ZÁPADOČESKÁ
UNIVERZITA
V PLZNI

Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

Semestrální práce z předmětu

Paralelní programování

Příklad: 5 – Hledání výskytů podřetězce

Jméno a příjmení: Pavel Gloss
Studijní číslo: A06082
Datum narození: 4.1.1984
Fakulta a obor: FAV – SWIN
Ročník: 5./6.
Datum: 14.ledna 2009
E-mail: pgloss@students.zcu.cz

Obsah

1 Zadání

Realizujte hledání všech výskytů daného podřetězce x1 v textovém souboru. Výstupem programu je seznam dvojic (řádek, pozice) nalezených výskytů.

Úlohu individuálně vypracujte ve dvou verzích:

1. Paralelní program pro systém se sdílenou pamětí (použitelné prostředky: prog. jazyk Java nebo vlákna POSIX).
2. Paralelní program pro systém s distribuovanou pamětí (použitelné prostředky: PVM nebo MPI) .

Povinné parametry:

- počet procesů / vláken (není nutné u MPI, tam se zadává počet vláken při spuštění prostředí)
- velikost kusu práce přidělované procesům (dynamické přidělování práce (Load-balance))
- řetězec x1
- název vstupního souboru
- bonus: přepínač case sensitive (rozlišování malých a velkých písmen)

Poznámky k řešení: není nutné řešit překrývání dvou nalezených podřetězců (plus body)

2 Analýza problému

Zadání můžeme řešit jako úlohu farmer-worker (viz přednášky):

Farmer:

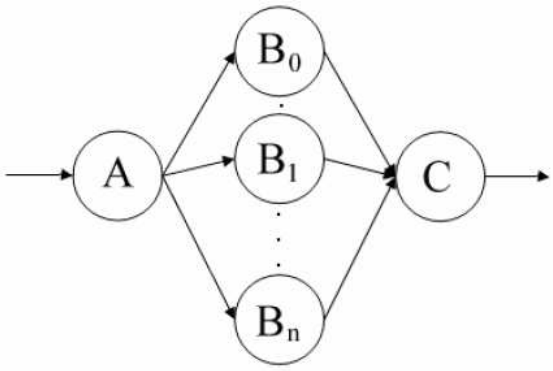
- načte vstupní data
- deleguje práci workerům

Dělníci:

- vykonají práci (tedy provádí hledání podřetězce na zadaném segmentu textu)
- výsledek odesílají zpět farmářovi

Postup zpracování lze rozdělit na následující kroky:

A - načtení souboru do paměti
B - nalezení pozic hledaného řetězce na přiděleném úseku práce (segmentu textu)
C - zkompletování všech výskytů



Je tedy vhodné paralelizovat druhý krok. Proces můžeme vyjádřit uvedeným precedenčním grafem.

2.1 Segmenty práce

Segment práce, kterou budou dělníci najednou vykonávat, se musí dopředu určit. To udělá farmář ze znalosti počtu vláken a velikosti práce. Protože jako data se načítají řádky textu, pak velikost práce bude znamenat, kolik řádků zvládne dělník prohledat v jednom pracovním cyklu. Podle toho farmář přidělí jednotlivé segmenty práce (skupiny řádků).

2.2 Prostředky k řešení

Úlohu jsem řešil pro dva systémy:

- Paralelní program pro systém s distribuovanou pamětí za pomoci prostředku PVM (v jazyku C)
- Paralelní program pro systém se sdílenou pamětí řešený v Javě

3 PVM

PVM je programový systém, umožňující vytváření a provozování paralelních programů založených na modelu předávání zpráv (message passing). V tomto obecném modelu se interakce paralelně běžících procesů uskutečňuje prostřednictvím posílaných zpráv, takže model je realizovatelný i na architekturách bez sdílené paměti (na tzv. volně vázaných systémech – např. počítače zapojené do sítě). K významným rysům PVM náleží podpora heterogenního výpočetního prostředí, obsahujícího počítače s různou technickou architekturou, výkonností, formátem dat apod., jako tomu je běžně u lokálních počítačových sítí.

PVM je volně šiřitelný a dostupný pro unixovské platformy (jsou buildy i pro Windows). Jeho masové rozšíření z něho dělá faktický standard, třebaže v poslední době se musí o popularitu dělit se standardem MPI (Messagge Passing Interface).

