SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

BIOINFORMATIKA - PROJEKT

SW# - LOKALNO PORAVNANJE GENOMA NA GPU (*GPU-ENABLED EXACT ALIGNMENTS ON GENOME SCALE, Korpar, Šikić, 2013*)

Mario Halambek

Tomislav Tunković

Kristijan Bartol

Zagreb, Siječanj, 2018.

Sadržaj

1. Uvod

"Poravnanje bioloških sljedova je najčešći prvi korak u bioinformatičkog analizi – bilo da je u pitanju pronalazak evolucijski očuvanih regija među vrstama, analiza genetske bolesti ili kreiranje rodovskog stabla. Upravo zbog svoje široke primjene, poravnanje dva biološka slijeda predstavlja jedan od najstarijih i najviše istraživanih problema u bioinformatici." [1]

Osnovni korak pri poravnanju bioloških sljedova je definiranje mjere udaljenosti dvaju sljedova. Osnovni primjer mjere udaljenosti je Hammingova udaljenost koja je jednaka broju različitih znakova. Ipak, ona nije definirana za nizove različite duljine. Umjesto nje češće se koristi neka varijanta mjere koju je predložio Levenshtein [2] (1965.) gdje je udaljenost dvaju nizova *s* i *t* jednaka minimalnom broju potrebnih modifikacija nad jednim znakom potrebnih za pretvaranje niza *s* u niz *t.* Ta udaljenost se naziva **udaljenost uređivanja** (engl. *edit distance*) s tri dozvoljene operacije kod udaljenosti uređivanja: zamjenom, umetanjem i brisanjem.

1.1. Udaljenost uređivanja – globalno poravnanje

Levenshtein nije predložio algoritam nalaženja poravnanja dvaju nizova. On je smišljen kasnije, a njegove se modifikacije s boljom vremenskom, odnosno, prostornom složenošću i dalje pronalaze. U osnovnoj varijanti se, ukratko, gradi tablica poravnanja općenite veličine *(n + k) x (m + l)*, gdje je *n* dužina niza *s*, a *m* dužina niza *t,* a *k* i *l* su brojevi praznina (operacija brisanja, odnosno, umetanja) koji se ubacuju u nizove i konačna vrijednost postupka je ona zapisana u elementu donjeg desnog kuta *(m + k, n + l).* Kreće se od elementa *(0, 0)* u gornjem lijevom kutu koji ima vrijednost *0.* Vrijednost svakog drugog elementa određuje se kao maksimalna vrijednost iz lijevog, gornjeg i gornjeg lijevog elementa, ako takvi postoje. Pseudokod osnovnog algoritma (bez rekonstrukcije poravnanja) dan je u nastavku:

```
EDIT_DISTANCE(s, t)
    for i <- 1 to |s| do
        V[i,0] = d*i
    end
    for j <- to |t|
        V[0,j] = d*j
    end
    for i <- 1 to |s| do
        for j <- 1 to |t| do
            MATCH = V[i-1,j-1] + w(s[i],t[j])
            INSERTION = V[i,j-1] + d
```

```
                DELETION = V[i-1,j] + d
                V[i,j] = min(MATCH, INSERTION, DELETION)
        end
end
```

1.2. Lokalno poravnanje

Algoritam opisan u prošlom odjeljku je algoritam globalnog poravnanja. To znači da tražimo udaljenost, odnosno, preklapanje čitavih nizova pa očekujemo da će nizovi s kojima radimo biti praktički jednake dužine i vrlo sličnog sadržaja. Međutim, danas ćemo najčešće tražiti preklapanje dvaju nizova od kojih je jedan znatno kraći od drugog (slika 1.1.c), npr. ako u genomu čovjeka (~6x10^9) želimo pronaći genetski zapis identičan ili sličan genetskom zapisu neke bakterije (~1x10^4).

Lokalno poravnanje izvodimo tako da zanemarujemo praznine na početku i na kraju (kraćeg) niza; jednostavno vrijednosti tih elemenata ne čuvamo u matrici preklapanja. U međuvremeno se pokazalo [3] da vrijednost 1 za sve nekompatibilnosti dvaju nizova (zamjena, dodavanje, brisanje) nije najpogodnija pa se od tada koriste negativne vrijednosti u tim slučajevima. Za preklapanja koristi se pozitivna vrijednost. Problem se tako pretvara u problem maksimizacije i u matrici preklapanja traži se maksimalna (nenegativna) vrijednost, tj. to više nije nužno donji desni element *(n, m)* (jer se traži regija u kojoj se kraći niz maksimalno preklapa s dužim, a ona nije nužno na kraju niza).

