Efektívne paralelné algoritmy
- zbierka riešených príkladov

Veronika Vlnková

2008

Efektívne paralelné algoritmy - zbierka riešených príkladov

DIPLOMOVÁ PRÁCA

Veronika Vlnková

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY
KATEDRA INFORMATIKY

Informatika

Doc. RNDr. Rastislav Královič, PhD.

BRATISLAVA 2008

Zadanie

Cieľom diplomovej práce je pripraviť zbierku príkladov k predmetu Efektívne paralelné algoritmy.

V prvej fáze je treba zozbierať príklady z rôznych zdrojov (učebnice, webové stránky podobných kurzov a pod.) a roztriediť ich podľa podobnosti a náročnosti techník, potrebných na ich riešenie. V druhej fáze treba ku každej skupine príkladov pripraviť vysvetlenie použitých techník, vymyslieť (modifikovať existujúce) príklady s rôznou náročnosťou, používajúce tieto techniky a navrhnúť príkladom riešenia.

Čestne prehlasujem, že som túto prácu písala samostatne s použitím uvedenej odbornej literatúry.

V Bratislave, 7. mája 2008

_____

Veronika Vlnková

Ďakujem svojmu diplomovému vedúcemu za poskytnuté materiály a cenné rady a pripomienky, docentovi Hudecovi z Fakulty informatiky a informačných technológií STU za poskytnutie zaujímavej literatúry a v neposlednom rade ďakujem svojej rodine za ich podporu pri písaní tejto práce, ako aj počas celého štúdia.

Abstrakt

Práca predstavuje zbierku príkladov z návrhu paralelných algoritmov. V úvode predstavíme používané modely PRAMu a krátko spomenieme používané miery zložitosti. V daľších kapitolách predložíme rôzne typy príkladov zoskupené podľa techník či štruktúr potrebných na ich vyriešenie.

V prvej kapitole sa venujeme príkladom, ktoré sa dajú efektívne vyriešiť použitím algoritmu na výpočet prefixových súm, pričom postupujeme od najjednoduchších, priamočiarych modifikácií algoritmu k zložitejším a menej zjavným spôsobom použitia.

V druhej kapitole uvádzame príklady, v ktorých základom riešenia je počítanie logických operácií (and, or, pozícia prvej jednotky...) na vhodne zvolenom boolovskom poli.

V tretej kapitole sú zaradené algoritmy, majúce na vstupe štruktúru spájaný zoznam. Časť kapitoly sa zaoberá riešením problémov priamo na tejto štruktúre, časť využíva algoritmus list ranking, pomocou ktorej vieme prerobiť spájaný zoznam na pole a pracovať ďalej na ňom.

V štvrtej kapitole sa venujeme rôznym triediacim algoritmom.

V piatej kapitole riešime úlohy z planárnej geometrie s využitím techniky divide & conquer.

Kľúčové slová: paralelné algoritmy, PRAM, zbierka

Obsah

Úvod

Existuje množstvo modelov slúžiacich na implementáciu paralelných algoritmov. My budeme v našej zbierke používať model parallel random access machine - PRAM.

PRAM je synchrónny shared memory model. Viacero procesorov v ňom môže pristupovať ku spoločnej pamäti, cez ktorú môžu komunikovať výmenou dát. To, že je model synchrónny znamená, že všetky procesory sú riadené spoločnými hodinami, pričom v každom takte má procesor povolené vykonať inštrukciu alebo ostať idle.

Rozlišujeme niekoľko variácií modelu PRAM v závislosti od toho, ako sa pristupuje k spoločnej pamäti:

- **EREW PRAM** (exclusive read exclusive write) – do jedného registra nemôže naraz pristupovať viacero procesorov
- **CREW PRAM** (concurrent read exclusive write) – povoľuje viacnásobný prístup len na čítanie
- **CRCW PRAM** (concurrent read concurrent write) – povoľuje viacnásobný prístup na čítanie aj na zápis. V závislosti od toho, ako narábame s viacnásobným zápisom, rozdeľujeme CRCW PRAM na tri typy:
  - **common –** povoľuje súčasný zápis, len ak sa všetky procesory snažia zapísať rovnakú hodnotu
  - **priority** – povolí zápis procesoru s najnižším indexom
  - **arbitrary** – uspeje náhodný procesor, pri tomto type PRAM môže nastať situácia, že pri opakovanom výpočte na rovnakom vstupe dostaneme rôzne výstupy.

Výhody modelu PRAM môžeme zhrnúť v nasledovných bodoch:

Je vyvinutých množstvo techník a metód, ako pristupovať k rôznym triedam výpočtových problémov.

Odstraňuje algoritmické detaily, týkajúce sa synchronizácie a komunikácie a pri návrhu algoritmov nám dovoľuje lepšie sa sústrediť na štruktúru problému.

Model zahŕňa explicitné porozumenie vykonania operácií v jednotlivých časových jednotkách a alokácie procesorov, ktoré prácu vykonávajú.

Návrhové paradigmy modelu PRAM sa ukazujú byť robustné a napríklad mnohé sieťové algoritmy možno odvodiť priamo z algoritmov na PRAM.

Dôležitou vlastnosťou modelu PRAM oproti iným je, že poskytuje prirodzené rozšírenie klasického sekvenčného modelu.

1. Prefixové sumy

Príklad 1.1 (prefixové sumy)

*Uvažujme n-prvkovú postupnosť $\{x_1, x_2, \ldots, x_n\}$ prvkov množiny S a binárnu asociatívnu operáciu *. Prefixové sumy tejto postupnosti sú čiastočné súčty (súčiny) definované takto*

$$s_i = x_1 * x_2 * \ldots * x_i, \quad 1 \leq i \leq n$$

Napíšte EREW PRAM algoritmus počítajúci prefixové sumy. Akú prácu si bude vyžadovať a aká bude časová zložitosť?

Riešenie

Triviálny sekvenčný algoritmus, ktorý by sumu s_i počítal ako súčet s_{i-1} a x_i je inherentne sekvenčný.

Na paralelný výpočet prefixových súm použijeme binárny vyvážený strom. Vstupné pole si uložíme do jeho listov a prechádzame hore po strome tak, aby sa nám v každom vrchole nachádzal súčet jeho synov.

Prefixová suma párneho listu i bude hodnota v koreni podstromu takého, že i je jeho posledný list. Prefixová suma nepárneho listu i bude súčet hodnoty v koreni podstromu, ktorého posledným listom je ľavý sused i, a hodnoty i. Prefixová suma prvého listu je jeho vlastná hodnota.

Algoritmus 1.1

Vstup: Pole X = (x_1, x_2, ... , $x_{n/2}$)

Výstup: Pole S = (s_1, s_2, ... , $s_{n/2}$)

```
begin
1: if n = 1 then {set s1: = x1; exit}
```

2: for $1 \leq i \leq n/2$ pardo

set $y_i := x_{2i-1} * x_{2i}$

3: *Rekurzívne vyrátaj prefixové sumy z* $\{y_1, y_2, \ldots, y_{n/2}\}$ *a ulož ich do* $\{z_1, z_2, \ldots, z_{n/2}\}$.

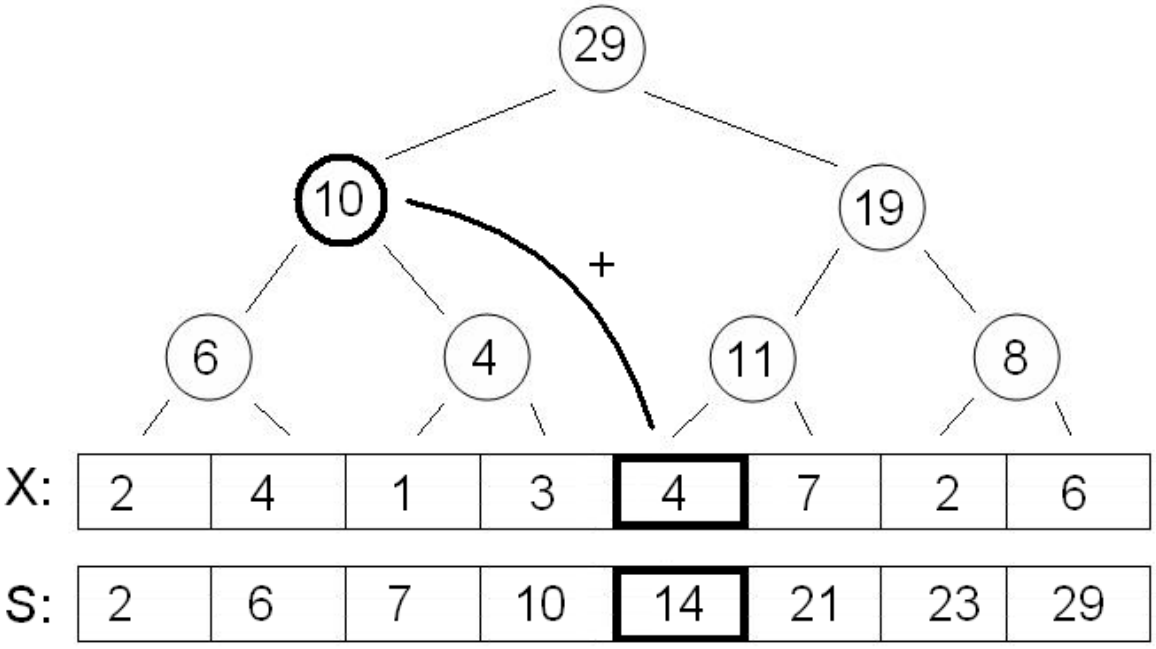
4: for $1 \leq i \leq n$ pardo

{i *párne* : set $s_i := z_{i/2}$

i = 1 : set $s_1 := x_1$

i *nepárne* > 1 : set $s_i := z_{(i-1)/2} * x_i$}

end



Algoritmus pracuje v čase T(n) = O(log n), používajúc celkovo W(n) = O(n).

