Návrh paralelních algoritmů

Úvod I.

Vývoj paralelního algoritmu je nunto chápat jako **optimalizaci**.

1. nejprve vždy vyvíjíme co **nejjednodušší sekvenční algoritmus bez optimalizací**

Úvod I.

Vývoj paralelního algoritmu je nunto chápat jako **optimalizaci**.

1. nejprve vždy vyvíjíme co **nejjednodušší sekvenční algoritmus bez optimalizací**
   - **za každou cenu se vyhýbáme předčasné optimalizaci**

Úvod I.

Vývoj paralelního algoritmu je nunto chápat jako **optimalizaci**.

1. nejprve vždy vyvíjíme co **nejjednodušší sekvenční algoritmus bez optimalizací**
   - **za každou cenu se vyhýbáme předčasné optimalizaci**
   - ta může zcela zbytečně poničit čistý návrh algoritmu

Úvod I.

Vývoj paralelního algoritmu je nunto chápat jako **optimalizaci**.

1. nejprve vždy vyvíjíme co **nejjednodušší sekvenční algoritmus bez optimalizací**
   - **za každou cenu se vyhýbáme předčasné optimalizaci**
   - ta může zcela zbytečně poničit čistý návrh algoritmu
   - bez základní jenoduché verze kódu nemůžeme poměřit přínos paralelizace

Úvod I.

Vývoj paralelního algoritmu je nunto chápat jako **optimalizaci**.

1. nejprve vždy vyvíjíme co **nejjednodušší sekvenční algoritmus bez optimalizací**
   - **za každou cenu se vyhýbáme předčasné optimalizaci**
   - ta může zcela zbytečně poničit čístý návrh algoritmu
   - bez základní jenoduché verze kódu nemůžeme poměřit přínos paralelizace
2. máme-li základní funkční kód, který není dostatečně výkonný, přistupujeme k optimalizacím

Úvod I.

Vývoj paralelního algoritmu je nunto chápat jako **optimalizaci**.

1. nejprve vždy vyvíjíme co **nejjednodušší sekvenční algoritmus bez optimalizací**
   - **za každou cenu se vyhýbáme předčasné optimalizaci**
   - ta může zcela zbytečně poničit čistý návrh algoritmu
   - bez základní jenoduché verze kódu nemůžeme poměřit přínos paralelizace
2. máme-li základní funkční kód, který není dostatečně výkonný, přistupujeme k optimalizacím
   - pokud je to možné, optimalizujeme jen sekvenční kód

Úvod I.

Vývoj paralelního algoritmu je nunto chápat jako **optimalizaci**.

1. nejprve vždy vyvíjíme co **nejjednodušší sekvenční algoritmus bez optimalizací**
   - **za každou cenu se vyhýbáme předčasné optimalizaci**
   - ta může zcela zbytečně poničit čistý návrh algoritmu
   - bez základní jenoduché verze kódu nemůžeme poměřit přínos paralelizace
2. máme-li základní funkční kód, který není dostatečně výkonný, přistupujeme k optimalizacím
   - pokud je to možné, optimalizujeme jen sekvenční kód
   - pokud to nepostačuje, přikročíme k paralelizaci

Úvod I.

Vývoj paralelního algoritmu je nunto chápat jako **optimalizaci**.

1. nejprve vždy vyvíjíme co **nejjednodušší sekvenční algoritmus bez optimalizací**
   - **za každou cenu se vyhýbáme předčasné optimalizaci**
   - ta může zcela zbytečně poničit čistý návrh algoritmu
   - bez základní jenoduché verze kódu nemůžeme poměřit přínos paralelizace
2. máme-li základní funkční kód, který není dostatečně výkonný, přistupujeme k optimalizacím
   - pokud je to možné, optimalizujeme jen sekvenční kód
   - pokud to nepostačuje, přikročíme k paralelizaci
3. během implementace optimalizací provádíme průběžné testy a kontrolujeme, zda optimalizovaný kód dává stále správné výsledky

Úvod I.

Vývoj paralelního algoritmu je nunto chápat jako **optimalizaci**.

1. nejprve vždy vyvíjíme co **nejjednodušší sekvenční algoritmus bez optimalizací**
   - **za každou cenu se vyhýbáme předčasné optimalizaci**
   - ta může zcela zbytečně poničit čistý návrh algoritmu
   - bez základní jenoduché verze kódu nemůžeme poměřit přínos paralelizace
2. máme-li základní funkční kód, který není dostatečně výkonný, přistupujeme k optimalizacím
   - pokud je to možné, optimalizujeme jen sekvenční kód
   - pokud to nepostačuje, přikročíme k paralelizaci
3. během implementace optimalizací provádíme průběžné testy a kontrolujeme, zda optimalizovaný kód dává stále správné výsledky
4. nakonec poměřujeme přínos optimalizace tj. výslednou **efektivitu paralelizace**

Úvod II.

Budeme se tedy zabývat:

1. návrhem paralelních algoritmů
2. (testováním paralelních algoritmů)
3. analýzou paralelních algoritmů

Návrh paralelních algoritmů

Metodiky návrhu paralelních algoritmů:

- **task/channel model** - Ian Foster
  - naprosto nejběžnější postup při návrhu paralelních algoritmů
- **bulk synchronous parallel model** - `www.bsp-worldwide.org`

Task/channel model I.

Tento model reprezentuje paralelní výpočet jako množinu úloh (*tasks*), které mezi sebou komunikují pomocí komunikačních kanálů (*channels*).

Task/channel model I.

Tento model reprezentuje paralelní výpočet jako množinu úloh (*tasks*), které mezi sebou komunikují pomocí komunikačních kanálů (*channels*).
Task představuje:

- program
- lokální paměť
  - instrukce a soukromá data
- vstupně/výstupní porty
  - task je používá ke komunikaci s ostatními tasky

Task/channel model II.

Channel je modelován jako:

- fronta zpráv spojující výstupní port jednoho tasku se vstupním portem nějakého jiného tasku
- data se na vstupu přijemce objevují ve stejném pořadí, v jakém bylo odeslána odesílatelem
- task nemůže přijímat data, která nebyla ještě odeslána
  - přijímání zpráv je vždy **blokující**
- task, který zrávu odesílá nemusí čekat, až druhý task zprávu přijme
  - odesílání zpráv je vždy **neblokující**
- přijímání zpráv je **synchronní**
- odesílání zpráv je **asynchronní**

Task/channel model III.

Výhodou *task/channel* modelu je, že jasně odlišuje přístup do lokální paměti a komunikaci mezi tasky.

- to je nezbytné pro návrh algoritmů pro architektury s distribuovanou pamětí
- umožňuje to efektivnější návrh algoritmů pro architektury se sdílenou pamětí

Task/channel model III.

Výhodou *task/channel* modelu je, že jasně odlišuje přístup do lokální paměti a komunikaci mezi tasky.

- to je nezbytné pro návrh algoritmů pro architektury s distribuovanou pamětí
- umožňuje to efektivnější návrh algoritmů pro architektury se sdílenou pamětí

Definition

Doba běhu paralelního algoritmu je pak definována jako čas, po který byl aktivní alespoň jeden task.

Fosterova metodika návrhu paralelních algoritmů

Ian Foster navrhl čtyři kroky vedoucí k návrhu paralelního algoritmu:

1. *partitioning* - rozdělení úlohy
   - celou úlohu rozdělíme na malé kousky - prvotní tasky - *primitive tasks*
   - snažíme se o co nejjemnější rozdělení
2. *communication* - komunikace
   - odvodíme, jak spolu musí jednotlivé podúlohy komunikovat
3. *agglomeration* - aglomerace
   - slučujeme malé podúlohy do větších za účelem redukce komunikace (počtu kanálů)
   - dostáváma tak vlastní tasky
4. *mapping* - mapování
   - jednotlivé tasky přidělujeme procesorům/výpočetním jednotkám

Partitioning I.

- jde o proces rozdělení celé úlohy na menší celky - *primitive tasks*.
- tento proces se také nazývá **dekompozice.**
- existují různé techniky dekompozice

Rekurzivní dekompozice

1. rekurzivní dekompozice

- je vhodná pro algoritmy navržené metodou "rozděl a panuj"
- metodu "rozděl a panuj"využijeme k vygenerování prvotních tasku
  - původní úloha se rozdělí na několik menších celků, na které se rekurzivně aplikuje stejný postup
  - zastavíme se většinou u velmi malých úloh, které jsou mnohem jednodušší k vyřešení
- typyckými příklady pro tuto dekompozici jsou - quick sort, rychlá fourierova transformace nebo binární vyhledávání v setříděném seznamu

Datová dekompozice I.

2. datová dekompozice

- je vhodná pro úlohy zpracovávající velké množství dat
- datovou dekompozici je možné provádět podle
  - vstupních dat
  - výstupních dat
  - podle vstupních i výstupních dat
  - podle mezivýsledků
- data rozdělíme na menší celky a k nim přidělíme prvotní tasky, které je budou zpracovávat

Datová dekompozice II.

Příklad: Násobení matic

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

- výpočet lze rozdělit na čtyři úlohy

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \end{aligned}$$

- jde o datovou dekompozici podle výstupních dat

Datová dekompozice III.

Příklad: Výpočet součtu, součinu nebo průměru dlouhé řady

- výstupem je jen jedno číslo, nelze tedy provést dekompozici podle výstupních dat
- zadanou posloupnost můžeme rozdělit na několik podposloupností
- každou podposloupnost zpracuje jeden prvotní task
- jde o dekompozici podle výstupních dat

Datová dekompozice IV.