Kad nađemo maksimalnu vrijednost u matrici, od nje tražimo optimalno poravnanje praćenjem unatrag. Pokazuje se i kako u ovom dijelu nema potrebe u matricu upisivati negativne vrijednosti [1], nego se one mogu zamijeniti nulama.

1.3. Smith-Watermanov algoritam

Sve pretpostavke i postupak opisan u prošlom odjeljku zapravo čine Smith-Watermanov algoritam. Pseudokod osnovnog Smith-Watermana dan je u nastavku:

```
SMITH_WATERMAN(s, t)
        M = 0
        V[0,0] = 0
        for i <- 1 to |s| do
                V[i,0] = 0
        end
        for j <- to |t|
                V[0,j] = 0
        end
```

```
        for i <- 1 to |s| do
            for j <- 1 to |t| do
                MATCH = V[i-1,j-1] + w(s[i],t[j])
                INSERTION = V[i,j-1] + d
                DELETION = V[i-1,j] + d
                V[i,j] = min(MATCH, INSERTION, DELETION)
        end
        M = max(M, V[i,j])
    end
```

Razlike koje se mogu primijetiti u odnosu na pseudokod određivanja udaljenosti uređivanja su izvan glavne, dvostruke *for* petlje, a to je da elemente prvog retka i prvog stupca postavljamo na nulu, te na kraju tražimo nenegativnu vrijednost *M* koja je ili maksimalna pozitivna vrijednost matrice, ili, ako takva ne postoji, *0.*

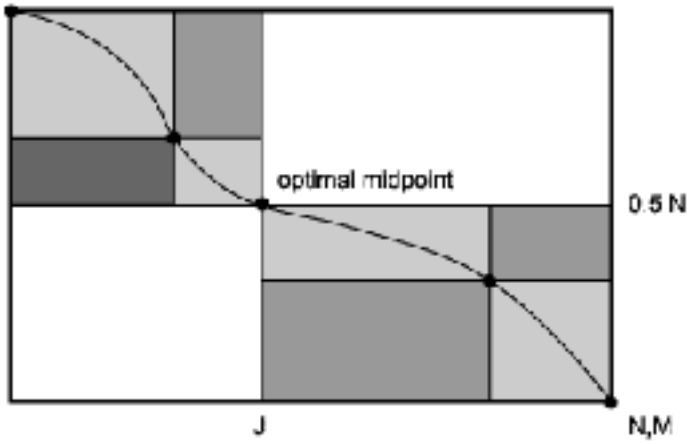
U slučaju da radimo i rekonstrukciju preklapanja, dodatno pamtimo i informaciju o smjeru iz kojeg smo došli.

2. Prostorna optimizacija Smith-Watermana (ideja)

Da bismo dobili lokalno poravnanje dvaju nizova duljine *m* i *n* Smith-Watermanom, moramo izračunati sve elemente matrice dimenzija *(m+1) x (n+1)* te spremiti sve informacije o smjerovima pa je vremenska i prostorna složenost osnovnog algoritma kvadratna - *O(n*m)* . Krenimo od smanjenja prostorne složenosti koja se može riješiti „čistim" algoritamskim postupkom bez posezanja za heuristikom.

2.1. Mayers-Millerov algoritam

Smanjenje prostorne složenosti temelji se na dvije obzervacije. Prva je da je poravnanje reverznih nizova s^r i t^r jednako poravnanju originalnih nizova *s* i *r.* Druga obzervacija je da je izračun svakog elementa matrice poravnanja potrebno znati samo vrijednosti u trenutnom i prethodnom retku. Dodatno, za rješavanje cijelog retka potrebno je znati samo vrijednosti u prethodnom retku i vrijednost rubnog uvjeta početnog stupca.