Viac o tomto algoritme a o jeho modifikáciach (napr. nerekurzívny výpočet) nájdeme v [1].

Príklad 1.2 (prefixové minimá)

Máme pole $A = (a_1, a_2, \ldots, a_n)$. Vypočítajte prefixové minimá na poli A. Teda pre každé i, $1 \leq i \leq n$, nájdite minimálny prvok postupnosti $\{a_1, a_2, \ldots, a_i\}$. Algoritmus by mal pracovať v čase O(log n) použijúc O(n) operácií na EREW PRAM.

Riešenie

Modifikujeme algoritmus na počítanie prefixových súm tak, že namiesto binárnej asociatívnej operácie * vyhodnotíme funkciu rátajúcu minimum dvoch prvkov.

Algoritmus Min

Vstup: Prvky x, y

Výstup: Minimum týchto dvoch prvkov

begin

1: if x < y then set Min: = x

   else set Min: = y

end

Algoritmus 1.2

Vstup: Pole $X = (x_1, x_2, \ldots, x_{n/2})$

Výstup: Pole $S = (s_1, s_2, \ldots, s_{n/2})$

begin

1: if n = 1 then {set s_1: = x_1; exit}

2: for $1 \leq i \leq n/2$ pardo

   set y_i: = Min(x_{2i-1}, x_{2i})

3: *Rekurzívne vyrátaj prefixové sumy z* $\{y_1, y_2, \ldots, y_{n/2}\}$ *a ulož ich do* $\{z_1, z_2, \ldots, z_{n/2}\}$.

```
4:  for 1 ≤ i ≤ n pardo
      {i párne        : set s_i: = z_{i/2}
       i = 1          : set s_1: = x_1
       i nepárne > 1  : set s_i: = Min(z_{(i - 1)/2}, x_i)}
end
```

Keďže minimum dvoch prvkov vieme zrátať s jedným procesorom v konštantnom čase, celková zložitosť vonkajšieho algoritmu nám zostane zachovaná a rovnako ako algoritmus 1.1, aj tento bude mať zložitosť T(n) = O(log n), W(n) = O(n).

Príklad 1.3 (sufixové sumy)

*Uvažujme n-prvkovú postupnosť {$x_1, x_2, \ldots, x_n$} prvkov množiny S a binárnu asociatívnu operáciu *. Sufixové sumy tejto postupnosti sú čiastočné súčty (súčiny) definované takto*

$$s_i = x_i * x_{i+1} * \ldots * x_n, \quad 1 \le i \le n$$

Ako by ste vyrátali sufixové sumy na EREW PRAM?

Riešenie

Sufixové sumy môžeme rátať rovnako ako prefixové, len otočíme vstup.

Príklad 1.4

V poli A máme uložených n prvkov. V poli L takom, že $L(i) \in \{1, 2, \ldots, k\}$ *, máme uložené zafarbenie prvku A(i), 1 ≤ i ≤ n. Napíšte optimálny EREW PRAM algoritmus pracujúci v čase O(log n), ktorý vypočíta poradie prvku vrámci svojej farby.*

Riešenie

Pole A si rozdelíme na úseky dĺžky log n. Na jednotlivých úsekoch si sekvenčne spočítame poradie prvku vrámci jeho farby na úseku a to nasledovne: Zoberieme si pomocné dvojrozmerné pole B, ktoré sme na začiatku inicializovali na nuly. Stĺpce poľa budú reprezentovať jednotlivé úseky, riadky jednotlivé farby. Políčko A(i) sa pozrie do poľa B na políčko prislúchajúce jeho úseku a farbe, k jeho hodnote pripočíta jedna. Tento výsledok si navyše zapamätá – označuje jeho poradie vrámci farby na úseku, túto hodnotu ešte ďalej využijeme.

Na riadkoch B máme teraz počet výskytov príslušnej farby v jednotlivých úsekoch. Spočítam si na nich prefixové sumy. Týmto sme si vypočítali počet výskytov príslušnej farby od začiatku poľa po koniec príslušného úseku.

Teraz si už vieme ľahko vypočítať poradie A(i) vrámci svojej farby na celom poli. A(i) si načíta hodnotu z B prislúchajúcu úseku pred ním a pripočíta ju ku svojmu poradiu vrámci farby na úseku. Aby sme predišli konfliktom s viacnásobným načítaním, aj tento krok spravíme vrámci úsekov sekvenčne.

Príklad 1.5

Máme dané celé čísla x a n, n > 0. Vypočítajte x^i pre všetky i, $1 \leq i \leq n$. Algoritmus by mal pracovať v čase O(log n) pri práci O(n) na EREW PRAM.

Riešenie

Vytvoríme si pole A dĺžky n a zinicializujeme ho na hodnoty x. Spočítame na ňom prefixové sumy s operáciou násobenie.

Algoritmus 1.5

Vstup: Pole X = (x_1, x_2, ... , $x_{n/2}$)

Výstup: Pole S = (s_1, s_2, ... , $s_{n/2}$), v ktorom $s_i = x^i$

```
begin
1:  for 1 ≤ i ≤ n pardo set x_i: = x
2: if n = 1 then {set s_1: = x_1; exit}
3: for 1 ≤ i ≤ n/2 pardo
      set  y_i: = x_2i - 1 * x_2i
4: Rekurzívne vyrátaj prefixové sumy z {y_1, y_2, ... , y_n/2}
a ulož ich do {z_1, z_2, ... , z_n/2}.
5:  for 1 ≤ i ≤ n pardo
       {i párne          : set s_i: = z_i/2
        i = 1            : set s_1: = x_1
        i nepárne > 1    : set s_i: = z_(i - 1)/2 * x_i}
end
```

Príklad 1.6

Napíšte algoritmus pracujúci v čase O(log n log m), ktorý vypočíta A^i, pre všetky $1 \le i \le m$, kde A je matica nxn. Akú prácu budeme potrebovať?

Riešenie

Úlohu vyriešime celkom jednoducho pomocou prefixových súm. Využijeme trik, ktorý sme použili v predchádzajúcom príklade. Do pomocného poľa si m-krát vložíme maticu A a zrátame na ňom prefixové sumy, pričom ako binárnu asociatívnu operáciu použijeme násobenie matíc.

Zložitosť algoritmu na počítanie prefixových súm na vstupnom poli dĺžky m je $T(m) = O(\log m)$, $W(m) = O(m)$. Operáciu *, o ktorej predpokladáme, že je na PRAM primitívna, nahrádzam algoritmom, ktorého zložitosť je $T(n) = O(\log n)$, $W(n) = O(n^3)$. Výsledná zložitosť teda naozaj bude $T(n\ m) = O(\log n \log m)$ a budeme potrebovať prácu $W(n\ m) = O(n^3 m)$.

Súčin dvoch matíc

Súčin AB, kde A a B sú matice nxn, vypočítame nasledovne:

Algoritmus ráta podľa nasledujúceho vzorca

$$c_{i,j} = \sum_{l=1}^{n} a_{i,l} b_{l,j} \quad .$$

Najprv v jednom kroku vynásobíme všetky dvojice l-tý prvok riadku a l-tý prvok stĺpca. Následne tieto súčiny zosumujeme na binárnom vyváženom strome.

Algoritmus beží v čase O(log n) používajúc $O(n^3)$ operácií. Nie je potrebné robiť žiadne simultánne zápisy, ani čítania a tak na výpočet môžeme použiť aj EREW PRAM.

Algoritmus Súčin dvoch matíc

```
Vstup: Matice A, B uložené v dvojrozmerných poliach nxn
Výstup: Matica C
begin
1: for 1 ≤ i, j, l ≤ n pardo
     set C'(i,j,l): = A(i,l) * B(l,j)
2: for h = 1 to log n pardo
     for 1 ≤ i, j ≤ n, 1 ≤ l ≤ n/2^h pardo
       set C'(i,j,l): = C'(i,j,2l - 1) + C'(i,j,2l)
3: for 1 ≤ i, j ≤ n pardo
     set C(i,j): = C'(i,j,1)
end
```

1. Prefixové sumy

Príklad 1.7 (*segmentové prefixové sumy)*

Máme pole $A = (a_1, a_2, \ldots, a_n)$ *celých čísel a boolovské pole B dĺžky n také, že* $b_1 = b_n = 1$. *Vypočítajte prefixové sumy pre každú sekvenciu* $(a_{i+1}, \ldots, a_j)$, $i \leq j$, *takú, že* $b_i = b_j = 1$ *a* $b_k = 0$ *pre všetky* $i \leq k \leq j$. *Algoritmus by mal pracovať na EREW PRAM v čase O(log n) a práci O(n).*

Riešenie

Každé políčko si zistí, či je na začiatku, na konci alebo vo vnútri segmentu a to tak, že sa pozrie na zhodné políčko v poli B a jeho ľavého suseda, či sa na nich nachádza jednotka alebo nula. Ak sa na b_i nachádza nula a na b_{i-1} jednotka, znamená to, že a_i je na začiatku segmentu. Ak sa na b_i nachádza nula a na b_{i-1} nula, znamená to, že a_i je vo vnútri segmentu. Ak sa na b_i nachádza jednotka a na b_{i-1} nula, a_i je na konci segmentu. Môže nastať aj situácia, že sa na b_i aj b_{i-1} budú nachádzať jednotky, toto sú jednoprvkové segmenty, ktorých hodnota zostáva zachovaná.

