SVEUČILIŠTE U ZAGREBU
PMF – MATEMATIČKI ODJEL

Sanja Singer i Saša Singer

Paralelni algoritmi 1

Predavanja i vježbe

Zagreb, 2003.

Sadržaj

1. Uvod **1**

1.1. Mali podsjetnik za ISO prefikse 1

1.2. Razlozi razvoja 2

1.3. Potencijalna upotreba 3

1.4. Primjer modela klime na Zemlji 4

1.5. Zašto je pisanje brzih programa tako teško 7

1.6. Osnovna klasifikacija paralelnih računala 8

1.6.1. SISD računalo 8

1.6.2. MISD računalo 9

1.6.3. SIMD računalo 10

1.6.4. MIMD računalo 11

1.7. Povijest računala u brojkama 12

1.8. Povijesni razvoj paralelnih računala 13

1.8.1. Prije 1980. 13

1.8.2. Rane osamdesete 14

1.8.3. Rođenje hiperkocke 15

1.8.4. Sredina osamdesetih 15

1.8.5. Kasne osamdesete 15

1.8.6. Devedesete godine – teraflops je dostignut 16

2. Paralelne arhitekture i modeli programiranja **18**

2.1. Programski model 18

2.2. Paralelizam, komunikacija i sinkronizacija 19

2.2.1. Paralelizam 19

2.2.2. Komunikacija 19
2.2.3. Sinkronizacija 21
2.3. Modeliranje komunikacija na računalima sa razdijeljenom memorijom 22
2.3.1. Model PRAM računala 22
2.3.2. Log$P$ model 25
2.4. Neke komunikacijske mreže na računalima sa razdijeljenom memorijom 27
2.4.1. Statičke mreže 29
2.4.2. Dinamičke mreže 30
2.4.3. Hijerarhijske mreže 31
2.4.4. Slojevite mreže 33
2.4.5. Kombinacija slojevite i hijerarhijske mreže 36

3. Mjere ponašanja (efikasnost) paralelnih programa 39
3.1. Ubrzanje, efikasnost, cijena 39
3.1.1. Donja i gornja granica 40
3.1.2. Ubrzanje i efikasnost 41
3.1.3. Cijena 46
3.1.4. Donja ograda za paralelno izračunavanje funkcije s n parametara 46

4. Brzi algoritmi na stablima 49
4.1. Emitiranje i redukcija na stablima 49
4.2. Zapis algoritama za paralelni prefiks i scan 50
4.2.1. PRAM forma algoritma 51
4.2.2. Algoritam za arhitekturu sa zajedničkom memorijom . . . 55
4.2.3. Algoritam za arhitekturu s razdijeljenom memorijom . . . 60
4.3. Povećanje efikasnosti 66
4.4. Množenje matrica reda n u vremenu $\mathcal{O}(\log n)$ 67
4.5. Invertiranje trokutastih matrica reda n u vremenu $\mathcal{O}(\log^2 n)$ . . . 68
4.6. Invertiranje punih matrica reda n u vremenu $\mathcal{O}(\log^2 n)$ 70

4.7. Rješavanje trodijagonalnog linearnog sistema reda n u vremenu $\mathcal{O}(\log n)$ . . . 75
4.8. Parno–neparna redukcija za trodijagonalne linearne sisteme . . . 77
4.9. Zbrajanje dva n–bitna cijela broja u vremenu $\mathcal{O}(\log n)$ . . . 79
4.10. Paralelno izvrednjavanje rekurzija . . . 80
4.11. Paralelno izvrednjavanje izraza . . . 83

5. Matrično množenje na hijerarhijskoj memoriji . . . 84
5.1. BLAS . . . 84
5.2. Tri implementacije množenja matrica . . . 88
5.2.1. Množenje matrica bez blokova . . . 88
5.2.2. Množenje matrica blokovima stupaca . . . 90
5.2.3. Množenje matrica kvadratnim blokovima . . . 93
5.3. Paralelno množenje matrica – razdijeljena memorija . . . 95
5.3.1. 1D blokovski raspored . . . 97
5.3.2. 1D blokovski raspored na sabirnici (bez emitiranja, s barijerom) . . . 98
5.3.3. 1D blokovski raspored na sabirnici (bez emitiranja i barijere) 101
5.3.4. 1D blokovski raspored na sabirnici (s emitiranjem) . . . 106
5.3.5. 1D blokovski raspored na prstenu procesora . . . 108
5.3.6. Cannonov algoritam na 2D torusu . . . 110
5.3.7. Množenje matrica na 3D torusu . . . 114
5.3.8. Množenje matrica na hiperkocki . . . 114

Literatura . . . 118

1. Uvod

U ovom poglavlju navest ćemo motivaciju za razvoj paralelnih računala, njihovu osnovnu klasifikaciju i povijesni razvoj.

1.1. Mali podsjetnik za ISO prefikse

Redovi veličina koji se danas spominju, vezani uz paralelna računala, su ogromni. Ovdje je podsjetnik za značenje prefiksa:

- kilo = 10^3 osnovnih jedinica (prefiks K);
- mega = 10^6 osnovnih jedinica (prefiks M);
- giga = 10^9 osnovnih jedinica (prefiks G);
- tera = 10^{12} osnovnih jedinica (prefiks T);
- peta = 10^{15} osnovnih jedinica (prefiks P);
- exa = 10^{18} osnovnih jedinica (prefiks E?);
- mili = 10^{-3} osnovnih jedinica (prefiks m);
- mikro = 10^{-6} osnovnih jedinica (prefiks μ);
- nano = 10^{-9} osnovnih jedinica (prefiks n);
- piko = 10^{-12} osnovnih jedinica (prefiks p);
- femto = 10^{-15} osnovnih jedinica (prefiks f);
- ato = 10^{-18} osnovnih jedinica (prefiks a?).

Broj realnih (floating point) operacija u sekundi označavat ćemo s flops (floating point operations per second). (Napomena: katkad u engleskom jeziku flops označava i običnu množinu od flop!)

1.2. Razlozi razvoja

Paralelna računala posljedica su tehnološke revolucije s početka osamdesetih godina — iste tehnološke revolucije koja je omogućila PC računala.

U posljednjih 15 godina dolazi do drastičnog povećanja gustoće tranzistora na chipovima:

- Intel 8086 (1980.) sadržavao je 50.000 tranzistora;
- Digital Alpha RISC (1992.) sadržavao je 1.680.000 tranzistora.

Također, dolazi do dramatičnog povećanja takta (clock-a) računala s 4 MHz na današnjih 400 MHz. Povećanje gustoće tranzistora i povećanje brzine rezultiraju ubrzanjem preko 1000 puta.

Tih otprilike milijun tranzistora smješteno je na površinu veliku približno 2 cm^2. (Začudo treća dimenzija debljine chipa ne igra ulogu, obično su chipovi tanki. Zašto?!) Tehnološki razlozi (veličina tranzistora), barem prema današnjim saznanjima, ne dozvoljavaju daljnja radikalna smanjenja chipa. Zbog brzine putovanja informacija kroz chip (koja ne može biti veća od brzine svjetlosti), daljnji faktor 10 u ubrzanju postiže se paralelizmom.

Za ilustraciju, pretpostavimo da želimo sagraditi jednoprocesorsko računalo brzine 1 Tflops-a s 1 TB memorije. Ako podaci koje dohvaćamo iz memorije u centralni procesor moraju putovati udaljenost r, oni moraju taj put prevaliti 10^{12} puta u sekundi (pretpostavka da za svaku operaciju trebaju operandi iz memorije). Ako podaci putuju brzinom svjetlosti $c = 3 \cdot 10^8$ m/s, onda je

$$r \leq \frac{c}{10^{12}} = 0.3\,\mathrm{mm} \quad !$$

Dakle, cijelo računalo trebalo bi stati u kutijicu stranice 0.3 mm.

Razmotrimo sad 1 TB memorije. Memorija se konvencionalno izrađuje kao planarna rešetka. U ovom slučaju ona bi imala dimenziju $10^6 \times 10^6$ riječi (byte-a). Ako je čitavo računalo smješteno u kutijicu s bridovima 0.3 mm, onda jedan byte mora stati u kvadrat s duljinom stranice p

$$p \leq \frac{0.3}{10^6} = 3 \cdot 10^{-7}\,\mathrm{mm} = 3 \cdot 10^{-10}\,\mathrm{m} \quad ,$$

odnosno, veličine manjeg atoma. A odakle bi izvirale žice?

Prema predviđanjima stručnjaka, negdje iza 2000. godine, i kućna PC računala postat će paralelna.

Vodeću ulogu u razvoju paralelnih računala imale su Sjedinjene Američke Države, gdje su federalne službe za znanost, tehnologiju i svemir (i naravno vojska)

inicirale razvoj i hardware-a i software-a superkompjutera. Poznato je da vlada može poticati razvoj, ali bez ozbiljne komercijalne i industrijske primjene razvoj stagnira (novac).

1.3. Potencijalna upotreba

Tradicionalno, ljudi prvo naprave neku teoriju (na papiru), a zatim je pokušavaju eksperimentalno dokazati (ili opovrgnuti) u laboratoriju – gradnjom prototipova. Umjesto gradnje prototipova, danas se mogu numerički simulirati "prototipovi".

Kad je napravljen projekt razvoja paralelnih računala, analizirani su i problemi koji bi se sekvencijalnim računalima teško mogli riješiti (zbog trajanja), a trebali bi se rješavati paralelnim računalima.

Na listi mogućih primjena našli su se slijedeći problemi:

- upotreba u naftnoj industriji – analiza novih naftonosnih polja i simulacija eksploatacije nafte iz ležišta već postojećih polja;
- model zagađivanja zraka i tla, simulacija klime na Zemlji, vremenska prognoza;
- simulacija opticanja fluida – primjena u avioindustriji, ali i primjena u automobilskoj industriji (autori navode simulaciju sudara?!);
- dizajniranje novih lijekova modeliranjem novih smjesa;
- identifikacija novih materijala s posebnim svojstvima (kao što je supravodljivost);
- simulacija elektro i plinske mreže s ciljem optimizacije proizvodnje i reakcije na kvarove;
- proizvodnja animiranih filmova;
- podrška geografskom informacijskom sistemu NASA-e, uključivo i analizu podataka u realnom vremenu sa satelita "Mission to Planeth Earth";
- vojna upotreba, posebno u područjima "komandiranje i kontroliranje", te kao podrška odlučivanju;
- telekomunikacijske svrhe za brze multimedijalne servere (elektronske – trenutne novine, Intrenet prijenosi događaja sa slikom, zvukom i masom dodatnih informacija uživo, predviđaju se Internet Sveučilišta i sl.);
- brzo pretraživanje dokumenata na Internetu.

Tipična pitanja u fazi dizajniranja paralelnih računala bila su:

- Da li su paralelna računala zapravo računala opće namjene? (Tj. da li su paralelna računala dovoljno fleksibilna kao sekvencijalna.)
- Da li je određeni tip računala bolji za neku svrhu nego drugi?
- Da li je za paralelno računalo potreban bitno paralelan software? Trebaju li za paralelna računala bitno drugačiji programski jezici?
- Treba li paralelno računalo biti izrađeno od malo superbrzih (skupih) chipova ili je bolje koristiti tisuće sporijih (jeftinijih) chipova — na primjer, chipova za PC računala? ("Armija mrava ili krdo slonova"?!)

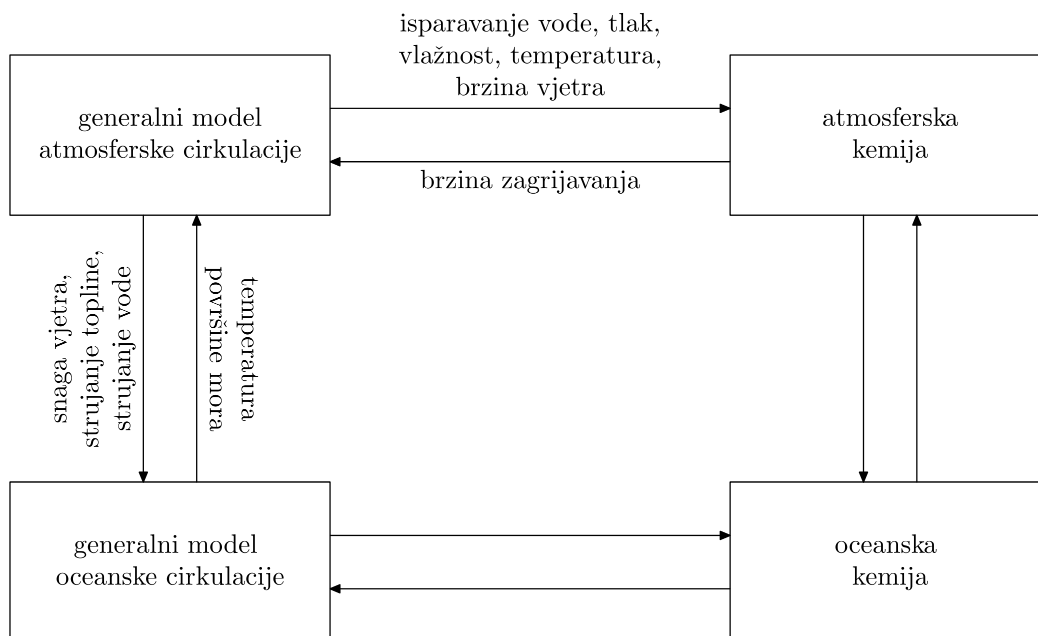
Postoji nekoliko razloga zašto je bolje napraviti simulaciju računalom:

- Stvarni fenomeni su katkad prekomplicirani da se modeliraju na papiru ili u laboratoriju (na primjer, simulacija klime).
- Katkad su pokusi vrlo skupi, vrlo opasni i spori, ili ih je vrlo teško napraviti (simulacija naftnih bušotina, aerodinamički tuneli, dizajn aviona i sl.).
- Većina fizikalnih procesa ne podnosi promjenu skale (projektiranje propelera brodova). Vrlo često, primjer pogrešne skale vidimo na filmovima (izgaranje male makete kuće koja gori normalnom brzinom – reskaliranje daje ogromne plamenove i ubrzano gorenje).

1.4. Primjer modela klime na Zemlji

Prognoziranje vremena za nekoliko dana unaprijed, kao i dugoročna simulacija klime (moderno: efekti El Niña), primjeri su koje možemo simulirati paralelnim računalom.

Dijelovi jednostavnog modela klime na Zemlji prikazani su na slijedećem dijagramu.



Najjednostavniji klimatski model funkcija je 4 argumenta: zemljopisne širine, dužine, visine (od tla) i vremena. Kao rezultat dobivamo 6 vrijednosti: temperaturu, tlak, vlažnost i brzinu vjetra (komponente u 3 smjera). Generalniji model mogao bi uključivati, na primjer, koncentraciju različitih plinova u atmosferi. Stvarni model atmosferskih procesa uključivao bi i stvaranje oblaka, količinu padalina, kemiju i sl.

Primijetimo da je klima neprekidna funkcija u svoje (barem) četiri varijable. Jasno je da računalom možemo simulirati kontinuirane procese samo u ponekim (diskretnim) točkama. Zbog toga, potrebno je diskretizirati klimatski model. Tada možemo aproksimirati klimu samo u diskretizacionim točkama – tj. računamo samo vrijednosti $klima(i, j, k, n)$, gdje i, j i k indeksiraju zemljopisnu širinu, zemljopisnu dužinu i visinu. Za razmak vremenskih koraka simulacije možemo izabrati dt, pa indeks n označava koji je to trenutak dt po redu (preciznije, vrijeme diskretiziramo kao $t = n \cdot dt$ i samo u višekratnicima od dt radimo simulaciju).

Algoritam za prognozu vremena (kratkoročnu) i simulaciju klime na Zemlji (dugoročnu) je, zapravo, funkcija koja preslikava stanje klime u trenutku t, tj. $klima(i, j, k, n)$, za sve i, j i k, u stanje klime u susjednom vremenskom trenutku $klima(i, j, k, n{+}1)$. Algoritam sadrži rješavanje sistema jednadžbi koje opisuju model klime — na primjer, Navier–Stokesove jednadžbe za gibanje plinova u atmosferi.

Pretpostavimo da diskretiziramo površinu Zemlje u polja stranice 1 km, te visinski u 10 točaka. Iz podataka o oplošju Zemlje, dobivamo da nam je za taj model potrebno približno $5 \cdot 10^9$ diskretizacijskih točaka. Ako koristimo 6 riječi (dugih 4 byte-a) po točki diskretizacije, potrebno nam je $10^{11} = 0.1$ TB memorije.

Pretpostavimo da nam je za vremensku prognozu za 1 minutu u odgovarajućoj diskretizacijskoj točki potrebno 100 flop-a. Drugim riječima, $dt = 1$ minuta, a za računanje svih $klima(i, j, k, n+1)$, za sve i, j i k, iz $klima(i, j, k, n)$, potrebno nam je $100 * 5 \cdot 10^9 = 5 \cdot 10^{11}$ flop-a. Jasno je da za predviđanje vremena za slijedeću minutu ne smijemo potrošiti više od 1 minute vremena rada računala (inače je jeftinije pogledati kroz prozor). Dakle, računalo mora imati brzinu od najmanje

$$5 \cdot 10^{11} \text{ flop-a/60 sekundi} \approx 8 \text{ Gflops-a} \quad .$$

Prognoza vremena za slijedećih 7 dana (s dozvoljenim vremenom računanja 24 sata) mora biti 7 puta brža, tj. računalo mora postizati brzinu od 56 Gflops-a. Predviđanje klime po tom modelu za slijedećih 50 godina (s dozvoljenim vremenom računanja mjesec dana) mora biti $50 \cdot 12 = 600$ puta brže, tj. računalo mora postizati brzinu od 4.8 Tflops-a.

Trenutni (1996.) modeli računanja imaju rezoluciju od 4° zemljopisne širine i 5° zemljopisne dužine, tj. približno 450×560 km. Cilj je profiniti mrežu 2 puta u svakoj dimenziji, tj. da rezolucija bude 2° zemljopisne širine i 2.5° zemljopisne dužine.

Veličina takve baze podataka je ogromna. NASA je poslala u svemir vremenske satelite za koje se očekuje da dnevno pošalju na Zemlju 1 TB informacija. Ukupno, za sve godine koje će sateliti biti u orbiti, to je približno 6 PB podataka, što ne može spremiti niti jedno postojeće računalo.

Drugi autori spominju da se, za današnji model 10–godišnje simulacije klime na Zemlji, zahtijeva otprilike 10^{16} flop–a. S računalima snage 10 gigaflops-a, za takvo računanje potrebno je otprilike 10 dana računanja (točnije 11.57 dana). Takva simulacija generira približno 10^{11} bitova podataka!

Znanstvenici bi željeli usavršiti postojeći model slijedećim poboljšanjima:

Današnji model	Željeni model	Povećanje ×
rezolucija 100 km	rezolucija 10 km	10^2–10^3
jednost. reprezentacija procesa	složena reprezentacija procesa	2–10
jednost. utjecaj oceana	složen utjecaj oceana	2–5
jednost. atmosferska kemija	složena atmosferska kemija	2–5
jednost. biosfera	složena biosfera	≈ 2
10–god. simulacija	100–god. simulacija	10–10^2

Sve zajedno daje povećanje trajanja računanja između 10^4 i 10^7 puta! Pretpostavimo li da je povećanje samo 10^4 puta, umjesto 11.57 dana, istim računalom

računali bismo $11.57 \cdot 10^4$ dana, što je 316 godina. Uz pretpostavku da u tih 316 godina računalo neprestano računa (ne pokvari se i ne nestane struje), simulacija klime bila bi besmislena, jer smo već saznali što se dogodilo (i umrli u međuvremenu).

1.5. Zašto je pisanje brzih programa tako teško

Godinama se kao jedna od mjera brzine računala koristi Linpack Benchmark. Test se sastoji u mjerenju brzine rješavanja sistema linearnih jednadžbi $Ax = b$ metodom Gaussovih eliminacija. Kao standardni test uzima se linearni sistem reda 100. Međutim, za paralelna računala, za takav test mogu se uzeti i mnogo veći redovi linearnog sistema.

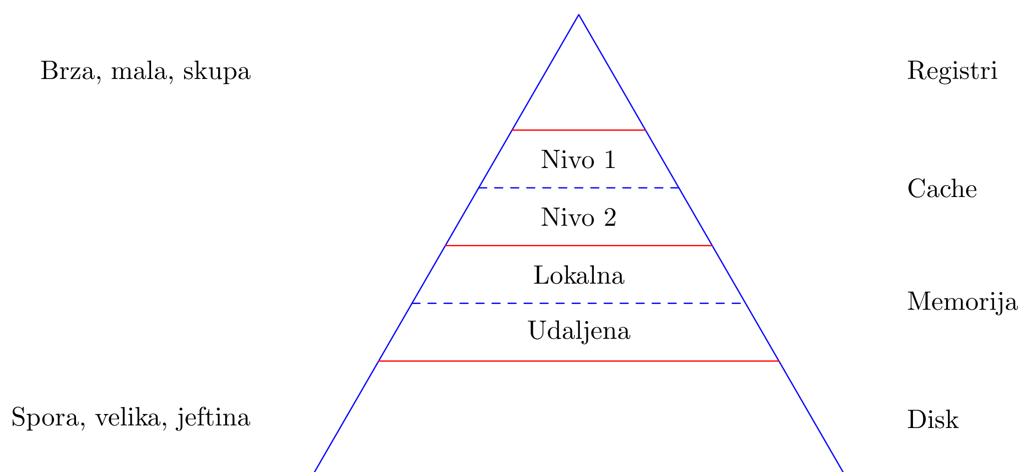
Vršna brzina je maksimalna moguća brzina računala, ako su svi procesori u svakom trenutku zaposleni i ako nema transfera podataka. Ali realni problemi najčešće nisu tako idealno paralelni! Uobičajeno, problemi imaju i sekvencijalnih dijelova, pa u nekim trenucima rade samo neki procesori, a drugi čekaju. Također, treba uračunati vrijeme dostupa do odgovarajućih podataka u memoriji, što je, uobičajeno, najsporiji dio posla. Zbog toga, mnogo je realnije mjeriti vremena na nekom problemu, a ne promatrati vršnu brzinu.

U siječnju 1996., najbrže računalo na svijetu bio je Intel Paragon s 6768 procesora (i860 chipovi), svaki s vršnom brzinom 50 Mflops-a, odnosno sveukupnom vršnom brzinom $50 \cdot 6768 = 338$ Gflops-a.

Ako se za Linpack test uzme matrica reda 128.600, računalo postiže brzinu 281 Gflops-a, odnosno za rješavanje čitavog problema potrebne su 84 minute.

Ako tim istim računalom pokušavamo riješiti linearni sistem reda 100, računalo će postići maksimalnu brzinu od samo 10 Mflops-a, koristeći jedan jedini procesor (od njih 6768). Rješavamo li linearni sistem reda 1000 na tom jednom jedinom procesoru, na tom računalu, postići ćemo brzinu 36 Mflops-a.

U čemu je tolika razlika, da brzina tako mnogo ovisi o dimenziji problema? Razlog leži u takozvanoj memorijskoj hijerarhiji. Sva računala, od onih jeftinijih do superkompjutera, imaju memoriju podijeljenu kao na slijedećoj slici:



Stvarne operacije mogu se obavljati samo na vrhu piramide – u registrima. Zbog toga, podaci koji su spremljeni na nižim hijerarhijskim nivoima moraju se pomaknuti u registre (podaci koji su bili tamo moraju se maknuti u memoriju nižu u hijerarhiji). Pomicanje podataka gore–dolje među raznim tipovima memorije je sporo, mnogo sporije nego što su to pojedine floating point operacije u registrima. Uobičajeno se troši mnogo više vremena na pomicanje podataka nego na koristan posao.

Dobro dizajnirani algoritam pokušava aktivne podatke držati što bliže vrhu hijerarhije u memoriji i minimizira transfer podataka među nivoima. Za mnoge probleme, kao što su Gaussove eliminacije, samo ako je problem dovoljno velik, vrijeme potrebno za dobivanje rezultata "pokrije" vrijeme potrebno za transfer podataka među razinama u hijerarhiji memorije.

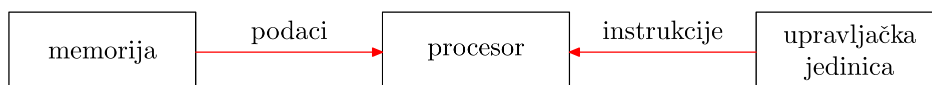
1.6. Osnovna klasifikacija paralelnih računala

Svako računalo, bilo ono sekvencijalno, vektorsko ili paralelno, izvršava niz instrukcija nad nizom podataka. Klasifikacija računala se vrši prema tome koliko se instrukcija izršava nad koliko podataka u jednom vremenskom trenutku. Klasifikaciju je dao Flynn (1966.). Razlikujemo:

1.6.1. SISD računalo

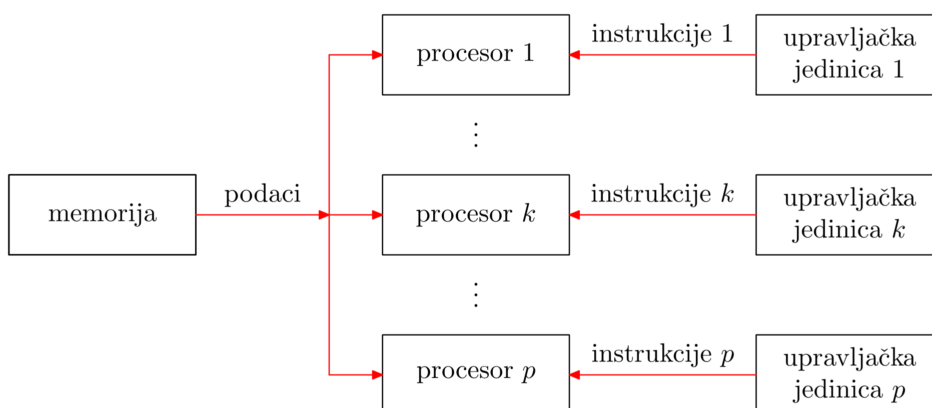
Jedan niz instrukcija izvršava se nad jednim nizom podataka (Single Instruction stream, Single Data stream). Ovo računalo je zapravo klasično sekvencijalno

računalo. Shematski, ono izgleda ovako:



1.6.2. MISD računalo

Više nizova instrukcija izvršava se nad jednim nizom podataka (Multiple Instruction stream, Single Data stream). Ovo računalo građeno je od p procesora, od kojih svaki ima svoju kontrolnu jedinicu. Memorija je zajednička za sve procesore. Svaki procesor izvršava nad istim (nepromijenjenim) podatkom različit posao. Shematski, ovo računalo izgleda ovako:



Evo i dva primjera za moguće korištenje takve vrste računala.

Primjer 1.6.1.

Zadatak je ispitati da li je cijeli pozitivan broj n prost. Jedan od načina rješavanja je da se ispita, da li je taj broj djeljiv bilo kojim prostim brojem manjim ili jednakim $n/2$. Pretpostavimo da MISD računalo ima upravo toliko procesora, koliko ima takvih prostih brojeva. Rezultat da li je broj n složen ili prost, dobit ćemo u prvom koraku, jer ako je barem jedan procesor odgovorio pozitivno, broj je složen. Realno je očekivati da je broj procesora manji od broja potencijalnih djelitelja. Tada svaki procesor ispituje neki podskup djelitelja.

Primjer 1.6.2.

U mnogim primjenama zadatak je klasificirati neki objekt, tj. odrediti kojoj klasi objekata pripada. Na primjer, kod ispitivanja specijalnim aparatom u dubokom moru mora se prepoznati da li je neki objekt: jato riba, alge, stijene, smeće ili nuklearna podmornica. Da bi se "prepoznao" neki objekt, potrebno je više uvjeta

koje on mora zadovoljavati. Uvjeti su, na primjer, veličina objekta (između nekih granica), oblik (elipsast, nepravilan), raspršena ili kompaktna sjena i sl.

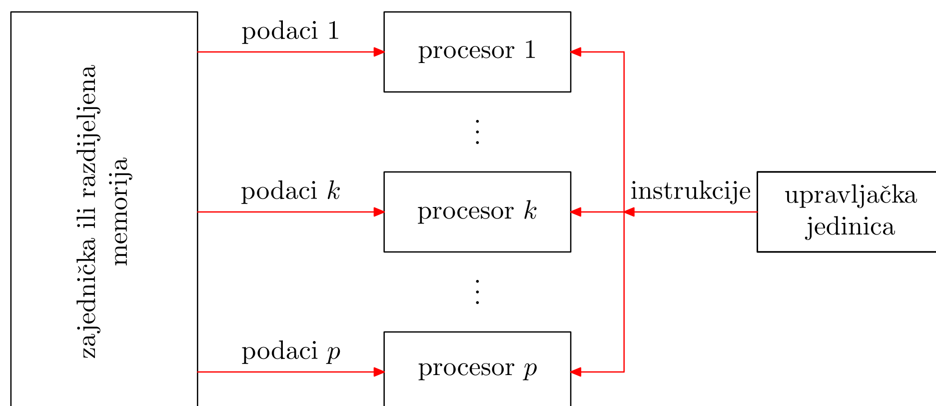
Svaki od procesora izvodi program koji prepoznaje točno jednu vrstu objekata. Svi skupljeni podaci šalju se svakom procesoru. Pretpostavka je da su vrste objekata disjunktne, tj. da se niti jedan objekt ne može svrstati u dvije ili više vrsta. Tada će najviše jedan procesor javiti potvrdan odgovor, tj. da je objekt prepoznat. Nažalost, u realnom svijetu vrlo je teško napraviti dobru podjelu objekata u vrste, tako da one budu disjunktne po izabranim karakteristikama. Osim toga, vrste objekata za prepoznavanje se neprestano mijenjaju, pa MISD računalo takve namjene nije nikad ni izrađeno.

Prethodna dva primjera pokazuju da MISD računala mogu biti korisna u mnogim specijaliziranim primjenama (posebno sagrađena računala za neki zadatak). Za općenitu primjenu nisu naročito pogodna zbog nefleksibilnosti.

1.6.3. SIMD računalo

Jedan niz instrukcija izvršava se nad više nizova podataka (Single Instruction stream, Multiple Data stream). U ovoj klasi računalo se sastoji od p identičnih procesora. Svaki od njih ima neki dio memorije u koji može spremati podatke i program. (Podrobnije o različitim memorijskim organizacijama kasnije!) Svi procesori upravljani su jednakim instrukcijskim nizom upravljačke jedinice.

Svi procesori rade sinkrono: u svakom koraku izvršavaju istu instrukciju. Može se dogoditi da neki procesori u nekom koraku čekaju (vrte praznu instrukciju). Gruba shematska skica izgleda ovako:



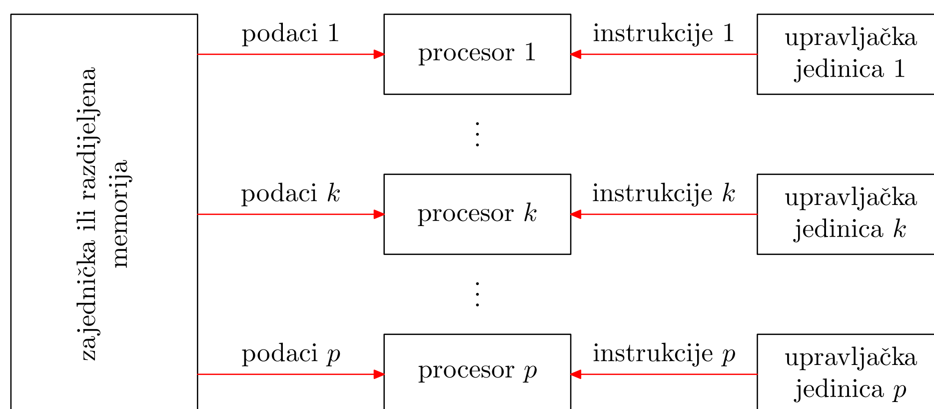
Jedna od podvrsta SIMD računala je vektorsko računalo, koje s vektorima postupa kao sekvencijalno računalo sa skalarima. Ova vrsta paralelizma je na nivou elementarnih operacija. Postoje dvije implementacije vektorskog računala: polje procesora (processor array) i niz uzastopnih procesora (pipelining).

Polje procesora “prima” svoj element vektora i obavlja operacije nad njim. Procesori su sinkronizirani. Nedostatak – broj procesora konstantan.

Kod niza uzastopnih procesora, vektori se dohvaćaju iz memorije i propuštaju kroz “cijev” procesora (element po element), tako da svaki procesor obavlja svoj dio posla.

1.6.4. MIMD računalo

Više nizova instrukcija izvršava se nad više nizova podataka (Multiple Instruction stream, Multiple Data stream). Ovo je najgeneralnije paralelno računalo. Svaki procesor može obavljati različitu operaciju. Generalno, pod ovom klasom računala možemo smatrati i nizove računala spojene mrežom. Shematski prikaz:



1.7. Povijest računala u brojkama

Snaga računala počevši od 1945. do danas raste eksponencijalno, u prosjeku za faktor 10 svakih 5 godina. Kako je to izgledalo kroz protekla desetljeća:

Godina (približno)	Računalo	Tip	Flops
1945	ENIAC	$\circ$	$5 \cdot 10^2$
1952	UNIVAC	$\circ$	$\geq 10^3$
1958	IBM 704	$\circ$	$\leq 10^4$
1960	IBM 7090	$\circ$	$\approx 10^5$
1964	CDC 6600	$\circ$	$\geq 10^6$
1968	CDC 7600	$\circ$	$\approx 10^7$
1977	Cray 1	$\circ$	$\geq 10^8$
1983	Cray X–MP	$\Diamond$	$\leq 10^9$
1987	Cray Y–MP	$\Diamond$	$\geq 10^9$
1991	Intel Delta	$\triangle$	$\approx 5 \cdot 10^{10}$
1992	Cray C 90	$\Diamond$	$\approx 10^{10}$
1994	IBM SP–2	$\triangle$	$\approx 10^{11}$

U prethodnoj tablici oznaka $\circ$ označava sekvencijalno računalo, oznaka $\Diamond$ umjereno paralelno računalo s 4–16 vektorskih procesora. Oznaka $\triangle$ označava masovno (bitno) paralelno računalo s redom veličine 100 do 1000 procesora.

Brzina računanja ovisi direktno i o vremenu potrebnom za izvođenje osnovnih računskih operacija. To vrijeme ovisi o brzini procesorskog sata (clock cycle) – između dva susjedna otkucaja, procesor je u stanju obaviti samo najprimitivniju operaciju (to ne mora biti osnovna računska operacija!). Tipične brzine procesorkih satova (u nanosekundama) bile su:

Godina (približno)	Računalo	Ciklus sata
1977	Cray 1	12.5
1987	Cray Y–MP	6.5
1992	Cray C 90	4.5
1989	IBM RS/6000 mod 530	40
1991	IBM RS/6000 mod 550	25
1994	DEC Alpha	10

Prva tri procesora pripadaju grupi vektorskih superkompjutera, a druga tri RISC

(reducirani skup instrukcija) mikroprocesora. Čini se da su obje skupine stigle do fizikalne granice.

Odakle dolaze te granice? Da bi zaobišli te granice, dizajneri procesora pokušavaju uspostaviti unutarnju paralelnost procesora na 64–bitnim operandima. Osnovni rezultat teorije složenosti za chipove vrlo visokog stupnja integracije (VLSI) kaže da je ta strategija skupa. Naime takav rezultat vrijedi za sva tranzitivna računanja, tj. ona za koja bilo koji izlaz može ovisiti o ulazu. Površina chipa A i vrijeme T potrebno za takvo računanje vezani su tako da AT^2 mora biti veće od neke funkcije (koja ovisi o problemu) dimenzije problema.

Jednostavno (neformalno) objašnjenje ovog rezultata je slijedeće. Svako računanje mora pomaknuti podatke s jedne strane kvadratnog chipa (stranice $\sqrt{A}$) na drugu. Količina informacija koju možemo prenijeti u jednom vremenskom trenutku ograničena je presjekom chipa, što daje brzinu transfera $\sqrt{A}T$, a odatle se dobiva i relacija za AT^2.

Da bi se smanjilo vrijeme za pomicanje informacije za neki faktor, presjek chipa mora se povećati za isti faktor. Ovaj AT^2 rezultat kaže ne samo to da je teško izrađivati pojedinačne chipove koji će raditi brže, nego da to možda (financijski) nije poželjno. Možda je jeftinije koristiti više sporijih komponenti.

Na primjer, ako se za računalo koristi površina n^2A silicija (figurativno), možemo napraviti n^2 komponenti svaka površine A, koje obavljaju neku operaciju u vremenu T. Ako izradimo samo jednu komponentu, moći ćemo tu istu operaciju obaviti u vremenu T/n. Zbog toga je višekomponentni sistem potencijalno n puta brži.

1.8. Povijesni razvoj paralelnih računala

Razvoj paralelnih računala možemo podijeliti u nekoliko razdoblja.

1.8.1. Prije 1980.

Postojala su paralelna računala i prije 1980. ali nisu imala široku primjenu na znanstveno računanje (scientific computing).

Prvo paralelno računalo bio je Illiac IV (1976.). Imao je osrednji učinak, bio je vrlo težak za programiranje i imao je nisku pouzdanost. Nije bio komercijalno računalo. Zanimljiviji podatak je da su se za njega posebno pisali kodovi za dinamiku fluida i tada je bio 6 puta brži od CDC 7600. Za ostale probleme bio je i sporiji od CDC 7600. Illiac IV bio je SIMD računalo s 32 procesora (svaki s lokalnom memorijom) povezana u prsten. Mogao se programirati u dva (visoka)

jezika FORTRAN-u i Glypnyr-u (paralelni jezik).

ICL Distributed Array Processor (DAP) bio je komercijalno računalo napravljeno u Engleskoj. Bio je SIMD računalo i mogao je imati 1 K ili 4 K jednobitnih procesora povezanih u matricu (dvodimenzionalno polje). Koristio se uglavnom na sveučilištima. Za funkcioniranje bio mu je potreban ICL mainframe (glavno računalo).

Treće važno paralelno računalo tog razdoblja je Goodyear Massively Parallel Processor (MPP). Projekti za gradnju su počeli 1969., ali su prodani vojsci i federalnoj službi (SAD) za kontrolu leta. Krajem sedamdesetih, MPP je instaliran u Goddard Space Flight Centre-u i znanstvenom centru NASA-e. Privukao je pažnju zbog toga što je na nekim primjenama mogao postići (nevjerojatnu) brzinu stotinjak Mflops-a. MPP je imao 16 K jednobitnih procesora, svaki s lokalnom memorijom, a mogao se programirati u Pascal-u i asembleru.

Posebnu pažnju treba još posvetiti Cray 1 računalu, koji je bio jednoprocesorsko vektorsko superračunalo (1976.). Mogao je operirati sa 64 parova podataka istovremeno.

1.8.2. Rane osamdesete

U ranim osamdesetim počinje razvoj MIMD računala.

Prvo takvo računalo je Denelcor Heterogeneous Element Processor (HEP). Bez obzira na jeziv odnos snage/cijene, privukao je pažnju zbog različitih mogućnosti programiranja. Bio je instaliran u Los Alamos-u, Argonne National Laboratory, Ballistic Research Laboratory i Messerschmidt-u u Njemačkoj (jedino stvarno radio na realnim problemima). Mogao je podržavati fini i grubi paralelizam.

