SVEUČILIŠTE U SPLITU

FAKULTET ELEKTROTEHNIKE, STROJARSTVA I BRODOGRADNJE

POSLIJEDIPLOMSKI DOKTORSKI STUDIJ ELEKTROTEHNIKE I INFORMACIJSKE TEHNOLOGIJE

KVALIFIKACIJSKI ISPIT

Paralelni sustavi za obradu podataka pohranjenih u formi grafa

LJILJANA DESPALATOVIĆ

Split, listopad 2016.

Sadržaj

1 Uvod

Prirodan način modeliranja mnogih diskretnih procesa i sustava su grafovi i mreže. Formu grafa i mreže uvode matematičari kao formalan prikaz problema za njihovo analitičko rješavanje. Izvan matematike, mreže i grafovi kao alat koriste se u mnogim drugim znanstvenin područjima za rješavanje raznih problema. Dok se matematičari bave pravilnim i sintetičkim mrežama i grafovima i njihovom analizom, u prirodi pronalazimo podatke koje možemo predstaviti u formi grafova i mreža koje su nepravilne i složene.

Neki primjeri sustava koji se mogu predstaviti u ovoj formi su društvene mreže, poznate već dugo u sociologiji, World Wide Web, Internet, transportne mreže, mreže citata, ali i mreže koje modeliraju sustave u biologiji (metaboličke mreže, mreže interakcija između proteina, genetičke regulatorne mreže itd.), u kemiji, geografskim informacijskim sustavima i prepoznavanju uzoraka (engl. *pattern recognition*).

Mreže se pojavljuju u raznim znanstvenim područjima, od neuroznanosti do sociologije, od ekonomije do informacijske tehnologije. Mreže koje dobijemo predstavljanjem podataka ovoj u formi zovemo kompleksne mreže (engl. *complex networks*) ili mreže stvarnog svijeta (engl. *real-world networks*).

Dok mreže u biologiji imaju mali broj čvorova, socijalne mreže i web mreže mogu dostići jako veliki broj čvorova. Facebook, najpopularnija društvena mreža na svijetu, ima oko milijardu i pol čvorova, dok se broj veza penje na 400 milijardi (12/2014) [22]. Web graf je primjer još veće mreže, sa 3.5 milijardi čvorova i 129 milijardi veza [113]. Analiziranje tako velikih grafova je praktički nemoguće na prosječnim korisničkim PC računalima. Velike mreže (*engl. large networks*) su mreže koje se mogu pohraniti u memoriji, dok za one koje su veće od toga, možemo reći da su ogromne (engl. *huge networks*).

Znanstvenici iz područja računalnih znanosti bave se mrežama sa sličnog gledišta kao i matematičari - unaprijeđuju alate za analizu mreža. Pri tom se u računalstvu bave s dva osnovna problema:

- Kako predstaviti mrežu i graf u formi pogodnoj za pohranu na računalu.
- Kako obraditi i analizirati mrežu i graf i doći do rezultata koji su na neki način optimalni.

Prvim problemom bave se tehnologije grafovskih baza podataka. Pritom danas govorimo o *big data* i distribuiranoj pohrani. Drugim problemom bave se sustavi za procesiranje grafova koji izvode algoritme nad kompletnim grafovima koji su uglavnom pohranjeni na eksternim uređajima.

U ovom radu dan je pregled područja obrade velikih grafova na računalima sa posebnim naglaskom na nove tehnologije Big Data ere. U drugom i trećem poglavlju dani su osnovni pojmovi i najpoznatiji algoritmi teorije grafova. Nadalje, u četvrtom poglavlju opisane su paralelni sustavi za analizu velike količine podataka. Sljedeća poglavlje donosi pregled modela za učitavanje i particioniranje grafova. Zadnje poglavlje je zaključak.

2 Pojam i svojstva grafa

2.1 Pojam grafa i mreža

Mrežu čini skup **čvorova** i **veza** između njih, gdje čvorovi i veze mogu imati svojstva (engl. *properties*). Matematički se modelira pomoću grafa $G = (V, E)$ u kojem su čvorovi reprezentirani vrhovima iz skupa $V \neq \varnothing$, a veze bridovima iz skupa bridova E, gdje svaki brid $e \in E$ spaja dva vrha $u, v \in V$. Svojstva čvorova modeliraju se skupovima funkcija vrhova, a svojstva veza skupovima funkcija bridova koje se zovu još i težine bridova. U ovom radu koristit će se ravnopravno pojam grafa i mreže.

Svaki brid $e \in E$ spaja dva vrha $u, v \in V$ koji se zovu **krajevi** od e. Kažemo još da su tada vrhovi u i v **incidentni** s e, a vrhovi u i v **susjedni** i pišemo $e = \{u, v\}$ [101]. **Petlja** je brid čiji se krajevi podudaraju, a brid čiji su krajevi različiti zove se **pravi brid** ili **karika**. **Stupanj vrha** v, $d(v)$ je broj bridova koje koji su incidentni sa vrhom v, pri čemu se svaka petlja računa kao dva brida. **Jednostavan** graf je graf bez petlji u kojem su svaka dva vrha spojena najviše jednim bridom.

Šetnja W u grafu G duljine k je netrivijalni konačni niz vrhova i bridova $W = v_0e_0v_1e_1 \dots e_{k-1}v_k$ takvih da su v_i i v_{i+1} krajevi od e_i, za svako $0 \leq i < k$. Vrhovi v_0 i v_k zovu se početak i kraj šetnje, a $v_1, v_2, \dots v_{k-1}$ unutarnji vrhovi od W. Broj k je duljina šetnje W. Šetnja je **zatvorena** ako je $v_0 = v_k$. Ako su u šetnji W svi bridovi različiti, kažemo da je W **staza**, a ako su na stazi svi vrhovi različiti, kažemo da je W **put**. Zatvorena šetnja, kod koje su početak i unutarnji vrhovi različiti zove se **ciklus**. Ciklus C_k duljine k zove se **k-ciklus**. Graf koji nema ciklusa je **aciklički graf**. **Udaljenost** $d_G(u, v)$ dvaju vrhova $u, v \in V$ je duljina najkraćeg puta između u i v u grafu G, a **dijametar** grafa $diamG = max\{d_G(u, v) : u, v \in V\}$.

Podgraf grafa $G = (V, E)$ je graf $H = (V', E')$, gdje je $V' \subseteq V$ i $E' \subseteq E$. Podgraf H je pravi podgraf grafa G ako je $V' \subset V$ i $E' \subset E$.

Graf sa n vrhova i m bridova može se prikazati pomoću $n \times m$ **matrice incidencije** u kojoj redci matrice odgovaraju vrhovima $v_i, 0 \leq i < n$, a stupci bridovima $e_j, 0 \leq j < m$, a vrijednosti matrice broj koliko su puta v_i i e_j incidentni. Nadalje, graf se može prikazati koristeći $n \times n$ **matricu susjedstva** A u kojoj redci i stupci odgovaraju vrhovima, a vrijednosti elemenata a_{ij} matrice broj bridova ili težina brida između vrhova v_i i v_j. Osim matričnog zapisa u računarstvu je uobičajena implementacija pomoću liste susjedstva gdje se svakom vrhu pridružuje lista njegovih susjednih vrhova.

U nastavku ćemo navesti još neke važne definicije iz teorije grafova.

Usmjereni grafovi Usmjereni graf D je uređeni par $D = (V, E)$ koja se sastoji od nepraznog skupa V vrhova i skupa E uređenih parova vrhova iz V koje zovemo **lukovi** ili usmjereni bridovi. Pišemo $e = (u, v)$ ili $u \to v$. Za usmjereni brid $e = (u, v)$ vrhovi u i v su krajevi od e, u je početni, a v krajnji vrh od e [6]. Primjerice, graf koji reprezentira Facebook prijateljstva je neusmjeren, dok je Twitter graf usmjeren. Usmjereni graf zovemo još i **digraf**.

Pojmovi **poddigraf**, usmjerena šetnja, staza, put i ciklus u usmjerenom grafu definiraju se analogno kao za grafove.

U usmjerenom grafu ulazni stupanj (engl. *in-degree*) vrha $v \in V$ je ukupan broj bridova iz E čiji je v krajnji vrh. Označava se $d_{in}(v)$. Izlazni stupanj (engl. *out-degree*) vrha v je

ukupan broj bridova čiji je v početni vrh i označava se $d_{out}(v)$. Vrijedi:

$$|E| = \sum_{v \in V} d_{in}(v) + \sum_{v \in V} d_{out}(v) \tag{2.1}$$

Povezanost u grafu Graf G je povezan ako između svaka dva vrha $u, v \in G$ postoji put između vrha u i vrha v. **Komponenta povezanosti** u grafu je povezani podgraf takav da unutar njega postoji put između svaka dva vrha, a ne postoji put prema ostalim vrhovima u grafu. U usmjerenim grafovima razlikujemo slabo (engl. *weakly*) i jako (engl. *strongly*) povezane komponente. Slabo povezane komponente su komponente usmjerenog grafa koje bi bile komponente povezanosti kada bi usmjerene bridove zamjenili neusmjerenima. S druge strane, jako povezane komponente su podgrafovi u kojima za svaka dva vrha u, v postoji usmjereni put od u do v i usmjereni put od v do u.

Regularni graf Graf je regularan ako su svi vrhovi istog stupnja. Ako je stupanj svih vrhova k kažemo da je graf **k-regularan**.

Bipartitni graf Graf $G = (V, E)$ je bipartitni graf ili bigraf ako je $V = V_1 \cup V_2$ i $V_1 \cap V_2$ i ako za svaki brid $e = \{u, v\} \in E$ vrijedi da je $u \in V_1$ i $v \in V_2$.

Potpuni graf Graf je potpun (engl. *complete*) ako su svaka dva vrha u njemu susjedna.

Šuma i stablo Šuma je aciklički graf. Stablo je povezan aciklički graf. Vrhovi grafa koji imaju stupanj 1 zovu se **listovi**.

Za graf $G = (V, E)$ sa $|V| = n$ vrhova sljedeće tvrdnje su ekvivalentne:

(i) G je stablo;

(ii) G je acikličan i ima točno $n - 1$ bridova;

(iii) G je povezan i ima točno $n - 1$ bridova;

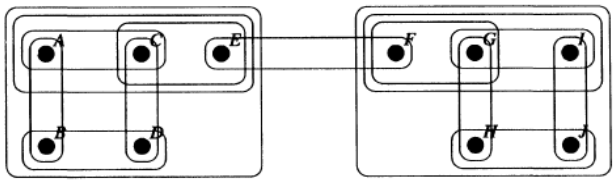
(iv) između svaka dva vrha u grafu G postoji jedinstveni put;

Nadalje vrijedi da za svaki brid e iz stabla G, preostali graf $G - e$ nepovezan. Dodavanjem brida e u stablo G, graf $G + e$ postaje graf s točno jednim ciklusom.

Usmjereni aciklički graf Usmjereni graf bez ciklusa zove se direktni aciklički graf (engl. *directed acyclic graph,* ***DAG***).

Težinski graf Graf $G = (V, E, w)$ gdje je $w : E \to \mathbb{R}$ funkcija koja svakom bridu $e \in E$ pridjeljuje vrijednost $w(e) \in \mathbb{R}$ se zove težinski graf.

Hipergraf Hipergraf je graf $G = (V, E)$, gdje je V skup vrhova, a E skup nepraznih podskupova od V, tj. neprazni podskup partitivnog skupa $\mathcal{P}(V)$. Primjer hipergrafa možemo vidjeti na slici 2.1



Slika 2.1: Hipergraf [54]

Sve pojmove koji nisu ovdje definirani zainteresirani čitatelj može naći u knjizi [12].

2.2 Kompleksne mreže

S aspekta graf analitike u računarstvu, različiti algoritmi se koriste s obzirom na strukturu grafova [60]. Tako različite algoritme primjenjujemo na grafove koji imaju svojstvo da je distribucija stupnjeva uniformna te su stupnjevi vrhova relativno mali i ne ovise o veličini grafa (primjerice mreža cesta ili željeznica), na slučajne i na nerazmjerne grafove.

Grafovima u prvoj grupi dijametar raste kako raste sam graf, odnosno prosječna udaljenost dvaju vrhova ovisi o veličini grafa. Slučajni graf je sintetički napravljen graf na način da se za svaka dva vrha u konačnom skupu vrhova odredi vjerojatnost da su susjedni. Erdős-Rényi model kreće od praznog grafa sa n vrhova u koji se dodaje m bridova na način da svaki od $\binom{\binom{n}{2}}{m}$ mogućih bridova ima jednaku vjerojatnost dodavanja u graf. Prvi brid se odabire između $\binom{n}{2}$ mogućih bridova, drugi između $\binom{n}{2} - 1$ itd. Ekvivalentno, svakom grafu G iz familije svih grafova sa n vrhova i m bridova (slika 2.2a) pridružuje se vjerojatnost

$$P(G) = \binom{\binom{n}{2}}{m}^{-1} \tag{2.2}$$

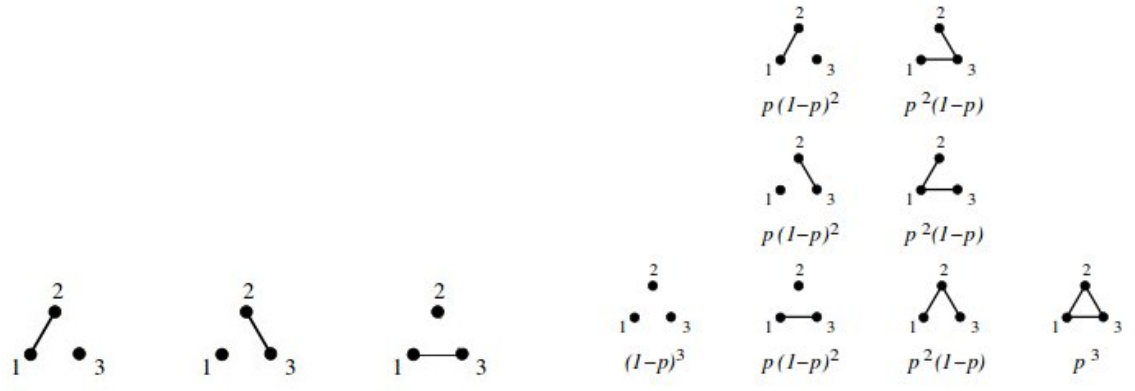
Takav graf se još zove uniformno slučajni graf (engl. *uniform random graph*) [32].

Sličan model je i Gilbertov model u kojem za dani broj vrhova n i vjerojatnost $0 \leq p \leq 1$ svaki od $\binom{n}{2}$ mogućih bridova ima vjerojatnost p da će se dodati u graf i $1 - p$ da neće. Ekvivalentno, svakom grafu G iz familije svih grafova sa n vrhova i m bridova (slika 2.2b) pridružuje se vjerojatnost

$$P(G) = p^m (1-p)^{\binom{n}{2}-m} \tag{2.3}$$

Ovakav graf zove se binomni slučajni graf (engl. *binomial random graph*). Oba modela spominju se u literaturi kao Erdős-Rényi slučajni modeli grafa jer su opisani u [27], ali je binomni slučajni graf već prije opisan u [36].

Za razliku od navedenih grafova, mreže stvarnog svijeta (engl. *real world networks*) ne ponašaju se poput grafova u klasičnim Gilbertovim i Erdős-Rényi modelima i često imaju zajednička topološka svojstva. Neka od njih su efekt malog svijeta (engl. *small world*



(a) Erdős-Rényi model $n = 3, m = 1$

(b) Gilbertov model $n = 3$

Slika 2.2: Familije slučajnih grafova [25]

effect), nerazmjernost mreže (engl. *scale-free*), grupiranje (engl. *clustering*) ili tranzitivnost (engl. *network transitivity*) i struktura zajednice (engl. *community structure*). Grafove kojima se predstavljaju takve mreže zovemo nerazmjerni grafovi, nerazmjerne ili kompleksne mreže. Ova četiri pojma ćemo detaljnije proanalizirati u nastavku izlaganja.

Nerazmjernost mreže (engl. *scale-free network*). Kod nerazmjernih mreža mali broj vrhova ima veliki broj veza, a veliki broj vrhova ima mali broj veza. Distribucija stupnjeva slučajnog grafa prati Poissonovu razdiobu, dok distribucija stupnjeva nerazmjernih mreža prati **krivulju potencije** (engl. *power law*) $f(k) = ak^{-\alpha}$ [96]. Takve mreže česte su u svijetu oko nas. Primjerice, da bi se opisala neka biološka funkcija potrebno je promatrati interakcije između proteina, DNK, RNK i molekula. Kompleksna interstanična mreža interakcija ima slična svojstva mreži web-stranica ili društvenim mrežama. Sve one imaju svojstvo nerazmjernosti [9].

Bitna svojstva nerazmjernih mreža su postojanje koncentratora (engl. *hub*) i dinamički rast (engl. *dynamic growth*). Modeli slučajnog grafa predpostavljaju da je broj vrhova n u grafu konstantan. Međutim, u većini mreža stvarnog svijeta broj vrhova raste. Dinamički rast je svojstvo mreže da raste s vremenom dodavanjem vrhova i bridova u nju. Koncentratori su vrhovi čiji je stupanj bitno veći od stupnja većine vrhova u mreži. Postojanje koncentratora može se objasniti evolucijom mreže: veća je vjerojatnost da će se novi vrhovi povezati sa vrhovima koji imaju veći broj veza. To svojstvo se zove preferencijsko povezivanje (engl. *preferential attachment*) [7].

Koristeći svojstva rasta i preferencijalnog povezivanja može se generirati nerazmjerna mreža. Prvi model nerazmjerne mreže, Barabási-Albert model, kreće od malog broja m_0 vrhova i u svakom koraku dodaje novi vrh sa $m, m \leq m_0$ bridova koji povezuju novi vrh sa m različitih vrhova u mreži. Pritom se pretpostavlja da je vjerojatnost p da će novi vrh biti povezan sa vrhom i ovisi o stupnju vrha k_i, tako da je $p(k_i) = k_i / \sum_j k_j$. Ovako generirana mreža ima distribuciju stupnjeva vrhova koja prati zakon potencije.

Grupiranje (engl. *clustering*) ili tranzitivnost (engl. *network transitivity*). U realnom svijetu, pa i svijetu društvenih mreža vjerojatnost da će nečija dva prijatelja (ili poznanika) i sami biti prijatelji je veća od vjerojatnosti da će dva slučajno odabrana čovjeka biti prijatelji. Tranzitivnost je tendencija prema povezanosti vrhova u grafu ukoliko imaju zajednički susjedni vrh [75]. Taj efekt opisuje koeficijent grupiranja C [76]:

$$C = \frac{3 \times c}{n} \tag{2.4}$$

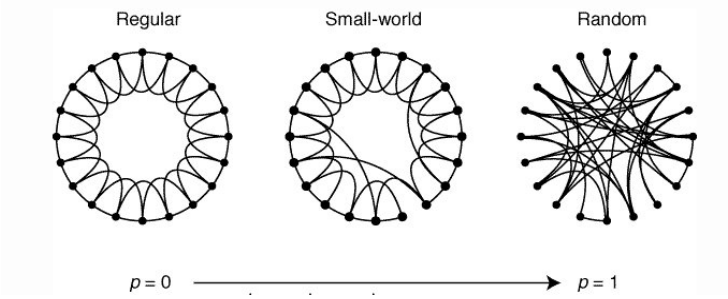
gdje je c broj ciklusa duljine 3, a n broj povezanih trojki vrhova u grafu tj. trojki u kojima je barem jedan vrh spojen s dva ostala vrha. Koeficijent C predstavlja vjerojatnost da su dva vrha povezana sa istim vrhom u grafu i sami povezani. Prosječna vrijednost koeficijenta C je najčešće između 0.1 i 0.5 u kompleksnim mrežama. U potpuno povezanom grafu, gdje je svaki vrh povezan sa svim ostalim vrhovima, koeficijent grupiranja je 1.

Iako se dugo smatralo da je svojstvo grupiranja karakteristično za socijalne mreže [75], pokazalo se da visok stupanj grupiranja imaju i mreže u biologiji. Primjerice, kod metaboličkih mreža nađeno je da je koeficijent grupiranja za jedan red veličina veći od očekivanog koeficijenta grupiranja u nerazmjernim mrežama [87]. Koeficijent grupiranja je velik i kod mreža proteinskih interakcija [8]. Koeficijent grupiranja i nerazmjernost mreže posljedica su hijerarhijske organizacije takvih mreža.

Efekt malog svijeta (engl. *small world effect*). Efekt malog svijeta opisuje svojstvo kompleksne mreže da je prosječna udaljenost između dva vrha logaritamska funkcija ukupnog broja vrhova. Ime potječe od "small world" eksperimenta u kojem je američki socijalni psiholog Stanley Milgram 1967. godine zamolio slučajno odabrane ljude u Nebraski da pošalju pisma slučajno odabranim ljudima u Bostonu, kojima su bila poznata imena, zanimanje i ugrubo opisana lokacija [11]. Zadatak odabranih pošiljatelja bio je da pismo pošalju nekome čiju adresu točno poznaju, a za koga pretpostavljaju da bi mogli poznavati osobu kojoj je pismo upućeno, sa molbom da proslijede njihovo pismo dalje prema navedenoj osobi. Na opće iznenađenje prosječan broj koraka na putu do ciljne osobe bio je šest.

Mreže malog svijeta opisali su Watts i Strogatz [106], koji su uočili da se regularnim i slučajnim grafovima ne mogu opisati biološke, tehnološke i socijalne mreže. U pokušaju da opišu mreže iz stvarnog svijeta, krenuvši od k-regularnog grafa, preusmjerili su, sa vjerojatnošću p, postojeće veze nekog vrha na slučajno odabrane vrhove (slika 2.3). Time se dobila mreža koja više nije regularna, ali nije ni potpuno slučajna. Takva mreža ima malu prosječnu udaljenost između dva vrha (kao što imaju slučajni grafovi) i veliki koeficijent grupiranja (kao što imaju regularni grafovi).

