Eötvös Loránd Tudományegyetem

Informatikai Kar

Komputeralgebra Tanszék

Párhuzamos sztringalgoritmusok és kapcsolódó adatszerkezetek

Témavezető:
Burcsi Péter
Egyetemi docens, PhD
Komputeralgebra tanszék

Szerző:
Megyesi Attila
Programtervező informatikus
MSC

Budapest, 2018

Tartalomjegyzék

1. fejezet

Bevezető

1.1. Motiváció

A mai világban a feldolgozandó adatok egyre nagyobb méretet öltenek. Gyakran már az adatok tárolása is problémát okozhat. Az adatok önmagukban semmit sem érnek, ezeket fel kell dolgozni ahhoz, hogy hasznosítható információt szolgáltassanak. A technológia egyre inkább az adatok elosztott feldolgozása felé hajlik, azonban gyakran előfordul, hogy szükségünk van olyan algoritmusokra és adatszerkezetekre, amelyek egy számítógép kapacitásait maximalizálják.

Dolgozatomban néhány bioinformatikában használt és felkapott sztring algoritmust tekintek át, megvizsgálva azok korlátait, optimalizációs lehetőségeit és párhuzamosítási lehetőségeit.

A dolgozat elkészítését több dolog is motiválja:

1. Az bioinformatikában használt sztring algoritmusok bemutató, egymást összehasonlító magyar nyelvű elérhető mű hiánya.

2. Az egyetemen tanult párhuzamossági- és memória felépítési ismeretek vizsgálata az adott sztring algoritmusokban.

3. A feldolgozott cikkekben használt párhuzamos algoritmusok és adatszerkezetek megismerése, megértése és elsajátítása.

4. Egy kiválasztott algoritmus párhuzamosítási lehetőségeinek vizsgálata, implementálása c++ nyelven.

1.2. Dolgozat felépítése

A dolgozat 2. fejezetében bemutatom a szuffix sorozat, a Burrows-Wheeler transzformáció és a leghosszabb közös prefix(ismertebb angol nevén longest common prefix, LCP) adatszerkezetet, illetve ezeknek naiv algoritmusait. Előtérbe kerül ezen algoritmusok felhasználási területe is.

A 3. fejezetben bevezetem a nagy mennyiségű adatok feldolgozásánál előkerülő nehézségeket. Tárgyalom a párhuzamos programok szükségességét és a vele járó bonyodalmakat. Különös figyelmet kap a számítógépek memória kapacitása és annak áthidalása. Bemutatom a c++ nyelv által kínált párhuzamosítási lehetőségeket.

A 4. fejezetben részletesen vizsgálom az említett algoritmusokban használt ötleteket,optimalizációkat,párhuzamosítási megoldásokat és futási- és tárigényüket. Továbbá bemutatom egy algoritmus általam javasolt módosítását illetve annak párhuzamosítás implementációját.

Az 5. fejezetben közlöm az általam készített algoritmus mérések alapján meghatározott optimális paramétereit futási időre és tárigényre nézve, a különböző párhuzamosítási módszerek hatékonyságát, illetve azoknak karbantarthatóságát. Összehasonlítom a külső memóriás algoritmusok(pSAscan, eSAIS) futási idejét.

Az utolsó fejezetben javaslatot teszek 1-1 megoldandó feladat paraméterterei által az optimális algoritmus kiválasztására. Továbbá levonom a következtetéseket a feldolgozott algoritmusokból.

1.3. Köszönetnyilvánítás

Szeretnék köszönetet mondani dr. Burcsi Péter Egyetemi Docensnek, hogy segítségemre volt a dolgozat elkészítésében. Köszönet illeti, hogy szakirodalmat biztosított, segített az algoritmusok megértésében és tanácsokat adott a dolgozat felépítésében és elkészítésében.

2. fejezet

Sztring adatszerkezetek ismertetése

2.1. Szuffix sorozat

A szuffix tömb vagy szuffix sorozat,angol nevén suffix array, továbbiakban SA, egy olyan sorozat, mely egy sztring szuffixeinek indexét tartalmazza a sztring szuffixeinek lexikografikus rendezése alapján. Gyakran használják az ún. teljes szöveg indexelésére (angolul full-text indexes), tömörítő algoritmusokban (bzip2 program csomag) illetve bioinformatikai algoritmusokban (például genomok elemzésére[1]), végül de nem utolsó sorban számos leghosszabb közös prefix algoritmus (angol nevén longest common prefix,LCP) alapját képezi.

Legyen S = S[1]S[2]...S[n] egy sztring, továbbá S[i,j] az S sztring i-től j-ig tartó részsztringje. Legyen A egy egész számokból álló sorozat, mely az [1..n] számok egy olyan permutációját tartalmazza, amelyre S[A[i-1]..n] $<$ S[A[i]..n] minden $1 < i \leq$ n-re. A sztring végére egy extremális karaktert fűzünk, amelyik nem szerepel a sztringben. Az irodalomban ez a karakter általában a $ karakter.

2.1.1. Példa. *Legyen az S = nagymama$. Az 2.1 táblázat bal oldalán láthatjuk az S sztringünknek az összes szuffixét. A szuffixhez tartozó indexet a szuffix kezdőbetűjének sztringbeli poziciója határozza meg. A 2.1 táblázat jobb oldalán látjuk a szuffixet lexikografikus rendezését az indexekkel együtt. A lexikografikus rendezéshez tartozó index oszlop adja a szuffix sorozatot.*

Szuffix	Index
nagymama$	1
agymama$	2
gymama$	3
ymama$	4
mama$	5
ama$	6
ma$	7
a$	8
$	9

Szuffix	Index
$	9
a$	8
agymama$	2
ama$	6
gymama$	3
ma$	7
mama$	5
nagymama$	1
ymama$	4

2.1. táblázat. Rendezetlen és rendezett szuffixek

A 2.2 táblázatban láthatjuk a nagymama$ sztringhet tartozó A szuffix sorozatot.

i	0	1	2	3	4	5	6	7	8
A[i]	8	7	1	5	2	6	4	0	3

2.2. táblázat. Szuffix sorozat

2.2. Burrows-Wheeler transzformáció

A Burrows-Wheeler transzformáció (továbbiakban BWT) alkalmazási területei nagyon hasonlóak a szuffix sorozat alkalmazási területével. A bzip2 tömörítő csomag alapját képezi, több szövegindexelési algoritmus is használja, továbbá genomok szekvencializálásában is fő szerepet játszik [3].

Legyen T ≡ T[0..n-1], ahol T[n-1] = $ lexikografikusan kisebb az összes T-beli karakternél. Az $\sum$ ábécé számossága korlátlan. A BWT legkézenfekvőbb definiálása n darab eltolással történik.Az így előállt szavakat lexikografikusan rendezzük, majd a rendezett szavak utolsó betűit egymás után fűzve és a sorrendet megtartva kapjuk a Burrows-Wheeler transzformáltat.

2.2.1. Példa. *A 2.3 táblázat első oszlopában látható, hogy minden egyes körben egyel körbe toljuk a sztringünket. A második oszlopban lexikografikusan rendezzük a létrejött szuffixeket. A harmadik oszlop a második oszlop minden egyes sorának utolsó karakterét tartalmazza. Az utolsó oszlop karakterei alkotják a Burrows-Wheeler transzformációt.*

Körbetolt	Rendezett körbetolt	BWT
almafa**$**	$almafa	a
a**$**almaf	a$almaf	f
fa**$**alma	afa$alm	m
afa**$**alm	almafa$	$
mafa**$**al	fa$alma	a
lmafa**$**a	lmafa$a	a
almafa**$**	mafa$al	l

2.3. táblázat. Burrows-Wheeler transzformáció

2.3. Kapcsolat az SA és a BWT között

2.3.1. Állítás. *A r.-dik karaktere a Burrows-Wheeler transzformációnak T[j-1], akkor és csak akkor, ha a T[j..n-1] szuffix az r.-dik szuffix pozíciót foglalja el a szuffixek lexikografikus rendezésben.*

A fenti állításnak köszönhetően a Burrows-Wheeler transzformáció könnyen előállítható a szuffix sorozat segítségével. Továbbá megjegyzem, hogy a BWT invertálható, tehát egyértelműen előállítható az eredeti sztring, feltéve, hogy egy extremális elemre végződik (jelen esetben a $). Az invertálás algoritmusait nem vizsgáltam a diplomamunkámban.

2.4. Leghosszabb közös prefix - LCP

A szuffix sorozatok alkalmazása során gyakran szükség van a leghosszabb közös prefix nevű adatszerkezetre, angol nevén longest common prefix (innen az LCP rövidítés).Az LCP két egymást lexikografikusan követő szuffix sorozat leghosszabb prefixe.

2.4.1. Értelmezés. $\forall i, j \in [0..n-1]$*: lcp(i,j) = a T[i..n-1] és T[j..n-1] szuffixek leghosszabb közös prefixének hossza.*

2.4.2. Értelmezés. $\forall i \in [1..n-1]$*: LCP[i] = lcp(SA[i],SA[i-1]).*

2.4.3. Példa. *Az T=cccccatcat sztringben az lcp(0,3)=2, mivel a 0. szuffix cccccatcat és a 3. szuffix ccatcat leghosszabb közös preffixének hossza 2.*

Legyen a T=NAGYMAMA, a 2.1 táblázatból már ismert szó és annak szuffix sorozata. A 2.4 táblázat lcp oszlopában látható az abban a sorban és a megelőző sorban lévő szuffix leghosszabb közös prefixének hossza.

Szuffix	Index	lcp
a	7	
agymama	1	1
ama	5	1
gymama	2	0
ma	6	0
mama	4	2
nagymama	0	0
ymama	3	0

2.4. táblázat. lcp értékek számítása

Ekkor az LCP tömb 2.5 táblázatban látható módon fog alakulni.

i	1	2	3	4	5	6	7	8
SA[i]	7	1	5	2	6	4	0	3
LCP[i]		1	1	0	0	2	0	0

2.5. táblázat. Szuffix sorozat és LCP tömb

3. fejezet

Az algoritmusok korlátai

Az fent bemutatott feladatokat megoldó algoritmusok váltakozó hosszúságú inputokra kell felkészüljenek. Előfordul, hogy csak egy pár megabyte-s fájlt szeretnék tömöríteni, de a bioinformatikában vagy a szöveg indexelésben előfordul, hogy több száz gigabyte hosszúságú bemenetekre kell hatékony megoldást nyújtsanak.

A fent említett probléma miatt nem is létezik optimális algoritmus, ezért több megközelítést dolgoztak ki a különböző méretű bemenetek kezelésére.

Az első szűk keresztmetszet a számítógép memóriája. A legtöbb algoritmus segédadatszerkezeteket használ, melyek az eredeti szöveg többszörösét is elérhetik, annak érdekében, hogy optimális futási időt nyújtsanak. Olyan szituáció is előfordul, amikor már a bemeneti szöveg sem fér el a memóriában vagy a segédadatszerkezetekkel együtt meghaladja a RAM mértékét. Ilyenkor úgynevezett külső memóriás algoritmusokat alkalmaznak.

A dolgozatban a belső memóriás algoritmus alatt, olyan algoritmust értek, ami a közvetlen elérésű - angol nevén random acces memory (RAM) - memóriában dolgozik csak és az általa használt adatszerkezetek is elférnek a RAM-ban.

A külső memóriás algoritmus olyan megközelítést jelent, mely a RAM méretbeli korlátai miatt, kénytelen átmenetileg a háttértárolóra menteni bizonyos adatokat.

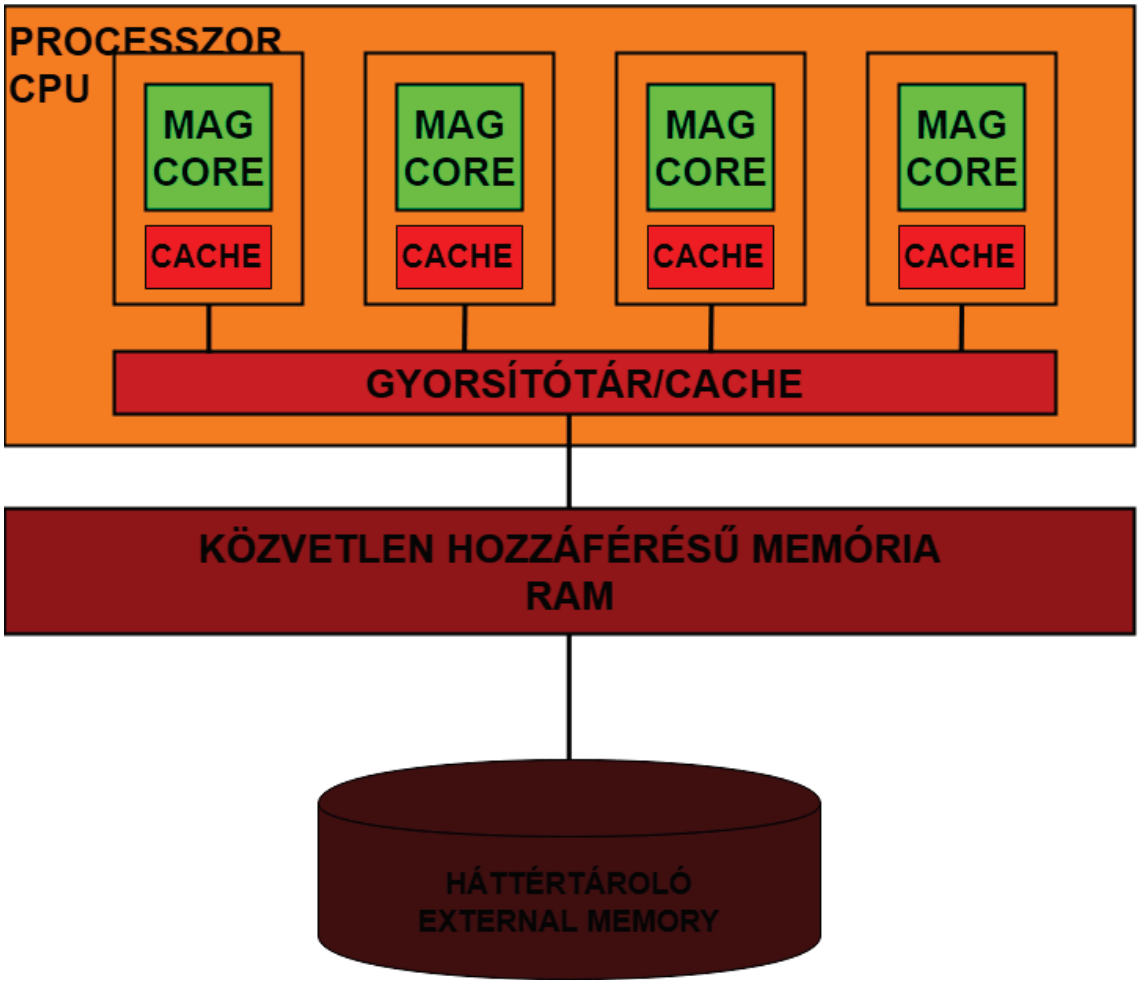
Az mai számítógépekben olyan processzorok találhatóak, amelyek több számítási egységgel, azaz maggal rendelkeznek. Ezek egyidejű használata, vagyis párhuzamos algoritmusok tervezése és implementálása ugyancsak csökkenteni tudja a futási időt. A dolgozatomnak nem célja a grafikus kártyákon futtatható algoritmusok elemzése.

Mint a számítógép memória felépítése, mint a párhuzamosítási lehetőség kihasználása különös figyelmet igényel az algoritmusok tervezése során. A következő két részben bemutatom, hogy miért.

3.1. Külső memória

Szeretném megjegyezni ezen a ponton, hogy bár a dolgozatomban nem szerepel a külső memóriás algoritmusok részleteinek bemutatása, mégsem mehetek el ezek mellett a részletek mellett, ugyanis nem hatékony háttértárhasználat mellett, a belső memóriában a párhuzamosítás semmit sem ér. Például, ha a futási idő 80%-a I/O műveletekből áll, akkor a futási idő maximum 20%-a párhuzamosítható, ami nagyon kevés, szinte elhanyagolható. Továbbá a párhuzamosítás során, a számítási egységekhez közel levő gyorsítótárak felépítése és szerepe is fontos.

