Optimalni vzporedni algoritmi v modelu dvojiškega razcepa

Sara Bizjak
sb6054@student.uni-lj.si

Bor Brecelj
bb3263@student.uni-lj.si

Zala Erič
ze2295@student.uni-lj.si

Laura Guzelj Blatnik
lg6256@student.uni-lj.si

POVZETEK

V članku je povzetih nekaj optimalnih osnovnih algoritmov v modelu dvojiškega razcepa, pri čemer se večinoma opiramo na vir [2]. Ti algoritmi so krčenje seznama, urejanje elementov, naključna permutacija in krčenje drevesa. Izkaže se, da imajo vsi algoritmi v omenjenem modelu optimalen čas glede na število vseh operacij in optimalen čas glede na najdaljšo verigo operacij.

V modelu dvojiškega razcepa se delovanje začne z enim samim procesom, ki se v nadaljevanju lahko rekurzivno razcepi na dva podprocesa naenkrat, in se zaključi, ko se zaključijo vsi podprocesi. Vsi podprocesi imajo dostop do skupnega pomnilnika, izvajajo pa se asinhrono.

Ključne besede

Model dvojiškega razcepa, PRAM, RAM, optimalni algoritmi

1. UVOD

Dandanes ima večina računalnikov večjedrni procesor, zato lahko več operacij izvajajo istočasno. Vendar ker hitrost delovanja programa ni odvisna le od strojne opreme, so se istočasno z razvojem računalnikov razvijali tudi vzporedni algoritmi. Paralelni algoritmi se tako dobro prilegajo modernim računalniškim arhitekturam, vendar jih moramo za vzporedno računanje seveda prilagoditi, če želimo doseči najvišjo možno stopnjo učinkovitosti.

V članku predstavimo rezultate optimalnih algoritmov v modelu dvojiškega razcepa. Glavni vir razprave je članek [2], kjer avtorji kot eni izmed prvih vpeljejo pojem dvojiškega razcepa. Ob tem smo naše delo vsebinsko smiselno razširili še z rezultati iz nekaterih drugih virov.

V modelu dvojiškega razcepa se podprocesi med sabo izvajajo asinhrono in imajo dostop do skupnega pomnilnika. Model uporablja standardne RAM ukaze s posebnim dodatnim ukazom `fork`, ki proces razcepi na dva podprocesa. Delovanje se začne z enim samim procesom, ki se v nadaljevanju lahko rekurzivno razveji na dva podprocesa naenkrat, in se zaključi, ko se zaključijo vsi procesi. Časovno zahtevnost merimo z veličinama *work* in *span*, ki merita število operacij vseh podprocesov skupaj in število operacij najdaljšega podprocesa v tem vrstnem redu.

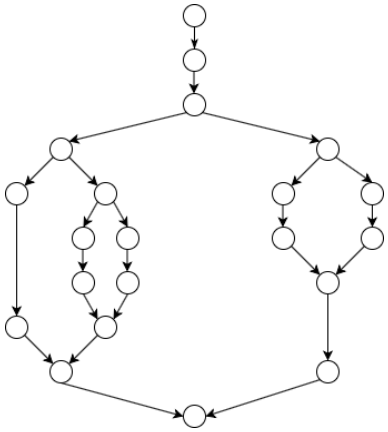
Za analizo vzporednih algoritmov se večinoma uporablja model PRAM (*angl. parallel random access machine*), ki predpostavlja, da se operacije izvajajo sinhrono, kar je v praksi težko izvedljivo. Prav zaradi tega se avtorji v referenčnem članku osredotočijo na optimalne algoritme v modelu dvojiškega razcepa, ki opisuje moderne večjedrne računalnike.

Problem nastane, če želimo prenesti optimalne algoritme iz PRAM modela v model dvojiškega razcepa, saj se poveča njihova časovna zahtevnost. Optimalni algoritmi v modelu PRAM torej niso več nujno optimalni v modelu dvojiškega razcepa. Iskanje optimalnih algoritmov v modelu dvojiškega razcepa je zanimiv problem, saj moramo uvesti drugačne algoritmične pristope.

2. MODEL DVOJIŠKEGA RAZCEPA

V tem poglavju podamo opis modela dvojiškega razcepa, ki spada med *večprocesne modele*. Večprocesni model je skupek dinamično zgrajenih procesov (*angl. threads*), ki praviloma potekajo asinhrono. Za izračun časovne zahtevnosti takega procesa moramo opazovati tako število operacij vseh procesov skupaj (v nadaljevanju *work*) kot število operacij najdaljšega procesa (v nadaljevanju *span*). Poznamo več inačic večprocesnih modelov, ki se razlikujejo po številu sočasno izvajanih procesov, sinhronosti in po načinu dostopanja do pomnilnika.

