Eötvös Loránd Tudományegyetem

Informatikai Kar

Programozási Nyelvek és Fordítóprogramok Tanszék

Osztott rendszerek specifikációja és implementációja

IP-08bORSIG

Dokumentáció az 3. beadandóhoz

Mikus Márk István
CM6TSV

2017. december 17.

1. Kitűzött feladat

Egy előre megadott fájlban képek szöveges reprezentációját találhatjuk (pixelenként RGB módon).

A program parancssori paraméterként kapja meg az alábbiakat:

- A képek átméretezési arányát %-ban. (pl. 50 - ekkor 50%-ra, azaz felére kell csökkenteni az összes képet. 25 esetén negyed-méretet kapnánk, etc.)
- Annak a fájlnak a neve, a képek primitív leírását tartalmazza. (pl. 'pictures.txt')
- A kimeneti fájl neve (pl. 'picross.quiz')

A fájlból (első param.) beolvasott képeket először a megadott arányban át kell méretezni.

Ezek után az így kapott kisebb képek színeit kell leképezni az előre megadott 8 szín valamelyikére.

Ezt követően az így kapott ábrákban minden sorra és oszlopra ki kell számolni, hogy egymás után hány azonos színű pixelt láthatunk (de nem szimplán azt, hogy az adott sorban/oszlopban hány különböző szín található).

A kapo tt eredményeket (a méretezett és megfelelő színre konvertált képeket és a hozzájuk tartozó címkéket) írjuk ki a kimeneti fájlba (3-ik paraméter)!

Az egyes feladatok részletezése lejjebb olvasható.

1.1. Bemeneti fájl felépítése

A paraméterben megadott fájl felépítése a következő:

`N`
P_1
P_2
...
P_N

Az első sorban található N szám alapján összesen N képre kell a később leírt számításokat alkalmazni, ezt N kép adatai követik az alábbi módon (a P_i kép leírása):

Először a kép szélességét (és egyben magasságát) olvashatjuk "pixelben" (s_i).

Az ezt követő (s_i) sorban összesen 3 x s_i x s_i pixel színkódja olvasható RGB formátumban, whitespacekkel tagolva (egy pixelt 3 szín alapján határozunk meg, így egy sorban összesen $3 * s_i$ adat olvasható.

Mivel a képünk négyzetes, azaz s_i x s_i méretű, összesen 3 x s_i x s_i adatot találunk).

Lehetséges részlet ekkor az i-ik képnél:

```
4
96 252 199 129 242 211 93 238 196 5 152 143
242 185 199 205 28 185 134 134 62 66 37 24
238 181 253 190 123 12 5 73 230 226 185 204
122 25 11 162 235 33 33 254 159 118 232 109
```

Ez alapján a kép bal felső pixele a (96, 252, 199), a tőle jobbra lévő pixel a (129,242,211) színt veszi fel.

A következő sorban az első pixel színe (242,185,199), ezt követi a (205,28,185) stb., stb.

Egy valid 'pictures.txt' fájl felépítése ekkor az alábbi lehet:[1]

```
2
4
96 252 199 129 242 211 93 238 196 5 152 143
242 185 199 205 28 185 134 134 62 66 37 24
238 181 253 190 123 12 5 73 230 226 185 204
122 25 11 162 235 33 33 254 159 118 232 109
2
91 64 228 191 3 105
241 77 140 185 50 163
```

A program egy lehetséges paraméterezése és futtatása tehát a következő:

```
> spawn -> master 50 pictures.txt picross.quiz
```

1.2. 1. részfeladat: Átméretezés

Bemenő adatként egy képet, és egy tömörítési arányt kapunk (hogy ez egy százalék, vagy az összevonandó pixelek száma soronként/oszloponként, az tetszőleges).

Az itt kiszámolt adat egy új kép, melynek mérete az eredeti méretének X%-a.

Az átméretezéshez egy nagyon egyszerű algoritmust fogunk használni, mely szimplán átlagolja az eredeti kép pixeleinek színét, és ebből állítja elő az új képpontnak megfelelő értéket.

