# Bevezetés

Manapság, az információs társadalom korában, egyre nagyobb jelentőséget kap az adatbányászat, és az információfeldolgozás. A különböző mintaillesztési algoritmusok jelentős szerepet töltenek be többek között a bioinformatika, a keresőmotorok illetve a természetes nyelvfeldolgozás területén is.

Másik, koránt sem elhanyagolható tényező az algoritmusok párhuzamosíthatóságának fontossága: ugyanis jelenleg az egyre nagyobb és nagyobb számítási teljesítményt már nem egy mag órajelének növelésével érik el a processzorgyártók, hanem több, párhuzamosan működő (viszont egyenként gyengébb) mag beépítésével a CPU-ba. Ez azért hasznos, mert így jóval kisebb az energia-felvétel, ugyan akkora (vagy akár nagyobb) névleges teljesítmény mellett. Ennek viszont csak akkor látjuk hasznát, ha az általunk fejlesztett algoritmusok megfelelően vannak párhuzamosítva, ellenkező esetben töredék részét tudjuk csak kihasználni ennek a névleges teljesítménynek.

Önálló laborom során ezzel a két témával foglalkoztam. Két különböző pontos mintaillesztési algoritmust vizsgáltam meg, egyrészt felfedve bennük a párhuzamosíthatósági lehetőségeket, másrészt pedig összehasonlítva a kettőt, megfigyelve, hogy milyen esetben melyik teljesít jobban.

Annak érdekében, hogy az eredményeim ne legyenek túlságosan implementációfüggőek, az algoritmusoknak két megvalósítását mutatom be.

A választásom egyrészt C++ nyelven az Intel Threading Building Blocks megoldására esett, másrészt pedig a Google – még egészen új – programozási nyelvére: a Go-ra.

Ebben a dokumentumban először röviden bemutatom az előzőleg említett két technológiát, majd áttérek a két mintaillesztési algoritmusra. Ezeknél először bemutatom az alap algoritmust, majd a párhuzamosítási lehetőséget, megmutatva a párhuzamosítással elért teljesítménynövekedést mind a két implementáció esetében. Végül pedig következik a két algoritmus összehasonlítása, ahol is levonom a megfelelő konklúziókat, miszerint mikor melyiket érdemes használni.

# Felhasznált platformok

Mind a két felhasznált platform esetében fontos volt, hogy a párhuzamosítás ne az alap (OS szintű) szálkezelést alkalmazza, mivel az erre a felhasználásra túlságosan erőforrás igényes: a szálak menedzselésével sokkal több processzoridő menne el, mint amit megengedhetünk magunknak. Ezzel szemben, egy sokkal finomabb felbontású megoldásra van szükség, ahol – a kifejezéseket a C# TPL-ből átemelve - „taszkokat” ütemezhetünk, amik már meglévő szálakon kerülnek futtatásra („threadpool”), így nem pazarolunk erőforrást a szálak létrehozására és megszüntetésére minden egyes párhuzamosan végzendő teendő esetén.

## Intel Threading Building Blocks (TBB)

Az Intel Threading Building Blocks az Intel nyílt forráskódú, platform független megoldása a párhuzamosításra, egy C++ template osztálykönyvtár formájában. Az 1.0-ás változat 2006-ban jelent meg, azóta már a 4.3-as verziószámnál tart – munkám során én ezzel dolgoztam, méghozzá Windows operációs rendszeren, Visual C++ fordítót alkalmazva.

A C++ STL-ből megszokott templatek logikáját követi, a párhuzamos végrehajtásra szánt programrészeket functorként veszi át (itt akár lambda kifejezést is adhatunk meg), és onnan pedig a felhasználó számára transzparens módon futtatja több szálon a programot: automatikusan felfedezve a szabad feldolgozóegységeket, és az adott futtatáshoz hozzászabva a taszkok ütemezését.

Egy példa a párhuzamos for ciklus működésére:

A hívás helye:

*tbb::parallel\_for(tbb::blocked\_range<int>(0, parts), ParallelMatcher(\*this, p, occurrencesArray, parts));*

És a functor deklarációja:

*class ParallelMatcher {*

*MatchAbleString &mas;*

*const Pattern &p;*

*std::vector<std::vector<int>\*> &array;*

*int parts;*

*public:*

*void operator() (const tbb::blocked\_range<int>& range) const;*

*ParallelMatcher(MatchAbleString &mas, const Pattern &p, std::vector<std::vector<int>\*> &a, int parts) : mas(mas), p(p), array(a), parts(parts) {}*

*};*

Ehhez még hozzátartozik a függvényhívás operátor definíciója. Látható, hogy ha nem használunk lambda kifejezéseket, akkor sajnálatos módon eléggé felduzzad a kód mennyisége, és ráadásul elkülönül a hívás helye, és a futtatott kód, de ez a C++ sajátosságaival jár, így ezt a kompromisszumot meg kell hoznunk. Nehezíti még a használatot a functor függvényhívás operátorának a keretrendszer által kötött argumentumai, de ez áthidalható az osztályban tárolt tagváltozókkal, mivel így a konstruktorban adhatjuk át a kívánt paramétereket.

