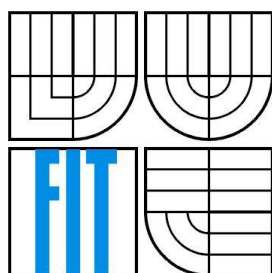


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



PRL

(TEMATICKÉ OKRUHY KE STÁTNICÍM 2009)

OBSAH

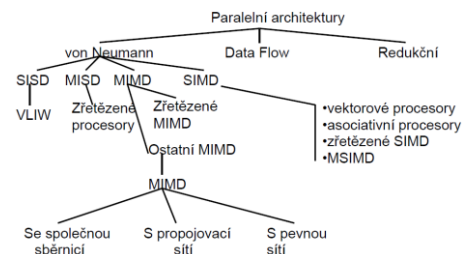
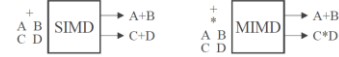
1	Vlastnosti paralelních a distribuovaných architektur.....	3
1.1	Ne-von Neumannovské architektury	3
1.2	von Neumannovské architektury.....	3
2	Základní typy topologií	4
2.1	MIMD – sdílená paměť a předávání zpráv	4
2.2	Propojovací sítě	5
3	Distribuované a paralelní algoritmy a jejich složitost.....	6
4	Řešení typických problémů paralelismu	6
5	Algoritmy řazení, algoritmy vyhledávání, vektorové a maticové algoritmy.....	7
5.1	Řadící algoritmy	7
5.2	Vyhledávací algoritmy	10
5.3	Vektorové a maticové algoritmy	11
6	Model PRAM (Parallel Random Access Machine), suma prefixů a její aplikace	12
6.1	PRAM	12
6.2	Suma prefixů.....	13
6.3	Využití sumy prefixů.....	14
7	Suma sufixů, Eulerova cesta, algoritmy nad seznamy, stromy a grafy.....	16
7.1	Algoritmy nad seznamy a suma sufixů	16
7.2	Algoritmy nad stromy.....	17
8	Interakce mezi procesy a typické problémy paralelismu	18
9	Předávání zpráv a jazyky pro paralelní zpracování	19
9.1	Předávání zpráv	19
9.2	Jazyky pro paralelní zpracování	20

1 VLASTNOSTI PARALELNÍCH A DISTRIBUOVANÝCH ARCHITEKTUR



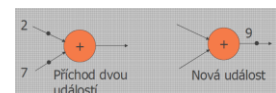
Flynnova klasifikace architektur procesorů:

- **SISD** – konvenční počítače;
- **SIMD** – vektorové počítače;
- **MISD** - řetězové;
- **MIMD**:
 - multiprocesory (sdílená paměť);
 - multikomputery (předávání zpráv).



1.1 NE-VON NEUMANNOVSKÉ ARCHITEKTURY

Dataflow – Provádí interpretaci grafu toku dat, program je tak převeden do takového grafu. Mezi používané jazyky patří VAL, Lucid, Id.



Redukční počítač – Pracuje na principu náhrady části výrazu jeho významem ($2 \cdot 3 \rightarrow 6$). Zpravidla se používá redukce řetězců, používá stromové hierarchie **T uzlů** (procesory a komunikace) a **L uzlů** (paměti).

1.2 VON NEUMANNOVSKÉ ARCHITEKTURY

VLIW (very long instruction word) – Jediný tok řízení, který řídí všechny procesory. Každá instrukce tak tvoří samostatné pole s operačním kódem pro všechny procesory, díky tomu existuje jen jediný *program counter PC*, který zpracovává instrukce sekvenčně. Je to architektura jednoduchá na HW implementaci a dobře škálovatelná, problémem jsou však podmíněné skoky (jedna subinstrukce provede skok, co ostatní?), problém toku dat (instrukce zpracované v jednom kroku nemůžou používat své výsledky → synchronizační NOOP zvětšující výsledné programy).

Superskalární procesory:

- **statické (in-order)** – Zpracovávají instrukce paralelně, ale v programovém pořadí, paralelní zpracování je možné u určitých (párovatelných kombinací) instrukcí;
- **dynamické (out-of-order)** – Zpracovávají více instrukcí paralelně i mimo programové pořadí, spekulativní provádění;

Zřetězené procesory – Lineárně propojené procesory, které řeší úlohy s proudovým charakterem, data procházejí postupně.



Vektorové procesory – Paralelně provádějí stejnou operaci nad různými daty. Lze dělit i na skupiny částečně nezávislých procesorů (MSIMD), nebo zcela nezávislých procesorů bez synchronizace (SPMD – same program multiple data). Výhodou těchto архитектур oproti MIMD jsou menší nároky na paměť, jeden instrukční tok a bez výraznější režie na synchronizaci.

Granularitou paralelismu rozlišujeme na několika úrovních:

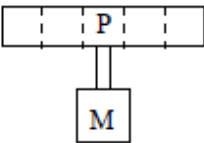

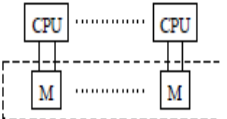
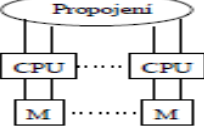
- **uvnitř instrukcí** – nejjemnější paralelismus;
- **mezi instrukcemi** – některé instrukce se provádějí paralelně, ale není to vidět úrovni programovacího jazyka (zřetězení u RISC);
- **mezi příkazy** – u vektorových počítačů nebo specializovaných koprocesorů;
- **mezi bloky procesu** – vlákna;
- **mezi procesy**.

2 ZÁKLADNÍ TYPY TOPOLOGIÍ

Synchronizace je vlastně zajištěním požadovaných časových vztahů mezi událostmi, využívá různých prostředků jako např. zasílání zpráv, semaforey, monitory, ad. Mezi typické synchronizační úlohy patří soupeření a kooperace.

Komunikace je přenášáním informací mezi subjekty (procesory, vlákna). Prostředky komunikace jsou např. sdílená paměť či různé druhy zasílání zpráv (broadcasting, kanály, RPC).

2.1 MIMD – SDÍLENÁ PAMĚŤ A PŘEDÁVÁNÍ ZPRÁV

Typ	Obrázek	Předávání zpráv	Sdílená paměť
Multitasking – Existuje jen jedno CPU, které přepíná kontext. Virtuální procesory jako je u technologie hyperthreading		Simulováno SW	Ano
Sdílená paměť – realizováno např. pomocí cache, jsou zde těsné vazby, které vedou k bojům o sběrnici.		Simulováno SW nebo HW	Ano
Virtuální sdílená paměť – každý procesor má svoji vlastní cache, které spolu komunikují a na venek se tváří jako jediný společný adresový prostor.		Simulováno SW nebo HW	Simulováno HW
Předávání zpráv – volně vázaná architektura propojená pouze procesory, typickým příkladem je počítačová síť LAN.		Ano	Simulováno SW

Sdílená paměť je paměť, do jejíhož prostoru mají přístup všechny procesory. Existují čtyři řešení současného přístupu k jedné buňce:

- **Exclusive read, exclusive write (EREW);**
- **Concurrent read, exclusive write (CREW);**
- **Exclusive read, concurrent write (ERCW)** – nemá opodstatnění, a tak se nepoužívá;
- **Concurrent read, concurrent write (CRCW).**

Protipól můžeme vidět v **předávání zpráv**, kdy má každý procesor svůj adresový prostor a paměť, přičemž vzájemnou komunikaci zprostředkovávají zprávy.

