Eötvös Loránd Tudományegyetem

Informatikai Kar

Programozási Nyelvek és Fordítóprogramok Tanszék

Osztott rendszerek specifikációja és implementációja

IP-08bORSIG

Dokumentáció az 1. beadandóhoz

Abonyi-Tóth Ádám
DKC31P

2017. október 26.

1. Kitűzött feladat

Adott egy bemeneti fájl, amelynek sorait szavanként hash-elni kell. Egy szó hashértéke a benne lévő karakterek hashértékének összege. A szó egy `'c'` betűjének hashkódja (`'kód'`) az alábbi módon áll elő:

```
'kód' : Nemnegatív egész szám := 1638; (0x666)
ha 'c' ASCII értéke páratlan:
    a 'kód'-ot bitenként shifteljük balra 11-et
egyébként:
    'kód'-ot shifteljük 6-ot bitenként balra.
'kód' := 'kód' XOR ('c' ASCII értéke BITENKÉNTI ÉSELVE 255-el); (0xFF)
Ha az így kapott 'kód' prím szám, akkor bitenként 'vagy'-oljuk, amennyiben
nem prím, 'és'-eljük össze 305419896-tal. (0x12345678)
```

A program feladata az, hogy a bemeneti fájl (`input.txt`) sorait párhuzamosan hash-elje, és az eredményt kiírja (a sorok sorrendjét megtartva) a kimeneti fájlba (`output.txt`).
A bemeneti fájl első sora egy N természetes számot tartalmaz, utána pedig N sornyi szöveges információ található. Feltehetjük, hogy a szövegben az angol ábécé betűit használó szavak, valamint általános írásjelek (pont, vessző, kérdő- és felkiáltójel, aposztróf, kettőspont stb.) találhatóak.
Példa bemenet:

```
4
Never gonna give you up, never gonna let you down
Never gonna run around and desert you
Never gonna make you cry, never gonna say goodbye
Never gonna tell a lie and hurt you
```

2. Felhasználói dokumentáció

2.1. Rendszer-követelmények, telepítés

A programunk több platformon is futtatható, dinamikus függősége nincsen, bármelyik, manapság használt PC-n működik. Külön telepíteni nem szükséges, elég a futtatható állományt elhelyezni a számítógépen.

2.2. A program használata

A program használata egyszerű, külön paraméterek nem kötelezőek, így intézőből is indítható. Ha a programot paraméterek nélkül futtatjuk, a bemenetét a futtatható állomány mellett kell elhelyezni az *input.txt* nevű fájlban. Ha pontosan egy parancssori paramétert kap a program, akkor az így megadott paraméter lesz a bemeneti fájl elérési útja.

Figyeljünk az inputfájlban található adatok helyességére és megfelelő tagolására, mivel az alkalmazás külön ellenőrzést nem végez erre vonatkozóan. A futás során az alkalmazás mellett található *output.txt* fájl tartalmazza a kapott eredményt, ahol az *i*-ik sor a bemeneti fájl *i+1*-ik sorának hashértéke.

3. Fejlesztői dokumentáció

3.1. Megoldási mód

A kódunkat logikailag két részre bonthatjuk, egy fő-, illetve több alfolyamatra. A fő folyamatunkat a `main()` és a `read()` függvények fogják megvalósítani, melyek feladata a bementi fájl beolvasása, illetve az alfolyamatok munkavégzési idejének mérése. Az alfolyamatok a bemeneti állomány egy-egy sorának hash-eléséért felelősek. A végeredményt a főfolyamat írja ki a kimeneti állományba.

3.2. Implementáció

A bemeneti fájlt `std::vector<std::string>` típussal fogjuk megvalósítani, míg az alfolyamatokat egy `std::future<std::string>` típusparaméterű vektorban fogjuk tárolni. A szükséges N folyamatot az `std::async()` függvény segítségével, azonnal fogjuk új szálon indítani, paraméterül a végrehajtandó `process_line()` függvényt, illetve a szöveg egy sorát fogjuk átadni.

