TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Miron Storožev

Graatsiliste graafide arvjada leidmine kasutades paralleelarvutusi

Bakalaureusetöö (9 EAP)

Juhendaja: Ahti Peder, PhD

Tartu 2018

Graatsiliste graafide arvjada leidmine kasutades paralleelarvutusi

Lühikokkuvõte:
Graatsiliste graafide kirjeldamine on üks graafiteooria lahendamata probleeme. Graafide üheks kirjeldamise võimaluseks on kasutada arvjada, ent senini pole suudetud leida piisaval hulgal graatsiliste graafide arvjada elemente. Eelnevalt on koostatud programm, mis leiab n-tipuliste graatsiliste graafide arvu, kuid selle keerukusest tuleneva pika tööaja tõttu pole tänaseni loendatud rohkemaid kui 11-tipulisi graatsilisi graafe.
Käesolevas lõputöös on paralleelarvutusi kasutades optimeeritud olemasolevat graatsiliste graafide loendamise programmi. Paralleelarvutuse rakendamise tulemusena kiirendatakse graatsiliste graafide leidmise protsessi. Käitades optimeeritud programmi klastril on leitud ja loendatud 12- ja 13-tipulised graatsilised graafid.
Bakalaureusetöö tulemusena valminud programmi võib kasutada ka suurema tipuarvuga graatsiliste graafide loendamiseks. Töö käigus leitud 12- ja 13-tipulisi graafe kasutatakse edaspidi graatsiliste graafide kirjeldamisel.

Võtmesõnad:
Graatsiline graaf, arvjada, paralleelarvutused, lõimed, klaster, programm

CERCS: P175 Informaatika, süsteemiteooria

Finding Sequence of Graceful Graphs Using Parallel Computing

Abstract:
Describing graceful graphs is one of many unsolved problems in graph theory. One option to describe graphs is to use their integer sequence. So far, describing graceful graphs using its number sequence was unattainable, since sequence was not long enough to make any conclusions based on it. The program, that finds number of graceful graphs with n vertices, already exists, but its time consumption caused by large computational complexity made counting graceful graphs with more than 11 vertices almost impossible.
The aim of this bachelor's thesis is to optimise existing graceful graphs counting program by using parallel computing. Optimised program was run on cluster to count graceful graphs with 12 and 13 vertices.
Optimised program can be used to count graceful graphs with even higher number of vertices. Numbers of graceful graphs with vertex number of 12 and 13 are added to number sequence and will be used to describe graceful graphs.

Keywords:
Graceful graph, sequence, parallel computing, threads, cluster, program

CERCS: P175 Informatics, systems theory

Sisukord

1 Sissejuhatus

Graafiteoorias on palju lahendamata probleeme. Üks enamtuntud lahendamata probleem on graatsiliste graafide kirjeldamine. Senini pole suudetud leida omaduste komplekti, millega saaks efektiivselt kontrollida, kas graaf on graatsiline või mitte. A. Pederi ja M. Tombaku artiklis [M11] on selgitatud, et kasutades suvalise struktuuri arvjada, on teatavatel juhtudel võimalik leida omaduste komplekt, millega saab antud struktuuri kirjeldada. Kui kahe struktuuri arvjadad on samasugused, saab mõlemat struktuuri kirjeldada samasuguste omadustega. Juhul, kui kahe struktuuri arvjadad on sarnased, võib ühte struktuuri kirjeldada teise struktuuri omaduste komplektiga, lisades või eemaldades struktuurist mõningaid omadusi. Erinevaid arvjadasid võib leida on-line arvjadade entsüklopeediast (*The On-Line Encyclopedia of Integer Sequences*) [N]. On leitud, et graatsiliste graafide arvjada (entsüklopeedias on arvjada id A243013, kuid arvjada viimane arv ei ole õige)

$$(1, 1, 1, 3, 5, 12, 37, 112, 340, 1078 \dots)$$

on väga lähedane märgendamata n tipu ja n-1 servaga graafide arvjadaga A001433

$$(1, 1, 1, 3, 6, 15, 41, 115, 345, 1103 \dots),$$

seega ka nende omaduste komplektid võivad olla sarnased. Vaatamata sellele avastusele pole suudetud kitsendada märgendamata graafide omaduste komplekti nii, et allesjäänud omadused kirjeldaksid graatsilisi graafe. Graatsiliste graafide omaduste komplekti leidmise teeb eriti keeruliseks andmete vähesus. Isegi tänapäeva võimsate arvutitega on suudetud leida vaid arvjada esimesed 11 liiget.

Käesoleva lõputöö eesmärk on optimeerida olemasolevat graatsiliste graafide loendamise programmi, kasutades paralleelarvutusi. Optimeeritud programmi kasutatades täiendatakse senini leitud graatsiliste graafide arvjada, loendades 12- ja 13-tipulisi graatsilisi graafe. Samuti on töö eesmärk leida vastavad graafid edasisteks uurimistöödeks.

Töö tulemused on vajalikud graatsiliste graafide kirjeldamiseks. Töös leitud 12- ja 13- tipuliste graatsiliste graafide arvud täiendavad senini leitud arvjada. Leitud arve kasutatakse eelnevalt kirjeldatud omaduste komplekti leidmiseks.

Töö raames valminud algoritm leiab programmile etteantud arvu n puhul, kui palju on unikaalseid n-tipulisi graatsilisi graafe. Programm kuvab graafide arvu ja soovi korral võib kirjutada graafid ka faili. Algoritm on kirjutatud nii, et programmi võib käitada mitme arvuti peal ja seejärel tulemused põimida. Samuti on algoritm

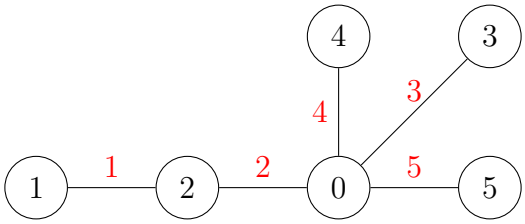
optimeeritud selliselt, et käitamisel kasutatakse ära kogu protsessori jõud, mille tulemusena töötab programm kiiremini.
Antud tööga valminud algoritmi võib arvutivõimsuse kasvades kasutada ka suurema tipuarvuga graatsiliste graafide arvu leidmiseks. Samuti on töö näiteks, kuidas jagada mahukas arvutusülesanne väiksemateks tükkideks.
Käesolev bakalaureusetöö koosneb kuuest peatükist, millest esimene on sissejuhatus ja viimane kokkuvõte. Töö teises peatükis antakse ülevaade taustinformatsioonist: selgitatakse graatsilise graafi mõistet ning kirjeldatakse, mis on paralleelarvutused ja milliseid paralleelarvutuse alaliike on antud töös kasutatud. Samuti selgitatakse mõistet klaster ning räägitakse, kuidas toimub klastril tööde käitamine. Peatükis kolm kirjeldatakse senini kasutatavat graatsiliste graafide arvu leidmise programmi ning selgitatakse, miks on antud algoritmi käitamine niivõrd ajamahukas. Neljandas peatükis räägitakse, kuidas on töö autor olemasolevat algoritmi paralleliseerinud. Samuti antakse ülevaade programmi üldistest muudatustest ning selgitatakse, kuidas käitati paralleliseeritud programm klastril. Töö viiendas osas räägitakse töö tulemustest: võrreldakse eelneva ja töö käigus loodud programmi kiirust ning antakse ülevaade, mida on optimeeritud algoritmiga leitud.

2 Taustinformatsioon

Käesolevas peatükis selgitatakse peamisi töös kasutatavaid mõisteid. Peatükis kirjeldatakse nii graafiteooriaga kui ka algoritmi paralleliseerimisega seotud termineid ning räägitakse, mis on klaster ja kuidas töötab klastril ressursside haldamine. Lisaks tutvustatakse jaga-ja-valitse meetodit.

2.1 Graatsiline märgendus ja graatsiline graaf

Graafi *graatsiliseks märgenduseks* nimetatakse märgendust, kus igale graafi tipule on omistatud unikaalne number vahemikus $\{0, \ldots, E\}$, kus E on graafi servade arv. Seejärel märgendatakse ära graafi servad nii, et iga serv tippudega x ja y saab väärtuseks tippudele omistatud arvude absoluutvahe $|f(x) - f(y)|$. Kui selle tulemusel on igal serval unikaalne väärtus, siis ongi antud graaf graatsiliselt märgendatud. [V01] Graafi, mida saab graatsiliselt märgendada, nimetatakse *graatsiliseks graafiks*.



Joonis 1. Näide graatsilisest graafist.

