Paralelni algoritmi na grafovima

Horvat, Jurica

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Jurica Horvat

PARALELNI ALGORITMI NA GRAFOVIMA

Diplomski rad

Voditelj rada:
prof. dr. sc. Mladen Jurak

Zagreb, srpanj, 2023.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Zahvaljujem se svom mentoru prof. dr. sc. Mladenu Juraku na strpljenju i korisnim savjetima prilikom izrade ovog rada.
Ovaj rad posvećujem svojoj obitelji uz veliku zahvalnost na podršci i motivaciji tijekom studiranja.

Sadržaj

Uvod

Matematički model grafa izuzetno je koristan alat za reprezentaciju odnosa između objekata. Primjerice, njime možemo opisati prijateljstva među ljudima na društvenim mrežama, gradove na karti i njihove udaljenosti. Grafovima možemo planirati projekte: rastavimo ih na zadatke, gdje svaki zadatak može započeti nakon što je neki skup zadataka već obavljen te svaki zadatak ima procjenu duljine trajanja. U računarskoj znanosti često grafom reprezentiramo razne automate ili Turingove strojeve.

Obično grafove koristimo da bismo na njih primijenili određene algoritme i tako doznali neke korisne informacije iz cijelog skupa objekata koji promatramo. U navedenim primjerima, to može biti najkraći put između dva grada ili najkraće moguće vrijeme u kojem projekt može biti izveden u cijelosti.

Naravno, od interesa su nam efikasni algoritmi po pitanju vremenske, ali i prostorne složenosti. Postoje različiti pristupi optimizaciji algoritama, a jedan od njih je paralelizacija. Njome nastojimo izvršiti neki algoritam koristeći raspodjelu posla na više procesnih elemenata i time poboljšati performanse danog algoritma. S obzirom da za reprezentaciju grafa trebamo memoriju, odmah se postavlja pitanje: hoće li procesni elementi dijeliti memoriju u kojoj je graf ili će komunicirati mrežom i na taj način razmjenjivati informacije o grafu. Oba pristupa se koriste u praksi, a u ovom radu se bavimo paralelnim algoritmima na sustavima s dijeljenom memorijom.

Najprije ističemo najbitnija svojstva takvih sustava i fenomene koji se javljaju kod programiranja u takvim okolnostima. Nakon toga navodimo mehanizme koje nudi moderni C++ za rješavanje tih fenomena. U središnjem dijelu rada odabiremo tri paralelna algoritma na grafovima i najprije opisujemo njihove sekvencijalne verzije, zatim korak po korak uvodimo paralelizaciju te ih u konačnici implementiramo u jeziku C++, u skladu sa standardom C++20. Na kraju ćemo testirati algoritme na specificiranim ulaznim podacima i diskutirati o njihovim performansama.

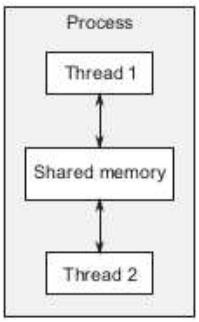
Poglavlje 1

Osnove programiranja sustava s dijeljenom memorijom

U ovom poglavlju ističemo osnovne koncepte vezane uz programiranje sustava s dijeljenom memorijom. Najprije opisujemo kako izgledaju takvi sustavi, a zatim česte izazove s kojima se susreću programeri u radu s ovakvim sustavima. Prvi pojam koji se veže uz paralelne sustave s dijeljenom memorijom je dretva (eng. *thread*).

1.1 Višedretvenost

Programi kakve ćemo pisati implementirajući paralelne algoritme u ovom radu sastojat će se od jednog *procesa* (eng. *process*). Taj proces će činiti barem jedna, a općenito više dretvi i imati će zajedničku memoriju za sve postojeće dretve, ali izoliranu od ostalih procesa koji se u tom trenutku izvode na računalu. *Dretva* ili *programska nit* (eng. *thread*) je niz instrukcija koji se izvodi u nekom procesu.



Slika 1.1: Shema sustava s dijeljenom memorijom

Na slici 1.1 vidimo skicu jednog procesa koji se sastoji od dvije dretve koje se izvršavaju istovremeno, uz uvjet da to dopuštaju sklopovlje i operacijski sustav. Jasno, potrebne su barem dvije procesne jedinice, tj. jezgre. Ukoliko to nije slučaj, dretve će se izvršavati naizmjence i to prema režimu kojeg implementira operacijski sustav.

Na prvi pogled, dijeljena memorija donosi očitu prednost: dretve ne trebaju komunicirati međusobno u svrhu razmjene podataka (varijabli), već ih samo čitaju i mijenjaju u zajedničkoj memoriji. No, što ako različite dretve nastoje u istom trenutku promijeniti vrijednost neke varijable? Koju vrijednost varijabla treba konačno zadržati? Ispada da programiranje u ovakvim okolnostima iziskuje neke posebne vještine programera - u vidu implementacije, testiranja, ali i dobre intuicije kod smišljanja algoritama.

1.2 Problemi s dijeljenom memorijom

Jedina jednostavna situacija u smislu dijeljene memorije u višedretvenom programiranju jest kada sve dretve imaju namjeru samo čitati iz memorije. Nije kompliciranа ni situacija kada imamo pisanje i čitanje uz vrlo specifične uvjete (npr. postoje pisanja u varijable, ali ni jedna dretva ne čita iz varijable koju je neka druga dretva mijenjala). Ali, u scenariju ”dretva A piše u varijablu x, dretva B čita vrijednost varijable x” imamo probleme s konzistencijom. Naime, u slučaju istovremenog zahtjeva za pisanjem od A te čitanjem od dretve B, možemo dobiti nekonzistentno stanje na način da dretva B pročita staru vrijednost od x.

Ovaj fenomen je poznat pod nazivom *stanje natjecanja* (eng. *data race*). Općenito, stanje natjecanja je svaka situacija u programiranju s dijeljenom memorijom u kojoj konačni ishod ovisi o relativnom poretku izvršavanja operacija dviju ili više dretvi. Napomenimo da stanje natjecanja nije nužno loša stvar za naše algoritme. Primjerice, imamo li u algoritmu neki globalni red na koji stavljamo neke podatke za obradu i na njega dodajemo elemente iz različitih dretvi, stanje natjecanja ne mora naštetiti konačnom ishodu. [4] Ovakvo benigno stanje natjecanja ćemo pronaći u BFS algoritmu za traženje najkraćeg puta kojeg obrađujemo kasnije.

1.3 Izbjegavanje stanja natjecanja

U nastavku navodimo neke od programerskih tehnika koje se koriste za izbjegavanje stanja natjecanja. Istaknimo za početak suštinu problema, odnosno razlog postojanja ovog fenomena. Uzmimo u tu svrhu precizniju verziju gornje situacije s promjenama varijabli. Recimo da dretva A želi inkrementirati cjelobrojnu varijablu x za 1 te u istom trenutku dretva B želi dohvatiti vrijednost varijable x. Problem je što operacija pisanja dretve A nije *atomarna*. To znači da je za njeno izvršavanje potrebno više instrukcija na nižoj razini. Konkretno, potrebno je iz memorije dohvatiti i spremiti vrijednost varijable x u procesorski

registar, izvršiti inkrement i na kraju spremiti dobivenu vrijednost u memoriju. Dretva B je mogla pročitati vrijednost varijable x u bilo kojem trenutku izvršavanja tih triju atomarnih procesorskih instrukcija. Idelano bi bilo da je pročitala na kraju posljednje instrukcije, ali to ne možemo garantirati bez dodatnih programerskih mehanizama.

Jedna opcija za rješavanje stanja natjecanja jest dizajniranje strukutre podataka na način da su modifikacije ostvarene kao nizovi nedjeljivih promjena, gdje svaka promjena čuva *invarijante strukture*. Invarijanta strukture je tvrdnja strukture koja je uvijek istinita. Na primjer, "ova varijabla sadrži broj elemenata u listi" je invarijanta strukture. Vrlo je prirodno za očekivati da se takve invarijante narušavaju kod kompliciranijih radnji nad strukturom. Stoga taj pristup rješavanja stanja natjecanja iziskuje dobro poznavanje memorijskog modela jezika u kojem programer radi kao i iskustvo programera. Ovakav pristup je poznat pod nazivom *programiranje bez zaključavanja* (eng. *lock-free programming*)

Sljedeći mehanizam koji opisujemo poznat je pod nazivom *software transactional memory*. Radi se o transkacijama kakve nalazimo i u radu s bazama podataka. Traženi niz modifikacija podataka i čitanja se sprema u transkacijske zapise i tada se izvrši čitav zapis odjednom. Ukoliko transakcija nije uspjela (npr. neka dretva već koristi strukturu koju trebamo u transakciji), ne izvršava se ni jedna instrukcija sa zapisa, već postupak kreće ispočetka.

Kritičnim dijelom koda smatramo sva čitanja ili pisanja povezana s varijablama odnosno strukturama kojima pristupa više dretvi.
Posljednji mehanizam koji opisujemo je takozvani *mutex*, skraćenica od *mutually exclusive* (hrv. *međusobno isključivo*). To je objekt koji nam osigurava da se pojedini dio koda izvršava od strane najviše jedne dretve. Objekt podržava dvije osnovne operacije: zaključavanje (eng. *lock*) i otključavanje (eng. *unlock*). Jasno, dretva koja uspije zaključati mutex, će ući u kritični dio koda. Ostale dretve neće moći ući u taj dio sve dok dretva koja je zaključala mutex ne otključa isti. [4]

Kako mutexom zaštititi neku varijablu ili općenito strukturu podataka? Imamo po jedan takav globalni objekt u programu za svaku strukturu podataka koja je dijeljena među više dretvi. U dijelovima koda gdje imamo pisanje ili čitanje vezano uz određenu strukturu, nastojimo zaključati odgovarajući mutex, potom izvršiti čitanje ili pisanje i otključati mutex po završetku. Kod pisanja, odnosno promjene neke strukture, na taj način osiguravamo da je samo dretva koja vrši tu promjenu svjesna međustanja u kojima se nalazi struktura.

1.4 Potpuni zastoj

Spomenimo još jedan bitan problem koji se javlja u sustavima s dijeljnom memorijom. Zapravo, on se javlja kao posljedica korištenja mutexa koje smo naveli kao mehanizam za sprječavanje stanja natjecanja. Do tog problema neće doći ukoliko imamo samo jedan mutex u cijelom programu - potrebna su barem dva. Zamislimo situaciju gdje imamo dvije

dretve: A i B te dva mutexa: M_1 i M_2. Recimo da je dretva A uspješno zaključala mutex M_1 i dretva B uspješno zaključala mutex M_2. Ako sada dretva A čeka na otključavanje mutexa M_2 da bi nastavila dalje, i istovremeno dretva B čeka na otključavanje od M_1 da bi nastavila dalje, imamo stanje u kojem ni jedna dretva neće napredovati. Takvo stanje nazivamo *potpuni zastoj* (eng. *deadlock*).

Koristan savjet za izbjegavanje ovakvih situcija je da ugnježđene mutexe uvijek zaključavamo u istom redoslijedu. Na taj način smo sigurno izbjegli potpuni zastoj. Ponekad je to jednostavno za primjeniti, a nekada gotovo nemoguće. Promotrimo sljedeću situaciju: imamo funkciju koja prima dvije instance neke klase i njen zadatak je da razmjeni vrijednosti nekih varijabli među tim instancama. Da bi ta funkcija radila ispravno u višedretvenom okruženju, moramo zaključati mutexe na obje instance klase. Primjerice, unutar funkcije to učinimo na način da najprije zaključamo mutex prvog argumenta funkcije, a zatim onaj drugi. No, time smo fiksirali poredak i riskiramo stanje potpunog zastoja - sve što je potrebno je da druga dretva pozove tu istu funkciju sa istim argumentima u drugom poretku. [4]

1.5 Sinkronizacija događaja

Prethodno smo razmatrali bitne programerske tehnike za zaštitu dijeljenih podataka kod programiranja u sustavima s više dretvi i dijeljenom memorijom. Sada navodimo još jedan bitan koncept u programiranju takvih programa. Motivirajmo ga najprije jednim vrlo praktičnim primjerom.

Zamislimo da se vozimo noćnim vlakom. Jedan način da se osiguramo da ćemo izaći na ciljanoj stanici je da ostanemo budni cijelu noć i pratimo stanice na koje vlak stiže. Drugi način je da provjerimo očekivano vrijeme dolaska vlaka na ciljnu stanicu, postavimo svoj alarm malo ranije od tog trenutka i odemo spavati. Taj način bi bio optimalniji za nas. No, ako je vlak u međuvremenu iz nekog razloga počeo kasniti, probudit ćemo se prijevremeno. Postoji mogućnost i da se baterija na mobitelu isprazni, alarm se ne oglasi i propustimo stanicu. Najbolji način bi bio kad bismo mogli otići na spavanje, a dolaskom na ciljnu stanicu bi nas netko probudio sa sigurnošću.

Vratimo se sada dretvama. Zamislimo jednostavnu situaciju s dvije dretve. Recimo da prva dretva čeka da druga dretva izvrši određeni zadatak. Tada prva dretva ima nekoliko opcija. Prvo, može neprestano prvojeravati vrijednost zastavice koja se čuva u zajedničkoj memoriji i zaštićena je mutexom. Druga dretva u tom slučaju postavi vrijednost te zastavice po završetku svojeg zadatka. Taj pristup je loš u dva pogleda. Prvo, dretva koja čeka da se postavi zastavica troši korisno procesorsko vrijeme neprestano provjeravajući vrijednost zastavice. Drugo, kada to radi, zaključava mutex dodijeljen toj zastavici pa dretva koja obavlja zadatak ne može postaviti vrijednost zastavice u trenutku kad je gotova sa zadatkom. Druga opcija je da dretva koja čeka bude na spavanju te periodički vrši provjeru

vrijednosti zastavice. Ovim pristupom smo riješili prvi problem: ne troši se korisno procesorsko vrijeme. Ipak, potencijalni problem se odmah javlja. Prekratak period spavanja bi mogao ipak trošiti korisno procesorsko vrijeme, a predugačak period može uzrokovati slabije performanse: dretva potencijalno spava dok je zadatak zapravo već obavljen i mogla bi krenuti dalje s radom. Ovakav pristup nikako neće biti dobar za algoritme koji trebaju raditi u realnom vremenu. Opcija koja je vrlo česta u modernim programskim jezicima se temelji na takozvanim *uvjetnim varijablama* (eng. *condition variables*). One funkcioniraju na principu da se dretve koje su zainteresirane za informaciju o završetku nekog događaja druge dretve pretplate na tu uvjetnu varijablu. Kada dretva koja obavlja zadatak završi, ona će obavijestiti o tome sve dretve koje su se pretplatile na odgovarajuću uvjetnu varijablu.

Još jedan koristan način za sinkronizaciju dretvi je korištenje *barijera* (eng. *barriers*). Barijera je sinkronizacijski mehanizam koji funkcionira na pricipu brojača. Imamo skup dretvi i svaka dretva može smanjiti u jednom ciklusu taj brojač za najviše 1. Kada brojač dostigne vrijednost 0, znamo da su sve dretve stigle do barijere i možemo nastaviti program s tom pretpostavkom. [4]

Poglavlje 2

Programsko sučelje jezika C++

U ovom poglavlju opisujemo sučelje jezika C++ prema standardu iz 2020. godine za programiranje sustava s dijeljenom memorijom. Pri tome navodimo realizacije svih programerskih mehanizama istaknutih u prethodnom poglavlju.

2.1 Dretve (programske niti)

Najprije navodimo klasu iz `std` biblioteke čiji instancirani objekti su u 1-1 korespondenciji s dretvama koje vidi operacijski sustav. To je klasa `std::thread` za čije korištenje trebamo uključiti zaglavlje <`thread`>. Pokažimo jednim jednostavnim primjerom kako se realizira višedretvenost u tom okruženju.

```
#include <iostream>
#include <thread>

void task(std::string name, int val){
  std::cout << "Radim zadatak s imenom: " << name <<
    " i argumentom " << val << std::endl;
}

int main(){
  std::thread t{task, "zadatak", 1}; // kreiranje pomocne programske
   niti
  std::cout << "ja sam u glavnoj dretvi" << std::endl;
  t.join();

  return 0;
}
```

Listing 2.1: objekt za dretvu

U kodu 2.1 vidimo jednostavni primjer kojim se demonstrira korištenje `std::thread` klase. Glavna (`main`) funkcija opisuje rad glavne dretve u procesu koji nastaje pokretanjem programa. U toj glavnoj dretvi napravili smo jednu pomoćnu dretvu čiji rad je opisan funkcijom `task`, te argumentima `name` i `val`. Prilikom same konstrukcije dretve (linija 10), ona se automatski i pokreće. Puno je zanimljivije pitanje prestanka njenog rada. Tipično želimo da glavna dretva završi posljednja i da su kod njenog prekida već završile i ostale dretve. Takvo ponašanje ostvarujemo pozivom funkcije `join()` na objektu koji je povezan s odgovarajućom dretvom. Ukoliko iz nekog razloga ne želimo pričekati da neka dretva završi prije završetka glavne dretve, možemo odmah nakon konstrukcije objekta odgovornog za dretvu (10. linija) dodati naredbu `t.detach()`. Ta naredba zapravo oslobađa dretvu kakvu konstruira operacijski sustav od našeg objekta u programu i nad njom više nemamo kontrolu. U tom slučaju nikako ne smijemo pozivati više `join` na tom objektu, jer on više nema kontrolu nad tom dretvom. Korištenje `detach` metode može biti opasno i biti uzrok raznih vrsti iznimki operacijskog sustava. Primjerice, završi li glavna dretva svoje izvođenje prije pomoćne dretve i ta pomoćna dretva pokuša dohvatiti neku globalnu varijablu, ona će ili dohvatiti neočekivanu vrijednost iz memorije koja se već promijenila ili izazvati iznimku jezgre operacijskog sustava. Iz tog razloga se `detach` koristi samo u vrlo specifičnim okolnostima.

Samim pozivom `t.join()` još uvijek nismo osigurali da će se ta naredba i izvršiti. To se može desiti u raznim situacijama kada se napusti doseg u kojem je prisutan poziv `join` naredbe. Primjerice, u slučaju da dretva `t` pozove `std::terminate` kod dohvata neke iznimke koja nije obrađena, pozvat će se `std::abort` i program neće stići izvršiti `join`. U tom slučaju opet imamo neočekivano ponašanje - kao da nismo ni koristili `join`. Taj problem možemo riješiti mehanizmom dohvata i obrade iznimki. U kodu 2.2 je prikazan primjer.

```
1  #include <iostream>
2  #include <thread>
3
4  void task(std::string name, int val) {
5    std::cout << "Radim zadatak s imenom: " << name <<
6      " i argumentom " << val << std::endl;
7  }
8
9  int main() {
10   std::thread t{task, "zadatak", 0}; // kreiranje pomocne programske
       niti
11   std::cout << "ja sam u glavnoj dretvi" << std::endl;
12   try{
13
14   }catch(...){
15     t.join();
16     throw;
```

```
17    }
18    t.join();
19    return 0;
20 }
```

Listing 2.2: obrada iznimke dretvi

Na prvi pogled, kod 2.2 izgleda kao rješenje situacije: u slučaju iznimke u dretvi koja izvršava funkciju `task`, obradit ćemo je u `catch` bloku i tom prilikom pozvati `t.join()`. U velikom broju pokretanja odnosno testiranja to će zaista raditi ispravno. Ipak, greška u ovom pristupu nažalost postoji. Da bismo je uspjeli uočiti moramo proučiti nižu razinu izvršavanja programa. U slučaju da sve prođe bez iznimke, program radi sigurno. Zamislimo sljedeći mogući scenarij: glavna dretva je kreirala pomoćnu koja obavlja neko složeno računanje. U nekom trenutku pomoćna dretva baca iznimku i upada u beskonačnu petlju. Tada kod obrade iznimke `t.join()` čeka da pomoćna dretva završi s radom, a to se nikada neće dogoditi zbog beskonačne petlje. Tada cijeli proces radi beskonačno dugo i moramo ga ručno uništiti korištenjem signala operacijskog sustava. Dakle, u samom programu ne možemo kontrolirati takve pojave, jer zaglavlje <`thread`> ne nudi takve mehanizme. Intuitivno, ispravni mehanizam bi bio da u slučaju dohvata iznimke u glavnoj dretvi uspostavimo neku vrstu komunikacije s pomoćnom dretvom i naredimo joj da završi. Srećom, u 2020. godini ne moramo to implemenirati sami. Novi standard jezika C++ nudi klasu `std::jthread` koja nudi upravo takav mehanizam. [5]

`std::jthread`

U imenu poboljšane klase `std::thread` dodano je samo slovo `j` na početak riječi, što nije slučajno. Slovo `j` na početku naglašava da se u destruktoru klase izvršava metoda `join`, ako je to moguće. Klasa zapravo u cijelosti poštuje *RAII* tehniku. *RAII* [1] je engleska kratica fraze *Resource Acquisition Is Initialization*, a značenje podrazumijeva enkapsulaciju svih resursa u klase, pri čemu vrijedi:

- konstrukor zauzima potrebnu memoriju i uspostavlja sve klasne invarijante ili baca iznimku ako to nije moguće,
- destruktor oslobađa memoriju i nikada ne baca iznimku.

Za upotrebu `std::jthread` dovoljno je opet uključiti samo zaglavlje <`thread`>. Štoviše, `std::jthread` nudi isto programsko sučelje kao i `std::thread`. To znači da u svojem potencijalno nesigurnom kodu programer može zamjeniti sve objekte tipa `std::thread` sa `std::jthread` i sve će raditi na sigurniji način.