Slika 2.1. Jednostavni prikaz Mayers-Millerovog algoritma poravnanja [4]

Originalni Mayers-Millerov algoritam traži optimalni *k* u srednjem retku matrice, točnije, pronalazi 2 susjedna elementa posljednjih redaka pri poravnanju gornje i donje polovice za koji je suma dvaju odabranih elemenata maksimalna. Taj se postupak dalje nastavlja za srednje retke u svakoj od polovica matrica pa je veličina novih matrica u tom idućem korak približno 4 puta manja. Rekurzija se nastavlja sve dok veličine matrica ne budu manje od 2x2. U tom trenutku izračunali smo sve zasivljene elemente sa *slike 2.1.*, a rekonstrukciju staze kojom smo kreirali poravnanje sljedova provodimo u dodatnom koraku u kojem moramo pronaći krajnje točke tražene staze (2 točke s oba kraja matrice, tzv. *startpointovi*).

Startpoint se nalazi u trećoj fazi postupka – rekonstrukciji. Taj element ima vrijednost poravnanja jednaku 0 i prvi je takav na putu od sredina prema krajevima. Kad nađemo obje ključne točke gornje i donje polovice, povezujemo taj put u konačno rješenje postupka – znamo maksimalnu vrijednost poravnanja i stazu.

3. Implementacija prostorne i vremenske optimizacije

U našoj implementaciji čitav postupak praktički izvodi jedna metoda (sve 3 faze). Glavne dijelove metode čine dvije skupine rekurzija. Jedna koja eliminira polovicu ukoliko sadrži poravnanje maksimalne vrijednosti 0 (verzija postupka rekonstrukcije opisan u prošlom poglavlju), a druga koja izvodi algoritam Mayers-Millera na preostalom dijelu matrice. Praznine s početka i kraja niza se zanemaruju s nekoliko *if* blokova za različite rubne slučajeve.

3.1. Nalaženje maksimalne vrijednosti – Smith-Waterman

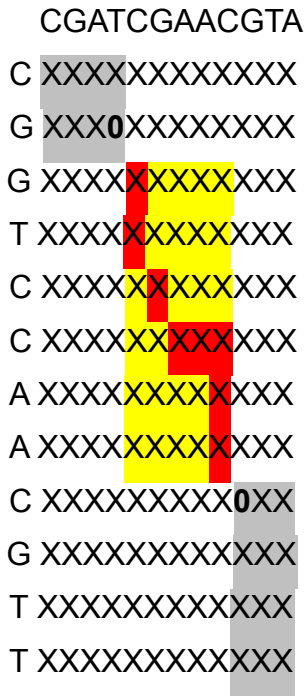
Kao i u originalnoj implementaciji [5], matrica se prepolavlja na dva dijela. Postupak na najvišoj razini izvodi jedna jezgra sekvencijalno, a računanje svake od polovica izvodi se paralelno za blokove veće od 32x32, što je ukratko opisano u idućem odjeljku. U svakom retku pamti se maksimalna vrijednost koja, ako postane veća od ukupne maksimalne vrijednost te polovice, postaje nova maksimalna vrijednost polovice i pamtimo njene koordinate. U trenutku kad obje polovice izvrše svoje računanje, imamo podatak o maksimalnoj vrijednosti obiju polovica i njihove koordinate. Postupak se u pseudokodu može prikazati na sljedeći način:

```
ALIGN(iRow0, iRow1, iCol0, iCol1)
    for iRow <- iRow0 to iRow1
        localRow[0] = row[0]-sc.k // take from last row
        for c <- iCol0 + 1 to iCol1
            r = max(localRow[c-1], row[c], row[c-1])
            localRow[c] = r
            if r > maxVal
                maxVal = r
                iRowMax = iRow
                iRowCol = c
        row = localRow
    return maxVal
```

Iz pseudokoda je vidljivo da je u svakom novom retku potrebno čuvati samo prethodni redak. Početna vrijednost novog retka je početna vrijednost prethodnog retka minus cijena praznine. U slučaju da je trenutna vrijednost poravnanja *r* postala veća od najveće vrijednosti poravnanja trenutne razine, *r* postaje nova najveća vrijednost poravnanja. Iz pseudokoda su izostavljeni detalji oko provjere je li niz praznina već otvoren ranije i detalji oko smjera iteriranja s obzirom na polovicu u kojoj trenutno računamo.

