Distribuované a paralelní algoritmy

Z FITwiki

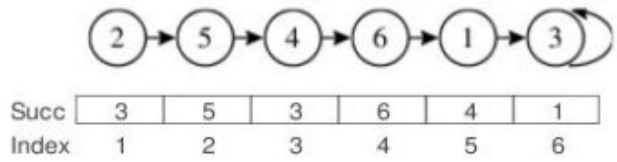
Seznamy, stromy, grafy

Seznamy

Lineární seznam

pole prvků v paměti (možnost přistoupit indexem), které obsahují hodnotu v_i a ukazatel na následníka $succ[i]$. Poslední prvek ukazuje sám na sebe.

Obsah



Predecessor computing

- počítá index předchůdce všech prvků
- t(n) = O(c)
- p(n) = n
- c(n) = O(n)

```
for i = 1 to n do in parallel
  Pred[Succ[i]] = i
```

Každý procesor vezme jeden prvek a následníkovi jeho prvku zapíše prvek jako předchůdce Pozn.: Je třeba zvlášť ošetřit první a poslední prvek seznamu

List ranking (sekvenční)

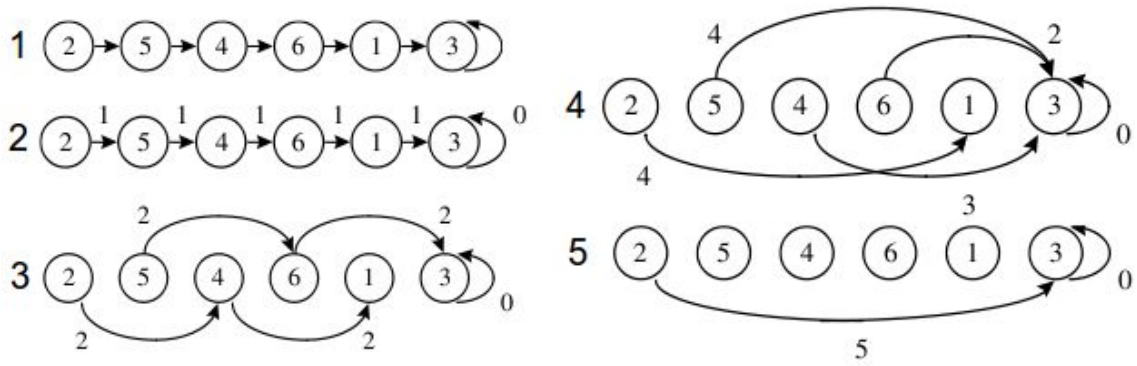
- přiřazuje prvkům rank = jejich vzdálenost od konce.
- Sekvenční časová složitost je $O(n)$
- V paralelním prostředí se používá technika **path doubling**.

List ranking (path doubling)

(Wyllie's algorithm)

```
Algorithm
Input: array Succ[1..n]
Output: array Rank[1..n]
for i=1 to n do in parallel
    if Succ[i]=i then Rank[i] = 0
                 else Rank[i] = 1
    for k = 1 to log n do
        Rank[i] = Rank[i] + Rank[Succ[i]]
        Succ[i] = Succ[Succ[i]]
    end for
end for
```

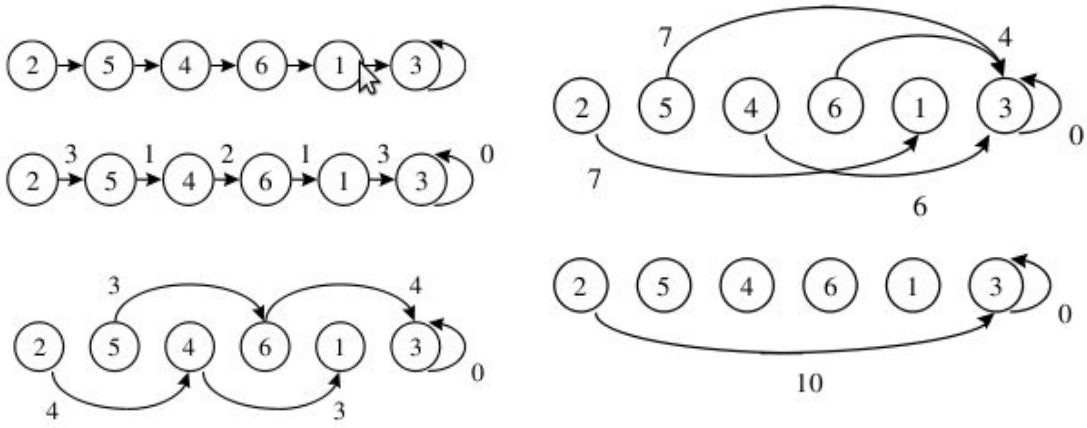
- 1) Paralelně se všem prvkům přiřadí RANK (0 pro poslední prvek, jinak 1).
- 2) Každý procesor prvku v $log(n)$ krocích
  - se počítá RANK jako `RANK[i] + RANK[Succ[i]]`
  - posune ukazatel `Succ[i] = Succ[Succ[i]]`



