BFS i DFS Obilazak Grafa (Paralelizacija)

Seminarski rad

Paralelno programiranje

Milan Cvijović, 6/24

Univerzitet Crne Gore

Prirodno Matematički Fakultet

Računarstvo I Informacione Tehnologije – Master

Uvod

Prije nego krenemo sa temom ovog rada, treba da se osvrnemo na osnove obilaska grafova i damo kratki uvod u projekat.

Osnove obilaska grafova

Često se u računarstvu susrećemo sa grafovima i samim tim postoji potreba za implementacijom adekvatnih tehnika za rad sa istim. Obilazak je jedna od neophodnih osnova za pretragu različitih grafova i stabala. Postoji dosta poznatih algoritama koji se koriste u ove svrhe, a mi ćemo obraditi dva koji su među najbitnijim – BFS i DFS. Ovi algoritmi imaju istu svrhu, ali različit princip rada.

Značaj BFS i DFS algoritama

Bitno je da za početak pojasnimo principe po kojim rade ovi algoritmi i koje su to sličnosti i razlike.

DFS (Depth-First-Search) je popularan algoritam koji se koristi za obilazak grafova ili stabala kao struktura podataka. Počinje od izvornog čvora i istražuje duboko svaku granu ili putanju prije backtracking-a. To znači da se kreće duboko kroz strukturu prije prelaska na druge grane, pa se zato i naziva depth-first. Obično se implementira pomoću rekurzije ili stack-a.

BFS (Breadth-First-Search) je algoritam koji istražuje čvorove nivo po nivo, počevši od izvornog čvora. On posjećuje sve čvorove na istoj dubini (nivou) prije prelaska na čvorove na sledećem nivou dubine. BFS se takođe koristi za pretragu grafova ili stabala.

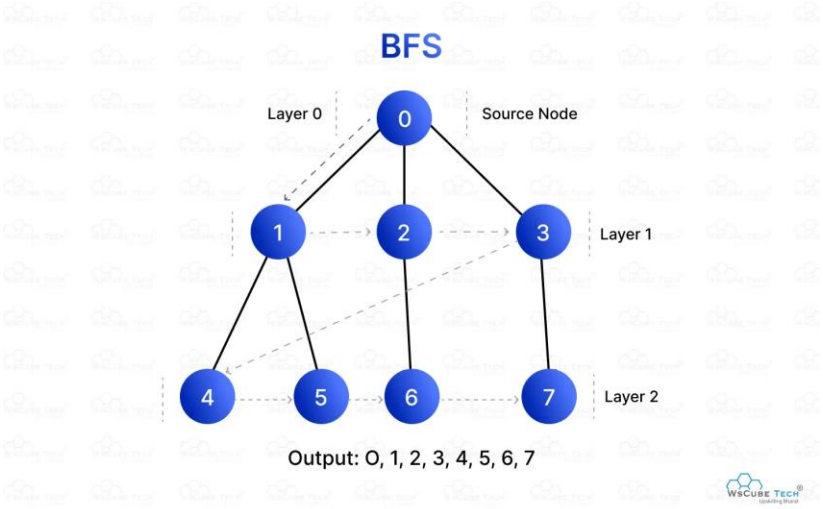
Pregled osnovnih algoritama BFS i DFS

Sada ćemo dati pregled rada i implementacije osnovnih algoritama BFS i DFS.

BFS pristup: pretraga grafa nivo po nivo, počevši od izvornog čvora, posjećivanje svih čvorova na jednom nivou prije prelaska na sledeći. BFS koristi red (queue) kako bi pratio čvorove koje treba posjetiti. Od početnog čvora BFS ide na sve direktne susjede (nivo 1), zatim se prelazi na sledeći nivo (nivo 2), ponavlja se procedura dok svi čvorovi nisu posjećeni.

Na sledećem primjeru dat je redosled posjećivanja:

- Kreće se od čvora 0
- Nivo 1: čvorovi 1, 2 i 3
- Nivo 2: čvorovi 4, 5, 6 i 7
- Izlaz: 0, 1, 2, 3, 4, 5, 6, 7.