2.2 PROPOJOVACÍ SÍTĚ

Propojovací sítě procesorů ovlivňují vhodnost jednotlivých typů algoritmů a efektivnost toku dat. Lze je dělit na:

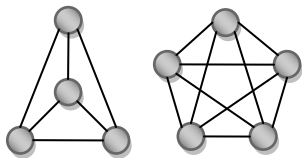
- **statické** – Kdy všechny uzly tvoří jen procesory, kanály jsou spojnice mezi uzly a jejich použití je zejména v systémech bez sdílené paměti. Mezi podstatné vlastnosti těchto sítí patří **průměr** (**diametr** – délka nejdelší z nejkratších cest mezi všemi dvojicemi uzlů), **konektivita** (**arc connectivity** – minimální počet hran, které je nutné odstranit pro rozdělení sítě na více částí), **šířka bisekce** (**bisection width** – minimální počet hran, které spojují dvě přibližně stejně velké poloviny sítě);

Úplné propojení

Diametr = 1

Konektivita = $p - 1$

Šířka bisekce = $\frac{p^2}{4}$

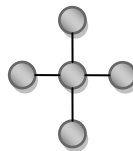


Hvězda

Diametr = 2

Konektivita = 1

Šířka bisekce = $\frac{p-1}{2}$

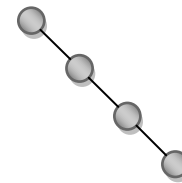


Lineární pole

Diametr = $p - 1$

Konektivita = 1

Šířka bisekce = 1

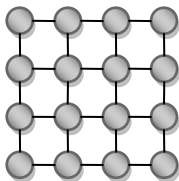


d-rozměrná mřížka

Diametr = $dp^{\frac{1}{d}}$

Konektivita = d

Šířka bisekce = $2p^{1-\frac{1}{d}}$

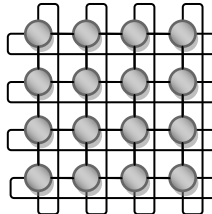


k-ární d-rozměrná kostka

Diametr = $dp^{\frac{1}{d}}$

Konektivita = d

Šířka bisekce = $2p^{1-\frac{1}{d}}$

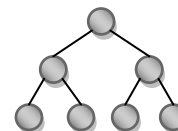


d-ární strom

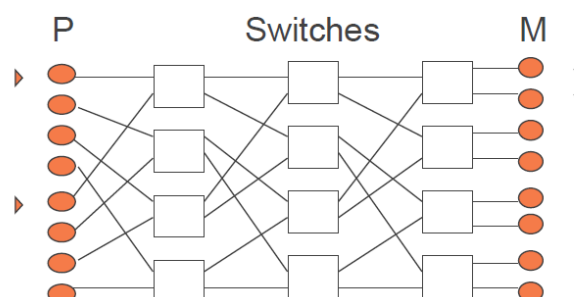
Diametr = $2 \log_d \frac{p+1}{2}$

Konektivita = 1

Šířka bisekce = 1



- **dynamické** – Uzly mohou být nejen procesory, ale i paměťovými moduly nebo přepínači. Často se využívají při implementaci architektur se sdílenou pamětí. Speciálním případem jsou tzv. **víceúrovňové propojovací sítě** (Omega, Butterfly, Benes), které spojují p procesorů s p paměťovými moduly pomocí přepínačů;



3 DISTRIBUOVANÉ A PARALELNÍ ALGORITMY A JEJICH SLOŽITOST

Všechny algoritmy lze analyzovat z různých hledisek, kdy můžeme tu kterou metriku $m(n)$ zařadit do příslušné třídy podle následujícího vzorce:

$$m(n) = \left\{ \overbrace{\underbrace{1}_{\text{sekvenční}}, \underbrace{c}_{\text{konstantní}}, \log n, n, n \cdot \log n}}^{\text{rozumný algoritmus}}, \overbrace{n^2, n^r, r^n}^{\text{nerozumný algoritmus}} \right\}$$

Obvykle používáme tyto metriky:

- **$p(n)$** je počet procesorů potřebných k řešení úlohy v závislosti na velikosti instance n ;
- **$t(n)$** je čas potřebný k řešení úlohy v jednotkách (krocích);
- **$c(n)$** je cena paralelního řešení dána součinem: $c(n) = p(n) \cdot t(n)$;

Algoritmus s optimální cenou, je algoritmus se sekvenčním časem. U ostatních algoritmů lze pozorovat hodnotu **zrychlení** $\left(\frac{t_{seq}(n)}{t(n)}\right)$ nebo **efektivnosti** $\left(\frac{t_{seq}(n)}{c(n)} \leq 1\right)$.

Složitost většinou rozumíme počet procesorů. Při výpočtu závislosti složitosti na délce vstupu je nejzajímavější nejhorší možný případ.

4 ŘEŠENÍ TYPICKÝCH PROBLÉMŮ PARALELISMU

Mezi typické problémy paralelismu patří synchronizace a komunikace. Tradičními úlohami jsou např. problémy producentů-konzumentů, večeřících písarů, aj.

viz POS

Mezi typické problémy paralelismu v rámci PRL pak patří řazení čísel, vyhledávání v posloupnostech, práce se seznamy, aj.

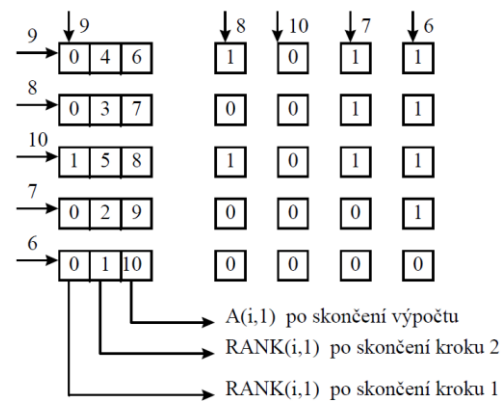
5 ALGORITMY ŘAZENÍ, ALGORITMY VYHLEDÁVÁNÍ, VEKTOROVÉ A MATICOVÉ ALGORITMY

5.1 ŘADÍCI ALGORITMY

Máme posloupnost n prvků $X = \{x_1, \dots, x_n\}$ a lineární uspořádání $<$, cílem **řazení** je pak vytvořit z prvků novou posloupnost $Y = \{y_1, \dots, y_n\}$, kde $\forall i \in \{1, \dots, n-1\}: y_i < y_{i+1}$. Přičemž v X si nejsou žádné dva prvky rovný.

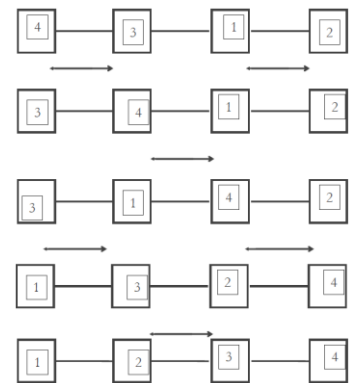
ENUMERATION SORT

- princip – správná pozice prvku ve výstupní seřazené posloupnosti je dána počtem prvků, které jsou menší než tento prvek;
- topologie – pro n prvků potřebujeme mřížku $n \times n$ procesorů, kde procesory jsou propojeny binárním stromem, kde každý procesor může uložit dva prvky do svých registrů A a B , přičemž je může pak porovnat a výsledek uložit do třetího registru $RANK$, pomocí strojového propojení pak může předat obsah kteréhokoli registru jinému procesoru;
- algoritmus – nejprve je každý prvek porovnán se všemi ostatními pomocí jedné řady procesorů, poté je správná pozice určena hodnotou registru $RANK$ a nakonec je prvek přesunut na správné místo;
- analýza – $t(n) = O(\log n), p(n) = n^2, c(n) = O(n^2 \log n) \Rightarrow$ NENÍ optimální.