Az új kép pixeleinek színét az eredeti adatból tudjuk kiszámolni a kövekező módon:

Ha p pixelt kell összevonni, akkor az új képünk $[i][j]$-ik pixelének színéhez az eredeti kép $[i * p + k][j * p + l](k, l \in [0..p)$ pixeleinek színét kell átlagolni.)

(Tehát ha 50%-ra méretezünk, akkor a kép 1/2 része lesz az eredetinek, azaz 2x2 pixelt vonunk össze. 25%-nál 1/4 részre 4x4 pixelt, 10% esetén 1/10-re 10x10 pixelt, etc. etc.)

[1] A fájlban tehát két "kép" található. Az első 4x4-es, a második 2x2-es.

A képet itt 1-től indexeljük (absztrakt algoritmus) - C++ban a vektort/tömböt azonban 0-tól, így a két külső ciklus más intervallumot jár be - [0..*newwidth*).

```
newwidth := orignalwidth / p
newheight := originalheight / p
result is image(newwidth, newheight)

for i in [1..newwidth] do
    for j in [1..newheight] do
        result[i,j] := average(original[i*p + k, j*p + l] ) (k,l ∈ [0..p) )
    endloop
endloop
```

A számolás során az egyes szín-komponenseket mindig *mod*256 kell érteni, az egyedi ábrázolási módoknál figyeljetek a túlcsordulások elkerülésére. Alapesetben [0..255] közé esik minden szín, amit ha 3 változó reprezentál, akkor az átlaguk sem lehet ennél nagyobb/kisebb, ebben az esetben ezzel nem kell külön foglalkozni. Ha egyetlen int-be csomagolva tároljátok azonban az adatokat (konzis "bitshift-magic módszer"), akkor azonban nem lehet simán összeadni a színeket - a két kék komponens összege túlcsordulhat a zöldbe, ami pedig a pirosba, eltorzítva az eredményt - ilyen esetben különös odafigyeléssel kell dolgozni.

Az adatcsatornában az így kapott képet kell továbbküldeni.

1.3. 2. részfeladat: Színkódolás

A csatornából érkező adatunk egy MxM-es kép (az M értéke képenként változik!). A feladatban megadott 8 alapszínt a 3 bites RGB-paletta alapján a következőknek tekintjük:

```
BLACK  (0,0,0)
RED    (255,0,0)
GREEN  (0,255,0)
BLUE   (0,0,255)
CYAN   (0,255,255)
MAGENTA(255,0,255)
YELLOW (255,255,0)
WHITE  (255,255,255)
```

Egy szín leképezését a fent megadott 8 szín valamelyikére vetítsünk komponensenként - amennyiben egy szín a [0..127] intervallumba esik, akkor az adott komponens 0 értéket vesz fel, ha a [128..255]-be, akkor az eredmény 255.

Az MxM-es kép mindegyik pixelét ily módon kell kódolni, hogy ezt az új képet küldhessük tovább a csatornában.

1.4. 3. részfeladat: Címkézés

A bemeneti adat elsősorban egy MxM-es kép, ebből kell címkéket készíteni (az M érték itt is változó) minden sorra és minden oszlopra. Az alábbi "kép" esetén pl:

```
BLACK RED GREEN BLUE
BLACK RED RED RED
BLACK RED RED YELLOW
MAGENTA RED YELLOW YELLOW
```

A sorokhoz tartozó címkék az alábbiak:

```
1 1 1 1
1 3
1 2 1
1 1 2
```

Ennek a jelentése az alábbi: Az első sorban négy, különböző színű pixel található. (B R G B) A második sorban egy különböző, majd 3 azonos színű pixel olvasható. (B R R R) A harmadik sorban egy egyedi, aztán két egy forma, végül az előtte lévőtől megint eltérő színű pixelt láthatunk. (B R R Y) Az utolsó sorban két különböző, majd egymással (de az előttük lévő pixellel nem) azonos színű 2 pixel látható. (M R Y Y)

Az oszlopokhoz ugyanitt a következőt rendelhetjük:

```
3 1
4
1 2 1
1 1 2
```

Az oszlopok címkéit analóg módon értelmezzük.A csatorában tovább az itt kapott képet, majd a sorok és oszlopok címkéit küldjük tovább.