V modelu dvojiškega razcepa imajo vsi procesi dostop do skupnega pomnilnika, vsak proces pa se lahko deli le na dva nova naenkrat, ki se v nadaljevanju izvajata vzporedno (kot prikazano na sliki 1). Vsak proces oz. naloga, ki se izvaja asinhrono z drugimi, deluje podobno kot RAM. To pomeni, da deluje na programu, ki je shranjen v skupnem pomnilniku, ima konstantno število registrov in uporablja standardne RAM ukaze. Pomembna nadgradnja pri modelu dvojiškega razcepa, v primerjavi z RAM, je dodan ukaz `fork`, ki proces razcepi na *podprocesa* (*angl. child thread*). V modelu se uporablja tudi posebna oznaka za konec, in sicer `endall`, ki zaznamuje konec celotnega postopka. Namesto ukaza `join`, ki v principu skrbi za to, da počaka na konec izvajanja drugega procesa iz zadnjega razcepa in šele nato nadaljuje z izvajanje, je v modelu prisoten ukaz `test-and-set` (v nadaljevanju `TS`), ki naenkrat prebere bit in ga nato nastavi na 1. S tem ukazom lahko `join` implementiramo tako, da se, ko oba (pod)procesa prideta do `join` ali `end`, izvede ukaz `TS` nad vnaprej določenim bitom, ki je bil nas-

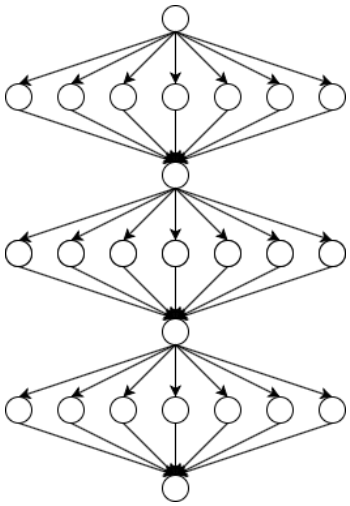


Slika 1: Prikaz delitve procesa v modelu dvojiškega razcepa.

tavljen na 0. Če `TS` vrne 0, pomeni, da se drugi proces še ni končal, in se proces, ki je klical `TS`, konča. Če pa `TS` vrne 1, pomeni, da se je drugi proces že končal in mora ta proces nadaljevati izvajanje. Omenimo še, da je uporaba ukaza `TS` v modelu smiselna, saj ga podpirajo vsi novejši procesorji.

Izvajanje operacije v modelu dvojiškega razcepa se začne z enim t. i. *začetnim* procesom, ki se razdeli na dva podprocesa (postopek se lahko nadaljuje rekurzivno na podprocesih) in zaključi, ko se izvede ukaz `endall`. To operacijo si tako lahko predstavljamo kot drevo, katerega vozlišča so *ukazi*, koren drevesa pa predstavlja ukaz začetnega procesa. Časovna zahtevnost take operacije je ovrednotena z *work* in *span* vrednostmi. *Work* je enak velikosti drevesa, to je številu vseh vozlišč v drevesu oz. številu vseh ukazov, *span* pa je enak globini drevesa oz. najdaljši poti ukazov od korena do lista drevesa.

Vredno je omeniti, da lahko opisan model iz dvojiškega posplošimo tudi na *več-razcepnega*. Ukaz `fork` bi v tem primeru sprejel tudi število, ki bi določalo, na koliko delov naj se proces deli. Taka delitev procesa je prikazana na sliki 2.



Slika 2: Prikaz delitve procesa več-razcepnega modela (v tem primeru je model 7-razcepni).

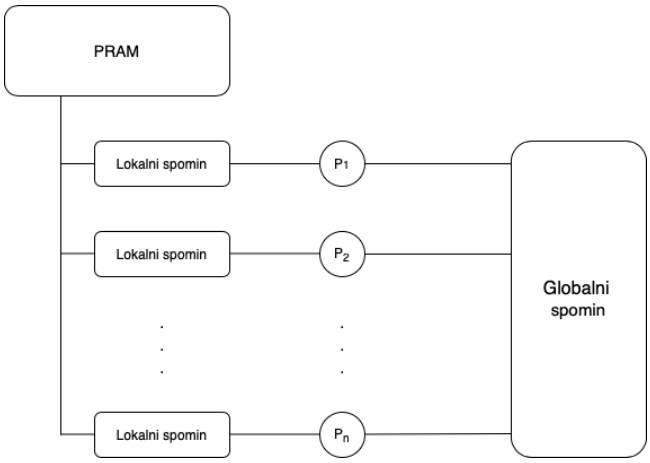
V referenčnem članku je obravnavan le model dvojiškega razcepa predvsem zaradi dejstva, da nekateri najučinkovitejši rezultati z nizkim številom operacij vseh procesov (z nizko zahtevnostjo *work*) veljajo *le* za dvojiške delitve. Zato se tudi tukaj v nadaljevanju osredotočimo le na algoritme v modelu dvojiškega razcepa.

2.1 PRAM

Model PRAM (*angl. parallel random access machine*) poznamo že iz poznih sedemdesetih let prejšnjega stoletja in je naravna vzporedna različica modela RAM (*angl. random access machine*).