Příklad: Spočítání výskytu zadaných sekvencí v jedné dlouhé posloupnosti (bioinformatika)

- nejprve můžeme provést dekompozici podle výstupních dat
- získáme několik podúloh, z nichž každá počítá výskyty jedné sekvence v celé posloupnosti
- na tyto podúlohy aplikujeme dekompozici podle vstupních dat
- každou podúlohu tak rozdělíme na několik menších, z nichž každá počítá výskyt dané sekvence v podposloupnosti původní posloupnosti

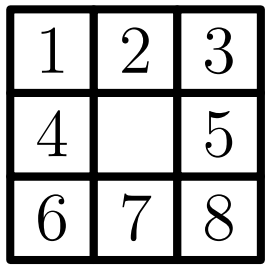
Dekompozice při prohledávání I.

3. dekompozice při prohledávání

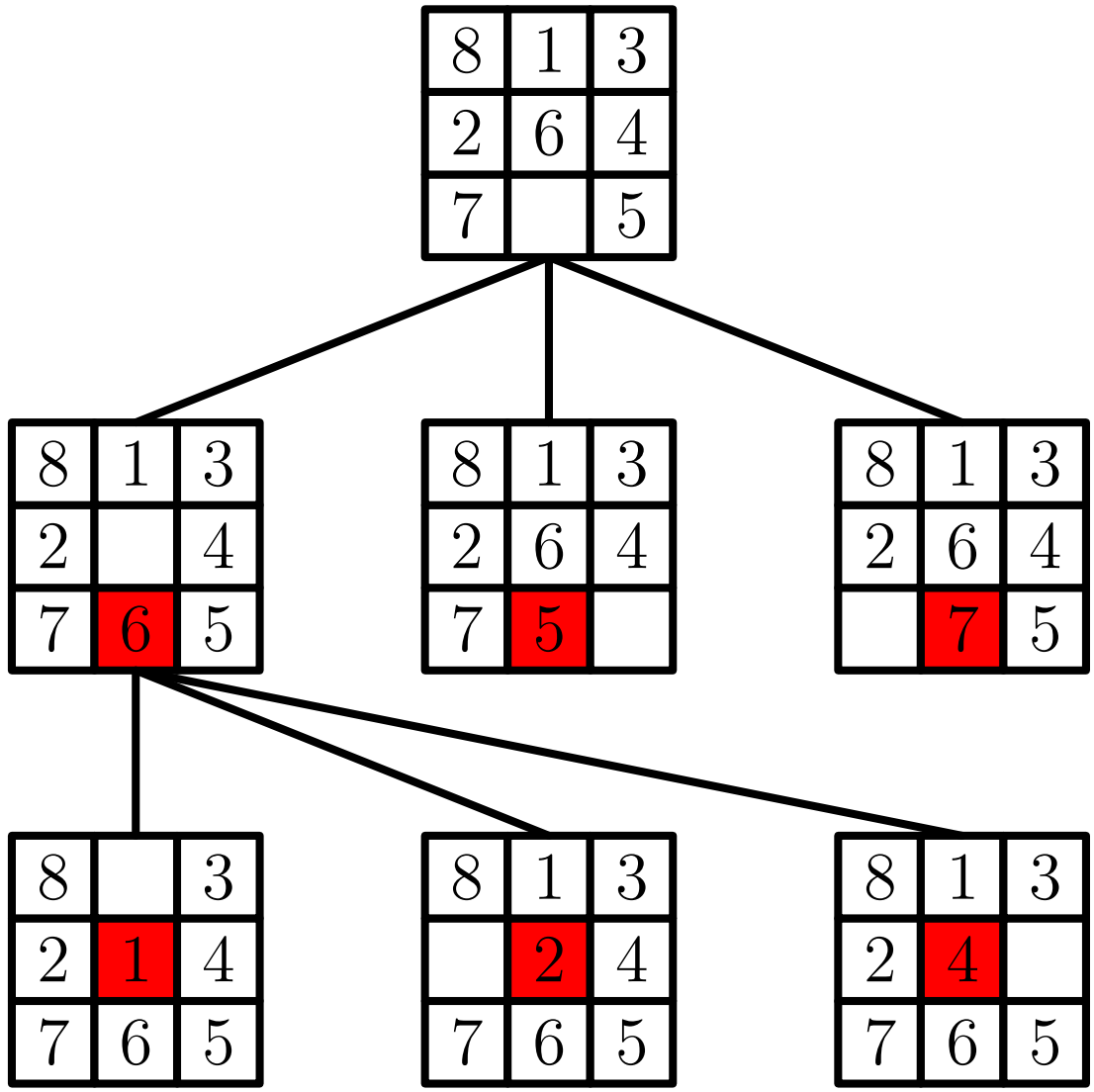
- využívá se při prohledávání stavového stromu
- vyskytuje se často v úlohách z umělé inteligence
  - hra 15, šachy

Dekompozice při prohledávání II.

Příklad: Hra Lišák - zjednodušená hra 15



Cílový stav hry



Dekompozice při prohledávání III.

- hru 15 (lišák) lze řešit prohledáváním stavového stromu
- strom začneme prohledávat sekvenčně
- s tím, jak se strom větví, vytváříme pro prohledání každé větve nový task
- prohledání různých větví může trvat různě dlouho

Spekulativní dekompozice

4. spekulativní dekompozice

- pokud se výpočet dělí na několik větví, lze výsledky jednotlivých větví vypočítat dopředu, a pak použít výsledek té větve, která bude opravdu provedena
- zpracování jednotlivých větví provádí nové tasky
- je dobré umět určit, které větve mají větší pravděpodobnost, že budou provedeny

Hybridní dekompozice

5. hybridní dekompozice

- někdy je nutné použít více z předchozích technik pro dekompozici

Charakteristika prvotních tasků I.

Prvotní tasky lze charakterizovat (klasifikovat) podle následujících kritérií:

Charakteristika prvotních tasků II.

1.způsob generování prvotních tasků

- **staticky** - všechny tasky jsou známé před započetím výpočtu
  - datová dekompozice většinou vede ke statickým prvotním taskům
  - rekurzivní dekompozice může v některých případech také vést ke statickým prvotním taskum
    - nalezení minima v setříděném seznamu
  - dekompozice při prohledávání může vést na statické p. tasky
    - strom prohledáváme sekvečně tak dlouho, až získáme dostatečně velký počet větví
    - ty pak prohledáme paralelně - jejich počet je ale konstantní
- **dynamicky** - tasky vznikají až za chodu programu
  - např. rekurzivní dekompozice u quick-sortu vede k dynamickým prvotním taskum
  - dekompozice při prohledávání může vést na dynamické p. tasky
    - při prohledávání stromu vytváříme nový task při každém větvení

Charakteristika prvotních tasků III.

2. velikost prvotních tasků

- velikostí zde myslíme čas potřebný k proběhnutí celého tasku
- tasky mohou být:
  - **uniformní**
    - násobení plné matice s vektorem, kdy jeden task provede násobení vektoru s jedním řádkem matice
  - **neuniformní**
    - paralelizace algoritmu *quicksort*
    - task je dynamický, ale ve chvíli, kdy je vytvořen, dokážeme určit jeho složitost

Charakteristika prvotních tasků IV.

3. znalost velikosti prvotních tasků

- zde rozhoduje, zda je velikost tasku
- například u hry 15 apod. nedokážeme předem určit velikost tasku
- u násobení matic to lze naopak snadno

Charakteristika prvotních tasků V.

4. velikost dat spojených s taskem

- zde záleží hlavně na poměru vstupních, výstupních dat a náročnosti výpočtu
  - suma dlouhé řady
    - velký objem vstupních dat, tomu úměrný objem výpočtu a velmi malý objem výstupních dat
  - hra 15
    - malý objem vstupních dat, relativně malý objem výstupních dat, často neúměrně náročný výpočet
  - quicksort
    - objem vstupních a výstupních dat včetně složitosti výpočtu jsou přibližně stejné

