Přidělování práce v prostředí s distribuovanou pamětí, možnosti urychlení výpočtu a přiřazení procesů na jednotlivé uzly.

Thursday, May 30, 2013 8:34 AM

Přidělování práce v prostředí s distribuovanou pamětí

Pozn. Přednáška z PPR b_advance.

Faktory ovlivňující rychlost výpočtu

- Virtuální topologie, komunikační schéma, distribuované aplikace
- Prezentovaná síťová topologie
- Fyzická topologie sítě
- Výkonnost jednotlivých uzlů v síti
- Výpočetní model distribuované aplikace
    - Každý proces může běžet podle vlastního programu

Řešení obecně

- Umístit procesy na uzly sítě takovým způsobem, aby běžely co nejrychleji a zároveň bylo komunikační zpoždění co nejmenší
    - Může se jednat i o protichůdné požadavky
- Zatížení a dostupnost uzlů se může měnit v čase
- Jsou tři typy úloh
    - S konečným časem výpočtu – distribuovaná aplikace má jenom něco spočítat a pak skončí
    - S teoreticky nekonečným časem výpočtu – distribuovaná aplikace poskytuje službu
        - Neplatí, že uzel musí být zcela vytížen výpočtem po celou dobu
    - Processing While in Transit – výpočet se nad daty provádí během jejich přenosu Např. Active Network

Možnosti urychlení

MPMD

multiple processes, multiple data

- Ve všech uzlech není stejný program
- Každý proces má svoje data

SPMD

- Do všech uzlů se zavede stejný program
- Do uzlů se zavedou specifická data procesů
- Každý proces dokáže zjistit svoje ID a z něj určí sousedy a svůj díl dat
- Jeden z procesů je řídící
    - Obvykle ten první vytvořený
        - ⇨ Mívá ID 0, ale konkrétně to závisí na sw
    - Zpracovává dílčí mezivýsledky
- V homogenním distribuovaném prostředí se zjistí celkový objem dat, počet spuštěných procesů a práce se přidělí najednou
    - Např. cluster z identických stanic a MPI
- V heterogenním prostředí je lepší využít dynamické přidělování práce
    - Uzly nemusejí být stejně výkonné
    - Ale i v případě, kdy uzly nejsou využívány jednouživatelsky
    - Např. model farmer-workers, kdy farmáři udělíme čestný titul identický program -> bude k tomu všemu ještě makat jako worker

Urychlení u Farmer-Worker v distribuovaném prostředí, pokud farmer jen kompletuje dílo od

workerů a pokud zanedbáme odeslání a příjem zprávy je přibližně S ~ N - 1, tj. přibližně lineární

Hodí se tehdy, když:
- Datový objem zpráv je malý
- Výpočet je v porovnání s objemem zpráv náročný
- Např. nemá smysl počítat pole integerů k součtu na jiný uzel, v dnešní době je **operace mov zhruba stejně časově náročná jako add, muselo by se tedy vykonat spoustu operací navíc** => zpomalení

V heterogenním prostředí se realizuje automatický **load balancing** viz dále, urychlení pak závisí na velikosti dat ke zpracování jedním procesem, je třeba najít ideální objem dat, který příliš nezpomaluje a zároveň poskytuje dobré urychlení
- Příliš malé objemy dat nevedou k urychlení
- Příliš velké objemy dat k urychlení už vedou, ale zase se zbytečně příliš čeká na procesy, které z různých důvodů pracují pomaleji než ostatní

Ideální velikost posílaných dat tak závisí na:
- Výpočetním výkonu uzlů
- Měla by uzel zaměstnat na tak dlouho, aby bylo možné úkolovat uzly v době, kdy ostatní počítají
  - V první vlně přidělit náhodné velikosti dat od jednoho až dvou násobků objemu zprávy
    - ⇨ Tím se na nějakou dobu zabrání konvergenci, kdy bude komunikační linka využívána všemi procesy
      - □ Eliminace komunikačního zpoždění překrytím výpočtem
        - ◆ Kromě neblokujících komunikačních operací
- Data by se měla spočítat v nějakém přiměřeném čase, aby se na poslední proces nečekalo celou věčnost

