Seminární práce do PDS

Hledání počtu výskytu slov

Christian Krutsche
KRU0144

Obsah

1 Úvod

Dnešní programovací a skriptovací jazyky umožňují jednoduchou práci s řetězci. Potřebujeme-li v textu něco najít nebo počet výskytu tohoto slova, moc práce nám to nezabere. Cílem této práce je vytvořit program, který v textu najde nejčetnější slova. Kromě toho bude schopen pracovat na více vláknech současně. V experimentální části si pak zkusíme porovnat zda-li je tento problém užitečné paralelizovat.

2 Obecný postup řešení problému

2.1 Nejideálnejší varianta procházení polem

Pokud bychom hledali v textu,kde všechna slova jsou oddělená mezerou, pak by stačilo rozdělit řetězec na místech mezer a uložit do pole. Následně procházet pole slovo po slově, a jestliže se slovo shoduje s hledaným, pak zvýšit počet výskytu. V praxi však tento přístup bude horší. Pro využití tohoto jednoduchého způsobu je potřeba mít text uzpůsobený (žádné jiné znaky, jen slova a mezery) a nemůžeme hledat žádný delší řetězec.

2.2 Důkladné porovnávání znaků

Další možností by bylo procházet text znak po znaku. V případě, kdy by se aktuální znak shodoval s prvním znakem hledaného slova, bychom začali porovnávat další znaky dokud by se nepotvrdila nebo neporušila rovnost části textu a hledaného slova. Tato metoda je náročnější pro vypočetní výkon, ale je přesná nezávisle na charakteru textu a hledaného slova(v tomto případě se může jednat i o věty). Případné řešení tohoto problému by se mohly lišit v krajních případech jakým může být například tento:

```
"Phasellus sagitititititititit nisi suscipit sem"
```

Hledejme slovo

```
"itit"
```

Po chvíli přemýšlení nám jistě dojde, že výsledek může být jak 3 tak 6 v závislosti na tom, jestli pokračujeme v hledání od následujícího znaku začátku nalezeného výskytu nebo od následujícího znaku konce nalezeného výskytu.

2.3 Nalezení mapy počtu výskytu pro každé slovo

Pravděpodobně je tento přístup nejnepravděpodobnější pro hledání počtu výskytu slova v textu. Jeho hlavní nevýhodou bude náročnost výpočtu, protože procházíme a počítáme každé slovo v textu. Dalším důvodem je problém stejný jak u prvního přístupu, je potřeba mít určitý charakter hledaného slova a vstupního textu pro úspěch. Celý výsledek je ukládán do tzv. **Mapy**, což je datová struktura, která umožní uchovat pro různé klíče různé hodnoty. V tomto případě se volí jako klíč slovo z textu a jako hodnota jeho počet výskytu. Pro zjištění počtu výskytu našeho slova v textu pak stačí přistoupit k oné Mapě a podívat se na hodnotu pod klíčem, kterým je naše slovo.

3 Popis algoritmu

V našem případě jsme si vybrali 3-tí přístup, protože chceme pořádně otestovat přínos paralelizace, a proto potřebujeme výpočetně náročný problém. Program je napsaný v jazyce **Go**, jehož nezanedbatelnými výhodami oproti konkurenčním jazykům je nativní podpora paralelizace. Dalším pozivitem je fakt, že jedno vlákno si veme 4KB operační paměti. Naproti tomu např. Java potřebuje celý 1MB pro jedno vlákno. [1] Proto pro dobře napsaný program v **Go** nebude velikost operační pamětí brzdou a takový program je schopen vytvořit tisíce paralelních vláken (tzv. goroutin).
Definujme si oddělovač jako jakýkoliv nealfanumerický znak (tj. znak, který není uvnitř slova, ale na jeho okraji). Příkladem takového oddělovače může být mezera, čárka, pomlčka, ale i třeba hvězdička.
Pro snadné pochopení algoritmu je kód zjednodušen. Pro podrobnosti nahlédněte do přiloženého programu.
Než začneme tvořit Mapu výskytů, rozdělíme si celý text tolik menších částí, kolik máme vláken. Je zřejmé, že tyto menší části nebudou přesně stejně velké, protože nechceme riskovat rozdělení nějakého slova.

```
for i := 0; i < numOfThreads; i++ {
    pivot = (i + 1) * partLength
    for pivot < len(text) && !isDivider(text[pivot]) {
        pivot++
    }
    go countOccurrencies(text[lastPivot:pivot], wordCountsChannel)
    lastPivot = pivot + 1
}
```

Nad těmito částmi pak spustím v jednotlivých vláknech funkce, které text projdou a pro každé slovo inkrementují jeho výskyt v mapě.

```
for i := 0; i < len(text); i++ {
    if isDivider(text[i]) && !isDivider(text[i+1]) {
        start = i + 1
    } else if !isDivider(text[i]) && isDivider(text[i+1]) {
        end = i + 1
        words = append(words, text[start:end])
    }
}
for i := 0; i < len(words); i++ {
    wordCounts[words[i]] = wordCounts[words[i]] + 1
}
```

Tyto mapy pak musíme složit do jedné velké.

```
for word, count := range wordCount {
    if word == "words_length" {
        allWordsLength += count
    } else {
        totalWordCounts[word] = totalWordCounts[word] + count
    }
}
```

