

PA160

Vyrovnnání zátěže (Load Balancing)

Distribuované plánování (Dist. Scheduling)

Odolnost proti výpadkům (Fault Tolerance)

Vyrovnaní zátěže

- Máme množinu úloh, mezi nimiž existují nějaké závislosti
 - Zpravidla vstup/výstupní
- Máme dále množinu procesorů, které jsou schopné vzájemně komunikovat
- *Rozložením zátěže* rozumíme takové přidělení úloh procesorům, které *minimalizuje celkový čas výpočtu*

Grafová reprezentace

- Máme množinu úloh N se závislostmi, kterou reprezentujeme jako graf $G(V, U)$
 - Vrcholy jsou procesy
 - Hrana odpovídá závislosti mezi procesy
- Graf potřebujeme rozložit následujícím způsobem:

$$N = N_1 \cup N_2 \cup \dots \cup N_p$$

při platnosti

$$|N_i| \approx |N|/p$$

kde $|N_i|$ je počet procesů připadajících na jeden procesor

Shrnutí

- Vlastnosti
 - Rovnoměrné rozložení zátěže
 - Minimalizace komunikace (minimum hran mezi jednotlivými podgrafy)
- Problém je NP úplný
 - Používáme heuristické přístupy
- Rychlost rozložení versus jeho kvalita
 - Dynamické
 - Statické

Vyrovnnání zátěže a plánování

- Vyrovnnání zátěže: jak rozdělit úlohy mezi procesory
- Plánování: v jakém pořadí je spustit
- Úzce spolu souvisí (často v distribuovaném systému synonyma)

Rozdělení úloh pro vyrovnání zátěže

Přístup k řešení problému je možno rozdělit podle následujících kritérií:

- Cena úlohy
- Závislosti mezi úlohami
- Lokalita

Cena úlohy

- Kdy známe cenu
 - Před spuštěním celého problému
 - V průběhu řešení, ale před spuštěním konkrétní úlohy
 - Až po dokončení konkrétní úlohy
- Variabilita ceny

Rozdělení do tříd podle ceny

1. Všechny úlohy mají stejnou cenu: *snadné*
2. Ceny jsou rozdílné, ale známé: *složitější*
3. Ceny nejsou známy předem: *nejtěžší*

Závislosti úloh

- Je pořadí spuštění úloh důležité?
- Kdy jsou známy závislosti
 - Před spuštěním celého problému
 - Před spuštěním úlohy
 - Plně dynamicky

Rozdělení do tříd podle závislosti

1. Úlohy jsou na sobě nezávislé: *snadné*
2. Závislosti jsou známé či predikovatelní: *složitější*
 - vlna
 - in- a out- stromy (vyvážené nebo nevyvážené)
 - obecné orientované stromy (DAG)
3. Závislosti se dynamicky mění: *nejtěžší*
 - Např. úlohy prohledávání

Lokalita

- Komunikují všechny úlohy stejně (nebo alespoň podobně)?
- Je třeba některé spouštět „blízko“ sebe?
- Kdy jsou komunikační závislosti známy?

Rozdělení do tříd podle lokality

1. Úlohy spolu nekomunikují (nejvýše při inicializaci): *snadné*
2. Komunikace má známý či predikovatelný průběh: *složitější*

- Pravidelný (např. mřížka)
- Nepravidelný

Např. PDE řešiče

3. Komunikace je předem neznámá: *nejtěžší*
- Např. diskrétní simulace událostí

Přístup k řešení

- Obecně záleží na tom, kdy je konkrétní informace známa
- Základní třídy:
 - Statické (off-line algoritmy)
 - Semi-statické (hybridní)
 - Dynamické (on-line algoritmy)
- Možné varianty (nikoliv vyčerpávající výčet):
 - Statické vyrovnání zátěže
 - Semi-statické
 - Samoplánování (self-scheduling)
 - Distribuované fronty úloh
 - DAG plánování

