Paralelni algoritmi

Paralelno izračunavanje više nije egzotična oblast, i već duže vreme je na glavnom pravcu razvoja računarstva. Razvija se vrlo brzo, čak i u odnosu na druge računarske oblasti. U upotrebi je više tipova paralelnih računara, sa brojem procesora u opsegu od 2 do 65536, i većim. Razlike između različitih postojećih računara vrlo su velike. Nemoguće je usvojiti jedan opšti model izračunavanja koji bi obuhvatao sve paralelne računare.

U ovom poglavlju nisu pokrivene sve (pa čak ni većina) oblasti povezanih sa paralelnim izračunavanjem. Prikazani su primeri korišćenja pojedinih modela izračunavanja i različite tehnike. Cilj je sticanje osnovnih znanja o paralelnim algoritmima i upoznavanje sa poteškoćama vezanim za njihovu konstrukciju. Na početku ćemo razmotriti zajedničke karakteristike paralelnih algoritama, zatim ukratko opisati neke osnovne modele paralelnog izračunavanja, a nakon toga videćemo primere paralelnih algoritama i nekih opštih tehnika.

Kada su u pitanju sekvencijalni algoritmi, osnovne mere složenosti su vreme izvršavanja i veličina korišćene memorije. Ove mere su važne i kod paralelnih algoritama, ali se mora voditi računa i o drugim resursima, posebno o broju procesora. Postoje problemi koji su suštinski sekvencijalni i koji se ne mogu "paralelizovati" čak ni ako je na raspolaganju neograničen broj procesora. Ipak, većina ostalih problema može se do nekog stepena paralelizovati. Što više procesora se koristi – do neke granice – algoritam se brže izvršava. Važno je proučavati ograničenja paralelnih algoritama, i biti u stanju okarakterisati probleme za koje postoje vrlo brza paralelna rešenja. Pošto je broj procesora ograničen, isto tako je važno da se procesori efikasno koriste. Sledeći važan element je komunikacija između procesora. Često je više vremena potrebno da dva procesora razmene podatke, nego da se izvrše jednostavne operacije sa podacima. Pored toga, trajanje razmene podataka može da zavisi od "udaljenosti" dva procesora u računaru. Prema tome, važno je minimizirati komunikaciju i organizovati je na efikasni način. Sledeće važno pitanje je sinhronizacija, koja je veliki problem kod paralelnih algoritama čiji se pojedinačni delovi izvršavaju na nezavisnim mašinama, povezanim nekom mrežom za komunikaciju, sa ograničenom informacijom o tome šta rade drugi delovi algoritma. Takvi algoritmi se obično zovu *distribuirani algoritmi*. Njih ovde nećemo razmatrati; ograničićemo se na modele sa potpunom sinhronizacijom.

Neki modeli paralelnog izračunavanja sadrže ograničenje da svi procesori u jednom koraku izvršavaju jednu istu instrukciju (nad eventualno različitim podacima). Paralelni računari sa ovakvim ograničenjem zovu se SIMD (skraćenica od Single–Instruction Multiple–Data) računari. Paralelni računari kod kojih svaki procesor može da izvršava različit program zovu se MIMD (skraćenica od Multiple–Instruction Multiple–Data) računari. Ukoliko se ne naglasi drugačije,

pretpostavlja se da su računari o kojima je reč MIMD računari.

Osnovni pojmovi

Postoji veliki broj različitih modela paralelnih računara i mi ćemo se u ovom materijalu ograničiti na osnovne modele. U ovom odeljku izložićemo neka opšta razmatranja i definicije koji se odnose na mnoge modele. Svaki od sledećih odeljaka pokriva jedan od tipova modela, sadrži njegov detaljniji opis i primere algoritama.

Performanse paralelnih algoritama

Vreme izvršavanja paralelnog algoritma označavaćemo sa $T(n, p)$, gde je n veličina ulaza, a p broj procesora. Odnos

$$S(p) = \frac{T(n, 1)}{T(n, p)}$$

zove se *ubrzanje* (eng. speedup) algoritma. Pritom za vrednost $T(n, 1)$ treba uzeti najbolji poznati sekvencijalni algoritam. Paralelni algoritam je najefikasniji kad je $S(p) = p$, tj. kad algoritam dostiže *savršeno ubrzanje*. Važna mera iskorišćenosti procesora je *efikasnost* (eng. efficiency) paralelnog algoritma, koja se definiše izrazom

$$E(n, p) = \frac{S(p)}{p} = \frac{T(n, 1)}{pT(n, p)}.$$

Efikasnost paralelnog algoritma može se videti kao ubrzanje po procesoru. Ona se računa kao odnos vremena izvršavanja najboljeg sekvencijalnog algoritma (koji se izvršava na jednom procesoru) i ukupnog vremena izvršavanja paralelnog algoritma na p procesora (ukupno vreme je stvarno proteklo vreme pomnoženo brojem procesora). Efikasnost ukazuje na udeo procesorskog vremena koje se efektivno koristi u paralelnom algoritmu u odnosu na sekvencijalni algoritam i važi $0 < E(n, p) \leq 1$. Ako je $E(n, p) = 1$, onda je količina izračunavanja obavljenog na svim procesorima u toku izvršavanja algoritma jednaka količini izračunavanja koju zahteva sekvencijalni algoritam. U tom slučaju postignuto je optimalno iskorišćenje procesora. Postizanje optimalne efikasnosti je retko, jer se u paralelnim algoritmima moraju izvršiti neka dopunska izračunavanja, koja nisu potrebna kod sekvencijalnog algoritma. Prilikom razvoja paralelnih algoritama jedan od osnovnih ciljeva je maksimiziranje efikasnosti.

Princip imitiranja paralelizma

Pri konstrukciji paralelnog algoritma mogla bi se fiksirati vrednost p, u skladu sa brojem procesora na raspolaganju, i pokušati sa minimiziranjem vrednosti $T(n, p)$. Nedostatak ovakvog pristupa je u tome što bi on mogao da zahteva novi algoritam, kad god se promeni broj procesora. Pogodnije bi bilo konstruisati algoritam koji radi za što je moguće više različitih vrednosti p.

Razmotrimo na koji način transformisati algoritam koji radi za neku vrednost p, u algoritam za manju vrednost p, bez značajne promene efikasnosti. U opštem slučaju, paralelni algoritam sa vremenom izvršavanja $T(n,p) = X$ može se transformisati u paralelni algoritam sa vremenom izvršavanja $T(n,p/k) \simeq kX$, za proizvoljnu konstantu $k > 1$. Drugim rečima, može se koristiti za faktor k manje procesora, čije je vreme rada onda duže za faktor k. Modifikovani algoritam može se konstruisati zamenom svakog koraka polaznog algoritma sa k koraka, u kojima jedan procesor emulira (paralelno) izvršavanje jednog koraka na k procesora. Ovaj princip nije uvek primenljiv. Na primer, moguće je da p nije deljivo sa k, ili da algoritam zavisi od načina povezivanja procesora ili da donošenje odluke o tome koje procesore emulirati zahteva takođe utrošak nekog vremena. Ipak, ovaj princip, takozvani *princip imitiranja paralelizma*, vrlo je opšt i koristan. On pokazuje da se može smanjiti broj procesora, ne menjajući bitno efikasnost. Pokažimo da efikasnost ostaje ista. Važi:

$$\begin{aligned} E(n,p) &= \frac{T(n,1)}{p \cdot T(n,p)} = \frac{T(n,1)}{p \cdot X} \\ &\simeq \frac{T(n,1)}{\frac{p}{k} \cdot kX} = E(n, \frac{p}{k}) \end{aligned}$$

Ako polazni algoritam (koji je konstruisan za velike vrednosti p) ima veliko ubrzanje, onda se mogu dobiti algoritmi koji postižu približno isto ubrzanje za bilo koju manju vrednost p. Prema tome, treba konstruisati algoritam sa što boljim ubrzanjem za maksimalni broj procesora, pri čemu efikasnost treba da bude dobra (što bliskija jedinici). Zatim, ako je na raspolaganju manji broj procesora, i dalje se može koristiti isti algoritam. S druge strane, paralelni algoritmi sa malom efikasnošću su korisni samo ako je na raspolaganju veliki broj procesora.

Primer 1. Pretpostavimo da je za dati problem za ulaz veličine n na raspolaganju sekvencijalni algoritam sa vremenom izvršavanja $T(n,1) = n$ i paralelni algoritam za n procesora sa vremenom izvršavanja $T(n,n) = \log_2 n$. Njegovo ubrzanje jednako je

$$S(n) = \frac{n}{\log_2 n}$$

dok mu je efikasnost

$$E(n) = \frac{n}{n \log_2 n} = \frac{1}{\log_2 n}.$$

Neka je $n = 1024$: za sekvencijalni algoritam važiće $T(n,1) = 1024$. Pretpostavimo da nam je na raspolaganju $p_1 = 256$ procesora. Na osnovu principa imitiranja paralelizma važi da je vreme izvršavanja paralelnog algoritma

$$T(n,p_1) = T(1024,256) = T(1024,1024/4) = 4\log_2 1024 = 4 \cdot 10 = 40$$

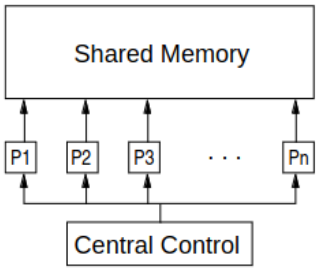
što je ubrzanje za faktor oko 25 u odnosu na sekvencijalni algoritam. S druge strane, za $p_2 = 16$ vreme izvršavanja paralelnog algoritma biće

$$T(n,p_2) = T(1024,16) = T(1024,1024/64) = 64\log_2 1024 = 640$$

što daje nedovoljno ubrzanje (manje od 2 sa 16 procesora). S obzirom na to da je paralelni algoritam imao malu efikasnost, on je kao što vidimo koristan samo ako je na raspolaganju veliki broj procesora.

