Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matic Pajnič

Implementacija paralelnega algoritma za analizo obogatenosti k-terk v genomskih zaporedjih

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: doc. dr. Tomaž Curk
Somentor: izr. prof. dr. Uroš Lotrič

Ljubljana 2015

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Sestava nukleotidnega zaporedja (A, T, C in G) določa strukturo in posledično funkcijo zaporedja. Uveljavljen način opisovanja genomskih zaporedij je preštevanje frekvenc pojavitve posameznih k-terk. S pomočjo permutacijskega testa se nato določi prag za statistično značilne pojavitve posameznih k-terk. Zaradi obsežnosti genomskih zaporedij, ki lahko obsegajo več milijard nukleotidov, predstavlja analiza sestave zaporedij računsko zahtevno opravilo. Preučite algoritem za preštevanje k-terk in ga implementirajte na grafičnih procesnih enotah. Poročajte o doseženih pohitritvah.

The composition of a nucleotide sequence (A, T, C and G) determines its structure and function. Counting k-mers frequency is an established way used to describe the nucleotide composition of sequences of interest. Permutation test is used to determine the statistical significance of observed k-mer frequencies. Genomic sequences can include billions of nucleotides, which renders k-mer enrichment analysis computationally expensive. Implement a parallel algorithm for k-mer enrichment analysis on graphics processor units and report on the achieved speed-up.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Matic Pajnič sem avtor diplomskega dela z naslovom:

Implementacija paralelnega algoritma za analizo obogatenosti k-terk v genomskih zaporedjih

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Tomaža Curka in somentorstvom izr. prof. dr. Uroša Lotriča,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 15. septembra 2015 Podpis avtorja:

Zahvaljujem se mentorju doc. dr. Tomažu Curku za potrpljenje in vztrajno pomoč pri izdelavi diplomskega dela.

Zahvaljujem se somentorju izr. prof. dr. Urošu Lotriču za podlago znanja, ki mi je pomagalo pri izdelavi diplomskega dela.

Zahvaljujem se staršem za vsa leta podpore.

Nazadnje pa se zahvaljujem določenemu prijatelju za iskro spodbude v pravem trenutku.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
ALE	arithmetic logic unit	aritmetično-logična enota
CPE	central processing unit	centralna procesna enota
CUDA	Compute Unified Device Architecture	/
DNA	deoxyribonucleic acid	deoksiribonukleinska kislina
FMA	Fast Multiply-Accumulate	/
GPE	graphics processing unit	grafična procesna enota
OpenCL	Open Computing Language	/
RNA	ribonucleic acid	ribonukleinska kislina

Povzetek

Cilj diplomske naloge je bil implementirati sekvenčni algoritem za iskanje krajših zaporedij v genomih, ga pohitriti s paralelizacijo ter implementirati paralelno verzijo na grafični kartici. Sekvenčni algoritem je moral poiskati krajša zaporedja znakov v danem zaporedju, ki predstavlja genom. Izračunati je moral frekvence pojavitev zaporedij in pogostost interakcij na podanih položajih ter na naključno premaknjenih položajih, za vsako krajše zaporedje posebej. Na podlagi pojavitev je nato moral določiti, katera krajša zaporedja so za določene položaje v genomu bolj značilna. Na podlagi podatkov o interakcijah med proteini in genomom na določenih položajih, ter na podlagi najdenih krajših zaporedij znakov, je moral nato izračunati in statistično ovrednotiti pogostost pojavitve interakcij. Sekvenčni algoritem smo implementirali v programskem jeziku C, paralelizacijo sekvenčnega algoritma pa smo izvedli na podlagi arhitekture OpenCL, ki omogoča implementacijo algoritmov na grafičnih karticah.

Ključne besede: paralelno računanje, GPE, sinhronizacija, niti, bioinformatika, proteini, DNA, RNA, geni.

Abstract

The goal of this thesis was to implement a sequential algorithm that would search for subsequences in a genome. To accelerate the execution time of this algorithm we designed a parallel version and implemented the parallel version on a graphics card. The sequential algorithm had to search for predefined subsequences in a genome that was represented as a sequence of characters. It had to calculate the frequencies of sequence occurrences and the frequencies of interactions on predefined positions and on randomly modified positions in the genome, for each subsequence. Based on these frequencies it had to identify sequences that were more frequent on certain locations in a given genome. Based on data about protein-RNA interactions on certain locations in the genome, and based on the found character sequences, the algorithm had to calculate and statistically evaluate the frequencies of interactions. The sequential algorithm was implemented in the C programming language, while the parallelization was implemented on the OpenCL architecture.

Keywords: parallel computing, GPU, synchronization, threads, bioinformatics, proteins, DNA, RNA, genes.

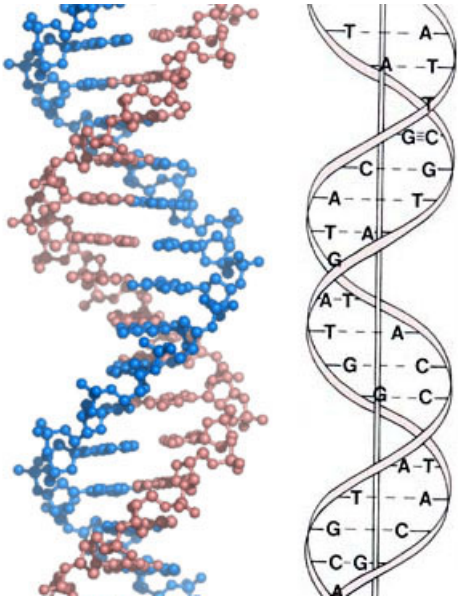
Poglavje 1

Uvod

1.1 Iskanje k-terk v bioloških zaporedjih

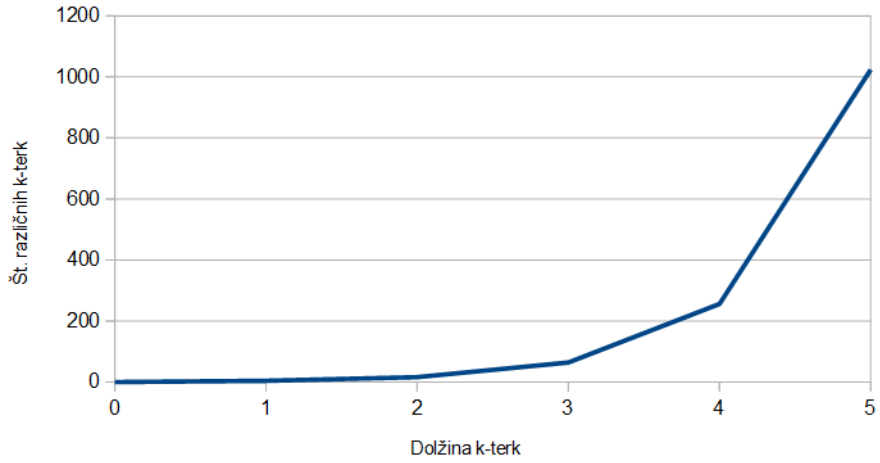
Eno od področij raziskovanja v bioinformatiki je analiza interakcij med med proteini in ribonukleinsko kislino (RNA). Taka interakcija nastane, ko se protein veže na molekulo RNA. RNA je prenašalka genetske informacije v vseh živih organizmih, torej vsebuje bistvene biološke informacije, ki jih organizem trenutno uporablja. Običajno je namen take interakcije modifikacija določene biološke funkcije RNA, npr., kako se bo izrazil določen gen. Do teh interakcij prihaja na točno določenih mestih na RNA. RNA nastane kot prepis podzaporedja genoma oz. DNA. DNA (deoksiribonukleinska kislina) je dvojna vijačnica, ki je sestavljena iz dveh med seboj komplementarnih zaporedij. Celoten genom je sestavljen iz več zaporedij, imenovanih kromosomi. Posamezni kromosomi so sestavljeni iz več milijonov zaporedij nukleotidov, ki jih označujemo s črkami A, C, G in T (glej sliko 1.1).

Eden od načinov analize takih interakcij je iskanje ter štetje podzaporedij nukleotidov ter štetje dogodkov (interakcij) na posameznih mestih v genomu. Ta podzaporedja oz. k-terke (ang. *k-mer*) so lahko različnih dolžin glede na željene vhodne parametre, ker pa se z njihovo dolžino eksponentno spreminja njihovo število (število različnih k-terk), se s tem močno spreminja tudi časovna zahtevnost take analize. V primeru DNK imamo 4 različne tipe nu-



Slika 1.1: Del dvojne vijačnice [14].

kleotidov (adenin - A, citozin - C, gvanin - G, timin - T), kar nam pri dolžini k da n^k različnih k-terk (4, 16, 64, ... različnih k-terk za $k = 1, 2, 3, \ldots$). Slika 1.2 ponazarja rast števila k-terk glede na njihovo dolžino [1].



Slika 1.2: Rast števila k-terk glede na dolžino.

1.2 Obstoječe rešitve

Obstajajo tako sekvenčne kot paralelne rešitve problema opisane analize interakcij med proteini in RNA. Razlikujejo se tako v pristopu k iskanju in štetju k-terk kot v dodatni količini parametrov, ki so pri tem upoštevani (npr., štetje intenzivnosti interakcij, velikosti intervalov v katerih se štejejo k-terke, idr.). Vse to vpliva na časovno zahtevnost algoritmov. Kljub temu je inherentna lastnost tega problema dejstvo, da gre za ogromne količine podatkov, na katerih se izvajajo med seboj enake si operacije (npr., štetje pojavitev vseh različnih k-terk v podintervalih genoma). Že samo iz dolžine celotnega genoma (človeški genom je sestavljen iz približno 3 milijard nukleotidov) sledi, da je problem zelo primeren za paralelizacijo. Če upoštevamo še, da pri takih analizah običajno analiziramo genom na velikem številu podintervalov, od nekaj sto tisoč do več sto milijonov lokacij, lahko s paralelizacijo reševanje tega problema bistveno pohitrimo.

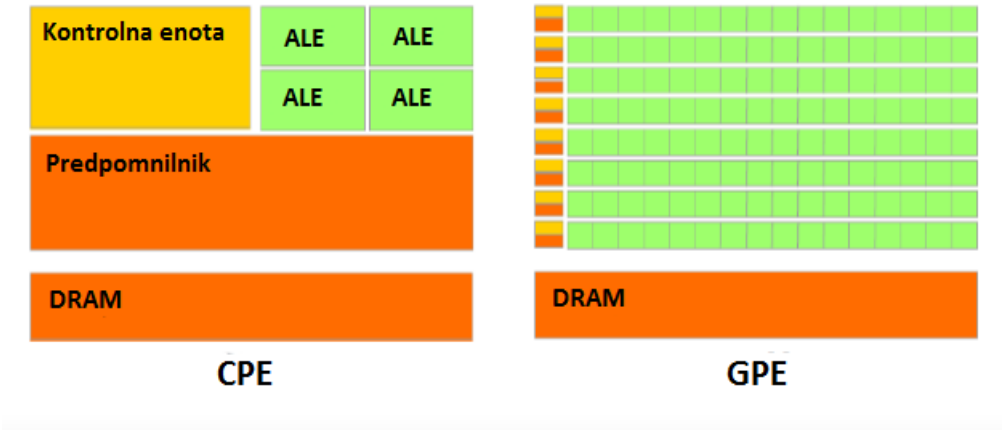
Obstaja več različnih pristopov k paralelizaciji problemov. Ti so odvisni predvsem od strojne opreme, na kateri se bo paraleliziran del rešitve izvajal. Tako med najbolj uporabljanimi pristopi ločimo paralelizacijo na:

- procesorju, ki je sposoben večnitnega izvajanja,
- grafični kartici,
- večračunalniških sistemih.

Procesorje, ki so sposobni večnitnega izvajanja, lahko ločimo na tiste, katerih namen je izvajanje v večji meri sekvenčnih programov, ter tiste, katerih namen je izvajanje močno paraleliziranih programov. Procesorji v prvi kategoriji so v zelo razširjeni uporabi tudi med potrošniki, ki na njih ne izvajajo močno paraleliziranih programov. Tipičen procesor take vrste dandanes lahko izvaja 4, 8 ali 16 niti hkrati. To povprečnemu uporabniku popolnoma zadostuje, izkaže pa se za nezadostno mero paraleliziranega izvajanja, če želimo problem kot je naš zares pohitriti. Procesorji druge vrste so namenjeni predvsem izvajanju paraleliziranih programov. Tipičen primer takih procesorjev je družina procesorjev Intel Xeon Phi. Procesorji iz te družine vsebujejo 60 oz. 61 procesorskih jeder, od katerih lahko vsaka izvaja do 4 niti hkrati. Tako strojno okolje je dosti bolj primerno za izvajanje paraleliziranih programov kot procesorji iz prve kategorije. Podjetje Intel je to družino procesorjev ustvarilo kot odgovor na grafične kartice. Te so sicer bolj efektivne z vidika samega paralelnega izvajanja algoritmov (sposobne so hkratnega izvajanja več niti kot procesor Xeon Phi), je pa paralelno programiranje na njih z vidika programerja bolj zahtevno [2].