- t(n) = O(log(n))
- p(n) = n
- c(n)=O(n.log(n)) (není optimální)

Paralení suma suffixů

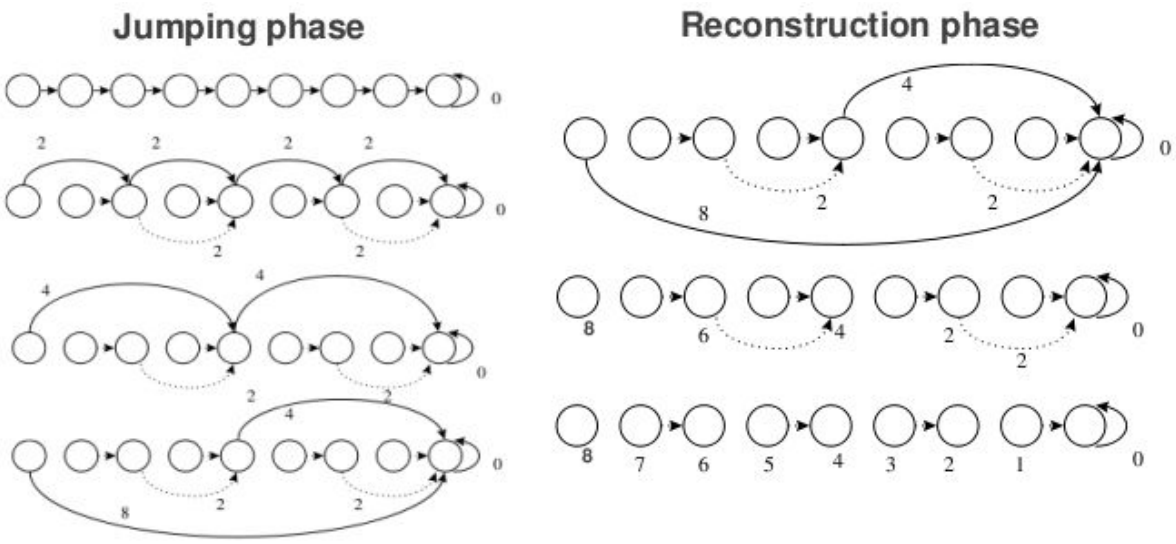
- je obdoba sumy prefixů, ale na seznamech (kde pevný bod je konec)
- Počítá se stejně jako list ranking, ale použije se zadaná operace $\oplus$



- t(n) = O(log n)
- p(n) = n
- c(n) = O(n.log n)

Optimalizace path doublig 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 (v každém kole odpojena polovina prcesorů)



- Postup:
  - Jumping phase
    - Nejprve každý procesor dostane vzdálenost 1,
    - pak pracuje každý druhý a zvýší ji na 2
    - pak každý čtvrtý a opět zvýší.
  - Recosntruction phase
    - přičítají mezivýsledky.

Problém v paralelismu
jak se určí "každý druhý" ?

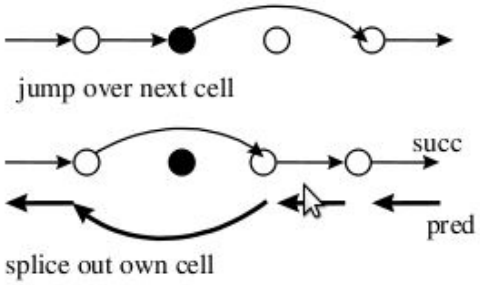
Random mating

- každý proces si náhodně vybere pohlaví
- každý female následovaný male se přeskočí a procesor se odpojí
- cena je stále neoptimální, ale množství práce je c(n) = O(n) (optimální)

