Algorytmy równoległe

Model obliczeń: PRAM (*Parallel Random Access Machine*):

- Wiele procesorów: $P_0,\ldots,P_{p-1}$,
- wspólna pamięć
- praca krokowa, zsynchronizowane fazy jednego kroku procesora:
  - odczyt z komórki pamięci
  - obliczenie lokalne w procesorze
  - zapis do komórki pamięci

  Czas jednego kroku: $O(1)$.

Algorytmy równoległe

Dostępy do pamięci: Concurrent/Exclusive Read/Write.
Możliwe wersje PRAM:
EREW, CREW, ERCW, CRCW.
Konflikty przy zapisie:

- *common-CRCW* – kolidujące procesory muszą zapisywać tę samą wartość (ograniczenie na algorytm)
- *arbitrary* – zapisywana jest dowolna z kolidujących wartości
- *priority* – wygrywa procesor o najmniejszym numerze
- *combining* – zapisywana jest pewna kombinacja kolidujących wartości (n.p. suma, maximum,. . .)

Pointer jumping

Problem *List ranking*: Mamy elementy w pamięci połączone wskaźnikami *next* w listę jednokierunkową. Chcemy dla każdego elementu i wyznaczyć jego odległość od końca listy $d[i]$. Tzn:

$$d[i] = \begin{cases} 0, & \text{jeśli } \textit{next}[i] = NIL \\ d[\textit{next}[i]] + 1, & \text{jeśli } \textit{next}[i] \neq NIL \end{cases}$$

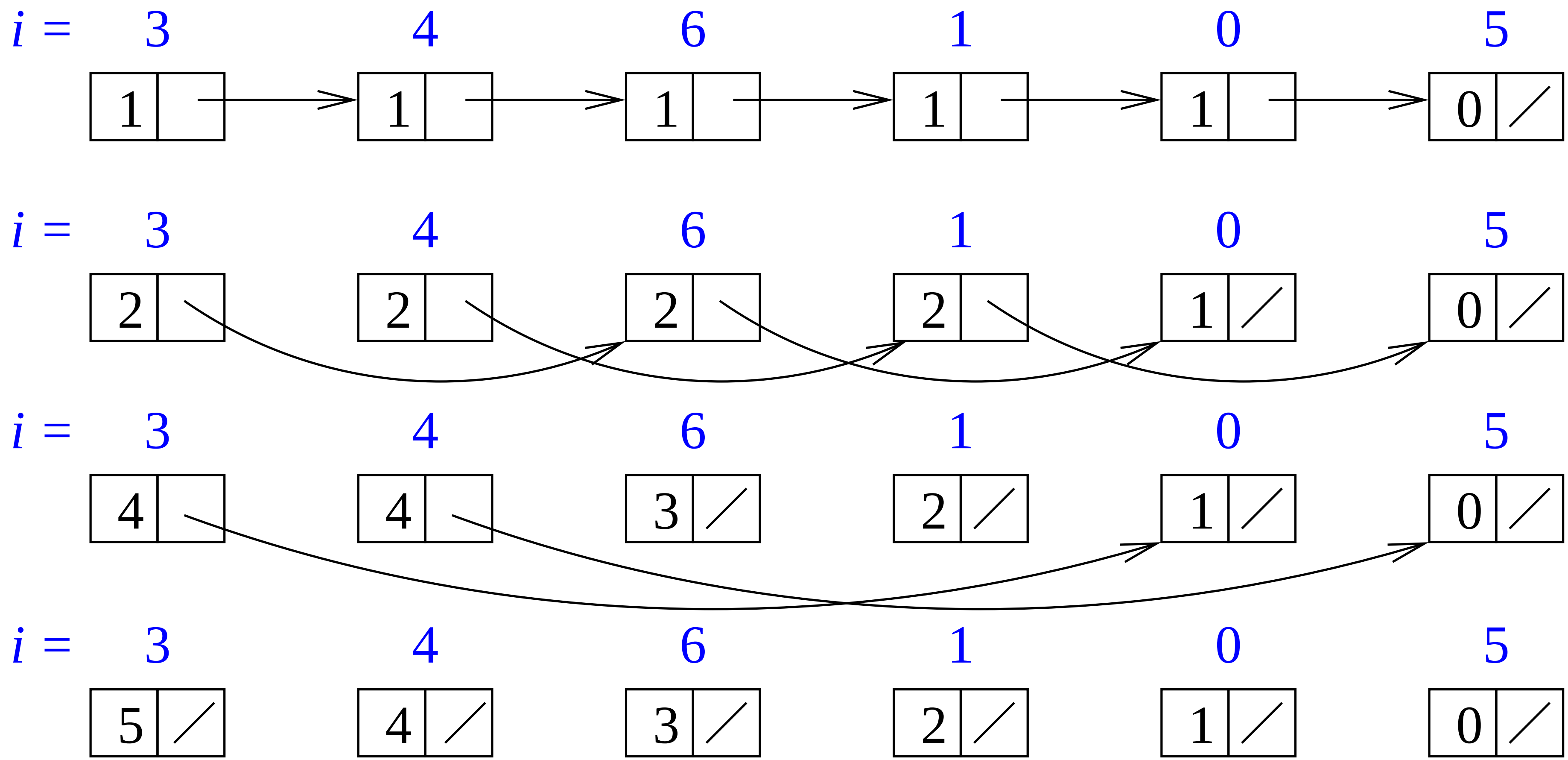
Zakładamy, że za każdy obiekt i na liście L odpowiada osobny procesor P_i (tzn. P_i modyfikuje wartości $d[i]$ oraz *next*$[i]$).