3.2. Prva razina rekurzije

Sad kad smo dobili maksimalne vrijednosti obiju polovica iterativnim prolaskom kroz svaku, možemo provjeriti te vrijednosti i ako je neka od njih 0, slobodni smo odbaciti tu polovicu jer na toj strani se sigurno ne nalazi optimalno poravnanje. Odluku možemo donijeti unutar jednog *if* bloka na način da opet pozovemo glavnu metodu za svaku polovicu.



Slika 3.1. Pronađeni start_pointovi (0) i staza optimalnog poravnanja (crveno)

Lako je primijetiti da će nas ova rekurzija u jednom trenutku dovesti do eliminacija određenih polovica na određenoj dubini (slika 3.1), osim u slučaju kad se optimalno poravnanje proteže oko čitave glavne dijagonale matrice, od gornjeg lijevog kuta do samog donjeg desnog. Ovim postupkom implicitno pronalazimo *starting_point*-ove gornje i donje polovice ekvivalentne originalnom algoritmu. Važno je primijetiti da će ova rekurzija odbaciti krajeve matrice, ne i srednji dio, inače bi imali nepovezanu stazu.

```
ALIGN_REC(iRow0, iRow1, iCol0, iCol1)
...
// odbaciti zadnji dio matrice?
if bestFwd.val >= valMax && bestFwd.val >= bestBak.val
        alignRec(iRow0, bestFwd.p.x-1, iCol0, bestFwd.p.y-1)
        sol.matches.push_back(bestFwd.p)
```

```
        return

// odbaciti prvi dio matrice?
if bestBak.val >= valMax && bestBak.val >= bestFwd.val
        alignRec(bestBak.p.x+1, iRow1, bestBak.p.y+1, iCol1)
        sol.matches.push_back(bestBak.p)
        return
```

Eksplicitno su navedeni dijelovi za ispitivanje odbacivanja oba dijela matrice. Iz pseudokoda je vidljivo da se u slučaju da ne zadovoljimo uvjete *if* bloka, ne granamo dublje što rezultira navedenim opisanim odbacivanjem donjeg, odnosno, gornjeg dijela matrice trenutne razine. Specifičnost je implementacije to što se pri grananju matricu zapravo ne prepolavlja na središnjem retku, nego je kao rubni redak i rubni stupac odabran upravo onaj u kojem smo pronašli trenutno maksimalnu vrijednost poravnanja.

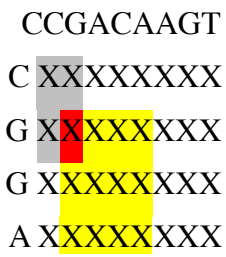
3.3. Druga razina rekurzije

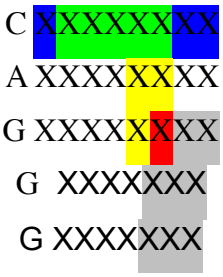
Dovršenje rekonstrukcije događa se na drugoj razini rekurzije. Ovdje se u biti implementira Mayers-Millerov algoritam. Ako znamo da smo prethodnim rekurzijama eventualno odbacili samo krajnje dijelove matrice, u ovom se dijelu možemo ponašati kao da radimo s čitavom početnom matricom, neovisno o trenutnoj dubini.

Sukladno Mayers-Milleru, zovemo rekurzivnu funkciju za oba dijela matrice:

```
alignRec(iRow0, iMidRow-1, iCol0, iColMax)
alignRec(iMidRow+1, iRow1, iColMax, iCol1)
```

Ove će rekurzije uz navedene *if* blokove iz prošlog odsječka osigurati da se izračunaju sve vrijednosti na traženom putu koje se spremaju *sol.matches.* Javlja se pitanje o ispravnosti mehanizama ovih rekurzija s obzirom da prva rekurzija zapravo ne radi s polovicama. Pogledajmo to samo na primjeru rekurzije prve razine u kojoj je *BestFwd.p.x != iMidRow* (slika 3.2.).





Slika 3.2. Odbacivanje krajeva matrice koji ne ulaze u poravnanje

Zasivljeni dijelovi matrice označavaju elemente koji se odbacuju jer se nalaze "iza" *starting_pointa* koji je označen crvenom bojom. U tim točkama vrijednost poravnanja je minimalna, a raste u smjeru glavne dijagonale. Elementi označeni žutom bojom su one matrice koje su preostale drugoj razini rekurzije na izračun. Plava bojom označen je srednji redak u trenutnom kontekstu, a zelena je samo preklapanje žute i plave boje. Sada je očito kako neće biti problema pri dubljem grananju rekurzija koje će prepolavljati žute dijelove na pola po retcima, iako žuti dijelovi nisu nužno nastali višestrukim prepolavljenjem.