A futás során feltehetjük az alábbiakat:

- A bemeneti fájl létezik.
- A bemeneti fájl felépítése megfelel a feladat specifikációjában leírtaknak.
- Az egyes színek rendre a [0..255] intervallumból veszik fel egész értékeiket.
- Az egyes képekhez tartozó adatok valódi adatokat tartalmaznak, nincs kevesebb/-több/más típusú adat, melyre a programot külön fel kell készíteni.

- A képek mérete (2^Mx2^M, M>=2)
- Az átméretezés során nem lesznek lógó sorok/oszlopok, azaz pl. ha felezni kell egy képet, akkor 50% esetén 100/50 = 2 -> 2x2 pixelt kell
- összevonni egy újjá (tehát nem foglalkozunk a 42,69%-os méretezéssel, hogy milyen módon kell számolni, az arány alapján a számolásban egyértelműen lehet indexelni a megfelelő sort/oszlopot az eredeti kép pixeleihez).
- Az összevonásnál a kisebb képpont-mátrixok egymástól diszjunkt részt fednek le (nincs overlapból adódó probléma).
- Nem méretezünk 0%-ra vagy annál kisebbre, illetve 100% fölé (identikus leképezés - 100% - azonban előfordulhat), azaz az első paraméter az [1..100] intervallumba eső egész szám.

1.5. 1. részfeladat: Szülőfolyamat

A szülőfolyamat (master) dolga az eredeti képek beolvasása, az adatcsatorna belső függvényeinek megfelelő gyerekfolyamatok (first, second, third) elindítása, a csatorna összeállítása megfelelő módon, illetve minden gyereknek azok működéséhez feltétlen szükséges adatok továbbítása. A képeket (és a plusz adatot, amit annak számítása igényel - átméretezési arány) az adatcsatornában elsőként szereplő gyereknek kell elküldeni! A master csak a csatorna végén lévő gyerektől fogadjon adatot! A harmadik gyerektől fogadott adatokat a szülőfolyamat írja ki a paraméterül kapott fájlba az alábbi módon: ($i \in [1..N]$, azaz ahány kép van)

P_i1
P_i2
..
P_in
L_iR
L_iC

Értelmezés: Az i-ik kép első sorának pixelelei '(R,G,B)' módon formázva (azaz zárójelezve, vesszőkkel elválasztva) szóközzel tagolva. (sor végén IS szóköz.) Az i-ik kép második sorának pixelelei '(R,G,B)' módon formázva hasonló módon, mind az n sorra. Az i-ik kép SOR-aihoz tartozó címkék, a fenti alakban (címkézés fejezet). Az i-ik kép OSZLOP-aihoz tartozó címkék, ahogy előbb is.

Az alábbi paraméterezés melletti futtatással a következő kimeneti fájlt kaphatjuk (a program egy lehetséges paraméterezése és futtatása tehát a következő):

```
> spawn -> master 50 pictures.txt picross.quiz
> cat picross.quiz
```

```
(255,255,255) (0,255,0)
(255,255,0) (0,255,255)
1 1
1 1
1 1
1 1
(255,0,255)
1
1
```

2. Felhasználói dokumentáció

A program egy konzolból futtatható alkalmazás, amelyet a feladat által leírt módon kell paraméterezni, a program a lefutása során a paraméterként megadott állományba (output) írja az eredményt.

2.1. Rendszer-követelmények, telepítés

A programunk több platformon is futtatható ($Windows/Linux$), amelyen fel van telepítve a PVM3 rendszer, amelynek része egy függvénykönyvtár, a démon és a konzol.

2.2. A program használata

A programot a PVM konzolon keresztül lehet futtatni. A PVM indításához szükséges parancs:

`pvm [Optional:] <hostsfile name>`[2]

Ennek hatására elindul a PVM-konzol. Ezek után a *spawn* kulcsszóval lehet meghívni a szűlőfolyamatot az alábbi módon:

`pvm> spawn -> master <szám> <input> <output>` Ahol a `<szám>`, a feladatban szereplő százalék, az `<input>` a bemeneti fájl, míg az `<output>` a kimeneti fájl neve.