Model RAM se kot osnovo praviloma uporablja za analizo zaporednih algoritmov. Pri modelu PRAM pa je prisotnih n procesorjev in vsi imajo dostop do (neomejenega) skupnega oz. globalnega pomnilnika, kamor zapisujejo vhode in izhode. Vsak procesor P_i hrani tudi svoj lokalni spomin. Posamezni procesor lahko v enem časovnem ciklu izvede celotno operacijo (aritmetično, logično, dostop do pomnilnika) [3].

Shema delovanja modela PRAM je prikazana na sliki 3, kjer procesorje označimo s $P_1, P_2, \ldots, P_n$.



Slika 3: Prikaz modela PRAM.

2.2 Primerjava med modelom PRAM in modelom dvojiškega razcepa

Potrebno je omeniti, da je model PRAM veliko zmogljivejši od modela dvojiškega razcepa. Izkaže pa se, da se na modernih računalniških arhitekturah bolje odreže model dvojiškega razcepa, saj na le-teh operacije velikokrat potekajo asinhrono, kar model dvojiškega razcepa upošteva, medtem ko model PRAM operacije izvaja sinhrono, torej istočasno [1]. Optimalen algoritem v modelu PRAM tako ni nujno optimalen v modelu dvojiškega razcepa. Medtem ko je zahtevnost veličine *work* pri obeh algoritmih asimptotično enaka, se modela razlikujeta v veličini *span*. Optimalni algoritmi v modelu PRAM so že dolgo znani, tako je v nadaljevanju smiselno predstaviti optimalne algoritme v modelu dvojiškega razcepa.

3. OPTIMALNI ALGORITMI V MODELU DVOJIŠKEGA RAZCEPA

Predstavimo nekaj novih optimalnih algoritmov za reševanje osnovnih problemov v modelu dvojiškega razcepa. Vsi predstavljeni algoritmi so optimalni glede na število operacij vseh procesov skupaj (*work*) in glede na čas izvajanja, ki je definiran kot največje število zaporednih operacij.

3.1 Pregled področja

V članku [4] avtorji predstavijo idejo za paralelizacijo zaporednih iterativnih algoritmov za naključno permutacijo, krčenje seznama in krčenje drevesa. Algoritmi niso zapisani in analizirani v modelu dvojiškega razcepa, jih pa avtorji empirično primerjajo z običajnimi iterativnimi implementacijami algoritmov. Časi izvajanja pokažejo, zakaj je sploh smiselno analizirati paralelne algoritme. Tabela 1 prikazuje čase izvajanja različnih implementacij omenjenih algoritmov.

Algoritem	(1) [s]	(40h) [s]	(zap) [s]
Krčenje seznama	92.1	4.62	38.8
Naključna permutacija	160	3.97	46
Krčenje drevesa	350	10.0	172

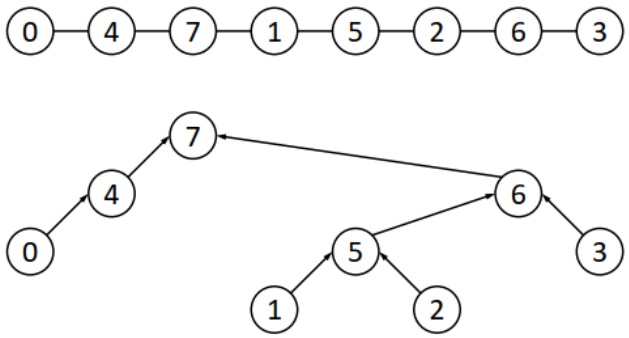
Tabela 1: Tabela je povzeta po članku [4] in prikazuje čase izvajanja različnih algoritmov. Z (1) je označen vzporeden algoritem z enim podprocesom, (40h) označuje 80 hiperpodprocesov in (zap) označuje zaporeden iterativen algoritem.

Očitno je, da je vzporedna implementacija vseh algoritmov veliko bolj učinkovita od zaporedne iterativne implementacije. Iz tabele razberemo, da je vzporedni algoritem z zgolj enim podprocesom celo počasnejši od iterativnega. Še več, izkaže se, da je uporaba vzporednega algoritma smiselna le, če uporabimo vsaj 4 podprocese. V tem primeru se čas izvajanja nezanemarljivo zmanjša. Empirično dokazani rezultati tako potrjujejo, da je vzporedne algoritme smiselno obravnavati.

3.2 Krčenje seznama

V tem razdelku si pogledamo problem, kjer kot vhod dobimo dvosmeren povezan seznam. Naša naloga je vsakemu vozlišču določiti njegov indeks oz. razdaljo do začetka seznama (*angl. list ranking problem*). Sekvenčni algoritem je enostaven, ker se moramo enkrat sprehoditi čez sezam. To pomeni časovno zahtevnost $O(n)$. Za vzporedni algoritem se problema lotimo s krčenjem seznama. To pomeni, da bomo seznam krčili, dokler ne dobimo seznama z enim samim vozliščem (*angl. list contraction*). Nato dobimo indekse vozlišč tako, da skrčen seznam nazaj razširimo. V modelu PRAM že poznamo optimalen algoritem s težavnostjo $O(n)$ pri veličini *work* in težavnostjo $O(\log n)$ pri veličini *span*. Ker algoritem temelji na več krogih, ki morajo biti sinhronizirani, ni optimalen v modelu dvojiškega razcepa.