Svaki procesor imao je mogućnost pipelininga pojedine instrukcije. Instrukcije različitih procesa stavljale su se u niz (queue) tako da se izvršavaju kad su operandi dovučeni iz memorije. Instrukcijski pipeline moglo je dijeliti do maksimalno 128 procesa.

Sve instrukcije (osim dijeljenja) trajale su 8 ciklusa. Do 16 procesora moglo je biti zajedno povezano, tako da se dobije grublje (large–grain) MIMD računalo. HEP je imao vrlo efikasan sinkronizacijski mehanizam. Mogao se porgramirati u FORTRAN-u, C-u i asembleru, imao je UNIX okolinu, a njegovo glavno računalo (front end) mogao je biti minikompjuter. Najveća zasluga tog računala – učenje paralelnog programiranja za nekoliko stotina ljudi.

U to vrijeme drugo značajno paralelno računalo bio je Cray X–MP/22 (1982.). Imao je samo 2 (super) procesora i služio je u znanstvene svrhe.

Još jedno MIMD računalo imalo je velik utjecaj na početku osamdesetih -

New York University Ultracomputer. Važnost tog računala je uglavnom u konceptu velikog broja (sporih) procesora (64) i različitih načina povezivanja među njima.

1.8.3. Rođenje hiperkocke

Možda najznačajnije računalo ranih osamdesetih je Caltech Cosmic Cube (hiperkocka) razvijena od Charlesa Seitza i Geoffreya Foxa. 1981. izrađena je šesto-dimenzionalna kocka (2^6 Intel 8086/87 procesora sa 128 KB memorije svaki). Ideja – rastom broja procesora, veze među njima trebale bi umjereno (a ne kvadratično) rasti, a da fleksibilnost ostane očuvana.

1.8.4. Sredina osamdesetih

Mnogo novih računala lansirano je od strane komercijalnih kompanija – najuspješnije Sequent i Encore. Računala koja su izrađena od te 2 kompanije imala su 16–30 procesora (shared memory), imala su Unix okolinu i dobar time–sharing. Bili su dobri za učenje programiranja i za manje poslove.

Još je jedno računalo obilježilo ovo razdoblje – Alliant. Inicijalni model imao je do 8 vektorskih procesora umjerene snage. Paralelne performanse – blizu tadašnjeg Cray-a.

Hiperkocke su razvijane u tom razdoblju od strane Intela-a, nCUBE, Ametek i Floating Point Systems Corporation. Hiperkocka nCUBe-a imala je do 1024 procesora i bila je instalirana u Sandia National Laboratories. Interesantna instalacija, jer je pokazano da ubrzanje procesa s jednog na sve procesore može biti i do 1000 puta.

Sredinom osamdesetih javljaju se i transpjuteri (T800 najpoznatiji) koji se stavljaju i u PC računala.

1.8.5. Kasne osamdesete

U ovom razdoblju počinje razvoj stvarno snažnih paralelnih računala.

Meiko računalo (Sveučilište Edinburgh) koristi 400 transpjutera T800 vezanih u matricu. Na temelju toga računala razavijen sistem predaje poruka (message passing system).

Krajem osamdesetih predstavljena su i slijedeća SIMD računala: CM–2 (Connection Machine), MasPar i novi DAP.

CM–2 bio je instaliran u Los Alamosu i imao je 64 K jednobitnih procesora, 2048 64–bitnih floating point procesora i 8 GB memorije. Linpack Benchmark

pokazuje 5.2 Gflops-a (red matrice 26.624). Razvijen CM FORTRAN (paralelni).

MasPar i DAP bili su manji sistemi i stavljani su u brodove i avione, a imali su i vojne svrhe.

Hiperkocke tog razdoblja: nCUBE–2 i Intel iPSC/860. nCUBE–2 – mogla je imati do 8 K procesora i imala je vršnu snagu 27 Gflops-a. Linpack Benchmark (red sistema 21.376) pokazuje 1.91 Gflops-a.

Intelova hiperkocka instalirana je u Oak Ridge-u (1990.) i imala je vršnu snagu 7 Gflops-a. Brzina komuniciranja među procesorima bila vrlo loša, pa se vršna snaga nije mogla ni izbliza doseći. Nasljednik ove hiperkocke je Touchstone Delta s 512 procesora i860 (snaga 32 Gflops-a vršno, Linpack Benchmark 13.9 Gflopsa (red sistema 25.000)). Slijedeći nasljednik je Intel Paragon (1992.) – komercijalno računalo koje je moglo imati do 4096 procesora i860 druge generacije. Vršna snaga 300 Gflops-a.

1.8.6. Devedesete godine – teraflops je dostignut

Kako je razvoj tekao dalje? Razvoj paralelnih računala u devedesetim godinama možda najbolje opisuje slijedeća tablica najbržih računala:

Godina	Računalo	Broj proc.	Brzina (Gflops-i)
1990.	Intel iPSC/860	128	2.6
1991.	Intel DELTA	512	13.9
1992.	Thinking Machines CM–5	1024	59.7
1993.	Intel Paragon	3744	143
1994.	Intel Paragon	6768	281
1996.	Hitachi CP–PACS	2048	368
1996.	ASCI Red	7264	1060
1997.	ASCI Red	9152	1340

U prethodnoj tablici, brzina označava tzv. MP Linpack test. Taj je test ekvivalentan uobičajenom Linpack testu, samo što on radi s matricama velikih redova. Rezultat takvog testa je, osim najveće postignute brzine i red matrice za koji se ta brzina postiže.

Zapravo, najbolji poticaj razvoju dalo je američko Ministarstvo energetike (ne obrane!), koje je započelo projekt ASCI (Accelerated Strategic Computing Initiative) kojemu je cilj izrada računala brzine 100 Tflops-a. Takvim bi se računalom mogla simulirati nuklearna eksplozija. Prvo su započeli razvojem "crvene", a zatim

i "plave" linije računala. Prvi cilj bio je dostići brzinu Linpack testa od jednog Tflops-a. Cilj je postignut krajem 1996., računalom iz "crvene" linije Intel ASCI Red, koji je postavljen u Sandia National Labs-u u Albuquerqueu. Zanimljivo je da je do danas to računalo prošlo nekoliko "dodavanja" i zamjena procesora snažnijima.

"Plava" linija, čiji predstavnici su IBM ASCI Blue Pacific (Lawrence Livermore National Laboratory) i SGI ASCI Blue Mountain (Los Alamos National Laboratory), trebala bi dostići brzinu od 3 Tflops-a.

Zanimljivo je da sva ta računala pripadaju imaju nekoliko tisuća procesora: Intel ASCI Red – 9632 procesora, IBM ASCI Blue Pacific – 7152 procesora (5808 u tzv. "zatvorenom" dijelu, a ostatak u "otvorenom" dijelu), SGI ASCI Blue Mountain – 6144 procesora. Da to nisu "kućni ljubimci", dovoljno je reći da im je cijena bila između 50 i 120 milijuna USD, "otisak stopala" (veličina tlocrta) između 100 i 750 m^2, potrošnja struje reda veličine 500 kW, težina oko 50 tona i da su im kablovi dugi oko 7.5 km.

Prema tablici objavljenoj u lipnju 2000. (tablica se objavljuje svakih 6 mjeseci, a može se naći na adresi `www.netlib.org/benchmark/top500.html`), brzine (u Gflops-ima) ta tri računala su:

Računalo	MP Linpack	Vršna brzina	Red matrice
ASCI Red	2379.6	3207	362880
Blue Pacific (zatvoreni dio)	2144	3868	431344
Blue Mountain	1608	3072	374400

Još je interesantnije da su procesori u tim računalima vezani na različite načine. Naime, idealno bi bilo da je svako računalo vezano sa svakim od preostalih računala, ali to je iz sličnih razloga kao i kod računala sa zajedničkom memorijom neostvarivo. Dovoljno je da jedan procesor "preko posrednika" može dostupiti do bilo kojeg preostalog, za što mu je potreban konstantan broj, ili maksimalno $\log_2 p$ veza po procesoru.

Za dva od tri najbrža računala javno je poznata arhitektura (barem njena gruba skica): procesori Intel ASCI Red računala vezani su u polje procesora, a procesori IBM ASCI Blue Pacific računala u tzv. omega mrežu.

Zanimljivo je da za treće iz klase najbržih računala na svijetu SGI ASCI Blue Mountain točna arhitektura nije javno poznata. Poznato je samo da se sastoji od 48 skupina (engl. clusters) od po 128 procesora koji imaju zajedničku memoriju.

Što se predviđa za bližu budućnost? Sredinom 2000. godine u Lawrence Livermore National Laboratory-ju predviđeno je instaliranje IBM računala snage 10 Tflopsa (iz milja nazvanog Baby Huey).

2. Paralelne arhitekture i modeli programiranja

Da bi odgovarajuća arhitektura pokazala punu snagu, moramo iskoristiti paralelizam, lokalnost i protočnost (prijevod od pipelining).

2.1. Programski model

Svaka paralelna arhitektura mora omogućavati tri funkcije:

- paralelizam – procesori moraju moći raditi istovremeno;
- komunikaciju među procesorima – procesori moraju moći između sebe razmjenjivati informacije;
- sinkronizaciju – na primjer: procesori se "slažu" oko vrijednosti neke varijable, ili vrijeme početka i kraja rada.

Programski model je sve ono što omogućava korisniku direktno programiranje. Na primjer, u programski model pripadaju: programski jezik, kompajleri, programske biblioteke, run–time sistem, itd.

Povijesno, ljudi su prvo dizajnirali paralelnu arhitekturu, a zatim programski model koji joj je odgovarao. Time je programski model bio usko povezan sa funkcijama paralelne arhitekture. Zatim je vrijeme pregazilo dotičnu arhitekturu, a programi nisu radili i korisnici su ponovno počinjali s razvojem programa.

Ipak, ljudi su se dosjetili da je mnogo jeftinije programski model razviti neovisno o arhitekturi, nego stalno mijenjati programe. Mijenjali i prilagođavali bi se kompajleri i biblioteke, tako da odgovaraju promjenama u organizaciji računala.

Na primjer, Connection Machine FORTRAN (CMF) u originalu je bio razvijen za Thinking Machines CM–2 računalo i bio je usko povezan s njegovom SIMD arhitekturom. Slijedeća generacija CM računala (CM–5) omogućavala je MIMD paralelizam, ali CMF kompajler je modificiran tako da su stari programi radili i na novoj arhitekturi. Jasno je da je pri tome način generiranja koda samog kompajlera značajno promijenjen.

Ovakvim načinom gledanja, koji razdvaja računala od programskog modela, omogućava se portabilnost programa (može se lako prenositi s jednog računala na drugo). Ipak, postoje i primjedbe na ovu filozofiju. Naime, moguć je veliki gubitak u performansi određenih računala, ako se programski model ne slaže dobro s njegovom arhitekturom. Jasno je da se svi programski modeli ne mogu jednako dobro implementirati na svim arhitekturama. Ostala su još mnoga neodgovorena pitanja vezana uz tu temu, pa su postojeći kompajleri, biblioteke i run–time sistemi obično: nepotpuni, "buhati" i neefikasni (ili u kombinaciji).

2.2. Paralelizam, komunikacija i sinkronizacija

2.2.1. Paralelizam

U prošlom poglavlju spomenuli smo dva osnovna tipa paralelnih računala:

- SIMD – u tu klasu pripadaju i vektorska (jednoprocesorska) računala, kao što je Cray T–90, CM–2 i jednoprocesorsko računalo RS 6000/590, gdje su instrukcije zbrajanja i množenja u pipelineu i kontroliraju se jednom stopljenom instrukcijom množenja i zbrajanja (MAF).
- MIMD – u tu klasu pripadaju gotovo sva komercijalna paralelna računala.

Uočimo da jedno računalo može pokazivati i SIMD i MIMD vrstu paralelizma, samo na drugačijem nivou (primjer, Cray T–90).

2.2.2. Komunikacija

Prije nego što opišemo načine komunikacije, uvedimo imena za različite memorijske lokacije koje instrukcijama možemo dohvatiti.

Memorija običnog sekvencijalnog računala sastoji se od **riječi**, a svaka riječ ima jedinstvenu **adresu**. Međutim, paralelno računalo može imati i više memorija, nazovimo ih Mem_i. Kao što smo u prošlom poglavlju najavili, kod SIMD i MIMD računala postoje dva načina organizacije memorije. Razlikujemo:

- računala sa **zajedničkom memorijom** (shared address space machines), tj. takvo računalo ima jednu globalnu memoriju za sve procesore;
- računala sa **razdijeljenom (distribuiranom) memorijom** (distributed address space machines), tj. svaki procesor takvog računala ima svoju lokalnu memoriju koja ne može biti adresirana od strane drugog procesora.

Kod računala sa zajedničkom memorijom, svaka riječ u memoriji ima jedinstvenu adresu, zajedničku za sve procesore. Na primjer, ako procesori $Proc_1$ i $Proc_3$

izvode instrukciju "load r1, 37", onda će se isti podatak iz zajedničke memorije (sa adrese 37) transportirati u registar $r1$ procesora $Proc_1$ i registar $r1$ procesora $Proc_3$.

Kod računala sa razdijeljenom memorijom, adrese memorije su lokalne za svaki procesor. Ponovno, neka procesori $Proc_1$ i $Proc_3$ izvode instrukciju "load r1, 37". Tada će procesor $Proc_1$ pročitati podatak sa adrese 37 iz memorije Mem_1 i transportirati u registar $r1$ procesora $Proc_1$, a procesor $Proc_3$ će to isto napraviti s podatkom sa adrese 37 iz memorije Mem_3 i transportirati u registar $r1$ procesora $Proc_3$. Jasno je da za računala s razdijeljenom memorijom, moraju postojati i druge instrukcije za transfer podataka (pored elementarnih "load", "store"), koje omogućavaju transfer podataka između procesora.

U današnje vrijeme ima uspješnih računala u obje ove klase memorija. Grubo govoreći, računala sa zajedničkom memorijom, uobičajeno, imaju bržu komunikaciju nego ona sa razdijeljenom memorijom i na prirodan način su vezana s programskim modelom (shared memory model). Zbog toga ih je obično lakše programirati, nego računala sa razdijeljenom memorijom. Programski model za računala sa razdijeljenom memorijom osniva se na prenošenju poruka (message passing). Dakako, računala sa zajedničkom memorijom teže je napraviti, ili su barem skuplja (po procesoru, za broj procesora ≥ 32), nego računala sa razdijeljenom memorijom.

Važno svojstvo komunikacije je i njena **cijena**. Pretpostavimo da s jednog procesora na drugi šaljemo n riječi podataka. Promatramo slanje n riječi istovremeno, zbog toga što hijerarhijska građa memorije računala pokazuje, da je jeftinije slati neku skupinu susjednih riječi odjednom, nego jednu riječ za drugom. Najjednostavniji model za vrijeme potrebno za takvo slanje je slijedeći:

$$\texttt{vrijeme za slanje } n \texttt{ riječi} = \texttt{latentnost} + \frac{n}{\texttt{širina vrpce}} \quad ,$$

gdje `latentnost` (latency) označava vrijeme za slanje jedne prazne instrukcije (jedinica – sekunde). `Širina vrpce` (bandwidth) (jedinica – riječi/sekundi) mjeri brzinu kojom riječi mogu proći kroz čitavu mrežu povezivanja procesora. Na primjer, poznata je formula za pipelining n riječi, kroz "pipu" sa s faza, ako svaka faza u pipeline-u traje t sekundi:

$$(s-1)t + nt = \texttt{latentnost} + \frac{n}{\texttt{širina vrpce}} \quad .$$

Mreža povezivanja ponaša se točno kao pipeline u prethodnom primjeru, "gura" podatke kroz mrežu određenom brzinom.

Da bismo imali osjećaj reda veličine cijene komunikacije, promjenimo jedinice za mjerenje latentnosti i širine vrpce u ciklus, odnosno riječi/ciklusu. Prisjetimo se da je ciklus vrijeme koje je potrebno računalu za izvršavanje jedne osnovne operacije, na primjer, jednog zbrajanja. Na računalima sa zajedničkom memorijom, gdje je komunikacija brža, vrijeme latencije može biti u stotinama ciklusa. S druge strane,

za mrežu radnih stanica vezanih Ethernet-om (PVM software), latencija može biti 10^5 ciklusa. Vrijeme po riječi (recipročno od širine vrpce) je, uobičajeno, mnogo niže nego latentnost (od $\mathcal{O}(10)$ do $\mathcal{O}(1000)$ MBytes/sekundi). Zbog toga je mnogo efikasnije poslati jednu veliku poruku, nego mnogo malih. Sporost komunikacije diktira da se dobri i loši paralelni algoritmi primarno razlikuju u količini komunikacije, a manje u količini računanja.

Katkad se računala sa zajedničkom memorijom sastavljaju kao skup klastera manjih računala sa zajedničkom memorijom. U tom slučaju, mnogo je brže pristupati memoriji računala iz vlastitog klastera, nego memoriji nekog drugog klastera. Takva računala zovu se računala s neuniformnim pristupom memoriji (Non Uniform Memory Access) i za simulaciju zahtijevaju najmanje dvije različite latentnosti i dvije različite širine vrpce (za blisku i udaljenu memoriju).

2.2.3. Sinkronizacija

Pod sinkronizacijom dva ili više procesora smatramo "dogovor" oko vrijednosti nekog podatka ili oko vremena. Postoje tri manifestacije sinkronizacije ili njenog pomanjkanja:

- međusobno isključivanje;
- barijere;
- konzistentnost memorije.

Međusobno isključivanje je dozvola samo jednom procesoru da pristupa jednoj memorijskoj lokaciji u jednom vremenskom trenutku. Za ilustraciju, uzmimo program koji računa zbroj p brojeva x_i, $i = 1, \ldots, p$. Pretpostavimo da procesor $Proc_0$ ima sve x_i na raspolaganju. Ostali procesori se ponašaju na slijedeći način: svaki procesor uzme odgovarajući x_i i zbroji ga na zbroj s i vrati procesoru $Proc_0$. Ovisno o redoslijedu pristupa procesora procesoru $Proc_0$, rezultat ovog zbrajanja može biti bilo koja parcijalna suma brojeva x_i. Kao primjer, uzmimo da treba zbrojiti samo dva broja x_1 i x_2. Postoje dvije mogućnosti za izvršavanje tog programa (vertikalni poredak je vremenski poredak!):

Procesor 1		Procesor 2	
`load` s	$(s = 0)$		
		`fetch` s	$(s = 0)$
$s = s + x_1$	$(s = x_1)$	$s = s + x_2$	$(s = x_2)$
`store` s	$(s = x_1)$		
		`store` s	$(s = x_2)$

Naravno, postoji mogućnost da je drugi procesor vremensku jedinicu ispred prvog i tad će rezultat biti x_1. Ovakav način rada procesora je čisto nedeterministički

i zove se **uvjeti trke** (race condition).

Međusobno isključivanje je mehanizam koji onemogućava uvjete trke, dozvolom pristupa varijabli samo od jednog procesora. To se obično implementira "test & set" instrukcijom, koja odgovarajuću riječ stavlja na odgovarajuću vrijednost dok se dovlači stara vrijednost.

Barijera stavlja svaki procesor u stanje čekanja u nekoj točki programa, dok svi ostali procesori ne stignu u tu točku. Barijere se mogu implementirati "test & set" instrukcijom, ali neka računala imaju i poseban hardware za to (CM–5).

Memorijska konzistentnost je vrsta problema vezana uz računala sa zajedničkom memorijom koja imaju cache-e. Pretpostavimo da oba procesora $Proc_1$ i $Proc_2$ učitaju memorijsku lokaciju 37 u svoj cache. Namjena cache-a je ubrzanje dohvaćanja podataka iz spore memorije, na primjer lokacije 37. Nakon čitanja i upotrebe lokacije 37, oba procesora žele spremiti različite podatke natrag u lokaciju 37. Inzistiranje da svaki procesor može dostupiti do bilo koje lokacije izgleda vrlo razumno. Takav način pogleda zovemo **konzistencija sekvencijalnog tipa** (sequential memory consistency). Ali takav pristup je vrlo skup za implementaciju, jer sva pisanja moraju ići u glavnu memoriju, i moraju se ažurirati svi cache-i u računalu (umjesto jednog – lokalnog cache-a). To nas vodi do takozvanog **modela slabije memorijske konzistencije** (weak(er) memory consistency model), koji ostavlja korisniku da izbjegava takve probleme.

2.3. Modeliranje komunikacija na računalima sa razdijeljenom memorijom

2.3.1. Model PRAM računala

Za modeliranje komunikacija na računalima sa razdijeljenom memorijom, kao teoretski model koristi se PRAM (Parallel Random Access Memory (Machine)) model. Takvo (teoretsko) računalo ima svojstvo da mu je komunikacija između proizvoljna dva procesora jednakog trajanja. Osim toga, smatra se da je cijena komuniciranja mnogo manja nego cijena obavljanja osnovnih aritmetičkih operacija.

Takav model je lako programirati i vrlo je omiljen među ljudima koji se bave teoretskim računarstvom. Da bismo potpuno opisali takav model, potrebno je još definirati što se zbiva, ako procesori pokušavaju istovremeno čitati iz jedne memorijske lokacije, ili u nju pisati.

Postoji nekoliko podmodela PRAM računala (prema broju procesora koji istovremeno mogu čitati i pisati u jednu memorijsku lokaciju):

- EREW (Exclusive Read – Exclusive Write) – samo jedan procesor može čitati

i samo jedan pisati istovremeno u jednu memorijsku lokaciju;

- CREW (Concurrent Read – Exclusive Write) – jedan procesor može pisati, a više njih čitati iz jedne memorijske lokacije;
- ERCW (Exclusive Read – Concurrent Write) – jedan procesor može čitati, a više njih pisati u jednu memorijsku lokaciju;
- CRCW (Concurrent Read – Concurrent Write) – više procesora može čitati i više njih pisati u jednu memorijsku lokaciju.

Jasno je da je istovremeno čitanje bezopasna operacija (iako ju je teško fizički realizirati), jer se ne mijenja podatak koji se učitava. Opasna je operacija istovremeno pisanje.

Postoje 3 "trika" kako se (teoretski) razrješava problem istovremenog pisanja:

- samo procesor s najmanjim indeksom može pisati;
- upisuje se zbroj svih brojeva koje su procesori htjeli upisati (upis = zbrajač);
- ako svi procesori žele upisati isti broj, upisuje se, a ako ne, lokacija ostaje nepromijenjena (upis = uspoređivač).

Primijetimo da posljednja dva načina razrješavanja problema pisanja u sebi sadrže još dvije "nevidljive" operacije – prva je zbrajanje, a druga uspoređivanje onoliko brojeva koliko ima procesora i to u konstantnom vremenu.

Primjer 2.3.1.

Simulirajte algoritam zbrajanja brojeva $x_1, \ldots, x_p$*, ako na raspolaganju imamo CRCW PRAM računalo s* p *procesora, a broj* x_i *je smješten lokalno u procesoru* $Proc_i$*. Rezultat se pohranjuje u lokaciju* s *(koja je na početku bila jednaka* 0*).*

Svi procesori (sinkrono) izvršavaju slijedeći program:

Procesor $Proc_i$

`load` s	$(s = 0)$
$s = s + x_i$	$(s = x_i)$
`store` s	$(s = \sum x_i)$

Posljednja naredba u algoritmu je implicitno zbrajanje svih rezultata, jer svi procesori pokušavaju pisati u istom vremenskom trenutku.

Primjer 2.3.2.

Pretpostavimo da imamo na raspolaganju PRAM (CRCW) računalo s $p = n^2$ *procesora. Istovremeno pisanje razrješavamo zbrajanjem brojeva koje želimo upisati. Na takvom računalu sortiramo polje* x *koje sadrži* n *različitih brojeva, a sortirano polje upisujemo u polje* y*.*

Algoritam sortiranja je vrlo jednostavan. Označimo procesore s $Proc_{ij}$. Procesor $Proc_{ij}$ ispituje da li je $x(i) > x(j)$ i upisuje rezultat u i–tu lokaciju polja count (koja je na početku bila jednaka 0).

Jasno je da (ako nemamo sve brojeve jednake), više procesora upisuje istovremeno rezultat u $count(i)$, pa se rezultati zbrajaju. Na kraju, u lokaciji $count(i)$ upisano je od koliko je brojeva broj $x(i)$ veći. Dakle, njegovo novo mjesto u sortiranom polju je $count(i) + 1$.

Svi procesori obavljaju slijedeći program:

Procesor $Proc_{ij}$
if $x(i) > x(j)$ **then** $count(i) = 1$
if $j = 1$ **then** $y(count(i) + 1) = x(i)$

Primijetimo da drugi redak u prethodnom programu obavlja samo p procesora, zato da bi se izbjeglo višestruko pisanje u odgovarajuće lokacije.

Na EREW računalu, kao najslabijem u klasi, možemo simulirati i preostala tri podmodela.

Istovremeno čitanje iz iste lokacije simuliramo procedurom **emitiranja** (broadcasting). Emitiranje se vrši na slijedeći način:

$Proc_1$ pročita vrijednost x
$Proc_1$ pošalje vrijednost x procesoru $Proc_2$
$Proc_1$ i $Proc_2$ pošalju x procesorima $Proc_3$ i $Proc_4$
$Proc_1$, $Proc_2$, $Proc_3$ i $Proc_4$ pošalju x $Proc_5$, $Proc_6$, $Proc_7$ i $Proc_8$
itd., sve dok svi procesori ne dobiju x.

Uočimo da je trajanje emitiranja jednako duljini slanja jedne informacije t, pomnoženoj s vremenskim trajanjem slanja. Označimo s k broj koraka slanja, potreban da svih p procesora dobije vrijednost x. Kad $Proc_1$ pročita x, x je dostupan samo njemu. U slijedećem koraku x je dostupan dvama procesorima, itd., sve dok nakon $k - 1$ koraka nije dostupan svima. Dakle, vrijedi

$$2^0 + 2^1 + 2^2 + \cdots + 2^{k-1} = \frac{2^k - 1}{2 - 1} = p \quad ,$$

odakle odmah slijedi

$$k = \mathcal{O}(\lg(p)) := \mathcal{O}(\log_2(p)) \quad .$$

Dakle, za emitiranje je potrebno $t \cdot \mathcal{O}(\lg(p))$ vremenskih trenutaka.

Na sličan način može se simulirati i višestruko pisanje, postupkom obrnutim od emitiranja. Pretpostavimo da se višestruko pisanje razrješava pisanjem, ako su svi brojevi x_i isti. Ako se istovremeno pisanje razrješava pisanjem zbroja svih brojeva, algoritam je sličan (vidjeti kasnije). Razrješavanje konflikta pisanja, dozvolom

pisanja procesoru s najmanjim indeksom, mnogo je jednostavnije (sami). Zbog jednostavnosti, pretpostavimo da je broj procesora p potencija od 2:

za sve $i = 1, \dots, p/2$, procesori $Proc_i$ i $Proc_{i+p/2}$ uspoređuju x_i
i definiraju (boolean) $y_i := (x_i = x_{i+p/2})$
za sve $i = 1, \dots, p/4$, procesori $Proc_i$ i $Proc_{i+p/4}$ uspoređuju x_i
i definiraju (boolean) $y_i := (x_i = x_{i+p/4})$ `and` y_i `and` $y_{i+p/4}$
za sve $i = 1, \dots, p/8$, procesori $Proc_i$ i $Proc_{i+p/8}$ uspoređuju x_i
i definiraju (boolean) $y_i := (x_i = x_{i+p/8})$ `and` y_i `and` $y_{i+p/8}$
itd., sve dok se problem ne svede na jedan procesor.

Uočimo da je u drugom koraku algoritma potrebno ispitati da li je, u prethodnom koraku, varijabla y_i bila `true` ili `false`, jer moglo bi se dogoditi, na primjer, $x_i = x_{i+p/4} = x_{i+3p/4} \neq x_{i+p/2}$! U svakom koraku prethodnog algoritma prepolavlja se broj procesora koji ispituju da li je bila jednaka vrijednost raznih x_i–ova. Zbog toga je i za ovaj algoritam potrebno

$$k = \mathcal{O}(\lg(p))$$

koraka. Prvi procesor će moći upisati rezultat x_1, ako je $y_1 =$ `true`. Ako konflikt pisanja razrješavamo pisanjem zbroja, u svakom koraku prethodnog algoritma, umjesto da se uspoređuju brojevi, oni se zbrajaju.

2.3.2. Log$P$ model

Umjesto detaljnog modela komuniciranja usko vezanog uz odgovarajuću arhitekturu, možemo izdvojiti bitne komponente koje utječu na komunikaciju.

Takav, kompromisni model zovemo Log$P$ model. Ime mu potječe od početnih slova parametara (engleskih) i nema veze s logaritmima! Ovaj model ima 4 parametra i to su:

- `L` latentnost (latency) – vrijeme potrebno za slanje poruke fiksne duljine (kroz mrežu) od procesora koji šalje poruku do procesora koji prima poruku;
- `o` višak vremena (overhead) – dodatno vrijeme koje procesor mora potrošiti ili za slanje ili za primanje paketa. Na primjer, to je vrijeme potrebno za kopiranje primljene poruke ili vrijeme potrebno za pakiranje poruke za slanje. Drugim riječima, vrijeme potrebno da se pošalje poruka s jednog procesora na drugi jednako je

  $$2 \cdot \mathtt{o} + \mathtt{L} \quad ,$$

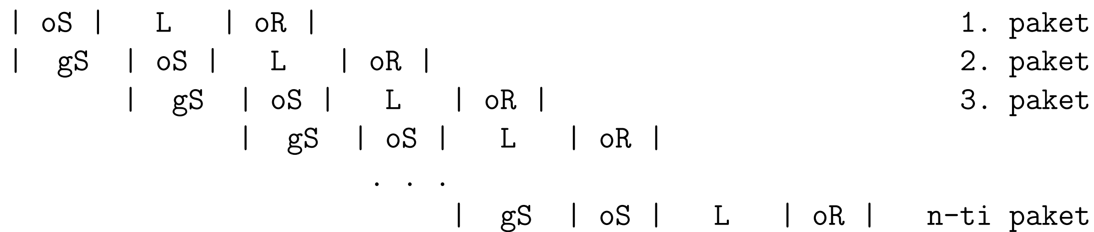
  tj. jedan `o` potječe od pakiranja poruke, jedan od raspakiravanja, a ostatak je putovanje kroz mrežu;

$\mathtt{g}$ razmak (gap) – minimalno vrijeme potrebno jednom procesoru između dva slanja poruke. Recipročna vrijednost od $\mathtt{g}$ je širina pojasa komunikacije po procesoru (communication bandwidth) za taj procesor;

p – broj procesora.

Dodatno, pretpostavljamo da mreža ima konačni kapacitet, tj. najviše $\mathtt{L}/\mathtt{g}$ poruka može putovati s jednog procesora na drugi. Ako neki od procesora pokuša poslati više poruka od te vrijednosti, poslat će se prvih $\mathtt{L}/\mathtt{g}$ poruka, a ostale će čekati slijedeće slanje.

Uobičajeno pojednostavljenje $\mathtt{Log}P$ modela je da se smatra da ne prelazimo dozvoljeni kapacitet mreže i da se velike poruke od n paketa (manjih poruka) šalju odjednom. Neka je $\mathtt{oS}$ dodatno vrijeme za slanje poruke, $\mathtt{oR}$ dodatno vrijeme za primanje poruke, $\mathtt{gS}$ razmak između slanja poruke (na procesoru koji šalje poruku), a $\mathtt{L}$ latentnost. Tada se slanje poruke s n paketa vremenski odvija ovako:



Proučavanjem prethodnog dijagrama dobivamo vrijeme potrebno za slanje n paketa:

$$t = (n-1) \cdot \mathtt{gS} + \mathtt{oS} + \mathtt{oR} + \mathtt{L} = n \cdot \mathtt{gS} + (\mathtt{oS} + \mathtt{oR} + \mathtt{L} - \mathtt{gS}) \quad .$$

Ako smatramo da su dodatna vremena za slanje i primanje poruke jednaka i da je razmak slanja poruka jednak razmaku primanja poruka, onda je prethodna formula jednaka:

$$t = n \cdot \mathtt{g} + (2 \cdot \mathtt{o} + \mathtt{L} - \mathtt{g}) := \alpha + n \cdot \beta \quad .$$

U slijedećoj tablici dane su vijednosti parametara α i β za neka računala. Duljina paketa koji se šalje je 8 byte-a. α i β su normalizirani tako da je jedinično vrijeme – vrijeme potrebno za jednu floating–point operaciju kod množenja 2 matrice s double precision elementima (8 byte-a). Usporedbe radi, dana je i performansa u Mflops-ima za to matrično množenje.

Računalo	α	β	Matmul	Software
Alpha + Ethernet	38.000	960	150	PVM
Alpha + FDDI	38.000	213	150	PVM
Alpha + ATM1	38.000	62	150	PVM
Alpha + ATM2	38.000	15	150	PVM
HPAM + FDDI	300	13	20	
CM–5	450	4	3	CMMD
CM–5	96	4	3	Active Mess.
CM–5 + VU	14.000	103	90	CMMD
iPSC/860	5.486	74	26	
Delta	4.650	87	31	
Paragon	7.800	9	39	
SP–1	28.000	50	40	
T 3D	27.000	9	150	Large Mess. (BLT)
T 3D	100	9	150	read/write

Objasnimo još neke oznake u tablici. Prva četiri podatka odnose se na mrežu DEC Alpha računala vezanih u mrežu različitim tipovima lokalnih mreža – od Ethernet-a (1.25 Mbyte-a u sekundi) do ATM (80 Mbyte-a u sekundi). HPAM je HP računalo s Active Message sistemom. CM–5 računalo predstavljeno je s 3 podatka – u varijanti s vektorskim procesorom i s dva različita sistema za prenošenje poruka. IBM SP–1 računalo sastoji se od RS 6000/370 procesora, a T3D je napravljen u Cray-u, a zapravo je mreža DEC Alpha računala s Crayevim upravljanjem memorije.

Zaključimo razmatranje prethodne tablice. Uočimo da je $\alpha \gg 1$ i $\beta > 1$, što pokazuje da je komunikacija vrlo spora, pa treba napraviti reda veličine barem 1000 floating–point operacija između dvije komunikacije. Osim toga, vrijedi $\alpha \gg \beta$, što pokazuje da je bolje poslati nekoliko većih poruka, nego više manjih.

2.4. Neke komunikacijske mreže na računalima sa razdijeljenom memorijom

Dizajniranje računala je uvijek "trgovina" između brze komunikacije (što je skupo, jer zahtijeva mnogo "žica") i cijene.

Mreža računala se tipično sastoji ne samo iz "žica" za povezivanje, nego i procesora za komunikaciju (routing processors). Ti procesori mogu biti vrlo jednostavni, ali mogu biti jednaki kao i procesori koje koristimo za računanje (Intel Paragon za obje svrhe koristi i860). Starija računala nisu imala procesore za komunikaciju. Danas, većina računala koristi posebne komunikacijske procesore, tako da oni mogu raditi u paraleli s procesorima koji računaju.

Mreže se mogu klasificirati po trima kriterijima:

- **topologiji** – koji procesori su direktno povezani s kojim;
- **dinamička – statička mreža** – da li se topologija mreže može dinamički mijenjati;
- **putu (algoritmu) slanja poruke** – način na koji se izabire put za poruku koja putuje s jednog procesora na drugi.

Stvarna računala su, uglavnom, hibridna i katkad imaju svojstva koja pripadaju različitim kategorijama.

Za svaku mrežu možemo definirati dvije tipične veličine:

- **dijametar** – maksimalni broj procesora (za prenošenje poruka) kroz koji poruka mora proputovati da bi stigla na odredište. Dijametar mjeri maksimalni razmak (broj žica među susjednim procesorima) za prenošenje poruka s jednog procesora na drugi;
- **širina presjeka** (bisection width) – najveći broj poruka koje mogu istovremeno biti poslane (bez da se koriste iste "žice", odnosno, istovremeno isti procesori za komunikaciju), bez obzira kojih $p/2$ procesora šalje poruke preostalim $p/2$ procesorima. Drugi način definicije – to je najmanji broj "žica" koje se moraju presjeći, da bi se mreža raspala na 2 nepovezana dijela.

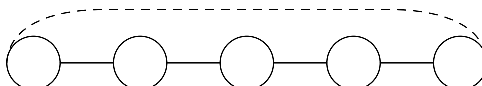
$\texttt{Log}P$ model i $\alpha + n \cdot \beta$ model komunikacija ignoriraju dijametar mreže, tj. ignoriraju činjenicu da se bližim susjedima u mreži poruka može prenijeti brže, nego udaljenijim (uzimajući u obzir samo prosječni α i β). To odražava stvarno stanje na modernim arhitekturama računala: najveći dio sporosti komunikacije dolazi zbog utroška vremena u software-u na ishodištu i odredištu, a mnogo manje zbog latentnosti mreže između. Posebno, to znači da u dizajnu algoritma, vrlo često možemo ignorirati topologiju mreže.

$\texttt{Log}P$ model samo djelomično (indirektno) koristi širinu presjeka i to kod ograničenosti kapaciteta mreže.

2.4.1. Statičke mreže

Linearno polje, prsten

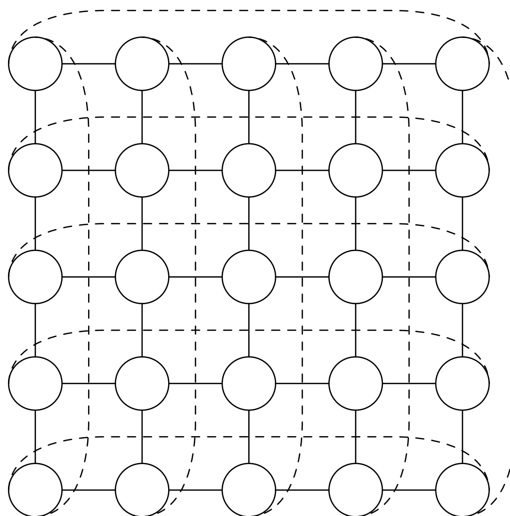
Linearno polje od p procesora povezano je dvosmjernim "žicama" kao na slici:



Ako su povezani prvi i zadnji procesor (na slici crtkano), takvo jednodimenzionalno polje zovemo prsten. U slučaju jednodimenzionalnog polja, dijametar mreže je p, a širina presjeka 1. U slučaju prstena, dijamatar je $p/2$, a širina presjeka je 2.

Dvo– i višedimenzionalna polja, torusi

Dvodimenzionalno polje od $p = q^2$ procesora poveznano je dvosmjernim "žicama" kao na slici:



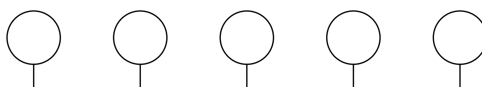
Ako su povezani prvi i zadnji procesor u svakom retku i stupcu (na slici crtkano), takvo dvodimenzionalno polje zovemo torus. U slučaju dvodimenzionalnog polja, dijametar mreže je $2q = 2 \cdot \sqrt{p}$, a širina presjeka $q = \sqrt{p}$. Naime, kombinatorika kaže da je najdalja komunikacija između dijagonalnih procesora. Za putovanje iz gornjeg lijevog procesora u donji desni, treba napraviti $q - 1$ korak udesno i $q - 1$ korak dolje (bez obzira na poredak koraka desno i koraka dolje). Pri tome se obiđe točno $2q - 1$ procesora (računajući polazni i završni).