List-Rank

List-Rank(L)
1 for each P_i in parallel do
2   if *next*$[i] = NIL$
3   then $d[i] \leftarrow 0$
4   else $d[i] \leftarrow 1$
5 while *istnieje* i *takie, że next*$[i] \neq NIL$ do
6   for each P_i in parallel do
7   if *next*$[i] \neq NIL$ then
8          $d[i] \leftarrow d[i] + d[$*next*$[i]]$
9   *next*$[i] \leftarrow$*next*[*next*$[i]$]

Uwagi: Linie 8 i 9 – w każdym podstawieniu: wszystkie odczyty argumentów przed wszystkimi zapisami wyników.

List-Rank



i = 3 4 6 1 0 5
1 1 1 1 1 0
i = 3 4 6 1 0 5
2 2 2 2 1 0
i = 3 4 6 1 0 5
4 4 3 2 1 0
i = 3 4 6 1 0 5
5 4 3 2 1 0

List-Rank – poprawność

Niezmiennik: Dla każdego i, na początku każdej iteracji pętli `while` suma wartości d w podliście o początku i jest równa odległości i od końca początkowej listy L.

Wiersze `1-4` wymuszają niezmiennik przed pierwszą iteracją.

Linie `8-9` zachowują niezmiennik: gdy następnik i jest "omijany" jego pole d jest dodawane do $d[i]$ (i to samo dla pozostałych elementów na nowej liście o początku i).

List-Rank – EREW

Algorytm jest typu EREW:
Brak kolizji zapisu – tylko P_i pisze do $d[i]$ i *next*$[i]$.
Brak kolizji odczytu – niezmiennik: dla każdych $k \neq l$, albo *next*$[k] \neq$*next*$[l]$, albo *next*$[k] =$*next*$[l] = NIL$.
Ten niezmiennik – prawdziwy na początku i zachowywany w wierszu 9.
Ponieważ wszystkie wartości *next* różne od NIL są parami różne, odczyty w wierszu 9 są rozłączne.

List-Rank – analiza

Niech n długość L.
Wiersze `1-4` – czas: $O(1)$.
Liczba iteracji:
W każdej iteracji – każda lista rozbijana na dwie: elementy z pozycji parzystych lądują w jednej, a z pozycji nieparzystych – w drugiej. Stąd: każda iteracja zmniejsza długość najdłuższej listy dwukrotnie. Po $\lceil \lg n \rceil$ iteracjach wszystkie listy są jednoelementowe.
Czas `List-Rank`: $\Theta(\lg n)$.
Praca: $\Theta(n \lg n)$ (Czas $\Theta(\lg n)$ i liczba procesorów $\Theta(n)$. Jeśli np. $n = 2^k$, to do końca aktywne $\geq n/2$ procesorów.)
`List-Rank` nie jest *sekwencyjnie-efektywny*: praca różni się od optymalnego algorytmu sekwencyjnego o czynnik większy niż stały.