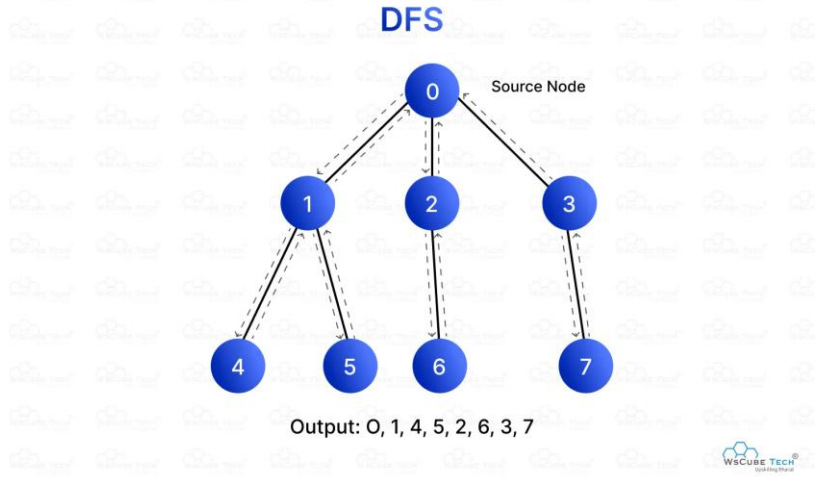


Slika 1

DFS pristup: pretraga duboko koliko je moguće duž grane prije backtracking-a. Koristi se stack ili rekurzija za praćenje čvorova i backtracking kada se dodje do kraja. Jedna grana se pretražuje u potpunosti prije nego se pređe na bilo koju drugu.

Redosled na sledećem primjeru:

- Kreni od čvora 0
- Posjeti čvor 1
- Posjeti čvor 4 (najdublji)
- Backtrack na čvor 1
- Posjeti čvor 5 (najdublji)
- Backtrack na čvor 0
- Posjeti čvor 2
- Posjeti čvor 6 (najdublji)
- Backtrack na čvor 2
- Posjeti čvor 7 (najdublji)
- Izlaz: 0, 1, 4, 5, 2, 6, 3, 7



Slika 2

Problemi klasičnog (sekvencijalnog) izvršavanja i potreba za paralelizacijom

Kod klasičnog (sekvencijalnog) izvršavanja datih algoritama, postoji nekoliko problema koji se javljaju u različitim scenarijima.

Na primjer, sa porastom veličine grafova dolazi se do velike složenosti, što ozbiljno smanjuje brzinu za ogromne grafove (npr. društvene mreže). Vremenska složenost u oba slučaja je O(V+E) gdje su V čvorovi, a E ivice.

Drugi problem može biti i prostorna složenost O(V). U slučaju velikih grafova velika je i potrošnja memorije reda i stack-a, pogotovo ako je za BFS graf širok ili ako se sa DFS ide preduboko u rekurzijama.

Takođe, ono što je i bitno u našem slučaju je neiskorišćenost višejezgarnih sistema u klasičnom izvršavanju, što znači da se ne dobija maksimalna efikasnost i brzina izvršavanja.

Iz prethodnog poglavlja može se zaključiti otkud potreba za potencijalnom paralelizacijom ovih algoritama. U nastavaku ćemo istražiti da li i kako se može izvršiti paralelizacija, koje su prednosti, potencijalni problemi i primjene.

Paralelizacija BFS algoritma

U ovom poglavlju bavićemo se detaljnije paralelizacijom BFS algoritma. Prvo ćemo dati pregled pseudokoda klasičnog BFS algoritma, a zatim analizirati mogućnosti za paralelizaciju.

Originalni BFS algoritam

Na osnovu prethodno opisanog BFS algoritma imamo sledeći pseudokod:

```
 1. function BFS(Graph G, start_node):
 2.     visited = array of size G.V, initialized to false
 3.     queue Q
 4.     visited[start_node] = true
 5.     enqueue start_node into Q
 6.     while Q is not empty:
 7.         level_nodes = empty list
 8.         level_size = size of Q
 9.         for i from 1 to level_size:
10.            node = dequeue from Q
11.            add node to level_nodes
12.            print node
13.         for each node in level_nodes:
14.            for each neighbor in G.adj[node]:
15.                if not visited[neighbor]:
16.                    visited[neighbor] = true
17.                    enqueue neighbor into Q
```

