1 Paralelní a distribuované architektury

Flynnova klasifikace procesorů:

- SISD – konvenční procesory
- SIMD – vektorové procesory
- MISD – řetězové procesory
- MIMD
  - multiprocesory (komukují sdílenou pamětí)
  - multicomputery a distribuované systémy (zasílání zpráv)

Flynnova klasifikace popisuje pouze Von-Neumannovské architektury. Data-flow a reukční architektury jsou také paralelní architektury.

Architektura data-flow nemá program a PC (Program Counter), provádí interpretaci grafu toku dat. Do uzlu vstupují operandy a vystupuje výsledek. V paměti udržuje struktury pro uzly, které definují operace, vstupy a výstupy, propojeno do orientovaného grafu. Paralelismus spočívá v několika operačních jednotkách, které vybírají události a pouští je grafem.

Redukční počítač bere výraz a nahrazuje jeho části výsledkem dané operace (`2*3` na `6`). Program se převede na strom a po zadání hodnot se strom redukuje až na jeden uzel.

VLIW (Very Long Instruction Word) je SISD, jedna velmi dlouhá instrukce (představuje několik menších pro různé procesory). Tyto procesory mohou pracovat paralelně, pokud je to možné (nejsou kolize, nečeká se na mezivýsledek). Jednoduchá implementace a škálovatelnost, ale některé instrukce špatné (skok využije jedinou podinstrukci, mezivýsledky apod). Pohybují se někde mezi statickými a dynamickými superskaláry.

Statické superskalární procesory zpracovávají sekvenční program na více procesorech *in-order*, takže paralelnost je jen za souběhu správných instrukcí.

Dynamické superskalární procesory zpracovávají sekvenční program na více procesorech *out-of-order*, např. spekulativní výpočty za skokem.

MISD jsou lineárně propojené procesory, řeší proudové úlohy, sekvenční průchod.

Zřetězené procesory využívájí několik paralelních postupů provádění (pipelines), protože jednotlivé části výpočetního řetězce by jednou pipeline byly nevyužity. Některé jednotky se pak mohou vyskytovat vícekrát (ALU).

SIMD jsou vektorové procesory, paralelně se provádí stejná instrukce na n procesorech a n částech vstupních dat (MMX, SSE). Rozlišují se podle toho, zda každý procesor má vlastní paměť, nebo je přístupná všem a alokuje se. Jednoduchá implementace, latence a *synchronizace* oproti MIMD. Ne všechny problémy jsou vhodné, nevyplatí se v malém počtu.

Granularita paralelismu – na úrovni instrukcí, mezi instrukcemi, mezi příkazy, mezi bloky procesu, mezi procesy

Synchronizace zajišťuje dodržení požadovaných časových vztahů. Řeší soupeření a kooperaci, používá zasílání zpráv, rendezvous, semafor, monitor a bariéru.

Komunikace je přenos dat mezi paralelními procesy, hlavními prostředky je zasílání zpráv a sdílená paměť.

2 Topologie

Multitasking – 1 CPU přepíná kontext (virtuální procesor), paměť je sdílená, předávání zpráv simulováno SW

Systém se sdílenou pamětí – CPU mají svou cache, zbytek na sběrnici (boj), předávání zpráv může být v HW nebo simulace SW

Virtuální sdílená paměť – CPU má svou paměť, ale je virtuálně spojena v simulovanou sdílenou, opět HW/SW simulované zasílání zpráv

Systém s předáváním zpráv – CPU vázány volně (např. počítačová síť), sdílená paměť simulovaná SW

Sdílená paměť:

- Všechny procesory mají přístup do celého paměťového prostoru.
- Řešení současného přístupu k jedné buňce:
  - EREW – Exclusive Read, Exclusive Write (velmi omezující)
  - CREW – Concurrent Read, Exclusive Write (časté, jednoduché)
  - ERCW – Exclusive Read, Concurrent Write (nedává smysl)
  - CRCW – Concurrent Read, Concurrent Write (složité)

Předávání zpráv:

- Každý procesor má vlastní adresový prostor.
- Také každý procesor má vlastní fyzickou paměť, přístup jinam komunikací.