Graf koji je k-regularan ima veliki koeficijent grupiranja, te u njemu prosječna duljina najkraćeg puta ovisi o k i to je veća što je k manji. Slučajni graf ima mali koeficijent grupiranja i malu prosječnu duljinu najkraćeg puta. Watts-Strogatzov model je kompromis između ta dva tipa grafa. Newman i Watts [77] su predložili modificirani model u kojem se veze ne prespajaju, već se regularnom grafu dodaju slučajne veze. Pokazali su da samo nekoliko dodanih slučajnih veza među vrhovima bitno skraćuje prosječni put između vrhova. S druge strane, kada se vjerojatnost p kreće oko 0.5, Watts-Newman model ima veliki koeficijent grupiranja.



Slika 2.3: Wattsov i Strogatzov model [106]

Struktura zajednice (engl. *community structure*). Kompleksne mreže imaju svojstvo da su vrhovi podijeljeni u grupe ili zajednice u kojima su veze među vrhovima unutar zajednice gušće od veza među vrhovima iz različitih zajednica. Mogućnost detektiranja zajednica u mrežama može imati praktičnu primjenu. Zajednice u društvenim mrežama predstavljaju ljude sa sličnim sklonostima ili interesima. U epidemiologiji se mogu koristiti rezultati da bi se prepoznale zajednice među ljudima i veze među njima u cilju sprječavanja širenja bolesti [57]. Kininmonth i dr. [56] su metode detektiranja zajednica koristili za analizu izmjene genetičkih informacija među koraljnim grebenima putem transporta larve. U mreži proteinskih interakcija zajednice predstavljaju funkcionalne module tj. proteine koji imaju slične biološke funkcije unutar stanice [19]. Zajednice na Internetu mogu predstavljati stranice sa sličnim sadržajima. Iz ovih primjera je jasno da detektiranje zajednica u kompleksnim mrežama ima široku primjenu u različitim znanstvenim disciplinama.

Zajednice u mreži predstavljaju particiju vrhova u grafu, te se za traženje zajednica mogu koristiti algoritmi za particioniranje grafa. Dijeljenje grafa na particije koristi se u distribuiranim algoritmima gdje se algoritmi izvode na klasterima računala te tipično dijele graf na particije jednake veličine. Metode particioniranja grafa za distribuirane sustave predstavljene su u poglavlju 6.2. Kod detektiranja zajednica particije vrhova ne moraju biti jednake veličine, pa se koriste adaptirani algoritmi za particioniranje [74]. Osim njih, za manje grafove se mogu koristiti i algoritmi koji računaju međupoloženost bridova (engl. *edge-betweenness*) tj. broj najkraćih putova u grafu koji prolaze kroz brid, te uklanjanjem bridova s najvećom međupoloženošću iz grafa dovode do raspada grafa na zajednice. Primjer takvog algoritma je Girvan-Newmanov algoritam [37].

Zajednice u mreži mogu biti disjunktni skupovi vrhova ili preklapajuće zajednice.

Modularnost Prilikom dijeljenja mreže na zajednice postavlja se pitanje optimalnosti rastava. Naime, algoritam u nekom trenutku treba zaustaviti. Nije potrebno razbiti mrežu na zajednice veličine jednog vrha. Kvalitativna mjera raspada mreže na zajednice naziva se modularnost [79].

Neka je dana particija mreže na k zajednica. Definiramo $k \times k$ matricu E tako da je e_{ij} udio broja bridova koji povezuju particiju i sa particijom j u ukupnom broju bridova. Tada se na dijagonali matrice E nalazi ukupan udio broja bridova koji se nalaze unutar iste particije, pa je $trag(E)$ udio broja bridova koji se neće eliminirati u postupku brisanja bridova prilikom detektiranja zajednica. Dobra particija je particija u kojoj će taj broj biti što veći (što više sačuvanih bridova). Međutim, taj broj će biti najveći u slučaju da su

svi vrhovi unutar iste particije, pa sam trag matrice ne daje nužno informaciju o strukturi zajednice. Zbog toga se definira suma retka (ili stupca) $a_i = \sum_j e_{ij}$ koja predstavlja udio bridova koji su barem jednim krajem u particiji i u ukupnom broju bridova.

Sada definiramo modularnost:

$$Q = \sum_i (e_{ii} - a_i)^2 = trag(e) - \|E^2\| \tag{2.5}$$

gdje je $\|\|$ suma elemenata matrice, kao udio broja bridova u mreži koji povezuju vrhove unutar particija umanjen za očekivani udio bridova unutar particija u slučajnom grafu koji ima istu podjelu vrhova na zajednice, ali slučajne bridove između vrhova (izvod dostupan u [23]). Ako udio bridova unutar zajednica nije bolji od slučajnog udjela, tada je $Q = 0$. Što se vrijednost više približava vrijednosti 1, to je jača struktura zajednica u mreži. U većini slučajeva vrijednost se kreće između 0.3 i 0.7 [79].

Vrijednost Q računa se prilikom svakog raspada zajednice na manje zajednice. Najveća vrijednost Q daje najbolje particioniran graf.

2.3 Reprezentacija grafa

Najčešća reprezentacije grafova su pomoću matrice susjedstva i liste susjedstva. Matrica susjedstva je kvadratna matrica u kojoj je na dijagonali broj a_{ii} bridova kojima su početni i krajnji vrh isti, a ostali elementi matrice a_{ij} težina ili broj bridova između vrhova i i j. Kako su matrice koje predstavljaju kompleksne mreže rijetke (engl. *sparse*), tj. imaju mali broj ne-nula elemenata, neefikasno ih je smještati cijele u memoriju, pa se koriste različite implementacije kako bi zapis matrice zauzimao što manje mjesta u memoriji, a da se istovremeno operacije matričnog računa mogu izvoditi efikasno. U nastavku ćemo dati pregled tehnika i struktura podataka pogodnih za reprezentaciju grafa.

Lista vrhova Vrhovi se spremaju u listu, a informacije o bridu unutar strukture podataka kojom je određen vrh. Prednost ovakvog pristupa je što se, u slučaju da obilazak grafa počinje od nekog vrha, podaci o bridovima fizički nalaze blizu incidentnih vrhova. S druge strane, ako je potreban pristup određenom vrhu, ovakav način pohranjivanja grafa je neefikasan, jer treba prolaziti po listi vrhova sve dok se ne nađe element u kojem je vrh incidentan sa traženm bridom.

Lista bridova Ovaj pristup graf prikazuje kao listu bridova. Sama struktura podataka koja predstavlja brid sastoji se od para (ili liste) vrhova incidentnih s bridom te težine brida ukoliko se radi o težinskom grafu. U ovakvom zapisu svaki vrh sudjeluje barem dva puta, pa postoji redundancija podataka. Osim toga, do informacija o bridovima se dolazi u linearnom vremenu.

Matrica susjedstva Za graf $G = (V, E)$ sa $n = |V|$ vrhova, matrica susjedstva A je $n \times n$ matrica takva da vrijedi

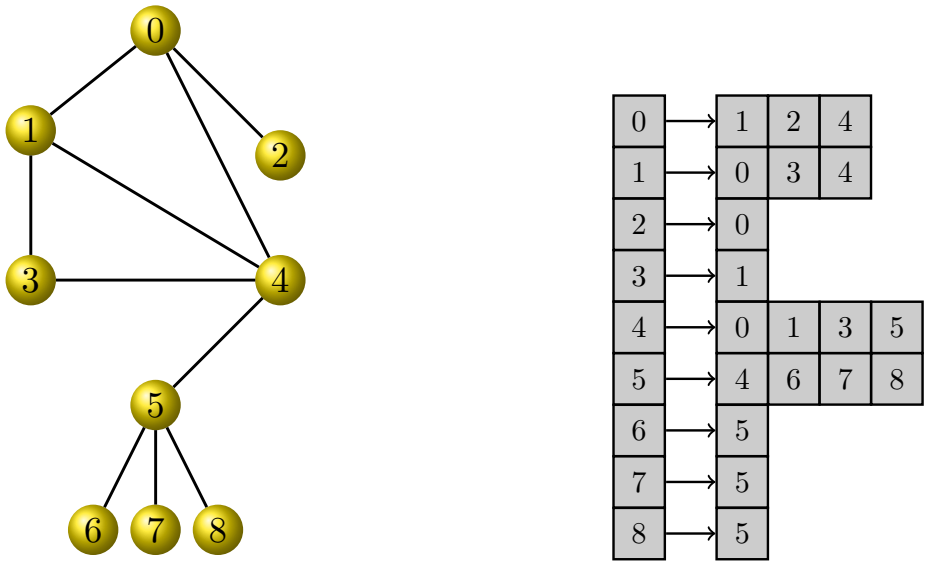
$$a_{ij} = \begin{cases} 1, & \text{if } e_{ij} \in E \\ 0, & \text{if } e_{ij} \notin E \end{cases} \tag{2.6}$$

Za težinske grafove vrijednost elemenata u matrici su težine na bridovima w_{ij}.

$$a_{ij} = \begin{cases} w_{ij}, & \text{if } e_{ij} \in E \\ 0, & \text{if } e_{ij} \notin E \end{cases} \tag{2.7}$$

Matrica susjedstva se koristi za guste grafove koji nemaju velik broj nula u svom zapisu pomoću matrice susjedstva. Pristup elementima matrice odvija se u konstantnom vremenu, ali prostorno matrica zauzima $n \times n$ memorijskih lokacija, što za rijetke grafove nije prihvatljivo. Osim toga, da bi našli sve vrhove incidentne s nekim vrhom v_{ij} trebamo proći sve elemente u i-tom retku čak i ako je v_{ij} malog stupnja.

Lista susjedstva Kod liste susjedstva se, za svaki vrh u grafu, kreira lista njegovih susjeda (slika 2.4). To je kompaktan način zapisivanja grafa čija je prostorna složenost proporcionalna broju vrhova i bridova u grafu. Pronalazak susjednih vrhova nekog vrha v je proporcionalan sa stupnjem vrha, pa je za taj problem ovaj zapis bolji od matrice susjedstva. Međutim, za ispitivanje da li su dva vrha susjedna potrebno je proći cijelu listu susjedstva jednog vrha, što u najgorem slučaju može biti $n - 1$ vrhova u jednostavnom grafu.



(a) graf (b) lista susjedstva

Slika 2.4: Prikaz grafa kao liste susjedstva

Prikaz grafova u formi matrica doveo je do razvoja grane teorije grafova - *computational* teorije grafova, koja je dobila na važnosti sa razvojem računala i tehnologija. Kako istovremeno grafovi realnog svijeta postaju sve veći, ali i rijeđi, bitno je odabrati što kompaktnije strukture podataka koje bi se koristile za implementaciju grafa. Jedan od načina je korištenje liste susjedstva koja sprema sve bridove grafa na način da u jednoj listi čuva sve vrhove i njihove susjede, a u drugoj broj koji određuje pomak (engl. *offset*) od početka liste na kojem počinje zapis sljedećeg vrha i njegovih susjeda (slika 2.5).

C = [1, 2, 4, 0, 3, 4, 0, 1, 0, 1, 3, 5, 4, 6, 7, 8, 5, 5, 5]
R = [0, 3, 6, 7, 8, 12, 16, 17, 18]

Slika 2.5: Kompaktni zapis pomoću pomaka grafa sa slike 2.4a

Uređene trojke Najjednostavniji način za implementaciju rijetke matrice je u formi niza uređenih trojki (engl. *triples*) gdje se za rijetku matricu $M \in \mathbb{R}^{m \times n}$ za svaki $M(i,j) \neq 0$ spremi u memoriju trojka $(i, j, M(i,j))$. Trojke u nizu mogu biti nesortirane, sortirane po redovima ili sortirane po redovima i po stupcima [14].

Compressed Row Storage (CRS) i Interleaved Compressed Row Storage (ICRS) format Format CRS za rijetku matricu $M \in \mathbb{R}^{m \times n}$ sa nnz ne-nula elemenata (engl. *number of nonzero elements*) definiran je sa tri vektora. Prvi vektor $val \in \mathbb{R}^{nnz}$sastoji se od ne-nula vrijednosti iz matrice M zapisanih redom prolazeći kroz matricu redak po redak. Drugi vektor $col \in \mathbb{R}^{nnz}$ sastoji se od pripadajućih indeksa stupaca u kojima se ne-nula elementi nalaze, dok su u trećem vektoru $roff \in \mathbb{R}^m$ indeksi koji označavaju početak pojedinog retka (engl. *row offset*) u vektorima val i col. Osim navedena tri vektora nekad se koristi i vektor $rcnt \in \mathbb{R}^m$ u kojem je zapisan broj ne-nula elemenata za svaki redak.

U sljedećem primjeru prikazan je zapis rijetke matrice u CRS formatu. Ideksi redaka i stupaca počinju od nule. Neka je dana matrica $M \in \mathbb{R}^{m \times n}$ rijetka matrica, gdje je $m = 5$, $n = 5$ i $nnz = 8$.

$$M = \begin{bmatrix} 0 & 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 10 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ -1 & 0 & 0 & 0 & 5 & 0 \\ 0 & -2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \end{bmatrix}$$

Tada su vrijednosti vektora $val \in \mathbb{R}^{nnz}$, $col \in \mathbb{R}^{nnz}$, $roff \in \mathbb{R}^m$ i $rcnt \in \mathbb{R}^m$ sljedeće:

$$val = \begin{bmatrix} 2 & 1 & 10 & 2 & -1 & 5 & -2 & 2 \end{bmatrix}$$
$$col = \begin{bmatrix} 1 & 2 & 2 & 4 & 0 & 4 & 1 & 3 \end{bmatrix}$$
$$roff = \begin{bmatrix} 0 & 2 & 3 & 4 & 6 & 7 \end{bmatrix}$$
$$rcnt = \begin{bmatrix} 2 & 1 & 1 & 2 & 1 & 1 \end{bmatrix}$$

Opisani format nije pogodan za paralelizaciju, pa se često koristi ICRS u kojem su u vektoru vrijednosti val navedene ne-nula vrijednosti u blokovima od L elemenata, gdje su u svakom bloku po jedna ne-nula vrijednost iz svakog retka. Ako neki redci imaju manje ne-nula elemenata, u tom vektoru će se pojaviti rupe (engl. *holes*). To utječe na veličinu zapisa tipično $5 - 10\%$, ali se ne smatra kritičnim jer je efikasnost veća nego kod CRS zapisa. Tipična veličina bloka je višekratnik broja 16 [45]. Za matricu M iz prethodnog

primjera i za veličinu bloka 8, vrijednosti vektora val, col, $roff$ i $rcnt$ su sljedeće:

$$val = \begin{bmatrix} 2 & 10 & 2 & -1 & -2 & 2 & - & - & 1 & - & - & 5 & - & - & - & - \end{bmatrix}$$
$$col = \begin{bmatrix} 1 & 2 & 4 & 0 & 1 & 3 & - & - & 2 & - & - & 4 & - & - & - & - \end{bmatrix}$$
$$roff = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & - & - \end{bmatrix}$$
$$rcnt = \begin{bmatrix} 2 & 1 & 1 & 2 & 1 & 1 & - & - \end{bmatrix}$$

Slično CRS formatu, za zapis grafova koristi se i *Compressed Column Storage* format, koji se još zove i Harwell-Boeing Format [26] u kojem se umjesto redova koriste stupci.

3 Graf algoritmi

Algoritme na grafovima generalno možemo podijeliti u dvije osnovne grupe [112]

- algoritmi prelaska preko grafa (engl. *traversal algorithms*) u kojima se prelazi po svim vrhovima grafa, jednom ili više puta,
- analitički iterativni algoritmi (engl. *analytically iterative algorithms*) u kojima se iterativno prelazi preko cijelog grafa sve dok se ne zadovolji neki uvjet konvergencije.

U algoritme prelaska preko grafa spadaju algoritmi za pretragu u dubinu (DFS, engl. *depth first search*) i u širinu (BFS, engl. *breath first search*), traženje najkraćih putova u grafu (engl. *single source shortest path*), minimalno razapinjuće stablo (engl. *minimum spanning tree*) i ispitivanje povezanosti (engl. *connectivity algorithms*).

Analitički iterativni algoritmi su PageRank algoritam, baziran na slučajnoj šetnji u grafu, komponente povezanosti (engl. *connected components*), detekcije zajednice (engl. *community detection*) i brojanje trokuta (engl. *triangle counting*).

Druga podjela algoritama dana je u [99]. Po toj podjeli algoritmi se mogu podijeliti u

- algoritme prelaska preko grafa,
- algoritme slučajne šetnje,
- algoritme agregacije (engl. *graph aggregation*) u kojima se računa statistika na kompletnom grafu.

Algoritmi agregacije su algoritmi ukrupnjavanja (engl. *graph coarsening*) koji su početna točka za particioniranje grafa u distribuiranim sustavima, sparsifikacija grafa (engl. *graph sparsification*) u kojima se smanjuje broj bridova u grafu i na taj način pojednostavljuje podjela na klastere, te algoritmi sumarizacije (engl. *graph summarization*) koji se kreću od računanja distribucije stupnjeva i udaljenosti među vrhovima do SNAP (*Summarization by Grouping Nodes on Attributes and Pairwise Relationships*) algoritama u kojima se sumarizacija radi na temelju korisnički definiranih svojstava vrhova i relacija među vrhovima.

Nadalje, Wang i dr. [105] dijele algoritme na

- fundamentalne algoritme za analizu grafova:
  - pretraživanje grafa (prelazak preko grafa (engl. *graph traversal*) i algoritmi za pretraživanje (engl. *search algorithms*)),
  - pronalaženje uzoraka (algoritmi najkraćeg puta, algoritmi mapiranja (engl. *matching algorithms*)),
  - centralnost grafa (engl. *centrality computing algorithms*),
  - rangiranje liste (engl. *list ranking algorithms*)),
  - particioniranje (povezane komponente (engl. *connected components algorithms*),
  - rezovi na grafu (engl. *graph-cut algorithms*)),

- napredne algoritme za analizu grafova:
  - indeksiranje i rangiranje (*pagerank*),
  - rudarenje podataka (algoritmi za klasteriranje i klasifikaciju),
  - strukturirani upiti (npr. RDF upiti),
  - algoritmi za preporuku (engl. *recommendation*).

U sljedećim podpoglavljima opisat ćemo neke od fundamentalnih algoritama za analizu grafa.

3.1 Osnovni graf algoritmi

3.1.1 Najkraći putevi u grafu

Naći najkraći put između dva vrha u grafu potrebno je u svakom navigacijskom softveru, softveru za planiranje avionskih ruta (u svijetu je 2013. godine bilo 41821 aerodroma[1]), planiranje transportnih ruta i slično. Najpoznatiji algoritam koji računa duljinu najkraćeg puta iz jednog vrha prema svim ostalim vrhovima u grafu je Dijkstra algoritam za usmjerene težinske grafove, čija se složenost kod jednostavne implementacije $O(|V|^2)$ može spustiti do $O(|E| + |V|log|V|)$ implementacijom pomoću Fibonacci hrpe [31], te **Bellman-Ford** algoritam koji je, iako sporiji od Dijkstra algoritma, primjenjiv na širi spektar problema uključujući one kod kojih težine vrhova mogu biti negativne. Složenost Bellman-Ford algoritma je $O(|V||E|)$ [39].

Osim traženja najkraćeg puta iz jednog vrha, čest je problem traženja svih najkraćih puteva u grafu (engl. *all pairs shortest path*). Postoje dvije vrste algoritama za taj problem, jedni koriste Dijkstrin algoritam koji se izvodi za svaki vrh u grafu, a drugi koriste **Floyd-Warshall** algoritam koji radi i za grafove koji mogu imati negativne težine bridova [85]. Floyd-Warshall algoritam ima složenost $O(|V|^3)$, pa nije dobar za izvođenje na jako velikim grafovima, međutim, kako koristi matričnu reprezentaciju, pogodan je za paralelizaciju [17].

3.1.2 Obilazak grafa

Algoritmi za obilazak grafa osnovni su algoritmi u teoriji grafova. Koriste se za traženje komponenti u grafu, traženje ciklusa, bipartitivnost grafa (bojanje vrhova sa dvije boje), traženje vrhova u grafu čijim se ukidanjem graf raspada na komponente i sl.

Pretraživanje u širinu (BFS, engl. *breath-first search*) pretražuje graf tako da redom posjećuje sve vrhove na istoj udaljenosti od ishodišnog vrha, počevši od udaljenosti 1, pa dok se ne obiđu svi vrhovi. Iz početnog vrha graf se obilazi tako da se posjete vrhovi koji su susjedni ishodišnom vrhu, a zatim se postupak ponavlja od tih susjednih vrhova, s tim da se ne posjećuju vrhovi koji su već bili posjećeni. Algoritam koristi strukturu podataka red (engl. *queue*) u kojoj sa početka dohvaća posjećeni vrh, a na kraj dodaje njegove susjede. Složenost ovog algoritma je $O(|V| + |E|)$, a za rijetke grafove kod kojih je $|E| = O(|V|)$ složenost je $O(|V|)$ [66].

S druge strane, **pretraživanje u dubinu** (DFS, engl. *depth-first search*) koristi strukturu podataka stog (engl. *stack*) iz kojeg se s početka uzima vrh, te se na početak stavljaju

[1] https://www.cia.gov/library/publications/the-world-factbook/fields/2053.html

svi njegovi susjedi. Na taj način pretraživanje ide od početnog vrha te obilazi (i obilježava) vrhove na udaljenosti $1, 2, \ldots$ sve dok ne dođe do lista u grafu ili dođe do već posjećenog vrha. Kad dođe do toga, vrati se do onog vrha čiji susjedi još nisu ispitani i nastavlja s istim postupkom.

Za traženje najkraćeg puta u netežinskom grafu pretraživanje u širinu daje optimalan rezultat, tj. sigurno daje najkraći put. Složenost oba algoritma je $O(|V| + |E|)$.

