PDV

Paralelní hardware

Hardwarová podpora pro paralelní výpočty: (super)skalární architektury, pipelining, spekulativní vyhodnocování, vektorové instrukce, vlákna, procesy, GPGPU. Hierarchie cache pamětí

Superskalární architektury

- Architektury procesorů, které umožňují vykonávat několik instrukcí současně (například součet prvků dvou polí)

Pipelining

Viz APO

Spekulativní vyhodnocování

- Viz APO

Vektorové instrukce

- Instrukce některých procesorů, které umožňují reprezentovat část pole jako vektor čísel
- Tyto procesory pak umožňují operace nad celými vektory

Vlákna, procesy

- Oboje v APO a OSY

GPGPU

- = Obecné výpočty na grafických procesorech
- Platforma CUDA umožňuje provádět paralelní výpočty na grafické kartě

Hierarchie Cache pamětí

- Taky APO

Konkurence

Komplikace v paralelním programování: souběh (race condition), uváznutí (deadlock), iluze sdílení (false sharing)

Souběh (race condition)

- Jev ke kterému může dojít když více vláken přistupuje paralelně ke dvěma nebo více proměnným
- Vlákna ve stejnou dobu přistupují ke stejné proměnné, takže výpočty provedené s touto proměnnou mohou být nekonzistentní

Deadlock (uváznutí)

- Popsáno v OSY i v DBS

Iluze sdílení (false sharing)

- Jev ke kterému dochází když jsou ve velké míře využívány prostředky k synchronizaci vláken
- Vlákna pak neustále čekají na jiná a dochází k minimální nebo žádné paralelizaci

Paralelismus v C/C++

Podpora paralelního programování v C a C++: pthreads, thread, jthread, atomic, mutex, lock_guard

Pthreads

- POSIX vlákna, která jsou využívána v programovacím jazyce C

- Mají definované mutexy, podmínkové proměnné, spinlocky i zámky pro čtení a zápis

Vlákna v C++11

- Definována v knihovně *thread* (*std::thread*)
- Vlákno se spustí při zavolání konstruktoru jeho třídy
- Ukončení pomocí funkce *join* nebo *detach*
- Vlákno může ukončit celou aplikaci pomocí *std::terminate*, pak je nutné předat výjimku pomocí *std::promise*

Vlákna v C++20 (jthread)

- Join je zavolán v destruktoru (je automatický)
- Je možné zažádat o ukončení vlákna pomocí *std::stop_token*

Atomické proměnné

- Definované šablonou *std::atomic<T>*
- Proměnné nad kterými lze definovat atomické operace, například *std::atomic<T>::fetch_add*
- Zajišťují, aby nad nimi nedošlo k konkurentním problémům

Mutex

- V C++ třída *std::mutex*
- Zámek, který umožňuje určitou část kódu zamknout tak, že v ní smí být pouze jedno vlákno najednou
- Funkce *lock()* a *unlock()*
- Automatickou správu mutexů poskytuje v C++ třída *std::lock_guard<T>*
  - Konstruktor automaticky volá lock, destruktor zámek odemyká
  - Parametrem šablony typicky bývá std::mutex

OpenMP

Podpora paralelního programování v OpenMP: sériově-paralelní model uspořádání vláken (fork-join), paralelizovatelná úloha (task region), různé implementace specifikace. Direktivy parallel, for, section, task, barrier, critical, atomic.

Sériově-paralelní model uspořádání vláken

- Je využívaný v OpenMP
- Paralelně vykonávaný kód obsahuje úseky, kde vznikne několik vláken, které začnou řešit a na konci úseku jsou opět spojena do jednoho
- Říkáme tomu také *fork-join model*

Paralelizace v OpenMP

- OpenMP používá preprocesorové direktivy ve tvaru `#pragma omp ...` k paralelizaci sériově napsaného kódu
- Pokud je část úlohy paralelizovatelná, můžeme pomocí OpenMP zavést tz. *task region* který je vykonán pouze jedním vláknem zatímco ostatní vlákna pokračují dále
- Na konci paralelního bloku se čeká až jej všechna vlákna dokončí, tomu se dá zamezit pomocí direktivy `nowait`

