PARALELNO PROGRAMIRANJE

⇒ program se može izvest:

- **VIŠEDRETVENO** - može se izvodit paralelno na jednom računalu
- **VIŠEPROCESNO** - može se izvodit paralelno na jednom ili više računala

⇒ glavni i dugoročni programski alat je MPI ("Message Passing Interface")

↳ najviše se bavimo višeprocesnim paralelnim programiranjem što MPI dobro podržava

⇒ ciljevi predmeta:

a) upoznat oblike paralelnog programiranja
b) isprobat neke tehnologije paralelnog programiranja
c) primjenit naučene oblike u ostalim oblicima
d) ocjenit ponašanje paralelnog programiranja

⇒ paralelno programiranje se najčešće koristi za modeliranje i simulaciju te obradu velike količine podataka

MODELI PARALELNIH RAČUNALA

⇒ paralelne programe pišemo za neki model, a ne za neko konkretno računalo (prenosivost)

⇒ FLYNNOVA TAKSONOMIJA:

a) SISD ⇒ klasičan Von Neumannov model

b) SIMD ⇒ jedna instrukcija na više podataka

⇒ svi procesori rade potpuno sinkronizirano (ako postoj „if" u kojoj jedni idu, a drugi ne, ovi drugi moraju čekati)

c) MIMD ⇒ paralelno izvođenje više različitih poslova (neovisni procesori)

⇒ PODJELA PO MEMORIJI:

a) ZAJEDNIČKA MEMORIJA ⇒ neovisni poslovi, al procesori dijele isti spremnik

⇒ lakše programiranje, al ograničenja skalabilnost

b) RASPODJELJENA MEMORIJA ⇒ više neovisnih procesora s neovisnim (raselbnim) memorijama

⇒ brz pristup lokalnoj memoriji, ali nešto teže programiranje

c) MJEŠOVITA MEMORIJA ⇒ procesor smije pristupat memoriji drugog procesora, ali je to sporije

MODELI PARALELNIH PROGRAMA

a) RAZMJENA PORUKA

- komunikacija porukama između procesora
- SPMD - „Single Program, Multiple Data"
  (isti program se izvodi na više računala)
  (ovo je model programa za razliku od SIMD modela koji je model računala
  ↳ nemoj to miješat)
- postoje različite uloge unutar programa
- MPMD - „Multiple Program, Multiple Data"
  (može se svesti na SPMD model)

b) PODATKOVNI PARALELIZAM:

- primjena iste operacije na više elemenata
- prilagođeni programski jezici

c) MODEL ZAJEDNIČKE MEMORIJE:

- svi radovi dijele istu memoriju
- sinkronizacija semaforima il' monitorima
- jednostavno programiranje (kao na OS-u)

d) SUSTAV ZADATAKA (GRAF)

- sustav radataka i kanala prikazuje se usmjerenim grafom

SVOJSTVA PARALELNIH ALGORITAMA

⇒ **POŽELJNA SVOJSTVA:**

a) ISTODOBNOST („concurrency")

b) SKALABILNOST („scalability")

↳ mogućnost proširenja na proizvoljan broj procesora

c) LOKALNOST („locality")

↳ veći omjer lokalnog rada u odnosu na udaljeni pristup memoriji

d) MODULARNOST („modularity")

↳ mogućnost uporabe dijelova algoritma unutar različitih programa

⇒ **AMDAHLOV ZAKON:**

$$\text{ubrzanje} = \frac{1}{s + \frac{p}{N_p}}$$

s ⇒ slijedni udio programa

p ⇒ paralelni udio programa

N_p ⇒ broj procesora

⇒ ZRNATOST („granularity"):

↳ omjer između količine računanja (lokalnog rada) i količine komunikacije

↳ dvije krajnosti:

- sitnozrnatost – puno komunikacije, lako ujednačiti
- krupnozrnatost – malo komunikacije, al teško ujednačit opterećenje

⇒ PODATKOVNA OVISNOST:

↳ ponekad za neki proces nedostaju podac da on nastavi s poslom

↳ rješava se na dva načina (VIDI SLAJDOVE!)

⇒ POTPUNI ZASTOJ:

↳ engl. „deadlock"

↳ ponovi ukratko uvjete za potpuni zastoj

↳ normalna stvar prilikom programiranja paralelnog programa

MESSAGE PASSING INTERFACE

⇒ MPI je samo standard - za korištenje je potrebna implementacija

⇒ zašto baš MPI?

↳ radi i na paralelnom računalu s raspodijeljenom memorijom (umrežena računala)

↳ ta malo kompliciva programiranje komunikacije, ali nam omogućava općenitost jer je MPI standardna **višeprocesna tehnologija** koja je vrlo rašrirena

⇒ broj procesa je po „defaultu" stalan od početka do kraja paralelnog programa

⇒ obično svi procesi izvršavaju isti program (isti .exe), pa je ovo SPMD model paralelnog programa

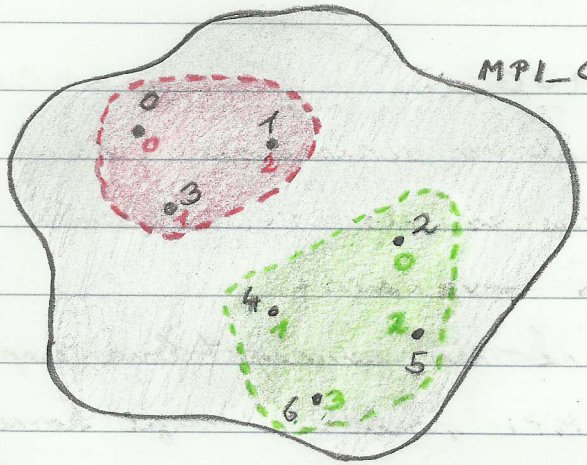
⇒ 4 OSNOVNE FUNKCIONALNOSTI:

1) otkriti ukupan broj procesa
2) identificirati vlastiti proces u grupi procesa
3) poslati poruku određenom procesu
4) primiti poruku (od određenog procesu)

⇒ MPI_COMM_WORLD:

↳ globalna grupa koja uključuje sve početne procese (svi koji žive od početka do kraja)

↳ može se dijeliti na podgrupe zbog raspodjele posla



⇒ da bi jedinstveno odredili proces, potrebno je navesti naziv grupe i indeks procesa u toj grupi (za početak imat ćemo samo globalnu grupu MPI_COMM_WORLD)

⇒ **POKRETANJE MPI PROGRAMA:** (na više računala)

↳ potrebno je:

- dijeljeni direktorij il ist .EXE na svakom računalu
- radni direktorij
- korisnički račun na svakom računalu
- program za pokretanje procesa („launcher")
- olakšavajuće: isključit „firewall"

⇒ skraćena sintaksa na radatke na papiru:

SEND(<ŠTO>, <KOME>)

RECV(<ŠTO>, <OD KOGA>)

↳ ako je <OD KOGA> postavljeno na '*', primamo poruku od bilo kojeg procesa

⇒ NAČINI KOMUNIKACIJE

↳ komunikaciju treba dobro isplanirati kako ne bi došlo do potpunog rastoja na MPI pozivu

↳ KADA NEKA MPI OPERACIJA ZAVRŠAVA?

- onda kada se sve upućene varijable (one korištene u MPI pozivu) mogu koristiti

↳ KADA SE KONTROLA VRAĆA POZIVATELJU MPI FUNKCIJE?

a) blokirajuća komunikacija („default")
- kontrola se vraća po završetku poziva MPI operacije

b) neblokirajuća komunikacija
- povratak je trenutan, ali se završetak MPI operacije mora provjeriti

⇒ mi trebamo pisati programe koj su neovisni o načinu rada MPI implementacije („synchronous" ili „buffered")

PROGRAM 0	PROGRAM 1
SEND(,1) →	RECV(,0)
RECV(,1) ←	SEND(,0)

komunikacija se ne presjeca pa sve dobro radi bez obzira o načinu rada MPI implementacije

PROGRAM 0	PROGRAM 1
SEND(,1) →	SEND(,0)
RECV(,1) ←	RECV(,0)

komunikacija se presjeca i ako se odluči slati u sinkronom načinu rbog velike poruke, dolazi do potpunog rastoja

NEMOJ PISATI OVAKVE PROGRAME!

⇒ GLOBALNA KOMUNIKACIJA:

↳ svojstva:

- svi moraju pozvati odredenu funkciju
- sve funkcije su blokirajuće
- nema oznake poruke
- količina poslanih podataka mora odgovarat količini primljenih podataka

↳ operacija slanja „jedan-prema-svima":

- funkcija BCAST

	BUFFER
ROOT	42
P1	/
P2	/
P3	/

→

	BUFFER
ROOT	42
P1	42
P2	42
P3	42

↳ operacija slanja „od-svih-prema-jednome":

- funkcija REDUCE
- skupljanje + operacija nad svim primljenim podacima

	SEND BUFFER	RECIEVE BUFFER
ROOT	2	/
P1	4	/
P2	8	/
P3	6	/

→

	SEND BUFFER	RECIEVE BUFFER
ROOT	2	20
P1	4	/
P2	8	/
P3	6	/

- redukcija je jako bitna operacija i nju ćemo puno koristiti
- binarna i asocijativna operacija je uvjet za ovu operaciju

↳ operacija SCATTER:

• raspršivanje podataka (jednolako) po procesima

	RECIEVE BUFFER	SEND BUFFER
ROOT	/	A B C D
P1	/	/
P2	/	/
P3	/	/

→

	RECIEVE BUFFER	SEND BUFFER
ROOT	A	A B C D
P1	B	/
P2	C	/
P3	D	/

↳ operacija GATHER:

• skupljanje na root proces

	SEND BUFFER	RECIEVE BUFFER
ROOT	A	/
P1	B	/
P2	C	/
P3	D	/

→

	SEND BUFFER	RECIEVE BUFFER
ROOT	A	A B C D
P1	B	/
P2	C	/
P3	D	/

⇒ NAPOMENA: potrebno je znati za BCAST i REDUCE (ove ostale samo informativno)

⇒ dodatna funkcija:

int MPI_Barrier (MPI_Comm comm)

↳ sve jedan proces ne nastavlja dok svi ne pozovu ovu funkciju

↳ služi za globalnu sinkronizaciju (najčešće za ispise i ostale interakcije s korisnikom)

ZAD: U nekom trenutku u svakom od „M" procesa nalazi se podatak „p". Odredi najveću vrijednost podatka „p" i tu informaciju proslijedi svim procesima. Napiši algoritam uz pomoć SEND i RECV gdje se komunikacija odvija u obliku lanca u dva prolaza.

NAPOMENA: za sve zadatke je pomoću:

M ⇒ broj procesa

ID ⇒ redni broj procesa

↳ za zadatak trebamo još dvije varijable:

p ⇒ podatak nekog procesa

max ⇒ najveća pomoću vrijednost nekog procesa

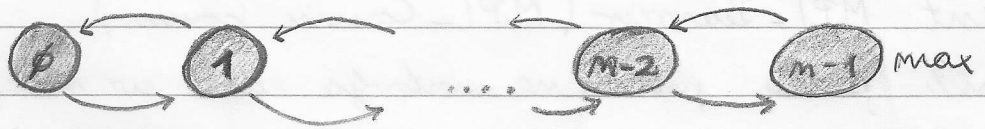
↳ algoritam lanca u dva prolaza:

- od 0 do zadnjeg i povratak vrijednosti,

a proslijedu veću

- zadnji stoga ima maksimum, pa samo

svima u natrag pošalje maksimum



↳ pseudokod:

```
AKO ( ID == 0 ) {
    SEND( p, 1 );
    RECV( max, 1 );
}
INAČE AKO( ID == n-1 ) {
    RECV( max, ID-1 );
    AKO( p > max )
        max = p;
    SEND( max, ID-1 );
}
INAČE {
    RECV( max, ID-1 );
    AKO( p > max )
        max = p;
    SEND( max, ID+1 );
    RECV( max, ID+1 );
    SEND( max, ID-1 );
}
```

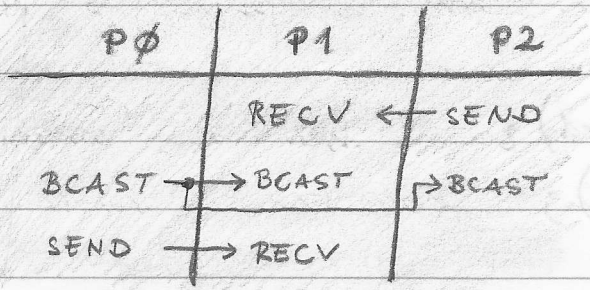
↳ složenost u ovisnosti o broju poslanih poruka:

$O(2n)$

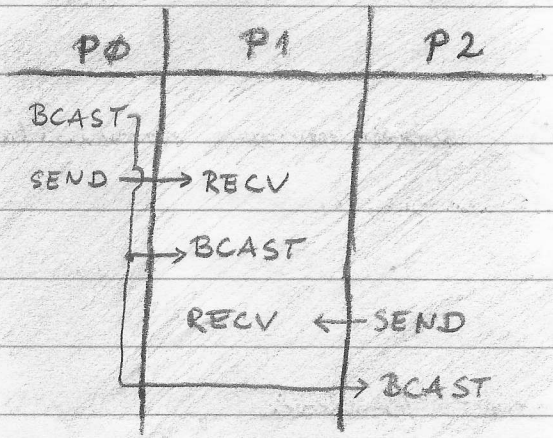
· primjet da se ovo može riješiti u logaritamskoj složenosti, ako grupiramo procese i među njima tražimo maksimum (grupe po dva: $O(\log_2 n)$)

PRIMJER 3 SA SLAJDOVA

⇒ jedan mogući scenarij:



⇒ drugi mogući scenarij:



↳ vidimo da je redoslijed poruka nije deterministički (engl. „ambigous"), pa na njega ne smijemo računati, ako želimo napisati ispravan program

NEBLOKIRAJUĆA KOMUNIKACIJA

⇒ koristi se za dodatno poboljšanje performansi ako se radi o velikim porukama

↳ npr. dok se prevrima velika poruka, možemo odmah vratiti kontrolu pozivajućem procesu

⇒ funkcije:

MPI_Irecv ⇒ „immediate receive"

MPI_Isend ⇒ „immediate send"

⇒ postoji samo „point-to-point" neblokirajuća komunikacija (nema globalnih neblokirajućih poziva)

⇒ PRIMJER KORIŠTENJA:

- dok primam podatke, vrati kontrolu procesoru da on radi nešto što ne uključuje podatke (npr. povećava preciznost nekog algoritma)
- s vremenom, procesor provjerava jesu li podaci stigli
- ako su podaci stigli, procesor može krenuti na posao vezan uz te podatke

PRVA DOMAĆA ZADAĆA

(PRIPREMA)

⇒ na početku, vilicu daj onu s manjim indeksom
↳ stoga, kako se vilica dijeli između dva filozofa, svaki filozof na početku jedini ima dvije, a ostali po jednu

⇒ uzet „ako ne jedem":
- mislim
- čekam drugu vilicu

} ovo su alternative

⇒ na "istovremeno" mišljenje i pitanje: „je l' me netko traži (je l' mi netko poslao poruku)"
↳ za to trebamo ASINKRONU KOMUNIKACIJU:
- ne znamo koliko poruka i ne znamo hoće l' uopće stati poruka

(funkcija: Iprobe)

ASINKRONA KOMUNIKACIJA

⇒ koristi se kada ne znamo ima li poruka za primit ili koliko tih poruka ima

⇒ <u>funkcija:</u>

MPI - I probe:

- ispitaj postoj li poruka koja je za mene poslana („je li me netko tražio?")
- „immediate probe"

SINKRONO PARALELNO RAČUNALO S ZAJEDNIČKOM MEMORIJOM

→ **model PRAM:**

- idealno paralelno računalo
- najlakše za razvoj paralelnih algoritama
  ↳ najlakše dokazat optimalnost nekog algoritma
- uz manje izmjene, algoritam za PRAM se može prebacit na realno računalo (uz očuvanje optimalnosti)

→ SVOJSTVA PRAM-a:

a) MIMD sa zajedničkom memorijom (ovo nije tako kod MPI-a)

b) proizvoljan il ograničen broj procesora (p)

c) sinkrono izvođenje: svi procesori jednako brzo izvode sve instrukcije

→ PRAM programiramo kao i do sada!
  ↳ pseudokodom uz proširenje:
  - BLOK „PARALELNO"

→ blok PARALELNO izvodi se paralelno na potrebnom broju procesora (koliko god treba)
→ nije moguće gniježđenje bloka PARALELNO

⇒ pristup memoriji unutar bloka PARALELNO:

EXCLUSIVE (E) - isključiv

CONCURRENT (C) - istovremen

⇒ inačice PRAM računala:

a) EREW - „exclusive read, exclusive write"
- najograničeniji model, pa ga po „defaultu" koristimo

b) CREW

c) ERCW

d) CRCW - najopćenitij model

⇒ dogovor o istovremenom pisanju:

1) PRIORITETNO:

↳ najmanji indeks uspije upisati rezultat

2) OPERACIJSKI:

↳ zadana operacija ima prioritet nad drugom

3) SLUČAJNO:

↳ slučajni procesor uspijeva zapisati podatak
(koristimo po „defaultu" jer je najopćenitije)

⇒ NAPOMENA: kako se u PRAM programima sve varijable globalne, ovdje nemamo naredbe SEND i RECV
(jer nema smisla - sve varijable su svima dostupne)

ALGORITAM ZBROJA PREFIKSA

⇒ engl. „all-prefix-sum"

⇒ jedan od najčešćih algoritama koji koristimo na PRAM računalima

⇒ ZADANO:

- binarna asocijativna operacija $\oplus$ s neutralnim elementom

⇒ algoritam vrši sljedeće preslikavanje vektora:

$$[a_0 \quad a_1 \quad \dots \quad a_{n-1}] \longrightarrow [a_0 \quad (a_0 \oplus a_1) \quad (a_0 \oplus a_1 \oplus a_2) \dots]$$

⇒ pretpostavljamo da su elementi u memoriji raspisani slijedno ili kao povezana lista

⇒ zbroj prefiksa se obično u kodu navodi „SCAN"

⇒ pomoćna operacija: „PRE-SCAN"

- pomaknut za jedno mjesto od „SCAN"-a
- stoga, mi ćemo napraviti „PRE-SCAN" jer se „SCAN" može dobiti onda u $O(1)$
  ↳ tako nam je lakše

⇒ za ostvarenje „PRE-SCAN"-a, trebamo reduciranje („reduce")

PR: Pokaži ostvarenje algoritma zbroja prefiksa na sljedećem nizu:

$A[] = [3\ 1\ 7\ 0\ 4\ 1\ 6\ 3]$; $n = 8$

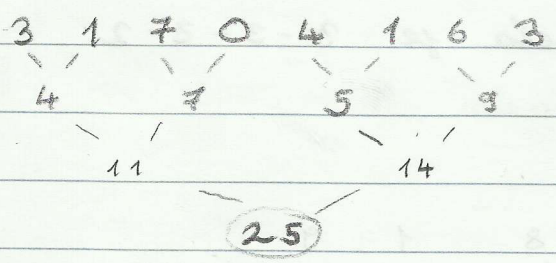
(idejno rješenje)

STABLO

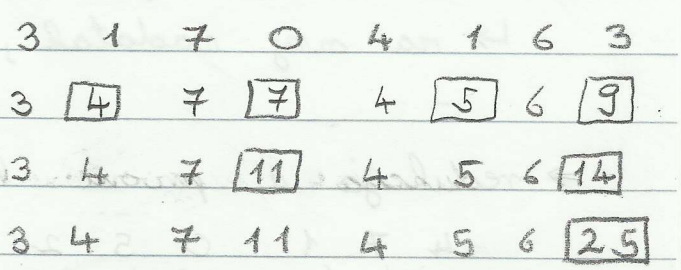
(egzaktno rješenje)

MEMORIJA

REDUCIRANJE



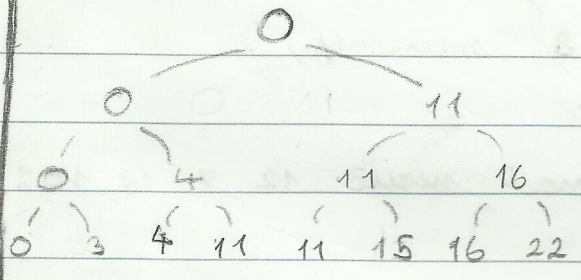
↳ za PRAM model, ovo je optimalan algoritam reduciranja

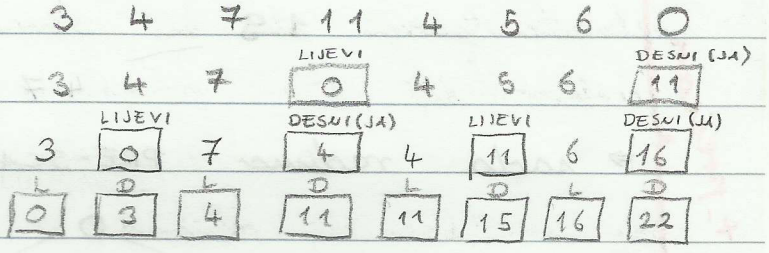


↳ u pseudokodu: +_REDUCE(A[])

↳ složenost: $O(\log_2 n)$

PRE-SCAN





↳ složenost: $O(\log_2 n)$

⇒ ukupna složenost:

$O(2 \log_2 n)$

⇒ dva problema: što ako n nije potencija broja 2, i što ako nemamo $n/2$ procesora?

PR: Provedi algoritam zbroja prefiksa za 4 procesora na zadanom nizu od 11 elemenata:

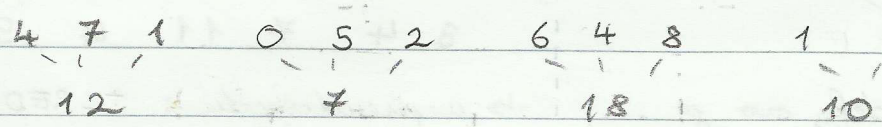
$A[] = [4\ 7\ 1\ 0\ 5\ 2\ 6\ 4\ 8\ 1\ 9]$

$p = 4 \qquad n = 11$

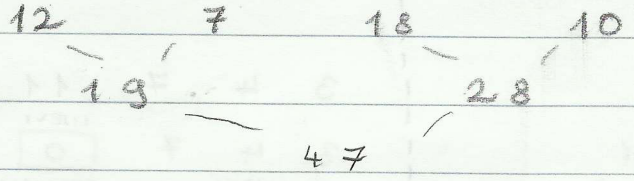
↳ ideja: svak procesor dobiva $\lfloor n/p \rfloor$ elemenata, a neki od njih i više

↳ na ovaj zadatak, raspodjela je 3-3-3-2

⇒ redukcija u prvom koraku:

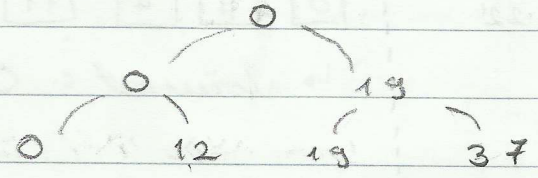


⇒ sada sigurno imamo potenciju broja dva elemenata, pa možemo reći klasičan algoritam reduciranja:



x - prescan

⇒ sada radimo PRE-SCAN na nizu 12 7 18 10:

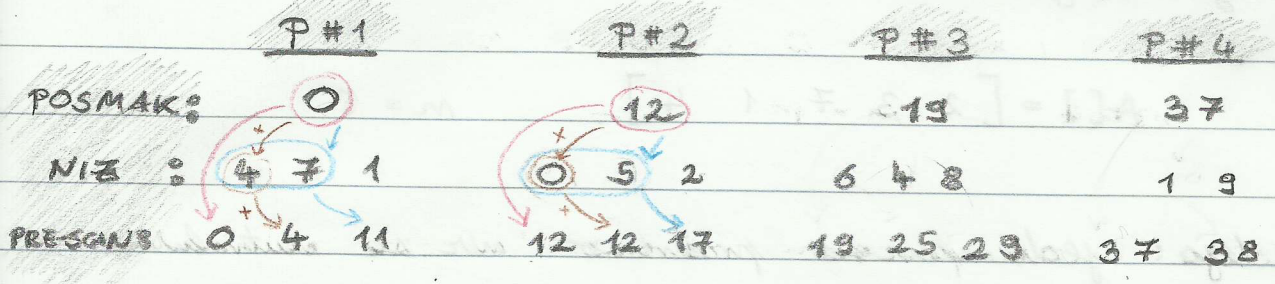


⇒ sada je još potrebno napraviti pre-scan za onaj početni niz od 11 elemenata

↳ ovdje, svaki procesor ne kreće od neutralnog elementa nego od rezultata pre-scana na gornjem nivou (tzv. „offset")

