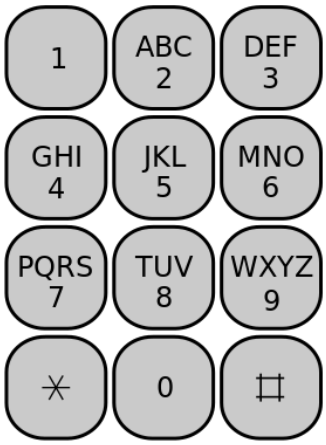
MOBILTELEFONOS ÜZENET SZÁMKÓDJÁNAK VISSZAFEJTÉSE

FELADATLEÍRÁS:

A feladat a képen látható módon számokhoz rendelt betűk alapján elkódolt üzenet visszafejtése. Például a “parallel” szó számkódja: 72725535.



A számsor melyről meg akarjuk állapítani milyen üzenetet tartalmaz, maximum egy SMS-hosszúságú lehet, tehát 160 karakter (ebben még nem vagyok biztos, a 160 lehet nagy vállalás, és túl számításigényessé teszi a feladatot, kezdetben 30 karakteres üzenettel próbálkozom majd). Az üzenet angol nyelven van értelmezve, értelmes angol kifejezésekből áll és az írásjeleket nem tartalmazza. Rendelkezésre áll továbbá egy forrás állomány (txt) melyben az összes értelmes angol szó össze van gyűjtve. A visszafejtendő üzenet tehát csak a szavakat tartalmazza “összeragasztva”, tehát szóközök nélkül.

VISSZAFEJTÉS:

Egy számsorhoz tartozó összes lehetséges értelmes szó-kombinációt vissza kell adni. Tehát pl. ha egy számsor többféleképp is felbontható értelmes szavakra, akkor az összes lehetséges megoldást vissza kell adni. Viszont azzal nem foglalkozik, hogy ezek a szavak együtt értelmes üzenetet tesznek-e ki.

A feladatot alapvetően rekurzív módszerrel lenne érdemes megoldani, viszont kellően hosszúra választva a bemeneti üzenetet, a rekurzív módszer nem lenne optimális. Így párhuzamos módszerrel már rövidebb futási idejű lehet a megoldás. A két megoldást különböző hosszúságú bemenetekre összehasonlítanám futási idő szempontjából.

ÖTLET A PÁRHUZAMOSÍTÁSRA

A bemeneti számsort az összes lehetséges módon felosztom. Ezt részhalmazokra bontom, ~2^{x-1} féle felbontás ha x=üzenet hossza. A felosztásokat úgy generálnám, hogy x..1 közötti hosszúságú elemekre bontom a bemenetet, és az összes lehetséges kombináció szerint lehetséges megoldást, “felosztást” képzek belőle:

Pl: 123 -> {1}{2}{3},{1}{23},{12}{3},{123}

Ezeken a felosztásokon belül vizsgálnám párhuzamosan több szállal az egyes részhalmazok helyességét. Amennyiben egy részhalmaz megfeleltethető egy számsornak akkor az a részhalmaz “validálásra kerül” és ha minden részhalmaz validálva lett, akkor az egy lehetséges megoldás. Amikor egy ilyen “felbontás” rossznak bizonyul: egy részhalmazához nem találunk a dictionary-ben kulcsot, azt el kell dobni mint lehetséges megoldás, többi részhalmazát nem vizsgáljuk tovább. /room for improvement/

Az angol szavakat asszociatív tömbbe rendezem: ahol a kulcs a szóhoz tartozó számsor, és az érték pedig egy lista mely az adott számhoz tartozó szavakat tartalmazza.

```
22476 : bairn cairn cairo
22487 : bahts baits
22493 : baize
22533 : baked baled caked
22537 : abler ables baker bakes balds baler bales cakes calfs
22543 : calif
22549 : calix
22552 : calla
22554 : calli
22556 : ballo
22557 : balks balls calks calls
22559 : balky bally
22566 : cajon
22567 : balms calms calor
22569 : balmy
22572 : balsa
22583 : calve
22586 : cajun
22599 : calyx
22625 : banal canal
```

DICTIONARY – KEY: STRING, VALUE: LIST<STRING>

Ez a gyűjtemény lenne a validálás alapja, a különböző szálak ebből olvasnának. A felbontások halmazát osztanám szét a szálak között, a feldolgozás párhuzamosan futna. A szinkronizáció, terheléselosztás megtervezése itt különösen fontos, mivel egy-egy ilyen felosztás különböző mennyiségű részhalmazból állhat, és az annyival több olvasás a szavak gyűjteményéből.