OpenMP direktivy

- `parallel` - Vytvoří blok, který začne vykonávat několik vláken najednou (počet lze specifikovat pomocí `num_threads(n)`)
- `for` - Pokud je napsáno uvnitř paralelní sekce před for cyklem, for cyklus je rozdělen tak, že každé vlákno vykoná nějakou jeho část (ne vždy žádoucí)
- `task` - specifikuje region, který je vykonán jen jedním vláknem a přeskočen ostatními
- `sections` a `section` - velmi podobné jako tasky, sections specifikuje část kódu ve které bude několik sekcí, kde každou ze sekcí vykoná jedno vlákno
- `barrier` - vynucuje část kódu, kde se bude čekat na všechna vlákna
- `critical` - vytváří kritickou sekci, kam smí přistupovat jen jedno vlákno najednou (ekvivalent) mutexů
- `atomic` - specifikuje operaci, která by měla být provedena atomicky (pouze jedním vláknem najednou)

Dekompozice

Techniky dekompozice programu: statické a paralelní rozdělení práce. Threadpool a Fronta úkolů. Balancování a závislosti (dependencies); na příkladech z řazení: quick sort, merge sort; na příkladech z numerické lineární algebry a strojového učení: násobení matice vektorem,

násobení dvou matic, řešení systému lineárních rovnic

Statické a paralelní rozdělelní práce

- *Statické rozdělení práce* spočívá v tom, že jednotlivým vláknům je předem určeno jakou přesně část práce mají vykonat
- Paralelní rozdělení práce jsem nedohledal, nejspíš mysleli dynamické
- *Dynamické rozdělení práce* spočívá v tom, že program přiděluje vláknům úkoly dynamicky na základě vytížení jednotlivých vláken
- Nevýhodou dynamického rozdělení je ale stále to, že vlákna musí přistupovat ke sdíleném zdroji (například fronta s úkoly)
- *Threadpool* je jakási množina vláken, která jsou dostupná pro paralelizaci nějakého problému
- Vlákna v threadpoolu čtou z *fronty úkolů*
- Každé vlákno může mít vlastní frontu úkolů, což řeší konkurenci, pak ale může dojít k nevyváženým frontám
  - Řešení je takové, že vlákno s prázdnou frontou může "ukrást úkol jiné frontě"

Závislosti

- Ne vždy je možné úkol triviálně dekomponovat, protože některé části problému mohou záviset na jiných
- Problém bude tedy splněn až tehdy když jsou splňeny všechny závislosti
- V openmp lze realizovat například pomocí `#pragma omp tasks depend([in/out/inout]:variables`

Paralelní quick sort

- Řadící algoritmy jako quick sort lze paralelizovat díky tomu, že jde o algoritmy typu rozděl a panuj
- V quicksortu konkrétně můžeme paralelně řadit menší pole, která byla od sebe rozdělena podle nějakého pivotu
- Zdrojový kód:

```
void qs(std::vector<int>& vector_to_sort, int from, int to) {
        if (to - from <= base_size) {
                std::sort(vector_to_sort.begin() + from, vector_to_sort.begin() + to);
                return;
        }

        //rozdeleni dle pivota (vector_to_sort[from])
        int part2_start = partition(vector_to_sort,from,to,vector_to_sort[from]);
        if (part2_start - from > 1) {
#pragma omp task shared(vector_to_sort) firstprivate(from,part2_start)
                {
                        qs(vector_to_sort, from, part2_start);
                }
        }
        if (to - part2_start > 1) {
                qs(vector_to_sort, part2_start, to);
        }
}
```

