Eötvös Loránd Tudományegyetem

Informatikai Kar

Programozási Nyelvek és Fordítóprogramok Tanszék

Osztott rendszerek specifikációja és implementációja

IP-08bORSIG

Dokumentáció az 2. beadandóhoz

Mikus Márk István
CM6TSV

2017. november 21.

1. Kitűzött feladat

Feladatunk annak eldöntése, hogy egy adott halmaznak létezik e olyan részhalmaza, melyben található elemek összege pontosan megegyezik egy előre megadott számmal!

A szükséges adatokat a program 3 parancssori paraméteren keresztül kapja még. Az első, egy egész értékű adat, mely a feladatban definiált számot jelzi, ezt kell valamilyen módon elérni a halmazelemek összegével. A második, egy fájl neve - ez tartalmazza a kiinduló halmaz elemeit (inputfájl). A felépítése az alábbi:

A fájl első sorában egy nemnegatív egész szám (N) áll - a kiinduló halmazunk elemszáma tehát N. A következő sorban összesen N db egész számot olvashatunk (pozitív és negatív egyaránt), melyek a halmaz elemeit jelölik (sorrendiséget nem kötünk meg köztük).

Egy megfelelő bemeneti fájl (például data.txt) ekkor:

```
6
3 34 4 17 5 2
```

A harmadik paraméter, annak a fájlnak a neve, melybe a feladat megoldása során kapott választ kell írni.

A kimeneti fájlban található információ az alábbi legyen:[1]

- létezik N összegű részhalmaz: `May the subset be with You!`
- nem létezik ilyen részhalmaz: `I find your lack of subset disturbing!`

A program egy lehetséges futtatása így az alábbi módon valósulhat meg:

```
pvm> spawn -> master 12 data.txt result.txt
```

A kapott kimenet (result.txt) az előző bemenetre tehát a következő:

`May the subset be with you!`[2]

A feladatot *Divide & Conquer* módszer alapján PVM3 használatával kell megoldani, az alábbiak alapján: Amennyiben az elvárt összeg 0, alapesetnél vagyunk, ezt ki tudjuk elégíteni az üres halmaz, mint részhalmaz segítségével, tehát *igaz*. Amennyiben az elvárt összeg nem nulla, de a halmazunk üres, erre nem tudunk megoldást adni, az eredmény *hamis*. Egyéb esetben két lehetőségünk van, az elérni kívánt (összegű) halmazba a jelenleginek egy tetszőleges elemét vagy beletesszük, vagy nem. Ha igen, akkor rekurzívan nézzük meg, hogy az a halmaz kielégíthető -e, melybe a most vizsgált elemet nem vesszük bele. Tehát pl. egy 10 összegű, K elemszámú halmaz akkor és csak akkor tartalmazza a 3-at, mint elemet, ha a 7 összegű, $K-1$ elemszámú halmazt ki tudjuk elégíteni (amely az itt vizsgált 3-at, mint elemet nem tartalmazza) a feltételek alapján. Ha nem, akkor a rekurzió azt dönti el, hogy egy kisebb elemszámú, de azonos súlyú halmazra teljesül e az elvárt feltétel. Ha az 5 összegű, K elemű halmazból ki akarjuk venni a 35-öt, mint elemet,

[1] Egyik esetben sem kell a sor végére enter!

[2] A 3, 4 és 5 elemek összege pontosan 12, így tudunk ilyen halmazt mutatni.

de a vizsgált súlyt nem akarjuk változatatni, akkor vizsgáljuk meg az 5 összegű, $K-1$ elemű halmazt (a 35 nélkül). Ha a rekurzió valamelyik ága igazat ad, akkor a válaszunk *igaz*, tehát tudunk ilyen halmazt mondani (az elemeket nem kell megadni), egyéb esetben *hamis*.

2. Felhasználói dokumentáció

A program egy konzolból futtatható alkalmazás, amelyet a feladat által leírt módon kell paraméterezni, azaz az első paraméter egy egész szám, a második az input fájl neve és a harmadik a kimenő állomány neve amibe a program bele írja az input adatokra adott válaszát.