Semi-statické vyrovnání zátěže

- Pomalá změna v parametrech, důležitá lokalita
- Iterativní přístup
 - Použije statický algoritmus
 - Použije jej pro několik kroků (akceptuje „mírnou“ nerovnováhu)
 - Přepočítá novým statickým algoritmem
- Často používán v následujících oblastech
 - Částicové simulace
 - Výpočty na pomalu se měnících mřížkách (gridy – ovšem v jiném smyslu než používány v předchozích lekcích)

Self Scheduling

- Centralizovaný pool připravených úloh
- Volné procesory vybírají z poolu
- Nové (pod)úlohy se do poolu přidávají
- Původně navržen pro plánování cyklů v překladači
- Vhodný pro
 - Množina nezávislých úloh
 - Úlohy s neznámými cenami
 - Lokalita nehraje roli
 - Centralizovaný pool snadno implementovatelný (např. SMP)

Varianty

- Self-scheduling nevhodné pro příliš malé úlohy
 - Sdružování úloh do shluků
 - * Pevná velikost
 - * Řízené sdružování
 - * Zužování (tapering)
 - * Vážené rozdělování

Pevná velikost

- Typický off-line algoritmus
- Vyžaduje velmi mnoho informací (počet a cena každé úlohy, ...)
- Je možné nalézt optimální řešení
- Teoreticky zajímavá, v praxi nepříliš použitelné

Řízené sdružování

- Použij velké shluky na začátku a menší na konci
 - Nižší režie na začátku, jemnější rozložení na konci
- Velikost shluku

$$K_i = \lceil \frac{R_i}{p} \rceil$$

kde R_i je počet zbývajících úloh a p je počet procesorů.

Zužování

- Analogické předchozímu, ale velikost shluku je funkcí i variace ceny úloh
- Využívá historická data
 - Malá variace \implies velké shluky
 - Velká variace \implies malé shluky

Vážené rozdělování

- Opět analogie self scheduling
- Bere do úvahy i výpočetní sílu uzlů
- Vhodné pro heterogenní systémy
- Používá rovněž historické informace

Distribuované fronty úloh

- Self-scheduling pro distribuovanou paměť
- Namísto centralizovaného poolu fronta úloh
- Vhodné
 - Distribuované systémy
 - Lokalita nepříliš důležitá
 - Pro statické i dynamické závislosti
 - Neznámou cenu úloh

Difuzní přístup

- Zavádí závislost na topologii (předchozí neuvažují)
- Vlastnosti
 - Lépe bere do úvahy lokalitu (resp. požadavky na ni)
 - Poněkud pomalejší
 - Musí znát cenu úlohy v okamžiku jejího vytvoření
 - Nepracuje se závislostmi mezi úlohami

Příklad

- Distribuovaný systém modelován jako graf
- V každém kroku se spočte *váha* úloh zbývajících na každém procesoru
- Procesory si tuto informaci vymění a následně provedou vyrovnání
- Možná vylepšení
 - Zohledňuje množství dříve zaslaných dat
- Lokalita není významným prvkem (přesto zlepšení proti náhodnému rozložení zátěže)

DAG plánování

- Opět grafový model
 - Uzly představují úlohy (výpočty; případně vážené)
 - Hrany reprezentují závislosti a případně komunikaci (mohou být rovněž vážené)
- Vhodné např. pro digitální zpracování signálu (DAG znám)
- Základní strategie: Rozděl DAG za minimalizace komunikace a zaměstnání všech procesorů (minimalizace času)
 - NP úplné
 - Oproti prostému rozdělení grafu bere do úvahy závislosti mezi úlohami

Praktické problémy

- Kdy je vhodné
 - Pro středně zatížené systémy
 - U nízko zatížených – vždy je volný procesor
 - U velmi zatížených – nehraje roli
- Podle čeho rozhodnout
 - Metriky určení výkonu
 - * Musí být snadno měřitelné
 - * Musí se promítat do optimalizovaných parametrů
 - Určení kvality
 - * Průměrná doba „reakce“