Graf závislostí tasků

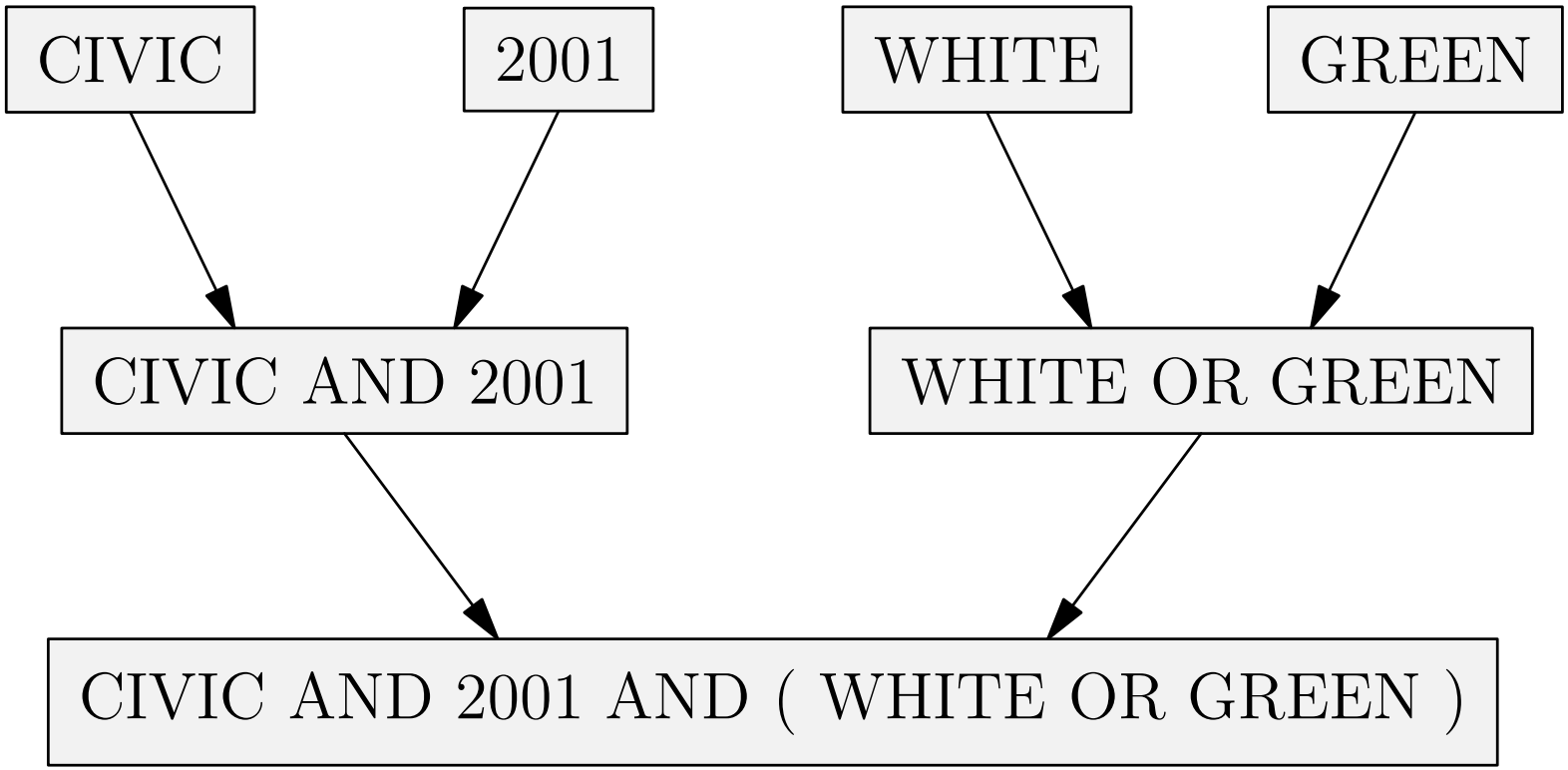
Task-dependency graph

- některé tasky mohou být spuštěny, až když jiné ukončily svou činnost
- kromě kominukace mezi tasky je toto další typ závislosti
- popisuje ho **graf závislosti tasků**
  - jde o orientovaný acyklický graf
  - uzly odpovídají jednotlivým prvotním taskům
  - uzly mohou být ohodnoceny podle množství výpočtů, jež je nutné provést k úplnému vyřešení tasku
  - task může být řešen, až když jsou vyřešeny všechny podproblémy, ze kterých do něj vede hrana
  - graf nemusí být souvislý
  - dokonce může být úplně bez hran

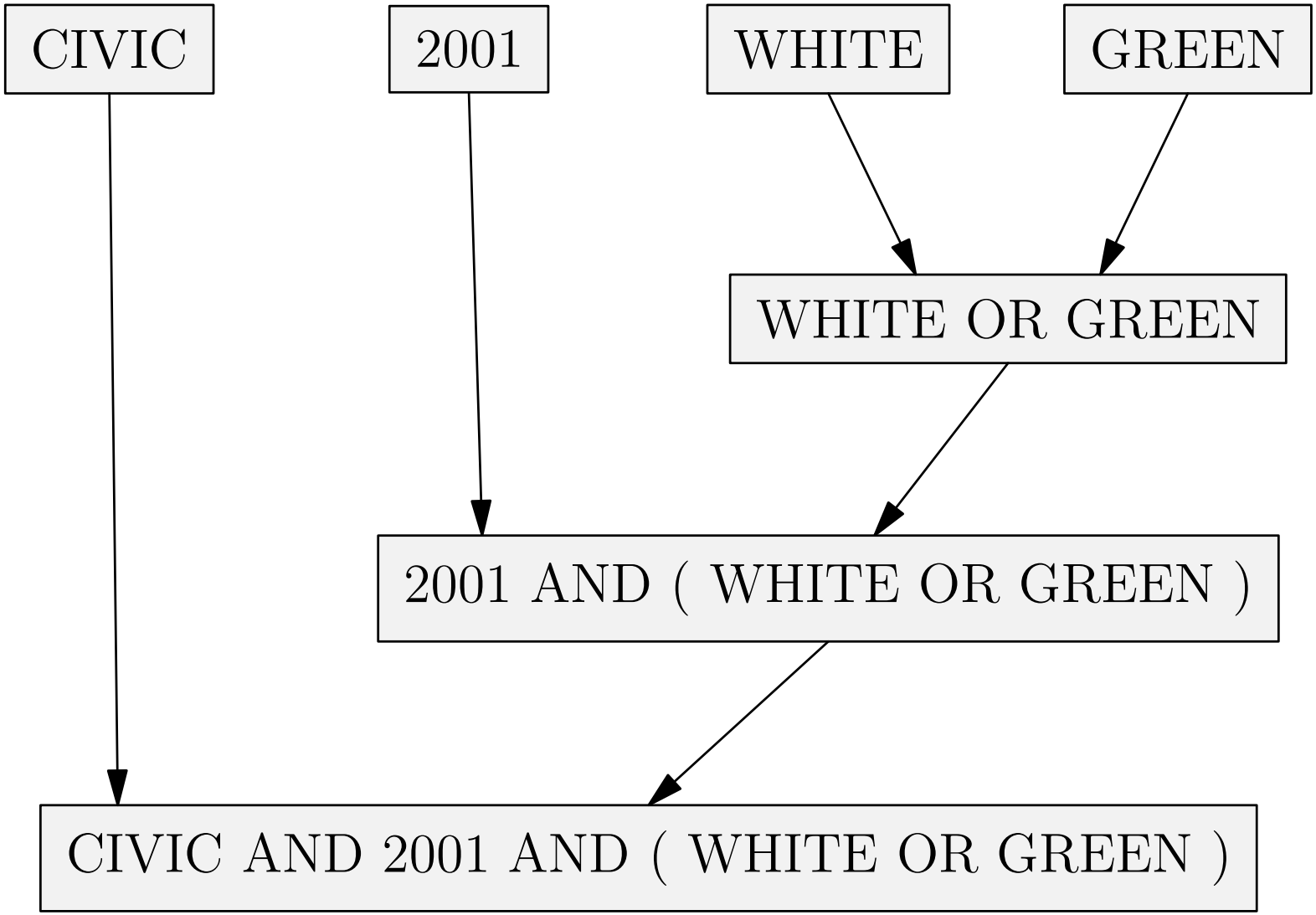
Graf závislostí tasků I.

Příklad: Chceme vyhodnotit následující SQL dotaz:

```
MODEL = "Civic"AND YEAR = "2001"AND ( COLOR =
"Green"OR COLOR = "White")
```



Graf závislostí tasků II.



Graf závislostí tasků III.

Podle grafu závislostí tasků lze popsat kvalitu navrhovaného paralelního algoritmu:

- **maximální stupeň souběžnosti** (*maximal degree of concurrency*)
  - udává, jaký je maximální počet tasků, jež mohou být zpracovány souběžně v libovolném stavu výpočtu
- **kritická cesta** (*critical path*)
  - je nejdelší cesta od některého počátečního k některému cílovému tasku
- **délka kritické cesty** (*critical path length*)
  - součet hodnot (udávajících náročnost tasku) uzlů podél kritické cesty
- **průměrný stupeň souběžnosti** (*average degree of concurrency*)
  - celková práce všech tasků/délka kritické cesty

Ohodnocení prvotních tasků

Dobře generované prvotní tasky by měly splňovat následující kritéria:

- mělo by jich být o jeden a více řádů více, než je počet procesorů
  - pokud to není splněno, jsme velmi omezeni v dalších krocích návrhu
  - může se stát, že nedokážeme efektivně obsadit všechny procesory
- redundantní výpočty a datové struktury jsou omezeny na minimum
  - není-li toto splněno, efektivita výsledného algoritmu může být nízká
  - může se snížit ještě více s rostoucí velikostí celé úlohy
- prvotní tasky by měly být přibližně stejně velké
  - pokud tomu tak není, může být problém rozdělit zátež rovnoměrně mezi všechny procesory
- počet prvotních tasku je rostoucí funkcí velikosti celé úlohy
  - pokud ne, může být problém s využitím velkého počtu procesorů pro velké úlohy

Komunikace

Po vytvoření prvotních tásků je nutné určit jejich způsob komunikace.
Existují dva způsoby komunikace:

- lokální komunikace
  - jeden task komunikuje s malým počtem jiných tasků
  - například výměna okrajových hodnot podoblastí při numerických výpočtech
- globální komunikace
  - komunikace, které se účastní velký počet tasku
    - často jsou to všechny
  - například může jít o výpočet sumy mezivýsledků z jednotlivých tasků

Komunikace výrazně příspívá k režijním nákladům paralelních algoritmů, protože u sekvečních se vůbec nevyskytuje.