⇒ stoga, procesori rade sljedeće:

	P#1	P#2	P#3	P#4
POSMAK:	0	12	19	37
NIZ:	4 7 1	0 5 2	6 4 8	1 9
PRE-SCAN:	0 4 11	12 12 17	19 25 29	37 38



⇒ složenost ovog algoritma:

$$\frac{n}{p} + \log_2 p + \log_2 p + \frac{n}{p} = 2\left(\frac{n}{p} + \log_2 p\right)$$

↳ ukupno: $\boxed{O\left(\frac{n}{p} + \log_2 p\right)}$

⇒ NAPOMENA:

: u kodu, operaciju SCAN rovemo ovako:

B[] = +- scan(A[])

ovaj niz A[] se ne mijenja

PR: Proveđi pre-scan za niz koji nije veličine jednake potenciji broja 2 ?!

$$A[] = [2\ 3\ 7\ 1\ 4] \qquad n = 5$$

↳ ideja: jednostavno proširimo niz sa neutralnim elementom za radnu operaciju:

2	3	7	1	4	0	0	0	
2	**5**	7	**8**	4	**4**	0	**0**	
2	5	7	**13**	4	4	0	**4**	
2	5	7	13	4	4	0	**17**	
2	5	7	13	4	4	0	**0**	„pre-scan"
2	5	7	**0**	4	4	0	**13**	
2	**0**	7	**5**	4	**13**	0	**17**	
0	**2**	**5**	**7**	**13**	**17**	**17**	**17**	

⇒ NAPOMENA: još nismo isprobali primjer s brojem procesora koji nije potencija broja dva

↳ napravi takav primjer ?! //

PR: Jedna od primjena pre-scan-a je provjera uzlaznosti niza. Napiši pseudokod za provjeru uzlaznosti niza A[] !

```
PARALELNO ( i = 0 DO n-2)
    [AKO ( A[i] ≤ A[i+1] )
        | P[i] = 1 ]

    // trebamo pomoćna polja
    K[i] = A[i];
    P[i] = 0;
    AKO ( K[i] ≤ A[i+1] )
        | P[i] = 1;

RBZ = OR_REDUCE( P[] )
```

NE MOŽE NA EREW PRAM-u !

ZAD: Napiši algoritam za CROW PRAM koji će za zadani niz P[] provjeriti ima li u nizu elemenata jednakih vrijednosti (duplikata).

↳ pretpostavka po defaultu:

- za niz od „n" elemenata na raspolaganju je „n" procesora, a rezultat mora biti zapisan u jednoj izlaznoj varijabli

↳ izlazna varijabla: duplih

```
duplih = 0;
ZA( i = 0 DO n-2)
    PARALELNO( j = i+1 DO n-1)
        AKO( P[j] == P[i])
            duplih = 1;
```

↳ riješimo ist zadatak za EREW:
(malo složenije)

```
PARALELNO( i = 0 DO n-1)
    K[i] = P[i];
    REZ[i] = 0;
ZA( d = 1 DO (n/2)+1)
    PARALELNO( i = 0 DO n-2)
        AKO( P[i] == K[(i+d)%n])
            REZ[i] = 1;
duplih = OR_REDUCE(REZ[]);
```

zašto ovdje smijemo do polovice (mislim da je krivo)

ZAD: Algoritam za CRCW i EREW PRAM koji će za zadani niz P[] odrediti broj jedinstvenih elemenata.

$$P[] = [\,1\ 2\ 1\ 3\ 4\ 2\ 5\ 1\,] \longrightarrow 5$$

CRCW

```
broj = 1;
ZA( i = 0 DO n-2) // gledaj sve od prvog do predzadnjeg
    inc = 1; // pretpostavi da nema duplikata
    PARALELNO( j = i+1 DO n-1) // gledaj sve dalje od „i"
        AKO ( P[j] == P[i] ) // našli smo istog
            inc = 0;
    broj += inc;
```

⇒ ovo je linearne složenosti ($O(n)$) ∇∘

OSTVARENJE ALGORITMA RADIX-SORT

⇒ koristi operaciju pre-scan za ostvarenje paralelnog algoritma sortiranja

⇒ PRETPOSTAVKA: ulazni brojevi su ograničene veličine (npr. 32-bita)

PR: Pogledajmo kako Radix-sort radi!

NIZ:	5	7	3	1	4	2	7	2	
BITOVI:	1	1	1	1	0	0	1	0	(zadnji bit)
NIZ'	4	2	2	5	7	3	1	7	
BITOVI:	0	1	1	0	1	1	0	1	(predzadnji bit)
NIZ''	4	5	1	2	2	7	3	7	
BITOVI:	1	1	0	0	0	1	0	1	(predpredzadnji bit)
NIZ''':	1	2	2	3	4	5	7	7	

pretpostavimo 3 bitni zapis

↳ ovo je bilo korištenjem operacije „split"

• „split" se može realizirat korištenjem pre-scana (vidi slajdove!)

↳ ne treba znat detaljno

1 ⇒ pogledajmo jednu iteraciju prethodnog primjera, ali sada tako da pribavimo realizaciju operacije „split":

NIZ:	5	7	3	1	4	2	7	2
BITOVI:	1	1	1	1	0	0	1	0
DOLJE:	0	0	0	0	[0]	[1]	2	[2]
PRESCAN UNATRAG:	4	3	2	1	1	1	0	0
GORE:	[3]	[4]	[5]	[6]	6	6	[7]	7
INDEX:	3	4	5	6	0	1	7	2
NIZ':	4	2	2	5	7	3	1	7

← +_prescan(!BITOVI)

NE TREBA ZNATI PROVESTI ▽

(treba znati da postoji ▽)

⇒ složenost jednog „split"-a je jednaka složenosti operacije prescan

⇒ ukupna složenost sortiranja:

$$O\left(\frac{n}{p}\log(n) + \log(p)\cdot\log(n)\right)$$

↳ za bazu 2 i vrijednost do „n" ▽

ASINKRONI PRAM

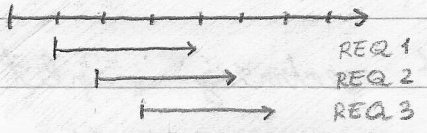
⇒ IZMJENE MODELA PRAM:

1) procesori su nesinkroni
   ↳ trebamo eksplicitnu sinkronizaciju!

2) pristup lokalnoj memoriji obavlja se u jednom koraku, a pristup globalnoj memoriji u „d" koraka

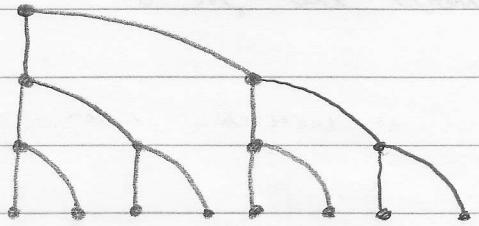
↳ napomena: uzastopni pristupi globalnoj memoriji mogu se ulančavati!

↳ npr. $d = 3$:



PR: Raspodjela jednog podatka na cijelo polje (8 elemenata)!

a) Prikaz binarnim stablom:



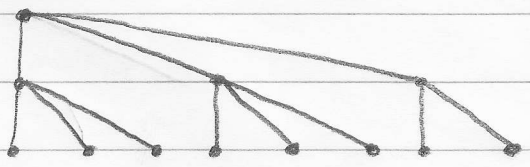
PRAM

$T_1 = 1$
$T_2 = 1$
$T_3 = 1$
$\Big\} T = 3$

APRAM ($d=3$)

$T_1 = 3$
$T_2 = 3$
$T_3 = 3$
$\Big\} T = 9$

b) Prikaz (d)-narnim stablom ($d = 3$):



PRAM

$T_1 = 2$
$T_2 = 2$
$\Big\} T = 4$

APRAM ($d=3$)

$T_1 = 4$
$T_2 = 4$
$\Big\} T = 8$

↳ vidimo da binarno stablo na APRAMU nije optimalno! (d)-narno stablo jest!

⇒ MODEL APRAM:

- postoj p - procesora
- svaki ima lokalnu memoriju
- globalna memorija („g" globalnih polja)
- funkcionira kao nesinkroni MIMD
- 4 vrste dodatnih instrukcija:

1) GLOBALNO ČITANJE

↳ prepisivanje iz globalne u lokalnu memoriju !!

2) GLOBALNO PISANJE (analogno)

3) LOKALNA OPERACIJA

↳ operacija u lokalnoj memoriji

4) SINKRONIZACIJA

↳ svi čekaju da svi dođu na isto mjesto u programu („barijera")

⇒ NAPOMENA: APRAM program je raspao na asinkronih odsječaka

↳ radnja instrukcija je uvijek barijera

⇒ OGRANIČENJE: u asinkronom odsječku, samo jedan procesor smije dirati jednu memorijsku lokaciju globalnog adresnog prostora

⇒ OCJENA SLOŽENOSTI:

↳ jedinična mjera: lokalna operacija

↳ globalni pristup: „d" - koraka

↳ uzastopnih „k" globalnih pristupa: $(d + k - 1)$ - koraka

↳ sinkronizacija: mjera $B(p)$ - koraka

• pretpostavka:

$$2 \leq d \leq B \leq p$$

PRILAGODBA PRAM ALGORITAMA ZA ASINKRONI MODEL

⇒ jedna PRAM instrukcija transformirana:

- globalno čitanje
- lokalna operacija
- globalno pisanje
- sinkronizacija

⇒ PRILAGODBA UZ JEDNAK BROJ PROCESORA (p):

↳ jedna EREW PRAM instrukcija na APRAM-u:

- čitanje (d)
- ograda (B)
- lokalna instrukcija
- pisanje (d)
- ograda (B)

} ukupno: $2B + 2d + 1$ ⇒ $O(B)$

PR: Neka su podaci za EREW PRAM: $p = 1024$, $B = \sqrt{p}$, $d = 3$. Na APRAM-u, jedna instrukcija će uzeti 71 korak!

⇓

velik trošak sinkronizacije

⇒ PRILAGODBA UZ p/B PROCESORA:

↳ manje procesora znači i manje troškova sinkronizacije (tražimo optimum)

↳ jedan APRAM procesor, simulira rad B PRAM procesora

↳ B' je trošak sinkronizacije p/B' procesora (ovo je rekurzivna definicija)

PR: Ako je $p = 1024$, $B = \sqrt{p}$, odredi B' ?

$$B' = \sqrt{\frac{p}{B'}} \Rightarrow B' = p^{1/3} \Rightarrow B' \approx 10$$

↳ svaki APRAM procesor radi sljedeće za jednu EREW PRAM instrukciju:

- čita za B' PRAM procesora $(d + B' - 1)$
- ograda (B')
- lokalne instrukcije (B')
- pisanje $(d + B' - 1)$
- ograda (B')

ukupno: $5B' + 2d - 2$
$\Rightarrow O(B')$

PR: Ako su podaci za APRAM $p = 100$, $B' = 10$

↳ koraci za jednu PRAM instrukciju:

$K = 54$ (nije optimum jer ovisi o izboru konstante)

↳ za isto podatke uz $B' = 16$, koraci za jednu PRAM instrukciju: $K = 48$

(ne treba znat naći optimum)

APRAM - REDUCIRANJE

⇒ reduciranje se često koristi, pa želimo maksimalne performanse na APRAM računalu

↳ dva pristupa uporabata do sada nisu optimalni jer su automatizirani (ne znaju ništa o problemu)

PR: Imamo sljedeće podatke. Pretvori u APRAM model!