Grafične kartice (GPE) so strojna oprema, ki je namenjena paralelnemu računanju. V praksi se večinoma uporabljajo za obdelavo grafičnih elementov, ki so na nivoju kode predstavljeni z matrikami oz. vektorji številskih vrednosti. Te matrike oz. vektorje so grafične kartice sposobne obdelati paralelno z veliko učinkovitostjo, saj te podatkovne strukture hkrati obdeluje veliko število aritmetično-logičnih (ALE) enot To število je neprimerno večje kot pri procesorjih v vsakdanji rabi (Slika 1.3). S programerskega vidika so grafične kartice sposobne ustvariti virtualno neskončno število niti, ki obdelujejo podane podatkovne strukture paralelno. To jih naredi zelo primerne

za reševanje paralelnih problemov, četudi je paralelno programiranje na njih lahko zelo zahtevno.



Slika 1.3: Razlika v številu ALE enot med običajnim procesorjem in grafično kartico [15].

Tretja rešitev vključuje izvajanje paralelnih programov na večračunalniških sistemih. Tu ne gre za izvajanje programa z več nitmi, temveč vsak procesor izvaja kodo z dostopi do lastnega pomnilnika. Ti procesorji med računanjem po potrebi komunicirajo med seboj s pošiljanjem sporočil (večinoma za potrebe delitve dela). Taki sistemi so, podobno kot grafične kartice, v teoriji lahko neomejeni v smislu mere paralelizma, saj je število procesorjev v takem sistemu načeloma lahko poljubno. Veliki sistemi vsebujejo več deset tisoč takih procesorjev. Očitna slaba stran take rešitve je njena cena, saj so taki sistemi izjemno dragi. Poleg tega je komunikacija med procesorji počasnejša kot v strojni opremi, kjer so si procesorski elementi fizično zelo blizu [3].

1.3 Cilji

Cilji diplomske naloge so:

- implementirati algoritem za iskanje k-terk v genomskih zaporedjih,
- implementirati algoritem na GPE,
- primerjati čase izvajanja sekvenčne in paralelne implementacije.

Algoritem ima naslednje vhodne parametre:

- datoteko, ki vsebuje človeški genom,
- datoteko, ki vsebuje koordinate lokacij v genomu, v katerih želimo izračunati obogatenost k-terk,
- relativne meje intervalov okolice vsake lokacije, v kateri naj algoritem išče k-terke,
- število permutacij vhodnih podatkov, ki naj jih algoritem izvede za določanje statistične značilnosti,
- dolžino k-terk,
- način štetja k-terk.

Prvi cilj diplomske naloge je implementacija algoritma, ki bo prebral datoteko s človeškim genomom in preštel k-terke izbrane dolžine ter intenzitete interakcij na izbranih intervalih okoli vsake lokacije, določene v datoteki s parametri za lokacije. Štetje je odvisno od izbranega načina štetja. Nad frekvencami pojavitev bo algoritem nato izvedel *Z-test*, ki nam bo povedal, kako se vrednosti na originalnih lokacijah (te bodo predstavljale vzorec) skladajo z vrednostmi na permutiranih lokacijah (te bodo predstavljale populacijo).

Drugi cilj diplomske naloge je pohitriti in implementirati paralelno verzijo algoritma na GPE.

Tretji cilj diplomske naloge je primerjati čase izvajanja sekvenčne in paralelne implementacije algoritma in s tem ovrednotiti paralelno implementacijo algoritma.

Poglavje 2

Podatki in metode

Sledi opis podatkovnih struktur, nad katerimi bo naš algoritem izvajal operacije, ter opis metod, ki jih bomo uporabili pri snovanju algoritma in interpretaciji rezultatov. Obseg posameznih podatkovnih struktur bo v veliki meri pogojeval snovanje paralelnega algoritma. Pri izbiri arhitekture se bomo odločali med arhitekturama OpenCL in CUDA. Ti sta na področju implementacije paralelnih algoritmov na GPE-jih najbolj razširjeni. Izbiro med arhitekturama bodo pogojevale predvsem operacije, ki jih bo naš algoritem izvajal nad opisanimi podatkovnimi strukturami.

2.1 Opis podatkov o interakcijah med proteini in RNA

Imamo več množic podatkov:

- množico k-terk,
- množico nukleotidov, ki predstavljajo genomsko zaporedje,
- množico lokacij v genomu.

Množica k-terk ima n^k elementov, kjer n predstavlja število različnih nukleotidov (v našem primeru 4), k pa celoštevilsko dolžino k-terk. k je eden od

vhodnih parametrov našega algoritma. Veljavne vrednosti parametra k so na intervalu 1 ... 8. Množica torej vsebuje vse možne kombinacije nukleotidov A, C, G, in T, določene dolžine k. Za dolžino k=2 množica vsebuje elemente: (AA, AC, AG, AT, CA, CG, CC, CT, GA, GC, GG, GT, TA, TC, TG, TT).

Genom je predstavljen z množico nukleotidov v določenem zaporedju. Množica lokacij vsebuje za vsako lokacijo v genomu več podatkov: oznako kromosoma, položaj (koordinato) v genomu, smer branja, ter intenziteto interakcije. Oznaka kromosoma označuje podinterval oz. podzaporedje celotnega genoma, v katerem obravnavamo določeno lokacijo (oz. neposredno okolico te lokacije).

Ker je genom predstavljen z dvojno vijačnico, moramo imeti informacijo o tem, katero stran oz. tirnico genoma trenutno beremo. To nam pove smer branja, ki nam pove, ali vzamemo genom tak kot je, ali iz njega podzaporedja na določenem intervalu izračunamo reverzni komplement genoma in tako dobimo drugo tirnico. Običajno, in to velja tudi za naš primer, je genom predstavljen s podatkovno strukturo, ki dejansko predstavlja le eno od dveh tirnic, saj drugo tirnico lahko izpeljemo iz prve. Reverzni komplement izračunamo tako, da vzamemo željeno podzaporedje genoma, ga obrnemo, ter vsako črko dobljenega zaporedja zamenjamo z njenim komplementom (črko C s črko G in obratno, ter črko A s črko T in obratno).

Intenziteta interakcije nam pove, koliko dogodkov (interakcij s proteinom) je bilo detektiranih na mestu lokacije v genomu.

2.2 Izčrpno iskanje k-terk

Proces štetja k-terk na določeni lokaciji (oz. v njeni neposredni okolici) je sledeč:

1. Za lokacijo preberemo parametre (oznaka kromosoma, lokacija, smer branja in intenziteta interakcije).
2. Preberemo dve podzaporedji genoma (meje oz. "širina" teh podzaporedij so preddoločene in enake za vse lokacije) glede na parameter smeri branja .
3. Za vsako od vseh možnih k-terk (dolžina teh in s tem njihovo število je preddoločeno) glede na način štetja ugotovimo oz. izračunamo:
   (a) ali se pojavi v tih dveh podzaporedjih, ali
   (b) kolikokrat se pojavi v teh dveh podzaporedjih, ter
   (c) v primeru pojavitve oz. pojavitev izračunamo intenziteto interakcije.
4. Permutiramo lokacijo (spremenimo njeno koordinato v določenem intervalu) in ponovimo meritev za p takih permutacij.

Natančnejša implementacija algoritma je razložena v poglavju 3.

2.3 Obdelava rezultatov

Rezultat štetja k-terk bo seznam k-terk z njihovimi frekvencami (št. pojavitev) in intenzitetami interakcij na originalnih ter na permutiranih lokacijah. Iz teh podatkov bomo izračunali, kakšna je statistična značilnost vrednosti na originalnih lokacijah (te predstavljajo vzorec) glede na vrednosti na permutiranih lokacijah (te predstavljajo populacijo). To bomo izvedli z *Z-testom*.

Z-test je statistični test, ki nam pove, kako se vzorec prilega glede na aritmetično sredino populacije ter glede na standardni odklon populacije. Metoda je primerna za vzorce, kjer je velikost dovolj velika (večja od 30), ali kjer so znana parametri populacije. V našem primeru bomo ocenili varianco populacije, tako da bo *Z-test* primeren statistični test za naše cilje. Enačba 2.1 prikazuje izvedbo *Z-testa*, kjer o predstavlja standardni odklon na populaciji, v vrednost na vzorcu, m pa aritmetično sredino na populaciji. V našem primeru torej o predstavlja standardni odklon vrednosti na permutiranih položajih, v vrednost na originalnem položaju, m pa aritmetično sredino vrednosti na permutiranih položajih. Vrednost je frekvenca pojavitve ali intenziteta interakcije, odvisno kaj v posameznem primeru merimo [4].

$$z = \frac{v - m}{o} \tag{2.1}$$

2.4 Opis arhitekture in načina programiranja GPE

2.4.1 Izbira arhitekture

Za paralelizacijo algoritma na GPE imamo na voljo dve razširjeni arhitekturi: CUDA in OpenCL. Razlik v načinu programiranja med arhitekturam je malo. Večje razlike so vidne v dodatnih funkcijah ter v podprtosti posamezne arhitekture s strani računalniške industrije.

CUDA je arhitektura za programiranje GPE-jev, razvito v okviru podjetja Nvidia. Slaba posledica tega je, da arhitektura deluje le na GPE-jih tega podjetja. Arhitektura v računalniški industriji posledično ni tako razširjena kot OpenCL. Po drugi strani CUDA ponuja programerju nekatere funkcije in zmogljivosti, ki jih OpenCL ne premore. Med temi sta najbolj očitni dve. CUDA omogoča izvajanja rekurzije, ter rahlo boljšo zmogljivost v primerjavi z OpenCL, za katero pa je potrebna večja mera optimizacije (težavnost te je seveda odvisna od računskega problema) [5].

OpenCL je arhitektura za programiranje GPE-jev, ki se je razvila v sodelovanju več organizacij, predvsem pa v okviru organizacije Khronos Group. V primerjavi s CUDA ta arhitektura sicer podpira nekaj manj dodatnih funkcij (npr., prej omenjeno pomanjkanje rekurzije), kljub temu pa je dosti bolj razširjena. V nasprotju s CUDA je primerna tako za programiranje GPE-jev kot centralnih procesnih enot (CPE-jev). Nudi tudi večjo skladnost s prevajalniki. V praksi se ta arhitektura uporablja napram CUDA predvsem zato, ker je običajno skladna z obstoječo strojno in programsko opremo. Za uporabo arhitekture CUDA je pogosto potrebna določena računalniška oprema (npr., za računanje s CUDA v dvojni natančnosti je potreben GPE serije TESLA) [6].

Zaradi večje razširjenosti, splošne podprtosti (tudi kar se tiče odpravljanja programskih hroščev), zanemarljive razlike v zmogljivosti in dejstva, da naš problem ne zahteva podprtosti dodatnih funkcij arhitekture CUDA (npr., re-

kurzije), bomo za paralelizacijo našega problema izbrali arhitekturo OpenCL.