Statické propojovací sítě:

- Všechny uzly jsou procesory.
- Všechny hrany jsou komunikační kanály.
- Neobsahují sdílenou paměť.
- *Průměr* je nejdelší délka nejkratších cest mezi všemi dvojicemi uzlů.
- *Konektivita* je minimální počet hran, které je nutné odstranit pro rozdělení na dvě části.
- *Šířka bisekce* je minimální počet hran, které spojují dvě přibližně stejně velké části sítě.

Typická statická propojení:

- Úplné propojení
- Hvězda
- Lineární pole
- D-rozměrná mřížka
- K-ární d-rozměrná kostka
- D-ární strom

Dynamické propojovací sítě:

- Uzly jsou procesory, paměťové moduly nebo přepínače.

- Často implementují sdílenou paměť.
- Implementace: křížový přepínač, sběrnice, ...

Víceúrovňové sítě spojují p procesorů s p paměťovými moduly pomocí $p.log(p)$ přepínačů.

3 Distribuované a paralelní algoritmy a jejich složitost

Počet procesorů p je odvozen on délky vstupu n. $p(n) = \{1, c, log(n), n, n.log(n), n^2, \ldots, n^r, r^n\}$.

Čas výpočtu t je také odvozen od n a je udáván v jednotkách (krocích).

Cena algoritmu $c(n) = p(n).t(n)$. Algoritmus s optimální cenou je stejně drahý jako sekvenční algoritmus (jde o cenu, ne rychlost) $c_{opt}(n) = t_{seq}(n)$.

Zrychlení paralelizací je dáno vztahem $t_{seq}(n)/t(n)$, efektivnost pak $t_{seq}(n)/c(n)$, nastavení je závislé na případu použití.

Složitostí většinou rozumíme počet procesorů. Při výpočtu závislosti na délce vstupu je nejzajímavější nejhorší případ, takže pokud jedna část algoritmu vyžaduje $p(n)$ procesorů a druhá $p(n^2)$ procesorů, výsledná složitost je $p(n^2)$.

4 Algoritmy řazení

Řazení bere posloupnost prvků a podle relace $>$ je seřadí. Zjednodušujeme na řazení pomocí operace porovnání. Také předpokládáme, že v posloupnosti nejsou žádné dva prvky rovny. Všechny posloupnosti tak lze vyjádřit (můžeme přidat index - složitost $O(n)$).

Optimální složitost podle sekvenčního algoritmu je $c(n) = O(n.log(n))$.

Enumeration sort:

- *Princip:* výsledná pozice prvku je dáná počtem prvků, které jsou menší
- *Topologie:* mřížka n krát n, řádky a sloupce jsou binární stromy v poli
- *Procesory:* registry A,B,RANK; do A,B zápis prvku, RANK inkrementace; možnost poslat registr synům
- *Algoritmus:* pomocí jedné řady se prvky porovnají (a přitom se mění RANK); správná pozice je RANK; nakonec se prvky přesunou stromem
- *Složitost:* $t(n) = O(log(n))$ – nejrychlejší paralelní řešení; $c(n) = O(n^2.log(n))$ – není optimální

Odd-even transposition sort

- *Princip:* paralelní bubble-sort, porovnávají se jen sousedé a mohou se přehodit
- *Topologie:* lineární pole n procesorů
- *Procesory:* obsahují jediný registr s hodnotou prvku
- *Algoritmus:* na počátku se pole naplní posloupností; v lichém kroku pracují liché procesory, v sudém sudé; porovná se s následníkem a případně prohodí hodnoty; algoritmus končí po n krocích (lze urychlit testem na prohození)

- *Složitost:* $t(n) = O(n)$ – nejrychlejší řešení pro lineární topologii; $c(n) = O(n^2)$ – není ideální

