Paralelní algoritmy

Paralelní algoritmy

Dosud jsme uvažovali jen **sekvenční** algoritmy:

- jsou vykonávány jen jedním procesorem
- množství „práce“ vykonané v jednom kroku je omezeno

Nyní se zaměříme na algoritmy, které pracují **paralelně**:

- jsou vykonávány velkým počtem procesorů
- tyto procesory pracují všechny současně — v jednom okamžiku je tak možné provést množství „práce“ úměrné počtu těchto procesorů
- paralelní provedení výpočtu tak může být výrazně rychlejší než sekvenční
- oproti sekvenčním algoritmům vyvstávají komplikace související se vzájemnou komunikací a synchronizací více procesorů

Paralelní algoritmy

Různé druhy výpočetních modelů pro paralelní a distribuované algoritmy je možné velmi zhruba rozdělit podle následujících kategorií:

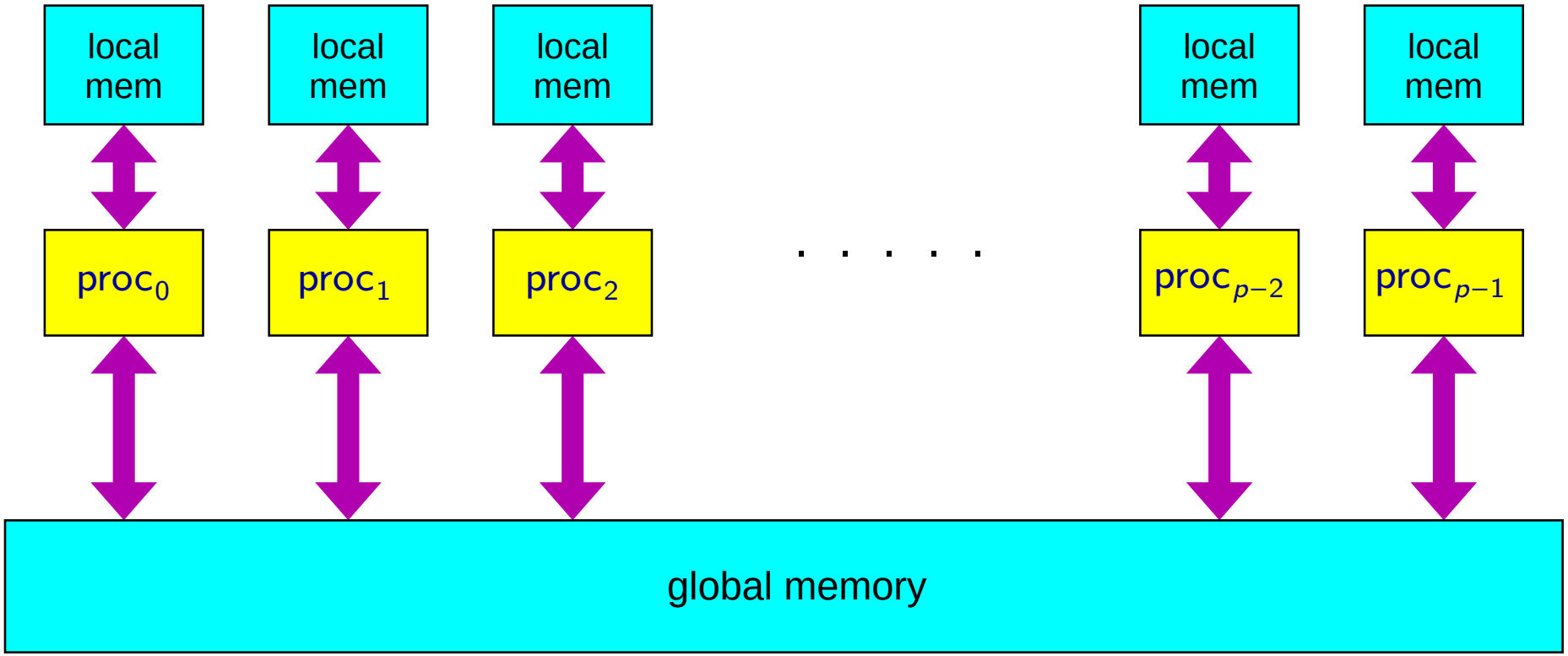
- Způsob vzájemné komunikace:
  - **sdílená paměť** — procesory sdílí společnou paměť, kam mohou všechny zapisovat a ze které mohou všechny číst
  - **posílání zpráv** — procesory jsou propojeny nějakým druhem komunikační sítě, přes kterou si mohou posílat zprávy
- Vzájemná synchronizace:
  - **synchronní** — všechny procesory jsou řízeny společným „hodinovým signálem“, instrukce jsou vykonávány na všech procesorech vždy všechny najednou přesně ve stejný okamžik
  - **asynchronní** — žádná synchronizace neexistuje, procesory mohou pracovat různě rychle, pořadí provedení operací provedených na různých procesorech není předvidatelné

PRAM

Zde se zaměříme na výpočetní model, který se nazývá **parallel random access machine** (**PRAM**):

- Jedná se o variantu strojů RAM, kde však máme k dispozici neomezený počet paralelně běžících procesorů.
- Každý z těchto procesorů vypadá velmi podobně jako běžný sekvenční stroj RAM, jaký jsme si popsali dříve.
- Tyto procesory sdílí **společnou globální paměť**.
- Kromě toho má každý procesor svou vlastní soukromou paměť.
- Procesory pracují **synchronně**.
- Procesory sdílí společný kód programu.
- Každý procesor má vlastní řídící jednotku — různé procesory tedy mohou ve stejný okamžik vykonávat různé instrukce programu.

PRAM


local mem
local mem
local mem
local mem
local mem
proc$_0$
proc$_1$
proc$_2$
. . . . .
proc$_{p-2}$
proc$_{p-1}$
global memory

PRAM

- Lokální i globální paměť jsou tvořeny buňkami, které jsou adresovány přirozenými čísly a obsahují jako své hodnoty celá čísla.
- Většina instrukcí stroje PRAM je stejných jako u standardního stroje RAM (přiřazení, aritmetické operace, podmíněné a nepodmíněné skoky) — tyto instrukce pracují s lokální pamětí procesoru
- Kromě instrukcí **load** a **store** pro práci s **lokální pamětí**:

$$R_i := [R_j] \qquad [R_i] := R_j$$

  má PRAM i instrukce **load** a **store** pro přístup ke **globální paměti**:

$$R_i := [R_j]_{glob} \qquad [R_i]_{glob} := R_j$$

PRAM

- PRAM nemá instrukce pro vstup a výstup:
  - předpokládá se, že vstupní data jsou na začátku uložena na nějakém stanoveném místě v globální paměti
  - podobně se předpokládá, že výstup bude na konci výpočtu zapsán na nějaké stanoveném místě v globální paměti
- procesory jsou indexovány — každý má přiděleno ID (0, 1, 2, ...):
  - každý procesor zná své ID a může ho používat při výpočtech (např. při indexování buněk globální sdílené paměti)
  - pro jednoduchost můžeme např. předpokládat, že na začátku výpočtu má každý procesor uloženo své ID v buňce na adrese 0 ve své lokální paměti

PRAM

- K dispozici je neomezený počet procesorů. V každém okamžiku však běží jen konečný počet z nich.
- Většinou se při popisu algoritmů předpokládá, že na začátku je automaticky spuštěno $p(n)$ procesorů (s ID $0, 1, \ldots, p(n)-1$), kde n je velikost vstupu a $p(n)$ je nějaká konkrétní funkce přiřazující velikosti vstupu počet procesorů, které budou použity pro výpočet.

  Počet použitých procesorů tak roste v závislosti na velikosti vstupu.
- Někdy se může uvažovat také varianta stroje PRAM, která má speciální instrukci FORK, která umožňuje aktivovat nějaký dosud nepoužitý procesor a spustit na něm provádění kódu od nějaké specifikované adresy.

  V této variantě se typicky předpokládá, že při spuštění programu se začne provádět tento program nejprve pouze na procesoru 0. Všechny ostatní procesory jsou na začátku neaktivní a jsou postupně spouštěny dynamicky během vykonávání programu.

PRAM

Jak bylo uvedeno, u strojů PRAM se předpokládá, že pracují **synchronně**.

- Jedna jednoduchá představa je, že v jednom kroku vždy všechny běžící procesory najednou vykonají každý jednu instrukci.

Ve skutečnosti většinou při popisu paralelních algoritmů pro PRAM není nutné předpokládat, že jsou procesory synchronizovány až na úroveň jednotlivých elementárních instrukcí.

Vhodnější je představa, že výpočet vypadá tak, že se neustále dokola střídají následující tří fáze výpočtu:

- **čtení z globální paměti** — všechny procesory najednou čtou z globální paměti (každý z nich čte nejvýše jednu buňku)
- **výpočet v lokální paměti** — každý procesor provádí výpočet, při kterém nepřistupuje ke globální paměti, a kdy provede $\mathcal{O}(1)$ kroků
- **zápis do globální paměti** — všechny procesory najednou zapisují do globální paměti (každý z nich zapisuje nanejvýš do jedné buňky)

Varianty strojů PRAM

Rozlišují se tři základní varianty strojů PRAM:

- **EREW — Exclusive-Read Exclusive-Write**:

  Je zakázáno, aby v jednom okamžiku přistupovalo více procesorů k jedné buňce.

  Pokud se najednou dva nebo více procesorů pokusí číst obsah jedné a té samé buňky nebo zapsat do stejné buňky, je to považováno za chybu — výpočet programu končí chybou.

- **CREW — Concurrent-Read Exclusive-Write**:

  Z jedné buňky může najednou číst libovolný počet procesorů.

  Zapisovat do jedné buňky však může v jednom okamžiku vždy jen jeden procesor.

- **CRCW — Concurrent-Read Concurrent-Write**:

  Při čtení i zápisu může k jedné buňce najednou přistupovat libovolný počet procesorů.

Varianty strojů PRAM

Varianta **CRCW** (**Concurrent-Read Concurrent-Write**) se dále dělí na následující tři podvarianty:

- **COMMON** — pokud do jedné buňky zapisuje najednou více procesorů, musí všechny zapisovat tutéž hodnotu.
  Pokud tomu tak není, je to považováno za chybu.
- **ARBITRARY** — pokud do jedné buňky zapisuje najednou více procesorů, je nedeterministicky vybrán jeden z nich a jim zapisovaná hodnota je do buňky skutečně zapsána.
  Hodnoty zapisované ostatními procesory jsou ignorovány.

  Nelze nic předpokládat o tom, který z procesorů bude vybrán.
- **PRIORITY** — pokud do jedné buňky zapisuje najednou více procesorů, je vybrán procesor s nejnižším ID a jím zapisovaná hodnota je do dané buňky skutečně zapsána.

  Tj. při zápisu mají procesory s nižším ID vyšší prioritu.

PRAM

Stroje PRAM jsou idealizovaným modelem paralelního počítače se sdílenou pamětí:

- V realitě je typicky počet dostupných procesorů většinou fixní a mnohem menší než velikost instancí, pro které chceme problém řešit.
- V realitě neexistuje implementace sdílené paměti, u které by se dala zaručit konstantou omezená doba přístupu nezávisející na tom, jak velký počet procesorů bude k paměti najednou přistupovat.
- Stroje PRAM slouží jako vhodný model pro prvotní, na konkrétním hardware co nejvíce nezávislý, návrh paralelních algoritmů, kdy zpočátku ignorujeme režii vzájemné komunikace mezi procesory a technické detaily s tím spojené.
- Při navrhování algoritmů pro PRAM jde hlavně o to, prozkoumat, jak dalece je možné řešení daného problému paralelizovat za ideálních podmínek.

PRAM

- Algoritmy pro PRAM tak mohou sloužit jako výchozí bod pro návrh paralelních algoritmů, které budou určeny pro více realistický a konkrétněji popsaný hardware, které budou detailněji řešit otázky související s distribucí práce na jednotlivé procesory a vzájemnou komunikaci mezi těmito procesory.

PRAM

- Při návrhu paralelních algoritmů pro PRAM se obvykle popisují tyto algoritmy pomocí pseudokódu nebo slovním popisem.

  Téměř nikdy se nepopisují až na úroveň jednotlivých elementárních instrukcí.

  Konkrétní detaily instrukcí a činnosti uvažovaného stroje PRAM tak většinou nejsou příliš důležité.

  Významné jsou však rozdíly mezi výše popsanými variantami EREW, CREW, CRCW a podvariantami COMMON, ARBITRARY a PRIORITY, které je třeba při návrhu algoritmů brát v úvahu.

Výpočetní složitost paralelních algoritmů

U paralelních algoritmů zamýšlených pro běh na stroji PRAM se z hlediska jejich výpočetní složitosti zkoumají dva parametry:

- **Doba výpočtu** — celkový počet kroků, přičemž se předpokládá, že všechny procesory provádějí krok najednou

  (U strojů PRAM se prakticky vždy uvažuje jednotková míra, kdy se počítá jen počet provedených operací, ne to, kolik bitů mají operandy zpracovávané při těchto operacích.)

- **Počet procesorů** — celkový počet procesorů, který je k dosažení dané doby výpočtu potřeba

Doba výpočtu (tj. **časová složitost**) i **počet procesorů** se pro daný algoritmus vyjadřují jako funkce $t(n)$ a $p(n)$, vyjadřující závislost doby výpočtu a počtu procesorů na velikosti vstupu n.

Typicky se také nevyjadřují přesně, ale jako asymptotické odhady.

Efektivní paralelní algoritmy

Za **efektivní paralelní algoritmy** jsou považovány ty, kde:

- **Časová složitost** $t(n)$ je **polylogaritmická**, tj. $\mathcal{O}(\log^k n)$, kde k je konstanta.
  (**Poznámka:** $\log^k n$ je zkrácený zápis pro $(\log n)^k$.)
- **Počet procesorů** $p(n)$ je **polynomiální**.

Problémy, pro které existuje takový efektivní paralelní algoritmus, jsou považovány za **dobře paralelizovatelné**.

Definice

Třídu NC (Nick's class) tvoří právě ty rozhodovací problémy, pro které existuje paralelní algoritmus s polylogaritmickou časovou složitostí při použití polynomiálního počtu procesorů.

Součet posloupnosti čísel

Příklad: Uvažujme následující problém:

Vstup: Posloupnost čísel $a_0, a_1, \ldots, a_{n-1}$.

Výstup: Součet $s = a_0 + a_1 + \cdots + a_{n-1}$.

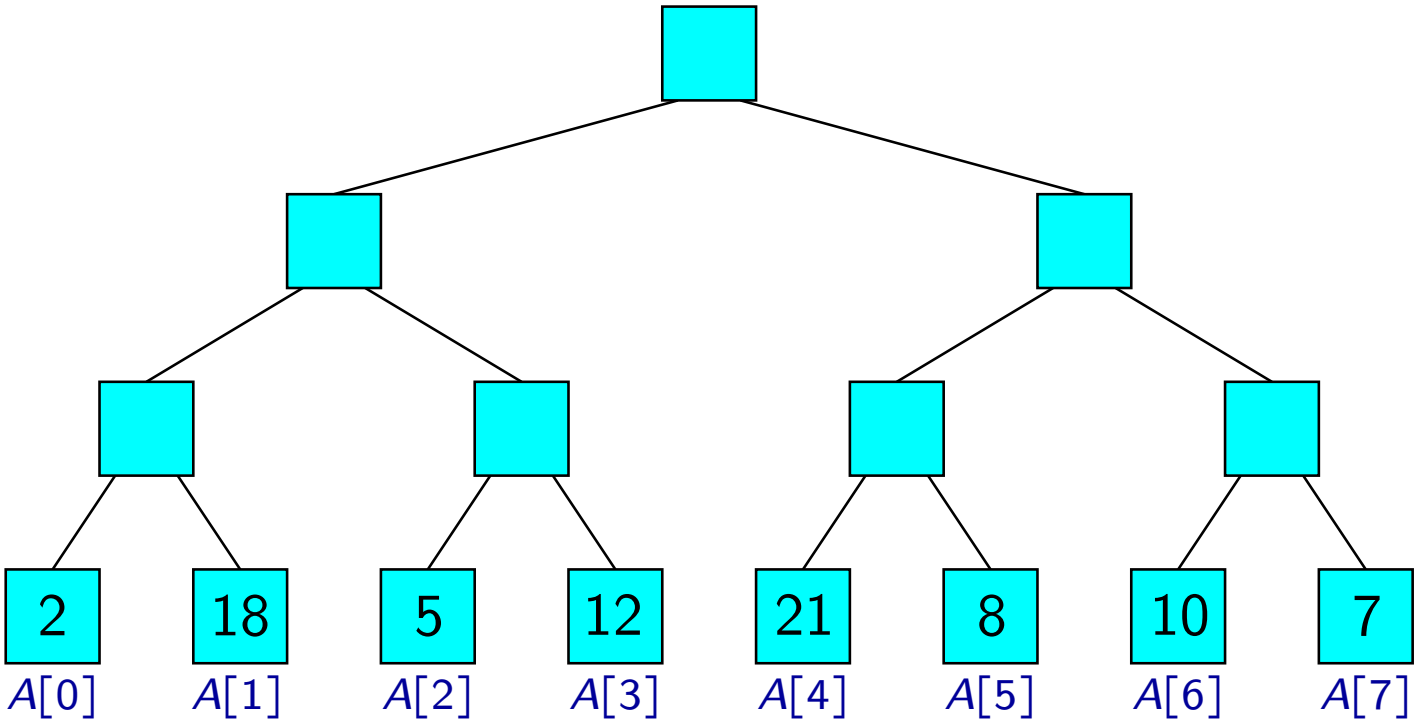
Poznámka: Pro jednoduchost předpokládejme, že $a_0, a_1, \ldots, a_{n-1}$ jsou „malá“ čísla, takže součet jakýchkoli dvou těchto čísel je možné provést v čase $\mathcal{O}(1)$, a každé z těchto čísel může být uloženo v jedné buňce paměti.