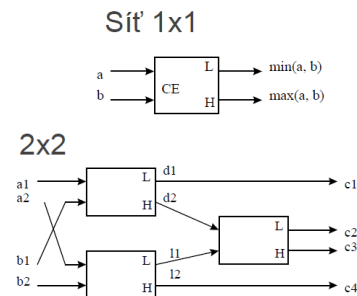
ODD-EVEN TRANSPORTATION SORT

- princip – paralelní bubble-sort s porovnáváním sousedů;
- topologie – lineární pole n procesorů s jedním registrem;
- algoritmus – na počátku každý procesor p_i obsahuje jednu z řazených hodnot y_i , v prvním kroku se každý lichý procesor spojí se svým sousedem p_{i+1} a vzájemně si porovnají hodnoty, je-li $y_i > y_{i+1}$, tak si prohodí hodnoty, v druhém kroku se svými sousedy spojí všechny sudé procesory a zopakují postup, nejpozději n krocích obsahují procesory od prvního k poslednímu seřazenou posloupnost;
- analýza – $t(n) = O(n), p(n) = n, c(n) = O(n^2) \Rightarrow$ NENÍ optimální, avšak algoritmus má nejlepší možnou časovou složitost dosažitelnou v lineární topologii.



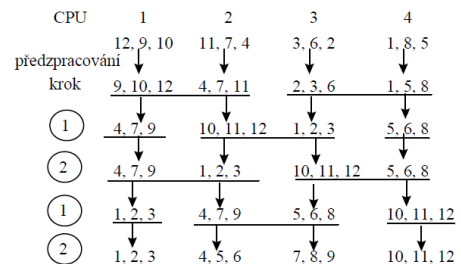
ODD-EVEN MERGE SORT

- princip – speciální řadící síť procesorů;
- topologie – v základní variantě síťově propojené procesory se dvěma vstupy, které jsou schopny seřadit na dvoj-výstup tak, že první výstup obsahuje menší a druhý větší z porovnávaných prvků;
- algoritmus – spočívá v kaskádovém zapojení sítě takovýchto procesorů, kde síť mohou používat i různé velké bloky (proměnlivé počty vstupů a výstupů), kdy buď začínáme od velkých bloků a postupujeme k menším nebo naopak;
- analýza – $t(n) = O(\log^2 n)$, $p(n) = n \log^2 n$, $c(n) = O(n \log^4 n) \Rightarrow$ NENÍ optimální.



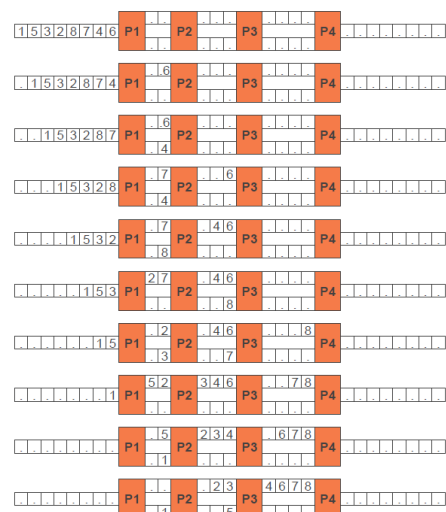
MERGE-SPLITTING SORT

- princip – lineární variantou odd-even transportation sortu, kde každý CPU se stará o více prvků;
- topologie – lineární pole N procesorů, kde každý obsahuje část řazené posloupnosti ($\frac{n}{2}$ čísel);
- algoritmus – procesor seřadí část posloupnosti, kterou obsahuje, pak se po krocích propojují vždy liché a pak sudé procesory, přičemž si propojující si vždy vezme menší polovinu čísel a propojuvaný tu větší polovinu čísel, nejpozději po $\frac{N}{2}$ iteracích je posloupnost seříděna;
- analýza – $t(n) = O\left(\frac{n \log n}{N}\right) + O(n)$, $p(n) = N$, $c(n) = O(n \log n) + O(nN) \Rightarrow$ což JE optimální, pokud počet procesorů $N \leq \log n$.



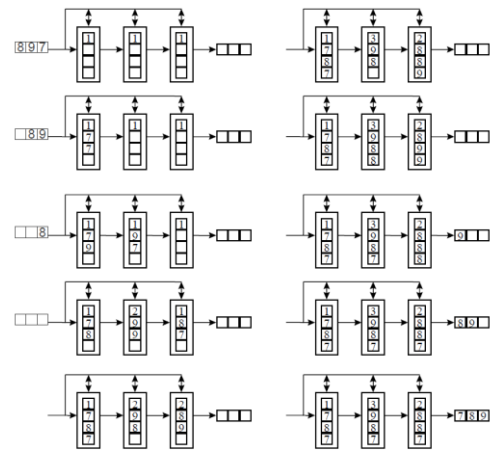
PIPELINE MERGE SORT

- princip – data nejsou uložena v procesorech, ale postupně do nich vstupují, přičemž každý procesor spojuje dvě seřazené posloupnosti;
- topologie – lineární pole $\log n + 1$ procesorů;
- algoritmus – podle pořadí dává procesor střídavě prvky ze vstupu do svých interních front, přičemž se vzrůstajícím pořadím roste krok střídání (a velikost interní fronty) mocninou 2, tzn., v prvním sloupci se střídá (1,2,1,2,...), v druhém (1,1,2,2,1,1,2,2,...), ve třetím (1,1,1,1,2,2,2,2,...);
- analýza – $t(n) = O(n)$, $p(n) = \log n + 1$, $c(n) = O(n \log n) \Rightarrow$ což JE optimální;



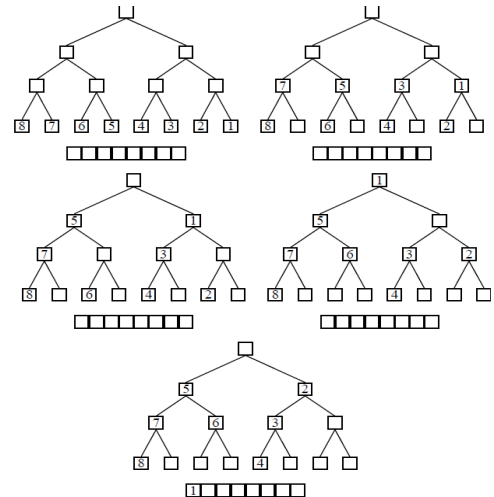
ENUMERATION SORT SE SBĚRNICÍ

- **princip** – místo mřížky použijeme lineární pole procesorů a společnou sběrnici schopnou v každém kroku přenést jednu hodnotu;
- **topologie** – lineární pole n procesorů, kde každý obsahuje čtyři registry: X_i (prvky posloupnosti x_i), Y_i (postupně prvky X), C_i (počet prvků menších než X_i), Z_i (seřazený prvek Y_i);
- **algoritmus** – nejprve se všechny registry C nastaví na 1, ze vstupu se přenesou čísla do volného registru X , provede se posun všech registrů Y a do prvního se replikuje vstup, procesory s neprázdnými registry X a Y provedou případnou inkrementaci registrů C , po vyčerpání vstupu a při odshiftování registrů Y , lze z registrů Z odebírat výslednou posloupnost;
- **analýza** – $t(n) = O(n)$, $p(n) = n$, $c(n) = O(n^2) \Rightarrow$ což NENÍ optimální.



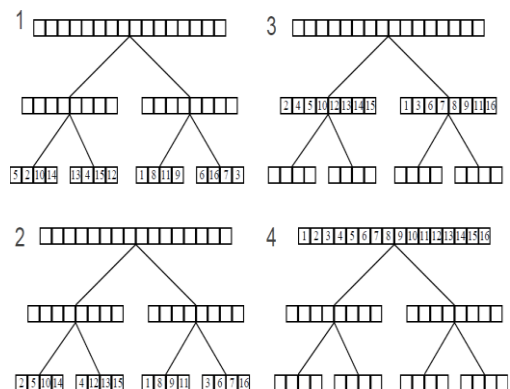
MINIMUM EXTRACTION SORT

- **princip** – stromem vybublává nahoru vždy nejmenší prvek;
- **topologie** – binární strom s n listy, hloubkou $\log n + 1$ a $2n - 1$ procesory, kde každý listový procesor obsahuje řazený prvek a každý nelistový procesor provádí porovnání;
- **algoritmus** – každý nelistový procesor porovná hodnoty svých dvou synů a menší z nich pošle svému otci, po $\log n + 1$ se první minimální prvek řazené posloupnosti dostane do kořene stromu;
- **analýza** – $t(n) = O(n)$, $p(n) = 2n - 1$, $c(n) = O(n^2) \Rightarrow$ což NENÍ optimální.