Joonisel 1 on näide kuuetipulisest graatsilisest graafist. Jooniselt võib näha, et iga serva märgend (joonisel märgendatud punasega) on võrdne serva ümbritsevate tippude absoluutvahega. Samuti on näha, et kõik servad on unikaalse väärtusega, mis teebki antud graafi graatsiliseks.

2.2 Graafi kujutamine arvjadana

David A. Sheppard [A76] on kirjeldanud oma artiklis, et igat märgendatud graatsilist graafi on võimalik kujutada arvjadana. Selleks luuakse nn *faktoriaalpuu*. Joonisel 2 on kujutatud faktoriaalpuu, millega on koostatavad kõik kuuetipulised (viieservalised) graatsiliselt märgendatud graafide arvjadad. Faktoriaalpuu iga tipu väärtuseks on kaks arvu. Arvud näitavad, milliste tippude vahel on graafis serv. Kui tipu väärtuseks on (0 2), siis on serv graafi tippude 0 ja 2 vahel. Joonist kasutades

kujutatakse ühte märgendatud graatsilist graafi arvjadana järgnevalt kirjeldatud viisil. Kujutamist alustatakse kõige ülemisest tipust. Antud juhul on selleks tipp (0 5). See tähendab, et tippude 0 ja 5 vahel luuakse serv. Serva loomisel jäetakse meelde valitud tipu indeks selles puus. Esimesel tasemel on see alati 0, kuna sellel tasemel on ainult üks valik. Seejärel liigutakse taseme võrra allapoole ja valitakse uuelt tasemelt omakorda suvaline tipp. Näiteks valitakse tipp (1 5). Meelde jäetakse antud olukorras indeks 1, kuna vasakult paremale lugedes on tipp (1 5) positsioonil 1. Protsessi korratakse, kuni algoritm jõuab kõige alumisele tasemele. Meelde jäetud indeksite jada ongi graafi kirjeldav arvjada. Arvjada loetakse paremalt vasakule. Näiteks kui on valitud järgmised tipud: (0 5), (1 5), (2 5), (3 5), (3 4), siis graafi kirjeldav arvjada on: 3,3,2,1,0 (joonis 4 vasakult esimene graaf).

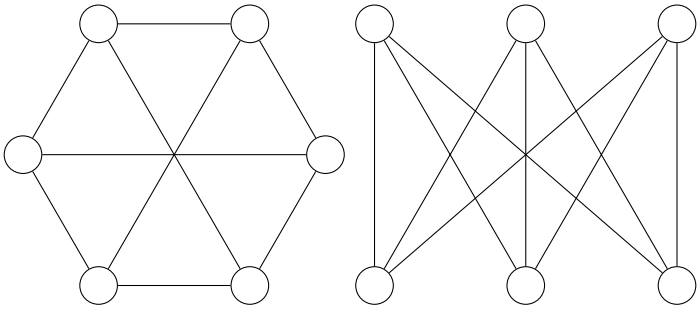
(0 5)
(0 4) (1 5)
(0 3) (1 4) (2 5)
(0 2) (1 3) (2 4) (3 5)
(0 1) (1 2) (2 3) (3 4) (4 5)

Joonis 2. Graatsiliste graafide loomise faktoriaalpuu.

Kuna graafide loomiseks läbitakse kõikvõimalikud faktoriaalpuu tippude kombinatsioonid, tekib olukord, kus tegemist on faktoriaalkeerukuse probleemiga. Kasutades joonisel 2 kujutatud faktoriaalpuud luuakse viieservalisi graafe $1 * 2 * 3 * 4 * 5 = 5! = 120$ tükki.

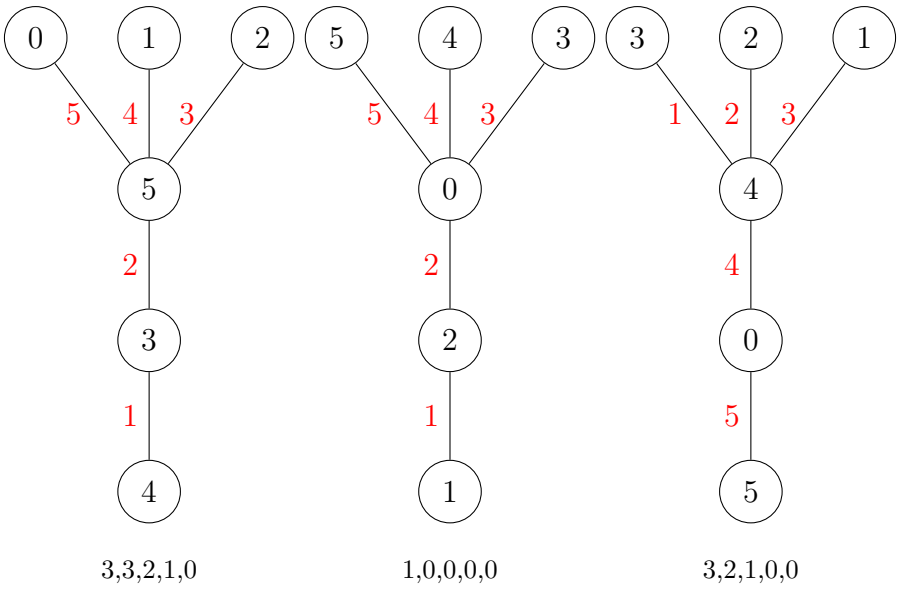
2.3 Isomorfism

Kahte graafi $G_1 = (V_1, E_1)$ ja $G_2 = (V_2, E_2)$ nimetatakse *isomorfseteks*, kui leidub selline bijektsioon $\varphi : V_1 \to V_2$, et graafis G_1 on tippude u ja v vahel serv parajasti siis, kui graafis G_2 on serv tippude $\varphi(u)$ ja $\varphi(v)$ vahel. V_1 ja V_2 on vastavate graafide tipud ning E_1 ja E_2 on servad.[R03] Lihtsustatult tähendab graafide isomorfsus seda, et graafi G_1 tippusid saab liigutada nii, et tekiks graaf G_2 ja vastupidi. Joonisel 3 on toodud näide kahest isomorfsest graafist.
Graafe loendatakse isomorfismi täpsusega, st isomorfseid graafe üksteisest ei eristata. Kui otsitakse teatava omadusega graafe, siis selle all mõeldakse, et graafid on tulemushulgas omavahel mitteisomorfsed. [R03] Graatsiliste graafide puhul on isomorfismi täpsusega loendamine keeruline, kuna võib osutuda, et ühte graafi saab mitmel viisil graatsiliselt märgendada. See tähendab, et arvjadade põhjal ei ole



Joonis 3. Näide isomorfsetest graafidest.

võimalik öelda, kas graafid on omavahel isomorfsed. Jooniselt 4 on näha, et kuigi igal kolmel graatsilisel graafil on erinev tippude ja servade märgendus ning erinev graafi kirjeldav arvjada, on nad struktuurilt identsed, seega isomorfsed.



Joonis 4. Kolm isomorfset graatsilist graafi erineva arvjadaga.

2.4 Paralleelprogrammeerimine

Traditsiooniliselt kirjutatakse arvutitarkvara jadamisi arvutuseks. Probleem jagatakse väiksemateks alamülesanneteks, mida jooksutatakse üksteise järel. Programm jookseb alati ühel protsessoril ja kindlal ajahetkel käitatakse vaid üks käsk. [B18] Paraku ei ole selline programmeerimise stiil alati efektiivne ja seetõttu ei sobi

mahukate arvutuste tegemiseks. Suurte arvutuste jaoks kasutatakse enamasti paralleelarvutusi.
Paralleelarvutustes kasutatakse korraga mitut arvutusressurssi. Probleem jagatakse osadeks, mida lahendatakse eraldi protsessoritel või tuumadel samaaegselt teiste osadega. Iga osa on omakorda jagatud alamülesanneteks, mis töötavad traditsioonilisel jadamisi viisil. [B18]
Riistvaraliselt jagatakse paralleelarvutusi mitmesse erinevasse klassi. Nende klassifitseerimine toimub taseme järgi, kus parallelism toimub. Näiteks võib parallelism toimuda tuumas, protsessoris või arvutite vahel. Antud töös on kasutatud kahte paralleelarvutuse klassi: mitmelõimelisust, mis on mitmetuumalise arvutamise alaliik, ja klasterarvutamist.