3.4. Višedretvenost za izračun poravnanja bloka

U slučaju računanja poravnanja za matricu veću od dimenzija 32x32 koristimo višedretvenu funkciju. S obzirom na veličinu dvaju blokova matrice i broj dretvi s kojima radimo dodijeljemo svakoj dretvi *(iCol1 – iCol0) * iThread / ctThreads* stupaca svakog retka te iterativno računamo vrijednosti po retcima, kao i u slučaju jedne dretve.

Može se primijetiti da će svaka dretva znati sama izračunati sve elemente osim onog prvog kojeg dakle čuvamo u zajedničkog memoriji – u polju koje sadrži vrijednosti rubnih elemenata za stupce prije bloka svake dretve. Nakon što izvršimo računanja, svaka će dretva imati lokalnu najbolju vrijednost poravnanja. Ukupna najbolja vrijednost je maksimalna vrijednost svih blokova dretvi. Specifična se logika ove CPU višedretvenosti može opisati pseudokodom:

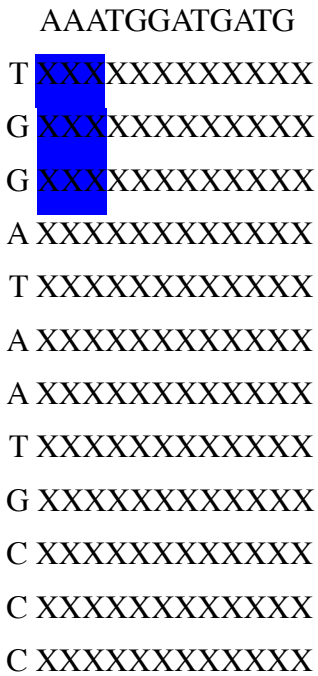
```
MULTITHREAD_ALIGN(iRow0, iRow1, iCol0, iCol1)
    vector<ThreadData> tShared

    // ...za sve dretve iThread...
    for iRow <- iThRow0 to iThRow1
        prevColRow = tShared[iThread].col[iRow-1]
        prevCol = tShared[iThread].col[iRow]
        for iCol <- iThCol0 to iThCol1
            // pronađi najbolji
```

```
                    tShared[iThread].best = bestFoundInCol
...
// Najbolja vrijednost poravnanja polovice
bestAlign = tShared[0].best
for iThread <- 1 to ctThreads
        if tShared[iThread].best > bestAlign:
                bestAlign = tShared[iThread].best
return best
```

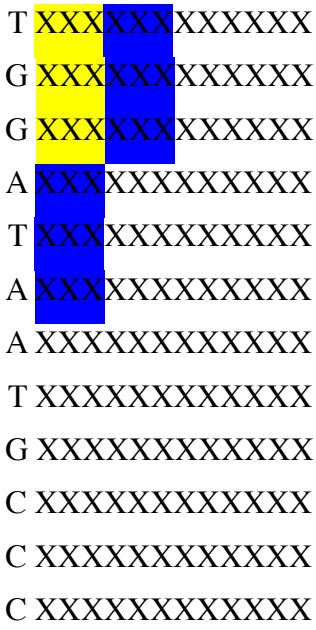
3.5 Implementacija na GPU – *wavefront* algoritam

Algoritam *wavefront* [6] prijedlog je paralelizacije izračuna maksimalne vrijednosti trenutne matrice i time smanjenja vremenske složenosti algoritma na *O(n).* Ideja je podijeliti matricu na CUDA blokove koji se izvode u vremenskoj složenosti *O(1).* Kao i ranije, postupak provodimo s obje strane matrice. Krećući se po glavnoj dijagonali matrice, rješavamo blokove koji se prostiru po antidijagonali (slike 3.3., 3.4. i 3.5.), otuda i naziv algoritma jer podsjeća na valove koji prolaze kroz matricu po glavnoj dijagonali.