Pokažimo sada primjerom kako riješiti problem koji smo prepoznali u kodu 2.2.

```
void task(std::stop_token st, std::string name, int val){
  while(!st.stop_requested()){
    ...
  }
}

int main() {
  std::jthread t{task, name, val};
  if(...){
    t.request_stop();
  }
  t.join();
  return 0;
}
```

Listing 2.3: rješenje sa `std::jthread`

U ovom pseudokodu uočavamo rješenje koje zapravo implementira intuiciju koju smo naveli malo prije: konstruiramo objekt tipa `std::jthread` predavši mu iste argumente kao i konstruktoru tipa `std::thread`. Argument koji trebamo dodati našoj funkciji `task` je `std::stop_token` pomoću kojeg ćemo dobiti signal ukoliko neka dretva (glavna u našem slučaju) zatraži prekid rada odgovarajuće dretve. U skladu s time moramo modificirati i kod funkcije `task`: primjerice, zatvorimo cijeli njen kod u jednu `while` petlju koja će u uvjetu samo pitati je li netko zatražio prekid rada ove dretve. S druge strane, u glavnoj dretvi možemo zatražiti prekid dretve pozivom metode `request_stop()` na odgovarajućem objektu. Postoji još jedan elegantan način reagiranja dretve na zahtjev za njenim prekidom.

```
void task(std::stop_token st, std::string name, int val){
  std::stop_callback cb{st, []{
    ...
  }};

  ...
}
```

Listing 2.4: rješenje sa `std::stop_callback`

U ovom slučaju, zahtjev za prekid rada dretve koja je opisana funkcijom `task` se obrađuje u lambda funkciji koju smo registrirali kao pozivajuću za `std::stop_token st`. Kod uništenja objekta `cb`, destruktor u skladu s RAII tehnikom odjavljuje pripadnu lambda funkciju koja se do tad pozivala na zahtjev za prekidom dretve pomoću `st`. Na taj način možemo registrirati više (različitih) lambda funkcija za obradu zahtjeva za prekidom dretve unutar funkcije `task`. [5]

Navedimo usput još jednu novost u standardu C++20 koja je vezana uz zaustavljanje dretvi pomoću komunikacije, ali ima i druge primjene. Riječ je o klasi `std::stop_source`. Njena svrha je da kreira zahtjeve za zaustavljanjem, primjerice objektima tipa

`std::jthread`. Zahtjev za zaustavljanjem kreiran od jednog `std::stop_source` objekta će biti vidljiv svim ostalim `std::stop_source` i `std::stop_token` objektima koji imaju isto asocirano stanje zaustavljanja. [2] U sljedećem primjeru pokazujemo kako se koriste takvi objekti.

```
#include <stop_token>
...

std::stop_source ssrc; // kreira dijeljeno stanje zaustavljanja
std::stop_token stok{ssrc.get_token()}; // kreira token za to stanje
    zaustavljanja
```

Listing 2.5: `std::stop_source`

Klasa `std::stop_source` sadrži svoj `std::stop_token` koji pruža API za zahtjeve za zaustavljanjem (`std::stop_requested`). Nije moguće napraviti `std::stop_token` objekt sa asociranim dijeljenim stanjem zaustavljanja na drugačiji način od onoga u 5. liniji u kodu 2.5 jer zadani konstruktor nema asocirano stanje zaustavljanja. Pokažimo za kraj koristan primjer koji pomoću `std::stop_source` objekta osigurava da sve dretve stanu na zahtjev jednog zajedničkog tokena.

```
#include <iostream>
#include <vector>
#include <thread>

int main(){
  std::vector<std::jthread> dretve;

  std::stop_source stopZaSve;
  std::stop_token tokenZaSve{stopZaSve.get_token()};

  for(int i = 0; i < 9; ++i){
    dretve.push_back(std::jthread{[](std::stop_token st){
        //...
        while (!st.stop_requested()){
        //...
        }
        },
        tokenZaSve //sada posaljemo token kao argument
        });
  }

  stopZaSve.request_stop();

  return 0;
}
```

Listing 2.6: `std::stop_source` za slanje zahtjeva nekom skupu dretvi

Bitno je naglasiti samo da u 18. liniji obavezno moramo poslati `tokenZaSve` kao argument lambda funkciji koju izvršava pojedini `std::jthread` objekt jer bi u protivnom taj sam `std::jthread` objekt stvorio svoj interni token i više dretve nebi imale zajedničko stanje zaustavljanja. U 22. liniji sada pozivom `stopZaSve.request_stop()` zahtjevamo da sve dretve stanu odjednom. Više detalja u vezi sa klasama `std::jthread`, `std::stop_source` i njihovim radom može se pronaći u [5] i [2].

2.2 Međusobno isključivanje

Sada opisujemo sučelje koje nudi C++ za mehanizam međusobnog isključivanja kojim smo se bavili u prošlom pogljavlju. Zaglavlje koje moramo dodati je <mutex>, a središnja klasa zaglavlja `std::mutex`.

```
1  #include <list>
2  #include <mutex>
3  #include <algorithm>
4
5  std::list<int> lista;
6  std::mutex M;
7
8  void dodaj_u_listu(int novi_element){
9    std::lock_guard<std::mutex> guard(M);
10   lista.push_back(novi_element);
11 }
12
13 bool prisutan_element(int vrijednost){
14   std::lock_guard<std::mutex> guard(M);
15   return std::find(lista.begin(), lista.end(),
16       vrijednost) != lista.end();
17 }
```

Listing 2.7: `std::mutex` za međusobno isključivanje

U kodu 2.7 vidimo jednostavnu upotrebu međusobnog isključivanja sa globalnom listom koju koristi više dretvi. `std::lock_guard` koji se pojavljuje u svakoj funkciji je omotač oko `std::mutex` objekta, a uloga mu je da podvrgne korištenje mutexa RAII principu koji smo već spomenuli prethodno. Preciznije, njegov konstruktor zaključava mutex `M`, a destruktor ga otključava. Vrijedno je spomenuti da zaglavlje <thread> nudi mehanizam za izbjegavanje mogućeg *potpunog zastoja* (eng. *deadlock*) kojeg smo opisali u prvom poglavlju. Radi se o funkciji `std::lock` koja prima proizvoljno mnogo mutexa i pritom ih zaključava na siguran način (izbjegava potpuni zastoj). Pokažimo primjerom njenu upotrebu.

```
1  friend void swap(X& lhs, X& rhs){
2    if(&lhs == &rhs)
```

```
3     return;
4   std::lock(lhs.m, rhs.m);
5   std::lock_guard<std::mutex> lock_a(lhs.m, std::adopt_lock);
6   std::lock_guard<std::mutex> lock_b(rhs.m, std::adopt_lock);
7   swap(lhs.val, rhs.val);
8 }
```

Listing 2.8: swap elemenata dva objekta pomoću `std::lock`

Funkcija `swap` najprije zaključa mutexe koji čuvaju objekte `lhs` i `rhs` koji su instance klase `X`. Zatim za takve zaključane mutexe napravi po jedan omotač tipa `std::lock_guard` uz dodatni argument `std::adopt_lock` koji će osigurati poziv konstruktora od `std::lock_guard` koji ne zaključava mutex po instancijaciji. [4]

2.3 Sinkronizacija dretvi

U ovoj sekciji promatramo sučelje koje nudi C++ za sinkronizaciju dretvi, tj. sinkronizaciju događaja koju smo spomenuli u prvom poglavlju.

`std::condition_variable`

`std::condition_variable` iz zaglavlja <condition_variable> je klasa za sinkronizaciju koja se primarno koristi zajedno s `std::mutex` objektom koji će blokirati jednu ili više dretvi dok neka druga dretva ne završi s modifikacijom dijeljene varijable i obavještavanjem objekta tipa `std::condition_variable` o tome. [2] Pokažimo sada primjerom kako pomoću ove klase realizirati čestu situaciju u programiranju s više dretvi: jedna dretva priprema podatke za obradu i stavlja ih u red, a druga skida podatke s reda i poziva funkciju za obradu na svakom od njih. [4]

```
1  std::mutex mut;
2  std::queue<data_chunk> red_s_podacima;
3  std::condition_variable cond_var;
4
5  void priprema_podataka(){
6    while(ima_podataka()){
7      data_chunk const data = pripremi_podatak();
8      {
9        std::lock_guard<std::mutex> lk(mut);
10       red_s_podacima.push(data);
11     }
12     cond_var.notify_one();
13   }
14 }
15
```

```
16 void dretve_za_obradu(){
17   while(true){
18     std::unique_lock<std::mutex> lk(mut);
19     cond_var.wait(lk, []{return !red_s_podacima.empty();});
20     data_chunk data = red_s_podacima.front();
21     red_s_podacima.pop();
22     lk.unlock();
23     obradi(data);
24     if(zadnji_podatak(data))
25       break;
26   }
27 }
```

Listing 2.9: sinkronizacija dretvi pomoću `std::condition_variable`

U 2. liniji stvorimo globalni `std::queue` koji će služiti za prijenos podataka za obradu među dretvama. Red s podacima će biti zaštićen mutexom `mut`. Kada dretva za pripremu podataka dohvati neki novi podatak za obradu, najprije zaključa `mut`. Zatim stavi taj podatak na kraj reda i otključa `mut` (RAII). Nakon toga poziva `std::contition_variable::notify_one` da bi obavijestila dretvu (jednu) koja je na čekanju uzrokovanom tom uvjetnom varijablom (ako takva postoji). S druge strane imamo dretvu koja je određena kodom funkcije `dretve_za_obradu`. Promotrimo rad jedne takve dretve. Najprije pomoću `std::unique_lock` zaključavamo `mut`. Nakon toga dretva poziva metodu `std::condition_variable::wait()` na objektu `cond_var`, kojoj prosljeđuje `lk` koji je omotač objekta `mut` kao i lambda funkciju koja izražava uvjet na čije ispunjenje se čeka da bi dretva nastavila s radom - red mora biti neprazan. Implementacija funkcije `wait` tada provjerava uvjet (dan prosljeđenom lambda funckijom). Ako je zadovoljen, dretva nastavlja s radom. Ako nije, funkcija `wait()` otključava `mut` i blokira dretvu. Kada je `cond_var` obavještena sa `notify_one` iz dretve koja priprema podatke, dretva za obradu se budi iz spavanja, zaključava mutex `mut` i ponovno provjerava uvjet iz lambda funkcije. Ako je on zadovoljen, dretva za obradu nastavlja s radom sa zaključanim mutexom `mut`. Ako uvjet nije zadovoljen, dretva za obradu otključava `mut` i nastavlja čekati (vraća se u red blokiranih dretvi). Kao omotač oko `mut` u funkciji za obradu podataka iz reda smo koristili `std::unique_lock` umjesto `std::lock_guard`. To je zato što dretva koja čeka na pojavu podataka u redu mora otključati i zaključati `mut` u više navrata (svakim pozivom `notify_one()` iz funkcije za pripremu podataka). Stoga omotač oko `mut` mora pružati tu fleksibilnost (funkcije `lock()` i `unlock()`). `std::lock_guard` je u tom smislu primitivna klasa jer ona točno jednom zaključa svoj mutex (u konstruktoru) i točno jednom ga otključa (u destruktoru). `std::unique_lock` je ovdje koristan i iz perspektive performansi: u 22. liniji otključavamo `mut` sa `unlock()`. Kada to ne bismo učinili, on bi se otključao tek na kraju dosega (kraj `while` petlje). To znači da bi red s podacima bio zaštićen za svo vrijeme obrade (funkcija `obradi`) i podaci bi morali čekati

dugo na ulazak u red u slučaju neke kompliciranije obrade, a za time uopće nema potrebe. [4]

Jednokratni događaji u budućnosti

Motivirajmo primjerom [4] iz stvarnog života sljedeću programersku tehniku u višedretvenom programiranju. Zamislimo da idemo na put avionom. Jednom kad dođemo do zračne luke i obavimo check-in, još uvijek moramo čekati na obavijest da je naš avion spreman za ukrcavanje, možda i nekoliko sati. U međuvremenu ćemo možda čitati neku knjigu, surfati internetom ili se najesti u skupom restoranu u sklopu aerodroma. U svakom slučaju, čekati ćemo na signal da je vrijeme da se ukrcamo u avion. I ne samo to: idući puta kada budemo putovali avionom, čekati ćemo drugi let, tj. na jedno ukrcavanje čekamo samo jednom.

Standardna biblioteka jezika C++ modelira ovakve vrste jednokratnih događaja pomoću predloška klase `std::future<>` iz zaglavlja `<future>`. Intuitivno, objekt tipa `std::future<>` reprezentira događaj koji će se dogoditi u budućnosti. Ako dretva treba pričekati za neki jednokratni događaj u budućnosti, ona čuva objekt tipa `std::future<>` koji se brine o tom događaju. On može sadržavati neke detalje o toj budućnosti (u primjeru: na kojim vratima će se odvijati ukrcavanje na naš avion), ali ne mora. Nadalje, dretva se može periodički konzultirati sa odgovarajućim `std::future` objektom da bi saznala ako se očekivani događaj dogodio ili još ne (u primjeru: pogled na ploču odlazaka) i to istovremeno s obavljanjem nekog drugog zadatka (u primjeru: obrok u restoranu). Alternativno, dretva može raditi neke zadatke bez da ispituje je li se očekivani događaj dogodio i tek kada joj za daljnje izvršavanje treba potvrda o izvršenju događaja, dretva stane i čeka sve dok se događaj ne desi.

Spomenimo još da u standardnoj biblioteci postoje dva različita `future` predložaka klase: `std::future<>` i `std::shared_future<>`. Razlika između njih je analogna razlici između `std::unique_ptr` i `std::shared_ptr` u jeziku C++. Instanca neke klase `std::future<>` je jedina instanca koja referira na fiksni događaj. S druge strane, više instanci tipa `std::shared_future` može referirati na isti događaj i tada sve instance koje referiraju na isti događaj u isto vrijeme saznaju da se događaj dogodio i mogu istovremeno pristupiti bilo kojem podatku vezanom uz taj događaj.

`std::async`

Zamislimo sada da imamo neki dugački izračun koji će proizvesti koristan rezultat za nastavak izvođenja dretve. Mogli bismo u tu svrhu stvoriti novi `std::jthread` objekt i izvršiti taj izračun, ali `std::jthread` ne nudi izravni mehanizam za povratak izračunate vrijednosti u dretvu iz koje je pokrenut. Tu će od koristi biti novi funkcijski predložak

`std::async<>`. Umjesto da pokrenemo novu dretvu sa `std::jthread` objektom, ovdje ćemo to učiniti sa funkcijom `std::async<>` koja će po konstrukciji nove dretve odmah vratiti `std::future` objekt. U trenutku kad nam zatreba vrijednost izračunavanja pomoćne dretve, pozvat ćemo `get()` funkciju na odgovarajućem objektu i dretva će biti blokirana sve dok taj `std::future` objekt ne postane spreman (kada izračun završi). U nastavku slijedi jednostavan primjer ovog pristupa.

```
#include <future>
#include <iostream>

int dugacak_izracun(){return 1;}
void radi_nesto_drugo(){}

int main(){
  std::future<int> rjesenje = std::async(dugacak_izracun);
  radi_nesto_drugo();
  std::cout << "Rjesenje je " << rjesenje.get() << std::endl;
  return 0;
}
```

Listing 2.10: čekanje na izračun pomoću `std::async`

Funkcijski predložak `std::async` dopušta pozivanje funkcije (kao nove dretve) s prosljeđivanjem argumenata na sličan način kao i `std::jthread` objekt. Ako je prvi argument u pozivu `std::async` pokazivač na funkciju koja je članica neke klase, tada sljedeći argument mora biti poveznica na objekt na kojem se izvršava ta funkcija: pokazivač na objekt, referenca ili obična varijabla. U nastavku slijedi primjer koji koristi takva prosljeđivanja argumenata.

```
#include <string>
#include <future>

struct X{
  void foo(int,std::string const&);
  std::string bar(std::string const&);
};

X x;
auto f1 = std::async(&X::foo, &x, 42, "hello");
auto f2 = std::async(&X::bar, x, "goodbye");

struct Y{
  double operator()(double);
};

Y y;
auto f3 = std::async(Y(), 3.141);
```

```
19 auto f4 = std::async(std::ref(y), 2.718);
```

Listing 2.11: argumenti za `std::async`

U 10. liniji će dretva izvršiti poziv `p->foo(42, "hello")`, pri čemu je `p = &x`. U 11. liniji će poziv biti `tmpx.bar("goodbye")`, pri čemu je `tmpx` kopija od `x`. U 18. liniji poziv je `tmpy(3.141)` pri čemu je `tmpy` dobiven konstruktorom premještanja iz `Y`. U 19. liniji poziv će biti `y(2.718)`.
Postoji još jedan dodatni argument koji može primiti funkcijski predložak `std::async<>`. On dolazi na prvo mjesto i tipa je `std::launch`. Ako ima vrijednost `std::launch::deferred`, dretva koja izvršava zadanu funkciju će započeti s radom tek kada se pozove `get()` ili `wait()` na `future` objektu kojeg je vratila `std::async`. U tom slučaju se kod izvršava sinkrono. Ukoliko na mjesto tog argumenta proslijedimo vrijednost `std::launch::async` tada naglašavamo da se funkcija mora izvršiti u zasebnoj dretvi. Ako pak taj argument izostavimo kod poziva `std::async<>`, on poprima zadanu vrijednost `std::launch::deferred | std::launch::async` koja znači da implementacija odlučuje sama o načinu izvršavanja.

`std::packaged_task`

`std::packaged_task<>` veže `std::future<>` s nekom funkcijom ili bilo čime što se može pozvati (funkcijski objekt, lambda). Predložak `std::packaged_task<>` je funkcijski objekt, čijim pozivom se poziva pridružena funkcija (ili pozivajući objekt) i na kraju njenog izvršavanja modificira članski `std::future<>` objekt. Takav princip se može koristiti za izgradnju upravitelja dretvama (eng. *task scheluder*) - na taj način se apstrahiraju detalji zadataka jer upravitelj se koristi samo instancama tipa `std::packaged_task<>` umjesto pojedinačnim funkcijama. Parametar predloška kod `std::packaged_task<>` je signatura funkcije koja će se pozvati. Primjerice, `int(std::string&, double*)` će stajati kada klasa omotava funkciju koja za argumente prima nekonstantnu referencu na `std::string` i pokazivač na `double`, a vraća `int`. Funkcija (ili pozivajući objekt) koju predajemo konstruktoru klase `std::packaged_task` ne mora imati identičnu signaturu onoj specificiranoj u parametru predloška. Dovoljno je da se ta signatura može dobiti implicitnom konverzijom odgovarajućih tipova. Povratni tip iz parametra predloška će odrediti i tip `std::future<>` objekta kojeg će dati metoda `get_future()`. U nastavku, u svrhu preciznijeg uvida, slijedi parcijalna definicija klase za jednu specijalizaciju `std::packaged_task<>`.

```
1 template<>
2 class packaged_task<std::string(std::vector<char>* ,int)>{
3   public:
4     template<typename Callable>
5     explicit packaged_task(Callable&& f);
```

```
    std::future<std::string> get_future();
    void operator()(std::vector<char>*, int);
};
```

Listing 2.12: parcijalna specijalizacija `std::packaged_task<>`

`std::promise`

Zamislimo situaciju u kojoj ne možemo zadatak reprezentirati samo jednom funkcijom ili zadatak čije rješenje će nastati iz više različitih izvora. Takve slučajeve možemo rješavati pomoću `std::promise<>` predložaka klase. U nastavku slijedi jedan primjer [4] takvog scenarija koji se javlja u razvoju aplikacija. Zamislimo da imamo aplikaciju koja barata s puno mrežnih veza istovremeno. Prirodno se nameće upravljanje sa svakom tom vezom u zasebnoj dretvi zbog jednostavnosti ideje, kao i samog programiranja. Taj pristup zapravo i radi kod manjeg broja veza, odnosno dretvi. Kada broj veza postane velik, operacijski sustav će posljedično trošiti velike resurse za stvaranje velikog broja dretvi i to će utjecati na performanse cijelog sustava. Stoga je u aplikacijama koje su izložene velikom broju mrežnih veza uobičajeno imati manji broj dretvi koje upravljaju vezama i to na način da svaka dretva upravlja s više njih odjednom. Zamislimo rad jedne takve dretve: paketi s podacima će dolaziti iz raznih mrežnih veza i biti će obrađeni u nepredvidivom redoslijedu, i analogno, podaci za slanje mrežnim vezama će biti stavljani u red u nepredvidivom redoslijedu. U većini slučajeva, neki drugi dijelovi aplikacije će čekati ili na uspješno slanje paketa ili na uspješno primanje niza paketa. `std::promise` u paru sa `std::future` nudi mehanizam za takve svrhe. Dretva koja čeka na podatke (slanje ili primanje) će se blokirati pomoću `std::future` (`get()` ili `wait()`), dok će dretva koja je zaslužna za dobavljanje/slanje podataka koristiti `promise` uparen sa spomenutim `std::future<>` objektom da bi spremila vrijednost od interesa i učinila ju dostupnom tom `future` objektu koji blokira dretvu. Svaki `std::promise<>` objekt ima pripadni `std::future<>` objekt koji dohvaćamo pomoću članske funkcije `get_future()`. Kada je vrijednost `promise` objekta postavljena sa `set_value()` u dretvi koja upravlja mrežnim vezama (dobavljanje i slanje podataka), odgovarajući `future` objekt postane spreman (više ne blokira dretvu) i može dohvatiti vrijednost. U nastavku slijedi skica koda za dretvu koja bi na opisani način upravljala mrežnim vezama u nekoj aplikaciji.

```
#include <future>

void obradi_veze(connection_set& veze){

  while(!done(veze)){
    veze_iterator end = veze.end();

    for(veze_iterator veza = veze.begin(); veza != end; veza++){
```

```
9
10      if(veza -> ima_dolazni_podatak()){
11        data_packet data = veza -> dolazno();
12        std::promise<payload_type> &p = veza -> get_promise(data.id);
13        p.set_value(data.payload);
14      }
15
16      if(veza -> ima_odlazni_podatak()){
17        outgoing_packet data = veza -> prvo_u_redu_odlazaka();
18        veza -> send(data.payload);
19        data.promise.set_value(true);
20      }
21    }
22
23  }
24
25 }
```

Listing 2.13: `std::future/std::promise` u mrežnoj aplikaciji

Dretva radi na sljedeći način. Dokle god je potrebno raditi nad mrežnim vezama, iterira po svim vezama i za svaku vezu provjerava postoji li podatak koji dolazi u aplikaciju odnosno podatak koji treba poslati iz aplikacije kroz odgovarajući mrežni kanal. Ukoliko postoji paket koji dolazi u aplikaciju putem trenutne veze (uvjet u 10. liniji), dohvatimo taj paket i iz njega pročitamo odgovarajući identifikator (`id`). Sada nad `std::promise<>` objektu koji je povezan s tim identifikatorom (pomoću *hash tablice* npr.) pozovemo funkciju `set_value()` s argumentom `data.payload` koji je zapravo traženi podatak koji dolazi. Nakon toga će dretva koja čeka na taj podatak imati spreman `std::future<payload_type>` objekt i on ju više neće blokirati. Vrlo je slično i sa drugim slučajem. Ako postoji podatak koji treba poslati putem trenutne veze (uvjet u 16. liniji), dohvatimo podatak koji se nalazi prvi u redu čekanja za slanje kroz tu vezu. Nakon toga ga pošaljemo (linija 18) i konačno odgovarajući `std::promise<bool>` objekt postavimo na `true`. To će signalizirati uspješno slanje poruke onoj dretvi aplikacije koja je inicirala isto.

Spomenimo još za kraj kako u ovakvom pristupu propagirati iznimke do kojih naravno može doći. Pomoću `std::promise<>` to možemo vrlo jednostavno. Recimo da je u dretvi koja izvršava neki bitni izračun došlo do iznimke (eng. *exception*). Ako smo ju uspjeli dohvatiti već u toj dretvi, možemo pozvati samo `moj_promise.set_exception(std::current_exception())`. Pripadni `future` objekt (u dretvi koja je pozvala ovu) će postati spreman i neće više blokirati dretvu - isto kao i da je ova dretva uspješno završila s izračunom. Jasno, na mjestu poziva `moj_future.get()` dobit ćemo iznimku i zato tu naredbu omotamo u `try/catch` blok da bi uspješno propagirali iznimku. Detalji se mogu pronaći u [4].

2.4 Atomske varijable i operacije

Atomskom operacijom nazivamo operaciju na nekom objektu (varijabli) koja je nedjeljiva. Kada takvu operaciju izvršavamo u nekoj dretvi, sve druge dretve vide stanje tog objekta (varijable) ili isto kao prije početka operacije ili ono nakon završetka cijele operacije. Važnost toga je u tome što objekti koji ne osiguravaju atomarnost svojih operacija nisu sigurni za višedretveno izvršavanje (bez dodatnih lokota i sličnih mehanizama). Primjerice, konstruiramo li u glavnoj dretvi neki `struct` sa 3 `int` varijable, druga dretva može pročitati vrijednost tog objekta s jednom ispravno inicijaliziranom varijablom, a druge dvije neinicijalizirane, ako konstruktor ne poštuje princip atomarnosti. To po definiciji uzrokuje stanje natjecanja. Programsko sučelje jezika C++ nudi konstrukciju atomarnih tipova varijabli i operacije nad njima. U sljedećih nekoliko odlomaka dajemo samo kratak uvid u programiranje s atomskim tipovima u C++ budući da ono zahtjeva dobro poznavanje memorijskog modela jezika, a to je tema koja izlazi iz okvira ovog rada. Zainteresiranog čitatelja upućujemo na [4].