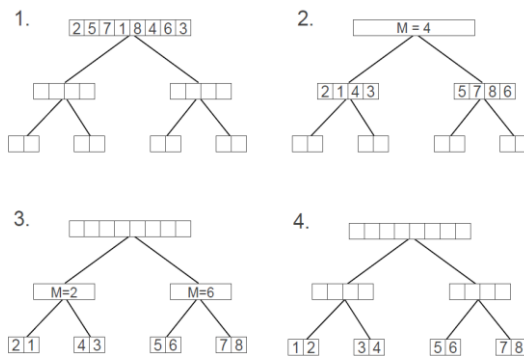
BUCKET SORT

- **princip** – merge splitting sort stromovou strukturou s m listovými procesory, kde $n = 2^m$;
- **topologie** – strom obsahuje 2^{m-1} procesorů, kde každý listový procesor obsahuje $\frac{n}{m}$ řazených prvků a umí je seřadit optimálním sekvenčním algoritmem (např. heap sort) a každý nelistový procesor pak umí spojit dvě seřazené posloupnosti;
- **algoritmus** – řazené prvky se rovnoměrně rozdělí mezi listové procesory, nejprve listové procesory seřadí své posloupnosti a pak postupně nelistové procesory spojují seřazené prvky do výsledné posloupnosti;
- **analýza** – $t(n) = O(n)$, $p(n) = (2 \log n) - 1$, $c(n) = O(n \log n) \Rightarrow$ což JE optimální.



MEDIAN FINDING AND SPLITTING

- **princip** – rozdělování posloupnosti mediánem;
- **topologie** – stejná jako bucket sort, tedy procesor na úrovni i zpracovává $\frac{n}{2^i}$ prvků, nelistový procesor umí nově najít medián optimálním algoritmem;
- **algoritmus** – jede se obráceným směrem, od vrcholu dolů k listům, kdy nelistový procesor postupně bere prvky své posloupnosti a když jsou menší než medián, tak je posílá svému potomkovi vlevo, jinak vpravo, a to do doby, než se zaplní listové procesory, které provedou finální setřídění;
- **analýza** – $t(n) = O(n)$, $p(n) = (2 \log n) - 1$, $c(n) = O(n \log n) \Rightarrow$ což JE optimální.

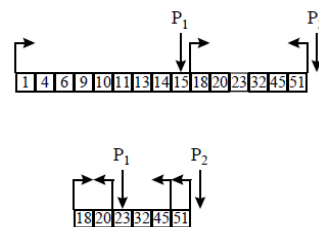


5.2 VYHLEDÁVACÍ ALGORITMY

Máme posloupnost n prvků $X = \{x_1, \dots, x_n\}$ a prvek \bar{x} , obecným úkolem **vyhledávacích algoritmů** je zjistit, zda $\forall i \in \{1, \dots, n\}: x_i = \bar{x}$ a případně zjistit i . **Optimální sekvenční algoritmus** pro problematiku vyhledávání závisí na tom, zda je vstupní posloupnost **neseřazená** (pak je časová náročnost $t(n) = O(n)$) nebo **seřazená** (pak je časová náročnost $t(n) = O(\log n)$).

N-ARY SEARCH

- **princip** – vyhledání v seřazené posloupnosti binární bisekcí;
- **topologie** – lineární pole N procesorů;
- **algoritmus** – každý procesor provede v rámci posloupnosti test, ve které polovině posloupnosti se námi hledané číslo nachází;
- **analýza** – $t(n) = O\left(\frac{\log(n+1)}{\log(N+1)}\right) = O(\log_{N+1}(n+1))$, $p(n) = N$, $c(n) = O(N \log_{N+1}(n+1)) \Rightarrow$ což NENÍ optimální.



UNSORTED SEARCH

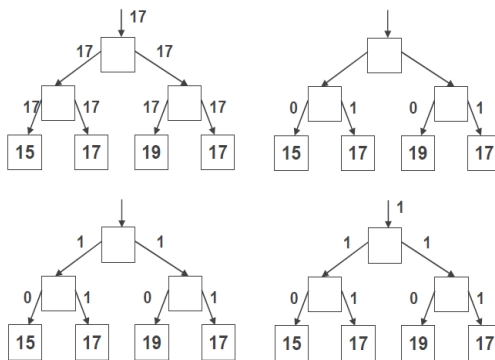
- **princip** – vyhledávání prvku v neseřazené posloupnosti;
- **topologie** – lineární pole N procesorů;
- **algoritmus** – paralelní mnohonásobné volání sekvenčního vyhledávání načteného prvku;
- **analýza** – podle použité architektury se liší výsledky:

$$\text{EREW a CREW: } t(n) = O\left(\log N + \frac{n}{N}\right), p(n) = N, c(n) = O\left(N \log N + \frac{n}{N}\right)$$

$$\text{CRCW: } t(n) = O\left(\frac{n}{N}\right), p(n) = N, c(n) = O(n) \Rightarrow \text{což JE optimální}$$

TREE SEARCH

- princip – vyhledávání v neseřazené posloupnosti pomocí stromové struktury;
- topologie – stromová architektura s $2n - 1$ procesory;
- algoritmus – kořen načte hledanou hodnotu \bar{x} a předá ji synům, až se dostane k listům, které obsahují prvky z prohledávané posloupnosti, všechny listy pak paralelně porovnají své x_i a \bar{x} s výsledkem 0 (rovná se) nebo 1 (nerovná se), nelistové procesory pak provádějí postupně logickou disjunkci hodnot svých synů, až se výsledek dostane do kořene, kde 1 znamená, že byl prvek nalezen a 0, že nebyl;
- analýza – $t(n) = O(\log n)$, $p(n) = 2n - 1$, $c(n) = O(n \log n) \Rightarrow$ což NENÍ optimální.



5.3 VEKTOROVÉ A MATICOVÉ ALGORITMY

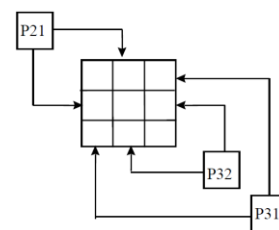
Transpozice matic má sekvenční složitost $O(n^2)$.

MESH TRANSPOSE

- princip – prvky se posílají maticí na svá nová místa v obou směrech;
- topologie – mřížka $n \times n$ procesorů, kde každý procesor má tři registry: A (který obsahuje vstupní hodnotu a výstup transpozice), B (hodnotu o pravého/horního souseda), C (hodnotu od levého/dolního souseda);
- algoritmus – vždy nejkrajnější prvky pošlou svou hodnotu sousedovi tak, aby dorazili na své místo (nejprve horizontálně a pak vertikálně);
- analýza – $t(n) = O(n)$, $p(n) = n^2$, $c(n) = O(n^3) \Rightarrow$ což NENÍ optimální.

