BFS i DFS Obilazak Grafa (Paralelizacija)

Projektni izvještaj

Paralelno programiranje

Milan Cvijović, 6/24

Univerzitet Crne Gore

Prirodno Matematički Fakultet

Računarstvo I Informacione Tehnologije – Master

Uvod

U prethodnom dijelu projekta, odnosno seminarskom radu obradili smo temu paralelizacije BFS i DFS algoritama za obilazak grafa. Dali smo početne ideje za implementaciju, a u ovom dijelu ćemo pokušati i da ih implementiramo u praksi. U nastavku ćemo obrađivati ideje i programske kodove ovih paralelizovanih algoritama.

BFS algoritam

Ideja

Prvo ćemo da se dotaknemo same ideje šta i kako je moguće uraditi. U seminarskom radu smo dali predlog da se paralelizuju for petlje klasičnog BFS algoritma, odnosno da se proširi postojeći kod. Međutim, kod prvobitnog pristupa smo imali problem kod pristupanja dijeljenim strukturama koje algoritam koristi. Potrebno je bilo zaključati ih u toku čitanja i pisanja, pa je paralelizacija gubila smisao. Nakon testiranja lako zaključujemo da takav pristup nije optimalan, pa tražimo drugo rješenje. U nastavku ćemo govoriti o novoj implementaciji.

Implementacija

Kao što smo već rekli, odlučili smo se za novu implementaciju paralelnog algoritma. Prvo ćemo dati kratko objašnjenje ideje, a zatim detaljan opis programskog koda. Ukratko, naš program se sastoji iz nekoliko ključnih koraka:

- Procesiranje svakog BFS nivoa paralelno koristeći više thread-ova
- Svako thread radi na dijelu trenutnog nivoa
- Da bi se izbjeglo konflikti, čuvaju se čvorovi sledećeg nivoa u privatnim listama
- Kada svi thread-ovi završe posao, liste se spajaju u glavni red

Na ovaj način prevazilazimo probleme koje smo imali kod prethodne implementacije. Iako su koraci bili jednostavniji, potrebni su bili zaključavanje i sinhronizacija, što sada nije slučaj.

Sada ćemo dati detaljniji uvid u dijelove programskog koda. Za početak možemo reći da je program implementiran tako da sadrži i sekvencijalnu i paralelnu verziju algoritma, dio za generisanje grafova za testiranje i samo testiranje performansi i ispis rezultata, ali tim ćemo se baviti kasnije. U nastavku su dijelovi koda ispraćeni objašnjenjima.

```
void parallelBFS(int start) {
        vector<bool> visited(V, false);
        queue<int> q;
        q.push(start);
        visited[start] = true;
```

Za početak inicijalizujemo funkciju i osnovne varijable koje koristimo. Vektor visited prati posjećivanje čvorova da bi se izbjeglo ponavljanje, q je standardni BFS red (queue) koji drži čvorove koji se obrađuju. Kreće se od datog čvora – start koji se postavlja kao prvi posjećen.

```
while (!q.empty()) {
	int level_size = q.size();
	vector<int> current_level;
```

Sledeći korak je priprema za obradu jednog nivoa. U okviru glavne petlje level_size predstavlja broj čvorova na istom BFS nivou (na istoj dubini). U current_level vektor se skladište čvorovi koje obrađujemo paralelno.

```
for (int i = 0; i < level_size; i++) {
	int node = q.front();
	q.pop();
	current_level.push_back(node);
}
```

U ovom koraku treba da skupimo sve čvorove sa jednog nivoa dubine i iz reda q ih prebacimo u current_level.

```
vector<vector<int>> next_level_threads(omp_get_max_threads());
```

Zatim pravimo vektore za svaki thread u koje smještamo čvorove koje otkrije. Ovako se izbjegava da više thread-ova pokušava da piše u isti vektor istovremeno. Ovo je zapravo vektor koji sadrži vektore, na unutrašnjem nivou su čvorovi koji pripadaju jednom thread-u, a spoljašnji nivo je lista tih vektora.

```
#pragma omp parallel
{
	int tid = omp_get_thread_num();
```

Otvaramo paralelni region i svakom thread-u se dodjeljuje ID (tid).

```
for (int i = tid; i < current_level.size(); i += omp_get_num_threads()) {
	int node = current_level[i];
```