A mai számítógépeknek a memória felépítése több szintből áll. A 3.1 ábra szemlélteti a legelterjedtebb felépítés egyszerűsített változatát. Az ábrán zöld szinnel jelöltem a processzorban lévő magokat, melyek képesek a számításaink elvégzésére. A ábrán a piros különböző árnyalataival jelölt gyorsítótár, közvetlen hozzáférésű memória(RAM) illetve háttértároló alkalmas az adatok rövid vagy hosszútávú tárolására. Ezekről a helyekből fogja a processzor az adatokat "megszerezni" a számításokhoz. A különböző tárhelyek elérési ideje nő azok processzotról mért távolságával.



3.1. ábra. Memória felépítése[20]

A cache elérése a leggyorsabb. Mai számítógépekben általában több cache szint is van. Ezeket L1,L2,L3-mal szokták jelölni az irodalomban. Általában nagyon kicsi méretűek. Pár kilobájttól pár megabájtig terjednek a cache szinttől függően. A cache kicsi mérete csak a lokalitás elvének kihasználását segíti: ha a processzornak szüksége van valamilyen adatra a RAM-ból, akkor nem csak azt az adatot olvassa be a RAM-ból, hanem annak környezetét is, feltételezve, hogy a környezetére is szükség lesz előbb vagy utóbb. Továbbá a kiírást is csak a cache-be teszi, majd különböző logikákat követve a cache vezérlő szinkronizálja ezeket a RAM-mal. Ha az adat olvasásnál nem található a gyorsítótárban, gyorsítótár hibának vagy cache miss-nek nevezzük.

A RAM elérése lassabb, mint a cache elérése, azonban gyorsabb, mint a háttértárolóé. A mai általános célű számítógépek RAM mérete általában 8-32 gigabyte, azonban vannak 64-128 gigabyte-os RAM-ok is. Egy bejegyzés alapján [15] a RAM elérése nanomásodpercben mérhető.

A háttértároló elérése a már említett bejegyzés[15] alapján millimásodperc nagyságrendű. Ez azt jelenti, hogy jóval több időre van szükség, hogy a processzortól a merevlemezre kerüljön az információ, mint a RAM-ba. A dolgozatomban több helyen hivatkozok I/O műveletekre a tömörség és az érthetőség érdekében. Ez alatt a háttértárolóra való írást és olvasást értem.

A fentebbi leírásból látszik, hogy az egyes tárhelyek elérési ideje között, nagyságrendbeli különbségek vannak, ezért az algoritmusok tervezése során erre különös figyelmet kell fordítani.

3.2. Processzor - Párhuzamosság

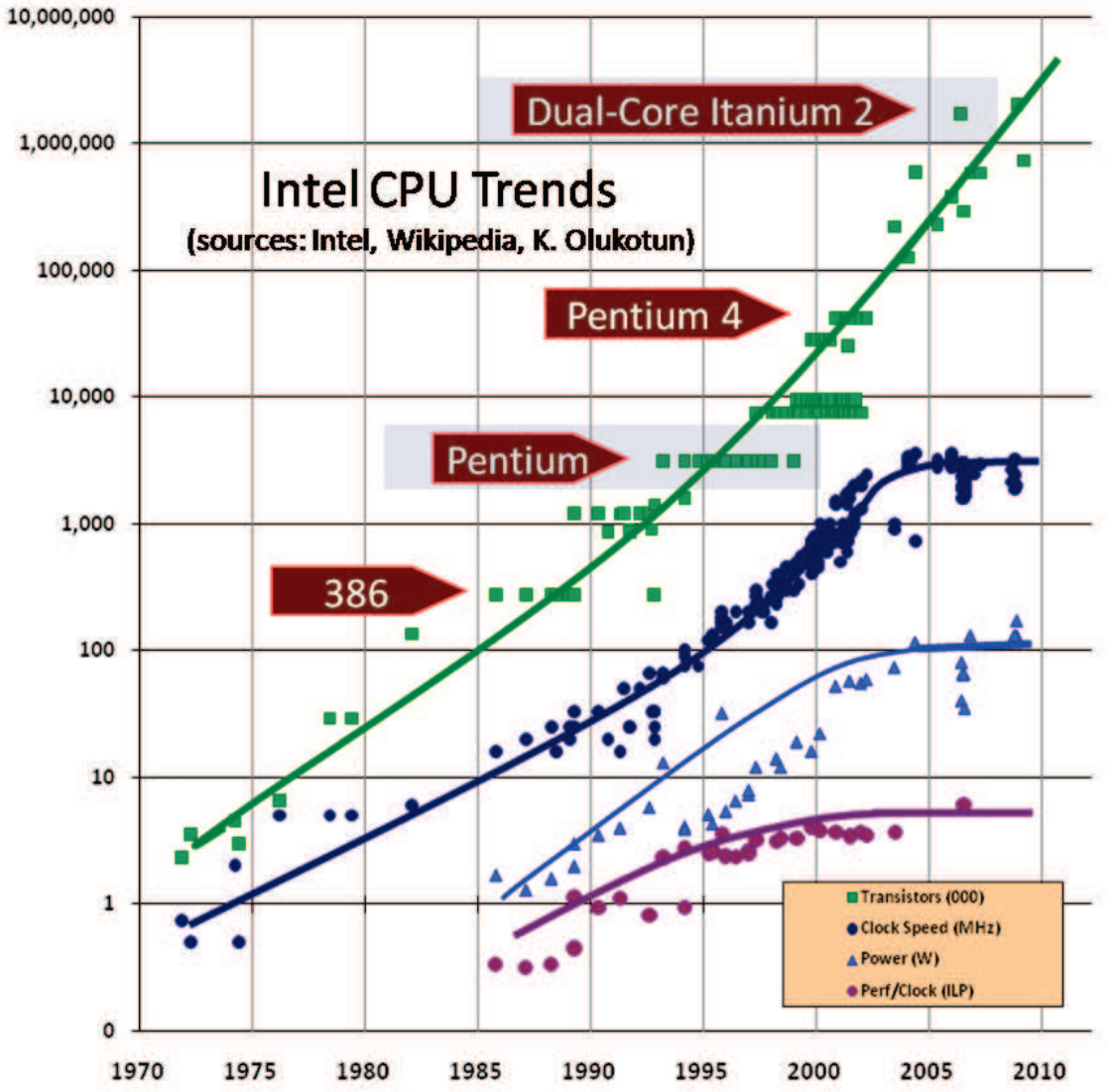
Ebben a fejezetben a fogalmak definiálása nem teljes, hisz ezeknek a fogalmaknak a definiálás egy teljes diplomamunkával is felér.

3.2.1. Processzorok

Az algoritmusok sebességbeli hatékonysága függ a processzorok gyorsaságától, pontosabban a processzorok órajelétől. Gordon E. Moore az Electronics Magazine 1965. április 19-i számában tett egy kijelentést a tranzisztorok számának növekedését illetően, melyről egy törvényt neveztek el: az integrált áramkörökben lévő tranzisztorok száma egységnyi területen – ami használható a számítási teljesítmény durva mérésére – minden 18. hónapban megduplázódik. A 3.2 ábrán zöld színnel a tranzisztorok száma látható, azonban itt nem az egységnyi területen lévő tranzisztorok számát, hanem a processzorban lévő tranzisztorok számát ábrázolják. A Moore törvényben szereplő tranzisztorok száma szoros kapcsolatban van az ábrán kékkel jelölt processzorok órajelével. A Moore törvény, körülbelül 2003-ig álta az időt ugyanis, ahogy az ábra mutatja a processzorok órajelének növekedése ebben évben kezdett eltérni a jóslattól. Ennek több oka is van, melyeket nem részletezek csak említést teszek róluk: hűtés, energia fogyasztás, gyártási költség [22]. Annak érdekében, hogy a gyártók megtartsák a sebességbeli növekedést új módszerhez fordultak. Több magot tettek a processzorokba, alacsonyabb órajellel ugyan, de több maggal tudták biztosítani a számítási kapacitás növekedését.

3.2.2. Párhuzamos program

A több mag megjelenésével együtt szükség volt újabb megközelítésű algoritmusokra ahhoz, hogy kitudjuk használni azokat. Párhuzamos programról beszélünk abban az esetben, ha több szál saját előforráson egyidejűleg hajtja végre a feladatát. A szálak egymás között úgynevezett osztott memórián keresztül tudnak kommunikálni, ami azt jelenti, hogy több mag írja és olvassa ugyan azt a memóriát. Azt az esetet, amikor több mag használja ugyan azt az erőforrást, és ezek közül legalább az egyik írja, versenyhelyzetnek nevezzük. Ezek a helyzetek -a helyes működés érdekében-



3.2. ábra. Processzorok történelme[22]

szinkronizálást igényelnek. Amennyiben a programunkban az erőforrások elérése determinisztikus párhuzamos programozásról, különben konkurens programozásról beszélünk. Vagyis, ha egy változót két szál különböző sorrendben írhat, akkor konkurens, programozásról, ha a változót meghatározott sorrendben érik el, párhuzamos programozásról beszélünk. Egy program tartalmazhat párhuzamos és konkurens részeket is.

Amdahl törvénye kimondja,hogy mennyi lehet a maximum gyorsulása egy párhuzamos programnak a feladat párhuzamosíthatóságának(F) és a szálak számának(p) arányában.Amennyiben megtudjuk becsülni a feladatunk párhuzamosíthatóságát, és ismerjük a processzorunkban levő magok számát, úgy az alábbi képlettel tudunk felső becslést adni a programunk gyorsulására.[5]

$$\text{Amdahl törvénye: gyorsulás} \leq \frac{1}{1 - F + \frac{1}{p}}$$

A szinkronizációkor a szálak kölcsönösen kizárják egymást, hogy egyszerre csak egyikőjük férjen hozzá az adott memóriaterülethez. A szinkronizáció költséges, ál-

talában zárolással szokták megoldani. A szinkronizáció biztosítja, hogy minden szál a ott lévő valós értéket lássa az adott memóriaterületen[21].

Az "A előbb-történt-mint B" (A happens-before B) reláció biztosítja [21], hogy az A által végrehajtott akció eredménye látható B számára. Ezt a relációt szokták alkalmazni bizonyos tulajdonságok leírására.Amennyiben az erőforrások elérésére ilyen relációkat definiálunk, párhuzamos programot kapunk.

3.2.3. C++11 - többszálúság

A tanulmányozott algoritmusok c++ nyelven készültek, ahogy az általam készített implementációk is. A c++ 2011-ben bevezetett szabványban definiálták a több szálúságot. Ezzel lehetővé tették nyelvi szinten több szálon futó, párhuzamos programokat definiálását.Bevezették a szál-,atomi-,mutex-,zárolás-,future-,promise- és async objektumokat, melyek felhasználásával helyes párhuzamos programokat készíthetünk.

A már előbb említett "happens-before" reláció több esetben teljesül, számunkra a legfontosabbak:

1. Új szál objektum (std::thread) létrehozása és annak végrehajtása között.

2. A szál végrehajtásának befejeződése és join() metódusának visszatérése között.

Az implementáció során egy tömbön asszociatív műveletet kellett végeznem, ezt párhuzamosítottam. Több módszert próbáltam ki:

1. A c++11-ben definiált szálakat felhasználva, szétosztottam a munkát egyenlő részekre a szálak között.

2. A jól ismert, gyakran használt OpenMP[23] külső könyvtárat használtam a párhuzamosításhoz. Ez fordító számára adott parancsok által automatikusan osztja szét a munkát a szálak között, továbbá bizonyos erőforrások használatára atomicitást is kérhetünk.

3. A szálak létrehozása és destruálása költséges. Ennek csökkentése érdekében a program futásának elején létrehoznak több szálat (thread pool), amelyek képesek feladatok végrehajtására. A feladatokat egy konkurens sorba szokták tenni, ahonnan a szálak kiveszik azokat és végrehajtják. Egy már meglévő c++11-es szabványon alapuló implementációt választottam ehhez. [24].

4. A c++17-es szabványban definiálták a standard párhuzamos algoritmusokat, azonban ezeket még egyetlen fordító sem támogatja. Az INTEL azonban készített egy referencia implementációt. Ezt is ki fogom próbálni.

4. fejezet

Algoritmusok áttekintése

4.1. Szuffix sorozatokat előállító algoritmusok

A második fejezetben ismertetett leggyorsabb szuffix sorozat implementáció Yuta Mori által implementált divsufsort [19] algoritmus. Az algoritmus egyik fő limitációja, hogy csak olyan sztringekre működik, melyek a memória(RAM) ötödében elférnek. Azokra a sztringekre, amelyek nem férnek el a memóriában, ismert az eSAIS algoritmus, mely külső memóriában igen gyorsan, azonban nagy tárigénnyel oldja meg a feladatot.

4.1.1. Divsufsort

A jelenleg ismert leggyorsabb belső memóriás szuffix sorozatot előállító algoritmus Yuta Mori által készített divsufsort [19]. Az algoritmus futási ideje legrosszabb esetben $\mathcal{O}(n\ log\ n)$. Tárigénye 5*n + $\mathcal{O}(1)$ byte. Az implementációs részleteket nem tárgyalja egyetlen cikk sem, csupán futási és tárigény összehasonlítások [18] találhatóak meg az algoritmussal kapcsolatban.Ezek alapján állítom, hogy a jelenlegi leggyorsabb. Annak ellenére, hogy az algoritmus nem párhuzamosított, említésre méltó, ugyanis számos párhuzamos implementáció összehasonlítási alapjaként szolgál, továbbá az egyik bemutatott algoritmus fő építő eleme is.

4.1.2. SAscan

Az SAscan[17] egy finn kutató csapat[6] által publikált és implementált szuffix sorozatok előállítására alkalmas algoritmus külső memóriában. Az algoritmus a leggyorsabb SA-k előállítására, ha a input hosszára igaz, $\frac{size(RAM)}{5} < size(input) < 5 * size(RAM)$. A következő részben bemutatom az SAscan párhuzamosított változatát is, azonban annak megértéséhez elengedhetetlen a kiinduló algoritmus megértése. A fejezetben bemutatott formális definíciók az eredeti cikkből származnak[17].

4.1.2.1. Alapok

4.1.3. Értelmezés. *Legyen egy X = X[0..m) sztring egy egészekből álló [0..σ) ábécé felett. Minden $i \in \{0, .., m-1\}$-re jelöljük X[i..m)-el az X szring m-i hosszú szuffixét, azaz X[i..m) = X[i]X[i+1]X[i+2]...X[m-1]. Hasonlóképpen jelöljük X[i..j)-vel*

az X[i]X[i+1]...X[j-1] sztring részsorozatát. Ha i=j, akkor az X[i..j) az üres sztring, melyet ϵ-nal jelölünk.

Ebben a fejezetben jelöljük SA_X-el az X sztring szuffix sorozatát.

4.1.4. Értelmezés. *A $SA_{X:Y}$ részleges szuffix sorozat, azoknak az XY szuffixeknek a lexikografikus rendezése, melyek az X-ben kezdődnek, vagyis $SA_{X:Y}[0..m)$ egy olyan sorozat, mely a [0..m) egészek permutációját tartalmazza, továbbá fennáll: $X[SA_{X:Y}[0]..m)Y < X[SA_{X:Y}[1]..m)Y < ... < X[SA_{X:Y}[m-1]..m)Y$.*

A cikk [17] megjegyzi, hogy $SA_{X:\epsilon} = SA_X$, illetve $SA_{X:Y}$ általában hasonló, de nem azonos az SA_X-el. Továbbá, az $SA_{X:Y}$-et megkaphatjuk az SA_{XY}-ból, ha eltávolítjuk az összes i $\geq$ m számot az SA_{XY}-ból.

4.1.4.1. Az algoritmus lépései

Legyen T[0..n) a szövegünk. A szöveget felbontjuk legfeljebb m hosszú részekre(blokkokra), ahol m-et úgy választjuk meg, hogy az algoritmus során használt összes $\mathcal{O}(m\ log\ n)$ bit elférjen a RAM-ban.