EREW TRANSPOZICE

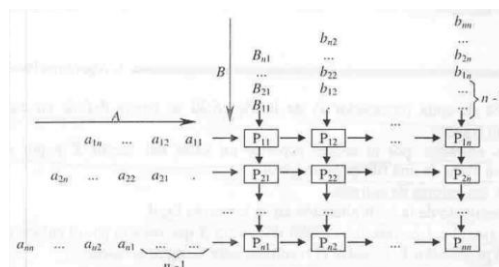
- princip – pokud je zajištěna exkluzivita přístupu, lze provést explicitní transpozici jednotlivých buněk matice;
- topologie – mřížka $n \times n$ procesorů;
- algoritmus – v jednom jediném kroku se spojí $\frac{n^2-n}{2}$ procesorů se svými protějšky odpovídající v matici místům transpozice a prohodí si hodnoty;
- analýza – $t(n) = O(1)$, $p(n) = n^2$, $c(n) = O(n^2) \Rightarrow$ což JE optimální.



Složitost optimálního algoritmu pro násobení matic není známa, odhaduje se $O(n^x)$, $2 < x < 3$.

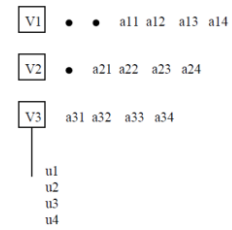
MESH MULTIPLICATION

- princip – využití matice pro postupné znásobování;
- topologie – mřížka $n \times k$ procesorů;
- algoritmus – prvky matic A a B se přivádějí do procesorů prvního řádku a prvního sloupce, odkud se pak každým dalším krokem postupně šíří výpočet do zbytku matice;
- analýza – $t(n) = O(n)$, $p(n) = n^2$, $c(n) = O(n^3) \Rightarrow$ což JE optimální.



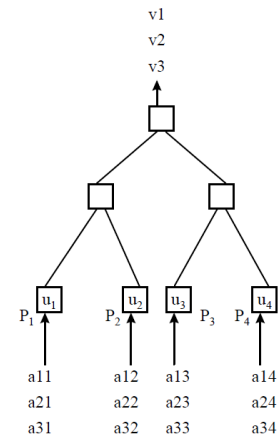
NÁSOBENÍ MATICE VEKTOREM

- **princip** – použít zjednodušený mesh multiplication tak, aby místo matice byl jedním ze vstupů vektor;
- **topologie** – lineární pole m procesorů, kde každý obsahuje prvek výstupního vektoru v_i ;
- **algoritmus** – stejným postupným způsobem se do nich zleva vsouvají prvky matice a z vrchu pak vektor;
- **analýza** – $t(n) = m + n - 1 = O(n)$, $p(n) = n$, $c(n) = O(n^2) \Rightarrow$ což JE optimální.



TREE MV MULTIPLICATION

- **princip** – časové nároky předchozího algoritmu lze ještě zlepšit použitím stromové etudy;
- **topologie** – architektura tedy obsahuje n listových a $n - 1$ nelistových procesorů;
- **algoritmus** – listové procesory násobí, nelistové sčítají;
- **analýza** – $t(n) = m + \log n - 1 = O(n)$, $p(n) = n$, $c(n) = O(n^2) \Rightarrow$ což JE optimální.



6 MODEL PRAM (PARALLEL RANDOM ACCESS MACHINE), SUMA PREFIXŮ A JEJÍ APLIKACE

6.1 PRAM

Model je rozhraní, které odděluje aplikaci od architektury.

PRAM (Parallel Random Access Machine) je synchronní model výpočtu, kde procesory komunikují pomocí sdílené paměti. Samotný RAM se skládá z procesoru (celočíslné i operace s pohyblivou čárkou) a paměti (kterou vyváří obecně neomezený počet registrů). Je to alternativní model k paralelnímu Turingovu stroji. Všechny RAMy jsou pak řízeny jedním společným programem. Výpočet probíhá pro kroci synchronně (čtení/zápis do sdílené paměti, lokální operace), přičemž procesory mohou používat i svůj index (unikátní číslo procesoru).

Omezením přístupu k paměti se zabývají dříve uvedené architektury EREW, ERCW, CREW a CRCW. U architektury CRCW je ještě dále potřeba specifikovat chování v případě zápisového konfliktu třemi způsoby:

- **common** – všechny zapisované hodnoty musí být shodné;
- **arbitrary** – zapisované hodnoty mohou být různé, zapíše se libovolná z nich;
- **priority** – procesory mají pevné priority, zapíše se hodnota toho nejprioritnějšího.

Relace $A \geq B$ udává vztah mezi algoritmy na architekturách tak, že co běží na B , bude beze změn běžet i na A (A je stejně tolerantní nebo tolerantnější k zápisovým konfliktům). Pak platí:

$$\text{priority} \geq \text{arbitrary} \geq \text{common} \geq \text{CREW PRAM} \geq \text{EREW PRAM}$$

Algoritmus Broadcast:

- princip - pro optimalizované rozdělování jedné hodnoty \bar{x} mezi všemi procesory;
- topologie - lineární pole n procesorů;
- algoritmus - v prvním kroku P_1 přečte \bar{x} a zpřístupní jej P_2 , v druhém kroku P_1 a P_2 zpřístupní hodnotu P_3 a P_4 , a tak to pokračuje, dokud neznají hodnotu všechny procesory.
- analýza - existuje několik variant pro různé architektury:
 - sekvenční: $t(n) = O(n)$
 - CREW: $t(n) = O(c)$
 - EREW: $t(n) = O(\log n)$

6.2 SUMA PREFIXŮ

Suma prefixů (někdy také **all-prefix-sum** nebo **allsums** či **scan**) je jeden ze základních kamenů stavby paralelních systémů. Suma prefixů je tedy operace, jejímž vstupem je binární asociativní operátor \oplus a uspořádaná posloupnost n prvků $(a_0, a_1, \dots, a_{n-1})$, jejímž výstupem je vektor $(a_0, a_0 \oplus a_1, \dots, a_0 \oplus a_1 \oplus \dots \oplus a_{n-1})$.

Operace **prescan** je variantou scanu, která pracuje s neutrálním prvkem I , její výsledek je pak posloupnost $(I, a_0, a_0 \oplus a_1, \dots, a_0 \oplus a_1 \oplus \dots \oplus a_{n-2})$.

Operace **reduce** má stejný vstup jako scan, ale vrací jen poslední prvek posloupnosti, tedy $a_0 \oplus a_1 \oplus \dots \oplus a_{n-1}$.

SCAN, ALLSUMS

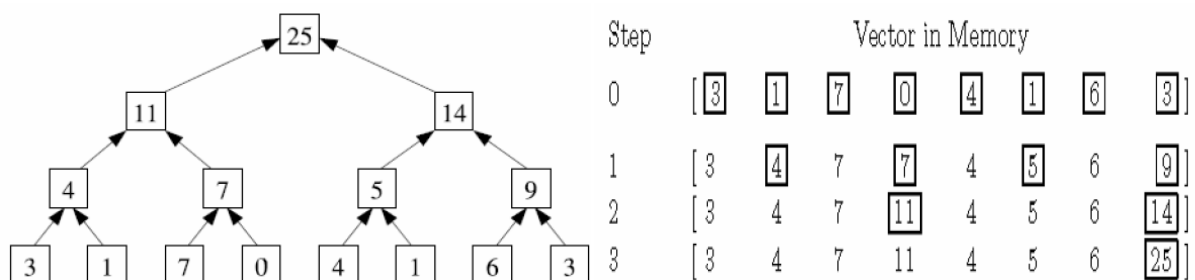
$I, a_0, a_0 \oplus a_1, \dots, a_0 \oplus a_1 \oplus \dots \oplus a_{n-2}, a_0 \oplus a_1 \oplus \dots \oplus a_{n-1}$

PRESCAN

REDUCE

PARALELNÍ VÝPOČET REDUCE

- princip - využití stromové struktury pro spočítání reduce za předpokladu, že je \oplus asociativní;
- topologie - stromová struktura $\frac{n}{2}$ procesorů (pro každou dvojici vstupu stačí jeden procesor) o výšce $\log n$;
- algoritmus - procesor sečte dvojici mu příslušejících čísel, přičemž v každém kroku stačí o polovinu méně procesorů než v předchozím, při použití sdílené paměti;
- analýza - $t(n) = O(\log n)$, $p(n) = \frac{n}{2}$, $c(n) = O(n \log n) \Rightarrow$ což NENÍ optimální;