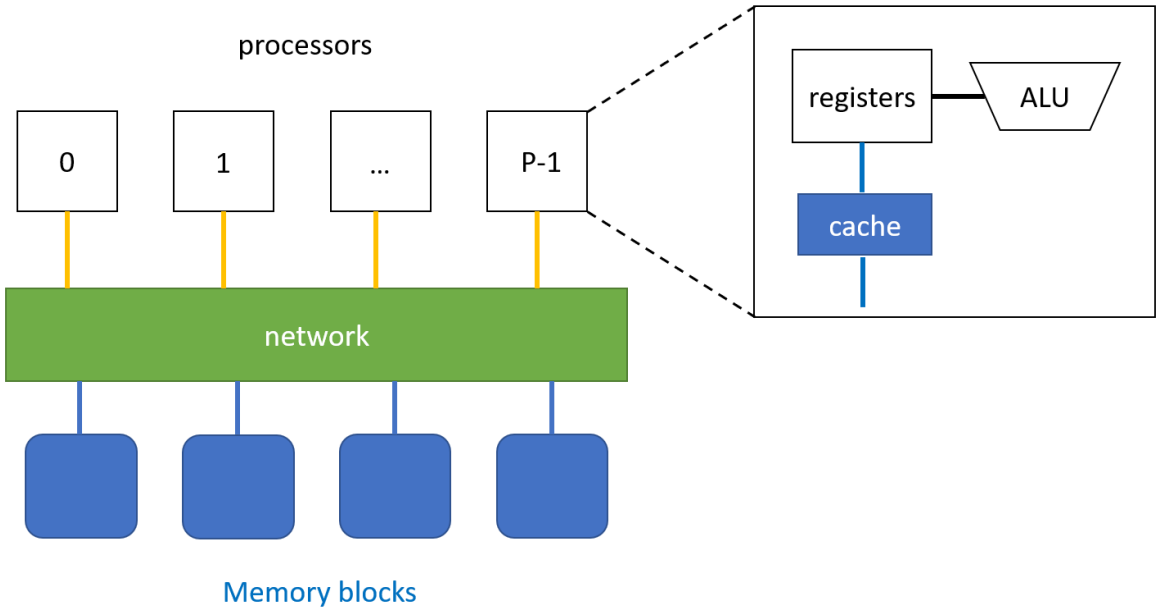
Mogućnosti paralelizacije

Prvi korak koji ćemo razmotriti je paralalizacija pomoću klasičnog PRAM pristupa, što je manje-više proširenje sekvencijalnog algoritma. Paralelno se mogu izvršavati for petlje, dok ažuriranje mora biti atomična operacija, što znači da se mora izvršiti u potpunosti bez ometanja i pauze.

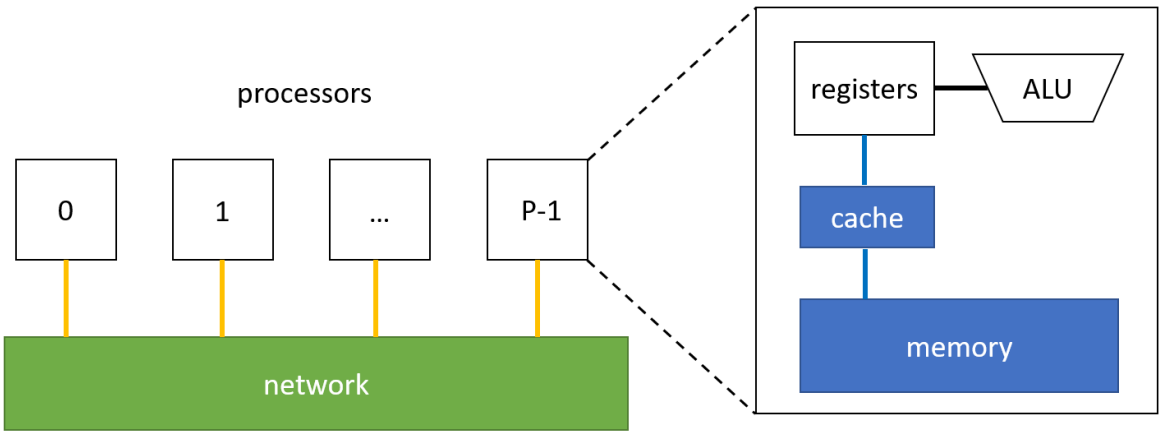
Međutim, postoje mali problemi kod ove jednostavne paralelizacije. Može se desiti da se tokom izvršavanja desi „data race“ između operacija koje dovode do potencijalnog višestrukog ažuriranja susjeda. Ovako se mogu nepotrebno trošiti resursi, ali uz pomoć sinhronizacije tačnost algoritma nije ugrožena.

Krenućemo od ovog pristupa, ali pomenućemo još nekoliko potencijalnih tehnika koje se mogu iskoristiti. Jedna od mogućnosti je paralelni BFS sa distribuiranom memorijom. Za razliku od pristupa sa dijeljenom memorijom, kod ovog pristupa se graf dijeli na više entiteta koji imaju zasebne memorije. Zbog različitih memorija entiteti moraju da komuniciraju kako bi razmjenili lokalne i distribuirane podatke. Ovakav način uključuje 1-D i 2-D particionisanje. 1-D particionisanje je jednostavniji način za kombinovanje paralelnog BFS i distribuirane memorije, dok je 2-D particionisanje druga opcija koja koristi prirodnu dekompoziciju matrice susjedstva kojom je graf predstavljen.