Optimalen algoritem v modelu dvojiškega razcepa temelji na ugotovitvi, da lahko naenkrat izbrišemo dve vozlišči, če nista sosednji. To dosežemo tako, da vsakemu vozlišču priredimo naključno prioriteto z uporabo naključne permutacije, ki je opisana v nadaljevanju. Torej, naenkrat lahko izbrišemo vsa tista vozlišča, ki imajo to prioriteto manjšo od obeh sosedov. Vrstni red brisanja lahko opišemo z drevesom, kot je prikazano na sliki 4.



Slika 4: Prikaz dvosmernega povezanega seznama z naključno izbranimi prioritetami vozlišč. Pod njim je drevo vrstnega reda brisanja vozlišč. Oče vozlišča je sosed, ki ima višjo prioriteto. Če imata oba soseda višjo prioriteto, za očeta vzamemo tistega z nižjo prioriteto. Preden zbrišemo vozlišče, moramo zbrisati vse njegove sinove.

Z L označimo dvosmeren povezan seznam dolžine n. Vsak element l_i ima shranjeno prioriteto ($l_i.p$), kazalec na naslednji element ($l_i.next$), kazalec na prejšnji element ($l_i.prev$) in zastavico ($l_i.flag$).

Ideja algoritma je, da za vsako vozlišče seznama ustvarimo en proces. Procesi, katerih vozlišče nima nižje prioritete od obeh sosedov, se takoj končajo. Ostali procesi zbrišejo svoje vozlišče in želijo nadaljevati z brisanjem sosednjega vozlišča z nižjo prioriteto. Lahko se zgodi, da dva procesa zbrišeta pripadajoče vozlišče in nato želita zbrisati isto vozlišče. V tem primeru moramo združiti procesa tako, da le en od njiju nadaljuje z brisanjem. Za to uporabimo ukaz `TS` nad zastavico ($c.flag$). Če je vrednost zastavice 0, potem jo proces nastavi na 1 in konča. Če pa je vrednost 1, potem proces ve, da je drugi proces že zaključil in nadaljuje z brisanjem. Da se algoritem res tako izvede, moramo na začetku nastaviti na 1 zastavice vseh vozlišč, ki imajo največ enega soseda z nižjo prioriteto, kar naredi prva vzporedna for zanka iz algoritma 1.

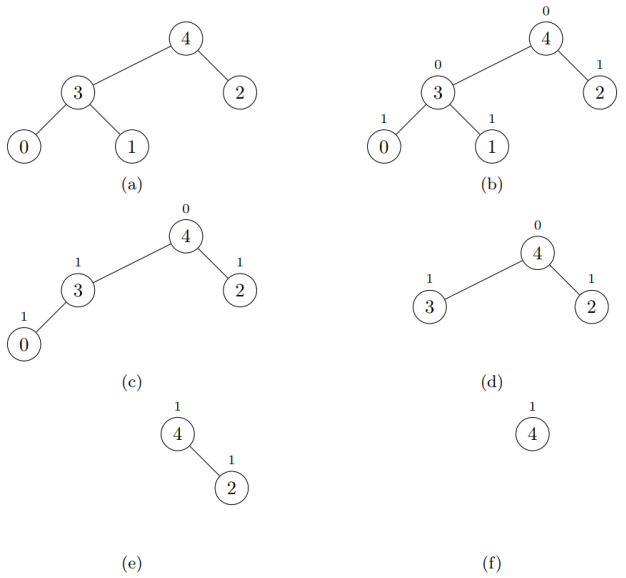
Druga vzporedna for zanka za trenutno vozlišče (c) najprej preveri ali so prioritete sosedov manjše. Nato briše vozlišča dokler ne pride do konca ali pa ukaz `TS` vrne 0. Postopek brisanja vozlišč je prikazan na sliki 5.

Ker vsako vozlišče enkrat zbrišemo se while zanka skupno ponovi n-krat, torej ima veličina algoritma *work* težavnost $O(n)$. Višina drevesa, ki prikazuje zaporedje brisanja vozlišč, je $O(\log n)$ z veliko verjetnostjo, ker je definirano z naključno permutacijo. To pomeni, da ima veličina algoritma *span* z veliko verjetnostjo težavnost $O(\log n)$.

Algoritem 1: Krčenje seznama, povzeto po [2].