VYLEPŠENÁ VARIANTA REDUCE

- **princip** – lze zlepšit, pokud máme dostatečně malé množství procesorů N , kde musí platit, že $\log N < \frac{n}{N}$, každý z procesorů;

$$\begin{array}{cccc} \underbrace{[4 \quad 7 \quad 1]}_{\text{processor 0}} & \underbrace{[0 \quad 5 \quad 2]}_{\text{processor 1}} & \underbrace{[6 \quad 4 \quad 8]}_{\text{processor 2}} & \underbrace{[1 \quad 9 \quad 5]}_{\text{processor 3}} \\ \text{Processor Sums} & = & [12 & 7 & 18 & 15] \\ \text{Total Sum} & = & 52 \end{array}$$
- **topologie** – stejná jako v případě nahoře;
- **algoritmus** – každý procesor provede sekvenční reduce pro svůj kousek posloupnosti o délce $\frac{n}{N}$ a výsledky těchto se pak zpracovávají už paralelně jako reduce;
- **analýza** – $t(n) = O\left(\frac{n}{N}\right)$, $p(n) = N$, $c(n) = O(n) \Rightarrow$ což JE optimální;

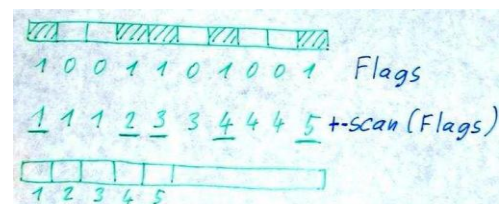
PARALELNÍHO VÝPOČTU PRESCAN na architektuře PRAM:

- **princip** – použití stromové struktury a její projití nejprve nahoru a pak zase dolů;
- **topologie** – stejná jako u základního reduce;
- **algoritmus** – Se skládá ze dvou částí **Upsweep** a **Downsweep**. V Upsweepu je algoritmus stejný jako reduce, jen každý uzel si pamatuje mezisoučet. V Downsweepu se kořen nastaví na hodnotu neutrálního prvku I , provede se $\log n$ kroků, ve kterých se pracuje vždy jen s procesory v dané úrovni, a to tak, že uzel dá svému pravému potomku svojí hodnotu \oplus hodnotu levého potomka a svému levému potomku dá svou hodnotu;
- **analýza** – složitost je stejná jako u (vylepšeného) reduce;

Step	Vector in Memory							
0	[3]	[1]	[7]	[0]	[4]	[1]	[6]	[3]
up	1	[3]	[4]	7	[7]	4	[5]	6
	2	[3]	4	7	[11]	4	5	6
	3	[3]	4	7	11	4	5	6
clear	4	[3]	4	7	11	4	5	6
down	5	[3]	4	7	[0]	4	5	6
	6	[3]	[0]	7	[4]	4	[11]	6
	7	[0]	[3]	[4]	[11]	[11]	[15]	[16]

6.3 VYUŽITÍ SUMY PREFIXŮ

Packing problem kdy máme k vstupních položek rozmístěných v poli o n pozicích (kde $k < n$) a kde je cílem vytvořit výstupní pole, kde položky zaujmají první pozice. Řešení leží ve spočtení pole binárních příznaků (1 – položka existuje, 0 – položka neexistuje), provést scan s operací + nad tímto polem a přesunout položky na správné pozice.



Problém viditelnosti, kdy máme zjistit, které body podél paprsku vycházejícího z místa X pozorovatele jsou viditelné, pokud máme na vstupu pozici X a matici terénu ve formě nadmořských výšek. Bod je viditelný, pokud žádný jiný bod mezi jím a pozorovatelem nemá větší vertikální úhel. Algoritmus řešení nejprve vytvoří vektor výšek bodů podél pozorovacího paprsku, pak přepočítá vektor výšek na vektor úhlů a nakonec pomocí operace prescan s operátorem MAX spočte vektor maximálních úhlů a pro zjištění viditelnosti bodu se určí úhel a porovná se s maximem.

Carry look-ahead parallel binary adder je paralelní sčítačka, která dvě binární čísla sečte v $\log n$ krocích, a to tak, že si předvypočítá všechny bity přenosu. Nejprve si spočítá pole $D = d_{n-1}, \dots, d_0$, kde $d_i \in \{\text{propagate}, \text{stop}, \text{generate}\}$. Dále se spočítá \odot -scan pole D , díky kterému se dostanou všechny bity přenosu v logaritmicke časě.

Paralelní radixsort nad n prvky je založen na radixu 2, kdy se v každém kroku beru v úvahu 1 bit klíče a pomocí operace *split* se přesouvají prvky s nulovým bitem na začátek a jedničkovým na konec posloupnosti. Pro prvky

s nulovým bitem se jejich pozice ihned zjistí pomocí \oplus -prescanu na invertované pole bitů, pro prvky s jedničkovým bitem pak pomocí \oplus -scanu a výsledek se odečte od n .

A	=	[5 7 3 1 4 2 7 2]
Flags	=	[1 1 1 1 0 0 1 0]
I-down	=	[0 0 0 0 0 1 2 2]
I-up	=	[3 4 5 6 7 7 7 8]
Index	=	[3 4 5 6 0 1 7 2]
permute(A, Index)	=	[4 2 2 5 7 3 1 7]

Operace **segmentovaného scanu** je definována nad asociativním binárním operátorem \oplus , uspořádanou posloupností n prvků $(a_0, a_1, \dots, a_{n-1})$ a uspořádanou posloupností příznaků (jež mohou nabývat jen dvou hodnot 0,1) $(f_0, f_1, \dots, f_{n-1})$, která vrací posloupnost $(s_0, s_1, \dots, s_{n-1})$, kde v s_i jsou sumy prefixů přes jednotlivé segmenty, přičemž hranice segmentů jsou dány hodnotou 1 v poli příznaků.

$$\begin{aligned}
 A &= (5, 1, 3, 4, 3, 9, 2, 6) \\
 F &= (1, 0, 1, 0, 0, 1, 0) \\
 \text{segmentovaný +scan} &= (5, 6, 3, 7, 10, 19, 2, 8) \\
 \text{segmentovaný MAX-scan} &= (5, 5, 3, 4, 4, 9, 2, 6)
 \end{aligned}$$

Paralelní quicksort používá u své paralelizace systém, kdy se jeden z prvků vybere jako pivot (buď první, nebo medián pole, nebo náhodně), kde se porovnání s tímto pivotem prvky rozdělí do tří skupin – větší, menší, rovny – a pro každou se pak rekurzivně volá další kolo paralelního quicksortu. V tomto případě účelně využijeme segmentovaného copy scanu a operace split z výše uvedeného radixsortu.

klíč	6,4	9,2	3,4	1,6	8,7	4,1	9,2	3,4
flags	1	0	0	0	0	0	0	0
Pivots	6,4	6,4	6,4	6,4	6,4	6,4	6,4	6,4
F	==	>	<	<	>	<	>	<
split	3,4	1,6	4,1	3,4	6,4	9,2	8,7	9,2
flags	1	0	0	0	1	1	0	0
pivots	3,4	3,4	3,4	3,4	6,4	9,2	9,2	9,2

7 SUMA SUFIXŮ, EULEROVA CESTA, ALGORITMY NAD SEZNAMY, STROMY A GRAFY

7.1 ALGORITMY NAD SEZNAMY A SUMA SUFIXŮ

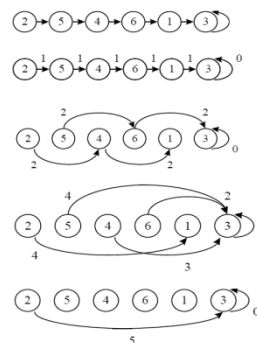
Tato skupina algoritmů pracuje s různě provázanými seznamy (každý prvek má následníka nebo předchůdce i následníka). Každý prvek tak má hodnotu a případné ukazatele.