Potrebno je uključiti zaglavlje <atomic>. Središnja stvar tog zaglavlja je predložak klase `std::atomic<T>`. Standard garantira da su sve operacije na takvim objektima atomske. Od posebnog interesa su nam tipovi za koje će `std::atomic<T>` raditi atomski bez korištenja lokota (mutexa). Radi se o tome da, ukoliko je tip `T` dovoljno kompliciran, `std::atomic<T>` objekt će interno koristiti lokot za postizanje atomarnosti, a to nam nije cilj. U tim slučajevima se zapravo i preporuča ručno upravljanje tipom pomoću lokota. Gotovo svi objekti tipa `std::atomic<T>` imaju definiranu funkciju članicu `is_lock_free()` koja vraća `true` u slučaju da se atomarnost ostvaruje bez internih lokota, a `false` inače. Naglasimo samo da funkcija `is_lock_free()` ovisi o arhitekturi hardvera na kojem se izvršava te da za neki tip `T` možda imamo atomarnost na nekom računalu dok na nekom drugom nemamo. U tu svrhu je u C++17 uvedena statička varijabla članica `static constexpr X::is_lock_free` koja ima vrijednost `true` ako atomarni tip `X` dopušta atomarnost bez unutarnjih lokota na svim hardverima koji mogu pokrenuti program koji nastaje kompiliranjem sa danim prevoditeljem, a `false` inače. Još jedan bitan tehnički detalj: standardni atomski tipovi ne dopuštaju kopiranje ni pridruživanje kopiranjem. [4]

`std::atomic_flag`

Najprimitivniji atomski tip objekta je `std::atomic_flag`. Za njega nije definirana spomenuta članska funkcija `is_lock_free()` i za sve operacije koje on nudi se garantira da se ostvaruju bez unutarnjih zaključavanja. To je ujedno i jedini tip koji ima tako jake garancije. Radi se naravno o primitivnom `boolean` tipu koji podržava dva moguća stanja (istina ili laž). Obavezno ga treba inicijalizirati s vrijednošću `ATOMIC_FLAG_INIT`. Do standarda C++17 ovaj tip je podržavao samo operacije `test_and_set()`, `clear()`, i operator pridruživanja =. Od standarda C++20 dodane su funkcije članice `test()`,

`wait()`, `notify_one()`, `notify_all()`. [5] Zbog svojih jakih svojstava atomarnosti `std::atomic_flag` se koristi i za konstrukciju drugih atomarnih tipova. Više detalja o tipu `std::atomic_flag` može se pronaći u [4].

Osnovni atomski tipovi

U ovoj točki ukratko opisujemo sučelje koje pružaju najčešće korišteni atomski tipovi. Počnimo sa `std::atomic<bool>`. On je malo složenija verzija tipa `std::atomic_flag` i iako ne dopušta konstruktor kopiranjem ni pridruživanje kopiranjem, možemo ga konstruirati iz ne-atomske `bool` varijable. Također, definiran je operator pridruživanja =. Ipak, razlika je u tome da je povratna vrijednost tog pridruživanja isključivo vrijednost, a ne referenca kao što smo navikli kod takvih operatora. Čitati iz varijable možemo pomoću članske funkcije `load()`, a pisati pomoću `store()`. Obje operacije su atomarne, a uvedena je i opracija `std::atomic<T>::exchange(T desired)` koja atomski može pročitati vrijednost, zamijeniti ju novom i vratiti prijašnju. Naravno, ova operacija radi za sve tipove `T` za koje je to moguće, a ne samo kod atomskog `bool` tipa. Integralni atomski tipovi podržavaju širi popis funkcija: `fetch_add()`, `fetch_sub()`, `fetch_and()`, `fetch_or()`. Za te funkcije su definirani i odgovarajući operatori (+, -, &, |), kao i operatori `++x` i `--x` i njihove postfiksne verzije. Primijetimo da nisu definirane operacije množenja, dijeljenja i bitovnog pomaka. No to ne predstavlja značajni gubitak budući da cjelobrojne atomske tipove obično koristimo kao brojače ili bitmaske za neka stanja, a za to imamo dovoljno operatora koji djeluju atomski. U standardu C++20 je izglađeno i sučelje za `float`, `double` i `long double`. Prije toga operacije poput `fetch_add()` ili operatora -= nisu bile atomske. [5].

Sinkronizacija dretvi s atomskim tipovima

Osim što nam omogućavaju atomske operacije, atomski tipovi zapravo predstavljaju i točke sinkronizacije dretvi, tj. mogu odrediti poredak izvršavanja instrukcija iz različitih dretvi. Gotovo svaka metoda na atomskim tipovima nudi mogućnost prosljeđivanja argumenta tipa `std::memory_order` koji će odrediti tip sinkronizacije. Sljedeći [4] primjer pokazuje na koji način atomski tipovi daju sinkronizaciju.

```
1  #include <vector>
2  #include <thread>
3  #include <atomic>
4  #include <chrono>
5  #include <iostream>
6
7  std::vector<int> podaci;
8  std::atomic<bool> spreman(false);
9
10 void reader_thread(){
```

```
11   while(!spreman.load()){
12     std::this_thread::sleep_for(std::chrono::milliseconds(1));
13   }
14   std::cout << "Dohvaceno = " << podaci[0] << "\n";
15 }
16
17 void writer_thread(){
18   podaci.push_back(42);
19   spreman = true;
20 }
21
22 int main(){
23   std::jthread citaj{reader_thread};
24   std::jthread pisi{writer_thread};
25   return 0;
26 }
```

Listing 2.14: sinkronizacija sa `std::atomic<bool>`

U 23. i 24. liniji pokrenemo dvije dretve. Jedna izvršava funkciju koja u globalni vektor `podaci` ubacuje vrijednost broj 42 i nakon toga postavlja globalni `std::atomic<bool>` na `true`. Druga dretva izvršava funkciju `reader_thread()` koja na početku ima jednostavnu `while` petlju: najprije atomski provjeri je li vrijednost zastavice `spreman` `true`. Ako nije, dretva odspava jednu milisekundu. Kada zastavica `spreman` postane `true`, dretva izađe iz petlje i ispiše dohvaćenu vrijednost `podaci[0]`. Uočimo, u ovom jednostavnom primjeru je od puno većeg interesa postojanje vrijednosti `podaci[0]` od toga kolika je ona. Kada će to biti ispunjeno? Nužan i dovoljan uvjet je očit: `podaci.push_back(42)` je morao završiti ne kasnije nego što je poziv `spreman.load()` vratio `true`. To općenito ne mora biti slučaj: prisjetimo se da kompajler može izmijeniti redoslijed nekih instrukcija u kodu ako to smatra optimalnim i naravno ako to ne narušava logiku programa. Kod višedretvenih programa kompajler može nesvjesno napraviti takve greške, no u našem primjeru smo sigurni da neće upravo zato jer atomski tip `spreman` u našem kodu koristi najjaču moguću sinkronizaciju. To je ona sinkronizacija u kojoj sve dretve vide jednaki poredak čitanja i pisanja na atomskoj varijabli. Uočimo da operacijama nad `spreman` nismo kao dodatni argument prosljedili vrijednost tipa `std::memory_order` jer smo koristili onu koja je zadana kada taj argument nedostaje: `std::memory_order_seq_cst`. Ona predstavlja upravo spomenuti *sekvencijalno konzistentni uređaj* koji je najjači oblik sinkronizacije. Detalji se mogu pronaći u [4].

Od standarda C++20 atomski tipovi pružaju još jedan način sinkronizacije dretvi. On se ostvaruje pomoću novih funkcija članica `wait()` i `notify_one()` odnosno `notify_all()`. Korištenjem atomskih varijabli i njihovih članica `wait()` i `notify_one()`/`notify_all()` se može simulirati rad `std::mutex` objekta što se ponekad isplati s obzirom da korištenje mutexa može biti skupo u nekim situacijama. Također, pomoću atomskih cjelobrojnih ti-

pova možemo konstruirati pravedan upravitelj dretvama, tj. ne opteretiti jednu dretvu sa puno podataka, a ostale dretve ostavljati slobodnima. Štoviše, u popularnom problemu koji se sastoji od globalnog reda i dretvi koje skidaju s njega podatak i zatim ga obrađuju, možemo osigurati da podaci s reda završavaju obrađeni u istom redoslijedu u kojem su bili pripremljeni u redu i pritom možemo koordinirati broj dretvi kojima dopuštamo da rade u svakom trenutku. Sljedeći [5] kod ilustrira takav program.

```
#include <iostream>
#include <queue>
#include <chrono>
#include <thread>
#include <atomic>
#include <semaphore>

using namespace std::literals;

int main(){
  char act_char = 'a';
  std::mutex act_char_mutex;

  std::atomic<int> maxTicket{0};
  std::atomic<int> activeTicket{0};

  constexpr int broj_dretvi = 10;

  std::vector<std::jthread> dretve;
  for (int i = 0; i < broj_dretvi; i++) {
    dretve.push_back(std::jthread{[&, i](std::stop_token st){
        while(!st.stop_requested()){
          char val;
          {
            std::lock_guard lg{act_char_mutex};
            val = act_char++;
            if(act_char > 'z')
              act_char = 'a';
          }

          int mojTicket{++maxTicket};
          int aktivan = activeTicket.load();
          while(aktivan < mojTicket){
            activeTicket.wait(aktivan);
            aktivan = activeTicket.load();
          }

          for(int j = 0; j < 10; j++) {
            std::cout.put(val).flush();
            auto dur = 20ms * ((i % 3) + 1);
```

```
41              std::this_thread::sleep_for(dur);
42            }
43
44            ++activeTicket;
45            activeTicket.notify_all();
46          }
47      }});
48    }
49
50    auto adjust = [&, oldNum = 0](int newNum) mutable{
51      activeTicket += newNum - oldNum;
52      if(newNum > 0)
53        activeTicket.notify_all();
54      oldNum = newNum;
55    };
56
57    for(int num : {0, 3, 5, 2, 0, 1}){
58      std::cout << "\n====== dozvola za " << num << " dretvi" << std::endl
       ;
59      adjust(num);
60      std::this_thread::sleep_for(2s);
61    }
62
63    for(auto& t : dretve){
64      t.request_stop();
65    }
66
67    return 0;
68  }
```

Listing 2.15: sinkronizacija sa `wait()` i `notify_all()`

Atomska varijabla `maxTicket` sadrži identifikator najkasnijeg dodijeljenog zadatka, dok je u `activeTicket` spremljen najveći indeks među zadacima koji se trenutno smiju izvršavati. U ovom primjeru nemamo eksplicitno definiran red kao strukturu već se vrijednost s početka reda generira ciklički (linije 25.-28.). Program radi na sljedeći način. Svaka dretva radi dokle god ne dođe zahtjev za njenim prekidanjem izvana. Najprije pomoću `act_char` izračuna sljedeću vrijednost u redu. Tada varijabli `mojTicket` daje do znanja da ima zadatak za obraditi i tom prilikom zapamti svoj indeks, tj. redni broj koji je pročitala od `mojTicket`. Nakon toga pogleda koja je maksimalna oznaka dretve koja ima dopuštenje za rad. Ako je ta oznaka strogo manja od rednog broja koji je dretva upravo dobila, tada idemo na spavanje, čekajući da varijabla `aktivan` poprimi novu vrijednost (34. linija). Ako pak je ta oznaka veća ili jednaka rednom broju dretve koji smo dobili u 31. liniji, ili je nakon nekog vremena dovoljno porastao maksimalni indeks dretve koja smije raditi, krećemo u obradu podatka (samo 10 puta ispišemo znak koji smo dohvatili). Na

kraju obrade atomarno povećamo za 1 maksimalni indeks zadatka koji se smije rješavati i o tome obavijestimo sve dretve koje čekaju. Bitno je da obavijestimo sve dretve jer nije unaprijed poznato koja će proći uvjet (indeksi). Lambda funkcija `adjust` u 50. liniji služi za ručno podešavanje broja dretvi (zadataka) koji dozvoljavamo da radi).

2.5 Novi sinkronizacijski mehanizmi u C++20

`std::latch`

`std::latch` [5] je klasa iz zaglavlja <latch> koja pruža mogućnost za sinkronizaciju višedretvenih programa, a uvedena je u standardu C++20. Ideja njenog rada je jednostavna: objekt se inicijalizira nekom nenegativnom cjelobrojnom vrijednošću koju razne dretve mogu atomarno smanjivati za 1, sve dok ne poprimi vrijednost 0. Dretve koje čekaju da ta vrijednost padne na nulu mogu svoje izvršavanje blokirati pozivom funkcije `std::latch::wait()` na odgovarajućem objektu. Takvo odbrojavanje do nule jedan `std::latch` objekt može provesti samo jednom, tj. za svako sljedeće treba konstruirati novi `std::latch` objekt. U nastavku slijedi jednostavan primjer korištenja klase.

```
1  #include <iostream>
2  #include <array>
3  #include <thread>
4  #include <latch>
5
6  using namespace std::literals;
7
8  void radi(char c){
9    //printamo svaki znak polovinu njegove ascii vrijednosti puta
10   for (int j = 0; j < c / 2; j++) {
11     std::cout.put(c).flush();
12     std::this_thread::sleep_for(100ms);
13   }
14 }
15
16 int main(){
17   std::array oznake{'.', '?', '8', '+', '-'}; //oznake zadataka koje
       radimo
18
19   std::latch sviGotovi{oznake.size()};
20
21   std::jthread t1{[oznake, &sviGotovi] {
22     for(unsigned i = 0; i < oznake.size(); i += 2){
23       radi(oznake[i]);
24       sviGotovi.count_down();
25     }
```

```
26   }};
27
28   std::jthread t2{[oznake, &sviGotovi] {
29     for (unsigned i = 1; i < oznake.size(); i += 2) {
30       radi(oznake[i]);
31       sviGotovi.count_down();
32     }
33   }};
34
35   std::cout << "\ncekamo dok ne zavrse svi zadaci\n";
36   sviGotovi.wait();
37   std::cout << "\nsvi zadaci su gotovi\n";
38   t1.join();
39   t2.join();
40 }
```

Listing 2.16: primjer upotrebe `std::latch` klase

Ovaj program skicira kako neki skup zadataka podjeliti nekom skupu dretvi (konkretno, dvije su radne dretve u primjeru) i pritom u fiksiranom dijelu koda osigurati da su svi zadaci obrađeni. Detaljniji opis slijedi u nastavku. Najprije konstruiramo 5 zadataka sa specifičnim oznakama (17. linija). Nakon toga inicijaliziramo `std::latch` objekt na vrijednost koja predstavlja broj zadataka. Nakon toga inicijaliziramo dvije dretve tako da obave taj skup zadataka alternirajućom podjelom: t_1 obavlja prvi, treći i peti zadatak, a t_2 riješi drugi i četvrti. Rad na zadataku je u ovom primjeru ispis oznake zadatka onoliko puta kolika je polovina (najveće cijelo) *ASCII* vrijednosti te oznake. Nakon što radna dretva završi neki zadatak, odmah atomarno smanjuje vrijednost objekta `sviGotovi` za 1. Bitno je primjetiti da u 36. liniji `wait()` nad `std::latch` objektom ne čeka da se dretve završe, već samo da vrijednost padne na nulu i time zadaci budu izvršeni. Za čekanje na kraj rada dretvi još uvijek trebamo koristiti `join()`, odnosno on se automatski pozove u destruktorima `std::jthread` objekata t_1 i t_2.

```
1 ?
2 cekamo dok ne zavrse svi zadaci
3 .?.?..?.?.?.?.?.??.?..?.?.?.?.??..??..??..?.?
4 8??8?8?88?8?8??88+8+8+8+8+8+8+8+8+8+8+
5 8+8+8+8+8+8+8+8+8+8+-+----------------------
6 svi zadaci su gotovi
```

Listing 2.17: mogući izlaz

Mehanizam kojeg nudi `std::latch` se može iskoristiti i za sinkronizaciju većeg broja dretvi u jednoj točki i tada ih pustiti da rade dalje. Jedna korisna primjena takve sinkronizacije je da na taj način možemo osigurati da više dretvi krene s radom u istom trenutku čak i kada inicijalizacije dretvi mogu potrošiti neko nezanemarivo vrijeme. Sljedećim primjerom prezentiramo tu upotrebu.

```
1  #include <iostream>
2  #include <array>
3  #include <vector>
4  #include <thread>
5  #include <latch>
6
7  using namespace std::literals;
8
9  int main(){
10   std::size_t brojDretvi = 10;
11
12   std::latch sviGotovi{int(brojDretvi)};
13
14   std::vector<std::jthread> dretve;
15   for (int i = 0; i < brojDretvi; ++i) {
16     std::jthread t{[i, &sviGotovi] {
17       std::this_thread::sleep_for(100ms * i);
18       sviGotovi.arrive_and_wait();
19       for (int j = 0; j < i + 5; ++j) {
20         std::cout.put(static_cast<char>('0' + i)).flush();
21         std::this_thread::sleep_for(50ms);
22       }
23     }};
24     dretve.push_back(std::move(t));
25   }
26
27   for(int i = 0; i < int(dretve.size()); i++){
28     dretve[i].join();
29   }
30
31   return 0;
32 }
```

Listing 2.18: sunkronizacija više dretvi pomoću `std::latch`

Način na koji osiguravamo da sve dretve u istom trenutku (približno) krenu raditi je sljedeći. U 12. liniji inicijaliziramo `std::latch` objekt na broj dretvi koje rade. Sada standardno pomoću petlje inicijaliziramo dretve i pritom populiramo vektor. Rad svake dretve je određen lambda funkcijom koja najprije odspava neko vrijeme. U 18. liniji radimo ključnu stvar: naredba `sviGotovi.arrive_and_wait()` radi najprije `count_down()` i zatim pozove `wait()` (sve na istom `latch` objektu). Zbog garantirane atomarnosti `count_down()` operacije, kada vrijednost objekta `sviGotovi` dođe do nule znamo da su sve dretve spremne za izvršavanje sljedeće naredbe - `for` petlje u 19. liniji.

```
1  02146835970214683597024631859704631287950463 12
2  879542671935842936587493658749658796578697897 8989
```

Listing 2.19: primjer izlaza kod pokretanja koda 2.16

Kao posljedicu ove sinkronizacije možemo primjetiti jednolikost u ispisu: vidimo da nemamo situaciju da neka dretva upiše više svojih identifikatora zaredom ili da se izmjenjuje samo neki manji podskup od svih dretvi.

std::barrier

`std::barrier` [5] iz zaglavlja <barrier> je novi sinkronizacijski mehanizam za višedretvene programe koji nam omogućava sinkronizaciju više asinkronih zadataka i to više puta. Prema načinu rada je vrlo sličan opisanom `std::latch` iz prošle točke. Isto se inicijalizira nekim nenegativnim brojem kojeg dretve atomarno mogu dekrementirati za jedan dok ne dođe do nule. Za razliku od `std::latch` objekta, kada se dosegne vrijednost nula, opcionalno se poziva unaprijed određena funkcija (eng. *callback* funkcija). Najbitnija razlika je da se dolaskom do vrijednosti nula vrijednost brojača natrag postavlja na onu kojom je objekt bio inicijaliziran. `std::barrier` je dakle vrlo koristan u situaciji kada više dretvi računa nešto zajedno i to rade više puta. Svaki puta kada su dretve odradile zadatak, mogu pozvati *callback* funkciju koja će procesuirati međurezultat, a nakon toga dretve kreću u novi krug računanja. U nastavku slijedi reprezentativni primjer takvog korištenja opisanog sinkronizacijskog mehazima.

```
#include <iostream>
#include <vector>
#include <thread>
#include <cmath>
#include <barrier>
#include <iomanip>

int main(){
  std::vector vrijednosti{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0};

  auto ispisiVrijednosti = [&vrijednosti]() noexcept{
    for (auto v : vrijednosti) {
      std::cout << std::fixed << std::setprecision(6) << v << " ";
    }
    std::cout << '\n';
  };
  ispisiVrijednosti();

  std::barrier sviGotovi{int(vrijednosti.size()), ispisiVrijednosti};

  std::vector<std::jthread> dretve;
  for(std::size_t i = 0; i < vrijednosti.size(); i++) {
    dretve.push_back(std::jthread{[i, &vrijednosti, &sviGotovi] {
        for(int j = 0; j < 5; j++) {
          vrijednosti[i] = std::sqrt(vrijednosti[i]);
          sviGotovi.arrive_and_wait();
```

```
27          }
28          }});
29    }
30
31
32    return 0;
33  }
```

Listing 2.20: 5 puta se procesuira međurezultat rada više dretvi sa `std::barrier`

U 9. liniji je inicijaliziran vektor vrijednosti čije korijene ćemo računati. U 11. liniji se inicijalizira lambda funkcija koja će imati ulogu spomenute *callback* funkcije vezane uz `std::barrier sviGotovi`. Pravilo je da *callback* funkcija mora biti `noexcept`. Jasno, brojač objekta `sviGotovi` je inicijaliziran s brojem elemenata u vektoru i objektu je pridružena spomenuta *callback* funkcija. Zatim za svaki element vektora `vrijednosti` konstruiramo dretvu koja će za njega računati korijene (23. linija). Promotrimo sada rad svake takve dretve. Ona 5 puta radi sljedeće: izračuna drugi korijen trenutnog broja `vrijednosti[i]` i spremi ga natrag u istu varijablu. Zatim dekrementira brojač od `sviGotovi` i ode na čekanje. Kada posljednja dretva to učini i postavi brojač na 0, poziva se specificirana *callback* funkcija i brojač se natrag postavlja na `vrijednosti.size()`. U nastavku slijedi izlaz programa - ovdje namjerno izostavljamo izraz "mogući izlaz programa" što smijemo zbog dobre sinkronizacije.

```
1 1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000
2 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
3 1.000000 1.189207 1.316074 1.414214 1.495349 1.565085 1.626577 1.681793
4 1.000000 1.090508 1.147203 1.189207 1.222845 1.251033 1.275373 1.296840
5 1.000000 1.044274 1.071075 1.090508 1.105823 1.118496 1.129324 1.138789
6 1.000000 1.021897 1.034928 1.044274 1.051581 1.057590 1.062697 1.067140
```

Listing 2.21: izlaz programa

Spomenimo još jednu praktičnu mogućnost kod predloška klase `std::barrier`. Radi se o funkciji članici `arrive_and_drop()`. Ona smanji brojač za jedan ali i osigura da se u idućoj fazi (nakon što se objekt ponovno inicijalizira na početnu vrijednost) taj početni broj smanji za jedan. Primjena toga je kad imamo više dretvi koje rade slično kao u kodu 2.18, ali uz dodatnu mogućnost da neke dretve nakon nekog broja obavljenih ciklusa zaustavimo, a ostale nastave raditi na isti način.