Paralelní merge sort

- Merge sort je taky algoritmus typu rozděl a panuj, takže po rozdělení na menší úseky můžeme předat podproblém jinému vláknu, vlákno pak bude řešit spojování dvou podpolí na menším úseku
- Zdrojový kód:

```
void ms(std::vector<int>& vector_to_sort, int from, int to) {
        if (to - from <= base_size) {
                ms_serial(vector_to_sort,from,to);
                return;
        }
        int middle = (to - from)/2 + from;
#pragma omp task shared(vector_to_sort) firstprivate(from,middle)
        ms(vector_to_sort, from, middle);
        ms(vector_to_sort, middle, to);
#pragma omp taskwait
```

```
        std::inplace_merge(...);
}
```

Paralelní násobení matice vektorem

- Nejefektivnější je násobit jednotlivé složky vektoru sloupci matice a sloupec nakonec sečíst
- Zdrojový kód (snad nebude potřeba):

```
void multiply(std::vector<int> &A, std::vector<int> &x, std::vector<int> &y) {
#pragma omp declare reduction(vec_int_plus : std::vector<int> : \
std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(),\
std::plus<int>())) initializer(omp_priv = omp_orig)
        int tmp;
#pragma omp parallel for num_threads(thread_count) reduction(vec_int_plus : y)
        for (int i=0; i<ROWS; i++) {
                tmp = 0;
                for (int j=0; j<COLS; j++) {
                        tmp += A[i * COLS + j]*x[j];
                }
                y[i] += tmp;
        }
}
```

Paralelní násobení matic

- Naivní přístup by bylo paralelizovat výpočet každého prvku c_{ij}, to je však dost neefektivní
- Lepší je rozdělit výslednou matici na submatice, které budou počítány
- Další přístup je analogie matice a vektoru, takže násobit řádky první matice s jednotlivými prvky řádků druhé matice

Paralelní řešení systému lineárních rovnic

- Řešení lineárních rovnic lze paralelizovat například při použití gaussovy eliminační metody
- V přednášce to není moc vysvětleno, tady je zdrojový kód:

```
void gauss_par(std::vector<double>& A) {
#pragma omp declare reduction(vec_int_plus : std::vector<double> : \
std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(),\
std::plus<double>())) initializer(omp_priv = omp_orig)
        for (int i=0; i<ROWS; i++) {
        // Make all rows below this one 0 in current column
#pragma omp parallel for num_threads(thread_count)
                for (int k=i+1; k<ROWS; k++) {
                        double c = -A[k * COLS + i]/A[i*COLS + i];
                        for (int j=i; j<ROWS; j++) {
                                if (i==j) {
                                        A[k * COLS + j] = 0;
                                } else {
                                        A[k * COLS + j] += c * A[i * COLS + j];
                                }
                        }
                }
        }
}
```

Selhání v distribuovaných systémech

Úvod do distribuovaných systémů (DS). Charakteristiky DS. Čas a typy selhání v DS. Detekce selhání v DS. Detektory selhání a jejich vlastnosti.

Distribuované systémy

- Distribuovaný systém je soubor výpočetních jednotek, které se uživateli jeví jako jeden

- Od paralelních se liší v tom, že výpočty v distribuovaných systémech probíhají na zcela oddělených zařízeních, které je tak obtížnější synchronizovat
- Jednotlivé výpočetní jednotky jsou propojeny komunikační sítí a komunikují spolu prostřednictvím zpráv

Charakteristiky distribuovaných systémů

- Výpočetní jednotky nemají sdílenou paměť
- Nesdílejí ani globální hodnoty (každá jednotka může mít jiný pojem o čase)
- Selhávají nezávisle na sobě, v případě že nějaká jednotka nekomunikuje, není možné určit co se s ní stalo
- U distribuovaných systémů je narozdíl od paralelních cílem zabezpečit trvalou dostupnost služeb

Selhání v distribuovaných systémech

- Pokud má n nezávislých procesů pravděpodobnost selhání p, pak je dostupnost distribuovaného systému

$$(1-p)^n$$

- Replikace nezávislých počítačů tedy zvyšuje odolnost vůči selhání
- Algoritmy distribuovaných systémů je potřeba navrhnout tak aby byly robustní vůči selhání