2.5 Mitmetuumaline protsessor ja lõimed

Mitmetuumaline protsessor koosneb kahest või enamast omavahel ühendatud tuumast. Suurema tuumade arvuga kaasneb tõhustatud jõudlus ja kuna tuumad on omavahel füüsiliselt ühendatud, siis nendevaheline kommunikatsioon on väga kiire. [M13] Mitmetuumalise protsessori maksimaalse efektiivsuse saavutamiseks kasutatakse lõimesid.
Lõimede kasutamine võimaldab arvutiprogrammil jagada üks ülesanne mitmeks koeksisteerivaks (ingl *coexisting*) ja sünkroonselt töötavaks alalmprotsessiks. Kuigi programmi lõimed jagavad omavahel mälu, on igal lõimel oma käsuloendur, väljakutsete magasin (ingl *call stack*) ja lokaalsed muutujad. See tähendab, et iga lõim on võimeline töötama iseseisvalt. [B06] Lõimesid võib käitada eraldi tuumadel. Õige lõimede arvu kasutamine tagab, et kõik alamprotsessid töötavad paralleelselt eraldi tuumadel, mis kindlustabki töö tõhususe.

2.6 Klaster

Tihtipeale on rakenduse käitamiseks vaja rohkem arvutusressursse, kui tavaarvuti suudab pakkuda. Sellisel juhul on suureks abiks klastri kasutamine. Klaster on töötlussüsteem (ingl *processing system*), mis koosneb omavahel ühendatud, kuid üksteisest sõltumatult töötavatest arvutitest. [R99] Arvutid töötavad koos, jättes mulje, et tegemist on superarvutiga. Klastri kasutamine võimaldab käitada üht programmi mitme arvuti peal. Selleks, et paralleliseerida programmi klasterarvutusi kasutades, tuleb programm jagada osadeks.
Klastrite korrektseks funktsioneerimiseks on oluline ressursside haldamine. Res-

sursside haldamisega kooskõlastatakse ligipääs täitmisüksustele ning operatiivmälu (RAM) ja püsimälu (ROM) eraldamine. Klastril võib olla tuhandeid kasutajaid ja igal kasutajal on võimalik käitada mitut rakendust erinevate nõuete ja käitusajaga. Sellises olukorras on klastri käsitsi haldamine võimatu. Õnneks on mitmeid häid tarkvarasid, millega saab kasutaja rakenduse töökoormust jaotada mitme arvuti vahel ja jälgida töö täitmise progressi. Üks tuntuimatest ressursside haldamise tarkvaradest klastril on Slurm. [M17]

2.7 Slurm

Slurm (*Simple Linux Utility for Resource Management*) on vabavaraline süsteem, mis on mõeldud Linuxi klastrite haldamiseks ja tööde järjekorda seadmiseks. Slurmil on kolm põhifunktsiooni: võimaldada ressursside eraldamist kasutajale, pakkuda raamistik tööde alustamiseks, teostamiseks ja tööprogressi jälgimiseks ning koostada tööde järjekord. Selleks, et Slurmiga programmi käitada, peab kasutaja looma nn töö, kus tuleb täpsustada, millist programmi soovitakse käitada, mitu arvutit on programmi jooksutamiseks vaja ja kui pikk on programmi tööaeg. [M17] Kui klaster kasutab partitsioone, siis tuleb ka märkida, millisel partitsioonil soovitakse töö käitada. Joonisel 5 on toodud näide Java programmi käitamisest Slurmiga.

```
#!/bin/bash

#SBATCH -partition=main
#SBATCH -nodes=1
#SBATCH -time=00:10:00

module load java 1.8.0_40
module load jdk 1.8.0_25

javac TestProgramm.java
java TestProgramm
```

Joonis 5. Näide Slurmi skriptist.

Koodilõigust võib näha, et lisaks partitsiooni, arvutite arvu ja tööaja määramisele tuleb Java programmi käitamiseks laadida moodulid „java” ja „jdk” (Java Development Kit). Lisaks sellele tuleb programm ka eelnevalt kompileerida. Programmi ühekordseks jooksutamiseks piisab koodi kompileerimisest ja seejärel selle käitamisest. Programmi käitamiseks paralleelselt mitme arvuti peal tuleb kasutada Slurmi

käsku *srun*. Selleks tuleb määrata, mitu korda soovitakse programm käitada ja mitu programmi käitatakse ühel arvutil. Joonisel 6 on koodijupp programmi käitamisest 24 arvuti peal.

```
#!/bin/bash

#SBATCH -partition main
#SBATCH -nodes 24
#SBATCH -ntasks 24
#SBATCH -ntasks per node=1
#SBATCH -cpus per task 4
#SBATCH -time 10:00:00

module load java 1.8.0_40
module load jdk 1.8.0_25
cd ProjektiKaust/src/

javac Klass1.java
javac Klass2.java

srun java Klass1
```

Joonis 6. Slurmi skript programmi käitamiseks 24 arvuti peal.

Lisaks eelnimetatud kohustuslikele väljadele on Slurmis võimalik kasutada mitmeid lisakäske RAM-i, protsessorite arvu, teavituste, teavituste tüübi ja muu määramiseks. Lisas I on toodud Slurmi enamlevinud käsud koos selgitustega.
Skripti käitamiseks klastris kasutatakse Slurmi käsku *sbatch*. Kui Slurm aktsepteerib skripti, saab kasutaja töö ID, millega saab töö täitmise progressi jälgida. Ebaõnnestunud skripti käitamisel annab Slurm tagasiside vea põhjusest. Joonisel 7 on kujutatud õnnestunud ja ebaõnnestunud skripti käitamine.

```
[kasutaja@rocket testkaust]$ sbatch bash_script.sh
Submitted batch job 2876103
[kasutaja@rocket testkaust]$ sbatch bash_script_will_fail.sh
sbatch: error: Batch job submission failed: Requested node
    configuration is not available
```

Joonis 7. Näide õnnestunud ja ebaõnnestunud Slurmi skripti käitamisest.

Kui tööskript on käitatud, siis on kasutajal võimalus töö progressi kohta saada lisainformatsiooni. Enne kui töö on klastril alanud, võib näha üldist tööjärjekorda. Kui

töö on klastril alanud, võib jälgida, millisel klastril, partitsioonil ja arvutil töö jookseb ning kui kaua on töö juba jooksnud. Töö lõppedes võib kuvada informatsiooni töö kestvuse, mälukasutuse jms kohta.

2.8 Jaga-ja-valitse

Jaga-ja-valitse (ingl *divide and conquer*) on algoritmi koostamise tehnika, kus probleem jagatakse mitmeks väiksemaks samasuguseks alamprobleemiks. Alamprobleemid lahendatakse järjest ära ja seejärel kombineeritakse alamtulemused kokku, lahendades sellega esialgse probleemi. [P95]
Jaga-ja-valitse tehnikal on mitu positiivset omadust. Tehnika kasutamine vähendab probleemi raskusastet, kuna jagab antud töö lihtsamini lahendatavateks alamprobleemideks. Kui alamprobleemideks jagamise ajaline keerukus on märgatavalt väiksem kui probleemi enda ajaline keerukus, siis toob tükkideks jagamine lisaks raskusastme vähendamisele endaga kaasa ka tõhusa ajalise võidu. Näiteks kui on tarvis lahendada ruutkeerukusega $O(n^2)$ ülesanne, kus tulemuse saamiseks tuleb võrrelda kõiki jada elemente omavahel ja andmete jagamine tükkideks toimub lineaarse keerukusega $O(n)$. Sellisel juhul andmete jagamine isegi kaheks osaks kiirendab tööaega märgatavalt. Väite selgitamiseks on toodud järgnevad arvutused: olgu järjendis k arvu, mida tuleb kõiki omavahel võrrelda. Võrdlemist võib lugeda elementaaroperatsiooniks. Kui lahendada probleem ühe osana, siis võrdlusoperatsioonide arv on

$$k * (k-1)/2 = k^2/2 - k/2.$$

Kui aga jagada andmed kaheks osaks, siis mõlemas osas on $k/2$ elementi. Seega on ühes osas tehtavate võrdluste arv

$$k/2 * (k/2-1)/2.$$

Kahe grupi peale sooritatakse võrdlusi kokku

$$k/2 * (k/2-1) = k^2/4 - k/2.$$

Kui tulemusele liita juurde ka andmete osadeks jagamine, siis keerukus kokku tuleb

$$k^2/4 - k/2 + O(k).$$

Kui andmete arv on suur ja andmed töödeldakse osadeks jaotamata, siis töötlemise keerukus tuleb ligikaudu $k^2/2$, sest arvesse läheb vaid suurima astmega liige. Kaheks

osaks jagamisel tuleb keerukus aga ligikaudu $k^2/4$. Seega juhul, kui osadeks jagamise keerukus on märgatavalt väiksem kui probleemi keerukus, siis suure andmehulga peal on probleemi jagamine isegi kaheks alamprobleemiks ajaliselt märgatavalt tõhusam.