Ispitivanje povezanosti u grafu radi se pomoću algoritama za pretraživanje. Ako je broj vrhova koje možemo obići počevši od vrha v jednak ukupnom broju vrhova u grafu, graf je povezan. Ako nije svaka sljedeća komponenta u grafu se detektira tako da se obilazak grafa nastavi od vrha koji ne pripada već detektiranim komponentama.

3.1.3 Minimalno razapinjuće stablo

Jedan od čestih optimizacijskih problema je traženje minimalnog razapinjućeg stabla (engl. *minimal spanning tree, MST*) u grafu. MST je stablo koje sadrži sve vrhove težinskog grafa, a suma težina bridova u njemu je minimalna. Koristi se prilikom dizajniranja telekomunikacijskih i računalnih mreža, transportnim mrežama, procesiranju slika, prepoznavanju govora (engl. *speech recognition*), a često je i podproblem nekog drugog problema kao što je problem trgovačkog putnika ili problem sparivanja [43]. Klasični algoritmi za traženje MST-a su Boruvka, Primov i Kruskalov algoritam za težinske povezane grafove. Pritom je složenost Primovog algoritma $O(|V|^2)$, a Kruskalovog $O(|E|log_2|E|)$. Veliki broj paralelnih algoritama za traženje minimalnog razapinjujućeg stabla bazirano je na Boruvka algoritmu kod kojeg složenost originalnog sekvencijalnog algoritma iznosi $O(|E|log|V|)$ [94] [18].

3.1.4 Traženje maksimalne klike u grafu

Klika (engl. *clique*) u grafu je potpun podgraf u grafu, tj. takav podgraf da su svi vrhovi unutar njega povezani. Problemi vezani uz detektiranje klike su traženje maksimalne klike u grafu po broju vrhova, traženje maksimalne klike u grafu po težini i za danu veličinu traženje klike koja je veća od te veličine. Svi ti problemi spadaju u NP-potpune probleme, pa se rješavaju optimizacijskim metodama. Traženje klika može biti interesantan problem kod proučavanja socijalnih mreža, a čest je u računalnoj kemiji (engl. *computational chemisty*) gdje se primjenjuje na problem pronalaženja podstruktura molekula sa nekim zajedničkim svojstvom [34].

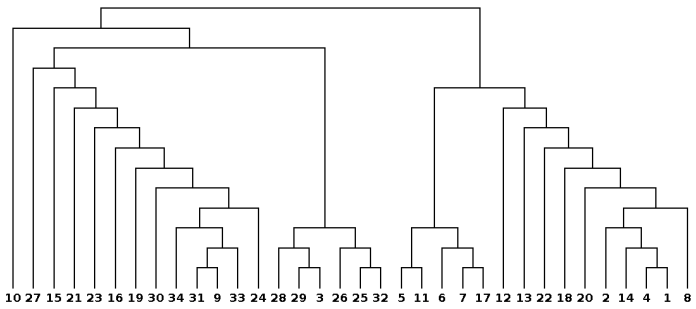
3.1.5 Detekcija zajednica

Metode za detekciju zajednica u kompleksnim mrežama baziraju se na hijerarhijskim metodama za grupiranje (engl. *hierarchical clustering method*) koje mogu biti aglomerativne i divizivne.

Hijerarhijsko aglomerativno grupiranje podataka. Tradicionalne hijerarhijske aglomerativne metode za grupiranje podataka kreću od grafa koji se sastoji samo od vrhova, a potom se dodaju bridovi i to tako da se krene od "jačih" prema "slabijim" bridovima gledajući težinu brida.

Težina brida između dva vrha može se računati na različite načine. Primjerice, težina brida može biti broj vršno-nezavisnih putova između dva kraja brida ili broj bridno-nezavisnih putova. Dva puta koja spajaju vrhove su **vršno-nezavisni** ako imaju zajednički samo početni i krajnji vrh, a **bridno-nezavisni** ukoliko nemaju niti jedan zajednički brid. Broj takvih putova između dva vrha ustvari označava broj vrhova (ili bridova) koje treba izbaciti iz grafa da bi vrhovi postali nepovezani [38]. Nadalje, težina brida može biti ukupan broj putova između dva vrha duljine l pomnožen sa α^l, gdje je α manji od recipročne vrijednosti najveće svojstvene vrijednosti matrice susjedstva (α se uvodi zbog konvergencije). U općenitom slučaju potrebno je izračunati težine svih $\frac{1}{2}n(n-1)$ parova vrhova, bez obzira da li su povezani, no u većini metoda nepovezanim vrhovima se pridjeljuje vrijednost 0 [78].

Proces dodavanja najjačih veza može se zaustaviti u bilo kojem trenutku, ali da bi se potpuno ispitala struktura zajednica u mreži potrebno je proces završiti do kraja. Kada su dodani svi bridovi, svi vrhovi su povezani pa tvore jednu zajednicu. Ovaj postupak se prikazuje **dendrogramom** tj. stablom koje prikazuje kako su se vrhovi i skupovi vrhova međusobno povezivali (slika 3.1).

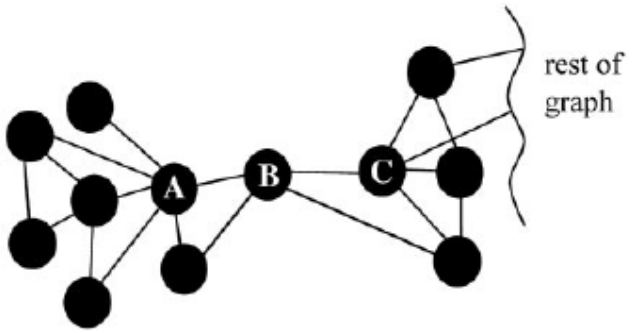


Slika 3.1: Dendrogram

Iako tradicionalna hijerarhijska metoda daje dobre rezultate u nekim slučajevima, pokazalo se da rezultati mogu biti bolji. Naime, vrhovi koji su spojeni samo jednim bridom sa ostatkom grafa dugo će biti izolirani, jer njihovi pripadajući bridovi imaju malu težinu, te se zbog toga dodaju na samom kraju.

Divizivne metode. Obratno od hijerarhijskog aglomerativnog grupiranja podataka, divizivne metode kreću od kompletnog grafa i iz njega izbacuju bridove sa najvećom težinom. Postupak izračunavanja težine svih (preostalih) bridova se prilikom svakog izbacivanja brida ponavlja, jer se izbacivanjem brida mijenja težina preostalih bridova. Takvo grupiranje provodi Girvan-Newman algoritam. Međutim, postoje modifikacije algoritama u kojima se mijenja poredak kojim su bridovi sa najvećim težinama uklonjeni stvarajući više mogućih struktura zajednica. Naime, u stvarnom svijetu moguće je da je neki vrh "između" dvije zajednice. Wilkinson i dr. navode mrežu funkcionalno povezanih gena u kojoj je to česta pojava [107]. Na slici 3.2 prikazan je graf u kojem nije jasno kojoj zajednici treba pripadati vrh B. Girvan-Newman algoritam će ukloniti brid BC kao brid sa najvećom težinom, iako brid AB ima jednako veliku težinu. Uklanjanjem brida BC, brid AB će postati dio zajednice u kojoj je vrh A i u sljedećoj iteraciji više neće imati

veliku težinu. Na taj način vrh B je pripao zajednici u kojoj je vrh A. Uklanjanjem brida AB i BC u istoj iteraciji, vrh B bi ostao izoliran. Zato Wilkinson i dr. u [107] predlažu metodu kojom se kreira više rastava na zajednice, a onda se analizom zajednica ocjenjuje pripadnost vrha pojedinoj zajednici.



Slika 3.2: Neodređenost pripadnosti zajednici vrha B [107]

3.1.6 PageRank algoritam

Inicijalno je PageRank algoritam nastao za potrebe web pretraživanja kako bi rangirao stranice na webu, ali se njegova primjena proširila i na druga područja. Jedan od primjera je predviđanje koje vrste bi mogle izumrijeti koje se vrši primjenom modificiranog PageRank algoritma na hranidbenu mrežu [3]. Drugi primjer su sustavi za preporuku (engl. *recommendation systems*) [73]. Nadalje, postoji primjena u kemiji gdje se algoritam primjenjuje na mrežu molekula vode gdje su molekule vrhovi u grafu, a bridovi su spojevi veze atoma vodika među molekulama [48].

PageRank algoritam računa važnost pojedinog vrha usmjerenog grafa. Za svaki vrh ulazni brid smatra se glasom (engl. *vote*). Međutim, svi bridovi nisu jednako važni, važno je iz kojih vrhova dolaze (slika 3.3a). Važnost vrha proporcionalna je važnosti vrha s koje dolazi brid. Primjerice, ako iz vrha v koji ima važnost r_v ide n veza prema drugim vrhovima, onda svaki vrh prema kojoj ide brid dobije $\frac{r_v}{n}$ glasova. Važnost (engl. *rank*) svakog vrha je suma glasova koje je dobio vrh putem ulaznih bridova.

Za graf na slici 3.3b možemo napisati sustav jednadžbi:

$$r_y = \frac{r_y}{2} + \frac{r_a}{2} \tag{3.1}$$

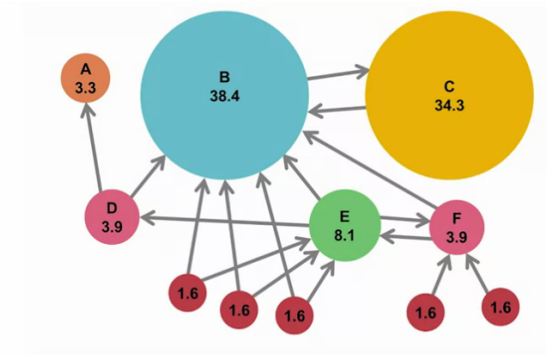
$$r_a = \frac{r_y}{2} + r_m \tag{3.2}$$

$$r_m = \frac{r_a}{2} \tag{3.3}$$

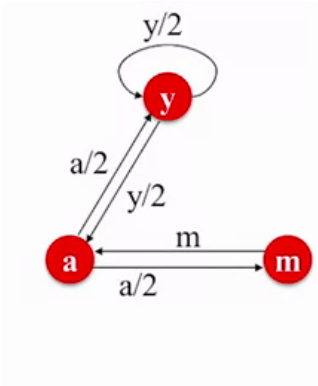
Kako gornje jednadžbe nemaju jedinstveno rješenje, uvodi se ograničenje

$$r_y + r_a + r_m = 1 \tag{3.4}$$

Za rješavanje sustava jednadžbi koristi se stohastička matrica susjedstva M. Neka je d_i



(a)



(b)

Slika 3.3: Važnost vrha u PageRank algoritmu ovisi o važnosti vrhova s kojih dolaze bridovi [61]

broj izlaznih bridova iz vrha i. Ako postoji brid iz i u j ($i->j$), $M_{ji} = \frac{1}{d_i}$, inače $M_{ji} = 0$. Suma elemenata svakog stupca je 1, pa je matrica stohastička po stupcima.

Nadalje, važnost vrha i je r_i, pa važnosti svih vrhova možemo predstaviti vektorom r takvim da je $\sum_i r_i = 1$.

Sada jednadžbu

$$r_j = \sum_{i->j} \frac{r_i}{d_i} \tag{3.5}$$

možemo zapisati u matričnom obliku

$$r = M \cdot r \tag{3.6}$$

pa je vektor r svojstveni vektor (engl. *eigenvector*) matrice za svojstvenu vrijednost (engl. *eigenvalue*) 1. Matrica ima svojstvenu vrijednost 1, jer je stohastička po stupcima, a kako je i vektor r stohastički, vrijedi $Mr \leq 1$. Nadalje, matrica M i M^T imaju iste svojstvene vrijednosti. Kako vrijedi $M^T \cdot e = e$ za vektor $e = \mathbf{1}$ koji se sastoji od samih jedinica, jer je suma elemenata redka matrice M^T jedan, to je $\lambda = 1$ svojstvena vrijednost i matrice M.

Za računanje svojstvenog vektora koristi se metoda iteracije eksponenta (engl. *power iteration*). Neka je $r_j^{(t+1)} = \sum_{i->j} \frac{r_i^{(t)}}{d_i}$, gdje je d_i izlazni stupanj vrha i.

- Pretpostavimo da graf ima n vrhova.
- Inicijaliziramo: $r^{(0)} = [\frac{1}{n} \dots \frac{1}{n}]^T$
- Iteriramo: $r^{(t+1)} = M \cdot r^{(t)}$
- Zaustavljamo se kada je postignuta željena preciznost npr. $\left|r^{(t+1)} - r^{(t)}\right|_1 < \varepsilon$

Norma $|x|_1 = \sum_{1 \leq i \leq n} |x_i|$ je L_1 norma, a može se koristiti bilo koja norma vektora.

Algoritam se bazira na slučajnim šetnjama kroz graf. U trenutku t šetač se nalazi na vrhu i. U trenutku $t+1$ odabrat će uniformno slučajni brid prema vrhu j. Proces se ponavlja beskonačno.

Neka je $p(t)$ vektor čija je i-ta koordinata vjerojatnost da je šetač na vrhu i u trenutku t. Funkcija p je distribucija vjerojatnosti. Tada je $p(t+1) = M \cdot p(t)$. Stacionarna distribucija slučajne šetnje $p(t)$ je stanje u kojem je $p(t+1) = M \cdot p(t) = p(t)$. Kako vektor važnosti r zadovoljava jednadžbu $r = M \cdot r$, r je stacionarna distribucija slučajne šetnje.

Da bi rješenje konvergiralo, trebamo analizirati probleme paukove zamke (engl. *spider trap*) i slijepe ulice (engl. *dead end*). Paukova zamka je problem kada u sljedećoj iteraciji dobijemo rezultat koji je već prije bio izračunat (izlazni bridovi su unutar male grupe vrhova, pa se slučajna šetnja odvija između tih vrhova). Problem slijepe ulice je situacija kada vrh nema izlaznih bridova, pa matrica M neće imati stupac sume 1 za taj vrh.

Problem paukove zamke može se riješiti tako da se u svakom koraku algoritma uvede vjerojatnost β s kojom se odabire jedan od bridova iz vrha na kojoj se trenutno nalazi šetnja, a sa vjerojatnošću $1-\beta$ se odabire bilo koji slučajno odabrani vrh (teleportacija). Vrijednost β je obično između 0.8 i 0.9.

Problem slijepe ulice rješava se tako da se šetnja sa vjerojatnošću $1/n$ nastavi sa slučajno odabranim vrhom. Odlazak na slučajni vrh omogućava proširenje grafa sa bridovima koji idu od trenutnog vrha prema slučajno odabranom vrhu. Sada matrica M u stupcu koji pripada vrhu koji nije imao izlaznih bridova, ima vrijednosti $1/n$. Tako smo dobili stahastičku matricu

$$A = M + a^T(\frac{1}{n}e) \tag{3.7}$$

gdje je

$$a = \begin{cases} 1, & \text{if } d_{out}(i) = 0 \\ 0, & \text{else} \end{cases} \tag{3.8}$$

Sada jednadžbu 3.5 možemo proširiti [82]

$$r_j = \sum_{i->j} \beta\frac{r_i}{d_i} + (1-\beta)\frac{1}{n} \tag{3.9}$$

Nadalje, kreira se Google matrica A koja je stohastička, neperiodična i ireducibilna, pa zadovoljava uvjete Markovljeve matrice za konvergenciju prema jedinstvenom pozitivnom stacionarnom vektoru. Matrica A dobijena je na sljedeći način

$$A = \beta \cdot M + (1-\beta)\frac{1}{n}e \cdot e^T \tag{3.10}$$

pa se metoda iteracije eksponenta primjenjuje da bi se izračunali svojstveni vektori matrice A iteriranjem $r^{(t+1)} = A \cdot r^{(t)}$.

3.1.7 Particioniranje grafa

Za razliku od metoda za detektiranje zajednica u grafu kojima se postiže rastav grafa na particije u kojima je broj bridova unutar particije veći, a između particija manji, te su particije nejednake veličine, kod particioniranja grafa broj i veličina particija su zadane i ne ovise samo o strukturi grafa. Zahtjevi pri particioniranju su takvi da particije moraju biti približno iste veličine ili težine, gdje težina particije može biti suma težina vrhova, a suma težina bridova koje povezuju particije što manja.

Particioniranje grafa je NP-problem, pa se koriste heuristički algoritmi koji traže

rješenja blizu optimalnih. Podjela samog grafa može se vršiti rekurzivno, dijeleći svaki put graf na dva dijela, ili podjelom na k particija odjednom.

Particioniranje se vrši izbacivanjem vrhova ili izbacivanjem bridova. Particioniranje grafa izbacivanjem vrhova se zove *vertex-cut partitioning*, a izbacivanjem bridova *edge-cut partitioning*. Cilj je naći skup vrhova ili bridova čijim se izbacivanjem graf raspada na dvije ili više particija približno jednake težine. Takav skup vrhova ili bridova nazivamo rez (engl. *cut*).

Najjednostavnija metoda particioniranje grafa je metoda temeljena na pretraživanju u širinu. Kako pretraživanje u širinu prelazi preko grafa tako da redom obilazi vrhove na jednakoj udaljenosti od ishodišnog vrha te kada obiđe sve vrhove na jednakoj udaljenosti prelazi na sljedeći nivo, vrhovi sa jednakom udaljenošći mogu se označiti istom brojčanom oznakom. Tada se particije kreiraju tako da se u jednu particiju stave vrhovi čija je oznaka manja od nekog zadanog broja, a u drugu preostali vrhovi.

Kernighan-Lin algoritam [63] polazi od podjele grafa na dva proizvoljna dijela jednake veličine, te u svakom koraku kreira novu particiju zamjenjujući parove vrhova iz različitih particija tako da se odabiru vrhovi čija će zamjena dovesti do najveće promjene veličine reza. Postupak se ponavlja tako da se isključe već zamjenjeni vrhovi sve dok više nema vrhova koji bi mogli biti zamjenjeni. Kad su sve zamjene napravljene, među particijama se bira ona koja ima najmanju veličinu reza. Kako rezultat ovisi o početnoj particiji, metoda se koristi za unaprijeđivanje neke već drugom metodom napravljene particije. Još jedna heuristička metoda bazirana na Kernighan-Lin algoritmu koja se primjenjuje na hipergrafove je **Fiduccia-Mattheyses** algoritam [91]. U njemu se po jedan vrh prebacuje u drugu particiju u svakoj iteraciji.

Ostali algoritmi za particioniranje koriste **spektralnu teoriju grafova**. Spektar grafa je skup svojstvenih vrijednosti matrice susjedstva grafa, ali spektralna teorija se primjenjuje i na drugim matricama korištenim u teoriji grafova (matrica centralnosti, matrica modularnosti,...). Jedna od tih matrica je i Laplacova ili Kirchoffova matrica $L(G)$ koja se dobije tako da se od dijagonalne matrice $D = diag(d_1, d_2, ..., d_n)$, u kojoj su d_i stupnjevi i-tog vrha, oduzme matrica susjedstva. Tako dobijemo $L = D - A$ odnosno

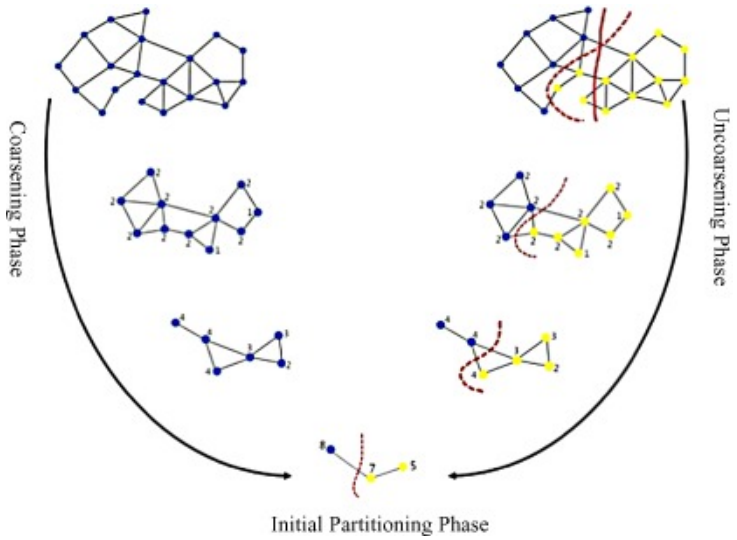
$$l_{ij} = \begin{cases} d_i, & \text{if } i = j \\ -1, & \text{if } v_{ij} \in E(g) \\ 0, & \text{if } v_{ij} \notin E(g) \end{cases} \tag{3.11}$$

Matrica L je (za neusmjerene grafove) simetrična i pozitivno definitna ($x^T Lx \geq 0, \forall x \in R^n$), pa su joj svojstvene vrijednosti nenegativne [51]. Jedna svojstvena vrijednost je 0 sa pripadajućim jediničnim svojetvenim vektorom $(1, 1, ..., 1)^T$, a druga najmanja svojstvena vrijednost određuje particioniranje grafa na dvije particije. Pripadajući svojstveni vektor se zove Fiedlerov vektor, a druga najmanja svojstvena vrijednost λ_2 predstavlja algebarsku povezanost (engl. *algebraic connectivity*) grafa i njena vrijednost je proporcionalna sa povezanošću grafa. Druga najmanja svojstvena vrijednost i njen pripadajući svojstveni vektor računa se Lanczosovim algoritmom [83], a particije veličine n se kreiraju tako da se vrhovi kojima odgovara prvih n najvećih vrijednosti Fiedlerovog vektora smještaju u jednu particiju, a ostali u drugu particiju [74].

Fidlerov vektor koristi se i za detekciju zajednica u mreži pri čemu se vektor sortira te se napravi $|V| + 1$ rezova između kojih se odabire onaj sa najmanjom težinom. Particioniranje

grafa pomoću Fiedlerovog vektora zove se spektralna bisekcija (engl. *spectral bisectioning*). Rekurzivnim ponavljanjem algoritma za spektralnu bisekciju graf se može particionirati na više disjunktnih skupova.