Modeli paralelnih računara

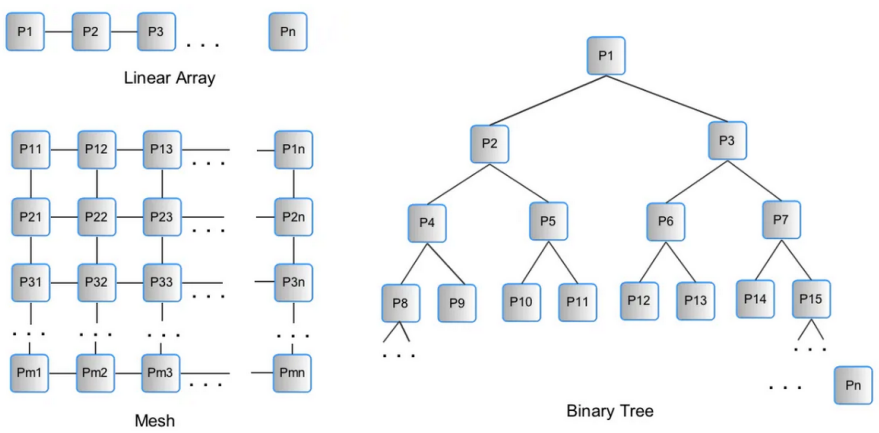
Modeli paralelnog izračunavanja međusobno se razlikuju uglavnom po načinu komunikacije i sinhronizacije procesora. Razmatraćemo samo modele koji podrazumevaju potpunu sinhronizaciju i različite načine povezivanja. *Modeli sa zajedničkom memorijom* pretpostavljaju da postoji zajednička memorija sa ravnomernim pristupom, tako da svaki procesor može da pristupi svakoj promenljivoj za jedinično vreme (slika 1). Ova pretpostavka o vremenu pristupa nezavisnom od broja procesora i veličine memorije nije baš realna, ali predstavlja prihvatljivu aproksimaciju. Modeli sa zajedničkom memorijom razlikuju se po načinu na koji obrađuju konflikte prilikom pristupa memoriji. Zajednička memorija je obično najjednostavniji način za modeliranje komunikacije, ali način koji je najteže hardverski realizovati.



Slika 1: Model paralelnog računara sa zajedničkom memorijom.

Drugi modeli pretpostavljaju da su procesori međusobno povezani posredstvom *mreže* (eng. network). Mreža računara se može predstaviti grafom, pri čemu čvorovi odgovaraju procesorima, a dva čvora su povezana granom ako između odgovarajućih procesora postoji direktna veza (slika 2). Svaki procesor obično ima lokalnu memoriju, kojoj može da pristupa brzo. Komunikacija između procesora se ostvaruje porukama, koje moraju da prođu više direktnih veza da bi došle do odredišta. Prema tome, brzina komunikacije zavisi od rastojanja između procesora koji razmenjuju poruke.

Još jedan mogući model paralelnih računara je model *sistoličkog računanja*. Sistolička arhitektura podseća na pokretnu traku u fabrici. Podaci se kreću kroz procesore ravnomerno, i tom prilikom se nad njima izvode jednostavne operacije. Umesto da pristupaju zajedničkoj (ili lokalnoj) memoriji, procesori dobijaju ulazne podatke od svojih suseda, obrađuju ih, i prosleđuju dalje.



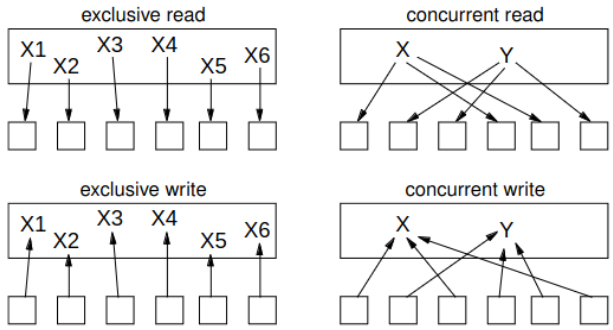
Slika 2: Primeri različitih mreža računara.

Algoritmi za računare sa zajedničkom memorijom

Računar sa zajedničkom memorijom sastoji se od više procesora i zajedničke memorije. U ovom odeljku bavićemo se samo potpuno sinhronizovanim algoritmima. Pretpostavljamo da se izračunavanje sastoji od koraka. U svakom koraku svaki procesor izvršava neku operaciju nad podacima kojima raspolaže, čita iz zajedničke memorije ili piše u zajedničku memoriju (u praksi svaki procesor može da ima i lokalnu memoriju).

Modeli sa zajedničkom memorijom razlikuju se po tome kako obrađuju memorijske konflikte (slika 3). Model EREW (Exclusive–Read Exclusive–Write) ne dozvoljava da dva procesora istovremeno pristupaju istoj memorijskoj lokaciji. Model CREW (Concurent–Read Exclusive–Write) dozvoljava da više procesora istovremeno čitaju sa iste memorijske lokacije, ali ne dozvoljava da dva procesora istovremeno pišu na istu lokaciju. Na kraju, model CRCW (Concurent–Read Concurent–Write) ne nameće nikakva ograničenja na pristup procesora memoriji.

Modeli EREW i CREW su dobro definisani, dok kod CRCW modela nije jasno šta je rezultat istovremenog pisanja od strane dva procesora na jednu istu memorijsku lokaciju. Ima više načina za obradu istovremenih pisanja. Najslabiji CRCW model, jedini koji će biti ovde razmatran, dozvoljava da više procesora istovremeno pišu na istu lokaciju samo ako zapisuju istu vrednost. Ako dva procesora pokušaju da upišu istovremeno različite vrednosti na istu lokaciju, prekida se sa izvršavanjem algoritma. Iako je to možda neočekivano, videćemo da je ovakav model vrlo moćan. Druga mogućnost je pretpostaviti da su procesori numerisani, i da, ako više procesora pokušaju istovremeni upis na istu lokaciju, realizuje se upis procesora sa najvećim rednim brojem.



Slika 3: Ilustracija različitih načina obrade memorijskih konflikata.

Algoritmi za nalaženje maksimuma

Problem. Pronaći najveći od n različitih brojeva, zadatih u nizu u zajedničkoj memoriji.

Ovaj problem rešićemo za dva različita modela sa zajedničkom memorijom: EREW i CRCW. Algoritmi za oba modela koriste tehnike koje se koriste pri rešavanju mnogih drugih problema.

Model EREW Direktni sekvencijalni algoritam za nalaženje maksimuma zahteva $n-1$ upoređivanja. Upoređivanje se može shvatiti kao partija koju igraju dva broja i u kojoj pobeđuje veći. Problem pronalaženja maksimuma je tada ekvivalentan organizovanju turnira, u kome je pobednik najveći broj u celom skupu. Efikasan način da se turnir organizuje paralelno jeste da se iskoristi drvo. Igrači se dele u parove za prvo kolo (pri čemu eventualno jedan igrač ne učestvuje, ako je ukupan broj igrača neparan), pobednici se ponovo dele u parove, i tako dalje do finala. Broj kola na turniru je $\lceil \log_2 n \rceil$.

Turnir se može transformisati u paralelni algoritam tako što se svakoj partiji u jednom kolu dodeli procesor (procesor igra ulogu sudije u partiji). Međutim, treba obezbediti da svaki procesor zna pozicije na kojima se nalaze njegovi "takmičari". To se može postići kopiranjem pobednika u partiji na poziciju sa većim indeksom od pozicija dva učesnika partije. Preciznije, ako partiju igraju x_i i x_j, $1 \leq i < j \leq n$, onda se veći od brojeva x_i, x_j kopira na poziciju j. U prvom kolu procesor P_i upoređuje x_{2i-1} sa x_{2i} $(1 \leq i \leq n/2)$, i zamenjuje ih ako je potrebno, tako da na veću poziciju ode veći broj. U drugom kolu procesor P_i upoređuje x_{4i-2} sa x_{4i} $(1 \leq i \leq n/4)$, i tako dalje. Ako je na primer $n = 2^k$, onda u poslednjem, k-tom kolu, P_1 upoređuje $x_{n/2}$ sa x_n, i eventualno ih zamenjuje. Nakon k-tog kola najveći broj nalaziće se na poziciji n. Pošto svaki broj u jednom trenutku učestvuje samo u jednoj partiji, dovoljan je model EREW. Vreme izvršavanja ovog jednostavnog algoritma je očigledno $O(\log n)$.

Algoritam koji smo upravo razmotrili zahteva $\lfloor n/2 \rfloor$ procesora, a njegova vremenska složenost je $T(n, \lfloor n/2 \rfloor) = \lceil \log_2 n \rceil$. Pošto za sekvencijalni algoritam za računanje maksimuma važi $T(n, 1) = n - 1$, efikasnost ovog paralelnog algoritma je $E(n, \lfloor n/2 \rfloor) = \frac{n-1}{\frac{n}{2} \log n} \simeq 2/\log_2 n$. Ako nam je na raspolaganju $\lfloor n/2 \rfloor$ procesora (na primer, ako je algoritam za nalaženje maksimuma deo drugog algoritma, kome je neophodno toliko procesora), onda je ovaj algoritam jednostavan i efikasan. Međutim, uz mali napor može se doći do algoritma sa vremenom izvršavanja $O(\log n)$ i efikasnošću $O(1)$.