3.1 Farmer

Hlavní aplikace, která vytvoří všechny potřebné dělníky (spouští je jako samostatné procesy), načte textový soubor a začne přidělovat dělníkům práci. Pokud nějaký dělník vrátí výsledek, je dělníkovi okamžitě přidělena další část práce a to tak dlouho, dokud se neprojde celé pole s prací. Po skončení hledání vyšle farmer zprávu multicastem všem dělníkům, aby se ukončili, a sám uloží výsledek a ukončí se.

3.2 Worker

Worker je samostatný program, ale je volán z prostředí PVM resp. z jiného programu (farmáře) pomocí pvm_spawn. Vykonává přidělenou práci. Dělník obsahuje nekonečnou smyčku ve které přijímá zprávy. Pokud přijatá zpráva znamená práci, načte si všechna data z přijímacího zásobníku a přijaté řádky textu zpracuje a výsledek pošle zpátky modulu farmer. Pokud zpráva je typu QUIT přeruší se nekonečná smyčka a dělník se ukončí.

3.3 Překlad a spuštění

Ke zdrojovým kódům je připojen soubor **makefile**, který pomůže s překladem. Dále je přiložen soubor **hostfile** s konfigurací uzlů clusteru Hydra.

Nejprve spustíme PVM daemona posloupností příkazů

- pvmd hostfile
- CTRL+Z
- bg

V adresáři se zdrojáky přeložíme pomocí make oba moduly

- **make EXE=farmer**
- **make EXE=worker**

Nebo je jednodušší použít připravenou dávku **mk** , ale je nutné pomocí chmod 777 nastavit práva na spuštění.

Nyní máme dva spustitelné soubory – farmer.exe a worker.exe.
Přeložený program lze (i opakovaně) spustit příkazem **./farmer.exe**

Program vypíše nápovědu, jaké parametry se mají zadat a ukončí se.

Před ukončením práce je nutné ukončit PVM daemona - spustit ovládací konzoli příkazem pvm a v ní napsat příkazy:

reset - ukončí všechny zbylé procesy
halt - ukončí samotného PVM daemona

4 Java

Paralelní procesy jsou vytvořena jako vlákna komunikující přes sdílenou paměť. Sdílenou paměť představuje třída s kritickou sekcí chráněnou monitorem.

4.1 Formát vstupních dat

Aplikace akceptuje libovolný textový soubor. Je však třeba si uvědomit, že jako jednotka práce slouží x řádků textu najednou. Proto je ideální textový soubor s více řádky, ideálně zhruba stejně dlouhými (délka však není nijak omezena).
Druhou možností je použít vestavěný LoremIpsum generátor a nechat si vygenerovat řádky textu automaticky.

4.2 Programátorská dokumentace

Byla vygenerována pomocí **javadoc**. Je přiložena v adresáři **doc**

4.3 Řešení

Program pracuje na principu farmer/worker s dynamickým přidělováním práce.

Volání výpočtu probíhá z GUI. Nejprve se načtou vstupní data ze souboru (textový soubor s jednotlivými řádkami textu) anebo se vygenerují Lorem ipsum generátorem (třída **LoremIpsum**). Načtená data představuje objekt třídy **Data**, který obsahuje seznamy řádků. Také je volána metoda **processFind**(int odRadku, int doRadku, ArrayList<Integer> polePozic[]) z jednotlivých dělníků, která hledá podřetězec this.needle v sdílených datech na zadaných indexech a ukládá výsledky do polePozic.

Ve třídě **Task** se vytvoří fronta s jednotlivými segmenty práce, kterou si rozebírají dělníci. Segment práce je reprezentován třídou **WorkPiece** a obsahuje jen dva indexy – od jakého řádku do jakého řádku se má provádět hledání (výpočet). To nám stačí, protože se těmito indexy odkazujeme do sdílených dat načteného textu.

Metodou **process**(int threads, int praceRadku, Data data, ArrayList<Integer> polePozic[]) ve třídě Task spustíme paralelní výpočet se zadaným počtem vláken a zadanou jednotkou práce tj. počet zpracovávaných řádků. Nastartují se všechna vlákna a počká, dokud všechna nedoběhnou. Každý dělník přidává na svoje zpracovávané indexy nalezené pozice. Nemůže dojít ke kolizi, protože indexy byly přiděleny jako segmenty práce metodou **prepareWork(..)** a to podle počtu vláken a počtu řádků jako jednotky práce. Do pole **polePozic** se na jednotlivé indexy, odpovídající číslování řádků, přidávají do seznamů jednotlivé nalezené pozice.