$n = 1024$, $B = \sqrt{p} = 32$, $d = 3$

a) algoritam za „n" procesora: B-nario stablo

↳ dubina stabla: $(\log(n) / \log(B)) = \log_B n$

↳ na svakoj razini stabla:

- B-1 čitanje: $d + B - 2$
- zadana operacija: $B - 1$
- pisanje rezultata: d
- ograda: B

} $3B + 2d - 3$

↳ na jednoj razini dobivamo $3B + 2d$ koraka što daje $O(B)$ složenost (za primjer: 102 koraka)

↳ da imamo dubinu stabla jednaku dva ($d = 2$), ukupno je izvršeno 204 koraka na svakoj razini stabla

↳ stoga, složenost je:

$$O(B \cdot \log(n) / \log(B))$$

b) algoritam uz manji broj procesora

↳ ovo je optimalniji algoritam od prethodnog, ali rahtjeva novi parametar T

↳ svaki procesor sljedno zbraja T elemenata

↳ pomoćni parametar: $T = B \cdot \log(n) / \log(B)$

• za primjer: $T = 64$

↳ novi broj procesora: n/T

↳ za naš primjer:

$$p = n/T = 16 \qquad B' = 4$$

POSTUPAK ALGORITMA

↳ svaki procesor sljedno zbraja T elemenata:

$$(d + T - 2) + (T + 1) + B' \Rightarrow O(T)$$

↳ za primjer: 132 koraka

↳ dolazeći niže je dubljine n/T reducira B'-nomim stablom:

• dubina $\log(n/T) / \log(B')$

↳ za primjer: 2

• na jednoj razini, radi se:

$3B' + 2d$ koraka

↳ za primjer: 18 koraka

↳ za primjer, ukupno: $132 + 2 \cdot 18 = 168$

↳ složenost:

$$O(T) = O\left(B \log(n) / \log(B)\right)$$

ZAD: Napiši algoritam na EREW PRAM koji će na dva ulazna niza znakova duljine „n" odrediti koji je prije po abecedi. Treba ispisati 'A', 'B' ili 1 (ako su jednaki!)

↳ ideja: iskoristiti procesore da u $O(1)$ usporedimo znakove na istim indeksima:

```
PARALELNO ( i = 0 DO n-1)
    AKO ( A[i] < B[i])
        REZ[i] = 1;
    INAČE AKO(A[i] > B[i])
        REZ[i] = -1;
    INAČE
        REZ[i] = 0
(*)
```

↳ sada, u polju REZ[] treba naći prvi različit od nule što možemo reduciranjem u $O(\log n)$:

- no, trebamo definirati svoju binarnu asocijativnu operaciju:

$$\underbrace{0\,0}_{0} \quad \underbrace{1\,0}_{1} \quad \underbrace{-1\,0}_{-1} \Rightarrow \text{NAĐI}(a,b) = \begin{cases} a, & a \neq 0 \\ b, & \text{INAČE} \end{cases}$$

```
(*) ODG = NAĐI_REDUCE ( REZ[] )
    AKO (ODG == 1)  ISPIŠI ( 'A' )
    INAČE AKO ( ODG == -1)  ISPIŠI ( 'B' )
    INAČE ISPIŠI ('1')
```

ZAD: U niru A[] duljine „n" zapisana je jedinstvena vrijednost na jednom mjestu, a ostale vrijednosti su jednake nuli. Napiši algoritam za CRCW PRAM koji će vrijednost iz neprazne elementa zapisati u sve elemente niza.

```
PARALELNO( i=0 DO n-1)
| AKO ( A[i] ≠ Ø)
|   | A[Ø] = A[i];
| A[i] = A[Ø];
```

} O(1)

↳ napišimo ovo za EREW (neće biti O(1) nego $O(\log(n))$)

· pronalaženje vrijednosti je isto, jer samo jedan proces ulazi u AKO:

```
PARALELNO( i=0 DO n-1)
| AKO ( A[i] ≠ 0)
|   | A[Ø] = A[i];
```

· ako želimo biti sigurni da ne narušavamo EREW model, radimo +_REDUCE:

A[n-1] = +_REDUCE(A[])

⇓

veća složenost, ali ne smeta zbog veće složenosti u drugom dijelu (raspodjela)

↳ raspodjela na EREW PRAM-u ide u logaritamskoj složenosti:

```
ZA ( d = log2(n) - 1  DO  0 )
    PARALELNO ( i = 0  DO  n-1  KORAK  2^(d+1) )
        A[ i + 2^(d) - 1 ] = A[ i + 2^(d+1) - 1 ];
```

ZAD: U nizu 4 su niza, pravi element su nula, a pozicije sa žetonom su označene jedinicama. Napiši algoritam za EREW PRAM koji za radnu niz P[] dužine „n" otkriva postoji li u njemu četiri žetona u nizu?

0011001110001111000 1....

ovo tražimo

00120012300012340001

suma niza jedinica | suma niza jedinica | suma niza jedinica

↳ definišemo svoju operaciju:

$$X(a,b) = \begin{cases} \emptyset, & b = \emptyset \\ a+b, & b \neq \emptyset \end{cases}$$

```
P[] = X_SCAN(P[]);
M   = MAX_REDUCE(P[]);
AKO (M > 3)
    | ISPIŠI ("DA")
INAČE
    | ISPIŠI ("NE")
```

↳ **ne valja** jer X nije asocijativna operacija, pa ne možemo sa ovom operacijom.

$$1\overset{X}{(}01\overset{X}{)} \neq (10)1$$

↳ sada napravimo dobro rješenje:

```
PARALELNO (i=0 DO n-5)
    | ZB[i] = P[i];
    | ZA (j=1 DO 3)
    |     | ZB[i] += P[i+j];
    ⊥

M = MAX_REDUCE (ZB[]);
AKO (M>3)
    | ISPIŠI ("DA")
INAČE
    | ISPIŠI ("NE")
```

⇒ NAPOMENA: pogledaj radatak 2.7, al ga ne trebaš rješiti u logaritamskoj složenosti nego u linearnoj

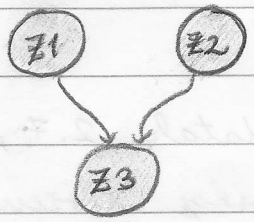
OBLIKOVANJE PARALELNIH ALGORITAMA

⇒ sada ćemo se skoncentrirat na odrestene računalne modele i odrestene programske modele kako bi na konkretnom sustavu prikazali oblikovanje paralelnih algoritama

⇒ MODELI KOJE KORISTIMO:

→ računalni model: MIMD raspodijeljeno memorije
- statna brzina komunikacije
- lokalni pristup brži od udaljenog

→ programski model: radaci, komunikacijski kanali



⇒ PARALELNI PROGRAM: skup radataka koj se mogu izvoditi istovremeno

⇒ ZADATAK: slijedni program koj izvodi lokalne operacije, slanje i primanje poruka te stvaranje novih radataka

⇒ napomena: primanje od sada smatramo isključivo blokirajuće

⇒ napomena: radaci se proizvoljno pridružuju procesorima

⇒ četiri faze razvoja:

1) PODJELA

· dekompozicija (rastav) problema

2) KOMUNIKACIJA

· odredivanje minimalne komunikacije

3) AGLOMERACIJA

· grupiranje zadataka i/il komunikacijskih kanala (ako je to isplativo)

4) PRIDRUŽIVANJE

· dodjeljivanje zadataka procesorima

FAZA PODJELE

⇒ **cilj**: postić sitnozornatu podjelu problema (dobiti što više zadataka)

⇒ idealni slučaj:

· podjela koja ne zahtjeva dupliciranje podataka ili računanja (vrlo rijetko u praksi)

⇒ DVIJE VRSTE PODJELE:

a) PODJELA PODATAKA

· podaci se dijele u manje cjeline koje obrađuju različiti zadaci

↳ napomenimo da u ovom koraku još ne gledamo količinu komunikacije, već samo dijelimo podatke na manje dijelove (bez analize komunikacije)

b) PODJELA RAČUNANJA

· dijel se izračunavanje, a ovisno o tome i podaci

⇒ POŽELJNA SVOJSTVA FAZE PODJELE:

1) broj zadataka je barem red veličine veći od broja procesora (time kasnije imamo veću slobodu)
2) bez uvišestručavanja podataka ili računanja
3) zadaci su podjednake veličine
4) broj zadataka ovisi o veličini problema (10x10 nije isto kao 1000x1000)

FAZA KOMUNIKACIJE

⇒ **cilj**: raspodjela i smanjenje ukupne količine komunikacije

⇒ PODJELA KOMUNIKACIJE:

↳ ima više podjela:

a) lokalna / globalna

b) strukturirana / nestrukturirana

c) statička / dinamička (statička znači da proces uvijek komunicira s istom grupom podataka)

d) sinkrona / asinkrona

⇒ najčešće optimiramo lokalnu komunikaciju

LOKALNA KOMUNIKACIJA:

↳ maska (engl. „stencil") - skup procesa (okolna susjeda) koji su potrebni za računanje nove vrijednosti

PR: Jacobijeva metoda i Gauss-Seidel metode (vrd sama okolina na slajdovima)

GLOBALNA KOMUNIKACIJA:

↳ koriste se komunikacijski obrasci (npr. B-norma stabla kod operacije redukcije)

OKREMI →

↳ obrazac redoslijeda — BITNO!

```
ZA ( i = 0 ; (i < log n) && ( proces_ID % 2^i == 0 ); i++ )
    dest_ID = ID XOR 2^i ;   // isključivo XOR
    AKO ( proces_ID % 2^(i+1) == 0 )
        RECV ( podatak, dest_ID );
        operacija ();
    INAČE
        SEND ( podatak, dest_ID );
```

ASINKRONA KOMUNIKACIJA:

↳ javlja se kad nismo sigurni hoće li komunikacije ući

PRI: Obrada velike količine podataka

OVO SU ALTERNATIVE

a) jednolika raspodjela podataka po radacima uz dodatu lokalnih i udaljenih zahtjeva

b) posebni radaci za računanje i radaci radnici za računanje

⇒ POŽELJNA SVOJSTVA FAZE KOMUNIKACIJE:

1) podjednaka količina komunikacije po radacima
2) manja količina lokalne komunikacije
3) istodobna komunikacija (ne da se slijedno dvaju zahtjeva)

FAZA AGLOMERACIJE

⇒ <u>cilj</u>: povećavanje zrnatosti (usklađivanje troškova komunikacije i računanja)

↳ najčešće se radnici okrupnjuju

⇒ <u>drugi cilj</u>: prilagodljivost na varijabilne ulazne uvjete (npr. varijabilan broj procesora)

⇒ <u>treći cilj</u>: smanjenje troškova implementacije

POVEĆANJE ZRNATOSTI

↳ najčešće ista količina komunikacije, al povećano količinu računanja

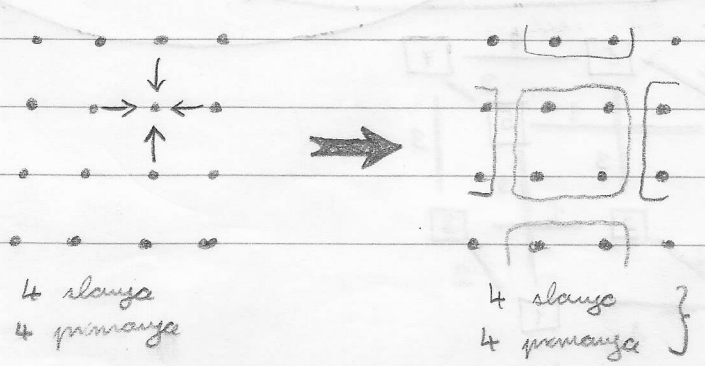
↳ no, možemo i:

a) smanjiti veličinu poruka

b) smanjiti broj poruka

↳ dajmo manje rečenom gore:

1) POVEĆAVANJE ZADATAKA - grupiramo više zadataka u jedan (vidi primjer s računanjem elemenata matrice na slajdu)



2) UVIŠESTRUČAVANJE RAČUNANJA - povećavanje količine računanja u cilju smanjenja komunikacije

- više varijabli nadataka računa istu stvar kako rezultat ne bi trebali iskomunicirati (već ga imaju lokalno)

↳ npr. operacija **all-reduce**:

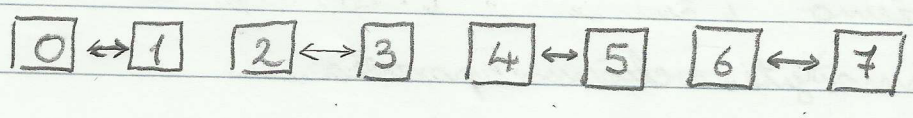
↳ rezultat redukcijske operacije mora na kraju imati svi radnici

↳ metode rješavanja:

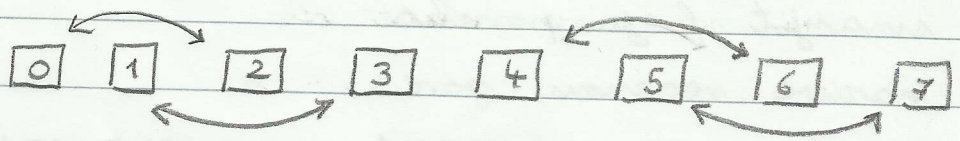
a) binarno stablo u dva prolaza
$2 \log(n)$ koraka

b) struktura hiperkocke (1 prolaz)
$\log(n)$ koraka

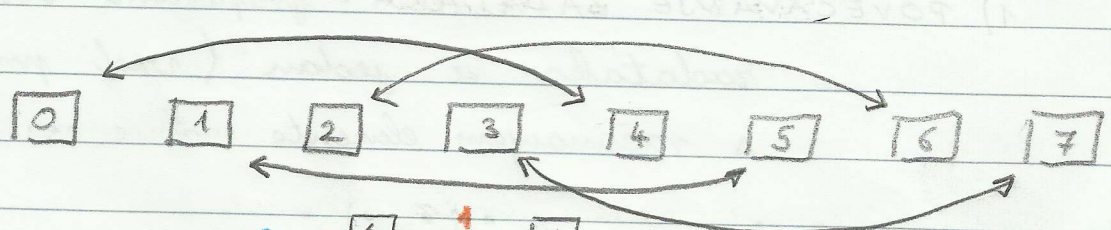
PR: Operacija **all-reduce** za 8 procesa (hiperkocka).