Predecessor computing:

- princip – dopočítání ukazatelů na předchůdce, je-li k dispozici lineární seznam následníků;
- topologie – lineární pole n procesorů;
- algoritmus – procesory paralelně přistoupí přes ukazatel ke svým následníkům a vytvoří u nich ukazatel na předchůdce za použití svého indexu, je potřeba ošetřit první a poslední prvek;
- analýza – $t(n) = O(c), p(n) = n, c(n) = O(n)$.

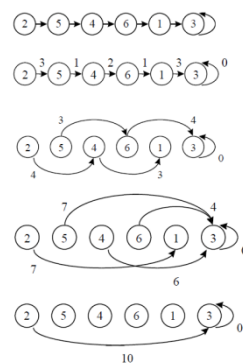
List ranking:

- princip – nalezení pořadí (*rank*) prvků seznamu (vzdálenost od konce seznamu);
- topologie – lineární pole n procesorů;
- algoritmus – používá se technika postupného zdvojování cesty;
- analýza – $t(n) = O(\log n), p(n) = n, c(n) = O(n \log n) \Rightarrow$ což NENÍ optimální.



Suffix je podseznam mezi prvkem a koncem seznamu. Operace **sumy suffixů** je tedy stejná jako suma prefixů, jen s tím, že pracuje se suffixy. Algoritmus pro paralelní sumu suffixů:

- princip – varianta list rankingu se vstupním podseznamem;
- topologie – lineární pole n procesorů;
- algoritmus – stejný jako list ranking, jen s tím rozdílem, že zde existuje vstupní pole, které je podseznamem (u list rankingu je tímto polem ohodnocení všech hran 1);
- analýza – $t(n) = O(\log n), p(n) = n, c(n) = O(n \log n) \Rightarrow$ což NENÍ optimální.



Algoritmus **vylepšeného list rankingu** odstraňuje provádění zbytečné práce tím, že v každém kole přestává pracovat polovina procesorů a celý algoritmus pracuje v jumping fázi a rekonstrukční fázi (kdy se přičítají mezivýsledky). Jak však v paralelismu určit, kdo je ten druhý, toto řeší varianty tohoto algoritmu jako **Random mating** (každý procesor si hodí korunou a přiřadí si pohlaví (žena nebo mutant), pokud je procesor ženského pohlaví a jeho následník je mutant, pak se prvek přeskočí a procesor se uvolní, protože žena s mutantem se nemůže pářit) nebo **Optimal list ranking** (je potřeba pevný počet stále pracujících procesorů – nejlépe $\frac{n}{\log n}$).

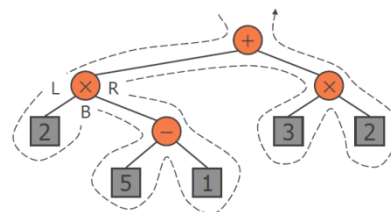
kde každý procesor má zásobník, přičemž se pokouší přeskočit následníka prvku na vrcholu zásobníku, pokud se mu to podaří, procesor se zabývá dalším prvkem na zásobníku).

k-obarvení seznamu je zobrazení $C: V \rightarrow \{0, 1, \dots, k-1\}$, kde pro $\forall i \in V: C_i \neq C_{i+1}$, řečeno selsky, barvy dvou sousedních vrcholů se neshodují. **2log n coloring** využívá index procesoru k určení barvy. Hodnota k je index nejnižšího bitu ID, ve kterém se sousedé liší, pak je barva $C = 2k + ID[k]$.

k-Ruling set (množina oddělovačů) je podmnožina seznamu taková, že žádné vybrané vrcholy nesousedí a je mezi nimi maximálně k nevybraných prvků. **2k-ruling set from k-coloring** vybere prvek tehdy, když jeho barva má menší index než předchůdce a následníka.

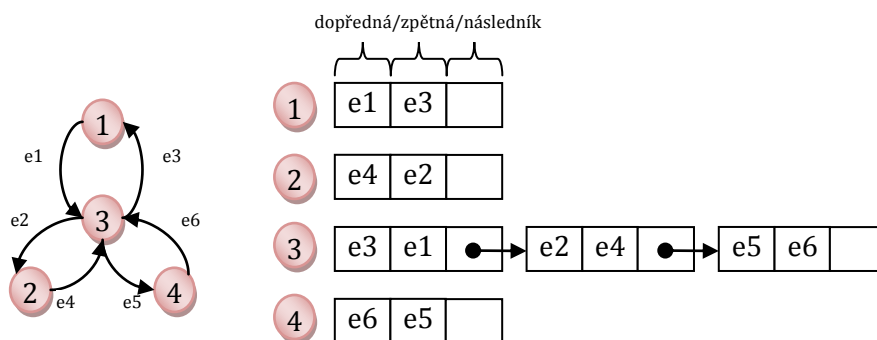
7.2 ALGORITMY NAD STROMY

Binární strom je abstraktní datová struktura, kde mezi sebou existují vazby otce (index i) a levého ($2i$) a pravého ($2i+1$) syna. Speciálním případem průchodu binárním stromem je preorder, inorder a postorder.



Strom (tree) je obecně orientovaný graf, který tvoří uspořádaná dvojice uzlů (vertices) a hran (edges) $T = (V, E)$. **Eulerovský graf** je takový, který vznikne z normálního nahrazením neorientovaných hran vždy dvojicí hran orientovaných.

Eulerovská kružnice je reprezentována funkcí následníka E_{tour} , kdy máme-li Eulerovský graf $\bar{T} = (V, \bar{E})$, pak $\forall e \in \bar{E}$ přiřazuje hranu $E_{tour}(e) \in \bar{E}$, která následuje e . Takovou funkci lze uložit a reprezentovat seznamem sousednosti:

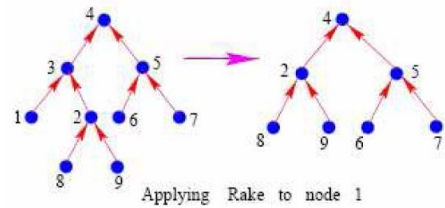


Eulerova cesta je tedy obecný případ průchodu stromem, kdy jím procestuje eulerovskou kružnicí. **Kořen stromu** je uzel stromu, ze kterého se rozhodneme eulerovskou kružnicí procházet, místo, kde přerušujeme eulerovskou cestu.

Mezi vyřešené paralelizovatelné problémy, které využívají výpočtu Eulerovy cesty, patří **Eulerova cesta s pozicemi** (vzdálenost hrany při průchodu eulerovskou hranou od kořene), **nalezení rodičů** (při každé dopředné hraně zaznamenáme, že rodičem uzlu, do kterého vstupujeme je uzel, ze kterého odcházíme), **přiřazení pořadí preorder vrcholům** (je dáno součtem 1 a všech dopředných hran, kterými jsme po cestě prošli).

Operace RAKE pro binární strom, kde je v nekořenový uzel, $parent(v)$ je rodičovským uzlem v a $sibling(v)$ je druhým potomkem $parent(v)$, je definována pro libovolný list stromu u , jehož rodič není kořenem tak, že:

- ze stromu odstraní u a $parent(u)$;
- připojí sourozenecký podstrom $sibling(u)$ na původní místo $parent(u)$.