Generalizacija na višedimenzionalna polja je očita. Ako imamo d–dimenzionalno polje sastavljeno od p procesora, njegov je dijametar $d \cdot p^{1/d}$, a širina presjeka $p^{(d-1)/d}$. Torusi su nešto bolje povezani. Arhitekturu dvodimenzionalnog polja procesora koristi Intel ASCI Red. Procesori Intel Paragona bili su vezani u dvodimenzionalni, a procesori Crayevog modela T3D u trodimenzionalni torus.

2.4.2. Dinamičke mreže

Sabirnica (bus)

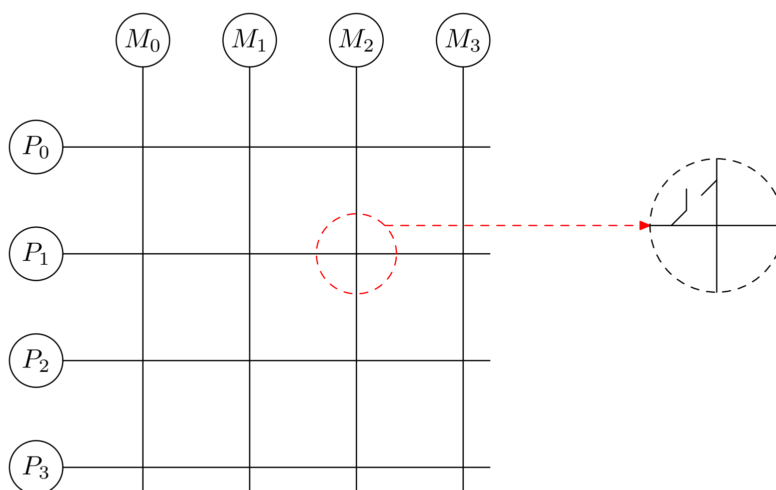
Sabirnica je najjeftinija i najjednostavnija dinamička mreža. Svi procesori dijele jednu sabirnicu (bus), kroz koju, u jednom vremenskom trenutku, podatke može slati najviše jedan procesor. Dijametar mreže je 1, jer je svaki procesor direktno povezan sa svim ostalima. Širina presjeka je, također 1, jer je dovoljno presjeći tu jednu jedinu žicu da se mreža raspadne u dvije komponente.



Na takav način se mogu povezati mreže radnih stanica, a sabirnica koja se koristi je Ethernet. Nešto pametnije mreže takve vrste imaju i male sklopke kojima su procesori povezani na sabirnicu (na primjer, ATM mreža).

Rešetka (crossbar, Xbar)

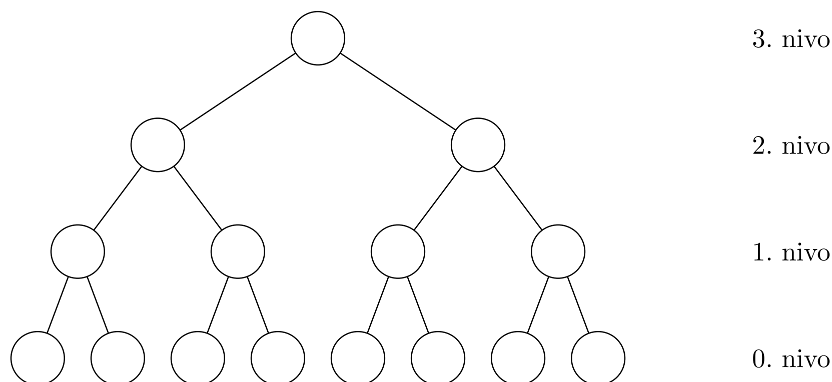
Rešetka je najskuplja dinamička mreža – svaki je procesor direktno povezan sa svakim (analogon potpunog grafa). Koristi se kod velikih računala, ali i kod manjih, gdje dolazi kao komponenta hibridne mreže. Rešetka ima p procesora, ali i p^2 sklopki (u p^2 križanja), tako da može biti povezana bilo koja permutacija $p/2$ procesora s bilo kojom permutacijom preostalih $p/2$. Zbog toga je širina komuniciranja $p/2$. Dijametar mreže je 1, jer bilo koja dva procesora mogu biti direktno spojena.



2.4.3. Hijerarhijske mreže

Stabla, debela stabla, piramide, debele piramide

Binarno stablo (potpuno balansirano) sastoji se od $p = 2^{q+1} - 1$ procesora složenih u nivoe od 0 do q, povezanih tako da svi procesori (osim korijena) imaju jednog oca, a svi procesori (osim listova) imaju dva sina.



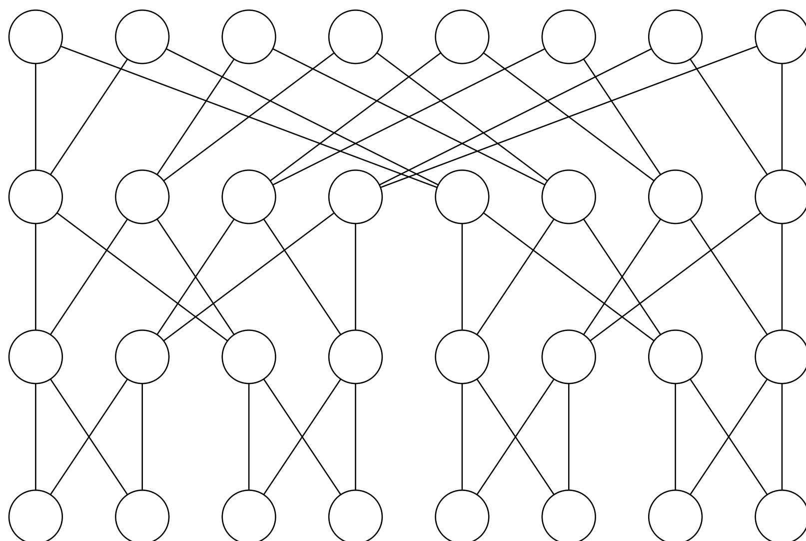
Uobičajeno je da su samo listovi (procesori na nivou 0) procesori za računanje, a svi ostali su komunikacijski procesori. Dijametar binarnog stabla je dvostruka dubina binarnog stabla $2 \cdot q = 2 \lg p$, što odgovara slanju podataka s najlijevijeg u najdesniji list. Širina presjeka je približno 1, jer, ako $p/2$ lijevih procesora pokušava poslati podatke u $p/2$ desnih procesora, u korijenu će nastati zastoj.

Zbog toga, trebalo bi "pojačati" veze između nivoa. Ako "žice" između nivoa k i $k+1$ imaju dvostruki kapacitet onih koje povezuju nivoe $k-1$ i k, onda smo otklonili usko grlo. Takva arhitektura se zove debelo stablo (fat tree). Takvu arhitekturu koristi CM–5 računalo.

Na primjer, za CM–5, brzina (širina pojasa) komunikacije je

za 4 najbliža procesora	20 Mbyte-a / sekundi
za 16 najbližih procesora	10 Mbyte-a / sekundi
za ostale procesore	5 Mbyte-a / sekundi

Povećani kapacitet "žica" među nivoima uobičajeno se ostvaruje tako da djeca imaju dva ili više očeva. Ako binarno stablo procesora s $q + 1$ nivoa (s prethodne slike) dopunimo tako da svako dijete ima 2 roditelja, dobivamo mrežu od 2^q isprepletenih binarnih stabala kao na slijedećoj slici



Ovakva arhitektura vrlo je slična slojevitoj mreži zvanoj leptir (vidjeti malo kasnije).

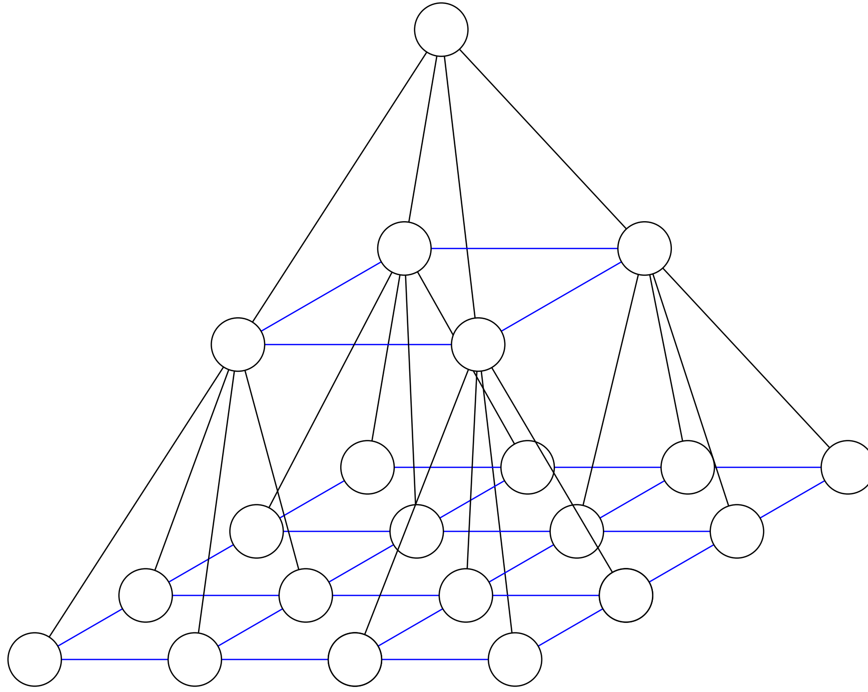
Poruka s procesora $Proc_i$ na procesor $Proc_j$ pomiče se prema korijenu stabla, dok ne naiđe na prvi zajednički prethodnik procesora $Proc_i$ i $Proc_j$. Takvih procesora ima više i svi se nalaze na istom nivou (u svakom procesoru za komunikaciju kroz koji prođe poruka postoje 2 puta). Da bi se balansiralo opterećenje, put između dva procesora bira se slučajno.

Dijametar takve mreže je približno $2 \lg p$, a širina presjeka je približno $p/\lg p$. Odakle to izlazi? Dijametar je isti kao i kod binarnog stabla s $q + 1$ nivoa, tj. $2q$. No, broj procesora u ovoj mreži je $p = (q + 1) \cdot 2^q$, jer svaki nivo ima 2^q procesora. Logaritmiranjem izlazi

$$\lg p = \lg(q + 1) + q \approx q \quad \text{(za iole veće } q\text{)}$$

pa je $2 \lg p \approx 2q$. Točna širina presjeka je $p/(q+1) = 2^q$ (dokažite!), a po prethodnoj relaciji je $q + 1 \approx \lg p$.

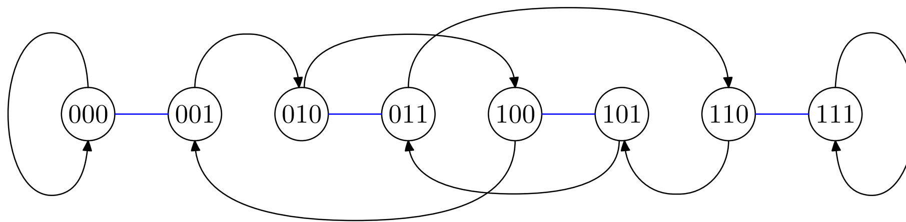
Nema nikakvog razloga zašto ne bismo generalizirali binarna stabla na k–narna, tj. da jedan otac ima točno k sinova. Ako za k uzmemo 4, dobivamo piramidu. Nivoe piramide, također, možemo povezati u mrežu – na primjer, svaki nivo piramide možemo organizirati kao dvodimenzionalno polje procesora.



2.4.4. Slojevite mreže

Savršeno miješanje (perfect shuffle), omega mreže

Mreža savršenog miješanja (s vezama za izmjenjivanje) sastoji se od $p = 2^q$ procesora. Procesori su vezani jednosmjernim vezama (veze savršenog miješanja) i dvosmjernim vezama (veze za izmjenjivanje), kao na slijedećoj slici:



Kad ne bi bilo veza za izmjenjivanje (vezuju parni procesor sa susjednim procesorom čiji je indeks za 1 veći), procesori bi bili razdijeljeni u disjunktne cikluse.

Procesore redom numeriramo od 0 do $p - 1$. Pravilo za konstruiranje veza savršenog miješanja je: procesor $Proc_i$ može direktno slati podatke procesoru $Proc_j$,

ako za indekse i i j vrijedi:

$$j = \begin{cases} 2i & , \text{za } 0 \leq i \leq \frac{p}{2} - 1 \\ 2i + 1 - p & , \text{za } \frac{p}{2} \leq i \leq p - 1 \end{cases} .$$

Još je jednostavnija definicija veza savršenog miješanja, ako umjesto dekadskih indeksa procesora koristimo binarne. Tada procesor $Proc_i$ može poslati podatke procesoru $Proc_j$, ako se j kao binarni broj dobiva cikličkom rotacijom binarnog broja i za jedno mjesto ulijevo.

Tipično, arhitektura savršenog miješanja se koristi na malo drugačiji način. Tom mrežom se obično vežu procesori za komunikaciju. Preciznije, imamo $2p$ komunikacijskih procesora podijeljenih u dvije skupine od p procesora i među njima uspostavljamo mrežu savršenog miješanja.

Uočimo da je u arhitekturi savršenog miješanja najdulji ciklus procesora međusobno povezanih, duljine q. To je jednostavno pokazati, jer za binarni zapis indeksa procesora koristimo točno q bitova (pa ciklus ne može biti veće duljine od q). Stavljanjem, na primjer, jedinice na proizvoljno mjesto (ostalo nule) i cikličkom rotacijom, postižemo ciklus duljine točno q.

Zbog toga, korisno je imati točno q skupina komunikacijskih procesora. Susjedne skupine procesora vezane su u mrežu savršenog miješanja (kao na slici). Takvu mrežu zovemo omega mrežom. Unutarnje skupine komunikacijskih procesora su zapravo jednostavne sklopke (switch-evi) (slika desno gore), koje poruku mogu propustiti "ravno", ako je i–ti bit indeksa procesora koji šalje poruku jednak i–tom bitu indeksa procesora koji prima poruku, ili "ukoso", ako to nije. Primijetimo da ove sklopke glume veze za izmjenjivanje u arhitekturi savršenog miješanja s vezama za izmjenjivanje.

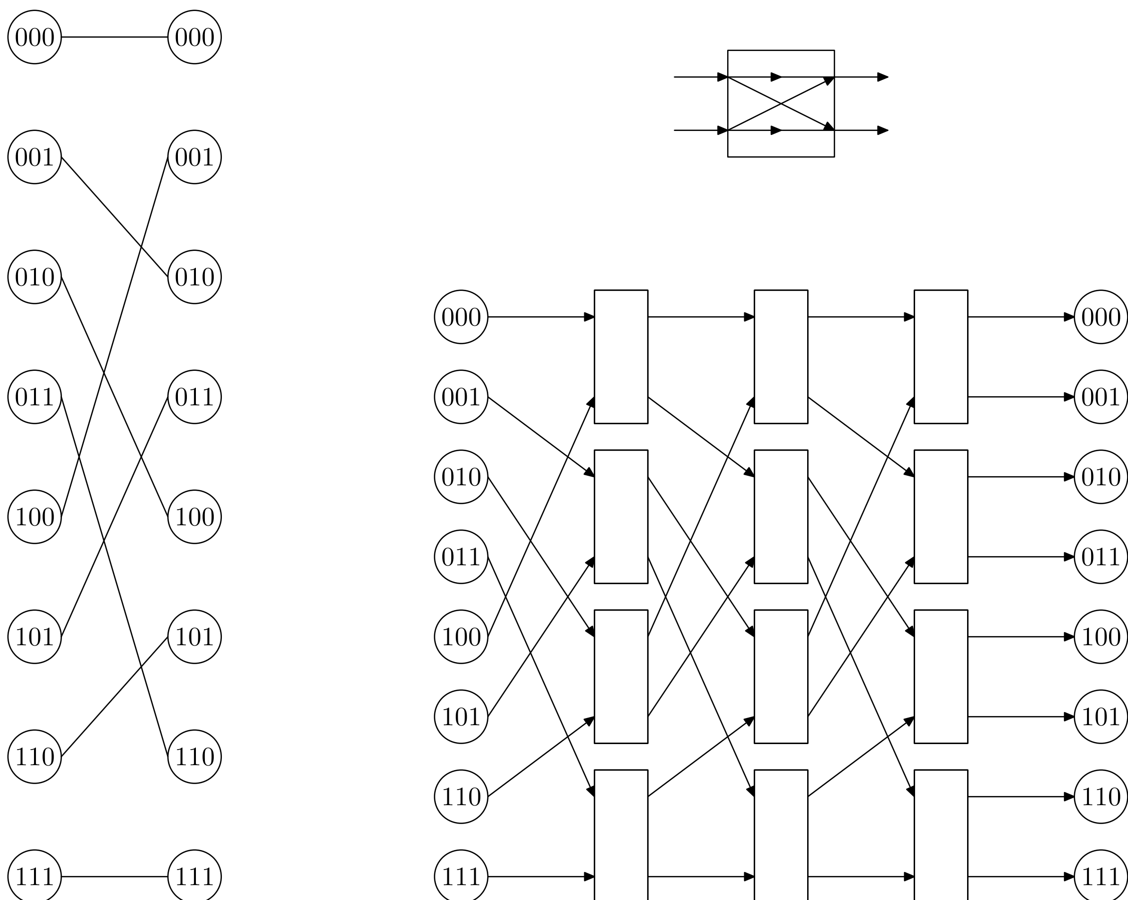
Dakle, poruka dinamički putuje kroz mrežu. Na primjer, ako imamo samo 8 procesora (kao na prethodnoj slici), onda indeks procesora ima samo 3 bita. Pretpostavimo da procesor $s_1s_2s_3$ želi poslati poruku procesoru $d_1d_2d_3$. On će to učiniti na slijedeći način:

Indeks pošiljatelja poruke	$s_1s_2s_3$	
Nakon prvog miješanja	$s_2s_3s_1$	
Nakon prve sklopke	$s_2s_3d_1$	("ravno" za $s_1 = d_1$, "koso" za $s_1 \neq d_1$)
Nakon drugog miješanja	$s_3d_1s_2$	
Nakon druge sklopke	$s_3d_1d_2$	("ravno" za $s_2 = d_2$, "koso" za $s_2 \neq d_2$)
Nakon trećeg miješanja	$d_1d_2s_3$	
Nakon treće sklopke	$d_1d_2d_3$	("ravno" za $s_3 = d_3$, "koso" za $s_3 \neq d_3$)
Indeks primatelja poruke	$d_1d_2d_3$	

Svako "miješanje" u prethodnom algoritmu računanja adresa (indeksa) je

primjena transformacije $s \to \mathrm{cshiftl}\,(s)$ (circular shift left by 1), tj. rotacija binarno zapisanog indeksa s, za jedno mjesto ulijevo.

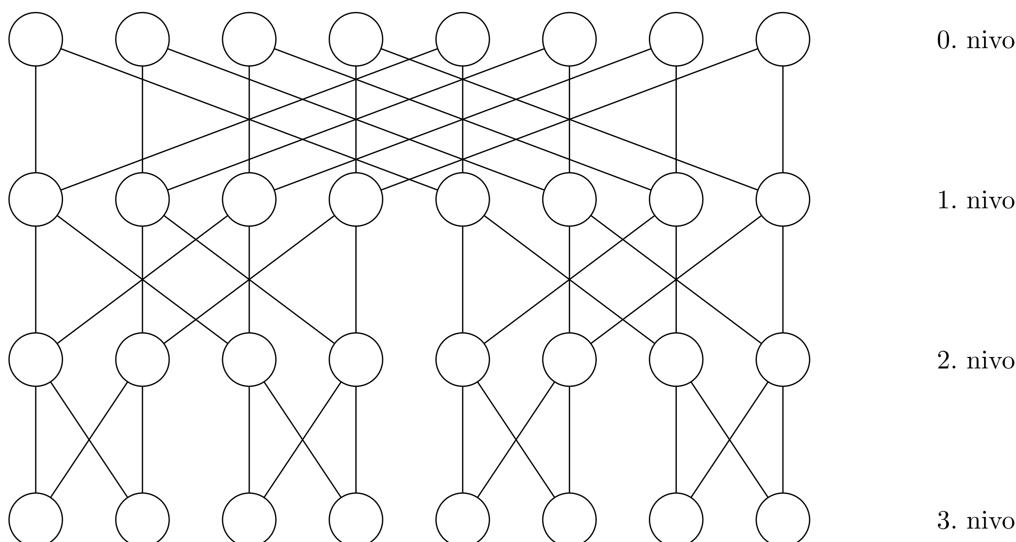
Na slijedećoj slici prikazana je omega mreža (desno dolje) i njeni sastavni dijelovi (sklopke) i mreža savršenog miješanja (lijevo):



Dijametar omega mreže je $q = \lg p$, jer poruka mora proputovati q sklopki. Širina presjeka je p. Takva mreža koristila se kod prvih modela paralelnih računala s početka osamdesetih godina: IBM RP3, BBN Butterfly i New York University Ultracomputera. Danas, omega mrežu koristi jedno od najbržih računala na svijetu – IBM ASCI Blue Pacific.

Leptir (butterfly)

Arhitektura leptira sastoji se od $p = (q+1) \cdot 2^q$ procesora podijeljenih u $(q+1)$ nivoa (indeksiranih od 0 do q). U svakom nivou ima točno 2^q procesora.

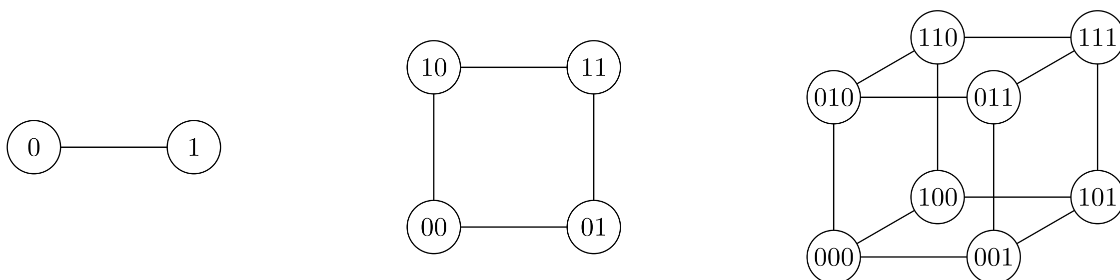


Ako na svakom nivou procesore indeksiramo od 0 do $2^q - 1$, onda je j–ti procesor na i–tom nivou vezan s dva procesora na $(i-1)$–om nivou: s j–tim i onim, čiji se indeks dobiva komplementiranjem i–tog najznačajnijeg bita binarne reprezentacije od j.

2.4.5. Kombinacija slojevite i hijerarhijske mreže

Hiperkocka

q–dimenzionalna hiperkocka ima $p = 2^q$ procesora. Ako procesore indeksiramo od 0 do $p-1$, za binarni zapis njihovih indeksa potrebno je q bitova. Po definiciji, procesori $Proc_i$ i $Proc_j$ su direktno vezani, ako se njihov binarni prikaz razlikuje u točno jednom bitu. Odatle odmah izlazi da je svaki procesor q–dimenzionalne hiperkocke direktno vezan s q drugih procesora.



Slanje poruka vrši se prema Grayevom kodu. q–bitni Grayev kod je permutacija cijelih brojeva od 0 do $2^q - 1$, tako da se binarni zapisi susjeda razlikuju za točno jedan bit. Isto vrijedi i za prvi i posljednji broj u nizu (tj. ciklički).

Grayev kod definira se rekurzivno. Jednobitni Grayev kod je

$$G(1) = \{0, 1\} \quad .$$

Neka je dan q–bitni Grayev kod

$$G(q) = \{g(0), g(1), \dots, g(2^q - 1)\} \quad .$$

$(q+1)$–bitni Grayev kod definira se iz q–bitnog na slijedeći način:

$$G(q+1) = \{0g(0), 0g(1), \dots, 0g(2^q - 1), 1g(2^q - 1), \dots, 1g(1), 1g(0)\} \quad .$$

Na primjer, 3–bitni Grayev kod je

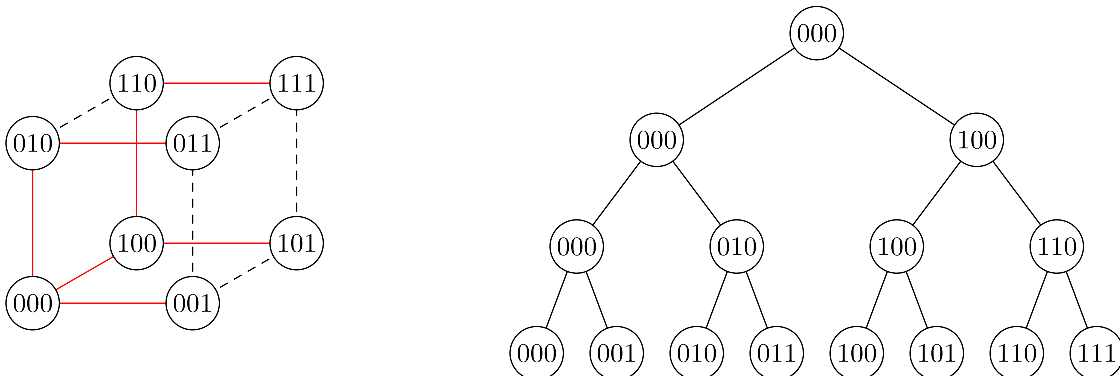
$$G(3) = \{000, 001, 011, 010, 110, 111, 101, 100\} \quad .$$

Zbog dovoljno veza koje postoje u hiperkocki, u nju možemo uroniti jednostavnije mreže koje imaju manje veza po čvoru, kao što je, recimo, binarno stablo ili prsten. Tako možemo lako prenijeti algoritam koji radi na arhitekturi hiperkocke na neku jednostavniju arhitekturu i obratno.

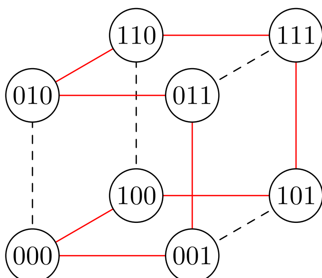
Pretpostavimo da imamo hiperkocku s $8 = 2^3$ procesora. Stablo se lako uranja u hiperkocku na slijedeći način:

- procesor 000 je korijen;
- djecu korijena dobivamo mijenjanjem prvog najznačajnijeg bita u adresi;
- djecu djece dobivamo mijenjanjem drugog najznačajnijeg bita u odgovarajućoj adresi;
- slijedeći nivo dobivamo mijenjanjem trećeg najznačajnijeg bita u odgovarajućoj adresi očeva.

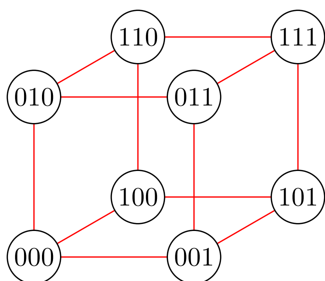
Primijetimo da je takvim postupkom svaki čvor sam sebi i lijevi sin. To neće predstavljati problem, sve dok algoritam koristi samo jedan nivo procesora u stablu, u jednom vremenskom trenutku.

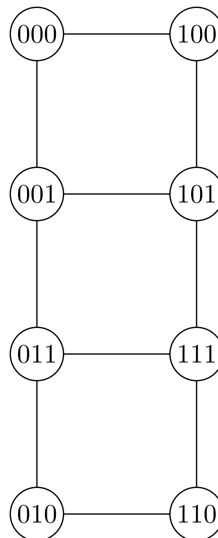


Ulaganje prstena u hiperkocku je još jednostavnije. Slijede se samo veze u odgovarajućem Grayevom kodu.



Također, u kocku možemo ulagati i višedimenzionalna polja procesora, tako da su susjedni čvorovi u mreži, susjedni čvorovi hiperkocke. Jedina je restrikcija da dimenzije takvog višedimenzionalnog polja moraju biti potencije od 2. Za ulaganje k–dimenzionalnog polja s $2^{q_1} \cdot 2^{q_2} \cdots 2^{q_k}$ procesora, potrebna je hiperkocka dimenzije $p = q_1 + q_2 + \cdots + q_k$. Ilustrirajmo to primjerom ulaganja 2–dimenzionalnog polja s $2^1 \cdot 2^2 = 2 \cdot 4$ procesora u 3–dimenzionalnu hiperkocku.





Takav način povezivanja korišten je kod ranijih serija Intelove hiperkocke i CM–2 računala.

3. Mjere ponašanja (efikasnost) paralelnih programa

Za paralelne programe, standardnim mjerama složenosti, a to su vrijeme, odnosno broj računskih operacija i prostor, dodajemo još i neke nove:

- broj paralelnih procesora – što nije jako bitno, osim ako računalo ima mnogo procesora i možemo birati konfiguraciju, tj. koliko će procesora računalo koristiti;
- vrijeme komuniciranja – kao dodatak na standardnu vremensku složenost.

3.1. Ubrzanje, efikasnost, cijena

Osnovni cilj paralelnog računanja je **ubrzanje** obzirom na standardno **sekvencijalno** računanje. Mjere o kojima ćemo govoriti opisuju taj odnos i vrlo često služe kao dobar kriterij za razlikovanje dobrih od loših paralelnih algoritama.

Neka je n neka razumna mjera veličine problema kojeg rješavamo. Na primjer: zbrojite ili sortirajte n brojeva, pomnožite dvije matrice reda n. Neka je p broj procesora koje imamo na raspolaganju za paralelno računanje.

U daljnjem, ignorirat ćemo detalje paralelne arhitekture (veza procesora i komuniciranje) i gledat ćemo globalno samo potrebno vrijeme za rješavanje problema veličine n.

Uvedimo slijedeće oznake:

- $T(p, n)$ – vrijeme za rješavanje problema veličine n na p procesora. Skraćena oznaka $T(p)$ podrazumijeva da sve ovisi i o n;
- $T(1, n)$ – vrijeme za rješenje istog problema sekvencijalnim algoritmom (skraćeno $T(1)$).

Vrijeme $T(1)$ služi kao referentno vrijeme za usporedbu sekvencijalnih i paralelnih algoritama i standardno se uzima da je $T(1)$ vrijeme **najboljeg** sekvencijalnog algoritma (ili barem najboljeg poznatog sekvencijalnog algoritma – ako nije

dokazano da je taj algoritam najbolji). Objasnimo pojam najboljeg sekvencijalnog algoritma i najboljeg poznatog sekvancijalnog algoritma.

3.1.1. Donja i gornja granica

Generalno, ako želimo analizirati neki novi sekvencijalni algoritam, nužno je postaviti dva pitanja:

- da li je optimalan;
- ako nije, kakav je njegov odnos prema najboljem postojećem algoritmu za taj problem.

Odgovor na prvo pitanje dobivamo uspoređivanjem broja operacija u algoritmu u najgorem slučaju, s poznatom donjom granicom za taj problem. Ako je taj broj operacija reda veličine donje granice, algoritam je optimalan.

Primjer 3.1.1.

Donja granica za računanje produkta dvije (opće) matrice reda n je n^2 aritmetičkih operacija.

Produkt te dvije matrice ima n^2 nezavisnih elemenata koje treba izračunati, pa algoritam sigurno zahtijeva n^2 operacija i ne može bolje.

Do danas nije poznat algoritam koji bi dvije (opće) matrice reda n pomnožio u $\mathcal{O}(n^2)$ operacija. Klasično množenje dvije matrice reda n zahtijeva $\mathcal{O}(n^3)$ operacija. Poznata su neka ubrzanja za matrično množenje.

Strassenov algoritam formulira se blokovski i osniva se na uravnoteženju trajanja zbrajanja dvije matrice reda m (što je $\mathcal{O}(m^2)$ operacija), odnosno klasičnog množenja za iste matrice ($\mathcal{O}(m^3)$ operacija).

Osnova algoritma je množenje dvije 2×2 matrice, što se može provesti sa 7 množenja i 15 zbrajanja. U prvi čas, ako smatramo da množenje i zbrajanje realnih brojeva traje jednako, ideja se nije činila naročito privlačnom. Ako pak 2×2 matrice zamijenimo $n/2 \times n/2$ matricama, kao posljedica izlazi da se množenje dvije $n \times n$ matrice može provesti sa 7 matričnih množenja matrica reda $n/2$ i odgovarajući broj zbrajanja matrica reda $n/2$, odnosno rekurzivno:

$$T(n) = \begin{cases} b & , \text{za } n \leq 2 \\ 7\,T(n/2) + an^2 & , \text{za } n > 2 \end{cases} ,$$

gdje su a i b konstante. Rješavanjem prethodne rekurzije dobiva se složenost $\mathcal{O}(n^{\lg 7})$ operacija. Victor Y. Pan radio je osamdesetih godina na generalizaciji Strassenovog algoritma – dijelio je matrice u sve veće blokove i pokušavao smanjiti broj množenja

u odgovarajućem bloku. Pokazao je da se dvije 70×70 *matrice mogu pomnožiti u* 143.640 *skalarnih množenja.*

Asimptotski najbrži, do sad poznati, algoritam konstruirali su Coppersmith i Winograd 1986., a potreban broj operacija je $\mathcal{O}(n^{2.376})$*.*

Zbog toga, u ovom slučaju, za uspoređivanje paralalnog i sekvencijalnog algoritma uspoređujemo sekvencijalni standardni i paralelni standardni algoritam, sekvencijalni Strassenov algoritam i paralelni Strassenov algoritam, itd.

Usko vezano uz množenje matrica, je i rješavanje sistema linearnih jednadžbi $Ax = b$. Na tu temu, dva velika matematičara Peter Alfeld i Nick L. Trefethen su sklopili i okladu 25. 6. 1985. u 100$ oko toga, da li će se do 31. 12. 1994. naći algoritam koji za rješavanje linearnog sistema reda n treba $\mathcal{O}(n^{2+\varepsilon})$ operacija.

Trefethen je okladu platio u veljači 1996. Oklada je obnovljena i na slijedećih 10 godina (do 1. 1. 2006.). Također, sklopljena je i dodatna oklada 13. 2. 1996. na 200$. Da bi dobio tu okladu, Peter Alfeld mora pokazati da ne postoji algoritam koji koristi $\mathcal{O}(n^2)$ operacija. Obratno, da bi dobio tu drugu okladu, Nick L. Trefethen mora konstruirati algoritam koji koristi $\mathcal{O}(n^2)$ operacija, što je, naravno, mnogo teže, nego samo pokazati da on postoji.

Primjer 3.1.2.

Donja granica za broj operacija za sortiranje n *slučajno poredanih brojeva (uspoređivanjem parova brojeva) je* $\mathcal{O}(n \log n)$ *operacija.*

Primijetimo da postoji $n!$ *mogućih rasporeda tih* n *brojeva. Za zapis svih tih rasporeda potrebno je* $\lg n!$ *bitova (svaki bit* 0 *ili* 1*). Budući da vrijedi*

$$n^{n/2} \leq n! \leq n^n \quad ,$$

logaritmiranjem izlazi da je potrebno najmanje $\mathcal{O}(n \lg n)$ *uspoređivanja.*

Postoje mnogi algoritmi koji dostižu tu donju granicu u najgorem slučaju. Takav je, na primjer, heapsort algoritam, pa za njega možemo reći da je optimalan. Jasno je da ćemo u ovom slučaju, paralelne algoritme uspoređivati s vremenom trajanja optimalnog sekvencijalnog algoritma.

3.1.2. Ubrzanje i efikasnost

Standardno gledamo slijedeće dvije mjere za paralelni algoritam.

- **Ubrzanje** (speedup), u oznaci $S(p)$, definiramo s

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

tj. to je mjera koliko puta brže računamo s p procesora, obzirom na samo 1 procesor. Bitno je da je referentni $T(1)$ dobar (najbolji u odgovarajućoj klasi), inače ćemo dobiti nerealno veliko ubrzanje. Idealni paralelni algoritam trebao bi imati ubrzanje p, međutim, kao što ćemo to kasnije vidjeti, to je nerealno.

- **Efikasnost** (efficiency), u oznaci $E(p)$, definiramo s

$$E(p) := \frac{S(p)}{p} \quad ,$$

tj. mjeri se koliko se dobro paralelizira algoritam. U idealnom slučaju, očekujemo $E(p) \approx 1$, tj. svi procesori su non–stop zaposleni i nema nepotrebnih višaka, obzirom na sekvencijalni algoritam.

Jednostavno je uočiti da vrijede neke ograde na $S(p)$ i $E(p)$.

Pseudoteorem 3.1.1. (Brent)

Vrijedi

$$S(p) \leq p \quad , \quad E(p) \leq 1 \quad .$$

Pseudo dokaz:

Jedan procesor može simulirati rad p procesora u najviše p puta više vremena, izvodeći redom, jedan po jedan korak svakog od p sekvencijalnih programa koji tvore paralelni program:

1. korak procesora $Proc_1$;
$\vdots$
1. korak procesora $Proc_p$;
2. korak procesora $Proc_1$;
$\vdots$
2. korak procesora $Proc_p$;
$\vdots$
n. korak procesora $Proc_1$;
$\vdots$
n. korak procesora $Proc_p$;
$\vdots$

pseudo ■

Uz određene teoretske pretpostavke na sekvencijalni i paralelni model računanja, prethodni pseudoteorem može postati pravi teorem.

Ova tvrdnja vrijedi, samo ako se za $T(1)$ uzima najbolji sekvencijalni algoritam. Njen dokaz ide na kontradikciju – jer bi, u protivnom, naša konstrukcija dala bolji sekvencijalni algoritam od polaznog.

Osim toga, bitne pretpostavke na model računanja i algoritam su, na primjer:

- algoritam je deterministički;
- memorija je sva ravnopravna – iste brzine, tj. nema hijerarhije.

To znači da sekvencijalno i paralelno računanje smijemo ravnopravno uspoređivati, po trajanju osnovnih operacija.

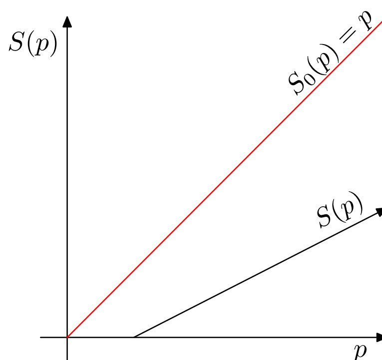
Standardni sekvencijalni model računanja je deterministički Turingov stroj s beskonačnom trakom ili beskonačnim RAM–om. Za paralelni model možemo uzeti p takvih Turingovih strojeva sa zajedničkom kontrolnom jedinicom.

Ako ne vrijede prethodne pretpostavke, može se dogoditi da dobijemo i nadlinearno ubrzanje, tj. kad gledamo $S(p)$ kao funkciju od p, dobijemo da je $S(p) > p$.

Standardni slučajevi kad se to može dogoditi su:

- nedeterministički algoritam – u paralelnom algoritmu uzimamo različite puteve obrade od sekvencijalnog algoritma;
- problem (tj. n) je prevelik za jedan procesor s hijerarhijskom strukturom memorije i on mora češće ići po podatke u sporiju memoriju, nego kad se isti problem raspodijeli na puno procesora. Ovo je vrlo čest i bitan slučaj u praksi.