Višerazinsko particioniranje grafa (engl. *multilevel graph partitioning*) bazira se na iterativnom pristupu u kojem se u svakoj iteraciji provode tri faze (slika 3.4) [92]. Prva faza je sažimanje (engl. *contraction, coarsening*) u kojoj se grupiraju vrhovi (obično aglomerativnim heurističkim metodama) u klastere, te se unutar klastera svi vrhovi zamjenjuju jednim, a svi bridovi između njih brišu. Na taj način se kreira graf manje veličine koji zadržava strukturu originalnog grafa. Postupak se zaustavlja kada je graf dovoljno malen da se može particionirati nekim od već spomenutih algoritama za particioniranje. Ta faza zove se inicijalno particioniranje (engl. *initial partitioning*). U fazi lokalnog unaprijeđenja (engl. *local improvement*) sažeti vrhovi se ponovo vraćaju u originalne vrhove, a vrhovi se pomiču između particija kako bi se postigla optimalna particija.



Slika 3.4: Faze višerazinskog particioniranja grafa [104]

Pregled složenosti osnovnih algoritama na grafu možemo vidjeti u tablici 3.1

3.2 Paralelni algoritmi

Grafovi su pogodni za opisivanje relacija među diskretnim objektima. Međutim, problemi koji se modeliraju pomoću grafa i rješavaju pomoću algoritama nad grafovima postaju sve veći, pa samim tim algoritmi koji se izvode sekvencijalno na jednom procesoru i sa relativno malom količinom memorije vrlo brzo dođu do granice kada se graf ne može učitati u memoriju i kada izvođenje algoritama previše traje. Stoga se koristi linearna algebra i matematičke metode koje u nekim slučajevima podižu granicu izvodivosti algoritma (u smislu veličine grafa), ali nedovoljno za velike grafove stvarnog svijeta. Možemo reći da sekvencijalni graf algoritmi nisu skalabilni.

Skalabilnost i efikasnost se može postići paralelizacijom. Ona omogućuje podjelu problema na manje dijelove koji će se izvršavati istovremeno. Time se može postići znatno ubrzanje izvođenja, ali uz dodatnu komunikaciju među procesorima, dodatni pristup

Tablica 3.1: Tablica složenosti osnovnih graf algoritama

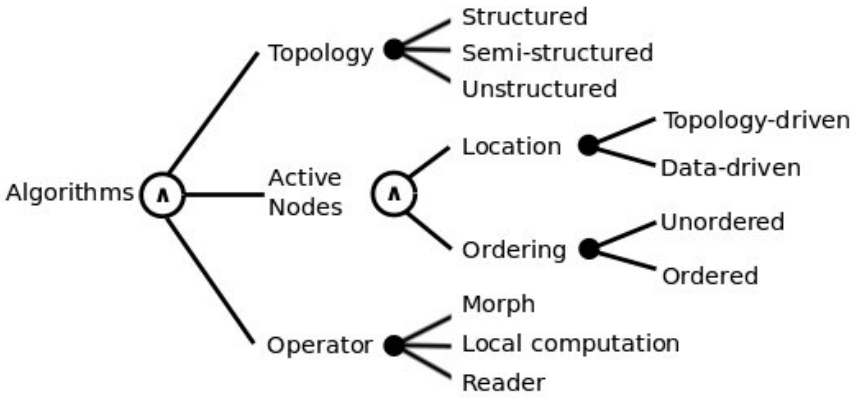
Algoritam	Složenost
Najkraći putevi u grafu iz jednog vrha	
Dijkstra (implementacija pomoću Fibonacci hrpe)	$O(\|E\| + \|V\|log\|V\|)$
Bellmand-Ford	$O(\|V\|\|E\|)$
Svi najkraći putevi u grafu	
Dijkstra	$O(\|V\|(\|E\| + \|V\|log\|V\|))$
Floyd-Warshall	$O(\|V\|^3)$
Pretraživanje u širinu	$O(\|V\| + \|E\|)$
Pretraživanje u dubinu	$O(\|V\| + \|E\|)$
Minimalno razapinjujuće stablo	
Boruvka algoritam	$O(\|E\|log\|V\|)$
Primov algoritam	$O(\|V\|^2)$
Kruskalov algoritam	$O(\|E\|log_2\|E\|)$
Struktura zajednice	
Girvan-Newman algoritam	$O(\|V\|^3)$
PageRank	$O(\|V\| + \|E\|)$

memoriji i složenije upravljanje poslovima (engl. *task management*). Da bi uspješno izvodili i skalirali paralelni algoritmi, nužno je da su dobro odabrani i povezani hardver na kojim će se izvoditi izračunavanja, softver koji će se koristiti te sami algoritmi. U numeričkom znanstvenom području postoji niz standardiziranih biblioteka (Boost, Cuda,...) koje zadovoljavaju potrebe velikog dijela numeričkih problema. Ipak, graf i graf algoritmi imaju specifičnosti koje mogu biti problem pri primjeni standardnih paralelnih sustava [65].

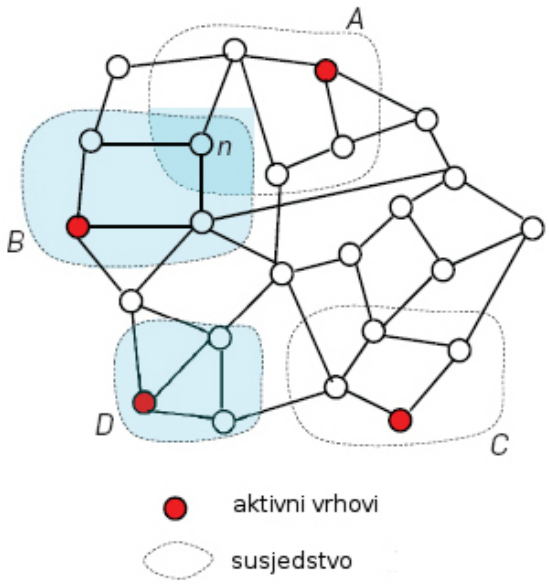
Grafovi su opisani neregularnim strukturama podataka (engl. *irregular data structures*) kao što su rijetke matrice, liste, stabla i sl. Algoritmi nad njima baziraju se na iterativnom izvođenju operacija nad dijelom vrhova (aktivni vrhovi) pa se operacije koje se ne preklapaju mogu izvoditi paralelno. Paralelna implementacija algoritama može biti *topology-driven* ili *data-driven*. Kod *topology-driven* implementacije svi vrhovi u grafu su aktivni, čak i ako nad nekima ne treba izvršiti nikakvu operaciju. S druge strane, *data-driven* implementacija izvodi, u jednom procesu ili *threadu*, operacije samo nad aktivnim vrhovima što zahtjeva praćenje stanja vrhova koji mogu postati aktivni kao posljedica izvršavanja drugog procesa ili niti (engl. *threada*) [72]. Graf algoritmi su uglavnom *data-driven*. Izvođenje ovisi o strukturi grafa koja nije *a priori* poznata, pa je particioniranje grafa težak problem. Nebalansiranost veličina particija može utjecati na skalabilnost. Osim toga, kod graf algoritama se javlja i problem lokalizacije (engl. *locality*). Naime, vrhovi koji su povezani ne moraju biti u istom skupu aktivnih vrhova, te stoga nisu u istom dijelu memorije tj. u istoj particiji. To za posljedicu ima da dohvaćanje podataka čini veliki dio ukupnog vremena izvršavanja algoritma, što je osnovni problem paralelnih implementacija.

U [60] je napravljena podjela algoritama nad grafovima s obzirom na tri kategorije ili dimenzije: topologiju grafa, lokaciju i poredak aktivnih vrhova i operator. Tu podjelu

nazivaju TAO analiza i možemo je vidjeti na slici 3.5. Podjela nije usko vezana za paralelne algoritme, već se i sekvencijalni algoritmi mogu promatrati kroz iste dimenzije. Topologija grafa je bitna kako bi se lakše odredili načini optimizacije algoritama [86]. Dimenzija aktivnih vrhova opisuje kako vrhovi postaju aktivni i u kojem poretku. Treća dimenzija je formulacija algoritama usmjerenih na podatke (engl. *data-centric*) koju nazivaju formulacija pomoću operatora. Operator predstavlja pravilo po kojem se radi *update* aktivnih vrhova. Svaka aktivnost operatora čita ili piše unutar dijela grafa koji su susjedni aktivnim vrhovima (slika 3.6). Operatori koji mogu mijenjati strukturu grafa tako što dodaju ili skidaju brid ili vrh iz grafa zovu se *morph* operatori. *Local computation* operatori mijenjaju vrijednosti labela vrhova, a ne mijenjaju samu strukturu grafa i to tako da čitaju labele susjednih vrhova te mijenjaju vrijednost aktivnih (*pull-style*) ili čitaju vrijednosti aktivnih vrhova te mijenjaju vrijednosti susjednih vrhova (*push-style*). Na kraju, operatori mogu biti čitači (engl. *reader*) ako se izvršavaju na *read-only* strukturi podataka.



Slika 3.5: TAO analiza [60]



Slika 3.6: Aktivni vrhovi i njihovo susjedstvo [60]

Pristup paralelnom rješavanju problema velikih grafova mora uzeti u obzir više parametara. Paralelno izvođenje uključuje prilagodbu algoritama za paralelno izvođenje, odabir arhitekture i samih modela programiranja. U sljedećim podpoglavljima predstavit će se različiti tipovi paralelnih arhitektura.

4 Graf orijentirane baze podataka

Tehnologije koje se bave izvođenjem algoritama nad velikim mrežama mogu se podijeliti u dvije grupe:

- S jedne strane su graf baze podataka koje, kao i tradicionalne baze podataka stavljaju fokus na trajnoj pohrani podataka (engl. *persistence*) i transakcijski sustav (OLTP, *Online Transaction Processing*) i koji omogućuju CRUD metode (Create, Read, Update, Delete) nad podacima.
- S druge strane su *Graph Computational Engine* ili *Graph Analytic Engine* koje izvode algoritme nad kompletnim grafovima koji su uglavnom pohranjeni na eksternim uređajima. Takvi sustavi služe za analizu grafa (OLAP, *Online Analytical Processing*) i uglavnom su distribuirani.

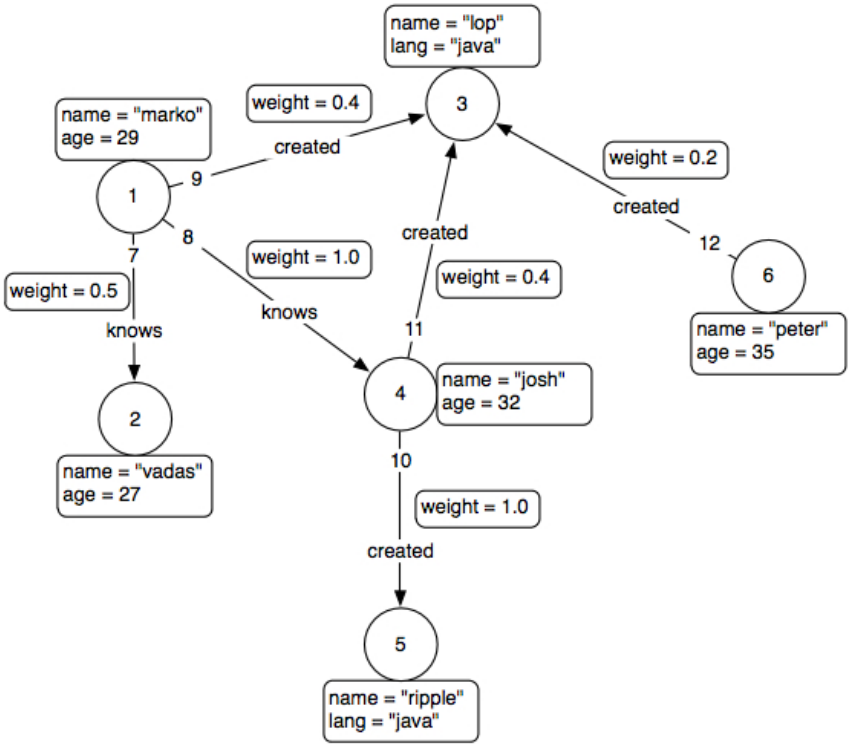
Primjer OLTP upita je da li postoji veza između dva čvora grafa, primjerice put između dva prijatelja u socijalnoj mreži, a primjer OLAP algoritma je računanje PageRanka za neku web stranicu u web grafu. Očito je kod OLTP upita bitno da je vrijeme odziva što manje, dok je za OLAP analitiku važan što veći protok (engl. *throughput*) i iskoristivost procesora.

OLTP upiti najbolje odgovaraju pretragama koje zahtijevaju pristup ograničenom skupu vrhova i koje koriste filtere ne bi li ograničili broj vrhova koji će se obići u potrazi za rješenjima. Kako za spremanje podataka OLTP koriste indeksiranje, broj zahtjeva za čitanjem na disku je ograničen. Takav ograničeni skup vrhova zajedno sa pripadajućim bridovima može se spremiti u memoriju, te se u memoriji mogu obavljati daljnji obilasci podgrafa. S druge strane, OLAP upiti su dobri za pretrage kojima se mora pristupiti velikom dijelu podataka, pa spremanje u memoriju nije efikasno. Graf se promatra kao niz zvjezdastih grafova koji se sastoje od vrha, njegovih svojstava, incidentnih bridova i njihovih svojstava, koji se onda linearno obrađuju, dok se svi ne obrade.

Graf orijentirane baze podataka najčešće koriste model grafa sa svojstvima (engl. *property graph*) u kojem je graf usmjeren multigraf s atributima i labelama. Pomoću takvog modela može se modelirati širok spektar drugih tipova grafova. Osnovne komponente modela su [1]:

- Graf – objekt koji sadrži vrhove i bridove.
  - Element – objekt koji ima jedinstveni identifikator, pridružen određeni broj ključ-vrijednost parova koji označavaju svojstva pridružena identifikatoru.
    - Vrh – objekt koji ima ulazne i izlazne bridove.
    - Brid – objekt koji ima početni i krajnji vrh i labelu koja označava tip relacije između pripadajućih vrhova.

Na slici 4.1 možemo vidjeti primjer takvog grafa. Graf baze podataka koriste se u područjima u kojima je bitna informacija o povezanosti kao što su socijalne mreže, sustavi za preporuku i u bioinformatici [70].



Slika 4.1: Primjer grafa sa svojstvima [1]

Prvi model pohrane podataka u formi grafa bio je model semantičke mreže [5] kojeg su predložili Roussopoulos i Mylopoulos još u 70-tim godinama prošlog stoljeća. Ipak relacijske baze podataka su s vremenom postigle potpunu dominaciju u pohrani podataka. Međutim, sada u XXI. stoljeću NoSQL (engl. *not only SQL*) baze se ponovo vraćaju na scenu, a glavni razlog za to je što su relacijske baze podataka neprikladne za velike nestrukturirane podatke koji se danas nesmiljeno produciraju. Za takve podatke potrebno je izvršavati tzv. *join* operacije za potrebe prelaska preko skupa podataka u potrazi za odgovorom na upit.

Važan pojam u implementaciji baza podataka je indeksiranje. U relacijskim bazama podataka pomoću indeksa se brzo pristupa ključu. Međutim, *join* operacija, kojom se spajaju dvije tablice u relacijskoj bazi, prolazi kroz sve indekse obiju tablica da bi pronašla podatke koji odgovaraju upitu. Kod graf baza podataka indeksi se koriste da bi se pristupilo početnom vrhu. Nakon toga, prolazi se grafom tako što se do sljedećeg vrha koristi incidentni brid. Takav pristup nazivamo susjedstvo bez indeksiranja (eng. *index-free adjancency*).

S aspekta baza podataka model podataka sastoji se od tri komponente:

- skup struktura podataka,
- skup pravila zaključivanja i
- pravila integriteta [4].

Strukture podataka odnose se na tipove objekata koji se koriste da bi modelirali podatke. Kod graf baza podataka to su grafovi sa svojstvima koji predstavljaju nadskup

grafova koji se mogu pojaviti pri modeliranju podataka. **Pravila zaključivanja** odnose se na jezike upita koji još uvijek nisu standardizirani. Tako neke graf baze podataka imaju svoj jezik upita (engl. *query language*), primjerice Neo4j baza sa jezikom Cypher, AllegroGraph realizira upite pomoću Prolog logičkog jezika, a većina graf baza pruža API u nekom od popularnih programskih jezika.

Što se ograničenja koja osiguravaju **integritet** tiče, većina današnjih graf baza podržava provjeru tipova i identiteta vrhova i bridova, ali ne više od toga (provjera kardinalnosti, integritet referenci – provjera da li postoje referencirani objekti i sl.).

Upiti nad grafovima se mogu podijeliti na [4]:

- upiti o susjedstvu,
- upiti dostupnosti, koji se odnose na provjeru da li postoji put između dva vrha u grafu,
- pronalaženje uzoraka u grafu (engl. *pattern matching*),
- agregacija (prosječne vrijednosti, suma, prebrojavanje, maksimum).

Najstarija i najkorištenija graf baza podataka je Neo4j sa jezikom upita Cypher [102]. Osim nje, popularne graf baze podataka uključuju Dex [68], AllegroGraph [2], HypergraphDB [50] i Titan [1].

5 Paralelni sustavi za procesiranje grafova

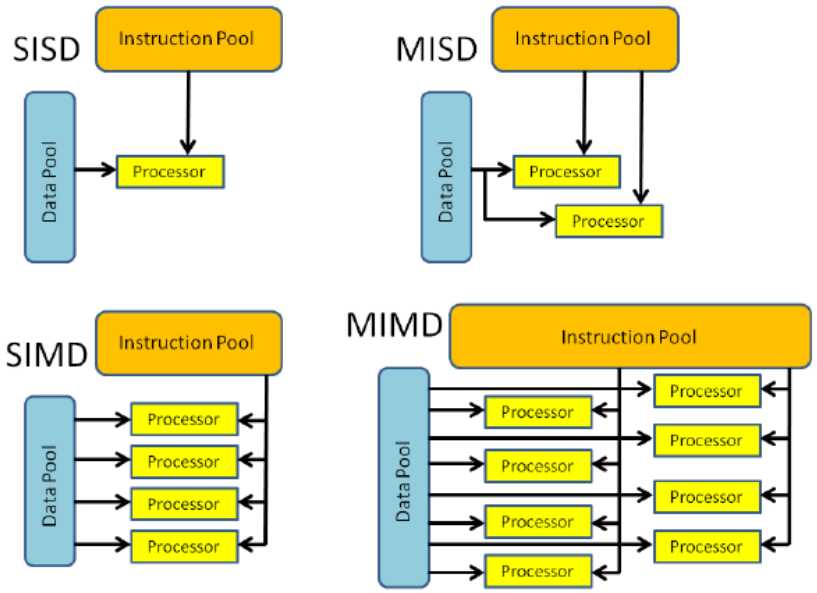
Prije nego krenemo u pregled paralelnih sustava za procesiranje grafova, navedimo i definirajmo osnovne pojmove koji se koristekada govorimo o paralelnom programiranju.

- **Propusnost (engl. *bandwidth*)** – Maksimalna količina (kapacitet) podataka koja u sekundi prolazi komunikacijskim kanalima (*bits per second*). Kod GPU usko grlo je memorija, a ne disk.
- **Protok (engl. *troughput*)** – Prosječan broj uspješno prenesenih poruka kroz komunikacijske kanale (*bits per second*).
- **Latencija (engl. *latency*)** – Vrijeme potrebno da se pošalje minimalna poruka iz jedne u drugu točku.
- **Ubrzanje paralelnog kôda (engl. *speedup of a parallel code*)** Omjer brzine izvođenja na N procesora u odnosu na vrijeme izvođenja na jednom procesoru. Važno je naglasiti da zbog potrebe za particioniranjem kopiranjem podataka ovo ubrzanje nikada nije N već manja vrijednost.
- **Uravnoteženje opterećenja (engl. *load balancing*)** – Distribucija podataka/posla po zadacima (engl. *tasks*) ili procesorima. Cilj je da procesori budu jednako opterećeni kako bi se postiglo manje vrijeme izvođenja.
- **Lokalnost (engl. *locality*)** – U bilo kojem trenutku program pristupa malom dijelu adresnog prostora. Spatial locality (podaci koji su blizu podacima kojima se zadnje pristupalo, vjerojatno će se pristupiti u budućnosti). Temporal locality (podaci kojima se već pristupalo, vjerojatno će se pristupiti u bliskoj budućnosti). CPU ne brine o lokalnosti, a GPU omogućava učitavanje podataka na on-chip memoriju prije nego što su potrebni.
- **Granularnost (engl. *granularity*)** – Udio izračunavanja (engl. *computation*) u odnosu na komunikaciju.
- **Fino granulirani (engl. *fine-grained*) vs. grubo granulirani (engl. *coarse-grained*) paralelizam** – Kod fino granulirane paralelizacije izračunavanje se izvodi nad malim dijelovima podataka uz čestu komunikaciju i sinkronizaciju (klasteri i neki SMP). Grubo granulirana paralelizacija je obrnuta - komunikacija se odvija povremeno, a izračunavanje izvodi nad većim količinama podataka (MMA).

5.1 Arhitektura distribuiranih sustava za paralelno izvođenje

Arhitektura računalnih sustava se tipično klasificira po Flynnovoj taksonomiji (slika 5.1) na četiri kategorije s obzirom na odnos broja instrukcija i količine podataka nad kojima se one izvode [16].

- **SISD, Single Instruction Single Data** predstavlja klasično, sekvencijalno, von Newmannovo računalo u kojem jedan procesor izvršava instrukcije nad jednim skupom podataka.



Slika 5.1: Flynnova taksonomija [49]

- **SIMD, Single Instruction Multiple Data** je model gdje računala izvode jednostruki slijed instrukcija nad više skupova podataka. Primjer za to su računala za procesiranje slike koja koriste višejezgrene procesore od kojih svaki izvršava iste instrukcije na grupama podataka.
- **MISD, Multiple Instruction Single Data** izvodi više istovremenih instrukcija na jednom skupu podataka. Ovaj model je najmanje zastupljen i uglavnom predstavlja konceptualni model [29].
- **MIMD, Multiple Instruction Multiple Data** je kategorija računala tj.sustava koji izvode višestruke instrukcije na grupama podataka. Takvi su višeprocesorski sustavi koji mogu imati zajedničku memoriju (engl. *shared memory*) ili distribuiranu memoriju (engl. *distributed memory*).

