T A R T U Ü L I K O O L

MATEMAATIKA-INFORMAATIKATEADUSKOND
Arvutiteaduse instituut
Infotehnoloogia eriala

Anti Alman

Piiratud võimsusega regulaaravaldistele sobituvate sõnade loendamine

Bakalaureusetöö (6 EAP)

Juhendajad: Meelis Kull
Sven Laur

Autor: ……	"….." märts	2013
Juhendaja: …….	"….." märts	2013
Juhendaja: …….	"….." märts	2013
Õppetooli juhataja: ……	"….." ……..	2013

2013 märts

Sisukord

Sissejuhatus

Bioinformatika üheks alamosaks on erinevate haiguste ja nende ravi uurimine eesmärgiga töötada tulevikus välja paremaid diagnostika- ja ravimeetodeid. Selle jaoks on mitmeid erinevaid võimalusi, milledest üks suund on reaalsetelt katsealustelt proovide võtmine ning analüüsimine. Analüüsi käigus saab võrrelda katsealuste proove tervete inimeste proovidega ja otsida katsealuste proovides leiduvat ühisosa, mida võib nimetada bioloogiliseks markeriks. Selle kaudu saab hinnata haiguse tõttu organismis toimunud muutusi, mis aitab meil haigust paremini mõista.

Selline analüüsimine on pikk ja keerukas protsess, mis hõlmab palju erinevaid samme, mis sooritatakse sageli erinevate inimeste poolt. Tuleb välja valida sobivad katsealused soovitud haiguse uurimiseks, võtta neilt proovid, töödelda proovid arvutile arusaadavaks, analüüsida nende proovide koostist ja ühisosa, võrrelda proove tervete inimeste proovidega ja esitada analüüsi tulemused. Nimetatud sammud kirjeldavad seda protsessi väga üldistatud kujul. Tegelikkuses jaguneb iga väljatoodud punkt omakorda veel mitmeks keeruliseks alamosaks.

Käesoleva bakalaureusetöö raames keskendun ühe konkreetse ülesande lahendamisele proovide võrdlemise etapis. Võrdluse sisendina on töö käigus kasutada suur kogus erinevate proovide andmeid ja ülesandeks on leida huvipakkuva bioloogilise markeri sagedus kõigis nendes proovides. Selleks otstarbeks loon kaks algoritmi ja kolm programmi, mis kõik on juba leidnud kasutamist reaalsete ülesannete lahendamisel (ärilises projektis). Lisaks võrdlen nende programmide omavahelist kiirust erineva hulga markerite sageduste samaaegsel otsimisel.

Bakalaureuse töö on valdavas osas praktilist laadi. Töö esimeses osas kirjeldan ülesande püstituse koos lühikese ülevaatega ülesande taustast, sisendiks olevad andmed ja loodava programmi väljundi. Sisu osas toon kolm programmi ja kirjeldan nende programmide loomisel kasutatud meetodeid, tekkinud probleeme ning puuduseid. Selle käigus puutun muuhulgas kokku ülesande paralleliseerimisega, hõreda andmestiku mälus hoidmisega, bitivektoritega, tööde järjekorra haldamisega ning Pythoni programmeerimiskeele mõningate eripäradega (võrreldes C++ keelega).

1 Ülesande püstitus

Käesoleva töö lähteandmestikuks on patsientidelt võetud proovid, millest osad on teatud tunnusega (nt haiged) ja teised ilma selle tunnuseta (terved) patsientidelt. Proovid on viidud masintöödeldavale kujule, nii et iga proovi iseloomustab umbes 1 miljon sõna, mille igaühe pikkus on 12 tähemärki, kusjuures iga sõna võib esineda mitu korda. Sõnade esinemissagedused iga proovi kohta on teada. Tähemärgid pärinevad kindlaksmääratud 20-märgilisest tähestikust

`A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y.`

Nimetatud andmed paiknevad ühises koondtabelis, mille veergudes on erinevad proovid ning ridades erinevad sõnad. Nende ristumiskohas on toodud vastava sõna esinemissagedus selles proovis (täpsem kirjeldus peatükis 1.1 Sisendandmed).

Sõnahulkade paremaks kirjeldamiseks võidakse kasutada regulaaravaldisi, kus on fikseeritud vaid mingid kindlad tähemärgid ja nendevahelised kaugused (kauguste märkimiseks kasutatakse iga tähemärgi koha peal punkti). Näiteks tähekombinatsioon `A..N.T.I.` moodustab ühe regulaaravaldise ja võib iseloomustada väga suurt hulka erinevaid sõnu.

Selgub, et niisuguste regulaaravaldiste põhjal on võimalik luua bioloogilisi markereid. Markerite loomise protsessi käigus on oluline kiirelt tuvastada, millised on mingi regulaaravaldise sagedused kõigis proovides, kuid see info koondtabelis puudub ja tuleb arvutada iga regulaaravaldise kohta eraldi olemasolevate andmete põhjal.

Käesoleva bakalaureusetöö ülesandeks on töötada välja ja implementeerida algoritm, mis lubaks võimalikult kiirelt leida suvalise regulaaravaldise esinemise sagedused kõigis proovides.

Keeruliseks teeb ülesande see, et sisendiks olev andmete maht on väga suur. Kokku on hetkel 1307 proovi ja umbes 79 miljonit erinevat sõna. Kõvakettal võtab koondtabeli fail ruumi umbes 193 GB. Sellest hoolimata sooviksime saavutada kiirust umbes 1 s regulaaravaldise kohta. See tähendab, et lihtsamad ühe protsessori tuuma peal jooksvad ja kõvaketta kiirusest sõltuvad lahendused ei ole antud ülesande jaoks piisavad ja tuleb leida parem lähenemine, kui võimalikud olemasolevad tööriistad (andmebaasid, `grep`

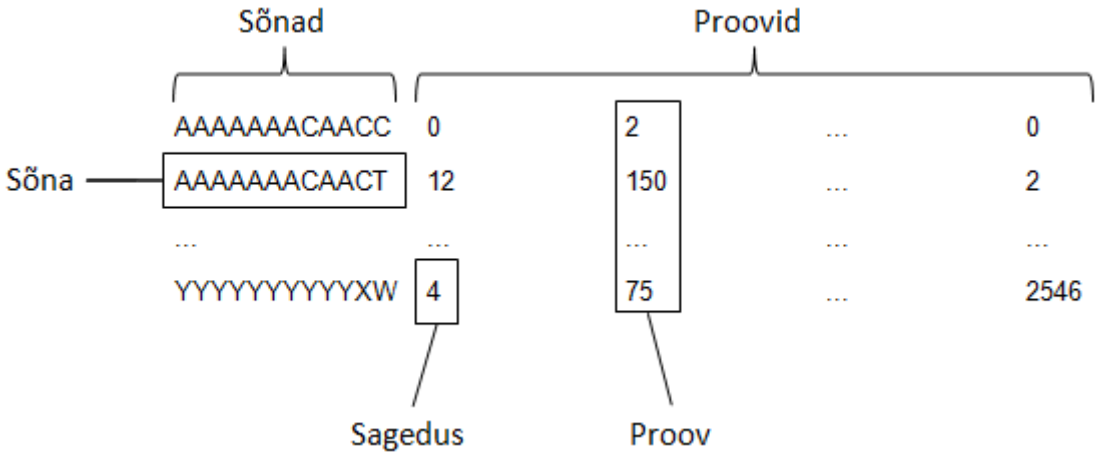
jne) suudavad pakkuda. Selle käigus saan teha ka mitmeid ülesande spetsiifilisi optimiseeringuid eesmärgiga tõsta algoritmi kiirust.

Valmiv programm läheb reaalselt kasutusse Tartu Ülikooli BIIT töögrupis ühe Eesti biotehnoloogia ettevõttega seotud projekti juures ja moodustab ühe komponendi suuremast töövoost.

1.1 Sisendandmed

Ülesande sisendiks on koondtabel (Joonis 1), kus on esitatud sõnad ja nende sõnade esinemissagedused kõigis proovides. Seda võib vaadata, kui tekstifailina olemasolevat andmebaasi, mille põhjal peab loodav programm sooritama loendamist vastavalt kasutajalt saadud päringutele (regulaaravaldiste loendile).

Tegu on tabulaatoriga eraldatud tekstifailiga, kus esimeses veerus on sõnad ja järgnevates veergudes nende sõnade esinemissagedused. Tabel ei sisalda päiseridu.



Joonis 1. Koondtabeli formaat

Koondtabeli erinevate elementidele viitamiseks kasutan edaspidi läbivalt mõisteid, mis on toodud tabelis 1.

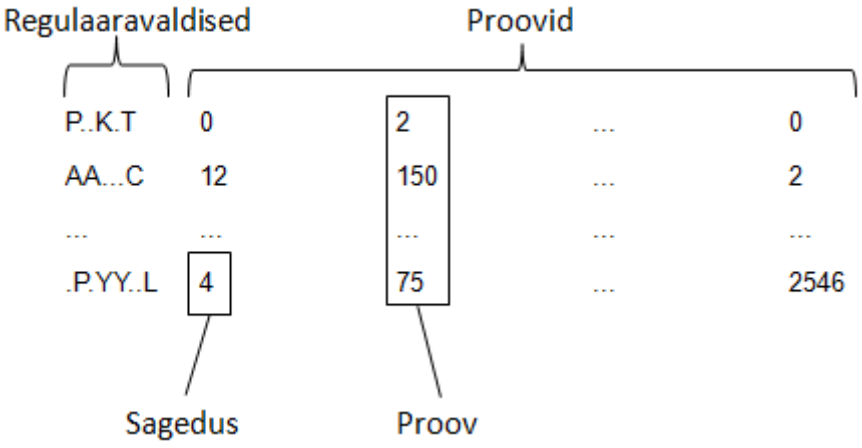
Sõnad	Hulk mille seast otsin regulaaravaldisele sobituvaid sõnu
Sõna	Üksikelement sõnade hulgast, mida võrdlen regulaaravaldisega
Proovid	Kõik proovid, kus on sõnade hulga iga elemendi esinemissagedused loetud
Proov	Sõnade hulga iga elemendi sagedus vastavas proovis
Sagedus	Vastaval real oleva sõna esinemiskordade arv vastavas veerus olevas proovis

Tabel 1. Koondtabeli elemendid

Tabeli 1 põhjal võib joonisel 1 toodud koondtabeli näitest välja lugeda, et sõna `AAAAAAACAACT` esineb esimeses proovis 12 korda ja teises proovis 150 korda.

1.2 Väljund

Programmi väljund (Joonis 2) on formaadilt väga sarnane koondtabelile. Iga sisendiks oleva regulaaravaldise kohta on väljundfailis üks rida, millele on kirjutatud regulaaravaldis ning selle regulaaravaldise sagedused kõigis proovides. Proovid on samas järjekorras nagu ka koondtabelis.



Joonis 2. Väljundi formaat

1.3 Programmi nõuded

Käesolevas peatükis toon välja olulisemad nõuded, millele peab bakalaureusetöö käigus valminud programm vastama.