Předpokládejme, že na začátku jsou hodnoty $a_0, a_1, \ldots, a_{n-1}$ zapsány v globální sdílené paměti v prvcích pole A (tj. v prvcích $A[0]$, $A[1]$,..., $A[n-1]$).

Po skončení výpočtu bude výsledek zapsán v globální proměnné s (která se nachází v globální paměti).

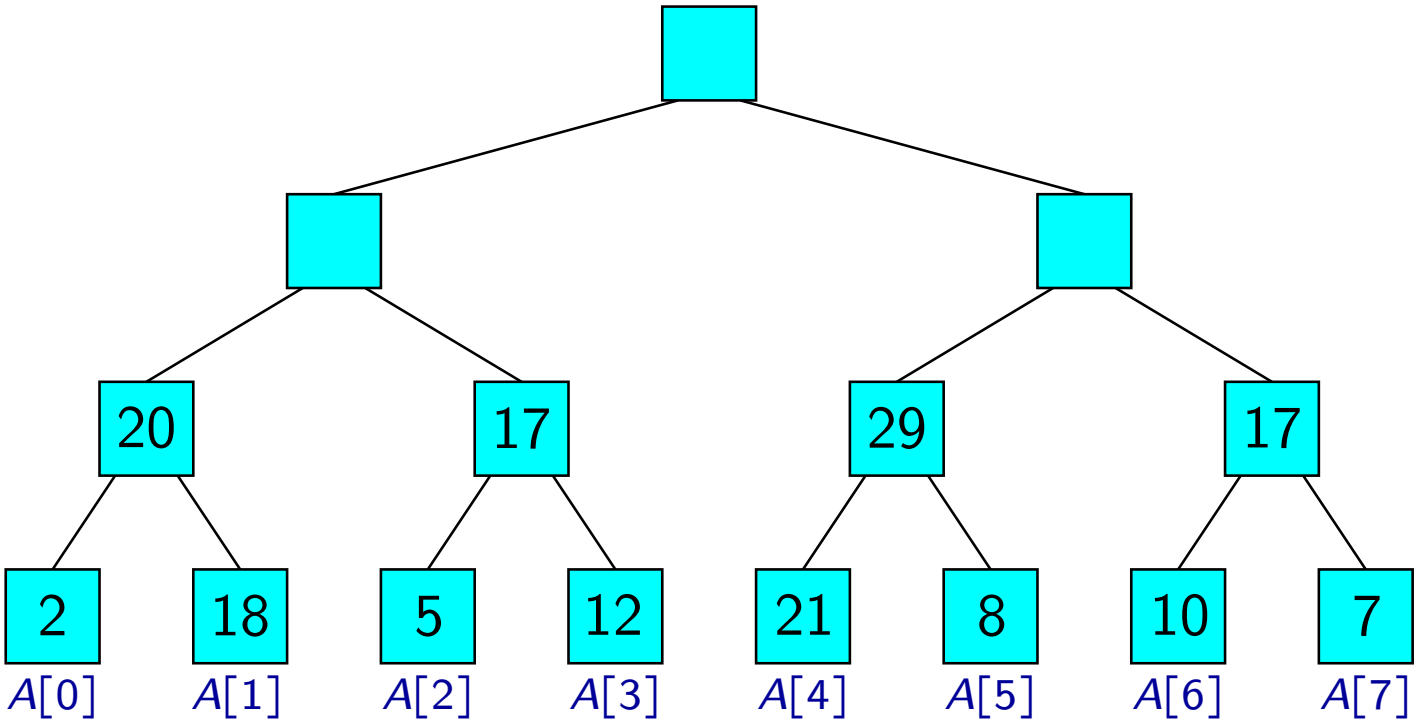
Součet posloupnosti čísel

Výpočet paralelního algoritmu si můžeme představit ve formě vyváženého binárního stromu, jehož listy odpovídají prvkům pole A.



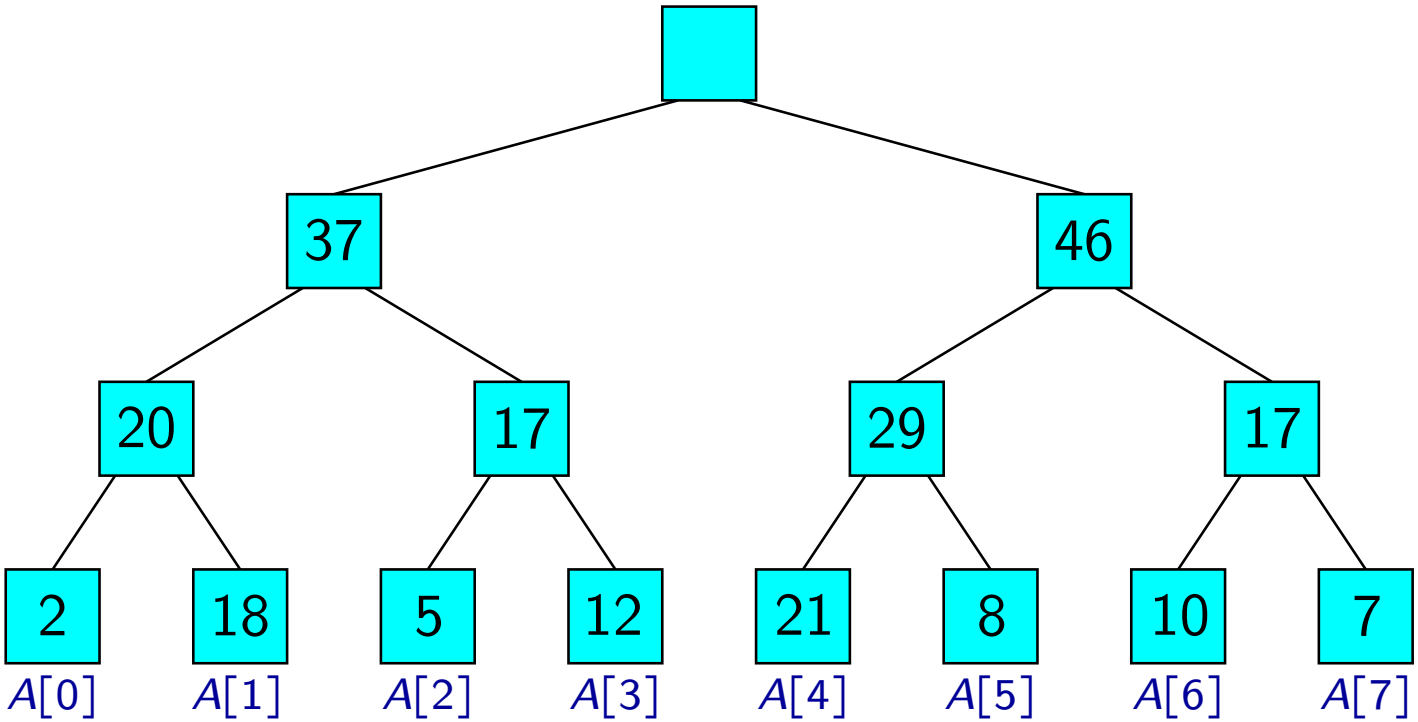
Součet posloupnosti čísel

Výpočet paralelního algoritmu si můžeme představit ve formě vyváženého binárního stromu, jehož listy odpovídají prvkům pole A.



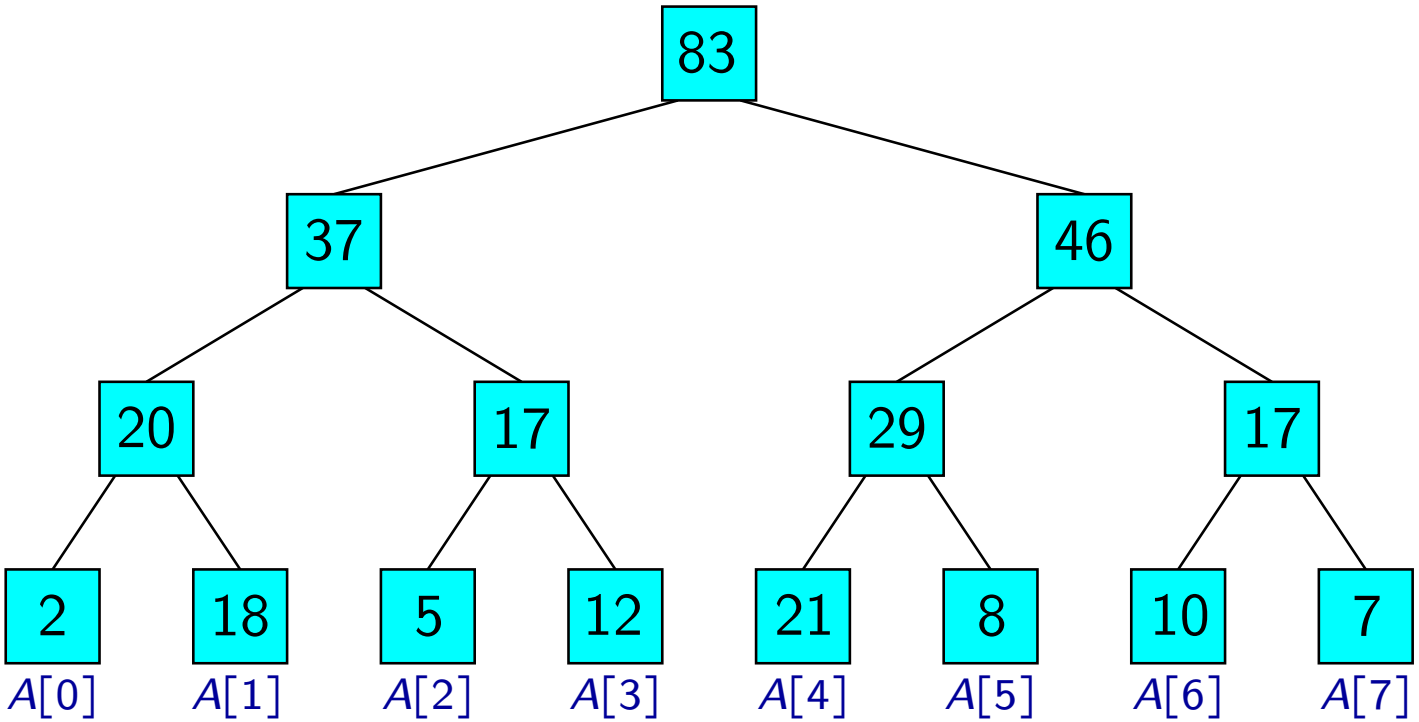
Součet posloupnosti čísel

Výpočet paralelního algoritmu si můžeme představit ve formě vyváženého binárního stromu, jehož listy odpovídají prvkům pole A.



Součet posloupnosti čísel

Výpočet paralelního algoritmu si můžeme představit ve formě vyváženého binárního stromu, jehož listy odpovídají prvkům pole A.



Součet posloupnosti čísel

Tento postup je možné implementovat například takto (předpokládejme pro jednoduchost, že n je mocnina dvojky):

- $k := 1$; $m := n/2$
- **while** $(k < n)$:
  - procesory s ID i, kde $0 \leq i < m$, provedou:
    $$A[i \cdot 2k] := A[i \cdot 2k] + A[i \cdot 2k + k]$$
  - $k := 2k$; $m := m/2$
- $s := A[0]$

Je zřejmé, že cyklus **while** proběhne $\log_2 n$ krát.

Celkem bylo použito $n/2$ procesorů.

Časová složitost tohoto algoritmu je $\mathcal{O}(\log n)$ při použití $\mathcal{O}(n)$ procesorů.

Součet posloupnosti čísel

Řekněme, že bychom neměli k dispozici $n/2$ procesorů, ale pouze p procesorů, kde p by bylo mnohem menší než n.

Paralelní algoritmus používající p procesorů, by mohl pracovat takto:

- Rozdělit pole A na p úseků, z nichž každý má délku $\lceil n/p \rceil$ nebo $\lfloor n/p \rfloor$.
- Každý úsek bude přidělen jednomu z p procesorů.
  Každý procesor spočítá součet čísel ve svém úseku.
- Součty sečtené jednotlivými procesory se sečtou dříve popsaným algoritmem, kterému bude stačit $p/2$ procesorů.

Časová složitost tohoto algoritmu je $\mathcal{O}(n/p + \log p)$ při použití p procesorů, kde $p \leq n/2$.

Brentova věta

Následující věta platí pro různé druhy paralelních algoritmů a netýká se jen strojů PRAM.

Věta (Brent)

Řekněme, že máme paralelní algoritmus, který vykoná celkem m operací, a kde doba jeho provádění při neomezeném počtu procesorů by byla t kroků.

Pokud bude k dispozici pouze p procesorů, je možné implementovat tento algoritmus tak, aby pro počet kroků t' platilo

$$t' \leq t + \frac{m - t}{p}$$

Brentova věta

Důkaz: Řekněme, že pro počty operací, které mohou být provedeny paralelně ve krocích $i = 1, 2, \ldots, t$, jsou m_1, m_2, $\ldots$, m_t.

Platí tedy

$$\sum_{i=1}^{t} m_i = m$$

V každém kroku i je možné provedení m_i operací rovnoměrně rozdělit mezi p procesorů:

- Provedení všech operací i-tého kroku tedy bude na p procesorech trvat $\lceil m_i/p \rceil$ kroků.

Brentova věta

Pro celkovou dobu simulace tedy bude platit:

$$t' = \sum_{i=1}^{t} \left\lceil \frac{m_i}{p} \right\rceil \leq \sum_{i=1}^{t} \frac{m_i + p - 1}{p} = \left(\sum_{i=1}^{t} \frac{m_i}{p} \right) + \left(\sum_{i=1}^{t} \frac{p-1}{p} \right)$$

$$= \frac{1}{p} \left(\sum_{i=1}^{t} m_i \right) + \frac{p-1}{p} \cdot t = \frac{m}{p} + t - \frac{t}{p} = t + \frac{m-t}{p}$$

Poznámka: Připomeňme, že pro libovolná přirozená čísla a a b platí

$$\left\lceil \frac{a}{b} \right\rceil \leq \frac{a+b-1}{b}$$

Optimální paralelní algoritmy

Efektivní paralelní algoritmus je považován za **optimální**, jestliže celkový počet provedených operací je (asymptoticky) stejný, jako je časová složitost nejlepšího známého sekvenčního algoritmu, který řeší daný problém.

Příklad: Dříve popsaný algoritmus pro počítání součtu posloupnosti čísel provede celkem $\mathcal{O}(n)$ operací.

Je zjevné, že jakýkoli sekvenční algoritmus musí provést $\Omega(n)$ operací.

Tento paralelní algoritmus je tedy v tomto smyslu optimální.

Hledání minima v čase $\mathcal{O}(1)$

Viděli jsme paralelní algoritmus pro spočítání součtu sekvence čísel pracující na stroji PRAM typu EREW:

- Časová složitost tohoto algoritmu byla $\mathcal{O}(n/p + \log p)$ při použití p procesorů, kde $p \leq n/2$.
- Při použití $n \,/\, \log_2 n$ procesorů bude tedy jeho časová složitost $\mathcal{O}(\log n)$.

Uvažujme následující problém:

Hledání minima z dané posloupnosti

Vstup: Posloupnost čísel $a_0, a_1, \ldots, a_{n-1}$.

Výstup: Minimum z této posloupnosti, tj. hodnota $\min\{a_0, a_1, \ldots, a_{n-1}\}$.

Hledání minima v čase $\mathcal{O}(1)$

Paralelní algoritmus pro stroj PRAM typu EREW pro hledání minima bude pracovat velmi podobně jako algoritmus pro počítání součtu a bude mít stejnou výpočetní složitost.

Na stroji PRAM typu **CRCW** ve variantě **COMMON** je možné najít minimum ze sekvence prvků v čase $\mathcal{O}(1)$, přičemž počet použitých procesorů bude polynomiální:

- Předpokládejme, že prvky $a_0, a_1, \ldots, a_{n-1}$ jsou uloženy v poli A (velikosti n).
- Algoritmus bude používat pomocné pole C rovněž velikosti n, jehož prvky budou booleovské hodnoty (typu $\mathrm{Bool} = \{0, 1\}$).
- Pole C bude inicializováno hodnotami 0 — na tuto inicializaci stačí n procesorů.