Prefixové sumy na segmentoch začneme rátať rovnako ako prefixové sumy na celom poli, s tým rozdielom, že keď sa majú spojiť dva uzly, z ktorých ľavý označuje koniec segmentu a pravý začiatok, tak ku spojeniu nedôjde. Nad poľom tak nebudeme stavať jeden vyvážený binárny strom, ale stromov niekoľko. Každý zodpovedajúci jednému segmentu.

Keďže staviame vyvážené binárne stromy s listami v poli dĺžky n, práca O(n) bude zachovaná. Rovnako výška žiadneho z nich nemôže byť väčšia ako log n, takže aj čas O(log n) zachováme.

Príklad 1.8

Máme pole $A = (a_1, a_2, \ldots, a_n)$ *celých čísel a boolovské pole B dĺžky n také, že* $b_1 = b_n = 1$. *Vypočítajte prefixové minimá pre každú sekvenciu* $(a_i, a_{i+1}, \ldots, a_j)$, $i \leq j$, *takú, že* $b_1 = b_n = 1$ *a* $b_k = 0$ *pre všetky* $i \leq k \leq j$.

Riešenie

Tento problém môžeme vyriešiť pomocou algoritmu na výpočet segmentových prefixových súm, ktorý modifikujeme podobne ako sme v príklade 1.7 modifikovali algoritmus na výpočet prefixových súm - namiesto binárnej asociatívnej operácie * vyhodnotíme funkciu rátajúcu minimum dvoch prvkov.

Príklad 1.9

Máme dané pole ľavých a pravých zátvoriek. Napíšte EREW PRAM algoritmus, ktorý v čase O(log n) pri lineárnej práci zistí, či ide o dobre uzátvorkovaný výraz.

Riešenie

Tento problém by sme štandardne riešili pomocou stacku, do ktorého vkladáme pri výskyte ľavej zátvorky, vyberáme pri výskyte pravej. Tento prístup sa môže zdať inherentne sekvenčný. Nie však keď stack nasimulujeme pomocou algoritmu na počítanie prefixových súm a to nasledovne: Za každú ľavú zátvorku do prefixovej sumy pripočítam 1, za pravú odpočítame 1.

V dobre uzátvorkovanom výraze sa nám nesmie vyskytnúť viac pravých zátvoriek, ako už bolo ľavých, preto žiaden prefix nemôže vyjsť záporný (nemohli sme zo stacku vyberať, keď tam nič nebolo). Okrem toho musí byť rovnaký počet ľavých aj pravých zátvoriek – celková suma teda musí vyjsť 0 (stack na konci musí byť prázdny).

```
Algoritmus 1.9
Vstup: Pole A
Výstup: Boolovská hodnota v D(1)
begin
1: for i = 1 to n pardo
     if A(i) = '(' then set B(i): = 1
```

```
        else set B(i):= - 1
2: Zrátaj prefixové sumy na poli B a ulož ich do poľa C
3: for i = 1 to n pardo
        if C(i) ≥ 0 then set D(i): = 1
        else set D(i): = 0
4: for h = 1 to log n pardo
        for 1 ≤ i ≤ n/2^h pardo
          set D(i): = D(2i - 1) & D(2i)
5: if C(n) ≠ 0 then set D(1): = 0
end
```

V prvom kroku si inicializujeme pomocné pole B. Tento krok je triviálny, zjavne sa neprekročí povolená práca a vykoná sa jednom takte.

Prefixové sumy na pomocnom poli B vieme zrátať v čase O(log n) s O(n) procesormi.

V treťom a štvrtom kroku overujeme, či nám niektorý prefix nevyšiel záporný. Do pomocného poľa D si pre každý prefix zapíšeme, či je jeho hodnota nezáporná. Na binárnom vyváženom strome vypočítame súčin (AND) všetkých jeho prvkov tak, že výsledok súčinu budeme mať na prvej pozícii. Naplnenie poľa D spraví n procesorov v jednom takte. Výpočet súčinu na binárnom vyváženom strome v čase O(log n).

V poslednom kroku overíme, či bol počet pravých aj ľavých zátvoriek rovnaký, teda či výsledná suma, pri počítaní prefixových súm, vyšla 0. Toto overenie nám vie vykonať na jeden krok jeden procesor.

Príklad 1.10

Nech A je pole dĺžky n, ktoré obsahuje iba čísla 0 alebo 1. Je dané číslo k < n. Nájdite EREW PRAM algoritmus pracujúci v čase O(log k) pri práci O(n), ktorý vypočíta

1. Prefixové sumy

pole B dĺžky n – k, pre ktoré platí

$$B(i)=1 \Leftrightarrow (\forall\, j \in \{i, ..., i+k\})\, A(j)=1 \quad .$$

Riešenie

Zadanie nám vlastne hovorí, že chceme vypočítať pole B, ktoré bude mať na i-tej pozícii jednotku práve vtedy, keď na tej istej pozícii a k pozíciach za ňou boli na vstupnom poli A samé jednotky. Pozície {i, i + 1, …, i + k} tvoria súvislý úsek, čo využijeme.

Rozdeľme si pole A na úseky dĺžky k. (Ak by sme chceli byť dôslední, môžeme posledný úsek poprípade doplniť nulami. Ide však len o technický detail.) Na každom takomto podpoli vypočítame prefixové a sufixové sumy s operáciou násobenia. Výsledky si uložíme do pomocných polí C (prefixové sumy) a C' (sufixové sumy).

Čo nám tieto sumy hovoria? Ak daná prefixová (sufixová) suma vyšla jednotka, znamená to, že všetky prvky príslušného intervalu boli jednotky. Ak by sa na intervale vyskytla čo i len jedna nula, súčin by musel byť nula.

Teraz už môžeme overiť, či sa nám na pozíciách {i, i + 1, ... , i + k} nachádzali samé jednotky. Interval {i, i + 1, ... , i + k} je dlhý k + 1, teda sa rozkladá práve cez dva naše k-prvkové úseky a tvorí nejaký sufix na prvom úseku a nejaký prefix na druhom. B(i) teda určíme vyšetrením dvoch hodnôt – sufixovej sumy na i-tej pozícii a prefixovej sumy na (i + k)-tej. Ak sú obe jednotky, museli byť aj na intervale {i, i + 1, ... , i + k} samé jednotky.

Zložitosť: Výpočet prefixových súm má zložitosť T(n) = log n, W(n) = n. Keďže pracujeme s úsekmi dĺžky k, zložitosť pre jednotlivé úseky bude T(k) = log k, W(k) = k. Paralelným výpočtom na všetkých úsekoch súčasne sa nám práca zvýši na n, časová zložitosť ostane zachovaná.

Overenie príslušných prefixových a sufixových súm vieme spraviť v niekoľkých krokoch, teda v konštantnom čase. Na výpočet jedného B(i) využijeme jeden procesor, celková práca v tomto kroku bude teda W(n) = n.

1. Prefixové sumy

Príklad 1.11

Nech A je ľubovoľné pole n prvkov z lineárne usporiadanej množiny M a x prvok množiny S. Ako určíme rank(x : A) na EREW PRAM v čase O(log n) a práci O(n)?

Riešenie

Rank(x : A) nám určuje počet prvkov poľa A, od ktorých je x väčšie. Porovnáme teda každý prvok z A s x a do pomocného poľa B si na príslušnú pozíciu zapíšeme výsledok porovnania. Ak bolo x väčšie, tak jednotku, inak nulu. Prvky poľa B spočítame.

Algoritmus 1.11

```
Vstup: A, x
Výstup: B(1)
begin
1: for 1 ≤ i ≤ n pardo
     if x > A(i) then set B(i): = 1
     else set B(i): = 0
2: for h = 1 to log n pardo
     for 1 ≤ i ≤ n/2^h pardo
       set B(i): = B(2i - 1) + B(2i)
end
```

Príklad 1.12

Predpokladajme, že A je usporiadané. Ako rýchlo vieme zistiť rank(x : A) s prácou O(n) v tomto prípade?

Riešenie

V konštantnom čase. Opäť porovnáme každý prvok poľa A s x. Ak bolo A(i) menšie, porovnáme x s jeho pravým susedom. Ak ten je od x väčší, rank(x : A) je i.

```
Algoritmus 1.12
Vstup: A, x
Výstup: r
begin
1: for 1 ≤ i ≤ n pardo
     if x > A(i) then
        if x < A(i + 1) then set r: = i
end
```

2. Logické operácie na boolovských poliach

Príklad 2.1

Máme n-prvkové pole A, v ktorom všetky prvky sú rôzne. Napíšte common CRCW PRAM algoritmus pracujúci v čase 0(1) pri práci $O(n^2)$, ktorý vypočíta maximum poľa.

Riešenie 1

Ukážeme si najprv algoritmus tak, ako bol odprezentovaný v [1].

Každé políčko A(i) porovnáme so všetkými prvkami poľa A a výsledok porovnania zapisujeme do pomocného dvojrozmerného poľa B tak, že do i-teho riadku a j-tom stĺpci zapíšeme, či A(i) je väčšie alebo rovné A(j). Ak sa v i-tom riadku nachádzajú samé jednotky, čiže A(i) bolo väčšie alebo rovné od všetkých prvkov, zapíšeme do poľa M na i-tu pozíciu jednotku.

Keďže maximum môže byť v poli, v ktorom sú všetky prvky rôzne, len jedno, v poli M sa jednotka bude nachádzať len raz a to na pozícii zhodnej s pozíciou maxima.

```
Algoritmus 2.1/1
Vstup: Pole A
Výstup: Pole M(i), v ktorom sa nachádza jednotka na rovnakej
pozícii, ako v poli A maximum
begin
1: for 1 ≤ i, j ≤ n pardo
     if (A(i) ≥ A(j)) then set B(i,j): = 1
                      else set B(i,j): = 0
2: for 1 ≤ i ≤ n pardo
```

```
     set M(i): = B(i,1) & B(i,2) & … & B(i,n)
end
```

Riešenie 2

Algoritmus môžeme ešte trochu zjednodušiť. Každé políčko má priradených n procesorov, ktoré ho porovnajú so všetkými ostatnými prvkami v poli. Porovnanie spravíme v dvoch krokoch tak, že po prvom porovnaní chcú všetky procesory zapísať jednotku, po druhom nulu. V jednom kroku chcú teda všetky procesory zapísať tú istú hodnotu a tak nepotrebujeme n políčok na zápis, ale stačí nám jedno. Pritom neporušíme common CRCW PRAM.