Jaga-ja-valitse algoritme on enamasti võimalik realiseerida paralleelarvutusi kasutades, kuna probleemi tükkideks jagamise käigus tekkinud alamprobleeme võib lahendada üksteisest sõltumatult ja seega ka paralleelselt. [SW96]

Jaga-ja-valitse tehnikat kasutatakse tihti algoritmides, kus esineb rekursioon. Põhiprintsiip sellisel juhul on jagada probleemi tükkideks nii kaua, kuni probleemi lahendus on triviaalne. Populaarsemad näited jaga-ja-valitse tehnika kasutamisest rekursioonis on kiirsortimine (ingl *quicksort*) ja põimemeetodil sorteerimine (ingl *mergesort*).

3 Olemasolev algoritm

Käesolevas peatükis selgitatakse olemasolevat graatsiliste graafide arvu leidmise algoritmi. Peatükis kirjeldatakse, kuidas antud algoritm loob ja võrdleb graafe, kuidas programmis iseloomustatakse graafe arvjadaga ning põhjendatakse, miks antud algoritmi kasutamine on niivõrd ajamahukas.

3.1 Programmi tööaeg

Olemasolevat programmi on eelnevalt käitatud ja leitud sellega kuni 11-tipuliste graatsiliste graafide arvud. Tabelis 1 on näidatud selle programmiga leitud graatsiliste graafide arvud.

Tabel 1. Senini leitud graatsiliste graafide arv.

Tippude arv	**Graatsilisi graafe**
1	1
2	1
3	1
4	3
5	5
6	12
7	37
8	112
9	340
10	1078
11	3620

Põhjus, miks senini on suudetud leida ainult kuni 11-tipuliste graatsiliselt märgendatud graafide arv, on programmi suur ajakulu. Antud algoritmi ajalise keerukuse hinnang on $O(n!^2)$, kus n on graafi servade arv. See tähendab, et programmi töötamise aeg kasvab servade arvu suurendamisel rohkem kui faktoriaalselt. Kui 9-tipulised (8-servalised) graatsilised graafid leiab programm umbes 22 sekundiga, siis 10-tipuliste (9-servaliste) graafide leidmiseks kulub programmil aega umbes 14 minutit ja 11-tipuliste (10-servaliste) graafide leidmiseks koguni 13 tundi. Tabelis 2 on näidatud algoritmi tööaja pikenemine graafi tipuarvu suurendamisel.

Tabel 2. Tippude arv ja umbkaudne aeg isomorfsete graatsiliste graafide leidmiseks.

Tippude arv	**Aeg**
1	$<$ 0.01 sek
2	$<$ 0.01 sek
3	$<$ 0.01 sek
4	$<$ 0.01 sek
5	$<$ 0.01 sek
6	~0.02 sek
7	~0.08 sek
8	~0.8 sek
9	~22 sek
10	~14 min
11	~13 tundi
12	~28 päeva
13	~5 aastat

Kui graafi tippude arv on väiksem kui 6, siis programmi tööaeg on väga väike ning selle täpne mõõtmine on keeruline. Seetõttu on tabelisse märgitud, et tööaeg on väiksem kui 0.01 sekundit. Tabelis ajad 12- ja 13-tipuliste graatsiliste graafide leidmiseks ei ole tegelikult mõõdetud, vaid ennustatud lähtudes eelnevate tulemuste suhetest. Näiteks 9- ja 10-tipuliste graatsiliste graafide arvu leidmise aja suhe on umbes 41 korda.

$$15\texttt{min} = 900\texttt{s}$$
$$900\texttt{s}/22\texttt{s} \approx 41$$

10- ja 11-tipuliste graafide arvu leidmise aja suhe on umbes 48 korda.

$$12\texttt{h} = 720\texttt{min}$$
$$720\texttt{min}/15\texttt{min} = 48$$

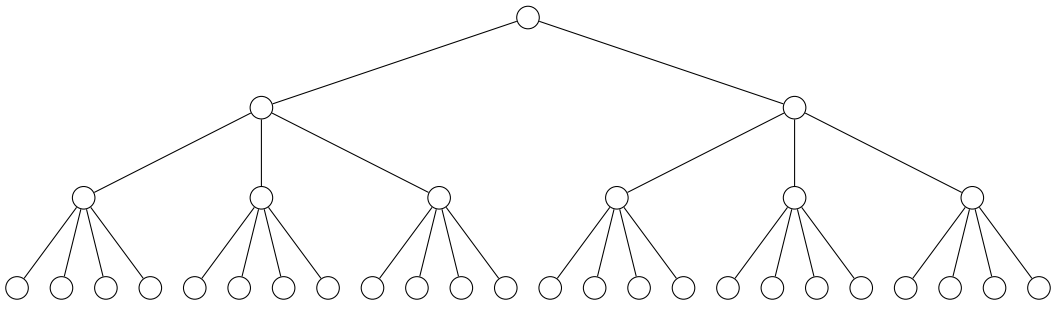
Siit võib ennustada, et 11- ja 12-tipuliste graatsiliste graafide arvu leidmise aja suhe on ligikaudu 55. Suhet kasutades võib samuti ennustada, et 12-tipuliste graatsiliste graafide arvu leidmiseks kulub umbkaudu 28 päeva.

$$12\texttt{h} * 55 = 660\texttt{h}$$
$$660\texttt{h}/24 \approx 28\texttt{p}$$

Sama arvutust on kasutatud ka 13-tipuliste graatsiliselt märgendatud graafide arvu leidmiseks. Algoritmi ajalist keerukust ja tohutut tööaega põhjustab nii graafide loomine kui ka isomorfismi kontroll.

3.2 Graafide loomine

Märgendatud graatsiliste graafide tipuarvuga $n+1$ loomise keerukus on $\Theta(n!)$, kus n on graafi servade arv. Antud keerukus tuleneb asjaolust, et luuakse kõikvõimalikud graatsiliselt märgendatud graafid. Graafide loomisel kasutatakse David A. Sheppardi kirjeldatud meetodit, kus graafi kujutatakse arvjadana. Arvjadade konstrueerimiseks kasutatakse rekursiivset funktsiooni, mille rekursioonipuu sügavus on võrdne graafi servade arvuga. Rekursiivsele funktsioonile antakse argumendina kaasa algselt tühi täisarvude massiiv. Funktsiooni põhitöö tehakse *for*-tsüklis, kus käiakse läbi kõik arvud vahemikus $\{0, \ldots, i\}$, kus i on antud rekursiooni tase. Rekursiooni tasemel 1 on vahemik $0 \ldots 1$, rekursiooni tasemel 3 on see $0 \ldots 3$. Iga arvu käsitlemisel luuakse koopia olemasolevast arvumassiivist, lisatakse arvumassiivi käsitletav arv ja liigutakse järgmisele rekursiooni tasemele, andes funktsioonile argumendiks tekitatud massiivi. Kuna Sheppardi notatsioonis loetakse arvjada paremalt vasakule, siis lisatakse igal tasemel uus arv arvjada esimeseks elemendiks. Selgub, et igal rekursiooni tasemel i kutsutakse funktsioon rekursiivselt välja $i+1$ korda. Kokku kutsutakse funktsioon rekursiivselt välja $i!$ korda. Kui funktsioon jõuab viimasele rekursiooni tasemele, siis ongi üks graaf arvjadana kirjeldatud. Kuna funktsioon jõuab rekursiooni viimasele tasemele $i!$ korda, siis kirjeldatakse kokku $i!$ graafi. Et aga rekursiooni sügavustase i on võrdne graafi servade arvuga n, siis kokku luuakse $n!$ graafi ja seega on ka algoritmi keerukus $\Theta(n!)$. Konstrueeritud graaf lisatakse graafide massiivi. Kui kõik graafid on loodud, kontrollitakse graafide isomorfsust üksteisega.



Joonis 8. Rekursioonipuu sügavusega 4.

Joonisel 8 on illustratsioon algoritmi rekursioonipuust, mille sügavus on 4. Võib näha, et viimasele rekursiooni tasemele jõutakse $4! = 24$ korda, seega luuakse ka 24 graatsiliselt märgendatud graafi.