Naš pseudoteorem kaže da u grafu $(p, S(p))$, pravac $S_0(p) = p$ odozgo ograđuje stvarno ubrzanje. Kvaliteta paralelne implementacije se često mjeri po tome, koliko je ponašanje našeg paralelnog algoritma blizu "idealnog" ubrzanja.



Idealno ubrzanje, uglavnom, nije dostižno — čak ni teoretski, jer se većina algoritama ne može idealno paralelizirati. Osim toga, nije smisleno (niti realno) neograničeno povećavati broj procesora p, ne vodeći računa o veličini problema n.

Naime, ovako definirane mjere $S(p)$ i $E(p)$ ovise, zapravo, samo o p (ne vidi se zavisnost o n). Ispada da, za razne n, ako povećavamo p, sve mjere treba prikazati nad (p, n) ravninom. Da bismo dobili realističnu mjeru ponašanja paralelnog algoritma, treba uzeti odgovarajući odnos između p i n.

Na primjer, ako fiksiramo n, očito je, da povećavanje p vodi, na kraju, u sve manju efikasnost. Za dovoljno velike p, većina procesora je besposlena, tj. vrijedi

$$\lim_{p\to\infty} E(p) = 0 \quad .$$

Ljudi kupuju paralelna računala, ne zato da bi postojeće probleme mogli riješiti brže, nego zato da mogu riješiti **veće** probleme. Zbog toga, u stvarnosti, veličina problema $n(p)$ raste u ovisnosti o broju procesora p.

Tada dobivamo **skalirano ubrzanje** (scaled speedup)

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

i **skaliranu efikasnost** (scaled efficiency)

$$E(p, n(p)) := \frac{S(p, n(p))}{p} \quad .$$

Obično pokušavamo sastaviti takozvani "skalabilni" (engl. "scalable") algoritam, u kojem je efikasnost $E(p, n(p))$ odozdo ograđena pozitivnom konstantom, tj. kako p raste, želimo da je

$$\lim_{p\to\infty} E(p, n(p)) = c > 0 \quad .$$

Na primjer, jedan od najčešćih slučajeva u praksi je, da $n(p)$ raste tako da je potrebna (iskorištena) memorija za algoritam konstantna po svakom procesoru – bez obzira na p. Tj. $n(p)$ raste tako da prostorna složenost paralelnog algoritma "linearno" ovisi o p, odnosno preciznije, svaki procesor koristi konstantnu količinu memorije. Na primjer, za matrično množenje $C = A \cdot B$ matrica reda N na p procesora, uzimamo $3N^2 = n(p) = p \cdot M$ (3 kao faktor, jer spremamo 3 matrice!), gdje je M konstantna količina memorije po procesoru.

Napomenimo još da ovisnost o n **svakako** treba uzeti u obzir. Nije smisleno unaprijed fiksirati n, pa onda odabrati najbolji pripadni sekvencijalni algoritam, baš za taj n, odnosno, praviti paralelni algoritam za taj fiksni n.

Ako želimo razumnu usporedbu, na neki način "n" mora biti parametar oba algoritma – tj. oba algoritma moraju ovisiti o n (bar u nekim granicama $n \leq n_{\max}$), a paralelni algoritam mora ovisiti još i o p (opet u nekim granicama $p \leq p_{\max}$). Obično

se $p_{\max}$ lako nalazi iz n ili $n_{\max}$, ili je jednostavno zadan arhitekturom računala, pa tako određuje $n_{\max}$.

Većina algoritama se ne može idealno paralelizirati, jer neke operacije moraju ići sekvencijalno. Amdahlov zakon daje vrlo jednostavnu ocjenu (odozgo) za ubrzanje koje možemo postići.

Zakon 3.1.1. (Amdahlov zakon)

Pretpostavimo da vrijedi Brentov pseudoteorem. Neka je $f < 1$ *dio ukupnog vremena* $T(1)$ *koji se troši na posao koji se može paralelizirati, a* $s = 1 - f$ *dio vremena* $T(1)$ *koji se troši na posao koji je sekvencijalan. Tada je*

$$T(p) \geq T(1)\frac{f}{p} + T(1)s \quad .$$

Primijetimo da smo ovdje iskoristili dio iz dokaza Brentovog pseudoteorema koji kaže da sekvencijalno računalo može simulirati paralelno. Nejednakost je posljedica toga što je $T(1)$ *najbolje moguće sekvencijalno vrijeme. Zbog toga je*

$$S(p) = \frac{T(1)}{T(p)} \leq \frac{1}{f/p + s} \leq \frac{1}{s}$$

a ocjena je neovisna o tome koliko procesora p *imamo. Za efikasnost vrijedi*

$$E(p) = \frac{S(p)}{p} \leq \frac{1}{f + sp} \to 0 \quad , \text{ za } p \to \infty \quad .$$

Dakle, ubrzanje je odozgo omeđeno s $1/s$ *i povećanje* p *preko* $f/s = f/(1-f)$ *ne može povećati brzinu (ubrzanje) za faktor veći od* 2.

Pouka Amdahlovog zakona je da algoritam ne smije imati serijsko usko grlo! Na primjer, ako je $s = 1\%$ (samo 1% programa je strogo sekvencijalno), ubrzanje je najviše

$$S(p) \leq 100$$

i ne isplati se koristiti više od $p = 100$ procesora, ako želimo visoku efikasnost.

Dakle, kod projektiranja algoritma treba vrlo pažljivo mjeriti performanse programa, tako da se otkriju eventualna serijska uska grla. Moguće je da se reformulacijom algoritma, dio tog serijskog uskog grla može otkloniti.

Na primjer, nađimo koji dio vremena smije trošiti sekvencijalni dio algoritma (u ovisnosti o p), ako želimo da algoritam ima efikasnost barem 50%:

$$E(p) \geq 0.5 \Longleftrightarrow S(p) \geq \frac{p}{2} \quad ,$$

pa iz Amdahlovog zakona dobivamo

$$\frac{p}{2} \leq S(p) \leq \frac{1}{f/p + s} \quad ,$$

odakle slijedi

$$f + sp = (1 - s) + sp = 1 + s(p - 1) \leq 2 \quad ,$$

odnosno

$$s \leq \frac{1}{p - 1} \quad ,$$

tj. sekvencijalni dio algoritma mora biti proporcionalan najviše inverzu broja procesora. Naravno, sekvencijalni dio algoritma ne možemo mijenjati (ili paralelizirati), pa prethodnu relaciju treba interpretirati kao ocjenu za broj procesora $p \leq 1 + 1/s$, za željenu efikasnost. Drugim riječima, ostale procesore u računalu (ako ih ima), treba dodijeliti (paralelno) nekom drugom poslu!

3.1.3. Cijena

Cijena paralelnog algoritma definira se kao:

$$C(p) := T(p) \cdot p \quad ,$$

tj. podrazumijeva se da svi procesori rade isti broj koraka.

Kažemo da je paralelni algoritam **optimalan** ako je

$$C(p) = \mathcal{O}(\text{donja granica za sekv. broj op., odn. vrijeme}) = \mathcal{O}(T(1)) \quad .$$

Paralelni algoritam sigurno nije optimalan, ako postoji sekvencijalni algoritam čije je vrijeme izvođenja $T(1) < C(p)$.

3.1.4. Donja ograda za paralelno izračunavanje funkcije s n parametara

Pretpostavimo da je dozvoljeno koristiti samo binarne operacije, tj. 2 operanda daju jedan rezultat. Pretpostavimo da te operacije traju podjednako dugo.

Jasno je da u jednom vremenskom koraku možemo izračunati funkciju koja ovisi o samo 2 parametra (1 binarna operacija). U drugom vremenskom koraku, svaki od dva operanda može ovisiti o 2 parametra, odnosno ukupno ulaz ovisi o najviše 4 nezavisna parametra. U svakom slijedećem koraku, najveći broj nezavisnih ulaznih parametara se udvostručuje.

Prema tome, u $k = \lg n$ vremenskih koraka možemo izračunati funkciju koja ovisi o najviše $2^k = n$ nezavisnih parametara. Preciznije rečeno

$$f(x_1, \ldots, x_n) = \Omega(\lg n) \quad ,$$

pri čemu se $\Omega(n)$ (čitati reda najmanje n) definira na slijedeći način:

Definicija 3.1.1.

Neka su $f, g : \mathbb{N} \to \mathbb{R}^+$. Funkcija g je reda najmanje f, u oznaci $\Omega(f(n))$, ako postoje pozitivne konstante n_0 i c takve da vrijedi

$$g(n) \geq c \cdot f(n) \quad , \quad \forall n \geq n_0 \quad .$$

Na primjer, zbrajanje n brojeva je funkcija n nezavisnih argumenata i ne može se obaviti u vremenu kraćem od $\mathcal{O}(\lg n)$. Algoritam je stablast – zbraja u paraleli $n/2$ parova brojeva, pa slijedećih $n/4$, sve dok ne preostane za zbrojiti 2 broja (vidjeti slijedeće poglavlje).

Teorem 3.1.1. (Munro–Paterson (1973.))

Ako se sekvencijalno računanje nekog rezultata (funkcijskih vrijednosti) sastoji od x elementarnih aritmetičkih operacija (maksimalno binarnih)

$$T(1) = x \quad (\textit{ili } \Theta(x))$$

(bitno samo da elementarne operacije podjednako traju), onda je vrijeme $T(p)$ potrebno za paralelno računanje istog rezultata s p procesora

$$T(p) \geq \begin{cases} \lceil \lg x + 1 \rceil & , \textit{ ako } x < 2^{\lceil \lg p \rceil} \\ \dfrac{x + 1 - 2^{\lceil \lg p \rceil}}{p} + \lg p & , \textit{ ako } x \geq 2^{\lceil \lg p \rceil} \end{cases} .$$

■

Primjer 3.1.3.

Pretpostavimo da moramo zbrojiti 8 brojeva. Sekvencijalno zbrajanje traje 7 vremenskih jedinica, dok paralelno zbrajanje traje 3 vremenske jedinice, ako imamo na raspolaganju 4 procesora. Tj. ovdje je $x = 7$, $p = 4$, pa je $x \geq 2^2$, odakle slijedi $T(4) = 3$.

Slijedeći teorem govori o ubrzanju, ako na raspolaganju imamo nedovoljan broj procesora.

Teorem 3.1.2. (Brent (1974.))

Ako je T paralelno vrijeme računanja za postupak od x elementarnih binarnih operacija, uz dovoljno veliki broj procesora (1. slučaj Munro–Patersonovog teorema), onda se isto računanje (postupak) može obaviti za vrijeme $T(p)$ sa samo p procesora, gdje je

$$T(p) = \left\lfloor T + \frac{x - T}{p} \right\rfloor \quad ,$$

(2. slučaj Munro–Patersonovog teorema). ■

Primjer 3.1.4.

Pretpostavimo da moramo zbrojiti 8 brojeva, a na raspolaganju imamo 3 procesora. Prema Brentovom teoremu imamo:

$$T(3) = \left\lfloor 3 + \frac{7 - 3}{3} \right\rfloor = 4 \quad .$$

Sastavite takav algoritam zbrajanja!

Primijetimo da prethodna dva teorema ne uzimaju u obzir komunikaciju među procesorima!

4. Brzi algoritmi na stablima

Mnogi algoritmi numeričke linearne algebre su stablasti. Kao što smo već pokazali, za izvrednjavanje bilo koje netrivijalne funkcije s n parametara, potrebno je najmanje $\lg n$ paralelnih koraka, uz pretpostavku da u svakom paralelnom koraku koristimo samo (unarne ili) binarne operacije.

Cilj poglavlja je pronalaženje najbržih mogućih paralelnih algoritama, bez obzira na potreban broj procesora ili na njihovu numeričku stabilnost. Neki od njih će biti samo akademskog karaktera, a neki će imati i veliko praktično značenje.

4.1. Emitiranje i redukcija na stablima

Na binarnim stablima s p procesora, emitiranje se odvija na prirodan način. Otac pošalje podatke svojim sinovima, oni svojim sinovima, sve dok podaci ne stignu do listova. Vremensko trajanje je dubina korištenog binarnog stabla, odnosno $\lg(p+1) - 1$ koraka.

Redukcija (s asocijativnim operacijama – zbog poretka računanja) je, na neki način, obratna operacija od emitiranja. Svaki sin pošalje podatke svom ocu, pa kao i u slučaju emitiranja, vrijeme trajanja je $\lg(p+1) - 1$ paralelnih koraka.

Redukciju možemo zamišljati i kao stablasto računanje vrijednosti nekog izraza. U listovima se nalaze polazni podaci (atomi izraza), a svi očevi obavljaju neku zadanu binarnu operaciju nad podacima iz svojih sinova i rezultat operacije šalju svom ocu. Krajnji rezultat (vrijednost izraza) je konačni podatak u korijenu stabla (nakon što on obavi svoju operaciju).

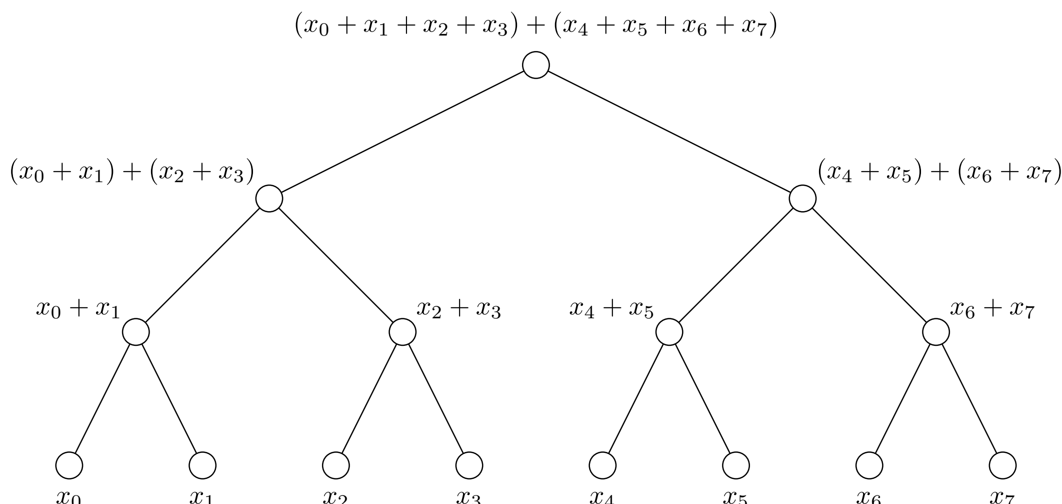
Općenito, ovakav postupak ovisi o tome koje binarne operacije treba obaviti (i kojim redom). Ako su sve te binarne operacije iste i još je ta operacija asocijativna, onda rezultat **ne ovisi** o redoslijedu njihovog izvršavanja. To omogućava jednostavnu paralelizaciju postupka izračunavanja, s vrlo velikom praktičnom primjenom.

Ilustrirajmo takvu redukciju na primjeru zbrajanja $n = 2^k$, $k \in \mathbb{N}$, brojeva (zanemarujući, nakratko, broj procesora p). Zbroj možemo pisati kao

$$S = x_0 + \cdots + x_{n-1} = (x_0 + \cdots + x_{n/2-1}) + (x_{n/2} + \cdots + x_{n-1}) \quad ,$$

što prikazuje operaciju na vrhu (u korijenu) stabla. Članove u zagradama možemo nezavisno, tj. **paralelno** izračunati. Rekurzivnom primjenom ovakvog raščlanjivanja na članove u zagradama (sve do razine osnovnih vrijednosti x_i), dobivamo algoritam paralelnog zbrajanja, poznat pod imenom **paralelni prefiks**.

Za $n = 8$, stablo računanja ima slijedeći oblik:



Ovaj algoritam vrijedi i za bilo koju drugu asocijativnu binarnu operaciju $\oplus$. Naziv "prefiks" dolazi od uobičajenog funkcijskog zapisa te operacije. Rezultat $a \oplus b$ pišemo kao $\oplus(a, b)$, tj. u prefiks notaciji.

Uz malo dodatnog posla, možemo izračunati i sve početne "parcijalne sume" zadanih vrijednosti, tzv. "scan" polaznog niza podataka $x_0, \ldots, x_{n-1}$.

4.2. Zapis algoritama za paralelni prefiks i scan

Uvedimo preciznu definiciju operatora scan.

Definicija 4.2.1.

Neka je zadan niz od n brojeva $x_0, \ldots, x_{n-1}$ i neka je $\oplus$ asocijativna binarna operacija. Rezultat primjene operatora scan(x) *(za zadanu asocijativnu binarnu operaciju $\oplus$) na niz $x = x_0, \ldots, x_{n-1}$ je niz $y = y_0, \ldots, y_{n-1}$, za koji vrijedi*

$$y_i = x_0 \oplus \cdots \oplus x_i \quad , \quad i = 0, \ldots, n-1 \quad .$$

Ovako definirani operator "scan" se katkad naziva i "prescan", jer računa početne parcijalne sume. Analogno se može definirati i operator "postscan" koji računa stražnje parcijalne sume.

Uočimo da komutativnost binarne operacije nije bitna. Za binarnu operaciju $\oplus$ možemo, na primjer, uzeti: zbrajanje, množenje, minimum, maksimum, matrično množenje, itd.

Ponovno, neka je zadan niz brojeva $x_0, \ldots, x_{n-1}$, $n = 2^k$, $k \in \mathbb{N}$. Treba naći $y = \text{scan}(x)$ tog niza, tj. niz $y_0, \ldots, y_{n-1}$,

$$y_i = x_0 \oplus \cdots \oplus x_i = \text{oznaka} = x(0:i) \quad .$$

Za rezultat primjene operacije $\oplus$ na bilo kojem bloku uzastopnih članova niza uvodimo oznaku

$$x_i \oplus \cdots \oplus x_j = \text{oznaka} = x(i:j) \quad , \quad i \leq j \quad .$$

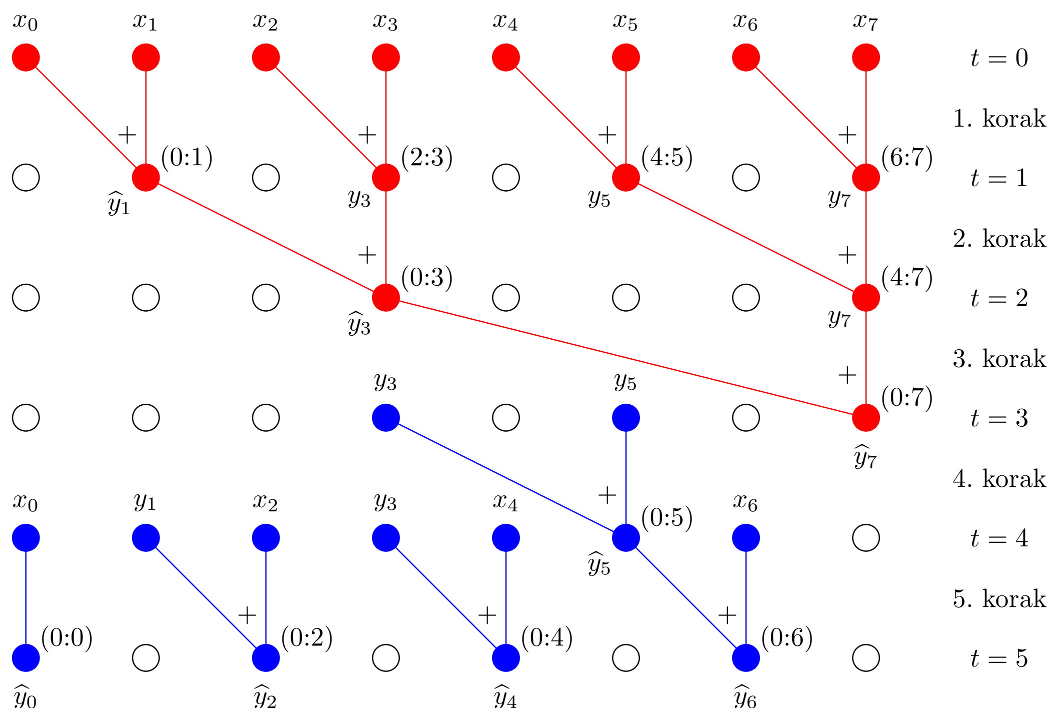
Radi jednostavnosti, za binarnu operaciju $\oplus$ uzmimo obično zbrajanje.

4.2.1. PRAM forma algoritma

Na početku, paralelni algoritam formuliramo sasvim općenito, u tzv. PRAM formi, ignorirajući broj procesora i veze među njima, tj. naznačavamo paralelizam dopušten (ili ograničen) samo međuzavisnošću podataka.

Pretpostavljamo da se zadani niz, na početku, nalazi spremljen u polju x, tako da je $x[i] = x_i$, za $i = 0, \ldots, n-1$, a rezultat treba spremiti u polje y, tako da je $y[i] = y_i$, za $i = 0, \ldots, n-1$.

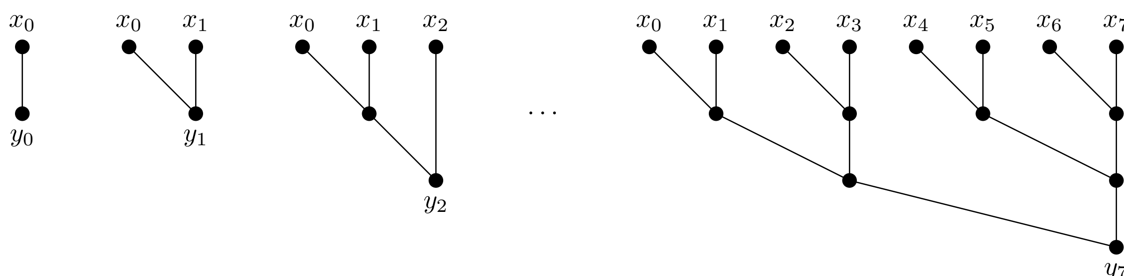
Graf računanja (ili ovisnosti podataka) za $n = 8 = 2^3$ ima oblik:



pri čemu, na prethodnom grafu, vrijednosti y_i su međuvrijednosti, a $\widehat{y}_i$ su konačne vrijednosti elemenata polja y. Uočimo da se u prva tri vremenska koraka obavlja penjanje po stablu (kao u redukciji za nalaženje ukupnog zbroja), a u posljednja dva dolazi do spuštanja po stablu (da nađemo preostale članove niza y).

Ako ovako podijelimo i vremenske korake, proces traje $2k-1$ koraka, a svaki se podatak koristi najviše jednom u danom trenutku (za što je dovoljan EREW model računala). Ako koristimo CREW ili CRCW model, odmah možemo uštediti jedan vremenski korak potreban za zbrajanje $y_3 + y_5$. Tada i cijelu drugu fazu algoritma možemo vremenski pomaknuti za jedno mjesto prema gore pa proces traje $2k - 2$ koraka.

Za početak, dokažimo da ovaj algoritam nije daleko od optimalnog algoritma za računanje niza $y = \text{scan}(x)$. Za dokaz, treba naći minimalni broj vremenskih koraka za (paralelno) računanje cijelog niza y, duljine $n = 2^k$. Tvrdimo da je taj broj jednak k. Promotrimo slijedeće "nezavisne" grafove za računanje $y_0, \ldots, y_{n-1}$.



Ako dozvolimo višestruko paralelno čitanje istih podataka, ove grafove možemo "preklopiti" na mjestu zajedničkih polaznih podataka (što odgovara paralelnom čitanju tih podataka). Sve ostale dijelove tih grafova (rezultate y_i) možemo nezavisno paralelno računati (uz dovoljan broj procesora). Odavde slijedi da je ukupan broj vremenskih koraka jednak maksimalnom broju vremenskih koraka potrebnih za računanje pojedinih članova y_i. Najdulje traje računanje zadnjeg člana y_{n-1} (sume cijelog niza x) i to točno $k = \lg n$ koraka. Međutim, y_{n-1} ovisi o $n = 2^k$ nezavisnih parametara, što znači da trebamo barem k vremenskih koraka za njegovo računanje. Dakle, najbrži algoritam za računanje niza y treba točno k vremenskih koraka.

Zaključujemo da je algoritam na bazi paralelnog prefiksa za nalaženje niza $y = \text{scan}(x)$ najviše za faktor 2 sporiji od najbržeg mogućeg, tj. ima optimalni red veličine.

Algoritam paralelnog prefiksa ima nekoliko prednosti pred opisanim optimalnim algoritmom. Kao prvo, nema višestrukog računanja istih međurezultata. To nije jako bitno, jer paralelizmom želimo smanjiti vremensku, a ne aritmetičku složenost. Nadalje, nema istovremenog višestrukog (paralelnog) čitanja istih podataka, pa možemo koristiti i EREW arhitekturu računala. Na kraju, paralelni prefiks se jednostavno i prirodno realizira na računalima s razdijeljenom memorijom uz standardne mreže povezivanja (očito je binarno stablo prirodna mreža).

Algoritam 4.2.1. (PRAM – algoritam)

```
{Penjanje}

m := n div 2;   {broj paralelnih operacija u pojedinom koraku}
step := 1;   {razmak indeksa za zbrajanje}
for i := 1 to m parallel do   {1. korak penjanja}
  y[2 * i − 1] := x[2 * i − 2] + x[2 * i − 1];   {= x(2i − 2 : 2i − 1)}
for j := 2 to k do   {j-ti korak penjanja}
    {ekvivalentno je while m > 1 do ili while step < n div 2 do}
  begin
  m := m div 2;   {m = n div 2^j = 2^(k−j)}
  step := 2 * step;   {step = 2^(j−1)}
  for i := 1 to m parallel do
    y[2 * i * step − 1] := y[(2 * i − 1) * step − 1] + y[2 * i * step − 1];
      {= x((2i − 2) · step : (2i) · step − 1)}
  end; {for j}
  {Ako je n > 1 (tj. k > 0), onda ovdje vrijedi:
    m = 1, step = n div 2 = 2^(k−1) i
    y[n − 1] sadrži čitavu sumu x(0 : n − 1).
   U protivnom, za n = 1 vrijedi m = 0.}

{Spuštanje}

for j := 1 to k − 2 do   {j-ti korak spuštanja - koristi samo y}
  begin
  m := 2 * m;   {m = 2^j}
  step := step div 2;   {step = n div (2m) = 2^(k−(j+1)) = 2^(k−j−1)}
  for i := 1 to m − 1 parallel do   {za 1 manje nego prije}
    y[(2 * i + 1) * step − 1] := y[(2 * i * step − 1] + y[(2 * i + 1) * step − 1];
      {= x(0 : (2i + 1) · step − 1)}
  end;   {for j}
m := 2 * m;   {m = 2^(k−1) = n div 2}
{ovdje ne treba: step := step div 2; jer izlazi točno step = 1}
parallel begin
  y[0] := x[0];
  for i := 1 to m − 1 parallel do
    y[2 * i] := y[2 * i − 1] + x[2 * i];   {= x(0 : 2i)}
parallel end;
```

Ako je $n = 1$, onda se petlja po j ne izvršava, a u zadnjem bloku je $m = 0$, pa se ne izvršava niti petlja po i. Tj. za $n = 1$, samo kopiramo $x[0]$ u $y[0]$.

Za $n > 1$, algoritam u j-tom koraku penjanja (za $j = 1, \ldots, k$) računa

$$y[i2^j - 1] := x((i-1)2^j : i2^j - 1) \quad , \quad i = 1, \ldots, 2^{k-j} \quad ,$$

a u j-tom koraku spuštanja (za $j = 1, \ldots, k-1$) računa

$$y[i2^{k-j} + 2^{k-j-1} - 1] := x(0 : i2^{k-j} + 2^{k-j-1} - 1) \quad , \quad i = 1, \ldots, 2^j - 1 \quad .$$

Analizirajte koje od ovih vrijednosti su i konačne za pripadne elemente u polju y, u fazi penjanja odn. spuštanja. Nakon toga, indukcijom se lagano dokazuje korektnost algoritma.

Ovako zapisan algoritam strogo odgovara ranijoj slici. Ako imamo dovoljan broj procesora na raspolaganju, ovaj algoritam traje najviše $2k - 1$ vremenskih koraka, a može i manje (ovisno o broju procesora i podjeli posla). To vrijedi za PRAM arhitekturu ili za odgovarajući pristup globalnoj memoriji, odnosno odgovarajuće veze među procesorima.

Zapis algoritma ne ovisi o broju procesora i nigdje se ne vidi koji procesor izvodi koju operaciju. Kod analize složenosti, broj procesora ulazi u igru, a jednako tako i detalji arhitekture.

Za praktičnu primjenu, ovaj je zapis preapstraktan. On je na nivou općeg ili potencijalnog paralelizma, tj. grafa ovisnosti podataka. Uočimo da se u prethodnom algoritmu i petlja po j može paralelizirati, uz dovoljan broj procesora.

Napomena 4.2.1.

Ako pretpostavimo da na početku algoritma vrijedi

$$y[i] = x[i] \quad , \quad i = 0, \ldots, n-1 \quad ,$$

onda se zapis algoritma bitno skraćuje, jer ne treba posebno pisati 1. korak penjanja i zadnji korak spuštanja.

Tu pretpostavku nije teško osigurati jednom vektorskom operacijom $y := x$ s, na primjer, n procesora, pri čemu svaki obavlja po jedno skalarno kopiranje vrijednosti. Vremensko trajanje raste za jednu vremensku jedinicu (uz n procesora).

Ovakav zapis algoritma koristi poneke lokacije $y[i]$ za spremanje međurezultata i to više puta, za razne podatke. Na primjer, $y[7]$ sadrži redom

$$x(6:7) \quad , \quad x(4:7) \quad , \quad x(0:7) = \text{konačni } y[7] \quad ,$$

U drugoj fazi, tj. kod spuštanja, sve izračunate vrijednosti su konačne, samo se koriste međurezultati u istim lokacijama (na primjer, $y[5]$).

Zadatak 4.2.1.

Napišite točno broj međurezultata y_i koji nisu i konačne vrijednosti tih lokacija. Napišite PRAM *algoritam koji koristi polje y samo za spremanje konačnih rezultata, a za međurezultate koristi pomoćno polje z i to svaku lokaciju u polju z koristi*

točno jednom za spremanje (pisanje). Dozvoljeno je više puta čitati isti podatak. Analizirajte koji podaci se više puta čitaju i u kojem koraku.

Napravite to isto, ali uz dodatno ograničenje da se svaki rezultat smije točno jednom čitati (za višestruko čitanje, svaki puta se mora iznova spremiti podatak).

Koliko se lokacija koristi u polju z u svakom od ova dva slučaja?

Zadatak 4.2.2. ("Pametni" – kratki zapis PRAM algoritma)

Pretpostavimo da polje x sadrži niz x spremljen tako da na početku vrijedi:

$$x[n+i] = x_i \quad , \quad i = 0, \ldots, n-1 \quad , \quad n = 2^k \quad ,$$

tj. $x_0, \ldots, x_{n-1}$ su na mjestima $x[n], \ldots, x[2n-1]$. Promotrimo slijedeći algoritam:

```
for j := k − 1 downto 0 do
  for i := 2^j to 2^(j+1) − 1 parallel do
    x[i] := x[2 ∗ i] + x[2 ∗ i + 1];
y[1] := x[1];
for j := 1 to k do
  for i := 2^j to 2^(j+1) − 1 parallel do
    if odd(i) then
      y[i] := y[(i − 1) div 2]
    else
      y[i] := y[i div 2] − x[i + 1];
```

(a) *Dokažite da na kraju algoritma vrijedi: $y[n+i] = \sum_{j=0}^{i} x_j = x[n] + \cdots + x[n+i]$, za $i = 0, \ldots, n-1$.*

(b) *Što je dobro, a što loše u ovom algoritmu?*

(c) *Da li se ovaj algoritam može generalizirati na bilo koju asocijativnu binarnu operaciju $\oplus$? Ako da, kako?*

(d) *Da li je ovaj algoritam "blizu" odgovora na prošli zadatak i što bi trebalo popraviti?*

4.2.2. Algoritam za arhitekturu sa zajedničkom memorijom

Pretpostavimo da imamo na raspolaganju paralelno računalo sa

(a) zajedničkom memorijom (shared memory);

(b) $p \geq \lceil n/2 \rceil$ procesora, tj. $p \geq 2^{k-1}$.

Zbog uvjeta (b), imamo dovoljno procesora da sve prirodno paralelne korake (operacije) zaista izvedemo paralelno (na primjer, 1. korak zbrajanja).

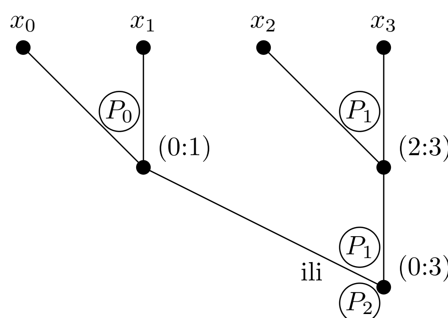
Zajednička memorija znači da su sve lokacije – adrese $x[i] = x_i$, $y[i] = y_i$, za $i = 0, \ldots, n-1$, globalne za cijelo računalo, tj. za sve procesore.

Ako dodatno pogledamo graf (stablo) procesa računanja, vidimo da gotovo nema konflikata u pristupu podacima, u smislu da više procesora istovremeno čita i/ili piše u istu lokaciju.

Jedini potencijalni konflikt je u završnom koraku penjanja i prvom koraku spuštanja – oba koraka koriste $y[n/2-1] = x(0 : n/2-1)$. Zbog toga smo ta dva koraka pisali vremenski odvojeno, što pokriva EREW (ekskluzivno čitanje je bitno) model. Ako je dozvoljeno istovremeno čitanje (CR), ova dva koraka mogu se spojiti u isti vremenski trenutak.

Veze među procesorima nisu bitne, jer koristimo zajedničku memoriju i nema konflikata čitanja i pisanja.

Sve ovisi o tome kako su procesori vezani na memoriju. Ako su procesori vezani kao stablo, onda se samo finalni rezultati pišu u zajedničku memoriju.



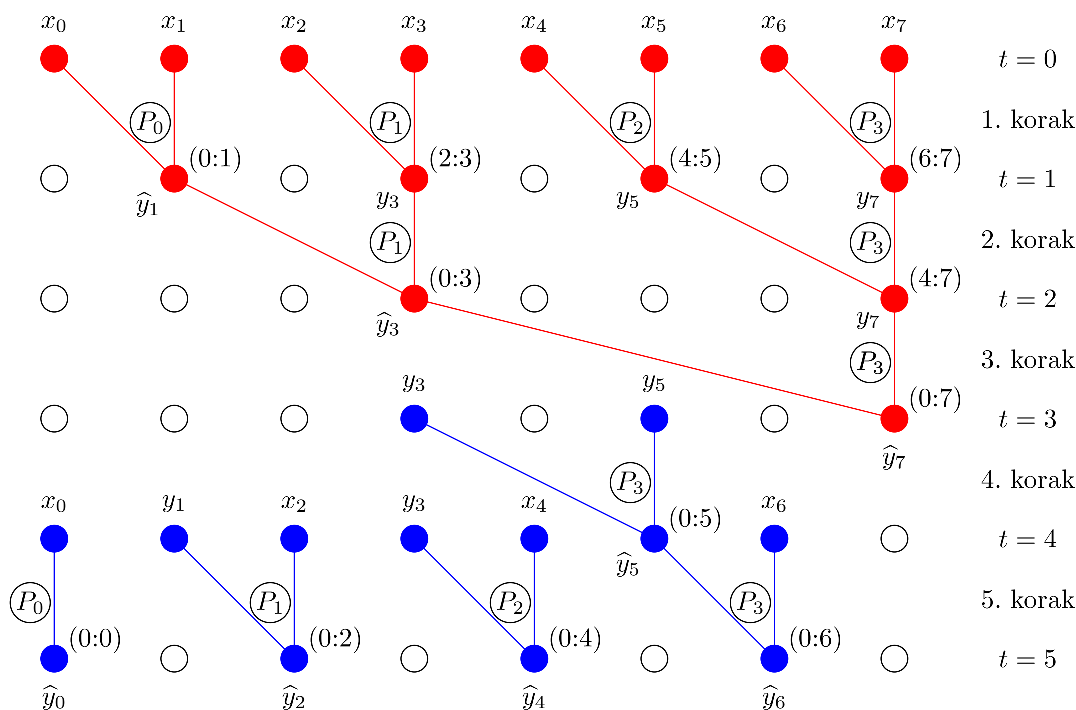
Postoje još dvije mogućnosti za ubrzanje.

- Međurezultat $(2 : 3)$ ne treba pisati kao $y[3]$, već se može lokalno čuvati u registru procesora P_1.
- Posljednju operaciju može obaviti procesor P_2, pa mu P_0 i P_1 šalju međurezultate (to traži barem $n-1$ procesora i vodi na algoritam za razdijeljenu memoriju).

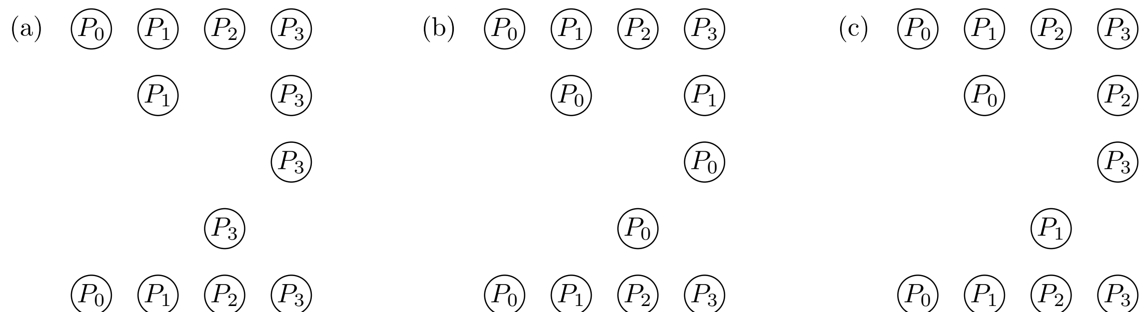
Slijedeći algoritam o tome ne vodi računa. Te detalje pokazujemo kasnije, u algoritmu s lokalnom ili razdijeljenom memorijom.

Algoritam za procesore sa zajedničkom memorijom koristi točno $p = n/2$ procesora, koji su indeksirani brojevima od 0 do $n/2 - 1 = 2^{k-1} - 1$.

Slijedeća slika prikazuje koji procesor obavlja koju operaciju s prethodnog grafa. Stablo procesa računanja za $n = 8 = 2^3$ s $n/2 = 4$ procesora P_0, P_1, P_2 i P_3:



Možemo izabrati i drugačije pridruživanje operacija procesorima (u svim koracima osim prvog i zadnjeg). Na slijedećoj slici prikazana su tri takva rasporeda. Raspored (a) odgovara prethodnoj slici, raspored (b) je simetričan rasporedu (a), dok je kod rasporeda (c) poštovano balansirano opterećenje procesora, tj. u unutarnjim koracima svaki procesor koristimo točno jednom.



Svi ovi rasporedi imaju fiksiran prvi i zadnji red (indeksi procesora rastu s indeksima podataka). Očito je da možemo izabrati i bilo koju drugu polaznu i završnu permutaciju indeksa procesora (ali to nema smisla za pregledni zapis algoritma).