1. Programm peab töötama 12 tähemärgi pikkuste sõnadega, mis kasutavad peatükis 1. Sissejuhatus toodud piiratud tähestikku.
2. Programm peab oskama töötada peatükis 1.1 Sisendandmed määratud andmestikuformaadiga.
3. Programmi väljund peab vastama peatükis 1.2 Väljund määratule.
4. Programm peab suutma loendada regulaaravaldisi, mis on kuni 12 tähemärki pikad ja võivad sisaldada punkte.
5. Programm peab suutma loendada suuremaid regulaaravaldiste komplekte mõistliku ajaga. Tuhande regulaaravaldise suuruse sisendi korral peab tulemuse saamiseks kuluma vähem kui 10 minutit.
6. Programm peab olema võimeline loendama mitut eraldi sisestatud regulaaravaldiste komplekti samaaegselt.
7. Uusi loendamistöid peab olema võimalik igal hetkel lisada.
8. Programmi peab olema võimalik kasutada shell skriptide osana. See tähendab, et töö käivitamiseks kasutatav programmiosa peab vajadusel enne väljumist ootama ära töö lõppemise.

1.4 Alternatiivsed lahendused

Lisaks käesoleva bakalaureuse töö sisus toodud lahendustele on veel mõned võimalikud lähenemised antud ülesande lahendamiseks. Käesolevas peatükis kirjeldan nendest kaks ja toon välja ka nende lahenduste puudused.

Esimeseks võimaluseks oleks kasutada regulaaravaldisele sobituvate sõnade leidmiseks Linuxi tööriista `grep`[1]. Tegu on üldjuhul väga hea ja kiire tööriistaga, millele saab anda otsimiseks ka mitmeid regulaaravaldisi korraga. Siinkohal on aga puuduseks see, et `grep` tagastab kõik read, mis vastavad vähemalt ühele sisendina antud regulaaravaldistest. Antud ülesandes on vaja leida read regulaaravaldise kaupa. Selleks oleks vaja `grep` käivitada iga regulaaravaldise kohta eraldi, mis on tõenäoliselt ajakulukas ja ei skaleeru suurte regulaaravaldiste komplektide korral hästi.

Saadud tulemuse töötlemiseks tuleks lisaks luua veel eraldi tööriist või skript, mis viiks tulemused peatükis 1.2 Väljund määratud kujule. Lisaks on `grep` piiratud ka kõvaketta kiirusega.

Teiseks võimalikuks lahenduseks oleks koondtabelis olevate andmete importimine andmebaasi (näiteks PostgreSQL) ja siis leida iga regulaaravaldise sagedus SQL päringute abil. Ka sellisel juhul tuleks luua eraldi skript või tööriist, mis viiks tulemused soovitud kujule, aga erinevalt `grep`'ist ei ole vaja andmete töötlemiseks lisaarvutusi teha, sest kõik arvutused saab sooritada SQL päringu käigus. Kuna üldjuhul paiknevad andmebaasi andmed siiski kõvakettal, siis oleks see lähenemine samuti piiratud kõvakettalt lugemise kiirusega.

2 Naiivne lahendus

Ülesande lahendamiseks olen kirjutanud kolm programmi ja kasutanud kahte erinevat algoritmi. Käesolevas peatükis kirjeldan selle, kuidas lahendasin ülesande esmakordselt. Seda võib vaadelda, kui lihtsat lähenemist püstitatud probleemile. Toon välja ka esialgse lahenduse põhilised puudused, mille tõttu lahendus ei olnud sobilik. Selle lahenduse jõudlust on hinnatud peatükis 5 Kiirustestid. Edaspidi kasutan sellele lahendusele viitamisel mõistet naiivne lahendus.

Naiivne lahendus loeb andmed kõvakettal asuvast koondtabeli failist ja töötleb neid andmeid rea kaupa. Implementatsioon on kirjutatud programmeerimiskeeles Python ja tugineb moodulitele `re` [2] ja `multiprocessing` [3]. Nendest esimene on kasutatav regulaaravaldise sõnale sobituvuse kontrolliks ja teine arvutuste paralleliseerimiseks. Programmi kasutamiseks peab olema arvutis Python 2.6 või uuem (programm ei tööta Pythoni versioonidega 3.x). Vajalikud moodulid on versiooniga 2.6 vaikimisi kaasas.

2.1 Algoritm

Algoritm on üpriski lihtne ja üldjoontes optimiseerimata. Iga sisendiks oleva regulaaravaldise kohta luuakse loend milles on iga proovi kohta üks arv algväärtusega 0. Seda loendit kasutatakse regulaaravaldisele vastavate ridade liitmise algpunktina.

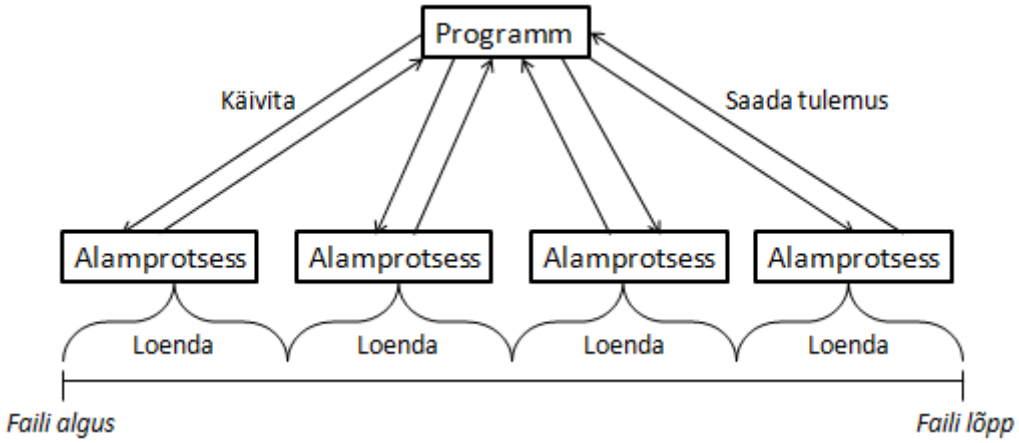
Algoritm töötleb koondtabeli faili rea kaupa, kontrollides iga rea juures, kas real olev sõna vastab mõnele sisendiks olevale regulaaravaldisele. Juhul kui vastab, siis liidetakse vastaval real olevad sagedused regulaaravaldise kohta käiva loendiga. Liitmist sooritatakse proovide kaupa. Lisaks arvestab algoritm ka sellega, et sama sõnaga võib sobituda rohkem kui üks sisendiks olev regulaaravaldis.

2.2 Paralleliseerimine

Soovitud kiiruse lootsin saavutada algoritmi paralleliseerimise abil. Peatükis 2.1 toodud algoritmi kirjeldusest saab järeldada, et igal real oleva sõna võrdlemine sisendiks olevate regulaaravaldistega on eraldiseisev ülesanne, mis ei sõltu teiste sõnade võrdlemisest. Samas regulaaravaldisele sobituvate sõnade sageduste liitmise lõpptulemus sõltub sellest, millistel ridadel olevad sõnad sobitusid. Aga kuna tegu on

liitmisega, siis ei oma tähtsust see, mis järjekorras sobituvad read leitakse ja liidetakse. Seda kasutasin ka algoritmi paralleliseerimisel ära.

Paralleliseerimise otsustasin teha selliselt, et peaksin algoritmi võimalikult vähe muutma. Selle jaoks jagab programm koondtabeli enam-vähem võrdse ridade arvuga P regiooniks, kus P on kasutatavate alamprotsesside ja ka regioonide arv. Iga regiooni kohta kutsub programm välja ühe alamprotsessi, mis loendab iga regulaaravaldise sagedused ainult selles regioonis. Kui kõik alamprotsessid on lõpetanud, siis liidan nende poolt arvutatud tulemused kokku, kasutades sama lähenemist nagu suvalisele regulaaravaldisele vastavate ridade kokku liitmisel.



Joonis 3. Naiivse lahenduse paraleliseerimine nelja alamprotsessi kasutamisel

Koondtabeli faili regioonideks jagamine ei tähenda siinkohal, et jagaksin koondtabeli kümneks eraldi failiks. Tegelikkuses ei muutu koondtabeli fail mitte kuidagi selle protsessi käigus. Selle asemel kasutan ära Pythoni failiobjekti meetodeid [4] `file.seek(n)`, `file.tell()` ja `file.readline()`.

Koondtabeli regioonideks jaotamiseks on programmil vaja teada faili suurust baitides. Selle saab kiirelt teada kasutades meetodit `os.path.getsize(file)`[5]. Failisuuruse jagan kasutatavate protsesside arvuga, mis annab mulle iga regiooni suuruse. Regiooni suuruse järgi panen paika iga regiooni algus- ja lõpppunkti, arvestades et esimese regiooni alguspunkt on 0 ja viimase lõpppunktiks on faili suurus. Lisaks kasutan ühe regiooni lõpu ja teise regiooni alguse täpseks paika panemiseks `file.readline()` meetodit, mille abil teen kindlaks, et iga regioon algab uue reaga.

Regulaaravaldise sageduste saamiseks tuleb liita kokku kõikidel sellele avaldisele vastavatel ridadel olevad sagedused proovide kaupa. Proovide kaupa liitmine tähendab seda, et kahe rea liitmisel liidan kokku esimese proovi kohta käivad sagedused, seejärel teise proovi kohta jne kuni on liidetud kõik sagedused.

2.3 Multiprocessing vs threads

Multiprocessing moodulit (nagu ka nimest võib järeldada) kasutatakse siis, kui on vaja kutsuda välja mitmeid alamprotsesse. Siinkohal võib tekkida küsimus, et miks otsustasin kasutada lõimede asemel alamprotsesse.

Nimelt ei annaks lõimede kasutamine reaalsetes oludes erilist kiiruse võitu, kuna Pythonis on kasutusel GIL (Global Interpreter Lock) [6][7]. GIL tähendab põhimõtteliselt seda, et sama Pythoni masinkoodi saab korraga jooksutada vaid üks lõim. Põhiline põhjus, miks GIL on kasutusel, on see et Pythoni mäluhaldus ei ole lõimede suhtes turvaline. Kahjuks tähendab see seda, et lõimede kasutamine antud juhul ei oleks mõistlik.

Siinkohal tulebki mängu multiprocessing moodul. Nimelt saab iga lõime asemel kasutada antud juhul alamprotsessi, mis teeb samad arvutused ja siis saadab tulemused ülemprotsessile tagasi. Sellega aga pääseme mööda GILi poolt tekitatud piirangutest ja saame antud ülesannet paralleliseerida.