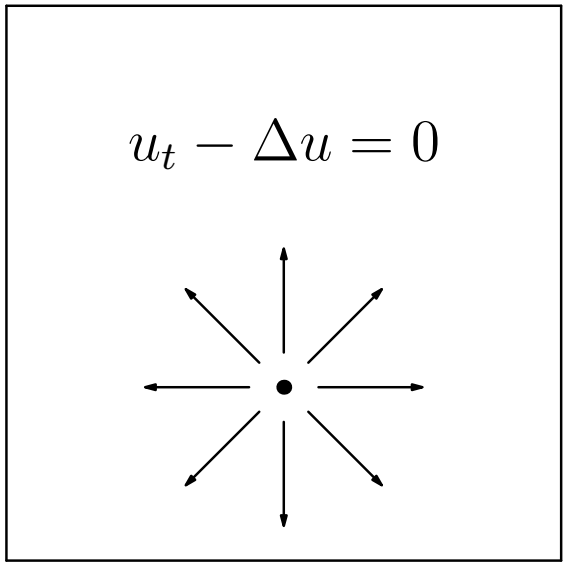
Graf interakcí I.

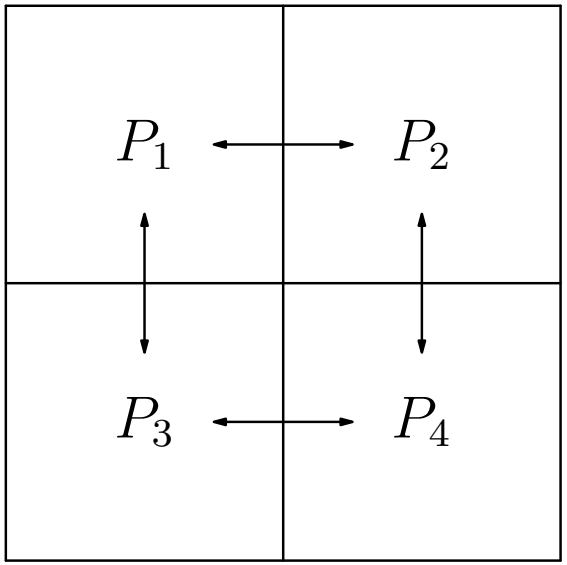
Komunikační vzor zachycuje tzv. **graf interakcí** (*task-interaction graph*).

- jeho vrcholy jsou tasky
- mezi dvěma vrcholy vede hrana, právě když spolu tyto dva tasky musí komunikovat
- graf závislostí je často podgrafem grafu interakcí

Graf interakcí II.

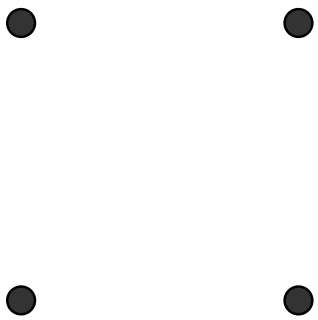
Rovnice vedení tepla

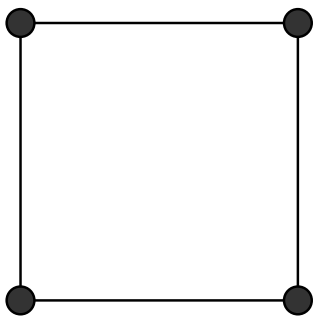




graf závislostí

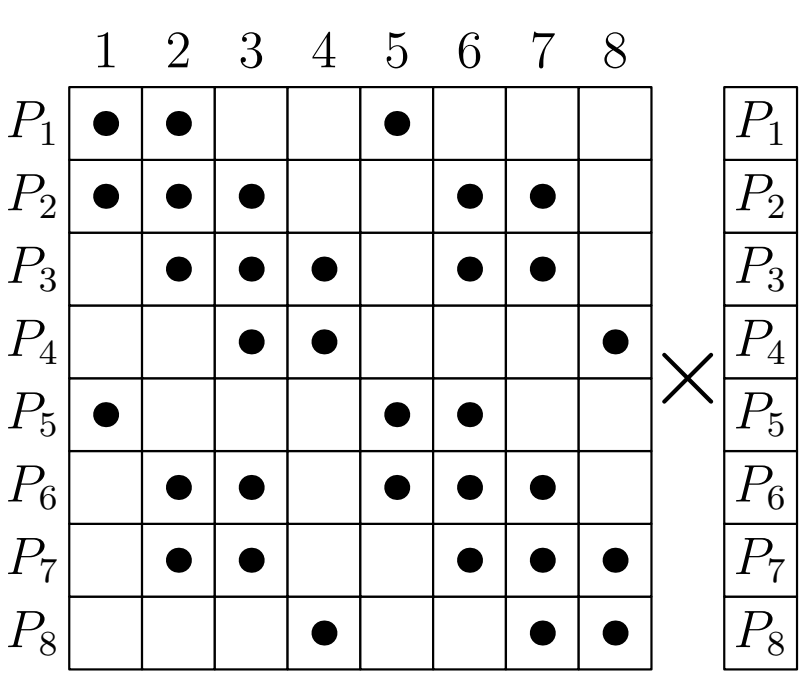
graf interakcí

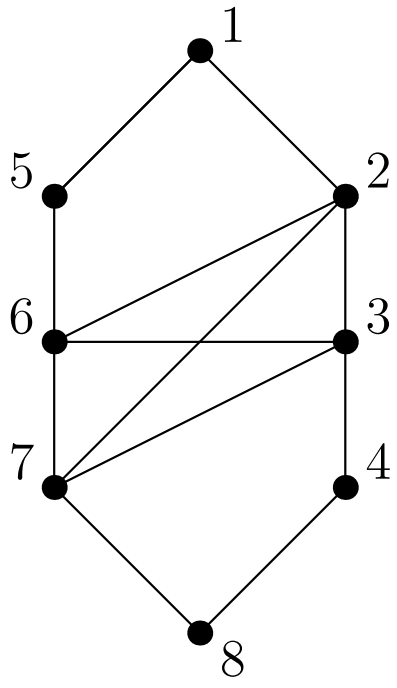




Graf interakcí III.

Násobení řídké matice a vektoru





Charakteristiky interakcí

Interakce lze dělit podle následujících kritérií:

- **statické a dynamické**
  - u statických interakcí se ví předem, kdy budou probíhat, je snažší je programovat
  - příklad dynamických je třeba hra 15
    - některé stavy mohou být prohledávány déle, než jiné
    - procesy, které mají již hotovou práci mohou převzít některé výpočty od ostatních
- **regulární a iregulární**
  - interakce mohou mít určitou strukturu
    - příklad regulárních interakcí - numerický výpočet na regulární síti
    - příklad iregulárních interakcí - násobení řídká matice krát vektor
- **interakce jen se čtením nebo i se zápisem**
  - to je důležité u systémů se sdílenou pamětí

Ohodnocení komunikace II.

Dobře navržená komunikace by měla splňovat:

- komunikační operace jsou dobře vybalancovány (rovnoměrně rozděleny) mezi všechny tasky
- tasky lze uspořádat do takové topologie, že každý task komunikuje jen s malým počtem sousedních tasků
- tasky mohou provádět komunikaci současně
- tasky mohou provádět výpočty současně

Aglomerace I.

- v prvním kroku návrhu paralelního algoritmu jsme se snažili o maximální paralelismus
- to většinou vede k příliš velkému počtu (primitivních) tasků
- jejich počet je často nutné zredukovat na počet vhodný pro danou paralelní architekturu
- aglomeraci lze určit podle grafu závislostí
  - menší tasky, které nemohou být zpracovány současně, je dobré spojit do jednoho většího
  - tím dochází k tzv. nárůstu lokality - (*increasing locality*)
- nebo podle způsobu komunikace mezi tasky
  - tasky, které se spojí do jednoho mezi sebou již nemusí komunikovat

Aglomerace II.

Dobře provedená aglomerace by měla splňovat:

- zvýší lokalitu výsledného paralelního algoritmu
- nově vytvořené tasky mají podobnou výpočetní a komunikační složitost
- počet tasků je rostoucí funkcí velikosti úlohy
- výsledný počet tasků je co nejmenší možný, ale větší nebo roven počtu procesorů cílové architektury
- náročnost úpravy sekvečního algoritmu na paralelní je přiměřená

Mapování

- jde o krok, kdy se p procesorům přidělují jednotlivé tasky
- u SMP architektur (se sdílenou pamětí) tuto práci obstarává operační systém
- snahou je maximalizace využití procesorů a minimalizace meziprocesorové komunikace
  - meziprocesorová komunikace roste, pokud dva tasky spojené channelem jsou mapovány na odlišné procesory a naopak
  - využití procesorů roste s počtem obsazených procesorů
- jde tedy o dva protichůdné požadavky
- mapujeme-li všechny tasky na jeden procesor, získáme minimální meziprocesorovou komunikaci, ale také minimální využití procesorů
- nalézt optimální řešení tohoto problému je NP-složitý (*NP-hard*) problém

