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

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 omrežjih, 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 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 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 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.

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 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 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 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 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 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 2.3. V splošnem podatkovne strukture delimo na dve skupini:

- 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.

V nadaljevanju dokumenta se bom najprej posvetil vprašanju, kako graf v funkcijskih jezikih sploh predstavimo, nato pa se bom lotil algoritmov ter podatkovnih struktur na grafih.

3 Predstavitev grafa na funkcijski način

Na GIT repozitoriju moje diplomske naloge je spisana učinkovita predstavitev grafa. Implementacija je je razdeljena na tri datoteke: "node.ml", "edge.ml", "graph.ml" Dodatno sem graf v OCamlu implementiral na dva načina:

- Prvi način je implementacija grafa s seznami. Najdemo jo lahko na tem naslovu.
- Drugi način pa je implementacija grafa z OCamlovimi arrayi ter množicami. To implementacijo pa lahko najdete na tem naslovu.

Analizi vsake vrste implementacije je spodaj dodeljena svoja podsekcija.

3.1 Implementacija grafa s seznami

3.1.1 Predstavitev vozlišča

Kot rečeno, je v datoteki "node.ml" definiran objekt vozlišča. Vsako vozlišče je identificirano z *id*-jem in vsebuje informacijo homogenega tipa *'a*, ki jo poimenujem *value*. V OCamlu bi tak tip zapisali takole:

Listing 1: Definicija vozlišča v OCamlu

```
type 'a node = {
  created_time: float;
  id: int;
  mutable value: 'a;
}
```

Na intuitiven način sem definiral še metode, ki bi bilo jih pri vsakem vozlišču smiselno imeti:

Listing 2: Preostale metode za vozlišča

```
let create_node id value = {
  created_time = Unix.gettimeofday ();
  id;
  value;
}

let equal_node a b =
  (a.created_time = b.created_time) && (a.value = b.value)

let hash_node node = int_of_float node.created_time

module NodeHashtbl = Hashtbl.Make(struct
  type t = int node
  let equal = equal_node
  let hash = hash_node
end)
```

Mogoče je še najbolj nova stvar definicija po meri narejene zgoščevalne tabele, ki vsako vozlišče zakodira po navodilih funkcije *hash node* (ki vsakemu vozlišču priredi njegov časovni žig),zapove, da sta elementa enaka, kadar metoda *equal node* tako pove, ter pove , da bodo vrednosti v tej kodirni tabeli vozlišča s celoštevilskimi tipi.

3.1.2 Predstavitev povezave

V datoteki "edge.ml" je definirana povezava. Povezava povezuje dve vozlišči. Povezava je lahko ali povezava ali pa nepovezana. Poleg že omenjenega atributa *is directed*, povezava vsebuje še sledeče informacije: *created time* - časovni žig ustvarjanja informacije, *src id* - ID vozlišča, v katerem se povezava začne ter *dest id* - ID vozlišča, v katerem se povezava zaključi

Tip povezave sem sprogramiral takole:

Listing 3: Definicija povezave v OCamlu

```
type edge = {
  created_time : float;
  src_id : id;
```

```
  dest_id : id;
  is_directed : bool
}
```

Na isti način kot pri vozlišču sem dodatno še definiral metodo *hash value*, poleg tega pa sem v datoteko "edge.ml" še dodal metodi *create edge from nodes* ter *equal edge* na sledeči način:

Listing 4: Preostale metode za povezave

```
let create_edge_from_nodes node1_id node2_id is_directed =
  {created_time = Unix.gettimeofday ();
    src_id = node1_id; dest_id = node2_id;
    is_directed = is_directed}

let equal_edge edge1 edge2 =
  edge1.src_id = edge1.src_id &&
  edge2.dest_id = edge2.dest_id
```

3.1.3 Predstavitev grafa

V datoteki "graph.ml" pa je definiran graf, ki povezuje vozlišča in povezave. Vsak graf vsebuje seznam vozlišč (gre za homogen seznam, kjer je vsak element *'a node*), seznam sosednosti (seznam seznamov, kjer i-ti element tega seznama vsebuje seznam povezav, povezanih z i-tim vozliščem) ter binarno vrednostjo, če je graf usmerjen ali ne. Dodatno sem dodal še zgoščevalno tabelo *lookup*, ki kot ključe hrani vrednosti vozlišč, kot vrednosti pa seznam vseh tistih indeksov vozlišč, pri katerih se ta vrednost pojavi.

Listing 5: Definicija grafa v OCamlu

```
type 'a graph = {
  nodes: 'a node list;
  adjacency_list: edge list list;
  lookup: ('a, int list) Hashtbl.t;
  is_directed: bool;
}
```

Naprej sem si zadal naloge, da implementiram osnovne metode, ki bi jih vsak graf potreboval:

- *create_empty_graph* - Ustvari prazen graf.
- *add_node_with_content* - V obstoječi graf doda novo vozlišče
- *remove_node* - Iz obstoječega grafa odstrani neko vozlišče
- *connect_nodes* - Poveže dani dve vozlišči

Spodaj prilagam implementacijo zgoraj navedenih metod:

Listing 6: Preostale metode za graf

```
let add_node_with_content graph content =
  let new_adjacency_list = [] :: graph.adjacency_list in
  let new_node = create_node (node_count graph) content in
  let new_nodes = new_node :: graph.nodes in
```

```
  let () =
    match Hashtbl.find_opt graph.lookup content with
    | Some ids ->
      Hashtbl.replace graph.lookup content (new_node.id::ids)
    | None -> Hashtbl.add graph.lookup content [new_node.id]
  in
  { graph with nodes = new_nodes;
  adjacency_list = new_adjacency_list }

let remove_node graph node_to_remove =
  let new_nodes = List.filter (
    fun node -> node.id <> node_to_remove.id) graph.nodes in
  let new_adjacency_list = List.filteri (
    fun i _ -> i <> node_to_remove.id) graph.adjacency_list in
  let updated_adjacency_list =
    List.map (fun edges -> List.filter (
      fun edge -> edge.src_id <> node_to_remove.id
      && edge.dest_id <> node_to_remove.id) edges)
      new_adjacency_list
  in
  Hashtbl.remove graph.lookup node_to_remove.value;
  { graph with
    nodes = update_node_ids new_nodes node_to_remove.id;
    adjacency_list = updated_adjacency_list;}

let connect_nodes graph node1 node2 =
  if not (List.exists (fun node -> node.id = node1.id) graph.nodes
    ) || not (
    List.exists (fun node -> node.id = node2.id) graph.nodes) then
    graph
  else
    let created_edge = create_edge_from_nodes
      node1.id node2.id graph.is_directed in
    let new_adjacency_list = List.mapi (fun i edges ->
      match (i = node1.id, i = node2.id) with
      | (true, _) -> created_edge :: edges
      | (_, true) when not graph.is_directed ->
        {created_edge with
        src_id=created_edge.dest_id;
        dest_id=created_edge.src_id} :: edges
      | _ -> edges
      ) graph.adjacency_list in
    { graph with adjacency_list = new_adjacency_list }
```

3.2 Implementacija grafa z OCamlovimi arrayi ter množicami

Glavna razlika med prejšnjo in to implementacijo, ki jo imenujem "Implementacija grafa z OCamlovimi arrayi ter množicami" je v definiciji tipa "graph". Tipa "node" ter "edge" sta definirano enako kot v definicijah 1 ter 3

Tokrat sem tip graf inicializiral z:

- arrayem, kjer je na i-tem mestu tega objekta shranjeno i-to vozlišče. V programski kodi ta podatek imenujem *nodes*

- *seznamom sosednosti*. Za razliko od prve implementacije, hranim seznam sosednosti kot array, kjer vsak element predstavlja množico celih števil. Na i-tem mestu v seznamu sosednosti je prisotno celo število j natanko tedaj, ko sta i-to in j-to vozlišče povezani.

- Dodatno definiram podatek *node_count* (slo. število vozlišč), ki hrani informacijo o tem, koliko vozlišč imamo v našem grafu. Kadarkoli bomo v graf dodajali ali odstranjevali vozlišče, se bo tudi ta podatek povečal oz. pomanjšal za 1.

- Polji *lookup* ter *is_directed* sta definirani kot prej.

Listing 7: Definicija posodobljenega tipa graf v OCamlu

```
type 'a graph = {
  nodes: ('a node Option.t) Array.t;
  adjacency_list: IntSet.t Array.t;
  lookup: ('a, int list) Hashtbl.t;
  is_directed: bool;
  node_count: int;
}
```

Metode *create_empty_graph*, *add_node_with_content*, *remove_node*, *connect_nodes* so implementirane ter počnejo enako kot v prejšnem razdelku. Implementirane metode je moč najti na tem naslovu.

3.3 Primerjava časovne kompleksnosti med obema implementacijama

Vsaka implementacija ima vsebinsko svoje prednosti in slabosti ter je verjetno od konteksta odvisno, katero uporabimo.

Glavna prednost implementacije 3.2 pred implementacijo 3.1 temelji na tem, da je časovna kompleksnost vpogleda poljubnega elementa v array $O(1)$, časovna kompleksnost poljubnega elementa v seznam pa $O(n)$, kjer je n dolžina seznama.

metoda/številka implementacije	**Št. 1**	**Št. 2**
Seznam vozlišč	$\underline{O(V)}$	$\underline{O(V)}$
Seznam povezav	$\underline{O(V+E)}$	$\underline{O(V+E)}$
Ali sta dani vozlišči sosednji	$O(V)$	$\underline{O(1)}$
create_empty_graph	$\underline{O(1)}$	$\underline{O(1)}$
add_node_with_content	$\underline{O(1)}$	$\underline{O(1)}$
remove_node	$O(E)$	$\underline{O(V)}$
connect_nodes	$O(V)$	$\underline{O(1)}$

Tabela 1: Časovna kompleksnost osnovnih grafovskih metod pri posameznih implementacijah. Za vsako metodo je podčrtana boljša implementacija

Opomba: Implementacija št. 1 označuje implementacijo grafov z seznami, implementacija št. 2 pa označuje implementacijo grafa s seznami. Opaziti gre, da je implementacija št. 2 občutno boljša kot prva - gre pa omeniti sledeče: druga

implementacija je spisana preko OCamlovih arrayev, ki zasedejo določen prostor, ko pa želimo dodati nov element, pa se treba celoten array skopirati v novega ter vanj dodati nov element. To sem v implementaciji delno rešil tako, da sem rezerviral dovolj velik array, da se to ne bi smelo zgoditi (od konteksta odvisno pa je, kolikšna ta številka dejansko bo). Vseeno, če pa to številko presežemo, bomo morali prvič, ko dodamo novo vozlišče, "plačati"$O(V)$ časovne komplekstnosti. Z različnimi amortizacijskimi strategijami, kot npr. da vsakič ko presežemo našo trenutno velikost arraya, to velikost podvojimo še vseeno lahko pridemo skozi konstantno ($O(1)$) amortizirano časovno kompleksnostjo pri dodajanju vozlišča v naš graf.

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