Semafori

Moderni C++ uvodi i sinkronizacijski mehanizam čiji rad se može usporediti sa radom semafora na razini operacijskog sustava. Radi se o predlošku klase `std::counting_semaphore<>` i klasi `std::binary_semaphore`. [5] Kasnije ćemo pomoću primjera objasniti parametar navedenog predloška klase. Ti predlošci predstavljaju

zapravo sličan tip sinkronizacije kao i onaj sa `std::mutex` objektima. Naime, ponovno ograničavamo pristup resursima (jednom ili više njih), ali semafori omogućavaju da pristup resursima nekoj dretvi može odobriti dretva koja ih uopće ne koristi (kod `std::mutex` objekta, `std::mutex::unlock()` mora izvršiti točno ona dretva koja je napravila i `std::mutex::lock()` - u suprotnom je ponašanje nedefinirano). Također, ove semafore možemo koristiti i u svrhu ograničavanja pristupa resursima. Primjerice, možemo ih koristiti u konstrukciji upravljača skupom dretvi (eng. *thread scheduler*) za variranje broja dretvi čiji rad dopuštamo. Idući primjer demonstrira upravo jednu takvu primjenu.

```
#include <iostream>
#include <queue>
#include <chrono>
#include <thread>
#include <mutex>
#include <semaphore>

using namespace std::literals;

int main(){
  std::queue<char> red;
  std::mutex red_mutex;

  for(int i = 0; i < 1000; i++){
    red.push(static_cast<char>('a' + (i % ('z' - 'a'))));
  }

  constexpr int brojDretvi = 10;
  std::counting_semaphore<brojDretvi> dopusteno{0};

  std::vector<std::jthread> skup_dretvi;
  for(int i = 0; i < brojDretvi; i++){
    std::jthread t{[i, &dopusteno, &red, &red_mutex](std::stop_token st)
    {
      while(!st.stop_requested()){
        dopusteno.acquire();
        char val;
        {
          std::lock_guard lg{red_mutex};
          val = red.front();
          red.pop();
        }

        for(int j = 0; j < 10; j++){
          std::cout.put(val).flush();
          auto spavaj = 130ms * ((i % 3) + 1);
          std::this_thread::sleep_for(spavaj);
```

```
37          }
38
39          dopusteno.release();
40        }
41      }};
42
43      skup_dretvi.push_back(std::move(t));
44    }
45
46    std::cout << "== cekam 2 sekunde (ni jedna dretva nema dozvolu)\n";
47    std::cout << std::flush;
48    std::this_thread::sleep_for(2s);
49
50    std::cout << "== dozvolimo rad za 3 dretve\n" << std::flush;
51    dopusteno.release(3);
52    std::this_thread::sleep_for(2s);
53
54    std::cout << "\n== dozvoljavamo rad za jos 2 dretve \n" << std::flush;
55    dopusteno.release(2);
56    std::this_thread::sleep_for(2s);
57
58    std::cout << "\n== kraj racunanja\n" << std::flush;
59    for (auto& t : skup_dretvi) {
60      t.request_stop();
61    }
62  }
```

Listing 2.22: jednostavni upravitelj dretvama

U 11. i 12. liniji inicijaliziramo red i pripadni `std::mutex` koji ga štiti. U 19. liniji inicijaliziramo konstantu koja označava maksimalan broj pristupa resursima koje štiti semafor. Ta konstanta mora biti `constexpr` jer je potrebno da bude poznata za vrijeme kompilacije s obzirom da ulazi kao parametar predloška za `std::couting_semaphore<>`. U `for` petlji u 22. liniji krećemo s konstrukcijom dretvi i populiranjem vektora `skup_dretvi`. Dretve ponovno konstruriramo prosljeđivanjem opcionalnog argumenta `std::stop_token` čije značenje je opisano u početku poglavlja. Dakle, svaka dretva radi dokle god ne dobije poziv za prekid rada (24. linija). Najprije (25. linija) pomoću poziva članske funkcije `acquire()` na semaforu se atomarno smanjuje vrijednost brojača semafora ako je ona pozitivna, dok se u protivnom dretva blokira, čekajući da se brojač inkrementira. Kada se dretva izbori za ulazak u kritičnu sekciju, zaključava `red_mutex` koji štiti red s vrijednostima, a zatim uzima vrijednost koja je na početku reda i makne ju. Nakon toga otključava `red_mutex` (RAII) i ispisuje dohvaćenu vrijednost 10 puta na standardni izlaz, uz dodano spavanje za bolji dojam konkurentnosti. U 39. liniji pozivamo člansku funkciju `release(int upd = 1)` semafora `dopusteno` koja atomarno povećava brojač semafora i time potencijalno neka dretva koja je blokirana sa `acquire()` dobiva

dopuštenje za ulazak. Primijetimo, prije `for` petlje smo brojač semafora inicijalizirali s nulom pa ni jedna dretva koja se konstrurira u `for` petlji neće krenuti s obradom. Tek u 51. liniji pomoću `dopusteno.release(3)` dajemo trima dretvama dopuštenje za prolazak `acquire` poziva u definiciji lambdi koje određuju rad dretvi. Nakon dvije sekunde dozvolimo rad za još dvije dretve (55. linija). U 60. liniji pomoću mehanizma pomognutog s `jthread` i `stop_token` objektima naređujemo dretvama da završe s radom. Razlog za parametrizaciju klase `std::counting_semaphore` je da za vrijeme kompajliranja prevoditelj može odabrati najefikasniju implementaciju klase za taj maskimalni broj pristupa. Napomenimo još da za `std::counting_semaphore<>` postoje i članske funkcije `try_acquire`, `try_acquire_for` i `try_acquire_until` čiji rad je prilično dobro opisan njihovim imenima, a detalji i primjena se mogu pronaći u [5]. Što se tiče klase `std::binary_semaphore`, ona je zapravo specijalni slučaj upravo opisane klase, tj. u zaglavlju `<semaphore>` imamo naredbu `using binary_semaphore = std::counting_semaphore<1>;` [2] Njegova primjena je dovoljno česta u praksi da je zaslužio posebni naziv. U [5] se nalazi jedan primjer sa `std::binary_semaphore` koji pokazuje kako dva takva semafora mogu imitirati rad jedne uvjetne varijable (`std::condition_variable`).

Poglavlje 3

Paralelizacija algoritama na grafovima

U ovom poglavlju ćemo iskoristiti stečeno znanje o programiranju višedretvenih programa s dijeljenom memorijom u programskom jeziku C++ za implementaciju triju poznatih algoritama na grafovima. Prije toga navodimo osnovne pojmove iz teorije grafova koji će nam trebati za razumijevanje algoritama i ukratko govorimo o reprezentaciji grafa u računalu.

3.1 Uvod

Najprije definiramo nekoliko pojmova iz teorije grafova koji će nam biti potrebni za razumijevanje algoritama. *Graf* je uređeni par (V, E), pri čemu V nazivamo skupom čvorova grafa, a E skupom bridova. Skup E sadrži uređene parove oblika (u, v), pri čemu su $u, v \in V$. Ukoliko za svaki par čvorova $v_1, v_2 \in V$ vrijedi da $(v_1, v_2) \in E$ povlači $(v_2, v_1) \in E$, kažemo da je graf (V, E) *neusmjeren*. U protivnom kažemo da je graf (V, E) *usmjeren*. *Izlazni stupanj* (eng. *outdegree*) čvora $u \in V$ je broj bridova oblika (u, v). Ulazni stupanj (eng. *indegree*) čvora u je broj bridova oblika (v, u). *Šetnja* u grafu $G = (V, E)$ je konačan niz čvorova $(v_1, v_2, ..., v_k)$, $k \geq 2$ za koji vrijedi $(v_{i-1}, v_i) \in E$ za sve $2 \leq i \leq k$. Kažemo da je šetnja *zatvorena* ako vrijedi $v_1 = v_k$. *Put* u grafu je šetnja u kojoj su svi čvorovi u parovima različiti, uz jedino dopuštenje $v_1 = v_k$ i tada govorimo o zatvorenom putu, odnosno *ciklusu*. Kažemo da su čvorovi x i y *povezani putem* u grafu $G = (V, E)$ ako postoji prirodan broj $k \geq 2$ i put $(v_1, ..., v_k)$ u grafu G takav da je $v_1 = x$ i $v_k = y$. Graf $G = (V, E)$ je *povezan* ako je svaki par različitih čvorova $u, v \in V$ povezan putem. Bridovima grafa ponekad pridružujemo težine, pa tako ističemo *težinske grafove* kao grafove kojima je pridružena i funkcija $c : E \to \mathbb{R}$ sa skupa bridova u skup realnih brojeva. *Stablo* je povezani graf koji ne sadrži ciklus. *Razapinjuće stablo* grafa $G = (V, E)$ je svako stablo $T = (V', E')$ za koje vrijedi $V' = V$ i $E' \subseteq E$. *Minimalno razapinjuće stablo* težinskog grafa $G = (V, E, c)$ je svako razapinjuće stablo $T = (V, E')$ grafa G koje minimizira izraz $cost(E'') = \sum_{e \in E''} c(e)$

po svim razapinjućim stablima (V, E'') grafa G.

3.2 Reprezentacija grafa u računalu

Kada govorimo o implementaciji algoritama na grafovima, najprije se postavlja pitanje reprezentacije grafa u memoriji računala. U ovoj točki navodimo tri moguća rješenja, no postoji i više njih. Bitno je samo imati na umu da različiti alogoritmi, za ostvarivanje svoje maksimalne efikasnosti traže potencijalno različita rješenja reprezentacije grafa. Čvorovi grafa u primjenama mogu biti razne vrste objekata - ne nužno jednostavni tipovi poput cijelih brojeva ili znakova. Mi ćemo ipak pretpostaviti da su naši čvorovi uvijek označeni prirodnim brojevima. To neće smanjivati općenitosti algoritama niti utjecati na njihove performanse - u općenitoj situaciji lako svim čvorovima grafa pridružimo različite cjelobrojne identifikatore (istu stvar primjenimo u skupu bridova). Za sve grafove $G = (V, E)$ stoga nadalje pretpostavljamo $V = \{1, 2, ..., k\}$ za neki prirodan broj k. Sljedeća rješenja se prema tome razlikuju samo u reprezentaciji bridova.

Popis bridova

Graf možemo izravno reprezentirati tako da u neki niz uređenih parova ubacimo sve bridove, tj. uređene parove vrhova kojima su određeni. U jeziku C++ primjerice možemo koristiti objekt tipa `std::vector<std::pair<int, int>>`. U slučaju težinskog grafa ubacujemo uređene trojke, gdje treći član predstavlja težinu brida. Pogodni tip za takav niz bi u jeziku C++ bio `std::vector<std::tuple<int, int, int>>`. Predložak klase `std::tuple<>` je uveden standardom C++11, a praktičan je za okupljenje više varijabli (ne nužno istog tipa) u jedan objekt. Detaljnije o tom tipu se nalazi na [3].

Niz susjedstva (eng. *adjacency array*)

U nekim algoritmima je krucijalno da se za svaki čvor u grafu efikasno dohvate svi bridovi koji izlaze iz njega. Rješenje reprezentacije bridova u takvim okolnostima može biti sljedeće. Imamo jedan niz *edges* koji će imati duljinu M, pri čemu je M ukupan broj bridova grafa. U taj niz ćemo najprije ubaciti identifikatore svih susjeda čvora 1 (prema kojima postoji izlazni brid iz 1). Neposredno nakon zadnjeg ubačenog elementa ubacimo identifikatore svih susjeda čvora 2. Tako ponavljamo redom za sve čvorove grafa. Paralelno s time održavamo i niz *begin_ptr*[] veličine $N+2$, pri čemu je N ukupni broj čvorova grafa. U tom nizu *begin_ptr*[v] drži indeks na oznaku prvog susjeda čvora v u polju *edges*. Zbog redoslijeda ubacivanja slijedi da je *begin_ptr*[$v + 1$] - 1 indeks posljednjeg susjeda čvora v, uz iznimku kada čvor v nema izlaznih bridova. Tada je *begin_ptr*[v] = *begin_ptr*[$v + 1$]. Sada možemo `for` petljom jednostavno proći po svim izlaznim bridovima čvora. U slučaju

da želimo pamtiti i težinu brida, možemo dodati novo polje, ili niz *edges* promijeniti tako da sadrži parove brojeva. Napomenimo samo da je ovo rješenje ispravno samo u slučaju *statičkih* grafova, tj. onih kojima se ne mijenjaju niti skup čvorova niti skup bridova prilikom izvođenja algoritma. U slučaju *dinamičkih* grafova se mogu koristiti razne varijante povezanih listi (eng. *linked list*). Takvo rješenje je detaljno opisano u [6]. Za algoritme u kojima ćemo mi koristiti niz susjedstva u ovom radu će ovo rješenje biti dovoljno.

Matrica susjedstva

Graf $G = (V, E)$ sa N čvorova možemo vrlo intuitivno reprezentirati kvadratnom matricom A reda N tako da definiramo $A_{ij} = 1$ ako u G grafu postoji brid (i, j). U protivnom stavimo $A_{ij} = 0$. Na sličan način reprezentiramo težinski graf $G = (V, E, c)$. Definiramo $A_{ij} = c(i, j)$ ako u G postoji brid (i, j). U suprotnom A_{ij} postavimo na neku vrijednost koja će dogovorno označavati nepostojanje brida (i, j). Matrice su očito vrlo praktične u slučaju grafova koji imaju puno bridova. Također, pomoću njih jednostavno implementiramo dodavanje odnosno brisanje brida iz grafa (ili promjenu težine): potrebno je samo dohvatiti i promijeniti jedan element matrice.

3.3 Pretraživanje u širinu

Algoritam pretrage u širinu (eng. *breadth-first search* ili *BFS*) jedan je od najpoznatijih algoritama na grafovima. Taj algoritam za ulaz ima graf $G = (V, E)$ i (fiksni) početni čvor s. Izlaz algoritma čine dvije korisne informacije za svaki od čvorova grafa: je li taj čvor dostižan od početnog čvora s i koji je najmanji broj bridova pomoću kojeg se taj vrh može dostići iz s. Vidjet ćemo u nastavku da pomoću BFS algoritma lako možemo dobiti jedno razapinjuće stablo grafa s korijenom u početnom čvoru s. U tom slučaju se izlazu algoritma dodaje i niz *parent* u kojem za svaki čvor v (osim početnog) $parent[v]$ označava roditelja od v u takvom stablu. Ovaj algoritam radi i za usmjerene i za neusmjerene grafove.

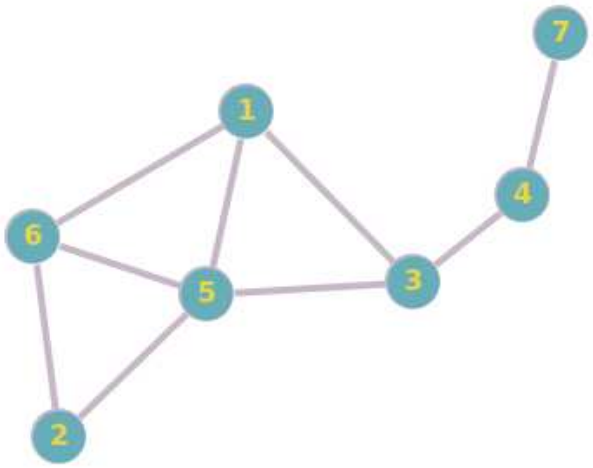
Sekvencijalni algoritam

Najprije opisujemo sekvencijalni algoritam, a zatim ćemo ga paralelizirati. U nastavku slijedi pseudokod algoritma [6] te zatim u nekoliko rečenica dajemo detaljniji opis.

Algorithm 1 Pretraživanje u širinu (BFS)

```
function BFS(s, V, E)
    dist ← [∞, ∞, ..., ∞]
    parent ← [⊥, ⊥, ..., ⊥]
    dist[s] ← 0, parent[s] ← s
    Q ← [s]
    Q' ← []
    l ← 0
    while Q ≠ [] do
        for each u ∈ Q do
            for each (u, v) ∈ E do
                if parent[v] = ⊥ then
                    parent[v] ← u
                    dist[v] ← l + 1
                    Q' ← Q' ∪ {v}
                end if
            end for
        end for
        (Q, Q') ← (Q', [])
        l ← l + 1
    end while
    return (dist, parent)
end function
```

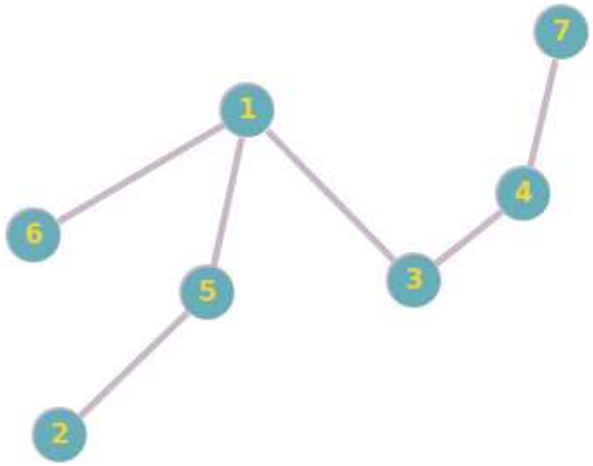
Opišimo ukratko koja je ideja ovog pseudokoda. Kada u opisu ovog postpuka kažemo "susjedi čvora u", mislimo na čvorove v za koje postoji brid (u, v). Algoritam nalazi čvorove grafa po slojevima. Slojeve numeriramo od 0 nadalje. Sloj 0 čini samo početni čvor s od kojeg započinjemo pretragu na grafu. Svi susjedi čvora s čine sloj 1 algoritma. Nadalje, za svaki $i \geq 1$ vrijedi: sloj $i + 1$ se sastoji od svih čvorova t koji su susjedi nekog čvora koji pripada sloju i, ali t nije susjed ni jednog čvora koji pripada nekom sloju $j \leq i - 1$. Ponekad, umjesto da kažemo da čvor v pripada sloju i, kažemo da je v na dubini ili udaljenosti i od početnog čvora s. Jasno, u algoritmu je dovoljno pamtiti samo posljednji pronađeni sloj i rezervirati spremnik za onaj idući. U danom pseudokodu je trenutni sloj predstavljen sa Q, a čvorove koje prepoznamo kao elemente idućeg sloja spremamo u Q'. Prilikom pronalaska čvora v koji će biti u idućem sloju ažuriramo i vrijednosti $parent[v] = u$ i $dist[v] = l + 1$, pri čemu je l oznaka trenutnog sloja, a u neki čvor iz trenutnog sloja Q koji ima susjeda v. Primijetimo da ovim postupkom, u slučaju povezanosti grafa, zaista gradimo razapinjuće stablo ulaznog grafa G ukorijenjeno u čvoru s. Prvo, povezanost grafa jamči da ćemo od čvora s doći do svih ostalih čvorova iz V. To znači da će naš algoritam svakom čvoru u nekom trenutku dodijeliti nekog roditelja (u `parent[]`), tj. dodati ga u stablo. Uočimo još da u ovom algoritmu ne možemo stvoriti ciklus na taj način jer se roditelj svakog čvora iz sloja $i \geq 1$ nalazi u sloju $i - 1$. Na taj način smo ispunili navedenu definiciju razapinjućeg stabla.



Slika 3.1: Primjer (neusmjerenog) grafa

Približimo slojevitost algoritma na jednostavnom primjeru grafa sa slike 3.1 s početnim čvorom označenim brojem 1. U sloju 0 se nalazi samo čvor 1. Njegovi izravni susjedni su čvorovi 3, 5 i 6 te stoga oni čine sloj 1. Prateći opis algoritma, zaključujemo da se u

sloju 2 nalaze čvorovi s oznakama 2 i 4. U sloju 3 se nalazi samo čvor 7 i on se nalazi na najvećoj udaljenosti (3) od početnog čvora. Primijetimo da ovaj graf ima cikluse (npr. 1-5-6) pa nije stablo. Razapinjuće stablo s korijenom 1 možemo prilikom izvršavanja algoritma konstruirati na sljedeći način: čvorovi 3, 5 i 6 zapamte čvor 1 kao svojeg roditelja jer je on čvor kojem su sva tri čvora izravni susjedi, a 1 se nalazi u sloju 0. Čvor 2 kao svog roditelja može zapamtiti ili čvor 5 ili čvor 6 (svejedno, ali uzmimo čvor 5 za primjer). Čvor 4 može kao roditelja zapamtiti samo čvor 3, a čvor 7 zapamti čvor 4 kao roditelja. Na taj način dobijemo razapinjuće stablo grafa sa slike 3.1 koje je prikazano na slici 3.2.



Slika 3.2: Razapinjuće stablo grafa sa slike 3.1.

Paralelni BFS

Pokušajmo sada paralelizirati ovaj algoritam. [6] Jasno je da ga ne smijemo paralelizirati na način da istovremeno radi na čvorovima koji pripadaju različitim slojevima. Takav pristup bi nekim čvorovima dodijelio prevelike udaljenosti, a neke bi možda i pogrešno proglasio nedostižnima. Dakle, preostaje nam paralelizirati sloj po sloj. Osnovna ideja je da fiksiramo broj radnih dretvi i tada prilikom izvođenja algoritma u svakom sloju ravnomjerno raspodjelimo posao među tim dretvama. Objasnimo prvo što podrazumijevamo pod poslom u svakom sloju. Cilj je za svaki čvor iz trenutnog sloja proći kroz sve njegove susjede i pritom prepoznati čvorove idućeg sloja (i zapamtiti ih u memoriji).