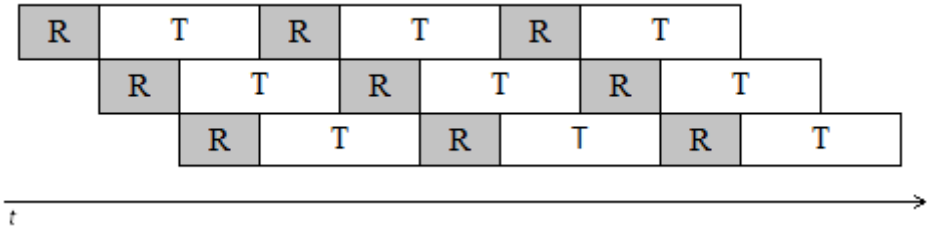
2.4 Kõvaketta pudelikael

Kuna iga regiooni töödeldakse samaaegselt ja kõik regioonid on tegelikult sama faili erinevad osad, siis võib tekkida kahtlus, et alamprotsesside selline kasutamine muudab programmi hoopiski aeglasemaks või ei anna soovitud kiiruse kasvu. Põhjenduseks oleks see, et kõvakettalt ei saa lugeda samaaegselt erinevaid andmeid. Programmi kasutamisel reaalses olukorras see siiski nii ei ole ja alamprotsesside selline kasutamine annab siiski märgatava kiirusevõidu. Vastavad mõõtmised on toodud peatükis 5 Kiirustestid.

Protsessid on kirjutatud selliselt, et üks protsess saab mingil ajahetkel, kas andmeid töödelda või neid kõvakettalt lugeda, aga ei saa teha mõlemat korraga. Seega, võib üks alamprotsess lugeda kõvakettalt andmeid samal ajal, kui teised töötlevad varem loetud

andmeid ja sellisel juhul ei sega alamprotsessid üksteist lugemise ajal. Ideaalsel juhul tähendab see, et üks protsess tegeleb alati faili lugemisega ja ülejäänud tegelevad alati loetud andmete töötlusega ning protsesside rollid vahelduvad aja jooksul.

Toon selgituseks ühe näite. Olgu kasutatavate alamprotsesside arv `N=3` ja oletame, et loetakse mitu rida korraga puhvrisse enne nende töötlemist. Olgu ühe puhvri jagu ridade lugemiseks (puhvri täitmiseks) kuluv aeg `R` ja nende ridade töötlemiseks kuluv aeg `T` (Joonis 4).



Joonis 4. Andmete lugemine ja töötlus mitme protsessiga

Antud olukorras on näha et juhul, kui ridade töötlemisele kuluv aeg `T` on võrdne või suurem kui kahekordne ridade lugemisele kuluv aeg `(N-1)*R`, siis ei jää alamprotsessid andmete lugemisel üksteise järgi ootama. Seega saavutame ideaalsel juhul peaaegu nii mitme kordse kiiruse võidu, kui mitut alamprotsessi kasutame.

Samas on ka selge, et antud lähenemine ei skaleeru väga hästi. Piisavalt suure alamprotsesside arvu korral jõuaksime ikka olukorrani, kus kõvakettalt lugemise kiirus muutuks programmi üldist kiirust aeglustavaks. Põhjuseks on siinkohal see, et suurendades protsesside arvu jõuame lõpuks punkti, kus kõik alamprotsessid ei jõua andmeid lugeda sama ajaga mis kulub ühel protsessil oma andmete töötlemiseks.

3 Baaslahendus

Soovitud kiiruse saavutamiseks on võrreldes esimese lahenduskatsega väga palju muudetud. Põhimõtteliselt on samaks jäänud vaid programmi sisend ja väljund. Põhjus on selles, et ma ei näinud võimalusi eelmise algoritmiga saavutada soovitud kiirust. Seega otsustasin kasutada ära seda, et programm hakkab jooksma masinas, kus on 256GB mälu. Sellest tulenevalt on mul võimalus kogu koondtabel mällu lugeda ja ainuüksi sellega saavutada üsna suur kiirusevõit.

Kuna sai võetud otsus, et hoian andmeid mälus, siis sellest tulenevalt võin ka andmete formaati mällu lugemise ajal muuta vastavalt sellele, mida pean efektiivsemaks. Olen seda võimalust püüdnud kasutada, et saavutada võimalikult hea loendamiskiirus. Kui oli valida mälu efektiivsuse ja arvutuskiiruse vahel, siis olen rohkem rõhku pannud arvutuskiirusele.

3.1 Koondtabel mälus

Antud ülesande lahendamisel eeldan, et reaalses olukorras on kasutada suhteliselt suure mäluhulgaga masin. Valitud lahenduse puhul on algses formaadis 193GB suuruse koondtabeli faili korral mälukasutus umbkaudu 19GB. Andmete mälus hoidmisel olen eelkõige püüdnud valida lahendused, mis lubavad algoritmi jaoks vajalikke tehteid sooritada võimalikult kiirelt. Võimalusel olen püüdnud ka mälukasutust hoida madalal, aga see on püstitatud ülesande seisukohast teisejärguline.

Mälus on koondtabel jaotatud kaheks osaks ja sarnaselt jaguneb ka programmi algoritm põhimõtteliselt kaheks osaks. Lisaks tasub veel märkimida, et koondtabel muutub väga harva. Andmete muutumisel või uute proovide lisandumisel koostatakse uus koondtabel ja seejärel laetakse see mällu. Praktikas on selle jaoks tavaliselt vajadus korra kuus või harvem. Programm ise ei muuda koondtabelit kunagi oma töö käigus, mis on mälukasutuse seisukohast oluline detail.

3.1.1 Koondtabeli sõnad

Koondtabeli sõnade mälus hoidmiseks sobiliku andmestruktuuri valimisel lähtusin sellest, et saaksin võimalikult kiirelt leida sisendiks olevale regulaaravaldisele sobituvatele sõnadele vastavad reanumbrid. Sellest tulenevalt võib valitud lahendus

tunduda esmapilgul võõras ja ebapraktiline. Samas olen kindel, et see lähenemine on ennast õigustanud.

Enne täpse kirjelduse andmist tuletan veel meelde, et koondtabelis leiduvad sõnad on kõik pikkusega 12 tähemärki ja nendes sõnades on kasutatud 20 tähelist tähestikku (täpsem kirjeldus peatükis 1 Ülesande püstitus).

Sõnade mälus hoidmiseks kasutan 240 `(12*20)` bitivektorit. Igas bitivektoris on üks element iga koondtabelis sisalduva sõna kohta. Seega on iga bitivektori pikkus võrdne koondtabelis sisalduvate sõnade arvuga. Lisaks vastab iga bitivektori esimene element koondtabeli esimesel real olevale sõnale, teine element teisel real olevale sõnale ja nii edasi.

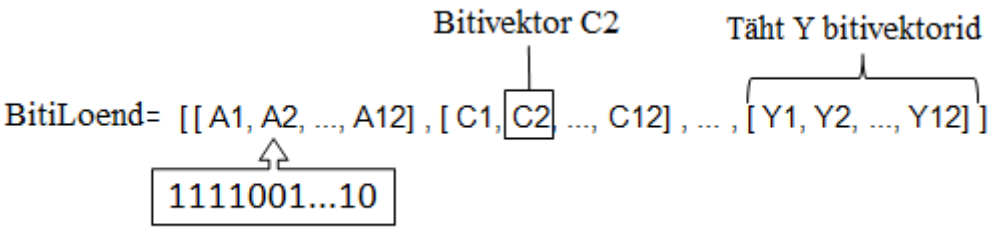
Kasutatavate bitivektorite hulk tuleneb otseselt sõnade pikkusest ja kasutatava tähestiku suurusest. Kõige lihtsam viis bitivektorite ja sisu seletamiseks on anda igale vektorile unikaalne tähis, mis iseloomustab vastava bitivektori poolt hoitavaid andmeid. Tähiste andmisel lähtun sõnade puhul kasutatavast tähestikust ja sõnade pikkusest (peatükk 1 Ülesandepüstitus). Iga bitivektori tähis koosneb ühest tähest ja ühest numbrist vahemikus 1-12

A_1, A_2 … A_{12}, C_1, C_2 … C_{12}, D_1 … Y_1, Y_2 … Y_{12}

ning iga bitivektori tähis on unikaalne.

Bitivektorite sisu seletamiseks valime näitena bitivektori A_1. Bitivektor koosneb suurest hulgast 1 ja 0 (`True` ja `False`) väärtustest. Näitena võetud tähisest (A_1) võime välja lugeda, et selles konkreetses bitivektoris sisalduvad väärtused käivad iga sõna esimesel positsioonil oleva tähe A kohta (kas vastavas sõnas on esimesel positsioonil täht A või mitte). Sama kehtib ka kõigi ülejäänud bitivektorite kohta. Kui näiteks G_{10} esimene element on 1 siis see tähendab, et koondtabeli esimesel real oleva sõna kümnes täht on G. Kui G_{10} teine element on 0 siis see tähendab, et teise sõna kümnes täht ei ole G.

Need bitivektorid olen omakorda paigutanud kahemõõtmelisse loendisse, mida kutsun edaspidi läbivalt nimega bitiLoend. Iga bitiLoendi alamloend hoiab endas ühe tähe kohta käivaid bitivektoreid selliselt, et esimene alamloendis olev bitivektor vastab sõna esimesele positsioonile, teine vastab teisele positsioonile jne.



Joonis 5. Bitivektoreid sisaldava andmestruktuuri bitiLoend ülesehitus

Joonisel 5 on toodud selgitav pilt bitiLoend struktuuri kohta, mille järgi võin viidata bitivektorile C2 kasutades koodis `bitiLoend[1][1]`. Lisaks võime lugeda välja, et koondtabeli esimese sõna teine täht on A, samas kui viimase sõna teine täht ei ole A.

Bitivektoritega töötamiseks kasutan Pythoni moodulit `bitarray 0.8.0`. Bitivektoritega sooritatavate tehete kirjeldus on toodud peatükis 3.2.1 Regulaaravaldisele vastavate ridade leidmine.

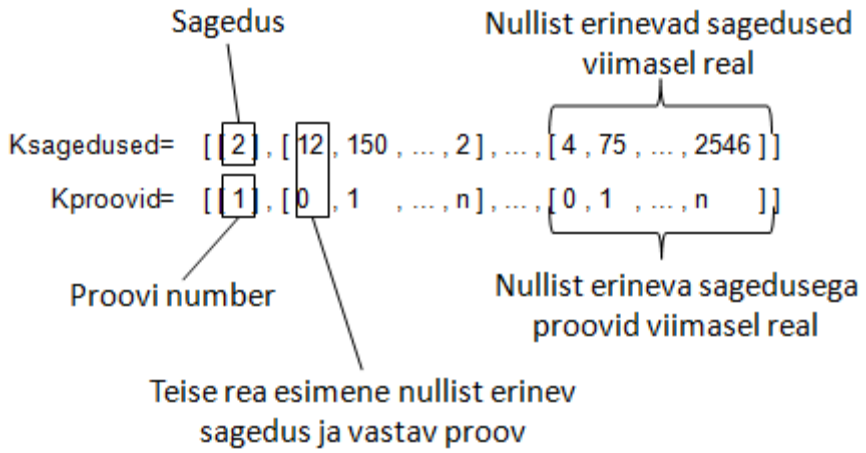
3.1.2 Koondtabeli sagedused

On teada, et koondtabelis on valdava enamiku sageduste väärtuseks 0. Seega on tegu hõreda maatriksiga. Hõredate maatriksite efektiivseks hoidmiseks on mitmeid erinevaid võimalusi nagu näiteks võtmete sõnastik (Dictionary of keys), loend loenditest (List of lists) [8] või koordinaatide loend (Coordinate list) [9].