Vhod: Dvosmeren povezan seznam L dolžine n. Vsak element l_i vsebuje naključno izbrano prioriteto $(l_i.p)$, kazalec na naslednji element $(l_i.next)$, kazalec na prejšnji element $(l_i.prev)$ in zastavico $(l_i.flag)$.

```
parallel foreach element l_i iz L do
    // Nastavi zastavico na 1, če ima element največ enega otroka
    l_i.flag ← (pri(l_i) < pri(l_i.prev)) or (pri(l_i) < pri(l_i.next))
parallel foreach element l_i iz L do
    c ← l_i
    // Nadaljuj, če je c list
    if (pri(c) < pri(c.prev)) and (pri(c) < pri(c.next)) then
        // Končaj, ko je seznam skrčen v en element
        while not (c.prev = null and c.next = null) do
            Izbriši c
            Naj bo c' tisti izmed c.prev in c.next z manjšo prioriteto
            // Končaj, če c ni zadnji otrok od c'
            if test-and-set(c'.flag) then break
            c ← c'
// Pomožna funkcija
Function pri(v)
    if v = null then return ∞ else return v.p
```



Slika 5: Postopek brisanja vozlišč seznama s prioritetami [0, 3, 1, 4, 2]**. (a) prikazuje drevo vrstnega reda brisanja vozlišč, (b) ima nad vozlišči še začetne vrednosti zastavic. Na začetku lahko vsa vozlišča z vrednostjo zastavice** 1 **zbrišemo naenkrat ali v poljubnem vrstnem redu. Za lepši prikaz postopka bomo izbrali en vrstni red in začeli brisanje z vozliščem** 1 **(c). Ta proces želi nadaljevati z brisanjem vozlišča** 3**, ampak konča in le nastavi vrednost zastavice na** 1**. Nadaljujemo z brisanjem vozlišč** 0 **(d),** 3 **(e) in** 2 **(f).**

3.3 Urejanje

V tem poglavju predstavimo nov pristop algoritma za urejanje s primerjanjem, ki razvrsti n elementov v $\mathcal{O}(n \log n)$ pričakovane veličine *work* in $\mathcal{O}(\log n)$ veličine *span*, kar je precej bolje, kot do tedaj odkriti agloritmi - najboljši do sedaj poznani rezultat glede na veličino *work*, urejanje s primerjanjem n elementov (*angl. comparison sort*), je v modelu dvojiškega razcepa zavzel kar $\mathcal{O}(\log n \log\log n)$ veličine *span*. Pseudo-koda paraleliziranega urejanja je opisana v algoritmu 2.

Algoritem je rekurziven in sloni na uporabi sortiranja vzorcev (*angl. sample sorting*). Algoritem v osnovnem primeru (to je takrat, ko velikost podproblema pade pod določen prag) zaporedno razvrsti elemente. V nasprotnem primeru pa, ko imamo podproblem velikosti n, enakomerno (z neko mero naključnosti) izbere $n^{1/3} \log_2 n$ vzorcev, ki jih potem vsakega posebej razvrsti s pomočjo algoritmov, ki imajo kvadratično zahtevnost *work*, saj elemente parno primerjajo med sabo. Ta dva koraka — delitev in razvrščanje vzorcev — lahko izvedemo z $\mathcal{O}(n)$ veličine *work* in $\mathcal{O}(\log n)$ veličine *span*.

Algoritem nadaljuje tako, da izbere pivote, s pomočjo katerih razdeli elemente po skupinah. To naredi tako, da za pivot izbere vsak $\log_2 n$-ti vzorec, elementi pa se tako razdelijo na $n^{1/3} + 1$ skupin, saj imamo $n^{1/3}$ pivotov. Algoritem potem vsaki skupini dodeli seznam velikosti $2c_1 r n^{2/3}$ (za določeno konstanto r in katerokoli konstanto $c_1 > 1$), kamor v nadaljevanju vzporedno pripisuje elemente. Element doda v skupino oz. pripadajoč seznam tako, da ga poskusi umestiti na naključno mesto v tem seznam s pomočjo `TS`, ki izbrano mesto rezervira. Če je mesto že zasedeno (`TS` se ne izvede), ponovno poskusi z naključnim izborom indeksa v seznamu, kar lahko ponovi največ $c_2 \log_2 n$-krat, kjer s c_2 označimo konstanto večjo od 1.

Pri poskusu dodajanja elementov v skupine (vrstica 7 v algoritmu 2) je največje število elementov, ki jih algoritem lahko dodeli isti skupini, enako $c_1 r n^{2/3}$ z verjetnostjo vsaj $a - n^{c_1}$ za določeno konstanto r in katerokoli konstanto $c_1 > 1$ (po lemi, dostopni v [2]).

Če v teh ponovitvah za kakšen element ne najde prostega mesta v seznamu, algoritem ponovi postopek in ponovno izbere $n^{1/3} \log_2 n$ vzorcev (to pomeni, da se vrne v četrto vrstico algoritma 2).

Ko algoritem razvrsti vse elemente po skupinah, pošlje te skupine v naslednji rekurzivni klic.