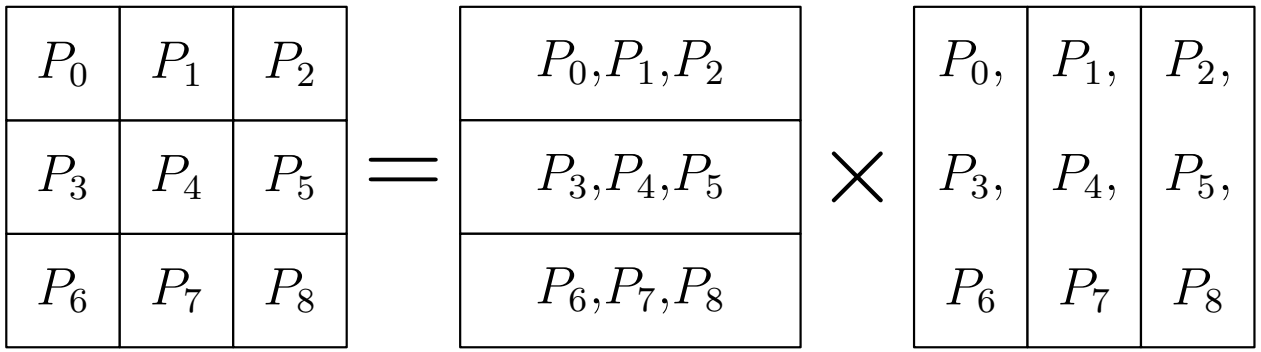
Mapování

- také se snažíme o vyvážené namapování tasků na procesory
  - jde o to, aby všechny procesory byly ideálně stejně vytížené
- způsoby mapování dělíme na
  - statické
  - dynamické

Statické mapování

1. Statické mapování

a. blokové mapování



Statické mapování

b. cyklické a blokově cyklické mapování

- používá se například u LU faktorizace
- zde se objem výpočtů liší pro různé prvky matice

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{bmatrix} \times \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix}$$

Statické mapování

Algoritmus pro LU rozklad:

```
for ( k = 1; k <= n; k ++ )
{
   // k-tý řádek dělíme pivotem
   for( j = k; j <= n; j ++ )
      A[ j ][ k ] /= A[ k ][ k ];

   // od řádků pod k-tým odečítáme j-tý
   for( j = k + 1; j <= n; j ++ )
      for( i = k + 1; i <= n; i ++ )
         A[ i ][ j ] -= A[ i ][ k ] * A[ k ][ j ];
}
```

Statické mapování

- blokové mapování by tu nebylo dobré
  - procesor s blokem v levém horním rohu by prováděl mnohem méně výpočtů než ten s pravým dolním blokem

P_0	P_1	P_0	P_1
P_2	P_3	P_2	P_3
P_0	P_1	P_0	P_1
P_2	P_3	P_2	P_3

Statické mapování

c. náhodné blokové mapování

- používá se tehdy, když problém nemá pevnou strukturu

Příklad: Mapování řádků řídké matice

- $V = \{0, 1, 2 \cdots 8\}$ - indexy řádků
- $random(V) = \{8, 2, 6, 0, 3, 7, 1, 5, 4\}$
- mapování - $\left\{ \underbrace{8, 2, 6}_{P_0}, \underbrace{0, 3, 7}_{P_1}, \underbrace{1, 5, 4}_{P_2} \right\}$

Statické mapování

d. mapování podle dělení grafů

- například když chceme rozdělit nestrukturovanou numerickou síť na podoblasti

Dynamické mapování

2. Dynamické mapování

- je vhodné tam, kde statické mapování rozděluje zátěž nerovnoměrně

a. centralizovaná schémata pro dynamické mapování

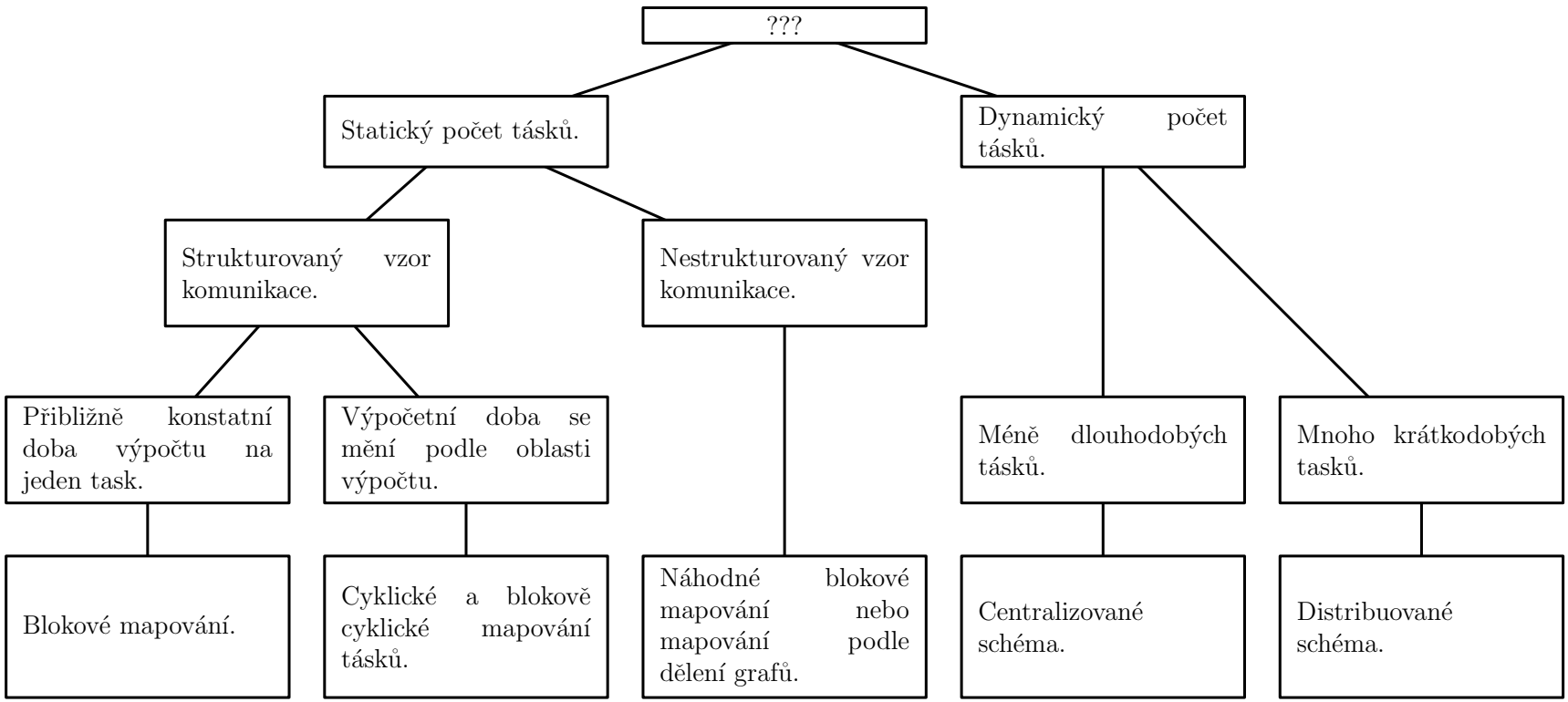
- spustí se speciální proces pro přidělování tasků
  - master/slave
  - producer/consumers
- existuje struktura, kam se ukládájí úlohy a odkud si je pracovní procesy berou
- pokud sem přistupuje hodně procesů, může docházet k velkým prodlevám
- můžeme se pokusit přidělovat větší úlohy a tím snížit počet přístupů
- s tím roste riziko nerovnoměrného rozdělení tasků

Dynamické mapování

b. distribuovaná schémata pro dynamické mapování

- tasky jsou nejprve rozděleny mezi procesy
- následně každý proces může poslat nebo přijmout určitý objem práce
- toto mapování je náročné na implementaci

Mapování - přehled



Možnosti snížení režie způsobené interakcemi

- maximalizace využití lokálních dat
  - tzv. *data locality*
  - při delších výpočtech se sdílenými daty je lepší vytvořit lokální kopii
- minimalizace objemu dat přenášených mezi procesy
  - tzv. *temporal locality*
  - někdy je lepší provést stejný výpočet na všech procesech, než jen na jednom a následně výsledek distribuovat mezi ostatní
- minimalizace četnosti interakcí
  - pokud to jde, slučujeme více interakcí (komunikačních operací) do jediné
- zabránit současnému přístupu více procesorů na jedno místo
  - to umožní provádět interakce současně

Možnosti snížení režie způsobené interakcemi

- současný běh interakce a výpočtu
  - nejprve provedeme výpočet s daty potřebnými pro interakci
  - následně se spustí interakce a provádí se výpočet na ostatních datech
  - to lze často využít při evolučních výpočtech na numerických sítích
    - celá síť se rozdělí na podoblasti
    - napočítáme novou časovou hladinou na okrajích podoblastí
    - hodnoty na okrajích je potřeba poslat sousedním procesům
    - spustíme interakci a současně napočítáváme novou časovou hladinu uvnitř podoblastí

Modely paralelních algoritmů

- datově paralelní
  - vyznačují se statickým mapováním
  - každý proces pracuje s různými daty
- úlohově paralelní
  - jde o algoritmy odvozené z grafu závislostí tasků
  - nebo algoritmy odvozené podle metody rozděl a panuj
- zásobník úloh
  - obsahuje centrální/distribuovanou strukturu s tasky pro jednotlivé procesy
  - tasky mohou být vytvářeny staticky na počátku nebo dynamicky za chodu
- master/slave
  - jeden nebo více procesů generuje tasky
  - ostatní procesy je provádějí
- pipelining
  - proud dat prochází od jednoho procesu ke druhému, a každý proces na nich provádí určitý dílčí výpočet
- hybridní úlohy

Analýza paralelních algoritmů I.