Typy selhání

- *Selhání procesu*:
  - *Havárie* - proces přestane zcela fungovat (přestane reagovat na zprávy)
  - *Byzantské selhání* - proces vykonává chybný algoritmus, ale stále reaguje na zprávy
- *Selhání kanálu*:
  - *Ztráta zprávy* - zpráva není doručena cílovému procesu
  - *Rozdělení* - procesy se rozdělí do disjunktních množin, které mezi sebou nedokážou komunikovat

Čas v DS

- Každý proces má *lokální hodiny*, které ale nemusí ukazovat přesný čas
- Synchronizace je možná jen s určitou přesností
- Na základě synchronizace rozlišujeme
  - *Synchronní systémy* - synchronní výpočty (horní odhad rychlosti), synchronní komunikace (horní odhad doby přenosu zprávy) a synchronní hodiny (horní odhad rychlosti driftu hodin)
  - *Asynchronní systémy* - žádné časové limity na rychlost procesů, dobu přenosu zpráv nebo drift lokálních hodin

Detekce selhání

- Systémy založené na skupinách procesů
- Předpokládáme FIFO kanál a havárii procesu (nikoliv byzantské selhání)

Centralizovaný heartbeat

- Zvolíme jeden proces p_j, kterému všechny ostatní posílají heartbeats
- Funguje velmi dobře pro všechny procesy kromě p_j
- Selhání p_j není nikdy detekováno
- p_j může být přetížen

Kruhový heartbeat

- Heartbeats jsou odesílány periodicky sousedům každého procesu
- Nevýhodou je, že je třeba udržovat kruh a že detektor selže při současném selhání více procesů

All-to-all heartbeat

- Každý proces posílá heartbeat všem ostatním procesům
- Selhání je prakticky vždy detekováno, ale občas může být detekováno i falešně (pokud jeden proces nedostane včas heartbeats, označí všechny ostatní za havarované)

Vlastnosti detektorů selhání

- *Přesnost* - Nedochází k mylné detekci

- *Úplnost* - Každé selhání je časem detekováno alespoň jedním funkčním procesem
- Úplnost je důležitější než přesnost
- *Rychlost detekce* - Čas od okamžiku, kdy první proces detekuje selhání
- *Škálovatelnost* - Počet posílaných zpráv a rovnoměrné rozložení komunikační zátěže

Detektor SWIM

- Tři druhy zpráv `ping`, `ping-req` a `ack`
- Každý proces p_i občas pošle ping náhodnému procesu p_j
- Pokud mu tento proces včas nepošle spátky ack, dojde k timeoutu a proces p_i pošle k náhodným procesům zprávu ping-req
- Tyto procesy potom zkusí poslat ping procesu p_j
- Pokud žádný z procesů nedostane zpět ack a nepošle ack procesu p_i, p_j je prohlášen za selhaný

Čas a kauzalita v DS

Čas a kauzalita v DS. Uspořádání událostí v DS. Fyzické hodiny a jejich synchronizace. Logické hodiny a jejich synchronizace.

Fyzické hodiny

- *Mimoběžnost hodin* (clock skew) - Rozdíl v času hodin dvou procesů
- *Drift hodin* - Rozdíl v rychlosti hodin dvou procesů
  - Mimoběžnost hodin se bude časem zvyšovat

Synchronizace fyzických hodin

- *Externí synchronizace* - Čas je udržován v rozmezí δ od času externích referenčích hodin
- *Interní synchronizace* - Rozdíl časů pro každý pár procesů je udržován v rozmezí δ
- Externí synchronizace implikuje interní synchronizaci
- Při synchronizaci musíme brát v úvahu latenci komunace mezi procesy *Cristianův algoritmus*:

$$C_i := t + \frac{T_{RT} - l_{min} + l'_{min}}{2}$$

- l_{min} a l'_{min} jsou minimální latence při komunikaci směrěm k hodinám resp. od hodin
- Druhý běžný způsob jak synchronizovat je NTP (*Network Time Protocol*), který vyšle dvě zprávy spočítá offset na základě časů odeslání a přijetí obou zpráv:

$$o = \frac{(t_1^r - t_2^r + t_2^s - t_1^s)}{2}$$

Logické hodiny

- Přiřazování logických časových značek různým událostem
- Značky popisují kauzální vztah mezi *událostm*i
- Díky časovým značkám mohou být události lineárně seřazeny
- Událost může být odeslání nebo přijetí zprávy
- Definujeme relaci "stalo se před" nad událostmi, značí se šipkou ($A \rightarrow B$) a je částečným uspořádáním

Synchronizace logických hodin

- *Lamportovy logické hodiny* - Každý proces má své logické hodiny, které se synchronizují podle přijímání zpráv
- Synchronizace:
  - Každý proces p_i si drží lokální logické hodiny C_i
  1. Po každé události v p_i se C_i inkrementuje o jedna
  2. Každé zprávě m je přiřazena časová značka $\mathrm{ts}(m) = C_i$
  3. Když proces p_j přijme zprávu, tak
     1. $C_j = \max\{C_j, \mathrm{ts}(m)\}$
     2. Provede krok 1 předtím než předá zprávu aplikaci
- Logické hodiny neimplikují kauzalitu, kauzalitu ale implikují *vektorové hodiny*, kde si každý proces drží informaci o stavu vektorových hodin všech ostatních procesů

Globální stav

Globální stav v DS a jeho výpočet. Řez distribuovaného výpočtu. Algoritmus pro distribuovaný globální snapshot. Stabilní vlastnosti DS.

Globální stav v DS

- *Globální stav* = množina lokálních stavů všech procesů a stavů všech komunikačních kanálů mezi nimi v jednom okamžiku
- *Globální snapshot* = záznam globálního stavu

Řez distribuovaného výpočtu

- Způsob jak zaznamenat globální snapshot bez použití globálních hodin
- *Řez* = časová hranice v každém procesu a v každém komunikačním kanále
  - Události, které nastaly před řezem jsou *v řezu*
  - Události, které nastaly po něm jsou *mimo řez*
- *Konzistentní řez* = řez, který splňuje kauzalitu, tedy pokud je událost f řezu a událost e se stala před řezem, pak je událost e také v řezu
- *Konzistentní globální snapshot* odpovídá konzistentnímu řezu
- Globální snapshot není konzistentní, pokud nemohl být v jeden okamžik zpozorován

Chandy-Lamport algoritmus pro distribuovaný globální snapshot

Zahájení tvorby snapshotu

Iniciující **proces** p_i odešle ZNAČKU ▪ všem ostatním procesům (i sobě)

Příjem ZNAČKY ▪ procesem p_i kanálem $C_{m,i}$

if (p_i dosud nezaznamenal svůj stav) **then**
  p_i **zaznamená** svůj stav;
  p_i **zaznamená** stav kanálu $C_{m,i}$ jako prázdnou množinu;
  p_i *každým* odchozím **kanálem** $C_{i,j}$ **odešle** jednu ZNAČKU ▪ (předtím než skrze $C_{i,j}$ pošle jakoukoliv jinou zprávu);
  p_i **zapne zaznamenávání** zpráv doručených skrze všechny ostatní příchozí kanály $C_{j,i}$ *kromě* $C_{m,i}$
else
  p_i **zaznamená** stav kanálu $C_{m,i}$ jako množinu všech zpráv, které p_i obdržel skrze $C_{m,i}$ od doby, kdy zahájil záznam $C_{m,i}$;
  p_i **ukončí** záznam kanálu $C_{m,i}$;
end if

- Výsledkem je vždy konzistentní řez

Stabilní vlastnosti DS

- *Stabilní vlastnost* je taková vlastnost, která zůstává splněna navždy potom, co je ve výpočtu už jednou splněna
- Příklad - výpočet skončil

Vzájemné vyloučení

Vzájemné vyloučení procesů v DS. Algoritmy pro vyloučení procesů a jejich vlastnosti.