A `process_line()` függvény feladata egy sor szavainak hash-elése, és a kiszámolt hash-értékekből a kimenet sorainak összeállítása. Az implementáció használhatott volna `std::istream_iterator<T>`-on alapuló `std::transform()` függvényt, de az iterátor csak pointeren keresztül érte volna el a stream-eket, ez pedig a feladat egyszerűségéhez képest felesleges overhead-et jelentett volna.

A szavak hashértékének kiszámítása triviális, a karakterek hashértékének számítása szigorúan a feladat specifikációja alapján történik. Az `is_prime(uint32_t)` függvényben azért van kibontva az első 5 prímmel való oszthatóság, mert a modern fordítók alkalmazhatnak olyan optimalizációkat, amik a fordítási időben ismert értékekkel való maradékszámítást gyorsabbá teszik, mint egy változó esetében. A `nontrivial_prime(uint32_t)` függvény egy template paraméterrel rendelkezik, mely azt jelzi, hogy mettől kezdje vizsgálni a prímszámokat. Mivel ez már fordítási időben ismert, érdemesebb így, template paraméterként átadni, hiszen így teret adhatunk egyéb, fordítási idejű optimalizációknak.

A feladat egyszerűsége révén egyetlen forrásfájlban, a `main.cpp`-ben található a teljes implementációs kód.

3.3. Fordítás menete

A programunk forráskódját a `main.cpp` fájl tartalmazza. A fordításhoz elengedhetetlen egy `C++11` szabványt támogató fordítóprogram a rendszeren. Ehhez használhatjuk az *MSVC*, *g++* és *clang* bármelyikét, de *Windows* operációs rendszer alatt *MinGW* fordítóval a fordítás sikertelen lehet, mert ezen a platformon az `std::future` típus implementációja hiányozhat.

A fordítás menete (4.8.4-es verziójú g++ használata esetén) a következő: `g++ main.cpp -std=c++11 -pthread`. A speciális, `-std=c++11` kapcsoló azért szükséges, mert alapértelmezés szerint ez a verziójú fordítóprogram még a régi, C++98-as szabványt követi, melyben a felhasznált nyelvi elemek még nem voltak jelen. A `-pthread` kapcsolóra azért van szükség, hogy a fordító, illetve az operációs rendszer megengedje a programnak a többszálúság használatát.

3.4. Tesztelés

A program tesztelése két részből állt, az egyik a megoldás helyességét vizsgálta, a másik a kód sebességét.

A helyesség vizsgálatához a programot lefuttattam a feladathoz mellékelt bemeneti fájlokra, és `git diff --no-index --color-words` paranccsal ellenőriztem, hogy nincs eltérés az elvárt és a kapott eredmények között.

A sebesség vizsgálatához a ugyanazt a szöveget tördeltem több sorra, ezzel ellenőrizve, hogy hogyan befolyásolja a futásidőt az állandó mennyiségű feladatra kiosztott szálak száma. A teszteket futtató processzor 4 logikai magot működtetett 2.9GHz-es órajelen. Az eredmények alább láthatóak.

Szálak száma	1	2	3	4	5	6	7	8	9	10	11	12	...	16
Futásidő (s)	1.082	0.551	0.434	0.349	0.449	0.328	0.312	0.305	0.302	0.316	0.316	0.318	...	0.316
Összköltség	1.082	1.102	1.302	1.396	2.246	1.969	2.181	2.443	2.717	3.164	3.477	3.815	...	5.06

A táblázat első négy oszlopában az összköltség *(futásidő × szálak száma)* nagyjából állandó, tehát itt a futásidő megközelítőleg fordítottan arányos a processzek számával.

A további oszlopokból az derül ki, hogy az összköltség hirtelen megnő, amint az indított szálak mennyisége meghaladja a processzor logikai magjainak számát, ami azt jelenti, hogy onnantól kezdve a gyorsulás nem lesz olyan nagy mértékű, mint amíg nem használtunk ki minden processzormagot. Azt is láthatjuk, hogy a futásidő egy idő után stabilizálódik $0.316s$ körül, akárhány szálat indítunk.