1) 

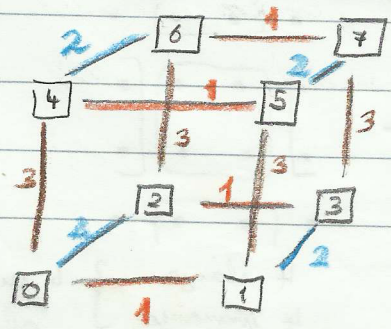
} u paralel

2) 

} u paralel

3) 

} u paralel



↳ ukoliko želimo uzastopno komunicirati (ulančani procesi), bolja je struktura leptira (VIDI SLAJDOVE!)

↳ obrazac hiperkocke BITNO!

```
ZA( i=0; i < log n; i++)
    dest_ID = proces_ID XOR 2^i;
    SEND(moj_podatak, dest_ID);
    RECV(novi_podatak, dest_ID);
    operacija();
```

↓

potencijalni deadlock ako se šalje velika poruka u MPI-u (SEND prijeđe u asinkrono izvođenje)

↳ rješenje je uvesti „if" kako bi uveli redoslijed jer se može desiti da svi ostanu na RECV jer SEND nije završio

⇒ NAPOMENA: kada svi nešto trebaju znati i treba biti u logaritamskoj složenosti, koristi obrazac hiperkocke

ZAD: 2.7 Svak proces treba saznat indeks svog sljedbenika i prethodnika!

```
brojevi [N] = {N, N, ..., N};
brojevi [RBR] = ID;
ZA ( i = 0; i < log N; i++)
    dest_ID = ID xOR 2^i;
    SEND ( brojevi [ ], dest_ID);
    RECV ( _brojevi [ ], dest_ID);
    ZA ( ∀ j  U  brojevi )
        AKO ( _brojevi [j] < N)
            brojevi [j] = _brojevi [j];

prethodnik = brojevi [RBR - 1];
sljedbenik = brojevi [RBR + 1];
```

PRILAGODLJIVOST

↳ želimo da broj radataka:

a) ovisi o veličini problema

b) veći od broja procesa

↳ ovo je drugi cilj faze angloneracije

SMANJENJE TROŠKOVA IMPLEMENTACIJE

↳ nije nam osobito naglašeno jer ovisi o radatku

↳ cilj je smanjiti trud za prilagodbu algoritma

⇒ POŽELJNA SVOJSTVA FAZE AGLOMERACIJE:

1) podjednaka složenost radataka

2) prilagodljivost broja radataka (veličini problema)

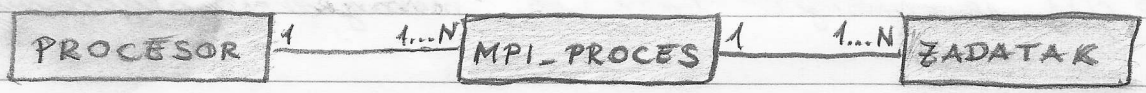
3) što manji broj radataka, ali i dalje veći od broja procesa

4) što manji troškovi prilagodbe

FAZA PRIDRUŽIVANJA

⇒ daje odgovor na pitanje:
„Koji procesor izvodi koj radatak?"

⇒ vrano za MPI:
- na jedan procesor možemo staviti 1 do N MPI_proces
- na jedan MPI_proces možemo staviti 1 do N radataka



⇒ najjednostavnije pridruživanje:
- jednolika raspodjela radataka po procesorima

(jedan radatak može imati više podatkovnih elemenata - npr. 1000x1000 je 100 matrica veličine 100x100)

⇒ optimizacija (poboljšanje) pridruživanja:
↳ dva načina:

a) UJEDNAČAVANJE OPTEREĆENJA

b) RASPOREĐIVANJE ZADATAKA

UJEDNAČAVANJE OPTEREĆENJA

↳ pretpostavka: stalan broj radataka

↳ dva tipa:

1) STATIČKO - samo na početku

2) DINAMIČKO - tijekom rada programa

↳ može biti:

a) lokalno - gledam samo susjedstvo
- brža, al manje učinkovita

b) globalno - gledamo sve procese
- sporije, al učinkovitije

↳ metode:

A) REKURZIVNA BISEKCIJA

↳ „podjeli-pa-vladaj"

↳ graf dijelimo tako da izmamo u obzir opterećenje procesa

↳ mogući načini dijeljenja:

1) po dulјoj dimenziji

2) po komunikacijskim i/il računalnim troškovima (rahtjeva globalno

B) LOKALNI ALGORITMI

↳ povremena usporedba opterećenja susjednih procesora

↳ ovi algoritmi imaju mal troškova, al daju sporu prilagodbu opterećenja

C) VJEROJATNOSNE METODE

↳ slučajno radatke bacamo po procesorima

↳ mali troškovi, rad uvijek

↳ problem je što je moguće da se generira puno komunikacije

D) CIKLIČKO PRIDRUŽIVANJE

↳ za „p" procesora, svakom procesoru ide svaki „p"-ti radatak

↳ po defaultu, MPI ovako radi

⇒ RASPOREĐIVANJE ZADATAKA

↳ pretpostavka: više radataka, kradeg trajanja (nastaju novi radaci, izvršavaju se i gase)

↳ metode:

A) VODITELJ / RADNIK

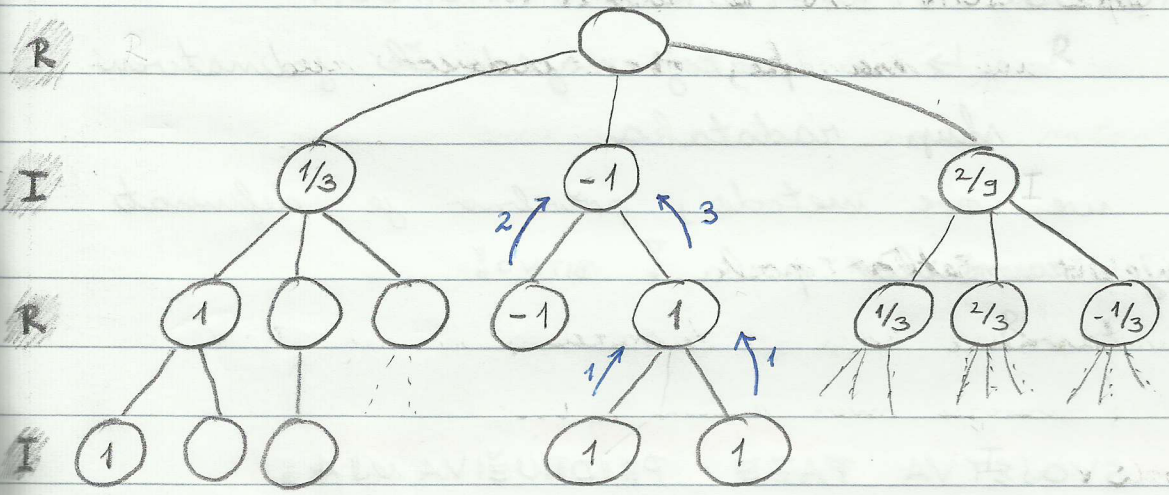
↳ jedan voditelj, ostali su radnici

↳ problem je zagušenje voditelja

PRIPREMA ZA DRUGU DOMAĆU ZADAĆU

⇒ za pridruživanje, najjednostavnije pridruživanje u širinu (nakon dubine D, ...)

⇒ koristi se brute-force pretraživanje: (R - računalo, I - igrač)



⇒ moramo stvoret barem 49 zadataka (dubina 2) kako bi se učinkovito rasporedio

⇒ RASPOREĐIVANJE ZADATAKA (nastavak)

↳ metode:

ovo ne prepričava na labos

A) VODITELJ / RADNIK

↳ jedan voditelj, ostali radnici

↳ problem je zagušenje voditelja

B) HIJERARHIJSKI VODITELJ / RADNIK

↳ definiraju se podgrupe radnika s dodatnim voditeljima

C) DECENTRALIZIRANA METODA

↳ ne postoji rasjedničko jedinstveni skup zadataka

↳ za ove metode, potrebno je definirat otkrivanje završetka posla ?! BITNO!

⇒ POŽELJNA SVOJSTVA FAZE PRIDRUŽIVANJA:

1) ako imamo voditelja, pobrini se da on nije preopterećen

2) osiguraj isplativ trošak usklađivanja opterećenja

⇒ NAPOMENA: jednoliko rasporedivanje po radacima može stvarati neusklađeno opterećenje ako radac nisu iste vrste (npr. vremenska prognoza - kod zadataka za dan se računa više stvari nego u noći)

KVANTITATIVNA ANALIZA PARALELNOG ALGORITMA

⇒ prvo, želimo definirat performanse po kojima ćemo ocjenjivati paralelni algoritam

⇒ sami kriteriji su ovisni o primjeni:

↳ npr. MULTIMEDIJA: bitan je **kapacitet** (PROTOČNOST) da možemo poslužit puno korisnika

↳ npr. SUSTAV UPRAVLJANJA: bitno je **trajanje odziva** („DELAY") kako autonomno vozilo ne bi bilo problematično

↳ npr. PLAČANJE RAČUNANJA: bitna je **učinkovitost** jer plaćamo broj korištenih procesora

⇒ MI PROMATRAMO:

A) ukupno trajanje izvođenja (T)

B) skalabilnost

⇒ DO SADA UPOZNATI NAČINI OCJENE PERFORMANSI:

a) AMDAHLOV ZAKON

↳ najčešći problem: neprimjenjivost u praksi

b) MJERENJE U ZADANOM OKRUŽENJU

↳ najčešći problem: nepotpuna informacija jer nemamo informaciju o skalabilnosti

c) ASIMPTOTSKA ANALIZA

↳ najčešći problemi: zanemarivanje (idealni model (PRAM) ili se mi nalazimo u području gdje zanemarivanje konstanti nije dobro)

⇒ DOBAR MODEL OCJENE PERFORMANSI:

↳ model koji ćemo mi koristit ▽:

↳ komponente:

T_R^i ⇒ trajanje računanja na „i" - tom procesoru

T_K^i ⇒ trajanje komunikacije na „i" - tom procesoru

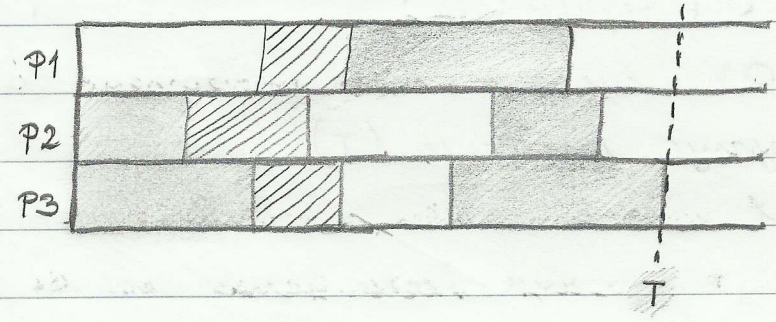
$T_č^i$ ⇒ trajanje čekanja na „i" - tom procesoru

↳ u odnosu na ove 3 komponente definiramo našu osnovnu mjeru:

TRAJANJE IZVOĐENJA: zbroj trajanja računanja, komunikacije i čekanja jednog procesora (odnosno za sve procesore)

⇓

prihvatit da je trajanje izvođenja jednako za sve procesore:



↳ određivanje trajanja izvođenja (T):

- promatramo bilo koji procesor

$$T = T_R^i + T_K^i + T_č^i$$

- promatramo prosjek na svima:

$$T = \frac{1}{P}\left(T_R + T_K + T_č\right)$$

↳ trebamo definirat izraze za T_R, T_K i $T_Č$; a za to trebamo uvest pretpostavke:

- multiračunalo, bez ograničenja propusnosti

TRAJANJE RAČUNANJA (T_R)

↳ trajanje računanja na svim procesorima (mislim da je to suma svih T_R^i)

↳ možemo odrediti na dva načina:

a) kao trajanje slijednog programa

b) kao funkcija veličine problema, broja zadataka i načina raspodjele zadataka po procesorima (OVO MI KORISTIMO!)