Na sledećim slikama su predstavljeni redom model sa dijeljenom i model sa distribuiranom memorijom.



Slika 3



Slika 4

Strategija i implementacija

Za početak ćemo krenuti sa prvim pristupom koji smo opisali, odnosno pristup sa dijeljenom memorijom uz korišćenje openmp. Kao što smo ranije opisali, koristićemo paralelne for petlje, a kod ažuriranja atomične operacije. Dijelovi koda koji ulaze u paralelizaciju su označeni počevši od linije 13.

```
 1. function BFS(Graph G, start_node):
 2.     visited = array of size G.V, initialized to false
 3.     queue Q
 4.     visited[start_node] = true
 5.     enqueue start_node into Q
 6.     while Q is not empty:
 7.         level_nodes = empty list
 8.         level_size = size of Q
 9.         for i from 1 to level_size:
10.            node = dequeue from Q
11.            add node to level_nodes
12.            print node
13.         for each node in level_nodes:   //START PARALLELIZATION
14.            for each neighbor in G.adj[node]:
15.                if not visited[neighbor]:    //CRITICAL SECTION
16.                    visited[neighbor] = true
17.                    enqueue neighbor into Q
```

Potencijalni problemi

Paralelizaciju algoritma pokušavamo u cilju povećavanja efikasnosti i brzine, međutim postoji nekoliko stvari koje treba uzeti u obzir. U zavisnosti od tipa grafa sa kojim radimo performanse mogu mnogo da variraju. Na primjer, pristup sa dijeljenom memorijom je pogodan za manje

ili srednje grafove i rad na mašini sa dosta jezgara. Distribuirani pristup je pogodan samo za ogromne grafove, poput društvenih mreža. Da bi se izbjegle potencijalne neželjene operacije potrebna je sinhronizacija pomoću atomičnih operacija. Takođe, u slučaju prevelike neregularnosti kod različitih nivoa grafa, dolazi i do neregularnog korišćenja memorije, pa je za takve slučajeve pogodno imati i load-balancing.

Paralelizacija DFS algoritma

Sada ćemo se osvrnuti i na DFS algoritam. U nastavku ćemo, kao i za BFS, analizirati originalni algoritam i mogućnosti za poboljšanja kroz paralelizaciju.

Originalni DFS algoritam

Na osnovu onoga što znamo o DFS do sada daćemo pregled pseudokoda za rekurzivnu i iterativnu implementaciju algoritma.

```
1. function RecursiveDFS(Graph G, node, visited):
2.     if visited[node] is true:
3.         return
4.     visited[node] = true
5.     print node
6.     for each neighbor in G.adj[node]:
7.         if not visited[neighbor]:
8.             RecursiveDFS(G, neighbor, visited)
```

Iterativna verzija:

```
 1. function IterativeDFS(Graph G, start_node):
 2.     visited = array of size G.V, initialized to false
 3.     stack S
 4.     push start_node into S
 5.     while S is not empty:
 6.         node = pop from S
 7.         if visited[node] is false:
 8.             visited[node] = true
 9.             print node
10.        for each neighbor in G.adj[node]:
11.            if not visited[neighbor]:
12.                push neighbor into S
```

Mogućnosti paralelizacije

Kao i kod BFS, postoji više načina da se izvrši paralelizacija. Opet će prvi korak biti razmatranje openmp pristupa, a zatim ćemo pomenuti i ostale. Pošto znamo da se DFS uglavnom implementira rekurzivno, dolazimo do problema. Klasična paralelizacija se ne slaže dobro sa

rekurzivnim pozivima, pa se rizikuje sa nekim neželjenim slučajevima. Zbog toga ćemo imati veće izmjene na algoritmu. Rekurzija u principu predstavlja problem za većinu tehnika za paralelizaciju. Osim toga razmatraćemo i nerekurzivni pristup pomoću stack-a.

Ostale tehnike se uglavnom vezuju na podjelu zadataka ili resursa na neki način radi boljih performansi. Takođe se može razmatrati distribuirani pristup kao kod BFS, gdje se veliki grafovi dijele kroz više entiteta. Tu se opet koristi MPI za komunikaciju između entiteta kako bi se osigurala sinhronizacija. Još jedan pristup je hibridni DFS koji kombinuje BFS i DFS na različitim nivoima (što nema mnogo smisla u našem slučaju). Paralelizacija na GPU je takođe teška za implementaciju zbog neslaganja sa rekurzijama, pa CUDA ili OpenCL pristupi nisu uobičajeni. Takođe napominjemo da su u ovom radu date samo ideje u manje detaljnom obliku, konkretna implementacija će biti dio završnog projekta.