Hledání minima v čase $\mathcal{O}(1)$

- Každé dvojici čísel i a j, kde $0 \leq i < j < n$, bude přiřazen jeden procesor.

  Celkový počet procesorů bude tedy $\mathcal{O}(n^2)$.

- Všechny tyto procesory provedou paralelně následující:

  Procesor přiřazený dvojici (i, j) (kde $0 \leq i < j < n$):

---

if $A[i] > A[j]$ **then**
  $C[i] := 1$
else
  $C[j] := 1$

- Je zjevné, že prvek $C[i]$ bude mít hodnotu 1 právě tehdy, když:
  - existuje nějaký prvek j, pro který platí $A[i] > A[j]$, nebo
  - existuje nějaký prvek j, pro který platí $A[i] = A[j]$ a $j < i$.

Hledání minima v čase $\mathcal{O}(1)$

- Po provedení výše uvedených instrukcí zbude jediný prvek i, pro který bude platit $C[i] = 0$.

  Tento prvek bude splňovat následující pro všechny j, kde $j \neq i$:

  - $A[i] \leq A[j]$
  - pokud $A[i] = A[j]$, tak $i < j$

- n procesorů pak pro $i = 0, 1, \ldots, n-1$ paralelně otestuje, zda platí $C[i] = 0$.

  Jediný procesor, pro který bude tato podmínka splněna, uloží hodnotu $A[i]$ jako hodnotu výstupu algoritmu.

Hledání minima v čase $\mathcal{O}(1)$

Na stroji PRAM typu CRCW COMMON je možné s $\mathcal{O}(n^2)$ procesory najít minimum v čase $\mathcal{O}(1)$.

Poznámka: Tento algoritmus ovšem není optimální, protože celkové množství provedených operací je $\Theta(n^2)$, zatímco přímočarý sekvenční algoritmus, který sekvenčně projde prvky pole, najde minimum v čase $\mathcal{O}(n)$.

Podobným způsobem je možné v konstantním čase s polynomiálním počtem procesorů (např. $\mathcal{O}(n^2)$ nebo $\mathcal{O}(n)$) na stroji PRAM CRCW COMMON řešit i některé další úlohy, např.:

- nalezení maxima v poli
- provedení operace **and** ($\wedge$) mezi všemi prvky pole tvořeného booleovskými hodnotami
- provedení operace **or** ($\vee$) mezi všemi prvky pole tvořeného booleovskými hodnotami

Návrh paralelních algoritmů

Ukážeme si některé techniky často používané při návrhu efektivních paralelních algoritmů:

- zdvojování
- spočítání výsledku nějaké asociativní binární operace pro všechny prefixy daného pole

Zdvojování

Řekněme, že máme dán jeden nebo více stromů, které jsou reprezentovány následujícím způsobem:

- Vrcholy jsou označeny indexy (např. $1, 2, \ldots, n$).

  Tyto indexy nejsou v žádném vztahu ke struktuře jednotlivých stromů ani k tomu, který vrchol patří do kterého stromu
  — pořadí jednotlivých vrcholů dané jejich indexy je zcela libovolné.

- Pole *parent* reprezentuje indexy rodičů jednotlivých vrcholů:
  - Pro každý vrchol i hodnota $parent[i]$ udává index jeho rodiče.
  - Pokud je vrchol i kořenem daného stromu, bude pro něj platit $parent[i] = i$.

Řekněme, že pro každý vrchol i chceme spočítat nějakou informaci, pro kterou je třeba projít od vrcholu i ke kořeni daného stromu, např.:

- vzdálenost od i ke kořeni daného stromu

Zdvojování

Algoritmus bude pracovat takto:

- **Inicializace:**

  Pro každý vrchol i se paralelně provede:

  $P[i] := parent[i]$
  if $P[i] \neq i$ **then** $dist[i] := 1$ **else** $dist[i] := 0$

- **Hlavní cyklus:**

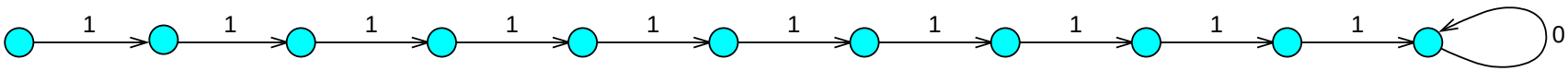
  $\lceil \log_2 n \rceil$ krát se zopakuje následující — paralelně pro každý vrchol i:

  ---

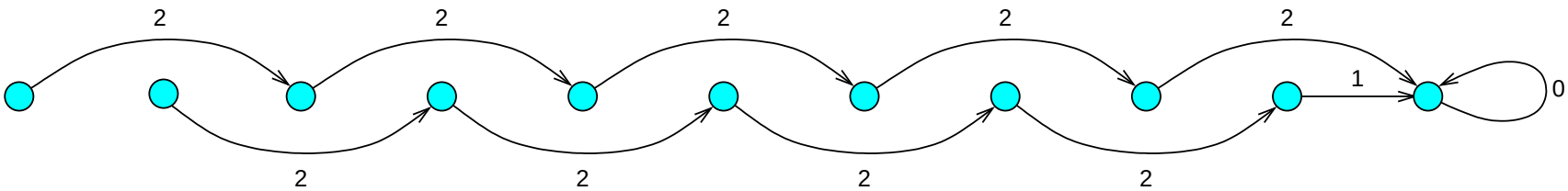
  if $P[P[i]] \neq P[i]$ **then**
  $dist[i] := dist[i] + dist[P[i]]$
  $P[i] := P[P[i]]$

Poznámka: Předpokládá se PRAM typu CREW.

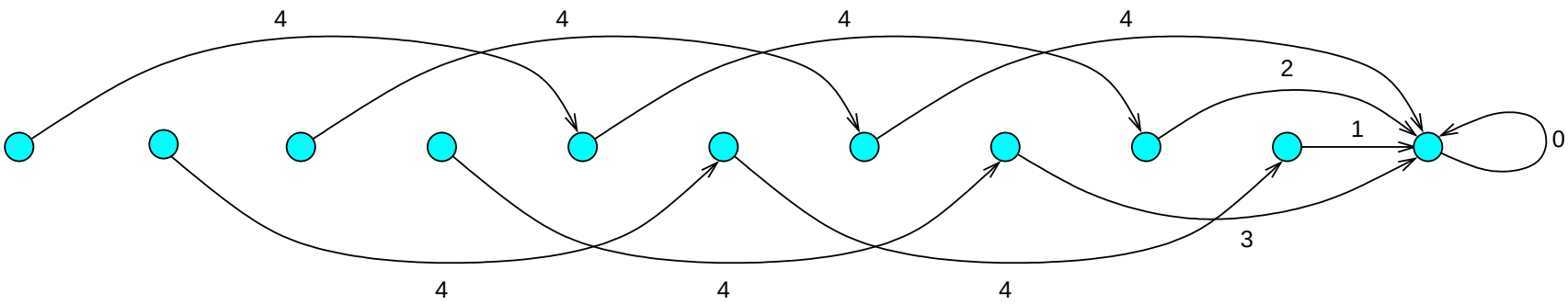
Zdvojování



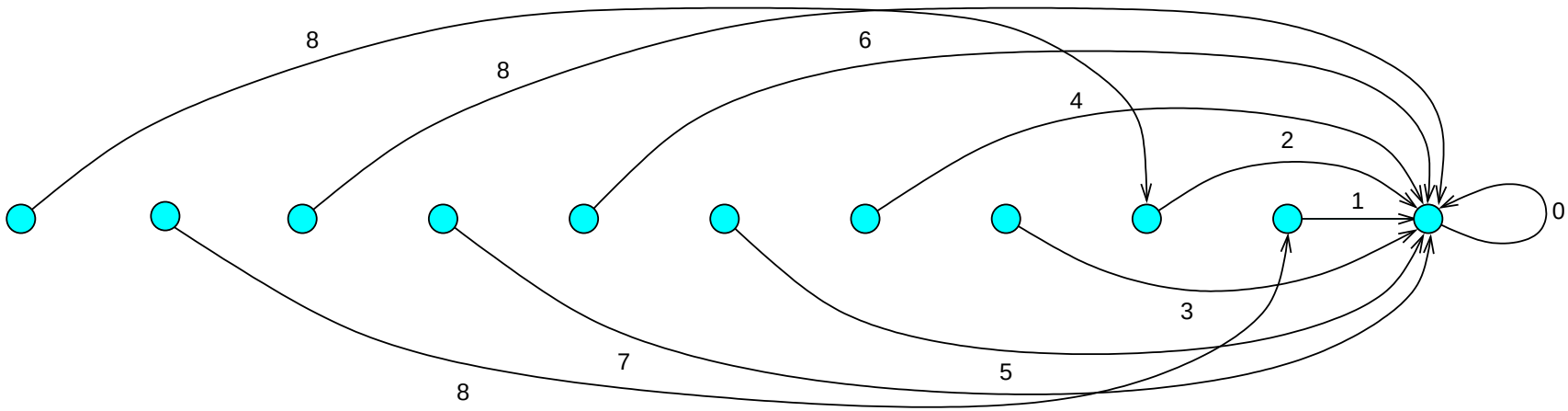
Zdvojování



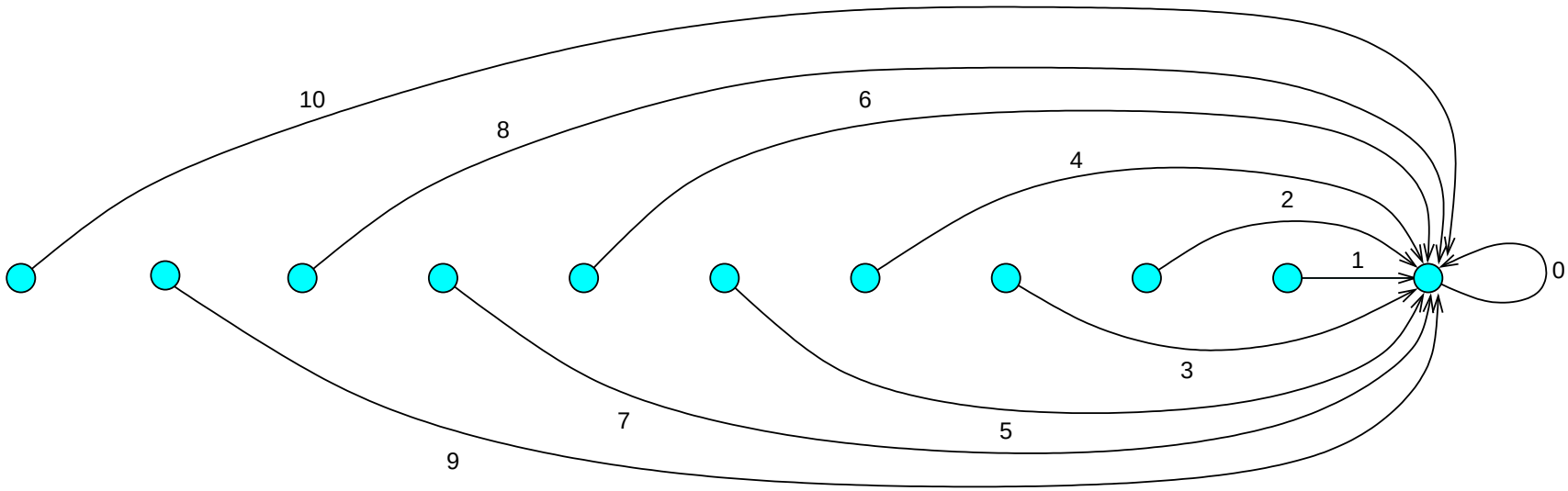
Zdvojování



Zdvojování



Zdvojování



Zdvojování

Je zjevné, že při použití n procesorů provede výše uvedený algoritmus $\mathcal{O}(\log n)$ kroků.

Počítání součtů prefixů posloupnosti

Součty prefixů

Vstup: Posloupnost čísel $a_0, a_1, \ldots, a_{n-1}$.

Výstup: Součty

$$s_i = \sum_{j=0}^{i-1} a_j$$

pro všechna $i = 1, 2, \ldots, n$.

Jednoduché řešení pracující s $\mathcal{O}(n^2)$ procesory v čase $\mathcal{O}(\log n)$:

- Na počítání každé z hodnot s_i vyčlenit samostatných i procesorů a použít pro ně dříve popsaný algoritmus pro počítání součtu sekvence čísel.

Tento algoritmus ovšem není optimální.

Počítání součtů prefixů posloupnosti

Popíšeme si způsob, jak si pro dosažení času $\mathcal{O}(\log n)$ vystačit s $\mathcal{O}(n)$ procesory, resp. jak při použití p procesorů, kde $p \leq n/2$, dosáhnout časové složitosti $\mathcal{O}(n/p + \log p)$

Algoritmus bude pracovat na stroji PRAM typu EREW.

Pro jednoduchost budeme předpokládat následující:

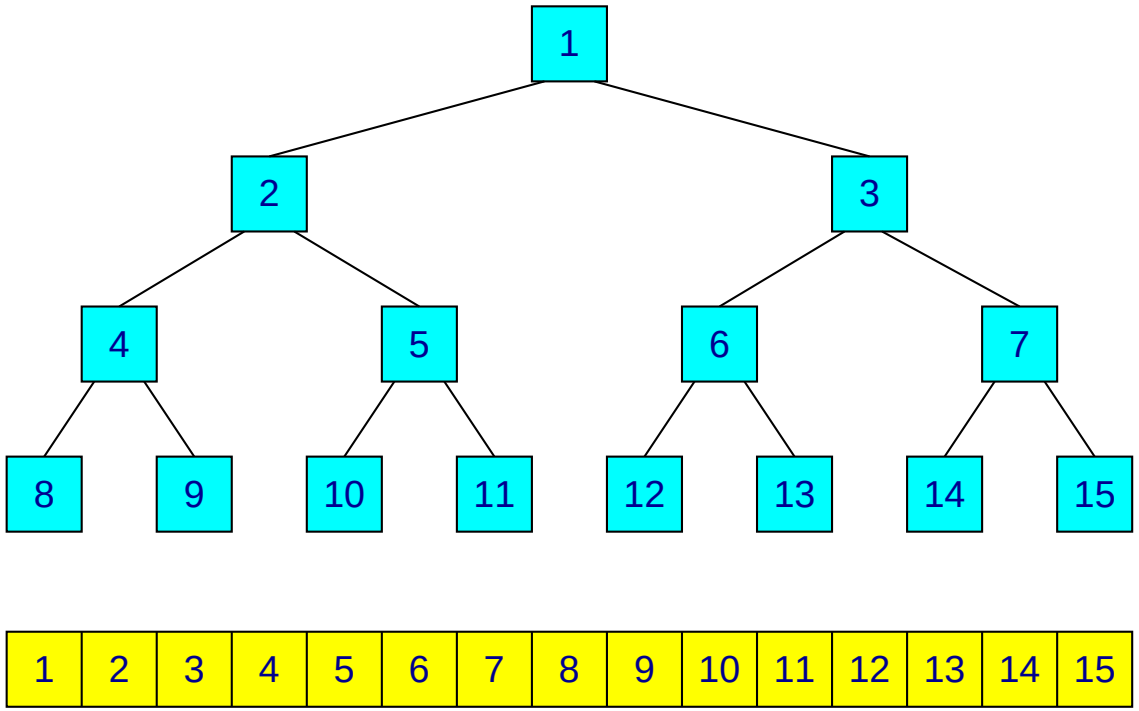
- n je mocnina dvojky, tj. $n = 2^m$ pro nějaké $m \in \mathbb{N}$

Tento algoritmus, podobně jako mnoho jiných efektivních paralelních algoritmů, používá jako datovou strukturu **vyvážený binární strom**:

- Hodnoty $a_0, a_1, \ldots, a_{n-1}$ budou na začátku uloženy v listech tohoto stromu.
- Hodnoty součtů $s_0, s_1, \ldots, s_{n-1}$ budou po skončení výpočtu také uloženy v listech tohoto stromu. (A hodnota s_n v kořeni.)

Uložení vyváženého binárního stromu v paměti

Vrcholy **vyváženého binárního stromu** mohou být uloženy v poli následujícím způsobem:



- Kořen je uložen v prvku s indexem 1.
- Potomci vrcholu s indexem i jsou uloženi v prvcích s indexy $2i$ a $2i + 1$.