Z výsledné mapy si pak můžeme vytáhnout četnost výskytu libovolného slova z textu. V programu je i seřazení těchto počtů pro jednotlivá slova sestupně, tj. výpis určité statistiky pro text. Pro řešení jsem si vytvořil pole slov (pouze jsem převzal klíče z mapy výskytů) a pro seřazení jsem použil paralelně Selection sort algoritmus nad rozdělenými poli slov. Pro spojení těchto polí jsem použil Merge sort. Dalo by se vše udělat Merge sortem, ale chtěl jsem zachovat počet vláken nezávislý na velikosti pole.
Pro použití je potřeba text, který je bez diakritiky. U hledáného slova se nerozlišuje velikost písmen, protože celý text i hledané slovo jsou nejprve převedeny do formátu s malými písmeny.
Pro první použití programu je vhodné začít příkazem zobrazujícím nápovědu:

```
go run main.go --help
```

4 Testování přínosu paralelizace

Pro testování programu jsem se rozhodl použít Bibli, hlavně protože je to opravdu dlouhý text, který není náhodně generován a je relativně přístupný.

Porovnejme si dobu, za kterou byl schopen program vytvořit mapu v závislosti na počtu vláken.

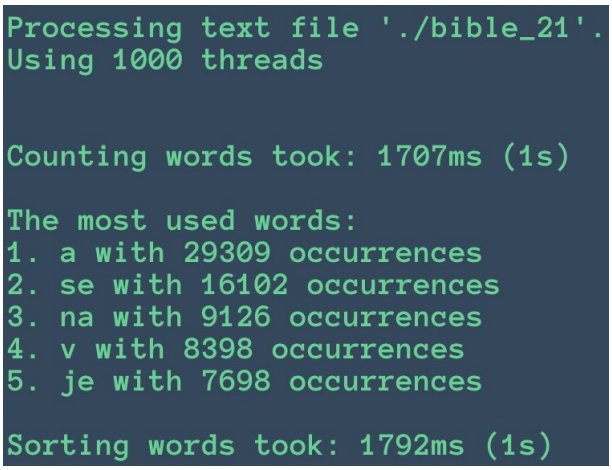
```
Processing text file './bible_21'.
Using 1 thread


Counting words took: 607635ms (10m 7s)

The most used words:
1. a with 29314 occurences
2. se with 16104 occurences
3. na with 9128 occurences
4. v with 8404 occurences
5. je with 7704 occurences

Sorting words took: 98674ms (1m 38s)
```

Obrázek 1: 1 vlákno

```
Processing text file './bible_21'.
Using 8 threads


Counting words took: 48243ms (48s)

The most used words:
1. a with 29313 occurrences
2. se with 16104 occurrences
3. na with 9128 occurrences
4. v with 8404 occurrences
5. je with 7704 occurrences

Sorting words took: 2536ms (2s)
```

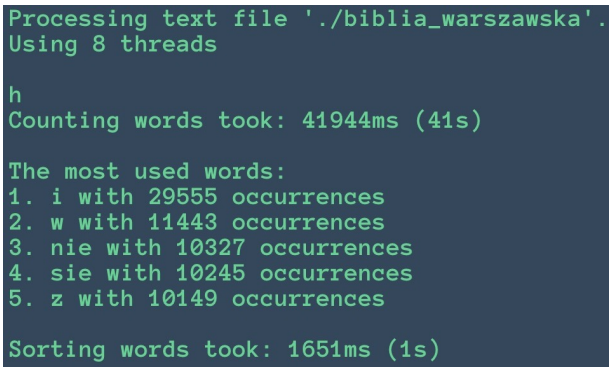
Obrázek 2: 8 vláken

Můžeme vidět, že časový rozdíl je markantní. Asi jsme nepovšimli fakt, že v žebříčku počtu výskytu slov se výsledky lehce liší. Je to pravděpodobně zapříčiněno špatným rozdělením slov. Při textu takového rozsahu je dost možné, že hraniční index dopadl do nějakého neošetřeného případu, čímž se slovo ztratilo.

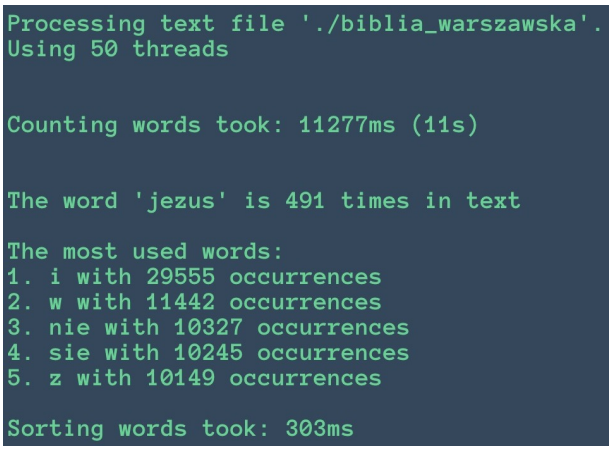


Obrázek 3: 1000 vláken

Ještě se podívejme na text v polském jazyce, kde si ukážeme i možnost vyhledání slova.



Obrázek 4: 8 vláken



Obrázek 5: 50 vláken a nalezený výraz

5 Závěr

Při tvoření tohoto programu jsem zlepšil znalost jazyka **Go** a vyzkoušel si paralelizovat práci s polem a řetězci. Z testovacích výsledku usuzuji, že práce s řetězci a poli je jak stvořena pro paralelizaci. Zrychlení při použití více vláken je obrovské. Faktem, který této myšlence napomáhá je potřeba operační paměti pro uložení dat. Protože pracujeme s velkým obsahem, pak rozdělení do menších paměťově méně náročných celků práci ulehčí.

Reference

[1] Cohen R., *Why you can have millions of Goroutines but only thousands of Java Threads*