Obliczenia prefiksowe

Dane: binarny, łączny operator $\otimes$ oraz lista: $< x_1, \ldots, x_n >$.
Wynik: lista $< y_1, \ldots, y_n >$ taka, że $y_1 = x_1$ i, dla $k > 1$, $y_k = x_1 \otimes x_2 \otimes \ldots \otimes x_k$.
Za element i odpowiada procesor P_i.

```
List-Prefix(L)
▷ i oznacza nr procesora a nie pozycję na L
for each P_i in parallel do
    y[i] ← x[i]
while istnieje i takie, że next[i] ≠ NIL do
    for each P_i in parallel do
        if next[i] ≠ NIL then
            y[next[i]] ← y[i] ⊗ y[next[i]]
            next[i] ← next[next[i]]
```

Czas: $\Theta(\lg n)$, praca: $\Theta(n \lg n)$, typ: EREW. (Te same argumenty co w `List-Rank`.)

List-Prefix – analiza

Niech $[i, j] = x_i \otimes x_{i+1} \otimes \ldots \otimes x_j$ dla $1 \leq i \leq j \leq n$.
Niezmiennik: Po t-tej iteracji `while` k-ty obiekt na L zapamiętuje w y wartość $[\max\{1, k - 2^t + 1\}, k]$.
Na początku k-ty obiekt ma obliczone $x_k = [k - 2^0 + 1, k]$, i na niego wskazuje *next* obiektu $k - 1$ jeśli $k - 1 \geq 1$.
Przed iteracją $t > 0$, y dla k-tego obiektu wynosi $[\max\{1, k - 2^t + 1\}, k]$, i na k-ty obiekt wskazuje *next* obiektu $k - 2^t$ jeśli $k - 2^t \geq 1$. W wierszu 6 iteracji t obiekt $k - 2^t$ zapisuje w k-tym obiekcie $[\max\{1, k - 2^t - 2^t + 1\}, k - 2^t] \otimes [\max\{1, k - 2^t + 1\}, k] = [\max\{1, k - 2^{t+1} + 1\}, k]$, a po wierszu 7 wskazuje na niego obiekt $k - 2^t - 2^t = k - 2^{t+1}$ jeśli $k - 2^{t+1} \geq 1$.

Metoda cyklu Eulera

Cykl Eulera w grafie – cykl, który przechodzi przez każdą krawędź dokładnie raz.
Podamy przykład metody cyklu Eulera dla wyznaczania głębokości węzłów w drzewie binarnym T.
Każdy węzeł i ma pola: *parent*$[i]$, *left*$[i]$, *right*$[i]$.
Każdemu węzłowi przypisujemy 3 procesory: A, B i C.