Strategija i implementacija

Ponovo krećemo od openmp pristupa, međutim kao što smo napomenuli, potrebno je izvršiti određene izmjene kod rekurzivnog algoritma jer klasična paralelizacija se ne poklapa sa rekurzijom.

Pseudokod za iterativnu verziju:

```
1. function parallel_dfs(graph, start):
2.     n = size of graph
3.     initialize visited array with false
4.     initialize shared_stack as empty stack
5.     push start node onto shared_stack
6.     parallel region:
7.         while true:
8.             node = -1
9.             acquire lock on shared_stack
10.            if shared_stack is not empty:
11.                pop node from shared_stack
12.            release lock on shared_stack
13.            if node == -1: break
14.            if visited[node] is true: continue
15.            set visited[node] to true
16.            print "Thread thread_number visited: node"
17.            for each neighbor in graph[node] in reverse order:
18.                if visited[neighbor] is false:
19.                    acquire lock on shared_stack
20.                    push neighbor onto shared_stack
21.                    release lock on shared_stack
22.        end parallel region
23. end function
```

Pseudokod za rekurzivnu verziju je dat u nastavku:

```
1. function dfs_parallel(node, graph, visited):
2.     acquire lock on mutex
3.     if visited[node] is true:
4.         release lock on mutex
5.         return
6.     visited[node] = true
7.     release lock on mutex
8.     print "Visited: node (Thread thread_number)"
9.     for each neighbor in graph[node]:
10.        create a parallel task:
11.            call dfs_parallel(neighbor, graph, visited) with neighbor as a
private variable
12.        end parallel task
13. end function
```

Potencijalni problemi

Kao što smo već i pomenuli, paralelizacija DFS algoritma je teža za implementaciju. Osim što je problematičan rad sa rekurzijama, potencijalni problem mogu biti i određeni tipovi grafova. Pošto grafovi mogu imati različite oblike i veličine, može se često desiti da postoje velike neregularnosti, odnosno velike razlike između putanja u grafu. Neke putanje mogu biti dosta kratke,a za to vrijeme druge mogu biti višestruko dublje pa se otežava balansiranje zadataka. Samim tim, vremena čitanja i upisivanja podataka se dosta razlikuju i bez sinhronizacije će se desiti „data race“ između thread-ova. Zbog toga se mora voditi računa o sinhronizaciji, pristupu upisu i čitanju podataka, podjeli slobodnih zadataka između slobodnih thread-ova. Ideja je da thread koji završi zadatak ranije može da ukrade slobodni zadatak dok je drugi thread zauzet. Podjelom zadataka i dekompozicijom grafa se može pomoći load balancing za grafove koji imaju netipične oblike.

Što se tiče efikasnosti, treba napomenuti da je za velike grafove poželjnije koristiti nerekurzivni pristup, ali i u tom slučaju preveliki grafovi mogu dovesti do pretjerane potrošnje memorije.

Primjene

Za kraj ćemo se osvrnuti i na praktične upotrebe ovih algoritama da bismo upotpunili priču o potrebi za većom efikasnosti u izvršavanju.

Danas jedan od najbitnijih primjera je u analizi društvenih mreža. Na primjer u pronalaženju najkraćih puteva do konekcija (LinkedIn), klasterisanje na osnovu interakcija korisnika (Facebook), predlozi za praćenje poznanika, prijatelja, itd. na osnovu zajedičkih interesovanja.

Drugi primjer je indeksiranje stranica na pretraživačima (Google), identifikacija pokvarenih linkova, struktura sajtova i konektivnosti za SEO optimizaciju i održavanje integriteta.

U sistemima za GPS i navigaciju je potrebno pronaći najkraće puteve između lokacija, generisati instrukcije za praćenje rute, itd. U igricama je potrebno pronaći sve moguće puteve kroz mapu ili slično.

Takođe se koriste i kod alokacije resursa, raspoređivanja, optimizacije i slično.

Literatura i reference

https://www.wscubetech.com/resources/dsa/dfs-vs-bfs

https://en.wikipedia.org/wiki/Parallel_breadth-first_search

https://www.lrde.epita.fr/~bleton/doc/parallel-depth-first-search.pdf

https://research.nvidia.com/sites/default/files/publications/nvr-2017-001.pdf

https://library.fiveable.me/data-structures/unit-11/applications-bfs-dfs/study-guide/tvkrm6qmUv4ZBNDc

Sadržaj