MPSD

- Step-locked
- Pásová výroba (pipeline)
- Části dat by měly být stejně velké ne podle objemu dat, ale výpočetně
- Problémem může být kapacita přenosových kanálů
  - Objem dat může být příliš velký a tak uzel může nějakou dobu čekat na data
  - Ideálně se v i-tém kroku
    - Počítají data
    - Data z i-1 kroku se posílají dalšímu uzlu v řadě
    - Přijímají se data, která se budou počítat v i+1 kroku
    - Překrytí doby komunikace dobou výpočtu => eliminace komunikačního zpoždění, pokud se data déle počítají, než přijímají
      - ⇨ Pokud ne, zmenšit objem posílaných dat
- N - počet úkolů
- Pošleme-li objem vstupních dat k nekonečnu, pak je urychlení N

Load-Sharing

- Předpokládá se prostředí pracovních stanic, které nemusejí být vždy plně vytíženy
- Jeden uzel se vyhradí jako master, kde se spustí aplikace
  - Ostatní uzly se označují termínem slave
- V okamžiku, kdy je master vytížen na maximum, zkusí se vyhledat nevytížený uzel

Červ

- jednotlivé procesy se mohou replikovat na nevytížených uzlech
- červ se skládá z několika segmentů – procesů
- počet segmentů je buď pevně stanoven, nebo se při pokročilejší implementaci stanovuje dynamicky podle okolností
- nápadně připomíná šíření virů
- červ musí být věrohodný pro uživatele pracovních stanic

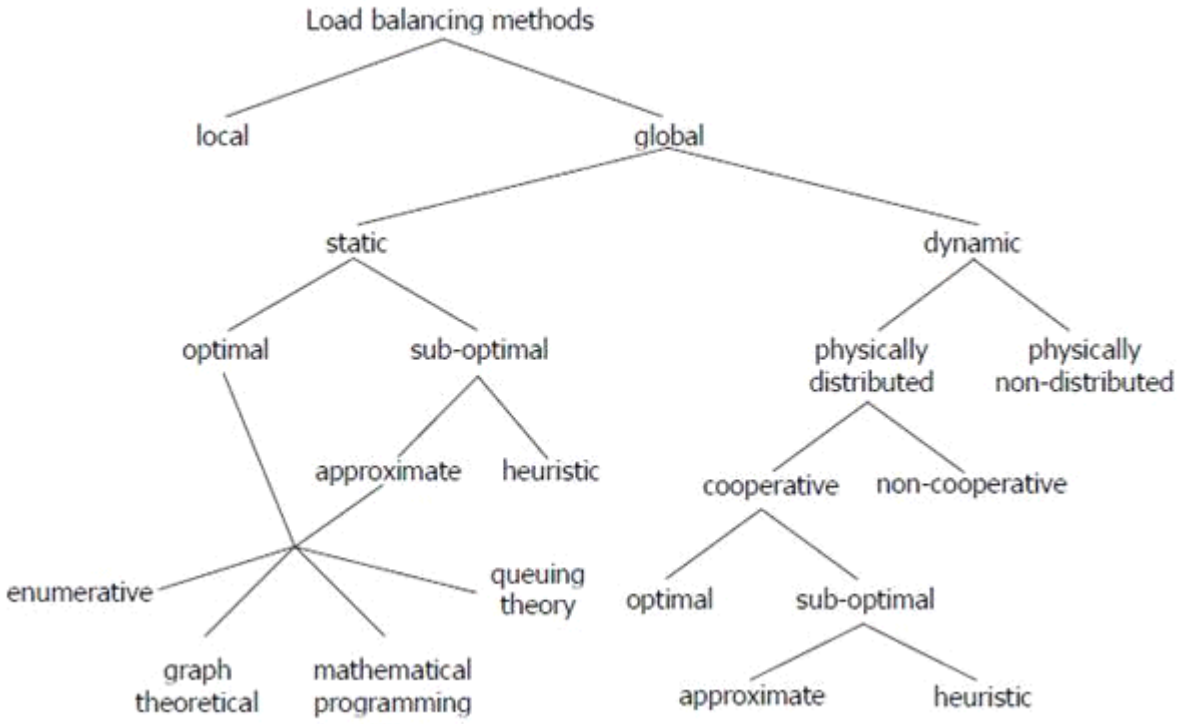
- komponenty červa
  - inicializační kód ke spuštění master
  - inicializační kód ke spuštění slave
  - výpočetní program
  - ze života červa
    - nalezení nevytíženého uzlu
  - žádný uzel nezná globální stav sítě, a proto se každý segment musí postarat o nalezení volné stanice sám za sebe
  - lze hledat například pomocí broadcast hodila by se synchronizace, protože několik segmentů může soupeřit o stejný uzel
    - uvolnění uzlu
  - segment se musí postarat, aby uzel byl zase viditelný jako dostupný i pro ostatní
    - kontrola růstu
  - čím větší červ, tím vyšší rychlost výpočtu, ale vznikají další problémy
- rychlost nemusí růst lineárně
- synchronizace
- stabilita