Razmotrimo prethodni paralelni algoritam za računanje maksimuma niza. Ukupan broj upoređivanja potrebnih za ovaj algoritam je $n - 1$, što je isto kao i za sekvencijalni algoritam. Razlog njegove male efikasnosti leži u tome što se većina procesora ne koristi u kasnijim kolima. Efikasnost prethodnog algoritma se može poboljšati smanjivanjem broja procesora i uravnotežavanjem njihovog opterećenja na sledeći način. Pretpostavimo da koristimo samo oko $n/\log_2 n$ procesora. Ulaz se može podeliti u $n/\log_2 n$ grupa (sa približno $\log_2 n$ elemenata u svakoj grupi) i zatim svakoj grupi dodeliti po jedan procesor. U prvoj fazi svaki procesor pronalazi maksimum u svojoj grupi koristeći sekvencijalni algoritam, koji se sastoji od oko $\log_2 n$ koraka. Posle toga ostaje da se odredi maksimum otprilike $n/\log_2 n$ maksimuma, pri čemu sad ima dovoljno procesora da se iskoristi turnirski algoritam. Vreme izvršavanja turnira je $T(n, \lceil n/\log_2 n \rceil) \simeq 2 \log_2 n$, a efikasnost ovog algoritma je

$$E(n) = \frac{T(n,1)}{pT(n,p)} = \frac{n-1}{\frac{n}{\log n} \cdot 2 \log n} \simeq \frac{1}{2}$$

Pokušaćemo sada da formalizujemo ovu ideju, koja omogućuje uštedu na broju procesora.

Za algoritam kažemo da je *statički* ako se unapred zna pridruživanje procesora operacijama. Dakle, unapred znamo za svaki korak i algoritma i za svaki procesor P_j operaciju i argumente koje P_j koristi u koraku i. Algoritam za nalaženje maksimuma je primer statičkog algoritma, jer se unapred znaju indeksi učesnika u svakoj partiji.

Lema 1 (Brentova lema)**.** Ako postoji statički EREW algoritam sa $T(n, p) = O(t(n))$, takav da je ukupan broj koraka (na svim procesorima) $s(n)$, onda postoji statički EREW algoritam sa $s(n)/t(n)$ procesora za koji važi

$$T(n, s(n)/t(n)) = O(t(n))$$

Primetimo da ako je $s(n)$ jednako sekvencijalnoj složenosti algoritma (kao što je to bio slučaj u prethodnom algoritmu), onda modifikovani algoritam ima efikasnost $O(1)$.

Dokaz: Neka je $T(n, p) \leq t(n)$ za sve dovoljno velike n, i neka je a_i ukupan broj koraka koje izvršavaju svi procesori u i-tom koraku algoritma, $i = 1, 2, \ldots, t(n)$. Tada je $\sum_{i=1}^{t(n)} a_i = s(n)$. Pretpostavimo da nam je

na raspolaganju $p' = \frac{s(n)}{t(n)}$ procesora. Ako je $a_i \leq s(n)/t(n)$, onda je raspoloživ broj procesora p' dovoljan za paralelno izvršavanje koraka i. U protivnom se korak i zamenjuje sa $\lceil a_i/p' \rceil = \lceil a_i/(s(n)/t(n)) \rceil$ koraka u kojima raspoloživih $p' = s(n)/t(n)$ procesora emuliraju korake, koje u originalnom algoritmu izvršava p procesora (koristeći princip imitiranja paralelizma). Ukupan broj koraka je sada

$$\sum_{i=1}^{t(n)} \left\lceil \frac{a_i}{s(n)/t(n)} \right\rceil \leq \sum_{i=1}^{t(n)} \left(\frac{a_i t(n)}{s(n)} + 1 \right) = t(n) + \frac{t(n)}{s(n)} \sum_{i=1}^{t(n)} a_i = 2t(n).$$

Prema tome, vreme izvršavanja modifikovanog algoritma je takođe $O(t(n))$. □

Ovo tvrđenje poznato je kao *Brentova lema*. Brentova lema pokazuje da je pod određenim pretpostavkama efikasnost paralelnog algoritma određena odnosom ukupnog broja operacija (operacija koje izvršavaju svi procesori) i vremena izvršavanja sekvencijalnog algoritma.

Primetimo da je u primeru računanja maksimuma niza važilo:

$$s(n) = \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ldots = n$$

a vreme izvršavanja je jednako $t(n) = O(\log n)$.

Ograničenje da algoritam bude statički je potrebno, jer se mora znati koje procesore treba emulirati. Brentova lema je tačna i za algoritme koji nisu statički, pod uslovom da se emulacija može lako izvesti. Primer gde se ova lema ne može primeniti je sledeći. Pretpostavimo da imamo n procesora i n elemenata. Posle prvog koraka neki procesori odlučuju (na osnovu rezultata prvog koraka) da prestanu sa radom. Isto se dešava i drugom, trećem koraku, itd. Ovaj algoritam je sličan turnirskom algoritmu, izuzev što se u ovom slučaju ne zna koji procesori odustaju od daljeg rada. Ako pokušamo da emuliramo preostale procesore posle na primer prvog koraka, potrebno je da znamo koji su još aktivni. Međutim, ta informacija nije dostupna, već da bi se to ustanovilo, potrebno je izvršiti neka dodatna izračunavanja.

Model CRCW Nameće se utisak da paralelni algoritam ne može da nađe maksimum za manje od $\log_2 n$ koraka, ako se koriste samo upoređivanja. Međutim, to nije tačno. Sledeći algoritam sa vremenom izvršavanja $O(1)$ ilustruje mogućnosti istovremenih upisa. Podrazumeva se varijanta istovremenih upisa, u kojoj dva ili više procesora mogu da pišu istovremeno na istu lokaciju samo ako zapisuju isti podatak. Zbog jednostavnosti pretpostavićemo da su svi elementi međusobno različiti.

Koristi se $n(n-1)/2$ procesora, tako da se procesor P_{ij} dodeljuje paru elemenata $\{x_i, x_j\}$. Pored toga, svakom elementu x_i pridružuje se zajednička (deljena)

promenljiva v_i, sa početnom vrednošću 1. U prvom koraku svaki procesor upoređuje svoja dva elementa i zapisuje 0 u promenljivu pridruženu manjem elementu. Pošto je samo jedan element veći od svih ostalih, samo jedna od promenljivih v_i zadržava vrednost 1. U drugom koraku procesori pridruženi pobedniku mogu da ustanove da je on pobednik i da objave tu činjenicu (na primer, upisivanjem njegove vrednosti u posebnu zajedničku promenljivu, rezultat). Ovaj algoritam zahteva samo dva koraka, nezavisno od n. Međutim, njegova efikasnost je vrlo mala, jer on zahteva $O(n^2)$ procesora. Naime, važi:

$$E\Big(n, \frac{n(n-1)}{2}\Big) = \frac{n-1}{\frac{n(n-1)}{2} \cdot 1} = O\Big(\frac{1}{n}\Big)$$

Ovo je takozvani *dvokoračni algoritam.*

Efikasnost dvokoračnog algoritma može se poboljšati kao i algoritma za model EREW. Ulazni podaci dele se u male grupe, tako da se svakoj grupi može dodeliti dovoljno procesora, da bi se maksimum grupe mogao odrediti dvokoračnim algoritmom. Sa opadanjem broja kandidata raste broj raspoloživih procesora po kandidatu, pa se može povećati veličina grupe. Dvokoračni algoritam omogućuje određivanje maksimuma u grupi veličine k sa $k(k-1)/2$ procesora, za konstantno vreme.

i	preostali broj elemenata	veličina grupe g	broj grupa
1	n	2	$n/2$
2	$n/2$	4	$n/8$
3	$n/8$	16	$n/2^7$
...			
k	$n/2^{2^{k-1}-1}$	2^{2^k-1}	$n/2^{2^k-1}$

Tabela 1: Ilustracija tehnike podeli i smrvi kod problema maksimuma za model CRCW

Pretpostavimo da imamo ukupno n procesora i da je n stepen dvojke. U prvom ciklusu veličina svake grupe je 2 i maksimum u svakoj grupi može se odrediti u jednom koraku. U drugi ciklus ulazi se sa $n/2$ elemenata, i n procesora. Ako formiramo grupe od po 4 elementa, imaćemo $n/8$ grupa, što nam omogućuje da svakoj grupi dodelimo 8 procesora. Ovo je dovoljno, jer je $4 \cdot (4-1)/2 = 6$. U treći ciklus ulazi se sa $n/8$ elemenata. Pokušajmo da odredimo najveću moguću veličinu grupe koja se može obraditi na ovaj način. Ako je veličina grupe g, onda je broj grupa $n/8g$, i za svaku grupu imamo na raspolaganju $8g$ procesora. Za primenu dvokoračnog algoritma na grupu veličine g potrebno je $g(g-1)/2$ procesora, pa mora da bude $g(g-1)/2 \leq 8g$, odnosno $g \leq 17$; jednostavnije je uzeti vrednost $g = 16$. Uopšte, u i-ti ciklus se ulazi sa $n/2^{2^{i-1}-1}$ elemenata, koji se dele na $n/2^{2^i-1}$ grupa po $g = 2^{2^{i-1}}$ elemenata, $i \geq 1$. Za nalaženje maksimuma u grupi dvokoračnim algoritmom dovoljno je $g(g-1)/2 \leq g^2/2 = 2^{2^i-1}$ procesora,

pa je za nalaženje maksimuma u svim grupama dovoljno

$$\frac{n}{2^{2^i-1}} \cdot 2^{2^i-1} = n$$

procesora. U naredni ciklus ulazi se sa po jednim elementom iz svake grupe, dakle sa $n/2^{2^i-1}$ elemenata, što indukcijom dokazuje ispravnost ove konstrukcije (tabela 1). Ukupan broj ciklusa do završetka algoritma ograničen je uslovom da je broj elemenata na početku i-tog ciklusa manji od jedan: $n/2^{2^{i-1}-1} \leq 1$, ili $i \geq \log_2(\log_2 n + 1) + 1$. Dakle broj ciklusa, a time i broj koraka prilikom izvršenja ovog algoritma je $O(\log \log n)$.