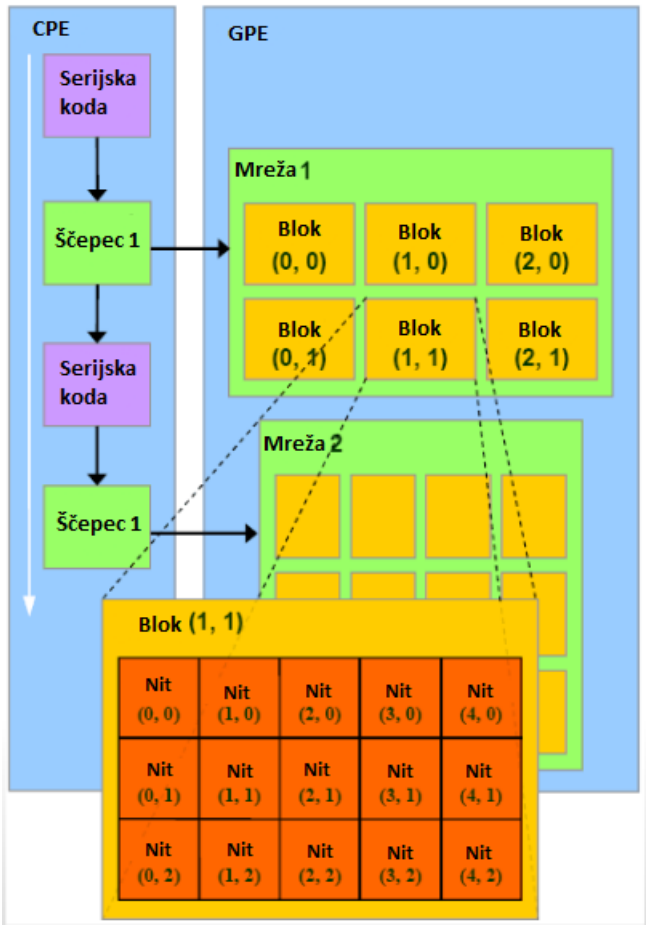
2.4.2 Način programiranja GPE

Izvajalni model arhitekture OpenCL je (enako kot pri arhitekturi CUDA) sledeč:

- Ustvarimo virtualno neomejeno število vzporednih niti, ki se bodo dinamično razvrščale in izvajale na strojni opremi.
- Na GPE gledamo kot na soprocesor, ki podpira neomejeno število niti. Ta GPE ima programsko viden hierarhičen pomnilnik.
- GPE uporabimo na problemih, kjer imamo visoko stopnjo podatkovnega paralelizma. (operacije se izvajajo na veliki količini podatkov, ki so porazdeljeni v podatkovno strukturo, ki je predstavljena z eno-, dvo-, ali trodimenzionalnim poljem).
- Program sestavljajo:
  - ščepci (ang. kernel): samostojni kosi programske kode, ki se izvajajo na GPE-ju,
  - serijska koda: koda, ki se izvaja na gostitelju (običajno CPE) Potrebna je predvsem za prenos kode posameznega ščepca na GPE ter prenos podatkovnih struktur z gostitelja na GPE in nazaj.

Organizacija niti v ščepcu je sledeča (glej sliko 2.1):

- Niti so logično porazdeljene v polje oz. blok niti:
  - blok niti je skupek do največ 1024 niti,
  - posamezne niti v bloku izvajajo enako programsko kodo,
  - vse niti se začnejo izvajati z istim ukazom,
  - niti lahko med seboj komunicirajo preko skupnega pomnilnika.



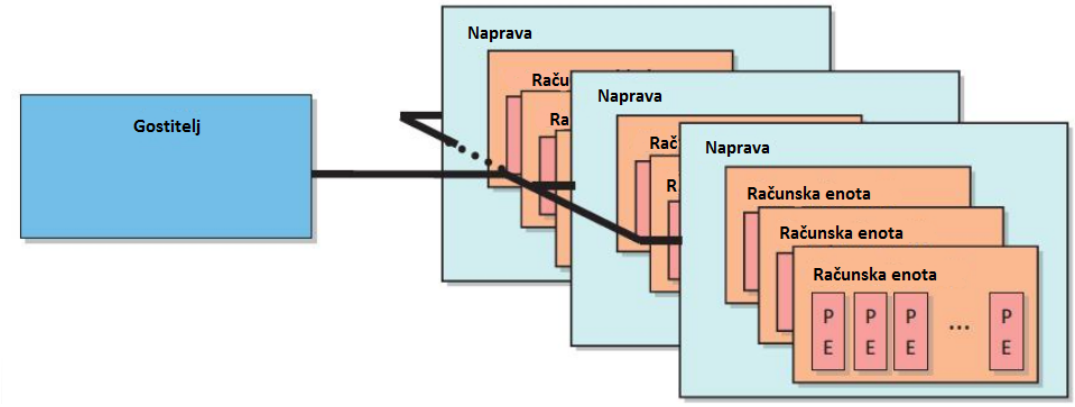
Slika 2.1: Organizacija niti v ščepcu [16].

- Mreža (ang. *grid*) niti je sestavljena iz med seboj neodvisnih blokov niti; blokov je v mreži največ 65535.

Glede na zgoraj abstraktno opisani izvajalni model arhitekture OpenCL in upoštevajoč delovanje GPE (organizacijo niti) lahko izvajanje paralelnih programov na GPE podrobneje opišemo z naslednjimi koncepti.

1. Platforma: Platformo predstavlja en gostitelj in več računskih naprav (glej sliko 2.2). Vsaka računska naprava je razdeljena na več procesnih elementov.

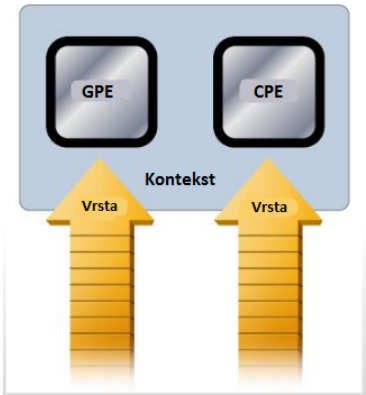
2. Kontekst: Kontekst je okolje za izvajanje ščepca in upravljanje s po-



Slika 2.2: Platforma gostitelj-računske enote [17].

mnilnikom. Med vključenimi napravami med izvajanjem programov poteka sinhronizacija. Kontekst vključuje: množico naprav, pomnilnik, ter ukazne vrste (preko teh poteka izvajanje operacij).

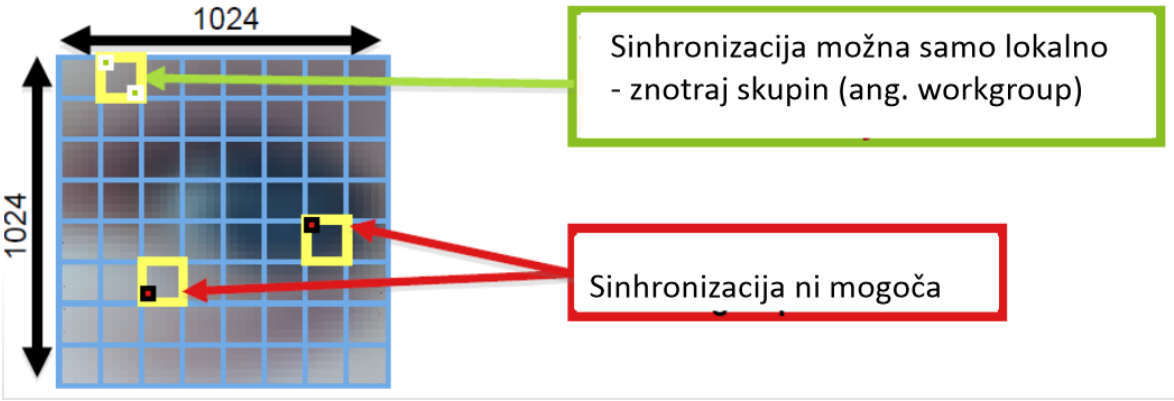
3. Ukazne vrste: vsi ukazi napravam so oddani preko ukaznih vrst (glej sliko 2.3). Vsaka naprava ima svojo (vsaj eno) ukazno vrsto. Več ukaznih vrst se uporablja za ukaze, ki med seboj ne potrebujejo sinhronizacije.



Slika 2.3: Ukazne vrste [18].

4. Izvajalni model ščepca: Za optimalno paralelizacijo algoritma najprej definiramo problemsko področje v eni/dveh/treh dimenzijah (glej sliko

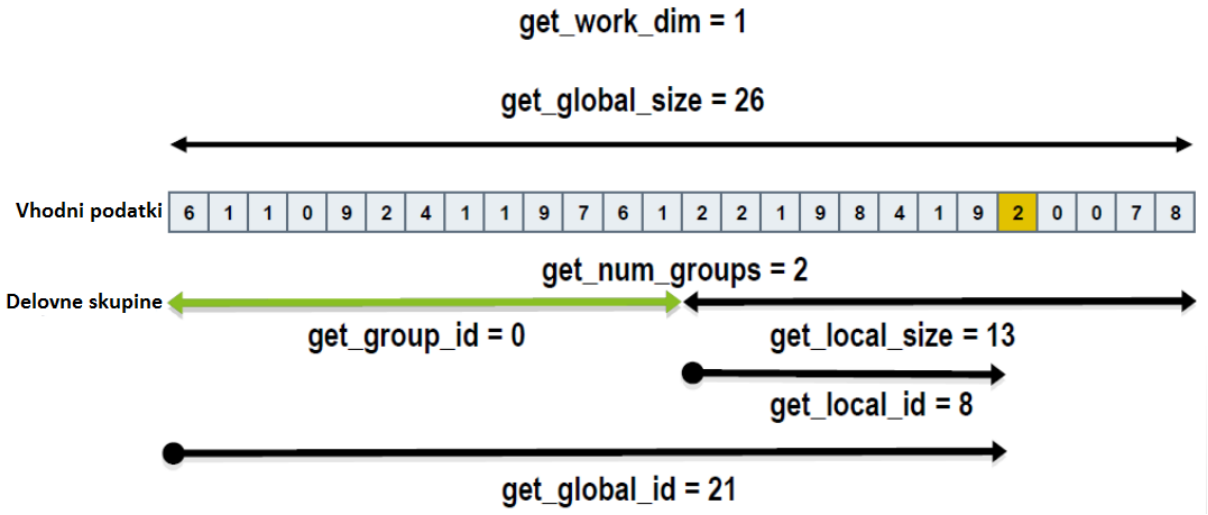
2.4 za delitev v dveh dimenzijah). Nato za vsako točko področja izvedemo ščepec. Pri tem ločimo globalne in lokalne dimenzije izvajalnega modela ščepca. Globalne dimenzije zaobsegajo vse niti, ki so ščepcem na voljo pri izvajanju. Lokalne dimenzije zaobsegajo samo niti v posamezni delovni skupini (ang. *workgroup*). S tem lahko pohitrimo izvajanje paraleliziranega problema, ker s tem izkoristimo lastnosti pomnilniškega modela GPE (več o tem pod razdelkom "pomnilniški model"). Niti so na opisan način razdeljene v delovne skupine, obenem pa so še vedno vidne z globalnega vidika. Zato jih lahko naslavljamo tako globalno kot lokalno. Tak način naslavljanja je nujen za delitev paralelnega reševanja problema na podskupine. Enostaven primer delitve 26 niti na dve skupini v eni dimenziji in različna načina naslavljanja teh niti z uporabo OpenCL metod je viden na sliki 2.5. Za reševanje določenih tipov problemov sicer razdelimo problemsko področje v več kot eno dimenzijo, vendar se bomo našega problema lotili z delitvijo problema v eni dimenziji [6].



Slika 2.4: Problemsko področje [19].

5. Pomnilniški model: Pomnilniški model pri računanju z GPE-ji delimo na:

   - privatni pomnilnik,



Slika 2.5: Delitev niti na delovne skupine [20].

- lokalni pomnilnik,
- globalni pomnilnik,
- pomnilnik konstant,
- pomnilnik na CPU.

Privatni, lokalni, globalni pomnilnik in pomnilnik konstant se fizično nahajajo na GPE. Z vidika strojne opreme tu ne gre za štiri fizično različne tipe pomnilnika na GPE. Ta delitev izhaja iz arhitekture OpenCL, ne pa neposredno iz arhitekture strojne opreme [7].

Privatni pomnilnik je namenjen posameznim nitim oz. njihovim podatkovnim strukturam. Velikost tega pomnilnika na posamezno nit je odvisna od števila niti. Vsak del privatnega pomnilnika je dostopen samo določeni niti [7].

Lokalni pomnilnik je omejen na delovno skupino, katere niti si ga delijo. Vsak del lokalnega pomnilnika je tako dostopen samo določeni skupini niti. Običajno se uporablja za delitev globalnih podatkovnih struktur in problemov na manjše dele struktur in na podprobleme. S tem lahko

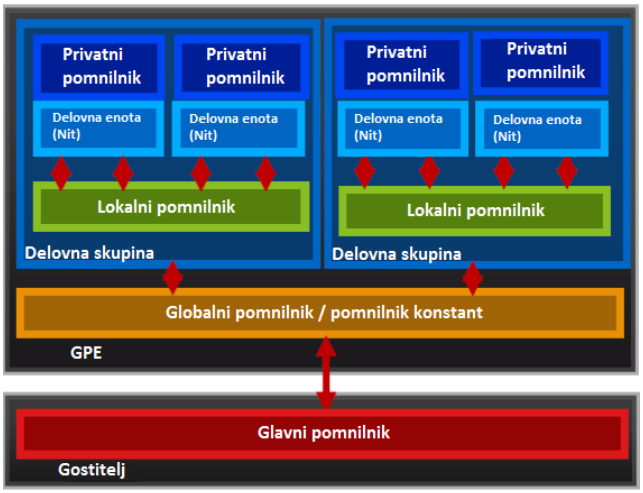
dosežemo večjo mero paralelizma, saj celoten problem oz. podatkovno strukturo lahko obdeluje več delovnih skupin hkrati [7].

