SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 2320

GPU implementacija vremenski učinkovitog algoritma za lokalno poravnavanje s linearnom memorijskom složenosti

Goran Žužić

Zagreb, lipanj 2012.

Hvala Mili Šikiću što me je uveo u područje koje me jedinstveno zanima i kojim ću se zasigurno još neko vrijeme baviti. Također, hvala na slobodi, mentoriranju i strpljenju koje mi je iskazao.

SADRŽAJ

1. Uvod

Biološki podaci se otkrivaju fenomenalnom brzinom. Primjerice GenBank, repozitorij svih javno dostupnih nukleonskih sekvenci, je već u travnju 2001. godine sadržavao preko 11.5 milijuna pojedinih sekvenci i u prosjeku se njegova veličina udvostručava svakih 15 mjeseci [1]. Ako tim brojevima dodamo mnoštvo ostalih projekata koji proučavaju genetsku i proteinsku strukturu zajedno s međusobnim interakcijama, počinjemo shvaćati nepreglednost otkrivenih informacija.

Kao rezultat te eksplozije, računala su postala neizbježan alat u biološkim istraživanjima zbog mogućnosti obrade velikih količina podataka koje dobivamo iz prirode. Bioinformatika, subjekt ovog uvoda, se često definira kao primjena računalnih tehnika kako bi se razumjele i organizirale informacije pridružene biološkim fenomenima.

Ovdje je bitno spomenuti pomalu neočekivanu povezanost biologije kao „analogne i deskriptivne“ znanosti te bioinformatike kao „digitalne i egzaktne“ grane. Ova unija se objašnjava činjenicom da se organizam kao takav može opisati informacijskom tehnologijom; njegova fiziologija je velikim dijelom određena genima na koje se može gledati kao potpunu digitalnu informaciju. Nadalje, veliki napredak u tehnologijama otkrivanja bioloških podataka prati sukladno povećanje obradbene moći računala koja omogućuju izračun kompleksnijih bioloških modela koji bolje simuliraju prirodne procese [2].

Veliki spektar područja kojim se bavi bioinformatika može biti klasificiran prema informacijama koje se koriste u obradi. Za neobrađene DNA sekvence, istraživanja uključuju razdvajanje kodirajućih i nekodirajućih regija, identifikacija introna, eksona i promotor regija, kao i usporedba dvaju DNA sekvenci kako bi se našli najveći preklapajući nizovi. Za proteinske sekvence, analize uključuju razvoj algoritama za usporedbu i poravnavanje višestrukih sekvenci (slično kao i kod DNA sekvenci), te potraga za funkcionalnim motivima u takvim poravnavanjima. Istraživanje strukturalnih podataka uključuju predviđanja 3D poravnavanja između dvaju proteina, proučavanje geometrijske strukture koristeći udaljenosti, kutove, površine i volumene kako bi se simuliralo kretanje među drugim makromolekulama i radi predviđanja energija

uključenih u spajanju (engl. docking) proteina [3]. Od navedenog, ovaj rad će se prvenstveno baviti poravnavanjem nukleinskih i proteinskih sekvenci te pripadajućim algoritmima.

1.1. Poravnavanje nukleinskih sekvenci

Problem poravnavanja sekvenci je način ocjenjivanja sličnosti između dvaju ili više sekvenci i kao takav zauzima jednu od ključnih uloga u bioinformatici. U praksi se poravnavaju DNA, RNA ili proteinske sekvence te im se pridružuje ocjena sličnosti na temelju dobivenog poravnavanja. Rezultati takvih poravnavanja ukazuju na funkcijske, strukturalne i evolucijske veze između danih sekvenci [4].

Kada pričamo o sekvencama, podrazumijevamo uređene nizove gdje jedan element predstavlja pojedini nukleotid ili aminokiselinu. Tako ćemo primjerice DNA sekvence predstaviti nizovima nad abecedom { A, G, T, C } koji redom predstavljaju nukleotide adenin, guanin, timin i citozin.

... A A G T C C G G ...

Slika 1.1: Manji dio DNA sekvence

S ciljem matematičkog definiranja problema poravnavanja dvaju sekvenci potrebno je proučiti kako se DNA može mijenjati kroz vrijeme. Evolucija unosi stalne i slučajne promjene u genetski kod pomoću djelovanja raznih fizikalnih i kemijskih procesa koje zovemo mutacije. Biolozi su izolirali 3 vrste mutacije koje su odgovorne za veliku većinu genetskih promjena [5]:

- Supstitucije jednog nukleotida drugim
- Ispuštanja niza nukleotida
- Umetanja niza nukleotida

Ocjenu sličnosti dvaju DNA sekvenci definirat ćemo kao minimalni broj gore navedenih mutacija da prvu sekvencu pretvorimo u drugu sekvencu[1]. Formalno, neka su A i B sekvence nad abecedom Σ te neka ϵ predstavlja prazan niz. Poravnati par ima oblik $\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right]$ gdje su $a, b \in \Sigma \cup \{\epsilon\}$. Poravnavanje A i B je konačni niz poravnatih parova $\{L_1, L_2, ..., L_k\}$ takav da konkatenacija gornjih elemenata producira A, dok konkatenacija donjih elemenata producira B. Poravnati parovi kojima je gornji element ϵ zovu

[1] Pritom neće nužno sve mutacije imati istu cijenu.

se umetački parovi, dok se oni kojima je donji element ϵ zovu brisački parovi. Preostali se parovi nazivaju supstitucijski. Nadalje, definiramo procjep unutar poravnanja kao niz uzastopnih umetačkih ili niz uzastopnih brisačkih parova. Za svaki umetački, brisački i supstitucijski par se definira sličnost kao $\sigma(\left[\begin{smallmatrix} x \\ y \end{smallmatrix}\right])$ koji sličnim parovima pridjeljuje pozitivnu ocjenu, dok ostalima pridjeljuje negativnu ocjenu. Takve sličnosti se obično pohranjuju u tzv. supstitucijske matrice koje su postale neophodne pri definiranju problema poravnavanja sekvenci. Primjeri takvih supstitucijskih matrica su PAM30, PAM70, BLOSUM62 i CCF53 [6].

Spremni smo dati matematičku definiciju problema globalnog poravnavanja. Potrebno je pronaći takvo poravnavanje sekvenci A i B čija je sličnost maksimizirana. Pritom se sličnost definira kao suma sličnosti svakog pojedinog para poravnavanja umanjenog za ukupni broj procjepa pomnoženim s procjepnom kaznom $g > 0$. Drugim riječima, sličnost poravnavanja se definira kao [7]:

$$\sum_{i=1}^{k} \sigma(L_i) - g \cdot |\text{procjepi}|$$

Takva definicija sličnosti se naziva *sličnost s afinim cijenama procjepa*.

1.1.1. Lokalna poravnavanja i višestruka lokalna poravnavanja

Do ovog trenutka smo definirali problem globalnog poravnavanja sekvenci. Iako je dotični problem vrlo bitan, puno je važniji problem *lokalnog poravnavanja* koji se slično definira. Nazovimo podsekvencom neke sekvence S bilo koji niz kojeg dobijemo brisanjem proizvoljnog broja znakova s početka i s kraja originalnog niza[2]. Problem lokalnog poravnavanja nizova A i B se definira kao maksimalno globalno poravnavanje bilo koje podsekvence od A i bilo koje podsekvence od B.

Sličan lokalnom poravnavanju je i problem višestrukih lokalnih poravnavanja. Definira se iterativno: prvo pronađemo najbolje lokalno poravnavanje i potom pogledamo sve njegove supstitucijske parove: svaki takav pojedini par supstituira i-ti znak niza A s j-tim znakom niza B. Mi potom zabranimo dani supstitucijski par u svim potonjim lokalnim poravnavanjima[3] i ponavljamo postupak traženja najboljeg preostalog lokalnog poravnavanja onoliko puta koliko poravnavanja želimo [7]. Ukupni broj traženih poravnavanja ćemo u ostatku rada označavati varijablom K.