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>`[3]

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 szerepelő keresett össteg, 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>`

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

3. Fejlesztői dokumentáció

A program két C++ 11. szabványnak megfelelő fordítási egységből áll. A főprogram, a főszál a *master.cpp* állományban, míg a megoldás magja a gyerekfolyamatokat implementáló *child.cpp* állományban található amely a rekurziót fogja adni a megoldáshoz.

3.1. Megoldási mód

A főszál feladata az adatok beolvasása, és a gyermek-lánc elindítása, végül az a válasz fogadása, majd a válasz megfelelő fájlba történő kiírása.

A gyemek feladata, hogy fogadja a megfelelő üzeneteket, és adatoktól függően vissza üzen a szülőnek az alapesetben foglalt válaszokkal, vagy indít két újabb gyeremeket, a kontextusnak megfelelő módosított paraméterekkel, bevárja a válaszukat és úgy üzen vissza a szülőjének.

3.2. Implementáció

3.2.1. Master

A fentebb leírt feladatot 4 alfeladatra bonthatjuk:

1. Bemeneti paraméterek egyszerű validálása és inicializálása megfelelő változókba
2. Adatok beolvasása
3. Gyermek folyamat indítása, üzenet küldése, illetve fogadása
4. Eredmény kiírása

Ugyanakkor implementálás során, optimalizálható a kód egyes részegységek összefogásával.

- A gyerekfolyamatot már a beolvasás előtt létrehozzuk, s így miután beolvassuk az adatokat rögtön el is küldjük.
- Ugyanígy a kiírás esetén is, egyből a válasz fogadásakor írjuk az eredmény fájlt, elkerülve ezzel felesleges másolásokat.

A folyamatot tehát, azzal kezdjük, hogy megvizsgáljuk, hogy minden paraméter adott-e az indításhoz, ha nem akkor rögtön kilépünk 1-es kóddal. (`exit(1)`) Ezután létrehozzuk a leendő egy darab kezdő gyerek-folyamatot, illetve vizsgáljuk annak valid létrejöttét, annak érdekében, hogy amennyiben nincs mód új gyermek létrehozására, úgy a program se használjon felesleges változókat. Ilyen hiba esetén a PVM-et is lezárjuk (`pvm_exit()`), illetve 2-es hibakóddal kilépünk a programból. (`exit(2)`) Amennyiben minden paraméter adott és a gyermek is létrejött inicializáljuk a paramétereket megfelelő változókba. A k egy egész értékű változó, míg a *input* és *output* `std::string` típusúak. Ezután elküdjük

a gyermeknek k-t, majd beolvasásra kerül N (egy pozitív egész szám (`unsigned int`-ként implementálva, hiszen ez jelképezi az átadandó halmaz elemszámát)), amit szintén elküldünk, hogy a gyermek tudja, hogy hány adatot kell fogadnia a következő üzenetben. Inicializáljuk a halmazunkat reprezentáló N elemű vektort (`std::vector<int>`), és feltöltjük a beolvasott értékekkel, majd elküldjük ezt is a gyereknek. Ezután már csak a választ kell megvárni, amelyet egy karakter formájában fogadunk (ez a legkisebb méretű üzenettípus, ami rendelkezésre áll, hogy egy boolean típust szimuláljunk), amitől függően a megfelelő szöveget írjuk az *output* fájlba.

3.2.2. Child

A gyermekfolyamatok feladatát maga 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:

Fogadunk egy üzenetet a szülőfolyamattól, amely egy egész szám lesz k. Amennyiben ez a szám 0, úgy alapeset szerint a válasz hamis, ha nem akkor fogadunk egy újabb üzenetet, ami egy pozitív egész szám, n lesz. Ha n 0, azaz üres halmazt foganánk a következő üzenetben, úgy már most tudunk válaszolni, hogy üres halmaz esetén (és mert k nem 0) a válasz hamis, egyés esetben pedig a rekurziós ágba lépünk.