3.3 Isomorfismi kontroll

Algoritmi teine pool on isomorfsete graafide eemaldamine. Isomorfsete graafide eemaldamise keerukus on $O(k^2)$, kus k on graafide arv. Selle põhjuseks on, et iga graafi isomorfsust tuleb kontrollida kõigi senini leitud unikaalsete graafidega. Algoritmi töö algab tühja massiivi loomisega unikaalsete graatsiliste graafide jaoks. Massiivi lisatakse kõikidest loodud graafidest esimene. Seejärel võetakse ette teine graaf ja kontrollitakse, kas ta on iga senini leitud unikaalse graafiga mitteisomorfne. Kui jah, siis graaf lisatakse unikaalsete graafide hulka. Vastasel juhul liigutakse edasi järgmise graafi juurde. Protsess kordub, kuni kõik graafid on läbi vaadatud ja alles on jäänud vaid unikaalsed graatsilised graafid. Kuna tulemushulka lisatavate graafide asukoht genereerimisel enne isomorfismi kontrolli ei ole teada, siis ei ole võimalik antud algoritmile anda täpset keerukuse hinnangut (Θ-relatsiooni mõttes). Seega ajalise keerukuse hinnang $O(k^2)$ on antud juhul mitte keskmise, vaid halvima juhu keerukus.
Isomorfismi kontrolliks kasutatakse olemasolevat algoritmi, mis võrdleb kahe graatsilise graafi külgnevusmaatrikseid. Algoritmi autoriks on Janno Siim ja seda on käsitletud tema bakalaureusetöös. [J]

4 Programmi paralleliseerimine

Selles peatükis räägitakse autori panusest töösse. Kirjeldatakse, kuidas programmi optimeerimiseks andmed tükkideks jagatakse ning kuidas on antud algoritmile rakendatud paralleliseerimine lõimede ja arvutite vahel. Samuti on selgitatud, kuidas on antud algoritmi jooksutatud klastril ning seejärel arvutite tulemused kokku põimitud.

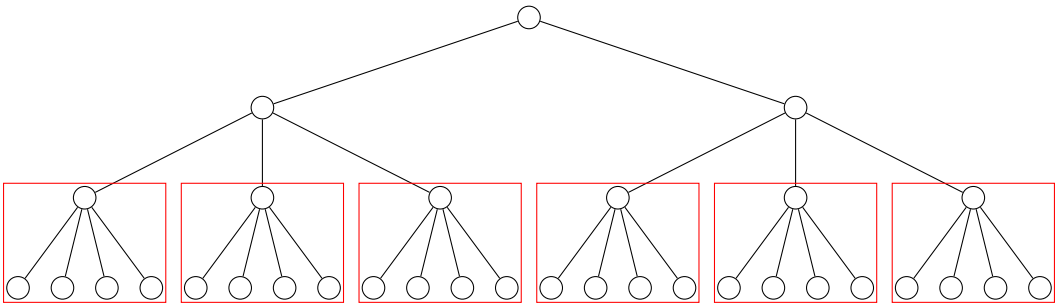
4.1 Üldised muudatused programmis

Programmi koodi loetavuse ja arusaadavuse parandamiseks ning paralleliseerimise kergemaks teostamiseks sooritati programmis mõningad muudatused. Koodi loetavuse hõlbustamiseks tekitati uus Java klass *GracefulGraph*. Klassis loodi põhifunktsioonid graafi kuvamiseks ja faili kirjutamiseks ning defineeriti võrdlemise meetod *equals*, mis tagastab tõeväärtuse, kas kaks graafi on omavahel isomorfsed. Peameetodis muudeti programmi struktuuri. Kui eelnevalt oli programm jagatud kaheks osaks: esimene osa koostas graafe ja teine eemaldas isomorfsed graafid, siis uues programmis saab tänu klassi *GracefulGraph* meetodile *equals* lisada graafid hulka (*java.util.HashSet<E>*). Kuna hulk sisaldab ainult unikaalseid elemente, siis juhul kui lisatav graaf on mõne graafiga isomorfne, jääb ta sinna lisamata. Selle tulemusena ei koosne uus programm enam kahest osast. Nii graatsiliste graafide loomine kui ka isomorfsete graafide eemaldamine sooritatakse ühe operatsioonina, mis muudab programmi paralleliseerimise kergemaks.
Eelnimetatud muudatused ei mõjutanud algoritmi ajalist keerukust ega tööaega. Muudatuste eesmärk oli muuta kood loetavamaks ja viia ta sobivale kujule paralleelarvutuste rakendamiseks.

4.2 Andmete jagamine osadeks jaga-ja-valitse meetodiga

Nagu mainitud alapeatükis 2.8, võib mahuka töö jagamine tükkideks kiirendada programmi tööaega. Selleks, et kontrollida, kas antud programmi tasub alamülesanneteks jagada, korraldati katse. Katse eesmärk oli uurida, kas antud probleemi jagamine osadeks vähendab tööaega ja millise osade arvu korral on see vähim.
Andmete jagamisel osadeks kasutati programmi rekursiivset ehitust. Programmi käitamisel anti talle ette täisarv m, mis näitab, mitmendal rekursioonipuu tasemel tuleb programm osadeks jagada. Kui funktsioon jõuab rekursiooni tasemele m, siis hakkab programm genereerima graafe osade kaupa. Varasemalt genereeris programm ühe graafi, kontrollis selle mitteisomorfsust teiste graafidega ja liikus

seejärel järgmise graafi juurde, kuni kõik graafid olid genereeritud ja kontrollitud. Praeguses lahenduses genereerib programm ainult ühe osa graafid ja kontrollib nende isomorfsust omavahel. Seejärel liigub ta järgmise osa juurde kuni kõik osad on läbitud. Lõpuks põimitakse tulemused kokku, kontrollides igas osas loodud mitteisomorfseid graafe teiste osade graafidega. Kuigi põimimine on ajamahukas, tehakse uue lahenduse puhul kokku vähem võrdlusoperatsioone, mis siiski lühendab programmi tööaega (peatükk 2.8). Lisame tähelepaneku, et graafide loomisel ega isomorfismi kontrollimisel ei ole graafide järjekord oluline. Tasemel m hargneb programm kokku $(m+1)!$ osaks, seega ka osade arv on $(m+1)!$. Joonisel 9 on näidatud programmi osadeks jagamist tasemel 2. Jaotatud osad on ümbritsetud punase kastiga. Jooniselt võib näha, et osadeks jagamine toimub rekursiooni teisel tasemel ja programm jaguneb kokku $(2+1)! = 6$ osaks.



Joonis 9. Rekursioonipuu osadeks jagamine tasemel 2.

Katse läbiviimisel käitati programm erinevate m väärtustega ja mõõdeti aeg. Saadud aeg kanti tabelisse (tabel 3). Sellest võib näha, et andmete ositi töötlemine parandab programmi tööaega. Aja paranemist näeb selgelt suuremate tipuarvudega graafide korral. Näiteks on näha, et 10-tipuliste graatsiliste graafide loendamisel on andmete jagamine 24-ks tükiks parandanud tööaega 141 ($810s - 669s = 141s$) sekundi võrra. Samuti võib täheldada, et mida suurem on loodavate graafide tippude (ja ka servade) arv, seda rohkemateks osadeks tuleb programm jagada. Kui 10-tipuliste graatsiliste graafide loendamisel oli tööaeg kiireim programmi jagamisel 24 osaks, siis 11-tipuliste graafide puhul tuli parima tööaja saavutamiseks jaotada programm 120 osaks.

Tabelist 3 on osad väärtused puudu, kuna antud algortitmi ei saa jagada rohkem kui $e!$ osaks, kus e on servade arv. See tähendab, et 6-tipulistel (5-servalistel) graafidel ei saa programmi andmeid enam $5! = 120$ osaks jagada. Jagamine ei ole võimalik, kuna sellisel juhul toimuks jagamine rekursiooni viiendast tasemest ja 6-tipuliste graafide arvutamisel ongi maksimaalne rekursiooni tase 5. 11-tipuliste

graafide loendamisel on mõned väärtused puudu, sest aja mõõtmine on niivõrd ajakulukas. Samtui on 12- ja 13-tipuliste graafide lahtrid tühjad, kuna neid pole eelnevalt suudetud välja arvutada.

Tabel 3. Algoritmi töökiiruse (sekundites) sõltuvus tippude ja osade arvust (parim aeg on märgitud rohelisega).

osi / tippe	**1**	**2**	**6**	**24**	**120**	**720**	**5040**
6	0.02	0.01	0.02	0.02			
7	0.08	0.07	0.06	0.07	0.07		
8	0.79	0.75	0.76	0.73	0.74	0.81	
9	21.8	20.3	18.4	18.2	18.4	19.8	22.1
10	810.4	799.3	763.9	669.3	679.4	793.9	855.7
11	12h48m11s			9h13m17s	8h8m35s	8h57m55s	
12						?	
13						?	