Legyen Y = T[i..n), illetve X = T[i-m..i). Tegyük fel, hogy már feldolgoztuk az Y-t és előállítottuk az SA_Y szuffix sorozatot. A következő lépésben előállítjuk az $SA_{X:Y}$-t, majd összefésüljük az SA_Y-nal, így megkapjuk az SA_{XY}-et.

Az $SA_{X:Y}$ és SA_Y elemei megtartják relatív sorrendjüket az SA_{XY}-ban. Ahhoz, hogy összefésüljük őket, szükségünk lesz egy $gap_{X:Y}[0..m]$ sorozatra, amely megmondja, hogy lexikografikusan hány SA_Y-beli szuffix szerepel $SA_{X:Y}[i-1]$ és $SA_{X:Y}[i]$ között. Formálisan:

4.1.5. Értelmezés.

$$gap_{X:Y}[0] = |\{j \in [0..|Y|) : Y[j..|Y|) < X[SA_{X:Y}[0]..m)Y\}|$$

$$gap_{X:Y}[i] = |\{j \in [0..|Y|) : X[SA_{X:Y}[i-1]..m)Y < Y[j..|Y|) < X[SA_{X:Y}[i]..m)Y\}|$$

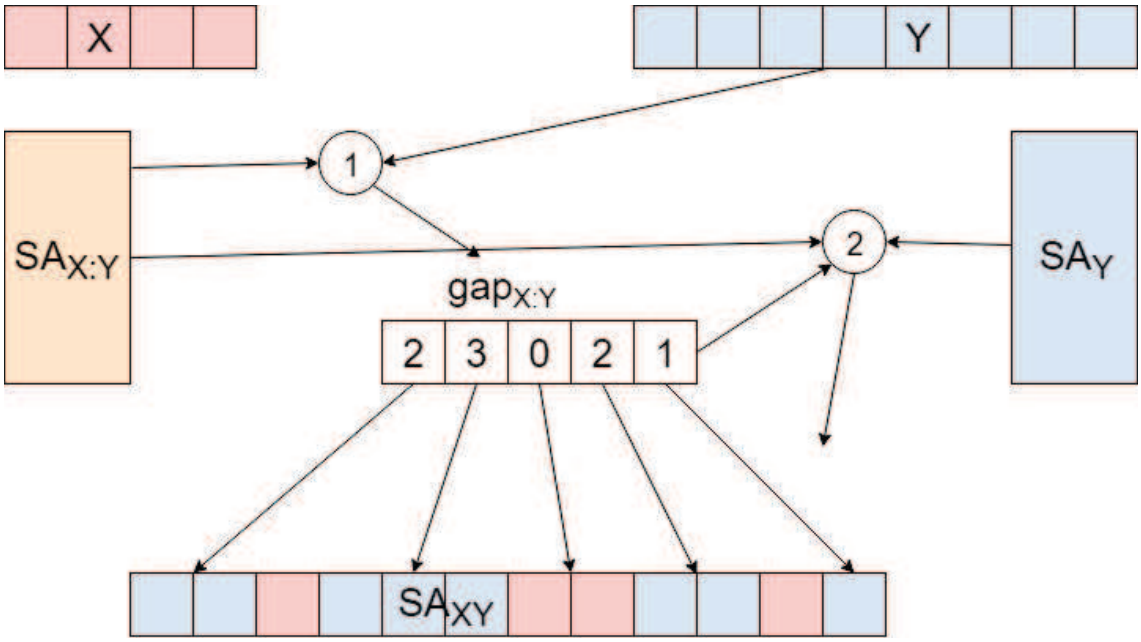
$$gap_{X:Y}[m] = |\{j \in [0..|Y|) : X[SA_{X:Y}[m-1]..m)Y < Y[j..|Y|)\}|$$

Megjegyzés: Az első képlet megadja, az Y-beli szuffix halmazának számosságát, melyben a szufixek lexikografikusan kisebbek, mint az XY-beli, X-ben kezdődő legkisebb szuffix. Az utolsó képlet megadja, az Y-beli azon szuffixek számosságát, amelyre a szuffixek lexikografikusan nagyobbak, mint az XY-beli X-ben kezdődő lexikografikusan legnagyobb szuffix.

Az algoritmust a 4.1 ábra szemlélteti. Az 1-essel jelölt rész ábrázolja a gap sorozat megalkotását, míg a kettes az összefésülést. Látható, hogy a $gap_{X:Y}$ megalkotásához, végig kell néznünk az Y-t. A $gap_{X:Y}$ segítségével könnyen előállítható az SA_{XY}, ahogy azt az ábra is mutatja.

Az ábrán 2-essel jelölt lépés nagyon költséges lemezről olvasás/írás szempontjából, ugyanis be kell olvasnunk hozzá az SA_Y-t és ki kell írnunk az SA_{XY}-t. Ennek elkerülése érdekében az összefésülés folyamatát későbbre halasztjuk és a háttértárra írjuk az $SA_{X:Y}$-t illetve a $gap_{X:Y}$-t.

Tegyük fel, hogy az $SA_{X:Y}$ vagy $gap_{X:Y}$ számítása közben össze kell hasonlítanunk két szuffixet, melyből legalább az egyik az X-ben kezdődik. Legrosszabb esetben a két szuffixnek lehet egy nagyon hosszú prefixe (többszörösen hosszabb,



4.1. ábra. Összefésülés a gap sorozattal[16]

mint az m, azaz a tárolt adatok hossza egyidejüleg). Ebben az esetben, lehetetlen lenne összehasonlítani a két szuffixet sok I/O művelet nélkül, feltéve, hogy nem rendelkezünk extra információval, mivel a háttértárolón lévő adatokat kellene olvasnunk hozzá. (például: extremális eset, ha az egész sztingünk egy karakterből áll).

4.1.6. Értelmezés. *Jelen esetben legyen gt_Y bitsorozat, amely eldönti minden szuffixről, hogy lexikografikusan kisebb-e, mint az Y. Formálisan, minden $i \in [0..|Y|)$:*

$$gt_Y[i] = \begin{cases} 1, & ha\ Y[i..|Y|) > Y \\ 0, & ha\ Y[i..|Y|) \leq Y \end{cases}$$

Két XY-beli szuffix összehasonlítása, melyből legalább az egyik X-beli, történhet karakterenkénti összehasonlítással, míg ütközést nem találunk a karakterek között, vagy az összehasonlítás el nem éri a hosszabbik szuffix végét. Formálisan, az i < j esetbe az $X[i..m]Y$ és $X[j..M]Y$ közötti sorrend eldöntésére, $X[i..m] < X[j..m]Y[0..j-i]$ vagy az $X[i..m] = X[j..m]Y[0..j-i]$ és $gt_Y[j-i]$ használható. Azonban látjuk, hogy ez még mindig legrosszabb esetben végig olvassa az Y-t. Ennek kiküszöbölésére ad megoldást [9], melyben értelmezi a következőt:

4.1.7. Értelmezés. $gt_{X:Y}[i] = \begin{cases} 1, & ha\ X[i..m)Y > Y \\ 0, & ha\ X[i..m)Y \leq Y \end{cases}$

Ebben az esetben a fentebb említett összehasonlítás elvégzéséhez elegendő $X[i..m-j+i) < X[j..m)$ vagy $X[i..m-j+i) = X[j..m)$ és $gt_{X:Y}[m-j+i] = 0$.

Az utolsó esetben az összehasonlítás legrosszabb esetben csak az X végéig tart, a gt_Y eldönti a sorrendet és nem kell tovább olvasnunk az Y-t.

Összefoglalva, minden m hosszú X blokkra:

1. Adott X, Y[0..m) és $gt_Y[0..m)$. Kiszámítjuk az $SA_{X:Y}$-t.
2. Adott X, $SA_{X:Y}$, Y és gt_Y. Kiszámítjuk az $gap_{X:Y}$-t és a gt_{XY}-t.

A kiszámított gt_{XY} bemenetként szolgál a következő blokk kiszámítására. A $SA_{X:Y}$-t és a $gap_{X:Y}$-t a háttértárolón tároljuk, amíg az összes blokkot fel nem dolgozzuk. Az algoritmus utolsó lépésében vesszük az összes részleges szuffix sorozatot, illetve a gap sorozatokat, majd ezek összefésülésével előállítjuk a végleges SA_T-t.

4.1.7.1. Az algoritmus részletei

Az első lépésben a részleges szuffix sorozat előállítását ($SA_{X:Y}$) a már említett és jelenleg leggyorsabbnak ismert divsufsort[19] algoritmussal végezzük el. Probléma viszont, hogy az algoritmust nem tudjuk módosítás nélkül használni, ugyanis nekünk a szuffixeink nem csak X-beliek, hanem XY-beliek, viszont csak az X-re szeretnénk alkalmazni. Tehát az algoritmusnak a rendezés során figyelembe kell vennie az X után szereplő, Y-beli prefixeket is a rendezésnél. Az algoritmus módosítását szeretnénk elkerülni több okból is:

1. az algoritmus megértése is komoly erőfeszítést igényel, a megfelelő módosítása pedig elképzelhető, hogy hatékonyságbeli problémákhoz vezet
2. ha a jövőben megjelenik egy hatékonyabb belső memóriás algoritmus, akkor azt is módosítanunk kell.

Ennek elkerülése érdekében megalkotunk egy Z sztringet úgy, hogy $SA_Z = SA_{X:Y}$, így nem kell módosítanunk az algoritmust és ha megjelenik egy hatékonyabb algoritmust azt minimális erőfeszítéssel integrálni tudjuk az SAscan-be.

4.1.8. Értelmezés. *[9]*

$$Z[i] = \begin{cases} X[i]+1, & \textit{ha } i = m-1 \textit{ vagy } X[i] > X[m-1] \textit{ vagy } X[i] = X[m-1] \textit{ és} \\ & gt_{X:Y}[i+1] = 1 \\ X[i], & \textit{különben.} \end{cases}$$

A Z sztring gyakorlatilag majdnem megegyezik az X-el. Csak abban az esetben különböző, ha az összehasonlítás átnyúlna Y-be is. Ebben az esetben kihasználjuk a $gt_{X:Y}$-t a versenyhelyzet eldöntésére, majd a megfelelő karakter értékét növeljük eggyel lexikografikusan. Ez további problémákat vet fel, melyek jelen esetben nem fontosak a diplomamunkám szempontjából és csak nagyon ritka esetben jönnek elő. Ezeknek a problémáknak a megoldására lásd az SAscan[9] 4. fejezetének végét.

A második lépés kicsit több körültekintést igényel.

A részleges Burrows-Wheeler transzformáltja egy X sztringnek a következő:

4.1.9. Értelmezés.

$$BWT_{X:Y}[i] = \begin{cases} X[SA_{X:Y}[i]-1], & \textit{ha } SA_{X:Y} > 0 \\ \$, & \textit{ha } SA_{X:Y} = 0 \end{cases}$$

4.1.10. Értelmezés. *Egy c karakterre és egy $i \in \{0..m\}$ indexre, $rank_{BWT_{X:Y}}(c,i)$ megegyezik azon karakterek számával, melyek azonosak c-vel, illetve $BWT_{X:Y}[0..i)$-ből vannak. $rank_{BWT_{X:Y}}(c,i) = |\{ch \in BWT_{X:Y}[0..i) | ch = c\}|$*

A $rank_{BWT_{X:Y}}(c,i)$ kiszámítása nagyon költséges, így érdekében, hogy a lekérdezés gyorsabb legyen, előfeldolgozzuk a részleges $BWT_{X:Y}$-t.

4.1.11. Értelmezés. *Legyen R[0..m) egy sorozat.*
$\forall i \in [0..m-1]$ *R[i]* $= rank_{BWT_{X:Y}}(i \ mod \ sigma, i)$.

Keressük meg azt a j pozíciót, melyre j mod σ = c.Ez egy i-hez közeli pozíció lesz. Ekkor végignézve i-től j-ig a $BWT_{X:Y}$-t megkapjuk a $rank_{BWT_{X:Y}}(c,i) - rank_{BWT_{X:Y}}(c,j)$ különbséget. Ebből már könnyen kitudjuk számolni a $rank_{BWT_{X:Y}}(c,i)$-t.

4.1.12. Értelmezés. *A* $sufrank_{X:Y}(S)$ *azon XY-beli szufixek, melyek X-ben kezdődnek és lexikografikusan kisebbek, mint az S. Vagyis, ha* $sufrank_{X:Y}(S) = k$ *(és* $0 < k < m$*), akkor* $X[SA_{X:Y}[k-1]..m)Y \ < S \leq X[SA_{X:Y}[k]..m)Y$.

4.1.13. Értelmezés. *Legyen C[0..σ]) egy egész számok sorozata, ahol a C[c] azon karakterek száma, melyre X[i] < c.*

4.1.14. Állítás. *Legyen* $k = sufrank_{X:Y}(S)$*. Ekkor bármely c-re,*

$$sukfrank_{X:Y}(cS) = C[c] + rank_{BWT_{X:Y}}(c,k) + \begin{cases} 1, & haX[m-1] = césY < S \\ 0, & különben. \end{cases}$$

A fentebbi állításnak köszönhetően az Y és a gt_Y visszafele történő végigolvasásával könnyen kiszámítható: $sufrank_{X:Y}(Y[j..|Y|)$ minden j = [|Y|-1 .. 0]-ra a képlet rekurzív alkalmazásával. Ezt az eljárást az angol irodalomban backward search-ként emlegetik [4]. Megjegyzem, hogy S = Y[j..|Y|) esetén, az Y<S összehasonlítás felcserélhető a $gt_Y[j] = 1$ összehasonlításával.

A sufrank rekurzív alkalmazásával kiszámoljuk, hogy az Y sztringben két szuffix közzé, hány X-beli szuffix ékelődik be lexikografikusan, melyből már könnyen tudjuk számolni a keresett $gap_{X:Y}$-t és a gt_Y-t a sufrank 4.1.12 értelmezése alapján.

4.1.14.1. Összefésülés

Minden $k \in [0..(n/m)), \text{legyen} X_k = T[k*m..(k+1)*m), Y_k = T[(k+1)*m..n), SA_k = SA_{X_k:Y_k}$ és $gap_k = gap_{X_k:Y_k}$. Az MERGE algoritmusban az előre kiszámolt részleges szuffixeket mozgatjuk a végleges szuffix sorozatba. Az algoritmus a következő invariánsok mellett működik:

1. Az i_k számontartja az SA_k részleges szuffix sorozatból a végleges sorozatba mozgatott szuffixeket.

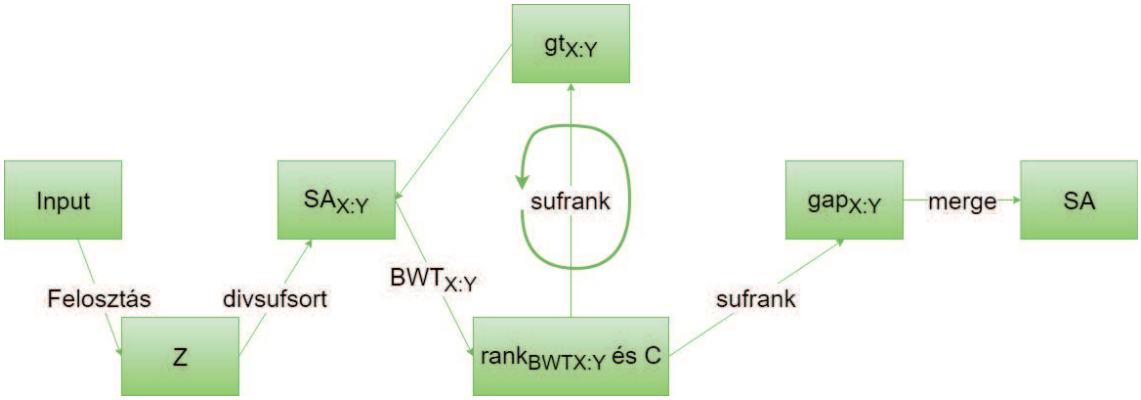
2. A $gap_k[i_k]$ számontartja az $SA_{k+1}, SA_{k+2}, .., SA_{n/m}$-ból még nem átmozgatott, olyan szuffixek számát, amelyek kisebbek, mint $SA_k[i_k]$

4.1.14.2. Az SAscan algoritmus nagy képe

Az SAscan algoritmus több egymást követő, egymással szorosan összefüggő nem triviális lépésekből áll, melynek között nehéz eligazodni. A 4.2 ábra a téglalapokkal jelöli az algoritmusban előálló különböző állapotokat, a téglalapok közötti nyilak pedig reprezentálják az állapot átmenetekhez szükséges függvényeket.