Odd-even merge sort

- *Princip:* síť složená ze speciálních procesorů
- *Topologie:* procesory propojeny tak, aby složením jednotlivých porovnání byla seřazená posloupnost
- *Procesory:* 2 vstupy a 2 výstupy, porovná vstupy a dá na výstupy `high` a `low`
- *Algoritmus:* spočívá v zapojení sítě, kaskáda 1×1, 2×2, $4 \times 4, \ldots$
- *Složitost:* $t(n) = O(log^2(n))$; $c(n) = O(n.log^4(n))$ – není optimální

Merge-splitting sort

- *Princip:* varianta odd-even sortu, každý procesor řadí krátkou posloupnost
- *Topologie:* lineární pole procesorů – $p(n) < n$
- *Procesory:* obsahuje m prvků, které umí seřadit optimálním sekvenčním algoritmem
- *Algoritmus:* místo porovnání sousedů se provede spojení posloupností ($O(n)$) a pak rozdělení na půl
- *Složitost:* $c(n) = O(n.log(n)) + O(n.p)$, optimální pro $p \leq log(n)$

Pipeline merge sort

- *Princip:* rozděleno na několik kroků, první spojuje posloupnosti délky 1, pak 2, atd.
- *Topologie:* linární pole procesorů – $p(n) = log(n) + 1$
- *Procesory:* umí spojovat dvě seřazené posloupnosti $O(n)$
- *Algoritmus:* ze vstupní posloupnosti se vezme první prvek a dá jej do jedné posloupnosti, druhý do druhé; další vybere vždy největší prvek a první dva dává do první posloupnosti, druhé dva do druhé; třetí krok také bere největší, ale střídá posloupnosti po čtyřech, atd.
- *Složitost:* $t(n) = O(n)$; $c(n) = O(n).O(log(n) + 1) = O(n.log(n))$ – optimální

Enumeration sort podruhé

- *Princip:* porovnání se všemi prvky a počet menších určuje pořadí
- *Topologie:* lineární pole n procesorů a sběrnice, která muže přenést jednu hodnotu
- *Procesory:* registr C (počet menších), X (prvek na dané pozici), Y (prvek posloupnosti, který se porovnává) a Z (výsledná pozice)
- *Algoritmus:* C se nastaví na 1; sběrnicí se pošle hodnota X a lineárně přes procesory Y prvnímu prvku; Y se posunou doprava a v dalším prvku se pošle X druhému a Y zase lineárně prvnímu; neprázdné registry se porovnají a případně inkrementuje C; po vyčerpání vstupu se X pošle do Z procesoru, který je určen C a to sběrnicí (protože Y se stále posouvá doprava)
- *Složitost:* $t(n) = n$; $c(n) = O(n^2)$ – není optimální

Minimum extraction sort

- *Princip:* stromem odebírá vždy nejmenší prvek
- *Topologie:* strom s n listy, $log(n) + 1$ úrovněmi a $2n - 1$ procesory

- *Procesory:* listový procesor obsahuje prvek posloupnosti, nelistové prvky umí porovnat syny
- *Algoritmus:* naplní se listy; v každém kroku otec vybere menší hodnotu; jakmile je v kořenu hodnota je to první prvek seřazené posloupnosti
- *Složitost:* $t(n) = O(n)$; $c(n) = O(n^2)$ – není optimální

Bucket sort

- *Princip:* stromem spojené procesory, které řadí menší posloupnosti a pak spojení
- *Topologie:* strom s m listy, kde $n = 2^m$
- *Procesory:* listové procesory řadí krátkou posloupnost, ostatní spojují syny – $O(n)$
- *Algoritmus:* –
- *Složitost:* $t(n) = O(n)$; $c(n) = O(n.log(n))$ – optimální

Median finding and splitting

- *Princip:* dělí posloupnost mediánem až na dvojice, které porovná
- *Topologie:* strom s m listy, kde $n = 2^m$
- *Procesory:* listové procesory porovnají dvojici, ostatní vyberou medián a rozdělí posloupnost – $O(n)$
- *Algoritmus:* je jasný, pro výběr mediánu je potřeba optimální, např. select
- *Složitost:* $t(n) = O(n)$; $c(n) = O(n.log(n))$ – optimální

5 Algoritmy vyhledávání

Vyhledávání zjišťuje přítomnost zadaného prvku v posloupnosti a případně i jeho pozici.