3.4 Naključna permutacija

Naključna permutacija se pogosto uporablja kot pomožna metoda pri ostalih algoritmih (npr. krčenje seznama), zato je algoritem naključne permutacije v vzporednem računanju pogosto obravnavan. Večina dosedanjih algoritmov ima polilogaritemsko globino in linearen *work*, poleg tega pa obstoječi algoritmi niso povsem naključni. Tako avtorji članka [4] predstavijo iterativen zaporedni algoritem, katerega cilj je naključno premešati elemente seznama. Sprehajamo se od konca seznama proti njegovemu začetku in ob tem vsak element zamenjamo z naključnim elementom, ki se nahaja

Algoritem 2: Urejanje s primerjanjem, povzeto po [2].

Function *Uredi(A)*
Naj bo $n = A$
if *n je konstanta* **then** Uredi osnoven primer in vrni rezultat
Naključno izberi $n^{1/3} \log_2 n$ vzorcev
Uredi vzorce s kvadratičnim algoritmom
Iz vzorcev izberi $n^{1/3}$ pivotov
Vse elemente iz A razdeli v skupine glede na izbrane pivote (skupine označimo s A_0, A_1, ..., $A_{n^{1/3}}$). Če ne uspe ponovi naključno Izbiranje vzorcev in nadaljuj od tam
parallel foreach $i \leftarrow 0$ *to* $n^{1/3}$ **do** Uredi(A_i)

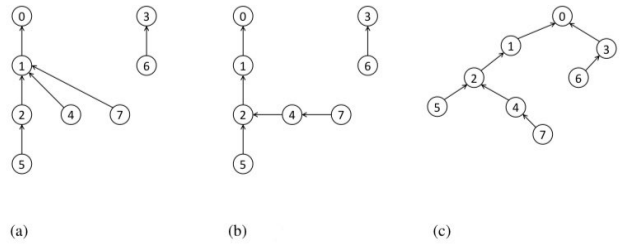
na tem mestu ali pred njim (seveda bi lahko elemente menjavali tudi v obratnem vrstnem redu). Algoritem uporablja pomožno tabelo H, v kateri je vsak element $H[i]$ uniformno naključno število med 0 in vključno i, pseudo-koda je opisana v algoritmu 3.

Algoritem 3: Naključna permutacija, povzeto po [2].

Function *NaključnaPermutacija(A, H)*
$A[i] \leftarrow i$ for all $i = 0, ..., n-1$
for $i \leftarrow n-1$ **to** 0 **do**
Zamenjaj($A[i]$, $A[H[i]]$)

V kolikor sta množici začetnih in končnih lokacij elementov neodvisni, lahko več menjav izvedemo vzporedno. Tako vpeljemo pojem *dominantnosti*, pravimo, da i dominira j v kolikor velja $H[i] = j$ za $i \neq j$. Glede na dominantnost elementov lahko definiramo *gozd dominantnosti* kot usmerjen graf z enakim številom vozlišč, kot je dolžina tabele A. V grafu sta vozlišči i in j povezani, v kolikor i dominira j. Ker lahko vsako vozlišče dominira največ eno izmed ostalih volišč, bo dobljeni graf gozd. Koreni dreves v gozdu dominantnosti bodo natanko elementi za katere velja $H[i] = i$. Primer gozda dominantnosti za tabelo $H = [0, 0, 1, 3, 1, 2, 3, 1]$ je prikazan na sliki 6(a). Nadalje vpeljemo še pojem *gozda odvisnosti*, kjer otroke vozlišč drevesa dominantnosti verižno povežemo v padajočem vrstnem redu. Natančneje, za vozlišče i z vhodnimi povezavami iz vozlišč $j_1 < \cdots < j_k$ dodamo povezave iz j_{l+1} v j_l za $1 \leq l < k$ (tako dobimo verigo) in izbrišemo povezave iz j_l v i za $l \geq 1$. Primer gozda odvisnosti za zgoraj definirano tabelo H se nahaja na sliki 6(b). Odvisnost med dvema vozliščema pomeni odvisnost pri izračunu – istočasno lahko namreč izvedemo menjave vseh listov v gozdu odvisnosti, na koncu pa bomo dobili enak rezultat, kot če bi menjave opravljali zaporedno. Torej, v enem vzporednem koraku algoritma lahko simultano zamenjamo položaje vseh elementov v listih gozda in jih nato izbrišemo. To lahko storimo, ker so listi v enakem ali različnih drevesih med seboj neodvisni.

Za analizo časovne zahtevnosti drevesa v gozdu odvisnosti združimo in jih povežemo v *povezano drevo odvisnosti*, slika 6(c). Za začetno tabelo A z n elementi tako dobimo binarno iskalno drevo, ki prav tako vsebuje n elementov, njegova globina bo tako $O(\log n)$ z veliko verjetnostjo. Posledično to pomeni, da bo algoritem naključne permutacije v modelu dvojiškega razcepa v pričakovanju dosegel težavnost $O(n)$ pri *work*, njegov *span* pa bo dosegel težavnost $O(\log n)$ z veliko verjetnostjo.