Problém vzájemného vyloučení

- *Kritická sekce* je část kódu u které potřebujeme zaručit, že ji v každém okamžiku vykonává maximálně jeden proces
- Definujeme funkce *enter()* a *exit()* pro vstup a výstup do kritické sekce
- Na úrovni operačního systému umíme problém řešit pomocí synchronizačních nástrojů, v distribuovaném systému potřebuje algoritmus

Požadavky na algoritmus pro vyloučení procesů

- *Bezpečnost* - Nejvýše jeden proces v kritické sekci
- *Živost* - Každý požadavek na vstup kritické sekce je časem uspokojen
- *Uspořádání* (není nutné) - Předchází-li žádost jednoho procesu do kritické sekce kauzálně žádosti jiného procesu, tento proces by se měl do kritické sekce dostat dříve
- Uvažujeme navíc, že procesy neselhávají a komunikační kanály jsou perfektní (FIFO, zprávy se neduplikují, nevznikají, neztrácejí) a uvažujeme asynchronní systém s konečnou latencí

Analýza výkonnosti algoritmů pro vzájemné vyloučení

- *Komunikační zátěž* = počet zpráv poslaných při každém vstupu a výstupu do KS
- *Zpoždění klienta* = zpoždění procesu při vstupu do KS za předpokladu, že jiné procesy na vstup nečekají
- *Synchronizační zpoždění* = interval mezi vystoupením jednoho procesu z KS a vstoupením dalšího

Centralizovaný algoritmus

- Je zvolen jeden koordinátor
- Ten spravuje speciální token, který držiteli umožní vstup do KS a frontu požadavků na vstup do KS
- Před vstupem do KS je poslán požadavek na token koordinátorovi, po přijetí tokenu algoritmus vstupuje do kritické sekce
- Po výstupu je token vrácen koordinátorovi
- Koordinátor po přijetí požadavku na token předá token pokud jej drží, pokud jej nedrží, tak tento požadavek přidá do fronty
- Ve chvíli kdy koordinátor obdrží token tak zkontroluje frontu zda tam není žádný požadavek a případně jej obslouží
- Bezpečnost je zaručena, a živost za našich předpokladů také
- Komunikační zátěž - 2 zprávy vstup, 1 výstup
- Zpoždění klienta - 2 latence
- Synchronizační zpoždění - 2 latence

Kruhový algoritmus

- N procesů v kruhu
- Proces může poslat zprávu následníkovi
- Mezi procesy koluje jeden token
- Před vstupem proces čeká dokud neobdrží token
- Po vstupu proces pošle token následníkovi
- Pokud proces obdrží token a nečeká na vstup, tak jej hned předá dále
- Komunikační zátěž - N vstup, 1 výstup
- Zpoždění klienta - 0 až N zpráv
- Synchronizační zpoždění - 1 až $N-1$ komunikačních latencí

Ricart-Agrawalův algoritmus

- Nepoužívá token, ale kauzalitu a multicast
- Nižší synchronizační zpoždění než kruhový algoritmus a nepoužívá centrální proces
- Každý proces si udržuje stav, který může nabývat 3 hodnot: WANTED, HELD, RELEASED (u všech procesů je inicializován na RELEASED)
- Před vstupem do KS:
  - Je nastaven stav na WANTED
  - Poslán multicast REQUEST všem ostatním procesům
  - Čeká na odpověď
  - Po přijetí OK je stav změněn na HELD a proces vstupuje do KS
- Po přijetí REQUEST
  - Pokud je stav WANTED a je časová značka přijaté zprávy nižší než čas ve kterém začal stav WANTED, je příslušný proces uložený do seznamu čekajících požadavků
  - Jinak je posláno OK
- Po výstupu z KS je stav nastaven na RELEASED a všem procesům ze seznamu je posláno OK
- V nejhorším případě je nutno čekat než všech $(N-1)$ pošle OK
- Je zachováno pořadí, požadavky s nižší lamportovou značkou mají přednost
- Komunikační zátěž - $2(N-1)$ vstup, $(N-1)$ výstup
- Zpoždění klienta - 1 latence
- Synchronizační zpoždění - 1 latence

Volba lídra

Volba lídra v DS. Algoritmy pro volbu lídra a jejich vlastnosti.