[2]Dopušteno je obrisati i nula znakova s bilo kojeg kraja.

[3]Ovdje ne zabranjujemo supstitucijski par između neka dva elementa abecede Σ, već zabranjujemo supstituciju i-tog elementa s j-tim elementom.

1.2. Cilj rada

Ovaj rad će dati pregled najboljih sekvencijalnih algoritama koji rješavaju problem lokalnog i višestrukog lokalnog poravnavanja sekvenci. Započet ćemo s jednostavnim Smith-Waterman algoritmom za afine cijene procjepa te ga potom optimizirati Hirscbergovom opservacijom kako bismo dobili linearno-memorijski Myers-Millerov algoritam. Poglavlje o sekvencijalnim algoritmima završit ćemo, za ovaj rad najvažnijim, Huang-Millerovim algoritmom koji u linearnoj memoriji i s sublinearnim vremenom s obzirom na K rješava problem višestrukog poravnavanja sekvenci.

Drugi dio rada bavit će se paralelnim izvedbama navedenih algoritama kako bi se postigla što bolja vremenska ušteda nad sekvencijalnim analozima. Dat ćemo pregled napravljenog (analoge Smith-Waterman i Myers-Miller algoritma) i istražiti moguća proširenja na Huang-Millerov algoritam.

2. Algoritmi za poravnavanje

U ovom poglavlju ćemo dati pregled algoritama za poravnavanje sekvenci. Predstavljeni algoritmi se generalno nadograđuju jedan na drugoga, pritom ilustrirajući povijest razvoja bioinformatičkih metoda. Međusobno se razlikuju u načinu implementacije, paraleliziranosti te vremenskom i memorijskom složenosti.

2.1. Sekvencijalni (CPU) algoritmi

2.1.1. Lokalno poravnavanje - Smith-Waterman algoritam

Osnovnu ideju za rješavanje problema globalnog poravnavanja sekvenci dali su S. Needleman i C. Wunsch 1970. godine u [8]. Sljedeći pomak došao je 1981. kada su T. Smith i M. Waterman u [9] predstavili jednostavnu preinaku Needleman-Wunschovog algoritma. Tako dobiveni algoritam je računao najbolje lokalno poravnavanje između sekvenci.

Uvedimo nekoliko oznaka koje ćemo koristiti kroz ovaj rad:

A
: Prva sekvenca

B
: Druga sekvenca

M
: Duljina prve sekvence, $|A|$

N
: Duljina druge sekvence, $|B|$

$A_{a..b}, B_{a..b}$
: Podsekvenca $A_a, A_{a+1}, ..., A_b$ odnosno $B_a, B_{a+1}, ..., B_b$

Kako bismo što elegantnije objasnili teoretsku podlogu Smith-Watermanovog algoritma, preuzet ćemo vrlo lucidnu notaciju predstavljenu u [7]. Definirajmo *graf poravnavanja* $G_{A,B}$ kao usmjereni graf s imenovanim bridovima koji se gradi specifično za dvije sekvence A i B:

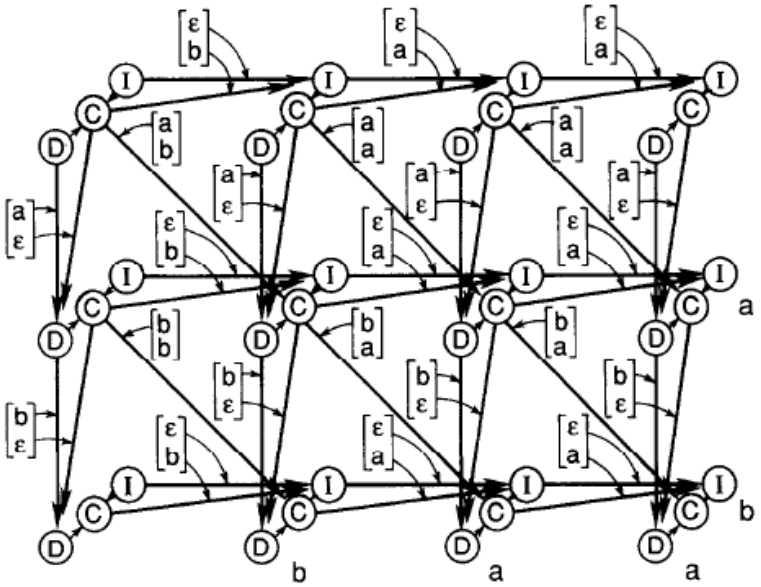
1. G ima 3(M+1)(N+1) vrhova koje ćemo označavati s $(i,j)_C, (i,j)_D, (i,j)_I$ gdje je $i \in \{0,1,...,M\}, j \in \{0,1,...,N\}$.

2. Za $i \in \{1,2,...,M\}, j \in \{0,1,...,N\}$ postoji brid $(i-1,j)_D \rightarrow (i,j)_D$ te $(i-1,j)_C \rightarrow (i,j)_D$ s oznakom brisačkog para $\left[\begin{smallmatrix} A_i \\ \epsilon \end{smallmatrix}\right]$.

3. Za $i \in \{0,1,...,M\}, j \in \{1,2,...,N\}$ postoji brid $(i,j-1)_I \rightarrow (i,j)_I$ te $(i,j-1)_C \rightarrow (i,j)_I$ s oznakom umetačkog para $\left[\begin{smallmatrix} \epsilon \\ B_j \end{smallmatrix}\right]$.

4. Za $i \in \{1,2,...,M\}, j \in \{1,2,...,N\}$ postoji brid $(i-1,j-1)_C \rightarrow (i,j)_C$ s oznakom supstitucijskog para $\left[\begin{smallmatrix} A_i \\ B_i \end{smallmatrix}\right]$.

5. Za $i \in \{0,1,...,M\}, j \in \{0,1,...,N\}$ postoji brid $(i,j)_D \rightarrow (i,j)_C$ te $(i,j)_I \rightarrow (i,j)_C$ koji oboje imaju praznu oznaku[1].



Slika 2.1: Graf poravnavanja $G_{A,B}$ za A=„ab“ i B=„baa“ preuzet iz [7]

Svojstvo grafa je da put od $(m,n)_C$ do $(i,j)_C$ predstavlja poravnavanje podsekvenci $A_{m..i}$ i $B_{n..j}$ ukoliko u poretku zapišemo oznake bridova na koje smo naišli na putu.

[1] Praznu oznaku brida ćemo prigodno označavati s $\left[\begin{smallmatrix} \epsilon \\ \epsilon \end{smallmatrix}\right]$ uz napomenu da $\sigma(\left[\begin{smallmatrix} \epsilon \\ \epsilon \end{smallmatrix}\right]) = 0$.

Težina puta se slično definira tako da na svakom bridu dodamo $\sigma([\begin{smallmatrix}x\\y\end{smallmatrix}])$ u globalnu sumu, uz dodatnu napomenu da za svaki brid u kojem započinjemo procjep dodatno oduzmemo g (dakle na bridovima $(i-1,j)_C \to (i,j)_D$ i $(i,j-1)_C \to (i,j)_I$).