Distribuirano paralelno izvođenje programa podrazumijeva izvođenje dijelova programa na više procesora istovremeno. Sustavi na kojima je moguće paralelno izvođenje obuhvaćaju velik raspon različitih arhitektura: od računala s jednim višejezgrenim procesorom, preko multiprocesorskih računala i SMP (engl. *Symmetric Multiprocessor*) radnih stanica sa nekoliko višejezgrenih procesora do velikih masivnih paralelnih distribuiranih sustava sa do preko tisuću procesora.

Distribuirani sustavi sa distribuiranom memorijom su najčešći oblik distribuiranih sustava, jer se sastoje od relativno jeftinih računala koja su povezana brzom mrežom. Programiraju se koristeći *message passing* model u kojem korisnik particionira podatke u memoriju na različitim računalima i određuje koji procesor izvršava koje zadatke. Tipično je procesor kojem su pridruženi određeni podaci odgovoran za sva izračunavanja nad

tim podacima (*owner-computes* model). Ovakva arhitektura se naziva i *shared-nothing* arhitektura.

Dva su načina na koje sustavi sa distribuiranom memorijom funkcioniraju: sinkrono i asinkrono. U sinkronom modelu procesor izvršava instrukcije nad lokalnim podacima, te komunicira putem poruka sa ostalim računalima. Međutim komunikacija se odvija istovremeno za sve procesore, pa oni koji obave svoj posao moraju čekati na ostale. S druge strane u asinkronim sustavima komunikacija se može odvijati bilo kad. U oba slučaja komunikacija među procesorima je obostrana (engl. *two sided communication*) i nije moguće da procesor piše ili čita iz tuđe memorije bez pomoći programa koji se izvršava na drugom procesoru.

Distribuirani sustavi sa particioniranim globalnim adresnim prostorom također odvajaju podatke, ali omogućuju procesorima operacije nad podacima drugog računala. U oba slučaja, distribuirani sustavi imaju ograničen broj niti (engl. *thread*), što može biti prepreka kod implementacije paralelnih algoritama nad jako velikim grafovima.

Distribuirani sustavi sa dijeljenom memorijom (engl. *shared-memory*) hardverski omogućuju pristup globalnoj memoriji. Primjer takvog sustava su sustavi sa simetričnim multiprocesorima (engl. *symmetric multiprocessors, SMP*), gdje svaki procesor može pristupiti globalnoj memoriji i koji koristi OpenMP ili POSIX niti i neki superkompjuteri (Cray MTA-2) koji omogućuju dinamičko stvaranje niti čiji broj nije ograničen brojem procesora, već je omogućena virtualizacija niti (*massively multithreaded architecture, MMA*).

5.1.1 Sustavi za obradu velikih podataka Hadoop i Spark

Kako se u ovom radu daje pregled područja vezanih uz procesiranje jako velikih grafova, prirodno je krenuti od toga što nam pružaju postojeći okviri za obradu velikih podataka (engl. *big data frameworks*).

Dva najpoznatija i najkorištenija okvira za rad sa velikom količinom podataka na distribuiranim platformama su Apache Hadoop i Apache Spark. Osnova Apache Hadoop sustava sastoji se se od datotečnog sustava *Hadoop Distributed File System* i jedinice za procesiranje *MapReduce*.

MapReduce paradigma inspirirana je *high-order* funkcijama (funktorima) koje su centralni elementi funkcionalnog programiranja. Funktori su funkcije koje primaju jednu ili više funkcija kao argumente (transformeri) ili vraćaju funkciju kao rezultat. *High-order* funkcija `map` prima transformer funkciju i listu, te primjenjuje transformer funkciju na svaki element liste. To se može zapisati kao $map(f, [l_1, l_2, \ldots, l_n]) \mapsto [f(l_1), f(l_2), \ldots, f(l_n)]$. Funkcija `map` ne mijenja veličinu liste.

S druge strane, funkcija `reduce` (`accumulate`, `aggregate`) procesira listu koristeći `combiner` funkciju te za rezultat vraća jedan element. Funkcija `combiner` se primjenjuje na dva elementa i vraća rezutat na koji se `combiner` funkcija ponovo primjenjuje u kombinaciji s ostatkom liste. $reduce(f, x, [l_1, l_2, \ldots, l_n]) \mapsto f[\ldots f[f[x, l_1], l_2] \ldots, l_n]$

MapReduce [24] je model kojim se automatizira paralelizacija na distribuiranim sustavima. Funkcija `map` procesira skup ključ-vrijednost parova te generira skup ključ-vrijednost parova na koje je primjenjena neka funkcija za transformaciju. Nakon toga, `reduce` funkcija

uzima izlaz iz `map` funkcije i kombinira ključ-vrijednost parove u manji skup ključ-vrijednost parova.

Hadoop [59] je implementacija MapReduce modela otvorenog kôda koja omogućuje distribuirano procesiranje velikih skupova podataka na klasterima servera kojih može biti do više tisuća. Svaki server može obavljati računanje i pohranu dijela podataka. Za pohranu podataka Hadoop koristi Hadoop Distributed File System (HDFS). Svaka od faza `map` i `reduce` se izvodi paralelno na skupovima ključ-vrijednost, s tim da se između `map` i `reduce` dijela odvija prijenos podataka između klastera tako da se podaci particioniraju i sortiraju. Faza `reduce` počinje tek kada faza `map` završi i kada se podaci prebace na odgovarajuće mašine.

Prednost Hadoop okvira je u tome što serveri mogu biti obična računala (engl. *commodity machines*), a sam okvir replicira podatke tako da je sustav otporan na kvarove. Nadalje, na programeru je samo da osigura četiri funkcije - za unos, izlaz, map i reduce. Ulazni podaci trebaju biti skup parova ključ-vrijednost koji se distribuiraju po *workerima* (*mapperima*), procesiranje se odvija odvojeno za svaku skupinu parova, a rezultati se na kraju spajaju (engl. *merge*).

Takav pristup nije dobar za graf algoritme koji trebaju izmjenjivati poruke među vrhovima ili se pomicati sa jednog vrha na drugi. Osim toga, graf se šalje od *mappera* do *reducera* u svakoj iteraciji, čak i kada se ne mijenja. Na kraju, ispitivanje konvergencije kao kriterija zaustavljanje iteracije može zahtijevati još MapReduce poslova [90]. Zbog tih razloga MapReduce okviri se ne smatraju pogodnim za procesiranje velikih grafova.

Spark [111] je drugi najpopularniji alat za procesiranje distribuiranih podataka na klasterima (engl. *cluster computing framework*), ali ne obavlja posao pohrane podataka tj. nema svoj *file system* već radi sa HDFS ili nekim drugim datotečnim sustavom. Za razliku od Hadoopa koji u ciklusima obavlja čitanje podataka sa klastera, izvođenje operacija, pisanje rezultata na klaster, pa onda ponovno čitanje ažuriranih (*updateanih*) podataka, izvođenje operacija itd., Spark u jednom koraku čita podatke, procesira ih te zapisuje rezultate. Međurezultate drži u memoriji na klasterima, pa je zbog toga brži od Hadoopa.

Osnovni objekti u Sparku su RDD (Resilient Distributed Dataset) distribuirani objekti koji su particionirani po klasterima, a mogu se i keširati u memoriji čvorova klastera. Oporavak od pogreške je implementiran unutar njih, tj. oni se automatski rekreiraju u slučaju pogreške.

Spark aplikacije sastoje se od pokretačkog programa (engl. *driver program*) koji unutar `main` funkcije pokreće paralelne operacije na klasterima.

Hadoop i Spark su primjeri okvira za velike podatke (engl. *big data framework* ili *big data analytic engine*), ali ni jedan ni drugi nisu dobri za obradu podataka u formi grafa. Zato su se razvili distribuirani sustavi optimizirani za procesiranje grafova koji uvode *graph-parallel computation* gdje se paralelizam postiže particioniranjem grafa, a same funkcije izvode nad vrhovima (*vertex-centric*) mijenjajući njihova svojstva/vrijednosti. Promijenjena svojstva vrhova propagiraju prema susjednim vrhovima tijekom izmjenjivanja poruka paralelnog sustava.

5.2 Distribuirani sustavi za procesiranje grafova

Sustavi za procesiranje grafova (engl. *graph processing systems*) su specijalizirani sustavi čije je svojstvo da imaju API koji obuhvaća kompleksne zavisnosti (engl. *dependencies*) unutar grafa i koristi strukturu grafa za smanjenje komunikacije među kompjuterima. Napravljeni su za obradu velikih grafova i imaju dva osnovna cilja: omogućiti jednostavnije programiranje algoritama nad grafovima tako što kao osnovne entitete (engl. *first-class citizents*) stavljaju vrhove i bridove grafova te efikasno računanje na grafovima tako da skupove podataka (engl. *dataset*) drže u memoriji, a ne zapisuju na disk nakon svake iteracije.

Prvi cilj doveo je do koncepta *think like a vertex* koji je dominantam u sustavima za procesiranje grafova, iako se u zadnje vrijeme pojavljuje i koncept *think like a graph* [99]. *Think like a vertex* programski model (*vertex-centric*) jednostavan je za programiranje, ali po njemu particioniranje ne odredjuje korisnik, što onemogućuje optimizaciju za određene algoritme. S druge strane, *think like a graph* koncept dopušta particioniranje korisniku čime se, uz smisleno particioniranje, algoritmi mogu bitno ubrzati jer se manje vremena troši na izmjenjivanje poruka između računala.

5.2.1 Message Passing Interface (MPI)

MPI je standardni interface za *message passing* programe. Postoji nekoliko MPI biblioteka namijenjenih procesuiranju grafova, ali one ne pružaju podršku za sinkronizaciju, raspoređivanje, komunikaciju i otpornost na greške. Primjer takvih biblioteka su Parallel Boost Graph Library [44], Multithreaded Graph Library [65] i CombBlas [15].

5.2.2 Sustavi bazirani na BSP modelu

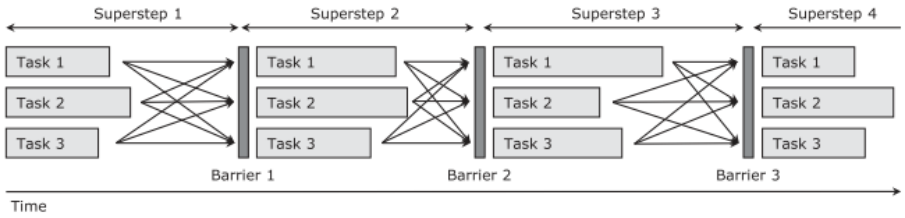
Pregel, 2010 [67] je platforma za procesiranje grafova koju je razvio Google. Bazira se na *Bulk Synchronous Parallel* (BSP) modelu koji se zasniva na slijedu superstepova [100]. BSP model se sastoji od

- parova procesor-memorija,
- mreže kojom se izmjenjuju poruke,
- hardvera koji podržava sinkronizaciju korištenjem barijere (engl. *barrier synchronization*).

Takva sinkronizacija omogućava da unutar jednog superstepa procesi koji dosegnu barijeru čekaju ostale, te se potom sinkroniziraju (slika 5.2). Sam superstep može biti računski (engl. *computational superstep*) ili komunikacijski (engl. *communication superstep*).

Pregel sustav promatra sve podatke kao usmjereni graf. Svaki vrh u grafu ima identifikator, korisnički definiranu vrijednost i izlazni brid, a svakom usmjerenom bridu je pridružen početni vrh, krajnji vrh i korisnički definirana vrijednost. Izlazni brid vrha i korisnički definirane vrijednosti mogu se mijenjati. Vrh ima dva stanja: aktivni i neaktivni. Ako vrh primi poruku, on postaje aktivan, a ako ne onda je neaktivan.

Samo izvođenje programa izvodi se u nekoliko faza. Više kopija programa pokreće se na klasterima. Jedno od računala postaje *master* i pridjeljuje svakom računalu (*worker*)



Slika 5.2: Bulk Synchronous Parallel model [69]

svoju particiju. Master zatim pokreće naredbe kojima se na ostalim računalima izvode superstepovi.

BSP model sastoji se od tri dijela. Prvi, *input step*, je distribuiranje vrhova po particijama. Broj particija je obično je veći od *workera*, pa na jednom *workeru* može biti više particija. Pregel particionira graf na temelju identifikatora vrha (predefinirana funkcija za particioniranje je hash funkcija). Particije se sastoje od skupa vrhova i njihovih izlaznih bridova.

Samo izvršavanje sastoji se od sekvenci superstepova. Superstepovi u Pregelu sastoje se od sekvenci iteracija u kojima se paralelno pozivaju instance korisnički definirane funkcije `compute()` za svaki vrh. Funkcija specificira operacije nad vrhom V i njegovim incidentnim bridovima, komunicira s globalnim (*aggregation*) varijablama i definira superstep S tako što prima poruke prethodnog superstepa S-1 i šalje poruke ostalim vrhovima koji ih primaju u sljedećem superstepu S+1. Nakon završetka superstepa vrhovi mogu postati neaktivni pozivanjem funkcije za zaustavljanje (`voteToHalt()`). Ponovo mogu postati aktivni ako prime poruku od nekog drugog vrha. Cijeli proces je gotov kada su svi vrhovi u neaktivnom stanju. Nakon izvođenja `compute()` metode obavlja se sinkronizacija prilikom koje se agregiraju vrijednosti globalnih varijabli. Da bi se smanjio broj poruka koji particije izmjenjuju, korisnik može definirati `combiner` funkciju u kojoj se agregiraju poruke (npr. min, max, sum). Na kraju se izvodi *output step*.

Pregel je napravljen za Google klaster arhitekturu [10] prema kojoj se svaki klaster sastoji od više od tisuću PC računala organiziranih u komunikacijske ormare (engl. *rack*), a sami klasteri nalaze se na različitim lokacijama. Taj sustav ima svoj servis za dodjelu imena računalima, pa Pregel koristi logička imena računala. S obzirom da je u distribuiranim sustavima uobičajen pad određenog postotka čvorova, Pregel omogućava provjeru (engl. *checkpointing*). Na početku superstepa *workeri* snime stanje svih vrhova, bridova i poruka. Kada se dogodi kvar, *master* notificira *workere* i oni se vrate u zadnje spremljeno stanje. Učestalost provjera određuje korisnik.

Pregel pruža C++ API koji skriva detalje implementacije distribuiranog sustava, a u prvi plan stavlja vrhove (*think like a vertex*), te je implementacija algoritama vrlo jednostavna. Međutim, Pregel je vlasnički (engl. *proprietary*) softver te nije dostupan za preuzimanje. Postoji implementacija u Javi koju su razvili studenti Sveučilišta u Santa Barbari, California i koje je otvorenog kôda (`http://kowshik.github.io/JPregel/#`). Uspješnija implementacija je Apachijeva implementacija otvorenog kôda, **Giraph, 2012**, koja se izvodi na Apache Hadoop klasterima i koja je dio Apache Hadoop ekosustava, te se samim tim dobro integrira sa drugim Hadoop tehnologijama. Međutim, umjesto

map i reduce poslova, Giraph ima samo map poslove (superstepove), što ukida potrebu za zapisivanjem međurezultata na disk između map i reduce faze, nema nepotrebnog sortiranja i obavlja sve poslove u memoriji.

Osim Giraph sustava, Apache razvija i Hama framework [93] koji implementira opći BSP model i koji ne služi samo računanju na grafovima, već uključuje i algoritme nad matricama, algoritme linearne algebre i slično.

Nakon Pregela i Girapha ravio se cijeli niz *frameworka* koji se logikom naslanjaju na Pregel. **DisNet, 2011** [62] je još jedan *master-worker* framework koji je, također usmjeren na vrhove (engl. *vertex-centric*) i sinkron, ali za razliku od Pregela cijeli graf se sprema u memoriju svakog *workera*. To ubrzava računanje, ali smanjuje skalabilnost. Osim toga, *workeri* mogu komunicirati samo sa *master* računalom, ali ne i međusobno.

Kod **GPS, 2013** [90] sustava ovorenog kôda API je proširen tako da omogućuje jednostavniju implementaciju globalnih, sekvencijalnih algoritama nad cijelim grafom, a ne samo usmjerene na vrhove, paralelne algoritme. Osim toga omogućuje reparticiju vrhova za vrijeme izračunavanja baziranu na porukama koji vrhovi izmjenjuju (tendencija da vrhovi koji često izmjenjuju poruke budu u istoj particiji), te sprema na istu particiju vrhove sa visokim stupnjem i njihove susjede čime štedi na porukama između dva *workera*.

GraphX, 2013 [109] [110] [41] je sustav za procesiranje grafova koji je dio Apache Spark projekta i koristi RDD, *fault-tolerance* i *task scheduling* iz Sparka koje kombinira sa API-jem za rad s grafovima. Ideja koja stoji iza GraphX-a je da se omogući korisniku da podatke vidi i kao graf i kao kolekciju RRD objekata. Tako omogućuje i *graph-parallel* i *data-parallel* računanje.

Graf je predstavljen kao graf sa svojstvima (engl. *property graph*), tj. usmjereni multigraf (graf u kojem između dva vrha može biti više bridova). Svaki vrh i brid mogu imati neku vrijednost. Bazira se na tablicama vrhova i tablicama veza među vrhovima.

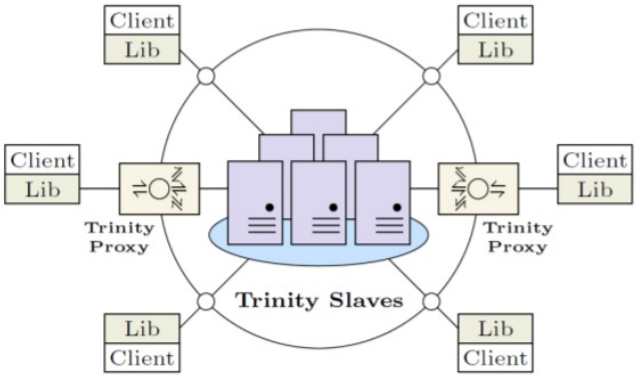
Trinity, 2013 [95], kasnije nazvan **Graph Engine**, kojeg je razvio Microsoft, primjer je distribuiranog sustava za *online* pretragu i *offline* analizu grafova sa particioniranim globalnim adresnim prostorom. Sustav se sastoji od više umreženih komponenti koje možemo klasificirati u tri skupine s obzirom na ulogu koji imaju u sustavu: klijent, *slave* i *proxy* (slika 5.3).

Slave čuva podatke te izvodi računanje nad njima što uključuje i komunikaciju porukama sa drugim komponentama sustava. *Proxy* ne sprema podatke već samo obrađuje poruke i služi kao posrednik između *slave* računala i klijent računala. Klijenti su aplikacije koje koriste Trinity biblioteku koristeći njen API.

Memorija je distribuirani *key-value store* koji se organizira kao 2^p memorijskih *trunkova*, gdje je za broj računala m, $2^p > m$. Podaci se zapisuju u ključ-vrijednosti parovima, gdje su ključevi 64-bitni ID-ovi, a vrijednosti su blobovi (Binary Large OBject) proizvoljne duljine. Parovi se dohvaćaju unutar globalnog adresabilnog adresnog prostora koristeći heširanje.

Trinity ima i dijeljeni distribuirani datotečni sustav TFS (Trinity File System) u kojem su spremljeni *trunkovi*, što povećava otpornost na pad dijelova sustava.

Za offline analizu grafova Trinity također koristi model usmjeren na vrhove (engl. *vertex-centric*), no za razliku od Pregela koristi restriktivniji model: u svakom superstepu vrhovi mogu primati poruke fiksnog broja vrhova (obično svojih susjeda) i mogu slati poruke fiksnom broju vrhova te modificirati njihove vrijednosti.



Slika 5.3: Trinity cluster [95]

5.2.3 Asinkroni sustavi

Prednost asinkronog pristupa je u tome što brzi *workeri* ne moraju čekati spore *workere*. S druge strane, sinkroni sustavi mogu grupirati poruke kako bi smanjili komunikaciju. Primjer asinkronog sustava je **GraphLab, 2012** [64] koji je također usmjeren na vrhove, ali za razliku od Pregela koji koristi poruke za interakciju među susjednim vrhovima, GraphLab koristi dijeljenu memoriju (engl. *shared state*).

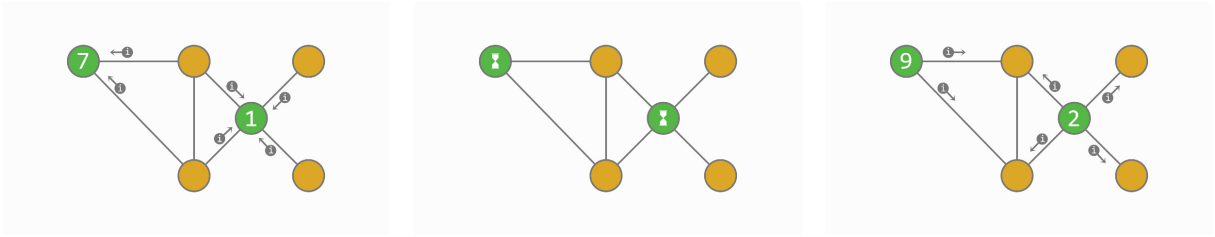
GraphLab se sastoji od tri dijela: data usmjereni graf $G = (V, E, D)$, gdje je D skup vrijednosti dodjeljenih vrhu ili bridu, *update* funkcije i *sync* operacije. Korisnički definirane *update* funkcije mogu mijenjati vrijednosti samog vrha, njegovih susjednih vrhova ili bridova.

GraphLab smanjuje samo vrijeme izračunavanja u odnosu na Pregel za 90%, ali i dalje troši puno vremena za kreiranje grafa i postprocesiranje.