- použití dvou procesorů místo jednoho prakticky nikdy nevede k ukončení výpočtu v polovičním čase
- paralelizace s sebou vždy nese určitou režiji navíc:
  - interakce a komunikace mezi jednotlivými procesy
  - prostoje procesorů
    - nerovnoměrné rozdělení práce
    - čekání na ostatní procesy
  - některé výpočty navíc oproti sekvenčnímu algoritmu

Naší snahou nyní bude odvození teorie pro ohodnocení úspěšnosti paralelizace dané úlohy.
Poznámka: p opět označuje počet procesorů, které se účastní paralelního výpočtu a n je velikost řešené úlohy (podle vstupních dat).

Analýza paralelních algoritmů II.

Definition

Sériový (sekvenční) čas běhu algoritmu (*serial runtime*) - $T_S(n)$ - je doba mezi spuštěním a ukončením výpočtu sekvenčního algoritmu na úloze o velikosti n.

Definition

Paralelní čas běhu algoritmu (*parallel runtime*) - $T_P(n, p)$ - je doba mezi spuštěním algoritmu a okamžikem, kdy poslední proces ukončí svůj výpočet.

Analýza paralelních algoritmů II.

Definition

Sériový (sekvenční) čas běhu algoritmu (*serial runtime*) - $T_S(n)$ - je doba mezi spuštěním a ukončením výpočtu sekvenčního algoritmu na úloze o velikosti n.

Definition

Paralelní čas běhu algoritmu (*parallel runtime*) - $T_P(n, p)$ - je doba mezi spuštěním algoritmu a okamžikem, kdy poslední proces ukončí svůj výpočet.

Poznámka: Pokud porovnáváme sekvenční a paralelní čas, měříme sekvenční čas na nejrychlejším známém sekvečním algoritmu. Navíc požadujeme nejrychlejší známý sekveční algoritmus pro danou velikost úlohy, ne asymptoticky nejrychlejší sekvenční algoritmus.

Analýza paralelních algoritmů II.

Definition

Sériový (sekvenční) čas běhu algoritmu (*serial runtime*) - $T_S(n)$ - je doba mezi spuštěním a ukončením výpočtu sekvenčního algoritmu na úloze o velikosti n.

Definition

Paralelní čas běhu algoritmu (*parallel runtime*) - $T_P(n, p)$ - je doba mezi spuštěním algoritmu a okamžikem, kdy poslední proces ukončí svůj výpočet.

Poznámka: Pokud porovnáváme sekvenční a paralelní čas, měříme sekvenční čas na nejrychlejším známém sekvečním algoritmu. Navíc požadujeme nejrychlejší známý sekveční algoritmus pro danou velikost úlohy, ne asymptoticky nejrychlejší sekvenční algoritmus.

Analýza paralelních algoritmů III.

Definition

Čas čistě sekvenční části algoritmu (*time of inherently sequential part*) - $P_S(n)$ je doba, za kterou proběhne výpočet neparalelizovatelné části algoritmu.

Definition

Čas paralelizovatelné části algoritmu (*time of parallelisable part*) - $P_P(n)$ je doba, za kterou proběhne výpočet paralelizovatelné části algoritmu při sekvenčním zpracování.

- $P_P(n) = T_S(n) - P_S(n)$

Analýza paralelních algoritmů IV.

Definition

Celková režie (*total overhead*) - $T_O(n, p)$ - je definována jako

$$T_O(n, p) = pT_P(n, p) - T_S(n).$$

Analýza paralelních algoritmů IV.

Definition

Celková režie (*total overhead*) - $T_O(n,p)$ - je definována jako

$$T_O(n,p) = pT_P(n,p) - T_S(n).$$

Definition

Urychlení (*speedup*) - $S(n,p)$ - je definováno jako

$$S(n,p) = T_S(n)/T_P(n,p).$$

Analýza paralelních algoritmů V.

- platí, že $S(n,p) \leq p$
  - kdyby $S(n,p) > p$, pak by žádný procesor nesměl běžet déle než $T_S(n)/p$
  - potom bychom mohli vytvořit sekvenční algoritmus, který bude emulovat paralelní výpočet a dostaneme menší $T_S(n)$
- v praxi lze ale často pozorovat **superlineární urychlení**, kdy je $S(n,p) > p$
  - rozdělená úloha se může vejít do vyrovnávacích pamětí procesorů, datová komunikace je tak rychlejší
  - dekompozice při prohledávání
    - tím, že prohledáváme současně více větví stavového stromu, můžeme řešení najít dříve
    - sekvenčně lze toto napodobit prohledáváním stromu do šířky - to je ale težší k implementování

Analýza paralelních algoritmů VI.

Definition

Efektivita (*efficiency*) - $E(n)$ - je definována jako

$$E(n,p) = S(n,p)/p \leq 1.$$

- je-li $S(n,p)$ lineární funkcí vůči p, pak $E(n,p) = E(n)$, máme algoritmus s efektivitou nezávislou na počtu procesorů, což by byl ideální stav
- s rostoucím počtem procesorů efektivita ve většině případů klesá

Analýza paralelních algoritmů VI.

Definition

Náklady (*cost*) - $C(n, p)$ - jsou definovány jako

$$C(n, p) = pT_P(n, p).$$

Definition

Řekneme, že algoritmus je nákladově optimální, pokud je $C(n, p) = \Theta(T_S(n))$.

Analýza paralelních algoritmů VI.

Definition

Práce (*work*) - $W(n,p)$ - je definována jako

$$W(n,p) = \sum_{i=0}^{p-1} t_i,$$

kde t_i je čistý čas výpočtu i-tého procesoru.

Amdahlův zákon

$$\begin{aligned} S(n,p) &= \frac{T_S(n)}{T_P(n,p)} = \frac{P_S(n)+P_P(n)}{P_S(n)+P_P(n)/p+T_O(n,p)} \\ &\leq \frac{P_S(n)+P_P(n)}{P_S(n)+P_P(n)/p} \end{aligned}$$

Označme $f = P_S(n)/(P_S(n)+P_P(n))$ čistě sekvenční část algoritmu. Pak je

$$\begin{aligned} P_S(n)+P_P(n) &= \frac{P_S(n)}{f}, \\ 1/f-1 &= \frac{P_S(n)+P_P(n)}{P_S(n)}-1 = \frac{P_P(n)}{P_S(n)}, \\ (1/f-1)P_S(n) &= P_P(n). \end{aligned}$$

Amdahlův zákon

Dostáváme

$$
\begin{aligned}
S(n,p) &\leq \frac{\frac{P_S(n)}{f}}{P_S(n) + \left(\frac{1}{f} - 1\right)\frac{P_S(n)}{p}} && (1)\\
&= \frac{\frac{1}{f}}{1 + \left(\frac{1}{f} - 1\right)\frac{1}{p}} && (2)\\
&= \frac{\frac{1}{f}}{\frac{f + \frac{1-f}{p}}{f}} = \frac{1}{f + \frac{1-f}{p}} && (3)
\end{aligned}
$$

Amdahlův zákon

Theorem

Amdahlův zákon: *Buď* $0 \leq f \leq 1$ *část výpočtů, které musí být prováděny čistě sekvenčně. Maximální urychlení* $S(n, p)$ *dosažitelné při použití* p *procesorů je*

$$S(n, p) \leq \frac{1}{f + (1 - f)/p}.$$

- Amdahlův zákon je založen na předpokladu, že se snažíme vyřešit problém dané velikosti, jak nejrychleji to jde

Amdahlův zákon

Z Amdahlova zákona lze snadno získat asymptotický odhad pro urychleni $S(n, p)$

$$\lim_{p\to\infty} S(n,p) \leq \lim_{p\to\infty} \frac{1}{f + (1-f)/p} = \frac{1}{f}.$$

To znamená, že výpočet nemůžeme nikdy urychlit více než $1/f$-krát.

Amdahlův zákon

Příklad: Odhady říkají, že 90% našeho algoritmu lze paralelizovat a zbývajících 10% musí být zpracováno jen na jednom procesoru. Jakého urychlení dosáhneme při použití 8 procesorů?

$$S(n,p) \leq \frac{1}{0.1 + (1 - 0.1)/8} \approx 4.7$$

To je výrazně méně než požadované urychlení 8. Minimalně ze tří procesorů nemáme žádný užitek.

Amdahlův efekt

U rozumně navržených paralelních algoritmů platí, že paralelní režie ma asymptoticky nižší složitost než výpočet paralelizovatelné části

$$T_O(n,p) = o(P_P(n))$$

- navyšování velikosti výpočtu způsobí výraznější růst $P_P(n)$ než $T_O(n,p)$
- při pevném počtu procesorů p je urychlení $S(n,p)$ rostoucí funkcí proměnné n

Amdahlův efekt

- Amdahlův zákon zkoumá možnost maximálního urychlení výpočtu pevně dané úlohy
- výsledek Amdahlova zákona je dost pesimistický pro paralelizaci
  - pokud čistě sekvenční část algoritmu tvoří 10%, pak nikdy nedosáhnem většího než desetinásobného urychlení
- paralelizace tedy neumožňuje řešit zadanou úlohu v libovolně krátkém čase
- ale už z Amdahlova efektu vidíme, že přínos paralelizace je spíše v tom, že dokážeme řešit větší úlohy
- paralelizace tedy umožňuje provádět přesnější výpočty, kvalitnější vizualizaci apod.

