struktura je **minljiva**, je podatkovna struktura, ki ne pomni svojih prejšnih 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 da so hitrejše 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.

Definicija 2.3. Funkcija $f\colon [a,b] \to \mathbb{R}$ je *zvezna*, če...

Osnovne rezultate o zveznih funkcijah najdemo v [6]. Navedimo le naslednji izrek.

Izrek 2.4. *Zvezna funkcija na zaprtem intervalu je enakomerno zvezna.*

Dokaz. Na začetku dokaza, če je to le mogoče in smiselno, razložite idejo dokaza.

Dokazovali bomo s protislovjem. Pomagali si bomo z definicijo zveznosti in s kompaktnostjo intervala. Izberimo $\varepsilon > 0$. Če f ni enakomerno zvezna, potem za vsak $\delta > 0$ obstajata x, y, ki zadoščata

$$|x - y| < \delta \quad \text{in} \quad |f(x) - f(y)| \geq \varepsilon. \tag{1}$$

□

Na enačbe se sklicujemo takole: Oglejmo si še enkrat neenačbi (1).

Če dokaz trditve ne sledi neposredno formulaciji trditve, moramo povedati, kaj bomo dokazovali. To naredimo tako, da ob ukazu za izpis besede *Dokaz* dodamo neobvezni parameter, v katerem napišemo tekst, ki se bo izpisal namesto besede *Dokaz*.

Dokaz izreka 2.4. Dokazovanja te trditve se lahko lotimo tudi takole... □

2.3. **Naslov morebitnega podrazdelka.** Besedilo naj se nadaljuje v vrstici naslova, torej za ukazom `\subsection{}` ne smete izpustiti prazne vrstice.

V tem podrazdelku si bomo ogledali še nekatere posledice zveznosti.

Lema 2.5. *Naj bo f zvezna in ...*

⋮

Na konec dela sodita angleško-slovenski slovarček strokovnih izrazov in seznam uporabljene literature. Slovar naj vsebuje vse pojme, ki ste jih spoznali ob pripravi dela, pa tudi že znane pojme, ki ste jih spoznali pri izbirnih predmetih. Najprej navedite angleški pojem (ti naj bodo urejeni po abecedi) in potem ustrezni slovenski prevod; zaželeno je, da temu sledi tudi opis pojma, lahko komentar ali pojasnilo. Slovarska gesla navajajte z ukazom `\geslo{}{}`. Med zaporednima geselskima ukazoma v LaTeX datoteki mora biti prazna vrstica, da so gesla izpisana vsako v svoji vrstici.

Pri navajanju literature si pomagajte s spodnjimi primeri; najprej je opisano pravilo za vsak tip vira, nato so podani primeri. Posebej opozarjam, da spletni viri uporabljajo paket url, ki je vključen v preambuli. Polje "ogled" pri spletnih virih je

obvezno; če je kak podatek neznan, ustrezno "polje" seveda izpustimo. Literaturo je potrebno urediti po abecednem vrstnem redu; najprej navedemo vse vire z znanimi avtorji po abecednem redu avtorjev (po priimkih, nato imenih), nato pa spletne vire, urejene po naslovih strani. Če isti vir citiramo v dveh oblikah, kot tiskani in spletni vir, najprej navedemo tiskani vir, nato pa še podatek o tem, kje je dostopen v elektronski obliki.

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

Literatura

[1] P. Chiusano, R. Bjarnason, *Functional Programming in Scala* **1**, Manning Publications Co., New York, 2015.

[2] P. Chiusano, R. Bjarnason *Naslov članka*, okrajšano ime revije **letnik revije** (leto izida) strani od–do.

[3] C. Velkovrh, *Nekaj navodil avtorjem za pripravo rokopisa*, Obzornik mat. fiz. **21** (1974) 62–64.

[4] P. Angelini, F. Frati in M. Kaufmann, *Straight-line rectangular drawings of clustered graphs*, Discrete Comput. Geom. **45** (2011) 88–140.

[5] I. Priimek, *Naslov knjige*, morebitni naslov zbirke **zaporedna številka**, založba, kraj, leto izdaje.

[6] J. Globevnik in M. Brojan, *Analiza I*, Matematični rokopisi **25**, DMFA – založništvo, Ljubljana, 2010.

[7] J. Globevnik in M. Brojan, *Analiza I*, Matematični rokopisi **25**, DMFA – založništvo, Ljubljana, 2010; dostopno tudi na `http://www.fmf.uni-lj.si/~globevnik/skripta.pdf`.

[8] S. Lang, *Fundamentals of differential geometry*, Graduate Texts in Mathematics **191**, Springer-Verlag, New York, 1999.

[9] I. Priimek, *Naslov članka*, v: naslov zbornika (ur. ime urednika), morebitni naslov zbirke **zaporedna številka**, založba, kraj, leto izdaje, str. od–do.

[10] S. Cappell in J. Shaneson, *An introduction to embeddings, immersions and singularities in codimension two*, v: Algebraic and geometric topology, Part 2 (ur. R. Milgram), Proc. Sympos. Pure Math. **XXXII**, Amer. Math. Soc., Providence, 1978, str. 129–149.

[11] I. Priimek, *Naslov dela*, diplomsko/magistrsko delo, ime fakultete, ime univerze, leto.

[12] J. Kališnik, *Upodobitev orbiterosti*, diplomsko delo, Fakulteta za matematiko in fiziko, Univerza v Ljubljani, 2004.

[13] I. Priimek, *Naslov spletnega vira*, v: ime morebitne zbirke/zbornika, ki vsebuje vir, verzija številka/datum, [ogled datum], dostopno na `spletni.naslov`.

[14] J. Globevnik in M. Brojan, *Analiza 1*, verzija 15. 9. 2010, [ogled 12. 5. 2011], dostopno na `http://www.fmf.uni-lj.si/~globevnik/skripta.pdf`.

[15] *Matrix (mathematics)*, v: Wikipedia: The Free Encyclopedia, [ogled 12. 5. 2011], dostopno na `http://en.wikipedia.org/wiki/Matrix_(mathematics)`.