PowerGraph, 2012 [40] je sličan GraphLabu, ali je prilagođen za rad sa nerazmjernim grafovima. Naime, GraphLab je namjenjen algoritmima u MLDM (Machine Learning Data Mining) području i u njemu se grafovi particioniraju u dva dijela: prvi na temelju specifičnog domenskog znanja (engl. *domain specific knowledge*) ili neke heuristike na k dijelova, gdje je k broj puno veći od broja računala, a particioniranje se izvodi brisanjem bridova (engl. *edge-cutting*). Svaki dio, koji se zove atom, spremljen je u odvojenoj datoteci na distribuiranom datotečnom sustavu. U svakom atomu su pohranjeni i *ghost* vrhovi, tj. skup vrhova koji su susjedni vrhovima i bridovima na granici particije. Metagraf od k-atoma se sprema u atom index datoteku. Nakon toga se particionira metagraf od k-vrhova na n particija, gdje je n broj računala.

Ovakav pristup particioniranju je problematičan za nerazmjerne mreže, jer su particije nebalansirane ako u samo od neke od njih stavimo vrhove sa velikim stupnjem, pripadajuće bridove i *ghost* vrhove. Zbog toga PowerGraph particije radi odstranjujući vrhove (engl. *vertex-cutting*). Na taj način svaki brid pripada jednoj particiji, a vrhovi sa velikim stupnjem nalaze se na više particija.

PowerGraph uvodi *gather-apply-scatter* (GAS) model po kojem se programi koji se izvode nad vrhovima dijele u tri faze. U *gather* fazi se prikupljaju podaci o susjednim vrhovima, u *apply* fazi program izvodi izračunavanje nad vrhom čiji su susjedi prikupljeni, dok se *scatter* osvježavaju (engl. *update*) vrijednosti bridova incidentnih tom vrhu (slika 5.4). I Pregel i GraphLab se mogu objasniti pomoću GAS modela. Kod Pregela je *gather* faza primanje poruka i izvođenja *combiner* funkcija, a *apply* i *scatter* su odvijaju unutar programa za izvođenje nad vrhovima (engl. *vertex* program), a kod GraphLaba *vertex* program obuhvaća *gather* i *apply* fazu, a *scatter* se postiže dijeljenim stanjima.



Slika 5.4: *Gather-apply-scatter* faze [33]

Osim sinkronih i asinkronih sustava, razvijen je i sustav koji omogućava dinamičko prebacivanje između sinkronog i asinkronog načina rada. **PowerSwitch, 2015** [108] ima također *vertex-centric* sučelje i izvodi programe nad vrhovma gdje u svakoj iteraciji mijenja stanje vrhova s obzirom na interakciju sa susjedima, ali niz iteracija grupira u epohe (engl. *epochs*) i kod prelaza iz jedne epohe u drugu odlučuje se za promjenu moda rada ovisno o brzini izvođenja u tekućem modu.

Odluka o tome treba li koristiti sinkrone ili asinkrone načine rada treba uzeti u obzir različite aspekte algoritma i podataka nad kojima se izvodi. Sinkroni sustavi imaju regularne intervale sinkroniziranja, ali sporo konvergiraju, dok je kod asinkronih situacija obrnuta. Nadalje, algoritmi koji se intenzivno oslanjaju na i/o operacije brže će se izvršavati na sinkronim, a CPU-intenzivni algoritmi na asinkronim sustavima. Skalabilnost je bolja kod sinkronih sustava jer se trošak koji se gubi čekanjem najsporijeg *workera* amortizira većom količinom aktivnih vrhova i izračunavanja nad njima u jednoj iteraciji. S druge strane, kod nekih algoritama je moguće da neće doći do konvergencije u sinkronom modu.

5.3 Paralelni sustavi za analizu velikih mreža na jednom računalu

Distribuirani sustavi za procesiranje velikih grafova na klasterima implementiraju upravljanje komunikacijom među klasterima i otpornost na greške (engl. *fault tolerance*). Međutim, particioniranje grafa na način da se minimizira komunikacija među klasterima, te da istovremeno particije budu balansirane je težak problem.

Distribuirani sustavi omogućavaju procesiranje jako velikih grafova u kratkom vremenu, ali njihovo korištenje nije jednostavno ni jeftino, pa se razvilo nekoliko sustava za paralelno procesiranje velikih grafova na jednom računalu koristeći višejezgrene procesore i GPU. U zadnjih nekoliko godina razvijeni su paralelni sustavi koji algoritme nad velikim grafovima izvode brzinom usporedivom sa brzinom izvođenja na distribuiranim sustavima istovremeno trošeći puno manje energije.

S obzirom da količina podataka može biti veća od raspoložive memorije na računalu, skalabilnost se postiže *out-of-core* izračunavanjem na način da se u memoriju učitavaju samo dijelovi grafa koji se dohvaćaju sa sporijih uređaja. Transfer podataka sa uređaja za pohranu trebao bi biti što manji i jednostavniji, a kod particioniranja podataka treba se voditi računa o lokalnosti, tj. omogućiti da se što više operacija izvodi na podacima unutar jedne particije.

5.3.1 Sustavi za izvođenje graf algoritama na višejezgrenim procesorima

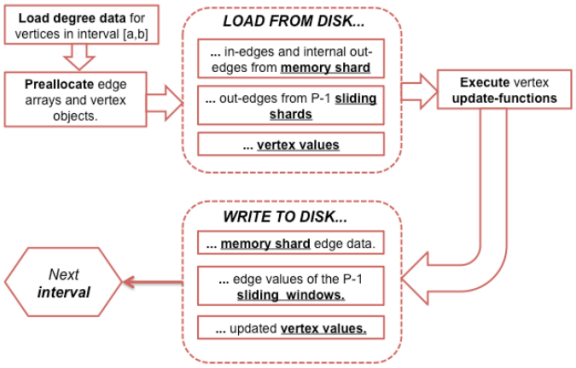
U zadnjih 10-tak godina razvili su se višejezgreni (engl. *multi-core*) procesori koji omogućuju istovremeno izvođenje na više jezgri procesora. Izvođenje graf algoritama na njima zahtjeva drugačiju implementaciju od klasičnih sekvencijalnih algoritama.

GraphChi, 2012 [58] [52] je sustav baziran na disku (engl. *disk-based*) otvorenog kôda za izvršavanje algoritama nad velikim grafovima na jednom računalu koji uvodi *parallel sliding windows (PSW)* metodu. Napisan je u C++, ali postoji i Java implementacija sa Scala API-jem koji je 2-3 puta sporiji. Koristi asinkroni model. Graf se sprema u CRS (engl. *Compressed Row Storage*) i CCS (engl. *Compressed Column Storage*) formatu opisanim u poglavlju 2.3.

Prednost korištenja *disk-based* izračunavanja je *random access* pristup podacima. PSW obrađuje jedan po jedan podgraf i procesira cijeli graf unutar jedne iteracije. Nakon toga uobičajeno slijedi sljedeća iteracija. Proslijeđivanje poruka vrhovima putem bridova se bazira na osvježavanju vrijednosti bridova.

Prema PSW metodi graf se dijeli na particije na način da se vrhovi podjele u P intervala koji su povezani sa *shardovima* u kojima se spremaju svi bridovi kojima je kraj (engl. *destination vertex*) unutar intervala. Bridovi u *shardu* su sortirani prema ulaznom vrhu. Sam broj particija P odabire se tako da se svaki pojedinačni *shard* može spremiti u memoriju.

U jednoj iteraciji prolazi se po intervalima, te se konstruira podgraf na način da se u njega dodaju svi ulazni bridovi pročitani iz *sharda* dok se izlazni bridovi čitaju iz *sliding shardova*. Nakon što je podgraf za interval učitan u memoriju, PSW pokreće korisnički definiranu *update* funkciju za svaki vrh paralelno (slika 5.5). Kako *update* funkcija može promijeniti vrijednosti bridova moglo bi se dogoditi da dva susjedna vrha pokušaju promijeniti isti brid što dovodi do stanja trke (engl. *race condition*). Da bi se to spriječilo susjedni vrhovi koji pripadaju istom intervalu se označavaju kao kritični i ažuriraju se sekvencijalno. To ograničava paralelizam [112].

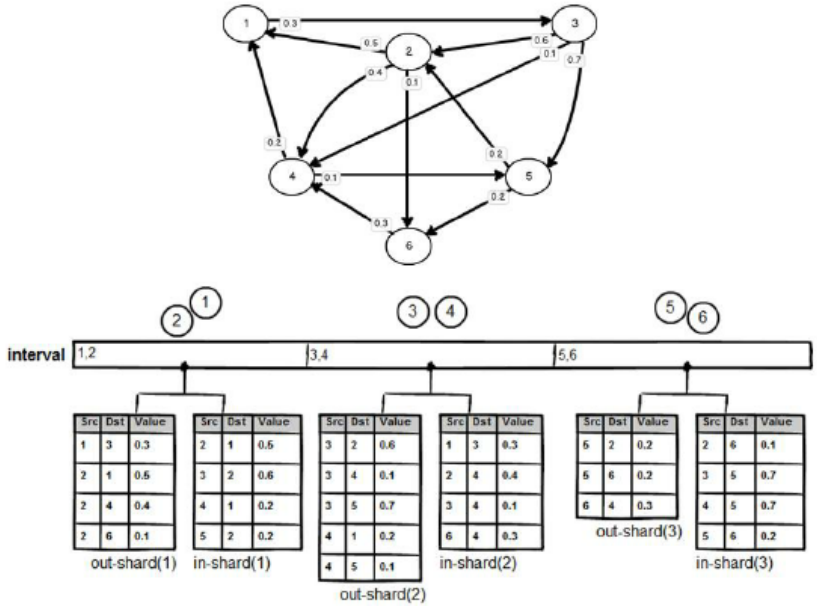


Slika 5.5: Dijagram GraphChi operacija [58]

Osim ograničenog paralelizma GraphChi ima još problem nepreklapanje ulazno izlaznih operacija i izvođenja zbog čega niti izvođenja (engl. *computation threads*) moraju čekati na i/o, što smanjuje performanse pogotovo kod velikih grafova koji u iteraciji obavljaju velik broj osvježavanja vrijednosti bridova koje onda treba zapisati na disk.

Zbog tih mana, predloženi su drugi modeli za paralelne graf sustave na jednom računalu, primjerice *pin-and-slide* implementiran u **TurboGraphu, 2013** [46] i **BiShard Parallel (BSP), 2014** [71]. Kod *pin-and-slide* modela izvođenje (engl. *computation*) se izvršava samo na jednom podskupu vrhova za koje se pronalaze stranice (engl. *page*) kojima pripadaju te se njih pridruži spremniku (engl. *pin to buffer*) dok se istovremeno dok se istovremeno može odvijati asinkroni i/o povratni poziv (engl. *callback*) kojim se ostatak vrhova čita sa SSD diska. Učitani vrhovi su pridruženi spremniku dok god ne završi procesiranje svih vrhova pripadajuće stranice. Kad završi procesiranje stranica se otkači (engl. *unpin*). Nakon toga prelazi se (engl. *slide*) na novu stranicu. Omogućivši paralelno izvršavanje asinkronih i/o operacija i niti izvođenja postigao se potpuni paralelizam.

Problem stanja trke koji se pojavio kod Parallel Sliding Windows modela BiShard model rješava tako da odvaja ulazne i izlazne bridove, tj. sprema dvije kopije svakog brida. To povećava količinu podataka zapisanih na disk, što se ne smatra kritičnim. Primjer spremanja ulaznih i izlaznih bridova u odvojene *shardove* može se vidjeti na slici 5.6.



Slika 5.6: BiShard primjer [71]

NXGraph, 2015 [21] uvodi podjelu grafa na intervale i *shardove* gdje su u intervalima vrhovi, a u *shradovima* bridovi i to tako da su bridovi u *shardu*, a njihovi pripadajući krajnji vrhovi (engl. *destination vertices*) u pripadajućem intervalu. Broj intervala i *shardova* je jednak. Nadalje, svaki *shard* dijeli na *sub-shardove* s obzirom na početne vrhove bridova. Na taj način premošćuje limit koji je imao GraphChi da svi početni i krajnji vrhovi brida moraju biti učitani u memoriju prije izračunavanja. Nakon učitavanja u memoriju sortiraju se bridovi u *sub-shardu* po krajnjim vrhovima. Intervali su učitani u memoriju cijelo vrijeme.

U [89] Roy i dr. uvode bridnocentričan (engl. *edge-centric*) pristup implementiran u **X-Stream, 2013** sustavu za procesiranje grafova. Sastoji se od dvije faze, *scatter* i *gather*. U *scatter* funkciji ulazni podatak je brid, a funkcija propagira vrijednost početnog vrha do krajnjeg vrha, te vraća vrijednost krajnjeg vrha. *Gather* funkcija akumulira poruke susjednih vrhova i mijenja stanje krajnjeg vrha.

Uobičajeni pristup skaliranju velikih grafova je sortiranje bridova po ulaznim vrhovima i *random acces* po sortiranim bridovima kako bi se došlo do njihovih incidentnih vrhova. Autori koriste činjenicu da je sekvencijalni pristup podacima brži od *random accesa* i na SSD diskovima i u memoriji, pa predlažu bridnocentričan pristup u kojem streamaju nesortiranu listu bridova sa uređaja za pohranu podataka na način da bridove smještaju u stream particije u kojima su sa bridovima i njihovi pripadajući početni vrhovi te njima pristupaju sekvencijalno unutar particije. Svaka stream particija sastoji se od skupa vrhova, liste bridova čiji je početni vrh u skupu vrhova i *update* liste koja se sastoji od osvježenih vrijednosti krajnjih vrhova koji se nalaze u skupu vrhova. Broj streaming particija je takav da particija stane u memoriju i one su fiksne veličine.

5.3.2 GPGPU (General purpose Graphics Processor Unit)

GPU je praktički dio svakog računala, bilo da se nalazi na posebnoj grafičkoj (video) kartici ili je ugrađen u matičnu ploču, pa može biti i na istom chipu kao i CPU [13]. Za razvoj GPU ubrzanog računanja (engl. *GPU-accelarated computinga*) možemo zahvaliti industriji igara gdje postoji ogromno tržište koje od igara očekuje sve bolju grafiku. S obzirom da se za grafiku koriste matrice i matrične operacije, vektori, meshevi i sl., prebacivanjem na GPU postiže se bitno ubrzanje, jer se takve operacije mogu izvoditi paralelno. Osim toga, za samo iscrtavanje grafike ista funkcija koja iscrtava piksel izvodi se paralelno na GPU. Međutim, kako kreiranje scena za igre i filmove nije samo iscrtavanje piksela, već i ozbiljno računanje raznih fizičkih pojava u svijetu koje igra predstavlja (gibanje tekućina, mijenjanje svojstava okoline zbog promjena temperature i sl.) tako je GPU *computing* postao zanimljiv i dijelu znanstvene zajednice koja se bavi računanjem visokih performansi (engl. *high-performance computingom*). Štoviše, zbog razvoja u tom smjeru, proizvođači su uveli i podršku za 64 bitne tipove veće preciznosti koji su nužni kod znanstvenih programa visokih performansi, a u samim igrama nisu bitni.

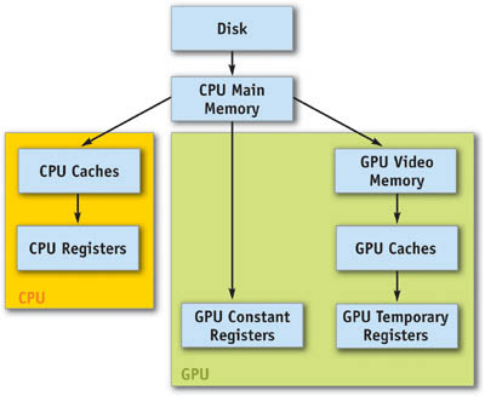
Danas se GPU sastoje od nekoliko stotina, pa i tisuća jezgri (NVidia Tesla K80 ima 2496 jezgri [80]) na kojima se može izvoditi više desetaka tisuća niti što GPU čini optimalnom za masovni paralelizam (u smislu cijene i energije). Osim toga hardver upravlja nitima što pojednostavljuje programe i olakšava skalabilnost i portabilnost. S obzirom na potencijale, u predhodnih desetak godina snažno se razvio i GPGPU (General-purpose computing on GPU), te se korištenje GPU pomaklo van znanstvenih interesa i industrije igara prema širokom polju problema. Ipak, GPU je napravljem za specifičnu klasu problema i nije jednostavan za programiranje, pa se primjenjuje kod aplikacija koje imaju velik broj računskih operacija, velik broj paralelnih niti i za koje je propusnost (engl. *throughput*) važnija od kašnjenja (engl. *latency*).

Nekoliko je programskih modela razvijeno za različite porodice grafičkih procesora, među kojima su najpoznatije CUDA (Compute Unified Device Architecture) za NVIDIA grafičke procesore [98] i CTM (Close To Metal) za ATI/AMD. To su sučelja za *general purpose computing* za paralelne arhitekture koje se sastoje od funkcija napisanih u programskom jeziku C.

GPU se sastoji od nekoliko multiprocesora (engl. *multiprocessors*, MP). Svaki multiprocesor se sastoji od jednakog broja jezgri koje izvode aritmetičko-logičke operacije. Unutar jednog multiprocesora sve jezgre izvode iste instrukcije, dok jezgre unutar različitih multiprocesora mogu izvoditi različite instrukcije. Svaki multiprocesor ima svoju malu dijeljenu memoriju, a zajedno dijele GPU globalnu memoriju. Model memorije prikazan je na slici 5.7.

Po Flynnovoj taksonomiji [30] prema kojoj se klasificiraju računalni sustavi s obzirom na podatke (jedan ili više) i instrukcije (jedna ili više) GPU multiprocesori spadaju u SIMD (Single Instruction, Multiple Data) model, često nazvan i vektorski procesor. Paralelno izvođenje znači izvršavanje više instanci istih niti.

Niti su podijeljene u blokove (engl. *blocks*) jednake veličine koji se dodjeljuju multiprocesorima, a same niti jezgrama unutar multiprocesora. GPU može upravljati većim brojem niti od broja jezgri, pa se vrijeme koje jedna nit potroši u komunikaciji s memorijom iskoristi za izvođenje druge niti (engl. *latency hiding*). Osim toga, nema ni zamjene konteksta koja kod CPU troši desetke tisuća taktova. Skup niti koji se istovremeno izvode na jednom multiprocesoru zove se *warp* i njegova veličina je fiksna, a programer određuje



Slika 5.7: Model memorije [107]

koliko će se niti izvršavati u istom *warpu*. Ako je broj niti veći od veličine *warpa*, niti se izvršavaju konkurentno.

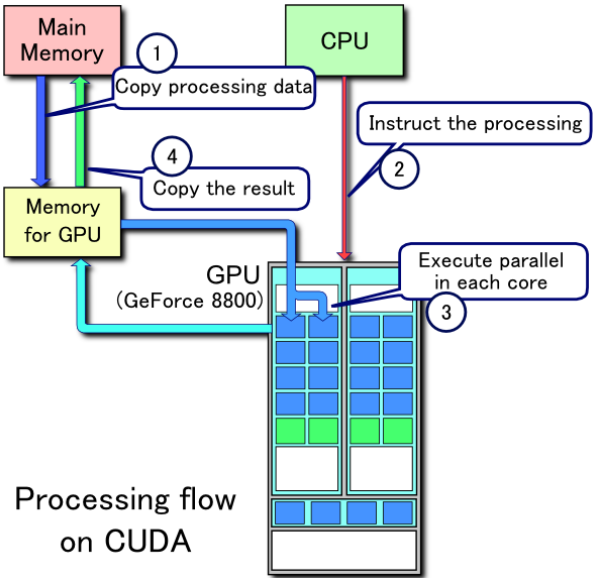
Izvođenje (engl. *single execution*) na GPU uređaju definira broj blokova, a kolekcija blokova koja se izvodi se zove grid. Svaka nit izvršava skup instrukcija koji se zove kernel.

Sam proces izvođenja programa prikazan je na slici 5.8. Odvija se tako što CPU prikupi ulazne podatke i prebaci ih u GPU memoriju. Zatim CPU poziva GPU program (ili skup kernela). GPU izvodi programe iz GPU memorije, a kad završi izvođenje CPU kopira rezultate nazad u CPU memoriju.

Implementacija paralelnih algoritama za grafove na ovakvim arhitekturama nije jednostavna, a kako se to područje počelo razvijati tek prije nekoliko godina ne postoji veliki broj biblioteka koje implementiraju takve algoritme. Zbog specifične arhitekture, paralelni algoritmi na GPU nemaju obrnuto proporcionalnu složenost sa brojem jezgri, kao što je to slučaj kod CPU višejezgrenih procesora. U [105] su navedeni problemi kod implementacije paralelnih algoritama za GPU. Kako je propusnost (engl. *bandwidth*) PCI Express sabirnice koja povezuje CPU i GPU ograničen (trenutno oko 16GB), poželjno je izvoditi više operacija nad jednim podatkom, što kod graf algoritama nije slučaj.

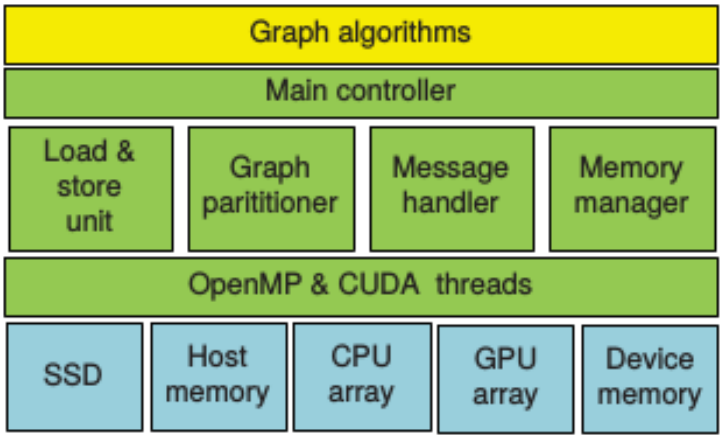
Platforma **gGraph, 2015** [112] za procesiranje grafova uvodi potpuni paralelizam: flash ssd i/o, višejezgreni CPU i višejezgreni GPU paralelizam.