4.3 Parima osade arvu leidmine

Töö põhieesmärk oli antud algoritmi kasutades leida 12- ja 13-tipulisi graatsilisi graafe ja loendada need. Nagu selgus eelnevast alapeatükist, tasub antud programmi töö jagada osadeks. Tabelist 3 nähtub, et mida suurem on loendavate graafide tippude arv, seda rohkem vähendab programmi tööaega graafide töötlemine ositi. 11-tipuliste graatsiliste graafide loendamisel oli tööaeg 120 osaks jagamisel lausa 4 tundi kiirem kui programmi tööaeg osadeks jaotamata juhul. 12- ja 13-tipuliste graatsiliste graafide loendamisel võib see vahe olla veelgi suurem, seega õige arvu m valik on väga oluline.
Arvu m leidmiseks koostati tabel, kus on kujutatud genereeritavate graafide arv ühes osas (tabel 4). Tabelite 3 ja 4 abil leiti seos, et parim tööaeg juhtudel n=12 ja n=13 saavutatakse siis, kui igas osas töödeldakse umbkaudu 10 korda rohkem graafe, kui on lõpptulemuses. Järelduse tegemiseks kontrolliti programmi tööaegu ainult $(m+1)!$ osadeks jagamisega ja tegelik parim osade arv ei ole teada. 10-tipuliste graafide puhul on see suhe

$$15120/1078 \approx 14$$

ja 11-tipuliste graafide puhul on see hoopiski

$$30240/3620 \approx 8.$$

Tabel 4. Ühes osas töödeldavate graafide arv (osade arv, millega tuli parim programmi tööaeg on märgitud rohelisega).

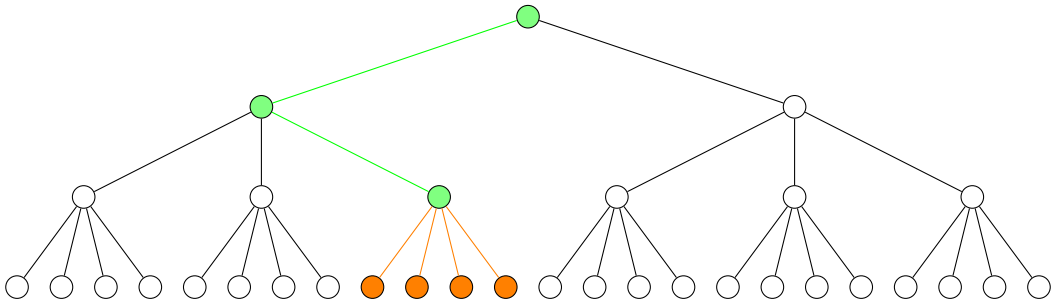
osi / tippe	**1**	**2**	**6**	**24**	**120**	**720**	**5040**
6	120	60	20	5			
7	720	360	120	30	6		
8	5040	2520	840	210	42	7	
9	40320	20160	6720	1680	336	56	8
10	362880	181440	60480	15120	3024	504	72
11	3628800	1814400	604800	151200	30240	5040	720
12	39916800	19958400	6652800	1663200	332640	55440	7920
13	479001600	239500800	79833600	19958400	3991680	665280	95040

Seega 12- ja 13-tipuliste graatsiliste graafide loendamisel valitakse selline osade arv, kus ühes osas töödeltavate graafide arv on umbes 10 korda suurem ennustatavast tulemusest. Kui vaadata senini loendatud graatsilisi graafe, siis tipuarvu kasvades ühe võrra suureneb tulemus umbkaudselt 3.5 korda. 9-tipulisi graafe on 340, 10-tipulisi 1078, seega suhe on $1078/340 = 3.2$. 11-tipulisi graafe on 3620, seega suhe 10-tipuliste graafide arvuga on $3620/1078 = 3.4$. Seega võib ennustada, et 12-tipulisi graatsilisi graafe on umbkaudselt $3620 * 3.5 = 12670$ ja 13-tipulisi graatsilisi graafe on umbkaudselt $12670 * 3.5 = 44345$. Siit võib järeldada, et 12-tipuliste graafide loendamiseks on kõige sobilikum jagada programm 720-ks tükiks, kuna sellisel juhul on ühes osas töödeldavate graafide arv 55440 (tabel 4) ja see on lähim ennustatava tulemuse kümnekordsele $12670 * 10 = 126700$. 13-tipuliste graatsiliste graafide loendamiseks on samuti mõistlik kasutada 720 osa.

4.4 Algoritmi paralleliseerimine arvutite vahel

Programmi paralleliseerimiseks arvutite vahel on tarvis jagada programm osadeks nii, et iga arvuti genereeriks erinevad graatsiliselt märgendatud graafid. Kuna kõikidel arvutitel käitatakse sama programm, siis erineva tulemuse saamiseks tuleb programm käitada erinevate argumentidega. Antud juhul antakse programmi käitamisel ette täisarv, mis määrab, millise rekursioonipuu haru programm käitab. Rekursioonipuu jagatakse harudeks alapeatükis 4.2 kirjeldatud jaga-ja-valitse

meetodi abil. Arvutite jõudmiseks õigete harudeni on loodud programmis massiiv teekondadest.
Teekonnad on defineeritud kahemõõtmelise massiivina, kus iga massiivi element on omakorda täisarvudest koosnev massiiv. Unikaalne arv, mis arvutile programmi käitamisel ette antakse, tähistab teekonna positsiooni kahemõõtmelises massiivis. Kui programmi on käitatud argumendiga 5, siis programm võtab teekondade massiivist viienda teekonna. Iga täisarv teekonna massiivis on indeks, mis tähistab, mitmes tipp tuleb antud rekursiooni tasemel valida.
Antud lähenemine tähendab, et kasutades Sheppardi graafide kujutamist arvjadana, paigutatakse arvjadasse esimesed k arvu juba ära ja edaspidi käsitletakse kõiki arvude kombinatsioone vahemikus $\{0, \ldots, m\}$, kus m on rekursiooni tase. Olgu kasutada 6 arvutit ja soovitakse emuleerida ühe arvuti töö neljatipuliste graatsiliste graafide loendamiseks. Sellisel juhul peaks iga arvuti läbi vaatama $4!/6 = 4$ graafi. Et seda teha, genereeritakse igale arvutile unikaalne teekond rekursiooni kolmanda tasemini, kasutades käitamisel programmile etteantud täisarvulist argumenti. Näiteks antakse arvutile number kolm ette massiiv [0,0,2], mis tähendab, et esimesel tasemel tuleb arvutil number kolm valida tipp indeksiga 0, teisel tasemel tipp indeksiga 0 ja kolmandal tasemel tipp indeksiga 2. Seejärel genereerib ja töötleb arvuti läbi kõik etteantud harus olevad graafid. Antud näite puhul tekib neli arvjada: [0,0,2,0], [0,0,2,1], [0,0,2,2] ja [0,0,2,3]. Kuna Sheppardi notatsioonis loetaks graafe paremalt vasakule, siis tekivad graafid: [0,2,0,0], [1,2,0,0], [2,2,0,0] ja [3,2,0,0]. Antud programmis paigutatakse igal rekursiooni tasemel läbitud element massiivi esimeseks elemendiks, seega rekursiooni viimasele tasemele jõudes on graafi kirjeldav arvjada õigel kujul. Joonisel 10 on kujutatud eelnevalt kirjeldatud protsess. Rohelisega on märgitud teekond, mille arvuti läbib. Oranžiga on märgitud programmi jagamine lõimedeks, mida selgitatakse järgmises alapeatükis.



Joonis 10. Arvuti teekond rekursioonipuu kolmanda tasemeni ja programmi jagamine lõimedeks.