Algorithm 1 Merge szuffix arrays

```
1: function MERGE(X_k, Y_k, SA_k, gap_k)        ▷ Where X_k - T[km..(k+1)m), Y_k -
   T[(k+1)m..n) ,SA_k = SA_{X_k:Y_k}, gap_k − gap_{X_k:Y_k}
2:     for k = 0 to (n/m) − 1 do
3:         i_k = 0
4:     for i = 0 to n − 1 do
5:         k = 0
6:         while gap_k[i_k] > 0 do
7:             gap_k[i_k] = gap_k[i_k] − 1
8:             k = k + 1
9:         SA_T[i] = SA_k[i_k] + k ∗ m
10:        i_k = i_k + 1
```



4.2. ábra. Az algoritmus lépéseinek szemléltetése

4.1.14.3. Az SAscan kiértékelése

Az cikk[17] alapján az algoritmus futási ideje $\mathcal{O}(\frac{n^2}{M} * log(2 + \frac{log\ \sigma}{log\ log\ n}))$, ahol n a bemenő szöveg mérete, M a RAM mérete, σ az abécé hossza. Konstans méretű abécére és elég nagy szövegre a futási idő $\mathcal{O}(n^2/M)$, ami azt jelenti,hogy ha a szöveg hossza nem sokkal nagyobb, mint a RAM mérete, akkor lineáris idejű algoritmust kapunk. Ellenkező esetben a futási idő drasztikusan megnő. Ezt a cikk [17] 5. fejezetében lévő diagrammok is szemléltetik.

A cikk mérései alapján az egy időpillanatban (angol irodalomban peak memory usage) használt maximális memória (RAM) 5.2*m byte. A háttértároló által használt memória pedig 6.5*n byte.

4.1.15. pSAscan

A pSAscan [11] algoritmus a már említett finn kutató csapat [6] munkája. A fentebb bemutatott SAscan [17] algoritmusból indultak ki és azt optimalizálták. Ebben a részben bemutatom, hogyan párhuzamosították az SAscan algoritmust, illetve milyen optimalizációkat vetettek be annak érdekében, hogy külső memóriában is hatékony legyen az algoritmus. A fejezetben a [11] cikk alapján mutatom be a pSAscan algoritmust.

Az előbi részben ismertett részleges szuffix sorozat,részleges Burrows-Wheeler transzformált definiciója, valamint a gt tömb formális definiciója nem változik.

4.1.15.1. Részleges szuffix sorozatok összefésülése

Ebben a részben bemutatásra kerül két egymás melletti részleges szuffix sorozat összefésülésének javított, szekvenciális változata. Feltételezzük, hogy minden szükséges adatszerkezet elfér a belső memóriában. A következő részekben kerülnek tárgyalásra a párhuzamosítási ötletek és a külső memóriás optimalizációk.

Legyenek adottak az $SA_{X:YZ}$ és az $SA_{Y:Z}$ részleges szuffix sorozatok, valamely X,Y és Z sztringekre. A feladat kiszámolni az $SA_{XY:Z}$ részleges szuffix sorozatot, amely nem más, mint a két részleges szuffix sorozat megfelelő összefésülése. Ehhez definiáljuk a $gap_{X:Y:Z}[0..|X|]$ sorozatot, ahol $gap_{X:Y:Z}[i]$ azon $SA_{Y:Z}$-beli szuffixek száma, melyek lexikografikusan $SA_{X:YZ}[i-1]$ és $SA_{X:YZ}[i]$ között helyezkednek el. Formálisan:

4.1.16. Értelmezés.

$$gap_{X:Y:Z}[0] = |\{j \in [0..n) : Y[j..n)Z < X[SA_{X:YZ}[0]..m)YZ\}|$$

$$gap_{X:Y:Z}[m] = |\{j \in [0..n) : X[SA_{X:YZ}[m-1]..m)YZ < Y[j..n)Z\}|$$

$$\forall i \in [1..m)\ gap_{X:Y:Z}[i] =$$

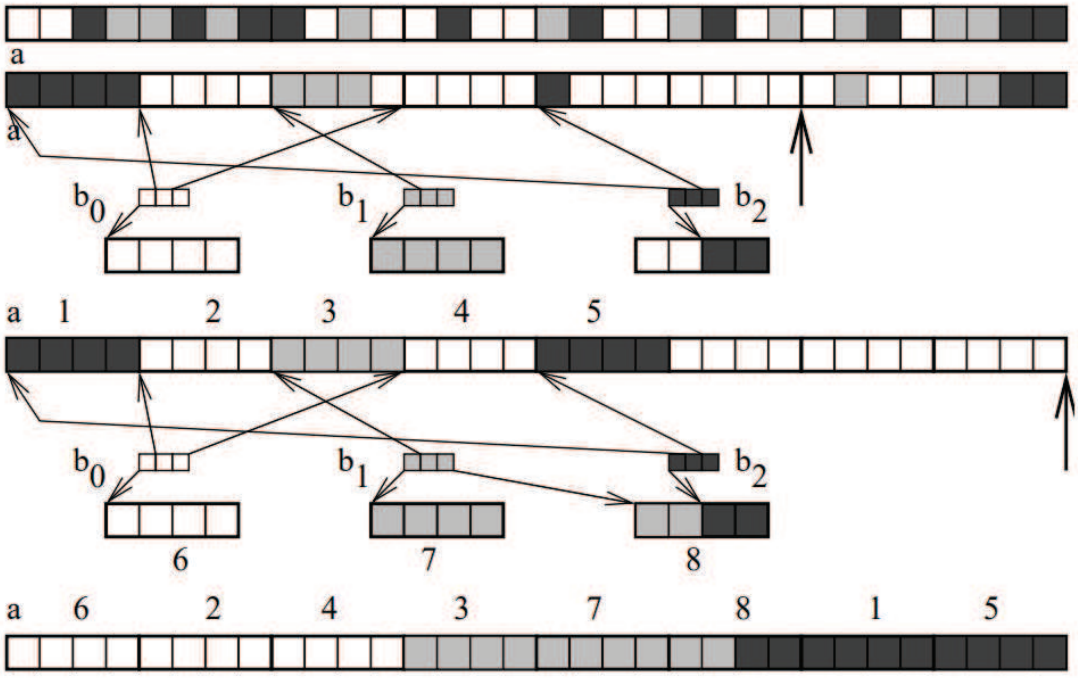
$$= |\{j \in [0..n) : X[SA_{X:YZ}[i-1]..m)YZ < Y[j..n)Z \leq X[SA_{X:YZ}[i]..m)YZ\}|$$

Az összefésülés folyamata kicsit eltér az SAscan [17] összefésüléséhez képest.

Első körben szükségünk lesz a $BWT_{X:YZ}$-re és $gt^{YZ}_{Y:Z}$-ra a gap sorozat kiszámításához. Az SAscan algoritmusban ezeket a sztringekből és a részleges szuffix sorozatokból számítottuk ki, melynek elméletben lineáris a költsége. A gyakorlatban ez viszont költséges, ugyanis a sztringekben való nem folytonos indexelés sok gyorsítótár hibához vezet (angol nevén cache miss), ami lassítja a futási időt, ahogy azt az előző fejezetekben már bemutattam.

Azzal a feltételezéssel élve, hogy két egymás melletti részleges szuffix sorozatnak $SA_{X:YZ}$ és $SA_{Y:Z}$-nek adott a $BWT_{X:YZ}$,$BWT_{Y:Z}$, $gt^{XYZ}_{X:YZ}$ és $gt^{YZ}_{Y:Z}$ kiszámoljuk a $BWT_{XY:Z}$ részleges Burrows-Wheeler transzformáltat és $gt^{XYZ}_{XY:Z}$ bit vektort. A részleges Burrows-Wheeler transzformáltakat együtt tároljuk a részleges szuffix sorozatokkal, így gyakorlatilag a részleges szuffix sorozatok összefésülésével megkapjuk a Burrows-Wheeler transzformáltat is. A keresett $gt^{XYZ}_{XY:Z}$ bit vektort a $gt^{XYZ}_{X:YZ}$ illetve a $gt^{YZ}_{Y:Z}$ bit vektorok konkatenációjából állítjuk elő. Az első bemeneti paraméter, míg a másodikat kiszámoljuk a visszafelé történő keresés során felhasználva, hogy $gt^{XYZ}_{Y:Z}[j] = 1$ akkor és csak akkor, ha $sufrank_{X:YZ}(Y[j..n]Z) > i_{XYZ}$, ahol i_{XYZ} az XYZ pozicja az $SA_{X:YZ}$-ben.

Az eredeti SAscan [17] algoritmusban az összefésülést a végére hagyták, azonban a pSAscan algoritmusban gyakran azonnal végre kell majd hajtani. Ez könnyen elvégezhető, amennyiben külön memória terület áll rendelkezésünkre a bemenetnek és a kimenetnek. A pSAscan során egy olyan technikát alkalmaztak, amelyiknek köszönhetően majdnem helyben megoldható az összefésülés. Az ötlet az, hogy felosztották a részleges szuffix sorozatokat kicsi blokkokra, melyeket lapoknak hívtak. Ezekre a lapokra mutatókat tárolnak egy tömbben, melyeket lap indexeknek hívnak. Továbbá pár extra lapot is tárolnak, melybe átmenetileg helyezik el az elemeket. Ahogy az algoritmus halad előre, az input elemekből output elemek lesznek, s a nem használt input elemek helyére bemásolják az output elemeket. Minden egyes indexelés a lapokra mutató tömbökön kell, hogy keresztül menjen. A 4.3 szemlélteti a módosítást.



4.3. ábra. Trükkös rendezés[13]

4.1.16.1. A részleges szuffix sorozatok párhuzamos összefésülése

Ez a rész egy több magos architektúrát feltételez, mely képes p darab szálat futtatni egyidejűleg. Továbbá feltételezi, hogy elég nagy osztott memória van a szálak között ahhoz, hogy minden adatszerkezet elférjen.

Az első feladat az SAscan [17] algoritmusban ismertetett R adatszerkezet előállítása, mely könnyen párhuzamosítható, mivel az adatszerkezet elemei egymástól függetlenek (emlékeztetőül R[i] = $rank_{BWT_{X:Y}}(i \ mod \ \sigma, i)$). Az R tömböt p egyenlő részre osztjuk. Mindegyik szálhoz, hozzárendelünk egy részt, amelybe csak az adott szál írhat. A szálak a számításokat csak olvasható adatszerkezeteken végzik, így elkerülve a versenyhelyzetet.

Az összefésülés legköltségesebb művelete futási idő szempontjából a rank lekérdezések miatt, a visszafele történő keresés (backward search). Ez könnyen párhuzamosítható, ha az Y-t felosztjuk p darab egyenlő hosszú szakaszra, majd mindegyiken külön indítjuk a visszafele történő keresést, mely az adott szakasz végétől indul. Mindegyik szál kiszámítja a kezdeti $sufrank$ értékét. Minden kiszámolt lépésben a szálaknak növelniük kell a gap sorozat megfelelő értékét. Ez körültekintést igényel, ugyanis nem engedhetjük, hogy a több szál egyszerre ugyan azt az értéket növelje. A zárolást költséges lenne, ezért máshoz kell folyamodni. Minden szál egy saját tárolóba gyűjti a növelt pozíciókat, majd ha megtelt a tárolója, rendezi azokat pozíció alapján, majd egy konkurens sorba teszi őket. Egy másik szál figyeli a sort és ha új tároló érkezett, akkor szétosztja p egyenlő részre, majd szétosztja p szál között a részeket, amelyek megnövelik az indexek alapján a gap sorozat megfelelő tagját. Mivel a tároló rendezett, ezért két szál sosem fogja ugyan azt az elemet egyszerre növelni, ezzel az algoritmus elkerüli az esetleg versenyhelyzeteket,szálbíztos marad.

A következő lépés a részleges szuffixek összefésülése a gap sorozat alapján. A

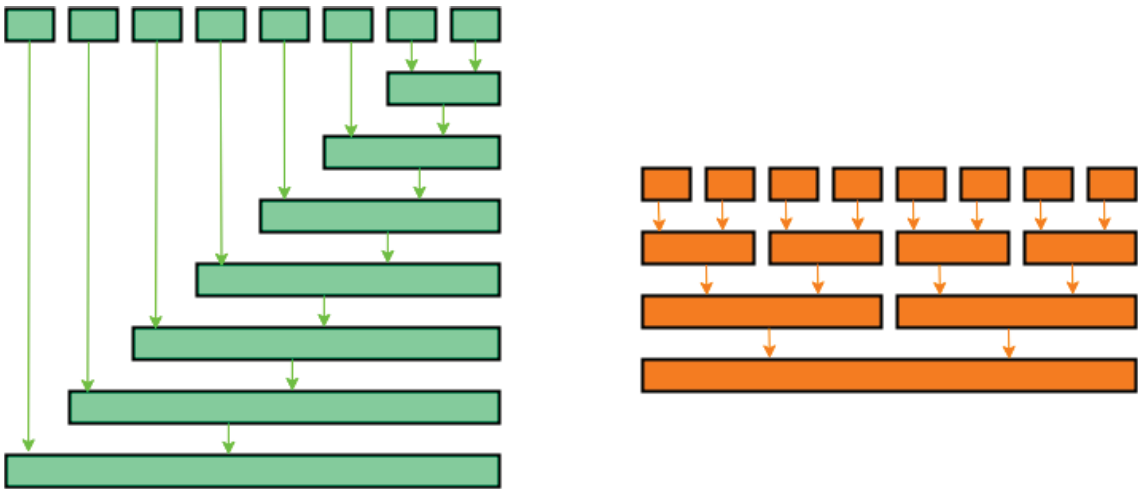
gap sorozaton előfeldolgozást hajtanak végre, p egyenlő távolságra lévő kumulatív összeg kiszámításával, így a bemeneti intervallumokat $\mathcal{O}(n/p)$ darab összehasonlítással a gap sorozaton megtudják állapítani. Következő lépésben minden szálhoz hozzárendelünk egy részt kimeneti sorozatból, ahova végrehajtja az összefésülést. Az összefésülést a fentebb említett, 4.3 ábrán szemléltetett, majdnem helyben rendező algoritmussal hajtja végre.

4.1.16.2. Párhuzamos szuffix sorozatot előállító algoritmus

Ebben a részben ugyan azt feltételezzük az architektúránkról, mint az előzőben.

Első lépésben felosztjuk a szöveget p darab egyforma hosszú blokkra. Legyen X az első blokk, Y a második, Z pedig a sztring Y utáni szuffixe. Ahhoz, hogy előállítsuk az $SA_{X:YZ}$-t, létrehozzuk a már ismertetett módon $\hat{X}$-et, melyre teljesül, hogy $SA_{\hat{X}} = SA_{X:YZ}$. Ehhez szükségünk van a $gt^{YZ}_{X:YZ}$. Ebben az esetben viszont, a $gt^{YZ}_{X:YZ}$-t az ismertettől eltérő módon számoljuk. Először kiszámoljuk $g\hat{t}_X = gt^{Y}_{X:Y}$-t. A számítás során megjelöljük azokat a biteket, ahol $X[i..m)Y[0..m-i) = Y$. Ezeket eldöntetlen biteknek nevezzük. Ebben az esetben $gt^{YZ}_{X:YZ}[i] = gt_Y[i]$.Tehát, ha $g\hat{t}_X[i]$ eldöntetlen bit, akkor függ az $g\hat{t}_Y[i]$. Amennyiben $g\hat{t}_Y[i]$ is eldöntetlen bit, függ a következő blokk i. bitjétől stb... Tehát, ha adott az összes $\hat{gt}$ vektor, akkor kiszámolhatjuk az összes eldöntetlen bitet. Ehhez $\mathcal{O}(m * p)$ munkát kell végezni, amelyik $\mathcal{O}(m)$ időben kerül p szálon. Az $\hat{X}$ ismeretében a divsufsort [19] lineáris implementációjával előállítjuk a részleges szuffix sorozatokat. Ezeknek az előállítása egymástól független, tehát párhuzamosítható.