Pomnilnik konstant je del globalnega pomnilnika, vendar so do njega možni samo bralni dostopi.

Globalni pomnilnik je dostopen vsem nitim. Vse niti si delijo vse podatkovne strukture v tem pomnilniku.

Pomnilnik na CPU vsebuje sekvenčni del programa in njegove spremenljivke oz. podatkovne strukture, torej tudi vse podatkovne strukture, ki se prenesejo na GPE(-je).

Prenos podatkovnih struktur pred začetkom izvajanja paralelnega dela programa poteka s prenosom iz pomnilnika na gostitelju (CPE) v globalni pomnilnik na GPE, iz tega pa po potrebi naprej v lokalni pomnilnik in naprej v privatni pomnilnik (glej sliko 2.6 za nazorno hierarhijo pomnilniškega modela).

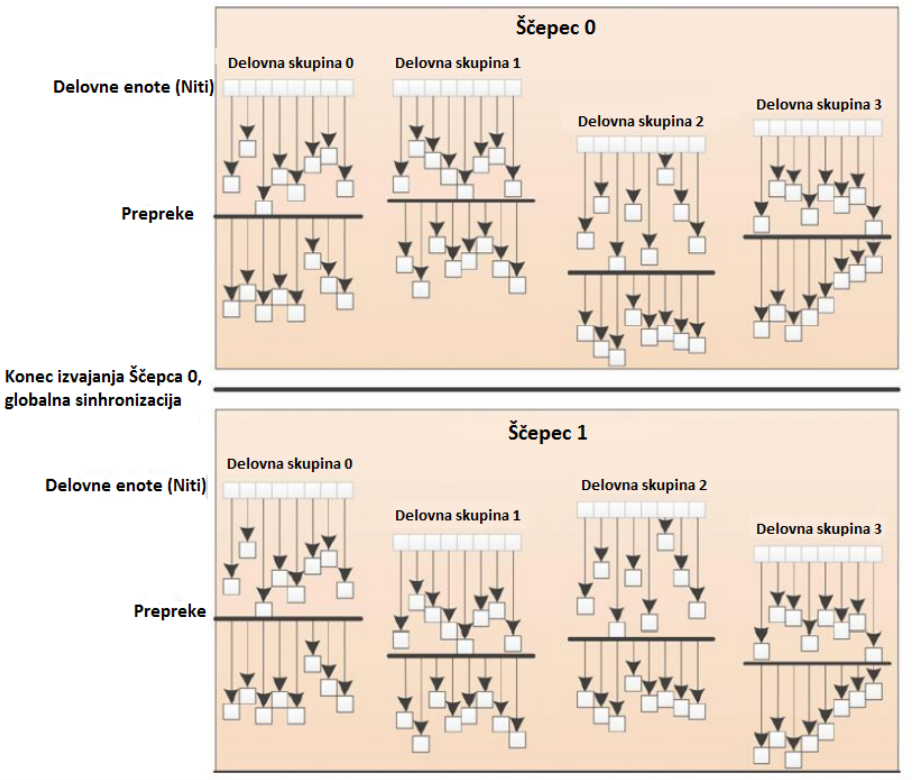


Slika 2.6: Hierarhija pomnilniškega modela [21].

Pri takem pomnilniškem modelu je potrebno skrbeti za konsistentnost. V danem trenutku brez tega namreč ni zagotovljeno, da bodo vse niti videle enako stanje v pomnilniku. Pri privatnem pomnilniku takih težav ni, saj je vsak del tega pomnilnika rezerviran za določeno nit.

Posledično ne more priti do t.i. "tekme za vire." Primer tekme za vire je situacija, v kateri bi dve niti skušali povečati vrednost v registru pomnilnika, vendar bi zaradi odsotnosti sinhronizacije med nitmi obe prebrali iz registra enako vrednost. Ta vrednost bi bila posledično povečana le s strani ene od niti, saj bi vsaka nit zapisala nazaj v register svojo vrednost. V lokalnem pomnilniku niti v isti delovni skupini dostopajo do skupnega pomnilnika, zato je tu potrebno poskrbeti za sinhronizacijo med nitmi. V globalnem pomnilniku tako posamezne niti kot posamezne delovne skupine lahko dostopajo do skupnega pomnilnika. Tu je potreba po sinhronizaciji med nitmi najbolj pogosta [7].

6. Izvajanje niti in sinhronizacija: Pri programiranju na GPE se programer sam odloči, koliko niti bo uporabil za reševanje določenega problema. Programer torej sam definira izvajalno okolje. Za čim večjo mero paralelizacije je dobro, da imamo niti čim več. Tako pohitrimo izvajanje programa, saj olajšamo delo razvrščevalniku, ki ima tako za izvajanje vedno na voljo več snopov niti. Izbira pravega načina delitve dela med nitmi je ključen faktor pri doseganju učinkovite paralelizacije. Potrebno je določiti globalno število niti ter število delovnih skupin. S tem posredno določimo tudi velikost posamezne skupine. Delovne skupine so si enake po številu niti in količini rezerviranega pomnilnika. Slika 2.7 prikazuje primer izvajanja programa na GPE, kjer se izvedeta dva ščepca eden za drugim. Znotraj vsakega problem rešujejo štiri delovne skupine s po osem delovnih enot oz. niti. S slike je vidno, da ko posamezne niti v delovni skupini končajo s svojim delom, programska prepreka (ang. *barrier*) poskrbi za sinhronizacijo. Niti, ki so končale z delom, počakajo ostale niti v delovni skupini, da končajo s svojim delom, preden nadaljujejo z izvajanjem programa. Podobno sinhronizacijo vidimo na višjem nivoju. Ko delovne skupine končajo s svojim delom, z izvajanjem naslednjega ščepca ne pričnejo, preden ostale delovne skupine končajo s svojim delom [6].

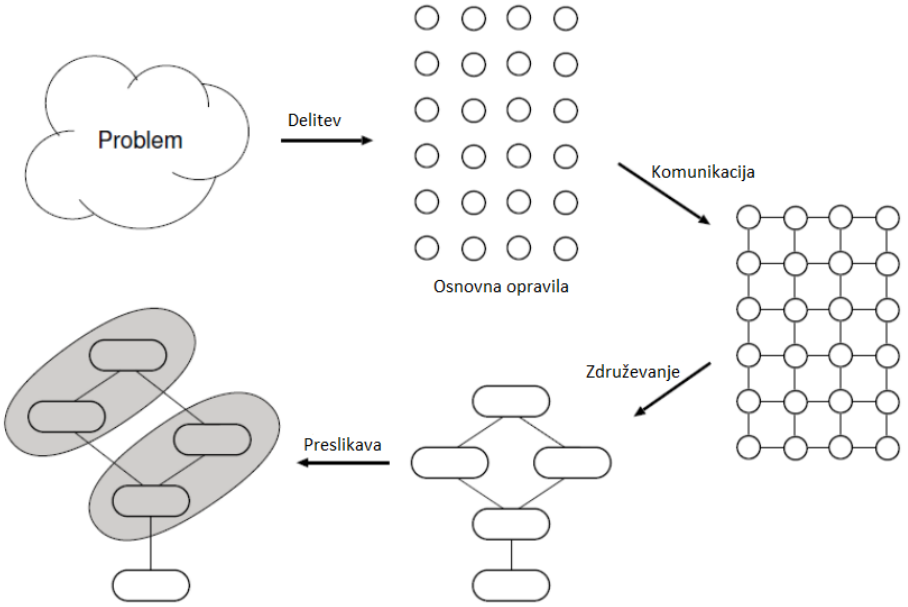


Slika 2.7: Sinhronizacija niti na lokalnem in globalnem nivoju [22].

2.5 Snovanje paralelnega algoritma

Pri snovanju paralelnega algoritma bomo uporabili Fosterjevo metodo. Ta loči snovanje paralelnega algoritma na 4 faze (glej sliko 2.8):

- delitev,
- komunikacija,
- združevanje,
- preslikava.



Slika 2.8: Fosterjeva metoda [23].

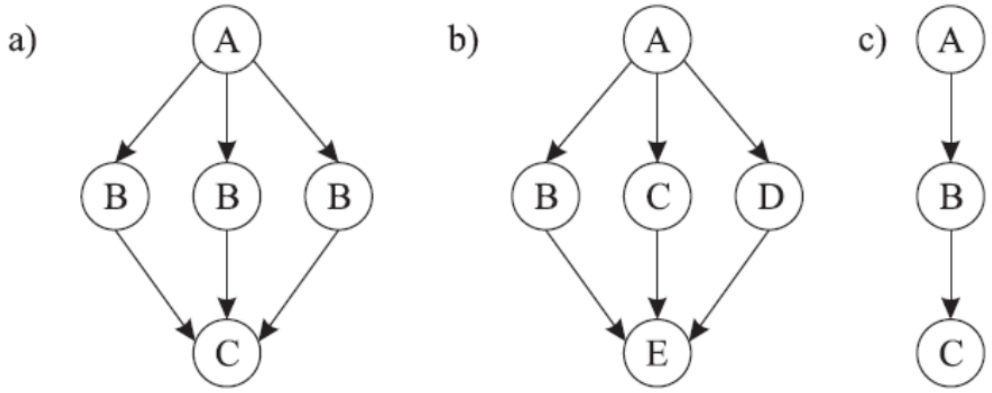
2.5.1 Delitev

Delitev je proces, s katerim razdelimo podatke in računske operacije na majhne dele [8]. Tu ločimo 2 načina delitve :

- podatkovna delitev (delimo podatke),
- funkcijska delitev (delimo računske operacije) [8].

Pri procesu delitve v algoritmu iščemo mesta, kjer se naloge lahko izvajajo sočasno. Pri tem moramo paziti, da to ne vpliva na izid algoritma. Pri podatkovni delitvi pri algoritmu iščemo podatke, ki jih lahko razdelimo in na posameznih delih sočasno izvajamo operacije. Pri funkcijski delitvi pri algoritmu iščemo operacije oz. naloge, ki jih lahko izvajamo sočasno. Slika 2.9 predstavlja grafe poteka različnih algoritmov. Ti algoritmi izvajajo naloge A, B, C, itd. Graf a ponazarja algoritem, kjer isto nalogo lahko sočasno izvajamo na različnih operandih. Graf b predstavlja algoritem, kjer lahko

sočasno izvajamo različne naloge. Graf c predstavlja algoritem, pri katerem nista možna ne podatkovna, ne funkcijska delitev.



Slika 2.9: Grafi poteka algoritmov z različnimi vrstami paralelizmov [24].

Naš algoritem bo izvajal relativno malo operacij na relativno velikih kosih podatkov. Glavni operaciji našega algoritma so:

- iskanje k-terk,
- izvajanje *Z-testa*.

Ti dve operaciji se ne moreta izvajati sočasno, saj je rezultat prve operacije vhodni parameter druge operacije. Funkcijskega paralelizma pri našem algoritmu torej ne bomo mogli izkoristiti.

Pri podatkovnem paralelizmu se je vedno dobro osredotočiti na največje in najpogosteje naslavljane podatkovne strukture v algoritmu. Iskanje k-terk bo v našem algoritmu potekalo na več lokacijah. Njihovo število lahko doseže več deset milijonov. Na vseh lokacijah se bodo izvajale identične operacije, tako da je potencial za podatkovni paralelizem tu velik.
Kvaliteto delitve lahko merimo po naslednjih kriterijih:

1. Število osnovnih opravil narašča z velikostjo problema (v idealnem primeru naj bi povečanje problema povečalo število osnovnih opravil, ne pa njihovo število).

2. Kompleksnost osnovnih opravil je približno enaka (v nasprotnem primeru je posameznim nitim oz. delovnim skupinam težko dodeliti približno enako količino dela).

3. Število identificiranih osnovnih pravil naj bi bilo za velikostni razred večje od števila paralelnih procesov (to zagotovi fleksibilnost v morebitnem naknadnem preoblikovanju algoritma).

4. Število redundantnih podatkov in reduntantnih računskih operacij je minimalno (v nasprotnem primeru se lahko zgodi, da algoritem ne bo mogel obvladati velikih problemov).

5. Identificirali smo alternativne načine delitve (tako vemo, da smo izbrali najboljšo, obenem pa nam to olajša morebitne naknadne spremembe algoritma).