TRAJANJE KOMUNIKACIJE (T_K)

↳ trajanje komunikacije na svim procesorima (opet, mislim da je to suma svih T_K^i)

↳ ocjenjuje se trajanjem slanja poruke:

t_s ⇒ trajanje postavljanja poruke

t_w ⇒ trajanje slanja riječi

$$\boxed{T_{S.P.} = t_s + t_w \cdot L}$$

broj riječi u poruci

trajanje slanja poruke

TRAJANJE ČEKANJA ($T_č$)

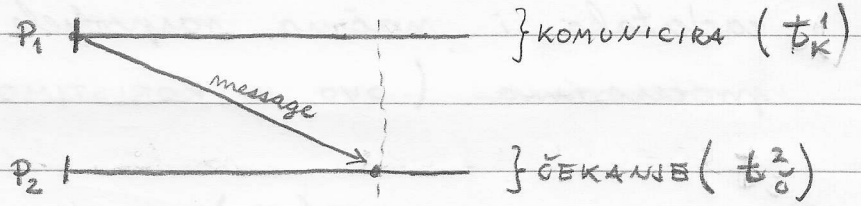
↳ trajanje čekanja na svim procesorima (također, valjda je to zbroj svih $T_č^i$)

↳ najčešći uzroci čekanja:

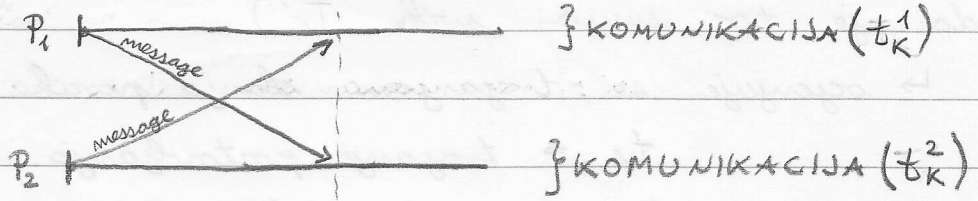
- nema posla → rješenje je uyednačiti opterećenje
- ima posla, al procesor nema podataka za računanje posla → rješenje je „prefatching" (unaprij dohvaćanje podataka u neblokirajućem modu i dotle računaj nešto)

↳ za razumijevanje, pogledajmo primjere:

a) JEDAN ZADATAK ŠALJE DRUGOME PORUKU

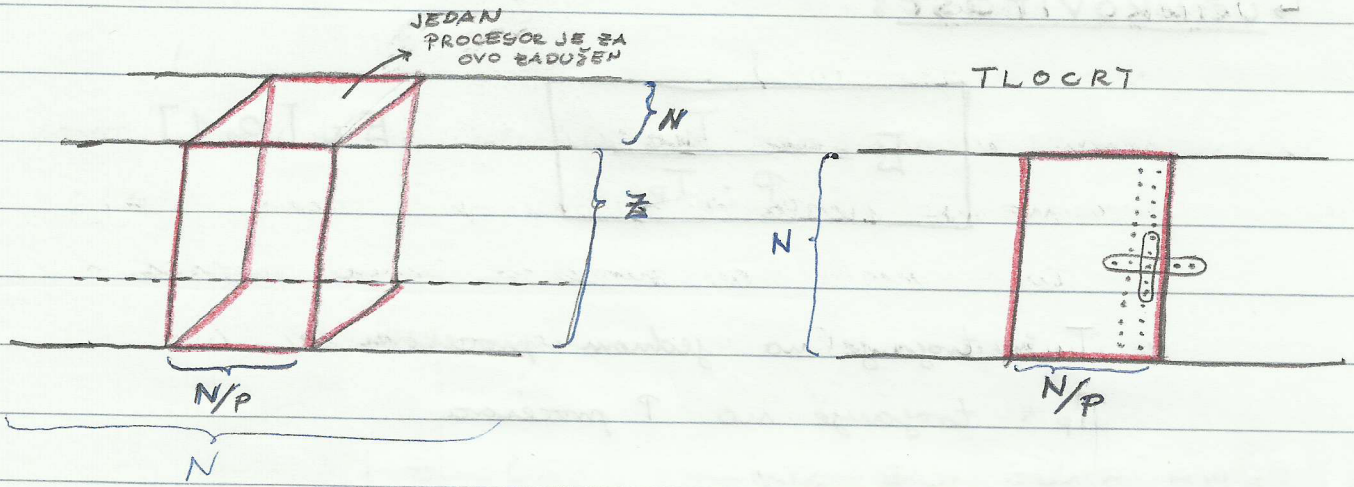


b) MEĐUSOBNO SLANJE



↳ stoga, pazi da ne povećaš komunikaciju i čekanje

PR: Primjenimo mjerenja na 1D atmosferski model



↳ RAČUNANJE: t_c

$$T_R^i = \frac{N}{P} \cdot N \cdot Z \cdot t_c = t_c \cdot \frac{N^2}{P} \cdot Z$$

↳ KOMUNIKACIJA: 2 poruke, $2 \cdot Z \cdot N$ točaka

$$T_K^i = 2 \cdot \left(t_s + t_w \cdot 2NZ \right)$$

↳ ČEKANJE: kako su procesori isti, vrijeme čekanja možemo zanemarit $\left(T_č^i = 0 \right)$

⇒ ukupno trajanje izvođenja:

$$T = \underbrace{t_c \cdot \frac{N^2}{P} \cdot Z}_{\text{KORISNO}} + \underbrace{2t_s + 4t_w NZ}_{\text{BESKORISNO ("OVERHEAD")}}$$

BITNO!

⇒ definiramo dvije dodatne mjere:

↳ **UČINKOVITOST:**

$$E = \frac{T_1}{P \cdot T_P} \qquad E \in [0, 1]$$

T_1: trajanje na jednom procesoru

T_P: trajanje na P procesora

↳ **UBRZANJE:**

$$S = \frac{T_1}{T_P} = P \cdot E \qquad S \in [0, P]$$

⇒ relativne veličine:

↳ u odnosu na isti algoritam (ali slijedni)

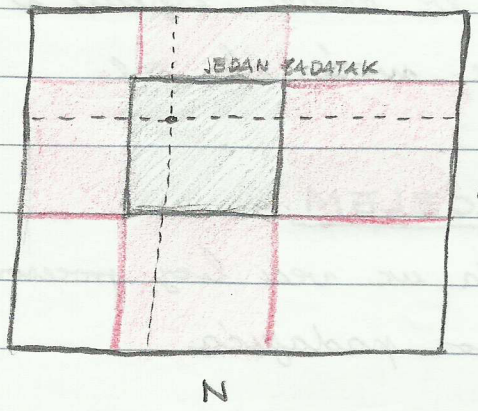
↳ to su ove gore mjere (ovo vodi u labos)

⇒ apsolutne veličine

↳ u odnosu na najbolji slijedni algoritam

↳ to je teško provedivo jer često ne znamo što je najbolji slijedni algoritam

ZAD: U jednom trenutku, paralelnim programom kvadratnu matricu veličine $N \times N$, a trošak jednog množenja je t_c. Matrica je na P radataka podijeljena tako da svaki radatak obrađuje podmatricu jednake veličine. Izvedite trajanje izvođenja i učinkovitost!



$N/\sqrt{P}$ - crveno područje trebamo samnati u svakom procesu

↳ vrijeme računanja: $T_R^i = t_c \cdot N \cdot \frac{N^2}{P}$

↳ vrijeme komunikacije: svima u retku i u stupcu trebamo poslati N^2/P podataka:

- ukupno $2(\sqrt{P}-1)$ poruka

$$T_K^i = 2(\sqrt{P}-1)\left(t_s + t_w \cdot \frac{N^2}{P}\right)$$

↳ vrijeme čekanja: svi rade istom brzinom (ravnomjerno)

⇒ odredimo učinkovitost:

$$E = \frac{T_1}{P \cdot T_P} = \frac{t_c \cdot N^3}{t_c \cdot N^3 + 2(\sqrt{P}-1)(t_s \cdot P + t_w N^2)}$$

ANALIZA SKALABILNOSTI

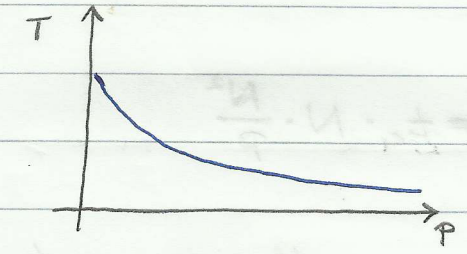
⇒ želimo pratiti kako se paralelna aplikacija ponaša kada dodamo još procesora

↳ to nam služi kako bi znali isplati li se dodati još računalne moći ili ne

A) PONAŠANJE UZ STALNI PROBLEM

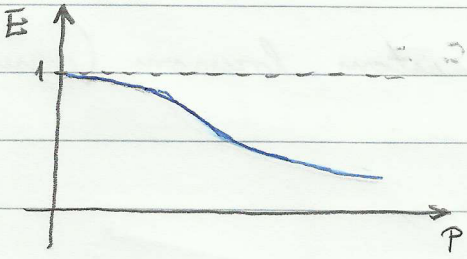
↳ „kako se program ponaša uz veći broj procesora?"

↳ UČINKOVITOST: monotono padajuća



trajanje u idealnom slučaju (može i rast ako je overhead prevelik)





↳ ako plaćamo broj procesora i trajanje tražimo najveći broj procesora uz minimalnu učinkovitost

BITNO!

ZAD: Paralelno računalo plaća se 1 kn po satu po procesoru. Trajanje slijednog programa je 10 dana, a učinkovitost paralelnog programa je $E = \frac{3}{2+P}$. ? Koje je minimalno trajanje koje možemo dobiti za 500 kn?

C ⇒ cijena $\quad C = P \cdot T_P \cdot 1\,kn/h \quad [c] = kn$

$T_1 = 240\,h$

$E_{MIN} = \frac{T_1}{P \cdot T_P} = \frac{T_1}{C_{MAX}} = 0.48$

$E_{MIN} = \frac{3}{2+P_{MAX}} \Rightarrow P_{MAX} = 4.25$

↓

možemo platiti najviše 4 procesora

↳ trajanje uz 4 procesora: $E_{P=4} = 0.5$

$$T_P = \frac{T_1}{P \cdot E_P} = 120\,h$$

⇒ odredimo minimalno trajanje za proizvoljan broj procesora:

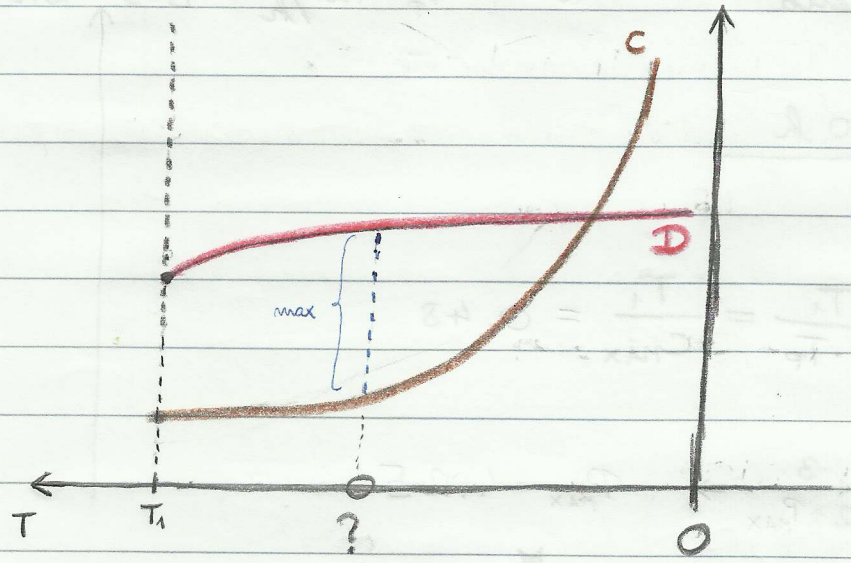
$$T_P = \frac{T_1}{P \cdot E_P} = \frac{T_1(2+P)}{3 \cdot P} = \frac{2}{3} \cdot \frac{T_1}{P} + \frac{T_1}{3}$$

$$\lim_{P \to \infty} T_P = \frac{T_1}{3}$$ //

↳ mi ra izvođenje paralelnog programa plaćamo neku cijenu (C)