Ravnomjerna raspodjela koju ćemo koristiti je sljedeća. Uzmimo da polje Q sadrži sve čvorove trenutnog sloja. Definiramo polje cijelih brojeva σ, sa $\sigma[j] = \sum_{k=0}^{j} outdegree[Q[k]]$, pri čemu je $outdegree[x]$ izlazni stupanj čvora x. Neka je $m_l = \sigma[|Q|]$ zbroj izlaznih stupnjeva svih čvorova u trenutnom sloju. Čvor $Q[j]$ ćemo

sada dodijeliti dretvi s indeksom $\left\lceil \frac{\sigma[j]p}{m_l} \right\rceil$, pri čemu je p konstanta koja označava broj radnih dretvi. Jasno, bitnije je obratno preslikavanje, tj. za svaku dretvu moramo znati koje će ona čvorove iz trenutnog sloja obraditi. Najprije, iz navedenog pravila pridruživanja čvora dretvama (izraz $\left\lceil \frac{\sigma[j]p}{m_l} \right\rceil$), očito je da će svaka dretva obrađivati neki neprekinuti podniz iz Q (možda i prazan). To vrijedi zbog monotonosti (rasta) polja σ, a zbog monotonosti sada možemo i efikasno za svaku dretvu odrediti granice u Q binarnim pretraživanjem, tj. prvi i zadnji čvor kojeg će obraditi ta dretva. Kako je cilj da svaka dretva prepozna sve čvorove idućeg sloja i zapamti ih, za to trebamo dodatni spremnik. Sve čvorove idućeg sloja spremamo u polje Q'. No, to ne činimo na način da svaka dretva ubacuje čvorove u Q' instantno pri njihovom pronalasku jer u tom slučaju trebamo koristiti neki način sinkronizacije jer je polje Q' izloženo pisanju od strane više dretvi istovremeno. Umjesto toga, svaka dretva j će stvoriti lokalno polje $Qp[j]$ u koje će spremiti čvorove koje je ona pronašla. Na kraju, prilikom dolaska do sinkronizacijske točke kada su sve dretve odradile svoj posao za trenutni sloj, svaka dretva će iskopirati sadržaj svog $Qp[j]$ u globalni Q'. Ovdje ćemo izbjeći stanje natjecanja jer u ovoj fazi možemo preprocesirati za svaku dretvu koliko ona novih čvorova ubacuje i u skladu s tim joj dodijeliti (matematičkim izrazom) jedinstveni interval polja Q' koji će ona mijenjati (i samo ta dretva). Također, kao što smo imali u sekvencijalnoj verziji algoritma, i ovdje ćemo imati polja *dist* i *parent* koja imaju ista značenja. Ta polja su globalna i korisno je primjetiti da uopće ne moramo brinuti o istodobnom pokušaju pisanja, čak ni u slučaju istog elemenata u polju. Takvu pojavu smo u prvom poglavlju označili kao *benigno stanje natjecanja*. Pojasnimo zašto ovdje stanje natjecanja ne može nanijeti štetu sigurnosti koda. Zamislimo da je algoritam u fazi kada obrađuje i-ti ($i \geq 1$) sloj. Neka su T_1 i T_2 dvije dretve koje pokušavaju istom čvoru u promijeniti $d[u]$. Ovdje je ključna jednakost vrijednosti na koje one postavljaju tu varijablu, a to je $i + 1$. Iz tog razloga je svejedno koja će dretva upisati tu vrijednost i hoće li obje upisati. Jedino je bitno da barem jedna upiše, a za to nam ne treba nikakva sinkronizacija. Benignost stanja natjecanja u polju *parent* je opravdana sličnim razlogom. Svejedno je koji čvor od v, w će biti proglašen roditeljom čvora u iz sloja $i + 1$, dokle god su v i w u sloju i te postoje izlazni bridovi iz oba čvora u čvor u. U nastavku dajemo implementaciju opisanog paralelnog algoritma uz dodatna pojašnjenja koda.

```
1  class Graph{
2    size_t n; //broj cvorova
3    size_t m; //broj bridova
4    std::vector<size_t> begin;
5    std::vector<int> adj; //niz susjedstva
6
7    public:
8
9    Graph(size_t n_, size_t m_) : n{n_}, m{m_}{}
10
```

```
  void setEdges(std::vector<size_t> begin_, std::vector<int> adj_){
    adj = adj_;
    begin = begin_;
  }

  std::vector<int>::const_iterator beginNeighbor(const int v) const{
    return adj.cbegin() + begin[v];
  }

  std::vector<int>::const_iterator endNeighbor(const int v) const{
    return adj.cbegin() + begin[v + 1];
  }

  size_t getNodesCnt() const{
    return n;
  }

  size_t getEdgesCnt() const{
    return m;
  }
};
```

Listing 3.1: klasa koja reprezentira graf

U kodu 3.1. vidimo implementaciju klase pomoću koje ćemo pamtiti graf. Varijable n, m označavaju broj čvorova i broj bridova, redom. Graf će interno biti reprezentiran nizom susjedstva. To znači da će u vektoru adj svakom čvoru biti dodijeljen neprekinuti interval u kojem će biti oznake svih čvorova prema kojima on ima izlazni brid. U vektoru $begin$ su zapisani počeci takvih intervala za sve čvorove. Primjerice, $begin[u]$ označava indeks prvog izlaznog brida čvora u u vektoru adj. Funkcije članice `beginNeighbor` i `endNegihbor` vraćaju konstantne iteratore na spomenute granice u vektoru adj.

```
template<class Iterator, class F, class B>
void prefixSum(Iterator outBegin, Iterator outEnd, int iPE,
    int p, Iterator tmp, F f, B &bar){

  const size_t begin = (outEnd - outBegin) * iPE / p;
  const size_t end = (outEnd - outBegin) * (iPE + 1) / p;

  size_t sum = 0, i = begin;
  for(; i != end; i++)
    *(outBegin + i) = (sum += f(i));
  *(tmp + iPE) = sum;
  bar.arrive_and_wait();
  size_t a = 0;
  for(i = 0; i < iPE; i++){
    a += *(tmp + i);
  }
```

```
17    for(i = begin; i != end; i++){
18      *(outBegin + i) += a;
19    }
20  }
```

Listing 3.2: predložak funkcije `prefixSum`

Predložak funkcije `prefixSum` implementira paralelni algoritam računanja prefiksnih suma. U našem slučaju, uz podjelu posla na više dretvi izračuna vrijednosti definiranog polja σ. Kao argumente prima iteratore na početak i kraj niza na kojem radi (kod nas će biti σ), indeks dretve koja je pozvala funkciju, ukupni broj dretvi koje sudjeluju u računanju niza, iterator *tmp* za pomoćni račun, funkcijski objekt f i referencu na barijeru *bar* . U 5. i 6. liniji se računaju granice *begin* i *end* koje određuju raspon u polju Q za koji ova dretva računa vrijednosti funkcije σ. Napomenimo da je F parametar predloška samo zbog kvalitete apstrakcije, a kod nas će to biti lambda funkcija koja za svaki i vrati $outdegree[Q[i]]$. Petlja s početkom u 9. liniji tako zapravo radi sljedeće: $\sigma[i]$ = (sum += $outdegree[Q[i]]$), za svaki $i \in [begin, end)$. Nakon toga, u 12. liniji pomoću barijere pričekamo da sve dretve dođu do te točke. Primijetimo, tada će svaka dretva djelomično izračunati σ vrijednosti za svoj interval, ali ne u potpunosti. Pošto je dretva uzela u obzir samo čvorove iz svog intervala, potrebno je na te vrijednosti dodati ukupnu sumu koju su izračunale sve dretve koje su po indeksu manje od trenutne. To je zato što su sve dretve po indeksu manje od trenutne dretve bile zadužene za elemente polja σ koji se nalaze prije *begin*. Upravo tu će pomoći polje *tmp* koje će imati veličinu koja odgovara broju radnih dretvi p i $tmp[j]$ će pamtiti ukupnu sumu koju je izračunala dretva s indeksom j. Primijetimo da polje *tmp* nije izloženo stanju natjecanja i da svaka dretva postavlja svoju vrijednost neposredno prije dolaska do barijere, u 11. liniji. Nakon toga u varijablu a sumiramo *tmp* vrijednosti svih dretvi s indeksom manjim od trenutne dretve. Potom je preostalo tu vrijednost a dodati izračunatim σ vrijednostima u intervalu $[begin, end)$. U nastavku slijedi implementacija glavnog dijela paralelnog algoritma.

```
1  constexpr std::size_t CACHE_LINE_SIZE =
2    std::hardware_destructive_interference_size; //za vrijeme kompilacije
3
4  typedef std::pair<std::vector<int>,
5    char[CACHE_LINE_SIZE - sizeof(std::vector<int>)]> PaddedVector;
6
7  const int INVALID_ID = -1;
8
9  void parallelBFS(unsigned p, const Graph &g, const int s,
10     std::vector<int> &d, std::vector<int> &parent){
11
12   d.resize(g.getNodesCnt());
13   parent.resize(g.getNodesCnt());
14   std::vector<int> Q;
```

```
15  Q.reserve(g.getN());
16  Q.push_back(s);
17
18  std::vector<PaddedVector> Qp(p);
19  std::vector<size_t> sigma(1), tmp(p);
20  sigma.reserve(g.getNodesCnt());
21  std::atomic<bool> done(false);
22  std::barrier barijera(p);
23
24  std::vector<std::jthread> threads(p);
25  for(unsigned i = 0; i < p; i++){
26    threads[i] = std::jthread([&](const unsigned iPE){
27        const size_t beginl = iPE * g.getNodesCnt() / p + 1;
28        const size_t endl = (iPE + 1) * g.getNodesCnt() / p + 1;
29
30        std::fill(d.begin() + beginl, d.begin() + endl,
31            std::numeric_limits<int>::max());
32        std::fill(parent.begin() + beginl,
33            parent.begin() + endl, INVALID_ID);
34
35        if(s >= beginl && s < endl){
36          d[s] = 0; parent[s] = s;
37        }
38
39        int l = 1;
40
41        for(; !done; l++){
42          prefixSum(sigma.begin(), sigma.end(), iPE, p, tmp.begin(),
43              [&](int j){ return g.endNeighbor(Q[j]) -
44                  g.beginNeighbor(Q[j]);}, barijera);
45
46          barijera.arrive_and_wait();
47
48          Qp[iPE].first.clear();
49          size_t ml = sigma.back(); //ml je suma svih outdegree-ova
50          Qp[iPE].first.reserve(2 * ml / p);
51
52          //std::upper_bound implementira binarno pretrazivanje
53          size_t curQ = std::upper_bound(sigma.cbegin(),
54              sigma.cend(), iPE * ml / p) - sigma.cbegin();
55          const size_t endQ = std::upper_bound(sigma.cbegin(),
56              sigma.cend(), (iPE + 1) * ml / p) - sigma.cbegin();
57
58          //po izlaznim bridovima svih mojih cvorova
59          for(; curQ != endQ; curQ++){
60            const int u = Q[curQ];
61            auto vIter = g.beginNeighbor(u);
```

```
62              auto vEnd = g.endNeighbor(u);
63              for(; vIter != vEnd; vIter++){
64                const int v = *vIter;
65                if(parent[v] == INVALID_ID){
66                  parent[v] = u;
67                  d[v] = l;
68                  Qp[iPE].first.push_back(v);
69                }
70              }
71            }
72
73            barijera.arrive_and_wait();
74
75            size_t outPos = 0;
76            for(int j = 0; j < iPE; j++){
77              outPos += Qp[j].first.size();
78            }
79            if(iPE == p - 1){
80              Q.resize(outPos + Qp[iPE].first.size());
81              sigma.resize(Q.size());
82              if(Q.empty()){
83                done = true;
84              }
85            }
86
87            barijera.arrive_and_wait();
88
89            std::copy(Qp[iPE].first.cbegin(), Qp[iPE].first.cend(),
90                Q.begin() + outPos);
91            barijera.arrive_and_wait();
92
93          }
94
95      }, i);
96    }
97
98  }
```

Listing 3.3: glavni dio algoritma

Funkcija `parallelBFS` kao argumente prima broj radnih dretvi za pretraživanje, referencu na objekt tipa `Graph`, početni čvor s od kojeg započinjemo pretragu u širinu, i reference na vektore d (ima ulogu *dist* u opisu algoritma) i *parent* u koje ćemo spremiti udaljenosti do čvorova te roditelje čvorova u stablu koje gradimo prema opisanim pravilima. Linije 12-16 inicijaliziraju vektore i pomoćne varijable za izvedbu algoritma. U 18. liniji svaka dretva inicijalizira svoj vektor u koji će spremati čvorove za idući sloj algoritma. Kasnije ćemo objasniti prethodno definirani tip `PaddedVector`. U 21. liniji je

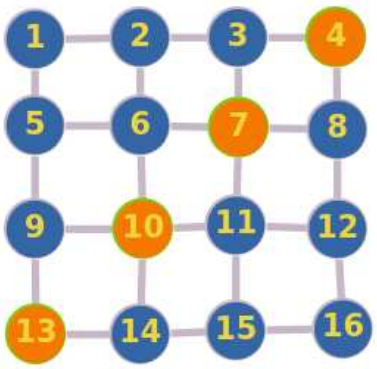
inicijalizirana varijabla *done* tipa `std::atomic<bool>` koja ima ulogu zastavice za signaliziranje kraja algoritma. Nju čita više dretvi, a biti će i upisana vrijednost `true` pa je zbog jednostavnosti atomska varijabla, a ne običan `bool` uparen s lokotom. U 22. liniji inicijaliziramo `std::barrier` varijablu *barijera* koja će imati ulogu da se u nekim točkama koda osigura da su sve dretve došle do tih mjesta. Početna vrijednost je postavljena na p, tj. broj dretvi. Prisjetimo se, svakim dolaskom njenog brojača do 0 on se resetira natrag na vrijednost p. U 24. liniji se definira vektor koji će sadržavati objekte tipa `std::jthread`, njih p. Destrukcijom tog vektora pozvat će se i destruktori svih elemenata tog vektora i u njima će se pozvati `join()` metode. U 26. liniji počinje definicija i-te lambda funkcije koja će opisivati rad dretve i, za sve $i \in [0, p-1]$. U linijama 27-37 dretve podjele posao u samoj inicijalizaciji vektora d i *parent*. Dretva koja je odgovorna za inicijalizaciju vrijednosti tih vektora u početnom čvoru s ujedno postavi $d[s] = 0$, $parent[s] = s$. U 39. liniji se inicijalizira varijabla l koja će biti lokalna za svaku dretvu i njena vrijednost je oznaka sloja koji se sljedeći računa (sloj 0 je obrađen na samom početku: sastoji se samo od čvora s, pa je sloj 1 prvi sljedeći). U 41. liniji započinje petlja koju vrti svaka dretva dok god algoritam nije završen, a petlja služi za prebacivanje u idući sloj. Na početku obrade svakog sloja pomoću opisanog predloška funkcije `prefixSum` na efikasan način računamo vrijednosti polja σ (u kodu pod nazivom *sigma*) koje nam služi za pravednu raspodjelu posla u trenutnom sloju. U 46. liniji pomoću barijere pričekamo da sve dretve stignu do te točke, tj. izračunaju vrijednosti svojeg intervala u polju *sigma*. U varijablu *ml* se sprema suma svih izlaznih stupnjeva čvorova u trenutnom sloju. U linijama 48. i 50. svaka dretva očisti, pa alocira svoje privremeno polje u koje će spremati čvorove koje prepozna kao elemente idućeg sloja. Primijetimo da smo u 50. liniji rezervirali svakoj dretvi veći vektor nego će ona trebati. To je učinjeno da bimo prilikom ubacivanja u vektor izbjegli dodatne alokacije koje `std::vector<int>` interno radi. Nakon toga, u linijama 53. i 55. svaka dretva računa granice svog intervala ($[curQ, ..., endQ]$) u trenutnom polju Q za koji će raditi posao i to pomoću binarnog pretraživanja i artitmetičkog izraza kojeg smo naveli ranije. U linijama 59-71 svaka dretva u svojem intervalu polja Q prolazi kroz izlazne bridove svih čvorova u intervalu i uvjetom u 65. liniji prepoznaje čvorove za idući sloj. Upravo u tom `if` bloku imamo spomenuto benigno stanje natjecanja. Preostalo je trenutni sloj Q zamijeniti sljedećim. U 73. liniji stoga pričekamo da sve dretve obave svoj zadatak prolaska po odgovarajućim izlaznim bridovima i u lokalne spremnike $Qp[j]$ spreme kandidate za idući sloj. U linijama 75-78 se računa indeks u globalnom Q', od kojeg nadalje će trenutna dretva stavljati čvorove koje je ona prepoznala kao elemente idućeg sloja. Zbog jednostavnosti smo fiksirali dretvu s indeksom $p-1$ koja će u linijama 80-84 inicijalizirati varijable i objekte potrebne za sljedeći sloj i provjeriti je li algoritam možda gotov. Jasno, algoritam je gotov ako ni jedna dretva nije pronašla čvorove kandidate za idući sloj. U 87. liniji imamo barijeru da bismo osigurali da su sve dretve došle do te točke (zapravo je najbitnije da ovdje dođe samo dretva $p-1$ jer trebamo imati ispravno alociran novi vektor

Q, ali nećemo puno dobiti na performansama zbog barijera koje su i prije i nakon ove) jer slijedi kopiranje čvorova iz lokalnih vektora $Qp[j]$ koji su prepoznati kao elementi idućeg sloja, u globalno polje Q (na taj Q za idući sloj smo se do malo prije referirali pomoću Q'. Za prelazak u novi sloj računanja potrebno je da sve dretve završe sav posao sa trenutnim slojem te stoga u 91. liniji ponovno koristimo barijeru. Primijetimo još da zbog stanja natjecanja koje smo proglasili benignim, postoji mogućnost da u idućem sloju imamo neki čvor koji se pojavljuje više puta. To se u pravilu neće događati često, ali i kad se desi, ne utječe na korektnost algoritma, već samo performanse.

Pojasnimo još ulogu tipa `PaddedVector` kojeg sve dretve koriste kao lokalni spremnik za čvorove koje će stavljati u idući sloj. Taj tip je par koji se sastoji od spremnika `std::vector<int>` u koji zapravo i spremamo te čvorove i niza od onoliko bajtova koliko ih fali od veličine praznog vektora do $CACHE\_LINE\_SIZE$. Potrebno je dakle opravdati drugi član tog para koji se očito nigdje ne koristi eksplicitno u kodu. Naravno, ovo je primjer situacije gdje žrtvujemo memoriju za performanse. U programiranju aplikacija s dijeljenom memorijom, jedan od problema s kojim se suočimo je lokalno pamćenje vrijednosti varijabli u procesorskoj jedinici radi bržeg ponovnog dohvata (eng. *cash*). Ako jezgra i upiše u varijablu b vrijednost x, a druga jezgra j ima spremljenu kopiju varijable b u brzoj memoriji (lokalnoj za tu jezgru), ona postaje nevažeća i potrebna je komunikacija među jezgrama i i j da bi se postigla konzistencija. Ta komunikacija postaje usko grlo i jedan je od glavnih razloga ograničene skalabilnosti programa s dijeljenom memorijom. Ova komunikacija među jezgrama se događa čak i u situacijama kada dvije dretve ne pristupaju nužno istoj memorijskoj ćeliji (uočimo, u našem kodu svaka dretva i pristupa samo svojem vektoru $Qp[i]$ pa stoga definitivno ne dijeli memoriju s nekom drugom dretvom). Komunikacija među jezgrama će se inicirati čim one pristupaju ćelijama koje su dovoljno blizu jedna druge, tj. nalaze se u istoj liniji cash memorije (eng. *cache line*). Ta pojava se naziva *lažno dijeljenje* (eng. *false sharing*) i u našem primjeru može biti izbjegnuta na način da svaki $Qp[j]$ bude dovoljno velik da popuni cijelu cash liniju. Upravo to je uloga u dodanog niza bajtova (za različite arhitekture računala će cash linije biti potencijalno drugačijih veličina). Od standarda C++17 možemo za vrijeme kompilacije pročitati vrijednost `std::hardware_destructive_interference_size` iz zaglavlja `<new>` koja nam daje minimalnu količinu memorije koju moramo imati između dva objekta da bismo izbjegli lažno dijeljenje. [2] Pomoću te kompilacijske konstante smo i definirali konstantu $CACHE\_LINE\_SIZE$. Sada svaka dretva može čitati i pisati u svoj vektor i takve opercije neće inicirati komunikaciju među jezgrama jer one ne spremaju iste cash linije u svoje brze memorije. [6]

Performanse paralelnog algoritma

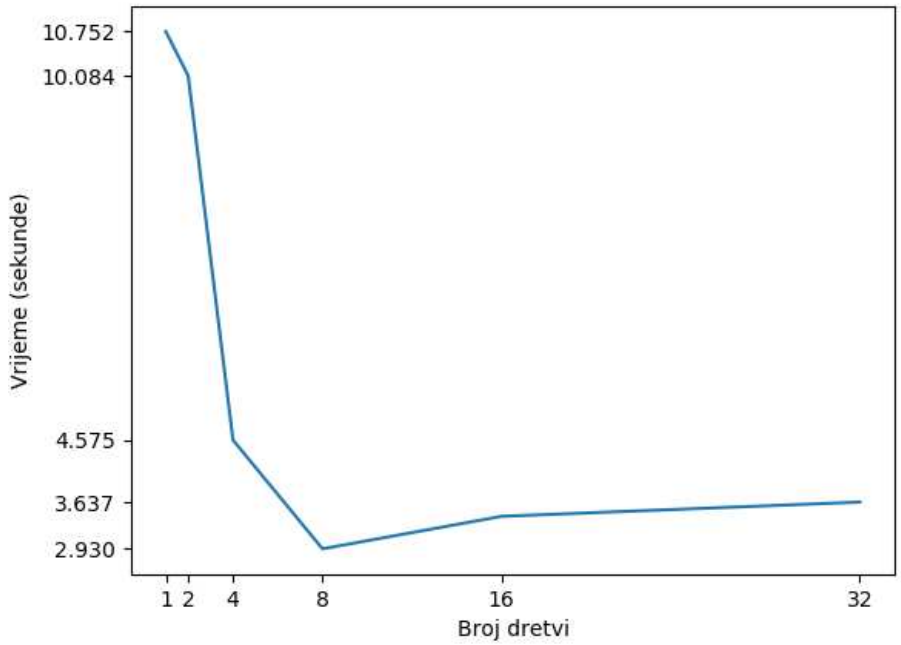
Korištenje ovog algoritma u nekim slučajevima neće poboljšati performanse (štoviše, u primjeru nekih grafova može pogoršati izvođenje: primjerice, u linijskom grafu gdje svaki sloj algoritma ima jedan čvor, a mi smo angažirali $p \geq 8$ (ili više) dretvi za obradu), ali postoje specijalni slučajevi grafova u kojima se pokazuje korisnim. To su grafovi u kojima broj čvorova u puno slojeva bude velik - da bi podjela posla imala smisla. Graf na kakvom smo mi testirali je tzv. *tablični graf* (eng. *grid graph*) sa 7000 redaka po 7000 čvorova - ukupno 49 000 000 čvorova. Na slici 3.3 se nalazi primjer tabličnog grafa sa 4 retka i 4 stupca. U takvom grafu broj čvorova po sloju najprije raste, a zatim pada. Narančastom bojom su označeni čvorovi koji se nalaze u sloju 3 (ako je početni čvor 1), i to je sloj s najviše čvorova za taj graf i početni čvor 1. Nadalje, algoritam smo testirali na računalu `prosper.math.hr` čije su relevantne specifikacije: sockets = 1, cores per socket = 24, threads per core = 2. Testirali smo izvođenje algoritma pomoću 1, 2, 4, 8, 16 i 32 radnih dretvi, a rezultati su prikazani na dijagramu sa slike 3.4. Prosječna vremena izvršavanja (od 20 pokretanja) se nalaze u tablici 3.1, a na slici 3.4 su rezultati vizualno prikazani. Program koji se izvodi pomoću jedne dretve je u prosjeku trebao 10.752 sekundi. Tek malo ubrzanje je pokazao program s 2 pokrenute dretve: 10.084 sekundi. Ogromno ubrzanje (dvostruko) je pokazao program koji je pokrenuo 4 dretve za obradu (4.575 sekundi). Gotovo dvostruko brži od njega je bio program s 8 pokrenutih dretvi (2.930 sekunde) i on je ujedno pokazao najbolju izvedbu. Za veći broj pokrenutih dretvi program ne pokazuje dobre performanse na ovom primjeru. Tako je program koji je pokrenuo 16 dretvi trebao prosječno vrijeme 3.421 sekundi, a program sa 32 dretve 3.637 sekundi.



Slika 3.3: Primjer tabličnog grafa dimenzija 4x4

Tablica 3.1: Rezultati paralelnog BFS-a

Broj pokrenutih dretvi	Prosječno vrijeme izvršavanja (sekunde)
1	10.752
2	10.084
4	4.575
8	2.930
16	3.421
32	3.637



Slika 3.4: Vrijeme izvođenja BFS-a u ovisnosti o broju korištenih dretvi

3.4 Topološko sortiranje

U ovoj točki se bavimo često korištenim algoritmom na specifičnoj klasi grafova: usmjereni grafovi bez ciklusa (eng. *directed acyclic graph*, skraćeno *DAG*). U takvim grafovima uvijek postoji (ne nužno jedinstven) poredak vrhova sa svojstvom da za sve bridove grafa (u, v) vrijedi: čvor u se u poretku nalazi prije čvora v. Takav poredak je poznat kao *topološko sortiranje grafa*. [6] Topološko sortiranje grafa ima primjenu u rješavanju raznih problema na grafovima u kojima je bitan poredak izračunavanja. Takvi primjeri su dinamičko programiranje te razne heuristike za rješavanje problema za koje ne znamo efikasne algoritme. U tom smislu je ovaj algoritam zapravo koristan alat za neke složenije algoritme na grafovima. Mi ćemo u ovoj točki zbog jednostavnosti, pretpostaviti da su grafovi u ulazu našeg algoritma usmjereni i bez ciklusa.