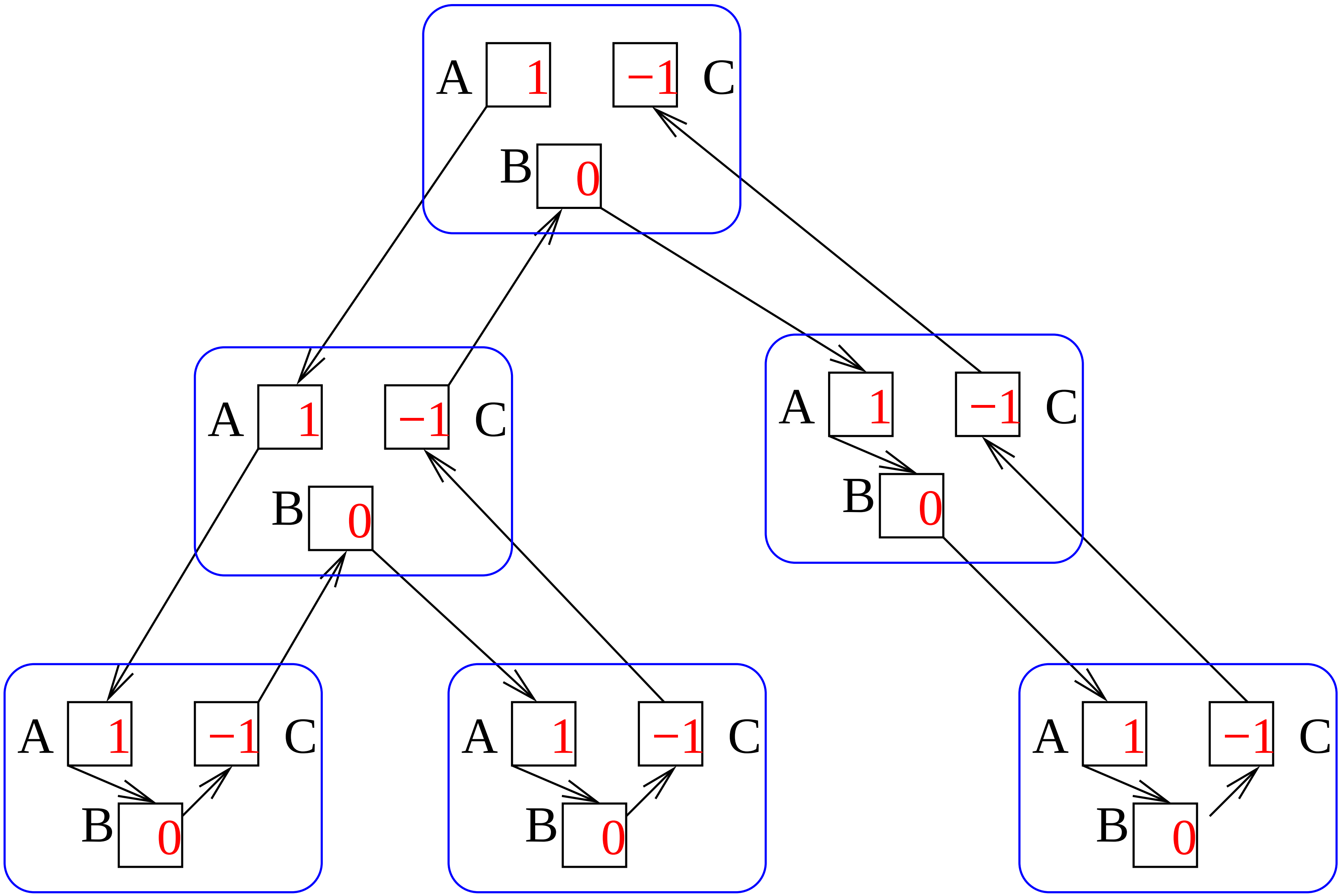
Metoda cyklu Eulera

Budujemy cykl Eulera dla drzewa T, traktując T jako graf skierowany, reprezentowany jako następująca lista procesorów:

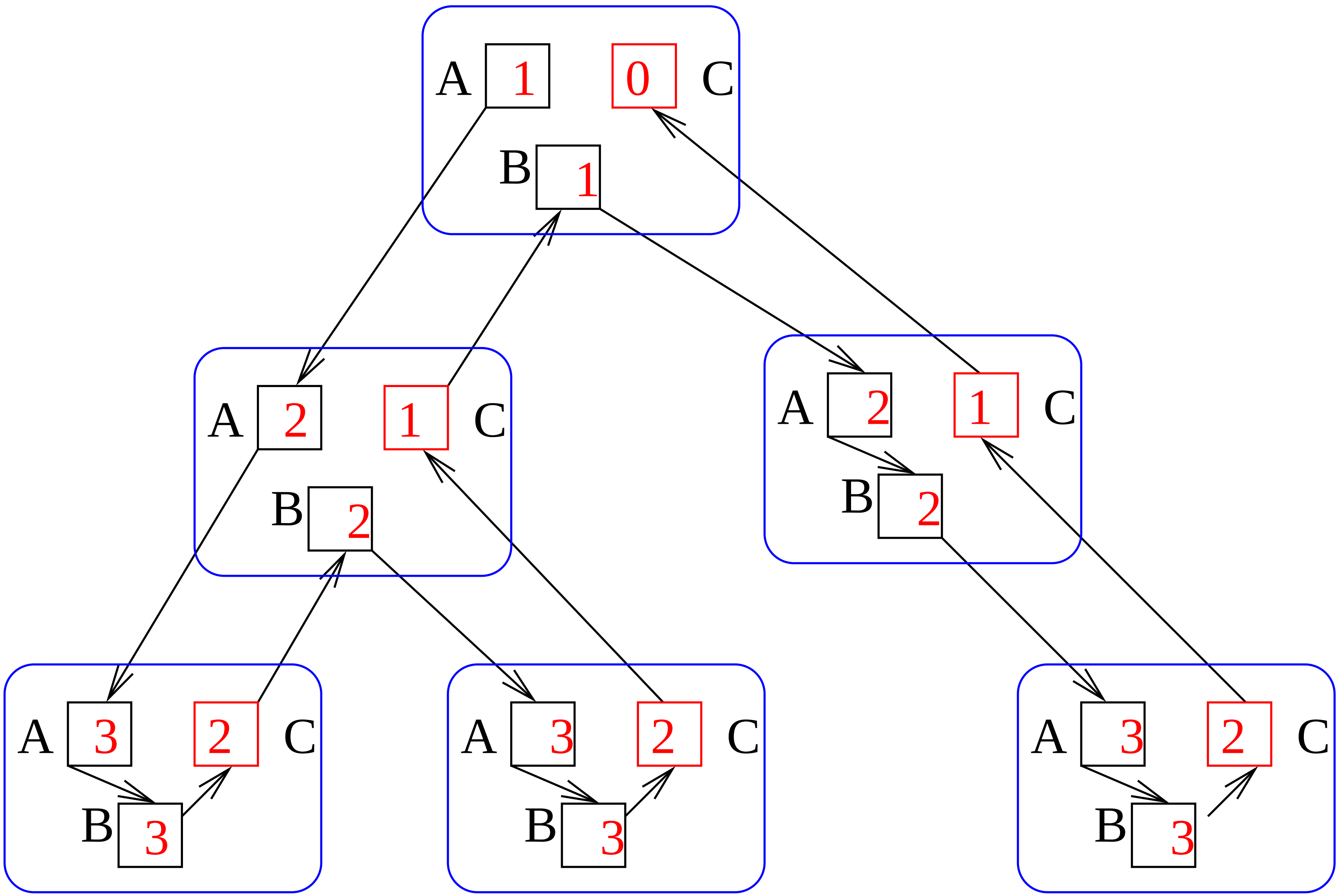
- Procesor A węzła i wskazuje na procesor A węzła *left*$[i]$, a jeśli *left*$[i]$ nie istnieje – na procesor B węzła i
- Procesor B węzła i wskazuje na procesor A węzła *right*$[i]$, a jeśli *right*$[i]$ nie isntnieje – na procesor C węzła i
- Procesor C węzła i wskazuje na procesor B (odp. C) węzła *parent*$[i]$ jeśli jest lewym (odp. prawym) synem, lub na NIL jeśli i – korzeń.

Umieszczamy 1 w każdym procesorze A, 0 w każdym procesorze B i -1 w każdym procesorze C i na skonstruowanej liście uruchamiamy `List-Prefix` z dodawaniem jako $\otimes$.

Metoda cyklu Eulera



Metoda cyklu Eulera



Metoda cyklu Eulera

Po zakończeniu – głebokość węzła jest w jego procesorze C:

Liczby 1, 0 i -1 zostały tak rozmieszczone, że suma w każdym poddrzewie i jest 0.

Przed wejściem do korzenia suma prefiksu jest 0 (głębokość korzenia).

Wartość prefixu zapisana w C węzła i jest taka jak przed wejściem do poddrzewa i.

Przy schodzeniu do *left*$[i]$ (przez A w węźle i) suma prefixu zwiększa się o 1 (*i będzie zapamiętana w* C[*left*$[i]$]).

Przy schodzeniu do *right*$[i]$ (przez B w węźle i) jest taka sama (*jak przy schodzeniu do left*$[i]$).

Czas: $O(\lg n)$ (inicjacja w czasie $O(1)$ i `List-Prefix` na liście długości $3n$)

Algorytm typu EREW.

Find-Roots

Dany las F, gdzie każdy węzeł i wskazuje na rodzica *parent*$[i]$. *parent*$[i] = NIL$ jeśli i – korzeń.
Wyznaczyć dla każdego i, *root*$[i]$ – korzeń drzewa zawierającego i.
Każdemu węzłowi i przypisany procesor P_i.

```
Find-Roots(F)
for each P_i in parallel do
    if parent[i] = NIL then
        root[i] ← i
while istnieje i taki, że parent[i] ≠ NIL do
    for each P_i in parallel do
        if parent[i] ≠ NIL then
            root[i] ← root[parent[i]]
            parent[i] ← parent[parent[i]]
```

Find-Roots

Algorytm typu CREW: wielu synów czyta informacje ze wspólnego rodzica, ale każdy zapisuje tylko swoje zmienne. Czas: $O(\lg d)$, gdzie d największa głębokość drzewa (podobnie jak w `List-Rank`, gdzie lista – ścieżka do korzenia.)