Iako je ovaj algoritam nešto sporiji od dvokoračnog ($O(\log \log n)$ u odnosu na $O(1)$), njegova efikasnost je mnogo bolja. Ona iznosi $O(1/\log \log n)$ u odnosu na $O(1/n)$ kod dvokoračnog algoritma. Opisana tehnika naziva se *podeli i smrvi* (eng. divide-and-crush), jer se ulaz deli u grupe, koje su dovoljno male da se mogu "smrviti" mnoštvom procesora. Primena ove tehnike nije ograničena na model CRCW.

Paralelni problem prefiksa

Paralelni problem prefiksa je važan jer se koristi kao osnovni element pri konstrukciji mnogih paralelnih algoritama. Neka je $\star$ proizvoljna asocijativna binarna operacija (operacija koja zadovoljava uslov $x \star (y \star z) = (x \star y) \star z$) za proizvoljne x, y i z), koju ćemo označavati imenom proizvod. Na primer, $\star$ može da označava sabiranje, množenje ili maksimum dva broja.

Problem. Dat je niz brojeva $x_1, x_2, \ldots, x_n$. Izračunati proizvode $x_1 \star x_2 \star \cdots \star x_k$ za $k = 1, 2, \ldots, n$.

Označimo sa $PR(i, j)$ proizvod $x_i \star x_{i+1} \star \cdots \star x_j$. Potrebno je izračunati $PR(1, k)$ za $k = 1, 2, \ldots, n$. Sekvencijalna verzija problema prefiksa je trivijalna — prefiksi se jednostavno izračunavaju redom. Paralelni problem prefiksa nije tako lako rešiti. Iskoristićemo metod dekompozicije, uz uobičajenu pretpostavku da je n stepen dvojke.

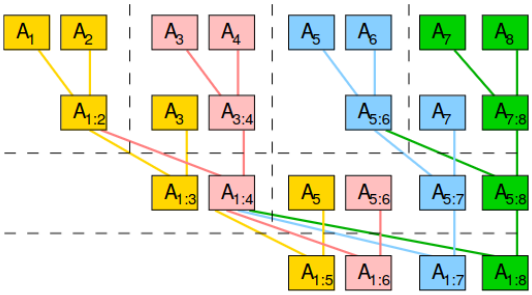
Induktivna hipoteza. Umemo da rešimo paralelni problem prefiksa za $n/2$ elemenata.

Slučaj jednog elementa je trivijalan. Algoritam započinje podelom ulaza na dve polovine, koje po induktivnoj hipotezi umemo da rešimo. Na taj način dobijamo vrednosti $PR(1, k)$ i $PR(n/2 + 1, n/2 + k)$ za $k = 1, 2, \ldots, n/2$. Prva polovina ovih vrednosti deo je konačnog rezultata. Vrednosti $PR(1, m)$ za $m = n/2 + 1, n/2 + 2, \ldots, n$ dobijaju se izračunavanjem proizvoda

$$PR(1, m) = PR(1, n/2) \star PR(n/2 + 1, m)$$

Oba ova člana poznata su po indukciji (već su izračunata; primetimo da je ovde iskorišćena asocijativnost operacije $\star$). Na primer, prilikom rešavanja dva potproblema veličine 2 dobijamo vrednosti: $PR(1, 1) = x_1$ i $PR(1, 2) = x_1 \star x_2$ i

$PR(3,3) = x_3$ i $PR(3,4) = x_3 \star x_4$. Kako bismo dobili sve prefikse potrebno je još pomnožiti $PR(3,3)$ i $PR(3,4)$ sa $PR(1,2)$ (slika 4).



Slika 4: Paralelno računanje prefiksa.

Algoritam je prikazan na slici 5. Činjenica da se prolasci kroz **do** petlju izvršavaju paralelno (istovremeno, na različitim skupovima procesora) u kodu je naznačena dodatkom `in parallel`.

Složenost. Ulaz je podeljen u dva disjunktna skupa u svakom rekurzivnom pozivu algoritma. Oba potproblema se mogu dakle rešiti paralelno u modelu EREW. Ako imamo n procesora za problem veličine n, onda se polovina njih može dodeliti svakom potproblemu. Kombinovanje rešenja potproblema sastoji se od $n/2$ množenja, koja se mogu izvršiti paralelno, ali je potreban model CREW, jer se u svakom množenju koristi $PR(1, n/2)$, odnosno $x[Srednji]$. Iako više procesora istovremeno čitaju $x[Srednji]$, oni pišu na različite lokacije, pa model CRCW nije neophodan. Vreme izvršavanja algoritma opisano je rekurentnom jednačinom:

$$T(n,n) = T(n/2, n/2) + O(1)$$

odakle sledi $T(n,n) = O(\log n)$. S obzirom na to da je vreme izvršavanja sekvencijalnog algoritma $O(n)$ efikasnost algoritma je $E(n,n) = O(1/\log n)$.

Ukupan broj koraka na svim procesorima može se opisati rekurentnom jednačinom

$$s(n) = 2s(n/2) + O(n), \;\; s(2) = 1$$

čije je rešenje $s(n) = O(n \log n)$. Stoga se efikasnost ovog algoritma ne može poboljšati korišćenjem Brentove leme. Da bi se poboljšala efikasnost, mora se smanjiti ukupan broj koraka.

Poboljšanje efikasnosti paralelnog prefiksa Ideja koja omogućuje efikasnije rešavanje ovog problema je korišćenje iste induktivne hipoteze, ali uz podelu ulaza na drugačiji način. Pretpostavimo ponovo da je n stepen dvojke i da imamo n procesora. Neka E označava skup svih elemenata x_i sa parnim indeksima i. Ako izračunamo prefikse svih elemenata iz E, onda je izračunavanje ostalih

```
Algoritam Paralelni_Prefiks_1(x, n);
Ulaz: x (niz sa n elemenata).
 {pretpostavlja se da je n stepen dvojke}
Izlaz: x (niz čiji i-ti element sadrži i-ti prefiks).
begin
    PP_1(1, n)
end
procedure PP_1(Levi, Desni);
begin
    if Desni − Levi = 1 then
        x[Desni] := x[Levi] ⋆ x[Desni] {⋆ je asocijativna binarna operacija}
    else
        Srednji := (Levi + Desni − 1)/2;
        do in parallel
            PP_1(Levi, Srednji); {dodeljeno procesorima od 1 do n/2}
            PP_1(Srednji + 1, Desni); {dodeljeno procesorima od n/2 + 1 do n}
        for i := Srednji + 1 to Desni do in parallel
            x[i] := x[Srednji] ⋆ x[i]
end
```

Slika 5: Algoritam $Paralelni\_prefiks\_1$.

prefiksa (onih sa neparnim indeksima) lako: ako je poznato $PR(1, 2i)$, onda se neparni prefiks $PR(1, 2i + 1)$ dobija izračunavanjem samo još jednog proizvoda

$$PR(1, 2i + 1) = PR(1, 2i) \star x_{2i+1}, \quad i = 1, 2, \ldots, n/2 - 1.$$

Prefiksi elemenata iz E mogu se odrediti u dve faze. Najpre se (paralelno) izračunavaju proizvodi $x_{2i-1} \star x_{2i}$, koji se zatim smeštaju u x_{2i}, $i = 1, 2, \ldots, n/2$. Drugim rečima, izračunavaju se proizvodi svih elemenata iz E sa svojim levim susedima. Zatim se rešava (indukcijom) problem paralelnog prefiksa za $n/2$ elemenata iz E. Rezultat za svako x_{2i} je tačan konačni prefiks, jer je svako x_{2i} već zamenjeno proizvodom sa x_{2i-1}. Pošto se znaju prefiksi za sve elemente sa parnim indeksima, preostali prefiksi se mogu izračunati u jednom paralelnom koraku na već spomenuti način. Lako se proverava da se ovaj algoritam može izvršavati u modelu EREW. Algoritam je prikazan na slici 6.

Složenost. Obe petlje u algoritmu `Paralelni_prefiks_2` mogu se izvršiti paralelno za vreme $O(1)$ sa $n/2$ procesora. Rekurzivni poziv primenjuje se na problem dvostruko manje veličine, pa je vreme izvršavanja algoritma opisano rekurentnom jednačinom:

$$T(n, n) = T(n/2, n/2) + O(1)$$

```
Algoritam Paralelni_Prefiks_2(x, n);
Ulaz: x (niz sa n elemenata).
 {pretpostavlja se da je n stepen dvojke}
Izlaz: x (čiji i-ti elemenat sadrži i-ti prefiks).
begin
    PP_2(1)
end
procedure PP_2(Korak);
begin
    if Korak = n/2 then
        x[n] := x[n/2] ⋆ x[n] {⋆ je asocijativna binarna operacija}
    else
        for i := 1 to n/(2 · Korak) do in parallel
            x[2 · i · Korak] := x[(2 · i − 1) · Korak] ⋆ x[2 · i · Korak];
        PP_2(2 · Korak);
        for i := 1 to n/(2 · Korak) − 1 do in parallel
            x[(2 · i + 1) · Korak] := x[2 · i · Korak] ⋆ x[(2 · i + 1) · Korak]
end
```

Slika 6: Algoritam $Paralelni\_prefiks\_2$.