Uložení vyváženého binárního stromu v paměti

Poznámky:

- Můžeme si představit, že strom je rozdělen na „vrstvy“ podle vzdálenosti od kořene — vrstva k je tvořena vrcholy, které mají vzdálenost od kořene k.
- Vrcholy z vrstvy k se nacházejí v poli v prvcích s indexy 2^k až $2^{k+1} - 1$.
- Indexy prvků pole v sobě obsahují informaci o cestě od kořene stromu k danému vrcholu i — v binárním zápise čísla i (bez nejvýznamnějšího bitu s hodnotou 1) jednotlivé bity reprezentují:
  - 0 — pokračovat do levého potomka
  - 1 — pokračovat do pravého potomka
- Pro vrchol i (kde $i > 1$) se index jeho rodiče spočítá jako $\lfloor i/2 \rfloor$.
- Strom, který má n listů, má celkem $2n - 1$ vrcholů.

Počítání součtů prefixů posloupnosti

Algoritmus bude používat dvě pole, jejichž prvky přísluší jednotlivým vrcholům vyváženého binárního stromu:

- pole A — v listech stromu jsou v poli A na začátku uloženy hodnoty prvků $a_0, a_1, \ldots, a_{n-1}$

  V ostatních vrcholech i bude po provedení výpočtu v $A[i]$ uložen součet hodnot ze všech listů podstromu, jehož kořenem je vrchol i.

- pole B — v prvcích $B[i]$ bude po skončení výpočtu uložen pro každý vrchol i součet všech hodnot a_j nacházejících se v celkovém stromu v listech nalevo od podstromu, jehož kořenem je vrchol i.

Počítání součtů prefixů posloupnosti

Předpokládejme, že $n = 2^m$. Algoritmus pracuje ve dvou fázích:

- V první fázi jsou spočítány hodnoty prvků pole A.

  Pro každou z vrstev $k = m-1, m-2, \ldots, 0$ se postupně provede následující:

  Pro vrcholy i s indexy 2^k až $2^{k+1}-1$ se paralelně provede:

  $A[i] := A[2*i] + A[2*i+1]$

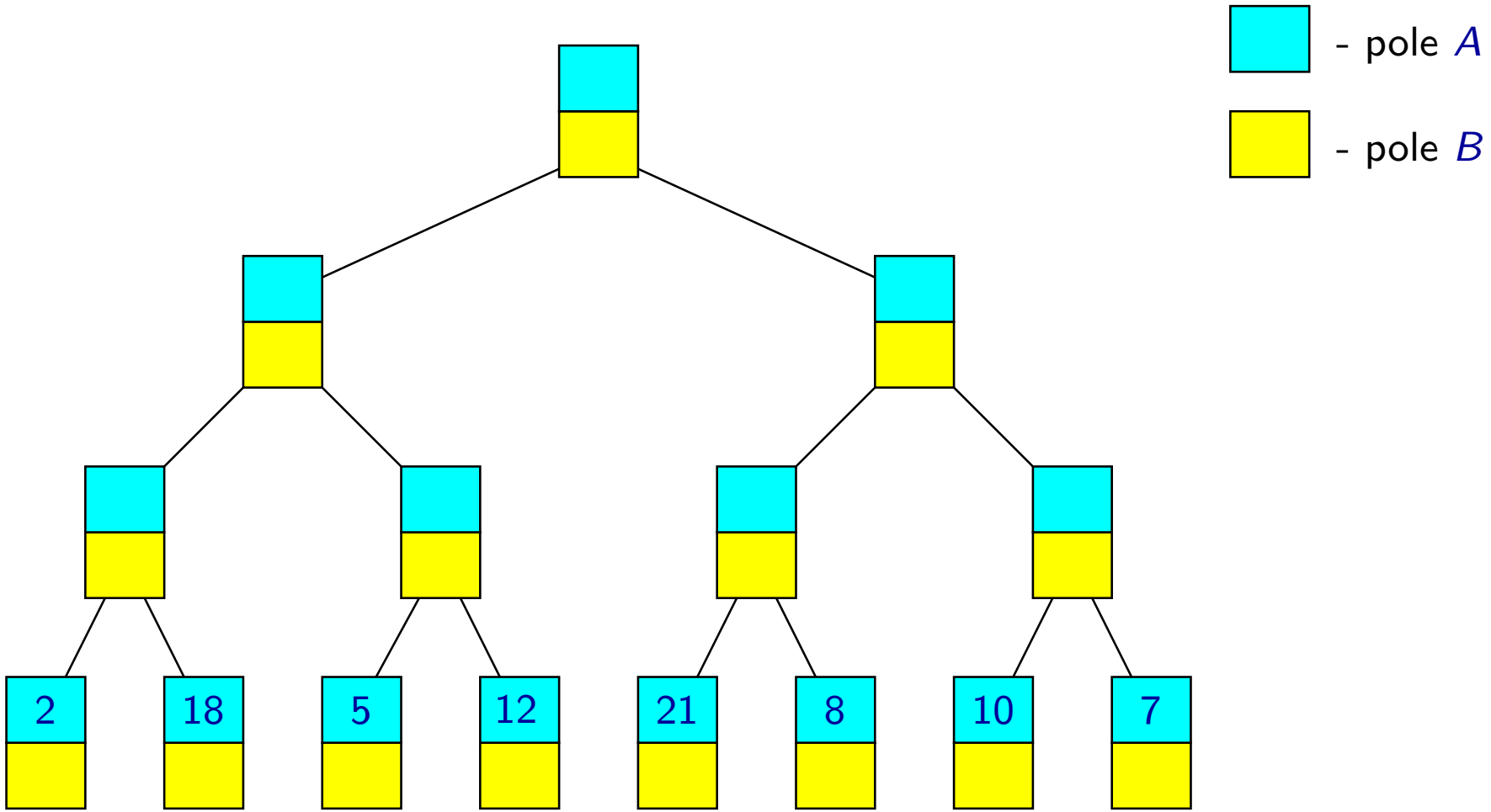
- V druhé fázi jsou spočítány hodnoty prvků pole B.

  Inicializuje se $B[1] := 0$.
  Pro každou z vrstev $k = 0, 1, \ldots, m-1$ se postupně provede následující:

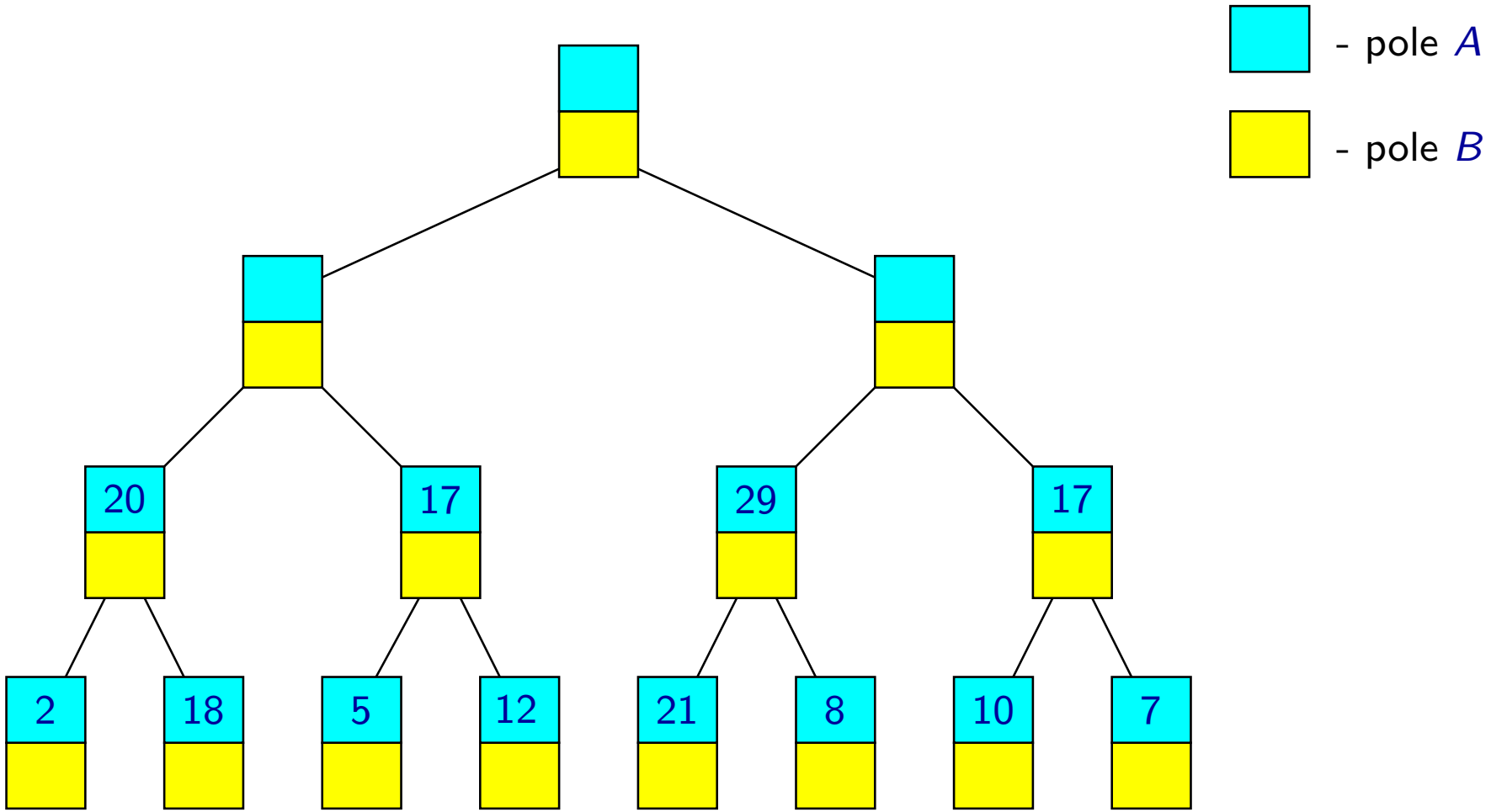
  Pro vrcholy i s indexy 2^k až $2^{k+1}-1$ se paralelně provede:

  $B[2*i] := B[i]$
  $B[2*i+1] := B[i] + A[2*i]$

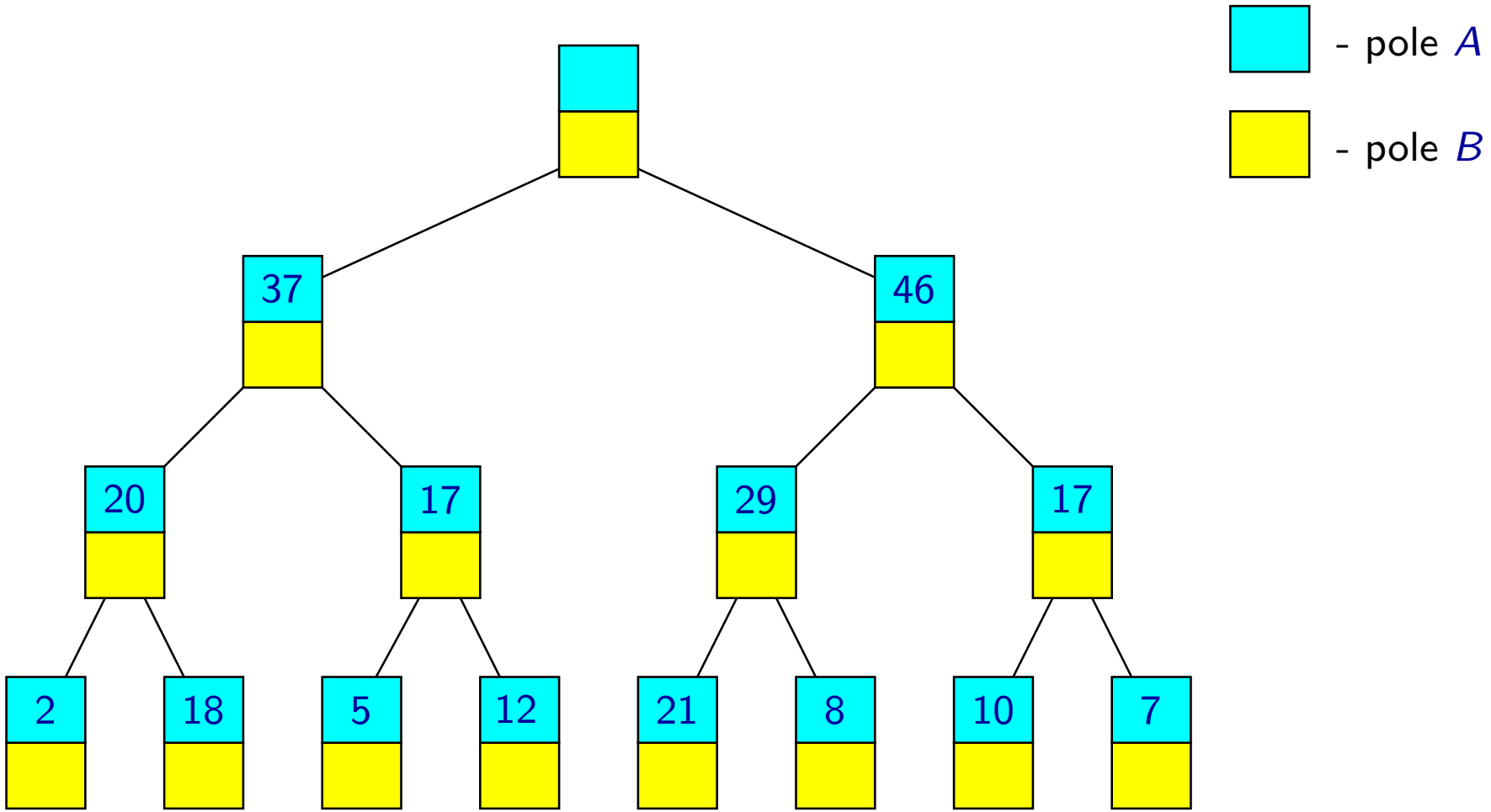
Počítání součtů prefixů posloupnosti



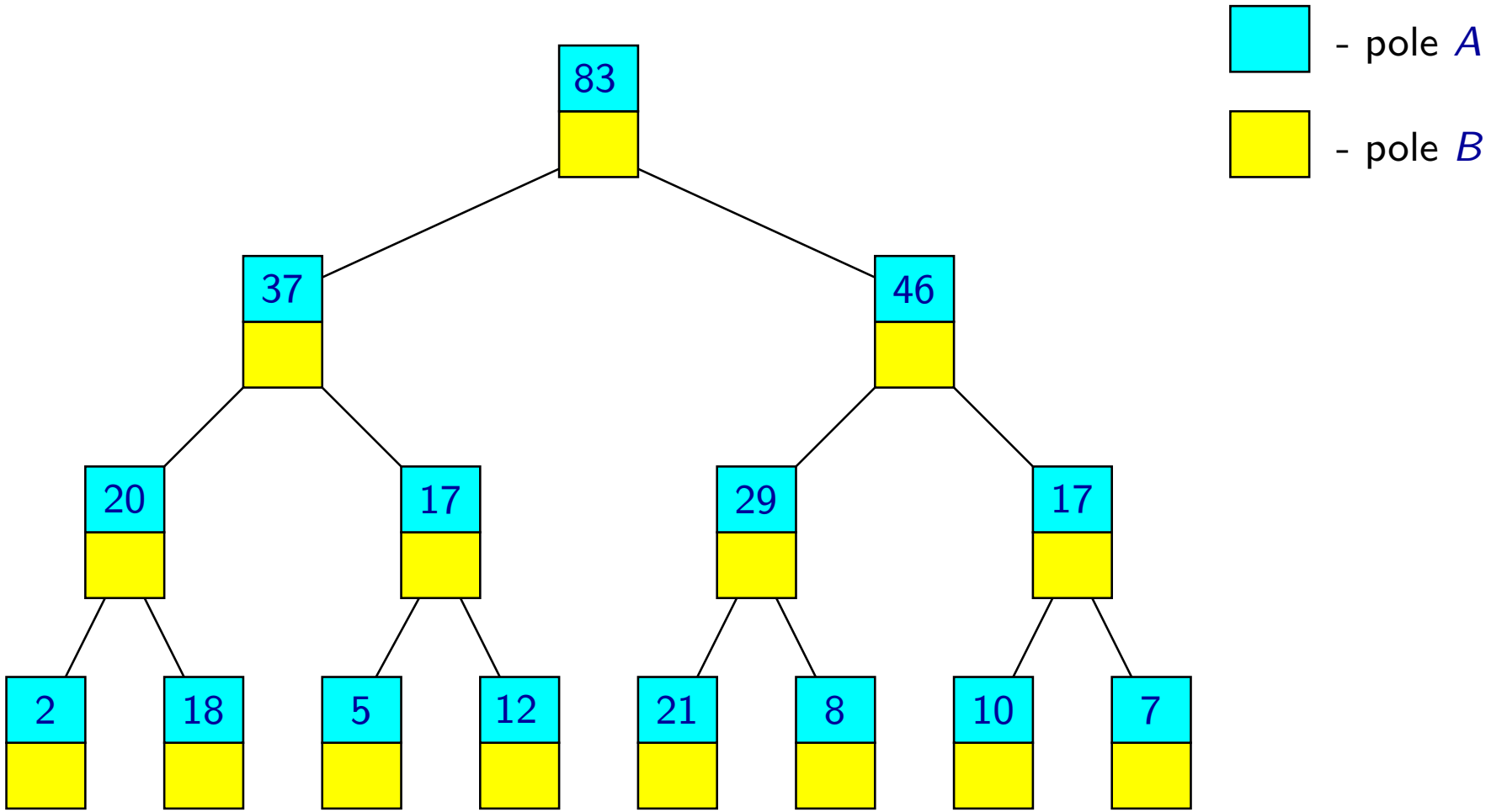
Počítání součtů prefixů posloupnosti



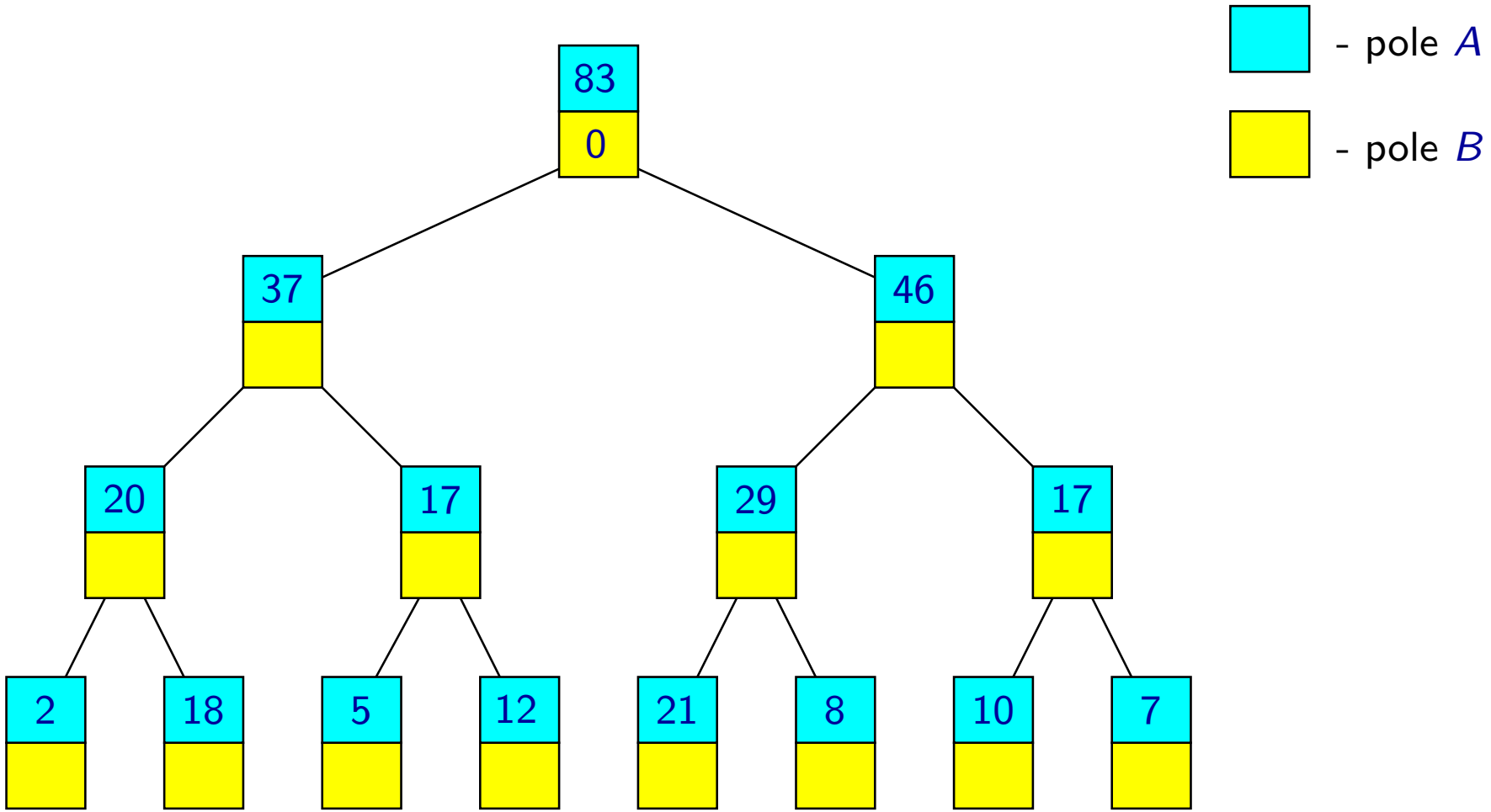
Počítání součtů prefixů posloupnosti



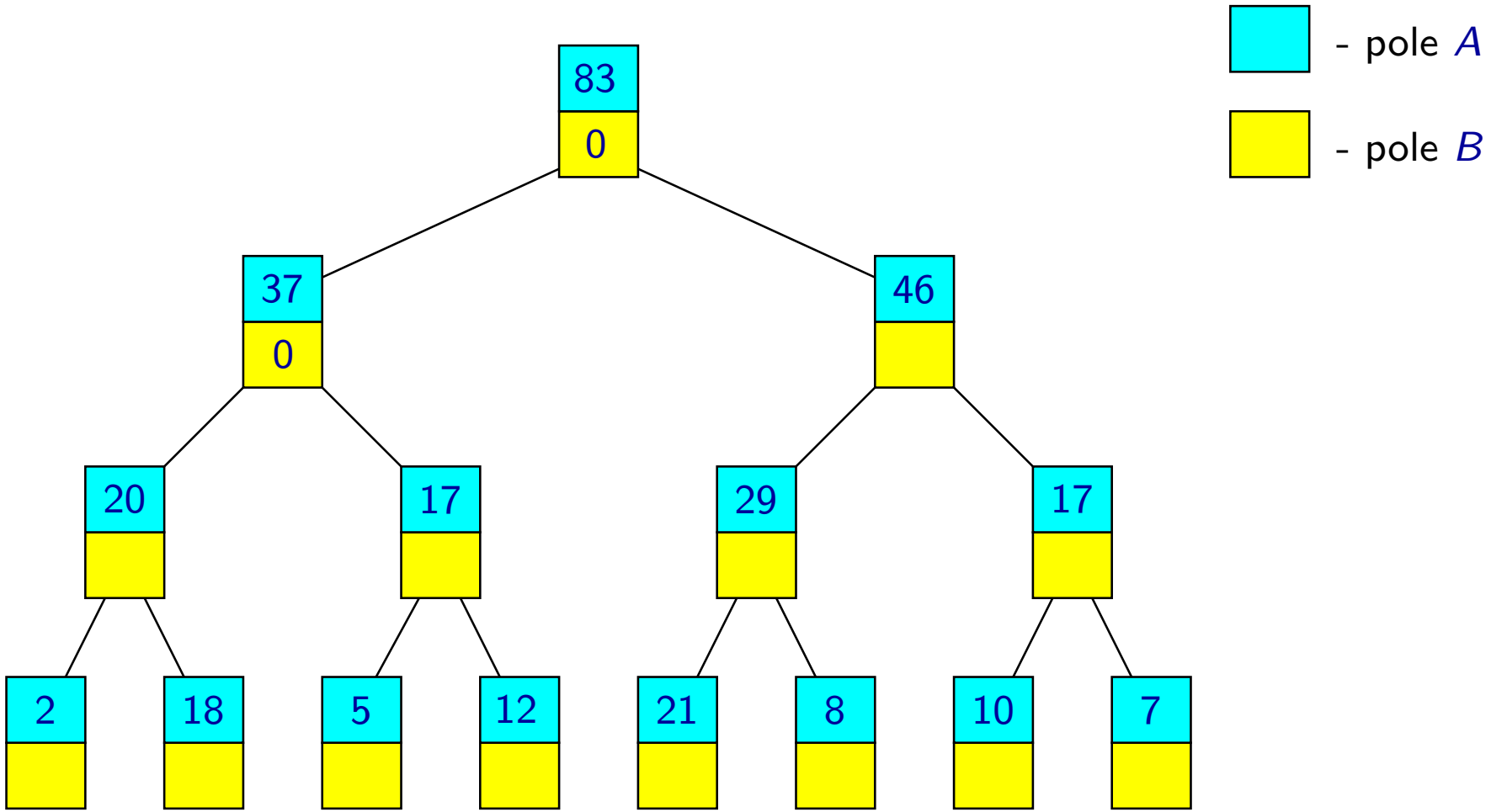
Počítání součtů prefixů posloupnosti



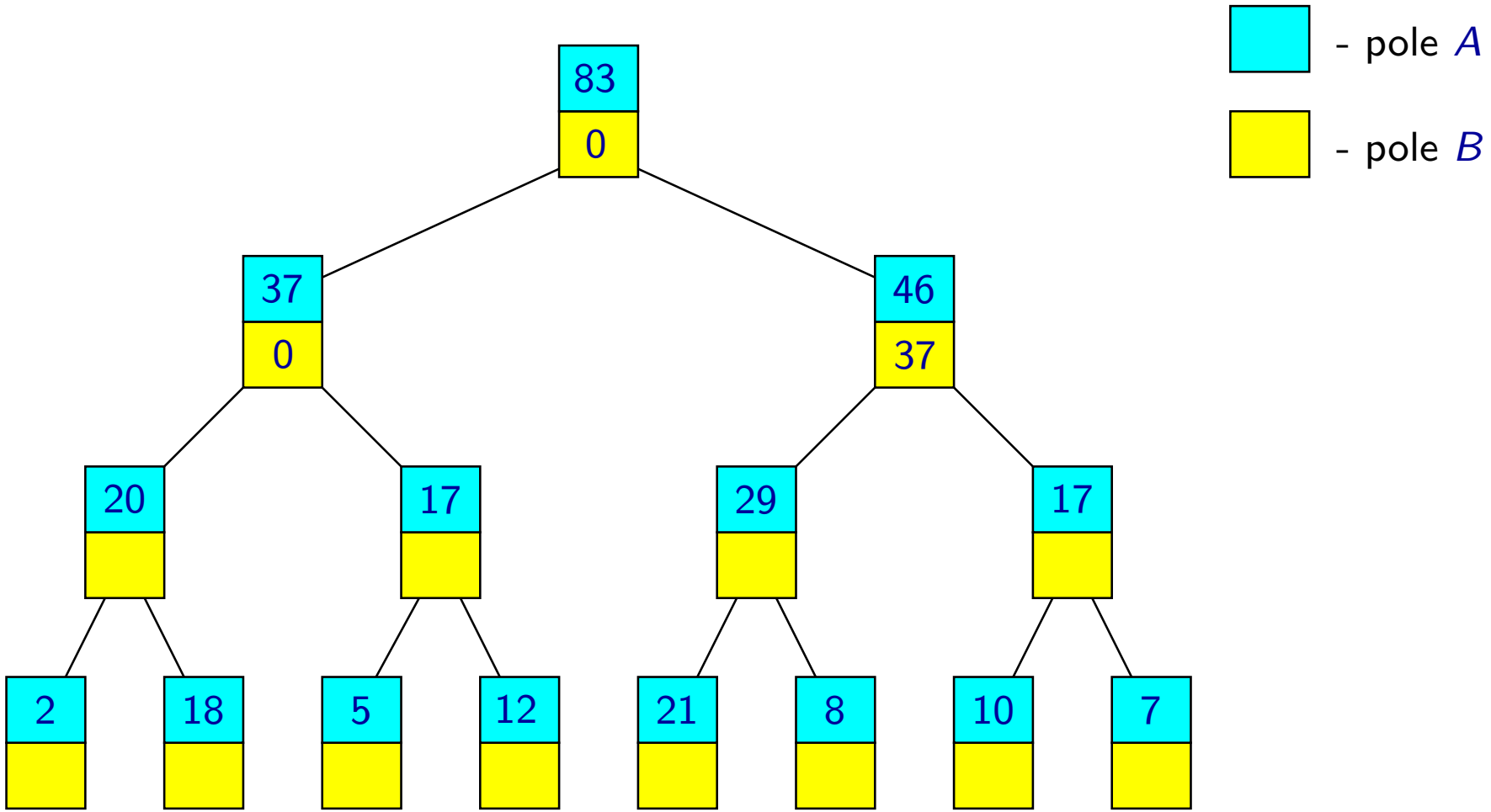
Počítání součtů prefixů posloupnosti



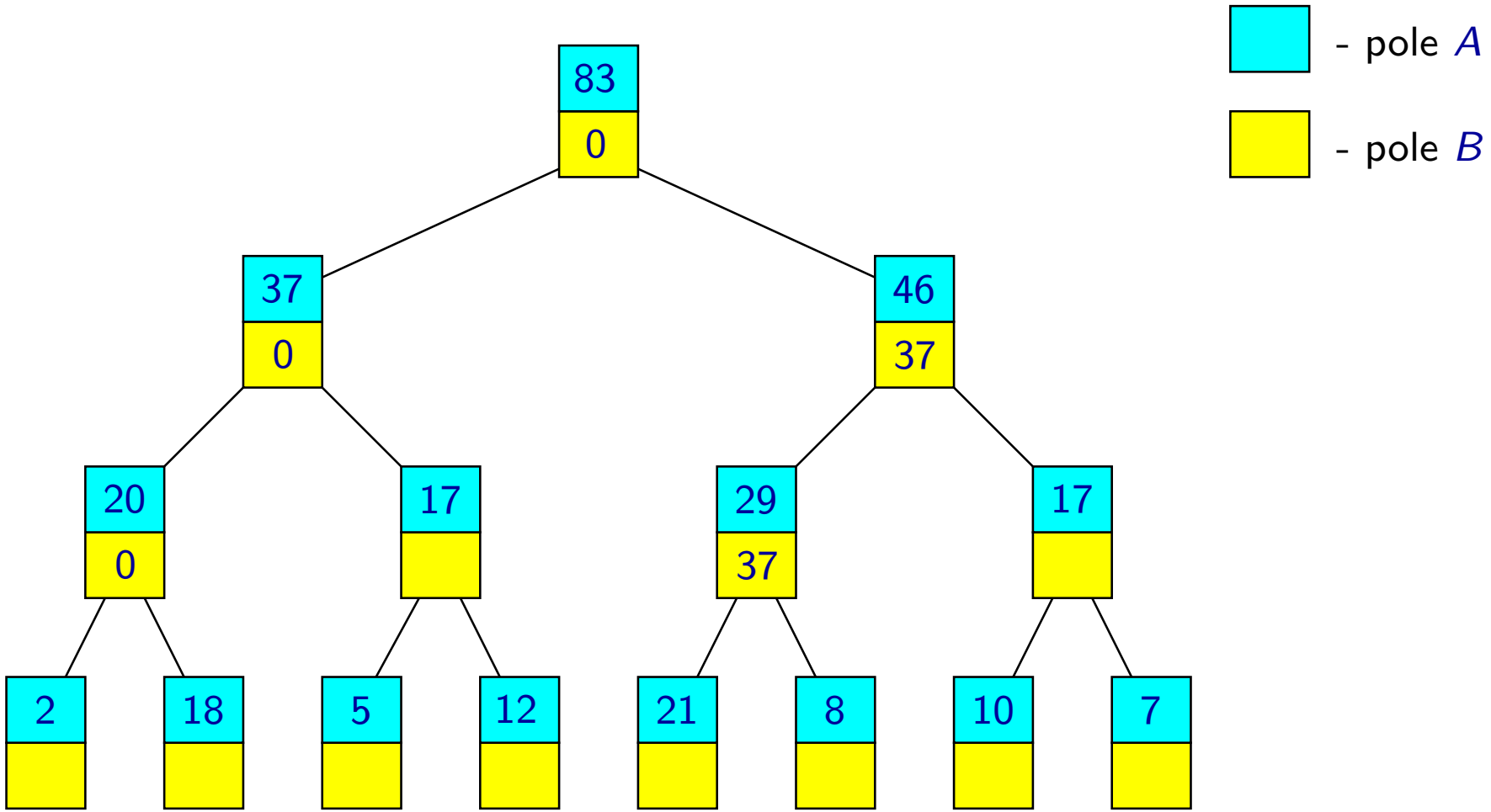
Počítání součtů prefixů posloupnosti



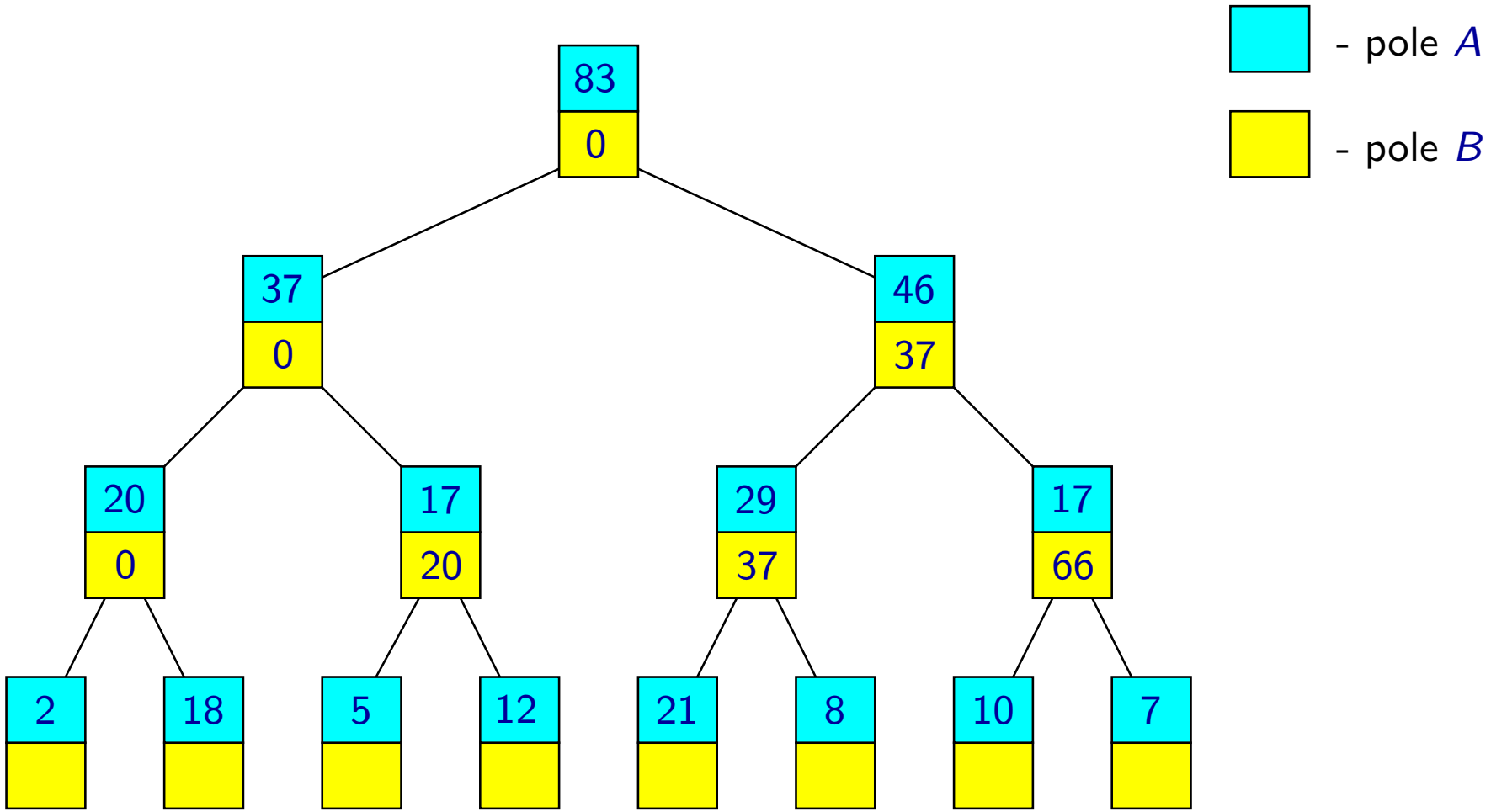
Počítání součtů prefixů posloupnosti



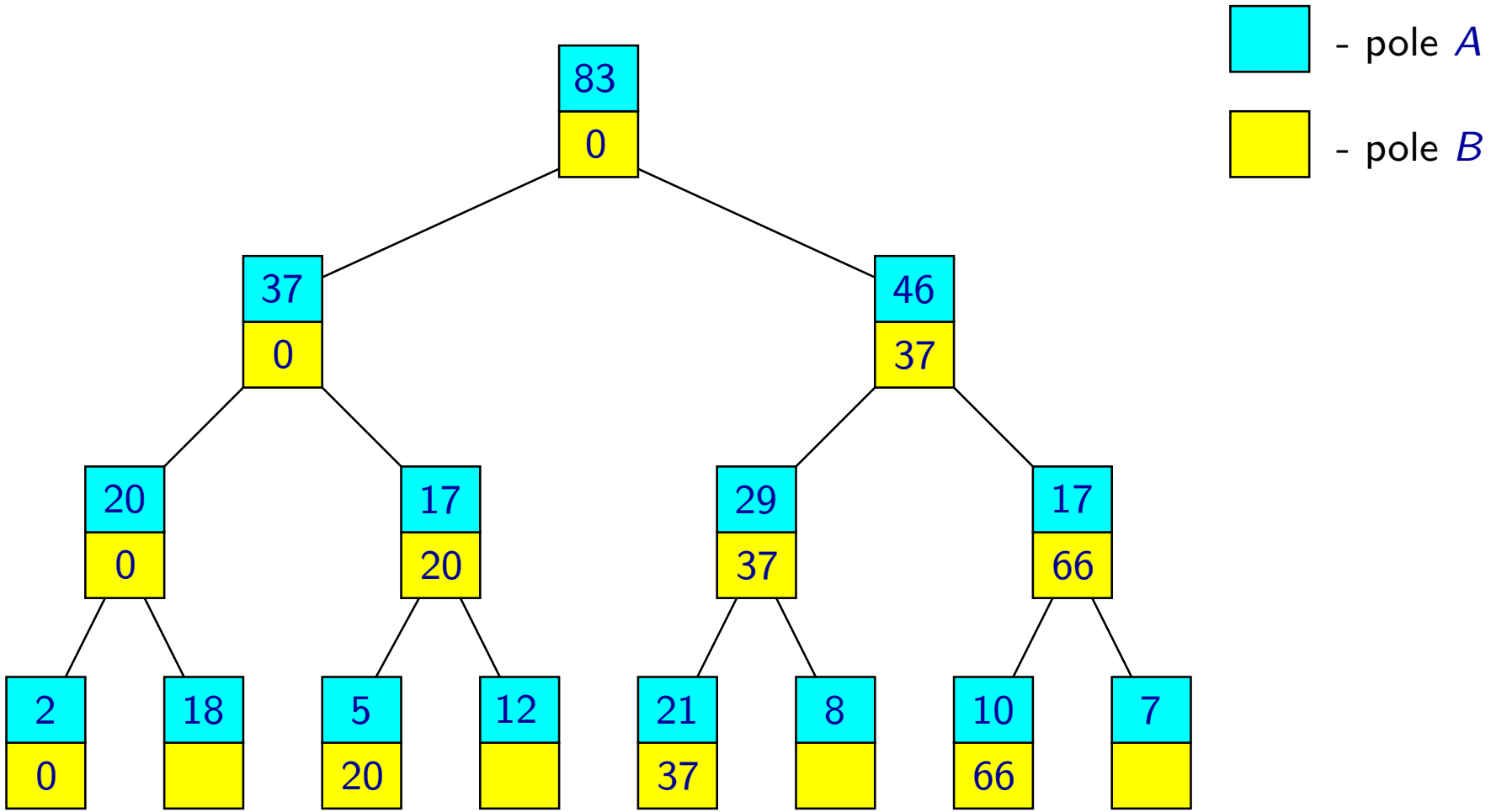
Počítání součtů prefixů posloupnosti



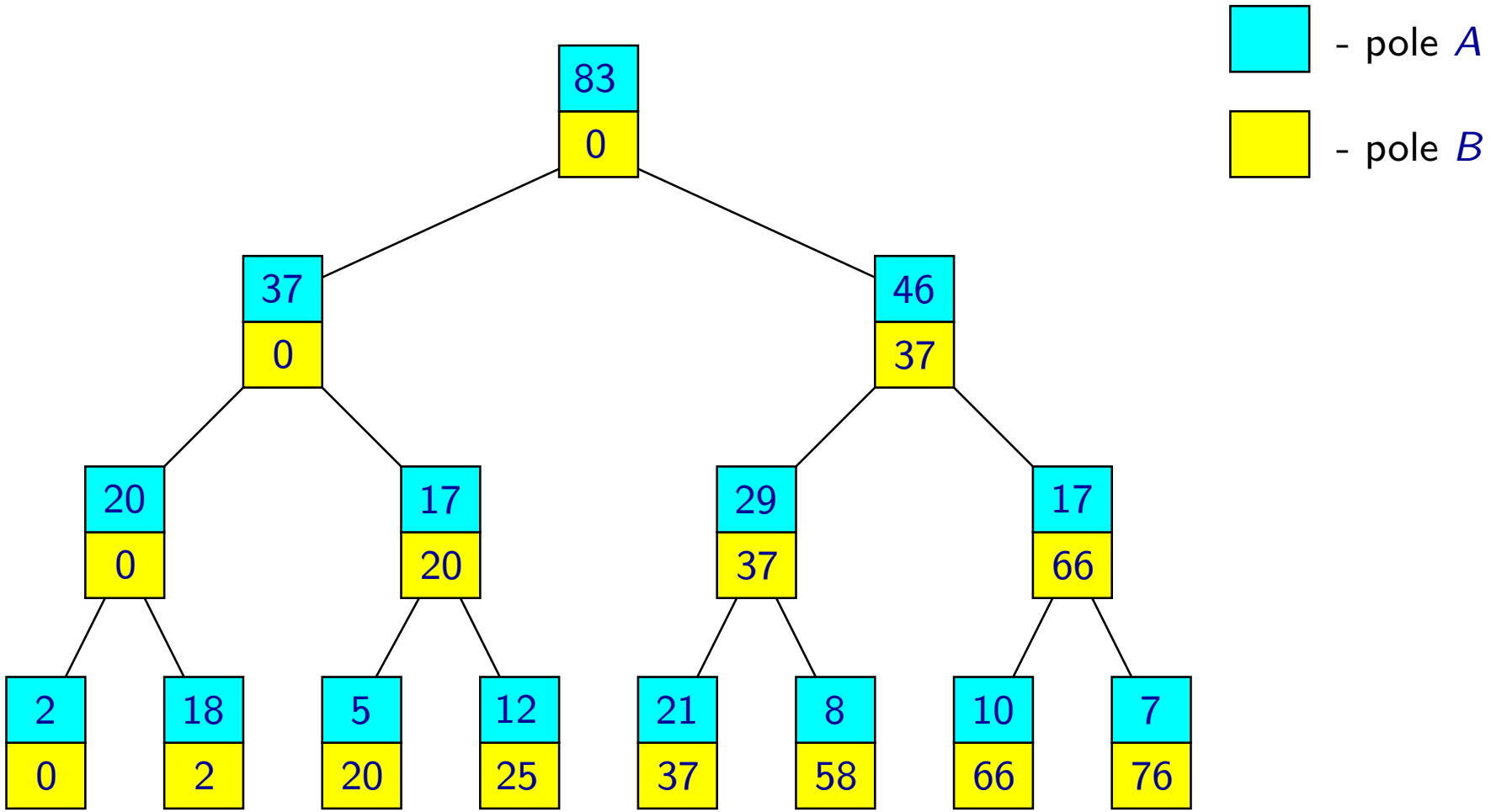
Počítání součtů prefixů posloupnosti



Počítání součtů prefixů posloupnosti



Počítání součtů prefixů posloupnosti



Počítání součtů prefixů posloupnosti

Součty všech prefixů posloupnosti je možné spočítat s $\mathcal{O}(n)$ procesory v čase $\mathcal{O}(\log n)$.

Lehce se ověří, že celkové množství provedených operací je $\Theta(n)$. Algoritmus je tedy optimální.

Podobně jako u předchozího algoritmu, který počítal součet pro celou posloupnost, ale ne pro jednotlivé prefixy, je možné výše uvedený algoritmus snadno upravit do podoby, kdy při při použití p procesorů, kde $p \leq n/2$, bude časová složitost $\mathcal{O}(n/p + \log p)$.

Při použití $\Omega(n \,/\, \log n)$ procesorů bude časová složitost tohoto upraveného algoritmu $\mathcal{O}(\log n)$.

Počítání součtů prefixů posloupnosti

Poznámka: U výše popsaného algoritmu nebylo příliš podstatné, že se jednalo zrovna o počítání součtu.

Stejný postup je možné použít pro libovolnou **asociativní binární operaci** $\circ$ pro spočítání všech hodnot s_i pro $i = 1, 2, \ldots, n$, kde

$$s_i = a_0 \circ a_1 \circ \cdots \circ a_{i-1}$$

Například:

- spočítání maxim a minim z posloupností $a_0, a_1, \ldots, a_{i-1}$
- spočítání hodnot

  $$a_0 \vee a_1 \vee \cdots \vee a_{i-1}$$

  za předpokladu, že hodnoty $a_0, a_1, \ldots, a_{n-1}$ jsou z množiny $\mathrm{Bool} = \{0, 1\}$

Počítání součtů prefixů posloupnosti

Počítání součtů prefixů je možné použít i pro jiné účely:

- Řekněme, že máme prvky $a_0, a_1, \ldots, a_{n-1}$ uloženy v poli A a v poli C máme pro každý prvek $A[i]$ uloženu hodnotu $C[i]$ z množiny $\{0, 1\}$.

  Chtěli bychom vykopírovat do pomocného pole D právě ty prvky $A[i]$, pro které platí

  $$C[i] = 1$$

  (Tj. chtěli bychom, aby prvních k prvků pole D obsahovalo právě ty prvky $A[i]$, kde $C[i] = 1$, kde k je celkový počet těchto prvků.)

  Index j v poli D, kam se má nakopírovat prvek $A[i]$, pro který platí $C[i] = 1$, se dá spočítat jako součet