Ülaltoodutest ei tundunud mulle ükski ideaalne ja seega otsustasin ise implementeerida mõnevõrra teistsuguse lähenemise, mis pakub mulle võimaluse kiirelt ridu kokku liita. Seda lahendust võib mõnes mõttes vaadata, kui kolme ülaltoodu segu.

Sageduste hoidmiseks kasutan kahte kahemõõtmelist loendit, mida kasutan paralleelselt ridade kokku liitmise etapis. Esimeses nendest kahest loendist hoian kõiki nullist erinevaid sagedusi ja teises nendele sagedustele vastavate indeksite numbreid. Edaspidi kutsun neid loendeid vastavalt Ksagedused ja Kproovid (eesliide K on tuletatud sõnast koondtabel).

Siinkohal on veel oluline märkida, et nii loendis Ksagedused kui ka loendis Kproovid vastab esimene alamloend koondtabeli esimesele reale. Proovide nummerdamine loendis Kproovid algab numbrist 0.



Joonis 6. Koondtabeli sageduste hoidmiseks kasutatavad loendid. Toodud arvud vastavad joonisele 1

Sageduste hoidmiseks kasutatavate loendite struktuuri selgitamiseks on toodud joonis 6. Selle joonise põhjal saame järeldada, et koondtabeli esimesel real on vaid üks nullist erinev sagedus. Selle sageduse väärtuseks on 2 (Ksagedused esimese alamloendi ainus element) ja tegu on koondtabeli teise prooviga (Kproovid esimese alamloendi ainus element on 1). Sarnaselt näeme, et koondtabeli teisel real on esimene nullist erinev sagedus esimeses proovis, selleks sageduseks on 12 ja teine nullist erinev sagedus on 150 ning leidub teises proovis.

Ehk, kui on vaja leida real 356 olevad nullist erinevad sagedused, siis selleks tuleb vaadelda loendeid Ksagedused ja Kproovid kohal `356-1=355`. Sellel positsioonil asuvaid alamloendeid (nimetame näite jooksul Ksagedused356 ja Kproovid356) tuleb vaadelda koos. Esimene number loendis Ksagedused356 näitab esimest nullist erinevat sagedust real 356 ja esimene number loendis Kproovid356 näitab mitmenda proovi kohta see sagedus käib.

See lahendus annab võimaluse suhteliselt kiirelt ja kerge vaevaga liita kokku valitud ridadel leiduvad sagedused proovide kaupa. See aga on väga oluline, et suudaksin saavutada loodetud loendamiskiiruse. Lisaks on tegu piisavalt mäluefektiivse viisiga andmete hoidmiseks. Liitmise kirjeldus on toodud peatükis 3.2.2 Sageduste liitmine.

3.2 Algoritm

Algoritmi olen endale lihtsuse mõttes jaganud kaheks osaks, sarnaselt koondtabeli mälus hoidmisele. Mõlemat algoritmi osa võib vaadelda kui eraldiseisvat ülesannet. Esimene osa algoritmist kasutab bitivektoreid ja teine osa sageduste loendeid. Sellise jaotuse tegin algselt selleks, et endale ülesannet lihtsustada ja et saaksin vajadusel ka mõlemat osa eraldi testida. Lisaks teeb sellise jaotuse kasutamine terve algoritmi selgitamise kergemaks.

Algoritm võtab sisendiks ühe regulaaravaldise korraga ja arvutab selle esinemissagedused. Iga regulaaravaldise jaoks täidetakse algoritm uuesti. See tähendab, et kui sisendiks on 100 regulaaravaldist siis läbitakse algoritm ka 100 korda.

Esimene algoritmi osa leiab kõik koondtabeli reanumbrid, millel olev sõna vastab sisendiks antud regulaaravaldisele. Nendest reanumbritest moodustatud loend on algoritmi teise osa sisendiks. Algoritmi teine osa liidab antud reanumbritel olevad sagedused indeksite kaupa kokku. Seejärel on ühe regulaaravaldise tulemus arvutatud ja valmis väljundisse kirjutamiseks.

3.2.1 Regulaaravaldisele vastavate ridade leidmine

Lisaks peatükis 3.1.1 Koondtabeli sõnad toodule kasutan ära ka seda, et bitivektoritega saab sooritada loogilisi tehteid. Algoritmis kasutan `&` ja `|` tehted. Nende tehete abil on võimalik igale sisendiks olevale piiratud võimsusega regulaaravaldisele vastavaid sõnu sisaldavate reanumbrite leidmine.

Parimal juhul ei ole selleks vaja ühtegi tehet sooritada (näiteks regulaaravaldise `K..........` korral on vastuseks bitivektor K_1) ja halvimal juhul on vaja sooritada 41 tehet. Näiteks regulaaravaldise `AKPYIGG` korral tuleb sooritada 6 `&` tehet iga positsiooni kohta, kus see regulaaravaldis võib sõnadega sobituda. Neid positsioone on kokku 6. Lisaks tuleb iga positsiooni kohta saadud tulemustega sooritada `|` tehted. Kuue positsiooni korral on neid tehteid vaja sooritada 5. Seega saame kokku sooritatavate tehete arvuks `6*6+5=41`.

Kui regulaaravaldisele lisada mõni punkt, siis see vähendab võimalike positsioonide arvu ja kui mõni täht asendada punktiga, siis see vähendab sooritatavate `&` tehete arvu. Sarnase arvutuskäigu tulemuseks saame, et kuue tähemärgi pikkuse regulaaravaldise

korral tuleb sooritada sama arv tehteid (5*7+6=41). Alla kuue tähemärgi ja üle seitsme tähemärgi pikkuste regulaaravaldiste korral jääb vajalike tehete arv juba väiksemaks.

Selgitame tehete sooritamist regulaaravaldise A.N.T näitel. Selle regulaaravaldisega sobivad kõik sõnad, millede esimene täht on A, kolmas täht N ja viies täht T. Seega saame sooritada bitivektoritega järgmise tehte $A_1\&N_3\&T_5$. Kui seejärel otsida tehte tulemuseks olnud bitivektorist kõikide tõeste bitide (väärtusega 1) indeksid, siis saame reanumbrid, kus sõna esimene täht on A, kolmas N ja viies T.

Siinkohal on aga oluline meeles pidada, et regulaaravaldis võib sõnaga sobituda mitmel erineval positsioonil. Näiteks A.N.T sobitub ka sõnadega, kus teine täht on A, neljas N ja kuues T. Selliseid sõnu ei oleks me eelmise tehtega tõenäoliselt leidnud. Et leida lisaks ka need sõnad, siis tuleb sooritada tehe $(A_1\&N_3\&T_5)|(A_2\&N_4\&T_6)$.

Ka eelnev tehe ei pruugi leida kõiki sõnu, mis sobituvad antud regulaaravaldisega. Selleks, et leida kõik sõnad, on meil vaja teada mitmel kohal regulaaravaldis üldse saab sõnale sobituda. Selle saame arvutada järgmise tehtega 12-x+1 (ehk 13-x), kus x on regulaaravaldises leiduvate tähemärkide arv, 12 on sõna pikkus ja ühe liidame sest 12 tähemärgi pikkune regulaaravaldis saab sobituda iga sõnaga ühel positsioonil.

Seega regulaaravaldis A.N.T saab sobituda 12-5+1=8 erineval positsioonil. Kõikide sõnade leidmiseks peame järelikult sooritama kaheksa & tehete komplekti. Nende tehete tulemused tuleb seejärel taandada üheks bitivektoriks kasutades selleks | tehteid. Kokku saame tehte

(A1&N3&T5)|(A2&N4&T6)|(A3&N5&T7)|(A4&N6&T8)|(A5&N7&T9)|(A6&N8&T10)| (A7&N9&T11)|(A8&N10&T12).

Kui selle tehte tulemuseks olevast bitivektorist leida kõikide bittide (väärtusega 1) indeksid, siis saamegi kõik reanumbrid, mis vastavad regulaaravaldisele A.N.T.

Eelnev kirjeldus näitab üldist põhimõtet, mille tuginedes on algoritmi esimene implementeeritud. Kiirema ridade leidmise saavutamiseks olen implementeerimise käigus algoritmi mõnevõrra muutnud.

Algoritmi kirjelduse kokkuvõttena võib öelda, et algoritm töötab vastavalt hetkel sisendiks olevale regulaaravaldisele. Iga regulaaravaldises leiduva tähe kohta võetakse

vastav bitivektor, nende bitivektoritega sooritatakse & tehe, mille tulemusena saadud bitivektoris vastab iga tõese väärtuse indeks regulaaravaldisele sobituvat sõna sisaldavale koondtabeli reanumbrile. Lisaks arvestatakse sellega, et regulaaravaldis võib sõnaga sobituda mitmel positsioonil. Selle arvestamiseks leitakse iga võimaliku positsiooni korral sobivad reanumbrid bitivektorina. Neid bitivektoreid on üldjuhul mitu ja lõpliku tulemuse saame sooritades nende bitivektoritega | tehted. Nende tehete tulemusest leiame kõigi tõeste väärtuste indeksid, milledest koosnev loend on algoritmi väljundiks.

3.2.2 Sageduste liitmine

Koondtabeli sageduste hoidmiseks kasutan kahte kahemõõtmelist loendit Kproovid ja Ksagedused ning sealjuures on mõlema elementide järjekord vastavuses bittide järjekorraga bitivektorites (peatükk 3.1.2 Koondtabeli sagedused).

Sageduste liitmine toimub algoritmi esimeses osas leitud reanumbrite loendi põhjal. Reanumbreid saan kasutada Kproovid ja Ksagedused loendis olevatele alamloenditele viitamiseks, ehk reanumbrite põhjal tean, milliseid alamliste tuleb liitmisel kasutada. Kproovid loendis olevate alamloendite sisu põhjal tean milliste indeksite sagedustele liita vastavad Ksagedused loendi alamloendites olevad arvud.

Liitmise alguses loon kõigepealt ajutise listi kus on iga proovi kohta üks arv mille väärtuseks on 0. Seejärel läbin liidetavate reanumbrite loendi ja liidan igal liidetaval real olevad nullist erinevad sagedused ajutisese loendisse. Liitmisel arvestan ka seda millise proovi juurde iga sagedus kuulub.

3.3 Paralleliseerimine

Teisel katsel loodud programmiga saavutasin piisava kiiruse väikesele kogusele regulaaravaldisele vastavate sageduste leidmiseks korraga leimiseks ilma, et peaksin kasutama mitut protsessori tuuma (vastavad testid on toodud peatükis 5 Kiirustestid). Sellest tulenevalt ei pidanud ka vajalikuks neid arvutusi paralleliseerida, kuigi see oleks võimalik ja annaks kindlasti mõningase kiirusekasvu.