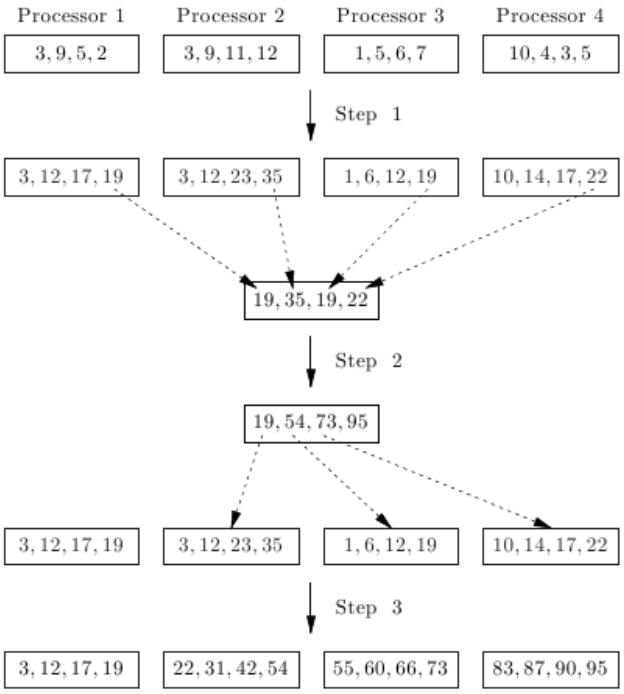
odnosno iznosi $O(\log n)$. Efikasnost algoritma jednaka je:

$$E(n,n) = \frac{n}{\frac{n}{2} \cdot \log n} = O\Big(\frac{1}{\log n}\Big)$$

Ukupan broj koraka u prethodnom algoritmu $s(n)$ zadovoljava rekurentnu jednačinu

$$s(n) = s(n/2) + n - 1,\ s(2) = 1$$

iz čega sledi da je $s(n) = O(n)$. Zbog toga se sada može iskoristiti Brentova lema za poboljšanje efikasnosti: algoritam se može promeniti tako da se za vreme $O(\log n)$ izvršava na samo $O(n/\log n)$ procesora, odnosno da mu efikasnost bude $O(1)$. Ideja na kojoj se zasniva ovaj algoritam jeste da se ulaz podeli na blokove od $\log n$ elemenata, tako da se svakom bloku dodeli po jedan procesor – ukupan broj potrebnih procesora za ovaj korak jeste $n/\log n$. Svaki od procesora može da primeni sekvencijalni algoritam za računanje prefiksa nad svojim elementima. Nakon toga se razmatra $n \log n$ elemenata – poslednji iz svakog bloka i nad ovim elementima se primenjuje paralelni algoritam za računanje prefiksa. Nakon toga se u dodatnih $\log n$ paralelnih koraka može svaki od elemenata iz nekog bloka pomnožiti najvećim elementom iz prethodnog bloka. Na ovaj način izračunavaju se konačne vrednosti prefiksa za svaki element niza (slika 7).



Slika 7: Ilustracija paralelnog računanja prefiksa u EREW modelu sa $n/\log n$ procesora za operaciju sabiranja.

Određivanje rangova u povezanoj listi

U paralelnim algoritmima mnogo je teže raditi sa povezanim listama nego sa nizovima, jer su liste suštinski sekvencijalne. Povezanoj listi može se pristupiti samo preko glave (prvog elementa), i lista se mora prolaziti element po element, bez mogućnosti paralelizacije. U mnogim slučajevima su, međutim, elementi liste (odnosno pokazivači na njih) smešteni u niz; pritom je redosled elemenata liste nezavisan od redosleda u nizu. Dakle, mi znamo lokacije elemenata liste, ali ne znamo njihov poredak, koji je poznat samo implicitno putem pokazivača samih elemenata liste. U takvim slučajevima, kad se listi pristupa paralelno, postoji mogućnost primene brzih paralelnih algoritama.

Rang elementa u povezanoj listi definiše se kao rastojanje elementa od kraja liste. Tako, na primer, prvi element ima rang n, drugi $n-1$, itd, a poslednji element rang 1.

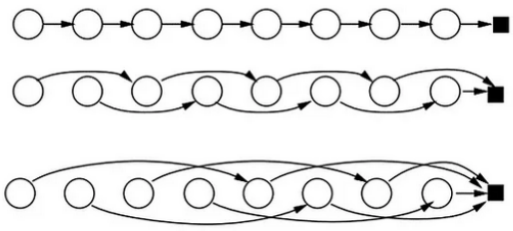
Problem. Data je povezana lista od n elemenata koji su smešteni u niz A dužine n. Izračunati rangove svih elemenata liste.

Posle izračunavanja svih rangova, elementi se mogu paralelno prekopirati na svoje lokacije u nizu, pa se ostatak izračunavanja može izvršiti direktno na nizu,

posle čega je njihova obrada mnogo jednostavnija.

Sekvencijalni algoritam za rešavanje ovog problema bi podrazumevao prosti prolazak kroz listu i bio bi linearne vremenske složenosti. Metod koji ćemo iskoristiti za konstrukciju paralelnog algoritma zove se *udvostručavanje* (eng. path doubling, pointer jumping) i omogućava praćenje putanja u strukturama podataka koje su bazirane na pokazivačima (povezane liste, usmereni grafovi) u vremenskoj složenosti koja je logaritamska u odnosu na dužinu najdužeg puta.

Svakom elementu niza dodeljuje se po jedan procesor. Na početku svaki procesor zna samo adresu desnog (narednog) suseda svog elementa u listi. Posle prvog koraka svaki procesor zna element na rastojanju dva (duž liste) od svog elementa. Ako u koraku i svaki procesor zna adresu elementa na rastojanju k od svog elementa, onda u narednom koraku svaki procesor može da sazna adresu elementa na rastojanju $2k$. Proces se nastavlja sve dok svi procesori ne dosegnu kraj liste.



Slika 8: Tehnika udvostručavanja za računanje rangova elemenata u listi.

Neka je $N[i]$ adresa najudaljenijeg elementa desno od elementa i u listi, koju zna procesor P_i. Na početku je $N[i]$ desni sused elementa i (izuzev za poslednji element u listi, čiji je pokazivač na desnog suseda **nil**). U suštini, procesor P_i u svakom koraku zamenjuje $N[i]$ vrednošću $N[N[i]]$, sve dok ne dostigne kraj liste. Neka je $R[i]$ rang elementa i. Na početku se promenljivoj $R[i]$ dodeljuje vrednost 0, izuzev za poslednji element u listi, za koga se ona postavlja na vrednost 1 (ovaj element se od ostalih razlikuje po pokazivaču $N[i]$, koji inicijalno ima vrednost **nil**). Kad procesor dobije adresu nekog desnog suseda sa rangom R različitim od nule, on može da izračuna svoj rang (odnosno rang svog elementa).

Naime, na početku samo element ranga 1 zna svoj rang. Posle prvog koraka element ranga 2 otkriva da njegov sused ima rang 1, pa zaključuje da je njegov sopstveni rang 2. Posle drugog koraka elementi sa rangom 3 i 4 ustanovljavaju svoje rangove, itd. Ako P_i ustanovi da $N[i]$ pokazuje na "rangirani" element ranga R posle d koraka udvostručavanja, onda je rang elementa i jednak $2^{d-1}+R$. Ovaj algoritam (slika 9) se može lako prilagoditi modelu EREW, dovoljno je da svaki procesor nezavisno izračunava svoju kopiju $D[i]$ promenljive D.

Složenost. Proces udvostručavanja omogućuje da svaki procesor dostigne kraj liste posle najviše $\lceil \log_2 n \rceil$ koraka, pa je $T(n,n) = O(\log n)$. Efikasnost algoritma

```
Algoritam Rangovi(N);
Ulaz: N (niz od n elemenata).
Izlaz: R (rangovi svih elemenata u nizu).
begin
    D := 1;
    {Svaki procesor može imati svoju lokalnu promenljivu D}
    {ovde je D zajednička promenljiva}
    do in parallel {procesor P_i je aktivan dok R[i] ne postane različito od nule}
        R[i] := 0;
        if N[i] = nil then R[i] := 1;
        while R[i] = 0 do
            if R[N[i]] ≠ 0 then
                R[i] := D + R[N[i]]
            else
                N[i] := N[N[i]];
                D := 2 · D
end
```

Slika 9: Paralelni algoritam za određivanje rangova elemenata povezane liste.

je $E(n,n) = \frac{n-1}{n \log n} = O(1/\log n)$. Popravka efikasnosti algoritma zahteva suštinski drugačiji pristup rešavanju.

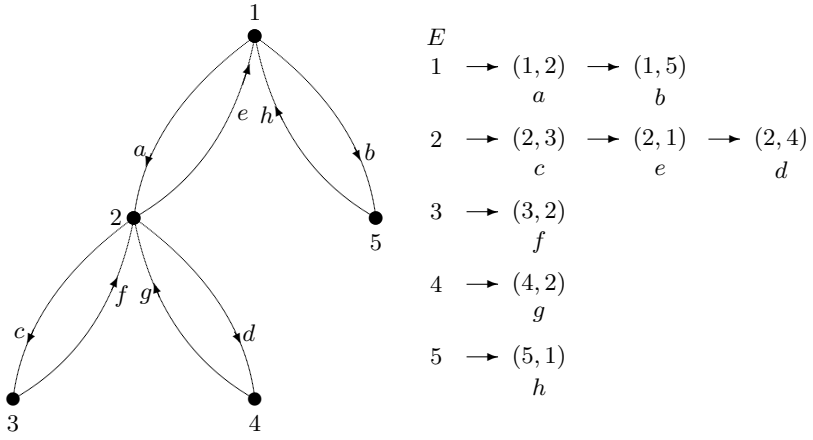
Na ovaj način može se, na primer, za svaki element liste izračunati suma sufiksa počev od te pozicije u listi.

Tehnika Ojlerovog obilaska

Mnogi algoritmi za rad sa drvetima mogu se paralelizovati tako da se paralelno obrađuje kompletna generacija čvorova (na primer, kod turnirskog algoritma za nalaženje maksimuma). Vreme izvršavanja takvog algoritma je proporcionalno visini drveta. Ako je drvo sa n čvorova dovoljno uravnoteženo i visina mu je $O(\log n)$, onda je ovaj pristup sasvim dobar. Međutim, ako drvo nije uravnoteženo, visina drveta može da bude u najgorem slučaju $n-1$, pa se mora tražiti neki drugi pristup. *Tehnika Ojlerovog obilaska* (eng. Euler tour technique) je alatka za konstrukciju paralelnih algoritama na drvetima, pogodna i u slučaju kada je drvo neuravnoteženo.