S time smo pretvorili problem lokalnog poravnavanja u traženje najtežeg puta na grafu. Ako primijetimo da graf nema ciklusa, poznato je da možemo efikasno izračunati duljinu najtežeg puta ukoliko topološki sortiramo vrhove i u tom poretku izračunamo maksimalnu moguću težinu puta do svakog pojedinog vrha. Na potonjoj slici je prikazan jedan očiti topološki poredak kojeg ćemo nadalje koristiti u ovom radu:

$$
\begin{aligned}
(0,0)_D, (0,0)_I, (0,0)_C,\\
(0,1)_D, (0,1)_I, (0,1)_C,\\
(0,2)_D, (0,2)_I, (0,2)_C,\\
...,\\
(0,M)_D, (0,M)_I, (0,M)_C,\\
(1,0)_D, (1,0)_I, (1,0)_C,\\
...,\\
(N,M)_D, (N,M)_I, (N,M)_C
\end{aligned}
$$

Lako je primijetiti da u njemu svi bridovi od nekog vrha idu isključivo prema vrhovima kasnije navedenim u nizu. Kako bi pokazali implementacijsku ideju, prikažimo s:

$C(i,j)$

Težinu najtežeg puta koji završava u $(i,j)_C$

$D(i,j)$

Težinu najtežeg koji završava u $(i,j)_D$

$I(i,j)$

Težinu najtežeg puta koji završava u $(i,j)_I$

I sada nam preostaje ići po topološkom poretku vrhova i računati redom $I(i,j), D(i,j), C(i,j)$ čije se formule jednostavno mogu interpolirati iz gornjeg grafa.

Kao zadnju napomenu, primijetimo kako su i vremenska i prostorna složenost algoritma točno $O(MN)$.

2.1.2. Hirschbergova optimizacija za linearnu memoriju - Algoritam Myers-Miller

U Smith-Waterman algoritmu, lako je primijetiti kako je memorijska složenost veći ograničavajući faktor od vremenske složenosti. Ukoliko poravnavamo dvije sekvence veličine 1 MB, potrebno će nam biti otprilike 1 sat vremena i 12 TB radne memorije[2]. Potrebno je pronaći način za smanjivanje memorijske složenost.

Pomak u razvoju se dogodio 1975. kada je D. S. Hirschberg u [10] dao linearni algoritam za problem Levenshteinove udaljenosti, problemu blizancu poravnavanja sekvenci. Ipak, do konačnog poboljšanja se došlo tek 1988. kada su E. W. Myers i W. Miller u [11] primijenili spomenutu Hirschbergovu optimizaciju za problem poravnavanja sekvenci.

Ideja poboljšanja je sljedeća: prvo je potrebno svesti problem lokalnog poravnavanja na problem globalnog poravnavanja, te je potom potrebno riješiti problem globalnog poravnavanja u linearnoj memoriji. Pritom je osnovna opservacija da analogno tome što možemo izračunati najteži put koji završava s nekim vrhom tako da obilazimo vrhove u topološkom poretku, možemo izračunati i najteži put koji započinje u nekom vrhu ukoliko vrhove obilazimo u obrnutom topološkom poretku.

Svođenje globalnog poravnavanja na lokalno poravnavanje

Prilikom računanja vrijednosti $C(i,j), D(i,j), I(i,j)$ po gore navedenom topološkom poretku, nije potrebno pamtiti sve prethodno izračunate vrijednosti. Potrebno je pratiti samo one vrijednosti koje se pojavljuju u prijašnjem retku. Nažalost, takav algoritam nije dobar iz razloga što nećemo znati rekonstruirati optimalan segment u kojem se dogodilo preklapanje[3]. Ipak, takav algoritam nas navodi na pravi trag jer ipak možemo saznati zadnji vrh na optimalnom putu. Označimo taj vrh s $(X,Y)_C$[4]. Sada računajući najteži put koji započinje u pojedinom vrhu i završava u $(X,Y)_C$ obilazeći vrhove u obrnutom topološkom poretku lako dolazimo do jednog od mogućih početaka optimalnog puta. Taj vrh označimo s $(Q,W)_C$.

[2]Ukoliko za svaki od $3NM$ vrhova spremamo 32-bitni podatak.

[3]U rječniku teorije grafova, znat ćemo težinu optimalnog puta, ali ne i koji je optimalan put.

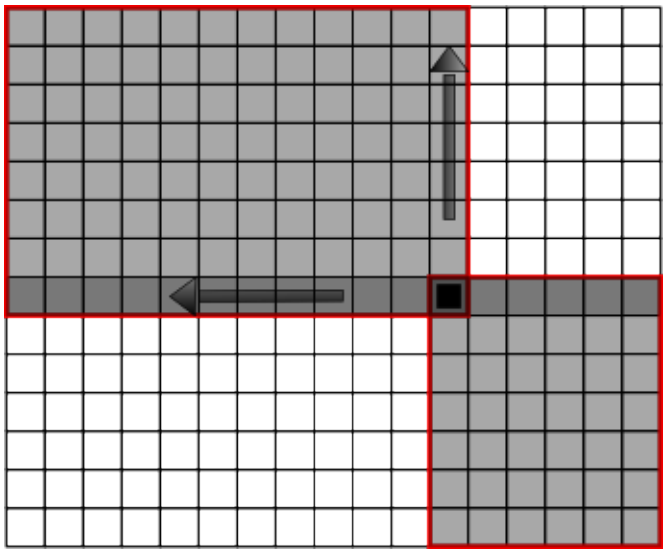
[4]Lako je primijetiti kako će optimalan put nužno počinjati i završavati u C-vrhu $(i,j)_C$ za neke i,j.

Slika 2.2: Svođenje problema globalnog poravnavanja na problem lokalnog poravnavanja.

Time smo problem lokalnog poravnavanja između sekvenci A i B sveli na problem globalnog poravnavanja između podsekvenci $A_{Q..X}$ i $B_{W..Y}$.

Globalno poravnavanje u linearnoj memoriji

Potrebno je pronaći najteži put koji započinje u $(0,0)_C$ i završava u $(M,N)_C$. Ako označimo $m := \lceil {}^{M}/_{2} \rceil$, lako je uočiti kako će takav optimalan put zasigurno prolaziti kroz barem jedan od vrhova oblika $(m,j)_C, (m,j)_I, (m,j)_D$ za neki $j \in \{0,1,...,M\}$.



Slika 2.3: Hirschbergovi rekurzivni poziv.

Potrebno je pronaći bilo koji od vrhova u m-tom retku kroz kojeg prolazi neki optimalan put. U tu svrhu ćemo za svaki vrh (m, j) izračunati najteži put koji završava u njemu, te najteži put koji započinje u njemu. Ukoliko tako dobivena dva broja posumiramo za svaki vrh, oni vrhovi kojima je suma maksimalna se zasigurno nalaze na nekom optimalnom putu. Tako dobiveni vrh u središnjem retku označimo s $(X, Y)_Z$, gdje su $i = \lfloor {}^{M}/_{2} \rfloor$, $Z \in \{C, I, D\}$.

Konačni algoritam dobivamo tako da rekurzivno pozovemo proceduru traženja najtežeg puta od $(0, 0)_C$ do $(X, Y)_Z$ te od $(X, Y)_Z$ do $(N, M)_C$. Ovdje je bitno primietiti da će svaki novi nivo rekurzije obrađivati dvostruko manje vrhova nego prethodna razina, stoga je ukupna vremenska složenost otprilike

$$\sum_{i=0}^{\infty} \frac{NM}{2^i} = O(NM)$$

dok je memorijska složenost očito $O(\min(N, M))$ jer nismo niti u jednom trenutku pamtili više od prethodnog retka matrica C, D, I.

2.1.3. Huang-Millerovo višestruko lokalno poravnavanje

Ako bi htjeli pomoću Myers-Millerovog algoritma pronaći K najboljih lokalnih poravnavanja (dakle riješiti problem višestrukog lokalnog poravnavanja koji je formalno definiran u uvodu), potrebno bi bilo pozvati prethodno objašnjeni algoritam točno K puta uz vremensku složenost $O(KMN)$. U praksi se pokazala potreba za rješenjem koje je skalabilnije za veće instance broja K.