Dla maszyn EREW rozwiązanie tego zadania wymaga czasu $\Omega(\lg n)$: Nawet, jeśli mamy tylko jedno drzewo o głębokości 1, to skopiowanie identyfikatora korzenia do jego $n-1$ synów wymaga czasu $\Omega(\lg n)$. (Każdy zapis i każdy odczyt może co najwyżej potroić łączną liczbę kopii identyfikatora w procesorach i komórkach pamięci.)

Fast-Max

Dane: tablica $A[0 \ldots n-1]$.
Wynik: maksimum z elementów A.
Algorytm common-CRCW. Używamy n^2 procesorów. Dla $0 \leq i, j \leq n-1$, procesor $P_{i,j}$ porównuje $A[i]$ z $A[j]$.

`Fast-Max(`A`)`

$n \leftarrow$ *length*$[A]$
for each $P_{i,0}$ **in parallel do**
    $m[i] \leftarrow TRUE$
for each $P_{i,j}$ **in parallel do**
    if $A[i] < A[j]$ **then**
        $m[i] \leftarrow FALSE$ $\triangleright$ $A[i]$ *nie jest maksimum*
for each $P_{i,0}$ **in parallel do**
    if $m[i] = TRUE$ **then**
        $max \leftarrow A[i]$
return max

Fast-Max

Czas $O(1)$. Możliwe konflikty zapisów w liniach 6 i 9, ale wtedy kolidujące wartości są takie same.
Rozwiązanie tego zadania na maszynach CREW wymaga czasu $\Omega(\lg n)$: Nawet policzenie alternatywy n bitów (przypadek, gdy wszystkie wartości są 0 albo 1) wymaga czasu $\Omega(\lg n)$.

Symulacja CRCW na EREW

Tw. Każdy p-procesorowy algorytm CRCW może być co najwyżej $O(\lg p)$ razy szybszy od najszybszego p-procesorowego algorytmu EREW dla tego samego problemu.

D-d. *Symulacja w czasie $O(\lg p)$ jednego kroku CRCW na EREW*:

symulacja zapisu: Pomocnicza tablica $A[0 \ldots p-1]$. Kiedy P_i z CRCW zapisuje x_i do komórki l_i, odpowiadający mu P_i' z EREW zapisuje (l_i, x_i) do $A[i]$. Sortujemy A ze względu na pierwsze współrzędne par. (Można to zrobić w czasie $O(\lg p)$). Dla $i = 1, \ldots, p-1$, P_i' odczytuje $A[i] = (l_j, x_j)$ oraz $A[i-1] = (l_k, x_k)$. Jeśli $l_j \neq l_k$ lub $i = 0$, to P_i' zapisuje x_i do komórki l_j. (Pary $(l_i, *)$ są obok siebie i najwcześniejsza wygrywa.)

symulacja odczytu: ćwiczenie. $\square$

Tw. Brenta

Tw. Każdy układ kombinacyjny, głębokości d i ograniczonym przez stałą stopniu wejściowym można symulować na p-procesorowym CREW PRAM w czasie $O(n/p+d)$.

D-d. Dane wejściowe układu – w pamięci. Na wyjście każdej bramki – osobna komórka pamięci. Symulacja 1 bramki – w czasie $O(1)$ przy użyciu jednego procesora: wczytanie (stałej ilości) wejść i zapisanie wyjścia.
Dla $i=1,\ldots,d$, niech n_i – liczba bramek o głębokości i. W i-tej fazie symulujemy n_i bramek o głębokości i. Dzielimy je na $\lceil n_i/p \rceil$ grup po p (ostatnia może mieć $\leq p$), i każdą grupę symulujemy równolegle.

Łączny czas: $\sum_{i=1}^{d} \lceil n_i/p \rceil \leq \sum_{i=1}^{d} (n_i/p+1) = n/p+d$ $\square$

Uwaga: Tu sieci komparatorów też można traktować jak układy kombinacyjne.

symulacje

Wn. Każdy n-elementowy układ o głębokości d i ograniczonych przez stałą stopniach wejściowym i wyjściowym bramek można symulować na p-procesorowym EREW PRAM w czasie $O(n/p + d)$.