- paralelizirano jer je dobit (D) veća nego kao kod izvođenja slijednog programa
- želimo maksimizirati zaradu (Z):

$$Z = D - C$$

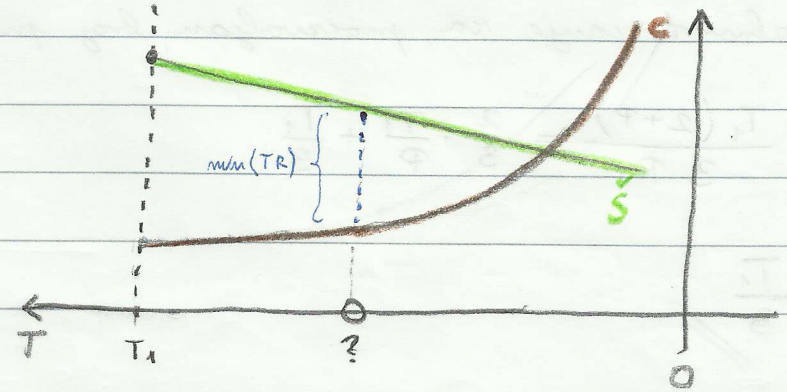


ZANIMA NAS OVO VRIJEME IZVOĐENJA - kada to imamo, imamo i bez procesora

- drugi scenarij je da paraleliziramo jer nam to umanjuje štetu (Š) koja uzrokuje veći ukupni trošak (TR):

$$TR = C + Š$$

- ukupni trošak želimo minimizirati



minimizirani trošak (zbroj cijene i štete)

ZAD: Paralelno računalo plaća se 1 kn/h po procesoru. Na raspolaganju je paralelni program čije se trajanje može izraziti kao $T_p = 50 + \frac{150}{p}$ [h]. Dobit od rezultata opisana je izrazom $D = \max(18(T_1 - T_p), 0)$ [kn]. Koje trajanje donosi najveću moguću zaradu i na koliko procesora?

$$Z = D - C = 18(T_1 - T_p) - p \cdot T_p$$
$$= 18T_1 - 18 \cdot \left(50 + \frac{150}{p}\right) - p \cdot \left(50 + \frac{150}{p}\right)$$
$$= 18T_1 - 1050 - \frac{2700}{p} - 50p$$

$$\frac{\partial Z}{\partial p} = -50 + \frac{2700}{p^2} = 0$$

$$p^2 = \frac{2700}{50} \Rightarrow p' = 7.348$$

↓

zaokružimo na 7 (da je bilo prvo 7.5, zaokružimo na 8)

↳ na 7 procesora, vrijeme trajanja je:

$$T_p = 71.43 \text{ h}$$

B) PROMJENJIVA VELIČINA PROBLEMA

↳ „ kako se mora mijenjati količina posla uz povećanje broja procesora kako bi učinkovitost ostala jednaka ? "

↳ druga (alternativna pitanja):

„ kako treba povećati broj procesora uz veću količinu posla kako bi učinkovitost ostala jednaka ? "

↳ IZOUČINKOVITOST:

- promjena količine posla u ovisnosti o broju procesora uz stabilnu učinkovitost
- izražava se $O()$ notacijom u ovisnosti o broju procesora
  - npr. $O(P^2)$, $O(P^{3/2})$
  - optimalno: $O(P)$ ⇒ koliko smo dodali procesora, toliko trebamo povećat i količinu posla

PR: 1D atmosferski model!

$$T_{1D} = t_c \cdot \frac{N^2}{P} \cdot Z + 2t_s + 4t_w \cdot N \cdot Z$$

$$E = \frac{t_c N^2 \cdot Z}{t_c \cdot N^2 \cdot Z + 2t_s \cdot P + 4t_w \cdot N \cdot Z \cdot P} \sim$$ želimo const (koj N uzeti?)

↳ izjednačimo: $N = P$

$$t_c N^2 Z \sim E(\dots)$$

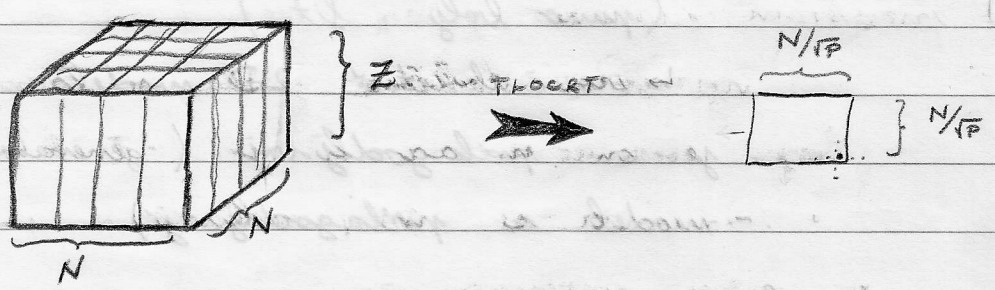
$$t_c P^2 Z \sim E\left(t_c P^2 Z + 2t_s P + 4t_w Z \cdot P^2\right) / : P^2 \Rightarrow t_c Z \sim E\left(t_c Z + \frac{2t_s}{P} + 4t_w Z\right)$$

↳ za prethodni primjer:

· količina posla je u brojniku i ona je razmjerna s N^2, a samim time i s P^2 jer smo uzeli $N = P$.

· stoga, izračunkovstost je kvadratna: $O(P^2)$

PR: 2D atmosferski model!



↳ računanje: $T_R^i = t_c \cdot \frac{N}{\sqrt{P}} \cdot \frac{N}{\sqrt{P}} \cdot Z = t_c \cdot \frac{N^2 \cdot Z}{P}$

↳ komunikacija: 4 poruke (svi susjedi), šaljem i primam samo dva rubna podatka: $Z \cdot \frac{N}{\sqrt{P}} \cdot 2$

$$T_K^i = 4\left(t_s + 2t_w \cdot Z \cdot \frac{N}{\sqrt{P}}\right)$$

⇒ učinkovstost:

$$E = \frac{t_c \cdot N^2 \cdot Z}{P \cdot (T_R^i + T_K^i)} = \frac{t_c \cdot N^2 Z}{t_c \cdot N^2 Z + 4t_s P + 8t_w Z N \sqrt{P}}$$

↳ odredimo složenost na prethodnu primjeru (2D atmosferski model):

- pokažimo $N = P$:

$$t_c P^2 Z \sim E\left(t_c P^2 Z + 4 t_s P + 8 t_w P^{3/2}\right)$$

(nije baš dobar „fit")

- pokažimo $N = \sqrt{P}$:

$$t_c P Z \sim E\left(t_c P Z + 4 t_s P + 8 t_w P Z\right) \quad / : P$$

$$T_R \sim N^2 \sim P$$

(puno bolji „fit")

↳ složenost 2D modela je $O(P)$, pa je on prilagodljiviji (generalno, višedimenzijski modeli su prilagodljiviji)

ZAD: Odredi izoučinkovitost modela za primjer s kvadratnjem matrice!

$$E = \frac{t_c N^3}{t_c N^3 + 2(\sqrt{P}-1)(t_s P + t_w N^2)}$$

↳ <u>pokojmo</u>: $N = \sqrt{P}$

$$E \sim \frac{t_c P^{3/2}}{t_c P^{3/2} + 2 t_s P^{3/2} + 2 t_w P^{3/2}}$$

} −1 u zagradi slobodno zanemarimo

$$T_p \sim N^3 \sim P^{3/2}$$

· izoučinkovitost je $O(P^{3/2})$

PROVJERA IMPLEMENTACIJE

⇒ ovdje se bavimo usporedbom stvarnog i predviđenog ponašanja („a posteriori")

⇒ UZROCI RAZLIKE U PONAŠANJU:

a) netočan ili neprecizan model (najčešće)
b) program nije u skladu s modelom
c) nepouzdani rezultati ili mjerenje

⇒ MOGUĆI NEDOSTACI MODELA: (detaljnije ovo pod a) gore)

a) nejednako opterećenje u radu
b) uvišestručeno računanje
c) nesklad algoritma i programskog alata
d) ne uzima se u obzir ograničenje komunikacijskog kanala (engl. „bandwidth")

PR: Provjera 1D atmosferskog modela?

$$T_R^i = t_c \cdot \frac{N^2 \cdot Z}{P} = t_c \cdot N \cdot Z \cdot \left(\frac{N}{P}\right)$$

ovo nije uvijek cijeli broj i to je uzrok neprecizног modela

↳ stoga, točna formula:

$$t_R^i = t_c \cdot N \cdot Z \cdot \left\lceil \frac{N}{P} \right\rceil$$

↳ VIDI SLAJDOVE ZA SLIKU!

⇒ NEPRAVILNOSTI UBRZANJA:

↳ ANOMALIJA UBRZANJA - program traje kraće od predviđenog, a razlika je veća za veći broj procesora

↳ SUPERLINEARNO UBRZANJE - ubrzanje veće od broja procesora (teoretski nije moguće, ali u praksi jest ☺)

↳ <u>mogući uzroci većih ubrzanja u praksi:</u>

- privremena memorija (CACHE, VIRTUALNA MEMORIJA)
- anomalije pretraživanja (STOHASTIČKI ALGORITMI)

↳ npr. do podrezivanja dođe puno prije kod paralelnog algoritma koji pretražuje u dubinu

MODULARNI PARALELNI ALGORITMI

⇒ paralelni program se promatra kao modul (crna kutija), a više modula se kombinira u paralelnu aplikaciju

↳ PROBLEM: moduli imaju fiksne podjele podataka (npr. prima samo matrice zapisane po stupcima, a ne po retcima što nije kompatibilno s drugim modulom)

↓

stoga, moduli zahtijevaju preraspodjelu podataka prije predaje podataka

⇒ NAPOMENA: ako su moduli takvi da omogućuju dinamičku podjelu podataka, onda se modul može unaprijed podesiti za željenu raspodjelu podataka i to znači da nije potrebno „preslagivanje" koje troši procesorske ciklus

⇒ NAČINI KOMPOZICIJE MODULA:

1) SLIJEDNA KOMPOZICIJA

↳ modul se izvode sljedno, jedan za drugim

↳ prednosti:

- jednostavnost, čitljivost i manja komunikacije među modulima

↳ nedostaci:

- komplicirana podjela podataka, neprilagodljivo broju procesora (neki modul bolje radi na više, a neki na manje procesora, a mi ih tjeramo na isti broj procesora)

2) PARALELNA KOMPOZICIJA

↳ različiti modul izvode se istovremeno na različitim procesorima

↳ prednosti:

- bolja raspodjela opterećenja

↳ nedostaci:

- veći troškovi komunikacije, različito trajanje modula (gubljenje vremena)

3) ZAJEDNIČKA KOMPOZICIJA

↳ različiti moduli izvode istovremeno na istim procesorima

↳ mora omogućit ili sam alat il operacijski sustav

↳ prednosti: podjela podataka neovisna o drugim modulima

↳ nedostaci: vremena konteksta, veći utrošak memorije (svi modul na istom procesoru)

⇒ MODULARNO PROGRAMIRANJE:

↳ modularnost i skrivanje podataka postiže se uporabom KOMUNIKATORA (komunikator definira grupu procesa koji komuniciraju)

↳ mehanizmi komunikatora:

a) KOPIRANJE KOMUNIKATORA

- dobiva se novi kontekst komunikacije jer je poruke nemoguće razjasniti
- funkcija MPI_Comm_dup
- omogućava slijednu kompoziciju

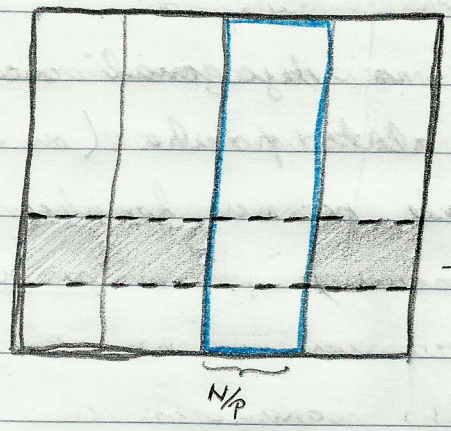
b) DIJELJENJE KOMUNIKATORA

- jedna grupa dijeli se na više podgrupa
- pripadnost istoj podgrupi definira se jednakom vrijednošću parametra „color"
- funkcija MPI_Comm_split
- omogućava paralelnu kompoziciju

ZAD: Paralelni program računa inverznost matrice $N \times N$ tako da transponira matricu, a ratim računa novu inverznost svakog elementa što traje t_c po elementu. Odredi trajanje, učinkovitost i neučinkovitost ako je matrica podijeljena:

a) po stupcima
b) po podmatricama

a)



→ trebat će dobiti ove svoje izračunate elemente

↳ KOMUNIKACIJA: „P-1" poruka (komunikacija sa svima), a veličina poruke je $\left(\frac{N}{P}\right)^2$ elemenata

$$T_K^i = (P-1) \cdot \left(t_s + t_w \cdot \frac{N^2}{P^2}\right)$$

↳ RAČUNANJE: $$T_R^i = t_c \cdot \frac{N}{P} \cdot N = t_c \cdot \frac{N^2}{P}$$

↳ ukupno trajanje: $$T = t_c \cdot \frac{N^2}{P} + (P-1) \cdot \left(t_s + t_w \cdot \frac{N^2}{P^2}\right)$$

↳ učinkovitost: $$E = \frac{t_c \cdot N^2}{t_c \cdot N^2 + (P-1)\left(t_s + t_w \cdot \frac{N^2}{P^2}\right)}$$

OKRENI!