Návrh přístupu

- Měření zátěže: fronty, využití CPU
- Rozhodování: statické, dynamické, adaptivní
- Součásti
 - Který proces přenést: preferovány nové procesy
 - Kdy proces přenést: většinou při dosažení nějaké úrovně (treshold)
 - Kam proces přenést: nejbližší soused (difuze), náhodně, ...
 - Kde a jaká informace je k dispozici
 - * Řízeno: požadavky (sender/receiver), časem (periodické), změnou stavu

Rozhodnutí řízeno vysílajícím (sender)

- Pouze nové úlohy
- Rozhodnutí: podle lokální kapacity
- Umístění
 - Náhodné: vyber a pošli
 - Limit: vyzkoušej n uzlů, pokud žádný pod limitem, úlohu nepřenášej
 - Nejkratší: poptej paralelně a náhodně n uzlů; přesuň na nejméně zatížený uzel pod limitem

Rozhodnutí řízeno přijímajícím (receiver)

- Pokud odcházející (končící) proces sníží zátěž pod limit, vyber proces odjinud
- Vhodné pro nové i částečně rozpracované úlohy
- Umístění:
 - Limit: vyzkoušej sekvenčně až n dalších uzlů
 - Nejkratší: poptej paralelně a náhodně n uzlů, vyber ten, který má nejvyšší zátěž nad limitem

Příklad: V System ze Stanfordu

- Výměna informací iniciována změnou stavu
 - Významné změny zátěže oznámeny všem uzlům
- M nejméně zatížených uzlů jsou přijímající, ostatní jsou posílající
- Přenosy iniciovány vysílajícím
- Umístění:
 - Náhodně vyber možného přijímajícího
 - Pokud je ještě přijímajícím (pod limitem), přesuň úlohu
 - V opačném případě zkus jiného

Příklad: Sprite z Berkeley

- Centralizovaná informace (u koordinátora)
 - Update iniciován změnou stavu
 - Přijímající: stanice bez interaktivního uživatele pod limitem
- Manuální selekční strategie (uživatel) – vždy vysílající
- Umístění: dotaz na koordinátora
- Stanice s úlohou se stane vysílajícím, pokud má proces a přijde uživatel

Migrace kódu a procesů

- Proces = kód + data + stack
- Migrace procesu (*silná mobilita*)
- Migrace kódu (*slabá mobilita*)
 - program vždy startuje z počátečního stavu
- Flexibilita
 - Dynamická konfigurace (na žádost)
 - Není třeba používat preinstalovaný software

Příklad: Sprite

- Migrace procesu (i běžícího)
- Přes sdílený systém souborů
- Přenos stavu
 - Všechno ulož do (sdíleného) swapu
 - Přesuň tabulky stránek a deskriptory souborů přijímajícímu
 - Založ proces u přijímajícího a nahraj nezbytné stránky
 - Předej řízení
- Jediný problém: komunikační závislosti
 - řešeno přesměrováním po přesunu

Migrace v heterogenních systémech

- Podporována pouze slabá mobilita v klasických modelech
- Rozvoj s využitím virtuálních strojů: skriptovací jazyky a Java

Odolnost proti výpadkům

- Primární problém distribuovaných systémů
- Základní složky
 - Rozpoznání výpadku
 - Reakce na výpadek
 - Dosažení konsensu

Klasický příklad pro konsensus

- Definice výchozího stavu
 - Město obklíčeno 4 armádami
 - Každá armáda má v čele generála
 - Rozhodnutí zaútočit musí udělat všichni 4 generálové současně
 - Komunikace spolehlivá, ale může trvat libovolně dlouho
 - Generálové mohou být zavražděni (armáda bez generála nebojuje)
- Je možné, aby se generálové shodli na rozhodnutí?