A MEGVALÓSÍTÁS

SZEKVENCIÁLIS MEGOLDÁS ELEMZÉSE, PÁRHUZAMOSSÁG FELDERÍTÉSE

A feladat szekvenciális megoldása a következő lépésekből áll:

A feladat bizonyos x-hosszúság felett válik számításigényessé, a fentebb említett 2^{x-1} méretű keresési tér miatt.

A 30 karakter hosszú bemenetre már túlságosan erőforrásigényes a feladat, a specifikációban ígértektől így eltérve max. 25 hosszú bemenetekre vizsgálom a továbbikban a programot.

Láthatjuk hogy a 25 karakterre is számottevő a szekvenciális megoldás futási ideje:

```
Run times for input with 25 length
sequential:
elapsed time: 12127 ms
solution count:6837
```

1. FIGURE - SZÁMÍTÁSIGÉNYESSÉG

Egy példa bemenetre

“reallylongmessagetodecode” : 732559566463772438633263 3

a következőképp alakulnak a futási idők a különböző részfeladatokat tekintve:

```
Message size: 25
Search space generated in: 8818 ms
Validate search candidates: 0 ms
Encode possible solutions to messages in: 601 ms
Search space size: 726
Solution candidates size: 726
All solution messages count: 465247
```

1. FIGURE –FUTÁSI IDŐK 25 KARAKTERES INPUTRA

A FELADAT FELBONTÁSA RÉSZEGYSÉGEKRE, DEKOMPOZÍCIÓ

FUNKCIÓK SZERINTI FELBONTÁS:

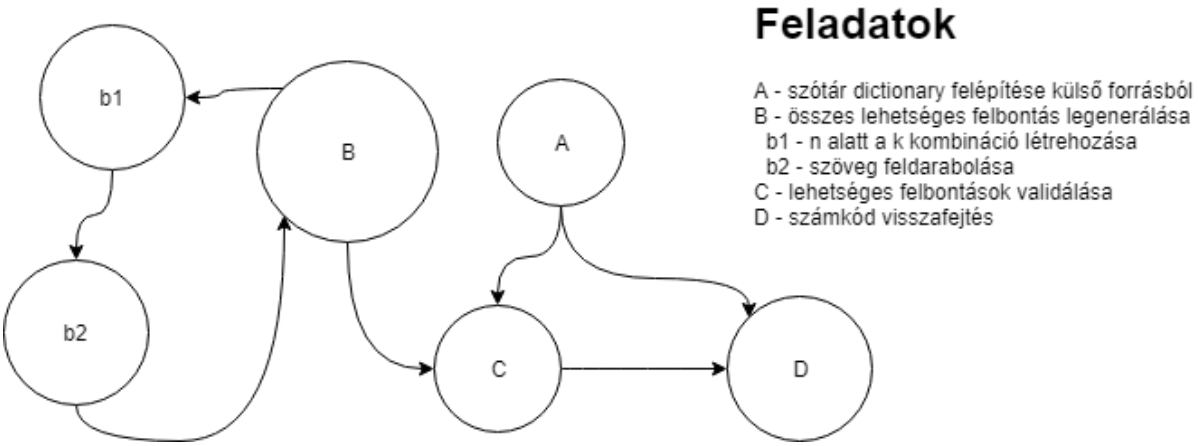


FIGURE 2 - FUNKCIÓK ÁBRÁZOLÁSA IRÁNYÍTOTT GRÁFAL

b1 funkció felel az n (0..n-1) db separator összes lehetséges kombinációjának generálásáért:

“123” : n=0 -> [0]

“123” : n=1 -> [1],[2]

“123” : n=2 -> [1,2]

Ez a részfeladat ciklikusan ismétlődő 0 és n-1 közötti paraméterekkel. Ez a függvény konkurens módon is futhatna, a működése elkülöníthető a különböző “daraboló” azaz separator méret szerint. Ez mind külön részfeladatként definiálható.