Optimální složitost podle sekvenčního algoritmu je $O(n)$ pro *neseřazenou* posloupnost – sekvenční vyhledávání; $O(log(n))$ pro *seřazenou* posloupnost – binární vyhledávání.

N-ary search

- *Seřazeno:* ano
- *Princip:* paralelní analogie k binárnímu hledání, zjišťuje se část, ve které prvek je
- *Topologie:* lineární pole m procesorů, kde $m < n$; CREW (hledaný prvek)
- *Procesory:* porovnávají prvek na svém místě s hledaným, registr, který říká, na které straně pokračovat
- *Algoritmus:* v každé iteraci se nastaví registry, v úseku, kde na obou stranách je hodnota odlišná se hledá v další iteraci
- *Složitost:* $t(n) = O(log_{m+1}(n+1))$; $c(n) = O(m.log_{m+1}(n+1))$ – není optimální

Unsorted search

- *Seřazeno:* ne
- *Princip:* paralelně volaný sekvenční algoritmus
- *Topologie:* lineární pole m procesorů, kde $m < n$
- *Procesory:* sekvenčně hledají prvek X s přidělenou posloupností

- *Algoritmus:* procesory načtou prvek do registru a provedou hledání, mohou nastavit flag nalezení
- *Složitost:* EREW – $c(n) = O(m.log(m) + n)$; CREW – $c(n) = O(n)$ (pokud zapisuje pouze jediný procesor)

Tree search

- *Seřazeno:* ne
- *Princip:* listy porovnají a výsledek se propaguje stromem
- *Topologie:* strom s $2n - 1$ procesory
- *Procesory:* listy umějí porovnat, otec logický OR vysledků synů
- *Algoritmus:* kořen načte hledaný prvek a stromem propaguje; listy obsahují prvky posloupnosti a porovnají; otec udělá OR
- *Složitost:* $t(n) = O(log(n))$; $c(n) = O(n.log(n))$ – není optimální

6 Maticové algoritmy

Složitost u maticových algoritmů nebývá založena na počtu prvků, ale na *počtu řádků/sloupců* n, také většinou uvažujeme čtvercové matice $n \times n$.

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

Mesh transpose

- *Princip:* prvky se posílají maticí na svá nová místa vždy v obou směrech
- *Topologie:* mřížka n krát n procesorů
- *Procesory:* 3 registry, A obsahuje výsledný prvek, B prvek od pravého/horního souseda, C od levého/dolního
- *Algoritmus:* vždy nejkrajnější prvky pošlou svou hodnotu sousedovi tak, aby dorazil na své místo (vždy pouze horizontálně a pak vertikálně
- *Složitost:* $t(n) = O(n)$; $c(n) = O(n^3)$ – není optimální

EREW transpose

- *Princip:* prvky se prohodí přímo mezi sebou paralelně
- *Topologie:* $(n^2 - n)/2$ procesorů (jeden na dva prvky a bez diagonály)
- *Procesory:* nejsou to ani procesory, pouze přehodí prvky
- *Algoritmus:* v jednom kroku se najednou paralelně přehodí potřebné prvky
- *Složitost:* $t(n) = 1$ – nejrychlejší; $c(n) = O(n^2)$ – optimální

Násobení matic $A(m, n)$ a $B(n, k)$ dává matici $C(m, k)$, kde $C_{ij} = \sum_{l=1}^{n} a_{il}.b_{lj}$.

Složitost podle sekvenčního algoritmu je $O(n^3)$, ale optimální není znám, pohybuje se někde mezi $O(n^2)$ a $O(n^3)$.

Obecné řešení paralelizuje pouze výslednou matici, ale výpočet sumy ponechává sekvenční.