  $$j = C[0] + C[1] + \cdots + C[i-1]$$

Součet velkých čísel

Uvažujeme následující problém:

Součet dvou velkých čísel

Vstup: Přirozená čísla a a b reprezentovaná binárně.

Výstup: Jejich součet $a + b$ reprezentovaný binárně.

Řekněme, že:

- čísla a a b jsou reprezentovaná každé n bity
- bity čísla a jsou uloženy v poli A
  (tj. $A[i]$, kde $0 \leq i < n$, reprezentuje i-tý bit čísla a)
- bity čísla b jsou uloženy v poli B
  (tj. $B[i]$, kde $0 \leq i < n$, reprezentuje i-tý bit čísla b)
- bity součtu $a + b$ budou po skončení výpočtu uloženy v poli S
  (tj. $S[i]$, kde $0 \leq i \leq n$, bude po skončení výpočtu obsahovat hodnotu i-tého bitu součtu $a + b$)

Součet velkých čísel

Označme zápisem $carry[i]$ hodnotu **přenosu** v i-tém bitu, tj. pokud bychom sčítali čísla tvořená bity

$$A[i-1]A[i-2]\cdots A[0] \qquad\qquad B[i-1]B[i-2]\cdots B[0]$$

hodnota $carry[i]$ by odpovídala bitu i tohoto součtu.

Předpokládáme, že $carry[0] = 0$.

Hodnoty $S[i]$ pro $i = 0, 1, \ldots, n-1$ je možné spočítat následovně:

$$S[i] := A[i] \oplus B[i] \oplus carry[i]$$

(Symbol $\oplus$ zde reprezentuje operaci *xor*, resp. sčítání modulo 2, což je totéž.)

Hodnota bitu $S[n]$ je stejná jako hodnota $carry[n]$.

Součet velkých čísel

Pokud bychom tedy znali hodnoty $carry[i]$ pro všechna i, hodnoty bitů $S[0]$, $S[1]$, ..., $S[n]$ by pak už bylo možné spočítat s n procesory v konstantním čase.

Hodnotu $carry[i+1]$ je možné spočítat z hodnot $A[i]$, $B[i]$ a $carry[i]$ následovně:

$$carry[i+1] := (A[i] \wedge B[i]) \vee ((A[i] \vee B[i]) \wedge carry[i])$$

Mohou nastat tři situace:

- Bity $A[i]$ i $B[i]$ mají oba hodnotu 1:
  pak $carry[i+1] = 1$ bez ohledu na hodnotu $carry[i]$
- Bity $A[i]$ i $B[i]$ mají oba hodnotu 0:
  pak $carry[i+1] = 0$ bez ohledu na hodnotu $carry[i]$
- Jeden z bitů $A[i]$ a $B[i]$ má hodnotu 1 a jeden hodnotu 0:
  pak $carry[i+1] = carry[i]$

Součet velkých čísel

Obecně můžeme pro libovolné i a j, kde $0 \leq i < j \leq n$, „efekt“ úseku bitů i, $i-1$, $\ldots$, $j-1$, daného hodnotami

$$A[i], A[i+1] \ldots, A[j-1] \qquad B[i], B[i+1] \ldots, B[j-1]$$

reprezentovat hodnotou z tříprvkové množiny $\{\mathrm{G}, \mathrm{R}, \mathrm{P}\}$, kde:

- G — $carry[j]$ bude vždy 1 bez ohledu na hodnotu $carry[i]$
- R — $carry[j]$ bude vždy 0 bez ohledu na hodnotu $carry[i]$
- P — $carry[j] = carry[i]$

Označme hodnotu tohoto efektu pro dané i a j zápisem $e(i,j)$.

Hodnotu $e(i, i+1)$ snadno spočítáme z hodnot $A[i]$ a $B[i]$.

Pro i, j, k, kde $0 \leq i < j < k \leq n$, je možné hodnotu $e(i,k)$ spočítat z hodnot $e(i,j)$ a $e(j,k)$ pomocí vhodně nadefinované binární asociativní operace $\circ$ na množině $\{\mathrm{G}, \mathrm{R}, \mathrm{P}\}$.