Sada se čvorovi procesiraju na osnovu ID-ja koji je prethodno određen, odnosno svaki n-ti čvor dobija svaki n-ti thread. Na primjer thread 0 dobija čvorove 0, N, 2N, itd. Ovako se balansiraju podaci sa kojima se radi bez kritičnih sekcija i sličnih operacija. Izostavićemo dio koda na koji ćemo se vratiti kasnije.

```
for (int neighbor : adj[node]) {
	if (!visited[neighbor]) {
		visited[neighbor] = true;
		next_level_threads[tid].push_back(neighbor);
	}
}
```

Dolazimo do posjećivanja susjeda. Svaki thread posjećuje susjede čvora koji mi je dodjeljen. Ako susjed nije posjećen označimo da je sada posjećen. Dodajemo ga na lokalnu listu tog thread-a.

```
for (int tid = 0; tid < omp_get_max_threads(); ++tid) {
	for (int node : next_level_threads[tid]) {
		q.push(node);
	}
}
```

U krajnjem dijelu ove funkcije, kada se paralelni region završi, sve lokalne liste se spajaju u globalni BFS red q. Ovo podešava red za sledeći BFS nivo. Pošto se ova operacija dešava izvan paralelnog regiona nema opasnosti od pogrešnog upisivanja podataka u slučaju data race-a. Sada možemo preći na analizu.

Analiza

Šta se zapravo postiže ovom funkcijom? Prvenstveno simuliramo BFS obradom jednog po jednog nivoa. Svaki nivo se obrađuje paralelno, koristeći nezavisne thread-ove i sinhronizacija se može izbjeći ovakvom implementacijom.

Na osnovu datog algoritma sproveli smo i testiranje rezultata, jer nam je potreban pokazatelj uspješnosti našeg zadatka. U main() funkciji našeg programa imamo nekoliko načina za generisanje različitih velikih grafova, a zatim pokretanje algoritma nad istim i utvrđivanje vremena utrošenog na izvršavanje kako paralelnog, tako i sekvencijalnog algoritma. Tako možemo izvršiti i poređenje efikasnosti dva algoritma i preispitati smisao ove implementacije.

Testiranje vršimo na velikom binarnom stablu i na velikom grid grafu. Veličine grafova možemo da podešavamo. Zbog ograničenih resursa računara zadržaćemo testove na nivou do milion čvorova jer je dovoljno, ali testovi su bili uspješni i na nivou do 100 miliona čvorova.

```
// Large binary tree for benchmark
Graph g(1000000);
for (int i = 0; 2 * i + 2 < 1000000; i++) {
	g.addEdge(i, 2 * i + 1);
	g.addEdge(i, 2 * i + 2);
}
// Large grid for benchmark
int rows = 1000;
int cols = 1000;
Graph g(rows * cols);

for (int r = 0; r < rows; ++r) {
	for (int c = 0; c < cols; ++c) {
		int node = r * cols + c;
		if (c < cols - 1) g.addEdge(node, node + 1);
		if (r < rows - 1) g.addEdge(node, node + cols);
	}
}
```

```
// Sequential BFS Timing
cout << "Sequential BFS: ";
double start_seq = omp_get_wtime();
g.sequentialBFS(0);
double end_seq = omp_get_wtime();
cout << "\nSequential BFS Time: " << (end_seq - start_seq) << " seconds\n";

// Parallel BFS Timing
cout << "\nParallel BFS: ";
double start_par = omp_get_wtime();
g.parallelBFS(0);
double end_par = omp_get_wtime();
cout << "\nParallel BFS Time: " << (end_par - start_par) << " seconds\n";

return 0;
```

Program je testiran na računaru sa Ryzen 5 4600H procesorom sa 6 jezgara i 12 thread-ova. Prvo testiranje pokrećemo na generisanom binarnom stablu sa milion čvorova, a zatim na generisanom grid grafu sa istim brojem čvorova. Rezultati su sledeći:

```
Sequential BFS:
Sequential BFS Time: 0.0699999 seconds

Parallel BFS:
Parallel BFS Time: 0.086 seconds
```

Slika 1

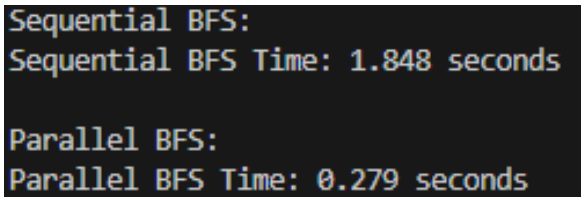
```
Sequential BFS:
Sequential BFS Time: 0.144 seconds

Parallel BFS:
Parallel BFS Time: 0.185 seconds
```