A rekurziós ág elején fogadjuk az n darab egész számot a szülőtől, amiket egy n elemű vektorban tárolunk, majd megkísérelünk létrehozni 2 gyerek folyamatot. Amennyiben létrejöttek, az első gyereknek elküldjük a k-t, $n-1$-et, továbbá azt az $n-1$ elemszámú halmazt reprezentáló vektort, amiből kivesszük "véletlenszerűen" az utolsó elemet.[4] A második gyereknek pedig elküldjük a $k - tail$ egész értéket, ahol $tail$ a vektorunk utolsó egész eleme, $n-1$-et, továbbá a vektor utolsó eleme nélküli vektort (Ez a `pop_back` metódussal került megvalósításra, mivel mindkét esetben a pop-olt vektorra van szükség).

Ezután fogadjuk a választ mindkét gyerektől, s válaszolunk a szülőfolyamatnak. (Amennyiben az első gyerek igazat ad, a másodiktól üzenetet nem is szükséges fogadni, hiszen azonnal tudunk üzenni a szülőfolyamatnak, hogy a válasz igaz, egyéb esetben meg kell várni a másik gyerek válasza lesz a szülő válasza. (Ha igaz, akkor igaz, ha hamis akkor hamis))

3.3. Fordítás menete

A programunk forráskódját a `master.cpp` és a `child.cpp` 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

[4]Mivel halmaz elemei között nincs sorrend, ezért a specifikáció véletlenszerűséget fogalmaz meg kiválasztásnál, de az algoritmus implementálása szempontjából hatékonyabb a vektor utolsó elemét, tekinteni mindig a "véletlen" kiválasztott elemnek

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.

3.4. Tesztelés

A program tesztelése során különböző méretű bemeneti fájlokkal, illetve paraméterekkel futtattam. (Az `atlasz.elte.hu` szerver 20 szálas korlátjából adódó lehetőségekhez mérten) A programom minden esetben a tőle elvárt kimenetet állította elő, így a tesztesetek alapján helyesnek gondolhatjuk a működését.

1. Hiányzó paraméterek tesztesete
2. k = 0 triviálisan igaz eset
3. |n| = 0 triviálisan hamis eset
4. k összegű részhalmaz létezése, n=1, n=2, n=3, n=4 esetekre [5]
5. k összegű részhalmaz nem létezése, n=1, n=2, n=3, n=4 esetekre [5]

A tesztelést az `atlasz.elte.hu` szerveren végeztem. Itt az *atlasz* fejgép szimbolizálja az 1 processzormagot. 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:

- k : A számláló érték
- n : Az aktuális halmaz elemszáma ($|n|$)
- T_1 : $k = 0$ eset, triviális igaz válasz, 1 gyermek
- T_2 : $n = 0$ eset triviálisan hamis eset, 1 gyermek
- T_{3_i} : $n = i$ illetve a k egy olyan érték amelyre a program igaz választ ad.($i = 1..4$)
- T_{4_i} : $n = i$ illetve a k egy olyan érték amelyre a program hamis választ ad.($i = 1..4$)
- N/A : Azt jelenti, hogy a szerver nem volt képes kezelni, ezt az esetet.

[5] n=5 eset már $2^n > 20$ szálat produkálna

	T_1	T_2	T_{3_1}	T_{4_1}	T_{3_2}	T_{4_2}	T_{3_3}	T_{4_3}	T_{3_4}	T_{4_4}
1 mag	∼0.1121 mp	∼0.1118 mp	∼0.0229 mp	∼0.0272 mp	∼0.0345 mp	∼0.0345 mp	N/A	N/A	N/A	N/A
2 mag	∼0.0116 mp	∼0.0025 mp	∼0.0148 mp	∼0.0151 mp	∼0.0177 mp	∼0.0263 mp	∼0.0384 mp	∼0.038 mp	N/A	N/A
3 mag	∼0.00358 mp	∼0.0114 mp	∼0.015 mp	∼0.0142 mp	∼0.0178 mp	∼0.0262 mp	∼0.0293 mp	∼0.0375 mp	∼0.0437 mp	∼0.042 mp
7 mag	∼0.00357 mp	∼0.00386 mp	∼0.00816 mp	∼0.0152 mp	∼0.0101 mp	∼0.0189 mp	∼0.0297 mp	∼0.0313 mp	∼0.033 mp	∼0.0343 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 magos megoldás mint a szekvenciális számítással ekvivalensnek tekinthető 1 magos verzió.