Če nad našim algoritmom izvedemo podatkovno delitev dela, to pomeni da bomo med niti oz. delovne skupine razdelili množico lokacij v genomu. Ker se nad vsemi lokacijami izvajajo identične operacije, s tem avtomatsko zadostimo drugemu kriteriju. Prvemu kriteriju je prav tako zadoščeno, saj z velikostjo množice lokacij premo sorazmerno narašča tudi število iskanj k-terk, ki jih algoritem izvede. Tretjemu kriteriju lahko zadostimo tako, da množico lokacij razdelimo med toliko niti oz. delovnih skupin, da je število lokacij za velikostni razred večje od števila niti oz. delovnih skupin. Če bomo poskrbeli, da niti oz. delovne skupine ne bodo porabljale preveč privatnega oz. lokalnega spomina, bo algoritem lahko obvladal tudi veliko vhodne množice lokacij. S tem bo izpolnjen četrti kriterij.

Da bi zadostili še petemu kriteriju, moramo obravnavati še alternativne načine delitve dela. Načine funkcionalne delitve in njihove neustreznosti za naš algoritem smo že obravnavali. Tako nam ostane še obravnavanje alternativnih podatkovnih delitev. Možne alternativne podatkovne delitve bi bile:

- Delitev dela po permutacijah: delo bi lahko razdelili tako, da bi vsaka nit oz. delovna skupina dobila v obdelavo celotno množico lokacij.

Vsaka nit bi tako opravila delo samo za eno permutacijo. Problem take delitve je, da je nivo paralelizma omejen s številom permutacij (torej z vhodnim parametrom, ki vpliva na rezultate), ne pa z velikostjo problema.

- Delitev dela glede na podzaporedja genoma: delo bi lahko razdelili tako, da bi vsak nit oz. delovna skupina dobila v obdelavo določen del genoma. V tem primeru je število niti oz. delovnih skupin bolj fleksibilno kot v prejšnjem primeru, vendar z velikostjo problema narašča velikost osnovnih opravil namesto njihovo število. Polega tega bi bilo potrebno zagotoviti, da posamezna nit oz. delovna skupina ne dobi v obdelavo lokacij, katerih koordinate so na robu ali izven njihovega podzaporedja genoma.

Tako smo zadostili vsem petim kriterijem delitve dela.

2.5.2 Komunikacija

Komunikacija med procesi (v našem primeru nitmi oz. delovnimi skupinami) predstavlja dodaten strošek paralelnega algoritma. Pri sekvenčnem algoritmu imamo samo en proces, tako da taka komunikacija ni potrebna. Poleg delitve dela je minimizacija komunikacije pomemben cilj v razvoju paralelnega algoritma. Pri tem ločimo dva tipa komunikacije:

- Lokalna komunikacija: poteka med sosednjimi procesi (npr., med nitmi). Časovni strošek take komunikacije je relativno majhen.

- Globalna komunikacija: poteka med množico opravil (npr., med delovnimi skupinami). Časovni strošek take komunikacije je relativno drag [9].

Naš cilj pri komunikaciji je torej minimizirati komunikacijo kot celoto, s poudarkom na minimizaciji globalne komunikacije.

Kvaliteto komunikacije lahko merimo po naslednjih kriterijih:

- Vsa opravila izvajajo približno enako število komunikacijskih operacij. Če je količina komunikacije med opravili preveč neuravnotežena, je algoritem manj učinkovit pri večjih problemih.
- Vsako opravilo naj bi komuniciralo z majhnim številom sosednjih opravil. Komuniciranje z velikim številom sosednjih opravil povzroči velik časovni strošek.
- Komunikacijske operacije morajo delovati sočasno. Če ne, algoritem ni učinkovit pri večjih problemih.
- Opravila lahko opravljajo svoje izračune sočasno. V nasprotnem primeru algoritem ni učinkovit pri večjih problemih.

Komunikacija je na prvi pogled potrebna pri naslednjih delih našega algoritma:

- Delitev lokacij genoma med niti oz. delovne skupine. Če tu za komunikacijo ni poskrbljeno, se lahko zgodi, da več niti oz. delovnih skupin obdela iste lokacije.
- Štetje k-terk na določeni lokaciji. Če je štetje k-terk na posamezni lokaciji paralelizirano, mora biti vsem nitim v delovni skupini, ki to lokacijo obdeluje, jasno, katere k-terke so bile že štete in katere ne. Prav tako mora biti poskrbljeno za to, da več niti ne obdeluje iste k-terke.
- Iteracije po lokacijah in permutacijah. Če bodo niti znotraj posamezne delovne skupine iterirale po lokacijah, ter za vsako lokacijo iterirale še po permutacijah, je potrebno poskrbeti za to, da vse niti hkrati obdelujejo isto vrednost (originalno ali permutirano).
- Združevanje rezultatov. Ko bodo delovne skupine končale z operacijami nad svojimi skupinami lokacij, bodo združile svoje rezultate v globalno podatkovno strukturo. Tu je potrebno poskrbeti za komunikacijo, da ne pride do dirke za vire.

Časovni strošek komunikacije med nitmi oz. delovnimi skupinami pri delitvi lokacij genoma med delovne skupine lahko popolnoma odstranimo. Če so lokacije zapisane v tabeli, lahko za preprečitev, da bi več delovnih skupin obdelovalo iste skupine lokacij genoma, poskrbimo tako, da vsaka delovna skupina zase rezervira n lokacij od indeksa i do indeksa i+n-1. n je v tem primeru lahko (za najbolj enostavno in učinkovito implementacijo) večkratnik števila niti v tej delovni skupini. i je večkratnik identifikacijske številke prve niti v delovni skupini. Oba večkratnika sta seveda odvisna od velikosti same delovne skupine.

Štetje k-terk na določeni lokaciji lahko sinhroniziramo s skupnimi spremenljivkami, ki ponazarjajo, katere k-terke so bile na posamezni lokaciji (oz. v intervalih okoli te lokacije) že štete. Da se izognemo dirki za vire med nitmi, moramo to izvesti s t.i. atomičnimi operacijami. To so operacije, ki jih lahko nad določenim naslovom v pomnilniku izvaja samo ena nit (oz. proces) hkrati. Za to, da več niti hkrati ne obdeluje iste k-terke, lahko poskrbimo z enako rešitvijo kot pri prejšnjem problemu, torej z rabo indeksiranja niti in k-terk (ki so predstavljene z neko tabelo).

Za komunikacijo med vsemi nitmi v posameznih delovnih skupinah lahko v iteracijah poskrbimo z uporabo programskih preprek. Za komunikacijo med nitmi v posamezni delovni skupini bomo lahko uporabili lokalno prepreko. Ta bo poskrbela, da se bodo niti iz iste delovne skupine na pravih točkah v programu počakale pred nadaljevanjem izvajanja programa.

Ob združevanju rezultatov moramo poskrbeti, da ne pride do dirke za vire pri pisanju v skupno podatkovno strukturo v globalnem pomnilniku. Tudi tu lahko za to poskrbimo z uporabo atomičnih operacij.

2.5.3 Združevanje

Prvi dve fazi snovanja paralelnega algoritma sta namenjeni maksimizaciji paralelizma in uvedbi komunikacije med procesi. Cilj faze združevanja je prilagoditev algoritma v trenutni fazi razvoja arhitekturi, na kateri se bo izvajal. To dosežemo s tem, da združimo posamezna osnovna opravila. To

storimo do te mere, da zmanjšamo časovni strošek, ki bi izviral iz ustvarjanja in komunikacije med procesi, od katerih bi sak izvajal le eno osnovno opravilo. Natančneje so cilji združevanja naslednji:

- zmanjšati strošek komunikacije,
- ohraniti skalabilnost algoritma,
- skrajšati čas programiranja [10].

Če bi torej naš algoritem oblikovali le na podlagi faze delitve in komunikacije, bi bil rezultat algoritem, pri katerem bi se lokacije genoma lahko porazdelile med niti na dva načina. Ali tako, da bi vsaka nit obdelala eno lokacijo, ali pa tako, da bi posamezna nit obdelala posamezno k-terko. V obeh primerih bi bil časovni strošek relativno velik. Veliko časa bi se namreč porabilo za ustvarjanje, nato pa še za komunikacijo med nitmi.

Kot rečeno, bomo opravila pri našem algoritmu združili tako, da bodo delovne skupine niti obdelovale vsaka svojo množico lokacij. Tako bomo poskrbeli, da časovni strošek ustvarjanja niti in komunikacije med njimi ne bo prevelik, obenem pa bo algoritem močno paraleliziran. Tako bomo zmanjšali strošek komunikacije, ohranili skalabilnost algoritma, ter skrajšali čas programiranja. Za komunikacijo med delovnimi skupinami neprimerno bo namreč lažje poskrbeti, kot bi bilo v primeru da niti ne bi združevali v delovne skupine.
Kvaliteto združevanja lahko merimo po naslednjih kriterijih:

1. Združevanje naj bi zmanjšalo časovni strošek s povečanjem lokalnosti.
2. Čas, ki ga porabimo za dodatno računanje, ki je posledica združevanja, mora biti manjši od časa, ki bi ga porabili za komunikacijo, če združevanja ne bi izvedli.
3. Dodaten pomnilni prostor, ki je potreben kot posledica združevanja, mora biti dovolj majhen, da je algoritem še vedno dobro skalabilen (da je še vedno učinkovit z večanjem velikosti problema).

4. Združevanje mora producirati opravila z med seboj podobnimi računskimi in komunikacijskimi časovnimi stroški.

5. Količina opravil se še vedno veča z velikostjo problema.

6. Rezultat združevanja je primeren za sodobne in prihodnje računalnike, na katerih naj bi se algoritem izvajal.

7. Števila opravil se ne da zmanjšati, brez da bi s tem povzročili neuravnoteženo obremenitev procesov, otežili programiranje, ali zmanjšali skalabilnost.

8. Združevanje mora opravičiti ceno spreminjanja sekvenčne kode v paralelno.

S tem ko smo niti združili v delovne skupine, se podatki, ki bi se izmenjali med temi nitmi, združi v opravila, ki so enotna nitim v posamezni delovni skupini. Z drugimi besedami, ker smo določena ločena opravila združili, smo prihranili na času, ki bi ga niti porabile za medsebojno komunikacijo. Čas, ki bi ga porabili za komunikacijo, če niti ne bi združili v delovne skupine, bi bil pri končnem združevanju rezultatov med nitmi relativno velik. Tako je zadoščeno tudi drugemu kriteriju. Tretji kriterij lahko izpolnimo tako, da niti združimo v dovolj majhno število delovnih skupin, da na dani arhitekturi ne presežemo dovoljene velikosti lokalnega pomnilnika. Množico lokacij genoma lahko razkosamo v enako velike dele, ki se porazdelijo med delovne skupine, ki so enako velike. Združevanje tako producira opravila z med seboj podobnimi računskimi in komunikacijskimi stroški. Ker lahko število delovnih skupin vežemo na velikost množice lokacij, lahko s tem zagotovimo skalabilnost algoritma in izpolnjenost petega kriterija. Obenem z združevanjem niti v primerno število delovnih skupin zagotovimo, da bo algoritem primeren tako za sodobne računalnike, kot za tiste v bližnji prihodnosti. Izpolnjenost četrtega in petega kriterija posredno vpliva tudi na izpolnjenost sedmega. Optimalno razmerje med številom niti v posamezni delovni skupini in številom samih delovnih skupin bo namreč produciralo

število opravil, ki ga zahteva sedmi kriterij. Glede na to da je za komunikacijo med delovnimi skupinami pri našem algoritmu lažje poskrbeti kot če dela niti ne bi združevali v delo delovnih skupin, smo zadostili tudi zadnjemu kriteriju kvalitetnega združevanja.

2.5.4 Preslikava

Cilj preslikave je dodeljevanje opravil procesnim enotam računalnika oz. strojne opreme, na kateri se bo paralelni algoritem izvajal. Ta korak je namenjen predvsem porazdeljenim sistemom, kjer mora programer sam poskrbeti za dodeljevanje dela posameznim procesnim enotam. V našem primeru se bo algoritem izvajal na GPE, ki sama poskrbi za dodeljevanje nalog [11].

Poglavje 3

Implementacija algoritma

Sledi strukturna analiza sekvenčnega in paralelnega algoritma. Sama predstava algoritmov ima zaradi boljše preglednosti obliko psevdokode. Psevdokodi sledi krajši opis strukture algoritma.

3.1 Sekvenčni algoritem

Sekvenčni algoritem lahko opišemo s psevdokodo algoritma 1:

```
while obstajajo neprebrana podzaporedja genoma do
    Preberi naslednje podzaporedje genoma;
    for l položajev do
        for p permutacij do
            for 4^k k-terk do
                if originalni položaj then
                    if k-terka v okolici then
                        Štej k-terko za originalni položaj;
                        Štej intenziteto interakcije za originalni položaj;
                    end
                end
                else if permutiran položaj then
                    if k-terka v okolici then
                        Štej k-terko za permutiran položaj;
                        Štej intenziteto interakcije za permutiran položaj;
                    end
                end
            end
            Permutiraj položaj;
        end
    end
end
Izvedi Z-test nad skupnimi rezultati vseh iteracij;
```

Algoritem 1: Sekvenčni algoritem.