Pomak u razvoju se dogodio 1991. kada su X. Huang i W. Miller u [7] predstavili algoritam koji pronalazi K najboljih lokalnih poravnavanja pomoću linearno memorije i u vremenskoj složenosti $O(MN + \sum_{i=1}^{k} |L_i|^2)$, gdje $|L_i|$ predstavlja duljinu i-tog najboljeg lokalnog poravnavanja. Negativne strane ovog algoritma jest da je relativno složen i kako ga nije jednostavno asimptotski analizirati do razine da je gornja vremenska složenost dobivena empirijski.

Potrebno je definirati rezidualne grafove poravnavanja. Neka je G_1 već dobro poznati graf poravnavanja $G_{A,B}$. Graf G_{n+1} za $n \in \{0, 1, ..., k-1\}$ definiramo tako da nađemo najteži put (optimalno lokalno poravnavanje) u grafu G_n te iz njega izbacimo (zabranimo) sve supstitucijske bridove u skladu s opisom zadatka.

Sada ćemo predstaviti pojednostavljeni i neformalni izgled osnovnog algoritma kojeg čemo postupno nadopunjavati detaljima:

1. Provedi Myers-Miller algoritam nad G_1 pamteći pritom pozicije[5] K najboljih puteva u grafu. Ukoliko dva takva puta počinju s istim vrhom, pamtimo samo bolji put. Dakle na kraju ćemo u strukturi imati K najboljih puteva koji počinju s različitim početnim vrhovima.

2. Iz strukture pročitaj poziciju najboljeg puta te potom taj podatak izbriši iz strukture.

3. Rekonstruiraj i ispiši najbolji pronađeni put pomoću Myers-Miller algoritma.

4. Zabrani sve supstitucijske parove upotrebljene u ispisanom poravnavanju.

5. Ponovno izračunaj one dijelove matrica $C, I, D, Prvi$ koji su se možda promijenili.

6. Uskladi strukturom s novoizračunatim podacima tako da sada sadrži $K - 1$ najtežih puteva koji počinju s različitim vrhom.

7. K puta ponovi korake 2-6.

Iz gornjeg pojednostavljenja vidljivo je da ćemo pri izračunu matrica C, I, D sada morati pamtiti dodatne podatke. Naime, za svaki element tih matrica pamtit ćemo i početni vrh u kojem počinje najteži put do tog elementa. Dodatno, ako postoji više mogućih početnih vrhova, pamtimo onaj koji je *topološki najmanji*. Tako dobivene tri matrice označimo s $Prvi_C(i,j), Prvi_I(i,j), Prvi_D(i,j)$. Izračun tih matrica ne povećava niti vremensku niti memorijsku O-složenost. Analogno s rezidualnim grafom poravnavanja definiramo i rezidualne matrice

$$C_n, I_n, D_n, Prvi_{C,n}, Prvi_{I,n}, Prvi_{D,n}$$

Glavna motivacija za izračun matrica $Prvi$ dana je u sljedećoj lemi:

Lemma 1. *Za neki $n \in \{1, 2, ..., k\}$ i neke C-vrhove u, v takve da najteži put u G_n završava u vrhu u[6] vrijedi da $Prvi_{C,n}(u) \neq Prvi_{C,n}(v) \implies C_n(v) = C_{n+1}(v) \wedge Prvi_{C,n}(v) = Prvi_{C,n+1}(v)$.*

Dokaz. Pokazat ćemo da najteži putevi koji završavaju u u odnosno v ne dijele niti jedan zajednički vrh, što će biti dovoljno da implicira istinitost ove leme. Pretpostavimo suprotno i recimo da najteži putevi koji završavaju u u, v dijele vrh w. U tome slučaju

[5] Pod „pozicijom“ smatramo poziciju početnog i krajnjeg vrha u putu.

[6] Dakle u n-tom koraku ćemo maknuti puti između $Prvi_{C,n}$ i u.

je nužno da najteži putevi $Prvi_{C,n}(u)..w$ i $Prvi_{C,n}(v)..w$ imaju istu težinu jer bi u suprotnom lakši dio puta mogli zamijeniti s onim težim, što je u kontradikciji s optimalnosti algoritma. No u ovom trenutku dolazimo do kontradikcije jer bi topološki manji vrh između $Prvi_{C,n}(u)$ i $Prvi_{C,n}(v)$ trebao biti početni vrh obijema putevima koji završavaju u u i v jer smo izjednačenja među početni vrhovima rješavali birajući topološki manjega. □

Glavni i najsloženiji korak kojeg treba dodatno pojasniti jest onaj u kojem ponovno izračunavamo dijelove matrice $C, I, D, Prvi$. Pretpostavimo da smo u nekom koraku ispisali i zabranili sve supstitucijske parove iz puta $F..u$ gdje su F, u neki C-vrhovi. Zbog leme 1, jedini podaci koji su se možda mogli promijeniti su oni kojima je vrijednost $Prvi = F$. No, prije nego što se upustimo u objašnjenje, potrebno je formalnije specificirati podatke koje pamti struktura.

Elementi strukture

Zamislimo da smo elemente matrice C podijelili u klase ekvivalencija s obzirom na vrijednost $Prvi_C$. Reprezentant takve klase je početni C-vrh F, dok cijenu definiramo kao najteži put koji započinje u F. Ukoliko uvedemo totalno poredak između klasa takav da je jedna klasa bolja od druge ako ima veću cijenu tada možemo s W_1 označimo cijenu $K.$ najbolje klase u G_1. Analogno tome, definiramo W_n kao cijenu $K + 1 - n.$ najbolje klase u G_n[7].

Svaki element strukture bit će sedmorka $S = (C, F, u, T, B, L, R)$, gdje:

C je cijena klase S.

F je početni C-vrh po kojem smo razdijelili elemente matrice C na klase ekvivalencije.

u je završni C-vrh najtežeg puta koji počinje u F. Ako ima više takvih, izaberemo bilo kojeg.

T, B, L, R su cijeli brojevi koji označavaju da svi elementi matrice C za koje vrijedi $C_n(x) > W_n$ i $Prvi(x) = F$ budu obuhvaćeni u pravokutniku dimenzija $[T, B] \times [L, R]$. Od svih mogućih pravokutnika izaberemo onaj najmanji.

Pojedinoj sedmorki S ćemo elemente označavati oznakama $S.C, S.F, ..., S.R$.

[7]Dakle u 1. koraku nas zanima $K.$ najbolja klasa, u 2. koraku $K - 1.$ najbolja, itd.

Ponovni izračun elemenata matrica

Kada u osnovnom algoritmu u $n.$ koraku zabranimo određene supstitucijske parove (4. korak u opisu algoritma) i maknemo najbolju klasu ekvivalencije iz strukture (2. korak), potrebno je uskladiti strukturu s novim podacima. Prije su u njoj bili sadržani podaci o najboljih $K + 1 - n$ klasa i svim vrhovima tih klasa za koje je $C_n(x) > W_n$. Nakon usklađivanja struktura bi trebala sadržavati sve podatke o $K - n$ klasa i svim vrhovima tih klasa za koje je $C_{n+1}(x) > W_{n+1}$.

Ukoliko označimo sedmorku najbolje klase u $n.$ koraku sa S, tada znamo da je dovoljno ponovno izračunati samo one vrijednosti u pravokutniku $[T, S.B] \times [L, S.R]$, gdje su T, L takve vrijednosti da nijedan put u G_n koji počinje izvan spomenutog pravokutnika i završava u pravokutniku $[S.T, S.B] \times [S.L, S.R]$ ima cijenu $\leq W_n$. Takav izbor T i L nam omogućuje da ponovno izračunamo elemente matrice zanemarujući sve što se nalazi na pozicijama „iznad ili lijevo“ od T, L i time čineći problem rekonstrukcije u linearnoj memoriji ekvivalentnim problemu lokalnog poravnavanja kojeg rješava Myers-Miller algoritam. Jedini problem je pronalazak vrijednosti T i L čime ćemo se baviti u ostatku odjeljka.