Programmi sisendiks on sageli 1000-10000 regulaaravaldist korraga ning on võimalik, et tulevikus isegi rohkem. Kui me eeldame, et iga regulaaravaldise kohta ajakulukulu 1,5 s siis 10000 regulaaravaldise korral kuluks natukene üle nelja tunni. See aga on juba

üsna pikk aeg tulemuste saamiseks. Sellest tulenevalt otsustasin paraliseerimise kohal minna lihtsama, aga võimalik et ka efektiivsema vastupanu teed.

Siinkohal on oluline märkida, et iga regulaaravaldise sageduste arvutamist võib vaadata, kui eraldiseisvat ülesannet. Pean silmas seda, et ühe regulaaravaldise sageduste arvutamine ei sõltu ühegi teise sageduste arvutamisest. Oluline on vaid see et väljundis esineksid regulaaravaldised samas järjekorras nagu sisendis.

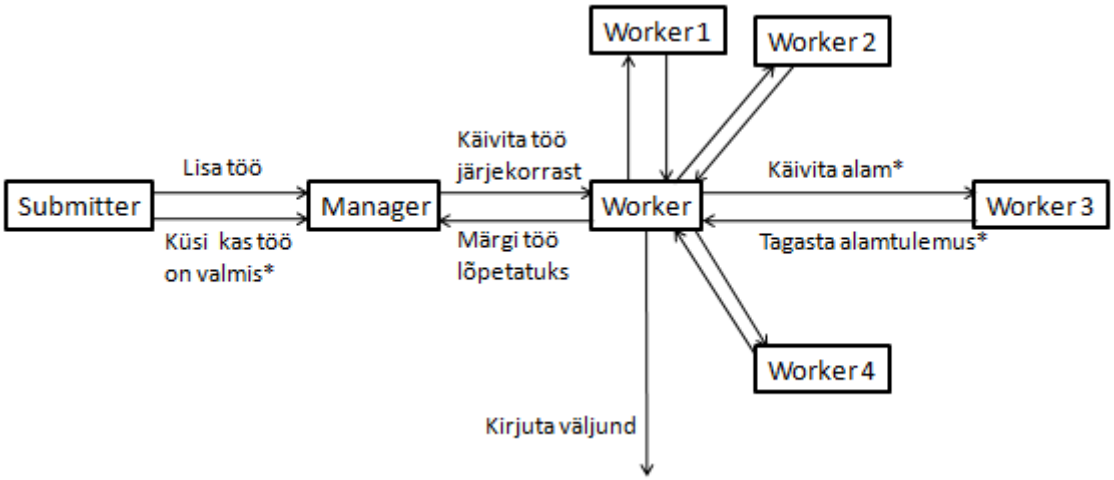
Oletame, et kasutatavate tuumade arvuks olen ma oma programmis määranud neli. Sellisel juhul jagan ka sisendiks olevad regulaaravaldised neljaks osaks. Kui jagamist ei ole võimalik täpselt teha, siis on viimane osa kõige suurem. Näiteks, kui sisendiks on 402 regulaaravaldist, siis esimese kolme osa suuruseks on 100 ja viimase suuruseks 102. Lisaks on jagamine teostatud selliselt et osade järjest väljastamisel saaksime sama regulaaravaldiste järjekorra kui sisendis.

Iga osa kohta loon eraldiseisva alamprotsessi, mis saab lugeda ülemprotsessi poolt mälus hoitavat koondtabelit. Igale nendest alamprotsessidest annan ette ühe eelnevalt loodud osadest. Seejärel loendab iga alamprotsess just temale antud regulaaravaldiste sagedused ja saadab loendustulemused lõpuks ülemprotsessile, mis kirjutab tulemused väljundfaili.

3.4 Programmi mudel

Selleks, et algoritmi reaalses olukorras kasutada, on vaja rohkemat kui lihtsalt andmete mällu lugemise ja algoritmi realiseerimine. On vajalik hallata tööde järjekorda, võimalust deemon protsessiga suhtlemiseks loendamise ajal, vajadusel töö lõppemiseni ootamine jne.

Nende ülesannete täitmiseks koosneb loodud programm kolmest komponendist: Manager, Worker ja Submitter. Nendel komponentidel on oma ülesanded, mis ei kattu teiste komponendi ülesannetega (joonis 7).



Joonis 7. Baaslahenduse komponentide suhtluse mudel (tärniga osad täidetakse vajadusel)

3.4.1 Manager

Manager on programmi keskne osa, millel on kõige rohkem erinevaid ülesandeid. Tegu on põhimõtteliselt deemoniga, mille käivitan hetkel kasutades Linuxi `nohup` [10] käsku. Manageri kirjutamisel olen kasutanud `Pyro4` [11] moodulit, mis lubab koos `nohup` käsu kasutamisega ülesse seada suhteliselt kergelt deemon protsesse ning nendega suhelda.

Käivitamisel alustab manager esimese asjana koondtabeli mällu laadimist. See protsess võtab üsna kaua aega (umbkaudu 5 tundi). Selle aja jooksul ei ole võimalik Manageriga praeguses implementatsioonis suhelda. Manager teeb ennast `Pyro4` abil nähtavaks peale seda, kui koondtabel on mällu laetud ja jääb ühendusi ootama. Manager ei ole välisvõrgust kättesaadav, kuna esiteks see pole hetkel vajalik ja teiseks oleks tegu turvariskiga.

Managerilt on võimalik küsida tervet tööde järjekorda, aktiivsete tööde arvu, kustutada töid, lisada töid jne. Praegu on Submitteris nendest kasutuses uue töö lisamine ja lisatud töö staatuse kontrollimine.

Töö lisamisel kontrollib Manager esiteks, kas antud töö on kohe võimalik käima panna. Juhul kui see on võimalik, siis märgitakse töö käivitatuks ja harutatakse [12] (fork) alamprotsess (Worker). Seejärel jääb Manager uusi ühendusi ootama. Kui Worker lõpetab loendustöö ja märgib töö lõpetatuks, siis manager kontrollib, kas on järjekorras

mõnda tööd mida saaks käivitada. Kui on, siis käivitab, kui ei, siis jääb uusi ühendusi ootama.

3.4.2 Worker

Worker on, nagu nimigi viitab, programmi tööline ehk siis komponent, mis tegeleb regulaaravaldiste loendamise tööga. See tähendab, et Workeri komponendis on implementeeritud nii algoritm ise, kui ka loenduse tulemusfaili kirjutamine. Lisaks jagab worker vajadusel regulaaravaldiste nimekirja osadeks ja kutsub välja alamtöölised (Sub-Worker).

Worker ei ole reaalsuses eraldisesisev programm. Tegu on põhimõtteliselt Pythoni mooduliga, mida kasutan eelkõige koodi kapseldamise eesmärgil. Worker käivitatakse Manageri poolt loendusülesande alustamisel. Selleks harutatakse alamprotsess, mis alustab loendustööd. Peale Workeri käivitamist jääb tema ülemprotsess (Manager) uusi ühendusi ootama. Alamprotsess (Worker) suhtleb Manageriga sama moodi nagu Submitter. Erandiks on see, et Worker saab Manageri poolt mälus olevaid andmestruktuure vabalt lugeda (sh tööde järjekord ja parameetrid) tänu sellele et Linux kasutab protsesside harutamisel efektiivsemaks mälu haldamiseks kopeeri-kirjutamisel metoodikat (copy-on-write) [13].

Kui sisendiks olevas regulaaravaldiste failis on rohkem kui neli regulaaravaldist, siis käivitab Worker neli alamprotsessi. Need neli käivitatakse kasutades Pythoni `multiprocessing` moodulit. Iga nendest neljast saab teatud osa regulaaravaldistest, mille sagedusi hakkab loendama (nagu Paralleliseerimine peatükis kirjeldatud). Peale iga regulaaravaldise loendamist saadetakse vastavad sagedused Workerile tagasi, kasutades `multiprocessing.Pipe(False)` objekte. Worker saab omakorda neid juba jooksvalt väljundisse kirjutama hakata.

Peale loendustöö lõpetamist ja väljundi kirjutamist suhtleb Worker Manageriga. Worker märgib enda poolt sooritatud töö lõpetatuks. Juhul kui, töö sisestamisel submitteri kaudu on antud `--noWait` lipp siis Worker ka eemaldab selle töö tööde nimekirjast. Peale seda Worker väljub. Väljumisega on baaslahenduses üks parandamata puudus, mis on kirjeldatud täpsemalt peatükis 3.5 Baaslahenduse puudused.

3.4.3 Submitter

Submitter on ainus komponent mida kasutajal läheb tavaoludes vaja. Tegu on programmiga mille abil saab uusi töid järjekorda lisada ja oodata nende tööde lõppemist. Selleks otstarbeks on programmil kolm parameetrit, mis on toodud tabelis 2.

Parameeter	Pikk nimi	Kirjeldus
-o	--output	Failitee ja nimi kuhu loendustulemus kirjutatakse
-m	--motifList	Fail, mis sisaldab reavahetusega eraldatud regulaaravaldisi
	--noWait	Programm ei oota töö lõppemist vaid väljub kohe kui töö on lisatud

Tabel 2. Baaslahenduse submitteri parameetrid ja nende selgitused

Submitter loeb regulaaravaldisi sisaldava faili läbi ja koostab nendest avaldistest loendi. See loend saadetakse koos noWait lipu ja output väärtusega managerile. Edasine tegevus oleneb noWait lipu väärtusest. Juhul, kui see on True, siis programm väljub kohe peale seda, kui töö on lisatud.

Juhul, kui noWait lipu väärtuseks on `False` (mis on ühtlasi ka vaikeväärtus), siis programm hakkab iga 60 sekundi järel Managerilt lisatud töö staatust küsima. Sellisel juhul väljub programm alles siis, kui Managerilt tuleb vastus, et töö on valmis. Selline ootamine on vajalik, et Submitterit saaks kasutada näiteks shell skripti osana.

3.5 Baaslahenduse puudused

Baaslahenduse tegemise käigus ilmnesid mõned probleemid. Esiteks tekkis kahtlus, et Pythonist tulenevalt kaotan ma kiiruses ja sama algoritmi implementeerimine C++ keeles annaks kiirema lõpptulemuse. Teiseks tegin programmi kirjutamisel ise vea, mis tekitas zombie [14] protsesse. Sellest veast ei oleks küll väga keeruline vabaneda, aga sai võetud otsus et kirjutan kogu programmi C++ keeles ümber. Peamist põhjust, miks selline otsus sai võetud, kirjeldan põhjalikumalt peatükis 3.5.1 Mäluprobleemid.