Sekvencijalni algoritam

Ponovno opisujemo najprije sekvencijalnu verziju algoritma, a zatim ćemo ga paralelizirati. U ulazu algoritma imamo neki usmjereni graf (V, E) za koji pretpostavljamo da nema ciklusa. Također smo kao ulaz dodali niz *indegree* u kojem pretpostavljamo preprocesirane ulazne stupnjeve svih čvorova grafa. Izlaz algoritma će biti jedno topološko sortiranje grafa koje vraćamo kroz niz *topo_sort*. Najprije taj niz inicijaliziramo kao prazan. Potom inicijaliziramo polja Q, Q' kao prazna i ona će služiti kao privremeni spremnici. Zatim u Q dodajemo sve čvorove koji su spremni za obradu, a to su oni kojima je ulazni stupanj na početku 0. Takve čvorove usput ubacimo i u *topo_sort*. Primijetimo da time ne narušavamo svojstvo topološkog sortiranja, bez obzira na poredak u kojem ubacimo takve čvorove. Nakon toga vrtimo petlju dokle god je Q neprazan. U svakom koraku takve petlje radimo sljedeće. Svakom čvoru iz Q prođemo po svim izlaznim bridovima te smanjujemo za 1 ulazni stupanj svim čvorovima na drugim krajevima takvih bridova. Čvorove čiji ulazni stupanj nakon takve operacije postane nula ubacujemo u Q' i *topo_sort*. Primijetimo ovdje ponovno da zbog nepostojanja ciklusa, ni jedan čvor kojeg ubacujemo u *topo_sort* u ovom koraku, nema izlaznih bridova prema čvorovima koji su već u nizu *topo_sort* te zbog toga i dalje vrijedi definicijsko svojstvo topoploškog sortiranja. Kad smo to napravili za sve čvorove iz Q, prebacimo sadržaj od Q' u Q, ispraznimo Q' idemo u sljedeći korak petlje. U nastavku se nalazi i pseudokod upravo opisanog algoritma.

Algorithm 2 Topološko sortiranje

```
function TopoSort(V, E, indegree)
    topo_sort ← []
    Q ← []
    Q' ← []
    for each node ∈ V do
        if indegree[node] = 0 then
            Q ← Q ∪ {node}
            topo_sort ← topo_sort ∪ {node}
        end if
    end for
    while Q ≠ [] do
        for each u ∈ Q do
            for each (u, v) ∈ E do
                indegree[v] ← indegree[v] − 1
                if indegree[v] = 0 then
                    Q' ← Q' ∪ {v}
                    topo_sort ← topo_sort ∪ {v}
                end if
            end for
        end for
        (Q, Q') ← (Q', [])
    end while
    return topo_sort
end function
```

Paralelno topološko sortiranje

Paralelizacija ovog algoritma će idejno i implementacijski biti vrlo slična prethodnom. U nastavku slijedi implementacija glavnog dijela opisanog algoritma, uz ispuštanje već navedenih definicija predloška funkcije `prefixSum`, klase `Graph` i još ponekih varijabli istih imena i tipova kao u prethodnom kodu. Nakon toga slijede dodatna objašnjena koda.

```
{
  std::vector<std::jthread> pool(P);
  //polja indegree i outdegree racunamo kod unosa grafa

  size_t last = 0;

  for(int p = 0; p < P; p++){
    std::jthread t{[&](const unsigned iPE){
      const size_t beginl = iPE * G.getNodesCnt() / P + 1;
      const size_t endl = (iPE + 1) * G.getNodesCnt() / P + 1;

      Qp[iPE].first.clear();
      Qp[iPE].first.reserve(2 * (endl - beginl) + 1);
      for(size_t i = beginl; i < endl; i++){
        if(indegree[i] == 0){
          Qp[iPE].first.push_back(i);
        }
      }

      barijera.arrive_and_wait();

      size_t outPos = 0;
      for(int j = 0; j < iPE; j++){
        outPos += Qp[j].first.size();
      }

      if(iPE == P - 1){
        Q.resize(outPos + Qp[iPE].first.size());
        last = Q.size();
      }

      barijera.arrive_and_wait();
      std::copy(Qp[iPE].first.cbegin(), Qp[iPE].first.cend(),
          Q.begin() + outPos);

      std::copy(Qp[iPE].first.cbegin(), Qp[iPE].first.cend(),
          topo_sort.begin() + outPos);

      barijera.arrive_and_wait();

```

```
41      int l = 1;
42      for(; !done; l++){
43        prefixSum(sigma.begin(), sigma.end(), iPE, P, tmp.begin(),
44            [&](int j) -> int{return outdegree[Q[j]];}, bar);
45        barijera.arrive_and_wait();
46        Qp[iPE].first.clear();
47        size_t ml = sigma.back();
48        Qp[iPE].first.reserve(2 * ml / p);
49
50        size_t curQ = std::upper_bound(sigma.cbegin(),
51          sigma.cend(), iPE * ml / p) - sigma.cbegin();
52        const size_t endQ = std::upper_bound(sigma.cbegin(),
53          sigma.cend(), (iPE + 1) * ml / p) - sigma.cbegin();
54
55        for(; curQ != endQ; curQ++){
56          const int u = Q[curQ];
57          auto vIter = G.beginNeighbor(u);
58          auto vEnd = G.endNeighbor(u);
59          for(; vIter != vEnd; vIter++){
60            const int v = *vIter;
61            if(--indegree[v] == 0){ //ovdje uvijek samo 1 dretva uspije
62              Qp[iPE].first.push_back(v);
63            }
64          }
65        }
66
67        barijera.arrive_and_wait();
68
69        outPos = 0;
70        for(int j = 0; j < iPE; j++){
71          outPos += Qp[j].first.size();
72        }
73        if(iPE == P - 1){ //samo jedna dretva alocira sredstva
74          Q.resize(outPos + Qp[iPE].first.size());
75          sigma.resize(Q.size());
76          if(Q.empty()){
77            done = true;
78          }
79        }
80
81        barijera.arrive_and_wait();
82
83        std::copy(Qp[iPE].first.cbegin(), Qp[iPE].first.cend(),
84            Q.begin() + outPos);
85        std::copy(Qp[iPE].first.cbegin(), Qp[iPE].first.cend(),
86            topo_sort.begin() + last + outPos);
87
```

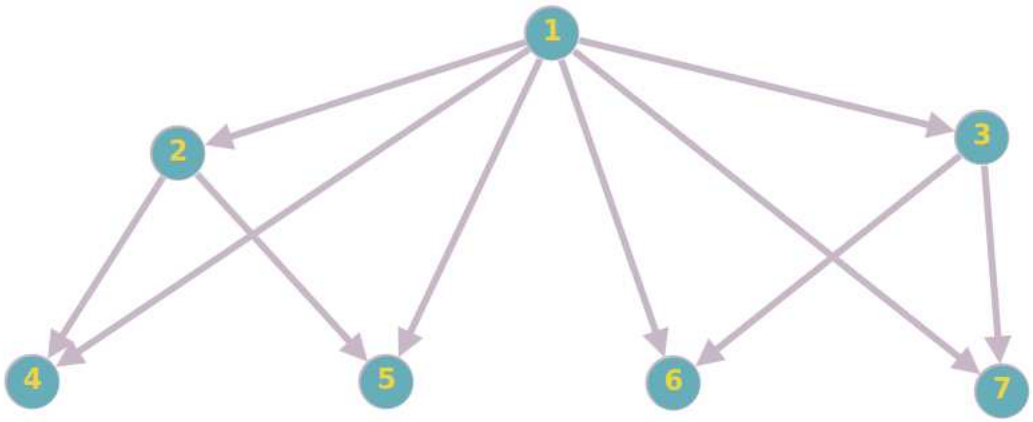
```
88          barijera.arrive_and_wait();
89          if(iPE == P - 1)
90            last += outPos + int(Qp[iPE].first.size());
91        }
92
93      }, p};
94
95      pool[p] = std::move(t);
96    }//kraj for petlje
97  }
```

Listing 3.4: paralelni obilazak DAG-a

Ponovno na početku konstruiramo skup od P radnih dretvi tipa `std::jthread`. U 9. i 10. liniji svaka dretva računa granice svojeg intervala iz $[1, ..., |V|]$ za koji će proći po svim čvorovima te ubaciti u svoj Qp one kojima je ulazni stupanj 0. Od 20. do 39. linije je već viđen postupak ubacivanja iz lokalnog spremnika Qp u globalni Q uz dodatak da se ovdje elementi ubacuju i u niz *topo_sort* na sasvim analogan način. Nakon toga se pokreće petlja u 41. liniji koja za trenutni skup Q raspodjeli posao među dretvama na sasvim jednak način kao što smo imali kod prijašnjeg algoritma - u obzir se uzimaju izlazni stupnjevi čvorova iz Q. Sada svaka dretva za svoj interval čvorova u Q radi sljedeće: za svaki čvor u prođe po svim njegovim izlaznim bridovima i za svaki brid (u, v) smanji ulazni stupanj čvora v za 1 (pravimo se da smo obrisali taj brid iako ga ne brišemo iz `Graph` objekta). Naglasimo samo da je u kodu *indegree* tipa `std::vector<std::atomic<int>>` i na taj način ostvarujemo atomarnost i izbjegavamo stanje natjecanja. Preciznije, uvjet u 61. liniji će biti istinit za točno jednu dretvu. Naime, izraz `--indegree[`v`]` će smanjiti tu atomsku varijablu za 1 i vratiti vrijednost koju je ona poprimila neposredno nakon toga, bez obzira što je u međuvremenu neka druga dretva već pokušala izvršiti dekrement na istom čvoru. Kod BFS-a smo u ovom djelu koda imali benigno stanje natjecanja koje nije narušavalo korektnost algoritma - bilo je svejedno hoće li čvor x u idući sloj biti ubačen od jedne ili više dretvi, no ovdje je nužno izbjeći stanje natjecanja. Kada bi dvije ili više različite dretve ubacile isti čvor x u budući skup Q (preko svojih Qp - ova), tada bi u idućem koraku algoritma svim susjedima čvora x bio smanjen ulazni stupanj za više nego što je potrebno (potencijalno bi $indegree[w]$ za neke w postao negativan), a to u ovom slučaju narušava korektnost algoritma. Varijabla *last* u ovom kodu je tipa `size_t` i samo služi kao indeks na prvo slobodno mjesto u vektoru *topo_sort* zbog lakšeg istovremenog ubacivanja elemenata iz više dretvi. Ostatak koda ponavlja operacije koje su već opisane u sekvencijalnoj verziji ovog algoritma ili kod paralelnog BFS-a.

Performanse algoritma

Paralelizirani algoritam će ponovno biti koristan ako nam većina skupova Q iz algoritma ima dovoljan broj čvorova i izlaznih bridova da podjela posla ima smisla. U tu svrhu smo testirali kod na sljedećem umjetnom primjeru. Graf je oblika usmjerenog potpunog binarnog stabla, tj. svaki čvor i (osim listova) ima izlazne bridove prema $2 \cdot i$ i $2 \cdot i + 1$. Zbog smanjenja trivijalnosti primjera, svakom čvoru dodamo izlazne bridove i prema čvorovima $2 \cdot i + 2, ..., min(V, 2 \cdot i + 500)$, pri čemu je $V = 2^{21}$ broj čvorova grafa. Broj bridova konačno ispadne 525 273 825. Na slici 3.5 se nalazi početni komad grafa na kojem testiramo.

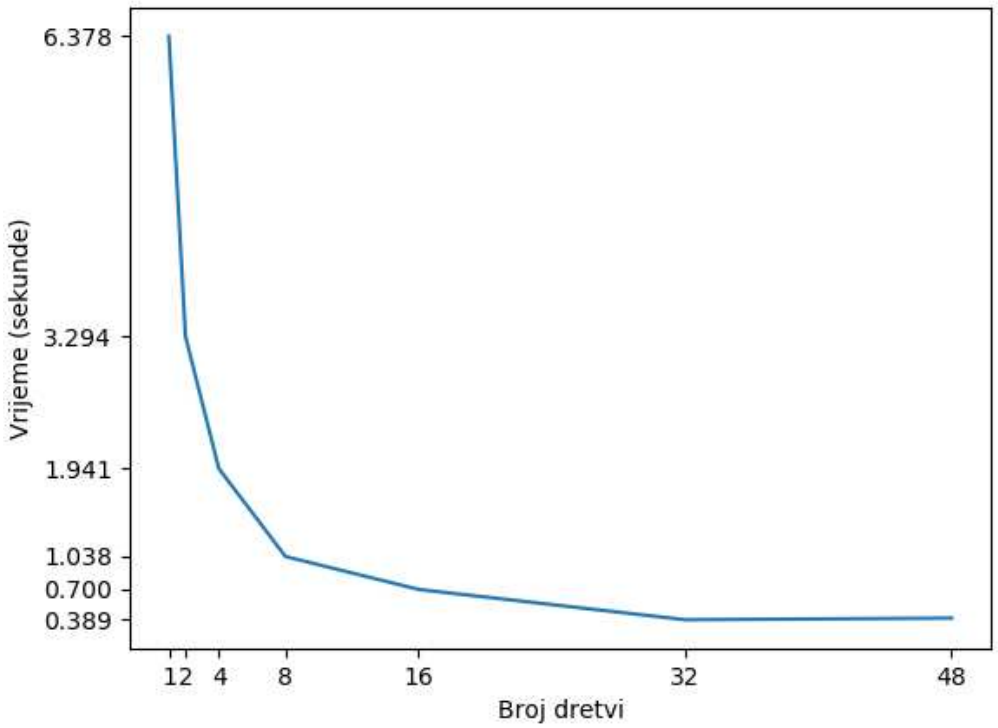


Slika 3.5: Početni komad velikog grafa

U tablici 3.2 i na grafu sa slike 3.6 je prikazano skaliranje algoritma, tj. njegovo izvođenje na različitom broju radnih dretvi. Testiranje ponovno provodimo na računalu `prosper.math.hr` čije smo relevantne specifikacije naveli ranije. Također, zbog nepredvidivosti ponašanja višedretvenih programa ponovno računamo prosječno vrijeme od 20 izvođenja nad istim ulaznim podacima. Pokretanjem samo jedne radne dretve, programu je potrebno 6.378 sekundi za računanje. Zatim se vrijeme izvođenja otprilike raspolavlja svakim dvostrukim povećanjem broja radnih dretvi (2, 4, 8, 16, 32). Najbrže izvođenje dobivamo sa 32 radne dretve (0.389 sekundi), dok sa 48 dretvi imamo prosječno vrijeme od 0.406 sekundi. Daljnjim povećanjem broja dretvi dobivamo još gore performanse. Ovi rezultati imaju smisla jer je produkt $\#sockets \cdot \#cores\_per\_socket \cdot \#threads\_per\_core$ za navedeno računalo jednak 48 pa će pokretanje većeg broja dretvi od 48 sigurno uzrokovati čestu promjenu konteksta na procesnim jedinicama.

Tablica 3.2: Rezultati paralelnog topološkog sortiranja

Broj pokrenutih dretvi	Prosječno vrijeme izvršavanja (sekunde)
1	6.378
2	3.294
4	1.941
8	1.038
16	0.700
32	0.389
48	0.406



Slika 3.6: Vrijeme računanja topološkog sortiranja u ovisnosti o broju korištenih dretvi

3.5 Boruvkin algoritam

U današnje vrijeme su najpoznatiji algoritmi za računanje minimalnog razapinjućeg stabla Kruskalov i Primov algoritam. Ipak, ni jedan od njih nije prikladan za paralelizaciju. Primov algoritam u svakom koraku nalazi novi čvor, čiji izbor ovisi o već izabranim čvorovima, dok Kruskalov algoritam u svakom koraku nalazi najlakši brid koji spaja neke dvije razdovjene komponente - izbor tog brida opet ovisi o prethodno izabranima. U ovoj točki se bavimo *Boruvkinim algoritmom* [6] za pronalazak minimalnog razapinjućeg stabla i on se vrlo jednostavno podvrgava paralelizaciji. Zbog jednostavnosti implementacije ćemo imati dvije pretpostavke na graf kojem računamo minimalno razapinjuće stablo: graf je povezan i težine bridova grafa su međusobno različite. Bez prve pretpostavke izlaz algoritma gubi smisao. Druga se može izostaviti, no prilikom implementacije će biti potrebno paziti na dodatne detalje. Naglasit ćemo u opisu algoritma gdje koristimo tu pretpostavku. Podrazumijevamo da su grafovi u ulazu neusmjereni i oznaka $\{u, v, c\}$ se odnosi na brid između čvorova u i v težine c. Pseudokodovi i implementacija u ovom radu su motivirani knjigom [6] te radom [7].

Sekvencijalni algoritam

Objasnimo za početak nekoliko dodatnih pojmova koje ćemo koristiti u daljnjem opisu algoritma. *Komponentom povezanosti* neusmjerenog grafa G smatramo povezani neusmjereni graf K čiji skup čvorova i skup bridova su podskupovi skupa čvorova grafa G i skupa bridova grafa G, redom. Za svaku komponentu povezanosti ćemo smatrati da ima svog čvora predstavnika (svi čvorovi svake komponente znaju tko je njihov predstavnik). Kada u narednim rečenicama kažemo "komponenta povezanosti v", mislimo na komponentu povezanosti kojoj je čvor v predstavnik. *Rez* u grafu $G = (V, E)$ je particija skupa čvorova $(S, V \setminus S)$, pri čemu su $S, V \setminus S$ oba neprazni. Za svaki rez $(S, V \setminus S)$ definiramo skup $E_S = \{\{u, v, c\} \in E : u \in S, v \in V \setminus S\}$.

Boruvkin algoritam radi u više iteracija i minimalno razapinjuće stablo izgrađuje postepeno. Kreće od praznog skupa T odabranih bridova grafa koji će ući u razapinjuće stablo, a završava sa $(|V| - 1)$ - članim skupom T, pri čemu je V skup čvorova grafa. Uz samu izgradnju skupa T ćemo pratiti i broj komponenata povezanosti koje implicira skup bridova T. Jasno, kada je $T = \emptyset$, broj komponenata povezanosti je $|V|$, jer ni jedna dva čvora nisu povezana bridom. S druge strane, kada broj različith komponenata povezanosti koje implicira skup bridova T postane 1, znamo da smo izgradili razapinjuće stablo inicijalnog grafa te algoritam završava. Primijetimo, na sličan način radi i Kruskalov algoritam, gdje u uzlaznom poretku po težinama ubacujemo bridove u T, ali samo one koji imaju svojstvo da će spojiti čvorove koji su u tom trenutku u različitim komponentama povezanosti. Posebnost Boruvkinog algoritma te prostor za paralelizaciju leži u mogućnosti dodavanja većeg broja bridova odjednom u T.

Osnovna ideja algoritma je, da u svakoj iteraciji, dokle god trenutni skup odabranih bridova za razapinjuće stablo T implicira barem dvije ili više različitih komponenti povezanosti, za svaku komponentu povezanosti pronađemo brid najmanje težine (jedinstven po pretpostavci) koji spaja neki čvor te komponente s nekim čvorom neke druge komponente i taj brid ubacimo u razapinjuće stablo (moramo samo paziti da ne ubacujemo cikluse, ali vidjet ćemo u nastavku da tu zapravo nemamo ozbiljnijih problema, upravo zbog navedenog kriterija odabira bridova). Korektnost Boruvkinog (i Kruskalovog) algoritma se opravdava sljedećom tvrdnjom koja je u literaturi poznata kao *svojstvo reza* (eng. *cut property*).

Lema 3.5.1. *Neka je $(S, V \setminus S)$ neki rez grafa $G = (V, E)$ te skup E_S definiran kao prije. Neka je $e \in E_S$ brid najmanje težine. Neka je sada $T' \subseteq E$ takav da je T' podskup bridova nekog minimalnog razapinjućeg stabla od G i T' ne sadrži ni jedan brid iz E_S. Tada je i skup bridova $T' \cup \{e\}$ također podskup bridova nekog minimalnog razapinjućeg stabla od G.*

Dokaz. [6] □

Primijetimo da nam lema 3.5.1 dopušta sljedeću operaciju: ukoliko imamo k različitih komponenti povezanosti $v_1, v_2, ..., v_k$ i za svaku od njih definiramo brid e_i kao brid minimalne težine od svih bridova koji spajaju neki čvor x iz komponente v_i sa čvorom y iz komponente v_j, za $i \neq j$, tada možemo dodati sve bridove e_i, $i = 1, ..., k$ u razapinjuće stablo i još uvijek ćemo "biti na putu do nekog minimalnog razapinjućeg stabla" ukoliko konzistentno provodimo ovaj postupak. U ovom dijelu smo prešutno iskoristili i uvodnu pretpostavku na graf: svi bridovi su različitih težina. Ta pretpostavka nam garantira da opisanim odabirom bridova $e_1, ..., e_k$ nećemo dodati ciklus u razapinjuće stablo. Mogu se eventualno pojaviti neki ciklusi duljine 2 - tada je $e_i = e_j$, za neke $i \neq j$. Primijetimo da takvi ciklusi zapravo nisu štetni, jer kolekciju bridova u minimalnom razapinjućem stablu tumačimo kao skup, pa dodavanjem e_i i e_j zapravo dodajemo jedan brid u razapinjuće stablo. Ciklusi sa većim brojem različitih čvorova od 2 nisu mogući i to vidimo na sljedeći način: neka je $(v_1, v_2, ..., v_{m-1}, v_m = v_1)$ neki ciklus od $m - 1$ različitih čvorova, za $m \geq 4$ i neka su $e_2, e_3, ..., e_m$ bridovi u tom ciklusu takvi da $e_i = (v_{i-1}, v_i)$ za $2 \leq i \leq m$, te su svi e_i odabrani na prethodno opisani način. Neka je $maxi \geq 2$ indeks brida najveće težine u tom ciklusu. Obratimo sada pozornost na brid $e_{maxi} = (x, y)$. Zbog činjenice da su x i y dio ciklusa sa barem 3 čvora, slijedi da je svaki od njih incidentan s još točno jednim bridom osim e_{maxi}, tj. x je incidentan s nekim bridom $e_{x_i} \neq e_{maxi}$, a y s nekim bridom $e_{y_i} \neq e_{maxi}$. Iz definicije e_{maxi} slijedi da i e_{x_i} i e_{y_i} imaju manju težinu od e_{maxi}. To daje kontradikciju s odabirom brida e_{maxi} u definiranom postupku neposredno nakon leme 3.5.1.

Dakle, ako počnemo graditi razapinjuće stablo od nule, tj. od stanja gdje imamo svaki čvor u svojoj komponenti ($k = |V|$) i provodimo gore opisanu operaciju dokle god je $k \geq 2$, lema 3.5.1 garantira da ćemo u trenutku kada k postane 1 dobiti jedno minimalno razapinjuće stablo originalnog grafa. Ta operacija je upravo i centralni dio Boruvkinog algoritma.

Da bismo efikasno provodili ovu operaciju spajanja komponenti više puta, podijeliti ćemo tu operaciju u četiri koraka, da bi osigurali lakšu izvedbu svake iduće operacije. U nastavku pomoću pseudokoda i dodatnih objašnjena opisujemo inicijalizaciju, a potom i sva četiri koraka u provođenju svake operacije. Dakle, inicijalizacija se provodi samo jednom i to na početku algoritma, a operacija se provodi dokle god broj različitih komponenti povezanosti ne postane 1.