A részleges szuffix sorozatok ismeretében kiszámoljuk a részleges Burrows-Wheeler transzformáltat illetve a gt sorozatot. Továbbá előre kiszámolunk $\mathcal{O}(p^2)$ sufrank értéket, annak érdekében, hogy ez rendelkezésre álljon az összefésüléskor. Az összefésülést több féle módon is elvégezhetjük. Az első lehetőség, amikor az eddig elkészült részhez, hozzávesszünk egy új részleges szuffixet és a kettőt összefésüljük (lásd a 4.4 ábrán zöld színnel).A második, amikor kiegyensúlyozott módon történik az összefésülés(lásd a 4.4 ábrán narancssárga színnel). Meglepetésemre nem a kiegyensúlyozott módon történő összefésülés a leggyorsabb.



4.4. ábra. Részleges szuffixok összefésülése

Az összefésüléskor szükséges idő megbecsülhető $a * l + b * r$ képlettel, ahol az l a bal oldali, míg az r a jobb oldali részleges szuffix sorozat hossza. Az a és b tapasztalati úton beállított konstansok. Mivel a futási idő hosszát a visszafele történő keresés befolyásolja a leginkább, így b > a. Ennek eldöntésére egy dinamikus algoritmust építettek be, amely a b/a arány és a p szálak alapján eldönti, hogy milyen formában érdemes összefésülni a részleges szuffix sorozatokat.

Az algoritmus tárigény $\mathcal{O}(12.25m + 7.125n)$ byte, a tapasztalati konstansok miatt.

4.1.16.3. Az párhuzamos algoritmus kiterjesztése külső memóriára

Ebben a fejezetben egy valós architektúrára mutatják be az algoritmust: A RAM mérete korlátos M, a külső memória mérete is korlátos, azonban feltételezik, hogy a sztring és a hozzá szükséges segéd adatszerkezetek elférnek benne, illetve p szál egyidejű futtatására képes az adott számítógép. A szálak osztott memórián keresztül tudnak egymással kommunikálni.

Az algoritmus alapötlete:

1. A szöveg felosztása m méretű blokkokra, melyek elég kicsik ahhoz, hogy a szükséges adatszerkezetek elférjenek a memóriában.

2. Minden X blokkra kiszámítja az $SA_{X:Z}$ részleges szuffix sorozatot és a $gap_{X:Z:\epsilon}$ gap sorozatot, ahol Z az X után következő szuffixxe a bemeneti szövegnek.

3. A részleges szuffix sorozatok összefésülése.

Az utolsó lépésben az I/O műveletek dominánsak, ezért a feladat nem párhuzamosíthatósága elenyésző.

Az második részben felhasználhatjuk az előzőekben bemutatott algoritmust kevés módosítással. A módosításra azért van szükségünk, mivel az $SA_{X:Z}$ a RAM-ban van, viszont a Z-t a háttértárolón tároljuk. A $gap_{X:Z:\epsilon}$ előállítása a visszafele történő kereséssel (backward search) történik a Z sztringen felhasználva a $BWT_{X:Z}$-t. A különbség az, hogy Z és $gt^{Z}_{Z:\epsilon}$ a háttértárolón vannak. Nagy bemenetre (n»m) ez $\Theta(n^2/m)$ műveletigényű, ami jóval több idő, mint a háttértárolóról írás/olvasás, mivel a rank és a gap sorozatok számolása költséges. Ezért a párhuzamosítás gyorsulást fog elérni.

Az blokkok hosszát, azaz m-t, úgy választjuk meg, hogy az összes szükséges segédadatszerkezet elférjen a RAM-ban. A *gap* sorozat előállításához körülbelül 5.2m byte-ra van szükség. A részleges szuffix sorozat előállításához viszont 10m byte-ra van szükség. Annak érdekében, hogy csak 5.2m byte extra memóriára legyen szükség a RAM-ban, minden X blokkot ketté osztunk: először kiszámoljuk $SA_{X_1:X_2Z}$-t, majd $SA_{X_2:Z}$-t és kiírjuk őket a külső memóriába. Kiszámoljuk a $gap_{X_1:X_2:Z}$ ahhoz, hogy össze tudjuk fésülni $BWT_{X_1:X_2Z}$-t és $BWT_{X_2:Z}$-t. Az $SA_{X_1:X_2Z}$-t és $SA_{X_2:Z}$ nem fésüljük össze $SA_{X:Z}$-ként, hanem $gap_{X_1:X_2:Z}$ és $gap_{X_2:Z:\epsilon}$ segítségével a végső összefésülésben fogjuk hozzáadni a végső szuffix sorozathoz.

4.1.16.4. pSAscan kiértékelése

A belső memóriás pSAscan algoritmus futási ideje $\mathcal{O}(n \log p)$ p szálon. A külső memóriás algoritmus futási ideje $\Theta(n^2/M)$, ahol M a RAM mérete. A belső me-

móriás verziónak 10*n byte RAM-ra van szüksége, külső memóriás algoritmusnak 7.5*n byte külső memóriára.

4.1.17. eSAIS

Az algoritmus működési elvének bemutatása nem célja a diplomamunkámnak. Az algoritmus viszont említésre méltó, ugyanis ez a jelenleg ismert, leggyorsabb külső memóriás algoritmus [11].

Az algoritmus legnagyobb hibája, hogy a tárigénye 28*n byte. Vagyis 28-szor több háttértárhelyre van szükség, mint a bemenet hossza. Az futási ideje arányos a háttértárolón egy optimális rendezési algoritmus futási idejével[2]. A fentebbi két részben láttuk, hogy az SAscan és pSAscan algoritmus futási ideje $\mathcal{O}(n^2/m)$ vagyis, ha a bemenet mérete csak konstransszorosa a RAM méretének, akkor a futási ideje $\mathcal{O}(n)$, azaz lineáris. A háttértárigénye 7.5*n byte. A lineáris futási idő, valamilyen konstanssal szorzódik az input méretétől függően. Gyakorlatilag, ha a bemenet egy kicsi konstansszorosa a RAM méretének, akkor az SAscan és pSAscan algoritmus gyorsabb, mint az eSAIS.

A diplomamunkám egyik célja ezeknek a feltételezéseknek az ellenőrzése mérésekkel.

4.2. Burrows-Wheeler transzformált

A Burrows-Wheeler transzformált kiszámolható az SA egyszerű végigolvasásával, amennyiben az elfér a memóriában, így a divsufsort [19] egy hatékony algoritmus lesz az BWT kiszámítására is. Abban az esetben, mikor a bemeneti sztring hossza meghaladja a RAM méretét, az SAscan algoritmus is jól teljesít, mivel a szuffix sorozat előállítása és összefésülés közben előállítja a BWT-ket is. Szuffix sorozat indexeket tárol, melyek általában 4, de akár 8-szor akkorák is lehetnek, mint amilyen hosszú a bemeneti sztringünk. Ez attól függ, hány bitre van szükségünk az input tárolásának hosszára. Előfordul olyan eset, amikor a szuffix sorozat már nem fér el a memóriában, viszont a karakter sorozat, amelyre ki kell számolnunk a Burrows-Wheeler transzformáltat pont elfér. Ebben az esetben használhatunk egy algoritmust, amelyik helyben állítja elő a BWT-t, mindössze pár extra byte-ra van szükség pár lokális változó tárolására.

4.2.1. BWT helyben történő előállítása

A Burrows-Wheeler transzformációt elő lehet állítani helyben, alig pár extra byte felhasználásával. A következőkben azt fogjuk megnézni, hogyan a [3] cikk alapján.

Adott T = T[0..n-1] sztring, ahol T[n-1] = \$. Bármelyik karakter elmozdítása a T-n belül olyan végzetes hibához vezethet, melynek köszönhetően nem ismerjük az eredeti sorrendet, így a BWT-t sem tudjuk előállítani. Az ötlet az, hogy jobbról balra haladva végezzük el a rendezést. Tegyük fel valamilyen S-re ($0 \leq s \leq n-3$), hogy az eredeti sztring T_{S+1} szuffixének már előállítottuk a Burrows-Wheeler transzformáltját - jelöljük ezt BWT(T_{S+1})-el - a T tömbben, az S+1 pozíciótól kezdődően. A következő algoritmus végrehajtásával $\forall i \in [0..s]$ előállítja a Burrows-Wheeler transzformáltját a T sztringnek:

1. Legyen c=T[i].

2. Keressük meg azt a p pozíciót melyre: T[p] = $. Vegyük észre, hogy a p-i pozíció a T tömbben a T_{i+1} szuffix sorrendbeli (rank) számát adja.

3. Számoljuk meg, hogy hány c-től kisebb karakter van a T[i+1..n-1] intervallumban, illetve hány c-vel megegyező a T[i+1..p] intervallumban.Majd adjuk hozzá az i-t.
$$r = \sum_{j=i+1}^{n-1} \chi(T[j] < c) + \sum_{j=i+1}^{p} \chi(T[j] = c) + i$$

4. Szúrjuk be a T[p] helyre a c karaktert. T[p] = c.

5. Toljuk el az összes T[i+1..r]-ben szereplő karaktert eggyel balra a T[i..r-1] helyre, majd szúrjuk be a T[r] helyre a $ karaktert.

4.2.2. Példa. *Legyen T = "almafa$". Tegyük fel, hogy i = 3. Ekkor c = 'm'. T = "almAF$A". A nagybetűvel jelölt karakterek a T sztringből a már előállított $BWT(T_{i+1})$. Ekkor az algoritmus lépéseinek eredményei a következőek lesznek:*

1. *c = 'm'*

2. *p = 6, mivel S[6]= $.*

3. *r = 4 + 0 + 3 = 7, mivel mind a 4 BWT-ben szereplő betű kisebb, mint az c='m'.*

4. *T = "almAFMA", mivel felülírtuk a $ karaktert a c='m' betűvel.*

5. *Először T = "alAFMAA", majd T = "alAFMA$", mivel először eltoljuk a karaktereket, majd felülírjuk a '$' karakterrel.*

Az algoritmus futási ideje $\Theta(n^2)$, mivel n-2 iterációban végig olvassa a sztringet az r értékének kiszámításához.Az algoritmus tárigénye a bemeneti sztringet leszámítva, a c,p,r és a ciklus iterációjához használt változók.

4.2.3. Kötegelt változat

A fenti algoritmusban a legköltségesebb, hogy minden egyes karakterre ki kell számolni az eltolás hosszát, majd minden karakterre el kell végezni a megfelelő eltolást. Egy optimalizációs ötlet, hogy egy köteg karakterre végezzük el a kiszámolást, majd a köteget egyszerre toljuk el. Az algoritmusban a kötegben található karakterek száma k. Az a cél, hogy a futási időt javítsuk,még akkor is ha ez a memória használat kárára. Gyakorlatilag, keressük, hogy ha van valamennyi fel nem használt memóriánk, hogyan tudnánk azt felhasználni a futási idő optimalizálására.

Első lépésben legyen m = n % k. Kiszámoljuk az input sztringünk Burrows-Wheeler transzformáltját a T(n-m..n-1] sztringre a fentebb bemutatott algoritmussal. Következő lépésben z darab k hosszú kötegre osztjuk a T[1..n-m] részsztringet, ahol l = n div k. Jobbról balra haladva, $\forall i \in [1..l]$-re hajtsuk végre a következő algoritmust, felhasználva, hogy $Z = T[(i+1) * k..n]$:

1. Állítsunk elő egy C statisztika tömböt, melyben minden T(i*k .. (i+1)*k] karakterre, számoljuk meg, hogy hány tőle kisebb karakter szerepel a Z sztringben. A C tömbböt a T(i*k..(i+1)*k] karaktereivel indexeljük, továbbá minden karakternek elég egyszer szerepelnie, rendezett sorrendben.

2. Állítsunk elő egy R_1 mátrix, mely segítségével gyorsan tudjuk számolni az alábbi lekérdezést: valamilyen j indexre és α karakterre, hány olyan $\hat{j}$ pozíció van Z-ben, melyre teljesük, hogy Z[j] = α és $j \leq \hat{j}$.

3. Legyen j a $ pozíciója a Z-ben. Hozzunk létre egy töréspont listát (innentől br), mely kezdetben a <$,j> párt tartalmazza.A lista az egészek alapján megtartja rendezettségét.

4. Jobbról balra haladva hajtsuk végre a következőket a s $\in$ (i*k..(i+1)*k] intervallumon:

   (a) Tekintsük az <$,j> párt a br listából.

   (b) Legyen c = T[s]

   (c) Legyen $\hat{r} = r_0 + r_1 + r_2$, ahol

   $$r_0 = \sum_{ch \in Z} \chi(ch < c), r_1 = \sum_{\hat{j}=s+1}^{j} \chi(Z[\hat{j}] = c), r_2 = \sum_{b \in br} \chi(b << \$, j >).$$

   Vegyük észre, hogy az r_0 a C tömb segítségével, míg az r_1 az R_1 tömb segítségével könnyen számolható. Az r_2 számítását lineárisan, vagy akár különböző wavelet fák segítségével könnyen számíthatjuk.

   (d) $\hat{r}$ kiszámítása után a <$,j> töréspontot <c,j>-re cseréljük a br listában, majd a listába rendezertten szúrjuk a <$,$\hat{r}$> töréspontot.

5. Összefésüljük a br listát a Z-vel.

4.2.4. Az algoritmus részletei

A C statisztika tömböt a Z egyszeri végigolvasásával feltudjuk tölteni. A Z minden c karakterére keressük meg a C tömbben azt a legkisebb betűt, amelyik nagyobb, mint a c. Ezt a pozíciót növeljük eggyel. A keresés történhet bináris kereséssel a karaktereken, mivel a tömb rendezett. Végül a C tömb minden elemére számoljuk ki a tömb prefix összegét az adott pozícióig.

Az R_1 mátrixot a $T(i * k..(i + 1) * k]$-ban előforduló karakterekkel,illetve egy [1..k] intervallummal indexeljük. A mátrix minden minden (x,y) eleme tárolja, hogy a Z sztringben az x-hez tartozó karakter hányszor fordul elő a Z[y*(n-s)/k .. (y+1)*(n-s)/k] részben. Az R_1 előállítása is Z egyszeri végigolvasásával előállítható: $\forall < c, i > \in zip(Z, N) : R_1[c, idivk] = R_1[c, idivk] + 1$. Gyakorlatilag minden egyes karakterre kiszámoljuk, bináris kereséssel, hogy az R_1 maátrixban, melyik karakterhez tartozik, illetve meghatározzuk, hogy a mátrix mely eleme reprezentálja azt a szakaszt, amelyikből a karakter származik. Az r_1 érték számítása az R_1 adatszerkezetből úgy történik, hogy kiszámoljuk hány egész intervallumon át kell megszámolni az egyező karaktereket, ezt kiolvassuk az R_1 mátrixból, majd végigolvassuk a maximum (n-s)/k hosszú maradék részt és megszámoljuk, hogy még hányszor fordul elő

az adott karakter.A fenti képletekben a div az egész osztást, míg a zip függvény a funkcionális nyelvekből ismert két halmaz elemeinek felsorolását jelenti párként.