b2 funkció felel a separator-ok szerinti tényleges felbontásért

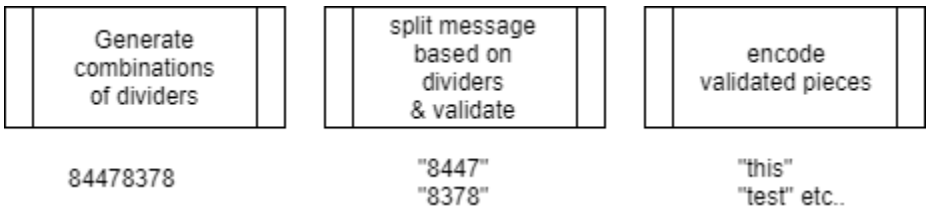
“123” : [0] -> {123}

“123” : [1],[2] -> {1}{23},{12}{3}

“123” : [1,2] -> {1},{2},{3}

B2 funkció is hasonlóan b1 hez, ciklikusan ismétlődő, részfeladatokra bontható. Továbbá a validálás hozzáfűzése is megfontolandó még ezen a szinten, hogy ne kelljen az egész keresési teret újra bejárni. Így a b2 funkció részévé válik a C funkció.

A funkcionális dekompozíció után a következőképp alakul a program felépítése:



3. FIGURE -FUNKCIONÁLIS DEKOMPOZÍCIÓ

Az egyes részfeladatokon belül többnyire egy-egy számításigényes ciklus van.

ADATOK SZERINTI FELBONTÁS

A legszámításigényesebb feladat, a keresési tér generálása, egymástól független adatok alapján számítható ki, aminek egy jelenntős része a darabolók kombinációinak generálása. Ezt felbonthatjuk különböző részegységekre, és azt külön szálak számára kioszthatjuk.

Továbbá, vannak olyan globális adatok, melyet minden szálnak el kell érni, ezek a kommunikációs pontok és ez alapján alakul ki a megoldáshalmaz:

Részhalmazok (“daraboló” generálás): (List<int[]>)

Keresési tér: (List<string[]>)

Szótár: (Dictionary<string,string>) validálás és dekódolás

Dekódolt output: (List<string>)

Ezen központi adatszerkezetekhez minden szál hozzáadja az általa feldolgozott részeladat eredményét, illetve olvas belőlük szükség esetén. Szegmentálással nem igazán osztható szét szálak között egyik adatszerkezet sem.

FELADATOK CSOPORTOSÍTÁSA, SZÁLAKHOZ/FOLYAMATOKHOZ RENDELÉSE

Az alábbiakban ismertetem a két párhuzamosítási tervet, amit alkalmazni fogok:

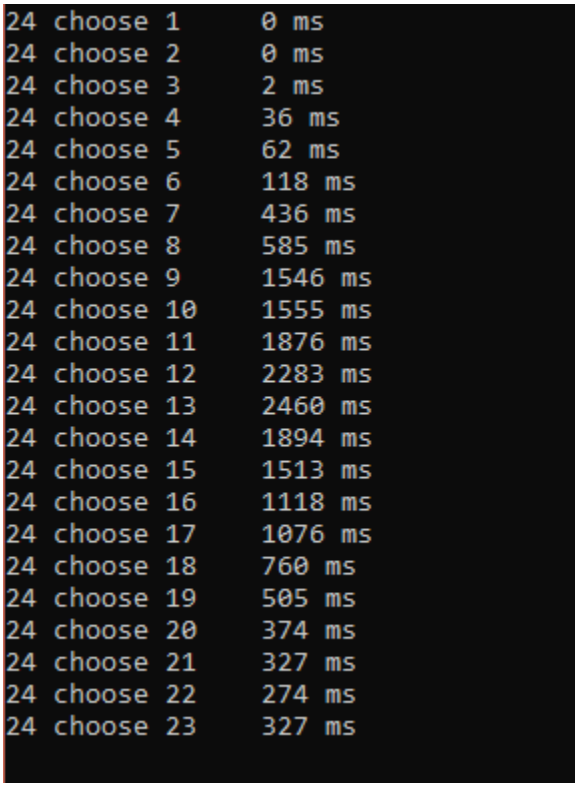
LOOP PARALLELISM

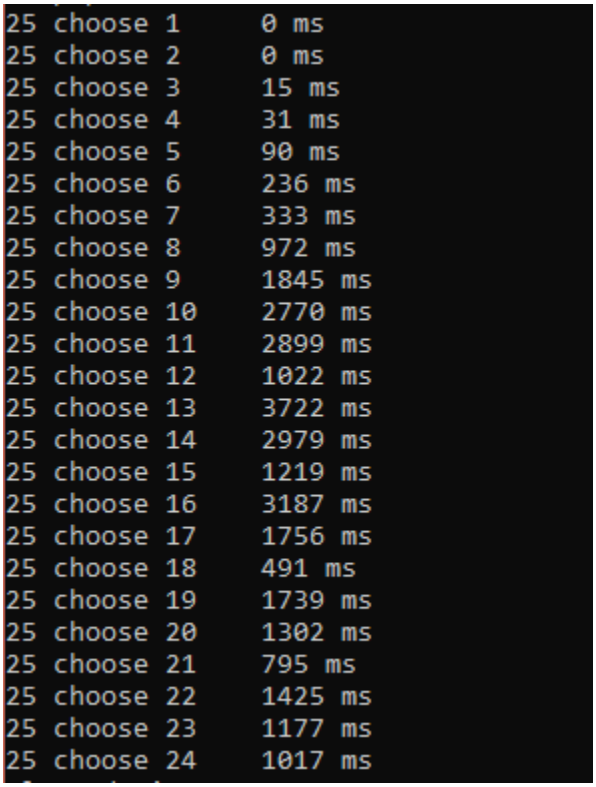
A funkcionális felbontás után nem sok kommunikációra van szükség az egyes szálak között, csak a részeredményeket kell bevárniuk. A feladat 3 számításigényes ciklus által definiálható, így jó megoldásnak ígérkezik a loop parallelism.

A közös adatok használatából fakadó overhead miatt félő, hogy nem lesz hatékony a megoldás. => kiküszöbölés : ezen adatokról másolat a különböző szálak számára?

TERHELÉSELOSZTÁS, LOAD BALANCING

A legszámításigényesebb részfeladat a kombinációk generálása, mely futási idők k függvényében így alakulnak 24, ill. 25 hosszú bemenetre:





Megfigyelhető, hogy körülbelül 3 részre osztható a terhelés mértéke. Az első harmad kis- , a középső harmad a leginkább- és az utolsó harmad közepesen erőforrásigényes.

Egy másik párhuzamosítási terv ez alapján:

Egy függvény állítja majd össze a feladatokat a worker szálak számára, hogy lehetőség szerint mind a 3 típusú kombinációból mind kapjon 1-et.

EREDMÉNYEK ÉRTÉKELÉSE, ÖSSZEGZÉS

A különböző megvalósítások futási ideje az input függvényében:

```
Run times for input with 4 length
sequential:
elapsed time: 6 ms
solution count:0

loop parallelism:
elapsed time: 51 ms
solution count: 0

load balancer with 8 task:
All task has finished
elapsed time: 58 ms
solution count:0
```

```
Run times for input with 9 length
sequential:
elapsed time: 5 ms
solution count:69

loop parallelism:
elapsed time: 59 ms
solution count: 69

load balancer with 8 task:
All task has finished
elapsed time: 45 ms
solution count:69
```

```
Run times for input with 16 length
sequential:
elapsed time: 27 ms
solution count:2643

loop parallelism:
elapsed time: 88 ms
solution count: 2643

load balancer with 8 task:
All task has finished
elapsed time: 17 ms
solution count:2643
```

```
Run times for input with 22 length
sequential:
elapsed time: 1599 ms
solution count:291887

loop parallelism:
elapsed time: 1422 ms
solution count: 291887

load balancer with 8 task:
All task has finished
elapsed time: 922 ms
solution count:291887
```

```
Run times for input with 24 length
sequential:
elapsed time: 26883 ms
solution count:12341696

loop parallelism:
elapsed time: 23572 ms
solution count: 12341696

load balancer with 8 task:
All task has finished
elapsed time: 23758 ms
solution count:12341696
```

```
Run times for input with 25 length
sequential:
elapsed time: 12778 ms
solution count:1935961

loop parallelism:
elapsed time: 12294 ms
solution count: 1935961

load balancer with 8 task:
All task has finished
elapsed time: 10411 ms
solution count:1935961
```

Megfigyelhető, hogy a kis karakterszámú üzenetekre a szekvenciális a legjobb megoldás, mivel nem teljesítenek jobban a többszálú megoldások, túl nagy a szál lérehozásnak és a szinkronizációnak az overheadje.

A loop parallelism megoldás során nem jobb a futási idő mint a szekvenciálisnál, ez amiatt lehet hogy a függőségek miatt az egyes ciklusok között meg kell várni hogy minden szál végezzen, és a load balancing nem felügyelt. Túl durván szemcsézett megoldás.

Ezeket a problémákat megpróbálva kiküszöbölni jött létre a load balancer-es megoldás: Ez a futási idők szempontjából az ideális megoldás a 3 közül.