Nemožnost shody

- Negativní teoretický výsledek (Fischer, Lynch, Paterson: JACM, **32:2**, 1985):
- *V asynchronních systémech nelze v konečném čase dosáhnout konsensu*

Formálnější definice

- Máme množinu distribuovaných procesů s počátečními stavy $\in \{0, 1\}$
- Požadujeme, aby se všechny shodly na jedné hodnotě
- Dodatečná podmínka
 - Musí existovat případ shody jak na stavu 0, tak na stavu 1 (triviální shoda není problém)

Silná shoda – podmínky

- Žádné dva procesy se neliší ve stavu
- Výsledný stav musí být výchozím stavem alespoň jednoho ze zúčastněných procesů
- Každý proces se v konečném čase rozhodne pro nějaký stav a toto rozhodnutí je nerevokovatelné

Slabá shoda – podmínky

- Žádné dva procesy se neliší ve stavu
- Může dojít k shodě na různých stavech
- Alespoň některé procesy se v konečném čase rozhodnou pro nějaký stav a toto rozhodnutí je nerevokovatelné

Vlastnosti modelu

- **Asynchronicita**
 - Neexistuje horní hranice pro čas, která proces potřebuje k jednomu kroku
 - Neexistuje horní hranice pro čas, který potřebuje doručení zprávy
 - Neexistují synchronizované hodiny
- **Předávání zpráv v point2point síti**
- **Předpokládáme:**
 - *Nejsou chyby v komunikaci*
 - *Proces buď funguje správně nebo se zhroutil*

Důsledky

- *Neexistuje deterministický algoritmus, který vyřeší problém shody v asynchronním systému s procesy, které se mohou zhroutit*
- Je totiž nemožné rozlišit následující případy
 - Proces neodpovídá, protože se zhroutil
 - Proces neodpovídá, protože je pomalý
- V praxi překonáváno zavedením timeoutů a ignorováním (případně „zabitím“) příliš pomalých procesů
- Timeouty součástí tzv. Failure Detectors

Fault tolerantní broadcast

- Problém shody by byl řešitelný, pokud by existoval vhodný typ fault tolerantního broadcastu
- Různé typy broadcastů
 - Základní spolehlivý
 - FIFO broadcast
 - Příčinný (Casual) broadcast
 - *Atomický broadcast* – ekvivalentní na řešení problému shody v asynchronním prostředí

Spolehlivý broadcast

- Je možno jej zkonstruovat pomocí dvoubodových primitiv `send` a `receive`
- Základní vlastnosti
 - **Správnost**: Pokud *korektní* proces p pošle broadcastem zprávu m , pak ji také eventuálně doručí.
 - **Shoda**: Pokud *korektní* proces p pošle broadcastem zprávu m , pak ji eventuálně doručí všechny *korektní* procesy.
 - **Integrita**: Jakoukoliv zprávu m proces doručí pouze jednou a pouze tehdy, pokud byla dříve poslána nějakým procesem p .

Difuzní algoritmus

- Jednoduché řešení
 - Používá `send` a `receive`
 - Princip
 - Proces p posílající broadcast označí posílanou zprávu m jednak svým identifikátorem, jednak pořadovým číslem poslané broadcastové zprávy a pošle ji všem svým sousedům
 - Přijetí zprávy se pak skládá z:
 - * Vlastního doručení zprávy (právě jednou, podle klíče odesilatele a pořadové zprávy)
 - * Pokud sám není původní odesílatel, pak ji odešle všem svým sousedům
- Přijetí se provede pouze jednou, další příšlé zprávy se stejným klíčem se ignorují