Slika 6: Primer gozda dominantnosti (a), gozda odvisnosti (b) in drevesa odvisnosti (c) pri naključni permutaciji, povzeto po [4].

3.5 Krčenje drevesa

Paralelni algoritmi za krčenje dreves imajo veliko interesa in raziskav zaradi visoke aplikativnosti na razna drevesa in grafe. Obstaja veliko različic, vendar v opisanem algoritmu predpostavljamo, da zmanjšujemo zakoreninjena binarna drevesa. To so drevesa, kjer specificiramo eno vozlišče kot koren, in kjer ima vsako notranje vozlišče dva otroka. Vsako zakoreninjeno drevo lahko skrčimo v linearnem času glede na število vseh operacij in logaritmičnem času glede na najdaljšo verigo operacij. Predpostavimo, da algoritem izvajamo nad drevesom T, ki ima n listov in $(n-1)$ notranjih vozlišč, ter da v vozlišču v levo poddrevo označimo z $v.lC$, desno pa z $v.rC$.

Izrojeno drevo, kjer so vsa vozlišča verižena, lahko predstavimo kot seznam. Za takšen primer ne poznamo optimalnega paralelnega algoritma za skrčenje z zahtevnostima $O(n)$ in $O(\log n)$.

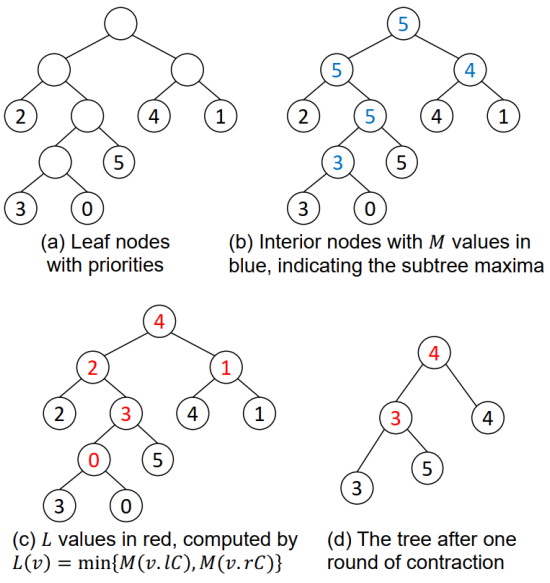
Tu poskusimo s paralelizacijo sekvenčnega iterativnega algoritma, ki obdeluje po en list. Taki operaciji bomo v nadaljevanju rekli operacija grabljenja (*angl. rake operation*). Natančneje, to je operacija, ki list l in njegovega starša v odstrani iz drevesa. Bratu lista l določimo za starša svojega starega starša (starša vozlišča v), staremu staršu lista l (staršu vozlišča v) pa za otroka določimo brata lista l namesto vozlišča v. To ponavljamo dokler ne ostane zgolj koren. V algoritmu 4 je prikazan takšen sekvenčni iterativni algoritem za krčenje drevesa. Za takšen algoritem mora biti drevo predstavljeno kot seznam vozlišč, kjer je prvih n elementov listov, ostalih $n-1$ elementov pa notranjih vozlišč. Ta postopek je zelo podrobno opisan v članku [4], ki je v našem izhodiščnem članku naveden kot vir, vendar vsebinsko smiselno oplemeniti obravnavane algoritme.

Da takšen algoritem paraleliziramo, vsakemu listu določimo naključno prioriteto, ki določa globalni razpored odstranjevanja vozlišč. Krčimo drevo na način, da na koncu ostane samo vozlišče z najnižjo prioriteto. Če hranimo v vozliščih še dodatne informacije, lahko to uporabimo pri različnih aplikacijah dreves. Želimo se izogniti operaciji grabljenja nad dvema vozliščema hkrati, kjer je eden starš od drugega, vendar pa to operacijo lahko simultano izvajamo na več listih. Da se odločimo, katera vozlišča so lahko naenkrat v

Algoritem 4: Sekvenčni algoritem za iterativno krčenje drevesa. Klic $sibling(T, i)$ vrne brata vozlišča i v drevesu T, klic $switchParentsChild(T, i, v)$ pa ponastavi kazalec otroka starša vozlišča i na vozlišče i namesto na vozlišče v.

```
Function SekvenčnoKrčenjeDrevesa(T)
    for i ← 1 to n do
        p ← T[i].parent
        // Če p ni koren drevesa
        if T[p].parent ≠ null then
            s ← sibling(T, i)
            T[s].parent ← T[p].parent
            switchParentsChild(T, p, s)
        else
            switchParentsChild(T, i, null)
```