Dogovor za zapis raspodjele operacija po procesorima

Pretpostavimo da svaki procesor ima registar–konstantu, zovimo ju MYPROC, koja sadrži njegov indeks. Za indeksiranje procesora u programu koristimo rezervirani globalni identifikator (konstantu) MYPROC, koja, zapravo, adresira odgovarajući registar procesora.

Ako neka naredba u programu sadrži parametar (identifikator) MYPROC, pretpostavljamo da se ta naredba izvodi paralelno — istovremeno na svim procesorima, s tim da svaki procesor koristi svoju vrijednost za MYPROC.

U paralelnim petljama, radi preglednosti programa, obično koristimo naredbu oblika

$$\textbf{if } \text{MYPROC} = i \textbf{ then } \langle \textit{naredba}_i \rangle \,;$$

sa značenjem da procesor P_i (čiji registar MYPROC sadrži vrijednost i) izvršava zadanu naredbu $naredba_i$. U ovom zapisu, i je indeks u okolnoj paralelnoj petlji, a $naredba_i$ je neka naredba koja, obično, ovisi o i (tj. sadrži identifikator i).

Potpuno isto značenje možemo postići i tzv. skraćenim zapisom. Tada ne pišemo okolnu petlju (s indeksom i) i prethodnu if–naredbu, već samo:

$$\textit{naredba}_{\text{MYPROC}} \,;$$

tj. u tijelu naredbe pišemo MYPROC, umjesto i. Uz dogovor o paralelnom izvršavanju ove naredbe po svim procesorima, efekt je isti, jer svaki procesor koristi svoju vrijednost za MYPROC.

Za analizu složenosti, pogodniji je dulji zapis, jer se paralelne petlje eksplicitno pišu, pa se lako nalazi potreban broj procesora (iz granica petlje). Naredbe unutar takvih petlji pišemo u terminima MYPROC, a ne indeksa petlje (tj. kao u skraćenom zapisu).

Ako je memorija zajednička, ovaj dogovor je dovoljan za zapis algoritama (jer svi procesori operiraju na zajedničkoj memoriji, tj. sve adrese su globalne).

Kod razdijeljene memorije, kad procesor može izravno adresirati samo svoje lokalne podatke, parametar MYPROC koristimo i za lokalno adresiranje. Tada, za komunikaciju među procesorima, koristimo posebne naredbe **send** i **receive** (pošalji i primi), oblika

$$\textbf{send}\,(\textit{podatak}, i) \quad , \quad \textbf{receive}\,(\textit{podatak}, i) \quad ,$$

sa značenjem "pošalji *podatak* procesoru P_i", odnosno "primi *podatak* od procesora P_i".

Algoritam za operaciju scan na računalu sa zajedničkom memorijom izlazi direktno iz PRAM algoritma.

Algoritam 4.2.2. (Algoritam na zajedničkoj memoriji)

{``Penjanje'' u stablu procesa računanja}

$m := n$ **div** 2; {broj paralelnih operacija u pojedinom koraku}
step $:= 1$; {razmak indeksa za zbrajanje}
for $i := 1$ **to** m **parallel do** {1. korak penjanja}
  if MYPROC $= i - 1$ **then**
    $y[2 * \mathrm{MYPROC} + 1] := x[2 * \mathrm{MYPROC}] + x[2 * \mathrm{MYPROC} + 1]$;
      $\{= x(2i - 2 : 2i - 1), \quad y[2i - 1] := x[2i - 2] + x[2 * i - 1]\}$
for $j := 2$ **to** k **do** {j-ti korak penjanja}
  begin
  $m := m$ **div** 2; $\{m = n \textbf{ div } 2^j = 2^{k-j}\}$
  step $:= 2 *$ *step*; $\{step = 2^{j-1}\}$
  for $i := 1$ **to** m **parallel do**
    if MYPROC $= i * step - 1$ **then**
      $y[2 * \mathrm{MYPROC} + 1] := y[2 * \mathrm{MYPROC} + 1 - step] + y[2 * \mathrm{MYPROC} + 1]$;
        $\{= x((2i - 2) \cdot step : (2i) \cdot step - 1)\}$
  end; {**for** j}
  {Ako je $n > 1$ (tj. $k > 0$), onda ovdje vrijedi:
    $m = 1$, $step = n \textbf{ div } 2 = 2^{k-1}$ i $y[n-1] = x(0 : n-1)$.
   U protivnom, za $n = 1$ vrijedi $m = 0$.}

{``Spuštanje'' u stablu procesa računanja}

for $j := 1$ **to** $k - 2$ **do** {j-ti korak spuštanja - koristi samo y}
  begin
  $m := 2 * m$; $\{m = 2^j\}$
  step $:=$ *step* **div** 2; $\{step = n \textbf{ div } (2m) = 2^{k-(j+1)} = 2^{k-j-1}\}$
  for $i := 1$ **to** $m - 1$ **parallel do** {za 1 manje, MYPROC za jedan više}
    if MYPROC $= (i + 1) * step - 1$ **then**
      $y[2 * \mathrm{MYPROC} + 1 - step] := y[2 * \mathrm{MYPROC} + 1 - 2 * step] +$
                    $y[2 * \mathrm{MYPROC} + 1 - step]$;
        $\{= x(0 : (2i + 1) \cdot step - 1)\}$
  end; {**for** j}
$m := 2 * m$; $\{m = 2^{k-1} = n \textbf{ div } 2\}$
{ovdje ne treba: *step* $:=$ *step* **div** 2; jer izlazi točno $step = 1$}
if MYPROC $= 0$ **then**
  $y[0] := x[0]$
else
  for $i := 1$ **to** $m - 1$ **parallel do**
    if MYPROC $= i$ **then**
      $y[2 * \mathrm{MYPROC}] := y[2 * \mathrm{MYPROC} - 1] + x[2 * \mathrm{MYPROC}]$; $\{= x(0 : 2i)\}$

4.2.3. Algoritam za arhitekturu s razdijeljenom memorijom

Na kraju, pretpostavimo da imamo paralelno računalo s razdijeljenom memorijom.

Razumno je pretpostaviti da je lokalna memorija svakog procesora relativno mala, u svakom slučaju, mnogo manja od n.

U protivnom, možemo pretpostaviti da svaki procesor ima svoju kopiju niza, pa smo u sličnoj situaciji kao da je memorija zajednička, osim što svaki procesor koristi svoje lokalne podatke.

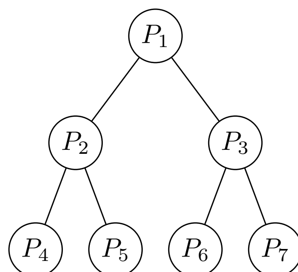
Granični slučaj, kad razumno veliki komad niza stane u lokalnu memoriju, na primjer $n/8$ ili $n/16$ (općenito $n/2^\ell$), nije posebno zanimljiv za praksu, jer n može varirati. Algoritam je tada kombinacija algoritama za zajedničku i razdijeljenu memoriju i treba sve što se može raditi lokalno, zaista i raditi lokalno.

Dakle, kao zaključak, pretpostavimo da u lokalnu memoriju stane malo podataka. Druga bitna stvar za algoritam je povezanost procesora, jer moraju komunicirati. Zbog toga, algoritam ne možemo razumno sastaviti bez dogovora o mreži povezivanja.

Prirodna arhitektura za paralelni prefiks je stablo procesora i to binarno stablo, jer koristimo binarne operacije.

Za prvi korak penjanja trebamo $n/2$ procesora – listova tog stabla, da obave $n/2$ paralelnih zbrajanja u istom vremenskom trenutku. Ako binarno stablo ima $2^{k-1} = n/2$ listova, onda čitavo stablo ima $2^k - 1 = n - 1$ procesora.

Procesore numeriramo redom brojevima od 1 (korijen stabla), a zatim nivo po nivo stabla, slijeva udesno rastućim indeksima. Ova numeracija je obratna od one za zajedničku memoriju, ali olakšava zapis algoritma. Osim toga, sad imamo $n - 1$ procesora, a a ne samo $n/2$.



Procesori u najdonjem redu (listovi) imaju indekse $P_{n/2}, \ldots, P_{n-1}$. Također, ako P_i nije korijen, onda je otac procesora P_i procesor $P_{i \,\mathbf{div}\, 2}$. Ako P_i nije list, njegov lijevi sin je P_{2i}, a desni sin P_{2i+1}.

Algoritam 4.2.3. (Algoritam na razdijeljenoj memoriji)

U zapisu algoritma pretpostavljamo da je $n \geq 4$ ili $k \geq 2$, bez dodatne provjere, tako da izbjegnemo ispitivanje n ili k prije pojedinih faza algoritma.

```
{Penjanje - 1. korak}
m := n div 2;
for i := 1 to m parallel do
  if MYPROC = i + m − 1 then
    begin
    L[MYPROC] := global x[2 ∗ MYPROC − n];    {x[2i − 2]}
    R[MYPROC] := global x[2 ∗ MYPROC + 1 − n];    {x[2i − 1]}
    S[MYPROC] := L[MYPROC] + R[MYPROC];
    send (S[MYPROC], MYPROC div 2);
      {ovdje se koristi k ≥ 2 - ima bar 2 nivoa}
    end;
for j := 2 to k − 1 do    {Penjanje - j-ti korak}
  begin
  m := m div 2;    {m = n div 2^j = 2^(k−j)}
  for i := 1 to m parallel do
    if MYPROC = i + m − 1 then
      begin
      receive (L[MYPROC], 2 ∗ MYPROC);
      receive (R[MYPROC], 2 ∗ MYPROC + 1);
      S[MYPROC] := L[MYPROC] + R[MYPROC];
      send (S[MYPROC], MYPROC div 2);
      end;
  end; {for j}
{Penjanje - k-ti korak, početak spuštanja}
m := m div 2;    {= 1}
if MYPROC = 1 then
  begin
  receive (L[MYPROC], 2 ∗ MYPROC);
  receive (R[MYPROC], 2 ∗ MYPROC + 1);    {Može se izbaciti!}
  S[MYPROC] := L[MYPROC] + R[MYPROC];    {Može se izbaciti!}
  send (L[MYPROC], 2 ∗ MYPROC + 1);
    {suma lijevog podstabla prenosi se u desno podstablo}
  end;
for j := 1 to k − 2 do    {Spuštanje - j-ti korak}
  begin
  m := 2 ∗ m;    {m = 2^j}
  if MYPROC = m then    {nema nule kao neutrala!!}
    send (L[MYPROC], 2 ∗ MYPROC + 1);
```

```
  else
    for i := 2 to m parallel do
      if MYPROC = i + m − 1 then
        begin
        receive (T[MYPROC], MYPROC div 2);
        send (T[MYPROC], 2 ∗ MYPROC);
        S[MYPROC] := T[MYPROC] + L[MYPROC];
        send (S[MYPROC], 2 ∗ MYPROC + 1);
        end;
  end;   {for j}

{Spuštanje - (k − 1)-i korak}

m := 2 ∗ m;   {m = 2^(k−1) = n div 2}
if MYPROC = m then
  begin
  global y[2 ∗ MYPROC − n] := L[MYPROC];
  global y[2 ∗ MYPROC + 1 − n] := S[MYPROC];
  end
else
  for i := 2 to m parallel do
    if MYPROC = i + m − 1 then
      begin
      receive (T[MYPROC], MYPROC div 2);
      global y[2 ∗ MYPROC − n] := T[MYPROC] + L[MYPROC];
      global y[2 ∗ MYPROC + 1 − n] := T[MYPROC] + S[MYPROC];
      end;
```

Na početku, procesori listovi (u stablu procesora) sadrže po dva odgovarajuća susjedna člana polaznog niza x. Ta dva člana možemo interpretirati kao njihovu djecu (u smislu podataka). U fazi penjanja, procesori računaju sumu djece (svojih podstabala) i šalju ju roditelju. Tako svaki procesor ima zbroj svih podataka **ispod** sebe. Na kraju penjanja, u korijenu dobivamo cijelu sumu (ako nam ona posebno treba).

Da bismo dobili scan polaznog niza, procesorima (koji nisu najljeviji u svom nivou) fali još suma svih podataka s indeksima manjim od njegove djece (tj. **lijevo** od njegove djece). U fazi spuštanja, procesor od roditelja dobiva taj podatak. Svom lijevom djetetu ga proslijeđuje, a za desno dijete mora dodati još sumu svog lijevog podstabla (zapamćenu iz faze penjanja). Na kraju, procesori listovi, kad dobiju podatak od roditelja, tom podatku dodaju svoje članove polaznog niza. Zbog toga, ti procesori, osim svog lijevog člana, pamte i sumu oba svoja člana niza.

Za realizaciju ovog postupka, svaki procesor ima 4 lokalne varijable (memorijske lokacije) L, R, S i T, za primanje, slanje i pamćenje podataka. Indeks varijable

označava kojem procesoru ona pripada. Točna značenja tih varijabli su:

- $L[i]$, $R[i]$ sadrže zbroj svih članova niza x u lijevom, odnosno desnom podstablu procesora P_i, respektivno. Ako je P_i list, te varijable sadrže njegove članove niza.
- $S[i]$ je njihov zbroj, tj. zbroj svih elemenata ispod (u podstablima) procesora P_i.
- $T[i]$ sadrži sumu sve ljevije djece (u fazi spuštanja).

U procesorima koji nisu listovi, samo L pamti podatke između faze penjanja i spuštanja, a sve ostale varijable služe za komuniciranje. U listovima, L i R sadrže članove niza, a S njihov zbroj. Osim L, još i S treba zapamtiti do faze spuštanja (zadnjeg koraka algoritma).

U pojedinoj fazi koristimo samo 3 varijable, pa jednu možemo eliminirati. Na primjer, R se koristi samo u penjanju, a T u spuštanju, pa ih smijemo poistovjetiti.

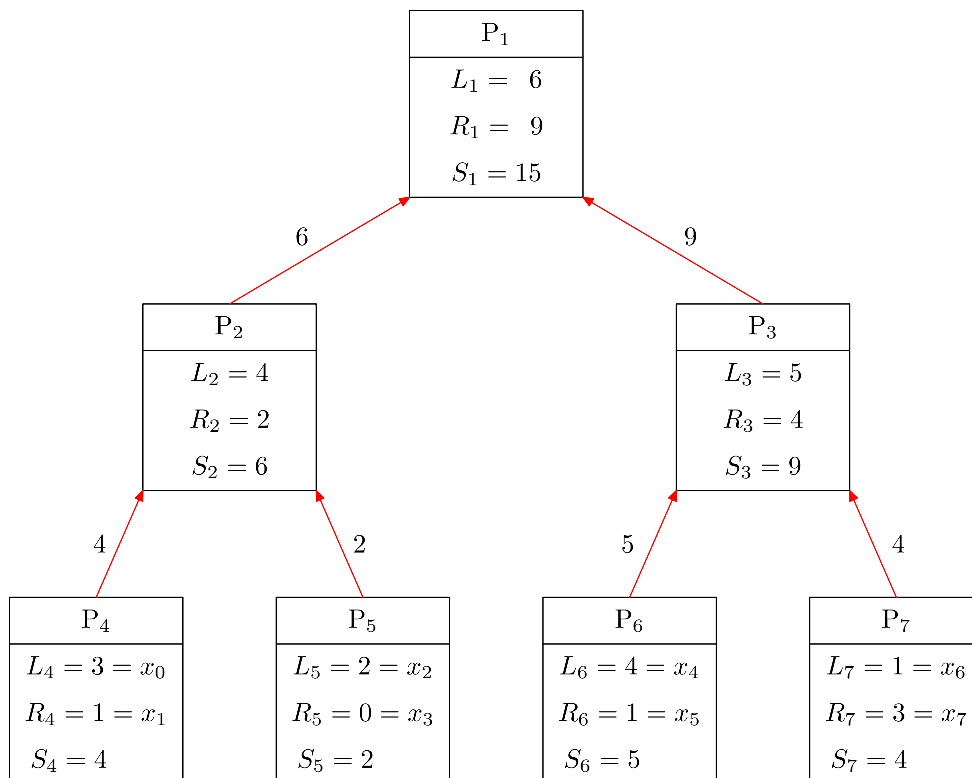
Na kraju, uočimo da korijen P_1 bespotrebno prima R i računa S, jer se S ne koristi poslije. U nekim praktičnim primjenama, korisno je odmah naći i sumu S cijelog niza x, s kojom P_1 može dalje operirati (ili ju poslati nekom drugom). Ako nam suma cijelog niza ne treba prije kraja algoritma, pripadne dvije naredbe u korijenu možemo izbaciti.

Ilustrirajmo rad algoritma za $n = 8$ i polazni niz podataka

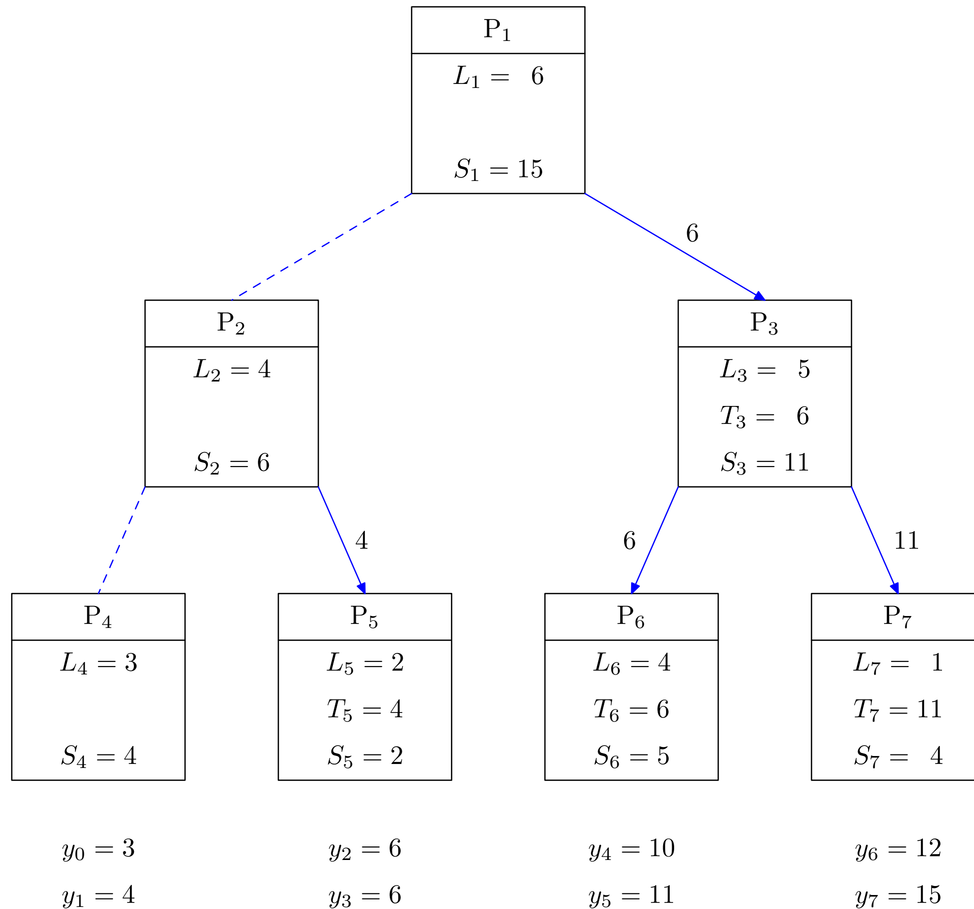
$$x = (3, 1, 2, 0, 4, 1, 1, 3) \quad .$$

Slijedeće dvije slike pokazuju stanje lokalnih varijabli i komunikaciju između procesora u pojedinim fazama algoritma.

Faza penjanja (prema korijenu stabla procesora) ima oblik



Faza spuštanja (prema listovima stabla procesora) ima oblik



Konačni rezultat je niz

$$y = \text{scan}(x) = (3, 4, 6, 6, 10, 11, 12, 15) \quad .$$

Napomenimo još neke činjenice, važne za razumijevanje paralelnog algoritma na razdijeljenoj memoriji.

Napomena 4.2.2.

Lokalno indeksiranje može se i ispuštati, ako vrijedi dogovor da su svi neindeksirani objekti lokalni. Operacije s globalnom memorijom mogu se odavde izbaciti i pretpostaviti da L i R u listovima već sadrže odgovarajuće vrijednosti iz polja x. Pri tome treba pretpostaviti da se punjenje tih vrijednosti (lokalnih objekata) iz polja x odvija paralelno u jednom koraku. U protivnom, može se promijeniti vremenska složenost cijelog algoritma. Na kraju, sprema se (globalno) polje y, također, u jednom vremenskom koraku.

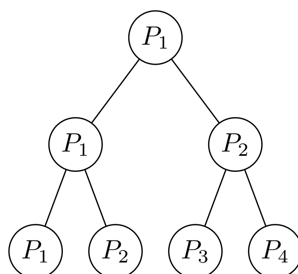
Primijetimo još da proces računanja ne mora vremenski odgovarati petlji po j. Operacije u lijevom podstablu mogu se odvijati i brže (ranije).

Zadatak 4.2.3.

Napišite pripadni algoritam u kojem petlje po j odražavaju optimalno vremensko odvijanje procesa.

Zadatak 4.2.4.

Prethodni algoritam može se realizirati i sa samo $n/2$ procesora. Tada više ne radimo s binarnim stablom. Kako treba postaviti procesore iznad listova, uz pretpostavku veza sličnih onima na stablu (2 ulaza i 1 izlaz)? Da li je u redu ovakav raspored:



Obratite pažnju na potrebnu lokalnu memoriju, tj. analizirajte što sve mora pamtiti pojedini procesor (posebno prvi).

Zadatak 4.2.5.

Modificirajte algoritam za razdijeljenu memoriju, ako procesore numeriramo kao u algoritmu za zajedničku memoriju, tj. da "listovi" imaju najmanje indekse.

4.3. Povećanje efikasnosti

Ako imamo $p \geq n/2$ procesora, onda paralelni prefiks niza duljine n traje $\mathcal{O}(\log n)$ koraka. Međutim, znamo da efikasnost teži u 0, za $n \to \infty$, tj. u prethodnom algoritmu, u kasnijim koracima penjanja i početnim koracima spuštanja, većina procesora ne radi ništa. Točnije:

1. korak penjanja – radi $n/2$ procesora;
2. korak penjanja – radi $n/4$ procesora;

$\vdots$

k. korak penjanja – radi $n/2^k$ procesora.

Pretpostavimo da na raspolaganju imamo $p < n/2$ procesora. Možemo postupiti na dva načina. Prvo, možemo koristiti Brentov teorem, tj. možemo naći potrebno vrijeme za izvođenje algoritma na nedovoljnom broju procesora. Drugi pristup je modifikacija algoritma.

Podijelimo niz x duljine n na p grupa: $p-1$ grupa ima $\lceil n/p \rceil$ članova, a posljednja ima preostatak, tj. $n-(p-1)\cdot\lceil n/p \rceil \le \lceil n/p \rceil$ članova. Svakoj grupi dodijelimo njezin procesor. Taj procesor obrađuje svoju grupu sekvencijalno, pa je za obradu potrebno vrijeme linearno ovisno o duljini grupe $\mathcal{O}(n/p)$. Grupe obrađujemo paralelno, pa nakon $\mathcal{O}(n/p)$ vremena imamo blok–prefiks veličine p. Taj blok–problem (blok–scan) možemo riješiti u $\mathcal{O}(\log p)$ vremena na računalu s p procesora. Na kraju, za svaku grupu treba napraviti pravi scan, što se radi sekvencijalno, za svaku grupu na njenom procesoru (za grupe, naravno, paralelno). Ovaj korak svodi se na dodavanje sume prethodnih grupa lokalnom scan-u grupe, što je linearno u n/p. Zbrajanjem potrebnih vremena za svaki korak, dobivamo ukupno vrijeme za ovaj algoritam, koje je jednako $\mathcal{O}(n/p+\log p)$. Ako uzmamo $p = n/\log n \le n/2$, za veće n, izlazi da je ukupno vrijeme $\mathcal{O}(\log n)$, tj. optimalno, ali sada uz optimalno iskorištavanje procesora (efikasnost više ne teži u 0, za velike n).

4.4. Množenje matrica reda n u vremenu $\mathcal{O}(\log n)$

Neka su A i B dvije matrice reda n. Pretpostavimo da imamo na raspolaganju $p = n^3$ procesora. Tada produkt matrica $C = A\cdot B$ možemo izračunati u vremenu $\mathcal{O}(\log n)$.

Uočimo da za množenje dvije matrice reda n treba izračunati točno n^3 produkata oblika $c_{ijk} = a_{ik}\cdot b_{kj}$, za što je potreban samo jedan vremenski korak. Nakon toga, svaki element c_{ij} matrice C

$$c_{ij} = \sum_{k=1}^{n} c_{ijk} \quad , \quad 1 \le i,j \le n \quad ,$$

dobivamo zbrajanjem n brojeva. Za zbrajanje n^2 skupina od n brojeva potrebno je $n^3/2$ procesora, a potrebno vrijeme je (paralelni prefiks) $\mathcal{O}(\log n)$.

Dakle, ukupno vrijeme potrebno za izvršavanje ovog paralelnog algoritma je, također, $\mathcal{O}(\log n)$.

Grubo skicirajmo algoritam:

for $i := 1$ **to** n **parallel do**
  for $j := 1$ **to** n **parallel do**
    for $k := 1$ **to** n **parallel do**
      $c[i,j,k] := a[i,k] * b[k,j];$

{`Paralelni prefiks`}

for $i := 1$ **to** n **parallel do**
  for $j := 1$ **to** n **parallel do**
    $c[i,j] := \sum_{k=1}^{n} c[i,j,k];$

Uočimo još da skalarni produkt dva vektora duljine n možemo izračunati u vremenu $\mathcal{O}(\log n)$. Naime, svaki element c_{ij} je, zapravo, jedan skalarni produkt vektora (i–tog retka matrice A i j–tog stupca matrice B).

4.5. Invertiranje trokutastih matrica reda n u vremenu $\mathcal{O}(\log^2 n)$

Za konstrukciju ovog algoritma potrebna nam je jedna lema.

Lema 4.5.1.

Neka je T regularna donjetrokutasta matrica reda n. Tada za proizvoljnu particiju matrice T oblika

$$T = \begin{bmatrix} A & 0 \\ C & B \end{bmatrix}$$

pri čemu su A i B kvadratne matrice redova m i $n - m$, vrijedi:

$$T^{-1} = \begin{bmatrix} A^{-1} & 0 \\ -B^{-1}CA^{-1} & B^{-1} \end{bmatrix} \quad .$$

Dokaz:

Uočimo da je trokutasta matrica regularna, ako su joj svi dijagonalni elementi različiti od nule. Dakle, ako je T regularna, regularne su i donjetrokutaste matrice A i B, pa ih smijemo invertirati.

Ostaje još dokazati da je $T \cdot T^{-1} = T^{-1} \cdot T = I_n$. Dokaz provodimo množenjem matrica. Kod množenja blok matrica treba samo paziti da su odgovarajući blokovi kompatibilni, tj. da su njihove dimenzije takve da ih (kao matrice) smijemo pomnožiti. Vrijedi:

$$T \cdot T^{-1} = \begin{bmatrix} A & 0 \\ C & B \end{bmatrix} \cdot \begin{bmatrix} A^{-1} & 0 \\ -B^{-1}CA^{-1} & B^{-1} \end{bmatrix} = \begin{bmatrix} I_m & 0 \\ CA^{-1} - CBB^{-1}A^{-1} & I_{n-m} \end{bmatrix} = I_n \quad .$$

Jednakost $T^{-1} \cdot T = I_n$ dokazuje se na isti način. (Ili još jednostavnije, zbog kvadratičnosti matrice T, ako je $T \cdot T^{-1} = I_n$, onda je T^{-1} i inverz od T.) ■

Algoritam koji koristi prethodnu lemu je tipa "podijeli pa vladaj" (divide–and–conquer). Radi jednostavnosti, pretpostavimo da matrica T ima red $n = 2^k$. Izaberimo matrice A i B reda $n/2$ (tj. $m = n/2$). Primjenom prethodne leme, invertiranje matrice T svodi se na dva invertiranja matrica reda $n/2$ i dva matrična množenja.

Matrice A i B su opet donjetrokutaste, pa na njih rekurzivno primjenjujemo prethodnu lemu, sve dok matrice A i B ne postanu reda 1. Tada je njihov inverz recipročna vrijednost odgovarajućeg broja.

Paralelni potprogram koji realizira prethodni algoritam je rekurzivan i glasi:

```
function TriInv(T, n);    { vraća T^{-1} }
begin
if n = 1 then
  T^{-1} := 1/T
else
  begin
  parallel begin
    A^{-1} := TriInv(A, n/2);
    B^{-1} := TriInv(B, n/2);
  parallel end;
  D := -B^{-1} C A^{-1};
  T^{-1} := [ A^{-1}  0      ]
            [ D       B^{-1} ];
  end;
end;    {TriInv}
```

Pretpostavimo da ponovno imamo na raspolaganju n^3 procesora. Složenost prethodnog algoritma izražena je rekurzivnom relacijom

$$T(n) = T(n/2) + \mathcal{O}(\log n) \quad ,$$

pri čemu prvi član s desne strane dolazi od paralelnog invertiranja matrica A i B, a drugi član zbog nalaženja matrice D, tj. matričnog množenja matrice C s odgovarajućim inverzima, što (uz dovoljan broj procesora) možemo realizirati paralelno u vremenu $\mathcal{O}(\log n)$ (vidjeti prethodni odjeljak).

Rješenje prethodne rekurzije možemo dobiti ili raspisivanjem, ili rješavanjem nehomogene rekurzije. Vrijedi redom:

$$\begin{aligned} T(n) &= T(n/2) + \mathcal{O}(\log n) \\ T(n/2) &= T(n/4) + \mathcal{O}(\log n/2) \\ T(n/4) &= T(n/8) + \mathcal{O}(\log n/4) \\ &\vdots \\ T(2) &= T(1) + \mathcal{O}(\log 2) \\ T(1) &= \text{const} \quad . \end{aligned}$$

Uvrštavanjem svake slijedeće jednakosti u prethodnu, dobivamo da se članovi oblika $\mathcal{O}(\log n/2^k)$ javljaju točno onoliko puta koliko je jednakosti, a to je $\lg n$. Zbog toga, možemo zaključiti da je složenost prethodnog algoritma $\mathcal{O}(\log^2 n)$.

Zadatak 4.5.1.

Precizno dokažite da je $T(n) = \mathcal{O}(\log^2 n)$.

Zadatak 4.5.2.

Kolika je složenost algoritma TriInv*, ako inverze matrica A i B računamo sekvencijalno, a ne paralelno, s tim da sve ostale operacije (množenje matrica) ostaju paralelne? Što dobivamo za složenost, ako je cijeli algoritam sekvencijalan? Usporedite rezultat s poznatim algoritmima za invertiranje trokutastih matrica.*

Zadatak 4.5.3.

U algoritmu TriInv *pretpostavili smo da matrica T ima red $n = 2^k$ i da je $m = n/2$, tj. blokovi A i B imaju isti red.*

- *Analizirajte složenost u funkciji od m, ako m varira od 1 do $n-1$. Pokažite da $m = n/2$ daje najmanju složenost.*
- *Ako red n matrice T nije potencija od 2, matricu T možemo proširiti (dopuniti) do trokutaste matrice T_1 čiji red je potencija od 2. Kako treba konstruirati T_1 i kako se dobiva inverz od T iz inverza od T_1?*

Modificirajte algoritam tako da radi (bez dopune matrice T) za bilo koji red n i analizirajte njegovu složenost u funkciji od n i m. Koji izbor za m daje najmanju složenost?

4.6. Invertiranje punih matrica reda n u vremenu $\mathcal{O}(\log^2 n)$

Algoritam za invertiranje punih matrica reda n konstruirao je Csanky (1977.). Algoritam zahtijeva četiri leme. Prva lema je Hamilton–Cayley-ev teorem.

Lema 4.6.1. (Teorem Hamilton–Cayley)

Neka je A zadana matrica reda n i neka je $p(x) = \det(xI - A)$ njen karakteristični polinom, u oznaci

$$p(x) = x^n + c_{n-1}x^{n-1} + \cdots + c_1 x + c_0 \quad .$$

Nultočke tog polinoma $\lambda_1, \ldots \lambda_n$ su karakteristične (svojstvene) vrijednosti matrice A. Matrica A poništava svoj karakteristični (svojstveni) polinom, tj. vrijedi

$$p(A) = A^n + c_{n-1}A^{n-1} + \cdots + c_1 A + c_0 I = 0 \quad .$$

Također, vrijedi $c_0 = (-1)^n \det A$ i $c_{n-1} = -\operatorname{tr} A$. ■

Dokaz prethodnog teorema može se naći u mnogim udžbenicima linearne algebre.

Ako je matrica A nesingularna, njena je determinanta različita od 0 ($c_0 \neq 0$), pa korištenjem Hamilton–Cayley-evog teorema, A^{-1} možemo izraziti kao polinom u A. Množenjem $p(A)$ s A^{-1} i dijeljenjem s $-c_0$, dobivamo

$$A^{-1} = \frac{1}{-c_0}\left(A^{n-1} + c_{n-1}A^{n-2} + \cdots + c_1 I\right) \quad .$$

Dakle, ako znamo izračunati koeficijente c_i, znamo izračunati i A^{-1}. Slijedeća lema daje način na koji to možemo učiniti.

Lema 4.6.2. (Newtonovi identiteti)

Neka je

$$p(x) = x^n + c_{n-1}x^{n-1} + \cdots + c_1 x + c_0$$

polinom stupnja n s korijenima (nultočkama) $\lambda_1, \ldots, \lambda_n$. Neka je

$$s_k = \sum_{i=1}^{n} \lambda_i^k \quad , \quad k \in \mathbb{N}_0 \quad ,$$

zbroj k–tih potencija korijena. Tada koeficijenti c_k i brojevi s_k zadovoljavaju slijedeći trokutasti sistem linearnih jednadžbi:

$$\begin{bmatrix} 1 & & & & & \\ s_1 & 2 & & & & \\ s_2 & s_1 & 3 & & & \\ s_3 & s_2 & s_1 & 4 & & \\ \vdots & \vdots & \vdots & \ddots & \ddots & \\ s_{n-1} & s_{n-2} & \cdots & \cdots & s_1 & n \end{bmatrix} \begin{bmatrix} c_{n-1} \\ c_{n-2} \\ \vdots \\ \vdots \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} -s_1 \\ -s_2 \\ \vdots \\ \vdots \\ -s_{n-1} \\ -s_n \end{bmatrix} \quad .$$

Dokaz:

Derivaciju karakterističnog polinoma izrazimo na dva načina. Prvo, vrijedi

$$p'(x) = nx^{n-1} + (n-1)c_{n-1}x^{n-2} + \cdots + c_1 \quad .$$

Ako su $\lambda_1, \ldots, \lambda_n$ nultočke polinoma p, onda ga možemo napisati u formi

$$p(x) = \prod_{i=1}^{n}(x - \lambda_i) \quad .$$

Deriviranjem tog zapisa za polinom p, dobivamo

$$p'(x) = \sum_{i=1}^{n} \frac{p(x)}{x - \lambda_i} \quad .$$

Za fiksni i, dijeljenjem izlazi

$$\begin{aligned}\frac{p(x)}{x-\lambda_i} &= x^{n-1} + (\lambda_i + c_{n-1})x^{n-2} + (\lambda_i^2 + c_{n-1}\lambda_i + c_{n-2})x^{n-3} + \\ &\quad \cdots + (\lambda_i^{n-1} + c_{n-1}\lambda_i^{n-2} + \cdots + c_1)x^0\end{aligned}$$

s ostatkom

$$\lambda_i^n + c_{n-1}\lambda_i^{n-1} + \cdots + c_1\lambda_i + c_0 = p(\lambda_i) = 0 \quad .$$

Drugim riječima, vrijedi

$$\frac{p(x)}{x-\lambda_i} = \sum_{k=0}^{n-1}\left(\sum_{\ell=k+1}^{n} c_\ell \lambda_i^{\ell-k-1}\right)x^k \quad ,$$

pri čemu definiramo $c_n = 1$. Zbrajanjem svih takvih relacija po $i = 1, \ldots, n$, izlazi

$$\sum_{i=1}^{n}\frac{p(x)}{x-\lambda_i} = \sum_{i=1}^{n}\left(\sum_{k=0}^{n-1}\left(\sum_{\ell=k+1}^{n} c_\ell \lambda_i^{\ell-k-1}\right)x^k\right) = \sum_{k=0}^{n-1}\left(\sum_{\ell=k+1}^{n} c_\ell\, s_{\ell-k-1}\right)x^k \quad .$$

Izjednačavanjem koeficijenata u $p'(x)$ dobivamo

$$\sum_{\ell=k+1}^{n} c_\ell\, s_{\ell-k-1} = (k+1)c_{k+1} \quad , \quad \text{za } k = 0, 1, \ldots, n-1 \quad .$$

Zamjenom varijabli $k' = n - k - 1$, dobivamo

$$\sum_{\ell=n-k'}^{n} c_\ell\, s_{k'+\ell-n} - (n-k')c_{n-k'} = 0 \quad , \quad \text{za } k' = n-1, \ldots, 0 \quad .$$

Analizirajmo detaljnije ove relacije. Uočimo da, po definiciji, vrijedi

$$s_0 = \sum_{i=1}^{n} \lambda_i^0 = n \quad .$$

Prvi član u sumi, za $\ell = n - k'$, glasi $c_{n-k'}s_0 = n\, c_{n-k'}$. Za $k' = 0$, to je i jedini član sume, pa cijela relacija glasi $n\, c_{n-k'} - n\, c_{n-k'} = 0$, tj. trivijalno.

Ako je $k' \geq 1$, suma ima barem 2 člana. Njen zadnji član, za $\ell = n$, glasi $c_n s_{k'} = s_{k'}$, jer je $c_n = 1$, po definiciji. Prebacimo taj član na desnu stranu. Dobivamo

$$\sum_{\ell=n-k'+1}^{n-1} c_\ell\, s_{k'+\ell-n} + k'c_{n-k'} = -s_{k'} \quad , \quad \text{za } k' = 1, \ldots, n-1 \quad .$$

To su upravo jednadžbe trokutastog linearnog sustava iz tvrdnje leme. ■

Slijedeće dvije leme pokazuju kako možemo izračunati koeficijente s_k.

Lema 4.6.3.

Trag matrice A definira se kao

$$\operatorname{tr} A = \sum_{i=1}^{n} a_{ii} \quad ,$$

ali vrijedi i

$$\operatorname{tr} A = \sum_{i=1}^{n} \lambda_i \quad .$$

Dokaz:

Izjednačavanjem koeficijenta karakterističnog polinoma p uz x^{n-1}

$$x^n + c_{n-1}x^{n-1} + \cdots + c_1 x + c_0 = \prod_{i=1}^{n} (x - \lambda_i)$$

dobivamo tvrdnju. (Ova lema standardni je dio Hamilton–Cayley-evog teorema i obično se tamo i izriče.) ■

Posljednja potrebna lema govori o odnosu svojstvenih vrijednosti matrice i svojstvenih vrijednosti potencije te iste matrice.

Lema 4.6.4.