A program hívásakor, a $/pvm3/$ könyvtárral egy szinten lévő fájlokra rá lát a rendszer, szóval amennyiben a fájlokat nem útvonallal adjuk meg, úgy alapértelmezetten itt keresi az output és input fájlokat.

A program nem csak PVM-konzolból, hanem futatható állományként is indítható, ha belenavigálunk a $/pvm3/bin/LINUX64/$ könytárba és az ott elkészített szimbolikus linkeket szólítjuk meg, az előbb említettekkel analóg módon:

`$ ~/pvm3/bin/LINUX64/master <szám> <input> <output>`

[2]Amennyiben megadjuk az opcionális paramétert, úgy a PVM-konzol a megadott fájlban meghatározott configurációval indul, s csatlakoztatja a fájlban felsorolt *blade*-ket

Figyeljünk a paraméterek helyességére illetve teljességére továbbá a bemeneti adat formátumának helyességére, mert a program külön ellenőrzést erre vonatkozóan nem végez. Ha nincs megfelelő számú paraméter a program kilép -1-es hibakóddal.

3. Fejlesztői dokumentáció

A program 5 db C++ 11. szabványnak megfelelő fordítási egységből áll. A főprogram, a főszál a *master.cpp* állományban, míg a megoldás egyes komponenseit alkotó programok a $first.cpp$, *second.cpp*, *third.cpp* állományban található továbbá van még egy segédállomány ($functions.h$) amely a komonensek által használt közös metódusokat tartalmazza.

3.1. Megoldási mód

A feladatot az Adatcsatorna tételére vezetjük vissza.

3.2. Adatcsatorna tétel

$D =< d_1, ..., d_N >$
$F(D) =< F(d_1), ..., F(d_N) >$
$F = f_M \circ f_{M-1} \circ ... \circ f_0$

$A = \underset{x_0}{Ch} \times \underset{\overline{x_0}}{Ch} \times ... \times \underset{x_{M+1}}{Ch} \times \underset{\overline{x_{M+1}}}{Ch}$
$B = \underset{x_0'}{Ch} \times \underset{\overline{x_0}'}{Ch} \times ... \times \underset{x_{M+1}'}{Ch} \times \underset{\overline{x_{M+1}}'}{Ch}$

3.3. Visszavezetés az adatcsatorna tételre

$D =< d_1, ..., d_N > \quad d_i \in \mathbb{Z}^{2^M \times 2^M} \ (i \in [1..N], M \geq 2)$
$F(D) =< F(d_1), ..., F(d_N) >$
$F = f_M \circ f_{M-1} \circ ... \circ f_0$ (bizonyítás teljes indukcióval)

Jelenegi esetben: $M := 2$

$F(x) = f_2 \circ f_1 \circ f_0$
Az egyes f_i a feladatban megfogalmazott 3 rész-számítás megvalósítása

Megoldóprogram:
$S = (||_{i=1}^{M} x_i =<>, \left\{ \square_{i=0}^{M} x_i, x_{i+1} := lorem(x_i), hiext(x_{i+1}, A_i \cdot lov(x_i)),\ ha\ x_i \neq <> \right\})$

A főszál feladata az adatok beolvasása, és a gyermek folyamatok, illetve a csatornára az adatfolyam elindítása, majd a válasz megfelelő fájlba történő kiírása.

Az első állomás feladata a kapott kép, tömörítése, továbbítása. A második állomás feladata a kapott kép színkódolása, továbbítása és a harmadiké pedig,a címkegenerálás illevte továbbítás az adatcsatorna végébe (master.cpp).