3.5.1 Mäluprobleemid

Teine lahendus sai kirjutatud selliselt, et oleks võimalik saada kasu Linuxi süsteemis kasutatavast kopeeri-kirjutamisel lähenemisest. Kopeeri-kirjutamisel tähendab põhimõtteliselt antud kontekstis seda, et iga harutatud alamprotsess saab vabalt lugeda kõiki ülemprotsessi poolt mälus hoitavaid andmeid, ilma et need andmed kopeeritaks. Kui mõni protsess muudab neid andmeid siis kopeeritakse terve mälu lehekülg selle protsessi jaoks [15].

Näitena võib tuua sellise olukorra, kus programm loeb 10GB andmeid mällu ja seejärel harutab 5 alamprotsessi. Iga alamprotsess saab mälus olevaid andmeid lugeda. Kui ükski kuuest protsessist ei muuda neid andmeid, siis võtab programm kokku umbes 10GB mälu. Kui iga alamprotsess muudab neid andmeid siis võtab programm kokku halvimal juhul üle 60GB mälusest andmetest tehakse iga alamprotsessi jaoks koopia. Seejuures on koopiad eraldiseisvad, ehk üks alamprotsess ei näe ühegi teise alamprotsessi (ega ka ülemprotsessi) poolt andmetes tehtud muudatusi.

Teise lahenduse kirjutamise alguses teadsin, et koondtabel võtab mälus umbes 15GB. Tööde järjekorra, loendamistöö ajal vajalike lisaandmete jms mälus hoidmiseks kulub selle kõrval marginaalne mäluhulk, mis on suure tõenäosusega alla 10MB. Lisaks arvestasin sellega, et peale koondtabeli mällu lugemist pole enam vajalik vastavaid andmestruktuure muuta, mis lubaks teoreetiliselt kasutada ära kopeeri-kirjutamisel lähenemist, et hoida mälukasutust madalal.

Kahjuks see aga Pythoni programmeerimiskeeles nii ei toimi ning ma sain sellele jälile alles siis, kui programm oli esimest päeva reaalselt kasutuses. Põhimõtteliselt võib öelda, et Pythoni programmeerimiskeeles saab kopeeri-kirjutamisel lähenemisest hoopiski kopeeri-ligipääsul, mis aga omakorda tähendab mitmeid kordi suuremat mälukasutust, kui baaslahenduse kirjutamisel eeldasin [16]. Näiteks ühe 100 regulaaravaldist sisaldava loendustöö jaoks oleks vaja praeguses implementatsioonis 6*16GB mälu (Manager, Worker ja neli alam-Workerit).

Põhjus tuleneb sellest, et iga Pythoni objektiga koos hoitakse ka vastavat refcounti. Refcount näitab seda, mitu viidet vastavale objektile parasjagu eksisteerib [17]. See arv võib muutuda programmi täitmise ajal väga tihti. Lisaks uue viite koodis lisamisele (mida saaks võib-olla vältida) võib refcount muutuda ajutiselt ka lihtsamate koodiridade täitmisel. Kahjuks tähendab refcounti muutmine kopeeri-kirjutamisel lähenemise seisukohast sama, kui vastava objekti andmete muutmine. Ehk siis kopeeri-kirjutamisel muutubki kopeeri-ligipääsul (copy-on-access) lähenemiseks.

Näitena võib vaadata järgmist koodi.

```
from bitarray import bitarray
from array import array

list1=[bitarray('00011'), bitarray('00010'), bitarray('11011')]
list2=[array('i',[0,0,0,0]),array('i',[1,1,1,1]),array('i',[2,2,2,2])]
```

```
def calculate(l1,l2):
       result1=l1[0]&l1[1]&l1[2]
       result2=l2[0][0]+l2[1][1]+l2[2][2]
return result1, result2

print calculate(list1,list2)
```

Selle koodi käivitamine muudab objektide list1 ja list2 refcounte print lause täitmisel. Isegi tehe `l1[0]&l1[1]&l1[2]` muudab vastavate objektide refcounte ja seega kopeeritaks seotud andmed mälus (kuigi andmed ise ei ole muutunud).

4 Lõpplahendus

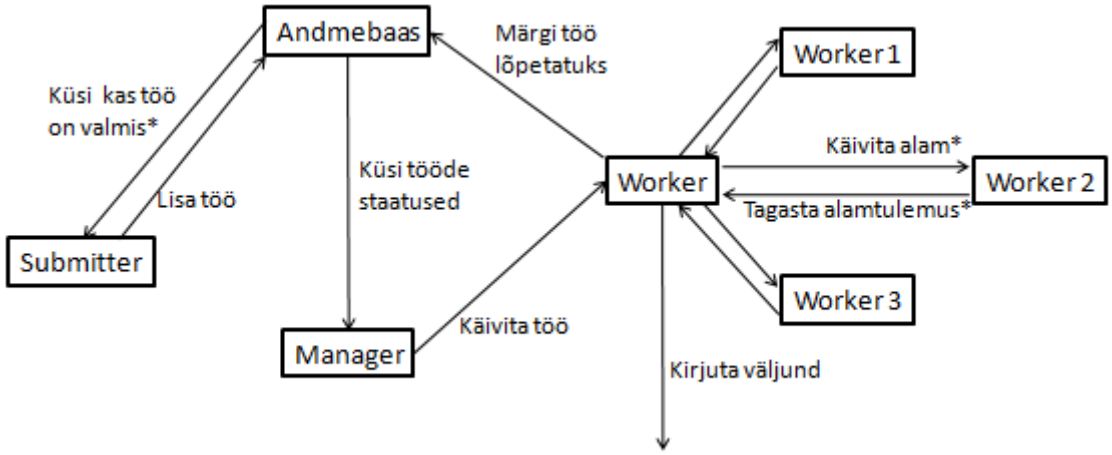
Lõplik versioon programmist on kirjutatud C++ keeles. Lisaks zombie protsesside ja mälukasutuse probleemide lahendamisele on programm ka kiirem. Kasutanud olen valdavas osas standardteegi poolt pakutavaid võimalusi. Programmis kasutatav algoritm on sama, mis pythoni versioonis, aga programmi mudel on mõnevõrra muutunud. Tööde järjekorda hoian sqlite3[18] andmebaasis, Workeri funktsionaalsus on kirjutatud manageri juurde ja kasutan harutamist palju rohkem. Järgnevalt kirjeldan muudatused sügavuti vastavates peatükkides.

Mäluprobleemi lahendamise kohta ei ole eraldi peatükki toodud, kuna see probleem lahenes C++ keele peale üleminekuga. Nimelt ei ole selles keeles taolist viitade loendamist nagu Pythonis.

4.1 Programmi mudel

Programmi mudel on üsna sarnane baaslahenduses implementeeritule. Mudeli kirjeldamisel kasutan lihtsuse mõttes terminit Worker hoolimata sellest, et Workeri funktsionaalsus on Manageri osana kirjutatud. Nimelt Workeri koodi osa täitmiseks harutatakse uus protsess, mis väljub peale lõpetamist ja Manager ei täida seda koodiosa kunagi.

Põhiliseks erinevuseks võrreldes baaslahendusega on see, et erinevad komponendid ei suhtle omavahel enam otse vaid suhtlus käib kaudselt läbi sqlite andmebaasi. Erandiks on siinkohal alam-Workerid, mis kasutavad oma ülemaga suhtlemiseks Linuxi süsteemi torusid (pipe)[19][20]. Lõpplahenduse mudel on toodud joonisel 8.



Joonis 8. Lõpplahenduse komponentide vahelise suhtluse mudel (tärniga osad täidetakse vajadusel)

4.2 Andmebaas

Tööde järjekorra hoidmiseks kasutan lihtsat sqlite andmebaasi. Andmebaas sisaldab vaid ühte tabelit workList. Nii andmebaasi kui ka vajaliku tabeli olemasolu kontrollib Manager igal käivitamisel. Juhul kui andmebaasi või workList tabelit ei eksisteeri, siis püüab manager need ise luua. See tähendab, et kasutaja ei pea andmebaasi loomiseks ise vaeva nägema eeldusel, et sqlite on süsteemis installeeritud. Andmebaasi kasutamine tähendab seda, et kasutajal on võimalik töid järjekorda lisada isegi siis kui Manager ei käi või ei ole andmeid mällu laadinud (eeldusel et vajalik andmebaas on koos korrektse workList tabeliga loodud). Tabel `workList` hoiab töö identifikaatorit, kahte töö kohta käivat parameetrit ja staatust (Tabel 3). Tabeli `workList` loomiseks kasutan käsku

```
CREATE TABLE IF NOT EXISTS workList (
      workID INTEGER PRIMARY KEY AUTOINCREMENT,
      status TEXT,
      motifFilePath TEXT,
      outFilePath TEXT
).
```

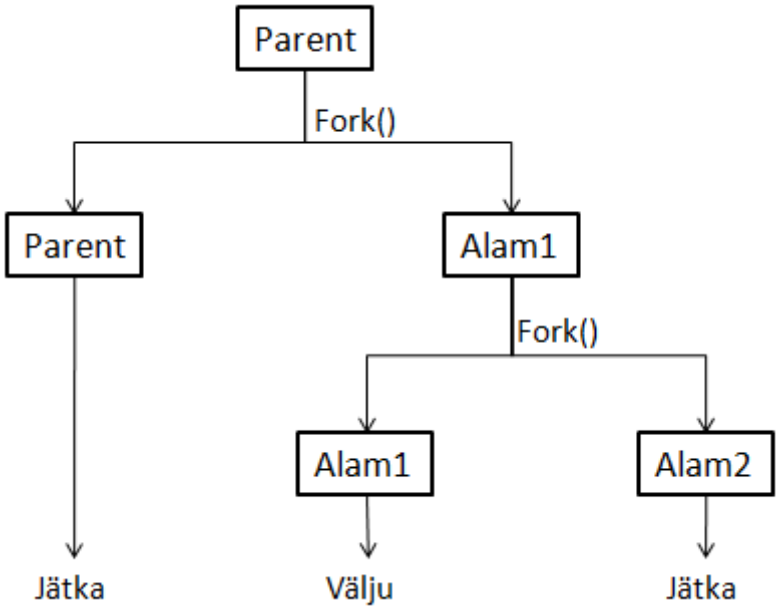
Staatus	Selgitus
Waiting	Töö on andmebaasi lisatud aga ei ole veel käivitatud
Running	Töö on käivitatud
Finished	Töö on lõppenud edukalt
Error	Töö on lõppenud veaga

Tabel 3. Võimalikud töö staatused sqlite andmebaasis koos selgitustega

4.3 Topelt harutamine

Tavaoludes peaks ülemprotsess ise harutatud alamprotsesside järgi ootama ja need nö kokku korjama ja sellega eemaldama vastavad kirjed protsesside tabelist. See oleks sobiv lahendus juhul, kui lubaksin samaaegselt vaid ühe töö jooksutamist. Reaalselt olen programmi aga implementeerinud nii, et samaaegselt võib jooksta rohkem kui üks töö ja sellest tulenevalt ei sobi ka lahendus, kus manager ootab oma alamprotsesside järgi. Lisaks ei oleks ka mingit muud põhjust miks ta peaks ootama, kuna iga töö valmimise info kirjutatakse andmebaasi ja väljundi kirjutamise sooritab Worker.