Slika 2

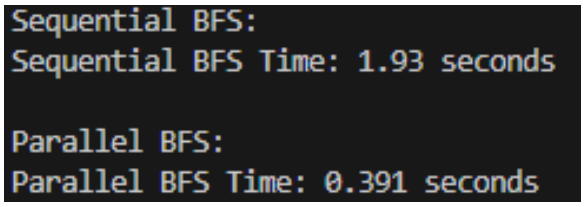
Na slikama vidimo rezultate testova redom za binarno stablo i grid graf. Ovakvi rezultati možda nisu očekivani jer vidimo blago sporije izvršavanje paralelnog algoritma, što dovodi u pitanje smisao paralelizacije na ovakav način. Da se zaključiti da troškovi pripreme i izvršavanja svih komandi u paralelnom regionu prevazilaze troškove izvršavanja sekvencijalnog algoritma ako imamo jednostavni obilazak. Zato ćemo se vratiti na dio koda koji smo ranije preskočili.

```
// Dummy code to simulate work
int dummy = 0;
for (int i = 0; i < 1000; ++i) {
      dummy += i * i;
}
// End of dummy code
```

Algoritam smo u osnovi testirali samo na jednostavnom obilaženju i posjećivanju čvorova. Ako u obje verzije dodamo ovaj kratki tzv. „dummy“ kod, možemo da napravimo lažni dodatni posao pri posjećivanju svakog čvora. Ovako simuliramo nešto realniji scenario gdje je potrebno ne samo posjetiti čvorove grafa, već ih i obrađivati kroz neke skuplje operacije izračunavanja. Sada opet testiramo obje verzije algoritma.



Slika 3



Slika 4

Na slikama su opet rezultati testiranja algoritama nad istim grafovima, ali ovaj put uz dodatni posao pri posjećivanju svakog čvora. U ovom slučaju vidimo značajno unapređenje brzine izvršavanja kod paralelnog algoritma kod oba grafa. Takođe, možemo primjetiti i da se brzine razlikuju i u zavisnosti od tipa grafa, gdje je u svim slučajevima izvršavanje brže ako radimo sa stablom.

Nakon dobijenih novih rezultata, možemo zaključiti da paralelizacija vjerovatno i nema smisla u koliko će se koristiti samo za jednostavno obilaženje grafa. Međutim, ako postoji dodatni posao pri posjećivanju čvorova, na primjer obrađivanje njihove vrijednosti i slično, može se postići značajno unapređenje efikasnosti. Ukratko, ukoliko radimo jednostavno obilaženje ili I/O operacije, često je pogodniji sekvencijalni pristup, dok je za skuplje operacije izračunavanja bolji paralelni pristup. Sada možemo preći i na DFS algoritam.

DFS algoritam

Ideja

Kao što smo ranije i napomenuli, paralelizacija DFS algoritma je teži zadatak u odnosu na BFS iz više razloga. Koriste se različite strukture podataka, postoji više načina implementacije, itd. Kao i kod BFS, opet smo nakon detaljnijeg rada izmijenili konačni implementaciju u odnosu na početnu ideju. Fokus je primarno postavljen na iterativnu implementaciju, zbog problema sa rekurzijama i paralelizacijom.

Implementacija

Možemo krenuti sa detaljima implementacije. Ukratko, osnovna ideja je sledeća:

- Koristi se dijeljeni stack koji čuva čvorove koji se posjećuju

- Svaki thread p okušava da uzme čvor sa dijeljenog stack-a
- Koristi atomic<bool> vektor da bezbjedno provjerava i označava čvorove koje posjeti
- Vraća neposjećene susjede nazad u dijeljeni stack koristeći kritični region

Cilj je implementirati program koji će imati bolje performanse u odnosu na sekvencijalni algoritam. Kao i ranije, krenućemo sa analizom programskog koda, korak po korak.

```
void parallelDFS(int start) {
    vector<atomic<bool>> visited(V);
    for (int i = 0; i < V; ++i) visited[i] = false;
```