Optimal list ranking

- simulace random mating
- pevný počet stále pracujících procesorů (n/log n procesorů, každý obsluhuje log n prvků)
- každý procesor má zásobník prvků, které má zpracovat
- prvky mají náhodně přiřazeno pohlaví
- v každém kroku se všechny procesory pokusí provést jump-over (přeskočení následníka prvku na vrcholu zásobníku)
  - (přeskakuje se female jehož následník je male)

- algoritmus může být nevyvážený
- řešení se nahrazením operace jump-over za splice-out (vypletení)



List coloring

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 indexu, ve kterém se sousedé liší, barva je pak $C = 2k + ID[k]$

Ruling set

k-ruling set
: množina nesousedících vrcholů, mezera mezi nimiž je široká maximálně k

$2k$-ruling set z k-coloring

: vybere prvek do ruling set tehdy, když jeho barva je nižší než barva předchůdce a následníka (hledáme lokální minima)

Stromy

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

Úroveň vrcholu
: počet hran mezi uzlem a kořenem

Eulerova cesta

Eulerova cesta

- je obecný případ průchodu stromem (linearizace)
- průchod stromem ve kterém se každá hrana projde právě jednou (jednou tam a jednou zpátky)

Eulerova kružnice

- strom se převede na orientovaný graf (každá hrana se nahradí za dvě orientované hrany v opačných směrech)
- je reprezentována funkcí `etour(e)`, která hraně přiřadí následující hranu v kružnici.
- Umožňuje projít všechny uzly grafu bez opakování hran na cestě.

Seznam sousednosti (adjacency list)

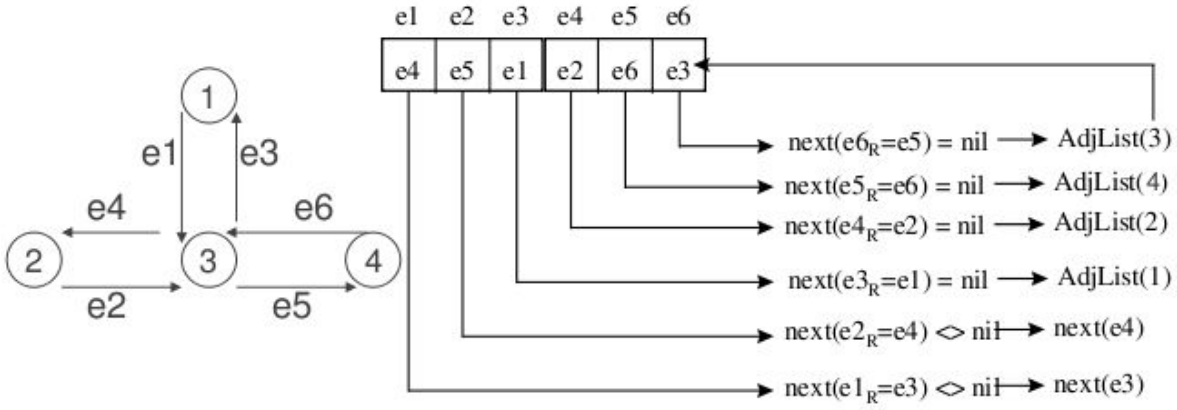
- slouží k reprezentaci stromu v podobě grafu pro eulerovu kružnici
- pro každý uzel je přiřazen lineárně vázaný seznam, každý prvek seznamu je dvojice hrana/opačná hrana (e, e_R)

Pozice uzlu

je vypočtena jako $2n - 2 - Rank(e)$

$Rank(e)$ je výsledek list rankingu - $O(log(n))$. Využívá se pro zjištění rodiče.

Eulerova kružnice




```
for i = 1 to 2n-2 do in parallel     { ei = (u, v) }
  if next(eR) <> nil  then Etour(e) = next(eR)
    else Etour(e) = AdjList(v) { first item of adj. list
  endif
endfor
```

Kořen stromu

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

Pozice hran

- vezmeme grav v podobě seznamu souslednosti
- převedeme na Eulerovu kružnici
- provedeme List ranking na kružnici (rank = opačné pořadí hran v kružnici)
- Pořadí získáme paralelním vypočtením 2n-2-rank (n je počet vrcholů)
- t(n) = O(log n) (nejhorší závislost má suma suffixů, ostatní jsou lineární)