Sam postupak je detaljno opisan u [7] na 351. stranici i ovdje ga navodimo zbog potpunosti. Idejno, sama ideja pronalaska tih vrijednosti je „obrnuti“ postupak poravnavanja koji započinje u donjem-desnom vrhu i nastavlja tražiti prema gornjem-lijevom vrhu. Analogno tome, matrice koje postupak izračunava označit ćemo s C', I', D' i $Zadnji$. Kao zadnju napomenu prije nego što predstavimo sam algoritam uvedimo oznake $i(x), j(x)$ koje redom označavaju redak i stupac vrha x unutar matrice. Također, neka oznaka $x \geq y$ za neka dva vrha x, y označava $i(x) \geq i(y) \wedge j(x) \geq j(y)$. Slijedi pseudokod pomoću kojeg određujemo (T, L):

Funkcija na ulaz dobija granice S.T, S.B, S.L, S.R najbolje sedmorke ;

ulaz : (t, b, l, r)

izlaz: (T, L)

$T \leftarrow T' \leftarrow t$; $L \leftarrow L' \leftarrow l$; gotov $\leftarrow$ false ;

repeat

rf $\leftarrow$ cf $\leftarrow$ true ;

while $(rf \wedge T > 0) \vee (cfL > 0)$ **do**

if $rf \wedge T > 0$ **then**

$rf \leftarrow false$; $T \leftarrow T - 1$;

Izračunaj $C', D', I', Zadnji$ za $(T, j)_X$ iterirajući po

$j = r, r-1, ..., L$ i $X = \{C', D', I', Zadnji\}$;

if $Zadnji_X(T, j) \geq (T', L')$ *za neki* $j \in [L, r]$ *i* X **then**

rf $\leftarrow$ true ;

end

if $Zadnji_X(T, L) \geq (T', L')$ *za neki* X **then**

cf $\leftarrow$ true ;

end

end

if $cf \wedge L > 0$ **then**

$cf \leftarrow false$; $L \leftarrow L - 1$;

Izračunaj $C', D', I', Zadnji$ za $(i, L)_X$ iterirajući po

$i = b, b-1, ..., T$ i $X = \{C', D', I', Zadnji\}$;

if $Zadnji_X(i, L) \geq (T', L')$ *za neki* $i \in [T, b]$ *i* X **then**

cf $\leftarrow$ true ;

end

if $Zadnji_X(T, L) \geq (T', L')$ *za neki* X **then**

rf $\leftarrow$ true ;

end

end

end

gotov $\leftarrow$ true ;

foreach *S* $\in$ *struktura* **do**

if $S.T \leq b \wedge S.L \leq r \wedge S.B \geq t \wedge S.R \geq l \wedge (S.T < t' \vee S.L < l')$ **then**

$t' \leftarrow \min(t', S.T)$;

$l' \leftarrow \min(l', S.L)$;

gotov $\leftarrow$ false ;

end

end

until *gotov = true* ;

Idejno, procedura započinje s pravokutnikom $[S.T, S.B] \times [S.L, S.R]$ i proširuje ga prema gore i prema lijevo sve do ne može zaključiti da svi putevi koji počinju izvan njega i završavaju u $[S.T, S.B] \times [S.L, S.R]$ nemaju cijenu $\leq W_n$.

Napomenimo još kako je memorijska složenost ovog algoritma $O(N + M)$ budući da Myers-Miller algoritam treba $O(\min(M, N))$, dok pronalazak vrijednosti T, L ipak zahtjeva $O(M * N)$ memorijskih lokacija. Zbog tog razloga je memorijska složenost upravo $O(M * N)$.

U drugu ruku, vremensku složenost je vrlo komplicirano analizirati. U prvom koraku modificirani Myers-Millerov algoritam radi u složenosti $O(MN \cdot S)$, gdje je S složenost pojedinih operacija nad strukturom. Zasad pretpostavimo da je $S \approx O(1)$, a detaljnije ćemo dotaknuti tu problematiku u sljedećem odjeljku. Nadalje, izračunavanje vrijednosti T, L i rekalkulcija dijela matrice $[T, S.B] \times [L, S.R]$ zahtjeva $O((S.B - T) \cdot (S.R - L))$ operacija. Iako nedokazano, empirijski se pokazalo kako je ta vrijednost za tipične genomske sekvence proporcionalna s $|L_n|^2$, gdje je $|L_n|$ duljina n. pronađenog poravnavanja. Iz prethodno rečenog zaključujemo kako je ukupna vremenska složenost $O(MN + \sum_{i=1}^{K} |L_i|^2)$, što je mnogo bolje od $O(KMN)$, složenosti koju bismo dobili s K pokretanja osnovnog algoritma.

Implementacijske napomene

Posebnu pozornost pri implementaciji treba obratiti na nekoliko stvari:

Zabranjivanje iskorištenih supstitucijskih parova Kako se provjera koja utvrđuje da li je pojedini supstitucijski par iskorišten u najunutarnjijoj petlji modificiranog Myers-Miller algoritma, potrebno je organizirati o takvim parovima na efikasan način. Autorova implementacija Huang-Miller algoritma gradi za svaki redak vezanu listu iskorištenih parova koja omogućuje odgovor na dani upit u O(1) vremenskoj složenosti.

Izvedba strukture Također u najunutarnjijoj petlji modificiranog Myers-Miller algoritma su i operacije koje usklađuju strukturu s novonastalim podacima. Potrebno je podržati brze operacije zamjene najgore klase i pronalazak klase s određenim početnim vrhom. Iako su obje operacije ostvarive u $O(1)$ vremenskoj složenosti pomoću hasha, takvo rješenje bi imalo veliku konstantu i bi usporavalo sveukupno izvođenje algoritma. Autorova implementacija slijedi naputak iz [7] i pamti klase u nizu uz dva dodatna pokazivača: jedan na najgoru klasu u strukturi, te

drugi na zadnju pristupljenu klasu. Ukoliko je potrebno pronaći klasu koja nije zadnja pristupljena, obavlja se linearno pretraživanje strukture. Za očekivati je da će matrica $Prvi_C$ imati mnogo grupa jednakih uzastopnih elemenata, stoga ova metoda empirijski pokazuje O(1) vremensku složenost.

2.2. Paralelni algoritmi na GPU

Cilj ovog rada je istražiti paralelne analoge prije navedenih sekvencijalnih algoritama. Kao platformu za paralelne izvedbe odabrali smo Compute Unified Device Architecture (CUDA), standard tvrtke Nvidia za programiranje njihovih grafičkih kartica. Specifičnost sklopovlja grafičkih kartica predstavlja činjenica da one sadrže velik broj procesora[8] slabijih performansi koji omogućuju masivno paralelno izvođenje korisničkih programa. Dodatnu prednost predstavlja činjenica što su takve kartice, za razliku od ostalih paralelnih platformi, relativno jeftine i široko dostupne velikoj masi ljudi. [12] [13]

2.2.1. Paralelni Smith-Waterman algoritam

Kako bismo uopće proučili mogućnosti paralelne izvedbe Smith-Waterman algoritma zanemarimo za početak da radimo na CUDA platformi. Za svaku efikasnu implementaciju algoritama na stvarnom paralelnom sklopovlju pretpostavka je da je moguće izvesti efikasan algoritam na PRAM modelu računala, jednostavnom matematičkom modelu koji je paralelni analogon klasičnog RAM računala. Pritom se zanemaruju mnogi praktični problemi poput utjecaja cachea, ograničenosti radne memorije ili neskalabilnosti sinkronizacije među procesorima s povećanjem broja procesora.