Operáciu AND sme vykonali skryto poradím, v akom sme výsledky zapisovali – najprv jednotku, potom, ak prvok maximom nie je, nulu. Na tomto mieste určite stojí za zamyslenie, ako sa takýmto spôsobom urobí OR – najprv zapíšu procesory, ktoré chcú zapísať nulu, potom procesory, ktoré chcú zapísať jednotku.

V tomto riešení už nemusíme použiť pomocné dvojrozmerné pole B, ale medzivýsledky zapisujeme priamo do poľa M.

Algoritmus 2.1/2

```
Vstup: Pole A
Výstup: Pole M(i), v ktorom sa nachádza jednotka na rovnakej
pozícii, ako v poli A maximum
begin
1: for i = 1 to n pardo
     for j = 1 to n pardo {
       if (A(i) ≥ A(j)) then set M(i): = 1
       if (A(i) < A(j)) then set M(i): = 0}
end
```

Riešenie 3

Všimnime si, že prvé porovnanie v predchádzajúcom algoritme bolo takmer zbytočné – keďže porovnávame so všetkými prvkami poľa, teda aj so sebou samým, porovnanie na rovnosť bude vždy pravdivé. Mohli by sme teda tento príkaz vynechať a nahradiť inicializáciou prvkov poľa M na jednotky.

Alebo môžeme porovnanie na rovnosť odstrániť a porovnávať na ostrú nerovnosť na oboch riadkoch (toto si môžeme dovoliť, keďže máme na vstupe samé rôzne prvky). Keďže maximum je len jedno, jednotka sa zapíše len na jedno políčko a druhé porovnanie už nie je potrebné. Stačí nám na začiatku inicializovať prvky poľa M na nuly.

Algoritmus 2.1/3

```
Vstup: Pole A
Výstup: Pole M(i), v ktorom sa nachádza jednotka na rovnakej
pozícii, ako v poli A maximum
begin
1: for i = 1 to n pardo {
     set M(i) = 0
     for j = 1 to n pardo
       if (A(i) > A(j)) then set M(i): = 1}
end
```

Posledné riešenie je najjednoduchšie a azda najlepšie je na ňom vidieť, že ide o výpočet maxima. Prvé a druhé riešenie však majú pre nás veľký význam z hľadiska použitých postupov. Tieto sa nám zídu pri riešení ďalších príkladov.

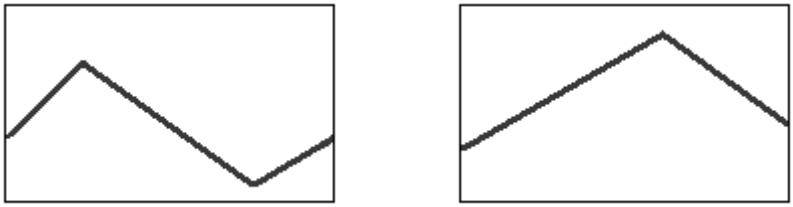
2. Logické operácie na boolovských poliach

Príklad 2.2

Máme n-prvkovú postupnosť A. Zistite v čase O(1) s použitím nanajvýš O(n) operácií na common CRCW PRAM, či je postupnosť bitonická.

Riešenie

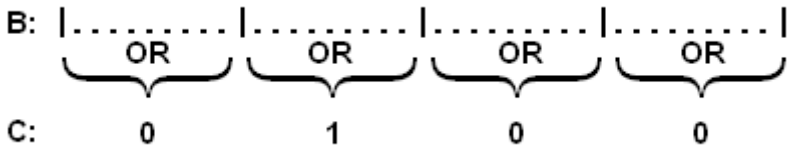
Využijeme vlastnosť bitonickej postupnosti, ktorá nám hovorí, že bitonická postupnosť má nanajvýš jedno minimum, maximum.



Najprv si overíme, či je len jedno minimum.

1. Každé políčko si pozrie svojich susedov, či nie je minimom. Ak minimom je, zapíše do pomocného poľa B dĺžky n na svoju pozíciu jednotku. Ak nie je, tak nulu.

2. Rozdelím si B na $\sqrt{n}$ úsekov dĺžky $\sqrt{n}$. Na každom úseku paralelne zistím (OR-om), či sa na ňom nachádza 1. Ak áno, zapíšeme do pomocného poľa C dĺžky $\sqrt{n}$ jednotku.



3. Overíme, či sa v poli C nachádza iba jedna jednotka – políčko, na ktorom sa jednotka nachádza, pozrie všetky ostatné, či ešte niekde ďalšia nie je. Pokiaľ je jednotiek

viac, znamená to, že bolo viac miním a teda postupnosť bitonická nie je.

4. Ak bola len jedna, tak spätne overíme úsek na B, ktorý jej prislúchal, či sa na ňom nenachádza jednotiek viac. Postupujeme rovnako ako v predchádzajúcom kroku.

Obdobný postup použijeme pri overovaní maxima.

Overme ešte, či nám sedí zložitosť. V 1. kroku každému políčku pridelíme jeden procesor, máme n políčok a teda W(n) = O(n). Porovnanie robíme s dvoma susedmi, čas je teda zrejme konštantný.

V 2. kroku sa opäť o každé políčko stará jeden procesor. Operáciu OR vykonáme v konštantnom čase tak, že v jednom kroku do príslušného C(i) zapíšu procesory, na ktorých políčkach bola nula, nulu, v druhom tie, na ktorých políčkach bola jednotka, zapíšu jednotku. Za zmienku stojí aj to, že sme vykonali simultánny zápis a neporušili sme pravidlá common CRCW PRAM – všetky procesory zapisovali rovnakú hodnotu.

Prezretie všetkých políčok v 3. a 4. kroku si vyžaduje n^2 operácií. My však pracujeme s poľom dĺžky $\sqrt{n}$ a teda W($\sqrt{n}$) = $\sqrt{n}^2$ = n.

Príklad 2.3

Máme boolovské pole A dĺžky n. Napíšte common CRCW PRAM algoritmus, ktorý v čase O(1) pri práci O(n) nájde najmenšie k také, že A(k) = 1.

Riešenie

1. Rozdelíme si vstupné pole na $\sqrt{n}$ úsekov dĺžky $\sqrt{n}$. Na každom úseku si zistíme, či sa na ňom nachádza aspoň jedna jednotka. Zvolíme rovnaký postup ako v 2. kroku v predchádzajúcom príklade. Výsledky si zapíšeme do pomocného poľa B.

2. Máme teda teraz pole dĺžky $\sqrt{n}$ s informáciou, v ktorých úsekoch sa nachádza jednotka. Nájdeme najmenšie k také, že B(k) = 1.

Teraz už môžeme hrubou silou – pre každé políčko s jednotkou pozrieme, či sa

pred ním nachádzajú už len nuly. Na to potrebujeme $\sqrt{n}^2$ = n procesorov, čo máme k dispozícii.

3. Ďalej pracujeme s úsekom na poli A, ktorý prislúchal „víťaznej" jednotke, teda s úsekom, na ktorom sa prvá jednotka bude nachádzať. Opäť hľadáme prvú jednotku na poli dĺžky $\sqrt{n}$. Zopakujeme teda postup z kroku 2.

Príklad 2.4

Ako rýchlo vieme vyriešiť predchádzajúci problém s rovnakou prácou na CREW PRAM?

Riešenie

Problém nastane pri zapisovaní. V algoritme 2.3 sa v jednom kroku na jedno políčko snaží písať nanajvýš $\sqrt{n}$ procesorov (v 1. kroku pri operácii OR). Pri exkluzívnom zápise si budú musieť najprv zvoliť procesor, ktorý bude zapisovať. Na voľbu použijeme binárny strom. V listoch má čísla procesorov. Ak je procesor ľavý, teda s menším číslom a chce zapisovať, postúpi o úroveň vyššie. Pravý procesor postúpi len, keď ľavý zapisovať nechce. Procesor, ktorý sa dostal až ku koreňu stromu, zapíše do políčka. Výška binárneho stromu s n listami je log n. My máme ale na vstupe len $\sqrt{n}$ procesorov, výška nášho stromu je teda $\log(\sqrt{n})$ a taká bude aj časová zložitosť algoritmu.

Stojí za zmienku, že pri takejto voľbe procesora na zápis nájdeme aj prvú jednotku daného úseku. Môžeme teda vynechať 3. krok. Algoritmus nám to síce trochu zrýchli, celkový odhad však zostane nezmenený.