A cikkben[3] bemutatott összefésülés egy relatív rendezést igényel, mivel előfordul a töréspont listában, hogy egyes pároknak a számértékük megegyezik. Ezt a cikk nem definiálta és nem is triviális. Az SAscan algoritmusban bemutatott gt tömbhöz kötődhet, amelyiknek a kiszámítása költséges. Ezért a töréspont lista kiszámítására egy módosítást tettem:

1. Tekintsük az <$,j> párt a br listából.
2. Legyen c = T[s]
3. Számoljuk ki az r_0-t és az r_2-t a fenti definiciók alapján.
4. $$r_1 = \sum_{\hat{j}=s+1}^{j-r_2} \chi(Z[\hat{j}] = c).$$
5. $\hat{r} = r_0 + r_1 + r_2$. $\hat{r}$ kiszámítása után a <$,j> töréspontot <c,j>-re cseréljük a br listában, majd a listába rendezertten szúrjuk a <$,$\hat{r}$> töréspontot.

A fő különbség az, hogy előbb megnézem, hogy hány megegyező karakter van az aktuálisan vizsgálandó karakterrel a br-ben, majd a r_1 számításakor nem a $ pozíciójáig, hanem a pozíciót r_2 karakterrel megegyező pozícióig állítom elő. Így meghatároztam a br listában az összefésülés után a karakterek abszolút pozícióját.

Az összefésülés a következő képen történik: legyen i = s+1, illetve j = s-k. Megnézem, hogy a br lista elején szereplő első töréspont indexe megegyezik-e az i-vel. Amennyiben igen kiírom a töréspontho tartozó karaktert a j. pozícióra, eltávolítom a töréspont lista első töréspontját és növelem a j-t; amennyiben nem egyeztek meg kiírom Z[i]-t a j. pozícióra és növelem az i-t és a j-t is. Addig folytatom a folyamatot, amíg a br lista ki nem ürül.

Az R_1 és C előállítása $\mathcal{O}(n * log\ k)$, míg a töréspont listán a műveletek wavelet fával $\mathcal{O}(logk)$ műveletbe kerülnek. Ezt n/k-szor kell végrehajtani. Az R_1 adatszerkezetnek legnagyobb a tárigénye: $\mathcal{O}(k^2)$. Tehát az algoritmus műveletigénye és tárigénye:

$$\mathcal{O}(\frac{n^2 * log\ k}{k}) \text{ és } \mathcal{O}(k^2)$$

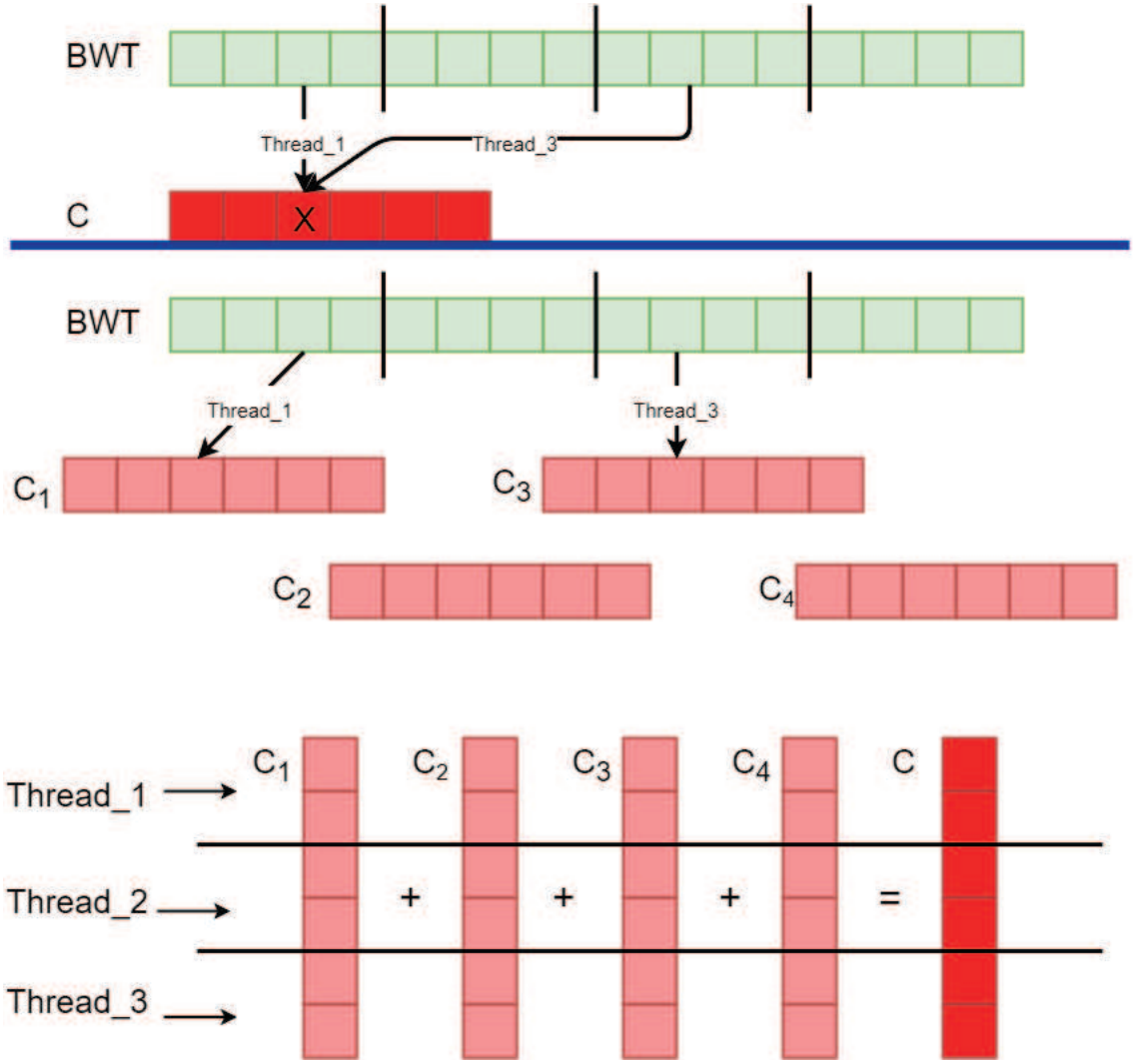
.

4.2.5. Kötegelt változat párhuzamosítása

Az algoritmusban az egyes karakterekhez tartozó $\hat{r}$ érték számításánál szükség van minden előző információra, vagyis a számítások függnek az előző lépéstől, ezért kevés lehetőségünk van a párhuzamosításra. Az algoritmusban az első és második lépésben a C és R_1 tömbök számítása viszonylag időigényes és kellően nagy is ahhoz, hogy ezek számításakor párhuzamosítsunk. Mindkét esetben a már kiszámított BWT-ben az adott karakter alapján kell a C és az R_1 tömbben is a számolt pozíción növelni az értéket.

Az ötlet az, hogy a BWT sorozatot osszuk fel annyi részre, ahány szálat képes futtatni a programunk, majd mindegyik részt adjunk oda egy szálnak, amelyik megnöveli a C és az R_1 tömbökben az adott pozíciókon az értékeket. Ebben az esetben

több szál ugyan azt a pozíciót is megpróbálhatja növelni, ami szinkronizációt igényel.A 4.5 ábrán a kék vonal feletti rész szemlélteti az esetet. A szinkronizációs költségek elkerülésének érdekében, minden szál létrehoz magának egy C-vel, illetve R_1-el azonos méretű tömböt, majd ezekben lokálisan számolja ki az eredményeket a saját szakaszára. Ezeket a lokális tömböket a végén vektoriálisan összeadjuk, így megkapva a C tömböt. Az összeadás is párhuzamosan történik. Minden lokális tömböt annyi egyenlő részre osztunk, ahány szálunk van, majd a szálak a megfelelő részeket fogják összegezni. A 4.5 ábrán a kék vonal alatti rész szemlélteti az esetet.



4.5. ábra. C tömb párhuzamos előállítása

4.2.6. A párhuzamosítás várható gyorsulása

A feladat nagyjából 3 részre osztható:

1. A C tömb előállítása
2. A R_1 tömb előállítása

3. A kötegek eltolásának számítása és az összefésülés

Valószínűleg a harmadik lépés a legköltségesebb, mivel ott az egész tömbben fölülírhatjuk az elemeket, míg az első két lépés csak rövid tömbökön végez írást, a hosszú tömbön csak olvasást végez. Bár Amdahl törvényében nincs definiálva, hogyan számoljuk a feladat párhuzamosíthatóságát, az első két lépésben a feladat a fentiek alapján jól párhuzamosítható, míg a harmadik rész egyáltalán nem, ebből kifolyólag a párhuzamosítással csak mérsékelt javulásra számíthatunk a futási időre nézve.

4.3. Leghosszabb közös prefix

A leghosszabb közös prefixet könnyen elő lehet állítani az SA sorozatból,ahogy azt a feladat bemutatásánál is láttuk. Ilyen algoritmus például a Kasai, Lee és Arimura [14] által javasolt lineáris idejű algoritmus. Továbbá ismert, hogy a már említett eSAIS [2] algoritmus az SA számítása közben, képes előállítani az LCP sorozatot is. Az első olyan algoritmus, amelyik a már kiszámolt SA-ból képes előállítani az leghosszabb közös prefix tömböt az LCPscan [7].

4.3.1. LCPscan

Az inverz szuffix sorozat,jelölése ISA, a SA szuffix sorozat egy olyan permutációja, melyre akkor és csak akkor ISA[i]=j, ha SA[j]=i. Gyakorlatilag az ISA[i] tárolja, hogy az i. szuffix hol található az SA tömbben. Az i. szuffixet lexikografikusan közvetlenül megelőző szuffix a Φ[i]: $\forall i \in [1..n] : \Phi[SA[i]] = SA[i-1]$

A PLCP tömb az LCP tömb egy permutációja, melyben az elemeket lexikografikus rendezésből a karakterek szövegbeli sorrendjére permutáljuk: $\forall i \in [1..n]$ PLCP[SA[i]] = LCP[i], vagyis $\forall i \in [0..n)$ PLCP[i]=lcp(i,$\Phi(i)$).

A 4.1 táblázatban látható a fentebb definiált tömbök a t=banana szó alapján.

T	b	a	n	a	n	a
SA	6	4	2	1	5	3
ISA	4	3	6	2	5	1
Φ	2	4	5	6	1	3
PLCP	0	3	2	1	0	0
LCP	0	1	3	0	0	2

4.1. táblázat. SA,ISA,Φ,PLCP,LCP

4.3.2. Az algoritmus lépései

Az algoritmus három fő lépésből áll:

1. Állítsuk elő a az ISA-t és Φ-t az SA-ból.
2. Állítsuk elő a PLCP-t az X-ből és a Φ-ből.
3. Állítsuk elő az LCP-t az ISA-ból és a PLCP-ből.

Az első lépéshez alkossuk meg $\forall i \in [1..n]$-re az (i,SA[i],SA[i-1]) hármasokat a szuffix sorozat végigolvasásával. Ezek után egy külső memóriás rendező algoritmussal rendezzük őket a második komponens szerint. Ekkor előáll az első komponensek sorozataként az ISA[0..n), illetve a harmadik komponensek sorozataként a Φ[0..n).

A harmadik lépéshez alkossuk meg $\forall i \in [0..n)$-re (ISA[i],PLCP[i]) párosokat, majd rendezzük egy külső memóriás rendező algoritmussal az első komponens szerint. Ezzel megkapjuk az LCP[1..n] sorozatot a párok második komponensének sorozataként.

A második lépésben a PLCP előállítása kicsit bonyolultabb. Osszuk fel a szöveg legfeljebb m hosszú, $\mathcal{O}(n/m)$ darab részre, majd dolgozzuk fel őket egyenként. Az m-et úgy választjuk meg, hogy egy m hosszú szöveg illetve a szükséges segédadatszerkezetek elférjenek a RAM-ban. Minden $0 \leq s \leq e < n$-re az X[s..e] blokkot szeretnénk feldolgozni, ezzel előállítva a PLCP[s..e)-t a $\Phi[s..e)$ felhasználásával. A második lépés végrehajtása alatt csak az X[s..e) részét tároljuk a RAM-ban, az X többi részét a háttértárolóról fogjuk beolvasni. Ebben a lépésben (i,j,l) hármasokat fojunk számolni, ahol $i \in [s..e), j = \Phi[i], l \in [0..lcp(i,j)]$. A számítások során az l értékét egyenként foguk növelni, amikor szükséges. A hármasokat a második komponens alapján töltjük fel, így az X egyszeri végigolvasásával megkaphatjuk az lcp-t.

Előfordulhat olyan eset, amikor a számítás során az [s..e) intervallumhoz tartozó leghosszabb közös prefixek számolása egy másik blokkban ér véget. Ennek kezelésére egy R_y halmazban tároljuk az y blokkban be nem fejezett hármasokat. Így minden blokknak van egy további bemenő R_s paramétere, illetve egy R_e kimenő paramétere,melyek tartalmazzák a be nem fejezett hármasokat. Az eljárás pszudokódja a 2 algoritmuson látható.

Algorithm 2 Blokk feldolgozás [7]

```
1:  function ProcessBlock(s, e, X, Φ[s..e), R_s)
2:      Q = R_s ⋃{(i, Φ, 0) : i ∈ [s..e)}
3:      Sort Q by the second component
4:      j_prevs, l_prev = 0
5:      L = R_e = empty
6:      for (i, j, l) ∈ Q do
7:          l = max(l, l_prev + j_prev − j)
8:          while i+l > e and j + l < n and X[i+l] = X[j+l do do
9:              l = l + 1
10:         if i + l ≥ e and j + l < n then
11:             R_e = R_e ⋃{(i, j, l)}
12:         else
13:             L = L ⋃{(i, l)}
14:         j_prev = j, l_prev = l
15:     Sort L by the first component
16:     e' = min({e} ⋃{i : (i, j, l) ∈ R_e})
17:     s' = min({s} ⋃{i : (i, l) ∈ L})
18:     Return PLCP[s'..e'),R_e
```

4.3.3. Irreducibilis LCP értékek

A blokkok feldolgozása során a sok ismétlődést tartalmazó szövegeknek (például T=aaaaaaaaaaa) az R_s és R_e halmazai nagyon nagyra nőhetnek. Ennek elkerülése érdekében az irreducibilis LCP [12] értékek fogjuk bevezetni.

4.3.4. Értelmezés. *PLCP[i] reducibilis, ha X[i-1]=X[*$\Phi[i] - 1$*]. Különben vagy ha az i=0 vagy* $\Phi[i] = 0$*, akkor irreducubulis.*

Ennek felhasználásához, szükség van egy lemmára.

4.3.5. Állítás. *Ha PLCP[i] reducibilis, akkor PLCP[i]=PLCP[i-1]-1.*

A 4.3.4 értelmezést figyelembe véve az algoritmus némi változtatást igényel: a szöveg végigolvasásakor csak az irreducibilis LCP értékeket számoljuk,vagyis ha X[i-1]=X[$\Phi[i] - 1$], akkor az (i,j,l) hármas számolását elvetjük. Ezek után az 4.3.5 állítás alapján kiszámoljuk az irreducibilis LCP értékeket.

4.3.6. Az algoritmus műveletigénye

Az algoritmus minden egyes blokk feldolgozásakor végigolvassa a szöveget. Ez adja a képlet első felét. A képlet második fele a külső memóriás rendezésekből adódik,ahol a B a háttértárolón a blokkok hossza.

$$\mathcal{O}(\frac{n^2}{m} + n * \log_{\frac{m}{B}} \frac{n}{B})$$

4.3.7. További optimalizálások

Az első lépésben a hármasok rendezése során az előállított Φ tömböt nem írjuk ki egyből a háttértárolóra, hanem kiválogatjuk az irreducibilis pozíciókat, majd megalkotjuk a (i,Φ) párokat, rendezzük és így írjuk ki. A ProcessBlokk eljárás kezdetén így kevesebb értéket kell beolvasni a háttértárolóról és elegendő csak összefésülni az R_s halmazzal, azok rendezése helyett. Ezzel csökkentjük az I/O műveletek számát, ami hatékonyabb algoritmust eredményez.

