Word Counter

Funkcionalita

- Načíst soubor podle parametru programu.
- Zjistit počet opakování daného slova v souboru.
- Paralelizovat běh pro více souborů.
- Detekovat přepínače v příkazové řádce.
- V případě nesouladu na vstupu upozornit uživatele a ukončit program.
- Uložit všechna slova do jednoho souboru

Návrh

Přepínače

V případě špatného vstupu, program vyhodí výjimku a vyšle uživateli pokyn pro zadání přepínače - - help, který uživateli vypíše obecný příklad vstupu.

[*Paralelizace* *rozpoznání velikosti písmen* *Smazání interpunkce* *soubor* *další soubor* … *n-ty soubor*]

Paralelizace

-st : „Single Thread“

-mt : „Multi Thread“

Rozpoznání velikosti písmen

-c: „capital“

Smazání interpunkce

-i: „interpunction“

Spuštění

Zkompilovaný soubor je třeba umístit do podsložky /compilated. Pro zadání zdrojových souborů je třeba soubor umístit do složky /resources následně pak vložit **jméno souboru** jako vstupní argument programu – **nenastavovat relativní cestu k souboru od zkompilovaného programu.**

Výstup pak bude ve složce /resources/counted.

Implementace

Program je koncipován do 3 částí – main, parallel, sequence. Main slouží k úvodnímu zpracování vstupu od uživatele a dle tohoto vstupu od uživatele deleguje požadavky do částí parallel a sequence.

Parallel – program běžící při požadavku na vice vláken.

Sequence – program běžící při požadavku na proces bez vláken navíc.

Třídy parallel a sequence dědí od společného předka counter. Přišlo mi vhodné dát těmto třídám společné rozhraní a zbytečně neporušovat „DRY“.

Třída counter poskytuje třídám parallel a sequence pro reprezentaci dat knihovní strukturu std::unordered_map. Z počátku implementace mě napadla možnost reprezentovat každé slovo instancí nějaké mé vlastní struktury. Usoudil jsem ale, že by vytváření těchto instancí bylo dost nákladné, nakonec bych je stejně musel uchovávat v jiné struktuře (např. std::vector). Samozřejmě jsem tímto řešením eliminoval prostor pro chybu při tvorbě vlastní struktury.

Finální implementace je mimo jiné rozdílná od prvotní také v užívání přepínačů. Uživatel nyní nemusí zadávat na vstup všechny druhy přepínačů. Minimální požadavek na vstup je tedy alespoň jeden textový soubor. Defaultně se program volá jako varianta se sekvenčním přístupem.

Problémy

Samotnou implementaci nedoprovázely výrazné problémy. Avšak při testování programu na různá vstupní data jsem zjistil, že rozdíl mezi během single a multi thread je opravdu malý. Nakonec se mi stalo, že single thread varianta „předběhla“ multi thread variantu. S tím samozřejmě nastala panika a pokusy o otestování. Dle statistik uvedených dále v dokumentu jsem usoudil, že problém nastane, když je v souboru vysokých počet různých slov (možná se pletu a budu rád za vysvětlení příčiny) – problém je dále zanalyzován v dokumentu.

Na doporučení jsem užil knihovní strukturu std::unordered_map, která naprosto markantně ovlivnila sekvenční přístup. Struktura std::map, obsažena v prvotní implementaci, poskytovala automaticky seřazený výpis slov – což přispívá k větší náročnosti programu. Ve finální práci je užita unordered_map.

Testovací data

Test1.txt, Test2.txt, Test3.txt – Každý soubor vychází z cca 350 slov, která jsou několikrát nakopírovaná za sebe.