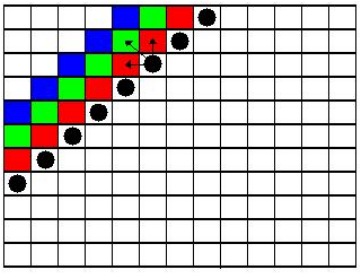
Jedini kriterij izvedbe na PRAM modelu je povoljni rezultat analize zavisnosti podataka. Dva podatka su nezavisna ukoliko se mogu simultano izračunati (svi ulazi potrebni za izračun podatka su već dostupni). Obično se zavisnost podataka proučava nad grafom zavisnosti koji je izomorfan s grafom prikazanim na slici 2.1, samo s obrnutim usmjerenjem bridova. U terminologiji grafova, kažemo da su dva vrha nezavisna ukoliko nije moguće naći put od jednog do drugog.

Teorija PRAM modela govori kako najdulji put u grafu zavisnosti odgovara najboljoj mogućoj vremenskoj složenosti algoritma kada bismo imali dostupni neograničeni broj procesora. Iz slike se može uočiti kako je duljina najdulje puta upravo $O(N+M)$, što je značajno poboljšanje od složenosti $O(MN)$ potrebne za sekvencijalni algoritam.

[8] Reda veličine 50-500 procesora po kartici.

S time rečenim, Brentov teorem [14] nam govori kako je na PRAM modelu moguće izvesti Smith-Waterman algoritam sa složenosti $O(M + N + {}^{MN}/_{p})$, gdje je p broj dostupnih procesora. Napomenimo kako gornja analiza također implicira kako bolje asimptotske složenosti nije niti moguće dobiti.

Radi ilustracije i pojašnjenja dat ćemo i eksplicitniji opis paralelnog algoritma za PRAM model. U terminologiji nezavisnosti podataka, uočavamo kako je svaka sporedna dijagonala matrica C, I, D nezavisna i moguće ju je izračunati u paraleli. Slika 2.4 ilustrira taj pristup na način da polja označena istom bojom mogu biti istovremeno izračunata. Uočavamo da sporednih dijagonala ima upravo O(M+N), što se podudara s brojem sekvencijalnih koraka koje će paralelni algoritam izvršiti. Algoritam u svakom pojedinom koraku u paraleli izračunava sve vrijednosti matrica na jednoj sporednoj dijagonali.



Slika 2.4: Nezavisnost podataka u matricama C, I, D.

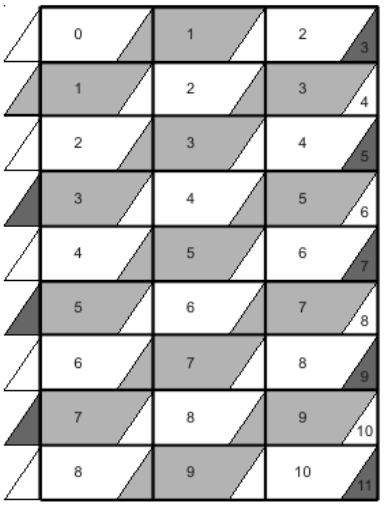
Izvedba na CUDA platformi

Gore opisani algoritam nije pogodan za izvedbu na CUDA arhitekturi zbog nepovoljnog poretka pristupa memoriji. Globalna memorija ima vrijeme pristupa i po nekoliko stotina ciklusa, stoga se pristupi globalnoj memoriji iz različitih procesora moraju međusobno spojiti u jedan unificirani pristup. To će se dogoditi kada procesori pristupaju uzastopnim memorijskim lokacijama. Također, ukoilko je količina memorije kojima pristupa jedna grupa procesora[9] dovoljno mala, poželjno je u početnom koraku algoritma prebaciti sve potrebne podatke iz spore globalne memorije u manju, ali mnogo bržu, dijeljenu memoriju koju dijeli upravo ta grupa procesora.

Iz gore navedenih razloga izvedba paralelnog Smith-Waterman algoritma koji u svakom koraku izračunava jednu dijagonalu podataka nije pogodna. Rješenje toga

[9]U CUDA terminologiji, radi se o grupama streaming procesora koji se nalaze na streaming multiprocesoru.

problema dali su 2010. Sandes i de Melo u [15]. Oni su grupirali elemente matrica C, I, D u blokove kao na slici 2.5 tako da jedna grupa procesora (tj. dretvi) u jednom koraku rješava jedan blok. Razlog za „kosom“ lijevom i desnom granicom je upravo gore navedena podatkovna nezavisnost. Dretve su organizirane tako da pojedina dretva u jednom koraku obrađuje jedan ili više redaka jednog bloka. Blokovi označeni istim brojem se izračunavaju u istom koraku.



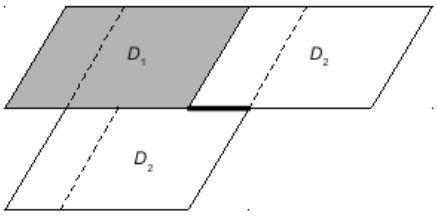
Slika 2.5: Grupiranje podataka matrica C, I, D u blokove, preuzeto iz [4].

Navedimo još kako se upravo zbog navedene geometrijske organizacije blokova na prikazani način javlja novi hazard koji je ilustriran na slici 2.6. Kako se idejno blokovi označeni bijelom bojom izračunavaju istovremeno i neovisno jedno o drugome[10], moguće je da podaci koji su označeni u slici podebljanom crtom nisu dostupni u trenutku kada ih donji D_2 blok zatreba. Iz toga razloga izračun blokova dijeli se u „kratku“ i „dugu“ fazu na način da se u kratkoj izračunavaju oni elementi koji su potrebni za sprječavanje hazarda, dok sve ostalo pripada dugoj fazi. Detalje oko same implementacije je moguće naći u [15] i [4].

2.2.2. Paralelni Myers-Miller algoritam

Prilikom proučavanja prethodno opisanog Smith-Waterman algoritma treba napomenuti kako se radi o metodi za strogi izračun elemenata matrica C, I, D. Ukoliko imamo na raspolaganju $O(MN)$ memorije možemo izračunate matrice pohraniti i time omogućiti

[10]Drugim riječima, mogu se izračunavati potpuno neproporcionalnim brzinama. Zbog ograničenosti sklopovlja Moguće je da se jedan blok u potpunosti izvršio, dok drugi još nije niti započeo.



Slika 2.6: Memorijski hazard, preuzeto iz [4].

izravnu rekonstrukciju rješenja. No, češći je slučaj kada si ne možemo priuštiti memorijsku složenost veću od $O(M + N)$, a u tome slučaju nam gornji algoritam sam po sebi uopće ne pomaže u pogledu rekonstrukcije rješenja.

Ipak, Korpar je 2011. u [4] pokazao kako je, analogno sekvencijalnoj metodi, moguće iskoristiti Hirschbergovu optimizaciju za rekonstrukciju optimalnog poravnavanja uz korištenje CUDA tehnologije.

Ideja je sljedeća - potrebno je koristiti paralelnu izvedbu Smith-Waterman algoritma u obje faze sekvencijalnog Myers-Miller algoritma: tijekom svođenja problema lokalnog poravnavanja na problem globalnog poravnavanja i tijekom rekurzivne rekonstrukcije globalnog poravnavanja u linearnoj memoriji.

Paralelno svođenje lokalnog na globalno poravnavanje Algoritam svođenja se sastoji u izračunavanju svih vrijednosti matrice C i pamćenju najvećeg viđenog elementa cijele matrice. Takav algoritam je ostvariv komponirajući algoritam opisan u prethodnom odjeljku zajedno s opće poznatim algoritmima redukcije, poput onog opisanog u [13].

Rekurzivna rekonstrukcija Operacija koju je potrebno podržati u ovom koraku jest pronalazak najvećeg elementa u središnjem retku matrica C, I, D, što je opet moguće izvesti komponirajući paralelnu izvedbu Smith-Waterman algoritma s paralelnom redukcijom. Bitno je napomenuti kako će zbog rekurzivnih poziva samog algoritma biti potrebno izračunavati dijelove matrica C, I, D s varijabilnim veličinama. Za efikasnu izvedbu algoritma potrebno je pronaći pragove ispod kojih nije isplativo prebacivati izračunavanje tih dijelova matrica na grafičku karticu, već je potrebno pozvati sekvencijalnu verziju istih algoritama. Korpar u [4] koristi genetsku metaheuristiku kako bi pronašao navedene pragove.