Nalezení rodičů

Dopředná hrana

position(e) < position (e_R)

tj. hrana na které nejdřív jdeme dopředu a pak až zpátky

Zpětná hrana

position(e) > position (e_R)

tj. hrana na které nejdřív jdeme dopředu a pak až zpátky

Pokud (u, v) je dopředná hrana pak u je rodičem v

```
for each edge e = (u, v) do in parallel
  if posn(e) < posn(eR) then
    parent(v) := u;
  endif
  parent(root) := nil;
endfor
```

Preorder

(Preorder – navštiv nejdřív otce, pak oba syny)

- Pořadí preorder vrcholu ve stromě je 1+počet dopředných hran, kterými jsme prošli po cestě z kořene k vrcholu

```
1) for each e do in parallel
     if e is forward edge then weight = 1
              else weight = 0
     endif
2) weight = SuffixSums(Etour, Weight)
3) for each e do in parallel
     if e=(u, v) is forward edge then
       preorder(v) = n - weight(e) + 1
     endif
   preorder(root) = 1
```

Počet následníků vrcholu

- Počet dopředných hran v segmentu Eulerovy cesty, počínajícím i končícím ve vrcholu

Úroveň vrcholu

- rozdíl dopředných a zpětných hran na cestě z kořene k vrcholu

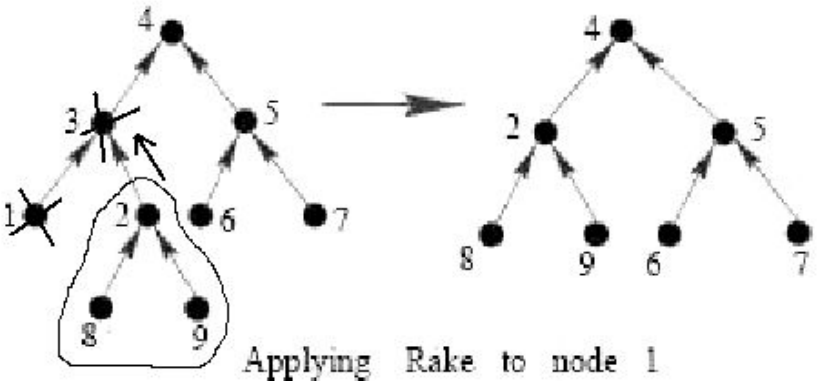
Tree contraction

Tree contraction

- používá se 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

- rake na **listový** uzel u
  - odstraníme uzlel u a jeho rodiče
  - druhý potomek rodiče u se připojí na místo, kde byl rodič u



Algoritmus tree contraction

- opakovaně aplikujeme RAKE a tím zmenšujeme strom
- snažíme se aplikovat pro co nejvíce listů paralelně
- nelze aplikovat operaci RAKE na vrcholy jež ve stromu sousedí (mají spol rodiče)
- Jak určit uzly na které lze aplikovat?

- Algoritmus:
  - Označíme listy jejich pořadím zleva doprava (pořadí na Eulerově cestě)
    - Každé hraně (v, p(v)), kde v je listem, přiřadíme váhu 1
    - Vyřadíme nejlevější a nejpravější list. (Tyto listy budou dva synové kořene až se podaří strom zmenšit na strom se třemi vrcholy)
    - Nad výsledným seznamem provedeme sumu suffixů a získáme listy, očíslované zleva doprava
  - Uložíme všech n listů do pole A
    - Aodd obsahuje prvky pole A s lichými indexy
    - Aeven obsahuje prvky pole A se sudými indexy
  - for i:=1 to log(n+1) do
    - RAKE na všechny Aodd, které jsou levými potomky
    - RAKE na všechny Aodd, které zbyly
    - A := Aeven

- počet listů se v každém kole změnší na polovinu
- t(n) = O(log n)

Algoritmus Tree search

- Vyhledávání v neseřazené posloupnosti
- Stromová architektura s 2n-1 procesory

- Algoritmus
  - 1. Kořen načte hledanou hodnotu x a předá ji synům ... až se dostane ke všem listům
  - 2. Listy obsahují seznam prvků, ve kterých se vyhledává (každý list jeden). Všechny listy paralelně porovnávají x a xi, výsledek je 0 nebo 1.
  - 3. Hodnoty všech listů se předají kořenu - každý ne list spočte logické or svých synů a výsledek zašle otci.