Príklad 2.5 (All Nearest Smaller Values – ANSV)

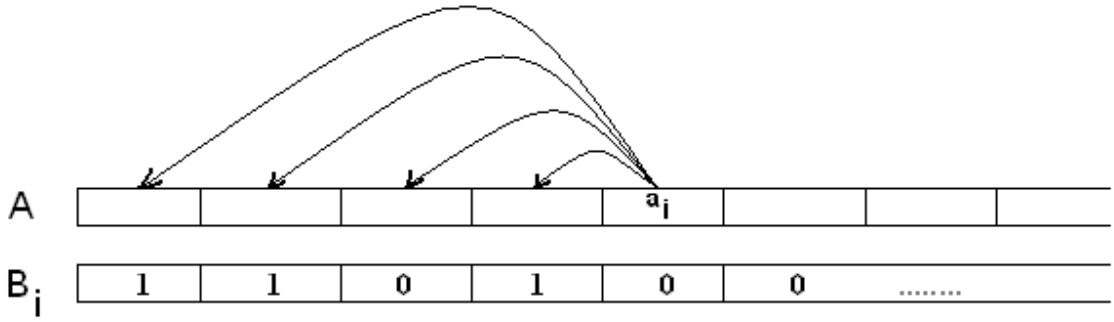
Máme pole A = (a_1, a_2, ... , a_n). Ľavá zhoda a_i, $1 \le i \le n$, je, ak také existuje, číslo a_k, také, že k je najväčší index taký, že $1 \le k \le i$ a $a_k < a_i$. Podobne definujeme pravú

zhodu. Nájdite ľavé a pravé zhody a_i pre všetky prvky poľa A v čase O(1) použijúc $O(n^2)$ na common CRCW PRAM.

Riešenie

Ukážeme si ako nájsť ľavé zhody. Riešenie pre pravé zhody je takmer rovnaké.

Pre každý prvok a_i pozrieme všetky prvky s nižším indexom (j < i) a porovnáme, či sú menšie. Ak áno, do pomocného poľa B_i zapíšeme jednotku (inak sú tam nuly).



Pomocou algoritmu 2.3 nájdeme najväčšie k také, že $B_i(k) = 1$.

Ľavá zhoda a_i je a_k.

Algoritmus 2.5

```
Vstup: Pole A
Výstup: Pole C
begin
1: for  1 ≤ i ≤ n pardo {
     for  1 ≤ j ≤ n pardo set Bi(j): = 0
     for  1 ≤ j ≤ i pardo
       if aj < ai then set Bi(j): = 1
```

```
        Nájdi na Bi najväčšie k také, že Bi(k) = 1
        set C(i): = A(k)}
end
```

Príklad 2.6

Napíšte common CRCW PRAM algoritmus na výpočet prefixových miním poľa $A = (a_1, a_2, \ldots, a_n)$ *pracujúci v čase* $O(1)$ *s* $O(n^2)$ *operáciami.*

Riešenie

1. A_i pozrie svojich predchodcov a zistí, či je od všetkých menší. Každému políčku sa priradí i procesorov, ktoré spravia porovnanie. Do pomocného poľa B zapíšu jednotku, ak A_i bolo menšie, nulu, ak nie. V dvoch krokoch, aby sme neporušili pravidlá common CRCW PRAM a v tomto poradí (vykonáme tak vlastne paralelne AND, pretože minimum musí byť od každého menšie).

2. Pokiaľ A_i minimom nie je, hľadáme najbližší menší index k taký, že A(k) samo sebe prefixovým minimom bolo. Postupujeme podobne ako pri hľadaní ľavej zhody – hľadáme najväčšie k také, že B(k) = 1.

Algoritmus 2.6

```
Vstup: Pole A
Výstup: Pole C
begin
1: for  1 ≤ i ≤ n pardo {
     for  1 ≤ j ≤ i pardo {
       if A(i) ≤ A(j) then set B(i): = 1
       if A(i) > A(j) then set B(i): = 0}
```

```
        if B(i) = 1 then C(i) = A(i)
        else {
          Nájdi na B najväčšie k také, že k < i a B(k) = 1
          set C(i): = A(k)}}
end
```

Príklad 2.7

Majme pole prirodzených čísel A dĺžky n. Nájdite common CRCW PRAM algoritmus, ktorý v čase O(1) a práci O(n^2) vypočíta pole L také, že

$$l_i = max\ \{ j \mid a_j = a_i, j < i\}.$$

Riešenie

Čiže pre každé a_i hľadáme najbližší prvok zľava s rovnakou hodnotou.

Každé a_i porovnáme so všetkými jeho predchodcami a výsledok porovnania zapíšeme do i-teho riadku pomocného dvojrozmerného poľa B na príslušnú pozíciu: $B_{ij}) = 1 \leftrightarrow (a_i = a_j, j < i)$.

Keďže sme do riadku B_i zapisovali jednotky len pre rovnaké prvky s menším indexom, stačí nám teraz nájsť pozíciu poslednej jednotky v riadku. Použijeme obdobný postup, ako pri riešení úlohy 2.3.

Porovnanie všetkých prvkov a naplnenie pomocného dvojrozmerného poľa spravíme na konečný počet krokov s použitím O(n^2) procesorov. Algoritmus 2.3 počítal na vstupe dĺžky n v čase O(1) pri práci O(n). Pri paralelnom výpočte na všetkých riadkoch súčasne sa čas nezmení, práca bude nW(n) = nO(n) = O(n^2).

Algoritmus 2.7

```
Vstup: Pole A
Výstup: Pole L
begin
1: for  1 ≤ i ≤ n pardo {
     for  1 ≤ j ≤ n pardo set Bi(j): = 0
     for  1 ≤ j ≤ i pardo
       if ai = aj then set Bi(j): = 1
     Nájdi na Bi najväčšie k také, že Bi(k) = 1
     set L(i): = A(k)}
end
```

Príklad 2.8

Nech A je pole obsahujúce n reálnych čísel. Napíšte CREW PRAM algoritmus, ktorý vypočíta pole L, pre ktoré platí, že L(i) = max j, kde j < i a súčasne A(j) > A(i). Algoritmus by mal pracovať v čase O(log n) a práci O(n).

Riešenie

Rátame takmer rovnako ako predošlý príklad, len nehľadáme A(i) = A(j), ale A(j) > A(i) a musíme zohľadniť, že teraz pracujeme s CREW PRAM.

Pri napĺňaní pomocného poľa B sme síce robili viacnásobné čítania, na jednu pozíciu ale zapisoval vždy len jeden procesor. Na nájdenie poslednej jednotky v riadku sme použili algoritmus 2.3. Ten však pracuje s common CRCW PRAM. Použijeme teda postup 2.4, ktorý modifikuje algoritmus 2.3 pre výpočet na CREW PRAM.

2. Logické operácie na boolovských poliach

Príklad 2.9

Máme utriedené pole A = (a_1, a_2, ... , a_n) také, že každé a_i je zafarbené farbou l_i, $l_i \in \{1, 2, \ldots, m\}$, m = O(log n). Napíšte CREW PRAM algoritmus pracujúci v čase O(log n) s O(n) operáciami, ktorý pre každé i, 1 ≤ i ≤ m, vypočíta minimálny prvok zafarbený l_i.

Riešenie

Rozdelíme si A na úseky dĺžky log n. Na jednotlivých úsekoch nájdeme minimá sekvenčne jedným prechodom úseku nasledovne: Máme pomocné dvojrozmerné pole, v ktorom riadky reprezentujú jednotlivé farby, stĺpce úseky. Hodnoty v ňom inicializujeme na nulu. Keďže vstupné pole je utriedené, na nájdenie minima danej farby na danom úseku nám stačí nájsť prvý výskyt prvku zafarbeného príslušnou farbou. Vezmeme prvok a_i a pozrieme na políčko na pomocnom poli, prislúchajúce danému úseku a farbe, ktorou je a_i zafarbené, či je hodnota na políčku rôzna od nuly. Ak je na políčku nula, minimum ešte nájdené nebolo, zapíšeme do políčka index i. (Nemôžeme zapísať priamo a_i, lebo to by sa mohlo rovnať nule.)

Máme teraz dvojrozmerné pole veľkosti log n x $\frac{n}{\log n}$. Na i-tom riadku sa nachádzajú úsekové minimá (resp. ich indexy) farby l_i. Keďže bolo vstupné pole utriedené, minimálny prvok farby l_i bude minimum toho úseku, na ktorom sa farba prvý krát vyskytla. Stačí nám teda nájsť prvý nenulový prvok na riadku. Využijem riešenie úlohy 2.3, respektíve 2.4, pričom namiesto prvej jednotky v boolovskom poli budeme hľadať prvý nenulový prvok.

Algoritmus 2.4 pracuje na CREW PRAM na poli dĺžky n v čase O(log $\sqrt{n}$) s O(n) operáciami. My máme na vstupe $\frac{n}{\log n}$ prvkov, čiže dobrý čas ostane zachovaný, operácií budeme potrebovať O($\frac{n}{\log n}$). Výpočet prebehne na všetkých riadkoch

paralelne, čiže log n W($\frac{n}{\log n}$) = log n O($\frac{n}{\log n}$) = O(n).

3. Zoznamy

Príklad 3.1 (Paralelný prefix)

Máme daný spájaný zoznam L dĺžky n. Nasledovník vrchola i je daný S(i), hodnota S posledného prvku zoznamu je 0. Každý vrchol si navyše pamätá hodnotu h(i). Napíšte EREW PRAM algoritmus pracujúci v čase O(log n), ktorý vypočíta na zozname prefixové sumy.