Neka je T neusmereno drvo. Pretpostavimo da je T predstavljeno listom povezanosti E, uz jednu dopunu. Kao i obično, postoji pokazivač $E[i]$ na početak liste grana susednih čvoru i (ako je ova lista prazna, onda $E[i]$ ima vrednost **nil**). Ta lista sastoji se od slogova koji sadrže odgovarajuću granu (i,j) (pri tome je dovoljno smestiti samo j, jer je i poznato) i pokazivač $Naredna(i,j)$ na narednu granu u listi. Svaka neusmerena grana (i,j) predstavljena je sa dve usmerene kopije: (i,j) i (j,i). Elementi liste sadrže i dodatni pokazivač: čvor

liste koji odgovara grani (i, j) sadrži pokazivač na granu (j, i). Ovo je potrebno da bi se grana (j, i) mogla brzo pronaći kad se zna adresa grane (i, j). Primer ovako predstavljenog drveta dat je na slici 10; pokazivači na kopije grana zbog preglednosti nisu prikazani.



Slika 10: Reprezentacija drveta.

Tehnika Ojlerovog obilaska zasniva se na ideji da se formira lista grana drveta, i to onim redom kojim se grane pojavljuju u Ojlerovom ciklusu za usmerenu verziju drveta (u ciklusu se svaka neusmerena grana pojavljuje dva puta, po jednom u oba smera). Kad se zna ova lista, mnoge operacije sa drvetom mogu se izvesti direktno na listi, kao da je lista linearna. Sekvencijalnim algoritmom lako se može pregledati drvo i usput izvršiti potrebne operacije. Ovakva "linearizacija" omogućuje da se operacije sa drvetom efikasno izvode paralelno. Videćemo dva primera takvih operacija, pošto prethodno razmotrimo formiranje Ojlerovog ciklusa.

Sekvencijalno nalaženje Ojlerovog obilaska drveta T (u kome se svaka grana pojavljuje dva puta) je jednostavno. Može se izvesti obilazak drveta koristeći pretragu u dubinu, vraćajući se suprotno usmerenom granom prilikom svakog povratka nazad u toku pretrage. Slična stvar se može izvesti i paralelno. Neka $w(i, j)$ označava granu koja sledi iza grane (i, j) u ciklusu. Ispostavlja se da se $w(i, j)$ može definisati sledećom jednakošću

$$w(i,j) = \begin{cases} Naredna(j,i) & \text{ako } Naredna(j,i) \text{ nije } \textbf{nil} \\ E[j] & \text{u ostalim slučajevima} \end{cases},$$

na osnovu koje se lako paralelno izračunava. Drugim rečima, lista grana susednih čvoru j prolazi se cikličkim redosledom (ako je (j, i) poslednja grana u listi čvora j, onda se uzima prva grana iz te liste, ona na koju pokazuje $E[j]$). Na primer, ako krenemo od grane a na slici 10, onda se ciklus sastoji od grana a, d, g, c, f, e, b, h, i ponovo a. Istaknimo to da činjenica da $Naredna(j, i)$ sledi iza (i, j) u ciklusu obezbeđuje da će grana (j, i) doći na red tek posle prolaska svih grana

susednih čvoru j. Na taj način se postiže da će poddrvo sa korenom u j biti kompletno pregledano pre povratka u čvor i.

Kad je lista grana u Ojlerovom obilasku konstruisana, proizvoljna grana (r, t) bira se za polaznu, a grana koja joj prethodi označava se kao kraj liste. Čvor r se bira za koren drveta. Posle toga se grane algoritmom *Rangovi* sa slike 9 mogu numerisati, u skladu sa svojim položajem u listi. Neka $R(i, j)$ označava rang grane (i, j) u listi. Tako je, na primer, $R(r, t) = 2(n - 1)$, gde je n broj čvorova.

Primetimo da kada imamo mogućnost da proizvoljno drvo transformišemo u listu koja odgovara Ojlerovom ciklusu, možemo na listi efikasno, u složenosti $O(\log n)$, rešiti problem paralelnog prefiksa. Pogodnim definisanjem asocijativne operacije i vrednosti za svaki čvor liste (tj. vrednosti za svaku granu u Ojlerovom ciklusu) možemo efikasno rešiti različite probleme nad drvetom.

Prikazaćemo sada dva primera operacija sa drvetom — dolaznu numeraciju čvorova, i izračunavanje broja potomaka za sve čvorove.

Računanje dolazne numeracije čvorova u drvetu Za granu (i, j) u ciklusu kažemo da je *direktna grana* ako je usmerena od korena, odnosno da je *povratna grana* u protivnom. Numeracija čvorova omogućuje razlikovanje direktnih od povratnih grana: grana (i, j) je direktna grana ako i samo ako je $R(i, j) > R(j, i)$. Pošto su dve kopije grane (i, j) povezane pokazivačima, lako je ustanoviti koja je od njih direktna grana. Šta više, ova provera se može obaviti paralelno za sve grane. Direktne grane su interesantne, jer određuju redosled čvorova pri dolaznoj numeraciji. Neka je (i, j) direktna grana koja vodi do čvora j (odnosno, čvor i je otac čvora j u drvetu). Ako je $f(i, j)$ broj direktnih grana koje slede iza (i, j) u listi, onda je dolazna numeracija čvora j jednaka $n - f(i, j)$. Vrednost dolazne numeracije korena r, jedinog čvora do koga ne vodi ni jedna direktna grana, je 1. Primenom varijante algoritma sa udvostručavanjem može se izračunati vrednost $f(i, j)$ za svaku direktnu granu (i, j). Precizniju razradu algoritma ostavljamo čitaocu kao vežbanje.

Primetimo da smo ovaj problem mogli videti i kao računanje prefiksne sume gde je vrednost svake direktne grane 1, a povratne 0.

Računanje broja potomaka čvorova u drvetu Drugi primer je izračunavanje broja potomaka svakog čvora u drvetu. Neka je (i, j) (jedinstvena) direktna grana koja vodi do zadatog čvora j. Posmatrajmo grane koje slede iza grane (i, j) u listi. Broj čvorova ispod j u drvetu jednak je broju direktnih grana ispod j u drvetu. Mi već znamo kako se paralelno izračunavaju vrednosti $f(i, j)$, jednake broju direktnih grana koje slede iza grane (i, j) u listi. Na sličan način $f(j, i)$ je broj direktnih grana koje slede iza grane (j, i) u listi. Lako je videti da je broj potomaka čvora j jednak $f(i, j) - f(j, i)$. Vreme izvršavanja oba opisana algoritma na modelu EREW je $T(n, n) = O(\log n)$.

Računanje dubine čvora u drvetu Razmotrimo problem računanja dubine svakog čvora u drvetu, odnosno dužine najkraćeg puta od korena do datog čvora. Specijalno, dubina korena je 0. Ovaj problem možemo rešiti primenom algoritma paralelnog prefiksa pri čemu je svakoj direktnoj grani dodeljena vrednost 1, a svakoj povratnoj grani vrednost -1.

Algoritmi za mreže računara

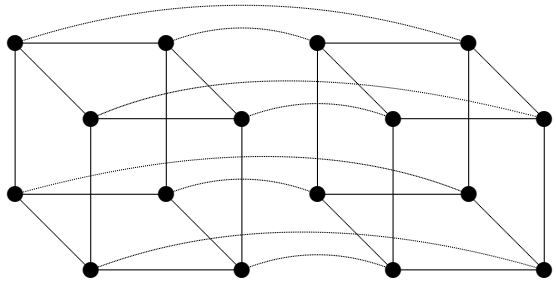
Mreže računara mogu se modelirati grafovima, obično neusmerenim. Procesori odgovaraju čvorovima, a dva čvora su povezana granom ako postoji direktna veza između odgovarajućih procesora. Svaki procesor ima svoju lokalnu memoriju, a posredstvom mreže može da pristupi lokalnim memorijama drugih procesora. Prema tome, sva memorija je na neki način zajednička, ali cena pristupa nekoj promenljivoj zavisi od lokacija procesora i promenljive. Pristup željenoj promenljivoj može biti brz koliko i lokalni pristup (ako je promenljiva u istom procesoru), ili toliko spor koliko i prolazak kroz celu mrežu (u slučaju kad graf ima oblik niza povezanih čvorova). Trajanje pristupa je negde između ove dve krajnosti. Procesori komuniciraju razmenom poruka. Kad procesor treba da pristupi promenljivoj smeštenoj u lokalnoj memoriji drugog procesora, on šalje poruku sa zahtevom za promenljivom. Poruka se usmerava kroz mrežu.

Više različitih grafova koriste se za mreže računara. Najjednostavniji među njima su linearni niz, prsten, binarno drvo, zvezda i dvodimenzionalna mreža. Efikasnost komunikacije raste sa brojem grana u mreži. Međutim, grane su skupe — to se može objasniti povećanjem površine koju zauzimaju veze, a time povećanjem dimenzija mreže i vremena komunikacije. Zbog toga se obično traži kompromis. Ne postoji tip grafa koji je univerzalno dobar. Pogodnost određenog grafa bitno zavisi od načina komuniciranja u okviru konkretnog algoritma. Međutim, postoje neke osobine grafova, koje mogu biti vrlo korisne za više različitih algoritama. U nastavku ćemo ih navesti, kao i primere mreža računara.

Bitan parametar mreže je *dijametar* odgovarajućeg grafa, tj. najveće rastojanje neka dva čvora u mreži. Dijametar određuje maksimalni broj grana koji poruka treba da prođe na putu do odredišta. Dvodimenzionalna mreža $n \times n$ ima dijametar $2n$, a kompletno binarno drvo sa n čvorova ima dijametar $2\log_2(n+1)-2$. Prema tome može da isporuči poruku mnogo brže nego dvodimenzionalna mreža. S druge strane, drvo ima usko grlo, jer sav saobraćaj iz jedne u drugu polovinu drveta prolazi kroz koren. Dvodimenzionalna mreža nema usko grlo i vrlo je simetrična, što je važno za algoritme u kojima je komunikacija simetrična.