Kot je vidno iz zgornje psevdokode, algoritem v zunanji zanki iterira skozi vsa podzaporedja genoma. Za vsako podzaporedje nato iterira skozi vse lokacije, ki so predstavljene s parametri: številka podzaporedja, koordinata lokacije, smer branja, intenziteta interakcije. Za vsako lokacijo nato za vsako k-terko najprej opravi meritev na originalni koordinati lokacije, nato pa ko-

ordinato še p-krat spremeni, ter vsakič opravi meritev. Meritev opravi glede na izbran način štetja. Preveri torej število pojavitev trenutne k-terke v okolici koordinate, ter šteje ali eno pojavitev, ali pa vse (enako za intenziteto interakcije).

3.2 Paralelni algoritem

Paralelni algoritem je sestavljen iz dveh modulov, kjer se en izvaja na gostitelju (CPE), drugi pa na GPE-ju. Algoritem, ki se izvaja na gostitelju, lahko opišemo s psevdokodo algoritma 2:

while *obstajajo neprebrana podzaporedja genoma* **do**
    Preberi naslednje podzaporedje genoma;
    Pripravi podatke na prenos (podzaporedje genoma, podatki o položajih);
    Prenesi podatke s CPE na GPE;
    Prenesi rezultate z GPE na CPE;
end
Izvedi Z-test nad skupnimi rezultati vseh iteracij;

Algoritem 2: Del paralelnega algoritma, ki se izvaja na gostitelju.

Algoritem, ki se izvaja na GPE, lahko opišemo s psevdokodo algoritma 3. Kot je vidno iz psevdokode za paralelni algoritem, večji del algoritma prenesemo na GPE. Na gostitelju se izvedejo le branje podzaporedij genoma, priprava in prenos podatkov na GPE in z GPE-ja nazaj na gostitelja, ter izvedba *Z-testa* na končnih podatkih.

Na GPE-ju se izvede računsko najbolj zahteven del celotnega algoritma. Na GPE se prenesejo največje podatkovne strukture, s katerimi operira algoritem (podzaporedje, parametri vseh lokacij za podzaporedje) in manjši potrebni podatki.

Parametri vseh lokacij za podzaporedje se nato razdelijo med delovne skupine. Vsaka delovna skupina tako obdeluje svojo skupino parametrov, sprotne rezultate pa zapisuje v svoj lokalni pomnilnik. Tako kot v sekvenčnem

algoritmu računa gostitelj, tako v paralelnem algoritmu delovne skupine na GPE-ju iterirajo skozi vse lokacije. Za vsako lokacijo nato iterirajo skozi k-terke in za vsako k-terko opravijo meritev. Meritev najprej opravijo na originalni koordinati, nato pa še na p permutiranih koordinatah. Tip meritve oz. tip štetja je tudi tu določen vnaprej. Ko delovne skupine obdelajo vse položaje, iz lokalnega spomina združijo vse rezultate v globalen spomin, iz katerega se nato rezultati prenesejo nazaj na gostitelja.

```
while obstajajo neprocesirani položaji do
    Daj naslednjo skupino neprocesiranih položajev prosti delovni skupini v
    obdelavo;
    for l položajev do
        for p permutacij do
            for 4^k k-terk do
                if originalni položaj then
                    if k-terka v okolici then
                        Štej k-terko za originalni položaj;
                        Štej intenziteto interakcije za originalni položaj;
                    end
                end
                else if permutiran položaj then
                    if k-terka v okolici then
                        Štej k-terko za permutiran položaj;
                        Štej intenziteto interakcije za permutiran položaj;
                    end
                end
            end
            Permutiraj položaj;
        end
    end
end
Združi rezultate vseh delovnih skupin;
```

Algoritem 3: Paralelni algoritem.

Poglavje 4

Teoretična analiza algoritma

Sledi teoretična analiza osnovanih algoritmov. Analize se bomo lotili z analizo časov izvajanja. Izpeljali bomo teoretična časa izvajanja za sekvenčni in paralelni algoritem. Nato bomo paralelni algoritem ovrednotili s primerjavo obeh teoretičnih časov izvajanja na podlagi Fosterjevih načel. Empirična analiza časov izvajanja bo opravljena v poglavju 5.

4.1 Teoretični čas izvajanja

Časa izvajanja obeh algoritmov bomo opisali z izpeljavo časov izvajanja posameznih podproblemov. Naš algoritem ima glede na način štetja k-terk dve delujoči verziji, katerih časa izvajanja sta si v najslabših primerih izvajanja enaka, v ostalih pa se razlikujeta. Način štetja kot parameter v primerjavi s parametri kot je število položajev in število k-terk na čas izvajanja ne more imeti velikega vpliva. Časa izvajanja si bosta zato zelo podobna pri obeh načinih štetja. Čas izvajanja bomo tako analizirali le za enokratno štetje [12, 13].

Za analizo časa izvajanja definiramo naslednje spremenljivke:

- g = dolžina genoma
- z = število podzaporedij genoma

- l = število lokacij
- p = število permutacij
- k = dolžina k-terk
- m = število mest v okolici posamezne lokacije, na katerih algoritem preveri prisotnost k-terk
- x = širina levega intervala okolice lokacije
- y = širina desnega intervala okolice lokacije
- s = število delovnih skupin niti
- n = število vseh niti

4.1.1 Sekvenčni algoritem

Čas izvajanja sekvenčnega algoritma je:

$$T_s = zT_1 + T_2, \tag{4.1}$$

kjer T_1 predstavlja obdelavo enega podzaporedja genoma, T_2 pa izvedbo *Z-testa*.
Čas obdelave enega podzaporedja genoma je enak:

$$T_1 = \frac{l}{z}T_4, \tag{4.2}$$

kjer je T_4 čas obdelave ene lokacije v genomu oz. podzaporedju genoma. Čas obdelave ene lokacije je enak:

$$T_4 = (1 + p)T_5, \tag{4.3}$$

kjer je T_5 čas iskanja vsek k-terk v okolici ene koordinate (enega položaja v genomu, katerega koordinata se z vsako permutacijo naključno spremeni v določenem obsegu). T_5 je enak:

$$T_5 = 4^k T_6, \tag{4.4}$$

kjer je T_6 čas iskanja ene k-terke v okolici ene koordinate. T_6 je enak:

$$T_6 = T_7 + T_8, \tag{4.5}$$

kjer je T_7 čas iskanja ene k-terke v levem intervalu okolice ene koordinate, T_8 pa čas iskanja ene k-terke v desnem intervalu okolice ene koordinate. Ta dva časa sta enaka:

$$T_7 = x - k + 1 \tag{4.6}$$

$$T_8 = y - k + 1 \tag{4.7}$$

Čas izvajanja *Z-testa* je enak:

$$T_2 = 4^k p, \tag{4.8}$$

saj algoritem za vsako k-terko izračuna standardni odklon na populaciji, ki jo predstavljajo rezultati za permutirane položaje.
Čas izvajanja sekvenčnega algoritma je tako enak:

$$T_s = z(\frac{l}{z}(1+p)4^k((x-k+1)+(y-k+1))) + 4^k p \tag{4.9}$$

4.1.2 Paralelni algoritem

Čas izvajanja paralelnega algoritma je:

$$T_p = z(T_2 + T_3) + T_4, \tag{4.10}$$

kjer T_2 predstavlja obdelavo podzaporedij na GPE, T_3 prenos podatkov na GPE ter nazaj na CPE, T_4 pa izvedbo *Z-testa*. T_2 je enak:

$$T_2 = T_5 + T_6, \tag{4.11}$$

kjer je T_5 čas obdelave vseh lokacij danega podzaporedja, T_6 pa čas združevanja rezultatov delovnih skupin. T_5 je enak:

$$T_5 = \frac{\frac{l}{z}}{s} T_7 + b = \frac{l}{zs} T_7 + b, \tag{4.12}$$

kjer je T_7 čas obdelave ene lokacije, b pa časovni strošek rabe preprek med iteracijami. Vidimo, da je T_5 v primerjavi s sekvenčnim algoritmom manjši za faktor s (število delovnih skupin niti), povečan pa za vrednost b. T_7 je enak:

$$T_7 = (1 + p)T_8, \tag{4.13}$$

kjer je T_8 čas iskanja vseh k-terk v okolici ene koordinate (enega položaja v genomu, katerega koordinata se z vsako permutacijo naključno spremeni v določenem obsegu). T_8 je enak:

$$T_8 = \frac{4^k}{\frac{n}{s}} T_9, \tag{4.14}$$

kjer je T_9 čas iskanja ene k-terke v okolici ene koordinate. Vidimo, da je čas iskanja vseh k-terk v okolici ene koordinate v primerjavi s sekvenčnim algoritmom manjši za faktor $\frac{n}{s}$ (število niti v eni delovni skupini). Čas iskanja ene k-terke v okolici ene koordinate je enak kot pri sekvenčnem algoritmu:

$$T_9 = (x - k + 1) + (y - k + 1) = x + y - 2k + 2 \tag{4.15}$$

T_6 je enak:

$$T_6 = \frac{4^k sc}{\frac{n}{s}} = \frac{4^k s^2 c}{n}, \tag{4.16}$$

kjer c predstavlja časovni strošek atomičnih operacij, ki skrbijo za pravilno združevanje rezultatov. Vidimo, da je čas združevanja rezultatov pogojen s številom rezultatov, ki jih je potrebno združiti, s številom delovnih skupin ki združujejo rezultate, s številom vseh niti oz. niti v vsaki delovni skupini, ter s ceno atomičnih operacij.

T_3 je enak:

$$T_3 = \frac{g}{z} + \frac{l}{z} + 4^k = \frac{g+l}{z} + 4^k, \tag{4.17}$$

saj je čas prenosa odvisen od dolžine podzaporedij genoma, števila lokacij za to podzaporedje, ter od števila rezultatov (to je pogojeno s številom k-terk).

Z-test se tako kot pri sekvenčnem algoritmu izvede na CPE, tako da je T_4 enak:

$$T_4 = 4^k p \tag{4.18}$$

Čas izvajanja paralelnega algoritma je torej:

$$T_p = z((((\frac{l}{zs} + b)((1+p)(\frac{4^k s}{n}(x+y-2k+2)))) + \frac{4^k s^2 c}{n}) + \frac{g+l}{z} + 4^k) + 4^k p \tag{4.19}$$

4.1.3 Primerjava teoretičnih časov izvajanja

Vidimo, da se enačbi časov izvajanja razlikujeta v več členih. Čas obdelave vseh lokacij danega podzaporedja je pri paralelnem algoritmu manjši za faktor, ki je enak številu delovnih skupin niti. Prav tako se zmanjša čas iskanja vseh k-terk v okolici ene koordinate. Ta se zmanjša za faktor, ki je enak številu niti v posamezni delovni skupini.

Po drugi strani se v enačbi časa izvajanja paralelnega algoritma pojavijo novi členi, ki predstavljajo časovne stroške. Ti členi predstavljajo čas prenosa podatkov med CPE in GPE, združevanje rezultatov med delovnimi skupinami z rabo atomičnih operacij, ter rabo preprek pri iskanju k-terk.

Ti dve enačbi potrdita načela Fosterjeve metodologije, da bo pretvorba sekvenčnega algoritma v paralelno obliko uspešna, če bo čas, prihranjen s povečanimi hitrostmi izvajanja glavnih opravil, večji od časovnih stroškov, ki jih povzroči raba sinhronizacijskih orodij. Pri našem algoritmu glavni opravili predstavljata obdelava vseh lokacij danega podzaporedja in iskanje vseh k-terk v okolici ene koordinate. Sinhronizacijska orodja, ki lahko povzročijo

časovne stroške, so atomične operacije in programske prepreke. Empirična analiza v poglavju 5 bo pokazala, ali bo prihranjen čas zaradi paraleliacije upravičil odatne časovne stroške zaradi sinhronizacijskih orodij.

Poglavje 5

Rezultati

Sledijo analiza dosežene pohitritve algoritma, analiza rezultatov izvajanja paralelnega algoritma, ter analiza pravilnosti rezultatov izvajanja paralelnega algoritma. Pohitritev bomo analizirali pri različnih vhodnih parametrih. Dobljene vrednosti (pohitritve) bomo med seboj primerjali in interpretirali razlike. Pri rezultatih izvajanja algoritma bomo interpretirali frekvence pojavitev in interakcij posameznih k-terk in iz njih izračunane *Z-teste*. Pravilnost rezultatov bomo interpretirali s primerjavo rezultatov med sekvenčno in paralelno rešitvijo.