Condor

- snaží se o fér využívání všech uzlů
- pokud některý uzel selže, proces se restartuje jinde
- arbitr, který rozhoduje o tom, kde se spustí proces
  - sám hledá použitelné uzly
  - centralizované místo
- možnost selhání
- checkpoints
  - procesy lze relokovat za běhu distribuované aplikace
- používá se ukládání obrazu procesu v paměti,
- ne rekonstrukce stavu
- uzly musejí být identické
- co s otevřenými soubory?
  - checkpoint se ukládá periodicky
- pokud selže výpočet, použije se poslední
- checkpoint
  - pre-emptivnost procesů je implementována pomocí checkpointů

Load-Balancing

- na rozdíl od load-sharingu se předpokládá, že celá síť je dostupná pro výpočet
- existuje několik různých metod, které se liší použitelností, účinností, náročností na zdroje (paměť, procesor, …), přesností, spolehlivostí, …



- statické
  - výpočet přiřazení procesů na uzly je proveden ještě před spuštěním distribuované aplikace
    - výpočet může běžet libovolně dlouho, abychom dosáhli požadované přesnosti předpovědi – pokud ji metoda umožňuje dosáhnout
  - nelze reagovat na dynamické změny v prostředí
  - vyžadují předem spoustu informací o chování sítě a aplikace (např. kom. zpoždění, doby běhu procesů)
    - nereálné požadavky nelze splnit
      - ⇨ vliv na přesnost a tedy i rychlost výpočtu
- dynamické
  - výpočet přiřazení procesů na uzly sítě se provádí za běhu distribuované aplikace
  - výpočet se odehrává v reálném čase a nemůže si proto dovolit konzumovat příliš mnoho zdrojů
    - umí se vyrovnat s dynamickými změnami
    - ⇨ procesy musí umět pre-empci
    - ⇨ potřebné informace lze zjistit až za běhu aplikace, nebo si jich část vyžádat předem
- pre-emptivní
  - procesy lze přerušit během výpočtu a přemístit je na jiný uzel, aby bylo možné kompenzovat změny v síti
  - např. některý z procesů mohl skončit svoji činnost, nebo se odebral do dlouhodobého wait-stavu
  - pokud tuto vlastnost procesy nemají, na změny lze reagovat až při vytváření
- centralizované
  - mají jeden centrální prvek, arbitr, který rozhoduje o rozdělování zátěže na jednotlivé uzly
  - centrální prvek je slabé místo, co se stane, když selže?
    - ⇨ centralizované správa vyžaduje komunikaci jednoho uzlu se všemi
  - možnost přetížení
  - arbitr má přehled o známé síti, a proto lze očekávat, že dokáže zátěž rozdělovat celkem efektivně bez rizik, která jsou jinak spojena s distribuovaným principem
- distribuované
  - rozhodování o rozdělování zátěže provádí několik až všechny procesy
  - mohou být distribuovány na několik uzlů
  - když jeden selže, nic se neděje, pokud ho výpočetní model aplikace nutně nepotřebuje k životu
  - procesy, které provádějí rozhodování mohou být buď specialisté, anebo to mají jako