Hiperkocka je popularna struktura koja kombinuje prednosti visoke simetrije, malog dijametra, mnoštva alternativnih puteva između dva čvora i odsustva uskih grla. d-dimenzionalna hiperkocka sastoji se od $n = 2^d$ procesora. Adrese procesora su d-torke brojeva iz skupa $\{0, 1\}$ (koje se mogu kodirati brojevima od 0 do $2^d - 1$). Prema tome, svaka adresa se sastoji od d bitova. Procesor P_i je povezan sa procesorom P_j ako i samo ako se binarni zapis i razlikuje

od binarnog zapisa j na tačno jednom bitu. Rastojanje između proizvoljna dva procesora je uvek manje ili jednako od d, jer se od P_i do P_j može doći promenom najviše d bitova, jednog po jednog. Na slici 11 prikazana je četvorodimenzionalna hiperkocka. Hiperkocka obezbeđuje bogatstvo veza, jer postoji mnogo različitih puteva između svaka dva procesora (odgovarajući biti mogu se menjati proizvoljnim redosledom). Hiperkocka se može takođe kombinovati sa nekom drugom mrežom, na primer umećući mreže umesto temena hiperkocke. U primeni se pojavljuju i druge strukture mreža.

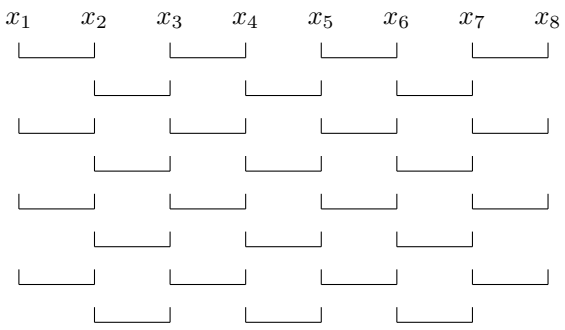


Slika 11: Četvorodimenzionalna hiperkocka.

Sortiranje na nizu

Razmotrićemo najpre jednostavan problem sortiranja na nizu procesora. Na raspolaganju je n procesora $P_1, P_2, \ldots, P_n$ i zadato je n brojeva $x_1, x_2, \ldots, x_n$. Svaki procesor čuva jedan ulazni podatak. Cilj je preraspodeliti brojeve među procesorima tako da najmanji od njih bude u P_1, sledeći u P_2, itd. U opštem slučaju moguće je dodeljivanje više ulaznih podataka jednom procesoru. Videćemo da se algoritam može prilagoditi i takvim uslovima. Procesori su povezani u linearni niz: procesor P_i je povezan sa procesorom P_{i+1}, $i = 1, 2, \ldots, n-1$. Pošto svaki procesor može da komunicira samo sa susedima, upoređivanje i razmenu podataka moguće je vršiti samo između elemenata koji su susedni u nizu. U najgorem slučaju algoritam zahteva izvršavanje $n-1$ koraka, koliko je potrebno da se podatak premesti sa jednog na drugi kraj niza. Algoritam se u osnovi izvršava u duhu bubble sort algoritma. Svaki procesor upoređuje svoj broj sa brojem jednog od svojih suseda, razmenjuje brojeve ako je njihov redosled pogrešan, a zatim isti posao obavlja sa drugim susedom (susedi se moraju smenjivati, jer bi se u protivnom upoređivali uvek isti brojevi). Isti proces nastavlja se sve dok brojevi ne budu poređani na željeni način. Koraci se dele na neparne i parne. U neparnim koracima procesori sa neparnim indeksom upoređuju svoje sa brojevima svojih desnih suseda; u parnim koracima procesori sa parnim indeksom upoređuju svoje sa brojevima svojih desnih suseda (slika 12). Na taj način su svi procesori sinhronizovani i upoređivanje uvek vrše procesori koji to i treba da rade. Ako procesor nema odgovarajućeg suseda (na primer prvi procesor u drugom koraku), on u toku tog koraka miruje. Ovaj algoritam se zove

sortiranje parno-neparnim transpozicijama (slika 13). Primer rada algoritma prikazan je na slici 14. Primetimo da se u ovom primeru sortiranje završava posle samo šest koraka. Ipak, raniji završetak teško je otkriti u mreži. Prema tome, bolje je ostaviti algoritam da se izvršava do svog završetka u najgorem slučaju.



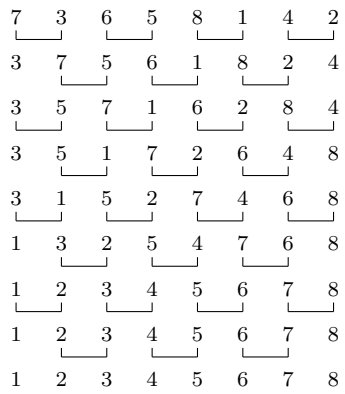
Slika 12: Sortiranje parno-neparnim transpozicijama.

Algoritam Sortiranje_na_nizu(x, n);
Ulaz: x (niz sa n elemenata, pri čemu je x_i u procesoru P_i).
Izlaz: x (sortirani niz, tako da je i-ti najmanji element u P_i).
begin
  do in parallel $\lceil n/2 \rceil$ puta
    P_{2i-1} i P_{2i} upoređuju svoje elemente i po potrebi ih razmenjuju;
      {za sve i, takve da je $1 < 2i \leq n$}
    P_{2i} i P_{2i+1} upoređuju svoje elemente i po potrebi ih razmenjuju;
      {za sve i, takve da je $1 \leq 2i < n$}
      {ako je n neparno, ovaj korak se izvršava samo $\lfloor n/2 \rfloor$ puta}
end

Slika 13: Algoritam za sortiranje na nizu procesora.

Algoritam `Sortiranje_na_nizu` izgleda prirodno i jasno, ali dokaz ispravnosti njegovog rada nije trivijalan. Na primer, element se u nekim koracima može udaljavati od svog konačnog položaja. U primeru na slici 14 broj 5 se kreće ulevo dva koraka pre nego što počne sa kretanjem udesno, a 3 ide do levog kraja i ostaje tamo tri koraka pre nego što krene nazad udesno. Dokaz ispravnosti rada paralelnih algoritama nije jednostavan, zbog međuzavisnosti delovanja različitih procesora. Ponašanje jednog procesora utiče na sve ostale procesore, pa je obično teško usredsrediti se na jedan procesor i dokazati da je to što on radi ispravno; moraju se posmatrati svi procesori zajedno.

Teorema 1. Na kraju izvršavanja algoritma `Sortiranje_na_nizu` dati brojevi su sortirani.



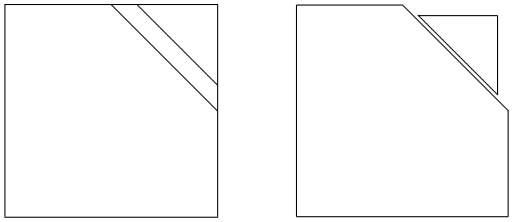
Slika 14: Primer sortiranja parno-neparnim transpozicijama.

Dokaz: Dokaz se izvodi indukcijom po broju elemenata. Pri tome se tvrđenje malo pojačava: sortiranje se završava posle n koraka, bez obzira da li je prvi korak paran ili neparan. Za $n = 2$ sortiranje se završava posle najviše dva koraka; sortiranje traje dva koraka ako je prvi korak neparan. Pretpostavimo da je teorema tačna za n procesora, i posmatrajmo slučaj $n+1$ procesora. Skoncentrišimo pažnju na maksimalni element, i pretpostavimo da je to x_m (u primeru na slici 14 to je x_5). U prvom koraku x_m se upoređuje sa x_{m-1} ili x_{m+1}, zavisno od toga da li je m parno ili neparno. Ako je m parno, nema zamene jer je x_m veće od x_{m-1}. Ishod je isti kao u slučaju da je broj x_m na početku bio u procesoru P_{m-1}, i da je zamena bila izvršena. Prema tome, bez gubitka opštosti može se pretpostaviti da je m neparno. U tom slučaju broj x_m se upoređuje sa x_{m+1}, zamenjuje, i kao najveći se premešta korak po korak udesno (dijagonalno na slici 14), sve dok ne dođe na mesto x_{n+1}, a onda ostaje tamo. To je pozicija koju x_m i treba da zauzme, pa sortiranje ispravno tretira maksimalni element.

Pokazaćemo sada indukcijom da je i sortiranje ostalih n elemenata korektno. Posmatrajmo dijagonalu nastalu kretanjem maksimalnog elementa (videti sliku 15). Upoređivanja u kojima učestvuje maksimalni element se ignorišu. Upoređivanja delimo na dve grupe, ona ispod, i ona iznad dijagonale.

Zatim "transliramo" trougao iznad dijagonale za jedno polje naniže i jedno polje ulevo. Drugim rečima, za upoređivanja u gornjem trouglu korak i se sada zove $i + 1$. Na primer, posmatrajmo upoređivanja 1 sa 8 i 4 sa 2 u prvom koraku na slici 14. Prvo upoređivanje je na dijagonali, pa se ignoriše; drugo je iznad dijagonale, pa se smatra delom koraka 2, umesto koraka 1. Prema tome, korak 2 sastoji se od upoređivanja 7 sa 5, 6 sa 1, i 4 sa 2. Međutim, ovo je regularni parni korak sa samo n elemenata. Korak 1 i sva upoređivanja u kojima učestvuje maksimalni element se sada mogu

prosto ignorisati, posle čega su preostala upoređivanja identična sa nizom upoređivanja koja se izvode pri izvršavanju algoritma sa n elemenata (pri čemu je prvi korak neparan). Prema induktivnoj hipotezi sortiranje n elemenata se izvodi korektno; prema tome, i sortiranje svih $n+1$ elemenata je korektno, a završava se posle $n+1$ koraka. □



Slika 15: Korak indukcije u dokazu ispravnosti sortiranja parno-neparnim transpozicijama, Teorema 1.