0. korak (inicijalizacija): Na samom početku imamo $|V|$ čvorova grafa $G = (V, E)$ za koji tražimo minimalno razapinjuće stablo, te svaki od njih predstavlja jednu trivijalnu komponentu povezanosti. Ovdje uvodimo i varijable koje ćemo koristiti u algoritmu. Skup čvorova V iz ulaza će biti varijabilan i njega ćemo modificirati kroz algoritam. Ideja je da on u svakoj iteraciji sadrži predstavnike trenutnih komponenti povezanosti. Primijetimo, na početku je to očito zadovoljeno jer je svaki čvor jedna zasebna komponenta povezanosti. Varijabla $cost$ će biti prvi član izlaza algoritma i predstavljati će ukupnu cijenu minimalnog razapinjućeg stabla, a drugi član izlaza algoritma je skup T i on će se sastojati od bridova koji čine minimalno razapinjuće stablo grafa G. Kako stablo gradimo postepenim dodavanjem bridova, a na početku nismo odabrali ni jedan brid, vrijednost od $cost$ je 0, a T je prazan skup. Varijabla *comp_cnt* označava trenutni broj komponenti povezanosti i mijenjat će se u svakoj iteraciji (smanjivati će se, i to uvijek za barem 1). Pomoću nje ćemo i detektirati kraj algoritma - kada poprimi vrijednost 1. Napomenimo još da varijabla *comp_cnt* u pseudokodu ima ulogu varijable k u prethodnom teoretskom razmatranju algoritma. Kako na samom početku imamo $|V|$ komponenti povezanosti jer nismo dodali ni jedan brid, inicijaliziramo *comp_cnt* sa $|V|$. Polje *root* ima ulogu pamćenja predstavnika komponenti, tj. u $root[u]$ piše oznaka čvora koji je predstavnik komponente povezanosti u kojoj se nalazi u. Prema prethodno opisanom stanju početka algoritma, stavljamo $root[u] = u$, za sve $u \in V$.

Algorithm 3 Boruvkin algoritam: inicijalizacija

```
G = (V, E)
cost ← 0
T = {}
comp_cnt ← |V|
for each u ∈ V do
    root[u] ← u
end for
```

1. korak operacije (minimizacija po bridovima): ovaj korak u svakoj operaciji služi za pronalazak bridova $e_1, ..., e_k$ ($k \longleftrightarrow comp\_cnt$) na način na koji smo opisali u teoretskom razmatranju algoritma. Preciznije, u ovoj fazi algoritma imamo *comp_cnt* ($comp\_cnt \geq 2$ jer je inače algoritam završio) čvorova u skupu V, koji označavaju predstavnike različitih

komponenti povezanosti s obzirom na trenutni skup odabranih bridova T (specijalno, u 1. koraku prve operacije je V skup svih čvorova grafa G jer je T prazan). Cilj je za svaki $u \in V$ pronaći brid najmanje težine od svih bridova koji spajaju u i neki $v \in V$, $v \neq u$. U tu svrhu na početku inicijaliziramo polje *min_out* sa uređenim parovima $(\infty, -1)$. Prvi element u paru $min\_out[u]$ označava težinu brida najmanje težine koji spaja komponentu u sa nekom drugom komponentom v, a drugi element u paru označava tu drugu komponentu v. S obzirom da u skupu V držimo predstavnike komponenti, a u konačnici nas zanimaju bridovi početnog grafa koji će ući u T, moramo znati koje bridove iz E ubacujemo u minimalno razapinjuće stablo, a ne koje komponente povezanosti smo spajali u nekom koraku. U tu svrhu definiramo polje *original_edge*, gdje će $original\_edge[u]$ biti konkretan brid iz početnog skupa E koji je u ovom koraku pronađen kao brid najmanje težine koji spaja komponentu $u \in V$ sa nekom drugom komponentom $v \in V$. Radi određenosti inicijaliziramo cijelo polje sa $\{-1, -1, 0\}$, iako nije bitno (jer pomoću *min_out* radimo usporedbe). Sada ćemo populirati definirana polja na prilično izravan način. Prolazimo po svim bridovima $e = \{u, v, c\} \in E$ te ignoriramo sve bridove za koje je $root[u] = root[v]$ jer oni spajaju čvorove koji su u istoj komponenti povezanosti (s obzirom na trenutni skup T) pa ih više ne možemo dodati u razapinjuće stablo. Svi ostali bridovi su potencijalno korisni. Stoga za svaki brid koristan $e = \{u, v, c\}$ radimo najprije provjeru za komponentu $root[u]$. Ako je $min\_out[root[u]].first > c$, tada smo za komponentu $root[u]$ pronašli brid manje težine koji spaja komponentu $root[u]$ sa nekom drugom, konkretno $root[v]$. U skladu s time i definicijom polja *min_out* te *original_edge* ažuriramo vrijednosti $min\_out[root[u]] = (c, root[v])$ i $original\_edge[root[u]] = e$. Potpuno analognu provjeru napravimo za čvor v.

Algorithm 4 Boruvkin algoritam: 1. korak svake operacije

```
min_out[] ← [(∞, −1), ..., (∞, −1)]
original_edge ← [{−1, −1, 0}, ..., {−1, −1, 0}]
for each e = {u, v, c} ∈ E do
    if root[u] ≠ root[v] then
        if min_out[root[u]].first > c then
            min_out[[root[u]] = (c, root[v])
            original_edge[root[u]] = e
        end if
        if min_out[root[v]].first > c then
            min_out[root[v]] = (c, root[u])
            original_edge[root[v]] = e
        end if
    end if
end for
```

2. korak operacije (brisanje malih ciklusa): cilj ovog koraka operacije je na ispra-

van način dodati odabrane bridove u razapinjuće stablo, ažurirati trenutnu cijenu *cost* te ažurirati predstavnike komponenti povezanosti - uočimo da dodavanjem ovako odabranih bridova (kao u 1. koraku operacije) u razapinjuće stablo smanjujemo broj komponenti povezanosti. U 1. koraku operacije smo izdvojili $|V|$ bridova koje bismo trebali dodati u T, tj. u razapinjuće stablo koje će na kraju biti minimalno. U teoretskom razmatranju algoritma smo napomenuli da biranjem bridova na način na koji to činimo, možemo u razapinjućem stablu koje gradimo stvoriti jednostavne cikluse (od 2 čvora). Napomenuli smo i da zapravo nije bitno ako u skup bridova dodamo dva brida koji čine trivijalni ciklus jer dodajemo dva puta isti brid pa je skup isti dodamo li samo jedan ili oba brida. Ipak, u kodu treba biti oprezan i paziti na takve cikluse. U nastavku opisujemo način na koji ispunjavamo ciljeve ovog koraka. Podsjetimo najprije, $min\_out[u]$ nam za komponentu u daje informaciju o bridu najmanje težine koji u spaja sa nekom drugom komponentom $v \in V$ te motivirani lemom 3.5.1 nastojimo sve takve bridove dodati u T. Stoga najprije iteriramo po svim komponentama (njihovim predstavnicima) $u \in V$ i prvo za svaki u provjerimo je li on dio malog ciklusa. Čvor $u \in V$ je dio malog ciklusa ako postoji $v \in V$ takav da $min\_out[u].second = v \wedge min\_out[v].second = u$. U samom kodu želimo razbiti takve cikluse i to činimo na način da uzmemo komponentu s manjom oznakom i za nju obrišemo brid prema drugoj komponenti (dogovorno je taj način - mogli smo i drugačije, samo je bitno da smo konzistentni s odlukom). Preciznije, pretpostavimo da je u dio malog ciklusa u paru sa v te vrijedi $u < v$. Tada ćemo postaviti $min\_out[u].second = u$, a $min\_out[v].second$ će i dalje pokazivati na u. Nakon što za svaki u u `for` petlji provjerimo je li dio ciklusa i po potrebi promijenimo njegov min_out, preostaje dodati brid koji određuje $min\_out[u]$ u razapinjuće stablo, tj. skup T. Ipak, sada moramo provjeriti je li $min\_out[u].second \neq u$, tj. osigurati da nije slučaj da je u otkriven kao član malog ciklusa i to kao onaj s manjom oznakom čiji smo brid odlučili obrisati. Ako je ta provjera prošla, u stablo T dodajemo brid $original\_edge[u]$ te cijenu tog brida pribrajamo trenutnoj cijeni razapinjućeg stabla. Također, s obzirom da sada spajamo komponente u i $min\_out[u].second$, moramo ažurirati i predstavnike tih komponenti, tj. potrebno je od trenutna dva predstavnika odabrati jednog. U tu svrhu stavljamo $root[u] = min\_out[u].second$. Primijetimo da na taj način nećemo doći u konflikte (cikličke definicije predstavnika) jer izvođenjem `for` petlje razbijamo sve male cikluse, pa polje min_out neće više implicirati ni jedan ciklus, a tada neće ni polje $root$.

3. korak operacije (kreiranje zvjezdastih stabala): primijetimo da nakon 2. koraka operacije još uvijek nemamo ispravno populirano polje $root$. Primjerice, za različite $v_1, v_2, v_3 \in V$ možemo imati $root[v_1] = v_2$, $root[v_2] = v_3$ i $root[v_3] = v_3$. Ta reprezentacija nije točna jer želimo spojiti sva tri čvora u istu komponentu, a time želimo i da sva tri pokazuju na istog predstavnika. U ovom koraku ćemo osigurati konzistentnu reprezentaciju, tj. za svaku novu komponentu povezanosti koja će nastati spajanjima nekih $v_1, v_2, ..., v_l \in V$ u 2. koraku ćemo osigurati da postoji $r \in \{1, ..., l\}$ takav da za sve $j \in \{1, ..., l\}$ vrijedi

Algorithm 5 Boruvkin algoritam: 2. korak svake operacije

```
for each u ∈ V do
    if min_out[u].second = v ∧ min_out[v].second = u ∧ u < v then
        min_out[u].second ← u
    end if
    if min_out[u].second ≠ u then
        root[u] ← min_out[u].second
        cost ← cost + min_out[u].first
        T = T ∪ {original_edge[u]}
    end if
end for
```

$root[v_j] = v_r$. Takva stabla su u literaturi poznata pod nazivom *zvjezdasta stabla* (eng. *star trees*) jer postoji centralni čvor (v_r kod nas) s kojim su svi ostali čvorovi iz stabla direktno povezani bridom. U algoritmu 6 vidimo rješenje problema. Dokle god postoji neki $u \in V$ za koji je $root[u] \neq root[root[u]]$, iteriramo po svim elementima $u \in V$ i pitamo je li $root[u] \neq root[root[u]]$. Ako je, postavimo $root[u]$ na $root[root[u]]$. Naglasimo ponovno da je završetak ovog koraka operacije garantiran u konačno mnogo koraka zahvaljujući eliminaciji ciklusa u prošlom koraku (u polju *min_out*, a iz toga je slijedilo i u polju *root*) te da će polje *root* na kraju ovog koraka implicirati prethodno opisano zvjezdasto stablo. Naravno, predstavnicima komponenti ćemo sada smatrati sve čvorove $u \in V$ za koje je $root[u] = u$. Dodajmo samo da je ovo vrlo efikasan način ažuriranja predstavnika: broj iteracija `while` petlje se može ograničiti sa $log_2(|V|)$ [6].

Algorithm 6 Boruvkin algoritam: 3. korak svake operacije

```
while ∃u : root[u] ≠ root[root[u]] do
    for each u ∈ V do
        if root[u] ≠ root[root[u]] then
            root[u] ← root[root[u]]
        end if
    end for
end while
```

4. korak operacije (kompresija i brisanje): opišimo sada kakvu situaciju imamo nakon prva 3 koraka svake operacije. Uspjeli smo dodati nekoliko bridova u razapinjuće stablo koje gradimo i pritom smo povezivali neke komponente povezanosti koje su do sada bile nepovezane. Također, uspješno smo ažurirali skup bridova i cijenu razapinjućeg stabla koje gradimo (koje će biti minimalno na kraju) te smo odabrali nove predstavnike komponenti. Pošto sada svi čvorovi unutar jedne komponente imaju jednaku važnost, možemo ih sve

poistovjetiti sa predstavnikom komponente u kojoj se nalaze. Stoga u V ostavljamo samo predstavnike svih komponenti (sjetimo se, to su svi čvorovi u za koje je $root[u] = u$). Nakon toga, iz skupa bridova izbacujemo sve bridove koji spajaju čvorove unutar iste komponente. Najprije dakle postavimo *comp_cnt* na 0 te prođemo po svim $u \in V$ i pritom povećavamo broj komponenti *comp_cnt* za 1 svaki puta kada naiđemo na u takav da je $u = root[u]$. Također, za sve u za koje je $u \neq root[u]$ ažuriramo skup predstavnika povezanih komponenti sa $V \leftarrow V \setminus \{u\}$ jer takvi čvorovi u više nisu predstavnici u svojim komponentama. Na sličan način prolazimo po svim bridovima $e = \{u, v, c\} \in E$ te u slučaju da se u i v sada nalaze u istoj komponenti povezanosti, izbacimo e iz E jer e više sigurno neće biti od koristi (stvorili bismo ciklus dodavši ga u razapinjuće stablo). U algoritmu 7 se nalazi pseudokod upravo opisanog djela algoritma.

Algorithm 7 Boruvkin algoritam: 4. korak svake operacije

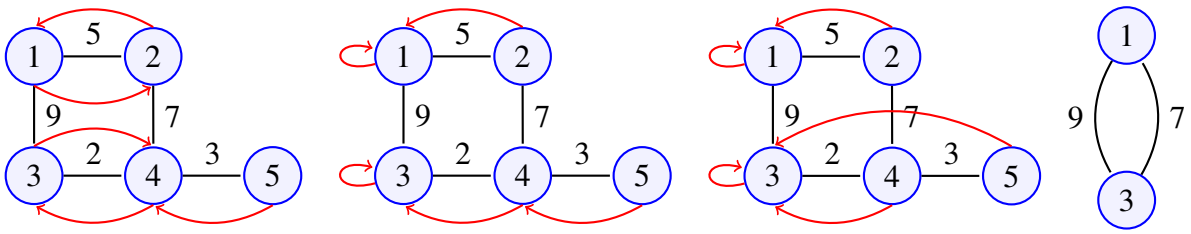
```
comp_cnt ← 0
for each u ∈ V do
    if root[u] = u then
        comp_cnt ← comp_cnt + 1
    else
        V ← V \ {u}
    end if
end for
for each {u, v, c} ∈ E do
    if root[u] = root[v] then
        E ← E \ {{u, v, c}}
    end if
end for
```

Algorithm 8 Boruvkin algoritam: ujedinjeni svi koraci

```
function BoruvkaMST(V, E)
    inicijalizacija()
    while comp_cnt ≥ 2 do
        1_korak_operacije()
        2_korak_operacije()
        3_korak_operacije()
        4_korak_operacije()
    end while
    return (cost, T)
end function
```

Provedimo po koracima upravo opisani algoritam 8 na jednostavnom primjeru sa slike 3.7. Navoditi ćemo samo relevantne promjene u varijablama, odnosno objektima iz algoritma. Ulazni graf čine svi čvorovi koji se nalaze na slici (a), a ulazni bridovi grafa su nacrtani crnom bojom na slici (a). Provedimo najprije fazu inicijalizacije: $cost = 0, T = \{\}, comp\_cnt = 5, V = \{1, 2, 3, 4, 5\}$. Kako je $comp\_cnt = 5 \geq 2$, ulazimo u `while` petlju u algoritmu 8. Nakon što provedemo 1. korak operacije, možemo vidjeti promjene u polju *min_out* na slici 3.7 (a). Crvenom bojom su označeni lukovi $x \rightarrow y$ za koje vrijedi $min\_out[x].second = y$. Sada uočavamo 2 mala ciklusa ($1 \rightarrow 2 \rightarrow 1$, $3 \rightarrow 4 \rightarrow 3$) implicirana tim lukovima. Rezultate 2. koraka operacije možemo vidjeti na 3.7 (b). Čvorovi 1 i 3 su imali manje oznake u svojim ciklusima pa smo njih odabrali kao one čije ćemo crvene lukove usmjeriti prema samima sebi (odnosno postaviti $min\_out[1].second = 1$ i $min\_out[3].second = 3$). Prateći algoritam 2. koraka, vidimo na na kraju imamo $T = \{\{1, 2, 5\}, \{3, 4, 2\}, \{4, 5, 3\}\}$ te $cost = 10$. U 3. koraku operacije želimo ažurirati predstavnike komponenti (komponenti povezanosti s obzirom na crvene lukove - formalno ih u ovom trenutku moramo gledati kao neusmjerene da bismo govorili o povezanosti, ali zanemarujemo taj detalj pošto nije bitan ni u kodu). Uočimo da je u ovom koraku čvor 5 jedini kojega treba ažurirati, tj. takav da $root[5] \neq root[root[5]]$. Naime, $root[5]$ će nakon 2. koraka biti 4, a $root[4]$ će biti 3. Stoga postavimo $root[5] = 3$. Kako vrijedi $root[3] = 3$, slijedi da smo gotovi s 3. korakom operacije. U 4. koraku operacije iz V izbacujemo sve čvorove koji nisu predstavnici svojih komponenti, a to su svi osim 1 i 3. Također, izbacujemo sve (crne) bridove koji sada povezuju čvorove iz istih komponenti. To su svi bridovi originalnog grafa osim $\{1, 3, 9\}$ i $\{2, 4, 7\}$. Nadalje, u ovom koraku izračunamo $comp\_cnt = 2$ pa idemo u još jednu iteraciju `while` petlje iz algoritma 8.

Sada je $V = \{1, 3\}$, $E = \{\{1, 3, 9\}, \{2, 4, 7\}\}$. U 1. koraku operacije nalazimo $min\_out[1] = (7, 3)$ i $min\_out[3] = (7, 1)$. U 2. koraku operacije razbijemo taj mali ciklus tako da stavimo $\text{min}\_out[1].second = 1$. Primijetimo, ostalo je $root[1] = 1$. Također, u ovom koraku imamo $T \leftarrow T \cup \{\{2, 4, 7\}\}$ i $cost \leftarrow 17$. U 3. koraku nemamo ni jedan ulazak u `while` petlju jer je $root[u] = root[root[u]]$ za $u = 1, 3$. Nakon 4. koraka ćemo imati $V \leftarrow V \setminus \{3\}$ jer $root[3] = 1$, a $E \leftarrow E \setminus \{\{1, 3, 9\}, \{2, 4, 7\}\}$ jer oba brida sada povezuju čvorove iz iste komponente ($root[3] = 1$) nakon 2. koraka. U 4. koraku također prebrojimo $comp\_cnt = 1$ te stoga algoritam završava. Kao izlaz dobivamo $T = \{\{1, 2, 5\}, \{3, 4, 2\}, \{4, 5, 3\}, \{2, 4, 7\}\}$ te $cost = 17$.



(a) 1. korak (b) 2. korak (c) 3. korak (d) 4. korak

Slika 3.7: Jedna operacija glavnog dijela Boruvkinog algoritma

Paralelni Boruvkin algoritam

Sada je preostalo prilagoditi ovaj algoritam višedretvenom izvršavanju s dijeljenom memorijom. Paralelizacija će se ponovno sastojati u podjeli posla među dretvama u svakom koraku operacije. U nastavku slijedi relevantan dio koda za inicijalizaciju te za svaki opisani korak operacije.

```
constexpr std::size_t CACHE_LINE_SIZE =
  std::hardware_destructive_interference_size;

typedef std::pair<std::vector<int>,
  char[CACHE_LINE_SIZE - sizeof(std::vector<int>)]> PaddedVector;

using edge = std::tuple<int, int, int>;

const int inf = int(2e9);
const std::pair<int, int> sentinel = std::make_pair(inf, -1);

size_t V, E; //broj cvorova i broj bridova
//ucitamo ih sa ulaza, kao i bridove grafa

std::vector<std::pair<edge, int>> edges(E); //brid, indeks

std::atomic<int64_t> mst_val = 0; //mst cost
std::vector<int> root(V + 1); //predstavnici komponenti

std::vector<std::mutex> mutexi(V + 1); //lokot za svaki cvor
std::vector<int> vertices(V); //predstavnici komponenti
std::vector<int> where(V + 1); //na kojem indeksu sam ja u vertices
std::vector<std::pair<int, int>> mini_out(V, sentinel);
std::vector<int> original_edge(V); //kao u pseudokodu; pravi brid
std::vector<int> mst_edges(V - 1); //indeksi svih bridova u MST-u
size_t last_mst_edge = 0; //prvi indeks na koji mogu dodat brid
std::barrier barijera(P);
std::vector<PaddedVector> Qp(P); //lokalni spremnici za sve dretve
std::vector<PaddedVector> my_mst_edges(P); //isto, za indekse bridova

std::vector<edge> input_edges(E);
for(int i = 0; i < E; i++){
  //ulaz(edge[i].first), edges[i].second = i;
  //pamtimo stvarne bridove jer se edges mijenja u algoritmu
  input_edges[i] = edges[i].first;
}

//inicijalizacija koju provede svaka dretva prilikom svojeg pokretanja
size_t begin = V * iPE / P + 1; //ukljuciva granica
size_t end = V * (iPE + 1) / P + 1; //iskljuciva granica
```

```
for(; begin < end; begin++){
  vertices[begin - 1] = begin;
  root[begin] = begin; //svaki cvor zasebna komponenta
  where[begin] = begin - 1; //zapamti indeks
}
```

Listing 3.5: inicijalizacija potrebnih varijabli i objekata

Kod inicijalizacije spomenimo samo da niz *vertices* ima ulogu skupa V u pseudokodu, no kako ga držimo kao vektor u kojem čvorovi mogu doći u proizvoljnom poretku, potrebno je pomoću dodatnog polja *where* zapamtiti gdje se koji čvor nalazi. To dodatno polje će koristiti i *mini_out* i *original_edge* jer oba polja rade na elementima iz *vertices*, a ne običnim indeksima. Primjerice, *mini_out*[i] nije ono što je u pseudokodu *min_out*[i], već je *mini_out*[i] ono što je u pseudokodu *min_out*[*vertices*[i]].

```
//svaka dretva popuni svoj dio opet
std::fill(mini_out.begin() + (V * iPE / P),
  mini_out.begin() + (V * (iPE + 1) / P), sentinel);
//original edge i ovdje pamti indeks pravog brida (ne modificiarnog)
std::fill(original_edge.begin() + (V * iPE / P),
  original_edge.begin() + (V * (iPE + 1) / P), -1);

barijera.arrive_and_wait();

begin = E * iPE / P; //ukljucivo
end = E * (iPE + 1) / P; //iskljucivo

for(int j = begin; j < end; j++){
  auto &[cur_edge, ind] = edges[j];
  auto &[u, v, c] = cur_edge;
  u = root[u]; v = root[v];
  if(u == v){ //u istoj komponenti su vec
    continue;
  }
  int &p_u = where[u], &p_v = where[v];
  //trazenje minimalnih bridova - vidjeti pseudokod za objasnjenje
  {
    std::lock_guard<std::mutex> lg1(mutexi[u]);
    if(c < mini_out[p_u].first){
      mini_out[p_u] = std::make_pair(c, v);
      original_edge[p_u] = ind;
    }
  }
  {
    std::lock_guard<std::mutex> lg2(mutexi[v]);
    if(c < mini_out[p_v].first){
      mini_out[p_v] = std::make_pair(c, u);
      original_edge[p_v] = ind;
```

```
34      }
35    }
36  }
37
38  barijera.arrive_and_wait(); //obavezno pricekati da svi ovo odrade
```