Krećemo od funkcije parallelDFS od startnog čvora. Pripremamo vektor visited veličine grafa. Postavljamo sve na false, a vektor je atomic<bool> kako bi se osiguralo bezbjedno pristupanje podacima.

```
stack<int> shared_stack;
shared_stack.push(start);
```

Inicijalizujemo stack koji dijele svi thread-ovi koji sadrži čvorove koji treba da budu posjećeni. Startni čvor se postavlja na stack za početak obilaska.

```
#pragma omp parallel
    {
        while (true) {
        int node = -1;
```

Otvaramo paralelni region i glavnu petlju, ukoliko nema dostupnih čvorova postavljamo čvor na -1.

```
#pragma omp critical
{
    if (!shared_stack.empty()) {
        node = shared_stack.top();
        shared_stack.pop();
    }
}
if (node == -1) break;
```

Ulazimo u kritičnu sekciju koja je potrebna da bi pristup diijeljenom stack-u bio bezbjedan. Zaključavamo pristup tako da samo jedan thread može u trenutku da uzima čvor sa stack-a. Ako je stack prazan, thread ne radi ništa. Ako nismo dobili novi čvor, odnosno stack je prazan, posao se završava i prekida se petlja.

```
bool expected = false;
    if (!visited[node].compare_exchange_strong(expected, true)) {
    continue;
}
```

Provjeravamo da li je čvor već posjećen, ako je drugi thread već posjetio čvor, provjera neće proći i čvor se preskače. Na ovaj način osiguravamo da isti čvor ne posjeti više thread-ova.

```
for (auto it = adj[node].rbegin(); it != adj[node].rend(); ++it) {
    if (!visited[*it]) {
        #pragma omp critical
        shared_stack.push(*it);
    }
}
```

Petlja prolazi kroz susjede trenutnog čvora u obrnutom redosledu, kako bi se simulirao klasični DFS pristup. Samo neposjećeni čvorovi se postavljaju na stack. Takođe koristimo kritičnu sekciju da bezbjedno upisujemo u stack.

Analiza

Opet kao i za prethodni algoritam dolazimo do analize. Nakon implementiranja programa potrebno je utvrditi performanse paralelizovanog algoritma. Ponovo generišemo velike grafove i vršimo testiranje i sekvencijalnog i paralelnog algoritma na njima. Ovaj dio programskog koda je skoro identičan kao i za BFS pa ga nećemo ponavljati. Prvo sprovodimo testiranje nad velikim binarnim stablom i grid grafom uz prosti obilazak.

```
Sequential DFS:
Sequential DFS Time: 0.148 seconds

Parallel DFS:
Parallel DFS Time: 0.648 seconds
```

Slika 5

```
Sequential DFS:
Sequential DFS Time: 0.237 seconds

Parallel DFS:
Parallel DFS Time: 1.084 seconds
```

Slika 6

Na prethodnim slikama su redom dati rezultati testiranja na binarnom stablu i grid grafu od milion čvorova uz prosti obilazak. Vidimo da rezultati paralelnog algoritma nisu zadovoljavajući s obzirom da je algoritam mnogo sporiji u odnosu na sekvencijalni. Ovo se dešava kao što smo već i napomenuli zbog teže implementacije paralelnog DFS algoritma, jer je potrebno i često zaključavanje i sinhronizacija.

Međutim, pokušaćemo još jedan test kao što smo uradili i za BFS. Dodajemo kratki lažni kod koji simulira operacije obrade čvorova kako bismo vidjeli rezultat u nešto realnijoj primjeni.

```
int dummy = 0;
for (int i = 0; i < 1000; ++i){
    dummy += i * i;
}
```

```
Sequential DFS:
Sequential DFS Time: 1.927 seconds

Parallel DFS:
Parallel DFS Time: 0.797 seconds
```

Slika 7

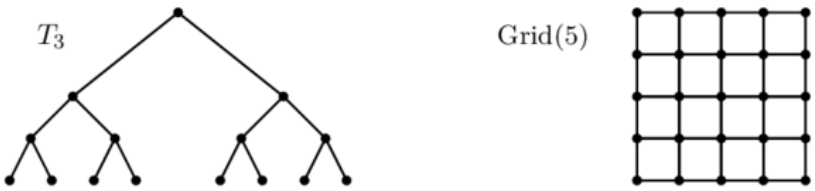
```
Sequential DFS:
Sequential DFS Time: 2.026 seconds

Parallel DFS:
Parallel DFS Time: 1.116 seconds
```