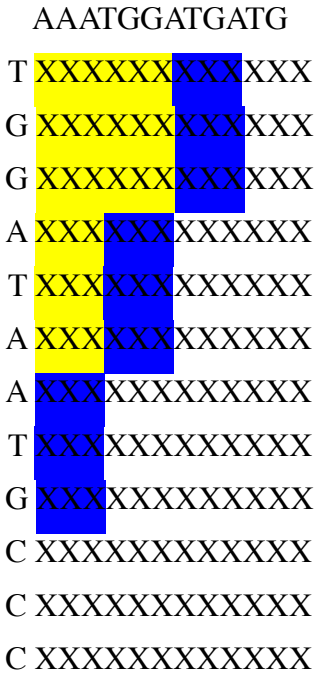


Slika 3.3. Prvi korak wavefront algoritma prve razine.

AAATGGATGATG



Slika 3.4. Drugi korak wavefront algoritma prve razine.



Slika 3.5. Treći korak wavefront algoritma prve razine.

Plava boja označava blokove na antidijagonali koji se trenutno računaju (ne nužno paralelno!), a žuta boja već izračunate blokove. Isti se postupak ponavlja na svim nižim razinama rekurzije (npr. slika 3.6), dokle god je veličina trenutnog bloka veća od 2000x2000. Osim značajki specifičnih uz CUDA programiranje, implementacija CUDA dijela se bitno ne razlikuje od opisanog postupka višedretvenosti na CPU.

Slika 3.6. Posljednji korak wavefront algoritma druge razine (druga polovica).

3.6. Napomena

Ukonnenov algoritam podrezivanja nije implementiran.

4. Rezultati

U tablicama se nalaze usporedbe izvodenja SW# implemetacije algoritma i SW implementacije ostavaren unutar ovog projekta.

Trajanje izvodenja algoritma u sekundama

Duljina sekvence	SW#	SW
100	1.06	1.22
1000	1.35	1.22
10000	1.70	1.46
100000	1.59	2.77
1000000	12.87	137.31

Iznosi poravnjanja (alignment score)

Duljina sekvence	SW#	SW
100	7	7
1000	10	10
10000	13	13
100000	17	17
1000000	19	19

5. Zaključak

Iako Ukonnenov algoritam podrezivanja nije implementiran, rezultati su ipak usporedivi s rezultatima koje daje SW# originalnih autora. Implementacija Ukonnena rezultate bi sigurno dodatno poboljšala. Vjerujemo da su dodatna poboljšanja, osim Ukonnenovog algoritma, moguća, ali nismo imali dovoljno vremena da ih pronađemo i implementiramo.

6. Literatura

[1] Šikić, Domazet-Lošo, "*Bioinformatika*", skripta iz kolegija Bioinformatika, Fakultet elektrotehnike i računarstva, str. 21, 2013.

[2] *Levenshtein distance, in Three Flavors* (2006., Spring). Izvor: https://people.cs.pitt.edu/~kirk/cs1501/Pruhs/Spring2006/assignments/editdistance/Levenshtein%20Distance.htm

[3] vlab.amrita.edu,. (2012.). *Global alignment of two sequences - Needleman-Wunsch Algorithm*. Izvor: vlab.amrita.edu/?sub=3&brch=274&sim=1431&cnt=1

[4] *Improvements in time and memory efficiency for global alignments* (2004.), Cold Spring Laboratory Press, Izvor: http://www.bioinformaticsonline.org/ch/ch03/supp-3.html

[5] Korpar, Šikić, *SW#-GPU-enabled exact alignments on genome scale* (2013., Srpanj), *Advance Access Publication, Sequence Analysis*

[6] Sandes, Melo, *Retrieving Smith-Waterman Alignments with Optimizations for Megabase Biological Sequences using GPU* (2013.), *IEEE Trans. Parallel Distrib. Syst.*

7. Sažetak

Smanjenje vremenske i prostorne složenosti algoritma Smith-Watermana. Prostorna složenost smanjuje se varijacijama Mayers-Millerovog algoritma kojim se kvadratna složenost smanjuje na linearnu u ovisnosti o dužoj sekvenci, a postupak spada u generičku skupinu *podijeli-pa-vladaj* algoritama. Implementacija je podijeljena u dvije skupine rekurzivnih poziva. Pri smanjenju vremenske složenosti koristi se CUDA paralelno programiranje u dijelu računanja vrijednosti matrice poravnanja. Paralelno se računaju blokovi matrice na antidijagonali krečući se prema krajevima po glavnoj dijagonali.