Kořen dostane 0 - nenalezeno, 1- nalezeno

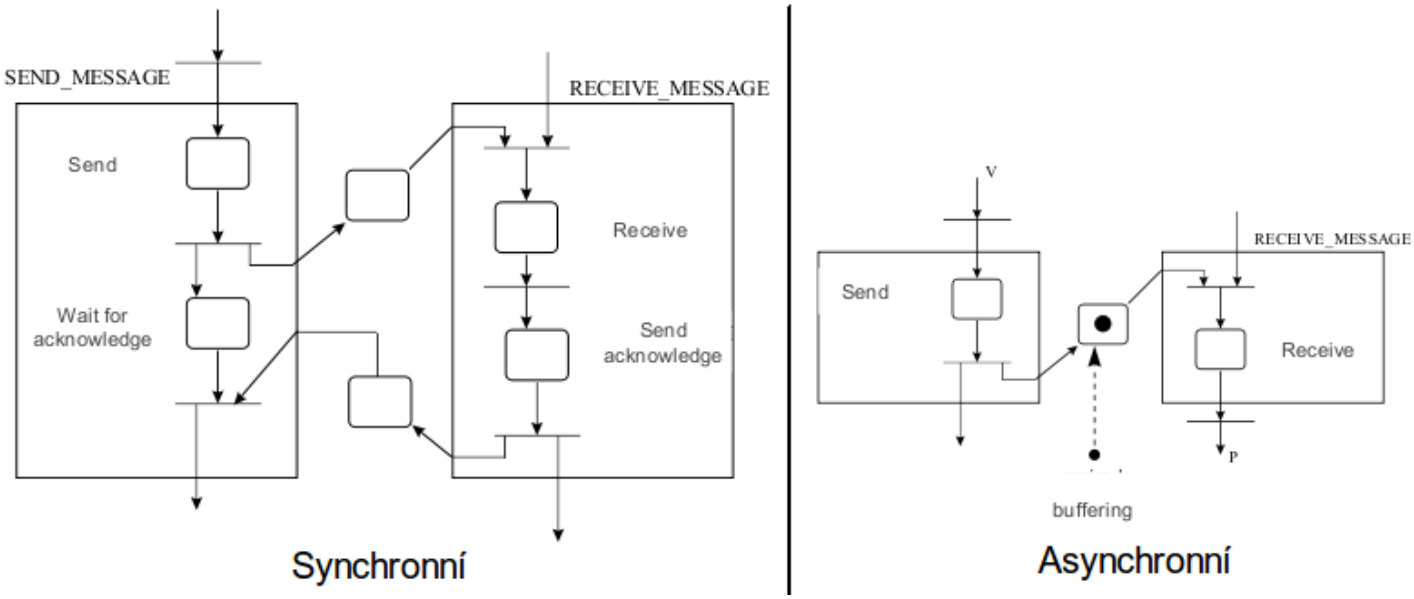
- Analýza
  - Krok (1) má složitost O(log n), krok (2) má konstantní složitost, krok (3) má O(log n).
  - t(n) = O(log n)
  - p(n) = 2.n-1

- $c(n) = t(n).p(n) = O(n.\log n) \rightarrow$ což není optimální

obrazek viz PRL H005- str.8

Předávání zpráv a knihovny pro paralelní zpracování (MPI)

Asynchroní/synchronní komunikace



Paralení výpočty

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 (může se zotavit při výpadku některých stanic)
- Dynamický (přidat/odebrat proces, přidat odebrat stanice, vyrovnání zátěže, chyby).

Implementace PVM

- Démon, bežící na stanicích.
- Démoni spolu komunikují.
- Spojení démonů tvoří jeden výkonný virtuální počítač.
- Démon má pod sebou procesy, kterým je nadřazen.

- První démon je označen jako master.
- Master se stará o nastavení, přidávání, hlídání.
- Výpadek mastera způsobí problémy

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 (pevný počet stanic, pevný počet úkolů, vyšší výkon).
  - MPIv2 zavádí možnost začínat a ukončovat skupiny úkolů
- Není odolnost proti chybám (pokud vypadne jedna stanice, je nutný výsledky zahodit).

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)
-

Kooperativní komunikace

- `send()` a `recv()`
- odesílatel i příjemce se přímo podílí na komunikaci

Jendostranná komunikace (One-sided)

- `Put()` a `Get()`
- pouze jeden proces se přímo podílí na komunikaci
- zápis/čtení z paměti druhého procesu bez jeho účasti
- součást MPIv2

Informace o prostředí

- kolik je dostupných procesorů?
- který procesor jsem já?