Mesh multiplication

- *Topologie:* mřížka n krát k procesorů
- *Procesory:* udržují průběžnou hodnotu výsledného prvku, na konci výsledek
- *Algoritmus:* prvky matic se přivádějí do prvního řádku resp. prvního sloupce a posléze se posílají mezi procesory dále
- *Složitost:* $t(n) = O(n)$; $c(n) = O(n^3)$ – není optimální

Násobení matice vektorem je specializovaný případ násobení matic – $k = 1$.

Linear array multiplication je specializovaný případ mesh multiplication (vektor přichází shora, matice zboku, jeden sloupec procesorů). $t(n) = O(n)$; $c(n) = O(n^2)$ – optimální

Tree MV multiplication

- *Topologie:* binární strom $2n - 1$ procesorů
- *Procesory:* listové násobí, nelistové sčítají
- *Algoritmus:* v listech je vektor, matice se přivádí po řádcích; výsledky násobení řádku se sčítají otcem až do kořene
- *Složitost:* $t(n) = O(n)$; $c(n) = O(n^2)$ – optimální

7 Model PRAM

PRAM (Parallel Random Access Machine) je synchronní model paralelního výpočtu pomocí procesorů se sdílenou pamětí a *společným programem*. Alternativa k paralelnímu Turingovu stroji.

Procesor:

- Aditivní a logické operace
- Multiplikativní operace
- Podmíněné skoky
- Přístup ke svému unikátnímu číslu (index)

Paměť:

- Náhodný přístup pro všechny procesory
- Reprezentovaná neomezeným počtem registrů
- Neomezená délka slova (dnes není vyžadováno)
- Módy přístupu EREW, CREW a CRCW

Výpočet probíhá synchronně po krocích – čtení, lokální operace, zápis.

Řešení CRCW konfliktů:

1. *COMMON* – všechny hodnoty musejí být stejné, jinak se nezapíše
2. *ARBITRARY* – zapíše se libovolná z hodnot
3. *PRIORITY* – procesory mají přidělenu prioritu

Broadcast je algoritmus pro EREW PRAM pro distribuci hodnoty na jednom místě v paměti do všech procesorů. Šíření je logaritmické – jeden procesor přečte, předá dalšímu, pak jsou dva, každý jednomu, atd. $t(n) = O(log(n))$

8 Suma prefixů

Suma prefixů (all-prefix-sums, allsums, scan) je základním kamenem paralelních algoritmů. Je to operace, jejíž vstupem je *uspořádaná* posloupnost a *binární asociativní* operace $\oplus$. Výsledkem je posloupnost $a_0, (a_0 \oplus a_1), (a_0 \oplus a_1 \oplus a_2), \ldots$.

Použití: vyhodnocování polynomů, rychlé sčítání v HW, lexikální analýza a porovnávání, řazení, hledání regulárních výrazů, odstranění označených prvků apod.

Sekvenční řešení prochází všechny prvky a nese si mezivýsledek, složitost $O(n)$.

Varianty:

- *Scan* je normální suma prefixů.
- *Prescan* je rozšířena o neutrální prvek na počátku výsledku a bez posledního prvku.
- *Reduce* je pouze hodnota posledního prvku scan.
- *Segmentovaný scan* je doplněn o pole příznaků, který určuje hranici segmentu, každý segment je scanován zvlášť.

Paralelní reduce je možno spočítat pomocí stromu procesorů, výsledek je v kořeni. $t(n) = O(log(n))$; $c(n) = O(n.log(n))$ – není optimální, lze vylepšit tak, že se paralelně počítá reduce menších posloupností

Algoritmy scan není nutné přímo uvádět, jelikož se jedná o prescan s použitím reduce.

Prescan se skládá ze dvou částí – *upsweep* a *downsweep*. Složitost je stejná jako u reduce.

Upsweep je totožný s reduce, ale mezivýsledky se nezahazují.

