Analiza serijske i paralelne implementacije algoritma bojenja povezanog grafa korišćenjem BFS obilaska u programskom jeziku C++

Vladimir Čornenki SV53/2021

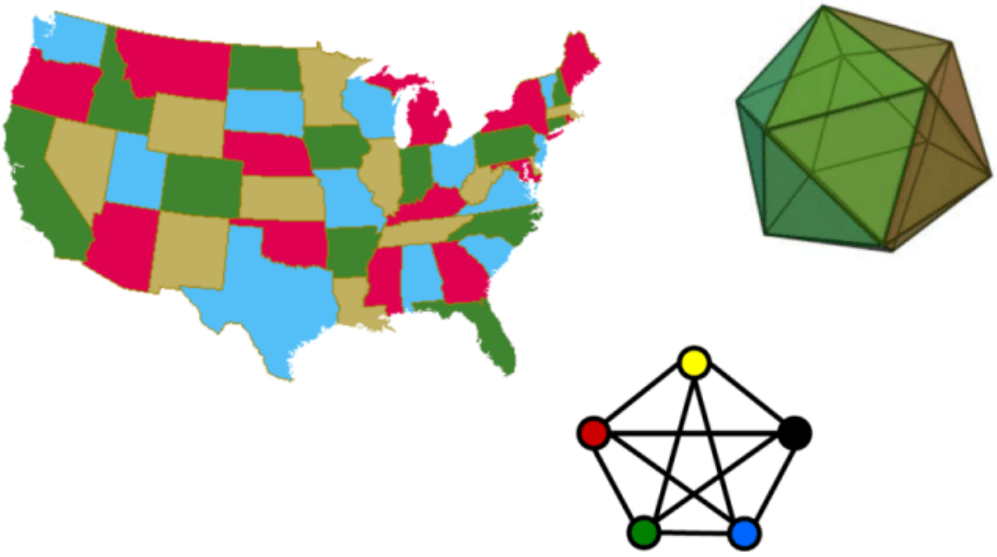
Danilo Cvijetić SV25/2021

Sadržaj:

Uvod

Problem obojenog grafa je veoma poznat problem u računarskim naukama. Postoji više verzija ovog problema. Najpoznatiji je NP-HARD problem gde se za određeni povezani graf traži najmanji mogući broj boja da bi se svi čvorovi toga grafa obojili, tako ga dva susedna čvora nemaju istu boju. Za ovaj problem konkretno se koriste razne heruristike, mašinsko učenje, evolutivni algoritmi i mnoge druge metode. Nas u ovom radu neće zanimati tačno ovaj problem već smo naš fokus usmerili na konkretno bojenje zadatog grafa proizvoljnim brojem zadatih boja poštujući gore navedena pravila.

Graf *G* je matematička struktura koja se sastoji od dva skupa: skup čvorova V(G) i skup grana E(G). Grane predstavljaju veze između pojedinačnih čvorova i ako postoji grana između dva čvora tada kažemo da su ta dva čvora povezana. Broj komponenti povezanosti predstavlja broj povezanih podskupova čvorova. Pod pojmom pravilnog bojenja grafa se smatra dodeljivanje boja čvorovima ili granama tako da svi susedni čvorovi/grane budu različite boje. Često se u literaturi izbacuje reč pravilno ako nemamo u razmatranju neke druge tipove bojenja. Mi ćemo se ovde baviti bojenjem povezanog grafa, tj. grafa koji ima samo jednu komponentu povezanosti i bavićemo se bojenjem čvorova.



Smatra se da problem bojenja grafa potiče iz oblasti geografije gde se javio problem bojenja mapa. Sve države (ili oblasti) na mapi trebale su da budu obojene

različitim bojama radi lakšeg uočavanja. Bojenje grafova ima široku praktičnu kao i teorijsku primenu. Osim uobičajenih vrsta problema, na problem bojenja mogu se zadati i ostala ograničenja u vidu načina dodele boja ili u vidu korištenja konkretnih boja. Ono uživa veliku popularnost u javnosti u vidu igre Sudoku. Ova oblast je i dalje jako aktivna oblast istraživanja. Bitnu ulogu ima u alokaciji registara u kompajlerima, raspoređivanja radio frekvencija da bi se izbegle interferencije, pravilnog rasporeda zelenih svetala na raskrsnici i mnoge druge primene.