4.3.8. Az algoritmus átalakítása

A fentebb bemutatott LCPscan [7] algoritmus futási ideje a gyakorlatban a külső memóriás rendezés futási idejétől függ. Annak érdekében, hogy a futási időből tudjanak faragni és a párhuzamosítás számottevő gyorsulást tudjon eredményezni a külső memóriás rendezést ki kell küszöbölni az algoritmusból. Ehhez pár fogalmat definiálunk, melyeket a [8] cikk alapján ismertetek.

4.3.9. Értelmezés. *Legyen* $q \geq 1$*. Legyen* $PLCP_q[0..n/q]$ *a ritka permutált leghosszabb közös prefix:* $PLCP_q[i] = PLCP[iq]$*, vagyis a PLCP minden q. elemét tartalmazza.*

4.3.10. Értelmezés. *Legyen* $\Phi_q[0..n/q]$*.* $\Phi_q[i] = \Phi[i * q]$*.*

Ekkor $PLCP_q[i] = lcp(q * i, \Phi_q[i])$. A ritka PLCP-t a PLCP tömörített reprezentálására használhatjuk, mivel a többi a PLCP többi elemére a következő korlátot ismerjük:

4.3.11. Állítás. $\forall i \in [0..n)$ *legyen:*

- $PLCP_q^{lo}(i) = max(0, PLCP_q[i/q] - (i - q * [i/q]))$
- $PLCP_q^{hi}(i) = \begin{cases} PLCP_q[i/q] + (q * [i/q] - i), & ha\ q * [i/q] < n \\ n - i - 1 & különben \end{cases}$

Ekkor $PLCP_q^{lo}(i) \leq PLCP[i] \leq PLCP_q^{hi}(i)$.

A fentebbi definíciókat ismerve, az $EM - S\Phi$ [10] algoritmus a következő képen néz ki:

1. Kiszámoljuk a $PLCP_q$ tömböt az X szövegből és az SA szuffix sorozatból. q-t úgy választjuk meg, hogy a $PLCP_q$ elférjen a RAM-ban, továbbá a szöveget felosztjuk olyan hosszú szegmensekre, hogy azok elférjenek a RAM-ban.
   (a) Kiszálmoljuk a Φ_q-t az SA végigolvasásával.$\forall i \in n$: ha SA[i mod q] = 0: $\Phi[SA[i]/q] = A[i - i]$. SA végigolvasása a háttértárolóról történik.
   (b) Minden szöveg szegmensre állítsunk elő egy fájlt. Állítsuk elő az összes (i,$\Phi[i]$) páros,úgy hogy i q-nak egész számú többszöröse, felhasználva a Φ_q-t. Írjuk ki az összes (i,$\Phi[i]$) párost a háttértárolóra, abba szövegszegmenshez tartozó fájlba, amelyik tartalmazza a $\Phi[i]$. Figyeljük meg, hogy a párok a fájlok miatt rendezve vannak.
   (c) Töltsük be az összes szegmenst a RAM-ba egyesével.Olvassuk be a szegmensekhez tartozó (i,Φ[i]) párokat, közben olvasva a szöveget, úgy hogy akkor érjük el az X[i]. karaktert, amikor az (i,$\Phi[i]$) párost dolgozzuk fel. Minden párosra számoljuk ki az lcp(i,$\Phi[i]$)-t és írjuk ki egy külön szegmenshez tartozó fájlba.Amikor az l=lcp(i,$\Phi[i]$)-t számoljuk, felhasználjuk azt a tényt, hogy $l \geq lcp(i', \Phi[i']) - (i - i')$, ahol $(i', \Phi(i')$ az előző lépésben kiszámolt pár, így az algoritmus sosem kell visszalépjen.
   (d) Rakjuk össze a $PLCP_q$-t, beolvasva a RAM-ba a $PLCP_q[i]$-t a $\Phi[i]$-t tartozó szegmenshez tartozó fájlból.
2. Számoljuk ki az LCP-t felhasználva a $PLCP_q$ a 4.3.11 állítás alapján. Ez alatt a lépés alatt a szöveg fél-szegmensekre osztjuk, és minden fél-szegmens párt betöltünk egyszer a RAM-ba.
   (a) Olvassuk végig az SA-t és hozzuk létre az összes (i,$\Phi[i]$) párt. Minden párra számoljuk ki az $l = PLCP_q^{lo}(i)$-t, majd írjuk ki az összes (i+l,$\Phi[i] + l$) párt a háttértárolóra, a fél-szegmens párok számára létrehozott fájlba.
   (b) Olvassuk be az összes fél-szegmenshez tartozó fájlt egyenként, majd számoljuk ki az összes (i,j) pára az lcp(i,j)-t és írjuk ki őket fájlba.
   (c) Olvassuk végig az SA-t és minden (i,$\Phi[i]$) párra számoljuk ki az $l = PLCP_q^{lo}(i)$ és olvassuk be az $l' = lcp(i + l, \Phi[i] + l)$ értéket a megfelelő fájlból. Ekkor $PLCP[i] = l + l'$ lesz a következő érték az LCP sorozatban.

4.3.12. Az algoritmus párhuzamosítása

Az algoritmus több lépésében az adatok feldolgozása egymástól függetlenül, egy intervallumon történik. Ezeket triviális párhuzamosítani a feladatok egyenlő mértékű szétosztásával a szálak között.

Az algoritmusnak van azonban nem triviálisan párhuzamosítható része, ilyen például az 1(c). A feldolgozandó párokat egyenletesen szétosztjuk a szálak között, majd mindegyik szál feldolgozza azokat. A feldolgozás folyamán a szálak általában által végigolvasott szöveg kevés. Vannak azonban olyan esetek, tipikusan ismétlődő szövegek, pl DNS, amikor nagy az átfedés a végigolvasott szöveg között. Mivel a szöveg olvasása a háttér tárolóról lassú, ezért a párhuzamosítás, nem fog nagy gyorsulást eredményezni. Az átfedés megszüntetése érdekében egy adott p-re, p»q, kiszámolunk egy szuper-ritka $PLCP_p$ tömböt, amely segítségével tudunk számolni egy alsó korlátot. Ennek az alsó korlátnak a felhasználásával megszüntethetjük az szöveg olvasásban történt átfedést. Annak köszönhetően, hogy kevesebb I/O műveletet hajtunk végre, gyorsulást fog eredményezni a párhuzamosítás.

A második nem triviális párhuzamosítás az algoritmusban a 2(b) lépés.Itt is a különböző lcp(i,j) értékeket egyenletesen szétosztjuk a szálak között. Az lcp(i,j) érték számításának ideje, függ a megegyező prefix hosszától. Ezért a szálak közötti terhelés elosztás nehéz, a nem megfelelő terhelés elosztása a magok nem optimális kihasználásával egyenértékű. A hosszú prefixek általában csoportosan jönnek. Ez azt jelenti, hogy egy szál futási ideje jóval hosszabb lesz mint a többi szálé, ami a gyorsulás kárára megy. Ennek kiküszöbölésére mérjük az átlagos lcp(i,j) számítási időt és ha valamelyik számítás átlép egy küszöböt, akkor annak számítását leállítjuk és egy tárolóba gyűjtjük. A számítások végén a tárolót párhuzamosan rendezzük a (j-i) különbségek alapján, majd a irreducibilis állítás felhasználásával kiszámítjuk ezeket az lcp(i,j) értékeket is. Ennek köszönhetően a sok ismétlődést tartalmazó szövegekben még 1 szálon is gyorsabb lesz az algoritmus.

A cikk [10] mérései alapján a párhuzamosítással már nyolc szálon felére tudták csökkenteni a futási időt.

5. fejezet

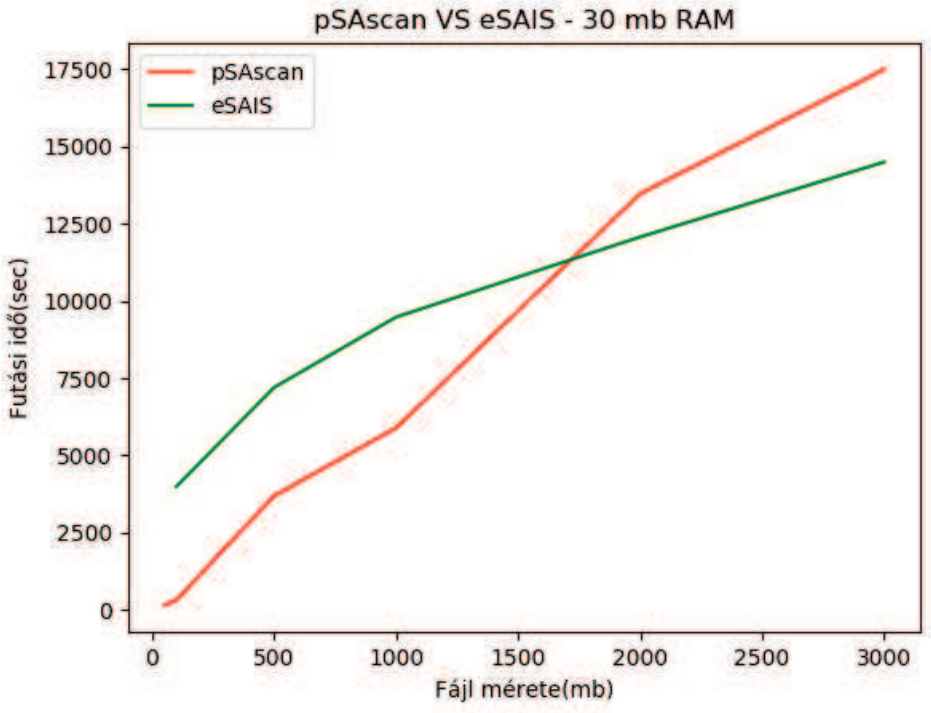
Mérési eredmények

A méréseket két laptopon végeztem el. Az egyikben egy Intel(R) Core(TM) i5-5200U, 2.20GHz processzor volt, amelyik 2 maggal rendelkezik és 4 szálat tud egyszerre futtatni. Ezentúl "Gép S"-sel fogok rá hivatkozni. A második gépben egy Intel(R) Core(TM) i7-7700HQ, 3.80GHz processzor volt, amelyik 4 maggal rendelkezik és 8 szálat tud egyszerre futtatni. Ezentúl "Gép L"-lel fogok rá hivatkozni.

A teszteseteket minden esetben random generáltam az angol ábécé kisbetűiből.

5.1. Külső memóriás algoritmusnak futási ideje

A fentiekben ismertetett pSAscan és eSAIS algoritmusokat futattam le különböző tesztfájlokra. A tesztek mérete 50 megabyte-tól 3 gigabyte-ig terjedt. Az algoritmusok számára rendelkezésre álló memóriát 30 megabyte-ra korlátoztam.

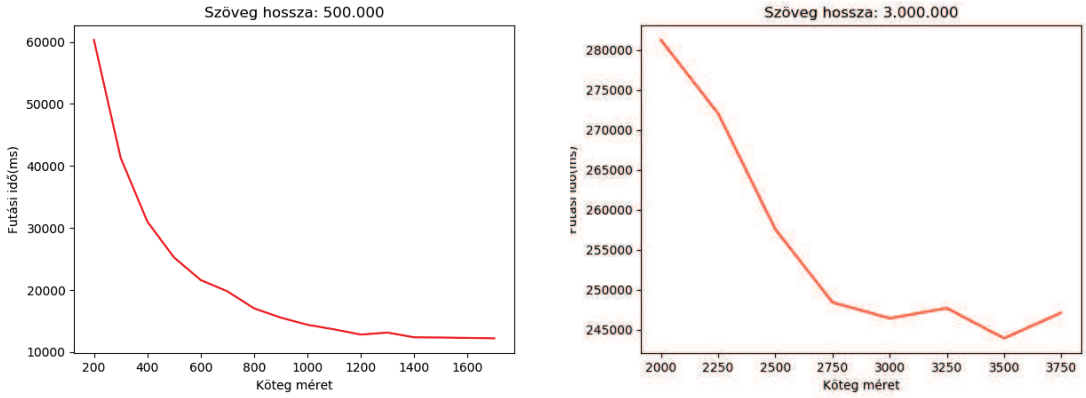


5.1. ábra. pSAscan és eSAIS

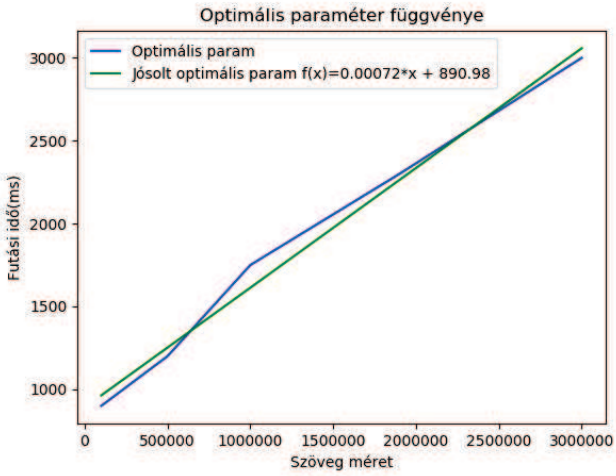
Az 5.1 ábra szemlélteti, hogy ha az input mérete nem haladja meg a rendelkezésünkre álló RAM 60-szorosát, akkor a pSAscan algoritmus a gyorsabb, ellenkező esetben az eSAIS. Az algoritmusok bemutatásakor láttuk, hogy az eSAIS aszimptotikusan gyorsabb, mint a pSAscan.Az eSAIS külsű memória használata 28*n byte, míg a pSAscan által használt mennyiség csak 7.5*n byte. Az I/O műveletek költsége miatt lehetséges, hogy egy aszimptotikusan lasabb algoritmus gyorsabban fut bizonyos inputokra.

5.2. Kötegelt változat optimális paramétere

Az előző fejezetben bemutatott kötegelt algoritmusban a kötegek mérete bemeneti paraméter. A köteg méretének növelésével nő a tárigény és csökken a futási idő, egy adott pontig. A 5.2 ábrán látható két bemeneti fájlra, hogy a köteg mérete alapján, hogy változik a futási idő különböző méretű bemeneti fájlokra. Ezeket a méréseket elvégeztem több bemeneti fájlra, majd minden fájlra a kapott legjobb idők alapján, meghatároztam az optimális kötegméretet a bemeneti szöveg méretének függvényében, ahogy azt a 5.3 ábra is mutatja. A méréseket a "Gép S"-en végeztem.



5.2. ábra. Köteg mérete és futási idő összehasonlítása

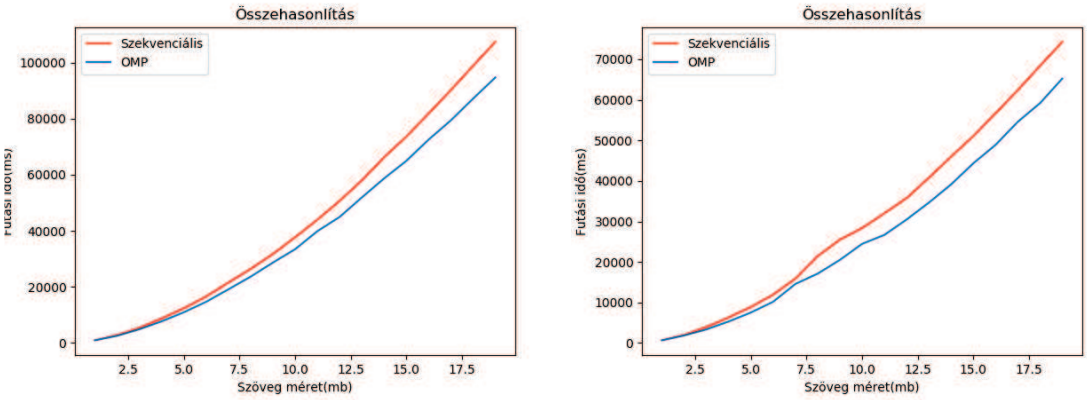


5.3. ábra. Optimális paraméter az input méretének függvényében

5.3. Párhuzamos implementációk futási ideje

A mérések során az előző részben meghatározott optimális paramétert használtam az input fájl mérete alapján kötegméretnek. A méréseket elvégeztem mindkét gépen, a grafikonok párban fognak szerepelni, bal oldalt az S, míg jobb oldalt az L gép.