Kod gGraph modela graf je usmjereni $G = (V, E)$, u kojem svaki vrh i svaki brid ima vrijednost. Programer implementira *update* funkciju koja se izvršava na svakom vrhu iterativno dok se ne zadovolji uvjet konvergencije. Pritom gGraph platforma koristi BSP model, prema kojem se nakon svake iteracije vrši sinkronizacija. Slično Pregelu, procesiranje se obavlja u superstepovima, gdje se svaki superstep sastoji od tri faze: izračunavanja (engl. *computation*), komunikacije (engl. *communication*) i sinkronizacije. U *computation* fazi CPU ili GPU izvršavaju instrukcije asinkrono dohvaćajući vrijednosti svojih (lokalnih) memorijskih lokacija. U fazi komukacije procesori izmjenjuju poruke potrebne za osvježavanje njihovog stanja prije sljedećeg superstepa. S obzirom da je graf podijeljen na particije u kojima se može dogoditi da su u različitim particijama susjedni vrhovi (engl. *boundary edges*) šalje se poruke susjednim vrhovima udaljenih particija. U fazi sinkronizacije poruka poslana u superstepu i posstaje dostupna u lokalnoj memoriji ciljnog procesora u superstepu $i + 1$.



Slika 5.8: Proces izvođenja na CUDA arhitekturi [47]

gGraph platforma sastoji se od 4 dijela (slika 5.9): jedinica za čitanje i pisanje, jedinica za particioniranje, upravljač porukama i upravljač memorijom. Sam graf pohranjen je na

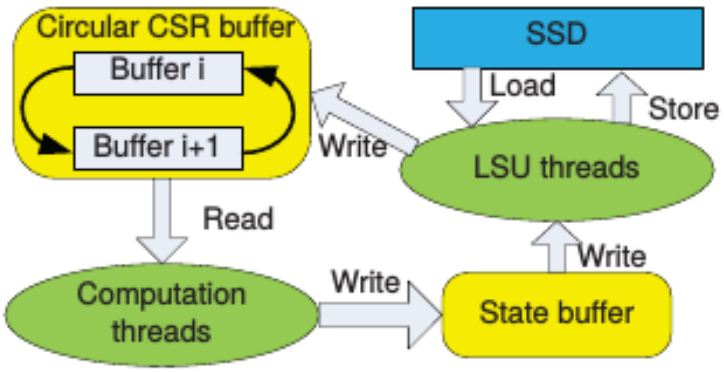


Slika 5.9: Arhitektura gGraph platforme [112]

SSD disku tako da su indeksi redaka pohranjeni u datoteci, a indeksi stupaca raspoređeni u P particija koje su podijeljene na blokove. Veličina particije je takva da može cijela stati u memoriju, a veličina bloka i veličina particije su konfigurabilne. Također postoji i datoteka s tabelom particija u kojoj su za svaki vrh zapisane vrijednosti parova (*partition_id, block_id*).

Učitavanje i pisanje na SSD disk izvodi *Load & Store Unit (LSU)* (slika 5.10). LSU *thread* dohvaća particiju tj. podgraf koji će sudjelovati u sljedećem superstepu, smješta ga u cirkularni CSR buffer koji se sastoji od dva dijela: jedan koji sadrži podgraf trenutnog superstepa i jedan u koji se sprema podgraf sljedećeg superstepa. Na taj način dok se

izvršava superstep paralelno se izvodi učitavanje podataka za sljedeći superstep, čime se znatno ubrzava izvođenje, jer su i/o operacije uglavnom najsporiji dio. Jedinica za



Slika 5.10: Load & Store Unit [112]

particioniranje grafa (engl. *graph partitioner*) dijeli dalje podgraf na $N + 1$ dijelova, gdje je N broj GPU jedinica, a jedan dio je zajednički za sve CPU jedinice, jer one dijele memoriju, dok GPU jedinice imaju svaka svoju memoriju.

Upravljač porukama (engl.*message handler*) također upravlja dvama bufferima za svaku GPU procesorsku jedinicu: ulazni spremnik (engl. *inbox buffer*) i izlazni spremnik (engl. *outbox buffer*) i jedan par ulazno/izlaznih spremnika za CPU. U izlaznom spremniku su granični vrhovi i koji su na udaljenim particijama i njihove pripadajuće poruke, a u ulaznom spremniku su vrhovi koji su udaljeni za neku drugu particiju i njihove poruke. Za svaku procesorsku jedinicu poruke se agregiraju kako bi se smanjila komunikacija.

Upravljač memorijom alocira, dealocira i ponovo koristi (engl. *recycle*) memoriju za cirkularni CSR *buffer* u kojem se sprema CSR podgraf, spremnik stanja (engl. *state buffer*) u kojem su vrijednosti vrhova i/ili bridova, *loading window* vrhovi i pripadajući bridovi za sljedeći superstep i za ostale manje spremnike. Osim toga, upravljač memorijom upravlja i transferima između glavne memorije (engl. *host memory*) i GPU memorije (engl. *device memory*).

U [112] autori su usporedili izvršavanje implementacija tri algoritma (komponente povezanosti grafa (engl. *connected components, CONN*), najkraći put iz jednog izvora (engl. *Single source shortest path, SSSP*)) i Page rank (PR), te utvrdili da gGraph nadmašuje GraphChi i PowerGraph, ali ne i GPS platformu. K tome je i energetska efikasnost najbolja među testiranim platformama.

Medusa, 2014 [114] je *framework* nastao s idejom da omogući programerima korištenje prednosti GPU pišući sekvencijalan kôd u C/C++. To je ustvari skup korisnički definiranih API funkcija koje *runtime* sistem izvršava paralelno. Inspiriran je BSP modelom, ali kao programski model ne koristi model usmjeren na vrhove (engl. *vertex-centric*) već Edge-Message-Vertex (EMV) model u kojem se izračunavanja mogu izvoditi i na vrhovima i bridovima, a jedinica procesiranja mogu biti i poruke. To je omogućeno kroz šest API-ja: Message API, Vertex API, Elist API, Edge API, Mlist API i Combiner API. Vertex i Edge API mogu slati poruke susjednim vrhovima. Razlog za to je postizanje *fine-grained* paralelizma koji je nužan za efikasno izvođenje na GPU. Naime, model usmjeren na vrhove

je *course-grained* paralelizam pri čemu vrhovi imaju različite stupnjeve i različiti broj poruka, što otežava optimizaciju memorije za bridove i poruke.

API funkcije se izvode nad svim vrhovima i bridovima grafa, što može dovesti do sporijeg izvođenja nekih algoritama. Zato je korisniku omogućeno postavljanje aktivnih vrhova ili bridova putem funkcije `SetActive()`.

6 Particioniranje za paralelnu obradu na grafičkim procesorima

Kod implementacije graf algoritama za paralelnu obradu susrećemo se sa različitim izazovima. Grafovi su sve veći, njihova struktura i zavisnosti nisu unaprijed određeni, pa odluka o tome na koji način će se graf učitati u memoriju, a onda rasporediti na paralelnu arhitekturu utječe na sam ishod i brzinu algoritama. Mnogi problemi koji se modeliraju pomoću grafa spadaju u diskretne optimizacijske NP-probleme koji nemaju jedinstveno rješenje već se heurističkim metodama približavaju optimalnom rješenju. S druge strane, neki problemi su *embarrassingly parallel*. Sustavi za procesiranje grafova implementiraju širok spektar graf algoritama za koje ne postoji jedinstveni optimalan način za učitavanje i particioniranje grafa koji bi bio zajednički za sve algoritme. Osim toga, grafovi imaju iregularnu strukturu što stavlja dodatne zahtjeve prilikom implementacije graf algoritama.

Nekoliko je osnovnih stvari o kojima treba razmišljati prilikom implementacije algoritama. GPU i CPU imaju različitu arhitekturu i to treba uzeti u obzir. CPU ima mali broj registara za svaku jezgru, što povlači za sobom velik broj zamjena konteksta (engl. *context switch*), dok je broj registara po GPU jezgri velik. S obzirom na to, CPU implementacija je bolja za mali broj kompleksnih poslova, dok je GPU bolji za izvođenje velikog broja jednostavnih poslova.

6.1 Streaming modeli

Kada se radi o velikoj količini podataka koje ne stanu u memoriju računala koriste se modeli za streamanje podataka (engl. *data streaming models*). Oni omogućavaju procesiranje ulaznih podataka onako kako se učitavaju koristeći ograničenu količinu memorije.

Za razliku od standardnog čitanja podataka možemo pretpostaviti da sustav nema kontrolu nad poretkom elemenata koji će se učitavati, da unaprijed ne moramo nužno znati veličinu podataka i da se do jednom obrađenih podataka ne može doći ukoliko nisu spremljeni u memoriju.

Za graf $G = (V, E)$ graf stream je niz bridova $e_{i_1}, e_{i_2}, ..., e_{i_m}$, gdje je $e_{i_j} \in E$, a $i_1, i_2, ..., i_m$ permutacija skupa $1, 2, ..., m$ [28]. Cilj streaming algoritama je smanjiti prostor ispod linearnog za veličinu ulaznih podataka ili vrijeme potrebno za izvođenje nad podacima.

Razlikujemo jednoprolazne (engl. *one-pass*) i višeprolazne (engl. *multi-pass*) *stream* modele. Kod jednoprolaznih modela ulaznim podacima se pristupa sekvencijalno samo jedan put, a količina memorije koja im se dodjeljuje je konstantna ili sublinearna u odnosu na veličinu problema (omeđena s gornje strane linearnom funkcijom od n). Algoritmi se izvode nad skupovima podataka koji su veći od raspoložive memorije, a najčešće ne daju točne rezultate već ih aproksimiraju. U višeprolaznom modelu podacima se može sekvencijalno pristupati više puta. Samim tim algoritmi dolaze do točnih rezultata, ali broj iteracija se povećava kako se povećavaju ulazni podaci, a samim tim i vrijeme izvršavanja algoritma.

Nekoliko je modela koji se koriste za streamanje grafova. U klasičnom stream modelu parametri su broj sekvencijalnih prolaza kroz podatke p i veličina memorije s. Osim toga bitno je i *per-item* vrijeme procesiranja t koje treba biti što manje. Cilj je pronaći algoritme čija je prostorna složenost funkcija broja vrhova ili da ona raste proporcionalno s brojem bridova. Algoritmi za rad s grafovima nisu pogodni za klasičan model [88], pa su

se razvili drugi modeli prilagođeni grafovima koji su proširenje klasičnog stream modela. To su semi-streaming, w-stream i sort-stream.

Semi-stream model dozvoljava veći s, pa je ograničenje na s je $O(n\dot{l}og^k(n))$ bitova (k je konstanta), a broj prolazaka kroz skup podataka je konstantan ili je logaritamska funkcija broja vrhova. U memoriju se spremaju svi vrhovi, ali ne nužno svi bridovi grafa [81]. U **w-stream** modelu dozvoljeno je i pisanje i čitanje streamova. U svakom prolazu zadnji zapisani stream, ili, u slučaju prvog prolaska, ulazni stream se čita, te se zapisuje novi stream. Veličina s za svaki međustream (engl. *intermediate*) je je linearna funkcija veličine originalnog streama s. **Sort-stream** je proširenje ovog modela sa mogućnošću sortiranja međustreamova u jednom prolasku.

6.2 Particioniranje za paralelnu obradu

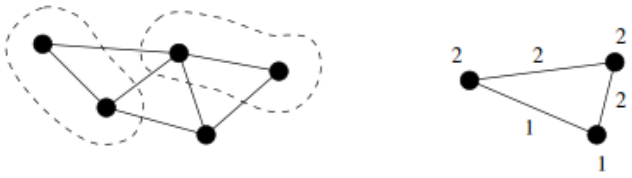
Particioniranje grafa je bitan predprocesorski korak za paralelne algoritme u kojem se graf dijeli između više procesorskih ili GPU jedinica. Kvaliteta particije utječe na efikasnost algoritma, jer smanjuje količinu komunikacije koja je u paralelnim sustavima skupa. Kvaliteta particioniranja određena je balansiranošću particija i minimalnim troškom koji je određen težinom bridnih rezova (engl. *edge cut*).

Nekoliko je problema vezanih uz particioniranje grafa na CPU-GPU arhitekturi naglašeno u [42]:

- GPU niti komuniciraju putem glavne memorije. Zato se zapisivanje u glavnu memoriju treba sinkronizirati, a s obzirom da konkurentnih niti može biti nekoliko tisuća, te da korišenje atomarnih operacija i *lockova* smanjuje performanse, to predstavlja izazov.
- Iregularnost grafa za posljedicu ima smanjenje performansi algoritama za particioniranje na GPU.
- Ograničenje memorije i latentnost transfera između CPU i GPU.
- Heterogeni sustavi kombiniraju SIMD i MIMD model, pa se program za particioniranje treba dekomponirati tako da se iskoriste prednosti i jednog i drugog modela.

Većina biblioteka za particioniranje koriste višerazinsku paradigmu (engl. *multilevel paradigm*) po kojoj se graf sažima (engl. *coarsening*), particionira i, na kraju opet ekstrahira (engl. *uncoarsening*) što je metoda opisana u poglavlju 3.1.7.

Najpoznatije biblioteke koje se koriste za sekvencionalno particioniranje grafa su Metis [53], Scotch [84] i Jostle [103] u kojima se faza sažimanja grafa (slika 6.1) sastoji od dva dijela: uparivanje (engl. *matching*) i stezanje (engl. *contraction*). Uparivanje M je skup nesusjednih bridova, a maksimalno uparivanje je takvo da ne postoji brid koji bi se mogao dodati u skup M, a da on i dalje ostane upareni skup. Za pronalaženje skupa M Metis koristi SHEM metodu (engl. *Sorted heavy edge matching*) po kojoj se prolazi kroz niz vrhova sortiranih uzlazno po stupnju i, za nespareni vrh v, u skup M stavlja incidentni brid najveće težine, a iz niza se izbacuju vrhovi incidentni s v. U fazi stezanja parovi vrhova incidentni s bridovima iz M se spajaju u jedan vrh. Postupak se ponavlja dok se ne zadovolje specifično definirani kriteriji. Kada je sažeti graf dovoljno malen, particionira se GGGP metodom (engl. *Greedy Graph Growing Partitioning*) po kojoj se počinje od



Slika 6.1: Faza sažimanja grafa [103]

slučajno odabranog vrha i koristeći BFS dodaju vrhovi u tzv. regije. Među odabranim kandidatima u regiju će se dodati vrhovi čijim će uključivanjem u regiju doći do najvećeg smanjenja težine bridnog reza. Regija raste dok u njoj nije oko polovice vrhova, te se tako graf podijeli na dva dijela. Postupak se rekurzivno ponavlja sve dok broj traženi particija nije postignut.

Faza ekstrakcije obuhavaća projekciju (engl. *projection*) u kojoj se sažeti vrhovi unutar particija vraćaju u originalne vrhove, a nakon svake projekcije se izvodi i rafiniranje (engl. *refinment*) u kojem se poboljšava bridni rez, tj. prolazi se kroz rubne vrhove te se oni premještaju iz jedne particije u drugu kako bi se dobila manja težina bridnog reza.

Tri spomenute biblioteke za sekvencijalno particioniranje grafa imaju svoje verzije za particioniranje na distribuiranim sustavima: ParMetis [55], PT-Scotch [20] i Parallel Jostle [103]. ParMetis implementira krupnozrnato (engl. *coarse-grained*) particioniranje u kojem inicijalno svaki procesor dobija $\frac{n}{p}$ vrhova, gdje je n broj vrhova, a p broj procesora. Faza uparivanja sastoji se od dva dijela. U parnim prolazima vrh v na procesoru p šalje zahtjev za uparivanjem vrhu u koristeći HEM (engl. *Heavy edge matching*) ako je $v > u$, a u neparnim ako je $v < u$. Nakon svake iteracije obavlja se sinkronizacija.

U fazi inicijalnog particioniranja svaki procesor particionira graf na dva dijela, a faza ekstrakcije se odvija slično kao u sekvencijalnom algoritmu, s tim da se na kraju svake iteracije šalju zahtjevi za prebacivanjem vrhova tako da prebacivanje ne utječe na balansiranost particija.

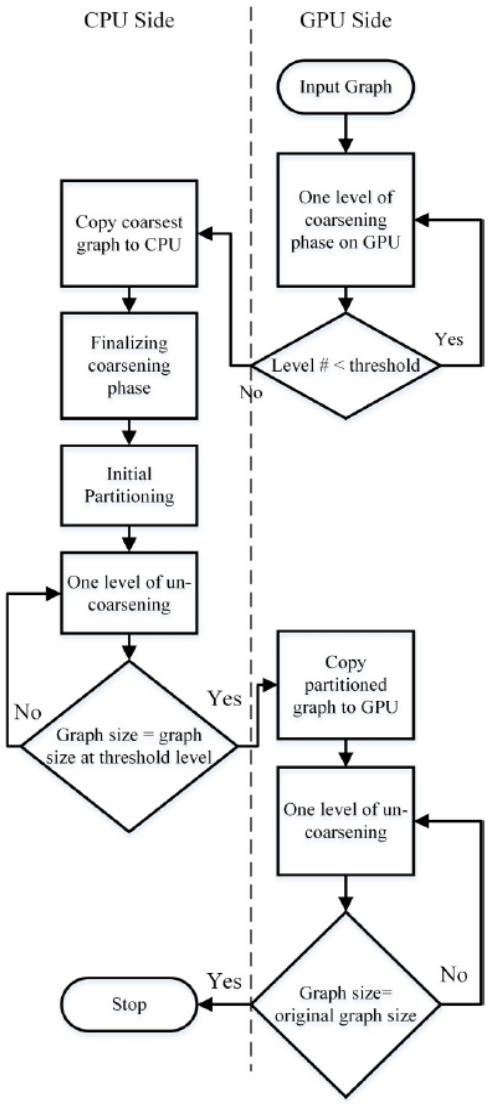
PT-Scotch u fazi uparivanja također šalje zahtjeve bazirane na HEM metodi, ali sa vjerojatnošću 0.5 da će nespareni vrh poslati zahtjev. Parallel Jostle uglavnom se koristi za particioniranje *mesheva* tj. planarnih grafova veće gustoće.

Za particioniranje na multijezgrenim arhitekturama sa dijeljenom memorijom koriste se biblioteke koje su proširenje Metisa, Gmetis i Mt-metis [97].

Prva biblioteka za višerazinsko particioniranje na heterogenenim CPU-GPU sustavima je CUDA implementacija GP-metis [42] najavljena 2016. godine. To je također implementacija višerazinskog algoritma koji se sastoji od sažimanja, inicijalnog particioniranja i ekstrakcije, s tim da sažimanje i ekstrakciju obavlja GPU, a inicijalno particioniranje CPU. Na slici 6.2 je prikazana schema particioniranja.

U svakoj iteraciji sažimanja alociranja su dva niza u globalnoj GPU memoriji, niz M duljine n, gdje je n broj vrhova grafa u kojem se nalaze parovi vrhova koji su upareni i koji će se sažeti u toj iteraciji, te niz mapiranih vrijednosti duljine n u kojem su labele vrhova koji će se sažeti u sljedećoj iteraciji grafa.

Na početku faze uparivanja vrhovi se podijele te se svakoj niti pridjeli skup vrhova.

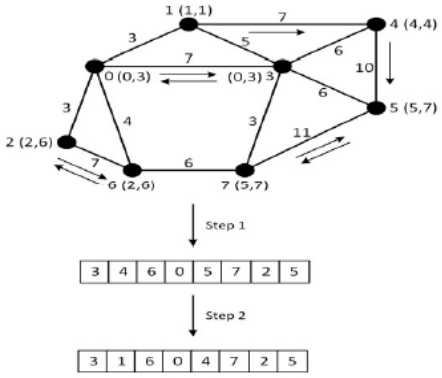


Slika 6.2: Schema particioniranja [42]

Svaka nit prolazi po svom skupu vrhova i za svaki vrh pronalazi odgovarajući par koristeći HEM metodu. Ukoliko su svi bridovi iste težine, koristi Random matching. Kako se niz M zapisuje u globalnu GPU memoriju može doći do konflikta, pa se pokreće još jedna kernel funkcija koja provjerava da li postoji vrh v sparen sa u takav da vrh u nije sparen sa v. Ukoliko postoji vrh v se uparuje sa samim sobom, te će se njemu pronaći par u nekoj od sljedećih iteracija (slika 6.3).

U fazi inicijalnog particioniranja na CPU koristi se Mt-metis koji iskorištava višejezgrenost CPU. Ekstrakcija se također odvija u dvije faze, projekcija i rafiniranje. Problem premještanja vrhova u fazi rafiniranja koji opet može dovesti do povećanja vrijednosti reznog brida, ali i do konflikta (*race condition*), rješava se tako da se vrhovi mogu premještati između dvije particije samo u jednom smjeru (uzlazno ili silazno).

Osim višerazinskog particioniranja postoje i druge, jednostavnije strategije, primjerice particioniranje po stupnjevima vrhova gdje se particije u kojima su vrhovi većeg stupnja



Slika 6.3: Uparivanje [42]

obrađuju na GPU, a particije u kojem su vrhovi manjih stupnjeva na CPU koje koristi Totem [35]. Takvi sustavi eksploatiraju činjenicu da je glavna memorija puno veća od GPU memorije, pa CPU može procesirati puno veće grafove, ali GPU može izvršavati paralelno puno više niti.

7 Zaključak

U posljednjih nekoliko godina došlo je do velikog razvoja paralelnih sustava za procesiranje velike količine podataka. Tehnologije velikih podataka (engl. *Big Data*) svoju primjenu nalaze i u mnogim poslovnim sustavima. Predvodnici razvoja sustava paralelne i distribuirane obrade su tehnologije Apache organizacije: Hadoop i Spark. Iako široko prihvaćeni i korišteni za analizu i rudarenje podataka, pokazali su se neprikladni za obradu velike količine podataka dane u formi grafa.