Základní koncepty MPI

- procesy mohou být seskupenu do skupin (groups)
- zprávy se vždy předávají a přijímají v daném kontextu (existuje výchozí kontext obsahující všechny procesy, MPI_COMM_WORLD)
- skupina a kontext = komunikátor
- proces je identifikován jeho rankem (jednoznačné ID) ve skupině příslušející komunikátoru
- zprávy mohou být označeny tagy, které pomáhají příjemci identifikovat zprávu (někdy nazýváno message types)
- podporuje blokující i neblokující zasílání zpráv
- podporují kolektivní operace (broadcast, reduce)

Datové typy

- data ve zprávách reprezentovány jako trojice (address, count, datatype)
- umožňují předávat data mezi platformami používajícími různé reprezentace dat
- datové typy
  - základní typy (MPI_INT, MPI_DOUBLE)
  - contiguous array (co to je??)
  - strided block (co to je??)
  - indexovaná pole
  - struktury

Synchronizace
: využívá se bariér (procesy čekají dokud všechny nedojdou k bariéře)

Kolektivní operace

Broadcast
: data poslána z jednoho procesu všem ostatním

Reduce
: data získána ze všech procesů, zkombinována a předána jednomu procesu

Scatter
: data z jednoho procesu rozdělena na části mezi všechny procesů

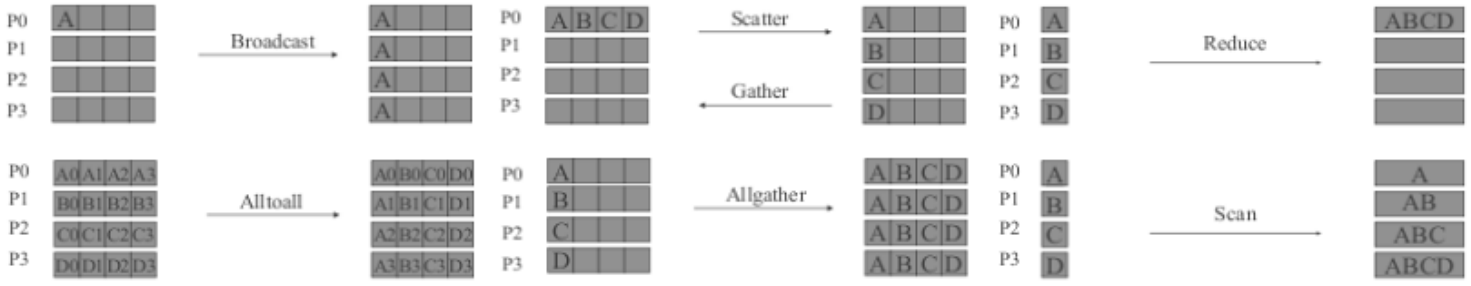
Gather
: části dat z různých procesů spojeny do jednoho procesu

Allgather
: části dat z různých procesů spojeny a předány všem procesům (gather+broadcast)

AllToAll
: data ze všech procesů rozdělena na části, příslušené řásti spojeny a předány konkrétním procesům (1 proc. první části, druhý drudé části, ...)

Scan
: každý proc obdrží zkombinovaná data předchozích procesů (a=a, b=a+b, c=a+b+c, ...)

Agregační funkce
: min, max, average, sum, and, or, xor, ...



Knihovny/jazyky

Někdo to pls zkontrolujte za správnost neručím je to vyčteno jen ze slides

OCCAM

- jazyka založený na formalismu Communicating Sequential Processes (CSP)
- imperativní procedurální jazyk
- výpočet reprezentován sítí komunikujících procesů

Základní prvky jazyka (procesy)

- přiřazení

```
x := y + 2
```

- vstup

```
keyboard ? char
```

- výstup

```
screen ! char
```

Komunikační kanály
    jeden proces do něj zapisuje ostatní čtou

Sekvence SEQ

- jednotlivé procesy jsou provedeny sekvenčně

```
SEQ
 proces
 proces
```

Paralelní spuštění PAR

- podprocesy jsou spuštěny paralelně
- proměnné zapisované v některé větvi nemohou být v ostatních větvích čteny/zapisovány

```
PAR
 proces
 proces
```

Replicated PAR/SEQ

- spuštění procesu v PAR/SEQ v několika exemplařích

```
PAR i = 0 FOR 4
  proces
```

Výběr ALT

- provede se pouze jeden z podprocesů (podle podmínky - guard)

```
ALT
 left ? packet -- guard
    proces 1
 right ? packet —- guard
    proces2
```