Slika 8

Sada vidimo rezultate nakon novog testiranja. Primjećujemo da uz dodatne operacije izračunavanja prilikom posjećivanja čvorova vrijeme izvršavanja sekvencijalnog algoritma značajno raste, dok kod paralelnog imamo samo blagi porast. Štaviše, razlika u vremenu izvršavanja je sada značajno promijenjena u korist paralelnog algoritma. I u ovom slučaju možemo primijetiti da paralelizacija algoritma dobija smisao samo u slučaju stvarnog posla prilikom posjećivanja grafova. Možemo zaključiti da je sekvencijalni pristup opet bolji za jednostavni obilazak i I/O operacije, dok je za kompleksnije i skuplje operacije izračunavanja znatno bolji paralelni algoritam.

Kompletne funkcije

Ovdje ćemo dati pregled oblika testiranih grafova i kompletnog koda funkcija koje smo koristili, a kompletan programski kod je dostupan u pratećim .cpp fajlovima.



Slika 9

Na slici je predstavljen osnovni oblik binarnog stabla i grid grafa kakvi su generisani za testiranje algoritama, a u nastavku su kompletne funkciije parallelBFS() i parallelDFS().

```
void parallelBFS(int start) {
        vector<bool> visited(V, false);
        queue<int> q;
        q.push(start);
        visited[start] = true;

        while (!q.empty()) {
            int level_size = q.size();
            vector<int> current_level;
            // Collect nodes for current level
            for (int i = 0; i < level_size; i++) {
                int node = q.front();
                q.pop();
                current_level.push_back(node);
            }
            // Thread-local queues for next level
            vector<vector<int>> next_level_threads(omp_get_max_threads());
            // Parallel processing of neighbors
            #pragma omp parallel
            {
                int tid = omp_get_thread_num();  // Get thread ID

                for (int i = tid; i < current_level.size(); i += omp_get_num_threads()) {
                    int node = current_level[i];
                    // Dummy code to simulate work
                    int dummy = 0;
                    for (int i = 0; i < 1000; ++i) {
                        dummy += i * i;
                    }
                    // End of dummy code
                    // Check each neighbor
                    for (int neighbor : adj[node]) {
                        if (!visited[neighbor]) {
                            visited[neighbor] = true;
                            next_level_threads[tid].push_back(neighbor);  // Add to thread-
local list
                        }
                    }
                }
            }
            // Merge thread-local next level lists into the main queue
            for (int tid = 0; tid < omp_get_max_threads(); ++tid) {
                for (int node : next_level_threads[tid]) {
                    q.push(node);
                }
            }
        }
    }
```

```
void parallelDFS(int start) {
        vector<atomic<bool>> visited(V);
        for (int i = 0; i < V; ++i) visited[i] = false;

        stack<int> shared_stack;
        shared_stack.push(start);

        #pragma omp parallel
        {
            while (true) {
                int node = -1;

                // Safely pop from shared stack
                #pragma omp critical
                {
                    if (!shared_stack.empty()) {
                        node = shared_stack.top();
                        shared_stack.pop();
                    }
                }

                if (node == -1) break;

                bool expected = false;
                if (!visited[node].compare_exchange_strong(expected, true)) {
                    continue;
                }

                // Begin dummy code
                int dummy = 0;
                for (int i = 0; i < 1000; ++i){
                    dummy += i * i;
                }
                // End of dummy code
                // Push neighbors in reverse order (DFS style)
                for (auto it = adj[node].rbegin(); it != adj[node].rend(); ++it) {
                    if (!visited[*it]) {
                        #pragma omp critical
                        shared_stack.push(*it);
                    }
                }
            }
        }
    }
```

Uputstvo za pokretanje programa

Kompletan programski kod se nalazi u fajlovima bfs_par.cpp i dfs_par.cpp u odgovarajućem repozitorijumu.

Kompajliranje programa se vrši sledećim komandama u terminalu odgovarajućeg foldera:

g++ -fopenmp bfs_par.cpp -o bfs_par

g++ -fopenmp dfs_par.cpp -o dfs_par

Nakon uspješnog kompajliranja programi se pokreću takođe kroz terminal:

./bfs_par

./dfs_par

Veličinu grafova na kojima se testira program podesiti u main funkciji.

Sadržaj