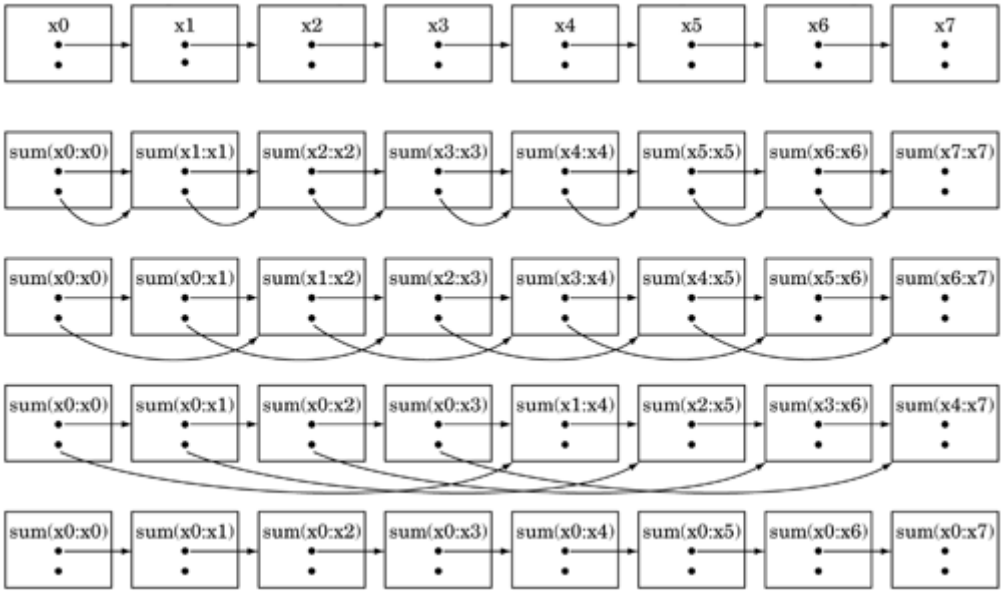
Riešenie

Úlohu budeme riešiť technikou pointer jumping. Popri presmerovávaní nasledovníkov, pripočítavame hodnotu vrchola k hodnote jeho nasledovníka.

Algoritmus si vyžaduje prácu O(n log n).

Algoritmus 3.1

```
Vstup: Zoznam L, hodnoty h(i)
Výstup: Zoznam L, hodnoty h(i)
begin
1: for i = 1 to n pardo {
     set S'(i): = S(i)
     while S'(i) ≠ 0 and S'(S'(i)) ≠ 0 {
       set h(S'(i)): = h(i) + h(S'(i))
       set S'(i): = S' (S'(i)); } }
end
```



Príklad 3.2 (List Ranking)

Máme daný spájaný zoznam L dĺžky n. Nasledovník vrchola i je daný S(i), hodnota S posledného prvku zoznamu je 0. Napíšte algoritmus pracujúci v čase O(log n), ktorý pre každý uzol vypočíta vzdialenosť od konca zoznamu.

Riešenie

Úlohu môžeme vyriešiť pomocou predchádzajúceho algoritmu. Na začiatku nastavíme každému vrcholu okrem posledného hodnotu 1. Pri presmerovávaní smerníkov, nebudeme pripočítavať hodnotu vrchola k hodnote jeho nasledovníka, ale hodnotu nasledovníka k hodnote vrchola.

```
Algoritmus 3.2

Vstup:  Zoznam L

Výstup:  Zoznam L, hodnoty rank(i)
```

```
begin
1: for i = 1 to n pardo {
     set S'(i) = S(i)
     if  S'(i) ≠ 0 then set rank(i): = 1
     else set rank(i): = 0
     while S'(i) ≠ 0 and S'(S'(i)) ≠ 0 {
       set rank(i): = rank(i) + rank(S'(i));
       set S'(i): = S' (S'(i)); } }
end
```

Algoritmus, ktorý sme si teraz ukázali, pracuje na EREW PRAM v čase O(log n) a práci O(n log n). Problém list rankingu ale vieme na EREW PRAM vyriešiť v rovnakom čase s počtom operácií O(n). Riešenie tohto algoritmu je však pomerne komplikované a predchádza mu množstvo teórie, na čo v tejto zbierke nie je priestor, preto ho neuvedieme. V ďalších príkladoch, aby sme mohli predviesť čo najefektívnejšie riešenia, keď sa budeme odvolávať na list ranking, budeme mať na mysli ten. Záujemcovia si ho môžu naštudovať v [1].

Príklad 3.3

Máme daný zoznam L dĺžky n, ktorého niektoré vrcholy sú označené červenou farbou. Nasledovník vrchola i je daný funkciou S(i), posledný prvok ukazuje sám na seba, funkcia Začiatok(L) ukazuje na prvý prvok zoznamu. Napíšte EREW PRAM algoritmus pracujúci v čase O(log n) s lineárnym počtom operácií, ktorý skonštruujte podzoznam L' zoznamu L, ktorý bude obsahovať červené prvky.

Riešenie

V prvom kroku presmerovaním smerníkov popreskakujeme vrcholy, ktoré nie sú

červené. Zoznam začínajúci počiatočným prvkom pôvodného zoznamu by teraz mal byť požadovaný červený podzoznam. Výnimku môžu tvoriť prvý a posledný prvok. Toto upravíme v kroku 2.

```
Algoritmus 3.3
Vstup: Zoznam L
Výstup: Podzoznam L s červenými vrcholmi
begin
1: for j = 1 to log n do
     for i = 1 to n pardo
       if S(i) ≠ červený then set S(i): = S(S(i))
2: for i = 1 to n pardo
     if i = Začiatok(L) then
       if i ≠ červený then set Začiatok(L): = S(Začiatok(L))
     if S(i) ≠ červený then set S(i): = i
end
```

[5]

Príklad 3.4

Majme daný zoznam L dĺžky n. Nasledovník vrchola i je daný S(i), hodnota S posledného prvku zoznamu je 0. Každý vrchol si navyše pamätá hodnotu h(i). Napíšte EREW PRAM algoritmus pracujúci v čase O(log n) s počtom operácií O(n), ktorý vypočíta minimálny prvok zoznamu.

Riešenie

Pomocou procedúry list ranking vieme určiť vzdialenosť vrchola od konca zoznamu na EREW PRAM v čase O(log n) s prácou O(n). Keď vieme vzdialenosť vrcholov od konca, môžeme si zoznam prerobiť na pole tak, že vrchol v(i) vložíme do poľa na pozíciu rank(i).

Na poli nájdem minimum v čase O(log n) s počtom operácií O(n) pomocou binárneho vyváženého stromu.

```
Vstup: Zoznam L
Výstup: A(1)
begin
1: List ranking
2: set A(rank(i)): = v(i)
3: for h = 1 to log n pardo
     for 1 ≤ i ≤ n/2^h pardo
       set A(i): = Min(A(2i – 1), A(2i))
end
```

Príklad 3.5

Majme ľubovoľný zakorenený strom T = (V, E) taký, že pre každý vrchol v máme daného nasledujúceho súrodenca s(v) a prvého potomka fc(v). (Pokiaľ vrchol ďalšieho súrodenca už nemá, s(v) = 0. Pokiaľ je v list, fc(v) = 0.) Napíšte algoritmus pracujúci v čase O(log n) pri práci O(n), ktorý každému vrcholu v určí rodiča p(v).

Riešenie

Určiť rodiča vrcholom, ktoré sú prvými potomkami, nie je problém. Každý vrchol

oznámi svojmu prvému potomkovi svoj index.

Ostáva nám určiť rodičov vrcholom, ktoré sú súrodencami prvých potomkov. Týchto máme vlastne určených zoznamom začínajúcom v každom vrchole, ktorý je prvým potomkom. Dĺžka takýchto zoznamov môže byť rôzna, v najhoršom prípade až n – 1 (ak je hĺbka stromu iba 1). Označme si dĺžky týchto zoznamov n_i, pričom vieme, že $n_i \leq n$ a $\sum n_i \leq n$.

Podobne ako v príklade 3.4 si zoznamy súrodencov prerobíme na polia. Poslať informáciu po poli v čase $O(\log n_i)$ s $O(n_i)$ procesormi už nie je problém.

Príklad 3.6

Máme kruh $K = (k_1, k_2, \ldots, k_n)$ reprezentovaný ako cyklický zoznam a hrany E spájajúce uzly v_i tak, že ľubovoľný uzol v_i je incidentný najviac s jednou hranou. Vieme zakresliť hrany do kruhu tak, že sa nebudú pretínať? Napíšte EREW PRAM algoritmus, ktorý toto zistí v čase O(log n) s použitím najviac O(n) operácií.

Riešenie

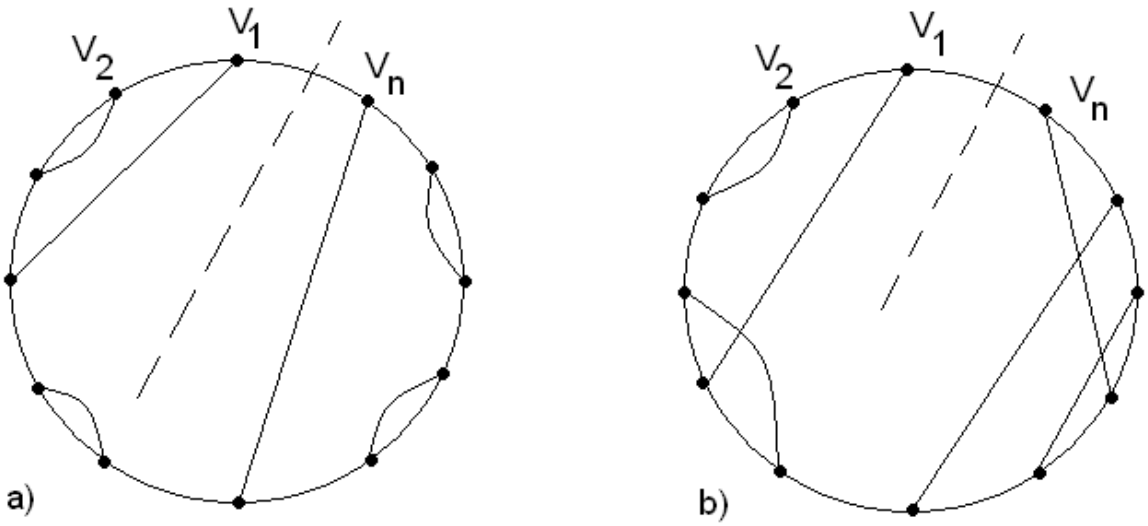
Kruh si najprv prerobíme na spájaný zoznam – vyberieme si vrchol, nech je to v_n a tam cyklický zoznam prerušíme. (Uvedomme si, ako dôležité je pre tento krok, že vrcholy máme očíslované.)