## A Go programozási nyelv

A Go programozási nyelvet a Google készítette, és 2009-ben jelent meg. (Jelenleg az 1.4.2-es verzió a legújabb, én ezt használom, Windows operációs rendszeren.) Ez egy compiler által fordított nyelv, mely szintaktikájában némi hasonlóságot mutat a C-vel, de azért néhány helyen - szándékosan, a dokumentációban ezen változtatások meg is vannak indokolva - jelentősen eltér. Elkészítésekor kiemelt cél volt átlátható, egyszerű nyelvet létrehozni, így sok (egyébként hasznos) funkciót hagytak ki belőle, amit más nyelvekben láthatunk: így például a genericitás, leszármazás, stb. nem része a nyelvnek. A C-hez képest viszont több támogatást ad, rendelkezésünkre állnak nyelvi elemként többek között a beépített string és változó méretű tömbök („slice”) is.

Számomra a leglényegesebb beépített nyelvi elem itt a string támogatás volt (teljes string, részstringek kezelése – ezeken összehasonlító funkciók hatékonyan, miközben a memóriában a string csupán egyszer szerepel), és a taszk alapú párhuzamosság („goroutines”) egyszerű megvalósítása.

Itt látható egy párhuzamosan futó for ciklus megvalósítása Go nyelven:

*ch := make(chan int)*

*for i := 0; i < parts; i++ {*

*go matchSubstr\_(str, pat, strlen, i, parts, occurrencesArray, ch)*

*}*

*for i := 0; i < parts; i++ {*

*size += <- ch*

*}*

Maga a párhuzamos függvényhívás csupán annyiban különbözik a szinkron változatától, hogy elé írjuk a go kulcsszót: a matchSubstr\_ függvény végig ugyan az marad, nem változnak a paraméterek, semmi. Az aszinkron hívások bevárásához van szükség a „channel”-re és a második for ciklusra: megvárjuk, amíg az összes hívott függvény beleszól a csatornába (itt pluszban még a csatornában hallott értéket fel is használjuk).

Látható, hogy a C++-os változathoz képest lényegesen rövidebb és egyszerűbb a kód, de ez annak tudható be, hogy míg ezt a nyelvet már szándékosan úgy alkották meg, hogy a párhuzamosság beépített nyelvi elem, addig a C++ megalkotása idején erről szó sem volt, ráadásul az Intel TBB még egy külső függvénykönyvtár is.

Sajnos ezen előnyeivel szemben azért hátránya is van a nyelvnek: mivel még fiatal, kiforratlan, több megoldásra váró problémát is felvet. Egyik ezek közül, hogy jelenleg még egy viszonylag kezdetleges garbage collectorral dolgozik. Szerencsére ez azoknál a stringméreteknél amivel én dolgozom, nem jelent problémát, észre sem vehető.

# Az algoritmusok

A megvizsgált algoritmusokban az a közös, hogy mind a kettő pontos mintaillesztést végez, mindkettő megkeresi az összes előforduló illeszkedést, és az ezeket tartalmazó listával tér vissza. Alapvetően mind a kettő egyszálú végrehajtással működött, melyhez megvizsgáltam a párhuzamosíthatósági lehetőségeket, és megvalósítottam egy többszálú változatot is.

A különbség az, míg a Suffix Array algoritmus esetén azon a stringen, amin az illeszkedéseket keressük, preprocesszálást végzünk, amiért cserébe a mintaillesztések gyorsak lehetnek, a Boyer-Moore algoritmus esetén ilyen nincs, ott csak a mintán végzünk némi előfeldolgozást (jellemzően sokkal rövidebb string – sokkal kevesebb idő), cserébe ott a mintaillesztések egyenként hosszabb időt vesznek igénybe.

A méréseket egy Intel Core i7 2630QM (4 fizikai, 8 logikai mag) processzoron végeztem.