Problém volby lídra

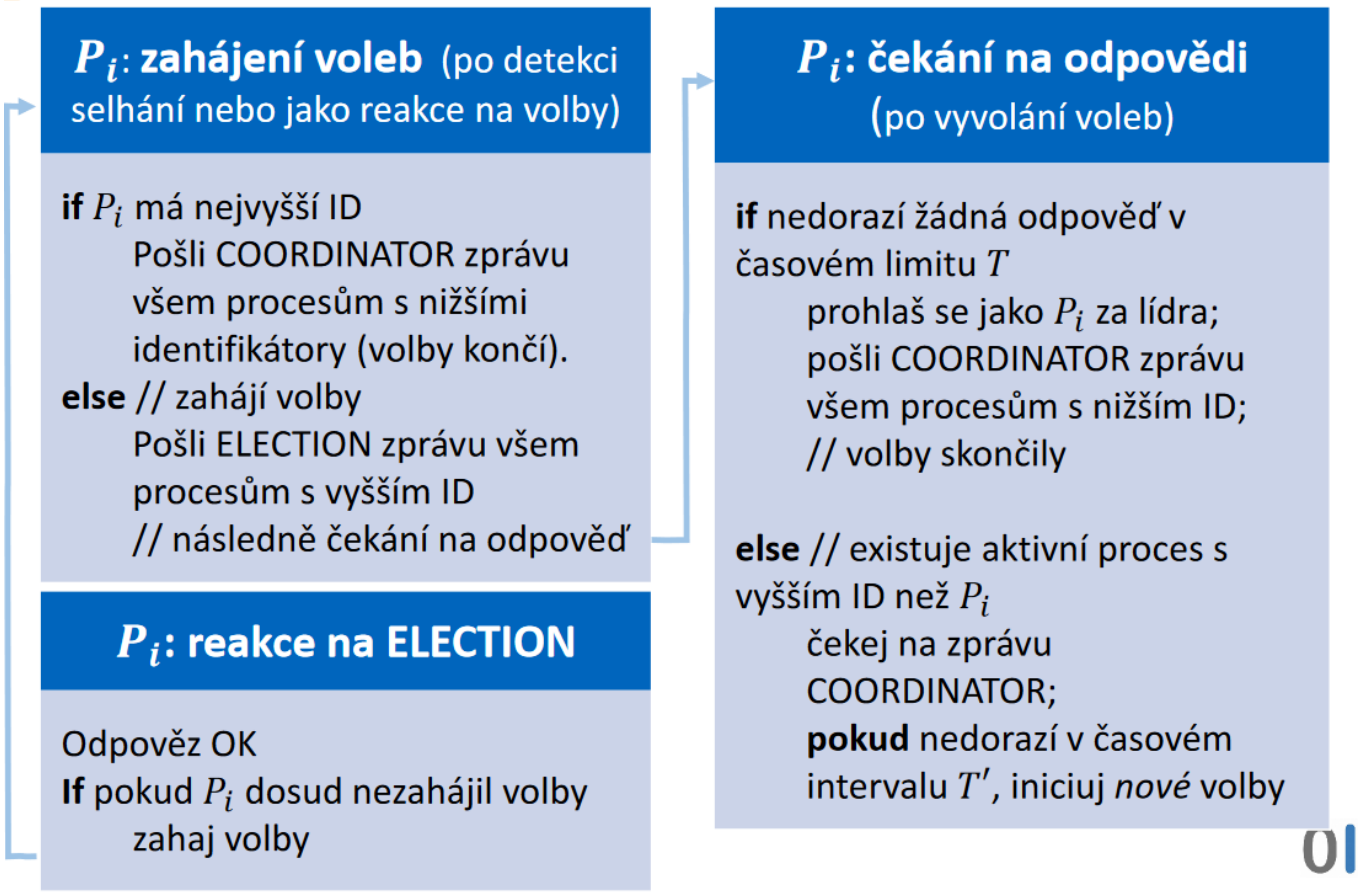
- Chceme ze skupiny procesů vybrat lídra a dát vědět všem procesům kdo je lídrem
- Předpokládáme, že procesy mohou havarovat
- Výsledek by neměl záviset na tom, který proces volbu vyvolal
- Jeden proces může v jeden okamžik vyvolat pouze jedny volby
- Více procesů smí současně vyvolat volby, ale požadujeme, aby se shodly na výsledku