Koncept rešenja

Naš problem sastoji se od generisanja nasumičnog grafa uz korištenje zadatih parametara. Svaki generisani graf ima jednu komponentu povezanosti.

Graf predstavljamo kao matricu gde jedan čvor može imati 0-8 susednih čvorova. Ova matrica će celo vreme biti grafički iscrtana da bi se mogao pratiti progres algoritma i vizualizovati parelelizam.

Na početku potrebno je generisati kvadratnu matricu zadate veličine (N) takvu da svako polje (i,j) u matrici M može da ima jednu vrednost iz skupa {0,1}. Ukoliko polje ima vrednost 1 to znači da se na tome polju nalazi čvor. Ukoliko je u polju vrednost 0 to znači da se ovo polje preskače pri bojenju. Na taj način formiramo mrežu suseda koja liči na lavirint. Takođe je potrebno ovu matricu generisati nasumično kako bi se algoritam testirao na različitim testnim podacima.

Parsiranjem generisane matrice formiramo graf. Za svako polje koje smatramo kao čvor, tj ima vrednost 1 u matrici uzećemo sva njegova susedna polja i formirati granu prema onim susednim poljima koja su takođe čvorovi.

Bojenje ovog grafa izvšićemo upotrebom Breath-First-Search obilaska odnosno obilaska grafa u širinu. Na taj način graf će da se boji ravnomerno od zadate tačke sve dok se sva polja ne oboje dozvoljenom bojom. Bojenje ovako generisanog grafa moći ćemo lako da vizualizujemo iscrtavanjem njegove matrice od koje je on nastao. Svaki put kada se neki čvor oboji u grafu, ažurira se i polje u matrici vezano za taj čvor i rezultat se iscrtava na ekranu.

Kada uspešno obojimo graf serijskim algoritmom, implementiraćemo i paralelni algoritam koji će bojenje grafa da vrši na sličan način, samo će na grafu biti izabrano više čvorova iz kojih ćemo da započnemo bojenje. Bojenje se vrši sve dok ima suseda

koji nisu obojeni. Heuristika za poređenje kvaliteta paralielizacije biće vreme koje je potrebno da se zadati graf oboji. Na kraju ćemo uporediti serijska i paralelna vremena.

Implementacija

Implementacija algoritma za bojenje grafa zahteva generisanje nasumične matrice zadatih veličina, njeno popunjavanje i parsiranje u graf. Generisanjem grafa možemo lako da pristupimo svim susedima nekog čvora i samim tim da ih i obojimo birajući boje koje ne postoje u listi njegovih obojenih suseda. U algoritam spada i iscrtavanje svih promena na korisnički interfejs.

Pri generisanju matrice algoritam se brine o tome da ne formira više od jedne komponente povezanosti tako što sekvencijalno prolazi kroz matricu i nasumično za svako polje bira da li je ono čvor ili ne i koliko ima suseda. Ovu istu formiranu matricu kasnije ćemo koristiti za grafičko iscrtavanje na korisničnki interfejs gde će svako polje koje je čvor biti bele boje, a polja koja nisu crne. Matrica je predstavljena kao konkurentni vektor ***tbb::concurrent_vector<int> matrixC***, a svakom (i,j) elementu se pristupa preko formule ***matrixC[i*COLUMN_NUMBER + j]***. Time smo omogućili efikasnije pristupanje matrici i kasnije efikasniju paralelizaciju gde izbegavamo da se neko polje boji više puta. Pored matrice susedstva formira se i matrica *tbb::concurrent_vector<bool> colored* koja za svako polje određuje da li je ono već obojeno ili nije.

Serijskim prolaskom kroz matricu *matrixC* možemo da generišemo graf koji joj odgovara. Graf je predstavljen u sledećem obliku:

```
class Graph {
    int V; // graph size
    list <int>* adj; // list of lists for every node
```