3.4. Implementáció

3.4.1. Master

A fentebb leírt feladatot további alfeladatokra bonthatjuk:

1. Bemeneti paraméterek egyszerű validálása és inicializálása megfelelő változókba
2. Adatok betöltése
3. Gyermekek folyamat indítása, csatornareálciók illetve komponensek felállítása
4. Adatok küldése (csatornába)
5. Adatok fogadása (csatornából)
6. Eredmény kiírása

A feltételek szerint bemeneti paraméterek validálásakor csupán elég megnézni, hogy meg lettek-e adva azok, illetve, hogy a p százalékot kifejező szám a 25, 50 illetve 100 értékek közül kerül ki. (Hiba esetén kilépünk a -1-es hibakóddal `return -1`) [3] Az *input* és *output* `std::string` típusú paraméterek, egyéb megszorítás nélkül.
Az adatok alapvető tárolására az `std::vector`-t használjuk. Az adatok betöltésekor, ahelyett, hogy egy újabb mélységű vektort használnánk az egyes színek tárolására, az integer típuson értlmezett bitshift-tel egy interben tároljuk a 3, 0 és 255 közötti számot.[4] (A `functions.h` állományban definiált `createRGB` ill. `fromRGB` metódsok felhasználásával)
A gyermek folyamatok indításakor leellenőrizzük, a PVM-ben szokásos módszerrel, a folyamatokat. Ha valamely `pvm_spawn` negatív eredménnyel szolgál, akkor kilépünk a -2-es hibakóddal. Miután felálltak a komponensek, elküldjük a csatorna kiépítéséhez szükséges adatokat az egyes gyermekeknek. Az első a p, szám mellett megkapja, hogy kinek kell majd továbbítania, illevte azt is, hogy hány kép fog áthaladni a csatornán. A második megkapja, az utóbbi két említett adatok felül azt is, hogy kitől kell fogadnia az adatokat. A harmadik, utolsó komponens azt kapja meg, hogy kitől fogadjon adatot, illetve szintén megkapja a csatornán áthaladó képek számát.

Miután elküldte a csatorna inicializálásához szükséges adatokat, a csatornát felkészültnek tekinti, az adatáramlásra. Az első gyermeknek elküldi a megfelelő képeket egymás

[3] Az egyelen olyan százalékok, amelyek esetében a 2 hatvány méretű mátrixokat diszjunk részmátrixokra bonthatjuk → Divide & Conquer

[4] 16-tal és 8-cal eltolva a megfelelő piros, és zöld koordinátákat

utána sorrendben. Miután az utolsót is elkülte, felkészül az ugyanennyi kép fogadására, amit folytatólagosan beleír a kimeneti állományba, amelyet miután megjött az utolsó kép is lezár. Ezután a program leáll.

3.4.2. First

Az első csatorna-állomás feldata, miután fogadta az inicializáláshoz szükségs adatokat, hogy adott számú képek mindeggyikét fogadja, azt tömörítse, majd továbbküldje. A képek fogadását a `functions.h` állományban található `recievePicture` metódus végzi. A komponens főfeladatát a tömörítést a *Divide & Conquer* módszer adja: feldaraboljuk a problémát egyszerűbb alproblémákra, amiket addig bontogatunk szét rekurzívan, ameddig triviális alapesetek segítségével megoldhatóvá válik a probléma. Így a megoldási elv a következő: Adott egy K szám, amely azt jelenti, hogy ilyen méretű mátrixot, már szekvenciálisan, a feladatban megadott dupla ciklus segítségével oldjuk meg. A tömörítés során ha az adott méretnél nagyobb méretű mátrixot kapu a függvény, akkor felosztja 4 diszjunk részmátrixra, s ezekre az `std::future` segítségével újra meghívja a tömörítőfüggvényt a negyedekre, majd összeállítja az eredmény-mátrixot. Az eredmény mátrixot a `functions.h` állományban található `sendPicture` metódussal továbbítja, a második állomásnak.

A K szám meghatározásában a feladat, szabad kezet adott, ugyanakor, a modellből következi, hogy 4 illetve annál kisebb méretű mátrixokra lehet elvégezni, a szekvenciális kiszámítást.

3.4.3. Second

A második komponens feladata a tömörített kép fogadása, színkódolása a feladatban leírt módon. Amennyiben a szín komponens egy koordinátája [0..127] -be esik akkor 0, amennyiben [128..255]-be, akkor 255 lesz a kódolt érték. Az implementációban az egyes pixel-sorokat egymástól elkülonítve, párhuzamosan számoljuk a `std::future` és `std::asnyc` segítségével. Az egyes sorokra, megadjuk, azt a sort, amely ugyanakkora elemszámú, de kódolt értékeket tartalmazza. Az eredmény-számítás fogadása után továbbítja a képt a harmadik komponensnek a már megszokott módon.