Downsweep:

1. Kořenu se přiřadí neutrální prvek.
2. Kořen přiřadí pravému synovi svou hodnotu $\oplus$ hodnotu levého syna.
3. Kořen přiřadí levému synovi svou hodnotu.
4. Opakuje se pro další úrovně.

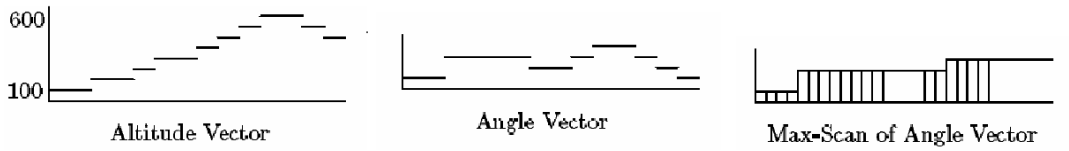
Packing problem bere vstupní pole, kde pouze některé hodnoty jsou důležité a tyto shrne na počátek pole.

1. Potřebné položky se označí flagem.
2. Spočte se +-scan flagů.
3. Každý prvek s hodnotou vyšší než levý soused se přesune na pozici danou touto hodnotou.

Problém viditelnosti – je dána výšková mapa (matice) a na ní pozorovací bod X. Je potřeba zjistit, které body jsou viditelné z bodu X.

1. Bod je viditelný, pokud žádný bod mezi ním a pozorovatelem nemá větší vertikální úhel.
2. Vytvoří se vektor výšek bodů na cestě paprsku.
3. Vektor výšek se přepočítá na vektor úhlů.
4. Pomocí max-prescan se spočte vektor maximálních úhlů.

5. Určí se úhel studovaného bodu a porovná s maximem.



Binární sčítačka – Carry Lookahead Parallel Binary Adder. U normální sčítačky paralelizaci vadí závislost na carry ze sčítání nižších bitů. Je potřeba carry předpočítat.

1. Vypočte se vektor s oborem hodnot "Propagate, Stop, Generate" podle tabulky.
2. Vypočte se scan nad vektorem opět pomocí tabulky.
3. Hodnota Generate říká, že na vyšším bitu bude potřeba uplatnit carry.
4. Carry je tak vypočteno logaritmicky, pak je možné sečíst v konstantním čase.

Radix sort je založen na dvojkovém radix sort, tedy prvky s bitem 0 se přemístí na počátek. To se provede pro všechny řády.

Operace split je operace přesunutí prvků na počátek, implementována jako scan a prescan, složitost jako scan.

Quicksort používá segmentovaný scan, segmenty jsou 3 – menší, rovné, větší než pivot. *Pivot* se nepočítá, většinou se bere náhodný/první prvek.

1. Kontroluje se, zda již není seřazeno (test mezi sousedy a AND-reduce výsledků).
2. Vybere se pivot každého segmentu, opět scan. Na počátku segment jeden.
3. V každém segmentu porovnej prvky s pivotem a rozděl.
4. Modifikovaný split pro přeřazení do segmentu.
5. Rekurzivně na segmenty quicksort.
6. Složitost $t(n) = O(n/m.log(n) + log(m).log(n))$; $c(n) = O(n.log(m) + m.log(n).log(m))$ – optimální pro malé m.

9 Seznamy

Lineární seznam je modelován jako pole (možno přistoupit indexem) prvků v paměti, které obsahují hodnotu a index následníka. Poslední prvek ukazuje sám na sebe.

Predecessor computing počítá index předchůdce v *konstantním čase* (`Pred[Succ[i]] = i`).

List ranking přiřazuje prvkům jejich vzdálenost od konce. Sekvenční složitost je $O(n)$. V paralelním prostředí se používá technika *path doubling*.

Path doubling:

1. Paralelně se všem prvkům přiřadí RANK (0 pro poslední prvek, jinak 1).
2. Každý procesor prvku v $log(n)$ krocích se počítá RANK jako `RANK[i] + RANK[Succ[i]]` a posune ukazatel `Succ[i] = Succ[Succ[i]]`.