Třída **ThreadCounter** hlídá, jestli všechny vlákna už skončila a blokuje hlavní vlákno do té doby, než ostatní skončí. Dělníka představuje třída **WorkingThread**, která běží, dokud není fronta prázdná. Pokud je fronta prázdná, zavolá ThreadCounter, že skončila a ukončí se. Pokud ve frontě je ještě nějaká práce, vezme si ji a zpracuje daný kus práce a to se celé opakuje, dokud fronta není prázdná.

4.4 Spuštění a ovládání

Program byl testován pod Windows na Java 1.6.0_04.

Javovské třídy zkompilujete příkazem javac *.java
nebo připraveným skriptem **compile.bat**

Program se spustí pomocí **run.bat** a spustí se grafické okno - **GUI**.

Nejprve musíme zadat vstupní text, ve kterém se bude prohledávat. Můžeme ho načíst ze souboru (libovolný textový soubor) anebo si nechat **vygenerovat dummy text Lorem ipsum** generátorem (wow!). Tak či onak, vstupní text se zobrazí v okénku „prohledavany text“.
Dále zadáme hledaný podřetězec a zaškrtneme, zda chceme brát ohled na velká a malá písmena (case sensitive).
Nyní nastavíme počet vláken (musí být > 1) a jednotku práce (neboli počet řádků zpracovaných jako jeden segment práce). Jednotka práce musí být > 1.

Pro zahájení výpočtu klikneme na tlačítko **Pracuj**. Pokud jsou parametry špatně zadány, je vypsána příslušná hláška a výpočet nemůže být zahájen. Proběhne paralelní výpočet a v okně „**výsledky**“ je nejprve zobrazena **doba trvání** paralelního výpočtu v milisekundách a dále nalezené pozice hledaného podřetězce. Čísla řádků i čísla pozic jsou indexovány od nuly.

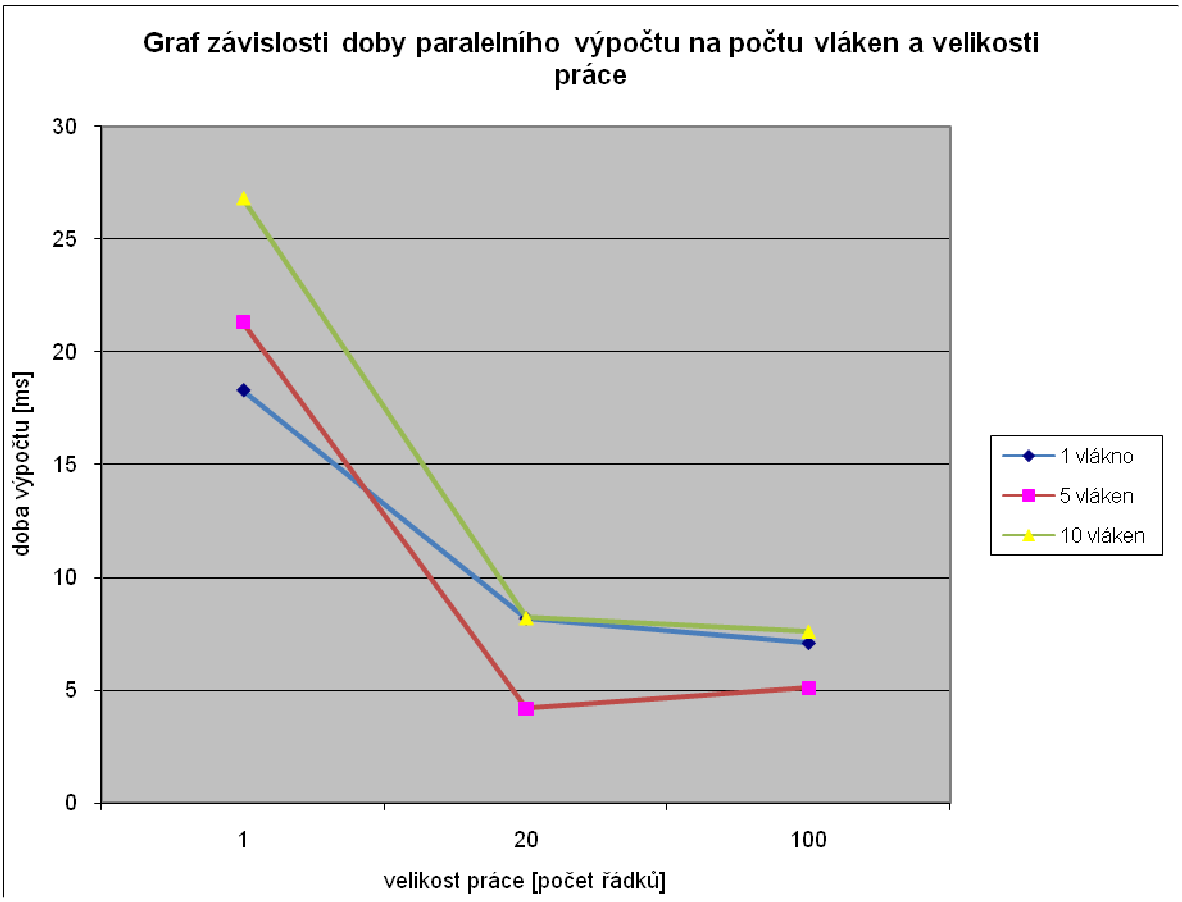
Příklad výsledku vypadá takto:

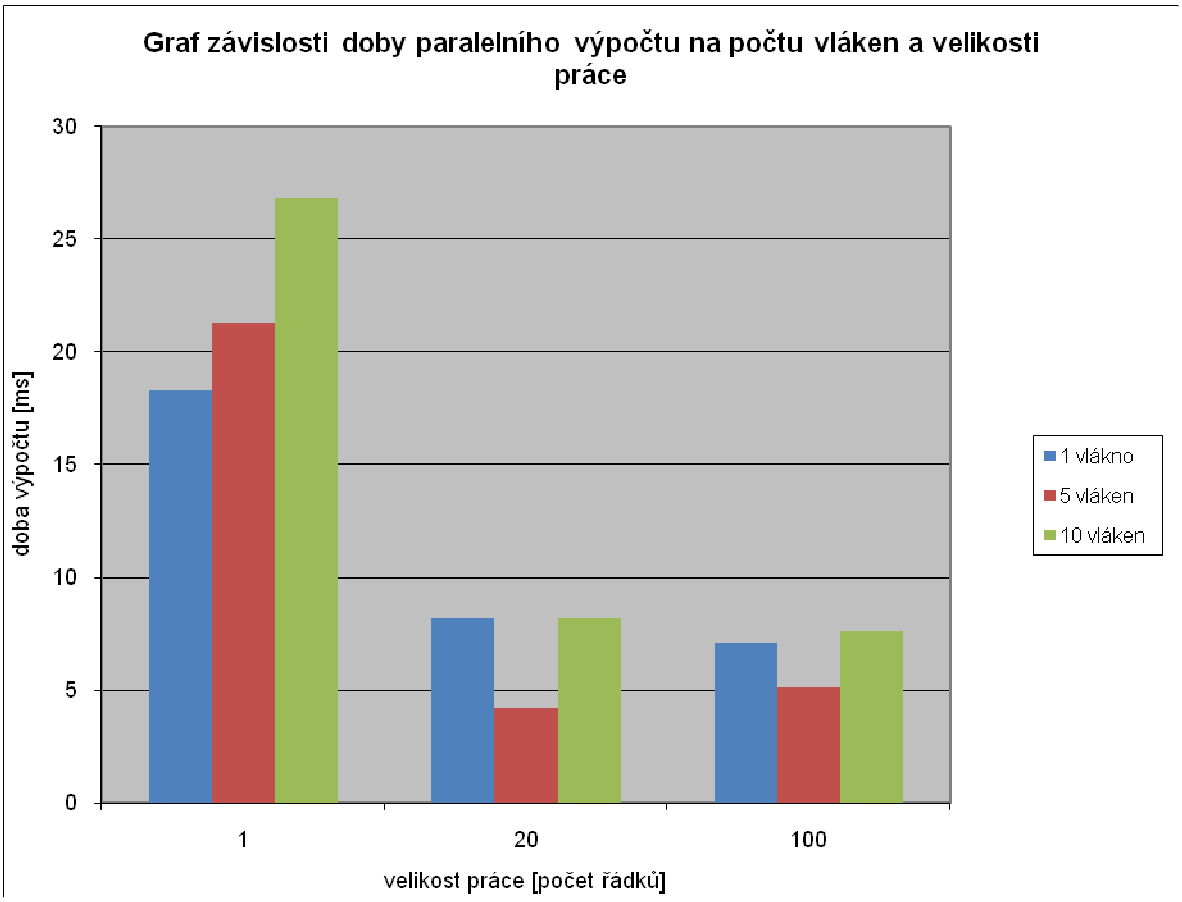
```
Doba vypoctu: 2.504509 ms

Vypisuji vysledky:
radek[0] --> 22,
radek[1] --> 11, 34, 71,
radek[2] --> 1, 5, 49, 68,
```