D-d. Symulacja taka jak w Tw. Brenta, z tym, że wartość obliczna na wyjściu bramki jest kopiowana przez symulujący procesor na te wejścia innych bramek, na których będzie potrzebna (lub na wyjścia układu) w stałym czasie, bo stopień wyjściowy bramki – stały. □

symulacje

Tw. Jeśli A działa na p-procesorowej maszynie PRAM w czasie t, to dla $p' < p$ istnieje algorytm A' dla tego samego typu PRAM p'-procesorowego, działający w czasie $O(pt/p')$.

D-d. Każdy odczyt i każdy zapis A można symulować w czasie $\lceil p/p' \rceil$. $\square$

Uwaga: Jeśli w (odp. t) jest dolną granicą na czas sekwencyjny (odp. równoległy), to najszybszy algorytm sekwencyjnie efektywny używa $O(w/t)$ procesorów. (Dla $\omega(w/t)$ procesorów nie ma algorytmu sekwencyjnie efektywnego.) Jeśli znajdziemy algorytm sekwencyjnie efektywny dla $\Theta(w/t)$ procesorów, to dla każdej liczby $p' = o(w/t)$ procesorów twierdzenie gwarantuje istnienie algorytmu sekwencyjnie efektywnego. (*Czas się zwiększy tyle razy ile razy zmniejszy się liczba procesorów.*)

Deterministyczne łamanie symetrii

Chcemy wybrać z listy pewien stały procent elementów, tak aby żadne dwa wybrane elementy nie sąsiadowały na liście.

Kolorowanie grafu nieskierowanego $G = (V, E)$ – funkcja $C : V \to N$ taka, że jeśli $C(u) = C(v)$, to $(u, v) \notin E$.

Niezależny zbiór wierzchołków grafu $G = (V, E)$ – podzbiór $V' \subseteq V$ taki, że każda krawędź jest incydentna z ≤ 1 wierzchołkiem z V'.

Maksymalny niezależny zbiór (MIS) – niezależny zbiór V' taki, że dla każdego $v \in V \setminus V'$ zbiór $V' \cup \{v\}$ nie jest niezależny.

Uwaga: każdy MIS na liście zawiera $\geq n/3$ obiektów (musi zawierać ≥ 1 spośród każdych 3 kolejnych elementów). Na liście można też wykonać `List-Rank` i wtedy elementy o parzystej odległości od końca stanowią MIS rozmiaru $\lceil n/2 \rceil$.

Six-Color

`Six-Color` – kolorowanie listy 6 kolorami. Z każdym obiektem listy x – związany procesor $P(x) \in \{0, \ldots, n-1\}$. Dla każdego x obliczamy ciąg kolorów $C_0[x], \ldots, C_m[x]$. *C_0 jest n-kolorowaniem: $C_0[x] = P(x)$.* Do zapisania $C_0[x]$ wystarczy $\lceil \lg n \rceil$ bitów.

Six-Color

`Six-Color` – kolorowanie listy 6 kolorami. Z każdym obiektem listy x – związany procesor $P(x) \in \{0, \ldots, n-1\}$. Dla każdego x obliczamy ciąg kolorów $C_0[x], \ldots, C_m[x]$. C_0 *jest* n*-kolorowaniem:* $C_0[x] = P(x)$. Do zapisania $C_0[x]$ wystarczy $\lceil \lg n \rceil$ bitów.

W iteracji obliczamy C_{k+1} *na podstawie* C_k*:* Niech r – liczba bitów do zapisania koloru w C_k oraz $C_k[x] = a =< a_{r-1}, \ldots, a_0 >$ i $C_k[$*next*$[x]] = b =< b_{r-1}, \ldots, b_0 >$. Oczywiście: $a \neq b$. Niech $i = \min\{j : a_j \neq b_j\}$. Ponieważ $0 \leq i \leq r-1$, i można zapisać za pomocą $r' = \lceil \lg r \rceil$ bitów jako $< i_{\lceil r' \rceil - 1}, \ldots, i_0 >$. Wyznaczamy $C_{k+1}[x] = < i_{\lceil r' \rceil - 1}, \ldots, i_0, a_i >$ ($\neq C_{k+1}[$*next*$[x]]$). Zmieniamy $r \leftarrow \lceil \lg r \rceil + 1$.