Součet velkých čísel

Hodnoty $carry[i]$ pro $i = 1, 2, \ldots, n$ je možné snadno spočítat z hodnot $e(0, i)$ a $carry[0]$.

- Pro spočítání všech hodnot $e(0, i)$ je možné použít stejný algoritmus, jako byl použit pro počítání součtů prefixů pole.

  Místo operace $+$ na číslech bude použita operace $\circ$ na množině $\{\mathrm{G}, \mathrm{R}, \mathrm{P}\}$.

Součet dvou n-bitových čísel je možné spočítat na stroji RAM typu EREW s p procesory v čase $\mathcal{O}(\log n + n/p)$.

Tj. pokud budeme mít k dispozici p procesorů, kde $p \in \Omega(n / \log n)$, bude doba výpočtu $\mathcal{O}(\log n)$.

Celkové množství provedených operací je $\Theta(n)$.

Součet velkých čísel

Součet dvou n-bitových čísel je možné spočítat na stroji RAM typu CRCW COMMON s $\mathcal{O}(n^3)$ procesory v čase $\mathcal{O}(1)$.

Myšlenka důkazu: Hodnotu $carry[i]$ pro $i = 1, 2, \ldots, n$ je možné vyjádřit následovně:

$$carry[i] \;:=\; \bigvee_{j=0}^{i-1} \Big((A[j] \wedge B[j]) \;\wedge\; \bigwedge_{k=j+1}^{i-1} (A[k] \vee B[k]) \Big)$$

Operace **and** ($\wedge$) a **or** ($\vee$) přes m operandů je možné na stroji PRAM typu CRCW COMMON počítat s m procesory v konstantním čase.

Poznámka: Existují způsoby, jak počet procesorů $\mathcal{O}(n^3)$ výrazně snížit.

Výpočet konečného automatu

Předpokládejme, že máme dán nějaký fixní deterministický konečný automat $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, tj. tento automat není u následujícího problému považován za součást vstupu.

Výpočet konečného automatu

Vstup: Slovo $w \in \Sigma^*$ délky n, tj. $w = a_1 a_2 \cdots a_n$.

Výstup: Posloupnost stavů $q_0, q_1, \ldots, q_n$ z množiny Q, kde stav q_i pro $i = 0, 1, \ldots, n$ je stav, ve kterém bude daný deterministický konečný automat $\mathcal{A}$ po přečtení prefixu $a_1 a_2 \cdots a_i$.

Myšlenka důkazu: Pro podslova $a_i a_{i+1} \cdots a_j$, kde $1 \leq i \leq j \leq n$, můžeme počítat funkce typu $Q \to Q$, které každému stavu $q \in Q$ přiřazují stav q', do kterého by se automat $\mathcal{A}$ dostal ze stavu q přečtením podslova $a_i a_{i+1} \cdots a_j$.

Výpočet konečného automatu

Skládání funkcí je asociativní operace.

Pro každé $i = 0, 1, \ldots, n$ se tak spočítá funkce přiřazující stavům $q \in Q$ stavy, do kterých by se automat $\mathcal{A}$ dostal přečtením prefixu $a_1 a_2 \cdots a_i$.

Opět je možné použít algoritmus pro počítání asociativních operací na prefixech pole.

Násobení matic

Násobení matic

Vstup: Čtvercové matice A a B velikosti $n \times n$.

Výstup: Matice $C = A \cdot B$.

Pro prvky $C[i][j]$ matice C (kde $i, j \in \{0, 1, \ldots, n-1\}$) platí

$$C[i][j] = \sum_{k=0}^{n-1} A[i][k] \cdot B[k][j]$$

Na stroji PRAM typu CREW tedy postačuje $\mathcal{O}(n^3)$ procesorů pro spočítání výsledné matice v čase $\mathcal{O}(\log n)$.

Násobení matic

Řekněme, že A je čtvercová matice velikosti $n \times n$.

Můžeme uvažovat **mocniny** matice A:

$$A^0 = I$$
$$A^1 = A$$
$$A^2 = A \cdot A$$
$$A^3 = A \cdot A \cdot A$$
$$A^4 = A \cdot A \cdot A \cdot A$$
$$\vdots$$

Tj. $A^0 = I$ a $A^{i+1} = A \cdot A^i$, kde I je jednotková matice.

Násobení matic

Umocňování matice

Vstup: Čtvercová matice A velikosti $n \times n$ a číslo k.

Výstup: Matice A^k.

Vzhledem k tomu, že násobení matic je asociativní operace, můžeme pro počítání mocniny matice A^k použít postupné umocňování na druhou a spočítat ho pomocí $\mathcal{O}(\log k)$ násobení matic.

Na stroji PRAM typu CREW je možné spočítat pro matici A velikosti $n \times n$ mocninu matice A^k pomocí $\mathcal{O}(n^3)$ procesorů v čase $\mathcal{O}(\log n \cdot \log k)$.

Speciálně v případě, kdy $k = n - 1$, bude časová složitost $\mathcal{O}(\log^2 n)$.

Násobení matic

Násobení matic nemusí pracovat jen se sčítáním a násobením.

Můžeme uvažovat i jiné druhy operací a jiné druhy množin, ze kterých jsou prvky matic.

Jedním příkladem je třeba:

- Prvky matic jsou čísla (např. přirozená, reálná, apod.) rozšířená o speciální hodnotu ∞.
- Místo operace $\cdot$ se bude používat operace $+$.
- Místo operace $+$ se bude používat operace $\min$.

To se dá využít například pro řešení problému, kdy máme dán orientovaný graf na n vrcholech, kde jsou hrany ohodnoceny, a chceme spočítat pro každou dvojici vrcholů u a v vzdálenost z u do v, tj. délku nejkratší cesty z u do v.

(Délka cesty je dána jako součet ohodnocení hran na této cestě.)

Násobení matic

K danému ohodnocenému orientovanému grafu $G = (V, E)$ s vrcholy očíslovanými indexy $1, 2, \ldots, n$ můžeme sestrojit matici A následovaně:

- $A[i][i] = 0$ pro všechna i
- $A[i][j] = w(i,j)$ pro všechny hrany $(i,j) \in E$, kde $w(i,j)$ je ohodnocení hrany (i,j) (předpokládáme, že $w(i,j) \geq 0$ a $j \neq i$)
- $A[i][j] = \infty$ pro všechny dvojice (i,j), kde $i \neq j$ a neexistuje hrana z i do j

Není těžké ověřit, že matice $D = A^{n-1}$ (počítaná pomocí operací $+$ a $\min$ místo $\cdot$ a $+$) splňuje následující:

- pokud $D[i][j] < \infty$, tak $D[i][j]$ udává délku nejkratší cesty z vrcholu i do vrcholu j
- pokud $D[i][j] = \infty$, tak neexistuje žádná cesta z vrcholu i do vrcholu j

Násobení matic

Následující problém je tedy možné řešit s $\mathcal{O}(n^3)$ procesory v čase $\mathcal{O}(\log^2 n)$:

Nejkratší cesty mezi všemi vrcholy v ohodnoceném grafu

Vstup: Orientovaný ohodnocený graf G s n vrcholy.

Výstup: Pro každou dvojici vrcholů u a v délka nejkratší cesty z u do v nebo informace, že žádná taková cesta neexistuje.

Násobení matic

Řekněme, že bychom se zaměřili na matice jejichž prvky jsou z množiny $\mathrm{Bool} = \{0, 1\}$.

Místo operací $\cdot$ a $+$ můžeme použít operace $\wedge$ a $\vee$.

Stejně jako v předchozím případě je možné provádět na stroji PRAM typu CREW násobení matic velikosti $n \times n$ s $\mathcal{O}(n^3)$ procesory v čase $\mathcal{O}(\log n)$ a výpočet hodnoty A^k s $\mathcal{O}(n^3)$ procesory v čase $\mathcal{O}(\log n \cdot \log k)$.

Všimněme si také, že platí následující:

Na stroji PRAM typu CRCW COMMON je možné provádět násobení booleovských matic velikosti $n \times n$ s $\mathcal{O}(n^3)$ procesory v čase $\mathcal{O}(1)$.

Matici A^k je pak možné spočítat v čase $\mathcal{O}(\log k)$.

Násobení matic

Podobně jako v předchozím případě uvažujme orientovaný graf G, kde ale hrany nejsou ohodnoceny.

Dosažitelnost v orientovaném grafu pro všechny dvojice vrcholů

Vstup: Orientovaný graf G s n vrcholy.

Výstup: Pro každou dvojici vrcholů u a v informace, zda existuje cesta z u do v.

Ke grafu G s n vrcholy můžeme sestrojit booleovskou matici A velikosti $n \times n$, kde $A[i][j]$ bude 1 právě pro ty dvojice (i, j), kde $i = j$ nebo existuje hrana z i do j.

Snadno se ověří, že v matici $D = A^{n-1}$ je $D[i][j] = 1$ právě tehdy, když existuje cesta z vrcholu i do vrcholu j.

Poznámka: Výše uvedená matice $D = A^{n-1}$ se často označuje jako **tranzitivní uzávěr** matice A.

Násobení matic

Z předchozího tedy plyne:

Tranzitivní uzávěr booleovské matice A velikosti $n \times n$ (a tedy dosažitelnost v orientovaném grafu G s n vrcholy) je možné řešit:

- na stroji PRAM typu EREW s $\mathcal{O}(n^3)$ procesory v čase $\mathcal{O}(\log^2 n)$,
- na stroji PRAM typu CRCW COMMON s $\mathcal{O}(n^3)$ procesory v čase $\mathcal{O}(\log n)$.

Násobení matic

Jedním z důsledků předchozího tvrzení je to, že každý problém ze třídy NL (tj. problém řešitelný nedeterministickým algoritmem s prostorovou složitostí $\mathcal{O}(\log n)$) je možné řešit efektivním paralelním algoritmem.

Věta

Každý problém ze třídy NL je možné řešit:

- na stroji PRAM typu EREW s polynomiální počtem procesorů v čase $\mathcal{O}(\log^2 n)$,
- na stroji PRAM typu CRCW COMMON s polynomiálním počtem procesorů v čase $\mathcal{O}(\log n)$.

Násobení matic

Myšlenka důkazu: Pro každý problém z NL existuje nedeterministický stroj $\mathcal{M}$ (např. nedeterministický Turingův stroj), který ho řeší s prostorovou složitostí $S(n) \in \mathcal{O}(\log n)$.

Pro daný vstup x velikosti n je možné sestrojit graf G, jehož vrcholy budou odpovídat konfiguracím stroje $\mathcal{M}$ velikosti $S(n)$ a hrany možným přechodům stroje $\mathcal{M}$ mezi těmito konfiguracemi.

Stačí zjistit, zda z vrcholu, který odpovídá počáteční konfiguraci, je dosažitelný nějaký vrchol odpovídající přijímající konfiguraci.

Důsledek

$\mathrm{NL} \subseteq \mathrm{NC}$

Třídění

Třídění

Vstup: Posloupnost $a_0, a_1, \ldots, a_{n-1}$.

Výstup: Prvky $a_0, a_1, \ldots, a_{n-1}$ setříděné od nejmenšího po největší.

Pro paralelní implementaci je vhodný například algoritmus **Merge-sort**.

Řada efektivních paralelních algoritmů pro problém třídění vychází právě z tohoto algoritmu.

Třídění

Algoritmus: Merge sort

```
MERGE-SORT(A, p, r):
    if r − p > 1 then
        q := ⌊(p + r) / 2⌋
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q, r)
        MERGE(A, p, q, r)
```

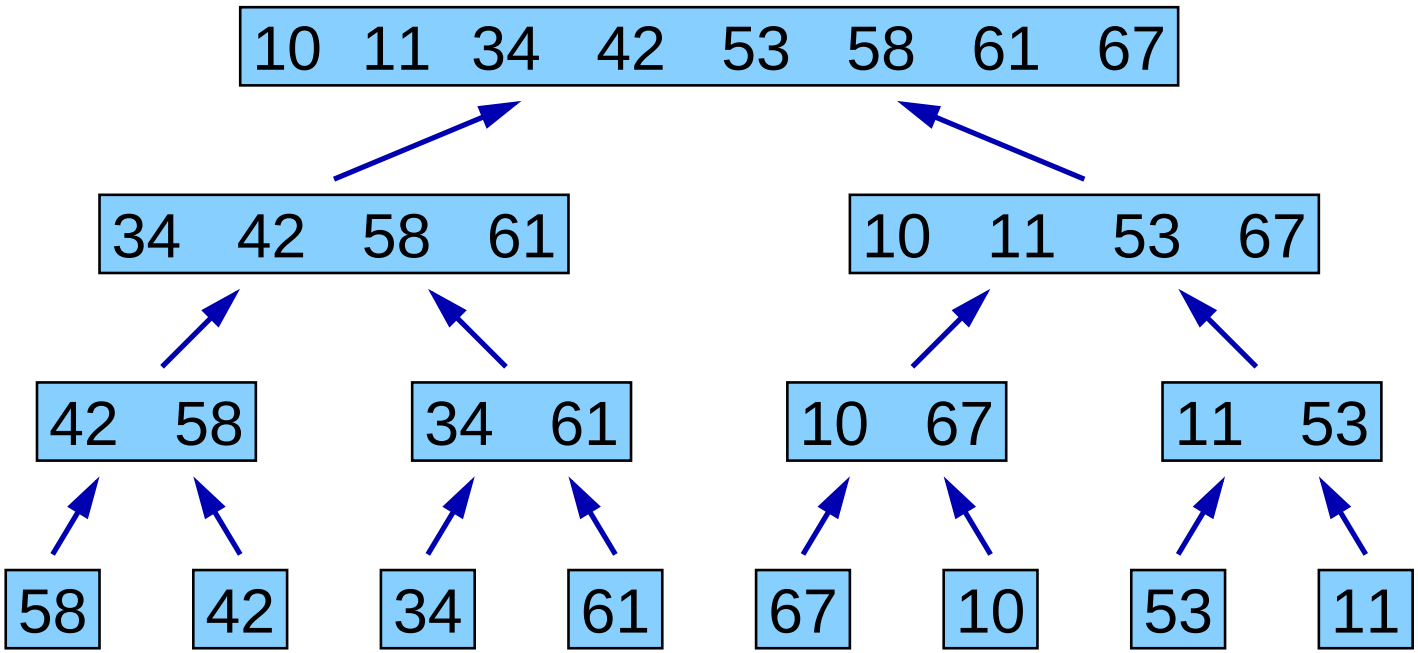
Obě rekurzivní volání funkce MERGE-SORT mohou být prováděny paralelně.

Pro setřídění pole A, které obsahuje prvky $A[0], A[1], \cdots, A[n-1]$, zavoláme MERGE-SORT$(A, 0, n)$.

Poznámka: Procedura MERGE(A, p, q, r) spojí setříděné posloupnosti uložené v $A[p \,..\, q-1]$ a $A[q \,..\, r-1]$ do jedné posloupnosti uložené v $A[p \,..\, r-1]$.

Třídění

Vstup: 58, 42, 34, 61, 67, 10, 53, 11



Strom rekurzivních volání má $\Theta(\log n)$ úrovní.

Třídění

Je zřejmé, že pro efektivní paralelní implementaci algoritmu Merge-sort stačí umět efektivně implementovat funkci MERGE, která:

- Dostane jako vstup dvě setříděné posloupnosti

  $$a_0, a_1, \ldots, a_{n-1} \qquad\qquad b_0, b_1, \ldots, b_{n-1}$$

  jejichž prvky jsou uloženy v polích A a B.
  (Obě tato pole jsou velikosti n.)

- Spojí tyto dvě posloupnosti délky n do jedné setříděné posloupnosti délky $2n$, která bude uložena v poli C (velikosti $2n$).

Ukážeme si jednu z jednoduchých paralelních implementací funkce MERGE, která pracuje na stroji PRAM typu CREW a s p procesory provede operaci sloučení dvou posloupností velikosti n v čase $\mathcal{O}(\log n + n/p)$.

Třídění

Předpokládejme pro jednoduchost, že všechny prvky v polích A a B jsou navzájem různé.

Označme $d = \lceil n/p \rceil$.

- Každému z prvků pole A s indexy $d \cdot i$, kde $i = 0, 1, \ldots, p-1$ bude přiřazen jeden procesor.

  Procesor i přiřazený prvku $A[d \cdot i]$ vyhledá sekvenčně pomocí binárního vyhledávání (půlením intervalu) v čase $\mathcal{O}(\log n)$ pozici v poli B, kam by byl zařazen prvek $A[d \cdot i]$, kdyby byl do pole B přidán.

- Podobným způsobem se pro každý z prvků $B[d \cdot i]$ vyhledá pozice, kam by byl tento prvek zařazen v poli A.

- Na základě těchto zjištěných indexů se pole A a B rozdělí na $2n$ „pruhů“, z nichž každý má délku nejvýše $2d$ (pruh obsahuje nejvýše d prvků z pole A a nejvýše d pruhů z pole B).