4.5 Závislost doby paralelního výpočtu

Pokusil jsem se změřit rychlosti pro různý počet vláken a různě velkou zadanou práci. Načítal jsem textový soubor o 4100 řádcích, každá řádka 75znaků. Výpočet probíhal na dvoujádrovém procesoru Intel Core Duo.
Udivilo mě, jak moc se při stejném spuštění a stejných parametrech liší časy výpočtu. Zde hodně hraje roli režie JVM. Proto jsem naměřil soubor hodnot, vyškrtal extrémy a zprůměroval zbylé hodnoty.

Počet vláken	Jednotka práce (počet řádek)	Doba výpočtu [ms]
1	1	18.3
1	20	8.2
1	100	7.1
5	1	21,3
5	20	4.2
5	100	5.1
10	1	26,8
10	20	8.2
10	100	7.6


Graf závislosti doby paralelního výpočtu na počtu vláken a velikosti práce
doba výpočtu [ms]
30
25
20
15
10
5
0
1
20
100
velikost práce [počet řádků]
1 vlákno
5 vláken
10 vláken



Z grafů je vidět, že více vláken přináší jisté urychlení. Bohužel počítaný výpočet je výpočetně zanedbatelný, takže zde hraje velkou roli i režie při vytváření a přepínání vláken.
Je však možné tvrdit, že pokud přidělujeme práci po větších segmentech (více řádků najednou), tak se výpočet urychluje.

5 Bonusy

PVM

- Překrývání podřetězců
- Měření času výpočtu

Java

- GUI
- Překrývání podřetězců (elegantně vyřešeno přidělováním po řádcích)
- Parametr case sensitive (rozlišovat velká a malá písmena)
- Vygenerovaná JavaDoc
- Načítání ze souboru / Lorem Ipsum generátor
- Měření doby výpočtu
- Závislost doby výpočtu pěkně zpracovaná v dokumentaci

6 Závěr

Oba programy pracují 100% > podle zadání.

Nejvíce času jsem strávil programováním v céčku pro PVM. Pracoval jsem v primitivním vývojovém nástroji Dev-C++, protože jsem neměl místo pro dokonalý nástroj od Microsoftu - Visual Studio .NET. Jazyk C opravdu nemám rád, protože mě nebaví půl dne hledat chyby v nealokované paměti apod ☹. Dále mám výtku k samotnému PVM a překládání na linuxovém clusteru.

Dva dny jsem hledal chybu při předávání zpráv. Farmář poslal práci dělníkům, a dělník poslal výsledek. Na straně farmáře jsem však přijímal úplně jiné údaje, než jsem odesílal z dělníka. Chybu jsem hledal v kódu. Ten však byl správně. Pak jsem zjistil, že v adresáři $HOME/pvm mám starší vývojovou větev programu a že už pracuji na novější v adresáři $HOME/pvm/nove. Podle cesty v hostfile se však z nové verze farmáře (pvm) spouštěly procesy starších verzí dělníků (pvm/nove) a jaksi si úplně neposílali kompatibilní zprávy. Tahle chyba mě dostala…

Nejsem fanda linuxu ani příkazové řádky. Navíc vzdálené připojení přes SSH z domova s pomalým internetem není to pravé ořechové pro komfortní práci…
Programovat naslepo, bez kontroly kódu a pak jednou za čas přeložit na Hydře a zjistit hromadu chyb mě brzy přestalo bavit. Stáhnul jsem si proto hlavičkový soubor „**pvm3.h**“ a už mi fungovala alespoň kontrola kódu. Dále mě vadilo na PVM, že na straně Workera (tedy proces spuštěný z PVM pomocí pvm_spawn(..)) nešlo vypisovat do konzole – potřeboval jsem kontrolní výpisy (to už vůbec nemluvím o nějakém debugování…).

To jazyk Java je jiné kafe (☺). V javě programuji rychle a efektivně a mohu se soustředit na podstatu problému. Když něco nefunguje jak má, spustím skvělý debugger. Pracuji v IDE Eclipse a Netbeans. Obě mají něco do sebe. Základ byl vytvořen v Eclipse a pak import do Netbeans a vytvoření GUI.

Pro výpočet ve vláknech v Javě jsem se pokusil i naměřit **závislost doby trvání výpočtu** na počtu vláken a velikosti přidělené práce.

Celkově byla práce poměrně obtížná.