5.1 Pohitritev

Pohitritve izvajanja algoritmov smo izmerili pri različno velikih vhodnih datotekah, ki so vsebovale različna števila parametrov o položajih v genomu, ter pri različnih dolžinah k-terk. Graf na sliki 5.1 prikazuje pohitritve pri teh konfiguracijah. Pri vseh testih so bila števila niti in delovnih skupin premo sorazmerna s številom položajev in manjša od tega števila za faktor 7. V predhodnih testiranjih se je v skladu s Fosterjevo metodo taka konfiguracija števila niti in delovnih skupin izkazala za optimalno.

Graf na sliki 5.1 prikazuje pohitritve izvajanja paraleliziranega algoritma pri različno velikih vhodnih datotekah podatkov o položajih v genomu, ter pri

različnih dolžinah k-terk. Vidno je, da je pohitritev izvajanja močno odvisna tako od velikosti vhodnih podatkov, kot od dolžine oz. števila k-terk, ki jih mora algoritem procesirati.

Pri procesiranju 16 k-terk (dolžina k=2) na 500000 položajih v genomu je paralelni algoritem dosegel 5,1-kratno pohitritev. Ko smo vhodne podatke povečali na 1500000 položajev, se je pohitritev povečala na 7,6. To je verjetno posledica dejstva, da pri premajhnem številu vhodnih podatkov pri dani konfiguraciji moči paralelnega računanja GPE-ja ni možno izkoristiti do večje mere. To je vidno iz relativno velike razlike v pohitritvi izvajanja algoritma nad 500000 in 1500000 vhodnih podatkov, kjer je razlika v pohitritvah znašala 49%, razlika med števili procesiranih položajev pa 300%. Nasprotno se je razlika v pohitritvi izvajanja algoritma nad 1500000 in 4500000 vhodnih podatkov izkazala za relativno majhno, saj je znašala le 3,9% pri povečanju vhodnih podatkov za 300%. 1500000 položajev se očitno izkaže kot vrednost, ki je že zelo blizu številu, pri katerem je paralelnost pri naši implementaciji z vidika števila položajev maksimalno izkoriščena.

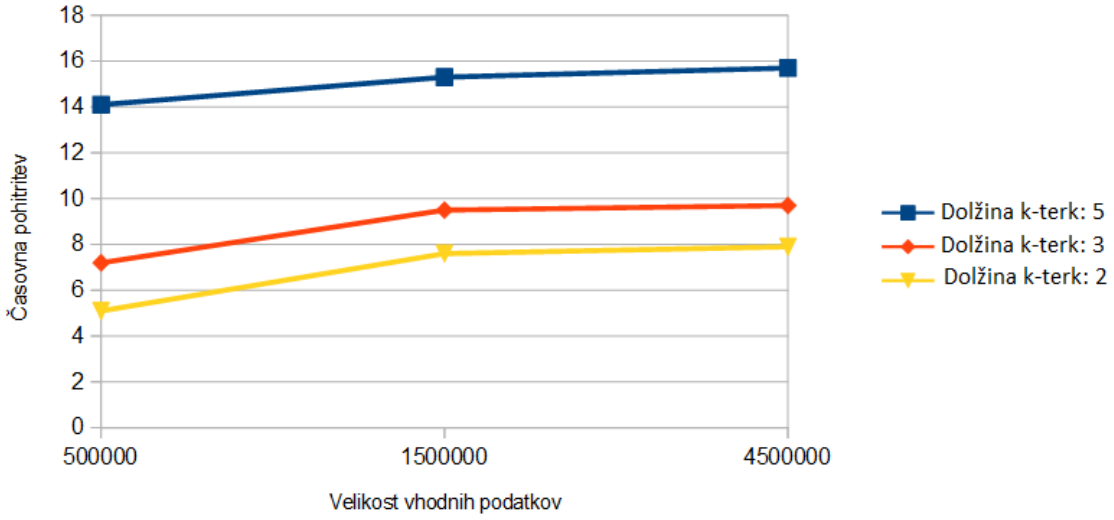
Pri procesiranju 64 k-terk (dolžine k=3) lahko na zelo podoben način zaključimo, da se večji izkoristek paralelne arhitekture GPE-ja pojavi pri številu procesiranih položajev, ki je blizu 1500000. Razlike v pohitritvah so podobne kot pri procesiranju 16 k-terk, in sicer 31,9% pri prehodu s procesiranja 500000 na procesiranje 1500000 položajev, ter 2,1% pri prehodu s procesiranja 1500000 na procesiranje 4500000 položajev. Pohitritvi v odvisnosti od števila položajev sta v primerjavi s procesiranjem 16 k-terk manjši. To je verjetno posledica nastavitev števila niti, ki so v vseh testih znašale 256 niti na delovno skupino. Pri procesiranju 16 k-terk je pri procesiranju vsakega položaja tako delalo le 16 od 256 niti na delovno skupino. Pri procesiranju 64 k-terk je delalo 64 od 256 niti na delovno skupino. Pri procesiranju 64 k-terk napram 16 je tako bil izkoristek niti večji za 300%. Ker je bil delež pohitritve, ki je izviral iz števila izkoriščenih niti v posamezni delovni skupini, večji od deleža pohitritve, ki je bil odvisen od števila procesiranih položajev, je seveda tudi razlika v pohitritvi v odvisnosti od števila položajev manjša

kot pri procesiranju 16 k-terk. Z drugimi besedami, zaradi večjega izkoristka posameznih niti pri procesiranju k-terk napram izkoristku delovnih skupin pri procesiranju položajev, je bil vpliv števila položajev na pohitritev manjši kot pri vhodnem parametru k=2.

Pri procesiranju 1024 k-terk (dolžina k=5) na enak način zaključimo, da se večji izkoristek paralelne arhitekture GPE-ja pojavi pri številu procesiranih položajev, ki je blizu 1500000. Razlike v pohitritvah pri prehodih med 500000 in 1500000 položaji na eni strani, ter 1500000 in 4500000 položaji na drugi strani, so bile pri procesiranju k-terk dolžine k=5 najmanjše. To gre pripisati predvsem dejstvu, da je bila izkoriščenost števila niti v posamezni delovni skupini pri procesiranju takega števila k-terk največja med vsemi testnimi izvajanji. Od 256 niti je vsaka nit za vsak položaj obdelala štiri k-terke, medtem ko je pri procesiranju 64 k-terk mirovalo oz. ostalo brez dela 192 (75%) niti v posamezni delovni skupini, pri procesiranju 16 k-terk pa kar 240 (93,75%) niti. Tudi pri parametru k=5 lahko zaključimo, da je bil delež pohitritve, ki je izviral iz števila izkoriščenih niti v posamezni delovni skupini, večji od deleža pohitritve, ki je bil odvisen od števila procesiranih položajev. Med vsemi testnimi konfiguracijami je bil pri k=5 ta delež največji.

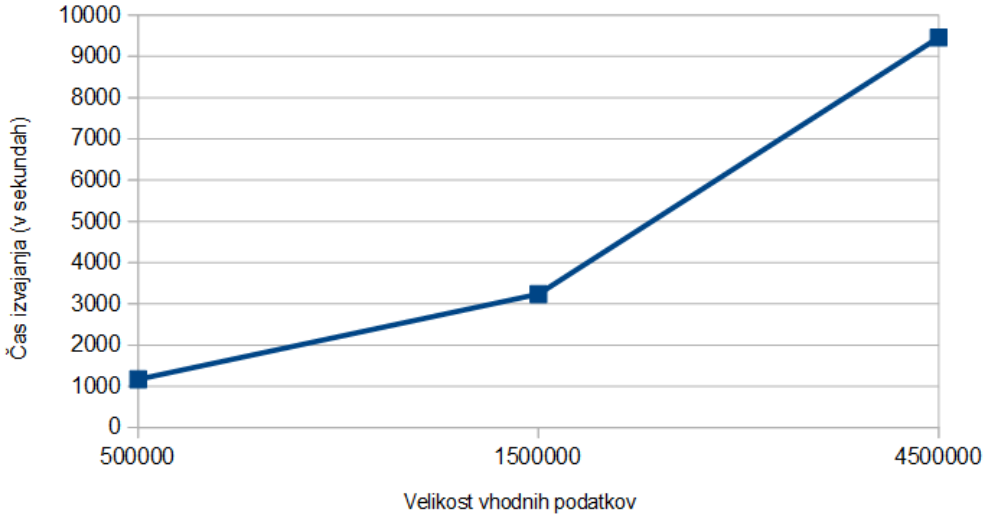
Samo število delovnih skupin ni vplivalo na razlike med časovnimi pohitritvami, saj je bilo razmerje med številom delovnih skupin in številom vseh procesiranih položajev pri vseh testiranjih konstantno.

Z grafa na sliki 5.2 so vidni časi izvajanja paralelnega algoritma pri procesiranju 1024 k-terk in pri različnih velikostih števila položajev. Časi izvajanja so skladni s pohitritvami, prikazanimi na grafu s slike 5.1, saj je pohitritev pri prehodu v velikosti števila procesiranih položajev s 1500000 na 4500000 manjša kot pri prehodu s 500000 na 1500000 procesiranih položajev. Če te razlike ne bi bilo, bi bil čas izvajanja premo sorazmeren s številom položajev, graf na sliki 5.2 pa bi vseboval premico. Graf naj služi tudi za lažjo predstavo deleža časa, ki ga paralelni algoritem porabi za pripravljanje in prenos podatkov z gostitelja na GPE in nazaj. Ti časi so vidni na grafu s slike 5.3 in so v primerjavi s časi procesiranja sami minornega pomena za končne čase



Slika 5.1: Pohitritev pri k=2.

izvajanja paralelnega algoritma pri vseh obravnavanih konfiguracijah vhodnih podatkov. Pri vseh konfiguracijah namreč znašajo približno 0,00026% celotnega časa izvajanja. Priprava in prenos podatkov torej doseže enak delež časa izvajanja paralelnega algoritma pri vseh konfiguracijah vhodnih podatkov.

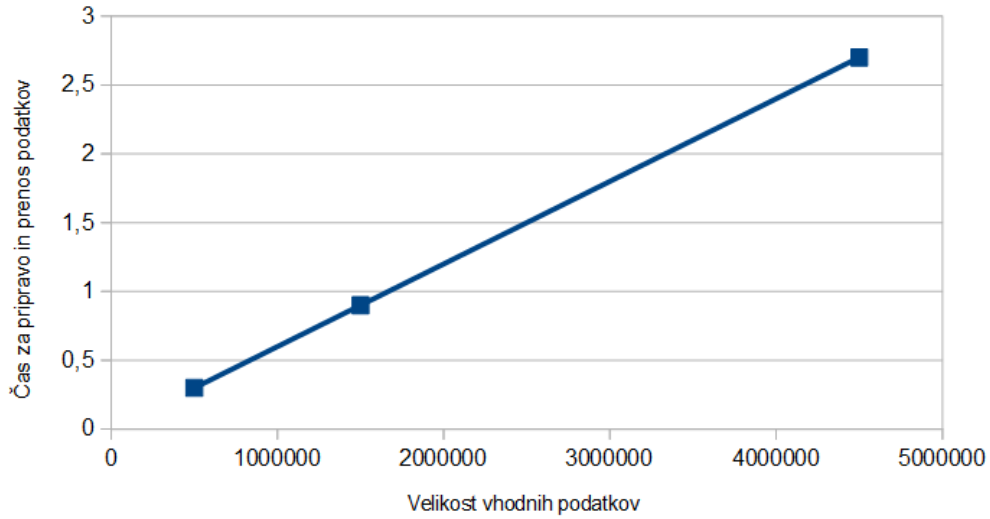


Slika 5.2: Časi izvajanja v odvisnosti od velikosti vhodnih podatkov pri dolžini k=5.

5.2 Odkrite k-terke

Tabeli 5.1 in 5.2 prikazujeta rezultate paralelnega algoritma za dolžino k=2 in za 450000 procesiranih položajev. Prvi stolpec v obeh tabelah vsebuje same k-terke. Drugi stolpec Tabele 5.1 vsebuje števila pojavitev (frekvence) posameznih k-terk na originalnih položajih v genomu, tretji stolpec vsebuje povprečja pojavitev (povprečja frekvenc) posameznih k-terk na permutiranih položajih. Četrti stolpec vsebuje rezultate *Z-testov* na podlagi drugega in tretjega stolpca.