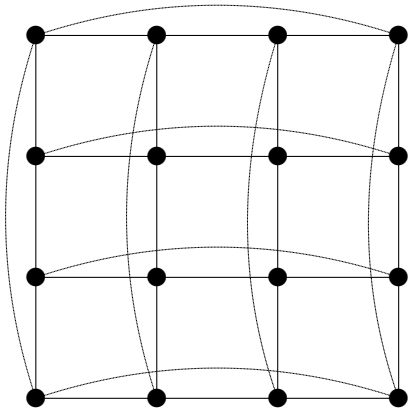
Do sada smo razmatrali slučaj jednog elementa po procesoru. Pretpostavimo sada da svaki procesor pamti k elemenata, i razmotrimo za početak slučaj samo dva procesora. Pretpostavimo da je cilj da procesori međusobno razmene elemente, tako da k najmanjih elemenata dođu u P_1, a k najvećih u P_2. Jasno je da u najgorem slučaju svi elementi moraju biti razmenjeni, pa je tada broj razmena elemenata $2k$. Sortiranje se može izvesti ponavljanjem sledećeg koraka sve dok je potrebno: P_1 šalje svoj najveći element u P_2, a P_2 svoj najmanji element u P_1. Proces se završava u trenutku kad je najveći element u P_1 manji ili jednak od najmanjeg elementa u P_2. Ovaj korak zove se *objedinjavanje-razdvajanje*. Koristeći ovaj korak kao osnovni u sortiranju parno-neparnim transpozicijama, dobija se uopštenje sortiranja na slučaj više elemenata po procesoru.

Iako je ovakav algoritam sortiranja optimalan za niz procesora, njegova efikasnost je mala. Imamo n procesora koji izvršavaju n koraka; prema tome, ukupan broj koraka je n^2. Mala efikasnost $E(n,n) = \frac{E(n,1)}{nE(n,n)} = \frac{n \log n}{n^2} = O(\log n/n)$ nije iznenađujuća, jer efikasan algoritam za sortiranje mora imati mogućnost da zamenjuje mesta međusobno udaljenim elementima. Niz procesora ne omogućuje takve zamene. U sledećem odeljku prikazaćemo specijalizovane mreže za efikasno sortiranje.

Množenje matrica na dvodimenzionalnoj mreži

Mreža računara koju ćemo sada razmotriti je dvodimenzionalna mreža $n \times n$. Procesor $P[i,j], 0 \leq i,j < n$ nalazi se u preseku i-te vrste i j-te kolone, i povezan je sa procesorima $P[i,j+1]$, $P[i+1,j]$, $P[i,j-1]$ i $P[i-1,j]$. Pretpostavlja se da su suprotni krajevi mreže međusobno povezani, odnosno da mreža liči na torus. Tako je, na primer, procesor $P[0,0]$ pored procesora $P[0,1]$ $P[1,0]$, povezan i sa procesorima $P[0,n-1]$ i $P[n-1,0]$, videti sliku 16. Drugim rečima, sva sabiranja i oduzimanja indeksa vrše se po modulu n (opseg vrednosti za indekse i, j je

$0, 1, \ldots, n-1$). Algoritam koji prikazujemo je simetričniji i elegantniji uz ovu pretpostavku; njegovo vreme izvršavanja jednako je (do na konstantni faktor) vremenu izvršavanja na običnoj mreži (onoj koja nije presavijanjem povezana kao torus).



Slika 16: Dvodimenzionalna presavijena mreža.

Problem. Date su dve $n \times n$ matrice A i B, tako da su njihovi elementi $A[i, j]$ i $B[i, j]$ u procesoru $P[i, j]$. Izračunati proizvod $C = AB$, tako da element $C[i, j]$ bude u procesoru $P[i, j]$.

Koristićemo običan algoritam za množenje matrica. Problem je premestiti podatke tako da se pravi brojevi nađu na pravom mestu u pravom trenutku. Posmatrajmo element $C[0, 0] = \sum_{k=0}^{n-1} A[0, k] \cdot B[k, 0]$, koji je jednak proizvodu prve vrste matrice A i prve kolone matrice B; redni brojevi vrsta i kolona su pri ovakvom označavanju za jedan veći od njihovih indeksa. Voleli bismo da broj $C[0, 0]$ bude izračunat u procesoru $P[0, 0]$. Ovo se može postići cikličkim pomeranjem prve vrste A ulevo, i istovremenim cikličkim pomeranjem prve kolone B uvis, korak po korak. U prvom koraku $P[0, 0]$ sadrži $A[0, 0]$ i $B[0, 0]$ i izračunava njihov proizvod; u drugom koraku $P[0, 0]$ dobija $A[0, 1]$ (od desnog suseda) i $B[1, 0]$ (od suseda ispod sebe) i njihov proizvod dodaje parcijalnoj sumi, itd. Vrednost $C[0, 0]$ se na ovaj način izračunava posle n koraka.

Problem je obezbediti da svi procesori obavljaju isti ovakav posao, a svi moraju da dele podatke međusobno. Potrebno je da podatke preuredimo tako da ne samo $P[0, 0]$, nego i svi ostali procesori dobiju sve potrebne podatke. Ideja je da se podaci u matricama ispremeštaju na takav način, da svaki procesor u svakom koraku dobije dva broja čiji proizvod treba da izračuna. Bitan je dakle početni raspored podataka. Problem rešava početni raspored takav da procesor $P[i, j]$ ima elemente $A[i, i + j]$ i $B[i + j, j]$, odnosno da drugi indeks elementa A bude jednak prvom indeksu elementa B (pri čemu se, kao što je rečeno, indeksi računaju po modulu n). Pošto se formira ovakav raspored, svaki korak se sastoji

a_{00} b_{00}	a_{01} b_{01}	a_{02} b_{02}	a_{03} b_{03}
a_{10} b_{10}	a_{11} b_{11}	a_{12} b_{12}	a_{13} b_{13}
a_{20} b_{20}	a_{21} b_{21}	a_{22} b_{22}	a_{23} b_{23}
a_{30} b_{30}	a_{31} b_{31}	a_{32} b_{32}	a_{33} b_{33}

a_{01} b_{10}	a_{02} b_{21}	a_{03} b_{32}	a_{00} b_{03}
a_{12} b_{20}	a_{13} b_{31}	a_{10} b_{02}	a_{11} b_{13}
a_{23} b_{30}	a_{20} b_{01}	a_{21} b_{12}	a_{22} b_{23}
a_{30} b_{00}	a_{31} b_{11}	a_{32} b_{22}	a_{33} b_{33}

Tabela 2: Početno razmeštanje elemenata matrica — priprema za paralelno množenje.

od istovremenog cikličkog pomeranja vrsta A i kolona B, čime $P[i,j]$ dobija elemente $A[i, i+j+k]$ i $B[i+j+k, j]$, $0 \leq k \leq n-1$, što su upravo oni elementi koji su mu potrebni. Do početnog rasporeda može se doći cikličkim pomeranjem vrste A sa indeksom i za i mesta ulevo, a kolone B sa indeksom i za i mesta naviše, $i = 0, 1, \ldots, n-1$. Algoritam je prikazan na slici 17. Na slici 2 prikazano je formiranje početnog rasporeda elemenata matrica za $n = 4$. Leva strana pokazuje početno stanje podataka, a desna njihov razmeštaj posle izvršavanja početnih cikličkih pomeranja.

Algoritam Množenje_matrica(A, B);
Ulaz: A i B ($n \times n$ matrice).
Izlaz: C (proizvod AB).
begin
  for $i := 0$ **to** $n-1$ **do in parallel**
    ciklički pomeri ulevo vrstu i matrice A za i mesta;
    {odnosno, izvrši $A[i,j] := A[i, (j+1) \bmod n]$ i puta}
  for $j := 0$ **to** $n-1$ **do in parallel**
    ciklički pomeri naviše kolonu j matrice B za j mesta;
    {odnosno, izvrši $B[i,j] := B[(i+1) \bmod n, j]$ j puta}
  {podaci su sada na željenim početnim pozicijama}
  for svi parovi i, j **do in parallel**
    $C[i,j] := A[i,j] \cdot B[i,j]$;
  for $k := 1$ **to** $n-1$ **do**
    for svi parovi i, j **do in parallel**
      $A[i,j] := A[i, (j+1) \bmod n]$;
      $B[i,j] := B[(i+1) \bmod n, j]$;
      $C[i,j] := C[i,j] + A[i,j] \cdot B[i,j]$
end

Slika 17: Algoritam za paralelno množenje matrica na mreži.

Složenost. Početna ciklička pomeranja vrsta A traju $n/2$ paralelnih koraka (kad

broj pomeranja postane veći od $n/2$, pomeranja se izvode u suprotnom smeru — udesno umesto ulevo); isto važi i za početna pomeranja kolona B. U narednih n koraka u svakom procesoru izvode se izračunavanja i pomeranja. Ti koraci mogu se izvršiti paralelno. Ukupno vreme izvršavanja je $O(n)$. Efikasnost algoritma je $E(n, n^2) = \frac{T(n,1)}{n^2 T(n,n^2)} = \frac{n^3}{n^2 \cdot n} = O(1)$, ako upoređujemo paralelni algoritam sa običnim sekvencijalnim množenjem matrica složenosti $O(n^3)$. Efikasnost je asimptotski manja, ako paralelni algoritam upoređujemo sa npr. Štrasenovim algoritmom za množenje matrica.