4.5 Algoritmi paralleliseerimine lõimede abil

Lisaks programmi jagamisele arvutite vahel paralleliseeriti antud programm ka arvutisiseselt. Arvutisisene paralleliseerimine võimaldab maksimaalselt kasutada protsessori tuumasid, kiirendades sellega programmi tööaega veelgi. Paralleliseerimiseks kasutatati lõimesid. Programmi jagamine lõimedeks sooritati kohe pärast andmete jagamist arvutite vahel. See tähendab, et pärast seda, kui iga arvuti oli jõudnud kindlale rekursiooni tasemele, kasutades arvutile antud teekonda, tehti programm mitmelõimeliseks. Eelnevas alapeatükis selgitati, et joonisel 10 on illustreeritud olukorda, kus arvuti saab graafide genereerimiseks endale teekonna [0,0,2]. Kui arvuti jõuab rekursiooni tasemele 3, valides tipu indeksiga 2, siis jagataksegi programm neljaks lõimeks nii, et iga lõim genereerib ja töötleb vaid ühte graafi. Näiteks lõim indeksiga 0 töötleb graafi, mille arvjada on [0,0,2,0].
Jooniselt 10 võib täheldada, et programmi lõimede arv sõltub sellest, mis tasemel toimub lõimedeks jagamine. Kui programmi käitamiseks kasutatakse kuut arvutit, siis alamprotsessi jagamine lõimedeks toimub rekursiooni kolmandal tasemel. Sellisel juhul jagatakse alamprotsess neljaks lõimeks nagu kõnealusel joonisel. Kui aga programmi käitamiseks kasutatakse 24 arvutit, siis lõimedeks jaotamine toimub rekursiooni neljandal tasemel ja programm jagatakse viieks lõimeks. Selline jagamine on antud juhul sobilik, kuna eesmärgiks on käitada programm klastril, kus igal arvutil on 20 tuuma.
Lõimede loomiseks kasutati Java klassi *ExecutorService* meetodit *newFixedThreadPool*. Kui tavalisel lõimedeks jagamisel luuakse lõim ja seejärel antakse talle ülesanne, siis klassi *ExecutorService* kasutades luuakse algul kindel arv lõimesid ning seejärel luuakse ülesanne ja saadetakse see meetodile *newFixedThreadPool*, mis ise määrab ülesande vabale lõimele. See muudab lõimede hallatavuse palju kergemaks ja selgemaks. Meetodi *newFixedThreadPool* loomisel anti talle argumendina kaasa täisarv $k + 1$, kus k on eelnevalt defineeritud rekursiooni tase, kus toimub programmi jagamine lõimedeks. Argument $k + 1$ määrab, mitu lõime peab programm tekitama.
Kui lõim on lõpetanud oma graafide töötlemise, lisab ta graafid lõpptulemusse, kus seejärel samuti võrreldakse kõiki graafe omavahel. Kuna aga graafide arv on lõpptulemusse lisamisel palju väiksem, kui lõim pidi esialgselt töötlema, siis sellisel puhul saavutatakse ajaline võit mitte ainult algoritmi paralleliseerimise pealt, vaid ka andmete tükkideks jagamise pealt (selgitatud alapeatükis 4.2). Joonisel 11 on näidatud koodilõik lõimede kasutamisest klassi *ExecutorService* abiga.

```
int divisionLevel = 3;
ExecutorService THP = Executors.newFixedThreadPool(divisionLevel+1);

if(currentLevel == divisionLevel) {
    for (int i = 0; i <= currentLevel; i++) {
        final int index = i;
        Runnable task = () -> {
            int[] sheppardArray = new int[edges.length + 1];
            sheppardArray[0] = index;
            System.arraycopy(edges,0,sheppardArray,1,edges.length);
            Set<GracefulGraph> intaskGraphs = generateGraphsSet(
                currentLevel+1,sheppardArray,result);
            graphs.addAll(intaskGraphs);
        };
        THP.execute(task);
    }
}
```

Joonis 11. Väljavõte koodist: klassi *ExecutorService* kasutamine lõimede haldamiseks.

4.6 Programmi käitamine klastril

Antud töös kasutati Tartu Ülikooli klastrit (*High Performance Computing Center of University of Tartu*), täpsemalt *Rocket*-klastrit. Klaster koosneb 135 arvutist, kus igal arvutil on 20 tuuma. Klastril kasutatakse Linuxi operatsioonisüsteemi ja ressursside haldamise tarkvara Slurmi. Nagu kirjeldatud alapeatükis 2.7, tuleb programmi käitamiseks klastril luua tööskript. Selle loomisel tuli silmas pidada, et igale arvutile tuleb anda unikaalne täisarv, mis kirjeldaks arvuti teekonda rekursioonipuus. Selleks loodi skriptis massiiv, mille elementide väärtused on vahemikus $\{0, \ldots, k-1\}$, kus k on soovitud arvutite hulk. Seejärel käitati programm käsuga *srun* ning anti käitavale programmile argumendiks iga kord erinev massiivi element. Töö skript on näidatud lisas II. Selleks, et valida, mitut arvutit ja mitut tuuma igas arvutis kasutada, lähtuti tabelist 3. 12-tipuliste graatsiliste graafide loendamisel tuli tabeli järgi kiireima tööaja saavutamiseks jagada töö 720-ks osaks. Selleks jagati töö 120 arvuti vahel ja arvutisiseselt jagati töö omakorda kuue lõime vahel. Seega kasutati 120 arvutit ja igal arvutil 6 tuuma, kokku 720 tuuma. Lõimede kasutamise puhul võis jääda mulje, et kuna klastril on igal arvutil 20 tuuma, siis ei kasutata arvuti võimsust täielikult ära ja enamus tuumasid on tööta. Tegelikkuses kasutatakse ära rohkem tuumasid, sest kui Slurmi tööskripti kirjutamisel

märgitakse, et kasutatakse ainult 6 tuuma, siis Slurm võib käitada ühe arvuti peal paralleelselt kuni kolm programmi, kasutades sellega ära $6 * 3 = 18$ tuuma ja jättes kaks tuuma teisele kasutajale oma programmi jooksutamiseks. Antud töö tulemuse saavutamine ei olene sellest, mitu arvutit tegelikult tööd teeb. Põhiline on, et tööd tehakse ära paralleelselt ja tööaeg oleks vähim.

4.7 Arvutite tulemuste ühendamine

Kui iga arvuti on oma töö lõpetanud ja kuvanud väljundfaili kõik arvutisisesed unikaalsed graatsilised graafid, siis töö teine pool on tulemuste ühendamine. Selleks loodi eraldi Java klass. Klassis jagatakse programm mitme lõime vahel, kasutades taaskord klassi *ThreadPoolExecutor* objekti *newfixedThreadPool*. Iga lõim saab endale ülesandeks sisse lugeda üks fail ja koostada failist graatsiliste graafide hulk. Seejärel lisatakse hulk järjekorda (*queue*). Objekti *newfixedThreadPool* kasutamine on siinkohal parem kui lihtne programmi jagamine lõimedeks, kuna selle rakendamisel saab lõimesid taaskasutada. See tähendab, et ühele lõimele saab anda mitu ülesannet. Antud olukorras saab ühte lõime kasutades järjest sisse lugeda mitu faili. Seega ei pea looma iga faili sisselugemiseks eraldi lõime.
Kui kõik failid on sisse loetud, siis hakatakse graafide hulki omavahel kokku põimima. Siinkohal taaskasutatakse eelnevalt loodud lõimesid. Igale lõimele antakse järjekorrast kaks hulka. Seejärel lõim põimib hulgad kokku *addAll* meetodiga ja lisab uue hulga tagasi järjekorda ning võtab järgmised kaks elementi. Kui järjekoda on jäänud vaid üks element, siis kontrollitakse, kas kõik lõimed on oma töö lõpetanud. Kui ei, siis oodatakse kuni iga lõim on seda teinud ja jätkatakse võrdlusprotsessiga. Kui kõik lõimed on oma töö lõpetanud, siis on järjekorras ainult üks hulk, mille suurus ongi otsitava tipusuurusega unikaalsete graatsiliste graafide arv. Graafide uurimiseks võib hulgas olevad graafid kuvada faili. Kuna väljundfailides olevate graafide arv on suhteliselt väike, siis tulemuste ühendamine on oluliselt vähem töömahukas, võrreldes tööga, mida iga arvuti pidi tegema nende failide loomiseks.

5 Tulemused

Antud peatükis räägitakse töö tulemustest. Peatükis võrreldakse paralleliseeritud ja paralleliseerimata programmi tööaegu ning räägitakse ka sellest, mida on suudetud leida, kasutades paralleliseeritud programmi.
Käesoleval tööl on kaks põhitulemit: programm paralleliseeriti ja programmi abiga loendati 12- ja 13-tipulised graatsilised graafid. Töö tulemusena on valminud mitmelõimeline programm, mida on võimalik käitada klastril, jagades töö mitme arvuti vahel. Selleks, et kontrollida, kas programm töötab õigesti, loendati paralleliseeritud programmiga kuni 11-tipulised graatsilisi graafe. Tulemus kattus esialgsel programmil saaduga.
Paralleliseeritud programmi tööaegu on võrreldud esialgse programmi tööaegadega. Võrdlemisel leiti, et paralleliseeritud programm loendab graatsilisi graafe esialgsest programmist kiiremini. Võrdluse tulemusi võib näha tabelis 5. Kui esialgu kulus 10-tipuliste graatsiliste graafide arvu leidmiseks 14 minutit, siis paralleliseeritud programm sai tööga hakkama umbes 20 sekundiga. Kiiruste vahe on umbes 42 kordne,

$$14\texttt{min}/20\texttt{s} = 42$$

Suuremat ajalist võitu võib näha 11-tipuliste graatsiliste graafide puhul, kus programmi käitamine ühel arvutil ja ühel lõimel võttis aega umbes 12 tundi ja 48 minutit, aga paralleliseeritud programmiga vaid 22 minutit. Paralleliseeritud programmi tööaeg oli esialgsest programmist umbes 35 korda kiirem.

$$12\texttt{h}48\texttt{min}/22\texttt{min} \approx 35$$

Tabel 5. Endise algoritmi ja uue programmi tööaja võrdlus.