$(P-1) \approx P$

$\downarrow$

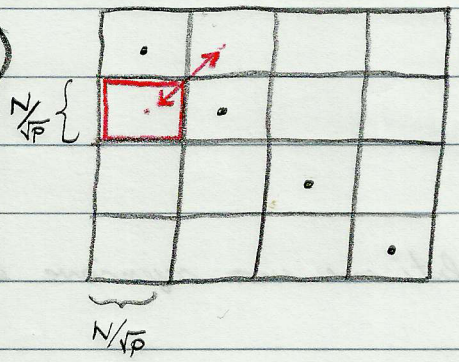
↳ izoučinkovitost: $E \cdot (t_c \cdot N^2) \sim t_c \cdot N^2 + t_s \cdot P^2 + t_w \cdot N^2$

$$N = f(P) = P$$

$$E \cdot (t_c \cdot P^2) \sim t_c \cdot P^2 + t_s \cdot P^2 + t_w \cdot P^2 \quad / : P^2$$

$$T_R \sim N^2 \sim P^2 \Rightarrow \text{izoučinkovitost je } O(P^2)$$

b)



→ element na dijagonali ne treboju slati poruke (oni čekaju na ostale, pa ne komplicirajo sam izračun)

↳ KOMUNIKACIJA: $T_K^i = t_s + t_w \cdot \frac{N^2}{P}$

↳ RAČUNANJE: $T_R^i = t_c \cdot \frac{N^2}{P}$

↳ ukupno trajanje: < analogno >

↳ efikasnost: $E = \frac{t_c \cdot N^2}{t_c \cdot N^2 + t_s \cdot P + t_w \cdot N^2}$

↳ izoefikasnost: $t \sim \frac{t_c \cdot \cancel{P}}{t_c \cdot \cancel{P} + t_s \cdot \cancel{P} \cdot t_w \cancel{P}}$

$N = f(P) = \sqrt{P}$

↳ izoučinkovitost je $O(P)$

PARALELIZACIJA EVOLUCIJSKIH ALGORITAMA

⇒ općenit pseudokod evolucijskog algoritma:

```
DOK NIJE (radovoljen_uvjet_raustavljanja()):
    evaluacija();
    selekcija();
    križanje();
    mutacija();
```

⇒ **paralelni evolucijski algoritmi:**

a) **trivijalno paralelni (TPEA)**

↳ više neovisnih pokretanja evolucijskog algoritma

↳ ovo je najučinkovitije

b) **raspodjeljeni (REA / DEA)**

↳ populacija je raspodjeljena na subpopulacije

(drugi naziv za subpopulaciju je „otok" il „deme")

↳ svaki otok evoluira zasebno



↳ uvodi se operator migracije kako bi genetski materijal jedne subpopulacije prešao u drugu (razmjena jedinki između otoka)

↳ parametri operatora migracije:

1) MIGRACIJSKI INTERVAL:

- broj iteracija između migracija
- može biti stalan ili ga možemo mijenjati u ovisnosti na stanje populacije (npr. prošlo je 100 generacija bez promjene najboljeg rješenja), a čak može biti i „random"
- migracija može biti <u>sinkrona</u> (svi u isto vrijeme razmjenjuju jedinke, pa je interval rasporedu za sve otoke) ili <u>asinkrona</u> (u različita vremena se jedinke šalju, pa svaki otok ima svoj migracijski interval)
- najčešće želimo da razmjena ne bude često

2) MIGRACIJSKA STOPA:

- broj jedinki koje migriraju
- obično malo (1 ili 2)

3) ODABIR JEDINKI ZA SLANJE:

- obično šaljemo bolje jedinke, pa koristimo operatore selekcije ra ovo

4) ODABIR JEDINKI ZA ZAMJENU:

- obično se biraju lošije jedinke (opet, operator eliminacijske selekcije)

5) TOPOLOGIJA RAZMJENE:

- tko s kim mijenja jedinke
- postoje razne topologije (prstenasta, kružna, ljestvičasta, dvoslojna / dvorazinska)

↳ REA je najpopularniji oblik paralelnog genetskog algoritma

↳ prednosti ovog modela: jednostavnost, prevencija lokalne konvergencije zbog raspodjeljenosti, mogućnost superlinearnog ubrzanja

↳ nedostaci ovog modela: puno novih parametara

↳ primjeti da je ovdje podjela **podatkovna** i **krupnozrnata**

c) masovno-paralelni (MPEA)

↳ rubni slučaj REA gdje jedan otok predstavlja jednu jedinku

↳ takva sitnozrnata podjela ima smisla na masovno-paralelnom računalu

↳ primjena selekcije i genetskih operatora definira se topologijom procesora

↳ danas je sličan popularan model: stanični EA („cellular EA")

d) globalno-paralelni (GPEA)

↳ više zadataka djeluje na jednoj zajedničkoj populaciji

↳ uobičajeno je: proces voditelj + radnici gdje voditelj ažurira populaciju

↳ podjela posla može biti:

1) voditelj: selekcija i operatori
radnici: evaluacija

2) voditelj: selekcija
radnici: evaluacija i operatori

↳ dvije varijante GPEA:

a) SINKRONI - voditelj čeka da svi radnici
završe svoj posao
- ima jednaka svojstva kao
slijedni evolucijski algoritam

b) ASINKRON - voditelj NE čeka na
sve radnike nego može napraviti
neke operacije sa starim
fitnessima kako nebi bio
„idle" predugo
- pristup podacima tada
nije međusobno isključiv

↳ prednosti ovog modela: nema dodatnih
parametara

↳ nedostaci ovog modela: loš je u raspodijeljenoj
okolini (zbog stvornatosti)

↳ zadnje je podjela računanja i ona je stvornata

c) hijerarhijski (HPEA)

↳ kombinacija nekih do sada nabrojanih varijanti

↳ može biti vrlo učinkovito jer se prilagođava
arhitekturama, ali može imati puno parametara

↳ npr. REA / GPEA

ZAD: Imamo sljedeći pseudokod:

VODITELJ

```
ZA (sve_radnike)
    pošalji N/R jedinki;
ZA (sve_radnike)
    primi N/R jedinki;
```

RADNIK

```
primi jedinke;
evaluiraj;
pošalji jedinke;
```

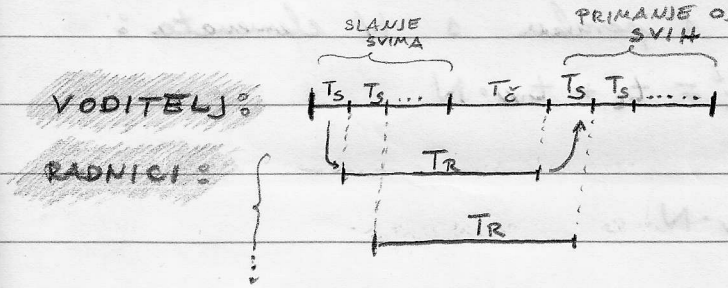
Dani su i sljedeći parametri:

$N \rightarrow$ veličina populacije $R \rightarrow$ broj radnika

$P = R + 1$ $t_c \rightarrow$ evaluacija 1 jedinke

$t_s, t_w \rightarrow$ slanje $t_c \gg t_s, t_w$

Odredi trajanje jedne evaluacije i optimalan broj radnika.



$$T_s = t_s + t_w \cdot \frac{N}{R}$$

$$T_R = t_c \cdot \frac{N}{R}$$

VODITELJ: $T = R \cdot T_s \cdot 2 + T_č$

$$T_č = \underbrace{T_R + T_s - R \cdot T_s}_{\text{IZ SLIKE}} = T_R + (1-R) \cdot T_s$$

$$T = 2RT_s + T_R + (1-R)T_s$$

$$T = t_c \cdot \frac{N}{R} + (R+1)\left(t_s + t_w \cdot \frac{N}{R}\right)$$

↳ ubrzanje: $S = \dfrac{t_c \cdot N}{t_c \cdot \frac{N}{R} + (1+R) \cdot T_s} = \dfrac{t_c \cdot N}{t_c(P-1) + P \cdot T_s}$

↳ optimalan broj radnika:

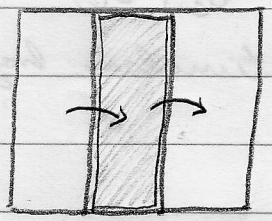
$$\frac{\partial T}{\partial R} = 0 \Longrightarrow R = \sqrt{\frac{N(t_c + t_w)}{t_s}}$$

ZAD: Paralelni algoritam računa svaki element matrice uz pomoć neposrednih susjeda „gore" i „lijevo" (matrica ima „spojene" rubove). Trošak računanja jednog elementa iznosi t_c. Odredi trajanje operacije na P radnatka te učinkovitost i ubrzanje ako je matrica podijeljena:

a) po stupcima!

b) po podmatricama! (matrica je $N \times N$)

a) $T_R^i = t_c \cdot \frac{N^2}{P}$



⇒ šaljem desnom, primam od lijevog ukupno 1 poruku s N elemenata:

$T_K^i = t_s + t_w \cdot N$

$$T = t_c \cdot \frac{N^2}{P} + t_c + t_w \cdot N$$

$$E = \frac{t_c \cdot N^2}{t_c \cdot N^2 + t_s \cdot P + t_w \cdot N \cdot P}$$

↳ za ubrzanje probamo sa $N = f(P) = P$

$$E \sim \frac{t_c \cdot P^2}{t_c \cdot P^2 + t_s \cdot P + t_w \cdot P^2} \quad / : P^2$$

$$E \sim \frac{t_c}{t_c + \underbrace{\frac{t_s}{P}}_{\text{NEŠTO S OVIM RADIMO (ZANEMARUJEMO ☺)}} + t_w} \Rightarrow T_R \sim N^2 \sim P^2$$

↳ ubrzanje je $O(P^2)$

b) ↳ dvije poruke: $T_K^i = 2 \cdot \left(t_s + t_w \cdot \frac{N}{\sqrt{P}}\right)$

↳ računanje: $T_R^i = t_c \left(\frac{N}{\sqrt{P}}\right)^2 = t_c \cdot \frac{N^2}{P}$

$$T = T_R^i + T_K^i$$

$$E = \frac{t_c \cdot N^2}{t_c \cdot N^2 + 2 t_s \cdot P + 2 \cdot t_w \cdot N \cdot \sqrt{P}}$$

↳ pokušajmo sa $N = \sqrt{P}$:

$$E \sim \frac{t_c \cdot P}{t_c \cdot P + 2 t_s \cdot P + 2 \cdot t_w \cdot P} \quad / : P$$

↳ izoučinkovitost je $O(P)$