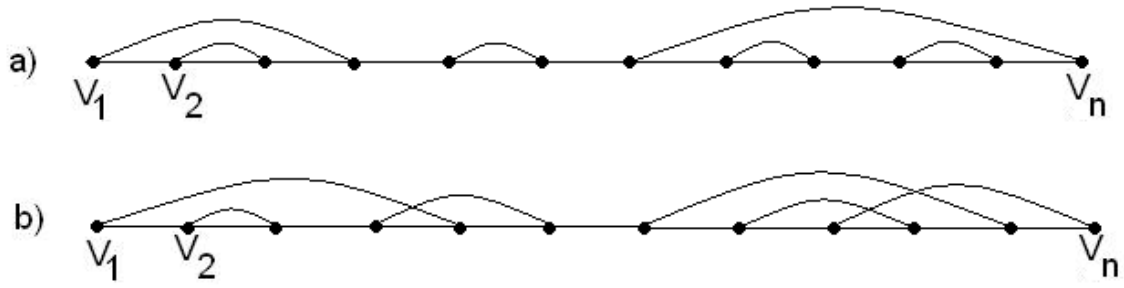
Zo zoznamu si podobne ako v príklade 3.4 urobíme pole. Zároveň si musíme prečíslovať hrany. To spravíme v kroku medzi list rankingom a prečíslovaním poľa – každý vrchol pozrie, aký rank vyšiel vrcholu, s ktorým ho spája hrana a hranu si podľa toho upraví.

Teraz, keď už máme problém pretransformovaný, poďme sa sústrediť na samotné riešenie problému. Ako sa nám čoskoro ukáže krok prerobenia kruhu na pole, nebol výhodný len z hľadiska ľahšieho riešenia problémov na poli, ale aj dobrý posun k nájdeniu triku na vyriešenie úlohy.

To, či sa hrany pretínajú zistíme nasledovne: Pre hranu idúcu z vrchola i do j, si zapíšeme interval <i, j>, i < j. Toto urobíme pre všetky hrany. Keďže z každého vrcholu ide najviac jedna hrana, takéto intervaly nemajú počiatočný ani koncový bod rovnaký. Všimnime si, že ak sa dve hrany krížia, tak intervaly, ktoré vymedzujú sa budú v prieniku tiež križiť – napr. pre hrany $v_a v_b$ a $v_c v_d$ <a, b> ∩ <c, d> = <c, b>. Pokiaľ sa nekrížia, tak prienik dvoch intervalov je buď prázdna množina alebo jeden interval sa nachádza v druhom celý. Viď obrázok.

Či <a, b> ∩ <c, d> = <c, b> vieme zistiť jednoducho prefixovými sumami. Na začiatok každého intervalu si do pomocného poľa vložíme + 1, na koniec – 1 a spočítame na tomto poli prefixové sumy ako v príklade 1.1. Prefixové sumy na začiatku a konci každého intervalu musia byť rovnaké. Prefixové sumy vieme vypočítať na EREW PRAM v čase O(log n) a v práci O(n). Ostatné kroky vieme s O(n) operáciami vykonať v čase O(1).




a)
V1
V2
Vn
b)
V1
V2
Vn

4. Triedenia

Príklad 1.1 (Odd-Even Transposition Sort)

Máme pole A n reálnych čísel. Utrieďte toto pole v lineárnom čase s lineárnym počtom operácií.

Riešenie

Triedenie, ktoré si ukážeme je veľmi podobné Bubble Sortu, len poradie, v akom sa porovnania vykonávajú sa líši. Porovnávame v dvoch fázach – v prvej porovnáme číslo na nepárnej pozícii s číslom na nasledujúcej párnej, v druhej číslo na párnej pozícii s číslom na nasledujúcej nepárnej. Ak je číslo s menším indexom väčšie, tak dvojicu prehodíme.

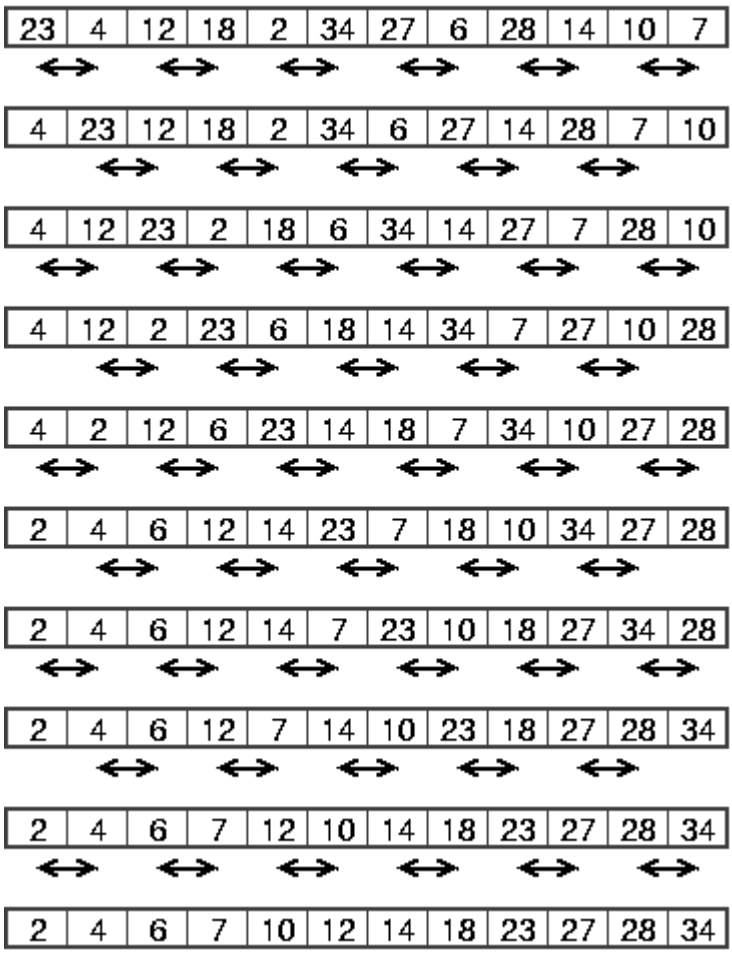
Korektnosť algoritmu môžeme dokázať indukciou. Najväčšie číslo, pokiaľ je potreba ho prehodiť, sa nikdy nezdrží. Na svoju pozíciu teda dorazí najneskôr na n – 1 krokov. Druhé najväčšie číslo sa zdrží len v prípade, že sa v prvom porovnaní stretne s najväčším. V takomto prípade musí jeden krok čakať. Potom však už nebude zdržané ani raz a nedorazí neskôr ako na n krokov.

Porovnania v jednej fáze môžeme vykonať paralelne. Porovnanie a prípadné prehodenie dvoch čísel vykoná jeden procesor – na jednu takúto fázu teda potrebujeme $\frac{n}{2}$ procesorov, čiže O(n) operácií. Ostatné kroky už musíme robiť sekvenčne a tak časová zložitosť bude O(n).

Algoritmus môže bežať aj na EREW PRAM.

Algoritmus

```
Vstup: Pole A

Výstup: Pole A

begin

for i = 0 to n do {

  for j = 1 to n/2 pardo

    if A(2*j) > A(2*j + 1) then Prehoď A(2*j) a A(2*j + 1)

  for j = 1 to n/2 pardo

    if A(2*j - 1) > A(2*j) then Prehoď A(2*j - 1) a A(2*j)

}

end
```



Príklad 4.2 (Rank Sort)

Máme pole A n reálnych čísel. Utrieďte paralelne toto pole využitím techniky rank sort.

Riešenie

V algoritme rank sort počítame počet čísel v poli, ktoré sú od prvku, ktorý chceme zaradiť menšie. Podobne, ako to bolo pri zoznamoch, počítame takto vzdialenosť prvku od začiatku poľa. Prvok potom na túto pozíciu zaradíme.

V sekvenčnom riešení problému hľadáme pozíciu prvku porovnaním so všetkými ostatnými, pričom, keď je od nejakého prvku väčší, pripočítame si do počítadla jedna. Postupným prechodom cez všetky prvky dostaneme časovú zložitosť $O(n^2)$. Paralelné riešenie sa nám priam núka. Každému prvku priradíme jeden procesor, ktorý mu rank v čase O(n) vyráta. Máme teda časovú zložitosť algoritmu O(n) pri práci O(n).

Algoritmus 4.2

```
Vstup: Pole A
Výstup: Pole B
begin
1: for i = 1 to n pardo {
     set rank: = 0
     for i = 1 to n do
       if A(i) > A(j) then inc(rank)
     B(rank): = A(i) }
end
```

Príklad 4.3

Vieme časovú zložitosť predchádzajúceho algoritmu znížiť na O(log n)? Koľko operácií budeme potrebovať?

Riešenie

Porovnanie vybraných čísel jedného s druhým môžeme robiť viacerými procesormi. Najlepší čas, konštantný, dostaneme, ak každému prvku priradíme n – 1 procesorov, ktoré ho porovnajú so zvyšnými prvkami poľa. Takýto postup je nám už dobre známy, napríklad z hľadania maxima v príklade 2.1. Problém však nastáva pri pripočítaní do počítadla. Doňho paralelný zápis nie je možný a môže si vyžadovať až n – 1 krokov. Na urýchlenie ale môžeme použiť binárny vyvážený strom, pomocou ktorého vieme súčet O(n) prvkov vykonať s O(n) operáciami v čase O(log n).

Výsledná zložitosť takto upraveného algoritmu bude $O(n^2)$ operácií v čase O(log n).