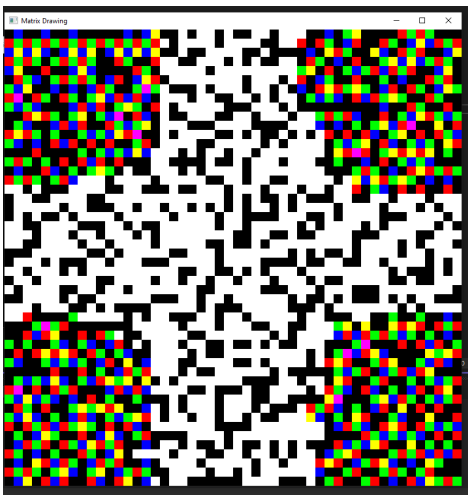
Broj čvorova V može maksimalno da bude veličine NxN odnosno kvadrat veličine matrice, ali je uvek manji od toga zbog postojanja “crnih” polja. Za svaki čvor ovde čuvamo i listu njegovih suseda pomoću liste adj.
Kada imamo izgenerisan graf možemo da otpočnemo bojenje istog. Ono se vrši korištenjem BFS oblilaska, tako što zadamo početni čvor i od njega krenemo.

```
void Graph::color_bfs(int startNode) {
    int v = startNode;
    int *queue = new int[ROW_NUMBER * COLUMN_NUMBER];
    int front = 0, rear = 0;
    colored[v] = true;
    queue[rear++] = v;
    while (front != rear) {
        v = queue[front++];
        int row = v / COLUMN_NUMBER;
        int col = v % COLUMN_NUMBER;
        set_color(get_available_color(v), row, col);
        list<int>::iterator it = adj[v].begin();
        for (it; it != adj[v].end(); it++) {
            if (!colored[*it]) {
                colored[*it] = true;
                queue[rear++] = *it;
            }
        }
    }
    delete[] queue;
}
```

BFS obilazak u širinu je takav da će početi iz proizvoljnog čvora koji se označava kao posećen i dodaje se kao jedini element reda. Potom se ponavljaju sledeći koraci sve dok red ne postane prazan: brisanje čvora sa početka reda i označavanje svakog njegovog susednog čvora kao posećenog i dodavanjem na kraj reda.

Za svaki čvor do toga dodjemo označimo da je obojen u matrici *colored* i nadjemo mu prvu dostupnu boju. Kada ustanovimo da BFS ne može da nađe više suseda koji nisu obojeni znamo da smo došli do kraja i da smo obojili celi graf jer smo na početku odredili da on ima jednu komponentu povezanosti.

Za vizualizaciju celog algoritma koristili smo **SFML C++** biblioteku i kreirali prozor koji iscrtava matricu u real-time. U BFS algoritam smo dodali ažuriranje stanja matrice koja se iscrtava čim neko polje postane obojeno. Da bi crtanje uvek bilo aktivno pokrenuli smo ga paralelno sa algoritmom za bojenje korišćenjem *tbb::parallel_invoke*. Ovim je serijski algoritam završen.



Paralelni algoritam koristi sve što i serijski s tim što BFS obilazak kreće iz više tačaka odjednom i izvršava se na različitim threadovima procesora. Za paralelizaciju korišten je **OneApi TBB** i task_group direktiva. Primer sa 4 taska za BFS i taskom za grafičko iscrtavanje bi izgledao ovako.

```
Graph g;
g.init_matrix();
g.generate_nodes();
g.parse_matrix();
task_group gr;
gr.run([&] {g.draw(); });
gr.run([&] {g.color_bfs(p0, 0); });
gr.run([&] {g.color_bfs(p1, 1); });
gr.run([&] {g.color_bfs(p2, 2); });
gr.run([&] {g.color_bfs(p3, 3); });
gr.wait();
```

Konkuretno preplitanje smo izbegli korištenjem concurrent vector-a za sve elemente grafa i matrice koji taskovi mogu simultano da čitaju i pišu.

Ispitivanje

Za ispitivanje brzine algoritma i sva testiranja korišćen je procesor AMD Ryzen 5 5600x sa 12 niti sa baznom frekvencijom 3,7GHz i maksimalnom frekvencijom 4,6GHz. Da bismo primetili razliku između serijske i paralelne implementacije algoritma, prilikom testiranja smo menjali broj niti koje se koriste prilikom paralelnog izvršavanja (2, 4, 6, 8 i 11) i veličinu grafa. Prilikom testiranja smo uočili da vreme izvršavanja paralelne implementacije značajno zavisi od izbora početnih tačaka svake niti i od generisanog grafa koji je svaki put drugačiji. Da bi smo imali što preglednije rezultate, sve testove smo vršili sa istim rasporedom početnih tačaka.