Six-Color

`Six-Color` – kolorowanie listy 6 kolorami. Z każdym obiektem listy x – związany procesor $P(x) \in \{0, \ldots, n-1\}$. Dla każdego x obliczamy ciąg kolorów $C_0[x], \ldots, C_m[x]$. *C_0 jest n-kolorowaniem: $C_0[x] = P(x)$.* Do zapisania $C_0[x]$ wystarczy $\lceil \lg n \rceil$ bitów. *W iteracji obliczamy C_{k+1} na podstawie C_k:* Niech r – liczba bitów do zapisania koloru w C_k oraz $C_k[x] = a =< a_{r-1}, \ldots, a_0 >$ i $C_k[\textit{next}[x]] = b =< b_{r-1}, \ldots, b_0 >$. Oczywiście: $a \neq b$. Niech $i = \min\{j : a_j \neq b_j\}$. Ponieważ $0 \leq i \leq r-1$, i można zapisać za pomocą $r' = \lceil \lg r \rceil$ bitów jako $< i_{\lceil r' \rceil -1}, \ldots, i_0 >$. Wyznaczamy $C_{k+1}[x] = < i_{\lceil r' \rceil -1}, \ldots, i_0, a_i >$ ($\neq C_{k+1}[\textit{next}[x]]$). Zmieniamy $r \leftarrow \lceil \lg r \rceil + 1$. Dla $r = 3$ nowy kolor może powstać przez dopisanie 0 lub 1 do numeru pozycji $0 =< 0, 0 >$, $1 =< 0, 1 >$ lub $2 =< 1, 0 >$ i ponownie $r = 3$. Mamy 6 kolorów – kończymy.

Six-color – analiza

Typ algorytmu: EREW ($P(x)$ czyta x i *next*$[x]$, zapisuje w x).
Niech r_i liczba potrzebnych bitów w kolorowaniu C_i.
Udowodnimy, że jeśli $\lceil \lg^{(i)} n \rceil \geq 2$, to $r_i \leq \lceil \lg^{(i)} n \rceil + 2$.

Podstawa: $r_1 \leq \lceil \lg^{(1)} n \rceil$.

Krok indukcyjny: Niech $r_{i-1} \leq \lceil \lg^{(i-1)} n \rceil + 2$.

$r_i \leq \lceil \lg r_{i-1} \rceil + 1 \leq \lceil \lg(\lceil \lg^{(i-1)} n \rceil + 2) \rceil + 1$

(*Z faktu, że* $\lceil \lg^{(i)} n \rceil \geq 2$*, wynika:* $\lceil \lg^{(i-1)} n \rceil \geq 3 > 2$).

$\leq \lceil \lg(2\lceil \lg^{(i-1)} n \rceil) \rceil + 1 =$

($\lg^{(i-1)} n > 2$ *i dla* $x \geq 1$*:* $\lceil \lg(2\lceil x \rceil) \rceil = \lceil \lg(2x) \rceil$ *(ćw.)*)

$= \lceil \lg(2 \lg^{(i-1)} n) \rceil + 1 = \lceil \lg(\lg^{(i-1)} n) + 1 \rceil + 1 = \lceil \lg^{(i)} n \rceil + 2$

Stąd $r_{\lg^* n} \leq \lceil \lg^{(\lg^* n)} n \rceil + 2 = 3$. ($\lg^* n = \min\{i : \lg^{(i)} n \leq 1\}$)

Zatem czas `Six-Color`: $O(\lg^* n)$.

List-MIS

Algorytm EREW, w czasie $O(c)$ oblicza MIS dla n-elementowej listy mając jej c-kolorowanie.
Dane: c-kolorowanie C.
Każdy element x ma atrybut *alive*$[x]$ (początkowo: $TRUE$). W i-tej iteracji procesor $P(x)$ sprawdza czy $C[x] = i$ oraz *alive*$[x] = TRUE$. Jeśli tak, to zaznacza x jako wybrany do MIS i ustawia pola *alive* następnika i poprzednika x na $FALSE$.

Otrzymany zbiór niezależny: Każda para sąsiadów ma różne kolory i ten o mniejszym kolorze albo jest niewybrany albo przy swoim wyborze zapewnił, że sąsiad jest nieżywy.

Otrzymany zbiór jest maksymalnym zbiorem niezależnym: Jeśli ani następnik ani poprzednik węzła y nie zostali wybrani to *alive*$[y] = TRUE$ i w kroku $C[y]$ element y został dołączony do MIS.