ADA

- používá mechanismus **randevousz**
- **Task** - oddělený sekvenční proces s lokálními daty
- tasky komunikují voláním vstupních procedur jiných tasků (volání i přijetí volání může čekat dokud není druhý připraven)

Accept

- definuje vstupní proceduru
- vnitřek procedury je spouštěn pouze při přijetí volání vstupní procedury
- parametry volání jsou předány do procedury

```
accept <entry-name> (<formal parameter list>) do
 ...
end;
```

Select

- čekání na accept více různých signálů
- podmíněná dostupnost vstupních procedur

```
select
  when <podmínka>
    accept entry1...
  or when <podmínka>
    accept entry2...
  else
```

end select;

Linda

- založeno na jazyce C
- **Tuple space** virtuální sdílená paměť, nástěnka na kterou procesy vyvěšují data a na které data hledají
- **tuple** - datová položka vkládaná na nástěnku, skládající se ze sekvence hodnot a formálních polí (slouží k vyhledávání tuple)
- asynchronní komunikace mezi procesy (proces vyvěsí data, jiný proc si data někdy později vezme)

out

- parametry jsou vyhodnoceny a výsledek je poté uložen na nástěnku
- volající ihned pokračuje, neblokující
- příklad out ('array', dim1, dim2)

eval

- pro každý parametr spustí proces, ten vyhodnotí parametr
- tyto vyhodnocené parametry jsou pak složeny do výsledné n-tice, která je vložena na nástěnku

- volající ihned pokračuje, neblokující
- ALE nechá po sobě spuštěné procesy, které vyhodnocují parametry a nevíme za jak dlouho skončí
- příklad eval ("test", F(param1), G(param2))

in

- vyhledá tuple odpovídající šabloně
- tuple je načten a odstraněn z tuple space
- blokuje dokud příslušný tuple není načten (tj. může čekat na jeho vygenerování)
- použitelné k synchronizaci mezi procesy
- existuje i v neblokující podobě inp, která se ihned vrátí

rd

- vyhledá tuple odpovídající šabloně
- tuple ja načten, ale je ponechán v tuple space
- blokuje dokud příslušný tuple není načten (tj. může čekat na jeho vygenerování)
- použitelné k synchronizaci mezi procesy
- existuje i v neblokující podobě rdp, která se ihned vrátí

Distribuovaný broadcast, synchronizace v distribuovaných systémech

Distribuovaný broadcast

- pojmy:
  - m - zpráva z množiny možných zpráv, každá zpráva zahrnuje odesílatele (sender(m)) a sekvenční číslo (seq(m))
  - funkce bcast(m) a deliver(m)
  - např. sender(m)=p a seq(m)=i znamená, že m je i-tá zpráva poslaná procesem p

Základní vlastnosti broadcastu

- Validity = Pokud korektní proces odešle broadcast, pak všechny korektní procesy tento broadcast obdrží (dříve či později)
- Agreement = Pokud korektní proces obdržel broadcast, potom všechny korektní procesy obdrží broadcast také (dříve či později)
- Integrity = Každý korektní procs obdrží broadcast maximálně jednou (nedojde k zacyklení)

Pokud platí všechny 3 vlastnosti, jde o **spolehlivý broadcast**.

Další vlastnosti broadcastu

- FIFO Order = pokud nějaký proces broadcastuje zprávu m před zprávou n, potom žádný korektní proces nepřijmne zprávu n, aniž by nejdříve přijal zprávu m (tj. zprávy jsou doručovány ve stejném pořadí jak byly odeslány)
- Causal Order = FIFO Order, ale v rámci všech procesů v systému (tj. Pokud máme 2 procesy a každý vysílá 2 broadcasty A (m,n), B(o,p) pak žádný proces nepřijme v pořadí (m,n,p,o) nebo (o,p,n,m) nebo (p,n,m,o)... Jediné správné je (m,n,o,p) nebo (o,p,m,n) nebo (m,o,p,n) nebo (m,o,n,p))
- Total Order = doručování přesně ve stejném pořadí, jak byly broadcasty odeslány

Klasifikace broadcastů

- Reliable (RBCAST) = Validity + Agreement + Integrity
- FIFO (FBCAST) = reliable + FIFO Order
- Causal (CBCAST) = reliable + Causal Order
- Atomic (ABCAST) = reliable + Total Order