Grafovi su iregularne strukture podataka koji imaju određene specifičnosti. U stvarnom svijetu kompleksni grafovi se pojavljuju kao nerazmjerne mreže čiji stupnjevi prate distribuciju zakona potencije, te kao mreže malog svijeta tj. kao mreže čiji je dijametar mali. Ove specifičnosti, kao i sama iregularnost grafova omogućavaju optimizacije namjenjene isključivo takvoj vrsti podataka kao što su praćenje aktivnih vrhova, *vertex-cut* particioniranje i dinamičko reparticioniranje.

Različiti odabir i vrste optimizacija doveli su do razvoja velikog broja različitih programskih okvira–*frameworka* čija su svojstva i osnovni koncepti opisani u ovom radu.

U radu su opisani koncepti na koje se oslanjaju sustavi za distribuiranu i paralelnu obradu velikih podataka pohranjenih u formi grafa. Distribuirana obrada provodi se na čvorovima koji su obična računala. Osim distribuirane obrade, u ovom radu dan je i pregled sustava za procesiranje grafova na jednom računalu koristeći višejezgrene procesore i/ili višeprocesorske grafičke kartice. Zbog bitno manjeg troška energije, a i samog hardvera, takvi sustavi su možda najzanimljiviji za razvoj, ali možda i najteži. Razvoj GPGPU programiranja se bazira na razvoju grafičkih aplikacija koje rade sa pravilnim matričnim strukturama podataka za razliku od grafova koji su nepravilni.

Prilagodba takvih programskih modela grafovima u ovom je trenutku izazov koji je u fokusu rada mnogih znanstvenika te se u tom smjeru očekuje i znanstveni doprinos buduće doktorske disertacije.

8 Literatura

[1] Property graph model. https://github.com/tinkerpop/blueprints/wiki/Property-Graph-Model. Accessed: 2016-09-30.

[2] Aasman, J. Allegro graph: Rdf triple database. *Cidade: Oakland Franz Incorporated* (2006).

[3] Allesina, S., and Pascual, M. Googling food webs: can an eigenvector measure species' importance for coextinctions. *PLoS Comput. Biol 5*, 9 (2009), e1000494.

[4] Angles, R. A comparison of current graph database models. In *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on* (2012), IEEE, pp. 171–177.

[5] Angles, R., and Gutierrez, C. Survey of graph database models. *ACM Computing Surveys (CSUR) 40*, 1 (2008), 1.

[6] Bang-Jensen, J., and Gutin, G. Z. *Digraphs: theory, algorithms and applications*. Springer Science & Business Media, 2008.

[7] Barabási, A.-L., and Albert, R. Emergence of scaling in random networks. *science 286*, 5439 (1999), 509–512.

[8] Barabási, A.-L., Ravasz, E., and Oltvai, Z. Hierarchical organization of modularity in complex networks. In *Statistical mechanics of complex networks*. Springer, 2003, pp. 46–65.

[9] Barabási, A.-L., and Oltvai, Z. N. Network biology: understanding the cell's functional organization. *Nature Reviews Genetics 5*, 2 (2004), 101–113.

[10] Barroso, L. A., Dean, J., and Hölzle, U. Web search for a planet: The google cluster architecture. *Micro, Ieee 23*, 2 (2003), 22–28.

[11] Boccaletti, S., Latora, V., Moreno, Y., Chavez, M., and Hwang, D. Complex networks: Structure and dynamics. *Physics Reports 424*, 4-5 (2006), 175–308.

[12] Bollobás, B. *Modern graph theory*, vol. 184. Springer Science & Business Media, 2013.

[13] Brodtkorb, A. R., Hagen, T. R., Schulz, C., and Hasle, G. Gpu computing in discrete optimization. part i: Introduction to the gpu. *EURO Journal on Transportation and Logistics 2*, 1-2 (2013), 129–157.

[14] Buluç, A. *Linear algebraic primitives for parallel computing on large graphs*. PhD thesis, UNIVERSITY OF CALIFORNIA Santa Barbara, 2010.

[15] Buluç, A., and Gilbert, J. R. The combinatorial blas: Design, implementation, and applications. *International Journal of High Performance Computing Applications* (2011), 1094342011403516.

[16] Chalmers, A., Reinhard, E., and Davis, T. *Practical parallel rendering*. CRC Press, 2002.

[17] Chapuis, G., Djidjev, H., Andonov, R., Thulasidasan, S., and Lavenier, D. Efficient multi-gpu algorithm for all-pairs shortest paths. In *IPDPS 2014* (2014).

[18] Chazelle, B. A minimum spanning tree algorithm with inverse-ackermann type complexity. *Journal of the ACM (JACM) 47*, 6 (2000), 1028–1047.

[19] Chen, C.-Y., Ho, A., Huang, H.-Y., Juan, H.-F., and Huang, H.-C. Dissecting the human protein-protein interaction network via phylogenetic decomposition. *Scientific reports 4* (2014).

[20] Chevalier, C., and Pellegrini, F. Pt-scotch: A tool for efficient parallel graph ordering. *Parallel computing 34*, 6 (2008), 318–331.

[21] Chi, Y., Dai, G., Wang, Y., Sun, G., Li, G., and Yang, H. Nxgraph: An efficient graph processing system on a single machine. *arXiv preprint arXiv:1510.06916* (2015).

[22] Ching, A., Edunov, S., Kabiljo, M., Logothetis, D., and Muthukrishnan, S. One trillion edges: graph processing at facebook-scale. *Proceedings of the VLDB Endowment 8*, 12 (2015), 1804–1815.

[23] Clauset, A., Newman, M. E. J. N., and Moore, C. Finding community structure in very large networks. *Physical Review E 70*, 6 (2004), 1–6.

[24] Dean, J., and Ghemawat, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113.

[25] Dorogovtsev, S. N. *Lectures on complex networks*, vol. 24. Oxford University Press Oxford, 2010.

[26] Duff, I. S., Grimes, R. G., and Lewis, J. G. Sparse matrix test problems. *ACM Transactions on Mathematical Software (TOMS) 15*, 1 (1989), 1–14.

[27] Erd6s, P., and Rényi, A. On the evolution of random graphs. *Publ. Math. Inst. Hungar. Acad. Sci 5* (1960), 17–61.

[28] Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., and Zhang, J. On graph problems in a semi-streaming model. In *Automata, Languages and Programming*. Springer, 2004, pp. 531–543.

[29] Flynn, M. *Computer architecture*. Wiley Online Library, 2008.

[30] Flynn, M. J. Very high-speed computing systems. *Proceedings of the IEEE 54*, 12 (1966), 1901–1909.

[31] Fredman, M. L., and Tarjan, R. E. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM) 34*, 3 (1987), 596–615.

[32] Frieze, A., and Karoński, M. *Introduction to random graphs*. Cambridge University Press, 2015.

[33] Fu, Z., Personick, M., and Thompson, B. Mapgraph: A high level api for fast development of high performance graph analytics on gpus. In *Proceedings of Workshop on GRAph Data management Experiences and Systems* (2014), ACM, pp. 1–6.

[34] Gardiner, E. J. Graph applications in chemoinformatics and structural bioinformatics., 2011.

[35] Gharaibeh, A., Costa, L. B., Santos-Neto, E., and Ripeanu, M. On graphs, gpus, and blind dating: A workload to processor matchmaking quest. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on* (2013), IEEE, pp. 851–862.

[36] Gilbert, E. N. Random graphs. *The Annals of Mathematical Statistics 30*, 4 (1959), 1141–1144.

[37] Girvan, M., and Newman, M. E. Community structure in social and biological networks. *Proceedings of the national academy of sciences 99*, 12 (2002), 7821–7826.

[38] Girvan, M., and Newman, M. E. J. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences of the United States of America 99*, 12 (2002), 7821–7826.

[39] Goldberg, A. V., and Radzik, T. A heuristic improvement of the bellman-ford algorithm. *Applied Mathematics Letters 6*, 3 (1993), 3–6.

[40] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI* (2012), vol. 12, p. 2.

[41] Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., and Stoica, I. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of OSDI* (2014), pp. 599–613.

[42] Goodarzi, B., Burtscher, M., and Goswami, D. Parallel graph partitioning on a cpu-gpu architecture. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (May 2016), pp. 58–66.

[43] Graham, R. L., and Hell, P. On the history of the minimum spanning tree problem. *Annals of the History of Computing 7*, 1 (1985), 43–57.

[44] Gregor, D., and Lumsdaine, A. The parallel bgl: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC) 2* (2005), 1–18.

[45] Haase, G., Liebmann, M., Douglas, C. C., and Plank, G. A parallel algebraic multigrid solver on graphics processing units. In *High performance computing and applications*. Springer, 2010, pp. 38–47.

[46] Han, W.-S., Lee, S., Park, K., Lee, J.-H., Kim, M.-S., Kim, J., and Yu, H. Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining* (2013), ACM, pp. 77–85.

[47] Harish, P., and Narayanan, P. Accelerating large graph algorithms on the gpu using cuda. In *International Conference on High-Performance Computing* (2007), Springer, pp. 197–208.

[48] Hudelson, M., Mooney, B. L., and Clark, A. E. Determining polyhedral arrangements of atoms using pagerank. *Journal of Mathematical Chemistry 50*, 9 (2012), 2342–2350.

[49] Ilg, M., Rogers, J., and Costello, M. Projectile monte-carlo trajectory analysis using a graphics processing unit. In *2011 AIAA Atmospheric Flight Mechanics Conference, Portland, OR, Aug* (2011), pp. 7–10.

[50] Iordanov, B. Hypergraphdb: a generalized graph database. In *International Conference on Web-Age Information Management* (2010), Springer, pp. 25–36.

[51] JIANG, J. An introduction to spectral graph theory.

[52] Jiang, Y., Zhang, D., Chen, K., Zhou, Q., Zhou, Y., and He, J. An improved memory management scheme for large scale graph computing engine graphchi. In *Big Data (Big Data), 2014 IEEE International Conference on* (2014), IEEE, pp. 58–63.

[53] Karypis, G., and Kumar, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing 20*, 1 (1998), 359–392.

[54] Karypis, G., and Kumar, V. Multilevel k-way hypergraph partitioning. *VLSI design 11*, 3 (2000), 285–300.

[55] Karypis, G., Schloegel, K., and Kumar, V. Parmetis. *Parallel graph partitioning and sparse matrix ordering library. Version 2* (2003).

[56] Kininmonth, S., Van Oppen, M. J. H., and Possingham, H. P. Determining the community structure of the coral seriatopora hystrix from hydrodynamic and genetic networks. *Ecological Modelling 221*, 24 (2010), 2870–2880.

[57] Kitchovitch, S., and Liò, P. Community Structure in Social Networks: Applications for Epidemiological Modelling. *PloS one 6*, 7 (2011), e22220.

[58] Kyrola, A., Blelloch, G. E., and Guestrin, C. Graphchi: Large-scale graph computation on just a pc. In *OSDI* (2012), vol. 12, pp. 31–46.

[59] Lam, C. *Hadoop in Action*, 1st ed. Manning Publications Co., Greenwich, CT, USA, 2010.

[60] Lenharth, A., Nguyen, D., and Pingali, K. Parallel graph analytics. *Communications of the ACM 59*, 5 (2016), 78–87.

[61] Leskovec, J., Rajaraman, A., and Ullman, J. D. *Mining of massive datasets*. Cambridge University Press, 2014.

[62] Lichtenwalter, R., and Chawla, N. V. Disnet: A framework for distributed graph computation. In *Advances in Social Networks Analysis and Mining (ASONAM), 2011 International Conference on* (2011), IEEE, pp. 263–270.

[63] Lin, S., and Kernighan, B. W. An effective heuristic algorithm for the traveling-salesman problem. *Operations research 21*, 2 (1973), 498–516.

[64] Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., and Hellerstein, J. M. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment 5*, 8 (2012), 716–727.

[65] Lumsdaine, A., Gregor, D., Hendrickson, B., and Berry, J. Challenges in parallel graph processing. *Parallel Processing Letters 17*, 01 (2007), 5–20.

[66] Luo, L., Wong, M., and Hwu, W.-m. An effective gpu implementation of breadth-first search. In *Proceedings of the 47th design automation conference* (2010), ACM, pp. 52–55.

[67] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2010), SIGMOD '10, ACM, pp. 135–146.

[68] Martinez-Bazan, N., Gomez-Villamor, S., and Escale-Claveras, F. Dex: A high-performance graph database management system. In *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on* (2011), IEEE, pp. 124–127.

[69] McCune, R. R., Weninger, T., and Madey, G. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR) 48*, 2 (2015), 25.

[70] Miller, J. J. Graph database applications and concepts with neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA* (2013), vol. 2324.

[71] Najeebullah, K., Khan, K. U., Nawaz, W., and Lee, Y.-K. Bishard parallel processor: A disk-based processing engine for billion-scale graphs. *International Journal of Multimedia & Ubiquitous Engineering 9*, 2 (2014), 199–212.

[72] Nasre, R., Burtscher, M., and Pingali, K. Data-driven versus topology-driven irregular computations on gpus. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on* (2013), IEEE, pp. 463–474.

[73] Natarajan, N., Shin, D., and Dhillon, I. S. Which app will you use next?: collaborative filtering with interactional context. In *Proceedings of the 7th ACM conference on Recommender systems* (2013), ACM, pp. 201–208.

[74] Newman, M. Community detection and graph partitioning. *EPL (Europhysics Letters) 103*, 2 (2013), 28003.

[75] Newman, M. E., and Park, J. Why social networks are different from other types of networks. *Physical Review E 68*, 3 (2003), 036122.

[76] Newman, M. E., Strogatz, S. H., and Watts, D. J. Random graphs with arbitrary degree distributions and their applications. *Physical Review E 64*, 2 (2001), 026118.

[77] Newman, M. E. J. Models of the small world. *Journal of Statistical Physics 101*, 3/4 (2000), 819–841.

[78] Newman, M. E. J. The structure and function of complex networks. *SIAM Review 45*, 2 (2003), 58.

[79] Newman, M. E. J., and Girvan, M. Finding and evaluating community structure in networks. *Physical Review E - Statistical, Nonlinear and Soft Matter Physics 69*, 2 Pt 2 (2004), 16.

[80] NVIDIA. Tesla k80 gpu accelerator board specification. http://images.nvidia.com/content/pdf/kepler/Tesla-K80-BoardSpec-07317-001-v05.pdf. Accessed: 2016-08-26.

[81] O'Connell, T. C. A survey of graph algorithms under extended streaming models of computation. In *Fundamental Problems in Computing*. Springer, 2009, pp. 455–476.

[82] Page, L., Brin, S., Motwani, R., and Winograd, T. The pagerank citation ranking: bringing order to the web.

[83] Parlett, B. N., and Scott, D. S. The lanczos algorithm with selective orthogonalization. *Mathematics of computation 33*, 145 (1979), 217–238.

[84] Pellegrini, F., and Roman, J. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *International Conference on High-Performance Computing and Networking* (1996), Springer, pp. 493–498.

[85] Pemmaraju, S., and Skiena, S. *Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica®*. Cambridge university press, 2003.

[86] Pingali, K., Nguyen, D., Kulkarni, M., Burtscher, M., Hassaan, M. A., Kaleem, R., Lee, T.-H., Lenharth, A., Manevich, R., Méndez-Lojo, M., et al. The tao of parallelism in algorithms. In *ACM Sigplan Notices* (2011), vol. 46, ACM, pp. 12–25.

[87] Ravasz, E., Somera, A. L., Mongru, D. A., Oltvai, Z. N., and Barabási, A.-L. Hierarchical organization of modularity in metabolic networks. *science 297*, 5586 (2002), 1551–1555.

[88] Ribichini, A. *Streaming Algorithms for Graph Problems*. PhD thesis, Universita' degli Studi di Roma "La Sapienza", Dipartimento di Informatica e Sistemistica, 2007.

[89] Roy, A., Mihailovic, I., and Zwaenepoel, W. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 472–488.

[90] Salihoglu, S., and Widom, J. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management* (2013), ACM, p. 22.

[91] Sanders, P., and Schulz, C. High quality graph partitioning. *Graph Partitioning and Graph Clustering 588*, 1 (2012).

[92] Schulz, C. *High Quality Graph Partitioning*. epubli, 2013.

[93] Seo, S., Yoon, E. J., Kim, J., Jin, S., Kim, J.-S., and Maeng, S. Hama: An efficient matrix computation with the mapreduce framework. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on* (2010), IEEE, pp. 721–726.

[94] Setia, R., Nedunchezhian, A., and Balachandran, S. A new parallel algorithm for minimum spanning tree problem. In *Proc. International Conference on High Performance Computing (HiPC)* (2009), pp. 1–5.

[95] Shao, B., Wang, H., and Li, Y. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), ACM, pp. 505–516.

[96] Strogatz, S. H. Exploring complex networks. *Nature 410*, 6825 (2001), 268–276.

[97] Sui, X., Nguyen, D., Burtscher, M., and Pingali, K. Parallel graph partitioning on multicore architectures. In *Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing* (Berlin, Heidelberg, 2011), LCPC'10, Springer-Verlag, pp. 246–260.

[98] Svensson, B. J., Sheeran, M., and Newton, R. R. A language for nested data parallel design-space exploration on gpus. Tech. rep., Technical Report 712. Indiana University, 2014.

[99] Tian, Y., Balmin, A., Corsten, S. A., Tatikonda, S., and McPherson, J. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment 7*, 3 (2013), 193–204.

[100] Valiant, L. G. A bridging model for parallel computation. *Communications of the ACM 33*, 8 (1990), 103–111.

[101] Veljan, D., Petkovšek, M., Pisanski, T., Svrtan, D., and Dujella, A. *Kombinatorna i diskretna matematika*. Algoritam Zagreb, 2001.

[102] Vukotic, A., Watt, N., Abedrabbo, T., Fox, D., and Partner, J. *Neo4j in Action*. Manning, 2015.

[103] Walshaw, C., and Cross, M. Jostle: parallel multilevel graph-partitioning software–an overview. *Mesh partitioning techniques and domain decomposition techniques* (2007), 27–58.

[104] Wang, W., Du, S., Guo, Z., and Luo, L. Polygonal clustering analysis using multilevel graph-partition. *Transactions in GIS 19*, 5 (2015), 716–736.

[105] Wang, Y., and Owens, J. Large-scale graph processing algorithms on the gpu. Tech. rep., Technical Report, Electrical and Computer Engineering Department, UC Davis, 2013.

[106] Watts, D. J., and Strogatz, S. H. Collective dynamics of small world networks. *Nature 393*, 6684 (1998), 440–442.

[107] Wilkinson, D. M., and Huberman, B. A. A method for finding communities of related genes. *Proceedings of the National Academy of Sciences of the United States of America 101 Suppl 1*, Suppl 1 (2004), 5241–5248.

[108] Xie, C., Chen, R., Guan, H., Zang, B., and Chen, H. Sync or async: Time to fuse for distributed graph-parallel computation. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2015), ACM, pp. 194–204.

[109] Xin, R. S., Crankshaw, D., Dave, A., Gonzalez, J. E., Franklin, M. J., and Stoica, I. Graphx: Unifying data-parallel and graph-parallel analytics. *arXiv preprint arXiv:1402.2394* (2014).

[110] Xin, R. S., Gonzalez, J. E., Franklin, M. J., and Stoica, I. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems* (2013), ACM, p. 2.

[111] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. Spark: cluster computing with working sets. *HotCloud 10* (2010), 10–10.

[112] Zhang, T., Zhang, J., Shu, W., Wu, M.-Y., and Liang, X. Efficient graph computation on hybrid cpu and gpu systems. *The Journal of Supercomputing 71*, 4 (2015), 1563–1586.

[113] Zheng, D., Mhembere, D., Burns, R., Vogelstein, J., Priebe, C. E., and Szalay, A. S. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (2015), pp. 45–58.

[114] Zhong, J., and He, B. Medusa: Simplified graph processing on gpus. *Parallel and Distributed Systems, IEEE Transactions on 25*, 6 (2014), 1543–1552.

9 Sažetak

U današnje vrijeme mnoge tvrtke temelje svoje poslovanje na znanju kojeg dohvaćaju iz velikih skupova podataka (engl. *Big Data*). Zbog toga se razvijaju različite tehnologije koje bi omogućile pohranu i ubrzanu obradu velikih skupova podataka.

Grafovi nam omogućavaju modeliranje kompleksnih odnosa među objektima i samim tim predstavljaju veliku pomoć u istraživanju strukture i svojstava objekata u različitim znanstvenim disciplinama.

Standardni sustavi za analizu velike količine podataka temeljeni na *map-reduce* paradigmi iako zreli za primjenu na velikim podacima, nisu optimalni za iregularne strukture podataka. U ovom je radu prikazan pregled razvoja paralelnih sustava za graf analitiku i to distribuiranih sustava, višeprocesorskih sustava i sustava koji koriste grafičke procesore.

U zadnjih nekoliko godina razvio se određen broj sinkronih distribuiranih sustava za analizu grafova (Pregel, GraphX, GraphEngine i dr.), asinkronih (GraphLab, PowerGraph), zatim sustava za višejezgrene procesore (GraphChi, TurboGraph, NXGraph) te sustava za izvođenje potpomognuto GPU procesorskim jedinicama (gGraph, Medusa).

Prilagodba sekvencijalnih algoritama paralelnom izvođenju nije trivijalan problem. Osim samog modeliranja algoritama, nužno je poznavati detalje hardvera, softvera koji omogućava komunikaciju među dijelovima sustava te dobro particionirati skup podatka. Međutim, razvoj takvih sustava tek je u povojima i za očekivati je da će u budućnosti sve više istraživača i kompanija istraživati ovo područje. Na kraju rada dan je pregled algoritama za streamanje i particioniranje podataka u formi grafa za bolju paralelnu obradu.

Budući istraživački rad biti će usmjeren na istraživanje performansi paralelnih algoritama za analizu velikih grafova u hibridnom okruženju.