FIFO Broadcast

- Spolehlivý broadcast neklade žádná omezení na *pořadí* doručení zpráv
- Je tedy možné získat následnou zprávu (z pohledu odesilatele) dříve, než je přijata původní
- FIFO broadcast: zprávy od jednoho vysílajícího musí být doručeny ve stejném pořadí, v jakém je vyslal
- FIFO broadcast = Reliable broadcast + FIFO uspořádání
 - Pokud proces pošle zprávu m dříve než zprávu m' , pak žádný *správný* proces nedoručí zprávu m' dříve než zprávu m .
- Je možno jej vytvořit jako rozšíření Reliable broadcastu

Příčinný broadcast

- FIFO broadcast stále není dostačující: je možno dostat zprávu od třetí strany, která je reakcí na zprávu původní dříve, než obdržíme původní zprávu.
- Řešení: příčinný broadcast
- Casual broadcast = Reliable broadcast + Příčinné uspořádání
 - Jestliže skupinové odeslání zprávy m příčinně předchází zprávu m' , pak žádný *správný* proces nedoručí zprávu m' dříve než m .
- Je možno vytvořit jako rozšíření FIFO broadcastu

Atomický broadcast

- Ani příčinný broadcast není dostačující: je občas třeba garantovat správné pořadí doručení všech replik
 - Dvě bankovní pobočky: jedna dostane dříve informaci o tom, že má přičíst úrok a teprve následně úložku, druhá naopak. Výsledkem je nekonzistentní stav.
- Atomic broadcast = Reliable broadcast + Úplné uspořádání
- Neexistuje v asynchronních systémech

Timed Reliable Broadcast

- Cesta k praktické realizaci
- Zavede horní limit na čas, do něhož se musí zpráva doručit
- Timed Reliable broadcast = Reliable broadcast + Timeliness
 - Existuje známá konstanta Δ taková, že jestliže zpráva m je skupinově vyslána v čase t , pak žádný *správný* proces ji nedoručí po čase $t + \Delta$.
- Dosažitelné v synchronních sítích
- Existuje transformace, která jakýkoliv Timed Reliable broadcast rozšíří na atomický broadcast.

Failure Detectors

- Zavedení částečné synchronizace
- Rozpoznání *špatných* (zhroucených) procesů
- Částečná synchronizace je skryta v detektorech zhroucení
 - Aplikace se od nich dozví, které procesy nekomunikují

Failure Detectors – základní vlastnosti

- Každý proces má lokální Failure Detector Modul
- Každý modul si drží seznam potenciálně zhroucených uzlů
- Lokální proces se ptá pouze lokálního modulu
- Moduly si mezi sebou vyměňují informaci
- Jsou *nespolehlivé* – potenciálně zhroucený uzel může být ze seznamu později odstraněn
- Aplikace pracuje se specifikací, nikoliv implementací

Perfektní detektor

- Základní vlastnosti
 - **Přesnost:** Žádný správný proces se nikdy nedostane do seznamu potenciálně zhroucených v žádném FD
 - **Úplnost:** Každý skutečně zhroucený uzel se jednou dostane do seznamu potenciálně zhroucených ve všech FD
- Vhodná abstrakce
- Těžce implementovatelné
- Existují *zeslabení* tohoto modelu

Zeslabení perfektního detektoru

- V úplnosti

- Každý skutečně zhroucený proces je eventuálně zařazen do seznamu některých správných uzlů

- V přesnosti

- Některé správné procesy nejsou nikdy zařazeny do žádného seznamu
- Případně slabší: Existuje čas, po jehož uplynutí není žádný správný proces zařazen v seznamu potenciálně zhroucených žádného správného FD
- Nejslabší: Existuje čas, po jehož uplynutí některé správné procesy nejsou nikdy zařazeny do seznamu žádného FD

Problém shody a FD

- *Problém shody je možno vyřešit za použití perfektního detektoru selhání*
- *Problém shody je možno vyřešit i za použití slabších FD*
- *Problém shody je možno vyřešit za použití FD založeném na zeslabeném předpokladu úplnosti i nejslabším předpokladu přesnosti (to je také nejslabší FD, jehož pomocí lze problém shody vyřešit)*