Ako su svojstvene vrijednosti matrice A jednake $\lambda_1, \ldots, \lambda_n$, *onda su svojstvene vrijednosti matrice* A^k *jednake* $\lambda_1^k, \ldots, \lambda_n^k$.

Dokaz:

Ako je λ svojstvena vrijednost matrice A, onda (po definiciji) postoji svojstveni vektor $x \neq 0$ takav da je

$$Ax = \lambda x \quad .$$

Za A^k vrijedi

$$A^k x = A^{k-1}Ax = \lambda A^{k-1}x = \lambda A^{k-2}Ax = \lambda^2 A^{k-2}x = \cdots = \lambda^k x \quad .$$

Prema tome, x je svojstveni vektor, a λ^k svojstvena vrijednost matrice A^k. ■

Sada možemo napisati osnovne korake algoritma za invertiranje matrica:

Algoritam 4.6.1. (Csanky-jev algoritam za invertiranje matrica)

1. `Izračunavanje potencija` A^k, $k = 2, \ldots, n-1$,
   `korištenjem paralelnog prefiksa`;
2. `Izračunavanje` $s_k = \operatorname{tr}(A^k)$;
3. `Rješavanje Newtonovih jednadžbi za` c_i;
4. `Izvrednjavanje` A^{-1} `korištenjem Hamilton-Cayley-evog teorema`.

Analizirajmo složenost prethodnog algoritma.

U prvom koraku računamo potencije matrice A, paralelnim prefiksom (tj. scanom) s operacijom matričnog množenja. Potrebno vrijeme je $\mathcal{O}(\log n)$ koraka paralelnog prefiksa, puta trajanje svake pojedine operacije u paralelnom prefiksu. Svako množenje matrica reda n možemo paralelno izvesti u vremenu $\mathcal{O}(\log n)$. Dakle, ukupno potrebno vrijeme za prvi korak je $\mathcal{O}(\log^2 n)$, uz dovoljan broj procesora. Trebamo barem $n^4/2$ procesora, zbog $n/2$ množenja matrica reda n, u prvom koraku paralelnog prefiksa.

Drugi korak zahtijeva izračunavanje n suma od po n brojeva. Te sume su nezavisne pa ih možemo računti paralelno, a za svaku koristimo paralelni prefiks (operacija je zbrajanje skalara). Uz pretpostavku $n^2/2$ procesora (što je osigurano 1. korakom), potrebno vrijeme odgovara jednom paralelnom prefiksu, a to je $\mathcal{O}(\log n)$.

Treći korak algoritma obavljamo invertiranjem trokutaste matrice, pa množenjem inverza s desnom stranom. Invertiranje trokutaste matrice zahtijeva paralelno vrijeme $\mathcal{O}(\log^2 n)$. Na kraju, za množenje matrice vektorom desne strane, treba izračunati još n skalarnih produkata, s redom $1, 2, \ldots, n$ članova. Zbog nezavisnosti, te produkte računamo paralelno. Najdulji skalarni produkt traje $\mathcal{O}(\log n)$. Dominantan član u ukupnom vremenu trećeg koraka je $\mathcal{O}(\log^2 n)$, a procesora ima dovoljno još iz prvog koraka.

Četvrti korak množi elemente matrica A^k skalarima i zbraja ih. Uz dovoljan broj procesora $(n^2/2)$, taj algoritam je, zapravo, n^2 nezavisnih paralelnih prefiksa (svaki je, zapravo, skalarni produkt!) i traje $\mathcal{O}(\log n)$ vremena.

Zbrajanjem vremena, zaključujemo da cijeli algoritam ima trajanje $\mathcal{O}(\log^2 n)$, uz barem $n^4/2$ procesora. Tablica složenosti pojedinih koraka je:

korak	paralelno vrijeme	broj procesora
1. korak	$\mathcal{O}(\log^2 n)$	$\mathcal{O}(n^4)$
2. korak	$\mathcal{O}(\log n)$	$\mathcal{O}(n^2)$
3. korak	$\mathcal{O}(\log^2 n)$	$\mathcal{O}(n^3)$
4. korak	$\mathcal{O}(\log n)$	$\mathcal{O}(n^3)$
ukupno	$\mathcal{O}(\log^2 n)$	$\mathcal{O}(n^4)$

Opisani algoritam je prilično nestabilan. Numerički je testiran za invertiranje matrice oblika $3 \cdot I_n$. Točnost računanja bila je 16 značajnih dekadskih znamenki (standardni "double precision"). Za $n > 50$, ovaj algoritam ne daje niti jednu točnu znamenku u inverzu. Uzrok nestabilnosti je veličina članova u produktu $c_k A^k$ (i c_k i elementi u A^k su ogromni). Rezultat invertiranja je $A^{-1} = 1/3 I_n$, tj. matrica s elementima $1/3$ ili 0. No, ti elemenati se računaju oduzimanjem velikih realnih brojeva. Zbog konačnosti prikaza, dolazi do kraćenja i gubitka točnosti.

Teoretski, mogli bismo prethodni algoritam modificirati tako da točnost računanja raste s porastom n. Ako u cijenu algoritma uključimo i tu dodatnu cijenu točnije aritmetike (u kojoj aritmetičke operacije na skalarima više nisu elementarne), moguće je pokazati da za vremensku složenost tog algoritma i dalje vrijedi $\mathcal{O}(\log^2 n)$. Međutim, taj neželjeni dodatak pisanja paketa za točnu aritmetiku čini ovaj algoritam neatraktivnim.

4.7. Rješavanje trodijagonalnog linearnog sistema reda n u vremenu $\mathcal{O}(\log n)$

Pretpostavimo da želimo riješiti linearni sistem $Tx = y$, gdje je T trodijagonalna matrica:

$$T = \begin{bmatrix} a_1 & b_1 & & & & \\ c_1 & a_2 & b_2 & & & \\ & c_2 & a_3 & b_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & c_{n-2} & a_{n-1} & b_{n-1} \\ & & & & c_{n-1} & a_n \end{bmatrix}.$$

Rješavanje Gaussovim eliminacijama bez pivotiranja provodi se u 3 koraka:

1. `Faktorizacija` $T = L \cdot U$, L `je donjetrokutasta s` 1 `na dijagonali,` `a` U `je gornjetrokutasta matrica`;
2. `Rješavanje trokutastog linearnog sistema` $Lz = y$;
3. `Rješavanje trokutastog linearnog sistema` $Ux = z$.

Pokažimo da korištenjem algoritama tipa paralelnog prefiksa trebamo točno $\mathcal{O}(\log n)$ operacija za svaki od ta tri koraka.

Prvo, množenjem $T = L \cdot U$, izlazi da matrice L i U imaju samo po jednu netrivijalnu sporednu dijagonalu, tj. oblik

$$L = \begin{bmatrix} 1 & & & & & \\ \ell_1 & 1 & & & & \\ & \ell_2 & 1 & & & \\ & & \ddots & \ddots & & \\ & & & \ell_{n-2} & 1 & \\ & & & & \ell_{n-1} & 1 \end{bmatrix}, \quad U = \begin{bmatrix} d_1 & e_1 & & & & \\ & d_2 & e_2 & & & \\ & & d_3 & e_3 & & \\ & & & \ddots & \ddots & \\ & & & & d_{n-1} & e_{n-1} \\ & & & & & d_n \end{bmatrix}.$$

Iz jednakosti $T_{ij} = (L \cdot U)_{ij}$, za $j = i-1, i, i+1$, dobivamo jednakosti:

$$c_{i-1} = \ell_{i-1} \cdot d_{i-1}$$

$$\begin{aligned} a_i &= \ell_{i-1} \cdot e_{i-1} + d_i \\ b_i &= e_i \quad . \end{aligned}$$

Odatle, odmah dobivamo rješenja za e_i. Rješavanjem prve jednadžbe po ℓ_{i-1} i supstituiranjem u drugu jednadžbu dobivamo

$$d_i = a_i - e_{i-1} \cdot c_{i-1}/d_{i-1} = a_i + f_i/d_{i-1} \quad ,$$

gdje je

$$f_i = -e_{i-1} \cdot c_{i-1} = -b_{i-1} \cdot c_{i-1} \quad .$$

Ako možemo riješiti tu rekurziju za sve d_i–ove, tada možemo izračunati sve $\ell_i = c_i/d_i$ u samo jednom (paralelnom) koraku.

Pokažimo kako možemo, korištenjem paralelnog prefiksa, izračunati članove te rekurzije u vremenu $\mathcal{O}(\log n)$. Napišimo d_i u obliku $d_i = p_i/q_i$ i uvrstimo to u rekurziju. Vrijedi

$$\frac{p_i}{q_i} = a_i + \frac{f_i}{p_{i-1}/q_{i-1}} = \frac{a_i \cdot p_{i-1} + f_i \cdot q_{i-1}}{p_{i-1}} \quad .$$

Sada stavimo

$$p_i = a_i \cdot p_{i-1} + f_i \cdot q_{i-1} \quad , \quad q_i = p_{i-1} \quad ,$$

odnosno

$$\begin{aligned} \begin{bmatrix} p_i \\ q_i \end{bmatrix} &= \begin{bmatrix} a_i \cdot p_{i-1} + f_i \cdot q_{i-1} \\ p_{i-1} \end{bmatrix} = \begin{bmatrix} a_i & f_i \\ 1 & 0 \end{bmatrix} \begin{bmatrix} p_{i-1} \\ q_{i-1} \end{bmatrix} \\ &= M_i \cdot \begin{bmatrix} p_{i-1} \\ q_{i-1} \end{bmatrix} = M_i \cdot M_{i-1} \cdots M_1 \cdot \begin{bmatrix} p_0 \\ q_0 \end{bmatrix} = N_i \cdot \begin{bmatrix} p_0 \\ q_0 \end{bmatrix} \quad , \end{aligned}$$

gdje je $f_1 = 0$, $p_0 = 1$ i $q_0 = 0$. Zbog toga, trebamo izračunati produkt od n matrica M_i, reda 2. Računanje se vrši paralelnim prefiksom, a matrično množenje 2×2 matrica je pridružena asocijativna binarna operacija. Potreban broj koraka je $\mathcal{O}(\log n)$, pa faktorizaciju matrice $T = L \cdot U$ možemo izvršiti paralelno u vremenu $\mathcal{O}(\log n)$.

Razmotrimo još i rješavanje trokutastog linearnog sistema $Ux = z$. Rješavanje sistema $Lz = y$ bit će analogno. Ovaj problem opet pretvaramo u paralelni prefiks, s množenjem matrica reda 2 kao binarnom operacijom. Izjednačavanje $(Ux)_i = z_i$ daje

$$d_i x_i + b_i x_{i+1} = z_i \quad .$$

Preuređivanjem dobivamo rekurziju

$$x_i = -b_i/d_i \cdot x_{i+1} + z_i/d_i = \alpha_i x_{i+1} + \beta_i \quad .$$

Taj linearni sistem rješavamo po opadajućim indeksima i. Zapisom x_i u obliku razlomka s nazivnikom 1, prethodnu relaciju pretvaramo u matrično množenje.

$$\begin{aligned} \begin{bmatrix} x_i \\ 1 \end{bmatrix} &= \begin{bmatrix} \alpha_i \cdot x_{i+1} + \beta_i \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha_i & \beta_i \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{i+1} \\ 1 \end{bmatrix} \\ &= P_i \cdot \begin{bmatrix} x_{i+1} \\ 1 \end{bmatrix} = P_i \cdot P_{i+1} \cdots P_{n-1} \cdot \begin{bmatrix} x_n \\ 1 \end{bmatrix} = Q_i \cdot \begin{bmatrix} x_n \\ 1 \end{bmatrix} \quad , \end{aligned}$$

gdje je $x_n = z_n/d_n$. Matrice Q_i možemo izračunati u vremenu $\mathcal{O}(\log n)$, korištenjem "paralelnog sufiksa". To je operacija tipa "scan", u kojoj se računaju stražnje (a ne prednje) parcijalne sume. Očito, dovoljno je preokrenuti numeraciju P_i matrica, da dobijemo paralelni prefiks.

Algoritam dekompozicije matrice T radi bez pivotiranja, što može biti uzrok nestabilnosti algoritma.

Kombiniranjem tehnika prošlog i ovog odjeljka, može se pokazati da je za rješavanje vrpčastog linearnog sistema reda n, širine vrpce m, potrebno vrijeme $\mathcal{O}(\log n \log m)$.

4.8. Parno–neparna redukcija za trodijagonalne linearne sisteme

Trodijagonalni linearni sistem reda n možemo riješiti i na drugačiji način u vremenu $\mathcal{O}(\log n)$. Ponovno, neka je zadan trodijagonalni linearni sistem $Tx = y$, gdje je T trodijagonalna matrica oblika:

$$T = \begin{bmatrix} a_1 & b_1 & & & & \\ c_1 & a_2 & b_2 & & & \\ & c_2 & a_3 & b_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & c_{n-2} & a_{n-1} & b_{n-1} \\ & & & & c_{n-1} & a_n \end{bmatrix} .$$

Osnovna ideja parno–neparne redukcije je rješavanje k–te jednadžbe u linearnom sistemu korištenjem dvije susjedne, tj. po varijablama x_{k-1} i x_{k+1}. Ako to napravimo za sve jednadžbe s neparnim indeksom, ostao je ponovno trodijagonalni linearni sistem (što treba još pokazati), ali s polovičnim brojem jednadžbi i varijabli. Radi jednostavnosti zapisa za sve jednadžbe, definirajmo $x_0 = x_n = 0$ i $c_0 = b_n = 0$.

U linearnom sustavu $Tx = y$, k–ta jednadžba je

$$c_{k-1}x_{k-1} + a_k x_k + b_k x_{k+1} = y_k \quad ,$$

odakle, uz pretpostavku $a_k \neq 0$, proizlazi:

$$x_k = \frac{1}{a_k}\left(y_k - c_{k-1}x_{k-1} - b_k x_{k+1}\right) \quad .$$

Korištenjem prethodne jednakosti za $k = i-1$ i $k = i+1$, eliminirali smo varijable x_{i-1} i x_{i+1} u i–toj jednadžbi. Vrijedi

$$\frac{c_{i-1}}{a_{i-1}}\left(y_{i-1} - c_{i-2}x_{i-2} - b_{i-1}x_i\right) + a_i x_i + \frac{b_i}{a_{i+1}}\left(y_{i+1} - c_i x_i - b_{i+1}x_{i+2}\right) = y_i \quad .$$

Sređivanjem izlazi:

$$-\frac{c_{i-1}c_{i-2}}{a_{i-1}}x_{i-2} + \left(a_i - \frac{b_{i-1}c_{i-1}}{a_{i-1}} - \frac{b_i c_i}{a_{i+1}}\right)x_i - \frac{b_i b_{i+1}}{a_{i+1}}x_{i+2} = y_i - \frac{c_{i-1}y_{i-1}}{a_{i-1}} - \frac{b_i y_{i+1}}{a_{i+1}} \quad .$$

Prethodna jednadžba vrijedi za sve parne indekse i, $1 \leq i \leq n$. Očito se radi o trodijagonalnom linearnom sistemu u varijablama $x_2, x_4, \ldots, x_{2\lfloor n/2 \rfloor}$.

Ovu proceduru eliminacije varijabli koristimo rekurzivno, sve dotle dok ne ostane ili jedna ili dvije jednadžbe. Nekom vrstom povratne supstitucije (u povratku iz rekurzivne eliminacije), tada se ta izračunata rješenja uvrštavaju u ostale jednadžbe (koje su bile eliminirane na tom koraku rekurzije).

Uočimo da se u svakom koraku eliminacije, broj varijabli smanji na približno pola, što znači da ukupno imamo $\mathcal{O}(\log n)$ koraka eliminacije. Ako imamo dovoljno procesora (tj. ako je broj procesora proporcionalan broju nepoznanica), onda svaki od procesora u svakom koraku obavlja nekoliko zbrajanja, množenja i dijeljenja. Preciznije, k–ti procesor obavlja samo eliminaciju nepoznanica u k–toj jednadžbi. Slično vrijedi i za povratnu supstituciju.

Prema tome, vrijeme potrebno za rješavanje trodijagonalnog linearnog sistema prema ovom algoritmu je

$$T(n) = \mathcal{O}(\log n) \quad .$$

Primijetimo da je broj ukupan broj računskih (aritmetičkih) operacija u samom algoritmu $\mathcal{O}(n)$, jer se u svakom koraku broj nepoznanica reducira na približno pola. To implicira da postoji sekvencijalni algoritam koji zadani problem rješava u vremenu $\mathcal{O}(n)$. Jedan takav algoritam su standardne Gaussove eliminacije za trodijagonalni linearni sistem. Taj algoritam je ekvivalentan sekvencijalnom izvođenju algoritma iz prethodnog odjeljka, bez svođenja na paralelni prefiks.

4.9. Zbrajanje dva n–bitna cijela broja u vremenu $\mathcal{O}(\log n)$

Neka su

$$a = a_{n-1}a_{n-2}a_{n-3}\cdots a_0 \quad \text{i} \quad b = b_{n-1}b_{n-2}b_{n-3}\cdots b_0$$

dva n–bitna binarna broja s binarnim znamenkama a_{n-1} do a_0, odnosno, b_{n-1} do b_0. Želimo izračunati zbroj ta dva broja, tj. broj $s = a + b$, gdje je

$$s = s_n s_{n-1} s_{n-2} s_{n-3} \cdots s_0 \quad .$$

(Zbroj se može produljiti za jednu znamenku.)

Standardni algoritam zbraja ta dva broja zdesna ulijevo, propagirajući prijenos duljine jednog bita (engl. carry–bit) c_i.

Algoritam 4.9.1. (Standardno zbrajanje binarnih brojeva)

Ako znamenku 0 *kodiramo kao false, a* 1 *kao true, pri čemu su* **xor***,* **or** *i* **and** *standardne logičke operacije, onda slijedeći algoritam zbraja dva n–bitna broja*

```
begin
c[−1] := false;
for i := 0 to n − 1 do
  begin
  c[i] := ((a[i] xor b[i]) and c[i − 1]) or (a[i] and b[i]);
  s[i] := a[i] xor b[i] xor c[i − 1];
  end;   {for i}
s[n] := c[n − 1];
end;
```

Najveći izazov prethodnog algoritma je brzo propagiranje prijenosa c_i zdesna ulijevo. Ako znamo prijenose c_i za svaki i, onda se znamenke s_i zbroja računaju paralelno, u jednom vremenskom koraku.

Označimo "propagiranje bita prijenosa" na i–tom mjestu (c_{i-1} se prenosi na c_i) s

$$p_i = a_i \textbf{ xor } b_i \quad ,$$

a "generiranje bita prijenosa" (c_i) na i–tom mjestu s

$$q_i = a_i \textbf{ and } b_i \quad .$$

Ove definicije reduciraju rekurziju za bit prijenosa c_i na "propagiraj c_{i-1} ili generiraj"

$$c_i = (p_i \textbf{ and } c_{i-1}) \textbf{ or } q_i \quad .$$

Prethodnu rekurziju izvrednjavamo korištenjem paralelnog prefiksa, pri čemu je pridružena asocijativna operacija matrično množenje dvije Booleove matrice (matrica s elementima *true* ili *false*) reda 2. Vrijedi:

$$\begin{bmatrix} c_i \\ \mathit{true} \end{bmatrix} = \begin{bmatrix} p_i \textbf{ and } c_{i-1} \textbf{ or } q_i \\ \mathit{true} \end{bmatrix} = \begin{bmatrix} p_i & q_i \\ \mathit{false} & \mathit{true} \end{bmatrix} \begin{bmatrix} c_{i-1} \\ \mathit{true} \end{bmatrix}$$
$$= C_i \cdot \begin{bmatrix} c_{i-1} \\ \mathit{true} \end{bmatrix} = C_i \cdot C_{i-1} \cdots C_0 \cdot \begin{bmatrix} c_{-1} \\ \mathit{true} \end{bmatrix} \quad .$$

Množenje Booleovih matrica je asocijativno, zbog toga što su operacije **and** i **or** distributivne i zadovoljavaju iste zakone distributivnosti kao obično zbrajanje i množenje (**and** odgovara množenju, a **or** zbrajanju).

Prethodni algoritam zove se još i "pogledaj unaprijed za prijenos" (eng. "carry look–ahead") i koristi se u svakom mikroprocesoru za cjelobrojno zbrajanje. Algoritam je u svojoj rudimentarnoj formi bio poznat Babbageu u devetnaestom stoljeću.

4.10. Paralelno izvrednjavanje rekurzija

Razmotrimo paralelno izvrednjavanje rekurzije

$$z_i = f_i(z_{i-1})$$

pri čemu je z_i broj, a f_i je racionalna funkcija. Pretpostavimo da racionalna funkcija f_i ima oblik

$$f_i(z) = \frac{\widetilde{p}_i(z)}{\widetilde{q}_i(z)}$$

pri čemu su $\widetilde{p}_i$ i $\widetilde{q}_i$ polinomi koji nemaju zajednički faktor. Također, neka je stupanj svake racionalne funkcije f_i jednak d (to je veći od stupnjeva polinoma $\widetilde{p}_i$ i $\widetilde{q}_i$). Na ovom mjestu prezentirat ćemo Kungov teorem koji govori o tome kad se z_n može efikasno paralelno izračunati (u ovisnosti o n), iz zadanog z_0, uz zadane racionalne funkcije f_i, za $i = 1, \ldots, n$.

Teorem 4.10.1. (Kung)

Neka je z_i definiran racionalnom rekurzijom $z_i = f_i(z_{i-1})$ fiksnog stupnja d, kao gore. Tada vrijedi:

1. *ako je $d = 1$, z_n se može izračunati u $\mathcal{O}(\log n)$ vremena, korištenjem paralelnog prefiksa, s matričnim množenjem matrica reda 2 kao asocijativnom operacijom;*

2. *ako je $d > 1$, izvrednjavanje z_n korištenjem samo racionalnih operacija (zbrajanja, oduzimanja, množenja i dijeljenja), traje $\Omega(n)$ vremena, tj. paralelno ubrzanje nije moguće.*

Drugim riječima, paralelni prefiks, s matričnim množenjem matrica reda 2 kao asocijativnom operacijom, nužan je i dovoljan za paraleliziranje svih racionalnih rekurzija koje se uopće mogu paralelizirati.

Dokaz:

Suština dokaza svodi se na činjenicu da supstitucija linearnih (racionalnih) izraza u linearne (racionalne) izraze **ne mijenja** stupanj, tj. izraz ostaje linearan. U protivnom, za izraze stupnja većeg od 1, supstitucijom se stupnjevi množe, tj. stupanj eksponencijalno raste s brojem supstitucija.

Prvo, razmotrimo slučaj $d = 1$. Tada svaka racionalna funkcija $f_i(z)$ ima oblik

$$f_i(z) = \frac{a_i \cdot z + b_i}{c_i \cdot z + d_i} \quad .$$

Upotrijebimo slični trik kao ranije, tj. napišimo z_i u obliku razlomka

$$z_i = \frac{p_i}{q_i} \quad ,$$

i nađimo rekurzije za brojnik i nazivnik. Supstutucijom dobivamo

$$\begin{aligned} z_i &= f_i(z_{i-1}) = \frac{p_i}{q_i} = \frac{a_i \cdot z_{i-1} + b_i}{c_i \cdot z_{i-1} + d_i} \\ &= \frac{a_i \cdot p_{i-1}/q_{i-1} + b_i}{c_i \cdot p_{i-1}/q_{i-1} + d_i} = \frac{a_i \cdot p_{i-1} + b_i \cdot q_{i-1}}{c_i \cdot p_{i-1} + d_i \cdot q_{i-1}} \quad . \end{aligned}$$

Napišimo prethodnu relaciju u obliku matričnog množenja:

$$\begin{aligned} \begin{bmatrix} p_i \\ q_i \end{bmatrix} &= \begin{bmatrix} a_i \cdot p_{i-1} + b_i \cdot q_{i-1} \\ c_i \cdot p_{i-1} + d_i \cdot q_{i-1} \end{bmatrix} = \begin{bmatrix} a_i & b_i \\ c_i & d_i \end{bmatrix} \begin{bmatrix} p_{i-1} \\ q_{i-1} \end{bmatrix} \\ &= M_i \cdot \begin{bmatrix} p_{i-1} \\ q_{i-1} \end{bmatrix} = M_i \cdot M_{i-1} \cdots M_1 \cdot \begin{bmatrix} p_0 \\ q_0 \end{bmatrix} = N_i \cdot \begin{bmatrix} p_0 \\ q_0 \end{bmatrix} \quad . \end{aligned}$$

Matrice N_i se dobivaju kao paralelni prefiks ("scan") niza matrica M_i. Dakle, sve matrice N_i, a onda i svi z_i, mogu se izračunati u $\mathcal{O}(\log n)$ vremena, kao što smo željeli pokazati.

Neka je sada $d > 1$. Želimo pokazati da izvrednjavanje z_n traje barem proporcionalno s n. Da bismo to postigli, promotrimo z_n kao racionalnu funkciju od polazne vrijednosti z_0 i upitajmo se koji je njen stupanj. Preciznije, ako z_n napišemo kao racionalnu funkciju od z_0, treba odrediti koji je maksimalni stupanj polinoma

koji se javlja u brojniku ili nazivniku. Budući da se d–te potencije polinoma javljaju u svakom od f_i–jeva, stupanj polinoma raste s faktorom d puta u svakom koraku, tj. stupanj od z_n je najviše d^n. Pažljivim sređivanjem dobit ćemo da je taj stupanj jednak baš d^n, osim u "sretnom" slučaju da zbrajanje rezultira kraćenjem vodećih članova polinoma (što je rijetkost i izuzetak od općeg rezultata).

Pretpostavimo da izvrednjavamo z_n u m paralelnih (vremenskih) koraka koristeći samo racionalne operacije. Pokazat ćemo da je tada stupanj od z_n (u ovisnosti o z_0) najviše 2^m. Da bismo "pokrili" korektan stupanj polinoma z_n, onda mora vrijediti $2^m \geq d^n$, odnosno, jednostavnije, $m \geq n$, zbog toga što je $d > 1$. Zbog toga trebamo više od n paralelnih koraka za izvrednjavanje z_n.

Pokažimo još da je nakon m vremenskih koraka, stupanj z_n jednak najviše 2^m. Dokaz se provodi indukcijom po m. Nakon $m = 0$ koraka, jedina mogućnost je $z_n = z_0$, tj. z_n ima stupanj $1 = 2^0$. Pretpostavimo da nakon $m - 1$ koraka, z_n ima stupanj 2^{m-1}. U m–tom koraku možemo primijeniti jednu od četiri dozvoljene racionalne operacije na dvije racionalne funkcije stupnja najviše 2^{m-1}. Sada je lako vidjeti da će primjena neke od te četiri operacije na te dvije racionalne funkcije, u najgorem slučaju, povećati njihov stupanj 2 puta (množenjem, nakon sređivanja brojnika i nazivnika!). ■

Naglasimo još jednom da je prethodni teorem istinit, jer dozvoljavamo samo zbrajanje, oduzimanje, množenje i dijeljenje. Ako dozvoljavamo, na primjer, transcendentalne operacije — druge korijene, eksponenciranje ili logaritmiranje, moguće je daljnje ubrzanje. Na primjer, za nalaženje pozitivnog drugog korijena iz nekog broja a, možemo koristiti Newtonovu metodu za rješavanje jednadžbe

$$g(x) = x^2 - a = 0 \quad .$$

To daje rekurziju

$$x_{i+1} = \frac{x_i^2 + a}{2x_i} = f(x_i) \quad ,$$

čiji je stupanj jednak 2 pa njeno paralelno ubrzanje nije moguće. Izvršimo zamjenu varijabli korištenjem transcendentalne (neracionalne) transformacije

$$x_i = \sqrt{a} \cdot \mathrm{cth}(y_i) = \sqrt{a} \cdot \frac{\exp(y_i) + \exp(-y_i)}{\exp(y_i) - \exp(-y_i)} \quad .$$

Dobivamo rekurziju

$$y_{i+1} = 2y_i \quad ,$$

koja, očito, ima stupanj 1, tj. može se ubrzati. Dakle, ovakva zamjena varijable smanjila je stupanj rekurzije.

Nažalost, time nismo ništa dobili. Ova rekurzija za y_i ima trivijalno rješenje $y_\infty = \infty$, odakle, očekivano, slijedi $x_\infty = \sqrt{a}\,\mathrm{cth}(\infty) = \sqrt{a}$. Međutim, ako ne znamo izračunati $\sqrt{a}$, a to je polazni problem, iz brzo izračunatih $\mathrm{cth}(y_i)$–ova (koji daju sve bolje aproksimacije jedinice), nećemo moći dobiti x_i–ove.

4.11. Paralelno izvrednjavanje izraza

Ovo poglavlje završimo Brentovim teoremom.

Teorem 4.11.1. (Brent)

Neka je E proizvoljni izraz koji se sastoji od operacija $+$, $-$, $\cdot$, $/$, zagrada i n varijabli, ali tako da se svaka pojava varijable računa zasebno. Ili drugačije: neka je E proizvoljno binarno stablo izraza s n listova (varijabli), čije je svaki unutarnji čvor označen s jednom od operacija $+$, $-$, $\cdot$, $/$. Tada se E može izračunati u $\mathcal{O}(\log n)$ vremena korištenjem komutativnosti, asocijativnosti i distributivnosti. ■

Moderniji dokaz prethodnog teorema koristi pohlepni algoritam, koji u svakom vremenskom koraku sve listove stabla (paralelno) sažima u njihove roditelje, a svi lanci (nizovi istih operacija) se izvrednjavaju korištenjem paralelnog prefiksa.

Algoritmi iz ovog poglavlja imaju, uglavnom, samo teorijsku važnost, u smislu — koliko brzo možemo neke operacije izvesti paralelno, ne vodeći računa o broju procesora i njihovoj povezanosti.

Ova situacija nije realna. Na stvarnim arhitekturama računala, dohvaćanje podataka iz memorije je bitno skuplja operacija od računanja, tj. treba voditi računa o mreži povezivanja — procesora međusobno, i procesora s memorijom. Također, ovi rezultati o paralelnoj vremenskoj složenosti podrazumijevaju da je broj procesora neograničen, u smislu da raste s veličinom problema. To, u praksi, nije slučaj. Broj procesora je (više, manje) fiksan, a zadanu veličinu problema treba pametno raspodijeliti na raspoložive procesore.

Drugim riječima, Brentov slučaj Munro–Patersonovog teorema **uvijek** dominira u praksi, ili:

- broj procesora je **mali** obzirom na veličinu problema koji se rješava,
- pristup podacima je (vrlo) spor, obzirom na brzinu računanja.

5. Matrično množenje na hijerarhijskoj memoriji

Na računalima s hijerarhijskom memorijom mogu se brzo realizirati neki osnovni algoritmi linearne algebre, kao što je matrično množenje. "Građevni blokovi" za dizajniranje takvih algoritama standardizirani su u biblioteci nazvanoj BLAS (Basic Linear Algebra Subroutines). Motivacija za takvu biblioteku je jednostavna: prenosivost programa s jednog računala na drugo, s ciljem da se maksimalno iskoristi snaga svakog pojedinog računala. BLAS je zbirka potprograma (uglavnom: FORTRAN 77, a, modernije: FORTRAN 90 ili 95, ili C, C++), koju proizvođači optimiziraju za svako računalo posebno (i trebali bi ju isporučiti zajedno s računalom, za potrebe tzv. znanstvenog računanja – "scientific computing").

5.1. BLAS

Projekt pisanja potprograma BLAS započet je 1973. kao privatna inicijativa za standardizaciju osnovnih algoritama. Ubrzo projekt podupire ACM–Signum, pa je uskoro publicirana i prva takva jednostavna biblioteka BLAS 1.

Danas postoji podjela BLAS-a na 3 nivoa, prema složenosti operacija koje sadrži. Standardno, BLAS potprogrami podržavaju realnu aritmetiku jednostruke i dvostruke točnosti (početna slova u nazivima potprograma: 'S', odnosno 'D', respektivno) i kompleksnu aritmetiku jednostruke i dvostruke točnosti (početna slova u nazivima potprograma: 'C', odnosno 'Z', respektivno). Podjela prema nivoima je slijedeća:

- BLAS 1 (ili BLAS nivoa 1) operira na vektorima (jednodimenzionalnim poljima) ili parovima vektora. Ako vektori imaju duljinu n, ti potprogrami su složenosti $\mathcal{O}(n)$ operacija na skalarima i, kao rezulatat, vraćaju ili vektor ili skalar. Ta grupa potprograma obuhvaća, na primjer:
  - `xaxpy` (prvi `x` = jedno od slova prema konvenciji o imenima potprograma – '`s`', '`d`', '`c`' ili '`z`') za računanje $y = y + a \cdot x$, gdje je a skalar, a x i y su vektori;

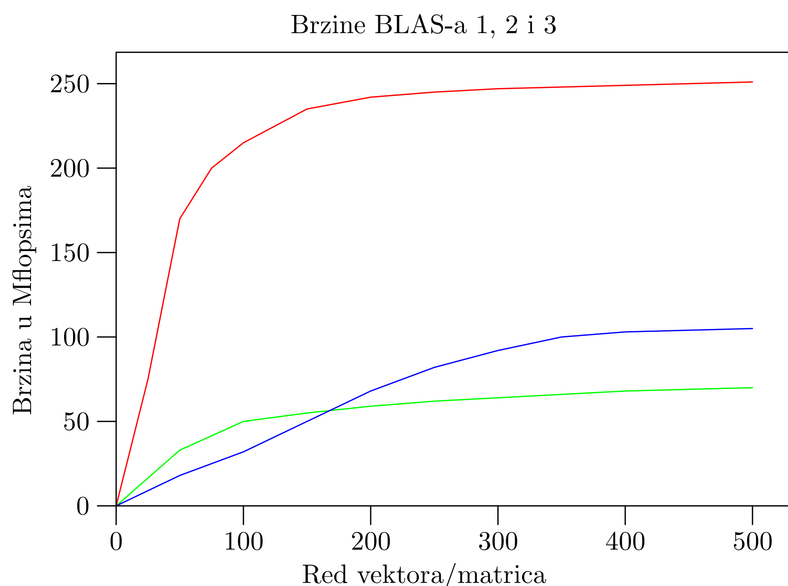
 - `xscal` za skaliranje vektora y skalarom a, tj. $y = a \cdot y$;
 - `xrot` primjenjuje Givensovu rotaciju za kut φ, s izračunatim $s = \sin\varphi$ i $c = \cos\varphi$, pri čemu se vektori x i y zamjenjuju, respektivno, s $c\cdot x+s\cdot y$, odnosno, $-s \cdot x + c \cdot y$;
 - `xdot` za računanje skalarnog produkta vektora $s = \langle x, y\rangle = \sum_{i=1}^{n} x_i \cdot \bar{y}_i$.

- BLAS 2 (ili BLAS nivoa 2) operira na matricama (dvodimenzionalnim poljima) i vektorima (jednodimenzionalnim poljima). Ako je matrica reda n, ti potprogrami su složenosti $\mathcal{O}(n^2)$ operacija na skalarima. Ta grupa potprograma obuhvaća, na primjer:
 - `xgemv` matrično–vektorsko množenje $y = y+A\cdot x$, x i y vektori, A matrica;
 - ažuriranje matrice matricom ranga 1, tj. računanje $A = A + y \cdot x^{\tau}$, A matrica, x i y vektori;
 - rješavanje trokutastih linearnih sistema $y = Tx$, T trokutasta matrica, y zadani vektor.

- BLAS 3 (ili BLAS nivoa 3) operira na parovima ili trojkama matrica (dvodimenzionalnim poljima). Ako su matrice reda n, ti potprogrami su složenosti $\mathcal{O}(n^3)$ operacija na skalarima. Ta grupa potprograma obuhvaća, na primjer:
 - matrično–matrično množenje $C = C + A \cdot B$, A, B i C matrice tipova (m, n), (n, k) i (m, k), respektivno;
 - čak preciznije, potprogram `xgemm` dozvoljava nešto općenitiju operaciju $C = b\cdot C + a\cdot \mathrm{op}\,(A)\cdot \mathrm{op}\,(B)$, pri čemu su a i b skalari, a $\mathrm{op}\,(\ )$ je opciono transponiranje matrica;
 - rješavanje trokutastih linearnih sistema s više desnih strana, tj. rješavanje $Y = T \cdot X$, gdje je T trokutasta matrica, Y zadane desne strane.

Matematički gledano, svi su ti potprogrami ekvivalentni. Na primjer, o skalarnom produktu možemo razmišljati kao o produktu $(1, n)$ i $(n, 1)$ vektora. Zbog toga, postavlja se pitanje kakve su razlike između algoritama. Odgovor je — performansa.

Na Sveučilištu Berkeley ispitivali su matrično množenje na RS 6000/590 računalu. Na slijedećoj slici, najviša krivulja dobivena je korištenjem BLAS-a 3 (matrično–matrično množenje), slijedeća po visini (za velike n) dobivena je korištenjem BLAS-a 2 (množenje matrice na vektor), a najniža (za velike n) korištenjem BLAS-a 1 (`saxpy` potprogram). Brzina dobivena BLAS-om 3 približno je jednaka maksimalnoj mogućoj brzini na tom računalu (266 Mflops-a). (Grafovi su "posuđeni" iz predavanja J. Demmela za kolegij CS 267.)



Da bismo kvantificirali usporedbe, uvedimo jednostavan model kako radi hijerarhijska memorija. Pretpostavimo da hijerarhijska memorija ima dva nivoa: brzi i spori dostup. Pretpostavljamo da su na početku svi podaci spremljeni u sporoj memoriji. Za svaki poziv potprograma `saxpy`, `sgemv`, odnosno `sgemm`, za matrice reda n i vektore duljine n, uspoređivat ćemo slijedeće podatke:

m – broj poziva spore memorije potrebnih da se pročitaju početni podaci iz nje i rezultati ponovo spreme u nju;

f – broj floating–point operacija;

q – prosječan broj flops-a po jednom pozivu spore memorije, tj. $q = f/m$.

Slijedeća tablica daje podatke o prethodnim veličinama za potprograme `saxpy`, `sgemv`, odnosno `sgemm`.

potp.	m	objašnjenje za m	f	q
`saxpy`	$3 \cdot n$	čitanje x_i i y_i jednom, pisanje y_i jednom	$2 \cdot n$	$2/3$
`sgemv`	$n^2 + \mathcal{O}(n)$	čitanje $A_{i,j}$ jednom + kao za `saxpy`	$2 \cdot n^2$	≈ 2
`sgemm`	$4 \cdot n^2$	čitanje $A_{i,j}$, $B_{i,j}$ i $C_{i,j}$ jednom, pisanje $C_{i,j}$ jednom	$2 \cdot n^3$	$n/2$

U prethodnoj tablici, očita je razlika u q–ovima za različite nivoe BLAS potprograma. Važnost veličine q leži u činjenici da se svaki podatak čita iz spore memorije

i na njemu se u prosjeku obavi q operacija dok je u brzoj memoriji. Što je veći q, algoritam više operira s podacima u brzoj memoriji, pa je efikasniji.