Algoritmus pro spolehlivý broadcast (RBCAST)

```
Odesílatel o:
bcast(R,m): //R = reliable bcast, m = zprava
 pridej do m odesilatele(m) a sekvencni_cislo(m);
 send(m) vsem sousedum, vcetne sebe;
```

```
Prijemce p:
deliver(R,m):
 pri receive(m) do
  if p jeste neprijal tento broadcast then
   if sender(m) != p then send(m) vsem sousedum;
   deliver(R,m)
  endif;
 enddo;
```

Synchronizace v distribuovaných systémech

V distribuovaných systémech není údaj o globálním čase (i kdyby byl, nejsme schopní kontrolovat zpoždění mezi doručením zpráv). Tedy každý uzel má vlastní hodiny. Řešíme problém vzájemného vyloučení pří požadavku více klientů v distribuovaném prostředí na sdílený zdroj. 2 možnosti:

- Lamportův algoritmus + optimalizace Ricart-Agrawala algoritmus (založeno na časových razítkách)
- Raymondův algoritmus (založen na tokenech)

Lamportův algoritmus

Lamportův algoritmus pro přístup do kritické sekce používá Lamportovy logické hodiny.

Lamportovy logické hodiny

- Není zde žádná souvislost s fyzickými hodinami, ve skutečnosti jde o časová razítka.
- Založeno na relaci mezi dvěma událostmi stalo_se_dříve(a,b).
- Vychází z pravidel fyzické kauzality (než něco přijmeme, musíme to odeslat)

Synchronizace hodin mezi 2 procesy probíhá takto:

1. Každý proces má vlastní čítač běžící různou rychlostí
2. Proces 1 chce odeslat zprávu procesu 2, společně se zprávou vloží hodnotu svého čítače
3. Proces 2 přijme zprávu od procesu 1 a zjistí časové razítko
4. Proces 2 porovná hodnoto svého čítače s hodnotou extrahovanou z časového razítka procesu 1
5. Proces 2 si nastaví hodnotu svého čítače na vyšší z těchto dvou hodnot a inkrementuje hodnotu
6. V tuto chvíli jsou procesy 1 a 2 synchronizovány (ale jen na chvíli, protože jejich čítače běží obecně různě rychle)

Lamportov algoritmus pre pristup do kriticke sekcie

wiki (http://en.wikipedia.org/wiki/Lamport%27s_Distributed_Mutual_Exclusion_Algorithm)

Proces vyšle žádost, a čeká až dorazí odpovědi od všech ostatních, a všechny žádosti v jeho frontě mají vyšší časovou značku.

- p posílá žádost Mp se svým timestampem
- přijetí žádosti od i: zapamatuje si žádost, pošle ACK s vlastním timestampem
- když dostane od někoho ACK, přidá si ho ke svému požadavku
- do kritické sekce proces vstoupí, když
  1. od všech ostatních dostal ACK
  2. a zároveň neví o žádném starším požadavku
- když skončí s kritickou sekcí, pošle ostatním release
- po přijetí release si proces vymaže k němu patřící žádost (a někdo další pak na základě toho může vlézt do kritické sekce)

R-A rozsirenie: zlucenie release and reply odpovede do jednej (delayed reply)

Raymondův algoritmus

Uzly, představující procesy v distribuovaném systému jsou uspořádány do binárního stromu. V tomto stromě se pohybuje "token", udávající povolení ke vstupu do kritické sekce. Procesy bojují o token, pouze ten který ho má smí vstoupit do KS. Problémem je přerušení některé větve v systému - pak procesy v této větvi nemohou vstoupit do KS.

Více v PDI:

https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/PDI-IT/lectures/lecture_2.pdf

https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/PDI-IT/lectures/PDI-Group_Communication.pdf

https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/PDI-IT/lectures/lecture3-part1.pdf

Další zdroje

[zcu.arcao.com/kiv/ds/ds.pdf] kapitola 4.3

Složitost distribuovaných a paralelních algoritmů

Počet procesorů

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

Čas

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

Cena

- 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

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

Složitost

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 $p(n^2)$.

Topologie distribuovaných a paralelních algoritmů

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,\,\dots

Víceúrovňové sítě

Spojují p procesorů s p paměťovými moduly pomocí $p.log(p)$ přepínačů.

Kategorie: Státnice 2011 | Paralelní a distribuované algoritmy

---

- Stránka byla naposledy editována 5. 6. 2011 v 10:48.