Lahenduseks on nö topelt harutamine, mis teeb igast alaprotsessist põhimõtteliselt eraldiseisva protsessi, mille kirje eemaldab operatsiooni süsteem ise protsesside tabelist peale seda kui vastav protsess on oma töö lõpetanud [21] [22]. Topelt harutamine tähendab lühidalt öeldes seda, et ülemprotsess harutab ühe alamprotsessi (nimetame Alam1) ja jätkab oma tööd. Alam1 harutab seejärel veel ühe alamprotsessi (nimetame Alam2) ja seejärel väljub. Selle tulemusena ei ole enam protsessil Alam2 otsest ülemat (kuna Alam1 väljus) ja sellest tulenevalt on Linuxi init protsessi kohus eemaldada vastav kirje protsesside tabelist kui Alam2 väljub. Kirjeldatu on toodud joonisel 9.



Joonis 9. Topeltharutamise selgitus

4.4 Programmi töövoog

Selles peatükis annan näitliku ülevaate sellest kuidas käitub programm alates käivitamisest kuni ühe loendustöö lõpetamiseni. Näide on toodud järjestikku käivate sammude kaupa.

Manager

1. Kasutaja käivitab Manageri käsuga nohup ./MotSum_manager > MotSum_manager.log 2>&1& (Manageri poolt `stdout` ja `stderr` väljunditese kirjutatav suunatakse faili MotSum_manager.log)
2. Manager kontrollib andmebaasi olemasolu ja loob selle vajadusel
3. Manager laeb koondtabeli mällu
4. Manager kontrollib iga minuti tagant kas andmebaasi on mõni töö lisatud.
5. Kasutaja lisab andmebaasi töö kasutades Submitterit
6. Manager leiab andmebaasist lisatud töö ja küsib selle parameetrid
7. Manager kontrollib kas jooksvate tööde arv on väiksem kui maksimaalselt lubatud
8. Manager käivitab workeri sooritades topelt harutamise
9. Manager jääb andmebaasist uusi töid kontrollima

Worker

1. Worker määrab käivitatud töö staatuseks “Running”
2. Worker loeb sisendiks oleva motiivide faili ja salvestab motiivid loendi
3. Worker jagab loendi nii mitmeks osaks, kui on lubatud protsesside arv
4. Worker harutab iga osa kohta alamprotsessi (sub-Worker)
5. Iga sub-Worker loeb oma osas olevate motiivide sagedused ja saadab loenduse tulemus Workerile
6. Worker loeb sub-Workerite tulemused
7. Worker kirjutab väljundfaili
8. Worker märgib töö staatuseks “Finished”
9. Worker väljub

5 Kiirustestid

Ülesande püstituse juures üheks tähtsaimaks kriteeriumiks on loodud algoritmi implementatsiooni kiirus. Käesolevas peatükis toon kiiruse mõõtmised kõigi kolme programmi kohta reaalsetes oludes. Kiirused on mõõdetud loendustöö algusest kuni tulemuse kirjutamiseni. See tähendab, et sisse pole arvestatud ajalisi viiteid, mis kuluvad regulaarselt töö staatuse kontrollimisele. Näiteks kolmanda lahenduskatse juures kulub töö sisestamisest kuni töö käivitamiseni kuni 60 sekundit.

Kõik kolm programmi on testitud 10 ja 100 regulaaravaldise suuruste töödega kasutades ühte alamprotsessi loendustöö kohta. Lisaks on kõik kolm programmi testitud ka kasutades kümmet alamprotsessi loendustöö kohta. Kümne alamprotsessiga loendused on testitud 10, 100, 1000 ja 10000 (ajapuudusel pole baaslahendust 10000 regulaaravaldisega testitud) regulaaravaldise suuruste loendustöödega. Nende testide tulemused näitavad ühest küljest seda kui hästi loodud programmid skaleeruvad mitmete tuumade kasutusele võtmisel ja teisest küljest programmide omavahelist kiiruseerinevust. Kõik toodud testid on jooksutatud viis korda ja iga testi juures on kasutatud samu sisendandmeid. Testide tulemused on toodud graafikutel 1 ja 2. Graafikutel olev ajakulu telg on antud logaritmilises skaalas.

Kõik testid on jooksutatud samas masinas, millel on 256GB mälu ja 32 protsessori tuuma. Seega on testimiseks kasutataval masinal piisavalt ressursse programmide jooksutamiseks. Samas testi tulemuste analüüsimisel tuleb silmas pidada, et seda masinat võivad samaaegselt kasutada mitmed erinevad kasutajad muude ressursimahukate programmide jooksutamiseks. See omakorda mõjutab ka käesoleva bakalaureusetöö raames arendatud programmide testimise tulemusi. Testide sooritamise ajal jooksis kasutatavas masinas paar muud ressursimahukat tööd ja sellet tulenevalt oleks ideaaloludes saadud ajad mõnevõrra madalamad.

5.1 Programmide kiirus ühe tuuma kasutamisel

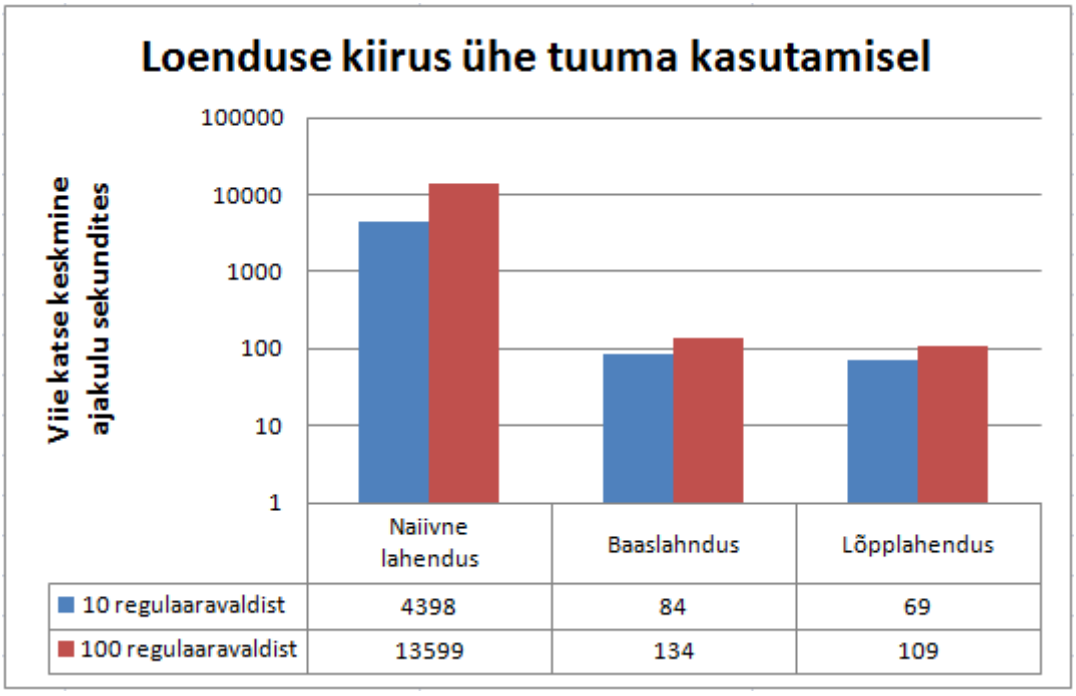
Ühe tuuma kasutamisel on naiivne lahendus väga aeglane (Graafik 1). Kümne regulaaravaldise sageduste leidmiseks kulub keskmiselt umbes 73 minutit ja saja

regulaaravaldise korral läheneb ajakulu juba neljale tunnile. Sellest lähtuvalt ei ole naiivne lahendus kasutamiseks piisavalt kiire ühe tuuma korral.

Baaslahendus ja lõpplahendus näitavad palju paremaid tulemusi leides 10 regulaaravaldise sagedused alla vähem kui 1,5 minutiga ning 100 regulaaravaldise sageduste leidmisele kulub umbkaudu 2 minutit. Mõlemal juhul on lõpplahendus oodatult natukene kiirem.

Eraldi tasub märkimist vähene ajakulu vahe 10 ja 100 regulaaravaldise vahel. See on tõenäoliselt põhjustatud sellest, et mõlemad programmid kasutavad uue loendustöö alustamiseks programmi harutamist (fork), millele võib antud juhtudel kuluda suur osa terve loendustöö peale kuluvast ajast.

Teine põhjus võib tuleneda operatsioonisüsteemist. Nimelt võib operatsioonisüsteem vajadusel otsustada mälus olevad andmed ajutiselt kõvakettale kirjutada. Kui neid andmeid on taas vaja siis laeb operatsioonisüsteem need andmed taas mällu, mis võtab ka omajagu aega.



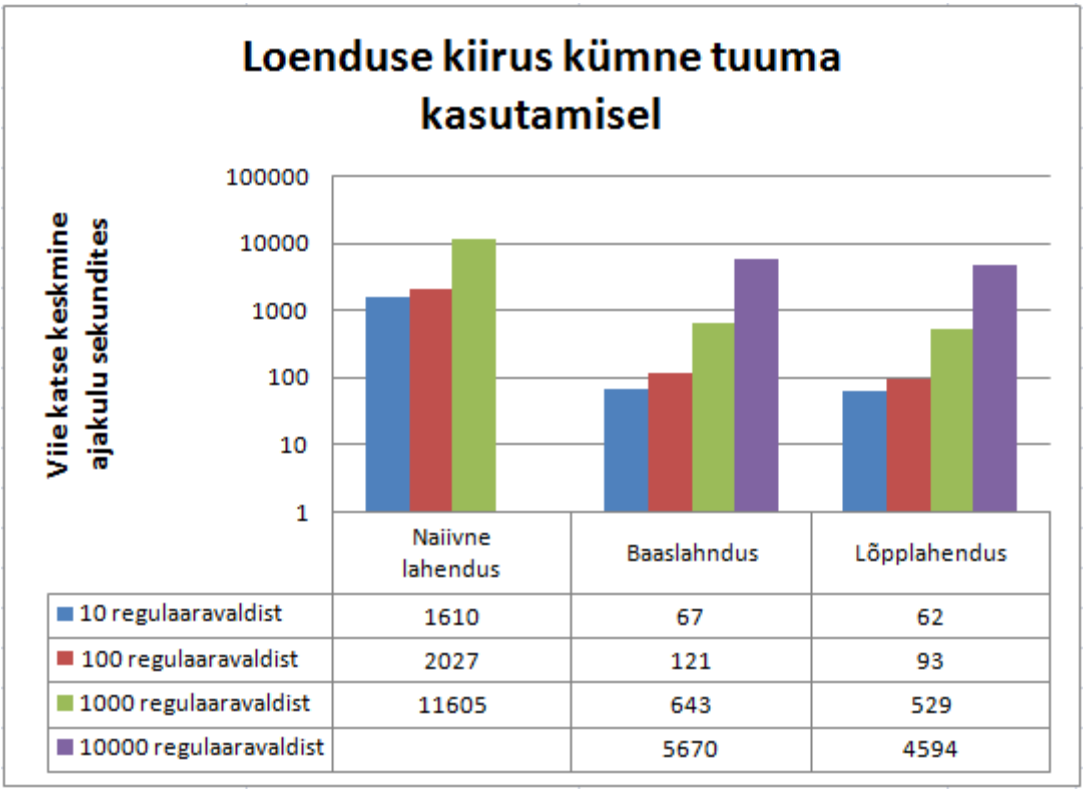
Graafik 1. loendamise ajakulu võrdlus ühe protsessori tuuma kasutamisel

5.2 Programmide kiirus mitme tuuma kasutamisel

Kümne tuuma kasutus on andnud kõige suurema võidu naiivses lahenduses 10 ja 100 regulaaravaldise suuruste tööde puhul. Sellest hoolimata on naiivne lahendus siiski umbes 24 korda aeglasem kui baas- või lõpplahendus. Naiivse lahenduse loendamiskiirus on vastuvõetav vaid väikeste loendustööde puhul. 1000 regulaaravaldise leidmiseks kulub juba üle kolme tunni, mis on juba liiga pikk aeg ja viitab selgelt, et lahendus ei ole kasutatav suuremahuliste loendustööde puhul.