Postoji još jedan način opisivanja veličine q u terminima "omjera promašaja" (engl. miss ratio). Omjer promašaja mjeri omjer broja memorijskih poziva spore memorije obzirom na ukupan broj memorijskih poziva. Pojam je vezan za brze "cache" memorije (male, skupe), obzirom na sporu, standardnu memoriju (veća, jeftinija). Dobar algoritam ima nizak omjer promašaja. Ako pretpostavimo da su na početku i kraju algoritma, svi podaci u sporoj memoriji i da svaka floating–point operacija uključuje 2 poziva memorije (bilo spore, bilo brze), možemo ograditi omjer promašaja s

$$\text{omjer promašaja} = \frac{\text{broj poziva spore memorije}}{\text{ukupan broj poziva memorije}} \geq \frac{m}{2 \cdot f} = \frac{1}{2 \cdot q} \quad .$$

Dakle, možemo zaključiti, što je veći q, nadamo se da je manji omjer promašaja.

Iskoristimo vrlo jednostavan primjer za ilustraciju, zbog čega se nadamo da veći q znači veću brzinu.

Primjer 5.1.1.

Pretpostavimo da se u brzoj memoriji floating–point operacije mogu odvijati brzinom od 1 *Mflops-a. Dovlačenje podataka iz spore memorije traje* 10 *puta dulje po podatku. Pretpostavimo da imamo jednostavni (sekvencijalni) algoritam:*

begin
$s := 0$;
for $i := 1$ **to** n **do**
  $s := s + f(x_i)$;
end;

pri čemu izračunavanje $f(x_i)$ traje $q-1$ operaciju na podatku x_i, što se može obaviti kad se podatak x_i dovuče u brzu memoriju (ne spremajući ga u sporu memoriju između operacija!). Pretpostavimo da se i s nalazi u brzoj memoriji. Za ovaj algoritam je očito $m = n$, $f = q \cdot n$. Pitamo se kolika je brzina ovog programa.

Pretpostavimo da se x_i na početku nalazio u sporoj memoriji. Za dohvaćanje svakog x_i–ja algoritam potroši 10 *mikrosekundi, odnosno za sve x_i–jeve ukupno $10 \cdot n$ mikrosekundi. Zatim, potrebno je još $q \cdot n$ floating–point operacija za izračunavanje s, što traje $q \cdot n$ mikrosekundi. Ukupno, potrošeno vrijeme je tada $t = q \cdot n + 10 \cdot n$ mikrosekundi, odnosno, posao se obavlja brzinom*

$$\frac{f}{t} = \frac{q \cdot n}{q \cdot n + 10 \cdot n} = \frac{q}{q + 10} \text{ Mflops-a} \quad .$$

Porastom q, približavamo se vršnoj brzini.

Obrazložimo još neke detalje našeg jednostavnog hijerarhijskog modela:

1. Postoje samo dva nivoa hijerarhije – brzi (cache) i spori (standardna memorija). U slučaju distribuirane memorije, brza odgovara lokalnoj memoriji procesora, a spora svoj preostaloj memoriji (pristup mrežom povezivanja procesora).
2. Mala brza memorija ima kapacitet M riječi (podataka), gdje je $M \ll n^2$, pa u nju možemo smjestiti samo manji dio $n \times n$ matrice. Jednako tako, pretpostavljamo da je $M \geq 4 \cdot n$, pa se u nju može smjestiti nekoliko cijelih redaka ili stupaca.
3. Svaki podatak iz spore memorije čita se zasebno (u praksi, obično se čita više riječi u cache ili nekoliko blokova memorije, ali to ne mijenja bitno osnovnu analizu).
4. Mi imamo potpunu kontrolu nad tim kako se podaci kreću između dva memorijska nivoa. To je najbolja moguća pretpostavka, jer, obično, hardware (cache ili sistem virtuelne memorije) donosi te odluke umjesto nas. Međutim, u slučaju paralelnog računanja, ako su ta dva nivoa memorije — lokalna memorija procesora i memorije udaljenih procesora, često imamo takvu eksplicitnu kontrolu transfera podataka (htjeli mi to ili ne).

Ovaj jednostavni model koristimo za detaljniju analizu nekoliko različitih algoritama za množenje matrica (operacija `xgemm`). Na kraju ćemo konstruirati algoritam koji "optimalno" koristi hijerarhijsku memoriju.

5.2. Tri implementacije množenja matrica

Promotrimo tri različite implementacije matričnog množenja i izračunajmo faktor $q = f/m$ za svaku. Te tri verzije algoritma su:

1. bez blokova (engl. unblocked);
2. s blokovima stupaca (engl. column blocked);
3. s kvadratnim (ravninskim) blokovima (engl. square blocked, 2D blocked);

U zapisu algoritama, osim računskih operacija, pišemo i operacije pristupa sporoj memoriji. Te operacije pišemo u komentarima, ne ulazeći u detalje njihove realizacije.

5.2.1. Množenje matrica bez blokova

Algoritam množenja matrica bez blokova je klasični algoritam množenja matrica.

Algoritam 5.2.1. (Množenje matrica bez blokova)

for $i := 1$ **to** n **do**
  begin
  {`Učitati redak` i `matrice` A `u brzu memoriju`}
  for $j := 1$ **to** n **do**
    begin
    {`Učitati` $C[i, j]$ `u brzu memoriju`}
    {`Učitati stupac` j `matrice` B `u brzu memoriju`}
    for $k := 1$ **to** n **do**
      $C[i, j] := C[i, j] + A[i, k] * B[k, j]$;
    {`Napisati` $C[i, j]$ `u sporu memoriju`}
    end; {**for** j}
  end; {**for** i}

Najdublja, unutarnja petlja po k računa, zapravo, skalarni produkt i–tog retka matrice A s j–tim stupcem matrice B, odnosno grafički:

Oznake $A(i, :)$ i $A(i, 1 : n)$ su skraćene oznake za i–ti redak matrice A, a $B(:, j)$ i $B(1 : n, j)$ za j–ti stupac matrice B (vidjeti malo kasnije).

Primijetimo da unutarnje dvije petlje u prethodnom algoritmu (po j i k) možemo interpretirati kao BLAS 2 operaciju množenja vektora i matrice (i–ti redak matrice A pomnožen matricom B daje i–ti redak matrice C). To sugerira da ovaj algoritam neće biti ništa bolji od ovih BLAS 2 operacija, jer se te operacije nalaze u unutarnjoj petlji (preciznije – unutar vanjske petlje po i, pa je $A(i, :)$ već u brzoj memoriji i nema prostora za uštedu).

Nađimo broj poziva m spore memorije. Uočimo da algoritam zahtijeva:

- n^3 poziva zbog učitavanja svakog stupca matrice B točno n puta.
- n^2 poziva zbog učitavanja svakog retka matrice A jednom. Taj redak držimo u brzoj memoriji za vrijeme izvođenja dvije unutarnje petlje.
- $2n^2$ poziva za po jedno učitavanje svakog elementa matrice C u brzu memoriju i, jednako tako, njegovo pisanje natrag u sporu memoriju.

Zbrajanjem nalazimo da je ukupan broj poziva spore memorije jednak

$$m = n^3 + 3n^2 \quad .$$

Odavde odmah možemo izračunati q

$$q = \frac{f}{m} = \frac{2n^3}{n^3 + 3n^2} \lessapprox 2 \quad .$$

Uočimo da q dobivamo kao kod potprograma `sgemv`, što ne iznanađuje, obzirom na već danu BLAS 2 interpretaciju. Za veće n, taj q je mnogo manji od gornje ograde $n/2$, što sugerira da se algoritam može poboljšati.

Zadatak 5.2.1.

Pokažite da drugačiji načini učtavanja matrice B u brzu memoriju ne mogu bitno popraviti m i q.

Zadatak 5.2.2.

U prethodnom algoritmu poredak petlji je (i, j, k), od izvana, prema unutra (po blokovima). Te 3 petlje možemo permutirati, s tim da svaka od njih i dalje ide od 1 do n. Može li to popraviti algoritam – dati veći q? (Odgovor: Ne! — Opravdajte.)

5.2.2. Množenje matrica blokovima stupaca

Poboljšanje algoritma iz prethodnog odjeljka dobivamo, ako matricu B rastavimo u blokove stupaca. Za zapis algoritma trebamo dodatne oznake. Matricu C promatramo kao blok–matricu oblika

$$C = [\, C_1, C_2, \ldots, C_N \,]$$

sastavljenu od N blok–stupaca C_j, $j = 1, \ldots, N$. Svaki blok–stupac C_j sastoji se od točno n/N potpunih stupaca, tj. C_j je tipa $(n, n/N)$, uz pretpostavku da N dijeli n. Matricu B particioniramo na isti način. Za kratko označavanje podmatrica i blokova koristimo slijedeću, tzv. "Matlab" notaciju.

Oznakom $A(i:j, k:\ell)$ označavamo podmatricu matrice A koja se nalazi na presjeku između i–tog i j–tog retka (oba uključena) i k–tog i ℓ–tog stupca (oba uključena). Takva podmatrica je tipa $(j - i + 1, \ell - k + 1)$. Preciznije:

$$A(i:j, k:\ell) = \begin{bmatrix} a_{i,k} & a_{i,k+1} & \cdots & a_{i,\ell} \\ a_{i+1,k} & a_{i+1,k+1} & \cdots & a_{i+1,\ell} \\ \vdots & \vdots & & \vdots \\ a_{j,k} & a_{j,k+1} & \cdots & a_{j,\ell} \end{bmatrix} \quad .$$

Dodatno, raspon $i:i$ pišemo samo kao i, tj. oznakom

$$A(i, k:\ell) = \left[\, a_{i,k} \; a_{i,k+1} \; \cdots \; a_{i,\ell} \,\right]$$

označavamo i–ti redak matrice A između stupaca k i ℓ.

Puni raspon $1 : n$ pišemo samo kao $:$, tj. oznaka

$$A(:, j) = \begin{bmatrix} a_{1,j} \\ a_{2,j} \\ \vdots \\ a_{n,j} \end{bmatrix}$$

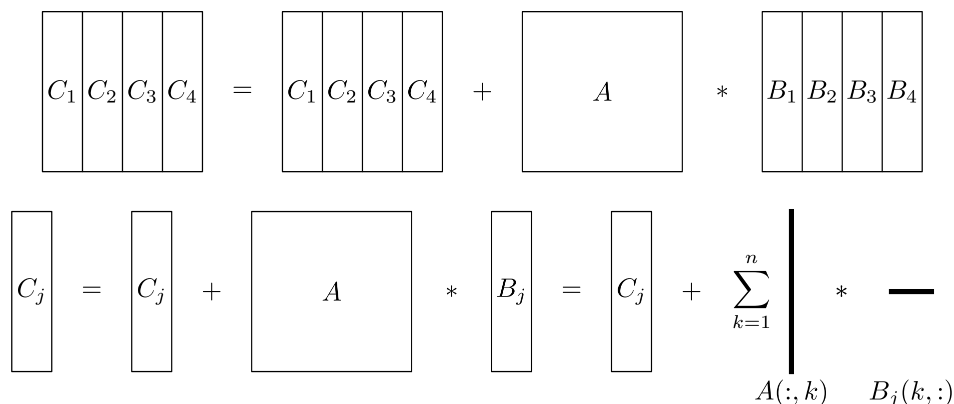
označava cijeli j–ti stupac matrice A.

Algoritam 5.2.2. (Množenje matrica s blokovima stupaca)

for $j := 1$ **to** N **do**
  begin
  {`Učitati blok-stupac` B_j `u brzu memoriju`}
  {`Učitati blok-stupac` C_j `u brzu memoriju`}
  for $k := 1$ **to** n **do**
    begin
    {`Učitati stupac` k `matrice` A `u brzu memoriju`}
    $C_j := C_j + A(:, k) * B_j(k, :)$;
      {`popravak (engl. update) matrice` C_j `matricom ranga` 1}
    end; {**for** k}
  {`Napisati` C_j `natrag u sporu memoriju`}
  end; {**for** j}

Unutarnja petlja po k više nije skalarni produkt, nego rang–1 popravak matrice C_j, matricom koja nastaje množenjem k–tog stupca matrice A s k–tim retkom matrice B_j.

Slikoviti prikaz procesa za $N = 4$ blok–stupca:



Pretpostavljamo da je brza memorija dovoljno velika da u isto vrijeme sadrži

blokove B_j, C_j i jedan stupac matrice A, tj.

$$M \geq 2 \cdot \frac{n^2}{N} + n \quad .$$

Odavde dobivamo gornju ogradu za veličinu bloka, odnosno donju ogradu na broj N blokova stupaca:

$$N \geq \frac{2n^2}{M-n} \sim \frac{2n^2}{M} \quad ,$$

uz pretpostavku da je M dosta veći od n, na primjer, $4n \leq M \ll n^2$.

Broj poziva spore memorije m uključuje:

- n^2 poziva zbog učitavanja svakog bloka B_j jednom (tj. cijeli B jednom).
- $N \cdot n^2$ poziva zbog učitavanja svakog stupca matrice A točno N puta.
- $2n^2$ poziva za po jedno učitavanje svakog elementa matrice C u brzu memoriju i, jednako tako, njegovo pisanje natrag u sporu memoriju.

Zbrajanjem nalazimo da je ukupan broj poziva spore memorije jednak

$$m = (N+3)n^2 \quad .$$

Odatle odmah slijedi (za malo veće N)

$$q = \frac{f}{m} = \frac{2n^3}{(N+3)n^2} = \frac{2n}{N+3} \sim \frac{2n}{N} \quad .$$

Algoritam možemo ubrzati tako da povećavamo q, tj. smanjujemo N (sve veći blokovi), dok je to moguće. Međutim, N je ograđen odozdo veličinom brze memorije

$$M \geq \frac{2n^2}{N} + n \quad .$$

Ako odaberemo najmanji mogući N, onda je

$$N \sim \frac{2n^2}{M} \quad ,$$

što daje

$$q \sim \frac{2n}{N} \sim \frac{M}{n} \quad .$$

Ako želimo dobiti razumno velik q (na primjer $q \geq 2$ ili više), onda M mora **rasti** (barem) proporcionalno s n. Osim toga, ta konstanta proporcionalnosti ne smije biti mala. Uz raniju pretpostavku $4n \leq M \ll n^2$, dobivamo $4 \leq q \ll n$, što je bitno lošije od gornje ograde $n/2$, a tek nešto bolje od prethodnog algoritma.

U praksi ni to, obično, nije moguće – najčešće je M fiksan. Za **fiksni** M, kad n raste, očekujemo da se algoritam **usporava** (jer pada q), što nije dobro svojstvo.

5.2.3. Množenje matrica kvadratnim blokovima

U ovom odjeljku matricu C promatramo kao blok–matricu (blok–)reda N, sastavljenu od $N \times N$ kvadratnih blokova C_{ij}, $i, j = 1, \ldots, N$, uz pretpostavku da N dijeli n. Svaki kvadratni blok C_{ij} je podmatrica od C reda n/N, tj. vrijedi

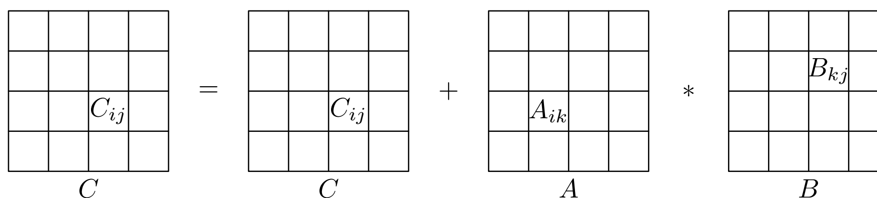
$$C = \begin{bmatrix} C_{11} & \cdots & C_{1N} \\ \vdots & & \vdots \\ C_{N1} & \cdots & C_{NN} \end{bmatrix} .$$

Matrice A i B particioniramo u blokove na isti način.

Algoritam 5.2.3. (Množenje matrica kvadratnim blokovima)

for $i := 1$ **to** N **do**
  for $j := 1$ **to** N **do**
    begin
    {`Učitati blok` C_{ij} `u brzu memoriju`}
    for $k := 1$ **to** N **do**
      begin
      {`Učitati blok` A_{ik} `u brzu memoriju`}
      {`Učitati blok` B_{kj} `u brzu memoriju`}
      $C_{ij} := C_{ij} + A_{ik} * B_{kj}$;
      end; {**for** k}
    {`Napisati blok` C_{ij} `natrag u sporu memoriju`}
    end; {**for** j, i}

Slikoviti prikaz procesa za blok–red $N = 4$ je:



$$\boxed{C_{ij}} \quad = \quad \boxed{C_{ij}} \quad + \quad \sum_{k=1}^{n} \boxed{A_{ik}} \quad * \quad \boxed{B_{kj}}$$

Unutarnja petlja – operacija $C_{ij} := C_{ij} + A_{ik} * B_{kj}$ je, zapravo, množenje matrica reda n/N (koje nismo posebno raspisali u pripadne 3 petlje – jer smatramo da je to, na primjer, osnovna BLAS 3 operacija).

Pretpostavljamo da je brza memorija dovoljno velika da sva 3 bloka C_{ij}, A_{ik} i B_{kj} odjednom stanu u brzu memoriju, tako da se ovo množenje matrica reda n/N

odvija isključivo u brzoj memoriji. To znači

$$M \geq 3\left(\frac{n}{N}\right)^2 \quad \text{ili} \quad N \geq \sqrt{\frac{3}{M}} \cdot n \quad ,$$

što je donja ograda za blok–red N.

Broj poziva spore memorije m uključuje:

- $N \cdot n^2$ poziva zbog učitavanja svakog bloka B_{kj} točno N^3 puta (tj. N puta cijeli B).
- $N \cdot n^2$ poziva zbog učitavanja svakog bloka A_{ik} točno N^3 puta (tj. N puta cijeli A).
- $2n^2$ poziva za po jedno učitavanje svakog elementa matrice C u brzu memoriju i, jednako tako, njegovo pisanje natrag u sporu memoriju.

Zbrajanjem nalazimo da je ukupan broj poziva spore memorije jednak

$$m = (2N + 2)n^2 \quad .$$

Dobivamo

$$q = \frac{f}{m} = \frac{2n^3}{2(N+1)n^2} = \frac{n}{N+1} \sim \frac{n}{N} \quad .$$

Posljednja približna jednakost vrijedi za N dosta veći od 1.

Ovo izgleda lošije od prethodnog algoritma (tamo je $q \sim 2n/N$), ali ovaj N nema isto značenje i ograničenja (ovaj N je približno korijen ranijeg). Kao i prije, algoritam ubrzavamo povećanjem q, tako da uzmemo najmanji mogući N. Iz uvjeta na M, to znači

$$N \sim \sqrt{\frac{3}{M}} \cdot n \quad \Rightarrow \quad q \sim \sqrt{\frac{M}{3}} \quad .$$

Ovaj q ne ovisi o n. Za konstantni M, dobivamo i konstantni q (što je bitno bolje od prethodnog algoritma).

Drugim riječima, zbog neovisnosti q o n, čini se da ovaj algoritam optimalno (do na konstantni faktor) koristi raspoloživu brzu memoriju.

To **nije** u kontradikciji s ranijom gornjom ogradom za q, oblika $q = n/2$. Naime, ako brza memorija M raste paralelno s n, na primjer, $M = 3n^2$, tako da sve 3 matrice čitave stanu u brzu memoriju, onda je $N = 1$ i tada je

$$q = \frac{n}{N+1} = \frac{n}{2}$$

(sada N nije bitno veći od 1), tj. optimalno.

Može se dokazati da je ovaj q ($\sim \sqrt{M/3}$) optimalan u smislu reda veličine (tj. do na faktor), jer vrijedi slijedeći teorem.

Teorem 5.2.1. (X. Hong i H. T. Kung, 1981.)

Za bilo koju blok–verziju ovog algoritma za množenje matrica vrijedi

$$q = \Omega(\sqrt{M}) \quad ,$$

tj. q raste barem kao konstantni višekratnik od $\sqrt{M}$. ■

Napomena 5.2.1.

Pod pojmom "ovog algoritma za množenje matrica" smatra se klasični algoritam s $2n^3$ operacija, tj. $f = 2n^3$. Sve naše blok–verzije nastale su iz istog osnovnog algoritma. Ako se uzme drugačiji, brži osnovni algoritam, ovaj teorem više nije primjenjiv.

Napomena 5.2.2.

Brži osnovni algoritam je, na primjer, Strassenov algoritam brzine

$$f = \mathcal{O}(n^{\log_2 7}) = \mathcal{O}(n^{2.81})$$

koji je rekurzivan i ima prirodnu blok–strukturu. U novije vrijeme, koristi se za velike n ($n \gtrsim 100$), na velikim računalima, za ubrzanje BLAS 3 operacija.

Postoje, asimptotski, još i brži algoritmi (rekord $f = \mathcal{O}(n^{2.376})$), ali su oni prespori (i prekomplicirani) za n–ove koji se danas mogu koristiti (obzirom na veličinu i brzinu računala).

5.3. Paralelno množenje matrica – razdijeljena memorija

Pokažimo kako se može implementirati matrično množenje – matrična operacija $C := C + A * B$, na paralelnim računalima s distribuiranom memorijom, uz nekoliko standardnih mreža komuniciranja.

Za predviđanje vremenskog trajanja, koristimo ranije modele komuniciranja.

Ponašanje ovih algoritama ovisi o nekoliko faktora:

1. Osnovni algoritam – koristimo standardni algoritam za množenje matrica koji zahtijeva $2n^3$ aritmetičkih operacija. Ako imamo p procesora, optimalni paralelni algoritam ima trajanje $2n^3/p$ vremenskih jedinica.

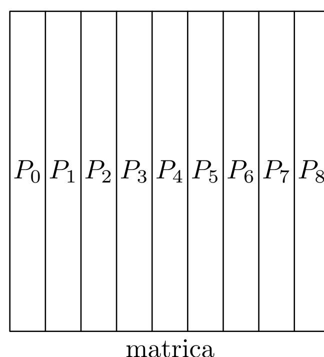
   Raspodjela (tzv. scheduling) ovih operacija po procesorima je zanimljiv dio konstrukcije algoritma (tj. koji procesori računaju koje dijelove produkta $A * B$ i kako se ti dijelovi akumuliraju na C).

2. Raspodjela podataka (data layout) – kako su A, B i C spremljeni po procesorima, koji dijelovi od A, B i C se pamte u kojim (lokalnim) memorijama procesora.

   Najčešće se koriste 2 osnovna rasporeda polaznih podataka (tj. stanja u početnom trenutku):

   (a) 1D blokovski (1D blocked) raspored – odgovara particiji matrice na blokove stupaca. Svaki procesor sadrži svoj blok stupaca. Pretpostavljamo da p dijeli n. Procesore indeksiramo linearno $P_0, \ldots, P_{p-1}$ – što je prirodno za 1D blokovski raspored;

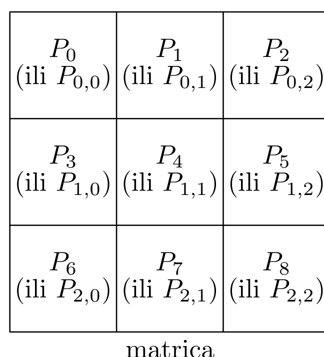
   Slika raspodjele blokova stupaca za $p = 9$:



matrica

   (b) 2D blokovski (2D blocked) raspored – odgovara particiji matrice na kvadratne blokove. Uz pretpostavku da je p potpun kvadrat, blokovi su reda $n/\sqrt{p}$ i svaki procesor pamti svoj blok. Procesore možemo indeksirati linearno indeksima $(0, \ldots, p-1)$ ili dvodimenzionalno, s dva indeksa, svaki indeks ide od 0 do $\sqrt{p} - 1$.

   Slika raspodjele blokova za $p = 9 = 3^2$:



matrica

3. Mreža povezivanja – procesori jedan drugom šalju pojedine blokove matrica.

   Pokazat će se da mreže veće povezanosti (valencije), poput dvodimenzionalnog polja, torusa i hiperkocke, omogućavaju brže algoritme od slabo povezanih mreža, poput prstena ili sabirnice.

   Ubrzanje ili uštedu na komunikaciji možemo postići na dva načina:

(a) mreže s većom bisekcijom dopuštaju da više nezavisnih podataka putuje mrežom u istom trenutku, bez kolizija.

(b) paralelizacijom komunikacije u svakom pojedinom procesoru.

Standardno, pojedini procesor P_i može, u danom trenutku, slati i/ili primati najviše jednu poruku (sekvencijalna komunikacija). Čak i ako dozvolimo istovremeno (sinkrono) slanje i primanje – najviše dvije veze (žice) iz tog procesora se mogu koristiti u svakom pojedinom trenutku, pa veća valencija nema odgovarajući utjecaj na brzinu.

Ako cijelu valenciju procesora (sve žice) možemo istovremeno koristiti za komuniciranje (paralelno komuniciranje) dobivamo bitno ubrzanje.

Modernije arhitekture to dopuštaju, ali na mrežama koje su mnogo slabije povezane od potpunog grafa (tj. valencija $\ll p$). Na primjer, u hiperkocki s p procesora, valencija je $\log_2 p$ (što je mali cijeli broj u današnjim računalima).

Napomena 5.3.1.

Geometrijski je prirodno koristiti $1D$ *blokovski raspored na mrežama koje imaju jednodimenzionalni oblik (sabirnica, prsten), a* $2D$ *blokovski na svim ostalim višedimenzionalnim topologijama. U prethodna dva odjeljka pokazali smo da* $2D$ *blokovski raspored osigurava brže algoritme. Analogiju s ranijom analizom dobivamo tako da smatramo da brža memorija odgovara lokalnoj memoriji, a spora memorija odgovara komuniciranju preko mreže.*

Drugim riječima, $1D$ *blokovski raspored nema smisla probati na višedimenzionalnim mrežama.*

5.3.1. 1D blokovski raspored

Radi jednostavnosti, pretpostavljamo da je n djeljiv s p. Označimo s $B(i)$ podmatricu tipa $n \times (n/p)$ matrice B, koja je na početku spremljena u procesoru P_i, za $i = 0, \ldots, p-1$. Podmatrice $A(i)$ i $C(i)$ definiramo na isti način.

Osim toga, neka je $B(j, i)$ podmatrica reda n/p od $B(i)$ (pa i od B), koja sadrži retke $j \cdot (n/p) + 1$ do $(j+1) \cdot (n/p)$ od $B(i)$, za $j = 0, \ldots, p-1$, tj. kao na

slici:

$$B(i) = \begin{array}{|c|} \hline B(0,i) \\ \hline B(1,i) \\ \hline \vdots \\ \hline B(p-1,i) \\ \hline \end{array}$$

Algoritam množenja dobivamo particijom matrice C na blokove stupaca:

$$C = [\, C(0), C(1), \ldots, C(p-1) \,] \quad ,$$

pa $C(i)$ računamo kao u algoritmu za množenje matrica blokovima stupaca

$$C(i) := C(i) + A * B(i) = C(i) + \sum_{j=0}^{p-1} A(j) * B(j,i) \quad .$$

(Okrugle zagrade koristimo za indeksiranje blokova.) Procesor P_i sadrži blokove $C(i)$ i $B(i)$, ali nema sve blokove $A(j)$ koje traži formula, već samo blok $A(i)$. Zbog toga, algoritam mora poslati svaki $A(j)$ svakom procesoru (tj. svaki $A(j)$ mora obići svaki procesor). Ovime smo implicitno pretpostavili da je lokalna memorija svakog procesora P_i dovoljna za pohranjivanje 4 bloka $A(i)$, $B(i)$, $C(i)$ i još jednog $A(j)$.

Ostatak algoritma ovisi o mreži – modelu komunikacije.

5.3.2. 1D blokovski raspored na sabirnici (bez emitiranja, s barijerom)

Uzmimo, za početak, najjednostavniji model mreže. Procesori se svi nalaze na jednoj žici — sabirnici (engl. bus). U bilo kom trenutku, najviše jedan procesor može slati i najviše jedan procesor može primati poruku. Pretpostavljamo da se slanje i primanje poruka obavlja istovremeno (sinkrono). U protivnom, vrijeme komuniciranja raste najviše za faktor 2.

Ovaj model odgovara vezi p računala povezanih jednim ethernet kablom.

U prvom trenutku, svi procesori računaju svoj komad produkta iz trenutno raspoloživih podataka, tj. svaki P_i računa svoj:

$$C(i) := C(i) + A(i) * B(i,i) \quad .$$

Nadalje, svaki procesor P_i mora poslati svoj $A(i)$ svim ostalim procesorima P_j, za $j \neq i$.

No, mrežom u danom trenutku može putovati najviše jedna poruka. Zbog toga, nakon svake poruke, radi sinkronizacije i izbjegavanja kolizija na mreži, postavljamo barijeru, što pišemo kao poziv procedure **barrier**( ).

Algoritam 5.3.1. (Množenje matrica: 1D raspored na sabirnici)

Algoritam bez emitiranja i s barijerama.

```
for i := 0 to p − 1 parallel do
  if MYPROC = i then
    C(MYPROC) := C(MYPROC) + A(MYPROC) ∗ B(MYPROC, MYPROC);
for i := 0 to p − 1 {parallel} do
  for j := 0 to p − 1 {parallel} do
    {u ovim dvjema petljama nema stvarnog paralelizma}
    if i <> j then
      begin
      if MYPROC = i then
        send (A(i), j)
      else if MYPROC = j then
        begin
        receive (A(i), i);
        C(MYPROC) := C(MYPROC) + A(i) ∗ B(i, MYPROC);
        end;   {else if}
      barrier ( );
      end;   {if, for j, i}
```

Napomena 5.3.2.

Prva petlja se katkad skraćeno piše samo kao:

$$C(\text{MYPROC}) := C(\text{MYPROC}) + A(\text{MYPROC}) * B(\text{MYPROC}, \text{MYPROC}) \quad ,$$

a podrazumijeva se da svaki procesor (istovremeno) izvršava svoj dio te naredbe, tj. da MYPROC *prolazi svim procesorima.*

Za analizu ovog algoritma koristimo slijedeće pretpostavke:

- svaka aritmetička (floating–point) operacija traje 1 vremensku jedinicu;

- priprema (inicijalizacija) poruke košta α vremenskih jedinica;
- slanje poruke traje β vremenskih jedinica po svakoj riječi (smatrat ćemo da je svaki podatak dugačak jednu riječ) – jedan podatak je, na primjer, jedan element matrice;
- slanje poruke od n riječi košta $\alpha + n \cdot \beta$ vremenskih jedinica;
- smatramo da je slanje i primanje poruke sinkrono.

Nađimo trajanje aritmetike (računanja) i komuniciranja u prethodnom algoritmu. Označimo s T_a ukupno trajanje aritmetičkih operacija, a s T_c ukupno trajanje komuniciranja.

Promotrimo unutarnju petlju (za fiksne i i j) algoritma. Neka je t_a trajanje aritmetičkih operacija u toj petlji, a t_c trajanje komuniciranja:

- aritmetika:
$$t_a = 2n(n/p)^2 = \frac{2n^3}{p^2} \quad ,$$
jer množimo blok $A(i)$, tipa $n \times (n/p)$, s (malim) blokom $B(i, \text{MYPROC})$, tipa $(n/p) \times (n/p)$. Isto trajanje ima i prva petlja (prije slanja $A(i)$).
- komunikacija:
$$t_c = \alpha + n(n/p)\beta \quad ,$$
jer blok $A(i)$ sadrži $n(n/p)$ riječi (podataka), a slanje i primanje je sinkrono.

Operacije u unutarnjoj petlji se izvode $p(p-1)$ puta (zbog barijere koja onemogućava paralelno računanje i komuniciranje). Ukupno trajanje aritmetike uvećava se još za trajanje prve petlje, prije slanja $A(i)$, i ono je jednako

$$T_a = (p(p-1)+1) \cdot t_a = (p(p-1)+1)\,\frac{2n^3}{p^2} \quad .$$

Ukupno vrijeme komuniciranja je

$$T_c = p(p-1) \cdot t_c = p(p-1)\,(\alpha + n(n/p)\beta) \quad ,$$

pa je ukupno vrijeme ovog algoritma

$$T(n,p) = T_a + T_c = (p(p-1)+1)\,\frac{2n^3}{p^2} + p(p-1)\,(\alpha + n(n/p)\beta) \quad .$$

Ako uzmemo samo članove s najvišim potencijama od p, a niže potencije zanemarimo, izlazi
$$T(n,p) \sim 2n^3 + p^2\alpha + pn^2\beta \quad ,$$
što je više od trajanja sekvencijalnog algoritma ($2n^3$) i **raste** s p, umjesto da pada, s povećanjem p.

Ovo je prilično loš paralelan algoritam. U stvari, on uopće nije paralelan, jer, zbog barijere, najviše jedan procesor može računati u svakom trenutku.

Rezultat, međutim, nije nerealan. Zamislimo da je n toliko velik da trebamo p procesora samo za spremanje matrice (nema velike zajedničke memorije), a svaki procesor ima ograničenu lokalnu memoriju (na primjer, konstantnu). Povećanje n onda povećava i p, a komunikacija, očito, raste s p (i nema paralelizma!).

5.3.3. 1D blokovski raspored na sabirnici (bez emitiranja i barijere)

Očekujemo da ćemo dobiti bolji rezultat ako uklonimo barijeru, tako da se računanje i komuniciranje mogu vremenski preklapati i tako da $A(i)$ možemo (ne nužno istovremeno) poslati većem broju procesora, pa oni mogu paralelno računati.

Algoritam 5.3.2. (Množenje matrica: 1D raspored na sabirnici)

Algoritam bez emitiranja i bez barijera.

```
for j := 0 to p − 1 parallel do
  if MYPROC = j then
    begin
    C(MYPROC) := C(MYPROC) + A(MYPROC) ∗ B(MYPROC, MYPROC);
      {Primi A(i) od prethodnih P_i}
      {petlja je sekvencijalna za fiksni MYPROC}
    for i := 0 to MYPROC − 1 {parallel} do
      begin
      receive (A(i), i);
      C(MYPROC) := C(MYPROC) + A(i) ∗ B(i, MYPROC);
      end;   {for i}
      {Pošalji A(MYPROC) svim preostalim P_i}
      {petlja je sekvencijalna za fiksni MYPROC}
    for i := 0 to p − 1 {parallel} do
      if MYPROC <> i then
        send (A(MYPROC), i);
      {Primi A(i) od slijedećih P_i}
      {petlja je sekvencijalna za fiksni MYPROC}
    for i := MYPROC + 1 to p − 1 {parallel} do
      begin
      receive (A(i), i);
      C(MYPROC) := C(MYPROC) + A(i) ∗ B(i, MYPROC);
```

end; {**for** i}
end; {**if**, **for** j}

Napomena 5.3.3.

Kao i prije, istovremeno komuniciraju samo 2 *procesora (sinkroni* **send** *i* **receive***), tj. u danom trenutku, na sabirnici može biti najviše* 1 *poruka i nju šalje jedan procesor (naravno!) i prima samo jedan procesor.*

Poredak slanja i primanja može se i drugačije zadati. Ovaj zapis je najlakši za čitanje, jer ranije poruke idu od procesora manjeg indeksa u procesor većeg indeksa.

VAŽNO: *Ovaj zapis* **ne zadaje** *u potpunosti vremenski redoslijed poruka (tko komu kada šalje)!*

Prethodni program je, dakle, **nedeterministički**:

- redoslijed slanja i primanja poruka ne mora biti isti kao u algoritmu s barijerama;
- to ne mijenja rezultat algoritma (tj. "utrkivanje" procesora nema utjecaja), jer dobivamo isti rezultat, neovisno o redoslijedu poruka. Bitno je samo da svaki procesor P_i pošalje svoj $A(i)$ svim preostalim P_j i primi od njih sve njihove $A(j)$ i izračuna pripadni dio produkta.

Pokažimo primjerom da i u našem zapisu ima više mogućih redoslijeda slanja i primanja poruka. Neka je $p = 4$. Tada imamo točno $12 = 4 \cdot 3$ poruka (P_i šalje, P_j prima, $j \neq i$, uz $i, j \in \{0, 1, 2, 3\}$).

Za svaki procesor, napravit ćemo (vremenski) redoslijed njegovih poziva procedura **send** i **receive** (jer on radi sekvencijalno – pa vremenski redoslijed ima smisla).

Pojedini procesori su stupci na slici, a redovi su pripadni pozivi. U svakom stupcu (za pripadni procesor) koristimo slijedeće oznake:

S_i – (**send**) pošalji procesoru P_i;

R_i – (**receive**) primi od procesora P_i.

U zagradi iza te oznake pišemo redni broj te poruke na sabirnici (od 1 do 12), onako kako to diktira naš zapis algoritma — svaki procesor mora prvo primiti podatke od prethodnih, pa poslati preostalima, pa primiti od onih s indeksom većim od njegovog.

Na primjer, "S2(5)" u stupcu procesora P_1 i "R1(5)" u stupcu procesora P_2, znači da je 5. poruka na sabirnici (vremenski gledano), ona poruka koju procesor P_1 šalje procesoru P_2.

redoslijed poziva	Procesor			
	P_0	P_1	P_2	P_3
1.	S1(1)	R0(1)	R0(2)	R0(3)
2.	S2(2)	S0(4)	R1(5)	R1(6 ili 7)
3.	S3(3)	S2(5)	S0(6 ili 7)	R2(9)
4.	R1(4)	S3(6 ili 7)	S1(8)	S0(10)
5.	R2(6 ili 7)	R2(8)	S3(9)	S1(11)
6.	R3(10)	R3(11)	R3(12)	S2(12)

Uočite da dvije poruke – poruka procesora P_1 za P_3 i poruka P_0 za P_2 mogu biti u bilo kom međusobnom redoslijedu – bilo koja od njih može biti šesta ili sedma.

Precizno, ovaj fenomen opisujemo ovako: Parcijalni uređaji događaja komuniciranja, određeni s p programa na pripadnim procesorima, ne čine (ne zadaju) potpuni uređaj događaja komuniciranja (na sabirnici).

Zadatak 5.3.1.

Pokažite da je za $p = 3$, redoslijed poruka jednoznačno određen (tj. nedeterminizam se prvi puta javlja za $p = 4$). Što se događa za $p = 5$?

Primijetimo da koraci računanja i komuniciranja u unutarnjoj petlji algoritma traju kao i ranije, tj.

$$t_a = \frac{2n^3}{p^2} \quad , \quad t_c = \alpha + n(n/p)\beta \quad .$$

Što treba očekivati za ponašanje algoritma? Ako je cijena (trajanje) komuniciranja dovoljno manja od cijene (trajanja) računanja, onda bi efikasnost trebala biti velika.

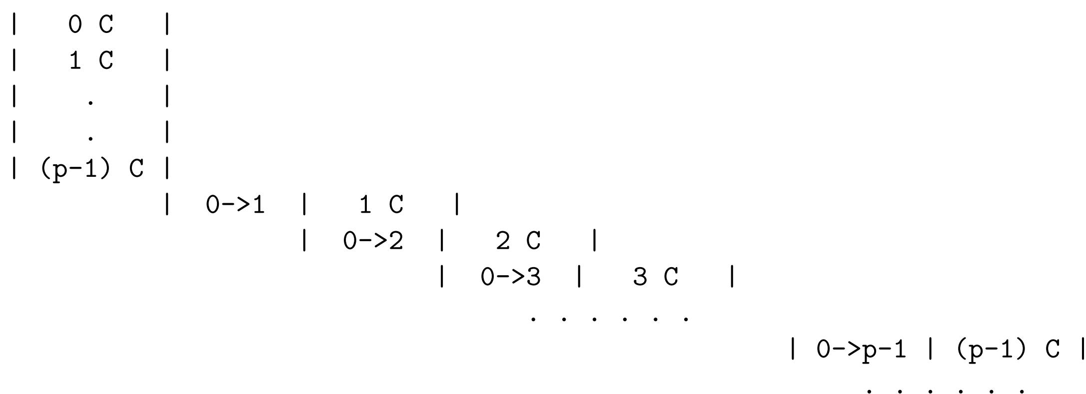
Obratno, ako je trajanje komuniciranja usporedivo s trajanjem računanja, ili ako dominira, očekujemo malu efikasnost.