3. $t(n) = O(log(n))$; $c(n) = O(n.log(n))$ – není optimální

Suma suffixů je obdoba sumy prefixů, ale na seznamech (kde pevný bod je konec). Počítá se stejně jako list ranking, ale použije se zadaná operace $\oplus$.

Vylepšení list rankingu (a sumy suffixů) spočívá ve snížení ceny, některé procesory totiž provádějí zbytečnou práci (počítají již spočítané věci nebo cyklí na konci). Řešením je odpojovat procesory a tím snížit cenu. Nejprve každý procesor dostane vzdálenost 1, pak pracuje každý druhý a zvýší ji na 2 pak každý čtvrtý a opět zvýší. V rekonstrukční fázi se pouze přičítají mezivýsledky.

Problém v paralelismu – jak se určí "každý druhý" ?

- *Random mating* – náhodně si vybere pohlaví, female následovaný male se odpojí
- *Optimal list ranking* – každý procesor má zásobník prvků, simulace random mating

List coloring je obarvení seznamu tak, aby sousedé neměli stejnou barvu. k-obarvení může použít k různých barev.

$2log(n)$ **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.

10 Stromy

Stromy jsou prezentovány podobně jako seznamy, ale vazba není na následníka $(i+1)$, nýbrž na syny ($2i$ a $2i+1$).

Eulerova cesta je obecný případ průchodu stromem (linearizace). Převádí strom na orientovaný graf (nahradí hranu dvěma opačnými) → Eulerova kružnice.

Eulerova kružnice je reprezentována funkcí `etour(e)`, která hraně přiřadí následující hranu v kružnici. Funkce je uložena jako seznam sousednosti.

Kořen stromu vznikne tak, že se v jednom bodě (kořeni) Eulerova kružnice přeruší.

Pozice uzlu je vypočtena jako $2n - 2 - Rank(e)$, kde $Rank$ je výsledek list rankingu – $O(log(n))$. Využívá se pro zjištění rodiče.

Tree contraction je operace použitá při výpočtu výrazů ve stromě. Eulerova cesta není použitelná. Každý list obsahuje operand a nelist operátor. Tree contraction strom postupně zmenšuje až do jediného uzlu, tedy výsledku.

RAKE operation odstraní daný uzel a otce, na jehož místo dosadí sourozence s jeho podstromem. Využívá se pro tree contraction.

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

1. Označíme listy zleva doprava.

2. Krajní se vyřadí (zůstanou poslední).
3. Nejprve se volá RAKE na liché a pak na sudé listy.
4. Redukce stromu za $t(n) = O(log(n))$.

11 Interakce mezi procesy

Interakce mezi procesy můžeme rozdělit na kooperaci (je potřeba synchronizace) a soupeření (je potřeba výlučný přístup).

Problém: jelikož procesy běží paralelně (popř. pseudoparalelně), není jasné pořadí vykonávání instrukcí obou programů.

Zde odkazuji na podklady předmětu POS, tomuto tématu se tam věnuje několik kapittol, stačí si odmyslet části spojené s OS.

Operátor `<await B->S>` je původně pouze teoretický operátor, implementovaný ve vyšších programovacích jazycích pro paralelní programování. Jeho implementace je problémová. Zajišťuje atomičnost `<>`, očekává splnění podmínky `B` a poté atomicky vykoná sekvenci příkazů `S`.

Critical Region ve vyšších jazycích jsou obalení použití semaforů pro přístup ke KS, klíčové slovo `region` a označení proměnných `shared`. Problém, pokud se zanořují, pak může dojít ke konfliktu. Jednoduchá implementace (téměř makro).

Conditional Critical Region rozšiřuje koncepci o podmínku, pokud je stanovena dobře, ke kolizi nedojde. Implementace je ovšem velmi složitá.

Základní algoritmy (chybné) by chtít neměli, pokročilé (bez chyby) se učit nebudu.