## Suffix Array algoritmus

Ez az algoritmus lényegében arról szól, hogy amikor megkapjuk az S stringet (az eredeti string, amin majd a mintaillesztéseket végezzük), akkor egy előfeldolgozási procedúrában elkészítjük hozzá a suffix array-t, amin onnantól kezdve már igen hatékonyan tudunk mintaillesztést végezni.

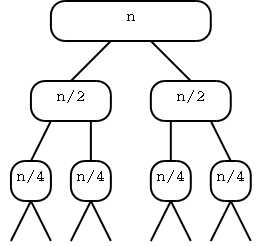
A suffix array egy rendezett tömb, amiben a string „utótagokat” tároljuk betűrendben: például a 4 betűs „word” szó esetében ilyenből 4 darab van: „word”, „ord”, „rd” és „d”. (Ez rendezve pedig a következőképp szerepel: „d”, „ord”, „rd”, „word”.)

Ha elkészítettük ezt a tömböt, akkor az összes illeszkedés megkereséséhez csupán két darab bináris keresést kell elvégeznünk: egyet a tömbben található legelső illeszkedés megkereséséhez, egyet pedig az utolsó megkereséséhez. Ha meg van ez a két elem, akkor annyi a teendőnk, hogy a tömbben a két elem között található minden suffix kezdőpontját visszaadjuk: ugyanis ezek lesznek az illeszkedések kezdőhelyei.

A bináris keresés miatt a mintakeresés  időben végezhető, így nagyon hatékony. Ami időigényes, az a suffix array felépítése: itt a rendezés miatt  időben tudunk csak dolgozni. Említésre méltó még a tömb memóriában foglalt helye: ez , mivel n darab pointert / integert tárolunk el, ami a suffix kezdőpontját jelzi a stringben.

Párhuzamosíthatóságnak az előfeldolgozás esetén van jelentősége: a mintaillesztés önmagában is elegendően hatékony. Az előfeldolgozás esetén viszont a rendezést végezhetjük párhuzamosan: mégpedig a párhuzamosan végzett merge sort segítségével.

Ahogy az *1.ábrán* is láthatjuk: ennek az algoritmusnak az a lényege, hogy kettéosztja a rendezendő tömböt, a két részt külön rendezi (rekurzívan), majd összefésüli a két rendezett résztömböt. Ez a párhuzamosítási szempontból azért jó, mert a kapott részek rendezését nyugodtan rábízhatjuk két külön feldolgozóegységre, ezek futása nem befolyásolja egymást, majd pedig ha mind a két egység elkészült, elvégezhetjük az összefésülést. A rekurziónak itt az szab gátat, hogy ha elegendően kicsi résztömböt kaptunk – vagy ha már annyi részre osztottuk a tömböt, hogy elfogytak a feldolgozóegységeink. Így azon a helyen már nem rekurzívan hívjuk a rendezőfüggvényünket, hanem egy tetszőleges – egyszálú – algoritmust alkalmazva rendezzük a résztömböt.



1. ábra

Elkészítve az algoritmust mind a két nyelven, méréseket végeztem a teljesítményükről. Mind a két esetben különböző hosszúságú angol szövegeken végzem el az előfeldolgozást, és az egyszálú algoritmus jelenti az alap szintet, amihez képest a párhuzamos változat teljesítménynövekedését vizsgálom. A mérési eredmények a *2. (C++)* és a *3. (Go) ábrán* láthatóak.

2. ábra (C++)

3. ábra (Go)

Látható, hogy jelen esetben a Go nyelv jobban teljesít: kétszeres stringméretek mellett produkál hasonló időértékeket, mint a C++ változat. Ennek oka abban keresendő, hogy a Go nyelv elkészítésénél fontos cél volt a hatékony stringműveletek végzése (mivel a Google berkeiben ez egy igen sokat használt funkció), és itt a beépített stringműveleteket lehetett használni (hisz a suffix array itt csupán résztömbök – „sliceok” – tömbje). Ezzel szemben a C++-os változat esetén a részstringek kezeléséhez char \* nullterminált stringeket kellett alkalmazni, és a C-s srtingfüggvényeket alkalmazni.

A Go nyelv esetében a párhuzamosítással 2,6 körüli speedup adódott, a C++-nál 1,5.

## Boyer - Moore algoritmus

Mint ahogy már előzőleg említésre került, ez az algoritmus nem végez előfeldolgozást azon a stringen, amire a mintát illesztjük, csupán a mintán. Ezáltal sokkal kisebb overhead-et jelent kevés mintaillesztés esetén, cserébe maga a mintaillesztés időigényesebb.