2.2.3. Paralelni Huang-Miller algoritam

U pokušaju paralelizacije Huang-Miller algoritma javlja se nekoliko problema. Kako bismo ušli u problematiku prisutnu s tom paralelizacijom prvo ćemo opisati probleme koji se javljaju, kao i njihova moguća rješenja, a potom ćemo izvesti općenite zaključke vezane za paralelizaciju spomenutog algoritma.

Podatkovna zavisnost prilikom određivanja vrijednosti T, L

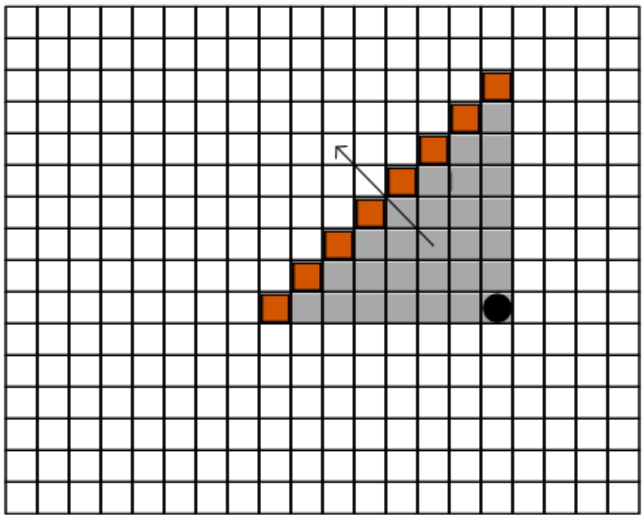
Problem: Sekvencijalni algoritam za izračunavanje vrijednosti T i L je ugrubo imao sljedeći oblik: započinjemo s pravokutnikom $[t, b] \times [l, r]$ i proširujemo ga prema gore i prema lijevo brzinom 1 element po koraku sve dok ne dođemo do situacije u kojoj možemo zaključiti da proširivanje više nije potrebno. Prilikom svakog proširenja, primjerice prema gore, potrebno je nanovo izračunati dio jednog retka matrica C', I', D' i $Zadnji$. Iako to ne stvara problem u sekvencijalnoj izvedbi, navedeni algoritam generira potpunu podatkovu zavisnost[11] unutar elemenata matrice koje je potrebno izračunati.

Analiza mogućih rješenja: Ovaj problem je moguće riješiti na dva načina: jednostavniji od njih uključuje inkrementalno smanjivanje vrijednosti T i L u koracima većim od 1. Dana metoda omogućuje razbijanje podatkovne zavisnosti i omogućuje izračunavanje dijelova matrice C postupkom analognim paralelnom Smith-Waterman algoritmu. Ona nije optimalna jer će dio matrice koje je potrebno ponovno izračunati biti mnogo veći nego je to originalno bilo potrebno. Autor je empirijski došao do zaključka da za veličinu koraka proporcionalnu veličini sekvence (primjerice, $n/100$), metoda se s porastom veličina nizova vremenski počinje ponašati kao K poziva paralelnog Myers-Miller algoritma.

Alternativno rješenje danog problema je promjena geometrije dijela matrice koji će biti potrebno ponovno izračunati. Umjesto pravokutnika, zanimat će nas pravokutni trokut kojemu su donji i desni brid paralelni osima matrica, dok mu je preostali brid paralelan sa sporednim dijagonalama. Sada u svakom koraku proširujemo trokut tako da pomaknemo dijagonalni brid jedan korak bliže ishodištu kao što je to ilustrirano na slici 2.7.

Primijetimo da su sada podaci koje je potrebno izračunati u sljedećem koraku (na slici označeni narančastom bojom) međusobno podatkovno nezavisni, stoga ih je moguće izračunati u punoj paralelizaciji.

[11] Kao što je to vidljivo iz slike 2.1, svi elementi jednog retka su međusobno podatkovno zavisni.



Slika 2.7: Podatkovno nezavisni algoritam traženja regije za rekalkulaciju.

Tablica zabranjenih supstitucijskih parova

Problem: U svakom koraku glavnog algoritma pronađeno je jedno poravnavanje podsekvenci i od toga trenutka nadalje svi upotrebljeni supstitucijski parovi u tom poravnavanju su zabranjeni. U tu svrhu je potrebno osmisliti efikasnu strukturu pohrane zabranjenih parova koja će moći paralelno posluživati sve dretve.

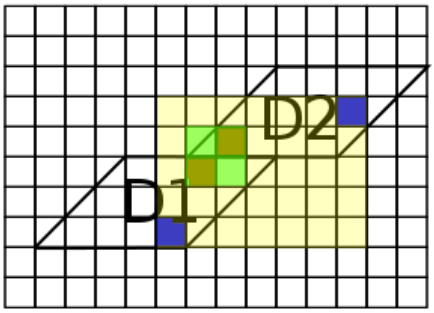
Analiza mogućih rješenja: Za početak primijetimo kako paralelna implementacija Smith-Waterman algoritma svakoj pojedinoj dretvi u jednom koraku daje ekskluzivno korištenje jednog ili više redaka. To svojstvo nam omogućuje da zabranjene parove pohranimo u stupčano sortiranu vezanu listu za svaki redak. Svaka dretva će pritom neovisno o drugima iterirati po zabranjenim parovima za svoj redak bez ikakve potrebe za sinkronizacijskim mehanizmima. Jedina negativna strana ovog pristupa je što generira divergenciju dretvi, vrlo nepoželjan efekt pri programiranju na CUDA platformi.

Sinkronizacija strukture K najboljih klasa

Problem: Vjerojatno najozbiljniji problem koji se javlja pri pokušaju paralelizacije Huang-Miller algoritma je potreba za konstantnom globalnom sinkronizacijom strukture koja čuva K najboljih klasa u svakom koraku početnog popunjavanja strukture. Originalna složenost popunjavanja strukture je O(MN) i time čini najunutarnjiju petlju cijelog programa. Svaka promjena na ovom dijelu koda direktno utječe na cjelokupnu efikasnost programa. Nažalost, direktna implementacija predviđenog algoritma (uz

sinkronizacijski mehanizam ulaska u monitor[12]) se u potpunosti serijalizira i kao rezultat tvori potpunu međusobnu zavisnost podataka.

Analiza mogućih rješenja: Autor nije uspio naći zadovoljavajuće rješenje ovog problema. Potencijalno rješenje bi bilo izgraditi strukturu za svaki blok podataka i potom izgraditi paralelnu metodu redukcije tih struktura. Time problem nije uopće riješen, jer se sada potpuna zavisnost podataka s cijele matrice preselila na potpunu zavisnost podataka nad cijelim blokom, koji će se opet morati serijalizirati (iako po prihvatljivijoj cijeni jer se radi o ulasku u monitor na razini bloka). Dodatne strana toga rješenja koje treba uzeti u obzir je neizvedivost potpuno točnog redukcijskog algoritma koji bi više takvih struktura reducirao u jednu. Pretpostavimo da svaki blok posjeduje svoju kopiju strukture, svoju kopiju vrijednosti W_n i da je pritom $K = 2$. Neka blokovi D_1 i D_2 kao na slici 2.8 sadrže u svojim strukturama klasu s po dva elementa koji redom imaju ocjenu sličnosti 10 (plava boja) i 100 (crvena boja). Ukoliko za reduciranu klasu vrijedi $W_n = 50$, u njoj se trebaju nalaziti samo oni crveni elementi koji su sadržani unutar zelenog kvadrata. No u drugu ruku mi iz podataka koje smo zapamtili u strukturi najbolje što možemo reći o crvenim kvadratima jest da se nalaze negdje unutar žutog pravokutnika. Ova nepreciznost će voditi do ogromnih pravokutnika $[S.T, S.B] \times [S.L, S.R]$ i rezultirat će s $O(kNM)$ algoritmom, tim više što su elementi strukture međusobno posloženi dijagonalno.