12 Předávání zpráv

Zasílání zpráv je založeno na dvou operacích – `send()` a `receive()`. Zprávy se posílají tzv. *kanálem.*

Asynchronní kanál neblokuje odesílání, data bufferuje v sobě, těžko se implementuje (buffer je problém).

Synchronní kanál blokuje (hned nebo pokud je naplněn buffer), buffer může být, ale je omezen, většinou bez bufferu, jde simulovat asynchronním kanálem (použití ACK).

OCCAM je programovací jazyk založený na CSP (Communicating Sequential Processes).

- Základem je proces, operace přiřazení `:=`, vstup `?` a výstup `!`.
- Sekvenční příkazy v oddílu `SEQ`.
- Paralelně prováděné příkazy v oddílu `PAR`.
- Podmínka na základě vstupního kanálu `ALT`.

ADA je imperativní objektový jazyk s validačními prostředky. Má silné prostředky pro zasílání zpráv.

- Konstrukce `accept` očekává příchod zprávy daného jména a parametrů, pak zpracuje úkol a odešle výsledek.

- Konstrukce `select` očekává více typů zpráv a pak vybere ošetřující kód.

Linda je paralelní programovací jazyk založený na C. Je založena na asynchroním zasílání zpráv přes globální prostor zpráv (global space).

- *Tuple Space* (nástěnka) – globální prostor zpráv (sdílená paměť).
- *Tuple* – n-tice obsahující pole s daty (zpráva).
- *Actual field* – pole zprávy, které je vyhodnoceno před odesláním
- *Formal field* – pole zprávy, které je předáno jako proměnná
- Vkládání na TS provádějí `out` a `eval`. Přijímání pak `in` a `rd`.
- `out` – generuje data (passive tuple), data vyhodnocena sekvenčně
- `eval` – generuje procesy (active tuple), které vypočítají data paralelně
- `in` – přijme a odebere z TS, blokující
- `rd` – pouze přečte z TS, blokující
- Požadavky na čtení zpráv jsou ve tvaru šablon (`("x",?y,?z)`).

13 Jazyky pro paralelní zpracování

Situace: obrovské sítě počítačů propojené rychlým spojením, je možné používat distribuované výpočty, potřeba jazyka a protokolu, nejpoužívanější jsou PVM a MPI

PVM (Parallel Virtual Machine)

- Vytvořen jednou skupinou.
- Distribuovaný operační systém.
- Přenosný mezi HW.
- Heterogenní (různé možnosti reprezentace dat).
- Velká odolnost proti chybám.
- Dynamický (přidat, odebrat proces, vyrovnání zátěže, chyby).

MPI (Message Passing Interface)

- Vytvořen fórem firem.
- Knihovna poskytující funkce.
- Přenosný mezi HW a SW (je to knihovna).
- Heterogenní (zabalení různých dat do daných typů).
- Zaměřen na vysoký výkon.
- Přesně definované chování.
- Statický (vyšší výkon).
- Není odolnost proti chybám (neurčitý výsledek).

Implementace PVM

- Démon, bežící na stanicích.
- Démoni spolu komunikují.
- Spojení démonů tvoří virtální paralelní stroj.
- Démon má pod sebou procesy, kterým je nadřazen.
- První démon je označen jako master.
- Master se stará o nastavení, přidávání, hlídání.

Implementace MPI

- Na každém počítači běží procesy (jeden na CPU).
- Procesy mají identifikaci.
- Procesy znají ID ostatních procesů.
- Procesy komunikují mezi sebou přímo.
- Proces neumí fork() (MPIv1)

Message Passing v MPI

- Kooperativní – explicitní `send()` a `recv()`.
- Jednostranné (MPIv2) – zápis do paměti cíle/čtení zdroje `Put()` a `Get()`.
- Procesy se sdružují do skupin a zprávy do kontextu.
- Komunikátor určuje kontext a skupinu (např. `MPI_COMM_WORLD`).
- Kolektivní operace – `bcast()` a `reduce()`.
- Neblokující operace – `Isend()` a `Irecv()`, pak `Iwait()`.
- `MPI_Barrier()` pro synchronizaci.