Az algoritmus pszeudokódja a következő (az eredeti cikkből idézve):

*stringlen 🡨 a string hossza*

*i 🡨 patlen*

*top: if i > stringlen then return false*

*j 🡨 patlen*

*loop: if j = 0 then return j + 1*

*if string(i) = pat(j)*

*then*

*j 🡨 j – 1*

*i 🡨 i – 1*

*goto loop*

*close*

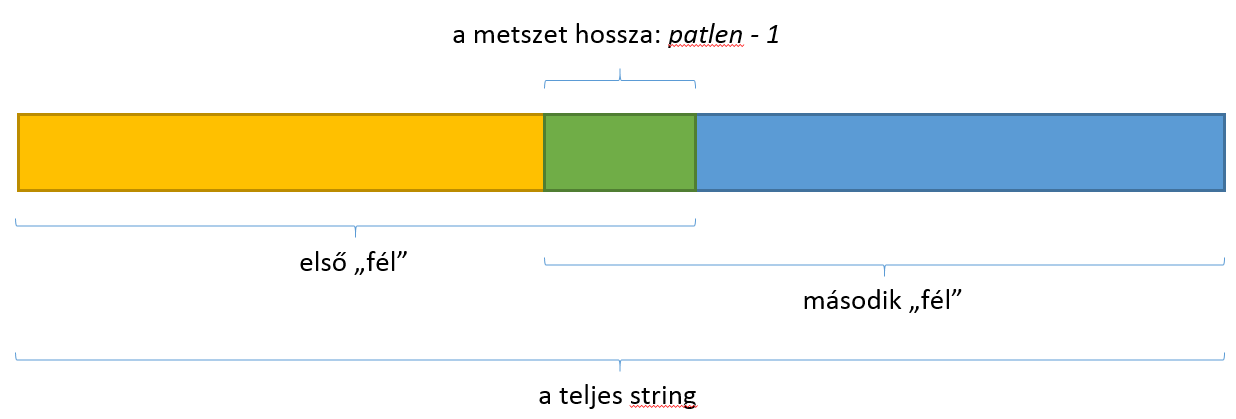
*i 🡨 i + max(delta1(string(i)), delta2(j))*

*goto top.*

Itt „string” az eredeti szöveg (amin a mintaillesztést végezzük), „pat” pedig az illesztett minta. A „delta1()” és „delta2()” függvények pedig a minta előfeldolgozásából keletkezett adatok, ezek teszik lehetővé az algoritmus számára, hogy ne minden egyes karaktert kelljen végigvizsgálnia, hanem nagyobb „ugrásokkal” haladhasson a szövegben.

Az algoritmus ebben a formájában még nem felelt meg az én elvárásaimnak: csupán az első találatot szolgáltatja. Azonban ezt át lehet hidalni azzal a trükkel, hogy ha visszatér az első találattal, akkor újra meghívjuk – immáron a stringnek a maradékával.

Párhuzamosításhoz azt az ötletet használtam fel, hogy ha több részre osztjuk fel a szöveget, akkor a részek egyenként nem függenek egymástól, így azokon a keresést lehet külön szálon futtatni. Azonban nem szabad elfeledkezni arról, hogy ha simán kettéosztjuk, akkor a kettévágás helyén lévő esetleges találatot elveszítenénk: így a *4. ábrán* lévő megoldáshoz folyamodtam: így semmiképpen sem veszítünk el találatot, és többször sem találjuk meg ugyan azt.



4. ábra

Ebben az esetben is elkészítettem mind a két nyelven az implementációt. Kétféle mérést is végeztem: az egyik bemutatja az algoritmus egy sajátosságát, miszerint minél hosszabb mintát illesztünk, annál nagyobb „ugrásokkal” tud haladni, éppen ezért annál gyorsabb lesz a működése. A másik féle mérés pedig a megszokott, egyszálú és párhuzamos változat összehasonlítása.

5. ábra(C++)

6. ábra(Go)

Az ábrákon látható, hogy ennél az algoritmusnál a két nyelv közel azonosan teljesít. Ennek oka itt az, hogy az algoritmus során nem jön ki a Go előnye a stringműveletek terén, hiszen itt karakterek összehasonlítását és tömbökben való lépkedést végzünk túlnyomó részben.

A „short pattern” mind a két esetben 9-11 karakteres angol szót jelent, a „long pattern” pedig 20 karaketeres, több szóból álló mintát. Észrevehetjük, hogy kétszeresére nőtt minta mellett nagyjából fele annyi időbe telt a mintaillesztés: azaz a minta hosszával lineárisan gyorsul az algoritmus.

A párhuzamosításból származó speedup a Go esetében 4 körüli, a C++ esetében pedig 6 körülinek adódott.