„vedlejšák“ ke své hlavní činnosti

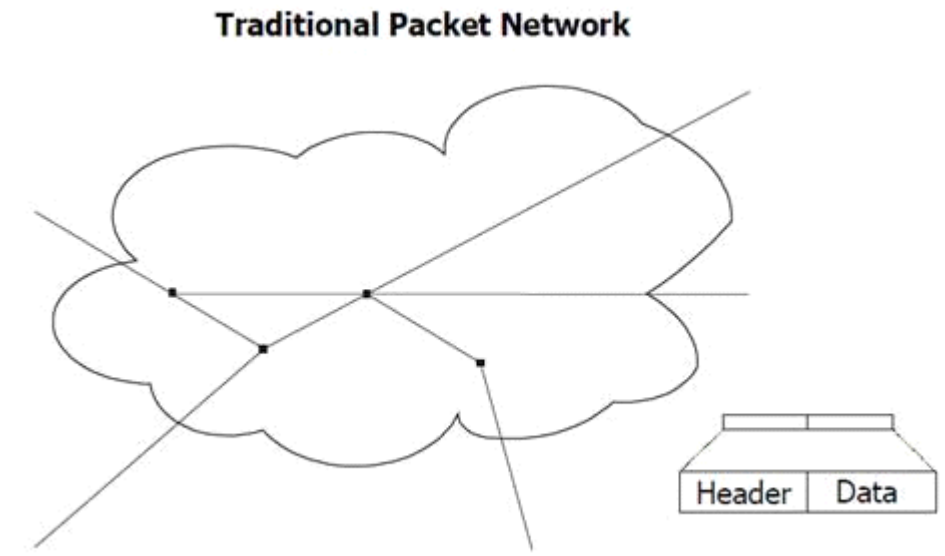
- adaptivní
  - síť prochází změnami během výpočtu – mění se stav uzlů
  - adaptivní metody berou do úvahy i několik předchozích stavů při rozhodování o přidělení zátěže
- kooperativní
  - každý proces se může rozhodovat buď sám za sebe, nebo může na rozhodnutí spolupracovat s ostatními
  - přímo – procesy spolupracují nad konkrétním rozhodnutí
  - nepřímo – procesy dávají informace o svých rozhodnutích k dispozici ostatním a ty je použijí při svých rozhodnutích
- sender-initiated
  - v okamžiku, kdy je uzel zatížen přes určitou mez, začne vyhledávat jiné uzly, kam by přemístil část své zátěže
  - je to režie navíc, protože čas potřebný na vyhledávání nových uzlů mohl být použit na běh procesů
- receiver-initiated
  - v okamžiku, kdy zátěž uzlu klesne pod určitou mez, začne vyhledávat jiné uzly, odkud by mohl převzít jejich zátěž
  - režie se projeví zvýšenou komunikací, uzel má dost volného výpočetního času, který může alokovat pro vyhledávání zátěže

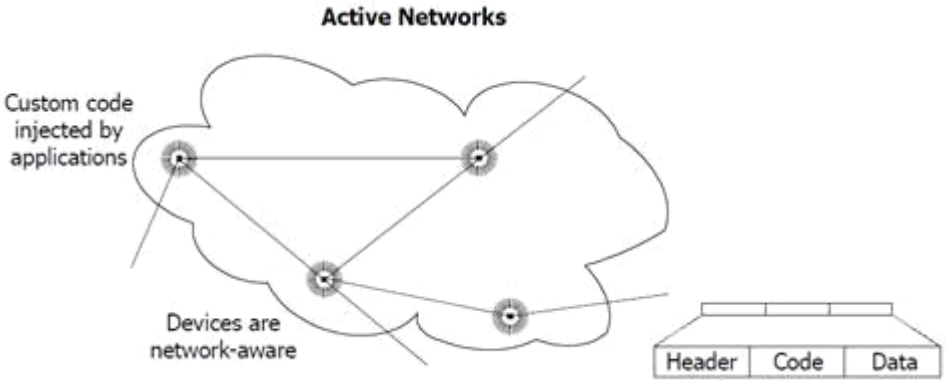
Load Redistribution

- load-balancing tradičně měří zátěž v počtu procesů, což není zrovna to nejlepší
- následující text bude o Load-Redistribution Method in Distributed Environment
- metoda vyžaduje pokročilou síťovou architekturu jako jsou Aktivní sítě