Rezultat *Z-testa* nam v tabeli 5.1 za določeno k-terko pove, za koliko standardnih odklonov se frekvenca k-terke na originalnih položajih razlikuje od povprečne frekvence na permutiranih položajih, če se frekvenca na permutiranih položajih porazdeljuje približno normalno. Če vzamemo za primer k-terko AA, nam vrednost *Z-testa* 0,32 pove, da se izmerjena frekvenca k-terke AA na originalnih položajih za 0,32 standardnega odklona razlikuje od povprečja izmerjenih frekvenc k-terke AA na permutiranih položajih. Pove



Slika 5.3: Časi pripravljanja in prenosa podatkov v odvisnosti od velikosti vhodnih podatkov pri dolžini k=5.

nam tudi, da se je k-terka AA na originalnih položajih pojavila večkrat, kot na povprečnem permutiranem položaju. Na podlagi vrednosti 0,32 lahko tudi rečemo, da obstaja 62,55% možnosti, da je ta vzorec k-terke AA (torej frekvenca pojavitev te k-terke na originalnih položajih) primerljiv s povprečno frekvenco na permutiranih položajih. Če za nasproten primer vzamemo k-terko GG, ki ima vrednost *Z-testa* -1,72, nam ta vrednost pove, da se je k-terka GG na originalnih položajih pojavila manjkrat, kot bi bilo s povprečne frekvence na permutiranih položajih to pričakovati. Z vrednostjo *Z-testa* -1,72 lahko tudi z verjetnostjo 4,27% trdimo, da je ta vzorec k-terke GG primerljiv s povprečno frekvenco na permutiranih položajih.

Drugi stolpec Tabele 5.2 vsebuje števila interakcij na originalnih položajih v genomu za posamezne k-terke, tretji stolpec vsebuje povprečja interakcij na permutiranih položajih za posamezne k-terke, četrti stolpec pa vsebuje rezultate *Z-testov* na podlagi drugega in tretjega stolpca. Če vzamemo za primer iste k-terke kot pri analizi prejšnje tabele, lahko iz vrednosti *Z-testa* 0,40 za k-terko AA zaključimo, da je bilo pri pojavitvah k-terke AA na originalnih

lokacijah več interakcij med genomom in proteini, kot bi bilo pričakovati na podlagi povprečja števila interakcij na permutiranih lokacijah. Na podlagi te vrednosti *Z-testa* lahko tudi z verjetnostjo 65,54% trdimo, da je ta vzorec k-terke AA primerljiv s povprečnim številom interakcij na permutiranih položajih. Po drugi strani lahko iz vrednosti *Z-testa* -1,87 za k-terko GG zaključimo, da je bilo pri pojavitvah k-terke GG na originalnih lokacijah manj interakcij med genomom in proteini, kot bi bilo pričakovati na podlagi povprečja števila interakcij na permutiranih lokacijah. Na podlagi te vrednosti *Z-testa* lahko tudi z verjetnostjo 3,07% trdimo, da je ta vzorec k-terke GG primerljiv s povprečnim številom interakcij na permutiranih položajih.

Iz teh vrednosti lahko zaključimo, da do interakcij med genomom in proteini definitivno bolj pogosto prihaja pri določenih k-terkah, kot pri drugih. Kakšni so razlogi za to ter kakšne so implikacije le-tega, ni bil predmet naše raziskave.

5.3 Pravilnost paralelne rešitve

Rezultati paralelne rešitve niso identični rezultatom sekvenčne rešitve. Razlike se pojavljajo pri štetju frekvenc na originalnih ter permutiranih položajih, ter pri štetju interakcij na originalnih in permutiranih položajih. Vrednosti frekvenc in interakcij v rezultatih paralelne rešitve v povprečju odstopajo od rezultatov sekvenčne rešitve za 0,1%. Posledično vplivajo na *Z-teste*, kjer se vrednosti od sekvenčne verzije razlikujejo za do 5%. Kljub temu so rezultati še vedno reprezentativni, saj razmerja med rezultati posameznih k-terk ostajajo enaka kot pri sekvenčni verziji.

Možnosti za izvor teh razlik je več. Lahko izhajajo iz nepopolne sinhronizacije med nitmi pri združevanju rezultatov. OpenCL namreč v času pisanja še ne omogoča atomičnih ukazov za spremenljivke tipa float, ki predstavljajo format, v katerem so zapisani rezultati našega algoritma (isto velja za format double, ki bi ga lahko uporabili namesto formata float). Ta problem smo skušali zaobiti z uporabo atomičnih operacij za spremenljivke tipa inte-

Tabela 5.1: Frekvence pojavitev in *Z-test* rezultati za k-terke dolžine 2.

k-terka	Frekvenca (orig.)	Frekvenca (perm.)	*Z-test* na frekvencah
AA	40474	39214	0,32
CA	40654	41704	-3,63
GA	38482	39415	-3,47
TA	41866	38894	7,81
AC	37041	37799	-4,29
CC	31005	33516	-4,38
GC	31919	35295	-7,51
TC	40203	39564	1,55
AG	40409	41650	-4,68
CG	10904	13878	-1,07
GG	30845	34393	-1,72
TG	42673	42826	-0,19
AT	42710	40987	6,12
CT	41938	41865	0,65
GT	40601	39980	2,44
TT	43093	40835	6,42

Tabela 5.2: Frekvence interakcij in *Z-test* rezultati za k-terke dolžine 2.

k-terka	Interakcije (orig.)	Interakcije (perm.)	*Z-test* na interakcijah
AA	49915	47983	0,40
CA	49614	51074	-3,69
GA	47050	48325	-3,56
TA	51410	47386	6,79
AC	45383	46311	-3,61
CC	37567	41211	-4,52
GC	39028	43472	-7,29
TC	48591	48376	0,45
AG	49511	51079	-4,64
CG	13137	17438	-1,26
GG	37600	42369	-1,87
TG	52106	52432	-0,33
AT	52360	49977	5,58
CT	51338	51275	0,36
GT	49937	48924	2,82
TT	52876	49857	6,07

ger in sprotnimi pretvorbami med formatoma. S tem smo sicer res zmanjšali razlike v rezultatih, vendar se taka rešitev očitno izkaže za nepopolno pri procesiranju velike količine podatkov, ki so jih zahtevali naši testi.

Drugi razlog bi lahko bila razlika v arhitekturah med GPE in gostiteljem, česar posledice so lahko zaokrožitvene napake pri operacijah z decimalno vejico. Tretji možen razlog je podoben drugemu. GPE uporabljajo računske enote FMA (Fast Multiply Accumulate), ki so namenjene hkratnemu izvajanju operacij seštevanja in množenja. Rezultat teh operacij je lahko zaradi te tehnologije na GPE drugačen kot na CPE, ki te tehnologije ne uporablja.

Poglavje 6

Sklepne ugotovitve

V diplomski nalogi smo najprej implementirali sekvenčni algoritem za iskanje določenih zaporedij v genomu, nato pa smo isti algoritem paralelizirali in ga implementirali na GPE-ju. Sekvenčno verzijo smo implementirali v programskem jeziku C, paralelno verzijo pa še z ogrodjem OpenCL, ki omogoča implementacijo algoritmov na GPE-jih. S tem smo dosegli večkratno pohitritev sekvenčnega algoritma z minimalnimi razlikami v rezultatih med sekvenčno in paralelno implementacijo.

Taka implementacija sekvenčnega algoritma za iskanje zaporedij v genomih ni nova, saj je to področje algoritmov že dokaj razvito v bioinformatiki. V tej diplomski nalogi je sekvenčna implementacija v večji meri služila kot primer algoritma, ki je zelo primeren za paralelizacijo.

Naša implementacija paralelnega algoritma lahko služi kot dober primer paralelizacije danega sekvenčnega algoritma po ustaljenih Fosterjevih načelih zasnove paralelnih algoritmov. Služi lahko tudi kot pokazatelj potenciala izrabe GPE-jev v praksi, s katerimi je možno določene algoritme pohitriti do take mere, da pohitritev izvajanja večkratno upraviči čas snovanja in implementiranja paralelnega algoritma.

Po drugi strani zaradi manjših razlik med rezultati sekvenčne in paralelne verzije implementiranega algoritma ta diplomska naloga služi tudi kot pokazatelj, kako lahko nepopolna sinhronizacija v paralelnem algoritmu vpliva na

rezultate, tudi če so ta odstopanja relativno majhna.

Naša paralelizacija algoritma, kot smo ga zasnovali, bi bila morda bolj uspešna, če bi arhitektura OpenCL podpirala atomične ukaze za spremenljivke tipa float. Ker temu ni tako, bi bilo možno algoritem izboljšati z drugačno zasnovo paralelizacije, verjetno pa taka zasnova brez drastičnih sprememb v sami sekvenčni implementaciji ne bi mogla doseči takega nivoja pohitritve, kot smo ga dosegli z našo implementacijo.

Literatura

[1] (1984) Nomenclature for Incompletely Specified Bases in Nucleic Acid Sequences. Dostopno na: http://www.chem.qmul.ac.uk/iubmb/misc/naseq.html.

[2] J. Fang, H. Sips, L. Zhang, C Xu, C. Yonggang, A. L. Varbanescu, "Test-Driving Intel Xeon Phi", v zborniku *Proceedings of the 5th ACM/SPEC international conference on Performance engineering*, New York, NY, ZDA, 2014, str. 137-148.

[3] Y. Gu, Q. Wu, M. Zhu, N. S. V. Rao, "Efficient pipeline configuration in distributed heterogeneous computing environments", v zborniku *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, New York, NY, ZDA, 2008, str. 432-432.

[4] R. C. Sprinthall, *Basic Statistical Analysis (9th ed.)*, Pearson Education, 2011, pogl. 5.

[5] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, S. Ioannidis, "Gnort: High Performance Network Intrusion Detection Using Graphics Processors", v zborniku *Recent Advances in Intrusion Detection*, New York, NY, ZDA, 2008, str. 116-134.

[6] H. Lee, M. Aaftab, *"The OpenCL Specification Version: 2.0 Document Revision: 26"*, Khronos Group, 3. pogl.

[7] H. Li, G. Fox, G. Laszewski, Z. Guo, J. Qiu, *Co-processing SPMD Computation on GPUs and CPUs on Shared Memory System*, Manson Publications, 2011, 7. pogl.

[8] B. W. Kernighan, P. J. Plauger, *The Elements of Programming Style*, McGraw-Hill, Inc., New York, NY, ZDA, 1982, 5. pogl.

[9] M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill Higher Education, 2004, 12. pogl.

[10] A. Burns, "Scheduling hard real-time systems: a review", *Software Engineering Journal*, št. 6, izv. 3, str. 116 128, 1993.

[11] N. Fisher, J. H. Anderson, S. Baruah, "Task partitioning upon memory-constrained multiprocessors", v zborniku *Embedded and Real-Time Computing Systems and Applications*, New York, NY, ZDA, 2005, str. 416-421.

[12] T. Cormen, C. Leiserson, R. Rivest, C. Stein, *Introduction to Algorithms, Third Edition*, MIT, 2009, str. 53.

[13] T. H. Cormen, *Introduction to Algorithms, Second Edition*, MIT, 2001, str. 35.

[14] (2015) Double Helix. Dostopno na: https://en.wikipedia.org/wiki/DNA.

[15] (2015) Cpu-gpu. Dostopno na: https://upload.wikimedia.org/wikipedia/ commons/c/c6/Cpu-gpu.svg.

[16] (2015) Cuda blocks. Dostopno na: http://www.dmi.unict.it/ bilotta/ gpgpu/ingv/cuda-blocks.png.

[17] (2015) Host, compute units. Dostopno na: http://developer-static.se-mc.com/wp-content/blogs.dir/1/files/2013/10/PlatformModel-660x360.jpg.

[18] (2015) Order queues. Dostopno na:
http://image.slidesharecdn.com/kite-opencl-course-141010090714-conversion-gate01/95/hands-on-opencl-42-638.jpg?cb=1413173038.

[19] (2015) OpenCL synchronization. Dostopno na:
http://image.slidesharecdn.com/ozviz2010-bednarz-introduction-to-opencl-131203161008-phpapp01/95/introduction-to-opencl-2010-28-638.jpg?cb=1386090150.

[20] (2015) Workgroup example. Dostopno na:
http://www.codeproject.com/KB/showcase/Work-Groups-Sync/image001.gif.

[21] (2015) Memory hierarchy. Dostopno na:
http://www.nehalemlabs.net/prototype/wp-content/uploads/2014/05/Fig1.png.

[22] (2015) Thread synchronization. Dostopno na:
https://www.mql5.com/en/articles/407.

[23] (2015) Foster method. Dostopno na:
http://www.mcs.anl.gov/ itf/dbpp/text/img160.gif.

[24] (2015) Grafi poteka. Dostopno na:
http://wiki.expertiza.ncsu.edu/ images/f/fb/Smid.png.