Az 5.4 ábrán a szekvenciális és az OMP könyvtár segítségével megvalósított párhuzamosított változat futási ideje látszik 20 darab kisebb bemenetre. Az OMP könyvtár a megfelelő paraméterbeállítások melett nagyon leegyszerűsíti a párhuzamos program írását, viszont csak nagyon általános feladatokra használható.

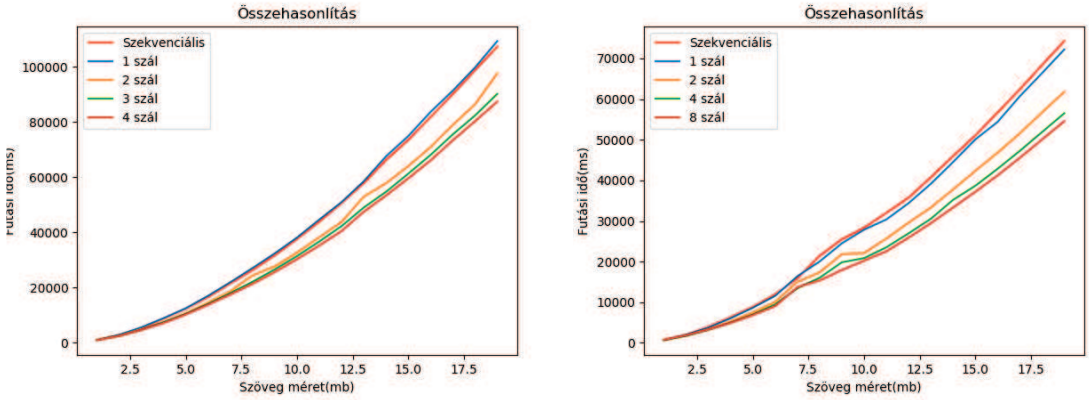


(a) Gép S (b) Gép L

5.4. ábra. Szekvenciális és OMP összehasonlítása

A következő méréshez implementáltam a BWT egyenlő részekre való felosztását, majd minden részhez új szál indításával végeztem el a 4.5 ábrán szemléltetett algoritmust. A 5.5 ábrán látható, hogy különböző számú szálak használata esetén, egyre jobban gyorsul az algoritmus.A piros görbe szemlélteti a szekvenciális futást, míg a kék az egyszálú futást. A különbség a kettő között, hogy az 1 szálú futás

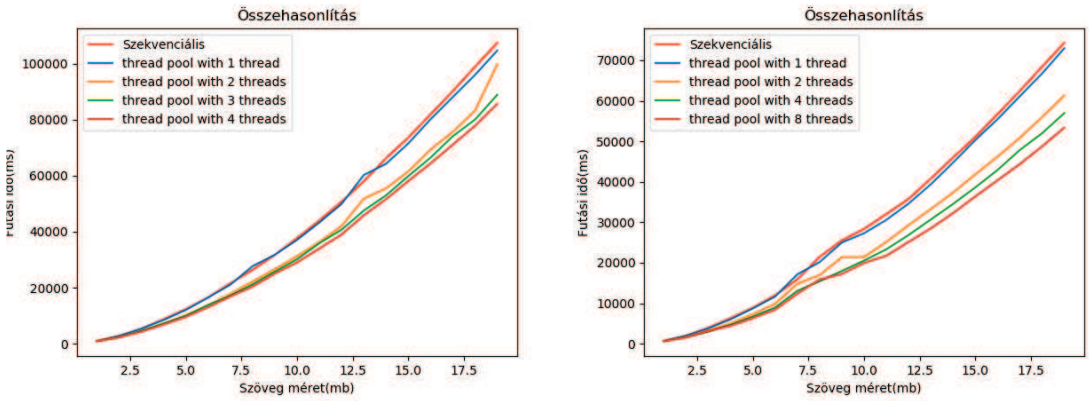
létrehozott 1-1 új szálat a számításokhoz. Ahogy az ábra is szemlélteti, ezek a futási idők majdnem megegyeznek.



(a) Gép S (b) Gép L

5.5. ábra. Különböző számú szálak esetén

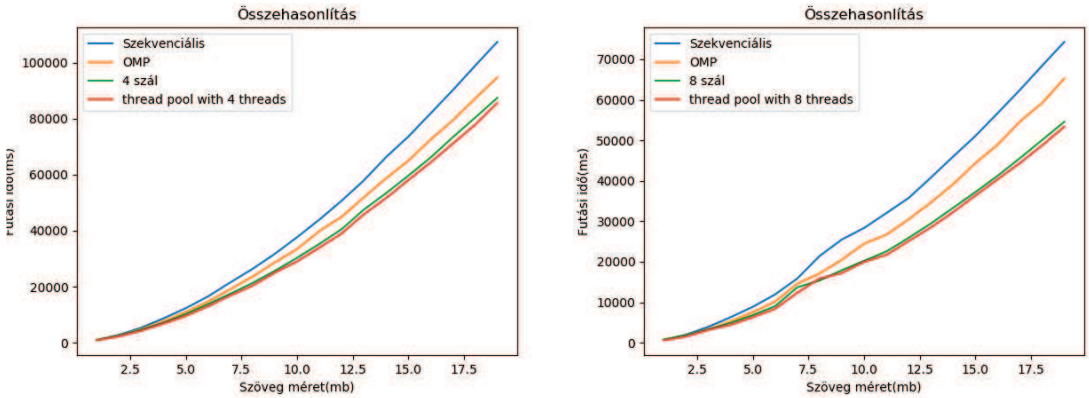
Az előző implementációban a szálak létrehozása költséges, ezért a 3. fejezetben említett angol nevén "thread pool" implementációt használtam. A program elején létrehoztam annyi szálat, ahányat az adott gép futtatni tud párhuzamosan, majd a feladatokat egy konkurens sorba helyeztem, ahonnan a szálak folyamatosan dolgozták fel azokat. A futási idő a 5.6 ábrán látható. Ezen az ábrán is jól látszik, hogy minél több szálat használunk, annál nagyobb gyorsulást fogunk elérni.



(a) Gép S (b) Gép L

5.6. ábra. Különböző számú szálat felhasználva a thread poolhoz

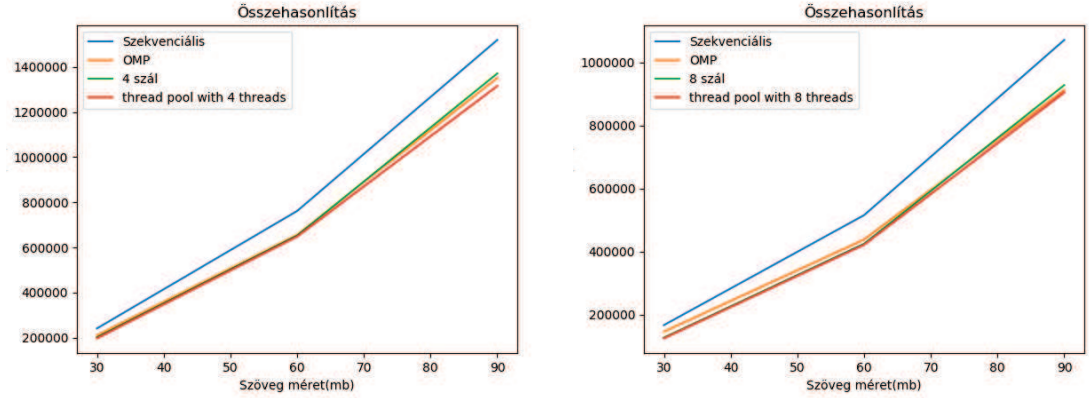
A 5.7 ábrák a különböző párhuzamosítási módszerek összefoglalója látszik kisebb fájlokra. Jól látszik, hogy a "thread pool" megközelítés érte el a legjobb eredményt, ugyanis ez kiküszöböli a sok szál létrehozásának költségét és testre szabható a párhuzamosítási feladat. Az OMP-vel is sikerült gyorsulást elérnünk, ám kevesebbet mint a másik két megközelítéssel, mivel ez a háttérben zárolást használ a szinkronizációhoz.



(a) Gép S (b) Gép L

5.7. ábra. Összes módszer összehasonlítása kisebb méretű bemeneten

A 5.8 ábrán a különböző párhuzamosítási módszerek látszanak nagyobb fájlokra. Itt a párhuzamosítási módok, nagyjából ugyan olyen eredményt értek el.



(a) Gép S (b) Gép L

5.8. ábra. Összes módszer összehasonlítása nagyobb méretű bemeneten

A dolgozat készítése során kipróbáltam a c++ 17-es szabványában definiált új párhuzamos algoritmusokat, azonban itt nem lehetett bevetni a 4.5 ábrán szemléltetett trükköt a szinkronizáció elkerülése érdekében, ez az implementáció lassabb lett, mint a szekvenciális megoldás. Az implementáció során a C tömbnek az elemei a nyelven definiált std::atomic változók voltak. Ez a típus konstrukció garantálja, hogy meg nem szakíthatóan hajtódnak végre rajta az utasítások, biztosítva a szinkronizációt.

6. fejezet

Összegzés

A dolgozatomban bemutatásra kerültek különböző sztring algoritmusok és a hozzájuk kapcsolódó adatszerkezetek párhuzamos előállításának módjai.

6.1. Szuffix sorozat

A szuffix sorozat előállításánál, amennyiben az input mérete nem haladja meg a RAM ötödét és nincs lehetőségünk több szálat használni a divsufsort [19] algoritmust érdemes használni.

Amennyiben lehetőségünk van több szálat használni és az input mérete meghaladja a RAM ötödét, illetve nem haladja meg a ram 60-szorosát, futási idő szempontjából a pSAscan [11] algoritmust érdemes használni futási idő szempontjából.

Amennyiben az input fájl hossza meghaladja a RAM 60-szorosát az eSAIS algoritmust érdemes használnunk.

6.2. Burrows-Wheeler transzformált

Amennyiben futási idő a fő szempont a fent említett algoritmusokat felhasználva, megfelelően paraméterezve könnyen előtudjuk állítani a BWT-t.

Bár nem gyakori eset, de nem elképzelhetetlen, hogy spórolnunk kell a memóriával: például egy kevesebb RAM-mal rendelkező vagy háttértárolóval nem rendelkező architektúrán kell kiszámolnunk a BWT-t vagy egy külső memóriás algoritmus részleges BWT-ket akar számolni helytakarékosan. Ebben az esetben érdemes lehet a helyben előállító Burrows-Wheeler transzformált párhuzamos, kötegelt változatát használni. Ennek tudomásom szerint még nem létezett implementációja. A dolgozathoz tartozó adathordozón megtalálható mind a szekvenciális, mind a párhuzamos implementáció.

6.3. Párhuzamosítási lehetőségek

A dolgozatom során kipróbáltam több párhuzamosítási lehetőséget a c++ nyelvben. Az OpenMP[23] és a c++17-ben definiált algoritmusok általános megvalósítások, könnyen használhatóak, viszont az általánosság miatt szinkronizációs adatszerkezeteket igényelnek. Ezek miatt könnyű velük dolgozni és karban tartani őket, viszont ha a feladatra nincs jól illeszkedő, általános minta, akkor nem optimálisak.

Az alacsonyabb szálszintű párhuzamosítást (std::thread vagy a thread pool) nagyobb eséllyel lehet a feladatra szabni, viszont gondos tervezést igényel, nehezebb karbantartani, ugyanakkor hatékonyabb.

6.4. Leghosszabb közös prefix

A dolgozatomban bemutattam egy külső memóriás, előre kiszámolt SA sorozatból LCP tömböt előállító párhuzamos algoritmust. Ennek bemutatása során előtérbe kerültek azok a problémák, amelyek miatt az algoritmus futási ideje párhuzamosítással nem javulna, továbbá különböző matematikai megfontolások segítségével ezen korlátok leküzdése, illetve a párhuzamosítás részleteinek bemutatása.

7. fejezet

Irodalomjegyzék

[1] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. The enhanced suffix array and its applications to genome analysis. In *Proceedings of the Second International Workshop on Algorithms in Bioinformatics*, WABI '02, pages 449–463, London, UK, UK, 2002. Springer-Verlag.

[2] Timo Bingmann, Johannes Fischer, and Vitaly Osipov. Inducing suffix and lcp arrays in external memory. In *Journal of Experimental Algorithmics*, volume 21, 09 2016.

[3] Maxime Crochemore, Roberto Grossi, Juha Kärkkäinen, and Gad M. Landau. Computing the burrows–wheeler transform in place and in small space. *Journal of Discrete Algorithms*, 32:44 – 52, 2015. StringMasters 2012 2013 Special Issue (Volume 2).

[4] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52:552–581, July 2005.

[5] Keijo Heljanko. Java concurrency. `https://mycourses.aalto.fi/pluginfile.php/364392/mod_resource/content/1/lecture3.pdf`. [Online; megnézve 29-április-2018].

[6] Juha Kärkkäinen. Finn kutató csapat. `https://www.cs.helsinki.fi/group/pads/`, 2017. [Online; megnézve 29-április-2018].

[7] Juha Kärkkäinen and Dominik Kempa. Lcp array construction in external memory. In Joachim Gudmundsson and Jyrki Katajainen, editors, *Experimental Algorithms*, pages 412–423, Cham, 2014. Springer International Publishing.

[8] Juha Kärkkäinen and Dominik Kempa. Faster External Memory LCP Array Construction. In Piotr Sankowski and Christos Zaroliagis, editors, *24th Annual European Symposium on Algorithms (ESA 2016)*, volume 57 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 61:1–61:16, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[9] Juha Kärkkäinen and Dominik Kempa. Engineering a lightweight external memory suffix array construction algorithm. *Mathematics in Computer Science*, 11:137–149, Jun 2017.

[10] Juha Kärkkäinen and Dominik Kempa. Engineering External Memory LCP Array Construction: Parallel, In-Place and Large Alphabet. In Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman, editors, *16th International Symposium on Experimental Algorithms (SEA 2017)*, volume 75 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[11] Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Parallel external memory suffix sorting. In *Combinatorial Pattern Matching*, pages 329–342, Cham, 2015. Springer International Publishing.

[12] Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. Permuted longest-common-prefix array. In Gregory Kucherov and Esko Ukkonen, editors, *Combinatorial Pattern Matching*, pages 181–192, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[13] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Automata, Languages and Programming*, pages 943–955, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[14] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In Amihood Amir, editor, *Combinatorial Pattern Matching*, pages 181–192, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[15] Sheryl Koenigsberg. Relative Speeds from RAM to Flash to Disk. `http://blog.infinio.com/relative-speeds-from-ram-to-flash-to-disk`, 2014. [Online; megnézve 29-április-2018].

[16] Juha Kärkkäinen. Parallel external memory suffix sorting. `http://www.cs.ucr.edu/~stelo/cpm/cpm15/28_Karkka.pdf`. [Online; megnézve 29-április-2018].

[17] Kempa D. Kärkkäinen, J. Engineering a lightweight external memory suffix array construction algorithm. In *CEUR Workshop Proceedings*, ICABD 2014, pages 53–60, 2014.

[18] Yuta Mori. Divsufsort bencjmarks. `https://github.com/y-256/libdivsufsort/blob/wiki/SACA_Benchmarks.md`. [Online; megnézve 29-április-2018].

[19] Yuta Mori. libdivsufsort. `https://github.com/y-256/libdivsufsort`. [Online; megnézve 29-április-2018].

[20] L.H. Pollard. *Computer design and architecture*. Prentice Hall, 1990.

[21] Zoltán Porloláb. Concurrency. `http://gsd.web.elte.hu/lectures/multi/slides/concurrency.pdf`. [Online; megnézve 29-április-2018].

[22] Herb Sutter. A fundamental turn toward concurrency in software-the face of hardware is changing, impacting the way you'll be writing software in the future.

Dr Dobb's Journal-Software Tools for the Professional Programmer, pages 16–23, 2005.

[23] Openmp. `https://www.openmp.org/`. [Online; megnézve 29-április-2018].

[24] Thread pool. `https://github.com/vit-vit/CTPL`. [Online; megnézve 29-április-2018].