Aktivní sítě

- stvořil Pentagon pro vyřešení nedostatků IP protokolu
- např. *Any-Cast* = metodologie pro adresování a routování, kdy jsou datagramy od jediného odesilatele routovány uzlu, který je topologicky nejblíže v dané skupině potenciálních příjemců (ačkoliv může být zasláno vícero uzlům, pokud mají stejnou adresu). Rozdíl od multicastu, který posílá všem ze skupiny najednou (a vždy), broadcastu, který zasílá jednoduše všem.
- Any-Cast je až v IPv6, aktivní sítě mají PAMcast – Programmable Any-Multicast – služba pro doručování zpráv, která generalizuje jak anycast tak multicast , která doručuje zprávy M z N příjemců, kde 1 <= M <= N
- IP
  - Dvojice vysílající a příjemce
  - Vysílající odešle paket na konkrétní adresu, včetně portu, kde předpokládá příjemce
  - Pokud tam není, paket se zahodí a vysílající zjistí chybu až timeoutem
- Aktivní síť
  - Paket, nazývaný capsule, je asociován s kódem, který se spustí na každém uzlu, kterým capsule prochází
    - Vždy je přítomný příjemce
  - Kód může manipulovat s daty, vlastnostmi (cíl, TTL, atd.) a vykonávat další uživatelsky definované činnosti
  - Proces se označuje termínem aktivní aplikace
    - Distribuovaná aktivní aplikace se skládá z několika aktivních aplikací, které mohou injektovat capsule a zároveň capsule může injektovat aktivní aplikace





- Komunikační model sám-sobě
  - Je možné injektovat capsuli do sítě, aby nasbírala potřebná data a pak je předala procesu, který ji injektoval
  - U IP by bylo nutné mít dopředu na každém uzlu, který by capsule mohla navštívit, spuštěný specializovaný proces
- Migrace procesů
  - Migrující proces změní svoji síťovou adresu, ale ještě ji nedal na vědomí ostatním procesům
  - Informaci o své nové síťové adrese zanechal na uzlu, odkud migroval
  - Capsule, která má doručit data, dorazí na uzel, odkud proces odmigroval, tam ho nenajde, ale použije svůj kód, aby si přečetla novou adresu a pouze změní svůj cíl
  - Ostatní procesy si mohou aktualizovat záznamy až později – lazy update, u IP je nutné vyřešit předem
- V aktivní síti je zapotřebí standardizace pouze dvou věcí
  - Programového kódu
    - Kód vykonává Execution Environment (EE), na jednom uzlu může být několik EE
  - Code distribution protocol
  - Vše ostatní je pak už aplikačně specifické
- Při přerozdělování zátěže (load redistribution) se procesy rozhodují samy za sebe, periodicky sledují své okolí (jako kolonie organismů)
  - Dosažení vyváženého stavu ne hned, ale postupně
- Capsule zjistí síťové okolí uzlu z hlediska topologie, výkonnosti, zatížení, komunikačního zpoždění apod.
  - Využije se při hledání výhodnějšího uzlu pro odmigrování
- **Rizika:**
  - *Masová migrace* - více procesů si vybere stejný uzel, ten se stane přetíženým

- *Oscilace* - proces se může pohybovat po síti, aniž by něco počítal
    - řešení - zavedení kreditů, od určitého počtu může migrovat
- *Zbytečná migrace*

Přiřazení procesů na jednotlivé uzly

- **Alokování uzlů**
    - 1 proces na 1 uzel
        - Např. pevně daná u paralelního počítače
        - 1 proces dokáže plně využít celý uzel, takže nemá smysl jich na jednom uzlu spouštět několik
        - OS uzlu neumí spustit více jak jeden proces najednou
    - Potenciálně nula až několik procesů na jeden uzel
    - Přidělení celé sítě pro jeden výpočet
        - Celkový čas výpočtu je pak dán
            - Dobou k zavedení programů, spuštění procesů a distribuce dat do uzlů
            - Vlastním výpočetem
            - Získáním výsledků z uzlů
    - Přidělení části sítě jednomu výpočtu
    - Několik paralelně běžících výpočtů
        - Na jednom uzlu může běžet několik procesů
        - Nelze se spoléhat na odvozená urychlení, protože ta nepočítala se zátěží, kterou vygeneruje neznámý kód
        - Nehodí se pro synchronní/lockCstepped algoritmy – na společném uzlu by dva spolupracující procesy na sebe musely čekat dobu výpočtu jednoho kroku
- **Identifikace procesů**
    - Jedinečná ID procesů
    - Interakce send/receive (vše ostatní je na nich postaveno)
    - Podle přidělení na uzly:
        - 1 uzel – 1 proces
        - Více procesů na uzlu
        - Více procesů na uzlu a procesy mohou migrovat (tabulka umístění procesů)

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>