Ispitivanje opisanog algoritma na pomenutom procesoru je dalo rezultate u sledećim tabelama:

Graf: 10x10			
tredovi	serijski	paralelno	ubrzanje
2	2.08	1.54	1.351
4	2.08	1.37	1.518
6	2.08	1.24	1.677
8	2.08	1.2	1.733
11	2.08	1.17	1.778

Graf: 50x50			
tredovi	serijski	paralelno	ubrzanje
2	26.9	13.95	1.928
4	26.9	7.62	3.530
6	26.9	5.56	4.838
8	26.9	4.91	5.479
11	26.9	3.82	7.042

Graf: 100x100			
tredovi	serijski	paralelno	ubrzanje
2	103.67	52.45	1.977
4	103.67	26.94	3.848
6	103.67	19.3	5.372
8	103.67	16.2	6.399
11	103.67	10.98	9.442

Graf: 150x150			
tredovi	serijski	paralelno	ubrzanje
2	233.08	116.41	2.002
4	233.08	59.39	3.925
6	233.08	42.17	5.527
8	233.08	31.95	7.295
11	233.08	27.25	8.553

Graf: 200x200			
tredovi	serijski	paralelno	ubrzanje
2	413.81	207.4	1.995
4	413.81	104.09	3.976
6	413.81	74.16	5.580
8	413.81	57.9	7.147
11	413.81	41.07	10.076

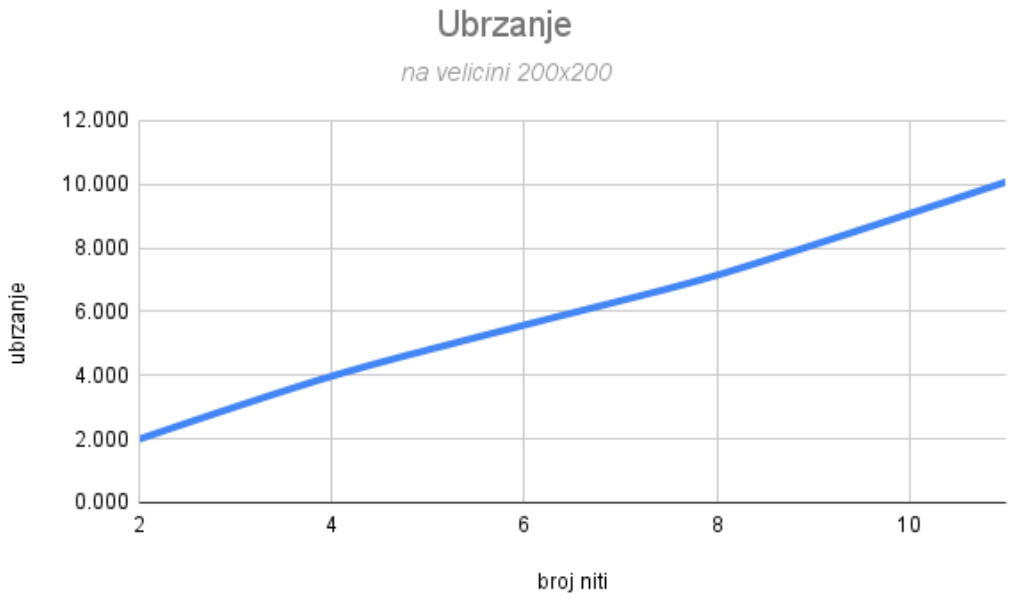
Graf: 500x500			
tredovi	serijski	paralelno	ubrzanje
2	2584.11	1291.45	2.001
4	2584.11	646.5	3.997
6	2584.11	458.04	5.642
8	2584.11	390.26	6.622
11	2584.11	265.397	9.737

Analiza rezultata

Posmatranjem rezultata možemo zaključiti da funkcija koja je zavisna od vremena izvršenja i veličine ulaznih podataka značajno sporije raste u slučaju

paralelnog izvršenja. Stoga možemo zaključiti da je paralelizacija bila uspešna. Na grafiku je prikazano vreme izvršavanja kada se paralelno pokrenu 4 tbb zadatka.

Da bismo bolje prikazali stepen paralelizma možemo prikazati sledeći grafik.



Ovde možemo da vidimo da ubrzanje linearno zavisi od broja niti odnosno zadataka koji se paralelno izvršavaju. Na maksimalnih 11 zadataka i veličini od 200x200 piksela program ima ubrzanje 10.076 što je prilično impresivno.