3.4.4. Third

Az utolsó állomás feladata, a címkék legenerálása, ehhez implementálni kell egy metódust, amely a fentebb említett módszerhez hasonlóan, soronként külön szálon indítva meghatározza az eredényt egy sorban, majd összeállítja azokat az eredeti vektornak megfelelően. A feladat része egy oszlopra vetített címke-mátrix generálása is, úgyhogy egy plusz metódus és az előbbi metódus segítségével könnyen kiszámítható az is. Ezt a függvény a `getColumnMatrix` metódus implementálja, amely megadja a sor alapú mátrixnak,

az oszlop alapú verzióját (transzponáltját). Erre meghívva az előbb említett `getRowSum` metódust, a feladat által kívánt második címkemátrixot kapjuk.

3.5. Fordítás menete

A programunk forráskódját a `master.cpp`, `first.cpp`, `second.cpp`, `third.cpp` és a `function.h` állomány tartalmazza. A fordításhoz szükséges `Makefile.aimk` állomány amely a fordítási információkat tartalmazza a forrásállományokkal egy szinten található. A fordításhoz a *g++* fordítóprogram szükséges, illetve az *aimk* program, aminek segítségével könnyen kialakítható a PVM fordításhoz szükséges környezet. A $Makefile$-ban meghatározott információk alapján lefordul az összes benne meghatározott fordítási komponens, amennyiben a $/pvm3/src/$ könyvtárban kiadjuk a következő parancsot:

```
aimk
```

Ezután, ha futtatható állományokat szeretnénk kapni, létre kell hozni a szimbolikus linkeket, ugyanerről a hyleről kiadva a következő parancsot:

```
aimk links
```

Ha mindkét parancs hibamentesen lefutott, lefordult a kód és a megfelelő linkek is létrejöttek. Így a konzolból (vagy intézőből) futtatható a program.

3.6. Tesztelés

A program tesztelése során különböző méretű bemeneti fájlokkal, illetve paraméterekkel futtattam. (Az `atlasz.elte.hu` szerver fejgépén illetve blade-jein) A programom minden esetben a tőle elvárt kimenetet állította elő, így a tesztesetek alapján helyesnek gondolhatjuk a működését.

1. Hiányzó paraméterek tesztesete
2. $p = 50$ és 3 kép esete (4, 8, 16 méretűek)
3. $p = 100$ 50 képre
4. $p = 25$ 30 képre

A tesztelést az `atlasz.elte.hu` szerveren végeztem. Itt az *atlasz* egy számítási egységet (*sz.e.*) jelent számunkra. A tesztelés során sorban adogattam hozzá a blade-ket, egészen 7 darab hostig. (8 hostot már nem mindig engedett a szerver) Az itt mért futásidők egy összefoglaló táblázatban:

- $T_1 : p = 50$ 3 képre (4, 8, 16 méretűek)
- $T_2 : p = 100$ 50 képre
- $T_3 : p = 25$ 30 képre
- N/A : Azt jelenti, hogy a szerver nem volt képes kezelni, ezt az esetet.

	T_1	T_2	T_3
1 *sz.e.*	∼0.1121 mp	N/A	∼0.9198 mp
2 *sz.e.*	∼0.1216 mp	N/A	∼0.5148 mp
3 *sz.e.*	∼0.0936 mp	∼0.4154 mp	∼0.6003 mp
7 *sz.e.*	∼0.0135 mp	∼0.3976 mp	∼0.5093 mp

Látható, hogy az egyes teszteset típusoknál mekkora gyorsulás érhető el a számítási egységek számának növelésével, és szinte minden esetben közel 3-szor gyorsabb volt a 7 blade-t csatlakoztatott megoldás mint a szekvenciális számítással ekvivalensnek tekinthető 1 számítási egységgel rendelkező verzió.