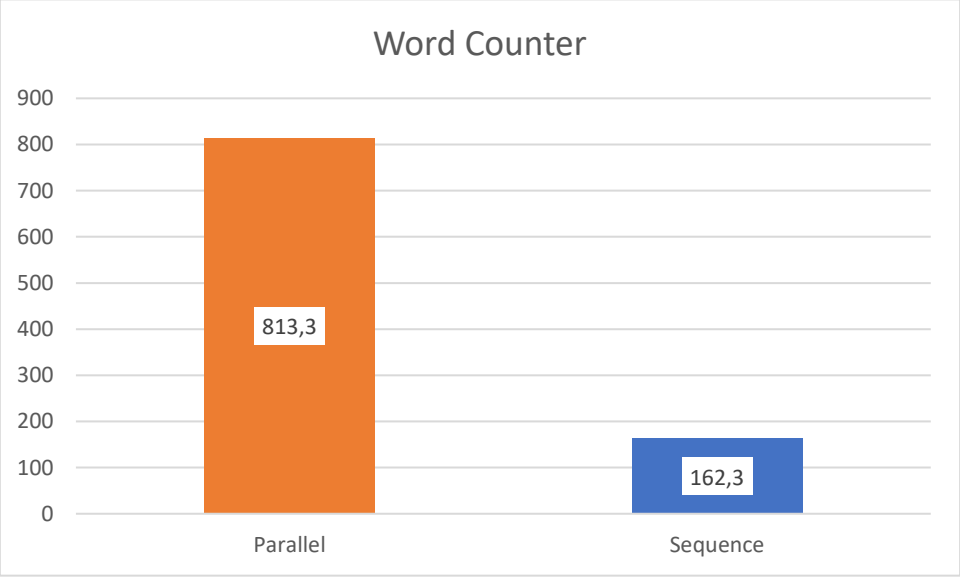
Lotr1.txt, Lotr2.txt, Lotr3.txt, Hobit.txt – Testovací data obsahující vysoký počet různorodých slov.

Statistiky

Testováno na: Intel Core i5-2500k 3,30GHz, 8GB DDR3 RAM, HDD - WD1002FAEX

Pro co nejlepší zanalyzování situace jsem přidal měření času pro zpracování jednotlivých souborů, ukládání dat a samozřejmě celého programu. Veškeré časy jsou uvedeny v milisekundách [ms]. Zároveň pro porovnání uvádím data, která jsem naměřil s prvotní implementací.

lotr1.txt lotr2.txt lotr3.txt		
	Parallel	**Sequence**
1	994	171
2	890	163
3	790	140
4	825	157
5	829	158
6	835	171
7	746	167
8	736	159
9	754	170
10	734	167
Avg.	**813,3**	**162,3**



Výsledek není vůbec zdařilý. Paralelní zpracování souboru je výrazně pomalejší. Při zpracování testovacích souborů test1.txt test2.txt test3.txt je paralelní zpracování efektivnější (pro tyto soubory statistiky neuvádím).

Závěr

Problém, který nastal se mi bohužel nepodařilo vyřešit. Paralelní aplikace běží na vstupních datech, která obsahují obrovské množství různorodých slov, a běží pomaleji než aplikace sekvenční. Jelikož nejsem schopný sám aplikaci testovat na OS linux, požádal jsem kolegu, který testoval na procesoru s 2 fyzickými jádry. Výsledky byly mnohem lepší, ačkoliv byla vícevláknová aplikace stále pomalejší výsledky se lišily v poměru cca 300ms : 180ms – paralelní : sekvenční.

Jelikož jsem do metod readfile a saveWords ve třídě counter umístil měření času, mohu vyloučit možnost pomalého zápisu na disk. Problém pravděpodobně nastává při čtení dat a vkládání do datové struktury std::unordered_map – tedy v metodě readFile. Nerad bych se pouštěl do vymýšlení přehnaných teorií, kde by mohl být problém… Možná problém paralelního přístupu na disk? Načítání velkého množství dat do mezipaměti? – to vše paralelně pak zpomaluje celý chod…? Skutečně nevím a nedokážu usoudit.

Zajímavé bylo však zjištění, jak markantně ovlivnila změna datové struktury sekvenční chod aplikace.