Tippude arv	**Esialgne algoritm**	**Uus algoritm**
8	~0.8 s	~1.2 s
9	~22 s	~5.1 s
10	~14 min	~1 min
11	~13 h	~22 min
12	~28 p	~16 h
13	~5 aastat	~15 p

Kasutades paralleliseeritud programmi suudeti leida 12- ja 13-tipuliste graatsiliste graafide arvud. 12-tipulisi graatsilisi graafe on 12737 ja 13-tipulisi graafe on 46376.

Lisaks töö põhieesmärkide saavutamisele leiti katset sooritades, et rekursioonipuuga loodud n graafi puhul on esimesed $n/2$ graafi ja teised $n/2$ graafi omavahel sümmeetrilised. Iga esimese poole graafide märgendus x on teises pooles asendatud märgendusega n-x, kus n on graafi servade arv. Näiteks kui rekursioonipuu esimeses pooles leidub graaf, mille arvjada on 3,3,2,1,0 ja graafi servade arv on 5, siis rekursioonipuu teises pooles leidub graaf, mille arvjada on 2,2,3,4,5. Eelnevast järeldub, et rekursioonipuu kõik esimese poole graafid on isomorfsed teise poole graafidega ja seega pole graatsiliste graafide loendamisel tarvis vaadata rekursioonipuu teist poolt. Sellist nähtust kirjeldab David A. Sheppard oma artiklis.[A76]

6 Kokkuvõte

Graatsiliste graafide kirjeldamine on üks suuremaid graafide valdkonna lahendamata probleeme. Senini pole leitud omadusi, millega saaks efektiivselt kontrollida, kas graaf on graatsiline.
Senini kasutatud algoritmiga on suudetud loendada kuni 11-tipulised graatsilised graafid. Antud töö eesmärk oli optimeerida olemasolevat graatsiliste graafide loendamise programmi ja kasutada seda 12- ja 13-tipuliste graatsiliste graafide arvu leidmiseks.
Töös on kirjeldatud, kuidas programmi efektiivsuse parandamiseks tehti graafide leidmise algoritm mitmelõimeliseks ja mitmel arvutil paralleelselt käitatavaks. Lisaks on näidatud, kuidas 12- ja 13- tipuliste graafide arvu leidmiseks jooksutati programmi klastril.
Töö tulemusena valmis programm, millega saab kordades kiiremini loendada graatsiliselt märgendatud graafe. Töö analüüsivas osas võrreldi paralleliseeritud ja paralleliseerimata programmide töökiirust. Paralleliseeritud programmi kasutades leiti, et 12-tipulisi graatsilisi graafe on 12737 ja 13-tipulisi on 46376. Olemasoleva ja töö käigus valminud programmide lähtekoodid on lisas III.
Lõputöö on suureks abiks graatsiliste graafide edasisel uurimisel. Edaspidi võib programmi rakendada ka veelgi suurema tipuarvuga graafide leidmiseks ning kasutada antud tööd tutvumaks võimalusega, kuidas jagada suuremahulisi töid väiksemateks alamprobleemideks.

Viidatud kirjandus

[A76] Sheppard D. A. The factorial representation of balanced labelled graphs. *Discrete Mathematics*, 15:379–388, 1976.

[B06] Goetz B. Java concurrency in practice. Pearson Education, 2006.

[B18] Blaise B. Introduction to parallel computing. `https://computing.llnl.gov/tutorials/parallel_comp/`, 2018. (17.04.2018).

[J] Siim J. Graafiteooria ülesannete komputeriseerimine. TÜ arvutiteaduse instituudi bakalaureusetöö, 2014. `https://comserv.cs.ut.ee/ati_thesis/datasheet.php?id=41201&year=2014`.

[M11] Peder A, Tombak M. Finding the description of structure by counting method: a case study. *SOFSEM 2011: Theory and Practice of Computer Science*, 2011.

[M13] Rouse M. multi-core processor. `https://searchdatacenter.techtarget.com/definition/multi-core-processor`, 2013. (10.04.2018).

[M17] Sterling T, Anderson M, Brodowicz M. High performance computing: Modern systems and practices. Elsevier Inc., 2017.

[N] Sloane N. The on-line encyclopedia of integer sequences. `http://oeis.org/`.

[P95] Brassard G, Bratley P. Fundamental of algorithmics. Pearson Education, 1995.

[R99] Baker M, Buyya R. Cluster computing at a glance. `http://academic.csuohio.edu/yuc/hpc00/lect/chapter-B1.pdf`, 1999. (03.05.2018).

[R03] Palm R. Diskreetse matemaatika elemendid. Tartu: Tartu Ülikooli Kirjastus, 2003.

[SW96] Achatz K, Schulte W. Massive parallelization of divide-and-conquer algorithms over powerlists. *Science of Computer Programming*, 26:59–78, 1996.

[V01] Vassilevska V. Coding and graceful labeling of trees. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.70.2416&rep=rep1&type=pdf`, 2001. (17.04.2018).

Lisad

I. Enamlevinud Slurmi käsud

käsk	**selgitus**
–job-name	töö nimi
–nodes	arvutite arv
–ntasks	alamülesannete arv
–cpus-per-task	CPU-de arv ühe ülesande kohta
–ntasks-per-node	mitu ülesannet ühel arvutil jooksutada
–mem	mälu hulk ühe arvuti kohta
–nodelist	nimekiri arvuti ID-dest, mida soovitakse kasutada
–exclude	nimekiri arvuti ID-dest, mida soovitakse vältida
–time	tööaeg
–chdir	kodukaust, kust leida käitatav programm
–partition	partitsioon, mida soovitakse kasutada
–mail-type	teavitused kasutajale.
–mail-user	kasutaja meiliaadress
–output	väljundfail. Vaikeväärtus on "slurm-%j.out", kus %j on töö ID
–error	fail, kuhu kirjutatakse juhul, kui programm viskab erindi

II. Slurmi skript programmi käitamiseks klastril erinevate argumentidega

```
#!/bin/bash
#SBATCH -partition main
#SBATCH -job name Graceful_graph_sequence
#SBATCH -nodes 24
#SBATCH -ntasks 24
#SBATCH -ntasks per node=1
#SBATCH -cpus per task 5
#SBATCH -mem 10GB
#SBATCH -time 10:00:00
#SBATCH -chdir /gpfs/hpchome/miron/thesis/
#SBATCH -output=/gpfs/hpchome/miron/thesis/graatsiline_%a.out
#SBATCH -array=0-23
#SBATCH -mail type=ALL
#SBATCH -mail user=miron.storozhev96@gmail.com

module load java 1.8.0_40
module load jdk 1.8.0_25
cd Graatsiline Graaf Arvjada/src/

javac Node.java
javac NodeList.java
javac VF2_State.java
javac IntPair.java
javac Isomorphism.java
javac GracefulGraph.java
javac Graceful_Main.java

srun java Graceful_Main ${SLURM_ARRAY_TASK_ID}
```

III. Lähtekoodid

Programmide lähtekoodid on avalikes Githubi repositooriumites.

1. esialgse programmi repositoorium:
   https://github.com/m1r0n/Graatsilised-OLD.git
2. töö käigus loodud programmi repositoorium:
   https://github.com/m1r0n/Graatsiline-Graaf-Arvjada.git

IV. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks

Mina, **Miron Storožev**,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose

   Graatsiliste graafide arvjada leidmine kasutades paralleelarvutusi,

   mille juhendaja on Ahti Peder

   1.1 reprodutseerimiseks säilitamise ja üldsusele kättesaadavaks tegemise eesmärgil, sealhulgas digitaalarhiivi DSpace-is lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;

   1.2 üldsusele kättesaadavaks tegemiseks Tartu Ülikooli veebikeskkonna kaudu, sealhulgas digitaalarhiivi DSpace´i kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.

2. olen teadlik, et punktis 1 nimetatud õigused jäävad alles ka autorile.

3. kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest tulenevaid õigusi.

Tartus, 14.05.2018