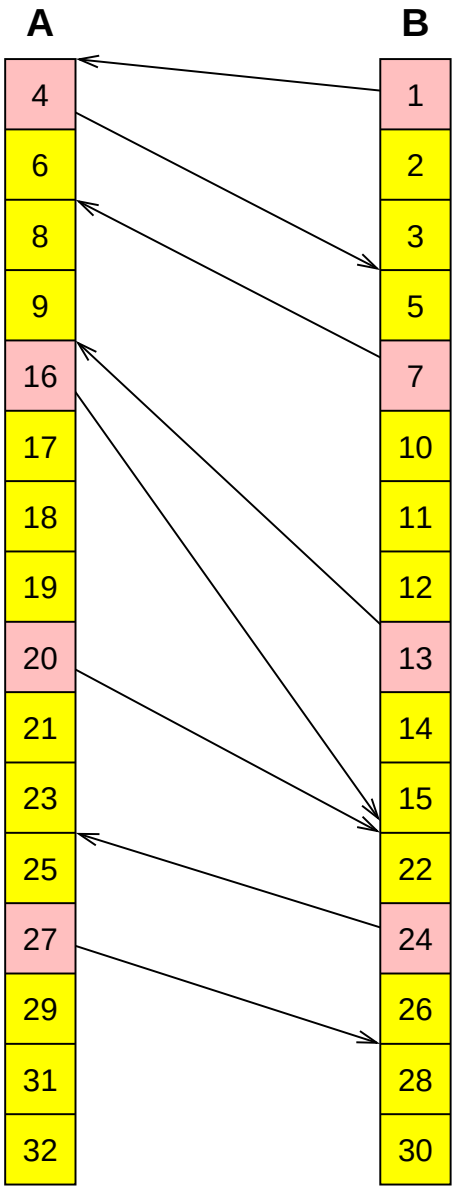
Třídění

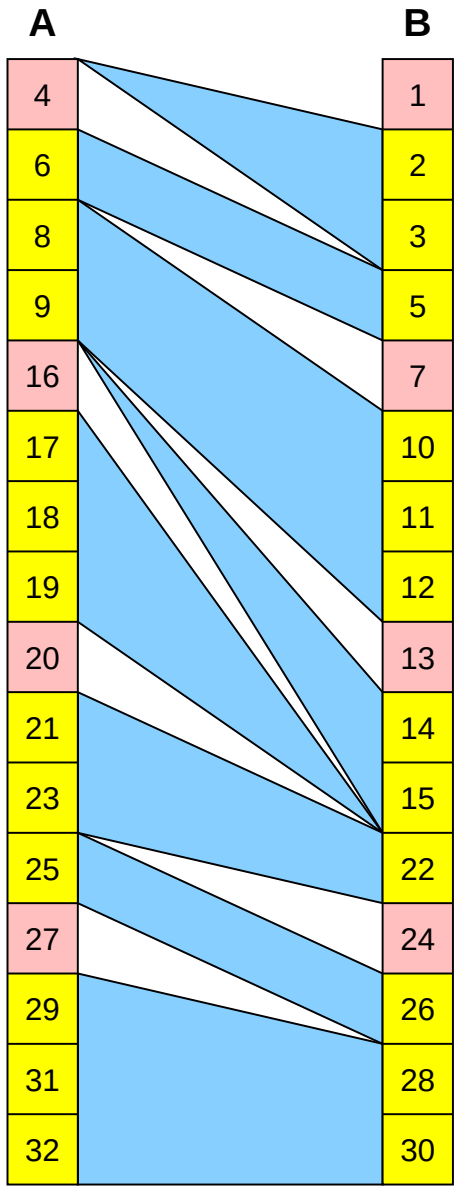
- Pro každý z pruhů se určí:
  - začátek a konec daného pruhu v poli A
  - začátek a konec daného pruhu v poli B
  - celkový počet prvků (z pole A i B) spadající do daného pruhu
- Pruhy budou ve výsledné setřídění posloupnosti setříděny za sebou.

  Pro každý z pruhů se spočítají indexy začátku a konce úseku v poli C, kam budou prvky z daného pruhu uloženy.
- Pro každý z pruhů se sekvenčně provede merge prvků z daného pruhu a jejich nakopírování do příslušného úseku v poli C.

  Každý z procesorů zpracuje dva pruhy.

Třídění


A
B
4
6
8
9
16
17
18
19
20
21
23
25
27
29
31
32
1
2
3
5
7
10
11
12
13
14
15
22
24
26
28
30


A
B
4
6
8
9
16
17
18
19
20
21
23
25
27
29
31
32
1
2
3
5
7
10
11
12
13
14
15
22
24
26
28
30

Třídění

Výše uvedená varianta paralelní implementace algoritmu Merge-sort pro PRAM typu CREW má při použití p procesorů časovou složitost $\mathcal{O}(\log^2 n + n \log n / p)$.

Při použití $\Omega(n \,/\, \log n)$ procesorů bude mít tedy výše uvedený algoritmus časovou složitost $\mathcal{O}(\log^2 n)$.

Existují efektivnější paralelní implementace třídění:

Existuje paralelní třídící algoritmus pro PRAM typu EREW, který má s n procesory časovou složitost $\mathcal{O}(\log n)$.

Vzájemná simulace variant strojů PRAM

Věta

Činnost stroje PRAM typu CRCW PRIORITY, který s p procesory vykoná t kroků, je možné simulovat strojem PRAM typu EREW s p procesory, který provede $\mathcal{O}(t \log p)$ kroků.

Myšlenka důkazu: Předpokládejme, že původní stroj PRAM (typu CRCW PRIORITY) pracuje po fázích, kdy střídavě všech p procesorů:

- čte z globální paměti
- provádí lokální výpočty, kde se ke globální paměti nepřistupuje
- zapisuje do globální společné paměti

První (tj. společné čtení z globální paměti) a třetí (tj. společný zápis do globální paměti) z těchto fází je třeba implementovat tak, aby provedení jedné takové fáze s p procesory trvalo $\mathcal{O}(\log p)$ kroků.

Vzájemná simulace variant strojů PRAM

Implementace fáze write:

- Bude použito pomocné pole A velikosti p, jehož prvky budou trojice tvaru

  $$(a, v, id)$$

  kde:

  - a — adresa buňky globální paměti, kam se bude zapisovat
  - v — zapisovaná hodnota
  - id — číslo procesoru, který zápis provádí

- Všechny procesory místo přímého zápisu do globální paměti zapíšou adresu a, kam chtějí zapisovat, hodnotu v, kterou chtějí zapsat, a své id do prvku $A[id]$.

  (Procesory, které nechtějí do globální paměti v daném kroku zapisovat, zapíšou do pole A nějakou speciální hodnotu, např. trojici (a, v, id), kde a bude -1.)

Vzájemná simulace variant strojů PRAM

- Pole A se setřídí:
  - Primárním kritériem pro třídění je hodnota adresy a.
  - Pokud mají dvě položky (a, v, id) a (a', v', id') v poli A stejnou adresu (tj. pokud $a = a'$), tak se třídí podle id a id'.
- Každý procesor se podívá do setříděného pole A na jednu položku:
  - Zjistí, jestli se jedná o položku s nejmenším id pro danou adresu — tj. jestli je buď první v poli A nebo předchozí položka odkazuje k jiné adrese.
  - Pokud ano, provede příslušný zápis do globální paměti.

Setřídění pole A je možné s p procesory provést v čase $\mathcal{O}(\log p)$.
Všechny ostatní kroky v této simulaci je možné provést v konstantním čase.

Vzájemná simulace variant strojů PRAM

Implementace fáze read:

- Bude použito pomocné pole B velikosti p, jehož prvky budou dvojice tvaru

  $$(a, id)$$

  kde:

  - a — adresa buňky globální paměti, ze které se bude číst
  - id — číslo procesoru, který z dané adresy potřebuje číst

- Všechny procesory místo přímého čtení z globální paměti zapíšou adresu a, ze které chtějí číst, a své id do prvku $B[id]$.

  (Procesory, které nechtějí z globální paměti v daném kroku číst, zapíšou do pole B nějakou speciální hodnotu, např. dvojici (a, id), kde a bude -1.)

Vzájemná simulace variant strojů PRAM

- Pole B se setřídí — kritériem pro třídění je hodnota adresy a.
- Každý procesor se podívá do setříděného pole B na jednu položku:
  - Zjistí, jestli se jedná o položku s nejmenším indexem pro danou adresu — tj. jestli je buď první v poli B nebo předchozí položka odkazuje k jiné adrese.
  - Pokud ano, přečte z adresy a hodnotu v.
- Následně je třeba provést rozkopírovaní každé přečtené hodnoty v ke všem položkám pole B, kde je uvedena stejná adresa jako adresa a, ze které byla čtena hodnota v.

  Použijí se další dvě pomocná pole velikosti p:
  - V — kopírované hodnoty v
  - C — booleovské hodnoty udávající pro každé i, zda byla již hodnota $V[i]$ zapsána

Vzájemná simulace variant strojů PRAM

Inicializace rozkopírování:

- Procesory i, které četly hodnotu v z globální paměti (z adresy $B[i].addr$), provedou:

  $V[i] := a$
  $C[i] := \textsc{True}$

- Ostatní procesorry provedou:

  $C[i] := \textsc{False}$

- Všechny procesory inicializují lokální proměnnou s na hodnotu 1.

Vzájemná simulace variant strojů PRAM

- Cyklus, který budou provádět paralelně všechny procesory i, dokud bude $s < p$:

```
if C[i] = True then
    if i + s < p and C[i + s] = False then
        V[i + s] := V[i]
        C[i + s] := True
s := s * 2
```

- Nakonec se paralelně rozkopírují hodnoty $V[i]$ procesorům, jejichž id je uvedeno v $B[i].id$.

Setřídění prvků pole B a následné rozkopírování vyžadují s p procesory čas $\mathcal{O}(p)$, ostatní kroky je možné provést v konstantním čase.

Vzájemná simulace variant strojů PRAM

Věta

Činnost stroje PRAM typu CRCW PRIORITY, který s p procesory vykoná t kroků, je možné simulovat strojem PRAM typu CRCW COMMON s $\mathcal{O}(p^2)$ procesory, který provede $\mathcal{O}(t)$ kroků.

Myšlenka důkazu:
Stačí umět simulovat jednu fázi zápisu s $\mathcal{O}(p^2)$ procesory v čase $\mathcal{O}(1)$.

Předpokládejme, že procesory simulovaného stroje jsou číslovány $0, 1, \ldots, p-1$.

Použije se pomocné pole C velikosti p.

Vzájemná simulace variant strojů PRAM

Simulace jedné fáze zápisu:

- Prvky pole C jsou inicializovány hodnotami 0.
- Pro každou dvojici (i, j), kde $0 \leq i < j < p$ se paralelně provede následující:
  - Pokud procesory i a j chtějí zapisovat na stejnou adresu a, nastaví se hodnota prvku $C[j]$ na 1.
- Pro každé i, kde $0 \leq i < p$, se paralelně provede následující:
  - Pokud je $C[i] = 0$, provede se zápis do globální paměti, který by prováděl procesor i simulovaného stroje.

Poznámka: Je zřejmé, že $C[i]$ je 0 právě u těch procesorů, které mají nejmenší *id* mezi všemi procesory, které chtějí zapisovat na stejnou adresu.

Dolní odhady složitosti pro stroje PRAM

Dá se dokázat, že například jakýkoli stroj PRAM typu CREW, který řeší následující problém, musí mít časovou složitost minimálně $\Omega(\log n)$:

OR sekvence bitů

Vstup: Sekvence bitů $a_0, a_1, \ldots, a_{n-1}$.

Výstup: Hodnota $a_0 \vee a_1 \vee \cdots \vee a_{n-1}$.

Je také dokázáno, že nemůže existovat stroj PRAM typu CRCW, který by řešil následující problém v čase $\mathcal{O}(1)$:

XOR sekvence bitů

Vstup: Sekvence bitů $a_0, a_1, \ldots, a_{n-1}$.

Výstup: Hodnota $a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1}$.

Poznámka: Symbol $\oplus$ zde reprezentuje operaci *xor*.

Rozklad neorientovaného grafu na souvislé komponenty

Rozklad neorientovaného grafu na souvislé komponenty

Vstup: Neorientovaný graf $G = (V, E)$ daný seznamem vrcholů a hran.

Výstup: Rozklad grafu G na souvislé komponenty.

Popíšeme si paralelní algoritmus pro stroj PRAM typu CRCW ARBITRARY, který bude pracovat s $\mathcal{O}(n + m)$ procesory (kde n je počet vrcholů a m počet hran grafu G), a který bude mít časovou složitost $\mathcal{O}(\log n)$.

Předpokládáme, že graf $G = (V, E)$ je zadán jako:

- pole vrcholů V velikosti n
- pole hran E velikosti m — předpokládá se, že každá hrana je v tomto poli uvedena dvakrát, jednou jako (u, v) a jednou jako (v, u). Pořadí hran není důležité.

Rozklad neorientovaného grafu na souvislé komponenty

Algoritmus bude používat následující tři pole velikosti n, jejichž indexy odpovídají vrcholům z množiny V.

- D — indexy rodičů jednotlivých vrcholů ve vytvářených stromech
- D_{prev} — uložení hodnot z pole D z předchozí iterace
- Q — booleovské hodnoty indikující, že pro daný vrchol došlo ke změně

Dále bude použita globální proměnná *changed* typu Bool.

Inicializace:

- pro každý vrchol $v \in V$ se paralelně provede:

  $D[v] := v$

- procesor 0 provede:

  changed := TRUE

Rozklad neorientovaného grafu na souvislé komponenty

Hlavní cyklus se bude provádět, dokud bude platit *changed* = TRUE.

Jedna iterace hlavního cyklu:

- **Krok 1** — pro každý vrchol $v \in V$ se paralelně provede:

  $Q[v] := \text{FALSE}$
  $D_{prev}[v] := D[v]$

- **Krok 2** — pro každý vrchol $v \in V$ se paralelně provede:

  $D[v] := D_{prev}[D_{prev}[v]]$
  if $D[v] \neq D_{prev}[v]$ **then**
  $\quad Q[D[v]] := \text{TRUE}$

Rozklad neorientovaného grafu na souvislé komponenty

- **Krok 3** — pro každou hranu $(u, v) \in E$ se paralelně provede:

---

if $D[u] = D_{prev}[u]$ **and** $D[v] < D[u]$ **then**
  $D[D[u]] := D[v]$
  $Q[D[v]] := \text{TRUE}$

- **Krok 4** — pro každou hranu $(u, v) \in E$ se paralelně provede:

---

if $D[u] = D[D[u]]$ **and** $Q[D[u]] = \text{FALSE}$ **then**
  if $D[u] \neq D[v]$ **then**
    $D[D[u]] := D[v]$

- **Krok 5** — pro každý vrchol $v \in V$ se paralelně provede:

$D[v] := D[D[v]]$

Rozklad neorientovaného grafu na souvislé komponenty

- **Krok 6** — procesor 0 provede:

  $changed$:= FALSE

- **Krok 7** — pro každý vrchol $v \in V$ se paralelně provede:

  if $Q[v]$ = **True then**

  $changed$:= TRUE

Vztah mezi paralelními algoritmy a obvody

Existuje úzký vztah mezi paralelními algoritmy a **booleovskými obvody**.

Problém A může být řešen booleovskými obvody v následujícím smyslu:

- Existuje nekonečná posloupnost obvodů $C_0, C_1, C_2, \ldots$
- Každý obvod C_n má přesně n vstupů.
- Předpokládáme, že vstupy problému A jsou kódovány jako sekvence bitů.
- Pokud je vstup x problému A reprezentován n bity a tyto bity budou přiřazeny jako hodnoty vstupů obvodu C_n, bude tento obvod generovat jako svůj výstup odpověď na příslušnou otázku problému A pro vstup x.

U obvodů se uvažují následující dva parametry:

- **velikost obvodu** — počet hran příslušného grafu obvodu
- **hloubka obvodu** — délka nejdelší cesty v grafu obvodu

Vztah mezi paralelními algoritmy a obvody

Dá se ukázat, že pro daný problém A existuje program pro stroj PRAM právě tehdy, když pro něj existuje posloupnost obvodů, která ho řeší, kde:

- Časová složitost daného stroje PRAM je asymptoticky stejná jako hloubka obvodů.
- Počet procesorů použitých strojem PRAM a velikost obvodu se liší nejvýše polynomiálně.

Tento vztah platí mezi následujícími typy strojů PRAM a následujícími typy obvodů:

- Stroje PRAM typu CREW odpovídají obvodům, kde hradla typu AND a OR mají vždy dva vodiče, které do nich vstupují, a libovolný počet vodičů, které z nich vystupují.
- Stroje PRAM typu CRCW COMMON odpovídají obvodům, kde hradla typu AND a OR mohou mít neomezený počet vodičů, které do nich vstupují i z nich vystupují.