Baaslahendus ja lõpplahendus näitavad samas üsna häid tulemusi. Loenduskiirus on piisav ka 10000 regulaaravaldise suuruste tööde puhul (baaslahendusel kulub umbes 95 minutit ja lõpplahendusel umbes 77 minutit).

Väikeste loendustööde puhul on baas- ja lõpplahenduse paralleliseerimine andnud marginaalse kiirusevõidu. Seda tõenäoliselt seetõttu, et alamprotsesside käivitamisele kulub suur osa tööajast. Samas suurendades sisendiks olevate regulaaravaldiste arvu sajalt tuhandeni ei tõuse loenduse ajakulu kümme korda (ajakulu tõus on umbes 5.8 kordne). Selle põhjal võin väita, et mitme tuuma kasutamine on üsna edukas. Ka sisendi suurendamisel tuhandelt kümne tuhandeni ei tõuse ajakulu kümme korda (tõus on umbes 8.7 kordne).



Graafik 2. Regulaaravaldiste loendamise ajakulu võrdlus kümne protsessori tuuma kasutamisel

6 Kokkuvõte

Käesoleva bakalaureusetöö ülesandeks oli luua ja implementeerida algoritm, mis leiab võimalikult kiirelt regulaaravaldisele vastavad sõnad ning arvutab regulaaravaldise sagedused proovides. Seda tuli teha eelnevalt koondtabelis olemasolevate andmete põhjal lähtudes selle tabeli formaadist. Sellest tulenevalt on valminud programmid oma ülesehituselt üsna ülesande spetsiifilised. Samas seatud eesmärk sai täidetud ja programmi esimesed kaks versiooni on ka mitmeid kuid juba edukalt kasutuses olnud. Kolmas versioon (C++ keelne) valmis bakalaureusetöö lõpuks ja ei ole kirjutamise hetkel veel reaalselt kasutusse võetud, aga kiirus ja tulemuse õigsus on testitud.

Antud ülesande esimese lahendusena sai esitatud nii öelda naiivne algoritm, mille puhul oli kasutatud mitut tuuma, et faili lugemist ja regulaaravaldiste sõnadega võrdlemist kiiremini sooritada. See algoritm on toodud rohkem lihtsa baaslahendusena, mille abil võrrelda, kui efektiivsed on baas- ja lõpplahendus.

Teise lahenduse juures olen kasutanud keerukamat lähenemist mis sisaldas endas andmete pidevalt mälus hoidmist, bitivektoritega tehete tegemist, hõreda maatriksi mälus hoidmist ja selle maatriksi ridade kokku liitmist. Lisaks sai arendatud lahendus tööde järjekorras hoidmiseks, mitme tuuma kasutamiseks ja tööde paralleelseks jooksutamiseks. Selle kõigega saavutasin küll soovitud kiiruse aga jooksin Pythoni programmeerimiskeelest tulenevate eripärade otsa, mis tõstsid mälukasutuse liiga suureks.

Mälukasutusprobleemide lahendamiseks sai sama algoritm ja programm tervikuna implemeneteeritud C++ programmeerimiskeeles. Selle käigus võtsin kasutusele lihtsa andmebaasi tööde järjekorras hoidmiseks ja muutsin sellest tulenevalt ka programmi struktuuri. Kuna välja töötatud algoritm andis muidu piisava kiiruse, siis seda ei pidanud enam muutma.

Loodud programmidest esimene (naiivne lahendus) omab süsteemile kõige väikseimaid nõudeid ja peaks olema kasutatav igas masinas, kus on isntalleeritud Python 2.6+. Teised kaks lahendust (baas- ja lõpplahendus) on mõeldud kasutamiseks suure mälumahuga Linuxi operatsioonisüsteemiga masinates, mis on üldjuhul ööpäeva

ringselt sisse lülitatud. Baaslahendus vajab Python 2.6+ versiooni ja lisamoodulit `bitarray 0.8.0`. Lõpplahendus vajab installeeritud `sqlite3` andmebaasi tarkvara ja kompileerimisel on kasutatud `gcc 4.1.2` kompilaatorit.

Lisaks on loodud programmide kiirus testitud erineva suurusega sisendite korral. Kiirustestide tulemusena selgus, et naiivne lahendus oli väga aeglane ja ei sobinud seetõttu hästi reaalsetes oludes kasutamiseks. Samas baas- ja lõpplahendus, mis kasutasid keerukamat algoritmi, olid enamikel juhtudel üle 25 korra kiiremad naiivsest lahendusest. Lõpplahenduses sai lahendatud ka põhilised baaslahenduse puudused, mille tulemusena on lõpplahendus reaalsetes oludes suhteliselt hästi kasutatav.

Bakalaureusetöö raames valminud C++ keelne programm töötab piisava kiirusega ja täidab püstitatud eesmärke. Kindlasti on veel mõningaid võimalusi selle programmi optimiseerimiseks ja kasutajasõbralikumaks muutmiseks, aga üldjoontes sai püstitatud ülesanne täidetud.

7 Kirjandus

Kõikide siin toodud URL'ide kättesaadavus on kontrollitud 07.03.2013. Kasutatud raamatud on kättesaadavad Google Scholar'i kaudu (http://scholar.google.com/) ja vastavad lingid on toodud iga raamatu kohta.

[1] „Unixhelp.ed.ac.uk – grep man page"
http://unixhelp.ed.ac.uk/CGI/man-cgi?grep

[2] „Python 2.6 – re module documentation"
http://docs.python.org/2.6/library/re.html

[3] „Python 2.6 – multiprocessing module documentation"
http://docs.python.org/2.6/library/multiprocessing.html

[4] „Python 2.6 – description of file objects"
http://docs.python.org/2.6/library/stdtypes.html#file-objects

[5] „Python 2.6 – os.path module documentation"
http://docs.python.org/2.6/library/os.path.html

[6] „The Python Wiki – Global InterpreterLock"
http://wiki.python.org/moin/GlobalInterpreterLock

[7] „Presentation by David Beazley – Understanding the Python GIL" 2010
http://www.dabeaz.com/python/UnderstandingGIL.pdf

[8] „SciPy v0.11 Reference Guide – Class scipy.sparse.lil_matrix"
http://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.lil_matrix.html

[9] „SciPy v0.11 Reference Guide – Class scipy.sparse.coo_matrix"
http://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.coo_matrix.html

[10] „Die.net – nohup man page"
http://linux.die.net/man/1/nohup

[11] „Pyro4 documentation"
http://pythonhosted.org/Pyro4/

[12] „Python 2.6 – description of os.fork method."
http://docs.python.org/2.6/library/os.html#os.fork

[13] Daniel P. Bovet, Marco Cesati "Understanding the Linux Kernel"
O'Reilly Media, Inc. 2008 – lk 388
http://books.google.ee/books?id=h0lltXyJ8aIC&printsec=frontcover#v=onepage&q&f=false

[14] „cdf.fnal.gov – Concept of zombie processes"
http://www-cdf.fnal.gov/offline/UNIX_Concepts/concepts.zombies.txt

[15] Daniel P. Bovet, Marco Cesati - **“**Understanding the Linux Kernel” 2008 – lk 28
http://books.google.ee/books?id=h0lltXyJ8aIC&printsec=frontcover#v=onepage&q&f=false

[16] „Stackoverflow – Sharing of memory between different processes in Python“
http://stackoverflow.com/questions/1268252/python-possible-to-share-in-memory-data-between-2-separate-processes/1269055#1269055

[17] „Python 2.6 – Reference Counting“
http://docs.python.org/2.6/c-api/refcounting.html

[18] „beej.us – Using Unix pipes in C++ aplications“
http://beej.us/guide/bgipc/output/html/multipage/pipes.html

[19] „Die.net – pipe man page“
http://linux.die.net/man/2/pipe

[20] W. Richard Stevens, Stephen A. Rago “Advanced Programming in the UNIX Environment: Second Edition” Addison Wesley Professional 2005 – lk 640
http://free4ebook.com/Advanced%20Programming%20in%20the%20UNIX%20Environment%20by%20W.%20Richard%20Stevens,%20Stephen%20A.%20Rago%20II%20Edition.pdf

[21] „A simple unix/linux daemon in Python by Sander Marechal“
http://www.jejik.com/articles/2007/02/a_simple_unix_linux_daemon_in_python/

Lisad

Bakalaureusetöö käigus loodud programmid on kättesaadavad GitHubi repositooriumist, mis asub aadressil: https://github.com/FableBlazeII/Alman_thesis.git

Abstract

Counting word frequencies based on limited regular expressions

Bachelor thesis (6 EAP)

Anti Alman

One of the tasks in bioinformatics is to find good biomarkers, that could help us in better understanding, diagnosing and treating different diseases. Finding such biomarkers involves analyzing large sets of biological data.

This bachelor's thesis concentrates on developing and implementing an algorithm for a subtask in a biomarker discovery pipeline. The pipeline itself is being developed at the BIIT group in the University of Tartu as part of an industrial collaboration. The input of this algorithm is data about a large number of different biological samples. The data about these samples is represented by using short words and corresponding frequencies, which allow us to find significant differences between samples. It is also known that in some cases a limited regular expression would be a much better representation of these differences. However the frequencies that correspond to any given regular expression need to be calculated based on words and the frequencies of these words.

This problem can be divided into two parts. First we need to find all of the words that match the given regular expression, this is achieved by using large bitvectors that will be constantly stored in memory. The second part concentrates on calculating the frequencies based on matching words. Speed is here achieved by storing frequencies in memory as a sparse array in format that allows fast adding of rows.

The resulting algorithm is implemented in both Python and C++. The details of these implementations are given and finally the speed of both of these implementations is measured against a naive solution.

The bachelors thesis results in an program that is able to find the frequencies of input regular expressions with sufficient speed.