Listing 3.6: 1. korak iteracije

U 1. koraku svake iteracije dretve najprije izračunaju granice intervala u vektoru bridova za koje će računati vrijednosti u vektoru *mini_out*. U ovom dijelu algoritma moramo izbjeći stanje natjecanja zbog istovremenog čitanja i pisanja u elemente polja *mini_out* i *original_edge*. U 16. liniji koda čvorove koji određuju brid poistovjetimo s predstavnicima komponenti (u pseudokodu smo eksplicitno izbacivali bridove koji su spajali čvorove iz istih komponenti, ovdje to radimo implicitno radi performansi) te u 17. liniji provjerimo spaja li taj brid čvorove iz različitih komponenti. Ukoliko su iz iste komponente, petlja se nastavlja. U protivnom ažuriramo polje *mini_out* na opisan način. Primijetimo da je nužno svaki puta uzeti reference na u i v da bismo mijenjali oznake tih čvorova u samim bridovima jer u protivnom se može desiti da neke bridove kasnije pogrešno protumačimo i stvorimo ciklus. Korisno je uočiti da ne moramo istovremeno zaključavati mutexe od oba čvora koji određuju trenutni brid, već možemo svaki čvor zaključati i obraditi pojedinačno. Upravo to činimo u blokovima s počecima u 22. i 29. liniji koda. Zaključavanje i otključavanje vršimo pomoću `std::lock_guard<>` (RAII). U 38. liniji pomoću barijere pričekamo da sve dretve završe, tj. da se prođe po svim bridovima.

```
1  //opet racunamo granice na isti nacin
2  begin = V * iPE / P;
3  end = V * (iPE + 1) / P;
4
5  //svaka dretva ce dodati neke bridove u globalni mst_edges
6  //ponovno koristimo princip brzih lokalnih spremnika (PaddedVector)
7  my_mst_edges[iPE].first.clear();
8  my_mst_edges[iPE].first.reserve(2 * (end - begin));
9
10 //sada se rjesavamo malih ciklusa i dodajemo
11 //bridove u razapinjuce stablo i zbrajamo cijenu
12 for(int j = begin; j < end; j++){
13   int &u = vertices[j];
14   if(mini_out[where[mini_out[j].second]].second == u &&
15       u < mini_out[j].second){
16     mini_out[j].second = u; //self loop
17   }
18   if(mini_out[j].second != u){ //dodajemu brid u razapinjuce stablo
19     root[u] = mini_out[j].second;
20     mst_val += int64_t(mini_out[j].first);
21     my_mst_edges[iPE].first.push_back(original_edge[j]);
22   }
```

```
}

//optimalnije je ako samo jedna dretva inicijalizira varijable
//kad se to moze
if(iPE == P - 1)
  comp = 0;

//pripremimo lokalni spremnik za ubacivanje predstavnika komponenti
//iz mojeg intervala (predstavnici komponenti za iducu operaciju)
Qp[iPE].first.clear();
Qp[iPE].first.reserve(2 * V / P + 1);

barijera.arrive_and_wait(); //svi moraju zavrsiti

//izracunaj koja je moja pocetna pozicija za ubacivanje
size_t myPos = 0;
for(int j = 0; j < iPE; j++){
  myPos += my_mst_edges[j].first.size();
}

//prebacimo mst bridove koje smo lokalno prepoznali u globalni vektor
std::copy(my_mst_edges[iPE].first.cbegin(),
  my_mst_edges[iPE].first.cend(),
    mst_edges.begin() + last_mst_edge + myPos);
```

Listing 3.7: 2. korak iteracije

U 2. koraku svake operacije algoritma (u kojem se rješavamo ciklusa veličine 2) nemamo stanje natjecanja iako to nije očito iz samog koda. Najprije u 2. i 3. liniji dretva izračuna granice intervala u vektoru *vertices* za koje će provjeriti jesu li dio spomenutih ciklusa. Primijetimo da za svaka dva čvora u, v koji su dio takvog ciklusa, `if` u 14. liniji prolazi samo onaj s manjom oznakom i tako izbjegavamo stanje natjecanja. Ostatak koda 2. koraka operacije je opisan komentarima.

```
//treba nam do-while i flag da bismo znali kad zaustaviti petlju
//u pseudokodu smo to samo matematicki izrekli

//gradimo zvjezdasta stabla pomocu root polja
bool flag = true;
do{
  flag = false;
  for(int j = begin; j < end; j++){
    int &u = vertices[j];
    if(root[root[u]] != root[u]){
      root[u] = root[root[u]];
      flag = true;
    }
  }
```

```
15 }while(flag);
16
17 //sada svaki cvor koji ce na kraju ostati predstavnik
18 //ide u sljedecu iteraciju
19 for(int j = begin; j < end; j++){
20   int &u = vertices[j];
21   if(root[u] == u){
22     Qp[iPE].first.push_back(u); //ubaci ga u lokalni spremnik
23     comp++; //broj komponenti povezanosti za sljedecu iteraciju
24     continue;
25   }
26 }
27
28 barijera.arrive_and_wait();
```

Listing 3.8: 3. korak iteracije

Uočimo da u ovom djelu koda imamo benigno stanje natjecanja. Prvo, svaka dretva piše u polje *root* samo za čvorove koji su iz njenog intervala. Jedini naizgled problematičan pristup je čitanje izraza $root[root[u]]$. Ipak, ponašanje našeg algoritma je takvo da pogrešno pročitana vrijednost iz takvih izraza (10. i 11. linija) može za posljedicu imati samo nekoliko dodatnih iteracija u `do-while` petlji. To vrijedi zato što polje *root* ne može implicirati cikluse (uklonili smo ih u prijašnjem koraku). Testiranjem na nekoliko velikih primjera smo utvrdili da je to vremenski jeftinije od zaključavanja mutexa pri svakom skoku petlje. U 28. liniji se čeka da sve dretve obave posao i da dobijemo željenu strukturu komponenata povezanosti.

```
1 //samo jedna dretva neka inicijalizira
2 if(iPE == P - 1){
3   last_mst_edge += myPos + my_mst_edges[iPE].first.size();
4   vertices.resize(comp); //comp predstavnika ima za iducu operaciju
5   mini_out.resize(comp);
6   original_edge.resize(comp);
7   V = comp;
8 }
9
10 //moja pozicija za ubacivanje u globalni vertices
11 myPos = 0;
12 for(int j = 0; j < iPE; j++){
13   myPos += Qp[j].first.size();
14 }
15
16 //ubaci i zapamti gdje je ostao koji cvor iz tvog intervala
17 size_t ptr = myPos;
18 for(int &node : Qp[iPE].first){
19   vertices[ptr] = node;
20   where[node] = ptr++;
```

```
21 }
22
23 barijera.arrive_and_wait();
24
25 //ako je ostala samo jedna komponenta, algoritam je gotov
26 if(comp == 1){
27   done = true;
28 }
```

Listing 3.9: 4. korak iteracije

U 4. koraku svake iteracije znamo broj komponenti, tj. broj čvorova koji idu u sljedeću operaciju (to su upravo predstavnici komponenti). U 2. liniji osiguramo da samo jedna dretva alocira vektore *vertices*, *mini_out* i *original_edges* za iduću iteraciju. U 3. liniji pomaknemo indeks na prvo slobodno mjesto u globalnom polju *mst_edges*. U 11. liniji računamo početni indeks za polje *vertices*, od kojeg dretva može stavljati svoje čvorove. Od 17. do 21. linije kopiramo elemente iz lokalnih vektora u globalni vektor *vertices* i pritom se za svaki čvor pamti njegova lokacija u tom globalnom polju pomoću vektora *where*.

Performanse algoritma

Algoritam smo testirali na dva slučajno generirana grafa koji zadovoljavaju pretpostavke iz uvoda: povezani su i svi bridovi im imaju međusobno različite težine. Prvi graf ima 500 000 čvorova i 1 549 479 bridova. Drugi graf ima 5 000 000 čvorova i 15 446 285 bridova. Testiranje smo ponovno provodili sa različitim brojevima dretvi i uzimali prosjeke od 20 izvršavanja. Također, ponovno smo provodili testiranje na računalu `prosper.math.hr` kao i za prethodne algoritme. Rezultati testiranja na oba grafa su prikazani u tablicama 3.3, 3.4 i na slikama 3.8, 3.9. Ponovno se algoritam dobro skalira do velikog broja dretvi, s obzirom na specifikacije računala. Najbolje izvođenje na prvom grafu smo dobili sa 32 pokrenute dretve - u prosjeku 0.258 sekundi, a na drugom grafu sa 36 dretvi - u prosjeku 4.082 sekundi.

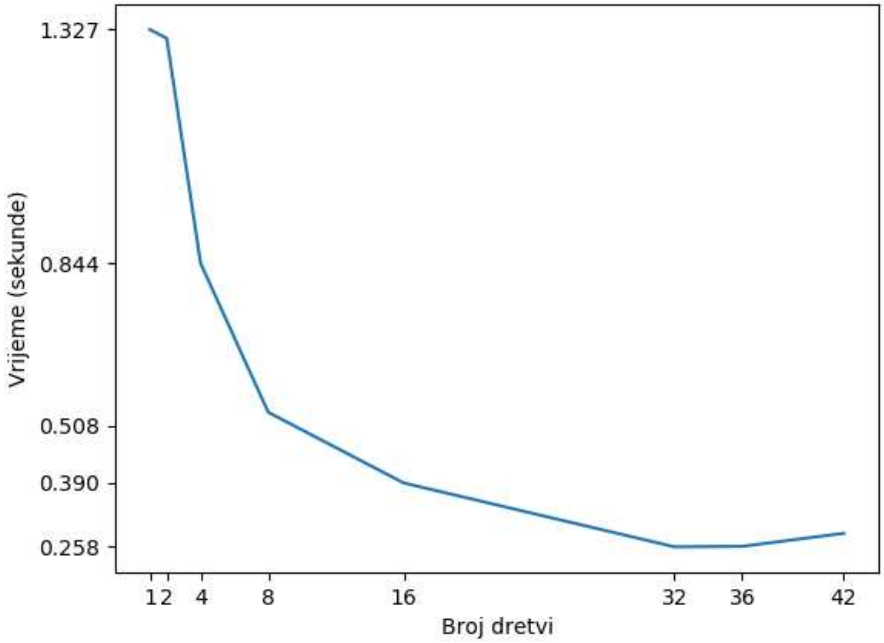
Zbog usporedbe smo implementirali i (sekvencijalni) Kruskalov algoritam i testirali njegovo prosječno vrijeme izvođenja na oba grafa. Na prvom grafu je završio s obradom u 1.669 sekundi u prosjeku, a na drugom za 20.685 sekundi u prosjeku.

Tablica 3.3: Rezultati paralelnog Boruvkinog algoritma na prvom grafu

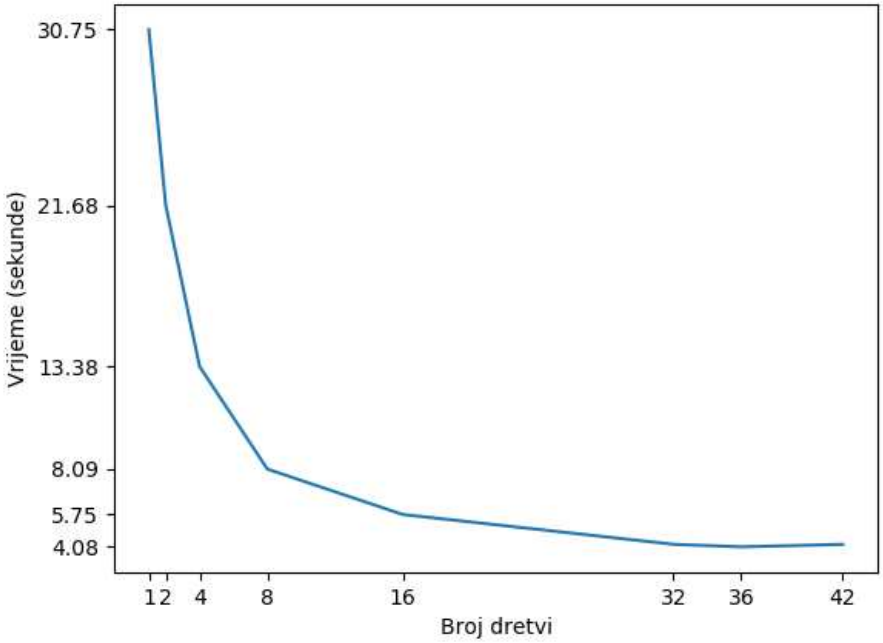
Broj pokrenutih dretvi	Prosječno vrijeme izvršavanja (sekunde)
1	1.327
2	1.309
4	0.844
8	0.508
16	0.390
32	0.258
36	0.259
42	0.286

Tablica 3.4: Rezultati paralelnog Boruvkinog algoritma na drugom grafu

Broj pokrenutih dretvi	Prosječno vrijeme izvršavanja (sekunde)
1	30.746
2	21.682
4	13.380
8	8.091
16	5.750
32	4.212
36	4.082
42	4.210



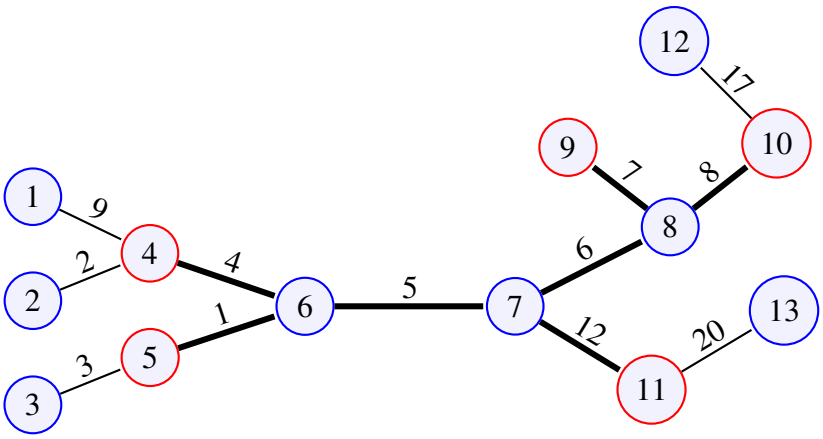
Slika 3.8: Vrijeme računanja paralelnog Boruvkinog algoritma u ovisnosti o broju korištenih dretvi na prvom grafu



Slika 3.9: Vrijeme računanja paralelnog Boruvkinog algoritma u ovisnosti o broju korištenih dretvi na drugom grafu

Primjena algoritma

U ovoj točki ćemo dati motivaciju za postojanje što bržeg algoritma za traženje minimalnog razapinjućeg stabla. Opisujemo problem optimizacije [6] koji se javlja u stvarnom svijetu, a Boruvkin algoritam je koristan alat za pokušaj pronalaska kvalitetnog rješenja. Zamislimo da imamo državu sa puno gradova, od kojih su neki naseljeni, a neki nisu. U državi postoje parovi gradova između kojih je moguće izgraditi cestu uz neki (nenegativni) trošak (koji općenito nije isti za različite parove gradova). Cilj nam je povezati sve naseljene gradove tako da ukupna cijena izgrađenih cesti bude minimalna. Formalna izreka ovog problema je: zadan je neusmjereni težinski graf $G = (V, E)$ i skup čvorova $S \subseteq V$. Potrebno je odabrati $T \subseteq E$ tako da čvorovi iz S budu povezani i to tako da suma težina bridova iz T bude minimalna. Taj problem je pozat pod nazivom *minimalno Steinerovo stablo* (eng. *minimum Steiner tree*). Primijetimo, ako je $S = V$, ovaj problem postaje problem minimalnog razapinjućeg stabla. No, čim je S pravi podskup od V situacija postaje zanimljiva (uočimo da je ponekad isplativo u T dodati i bridove koji ne spajaju nužno samo čvorove iz S). Na slici 3.10 se nalazi graf sa 13 čvorova u kojem je podebljanim bridovima određeno jedno Steinerovo stablo za $S = \{4, 5, 9, 10, 11\}$. Za problem minimalnog Steinerovog stabla nije poznat algoritam polinomijalne vremenske složenosti pa se tako rješava raznim postupcima traženja što boljeg rješenja, umjesto onog najboljeg. Jedan takav pokušaj u svom izračunu koristi minimalno razapinjuće stablo i opis tog pristupa navodimo u nastavku.



Slika 3.10: Jedno Steinerovo stablo za $S = \{4, 5, 9, 10, 11\}$

Nad zadanim skupom čvorova S ćemo napraviti pomoćni graf H koji će biti potpuni, tj. između svaka dva čvora iz S će postojati brid. Za svaka dva čvora $u, v \in S$ ćemo definirati težinu brida $\{u, v\}$ u grafu H sa cijenom najkraćeg puta od u do v u originalnom grafu G. Neka je T_A minimalno razapinjuće stablo takvog grafa H i c_H vrijednost tog razapinjućeg

stabla (suma težina bridova). Sada u takvom stablu T_A svaki brid $\{x, y\}$ zamijenimo sa nizom čvorova i bridova iz G koji se nalaze na najkraćem putu od x do y. Tako dobiveni graf očito ima istu sumu težina bridova c_H, ali može sadržavati paralelne bridove (više bridova između istog para čvorova) i cikluse. Iz takvog grafa uklonimo sve paralelne bridove i uklanjamo bridove iz ciklusa dokle god postoji neki ciklus. Na kraju dobivamo jedno Steinerovo stablo za S koje ima cijenu najviše c_H. Za kraj navodimo tvrdnju i dokaz o gornjoj ogradi na takav c_H

Lema 3.5.2. *Za vrijednost c_H koju nalazimo opisanim postupkom vrijedi: $c_H \leq 2 \cdot c(T_{opt})$, pri čemu je T_{opt} neko minimalno Steinerovo stablo.*

Dokaz. Dovoljno je dokazati da konstruirani graf H ima razapinjuće stablo čija je cijena najviše $2 \cdot c(T_{opt})$ (tada ćemo, primjerice Boruvkinim algoritmom i pronaći takvo stablo). Fiksirajmo jedno minimalno Steinerovo stablo T_{opt} i definirajmo šetnju C u T_{opt} tako da ona krene od nekog čvora r u T_{opt}, prođe po svim čvorovima iz T_{opt} i na kraju se vrati u čvor r. Na taj način svaki brid iz T_{opt} obilazimo točno dva puta. Primjerice, za Steinerovo stablo sa slike 3.10 i početni čvor 5 šetnja C može biti $5 \to 6 \to 4 \to 6 \to 7 \to 8 \to 9 \to 8 \to 10 \to 8 \to 7 \to 11 \to 7 \to 6 \to 5$. Iz takve šetnje C ćemo izbaciti sve čvorove koji se nalaze u $V \setminus S$. Primijetimo da tada dobivamo jednu šetnju u grafu H u kojoj je suma težina bridova najviše $2 \cdot c(T_{opt})$. To vidimo na sljedeći način. Uzmimo iz originalne šetnje C (prije izbacivanja spomenutih čvorova) neka dva čvora $u, v \in S$ i dio šetnje $(u, x_1, x_2, ..., x_k, v)$ iz C t.d. su $x_i \in V \setminus S$, za $1 \leq i \leq k$ (gledamo samo parove čvorova $u, v \in S$ za koje postoje takvi dijelovi šetnje C). U grafu H između u i v imamo brid težine jednake cijeni najkraćeg puta između u i v u G. Stoga u šetnji C iz T_{opt} ne možemo imati put s manjom cijenom između ta dva čvora. Kada to uvažimo za svaka dva susjedna čvora $u, v \in S$ koji ostanu u C nakon izbacivanja, slijedi da dobivena šetnja u H ne može imati cijenu veću od $2 \cdot c(T_{opt})$. U toj šetnji se nalaze svi čvorovi iz S u jednom velikom ciklusu. Izbacivanjem bilo kojeg brida iz te šetnje dobivamo jedno Steinerovo stablo za skup S u G sa cijenom najviše $2 \cdot c(T_{opt})$.

□

Bibliografija

[1] https://en.cppreference.com/w/cpp/language/raii.

[2] https://en.cppreference.com/w/cpp/thread.

[3] https://en.cppreference.com/w/cpp/utility/tuple.

[4] Williams A., *C++Concurrency in Action, Second Edition*, Manning Publications Co., 2019.

[5] Josuttis M. N., *C++20 - The Complete Guide*, Leanpub.com, 2022.

[6] P. Sanders, K. Mehlhorn, M. Dietzfelbinger i R. Dementiev, *Sequential and Parallel Algorithms and Data Structures*, The Basic Toolbox, Springer, 2019.

[7] Zhou W., *A Practical Scalable Shared-Memory Parallel Algorithm for Computing Minimum Spanning Trees*, http://algo2.iti.kit.edu/documents/Theses/msThesisZhou.pdf.

Sažetak

U ovom diplomskom radu analiziramo poznate algoritme na grafovima i implementiramo ih u jeziku C++, koristeći paralelizaciju kao sredstvo za optimizaciju.

U uvodnom poglavlju smo se bavili izazovima koji se općenito javljaju kod programiranja višedretvenih aplikacija s dijeljenom memorijom. U drugom poglavlju smo stvorili bazu za konkurentno programiranje u modernom jeziku C++20, gdje smo istaknuli nekoliko korisnih i često korištenih sinkronizacijskih mehanizama te naveli korisne primjere.

U glavnom dijelu rada smo se bavili s tri algoritma: pretraživanje u širinu (BFS), topološko sortiranje usmjerenih grafova bez ciklusa te Boruvkin algoritam za traženje minimalnog razapinjućeg stabla grafa. Za svaki od tih algoritama smo najprije opisali sekvencijalnu verziju, a onda ju paralelizirali i pritom detaljno analizirali mjesta izložena stanjima natjecanja. Na kraju smo sva tri algoritma implementirali u jeziku C++ i testirali njihova izvođenja sa različitim brojem korištenih dretvi, na odabranim testnim primjerima.

Na samom kraju rada je iznesen motivacijski primjer, kojim se ističe korisnost algoritama na grafovima (konkretno, algoritama za traženje minimalnog razapinjućeg stabla) u stvarnom svijetu.

Summary

In this thesis, we analyze well-known algorithms on graphs and implement them in the language C++, using parallelization as a tool for optimization.

In the introductory chapter, we dealt with the challenges that generally arise when programming multithreaded applications with shared memory. In the second chapter, we created a base for concurrent programming in the modern C++20 language, where we highlighted several useful and frequently used synchronization mechanisms and provided useful examples.

In the main part of the thesis, we covered three famous algorithms: breadth-first search (BFS), topological sorting of directed acyclic graphs, and Boruvka's algorithm for finding the minimum spanning tree of a graph. For each of these algorithms, we first described sequential version, then parallelized it step by step and thoroughly analyzed the places exposed to data races. In the end, we implemented all three algorithms in the C++ and tested their performances with a variable number of running threads.

At the very end of the thesis, a motivational example is presented, which highlights the usefulness of algorithms on graphs (specifically, algorithms for finding minimum spanning tree) in the real world.

Životopis

Rođen sam u Čakovcu, 9. svibnja 1999. godine. Osnovnu i srednju školu sam završio u rodnom gradu: 2014. završavam 3. osnovnu školu u Čakovcu, a 2018. sam maturirao kao učenik Gimnazije Josipa Slavenskog Čakovec. Iste godine sam upisao i preddiplomski studij *Matematika* na matematičkom odsjeku Prirodoslovno - matematičkog fakulteta u Zagrebu, a 2021. godine ga završio. Svoje školovanje sam nastavio na istom odsjeku, a upisao sam diplomski studij *Računarstvo i matematika* u listopadu 2021. godine.