Izračunajmo ukupno vremensko trajanje algoritma. Prvo nacrtajmo vremensko ponašanje algoritma, tj. nacrtajmo pojedine operacije na vremensku os, uz pretpostavku da je $t_c \leq t_a$ (suprotni slučaj analiziramo malo kasnije).

Uvedimo slijedeće oznake na vremenskoj osi:

- `i -> j` — poruka koju P_i šalje procesoru P_j. Njeno trajanje je t_c.
- `i C` — procesor P_i računa. Trajanje računa je t_a.

Radi jednostavnosti, pretpostavljamo da se komuniciranje odvija istim redoslijedom kao u algoritmu s barijerama (iako to nije bitno za ukupno vrijeme – samo za skicu).



Uočite da pretpostavka $t_c \leq t_a$ znači da u dijelu vremena imamo paralelno računanje (na primjer, `1 C` i `2 C` se preklapaju u dijelu vremena).

Očito se čitav postupak odvija u obliku protoka kroz cijev (pipeline). Treba izračunati (vremensku) duljinu cijevi, tj. njeno trajanje. Za to treba pogledati da li ima "mjehurića" u cijevi, tj. da li komunikacioni koraci mogu započeti bez prethodnog zastoja (jer odgovarajući procesor još radi prethodni račun).

Što bi se odvijalo dalje, iza nacrtanog dijela algoritma? Prva slijedeća komunikacija (iza `0->p-1`), koja nije nacrtana, je `1->0`. Poruka `1->0` može početi, nakon što `0->p-1` završi, bez čekanja, samo ako je procesor P_1 nezaposlen — tj. ako je završio računanje `1 C`, nakon poruke `0->1`.

To se događa u slučaju da je

$$t_a \leq (p-2) \cdot t_c \quad , \quad p \geq 2 \quad .$$

Uz tu pretpostavku, nema "mjehurića" u cijevi, pa je ukupno vrijeme

$$T(n,p) = p(p-1) \cdot t_c + 2 \cdot t_a \quad , \tag{5.3.1}$$

tj. jednako je trajanju svih $p(p-1)$ poruka, plus trajanje računanja prije slanja prve i nakon slanja zadnje poruke. Naravno, ista ova relacija vrijedi i za $t_c \geq t_a$.

Ako nema mjehurića, dobivamo gornju ogradu za vrijeme

$$T(n,p) \leq p(p-1) \cdot t_c + 2(p-2) \cdot t_c = (p^2 + p - 4) \cdot t_c \quad .$$

Općenitu donju ogradu za $T(n,p)$ možemo dobiti (neovisno o mjehurićima) iz činjenice da svaki procesor mora izračunati svoj dio matrice C, za što mu treba svih

p dijelova matrice A i još mora poslati svoj dio od A svim ostalim procesorima ($p-1$ poruka) i od svakog dobiti njegov dio matrice A (još $p-1$ poruka). Dobivamo:

$$T(n,p) \geq p \cdot t_a + 2(p-1) \cdot t_c \quad .$$

Ako nema mjehurića, tj. ako je t_c odozdo ograđen s

$$t_c \geq \frac{t_a}{p-2}$$

imamo

$$T(n,p) \geq p \cdot t_a + \frac{2(p-1)}{p-2} \cdot t_a > (p+2) \cdot t_a > p \cdot t_a \quad .$$

Uz odabrani standardni algoritam množenja matrica, najbolji, (ujedno i uobičajeni) sekvencijalni algoritam ima trajanje $2n^3$, pa je najbolje moguće vrijeme paralelnog algoritma $2n^3/p$. No, ovdje je $t_a = 2n^3/p^2$, pa je:

$$\text{idealni } T(n,p) = \frac{2n^3}{p} = p \cdot t_a < (p+2) \cdot t_a < T(n,p)$$

i vidimo da se idealno ubrzanje skoro dostiže, ako je t_c na donjoj granici

$$t_c \sim \frac{t_a}{p-2} \quad ,$$

pa je

$$T(n,p) \gtrapprox \frac{p+2}{p} \cdot \text{idealni } T(n,p) \quad .$$

Ako je broj procesora p dovoljno velik, a komunikacija dovoljno brza i nema mjehurića, dobivamo skoro idealno ubrzanje. Tada činjenica da je sabirnica serijsko usko grlo nema bitni utjecaj.

Ako je komunikacija još brža (što nije realistično!), tj. ako vrijedi

$$t_c < \frac{t_a}{p-2} \quad ,$$

onda imamo rupe – mjehuriće na vremenskoj osi, pa više ne vrijedi polazna relacija (5.3.1) za $T(n,p)$.

Zadatak 5.3.2.

Nađite u tom slučaju točnu relaciju za $T(n,p)$ i pokažite da se ubrzanje malo povećava, ali **ne bitno**.

Za praksu je puno zanimljivije promatrati slučaj kad t_c raste obzirom na t_a. Dakle, uzmimo opet da je

$$t_c \geq \frac{t_a}{p-2}$$

i promatrajmo što se događa kad t_c raste. Očito, ubrzanje se smanjuje!

U trenutku kad je $t_c = t_a$, imamo

$$T(n,p) = p(p-1) \cdot t_c + 2 \cdot t_a = (p^2 - p + 2) \cdot t_a = \frac{p^2 - p + 2}{p^2} \cdot 2n^3 \approx 2n^3$$

(ako uzmemo samo vodeći član brojnika), tj. trajanje je približno jednako sekvencijalnom algoritmu – paralelizam ništa ne pomaže.

Ako je $t_c > t_a$, algoritam je sporiji od sekvencijalnog, što je očito iz vremenskog grafa, jer nema paralelnog računanja, a samo se **dio** komunikacija izvodi paralelno s računanjem.

Nađimo još i efikasnost algoritma, uz pretpostavku $t_c \geq t_a/(p-2)$. Iskoristimo li da je $2n^3 = p^2 \cdot t_a$, dobivamo

$$\begin{aligned} E(p) &= \frac{S(p)}{p} = \frac{T(1,p)}{p \cdot T(n,p)} = \frac{2n^3}{p \cdot T(n,p)} = \frac{p \cdot t_a}{T(n,p)} \\ &= \frac{1}{\dfrac{T(n,p)}{p \cdot t_a}} = \frac{1}{\dfrac{2}{p} + (p-1)\,\dfrac{t_c}{t_a}} \\ &= \frac{1}{\dfrac{2}{p} + \dfrac{p^3 - p^2}{2n^3} \cdot \alpha + \dfrac{p^2 - p}{2n} \cdot \beta} \quad . \end{aligned}$$

Ako je omjer t_c/t_a blizu $1/(p-2)$, efikasnost je blizu 1. Za t_c/t_a blizu 1, efikasnost je skoro $1/p$, tj. nema paralelnog ubrzanja.

Gledano u funkciji n i p, vidimo da je t_c/t_a mali, ako je $n \gg p$, i ako α i β nisu jako veliki.

Ako uzmemo fiksni p, a n raste, onda t_a raste kao n^3, a t_c kao n^2, pa t_a dominira za velike n (tj. t_c/t_a je mali), što daje visoku efikasnost. To nije jako realno! Ako p raste linearno s porastom n (tj. lokalna je memorija konstantna), onda t_a raste linearno, kao i t_c, pa je omjer približno konstantan, tj. efikasnost je približno konstantna, što je bolje od prošlog algoritma.

5.3.4. 1D blokovski raspored na sabirnici (s emitiranjem)

Osnovna prepreka efikasnosti ranija dva algoritma je stroga sekvencijalnost poruka i ograničenje da jednu poruku prima samo jedan procesor.

Pretpostavimo stoga da jednu poruku na mreži može primiti više procesora – u ovom slučaju, svi preostali na sabirnici.

Takvo slanje poruke, u kojem jedan procesor šalje poruku svim preostalim procesorima, zove se emitiranje (engl. broadcast). U algoritmu, emitiranje zapisujemo pozivom procedure **broadcast**, u obliku **broadcast** (*podatak*).

Algoritam 5.3.3. (Množenje matrica: 1D raspored na sabirnici)

Algoritam s emitiranjem i bez barijera.

for $i := 0$ **to** $p-1$ {**parallel**} **do** {`ovo ide sekvencijalno`}
  for $j := 0$ **to** $p-1$ **parallel do**
    if MYPROC $= j$ **then**
      begin
      if MYPROC $= i$ **then** {P_i `šalje` $A(i)$ `svim preostalim`}
        broadcast $(A(i))$
      else {`svi preostali sinkrono primaju` $A(i)$}
        receive $(A(i), i)$;
      $C(\text{MYPROC}) := C(\text{MYPROC}) + A(i) * B(i, \text{MYPROC})$;
      end; {**for** j, i}

Uočite da prve faze ranijih algoritama nema, ona se obavlja u unutarnjoj paralelnoj petlji. Procesor P_i računa sa svojim (početnim) $A(i)$, nakon što ga pošalje ostalima, tj. u isto vrijeme kad i svi ostali procesori računaju s tim istim $A(i)$.

Uz isti model komunikacije, ukupno vrijeme trajanja ovog algoritma je

$$T(n,p) = p \cdot (t_c + t_a) \quad ,$$

jer u svakom vremenskom koraku (po i), P_i pošalje $A(i)$ svima ostalima (samo jedna komunikacija u trajanju t_c), a zatim svih p procesora paralelno računa svoj dio matrice C – dodajući onaj dio koji ovisi o $A(i)$.

Uvrstimo t_a i t_c, koji su isti kao ranije, u formulu ukupnog trajanja:

$$T(n,p) = p \cdot (t_c + t_a) = p \cdot \left(\alpha + \frac{n^2}{p} \cdot \beta + \frac{2n^3}{p^2}\right) = \frac{2n^3}{p} + p \cdot \alpha + n^2 \cdot \beta \quad .$$

Pripadna efikasnost je

$$E(p) = \frac{2n^3}{p \cdot T(n,p)} = \frac{1}{1 + \dfrac{t_c}{t_a}} = \frac{1}{1 + \dfrac{p^2}{2n^3} \cdot \alpha + \dfrac{p}{2n} \cdot \beta} \quad .$$

U usporedbi s algoritmom na sabirnici bez emitiranja, član t_c/t_a u nazivniku efikasnosti je $p-1$ puta manji, tj. efikasnost je mnogo manje osjetljiva funkcija od p.

Slično kao i prije, zbog toga što očekujemo $\alpha \gg 1$ i $\beta \gg 1$, mora biti $n \gg p$ za efikasni paralelizam. Međutim, donja ograda za n/p za postizanje efikasnog paralelizma je ovdje mnogo manja nego na sabirnici bez emitiranja.

Ovdje je i komunikacija paralelizirana, a ne samo računanje, što je, očito, bolje.

5.3.5. 1D blokovski raspored na prstenu procesora

Slično ponašanje možemo postići i na prstenu od p procesora (bez upotrebe emitiranja), jer istovremeno imamo opet p poruka u mreži. Potrebni komadi matrice A kružno putuju (paralelno) uokolo prstena.

Radi jednostavnosti, opet pretpostavljamo sinkrono slanje i primanje poruka. U najgorem slučaju, dobit ćemo da komunikacija traje za faktor 2 dulje od sinkrone. To dobivamo tako da, umjesto istovremenog slanja i primanja, prvo neparni procesori šalju parnima, a zatim parni neparnima. Ovo je korektno, ako prsten ima paran broj procesora, jer susjedi u prstenu onda imaju različitu parnost. Ako imamo neparan broj procesora u prstenu, razmislite o tome i riješite kao zadatak.

Algoritam 5.3.4. (Množenje matrica: 1D raspored na prstenu)

for $j := 0$ **to** $p-1$ **parallel do**
  if MYPROC $= j$ **then**
    begin
      {`kopiranje svog komada matrice` A `u lokalni pomoćni`}
      {`prostor` T `koji služi za komunikaciju`}
    $T := A(\text{MYPROC});$
    $C(\text{MYPROC}) := C(\text{MYPROC}) + T * B(\text{MYPROC}, \text{MYPROC});$
    end; {**for** j}
for $i := 1$ **to** $p-1$ {**parallel**} **do** {`ovo ide sekvencijalno`}
  for $j := 0$ **to** $p-1$ **parallel do**
    if MYPROC $= j$ **then**
      begin
      send $(T, (\text{MYPROC} + 1)$ **mod** $p)$; {`slanje desnom`}
      receive $(T, (\text{MYPROC} - 1)$ **mod** $p)$; {`primanje od lijevog`}
        {`lokalni` T `sadrži polazni` $A((\text{MYPROC} - i)$ **mod** $p)$}
      $C(\text{MYPROC}) := C(\text{MYPROC}) + T * B((\text{MYPROC} - i)$ **mod** $p, \text{MYPROC});$
      end; {**if**, **for** j, i}

Uz isti model komuniciranja, t_a i t_c u unutarnjoj petlji su isti kao ranije. Svaki procesor računa p puta, a komunicira $p-1$ puta (jedna komunikacija između svaka

dva računanja), pa je

$$\begin{aligned} T(n,p) &= p \cdot t_a + (p-1) \cdot t_c = p \cdot \frac{2n^3}{p^2} + (p-1) \cdot \left(\alpha + \frac{n^2}{p} \cdot \beta \right) \\ &= \frac{2n^3}{p} + (p-1) \cdot \alpha + \frac{p-1}{p} \cdot n^2 \beta \quad , \end{aligned}$$

što je malo bolje od prethodnog algoritma s emitiranjem. Pripadna efikasnost je:

$$E(p) = \frac{2n^3}{p \cdot T(n,p)} = \frac{1}{1 + \frac{p-1}{p} \cdot \frac{t_c}{t_a}} = \frac{1}{1 + \frac{p(p-1)}{2n^3} \cdot \alpha + \frac{p-1}{2n} \cdot \beta} \quad .$$

Ponovno, točno za jedan t_c bolje od algoritma na sabirnici s emitiranjem.

Napomena 5.3.4.

Posljednja dva algoritma su optimalna, što se reda veličine tiče, tj. do na konstantni faktor.

Dokaz:

Osnovni algoritam za množenje matrica, uz 1D blokovski raspored po procesorima, daje raniju formulu

$$C(j) = C(j) + \sum_{i=0}^{p-1} A(i) * B(i,j) \quad .$$

Svaki sumand (tj. produkt $A(i)*B(i,j)$) se akumulira — zbraja na $C(j)$ u procesoru P_j. To je posljedica 1D blokovskog rasporeda, u kojem je blok $C(j)$ rezultata C spremljen u procesoru P_j, tj. nije bitno gdje se taj sumand računa! To znači da se neki blok podataka veličine $n \cdot (n/p)$ (na pr., faktor $A(i)$ ili čitav sumand) miče sa svakog procesora P_i na svaki procesor P_j, a to zahtijeva barem $p-1$ poruka veličine n^2/p.

Uočimo da je broj sumanada $p(p-1)$ i svaki procesor ih treba p komada, od kojih je najviše jedan njegov — zbog početnog rasporeda matrica. Dakle, mora ih primiti (barem) $p-1$, što je jednako broju poruka. Dodatni faktor p se gubi paralelizacijom komuniciranja (p istovremenih poruka u mreži). Kod emitiranja, faktor je $p-1$ u paralelizaciji, što je istog reda veličine kao p.

Dakle, za 1D blokovski raspored na sabirnici ili prstenu, to je donja ograda za paralelno vrijeme

$$T(n,p) \geq p \cdot t_a + (p-1) \cdot t_c \quad .$$

Na prstenu, ona je dostižna, a na sabirnici se skoro dostiže, ako koristimo emitiranje.

■

Za prsten, optimalnost se još lakše vidi direktno iz algoritma. Svi procesori su točno jednako opterećeni i rade non–stop — računaju ili komuniciraju (a to dvoje ne ide simultano u istom procesoru). Kad komuniciraju, sve veze među procesorima su potpuno iskorištene, pa nema prostora za uštedu.

Napomena 5.3.5.

Prsten ($1D$ torus) može se uložiti u bilo koji višedimenzionalni torus i hiperkocku (s dovoljnim brojem procesora), tj. uz dovoljnu duljinu stranice torusa, odnosno dovoljnu dimenziju hiperkocke.

Zbog toga, prethodni algoritam radi i na takvim mrežama. Međutim, postoje i mnogo bolji algoritmi, koje opisujemo u nastavku.

5.3.6. Cannonov algoritam na 2D torusu

Prvo ćemo opisati Cannonov algoritam za množenje matrica uz 2D blokovski raspored, bez pozivanja na mrežu, a kasnije dodajemo i prirodni oblik mreže (u ovom slučaju, 2D torus).

Pretpostavljamo da je broj procesora p potpun kvadrat, tj.

$$p = s^2$$

i da je n djeljiv sa $s = \sqrt{p}$. Procesore označavamo prirodno, s dva indeksa, $P_{i,j}$ za $i, j \in \{0, 1, \ldots, s - 1\}$.

Analogno, $B(i, j)$ označava kvadratni blok — podmatricu od B, reda n/s, koja je na početku spremljena u procesoru $P(i, j) = P_{i,j}$. Isto vrijedi za $A(i, j)$ i $C(i, j)$.

Napomena 5.3.6.

Ovaj $B(i, j)$ je različit od onog $B(i, j)$ iz $1D$ blokovskih algoritama. Ovaj ima red $n/s = n/\sqrt{p}$, a raniji je bio reda n/p.

Kao osnovni algoritam za množenje matrica uzimamo standardni algoritam s $2n^3$ operacija, samo pisan u 2D blokovskoj formi. Svaki blok $C(i, j)$ se računa kao

$$C(i, j) = C(i, j) + \sum_{k=0}^{s-1} A(i, k) * B(k, j) \quad , \quad \text{za } i, j = 0, \ldots, s - 1 \quad .$$

Cannonov algoritam preuređuje redoslijed sumacije u ovoj sumi (unutarnjoj petlji algoritma) u oblik:

$$C(i, j) = C(i, j) + \sum_{k=0}^{s-1} A(i, (i + j + k) \textbf{ mod } s) * B((i + j + k) \textbf{ mod } s, j) \quad .$$

Vrijednosti $(i + j + k)$ **mod** s prolaze svim vrijednostima od 0 do $s - 1$, ali ne tim redoslijedom!

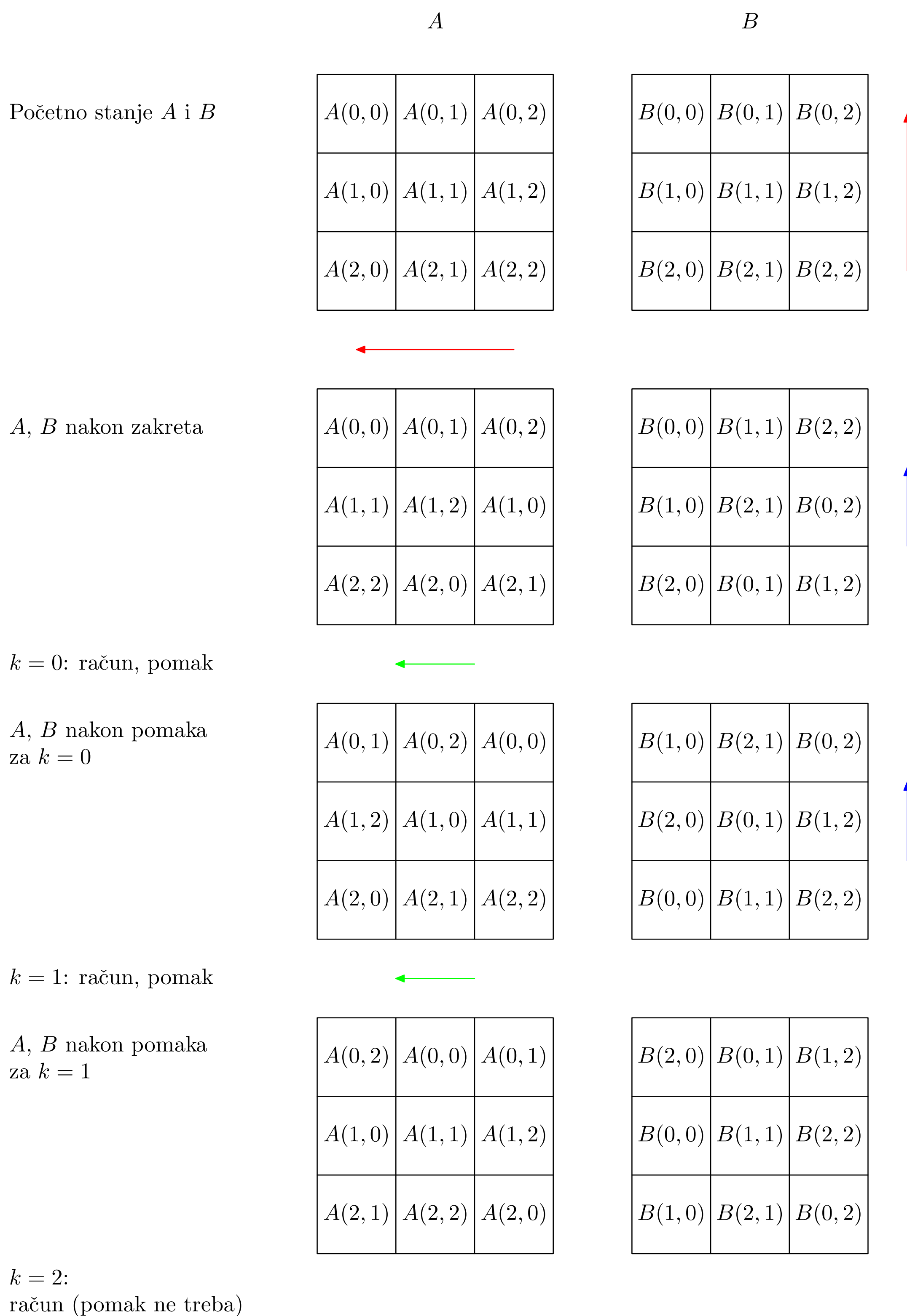
Algoritam 5.3.5. (Množenje matrica: Cannonov algoritam)

```
{''Zaokretanje'' ili ''ukošavanje'' matrice A (engl. ''skew'')}
for i := 0 to s − 1 parallel do
  kružno ulijevo rotiranje blok-retka i od A za i mjesta,
    tako da A(i, j) bude prepisan s A(i, (j + i) mod s);
{''Zaokretanje'' ili ''ukošavanje'' matrice B}
for j := 0 to s − 1 parallel do
  kružno nagore rotiranje blok-stupca j od B za j mjesta,
    tako da B(i, j) bude prepisan s B((i + j) mod s, j);
{Množenje}
for k := 0 to s − 1 {parallel} do    {ovo ide sekvencijalno}
  for i := 0 to s − 1 parallel do
    for j := 0 to s − 1 parallel do
      begin
      C(i, j) := C(i, j) + A(i, j) ∗ B(i, j);
        {A(i, j) i B(i, j) su trenutne vrijednosti ovih blokova u P(i, j)}
      kružno ulijevo rotiranje svakog blok-retka od A za 1,
        tako da se A(i, j) prepisuje s A(i, (j + 1) mod s);
      kružno nagore rotiranje svakog blok-stupca od B za 1,
        tako da se B(i, j) prepisuje s B((i + 1) mod s, j);
      end;    {for (i, j), k}
```

Zadnje rotacije za vrijednost $k = s - 1$ ne treba raditi. Iz ovog algoritma, iako se ne spominju procesori $P(i, j)$, očito je koji procesor obavlja koju računsku operaciju i tko kome šalje kakve poruke.

Algoritam je najlakše predočiti slikom koja pokazuje stanje lokalnih blokova $A(i, j)$, $B(i, j)$ nakon pojedinih koraka algoritma. U odgovarajuće polje $A(i, j)$, odnosno $B(i, j)$, upisujemo koji blok **polazne** matrice se, u tom trenutku, nalazi u pripadnom lokalnom bloku u procesoru $P(i, j)$.

Uzmimo $p = 9$ procesora, tj. blok–red je $s = 3$.



Nakon početnog "zakreta" i svakog daljnjeg pomaka, u procesor $P(i,j)$ stižu upravo blokovi oblika $A(i,k)$ i $B(k,j)$ iz polazne matrice. Redoslijed k–ova daje različite permutacije na raznim procesorima, a zadan je s $(i+j+k)$ **mod** s.

Na primjer, pogledajmo kako se formira rezultat $C(1,2)$ u procesoru $P(1,2)$. Nakon početnog zakreta, u $P(1,2)$ se nalaze blokovi $A(1,0)$ i $B(0,2)$, koje množimo i dodajemo na početni $C(1,2)$. Nakon prvog pomaka ($k=1$), stižu blokovi $A(1,1)$ i $B(1,2)$, koje množimo i akumuliramo na $C(1,2)$. Na kraju, nakon pomaka za $k=2$, stižu blokovi $A(1,2)$ i $B(2,2)$, koje množimo i akumuliramo.

Ovaj algoritam dobro odgovara 2D torusu s $p = s \times s$ procesora – blok–stupci i blok–reci mogu nezavisno putovati po stupcima, odnosno recima mreže.

Trajanje pojedinih dijelova algoritma je:

(1) "Zakret" A: svaki blok–redak matrice A putuje nezavisno od ostalih – tj. paralelno sa svim ostalima. Zbog kružne veze u retku, poruku šaljemo kraćim putem (lijevo ili desno). Uz pretpostavku da svaki procesor simultano šalje i prima, trajanje je najviše

$$T_1 = \frac{s}{2} \cdot \left(\alpha + \left(\frac{n}{s}\right)^2 \cdot \beta \right) = \frac{\sqrt{p}}{2} \cdot \alpha + \frac{n^2}{2\sqrt{p}} \cdot \beta \quad ,$$

jer imamo najviše $s/2$ poruka (najveća udaljenost bilo koja 2 procesora), a svaka poruka je jedan blok, tj. duljine $(n/s)^2$.

U nekim recima (na primjer, početnim), zakret traje i mnogo kraće, jer elemente treba maknuti za manje mjesta ulijevo (pripadni procesori su mnogo bliže). Međutim, T_1 se dostiže za $i \approx \lfloor s/2 \rfloor$.

(2) "Zakret" B: trajanje je potpuno isto kao i trajanje zakreta A (samo se zamijene uloge redaka i stupaca). Ova operacija ne ide paralelno sa zakretom od A, jer isti procesori ne mogu slati/primati više od jedne poruke. (Ako procesor može paralelno komunicirati i po retku i po stupcu, onda se zakret od B može odvijati paralelno sa zakretom od A).

Dakle: $T_2 = T_1$.

(3) Pomak A ulijevo za jedno mjesto, odnosno pomak B nagore za jedno mjesto: obavlja se paralelno u svakom retku (a zatim u svakom stupcu). Budući da prvo putuje jedna poruka (zapravo, puno njih sinkrono) za A, a zatim jedna za B, svaka duljine $(n/s)^2$, onda je trajanje

$$T_3 = 2 \cdot \left(\alpha + \left(\frac{n}{s}\right)^2 \cdot \beta \right) = 2 \left(\alpha + \frac{n^2}{p} \cdot \beta \right) \quad .$$

(4) Lokalno množenje bloka od A s blokom od B i akumulacija na C izvodi se paralelno na svim procesorima i traje kao standardno množenje matrica reda

n/s, tj.

$$T_4 = 2\left(\frac{n}{s}\right)^3 = \frac{2n^3}{p^{3/2}} \quad .$$

Ukupno paralelno trajanje prethodnog algoritma je:

$$\begin{aligned} T(n,p) &= 2T_1 + (s-1)\cdot T_3 + s\cdot T_4 = (3s-2)\cdot\left(\alpha + \left(\frac{n}{s}\right)^2\cdot\beta\right) + \frac{2n^3}{s^2} \\ &= \frac{2n^3}{p} + (3\sqrt{p}-2)\cdot\alpha + \frac{(3\sqrt{p}-2)\,n^2}{p}\cdot\beta \quad . \end{aligned}$$

Efikasnost je:

$$E(p) = \frac{1}{1 + \dfrac{(3\sqrt{p}-2)\,p}{2n^3}\cdot\alpha + \dfrac{3\sqrt{p}-2}{2n}\cdot\beta} \quad .$$

U usporedbi s prstenom, vidimo da je:

- aritmetičko vrijeme isto;
- komunikacijsko vrijeme je ovdje manje i to za faktor

$$\frac{p-1}{3\sqrt{p}-2} \approx \sqrt{p}/3 \quad .$$

Taj faktor je reda veličine $s = \sqrt{p}$, što odražava činjenicu da je bisekcija 2D torusa za faktor $\sqrt{p}$ veća od one na prstenu (a naš algoritam paralelno komunicira, tako da bisekcija dolazi do izražaja).

5.3.7. Množenje matrica na 3D torusu

Na 3D torusu trajanje komunikacija može se smanjiti za dodatni faktor $p^{1/6}$, obzirom na 2D torus. U formuli:

$$C(i,j) = C(i,j) + \sum_{k=0}^{s-1} A(i,k) * B(k,j)$$

množenje $A(i,k) * B(k,j)$ obavlja procesor $P(i,j,k)$ u 3D torusu. Sume se zatim akumuliraju duž redova ove mreže (R. Agarwal, 1995.). Pokušajte sami sastaviti odgovarajući algoritam!

5.3.8. Množenje matrica na hiperkocki

Torus možemo uložiti u hiperkocku i tako preslikati pripadni Cannonov algoritam. Međutim, postoji i bolje rješenje. Hiperkocka ima veći broj žica po procesoru od pripadnog torusa, što omogućava ubrzanje početne faze zakreta u algoritmu.

Prvo treba polazni 2D raspored blokova preslikati na hiperkocku. Za to koristimo ulaganje 2D polja i 2D torusa u hiperkocku, koje smo već ranije opisali.

Neka je 2D polje, odnosno torus, kvadratnog oblika sa stranicom $s = 2^m$, tj. cijelo polje (ili torus) ima $2^m \times 2^m$ procesora ($p = 2^{2m}$), i neka je

$$G(m) = \{g(0), \ldots, g(2^m - 1)\}$$

Grayev m–bitni kôd. Tada procesoru (i, j) iz polja/torusa pridružujemo procesor

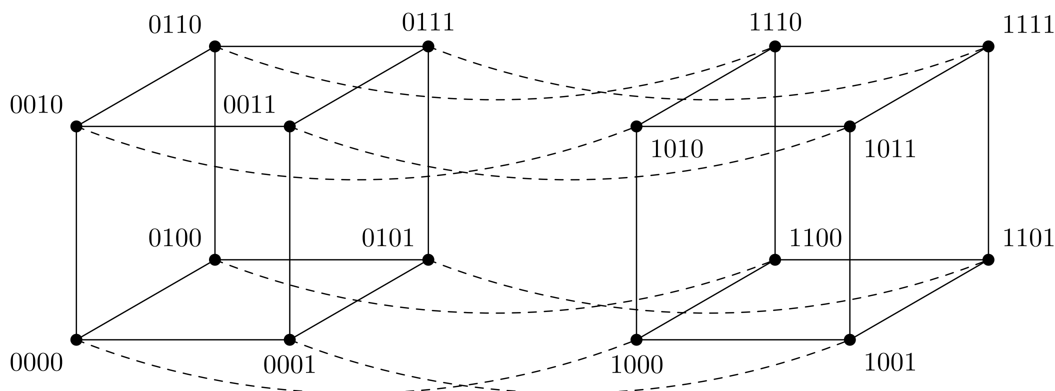
$$g(i) \cdot 2^m + g(j)$$

u hiperkocki (konkatenacija m bitova od $g(i)$ s m bitova od $g(j)$ daje adresu u hiperkocki dimenzije $2m$).

Na primjer, za $m = 2$ imamo slijedeće preslikavanje 4×4 polja ili torusa u 4D hiperkocku. Polja sadrže odgovarajuće adrese u hiperkocki.

0000	0001	0011	0010
0100	0101	0111	0110
1100	1101	1111	1110
1000	1001	1011	1010

Iz ovog preslikavanja je očito da se reci matrice, odnosno, reci procesora iz polja ili torusa, preslikavaju u nezavisne pod–hiperkocke polazne hiperkocke. Isto vrijedi i za stupce.



To znači da "višak" žica – dodatne veze u hiperkocki, možemo iskoristiti za ubrzanje faze zakreta u Cannonovom algoritmu. Autori te modifikacije su Dekel, Nassimi i Sahni.

Bez smanjenja općenitosti (za pojednostavljivanje zapisa) možemo pretpostaviti da su blokovi $A(i,j)$, $B(i,j)$ i $C(i,j)$ spremljeni u procesoru $i \cdot 2^m + j$.

Algoritam 5.3.6. (Množenje matrica: Dekel, Nassimi, Sahni)

Množenje matrica na hiperkocki.

```
for k := 0 to m − 1 do
  begin
  jk := 2^k and j;   {logički ''i'' brojeva 2^k i j, po bitovima}
  ik := 2^k and i;   {logički ''i'' brojeva 2^k i i, po bitovima}
    for i := 0 to s − 1 parallel do
      for j := 0 to s − 1 parallel do
        begin
        zamijeni A(i, j xor ik) i A(i, j);
        zamijeni B(jk xor i, j) i B(i, j);
          {xor = logički ''ekskluzivni ili'' brojeva, po bitovima}
        end;   {for i, j}
  end;   {for k}
for k := 0 to s − 1 do
  for i := 0 to s − 1 parallel do
    for j := 0 to s − 1 parallel do
      begin
      C(i, j) := C(i, j) + A(i, j) ∗ B(i, j);

      kružno ulijevo rotiranje svakog blok-retka od A za 1,
        u poretku Grayevog kôda;

      kružno nagore rotiranje svakog blok-stupca od B za 1,
        u poretku Grayevog kôda;
      end;   {for (i, j), k}
```

Promotrimo rad algoritma. Nakon faze zakreta, polazni $A(i,j)$ dolazi na mjesto $A(i, j \textbf{ xor } i)$. To postižemo promjenom po jednog bita od j u svakom trenu (od bita $k = 0$, do bita $k = m - 1$), tako da dobijemo odgovarajući bit u $j \textbf{ xor } i$.

U svakom koraku, $j \textbf{ xor } ik$ se može razlikovati od j samo u k–tom bitu, pa zamjena sadržaja s drugim procesorom treba samo komunikaciju među najbližim (susjednim) procesorima. To znači da se navedene zamjene zaista mogu izvesti paralelno, u m vremenskih koraka.

Analogno, $B(i,j)$ se pomiče u $B(j \textbf{ xor } i, j)$, jedan po jedan bit u svakom od m vremenskih koraka.

Trajanje ovakve faze zakreta je

$$T_z = 2m\left(\alpha + \left(\frac{n}{s}\right)^2 \cdot \beta\right) \quad ,$$

što je za faktor

$$\frac{\sqrt{p}}{2} \cdot \frac{1}{2m} = \frac{\sqrt{p}}{2\lg p}$$

bolje od Cannonovog algoritma na torusu. Ostatak algoritma (množenje, akumuliranje, micanje) ima istu cijenu, odnosno trajanje.

Za još brži algoritam, možemo pretpostaviti da svih $\lg p$ žica vezanih na svaki procesor u hiperkocki, možemo koristiti istovremeno – što daje $\lg p$ paralelizam komunikacije. Autori tog algoritma su Ho, Johnsson i Edelman.

Ovaj algoritam koristi "sve žice, svo vrijeme" na hiperkocki, pa je optimalan u tom smislu paralelne komunikacije. Odgovarajući hardware bio je realiziran u računalu CM–2, a ovaj algoritam se koristio za množenje matrica u pripadnoj biblioteci potprograma.

Usporedni pregled broja koraka za pojedine operacije u ova 3 algoritma za množenje $n \cdot 2^n \times n \cdot 2^n$ matrica na $2n$ dimenzionalnoj hiperkocki:

Algoritam	započinjanje (startanje) poruka	koraci slanja poruka	aritmetički koraci
Cannon	$2 \cdot (2^n - 1)$	$2n^2 \cdot (2^n - 1)$	$2n^3 \cdot 2^n$
Dekel, Nassimi, Sahni	$n + 2^n - 1$	$n^3 + n^2 \cdot (2^n - 1)$	$2n^3 \cdot 2^n$
Ho, Johnsson, Edelman	$n + 2^n - 1$	$n^3 + n \cdot (2^n - 1)$	$2n^3 \cdot 2^n$

Literatura

[1] KEY = `Akl-89`

S. G. Akl, *The Design and Analysis of Parallel Algorithms*, Prentice–Hall International, Englewood Cliffs, New Jersey, 1989.

[2] KEY = `Bertsekas-Tsitsiklis-89`

D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computation — Numerical Methods*, Prentice–Hall International, Englewood Cliffs, New Jersey, 1989.

[3] KEY = `Brent-74`

R. P. Brent, *The parallel evaluation of general arithmetic expressions*, J. Assoc. Comput. Mach., 21 (1974), pp. 201–206.

[4] KEY = `Demmel-96`

J. W. Demmel, *Applications of parallel computers*. Lecture notes for CS267, CS Department, University of California, Berkeley, 1996.

— `http://http.cs.berkeley.edu/~demmel/cs267/`.

[5] KEY = `Foster-95`

I. Foster, *Designing and Building Parallel Programs*, Addison–Wesley, Reading, Massachusetts, 1995.

— `http://www.mcs.anl.gov/dbpp/`.

[6] KEY = `Fox-Williams-Messina-94`

G. C. Fox, R. D. Williams, and P. C. Messina, *Parallel Computing Works*, Morgan Kaufmann Publishers, 1994.

— `http://www.npac.syr.edu/copywrite/pcw/`.

[7] KEY = `Gibbons-Rytter-88`

A. Gibbons and W. Rytter, *Efficient Parallel Algorithms*, Cambridge University Press, Cambridge, 1988.

[8] KEY = `Krishnamurthy-89`

E. V. Krishnamurthy, *Parallel Processing: Principles and Practice*, Addison–Wesley, Sydney, 1989.

[9] KEY = `Kronsjoe-85`

L. Kronsjö, *Computational Complexity of Sequential and Parallel Algorithms*, John Wiley and Sons, Chichester, 1985.

[10] KEY = `Lakshmivarahan-Dhall-90`

S. Lakshmivarahan and S. K. Dhall, *Analysis and Design of Parallel Algorithms: Arithmetic and Matrix Problems*, McGraw–Hill, New York, 1990.

[11] KEY = `Mody-88`

J. J. Mody, *Parallel Algorithms and Matrix Computations*, Oxford University Press, Oxford, 1988.

[12] KEY = `Quinn-87`

M. J. Quinn, *Designing Efficient Algorithms for Parallel Computers*, McGraw–Hill, New York, 1987.

[13] KEY = `VandeVelde-94`

E. F. Van de Velde, *Concurrent Scientific Computing*, Springer–Verlag, New York, 1994.