Paralelní RAKE nesmí být aplikován na sousedící uzly (konflikt), snaha co nejvíce RAKE najednou:

- 1) Označíme listy zleva doprava;
- 2) Krajní se vyřadí (zůstanou jako poslední, až bude strom redukován na 2 listy a kořen);
- 3) Nejprve se volá RAKE na liché a pak na sudé listy takto upraveného seznamu vrcholů;
- 4) Redukce stromu za $t(n) = O(\log n)$.

Při **Tree contraction** se řeší paralelní vyhodnocení aritmetického výrazu uloženého ve formě stromu, kde každý list obsahuje operand a každý list operátor. Tato technika opakovaně zmenšuje strom, až do velikosti jednoho vrcholu, a to tím, že spojuje listy do rodičovského uzlu. Používá se právě operace RAKE v co největší míře (musí se ohlídat, aby nebyla RAKE aplikována současně na vrcholy, jejichž rodiče ve stromě sousedí).

8 INTERAKCE MEZI PROCESY A TYPICKÉ PROBLÉMY PARALELISMU

Interakci můžeme rozdělit na kooperaci (je potřeba synchronizace) nebo soupeření (je potřeba vzájemné vyloučení).

Kritická sekce KS je část programu, ve které dochází k přístupu ke sdíleným prostředkům, obvykle maximálně jeden proces se může v daném čase nacházet v kritické sekci, aby bylo dodrženo integrity hodnot. Obvykle řešíme tak, že každé sdílené proměnné přidělíme vlastní prostředek vzájemného vyloučení (semafor, monitor).

Monitor je abstraktní struktura schopná vzájemného vyloučení (**mutex** = mutual exclusion), kdy na sdílenou proměnnou je dán zámek při přístupu nějakým programem. Synchronizace je pak zaručeno zavedením podmínky, která musí být splněna, aby mohl (jiný) proces vstoupit do KS, jinak je zařazen do fronty čekání.

Typické problémy paralelismu procesů jsou **problém večeřících filozofů** či **producent-konzument**.

9 PŘEDÁVÁNÍ ZPRÁV A JAZYKY PRO PARALELNÍ ZPRACOVÁNÍ

9.1 PŘEDÁVÁNÍ ZPRÁV

Nejprve definujeme nutná primitiva, a to operace:

- `send(channel, message)`
- `receive(channel, message)`

Kanály umožňují komunikaci mezi procesy. Kanály lze dělit na:

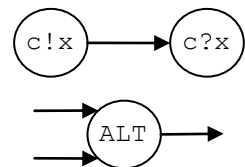
- **asynchronní** – neomezená kapacita, neblokující `send`, složité na implementaci (např. email);
- **synchronní** – omezená kapacita, operace `send` blokuje, lze simulovat asynchronním kanálem za použití potvrzení (acknowledge).

OCCAM

Je formalismem a jazykem založeným na CSP (communicating sequential processes), vyvinutý Tonym Děvkou a Davidem Mayem z Anglie. Primitivy v OCCAMu jsou procesy pěti různých druhů:

- přiřazení (`x := y + 2`);
- vstup (`keyboard ? char`);
- výstup (`green ! char`);
- SKIP (NOOP, který ukončuje);
- STOP (NOOP, který neukončuje).

OCCAM používá nebufferované dvoubodové kanály, které deklarují typ protokolů. Používá dvě programové závorky, které určují způsob zpracování SEQ (sekvenční) a PAR (paralelní). Dále obsahuje instrukci ALT pro alternativní výběr mezi dvěma běhy.



Aplikace v OCCAMu je sepsána jako síť komunikujících procesů. **Konfigurace** mapuje komponenty na fyzické procesory a linky, které realizují paralelní proces, přičemž samotná konfigurace by nijak neměla ovlivnit správnost.

ADA

Je imperativní objektový jazyk. Existuje v něm struktura úkol (task), která představuje jeden sekvenční proces, který obsahuje lokální data. Jednotlivé úlohy mezi sebou komunikují pomocí spouštění vstupních bodů dalších úkolů a *rendezvousingují* se nad takto spuštěnými úlohami. Významné jsou dvě konstrukce, a to `accept` (která slouží v případě očekávání zprávy) a `select`, jenž provádí výběr z více typů došlých zpráv.

LINDA

Je paralelní koordinační programovací jazyk založený na jazyku C, který spoléhá na asynchronní asociativní komunikaci pomocí sdíleného globálního prostoru zvaného **Tuple space**. Kde **tuple** (*n*-tice) je fundamentální datová struktura tvořená polem dat, která mohou být buď **actual field** (je vyhodnoceno před odesláním) nebo **formal field** (je předáno jako proměnná). Využívá

operace vkládající na nástěnku `out` (sekvenčně data) a `eval` (paralelně procesy) a operace vybírající z nástěnky, kterými jsou `in` (vyjme tuple z nástěnky) a `rd` (přečte data z nástěnky bez jejich vyjmutí).

9.2 JAZYKY PRO PARALELNÍ ZPRACOVÁNÍ

Trendem dneška jsou clustery počítačů provádějící distribuovaný výpočet, nástrojem pro jejich programování a komunikační protokol jsou jazyky PVM a MPI.

PVM

Je vyvíjeno od roku 1989 jako distribuovaný operační systém, který je přenositelný na různé platformy a umí se vypořádat se selháními. Je to tedy heterogenní koncepce, která zavádí **virtuální stroj**, jenž představuje rozběhlou paralelní aplikaci, každá stanice má tak spuštěného svého démona. PVM díky tomu může při běhu aplikace vytvářet popřípadě rušit procesy, je tedy **dynamické**. PVM je schopno vypořádat se s pádem procesu a přerozdělit práci jinému volnému, pokud však nedošlo k pádu **master procesu** (řídí činnost ostatních).

MPI (MESSAGE PASSING LIBRARY)

Je knihovna pro psaní aplikací, která je velmi výkonná a má dobře definované chování, která se vyvíjí za přispění konsorcia firem od roku 1992. Na začátku běhu programu jsou množině procesorů přiděleny procesy ke zpracování, každý procesor pak zná svůj rank (`MPI_COMM_RANK`) a ví celkový počet všech procesorů (`MPI_COMM_SIZE`). Od startu programu žádné nové procesy nemohou vznikat, mohou však vznikat skupiny úkolů, je tedy **statické**. Při pádu některého z procesů dojde k zastavení celého systému.

MPI všeobecně rozšiřuje model předávání zpráv, není jazykem a překladačem podmíněná a není zaměřena na žádnou určitou platformu – usnadňuje život koncovým uživatelům a programátorům. Systém zasílání zpráv dělá výměnu dat kooperativní (data jsou na jedné straně explicitně zaslána a na druhé čtena), umožňuje ale i jednostrannou komunikaci např. vzdálené operace s pamětí. Procesy jsou děleny do **skupin**, každá zasláná zpráva musí obsahovat **kontext**, přičemž dohromady skupina a kontext tvoří **komunikátor**. Zasláné zprávy lze označovat i **tagem**, který pomáhá v procesu zpracování příjemci v rámci dané skupiny. MPI obsahuje kvůli přenositelnosti mezi architekturami i vlastní definice obvyklých datových typů.

MPI obsahuje např. tyto zajímavější funkce:

- `MPI_SEND(start, count, datatype, dest, tag, com);`
- `MPI_RECV(start, count, datatype, dest, tag, com, status);`
- `MPI_BCAST()` – je schopna redistribuovat data z jednoho procesu ostatním;
- `MPI_REDUCE()` – kombinuje data ze všech procesů v rámci komunikátoru a vrací je v jediném procesu;
- `MPI_SCAN()`, `MPI_GATHER()`, `MPI_SCATTER()`;
- `MPI_BARRIER()` – prostředek synchronizace;
- `MPI_MAX()`, `MPI_MIN()`, `MPI_SUM()`, `MPI_BOR()`, `MPI_LXOR()`, ...