Príklad 4.4

Máme pole A obsahujúce n celých čísel z rozsahu 0 ... log n. Napíšte EREW PRAM algoritmus pracujúci v čase O(log n) a práci O(n), ktorý toto pole utriedi.

Riešenie

Vstupné pole A si rozdelíme do úsekov dĺžky log n.

Na nich paralelne spočítame výskyty jednotlivých čísel v danom úseku. Tieto početnosti si ukladáme do pomocného poľa B tak, že na i-tej pozícii sa nachádza výskyt čísla i mod log n v príslušnom úseku (konkrétne (i div log n + 1)).

Paralelne pracujeme na úsekoch dĺžky log n s poľom dĺžky n, teda T(n) = O(log n), W(n) = O(n). Zinicializovanie (vynulovanie) poľa B má T(n) = 1.

Algoritmus 4.4

```
Vstup: Pole A
Výstup: Pole A
begin
1: for i = 0 to n pardo
     set B(i): = 0;
2: for i = 0 to n/log n pardo
     for j = i*log n to i*log n + log n
       inc(B(A(j)*i));
```

Vypočítame výskyty v celom poli a to nasledovne: Každý proces spočíta výskyty jedného čísla c, teda c-ty proces spočíta políčka c + i log n a výsledok si uloží do pomocného poľa C na políčko C(c).

Keďže políčok je zjavne $\frac{n}{\log n}$ a spočítať sumu vieme v čase T(n) = log n ,

$T(\frac{n}{\log n}) \leq O(\log n)$. Pre W(n) = log n * W($\frac{n}{\log n}$) = log n * O($\frac{n}{\log n}$) = O(n).

```
3: for j = 0 to log n pardo
     for 1 ≤ i ≤ n/log n pardo
       set X(j,i): = B(i*log n + j);
     for k = 1 to log( n/log n ) do
       for 1 ≤ i ≤ (n/log n)/2^k pardo
         set X(j,i): = X(j,2i - 1) + X(j,2i);
     set C(j): = X(1);
```

Teraz už máme pole C, dĺžky log n, v ktorom sa na i-tej pozícii nachádza celkový

výskyt čísla i.

Jedným sekvenčným prechodom pole upravíme tak, že na i-tej pozícii budú všetky výskyty čísla i a čísel menších od neho (čiže najďalejší možný výskyt i v konečnom poli).

Keďže C je dĺžky log n, T(n) = W(n) = log n.

```
4: for i = 1 to log n − 1 do
     set C(i): = C(i) + C(i - 1)
```

Aby sme v ďalšom predišli konfliktom na EREW pamäti, nakopírujeme si odkiaľ sa má číslo vypísať do pomocného poľa D.

Opäť pracujeme sekvenčne s poľom dĺžky log n, čiže T(n) = W(n) = log n.

```
5: for i = 0 to log n − 1 do
     if i = 0 then set D(i) = 0
     else set D(i): = C(i - 1) + 1
```

Pomocou získaných informácií prepíšeme pole A utriedenými hodnotami. T(n) = 1, W(n) = n.

```
6: for i = 0 to log n − 1 pardo
     for j = D(i) to C(i) pardo
       set A(j): = i;
end
```

Príklad 4.5

Vieme utriediť pole dĺžky n v konštantnom čase? Koľko procesorov by sme potrebovali?

Riešenie

Vieme. Výpočet je však veľmi neefektívny a reálne v praxi nepoužiteľný – na common CRCW PRAM budeme potrebovať až n.n! procesorov.

Je však zaujímavé vedieť, že paralelné triedenie v konštantnom čase je vôbec možné.

Algoritmus 4.5

```
Vstup: Pole A
Výstup: Pole A
begin
1: for 0 <= p < n! pardo
      b(p) = true
 2: for 0 <= p < n!, 0 <= i < n - 1 pardo
      if a(p(i))  > a(p(i + 1)) or
         (a(p(i)) = a(p(i + 1)) and p(i) > p(i + 1)) then
        b(i) = false
 3: for 0 <= p < n!, 0 <= i < n pardo
       if (b(p)) {c(p(i)) = a(i); a(i) = c(i)}
end
```

[4]

5. Planárna geometria

Príklad 5.1

Máme množinu M, pozostávajúcu z n bodov v rovine. Ďalej máme bod P, ktorý do M nepatrí. Voronoiov mnohouholník V(P) je taký mnohouholník, ktorý ohraničuje všetky body Q, ktoré sú bližšie k P ako k ľubovoľnému bodu z M. Nájdite CREW PRAM algoritmus pracujúci v čase O(log n) a práci O(n log n), ktorý pre danú rovinu M a bod P nájde Voronoiov mnohouholník V(P).

Riešenie

Zoberme si bod A z daných n bodov v množine M a bod P. Body, ktoré sa nachádzajú bližšie k bodu P ako k bodu A, ležia v rovine, ktorú si určíme nasledovne: Označme si S stred medzi bodmi A a P, S = (A + P)/2. Vezmime si smernicu priamky AP a jej normálu, ktorá prechádza bodom S. Táto normála bude určovať polrovinu, v ktorej sa nachádzajú všetky body, ktoré sú bližšie k P ako A.

Spravíme prienik všetkých takýchto polrovín. Tento prienik je Voronoiov mnohouholník.

Prienik polrovín vieme na CREW PRAM spraviť v čase O(log n) a práci O(n log n). Konkrétny algoritmus môžeme nájsť napríklad v [1] str. 272 - 277.

Príklad 5.2

Máme v rovine N obdĺžnikov s hranami rovnobežnými so súradnicovými osami. Nájdite CREW PRAM algoritmus pracujúci v čase O(log n) a práci O(n^2), ktorý vypočíta plochu ich zjednotenia.

Riešenie

Príklad riešime rovnako, ako by sme riešili jeho sekvenčné riešenie.

Predpokladajme, že všetky súradnice sú rôzne. Najskôr si koncové body obdĺžnikov utriedime podľa x-ovej súradnice, čím nám vzniknú „pásy“, tvorené susednými súradnicami v utriedenom poradí x-vých súradníc, také, že vnútri týchto pásov sa nenachádzajú žiadne ďalšie body.

Každý takýto pás sa dá spracovať paralelne a teda problém ešte spočíva vo vypočítaní plochy zjednotenia v jednom takomto páse. (Výsledné hodnoty jednotlivých pásov sa ľahko spočítajú v čase O(log n) do jednej premennej napríklad algoritmom na výpočet prefixových súm.)

Poďme sa teda zaoberať tým, ako vypočítame plochu zjednotenia obdĺžnikov v konkrétnom páse. Všimnime si, že obdĺžniky môžeme rozdeliť do pásov aj podľa súradníc y-nových vrcholov. Pre každý x-ový pás je potom dôležité, ktoré y-nové pásy sú v ňom pokryté a ktoré nie.

Koncové body si utriedime aj podľa y-nových súradníc.

Vytvoríme si dvojrozmerné pomocné pole, v ktorom každému x-ovému pásu bude prislúchať jeden riadok, ktorý naplníme nasledovne: Ak príslušný obdĺžnik do x-ového pásu zasahuje, vložím na index prislúchajúci konkrétnemu y-novému pásu + 1, ak na ňom obdĺžnik začína, - 1, ak na ňom obdĺžnik končí. Ak obdĺžnik do pásu nezasahuje, zapíšeme nulu.

Na riadkoch spočítame prefixové sumy. Na miestach v pomocnom poli, ktoré reprezentujú pokrytú plochu, nám vyjdu jednotky, na nepokrytých miestach nuly. Teraz nám ostáva už len spočítať počet jednotiek v poli.

Vypočítať prefixové sumy, či sumy, na poli dĺžky n vieme v čase O(log n) a práci O(n). Keďže paralelne na n poliach, práca nám stúpne na $O(n^2)$.

Záver

Zbierka je zameraná hlavne na študentov fakulty navštevujúcich prednášku Efektívne paralelné algoritmy, ale dobre poslúži aj študentom podobných kurzov na iných školách, ich vyučujúcim či jednoducho záujemcom o paralelné programovanie. Je vhodná ako doplnok k prednáškam či k [1], ale dá sa použiť aj samostatne. Pokiaľ je nám známe, jedná sa vôbec o prvú učebnicu so zameraním na programovanie na PRAMoch v slovenskom jazyku.

Veríme, že zbierka paralelné programovanie čitateľovi priblížila a vďaka forme a postupnosti, akou je napísaná, si jednoducho osvojil postupy, techniky a triky využívané v paralelnom programovaní a podobné, ale aj zložitejšie úlohy, bude teraz ľahko vedieť vyriešiť sám.

Zoznam použitej literatúry

1. J. Jaja, An Introduction to Parallel Algorithms, Addison-Wesley, Reading, MA, 1992
2. B. Wilkinson, M. Allen, Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, PrenticeHall, Englewood Cliffs, NJ, 1998
3. W. Petersen, P. Arbenz, Introduction to Parallel Computing A Practical Guide with Examples in C, Oxford University Press, 2004
4. Jop Sibeyn, Lecture Notes Parallel Algorithms, http://users.informatik.uni-halle.de/~jopsi/dpar03/notes.shtml
5. Alan Gibbons, Wojciech Rytter, Efficient Parallel Algorithms, Cambridge University Press, 1998
6. David Culler, J.P. Singh, Anoop Gupta, Computer Architecture: A Hardware/Software Approach, Morgan Kaufmann, 1998
7. Timothy G. Mattson, Beverly A. Sanders, Berna L. Massingill, Patterns for Parallel Programming, Addison Wesley Professional, 2004
8. Uzi Vishkin, Thinking in Parallel: Some Basic Data-Parallel Algorithms and Techniques, http://www.umiacs.umd.edu/users/vishkin/PUBLICATIONS/classnotes.pdf