obdelavi, definiramo vrednost $M(v)$ vsakega notranjega vozlišča v kot najnižjo prioriteto (najvišjo vrednost) vseh listov v njegovih poddrevesih. Glede na $M(\cdot)$, definiramo še $L(v) = min\{M(v.lC), M(v.rC)\}$, če je v notranje vozlišče, oziroma definiramo L kar kot svojo lastno prioriteto, če je v list. V sliki 7 je v `b` in `c` prikazano, kako na drevesu z naključno generiranimi prioritetami (iz `a`) izračunamo vrednosti M in L. Glede na izračunane vrednosti $L(\cdot)$ paralelni algoritmi vedo, ali lahko pograbijo v: če ima starš vozlišča v manjšo vrednost L kot bratje vozlišča v in stari starš vozlišča v, ga lahko. Sicer počaka vozlišče na naslednji krog krčenja. Na sliki 7 je v `d` prikazano drevo po prvem krogu krčenja. Vidimo, da je že v prvem krogu algoritem lahko pograbil tri pare listov in njihovih staršev, namesto enega. Vozlišča, ki so po zgornjem pogoju kvalificirala za paralelno grabljenje, so bili list z vrednostjo 2 in njegov starš, list z vrednostjo 0 in njegov starš, ter list z vrednostjo 1 in njegov starš. Drevo smo tako že v prvem krogu bolj učinkovito skrčili.



Slika 7: Prikaz izračuna vrednosti $M(\cdot)$ in $L(\cdot)$ pri drevesu z določenimi prioritetami, ter paralelna okrajšava tako določenega drevesa. Slika je vzeta iz članka [2]

.

Računanje $M(\cdot)$ in $L(\cdot)$ z visoko verjetnostjo porabi $O(n)$ glede na število vseh operacij in $O(logn)$ glede na najdaljšo verigo operacij.

4. REZULTATI

Vsi opisani algoritmi so optimalni glede na veličini *work* in *span* v modelu dvojiškega razcepa. Njihove časovne zahtevnosti povzema tabela 2. Potrebno je omeniti, da za vse algoritme v tabeli zapisano vrednost veličine *span* dobimo z veliko verjetnostjo. Časovna zahtevnost *work* pa je za algoritme razvrščanja, naključne permutacije in krčenja drevesa definirana v pričakovanju.

Algoritem	*work*	*span*
Krčenje seznama	$O(n)$	$O(\log n)$
Urejanje	$O(n \log n)$	$O(\log n)$
Naključna permutacija	$O(n)$	$O(\log n)$
Krčenje drevesa	$O(n)$	$O(\log n)$

Tabela 2: Tabela prikazuje optimalne časovne zahtevnosti algoritmov v modelu dvojiškega razcepa.

V optimalnost algoritmov glede na veličino *work* se lahko hitro prepričamo, saj za vse te probleme poznamo optimalne sekvenčne algoritme z enakimi časovnimi zahtevnostmi. Da se prepričamo, da so opisani algoritmi optimalni glede na veličino *span*, se moramo spomniti, da se vsak algoritem razcepi na n podprocesov. Ker predstavljen model dovoli le dvojiške razcepe, potrebujemo $\log n$ korakov že za njihov razcep. Tako ne obstajajo algoritmi z veličino *span* manjšo od $O(\log n)$, torej so opisani algoritmi optimalni tudi glede na veličino *span*.

5. ZAKLJUČEK

V članku predstavimo model dvojiškega razcepa, ki vsako operacijo razdeli na dva asinhrona podprocesa. Ravno asinhrono delovanje podprocesov je razlog, da se model dobro prilega današnjim arhitekturam računalniških procesorjev. Omejitev, da se lahko proces razcepi le v dva podprocesa naenkrat, omogoča učinkovito razporejanje procesov. Uporaba ukaza `TS` namesto `join` pa ohranja model čim bolj preprost.

Nadalje predstavimo nekaj algoritmov, ki so za opisan model optimalni. Predstavljeni algoritmi rešujejo osnovne probleme iz področja vzporednega računanja, kar pomeni, da so ideje iz teh algoritmov, uporabne tudi v zahtevnejših problemih.

Zanimivo bi bilo primerjati, kakšne časovne zahtevnosti opisanih algoritmov bi dobili, če bi namesto dvojiškega uporabili več-razcepni model. Prav tako bi bilo zanimivo primerjati čase izvajanja algoritmov v modelu dvojiškega razcepa in več-razcepnem modelu.

6. VIRI

[1] Z. Ahmad, R. Chowdhury, R. Das, P. Ganapathi, A. Gregory, and M. M. Javanmard. Low-depth parallel algorithms for the binary-forking model without atomics, 2020.

[2] G. E. Blelloch, J. T. Fineman, Y. Gu, and Y. Sun. Optimal parallel algorithms in the binary-forking model. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '20, page 89–102, New York, NY, USA, 2020. Association for Computing Machinery.

[3] J. F. JaJa. *PRAM (Parallel Random Access Machines)*, pages 1608–1615. Springer US, Boston, MA, 2011.

[4] J. Shun, Y. Gu, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons. *Sequential Random Permutation, List Contraction and Tree Contraction are Highly Parallel*, pages 431–448.