Kruhový algoritmus

- Procesy jsou uspořádané do kruhu a jsou schopné detekovat selhání ostatních procesů
- Zahájení voleb, P_i pošle do kruhu zprávu $ELECTION(i)$
- Zpracování zprávy $ELECTION(j)$
  - Pokud $i < j$ pak je zpráva přeposlána
  - Pokud $i > j$ pak je zpráva nahrazena zprávou $ELECTION(i)$
  - Pokud $i = j$ pak je P_i nový koordinátor je
- Po přijetí zprávy $ELECTED(j)$ si proces poznamená, že P_j byl zvolen a pokud $i \neq j$, pak přepošle zprávu dál
- Procesy ignorují zprávy $ELECTION$ a $ELECTED$ zpráv s nižším ID než je maximální ID

Algoritmus Bully

- Proces, který detekoval selhání dosavadního lídra vyzve procesy s vyšším ID ve volbách



Konsensus

Konsensus v DS. FLP teorém. Algoritmy pro distribuovaný konsensus.

Problém konsensu

- Každý proces P má vstupní proměnnou x_p a výstupní proměnnou y_p
- Cílem je, aby se všechny procesy shodly na hodnotě výstupní proměnné
- Mnoho problémů v DS je ekvivalentních konsensu:
  - Perfektní detekce selhání
  - Volba lídra
  - ...
- V systémech bez selhání je problém konsensu řešitelný
  1. Jednotlivé procesy si vzájemně pošlou hodnoty vstupních proměnných
  2. Provedou stejný výpočet
  3. Výstupní proměnnou nastaví na výsledek výpočtu

FLP teorém

- V asynchronním distribuovaném systému nelze dosáhnout současně bezpečnosti a živosti výpočtu, pokud v něm může docházet k selháním
  - *Bezpečnost* = garance, že nikdy nedojde k nečemu špatnému
  - *Živost* = garance, že časem dojde k něčemu dobrému

Algoritmus Paxos

- Pracuje v kolech, která jsou asynchronní
  - Je-li proces v kole j a dorazí mu zpráva z kola $j+1$, přeruší činnost v rámci kola j a přesune se do kola $j+1$
  - Využívá timeouty
- Každé kolo je rozděleno do tří fází
  - *ELECTION* - je zvolen leader
  - *BILL* - zvolený lídr navrhne hodnotu, ostatní procesy potvrzují
  - *LAW* - lídr rozešle všem ostatním procesům finální hodnotu

Algoritmus RAFT

- Další algoritmus pro distribuovaný konsensus
- Není v požadavích takže jenom stručně
- Čas je rozdělený do epoch a každý server si uchovává číslo epochy, epochy pak slouží k identifikaci zastaralých informací
- Servery můžou být trojího typu: následovník, lídr a kandidát
- Když následovník neobdrží pravidelný heartbeat od lídra, vyvolá novou volbu lídra
- V každé epoše smí zvítězit maximálně jeden lídr
- Všechny servery si udržují logy, kde každý element (záznam) má vždy index, epochu a příkaz
- Záznam je *potvrzený* když je uložen na většině serverů
- Logy jsou *konzistentní* když na každé pozici obsahují záznam se stejným číslem epochy a stejným příkazem