Gustavsonův-Barsiho zákon

- nyní se tedy nebudeme snažit zkracovat čas výpočtu
- místo toho se pokusíme v daném čase výpočítat co největší úlohu
- u většiny paralelizovatelných úloh s rostoucí velikostí úlohy roste velikost čistě sekvenční části řádově pomaleji, než celková velikost

Víme, že

$$S(n,p) = \frac{P_S(n) + P_P(n)}{P_S(n) + P_P(n)/p + T_O(n,p)}.$$

Označme jako s časový podíl, který zabere zpracování čistě sekvenční části při paralelním výpočtu, tj.

$$s = \frac{P_S(n)}{P_S(n) + \frac{P_P(n)}{p} + T_O(n,p)}.$$

Gustavsonův-Barsiho zákon

Pak dostáváme

$$
\begin{aligned}
1 - s &= \frac{\frac{P_P(n)}{p} + T_O(n,p)}{P_S(n) + \frac{P_P(n)}{p} + T_O(n,p)}, \\
P_S(n) &= \left(P_S(n) + \frac{P_P(n)}{p} + T_O(n,p) \right) s, \\
P_P(n) &= \left(P_S(n) + \frac{P_P(n)}{p} + T_O(n,p) \right) (1-s)\, p - p T_O(n,p).
\end{aligned}
$$

A dále

$$
\begin{aligned}
S(n,p) &= \frac{P_S(n) + P_P(n)}{P_S(n) + P_P(n)/p + T_O(n,p)} \\
&= \frac{(P_S(n) + P_P(n)/p + T_O(n,p))\,(s + (1-s)\,p)}{P_S(n) + P_P(n)/p + T_O(n,p)} - \\
&\quad \frac{p T_O(n,p)}{P_S(n) + P_P(n)/p + T_O(n,p)}
\end{aligned}
$$

Gustavsonův-Barsiho zákon

Celkem tedy

$$S(n,p) = s + (1-s)p - \frac{pT_O(n,p)}{P_S(n) + P_P(n)/p + T_O(n,p)}.$$

Předpokládáme-li

$$T_O(n,p) = o(P_P(n)) \text{ a } P_S(n) = o(P_P(n)),$$

pak dostáváme

$$S(n,p) = s+(1-s)p-O(P_P(n)^{-1}) = p+(1-p)s-O(P_P(n)^{-1}).$$

Gustavsonův-Barsiho zákon

Theorem

Gustavsonův-Barsiho zákon: *Mějme paralelní program řešící problém velikosti n na p procesorech. Buď s část z celkového času výpočtu potřebná ke zpracování čistě sériové části výpočtu. Předpokládájme*

$$T_O(n,p) = o(P_P(n)) \text{ a } P_S(n) = o(P_P(n)),$$

Pro maximální dosažitelné urychlení pak platí

$$S(n,p) = p + (1-p)s - O(P_P(n)^{-1}).$$

Poznámka: Mnoho textů o paralelizaci uvádí jen $S(n,p) \leq p + (1-p)s$.

Gustavsonův-Barsiho zákon

- Amdahlův zákon vychází ze sekvenčního výpočtu a odvozuje, kolikrát rychlejší může být tento výpočet s využitím paralelizace
- Gustavsonův-Barsiho zákon vychází z paralelního výpočtu a vyvozuje, kolikrát déle by trval tento výpočet bez paralelizace
  - neber ale v úvahu superlineární urychlení, tedy fakt, že paralelní architektura může mít výhodu ve větším množství rychlejší paměti

Gustavsonův-Barsiho zákon

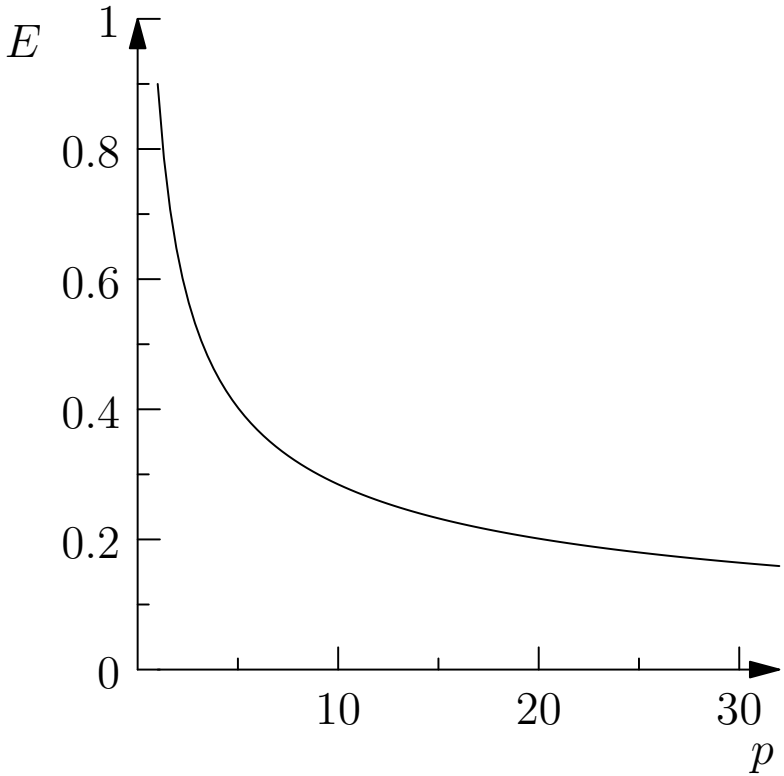
Příklad: Výpočet běžící na 64 procesorech trvá 220 sekund. Měření ukazuje, že 5% z celého času výpočtu zabere čistě sekvenční část algoritmu. Jakého urychlení bylo dosaženo?

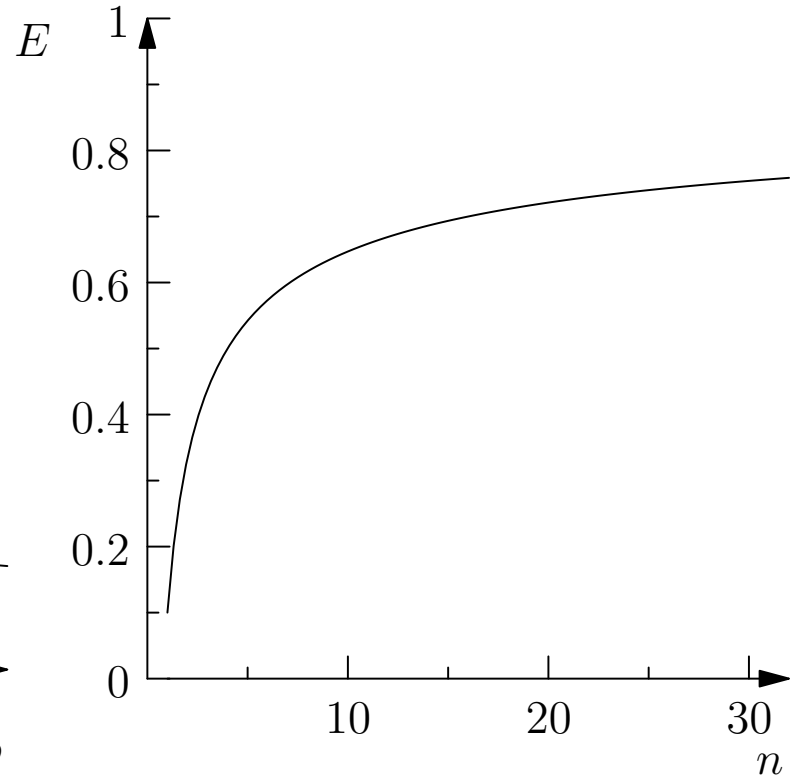
$$S(n,p) = 64 + (1 - 64) \cdot 0.05 = 64 - 3.15 = 60.85.$$

Efektivita a škálovatelnost

Závislost efektivity na

- rostoucím počtu procesorů p (plyne z Amdahlova zákona)
- a velikosti úlohy W (plyne z Gustavsonova-Barsiho zákona)





Škálovatelnost paralelních algoritmů

- jde o vlastnost paralelního algoritmu využít efektivně velký počet procesorů
- otázka je, o kolik musíme zvětšit danou úlohu, abychom po přídání určitého počtu procesorů zachovali zvolenou efektivitu
- čím méně je nutné velikost úlohy zvětšovat, tím lépe
- velikost úlohy nebudeme poměřovat velikostí vstupních dat n, ale pomocí $T_S(n)$

Škálovatelnost paralelních algoritmů

Platí

$$T_P(n,p) = \frac{T_S(n) + T_O(n,p)}{p},$$

a tedy

Škálovatelnost paralelních algoritmů

Platí

$$T_P(n,p) = \frac{T_S(n) + T_O(n,p)}{p},$$

a tedy

$$S(n,p) = \frac{T_S(n)}{T_P(n,p)} = \frac{T_S(n)p}{T_S(n) + T_O(n,p)}.$$

Škálovatelnost paralelních algoritmů

Platí

$$T_P(n,p) = \frac{T_S(n) + T_O(n,p)}{p},$$

a tedy

$$S(n,p) = \frac{T_S(n)}{T_P(n,p)} = \frac{T_S(n)p}{T_S(n) + T_O(n,p)}.$$

Potom

$$E(n) = \frac{S(n,p)}{p} = \frac{T_S(n)}{T_S(n) + T_O(n,p)} = \frac{1}{1 + \frac{T_O(n,p)}{T_S(n)}},$$

Škálovatelnost paralelních algoritmů

Platí

$$T_P(n,p) = \frac{T_S(n) + T_O(n,p)}{p},$$

a tedy

$$S(n,p) = \frac{T_S(n)}{T_P(n,p)} = \frac{T_S(n)p}{T_S(n) + T_O(n,p)}.$$

Potom

$$E(n) = \frac{S(n,p)}{p} = \frac{T_S(n)}{T_S(n) + T_O(n,p)} = \frac{1}{1 + \frac{T_O(n,p)}{T_S(n)}},$$

tedy

$$\frac{1}{E} = 1 + \frac{T_O(n,p)}{T_S(n)} \quad \Rightarrow \quad \frac{1-E}{E} = \frac{T_O(n,p)}{T_S(n)} \Rightarrow T_S(n,p) = \frac{ET_O(n,p)}{1-E}.$$

Škálovatelnost paralelních algoritmů

Definition

Funkce **izoefektivity** udává vztah mezi počtem procesorů p a velikostí úlohy n, kdy má paralelní algoritmus stejnou efektivitu.