Slika 2.8: Neizvedivost redukcije struktura.

Izvedivost izvedbe Huang-Miller algoritma na CUDA platformi

Zbog prethodno navedenih razloga autor smatra da se Huang-Miller algoritam ne može efikasno paralelizirati na CUDA grafičkim karticama. Od navedena tri problema vrijedi da prva dva samo znatno otežavaju i usporavaju eventualnu implementaciju, dok

[12]engl. mutex

zadnji navedeni problem čini potencijalnu implementaciju potpuno neisplativom dok se ne pronađe alternativni način zaobilaska podatkovnih zavisnosti.

3. Pregled i usporedba danih metoda

Praktična primjenjivost paralelnih verzija algoritama za poravnavanje sekvenci najbolje je ilustrirana u [16] gdje se navodi kako je ubrzanje na relativno staroj grafičkoj kartici otprilike 30 puta od strogo sekvencijalnog rješenja. Slično tome, Korpar u [4] na novijoj kartici navodi ubrzanja s faktorom reda veličine 400.

Kako je naglasak ovog rada ipak na najboljem mogućem algoritmu za problem višestrukog poravnavanja sekvenci, u sljedećim tablicama ćemo usporediti vremena izvršavanja najboljih dostupnih algoritama koji rješavaju taj problem za različite vrijednosti broja K. U donjima tablicama moguće je vidjeti vremena izvršavanja diskutiranih algoritama na proteinskim sekvencama duljina 35213×36805 (tablica 3.1).

Testirani algoritmi su redom:

- Sekvencijalni Smith-Waterman algoritam koji se pokreće K puta.
- Sekvencijalni Huang-Miller algoritam koji pronalazi K najboljih poravnavanja sekvenci.
- Paralelni Smith-Waterman algoritam koji se pokreće K puta.

K	$K\cdot$ seq-SW	HM	$K\cdot$ cuda-SW
10	431s	972s	4s
50	2440s	3582s	21s
100	4743s	5919s	37s
1000	61286s	26696s	455s

Slika 3.1: Vremena izvršavanja za proteinske sekvence duljine 32503×36805

Bitno je napomenuti kako svi ti algoritmi imaju linearnu memorijsku složenost.

4. Zaključak

Iz prethodno dobivenih rezultata se jasno naslućuje kako je za problem višestrukog poravnavanja sekvenci najbolji dostupni algoritam za sve praktične primjene upravo paralelni Smith-Waterman koji se pokreće K puta i pritom u svakoj iteraciji otkriva sljedeće najbolje poravnavanje.

Razlog tome leži u pretpostavki da paralelna izvedba Huang-Miller algoritma nije moguća zbog intrinzičkih podatkovnih zavisnosti koji onemogućuje isplativost implementacija na CUDA platformi. Ukoliko bi poboljšana modifikacija navedenog algoritma sa svojstvom da ju je lakše paralelizirati bila razvijena, najvjerojatnije bi takav algoritam imao prednost nad diskutiranim metodama.

Doduše, zanimljivo je primijetiti kako je Huang-Miller algoritam skalabilniji s povećanjem broja K. Dok se ostali algoritmi s povećanjem K ponašaju linearno uz goru konstantu, konstanta u implementaciji Huang-Miller algoritma se smanjuje. Doduše, na današnjim računalima algoritam postaje isplativ nad paralelnim Smith-Watermanom tek za nepraktično velike sekvence[1].

[1] Ova tvrdnja je dobivena iz interpolacije podataka i nije eksperimentalno potvrđena.

LITERATURA

[1] D.A. Benson, I. Karsch-Mizrachi, D.J. Lipman, J. Ostell, i D.L. Wheeler. Genbank: update. *Nucleic acids research*, 32(Database issue):D23, 2004.

[2] N.M. Luscombe, D. Greenbaum, M. Gerstein, et al. What is bioinformatics? a proposed definition and overview of the field. *Methods of information in medicine*, 40(4):346–358, 2001.

[3] M. Feig, A. Onufriev, M.S. Lee, W. Im, D.A. Case, i C.L. Brooks III. Performance comparison of generalized born and poisson methods in the calculation of electrostatic solvation energies for protein structures. *Journal of computational chemistry*, 25(2):265–284, 2004.

[4] M. Korpar. Implementacija smith waterman algoritma koristeći grafičke kartice s cuda arhitekturom. *Fakultet elektrotehnike i računarstva, završni rad*, 2011.

[5] G. Žužić. Usporedba dna sekvenci: Smith-waterman algoritam. *Fakultet elektrotehnike i računarstva, seminarski rad*, 2011.

[6] I. Čanadi. Algoritmi za poravnavanje dviju sekvenci. *Fakultet elektrotehnike i računarstva, seminarski rad*, 2009.

[7] X. Huang i W. Miller. A time-efficient, linear-space local similarity algorithm. *Advances in Applied Mathematics*, 12(3):337–357, 1991.

[8] S.B. Needleman, C.D. Wunsch, et al. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[9] T. Smith i M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[10] D.S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[11] E.W. Myers i W. Miller. Optimal alignments in linear space. *Computer applications in the biosciences: CABIOS*, 4(1):11–17, 1988.

[12] D. Kirk, W.H. Wen-mei, i W. Hwu. *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2010.

[13] C. Nvidia. Nvidia cuda programming guide, 2011.

[14] S. Chatterjee i Prins J. Parallel computing pram algorithms, 2009. URL `http://www.cs.unc.edu/~prins/Classes/633/Handouts/pram.pdf`.

[15] E.F.O. Sandes i A.C. de Melo. Cudalign: using gpu to accelerate the comparison of megabase genomic sequences. U *ACM Sigplan Notices*, svezak 45, stranice 137–146. ACM, 2010.

[16] Y. Munekawa, F. Ino, i K. Hagihara. Design and implementation of the smith-waterman algorithm on the cuda-compatible gpu. U *BioInformatics and BioEngineering, 2008. BIBE 2008. 8th IEEE International Conference on*, stranice 1–6. IEEE, 2008.

GPU implementacija vremenski učinkovitog algoritma za lokalno poravnavanje s linearnom memorijskom složenosti

Sažetak

Višestruko poravnavanje sekvenci jedan je od najvažnijih problema u bioinformatici za koje nije razvijeno efikasno egzaktno rješenje. Cilj rada je pronaći najisplativiji algoritam koji egzaktno i u linearnoj memoriji rješava dani problem. Istraženi će biti sekvencijalne i paralelne verzije Smith-Watermanovog, Myers-Millerovog i Huang-Millerovog algoritma te se pokazuje kako zadnji od navedenih nije pogodan za paralelizaciju. Rad je zaokružen pregledom i usporedbom svih dostupnih metoda.

Ključne riječi: bioinformatika, višestruko poravnavanje sekvenci, Smith-Waterman, Myers-Miller, Huang-Miller, linearna memorija, paralelizacija, CUDA.

GPU implementation of an efficient linear-memory algorithm for local sequence alignment

Abstract

Multiple sequence alignment is one of the most important bioinformatics problems that doesn't have an efficient exact solution. The goal of this article is to find the most efficient algorithm that will solve the problem in linear memory. The focus will be put on sequential and parallel variants of the popular Smith-Waterman, Myers-Miller and Huang-Miller sequence alignment algorithms implemented. It is also shown that the latter one is not suitable for parallelization. The article is concluded with a summary and comparison of the most efficient available methods.

Keywords: bioinformatics, multiple sequence alignment, Smith-Waterman, Myers-Miller, Huang-Miller, linear memory, parallelization, CUDA.