Theorem

Označíme-li $K = \frac{E}{1-E}$ *konstantu určující požadovanou efektivitu, pak funkce izoefektivity je dána vztahem*

$$T_S(n) = KT_O(n,p) \Rightarrow n = T_S^{-1}(KT_O(n,p)).$$

Pozn.: Čím menší tato funkce je, tím lépe.

Nákladově optimálního algoritmus

Definition

Algoritmus je nákladově optimální, práve když platí

$$pT_P(n,p) = \theta(T_S(n)).$$

Platí

$$T_P(n.p) = \frac{T_S(n) + T_O(n,p)}{p}.$$

Tedy

$$pT_P(n,P) = T_S(n) + T_O(n,p) = \theta(T_S(n)) \Leftrightarrow T_O(n,p) = O(T_S(n)).$$

Nákladově optimálního algoritmus

Definition

Algoritmus je nákladově optimální, práve když platí

$$pT_P(n,p) = \theta(T_S(n)).$$

Platí

$$T_P(n.p) = \frac{T_S(n) + T_O(n,p)}{p}.$$

Tedy

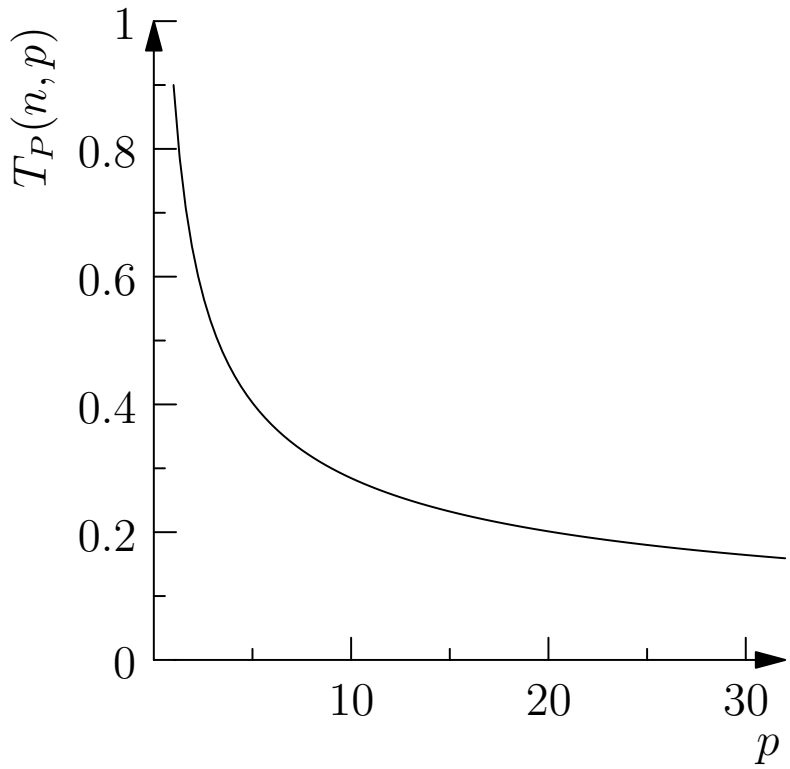
$$pT_P(n,P) = T_S(n) + T_O(n,p) = \theta(T_S(n)) \Leftrightarrow T_O(n,p) = O(T_S(n)).$$

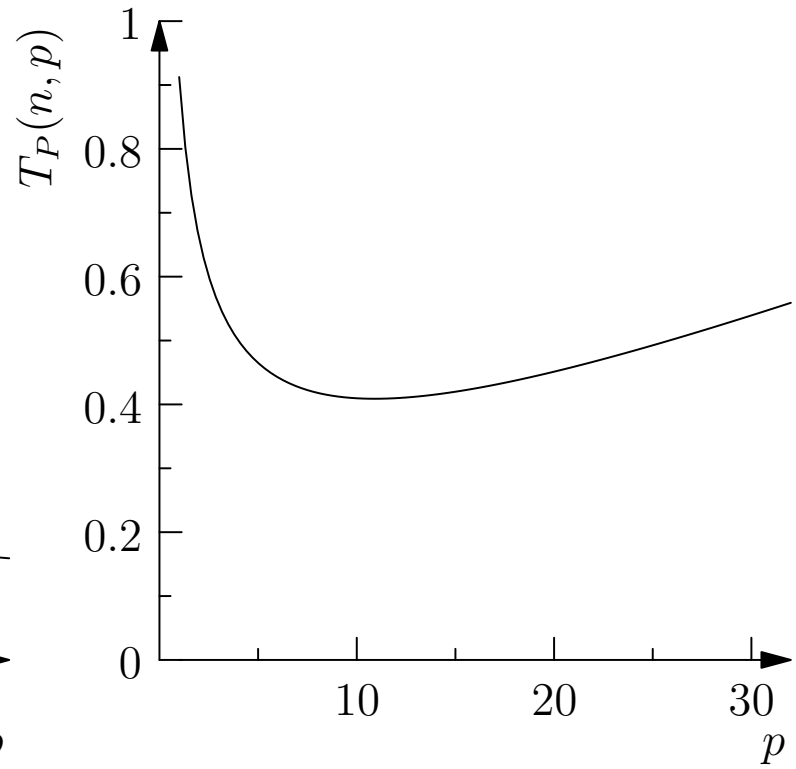
Theorem

Algoritmus je nákladově optimální, právě když paralelní režie nepřevyšuje řádově velikost úlohy.

Nejkratší a nákladově optimální nejkratší čas

Dvě možnosti, jak se může chovat paralelní čas s rostoucím počtem procesorů.





Nejkratší a nákladově optimální nejkratší čas

Hledáme minimum funkce paralelního času $T_P(n, p)$, tj.

$$\frac{d}{dp} T_P(n, p) = 0 \rightarrow T_P^{min}(n, p^*).$$

Problém je, že p^* může být příliš velké. Chceme najít p^* tak, aby byl výpočet nákladově optimální.

Nejkratší a nákladově optimální nejkratší čas

Musí tedy platit (pro pevně dané n)

$$T_S(\cdot) = \Omega(T_O(\cdot, p)),$$

tj.

$$p = O(T_O^{-1}(T_S(\cdot))).$$

Pro nákladově optimální algoritmus platí $pT_P(n,p) = \theta(T_S(n))$ tj.,

$$T_P(n,p) = \theta\left(\frac{T_S(n)}{p}\right).$$

Spolu s $p = O(T_O^{-1}(T_S(n)))$ dostáváme spodní odhad pro nákladově optimální nejkratší čas

$$T_P^{opt}(n,p) = \Omega\left(\frac{T_S(n)}{T_O^{-1}(T_S(n))}\right).$$

Modely ideálních paralelních architektur

- když analyzujeme složitost paralelních architektur, je často potřeba předpokládat určité vlastnosti paralelní architektury, pro kterou je navržený
- teoreticky se paralelní architektury popisují pomocí PRAM

Modely ideálních paralelních architektur

PRAM = Parallel Random Access Machine

- jde o model architektury se sdílenou pamětí
- stroj má p procesorů a globální paměť neomezené kapacity se stejně rychlým přístupem na jakoukoliv adresu pro všechny procesory
- modely PRAM se dělí podle ošetření přístupu více procesorů na stejnou adresu

Modely ideálních paralelních architektur

- **EREW PRAM** = Exclusive Read Exclusive Write PRAM
  - ani čtení ani zápis nelze provádět současně
  - je to nejslabší PRAM
- **CREW PRAM** = Concurrent Read Exclusive Write PRAM
  - možné současné čtení více procesorů z jedné adresy
  - to umí dnešní GPU
- **ERCW PRAM** = Exclusive Read Concurrent Write PRAM
  - umožňuje současný zápis
- **CRCW PRAM** = Concurrent Read Concurrent Write PRAM
  - umožňuje současné čtení i zápis

PRAM CW protokoly pro zápis

Současný zápis je možné ošetřit následujícími protokoly:

- **common** (obyčejný)
  - všechny zapisované hodnoty musí být stejné
- **arbitrary** (náhodný)
  - náhodně se vybere jeden proces, který zápis provede
  - u ostatních zápis selže
- **priority** (prioritní)
  - procesy mají dané priority, podle ktertých se určí, kdo provede zápis
- **sum** (součet)
  - zapíše se součet všech hodnot