Turun yliopisto
University of Turku

Hajautetut ohj.järjestelmät ja pilvipalvelut Harjoitustyö 1

Laskennan nopeuttaminen säieohjelmoinnilla

Ryhmä:
Ei ryhmää

Harjoitustyö 1
Laskennan nopeuttaminen säieohjelmoinnilla
16. marraskuuta 2018
13 s., 1 liites.

Turun yliopisto
Tulevaisuuden teknologioiden laitos
Tietotekniikka
Hajautetut ohj.järjestelmät ja pilvipalvelut

Sisältö

1 Johdanto

Nykyään on tavallista, että muutaman euron arvoisista sulautetuista järjestelmistä alkaen tietokoneet sisältävät useampia ns. suoritinytimiä ja sitä kautta kyvyn laskea yhden sijaan useita laskutoimituksia samanaikaisesti. Erilaiset laskentatehtävät rinnakkaistuvat eri tavalla, mutta laskennan tehostamisen kannalta helpoin algoritmien kategoria on ns. helposti rinnakkaistuvat ongelmat (*embarrassingly parallel*). Nämä algoritmit voidaan osittaa niin, että osat voidaan laskea täysin itsenäisesti ilman kommunikointia osien välillä. Käytännössä tällaiset ongelmat voidaan jakaa aluksi osaongelmiin, käynnistää kunkin osan suoritus rinnakkain ja/tai peräkkäin ja odottaa, kunnes kaikki laskenta päättyy, jonka jälkeen tulokset kootaan.

1.1 Mandelbrot-fraktaali

Tässä harjoitustyössä sovelletaan yhtä tällaista ongelmaa – ns. Mandelbrotin fraktaalijoukon kuvan piirtämistä rinnakkaisesti. Kuvan piirtämisessä keskeinen osa on ns. ydinfunktio (*kernel*), joka saa syötteenä kuvapisteen koordinaatit ja generoi iteratiivisella kompleksilukuaritmetiikalla kuvapisteelle värin annetusta paletista. Jokaisen pisteen väriarvo voidaan laskea toisista pisteistä riippumatta – laskennan perustaksi riittää siis pelkkä pisteen koordinaatti. Säieohjelmoinnin tekniikoiden vastuulle jää koordinoida korkean tason suoritus niin että kaikki kuvan $leveys \times korkeus$ pistettä tulevat lasketuksi mahdollisimman tehokkaasti.

1.2 Säikeistystehtävät

Tässä harjoitustyössä on seuraavat kolme ratkaistavaa säieohjelmoinnin ongelmaa:

1. Ratkaistaan ongelma *staattisella aikataulutuksella*, so. tilanne, jossa kukin säie $p_1, ..., p_n$ saa tehtäväksi kiinteän määrän tehtäviä $t_1, ..., t_m$. Kutakin tehtävää vastaa yhtäsuuri kuva-alue kokonaiskuvasta. Esimerkki: $n = 4$ ja $m = 2$. Nyt jokainen prosessori saa kaksi tehtävää, joista kumpikin vastaa $\frac{1}{n \times m} = \frac{1}{8}$ koko kuvasta. Kuvan voi jakaa esim. allekkain kahdeksaan vaakasuoraan siivuun. Kun säie on suorittanut tehtävänsä t_1, jatkaa se suoraan tehtävästä t_2. Nyt jos jonkin muun säikeen tehtävä on vielä keskeneräinen, näitä tehtäviä ei jaeta säikeiden välillä vaan nopeiten valmistuvat jäävät odottamaan.

2. Ratkaistaan ongelma *dynaamisella aikataulutuksella*, so. tilanne, jossa kukin säie $p_1, ..., p_n$ saa tehtäväksi vaihtelevan määrän tehtävistä $t_1, ..., t_m$. Kutakin tehtävää vastaa tässäkin yhtäsuuri kuva-alue kokonaiskuvasta, mutta algoritmin ero syntyykin siinä, miten tehtäviä jaetaan. Esimerkki: $n = 4$ ja $m = 64$. Nyt jokainen säie saa tehtävän, joka vastaa $\frac{1}{m} = \frac{1}{64}$ koko kuvasta. Kuvan voi jakaa esim. ”tiiliin”, niin että kokonaiskuva koostuu 8×8 pienemmästä ruudusta, joiden kunkin koko on $\frac{leveys}{8} \times \frac{korkeus}{8}$. Suoritettuaan tehtävänsä säie poimii jonosta uuden tekemättömän tehtävän, kunnes kaikki tehtävät on laskettu.

3. Käyttöliittymän (piirretään uudelleen 50 ms välein) irrottaminen laskennasta:

```
1 class Ticker implements Scheduled {
2   public int tickDuration() { return 50; }
3   public void tick() {
4       ...
5       canvas.redraw();
6       ...
7   }
8 }
```

Nyt ongelma on, mikäli fraktaalin iteraatioiden maksimimäärää kasvattaa tai ohjelmassa hiirellä navigoi työläästi laskettavaan kohtaan, laskenta voi kestää yli 50 millisekuntia, jolloin keskeneräistä laskentaa voi kumuloitua, minkä seurauksena koko käyttöliittymä voi ”halvaantua”. Ongelmaan ratkaisuna on ns. työjono, joka on säieturvallinen samanaikainen jonorakenne, johon yhteen päähän tallennetaan seuraava piirrettävä kuva ja toisesta päästä luetaan käyttöliittymän esitettäväksi jonoon kertynyt kuva. Ongelma on klassinen tuottaja–kuluttaja -ongelma samanaikaisuuden kirjallisuudesta. Tässä lisähaasteina:

- Käyttöliittymää saa JavaFX:ssä päivittää vain JavaFX:n käyttöliittymäsäikeellä tai ohjelma kaatuu. JavaFX-säie tarjotaan automaattisesti ja kyseinen säie kutsuu Ticker-luokan metodeita ja oletuksena sitä kautta koko piirtoa alusta loppuun.

- Toiseksi, kuvat vievät kohtalaisen paljon muistitilaa, joten työjono ei voi olla mielivaltaisen suuri vaan jonon pituus kannattaa rajoittaa, esim. sulavan toiminnan takaamiseksi 2–3 kuvaan (jonossa on tällöin valmiina kuvia, mikäli laskenta tilapäisesti kestää yli 50 ms)

- Jonosta kulutetaan kuvia pois tasaisella 50 ms nopeudella (+ piirto JavaFX:llä kesti harj.työn laatijan koneella lähes vakioajan 1-2 ms), joten jos tuottaja pystyy tuottamaan kuvan välillä nopeammin, tuottaminen tulisi jäädyttää säietekniikoilla, kunnes jonoon mahtuu taas kuvia.

- Toisaalta, kuluttaja ei saa vastaavalla tavalla jäädä odottamaan jonosta kuvaa, mikäli seuraava näytettävää kuvaa ei ole, sillä tämä jäädyttäisi käyttöliittymän kuten alkeellinen ratkaisu ilman työjonoa. Jos jono on tyhjä, käyttöliittymän tulee jatkaa vanhan kuvan näyttämistä.

1.3 Alustava toteutus

Tehtävänannon pohjana käytettävä toteutus mallintaa Mandelbrot-fraktaalin laskennan peräkkäisesti. Toteutus on tehty Java-kielellä ja hyödyntää grafiikan esittämiseen modernia JavaFX-kirjastoa. Työssä sekä kytkös JavaFX:ään että laskennan ydinfunktio (sekä muu työn tekoa helpottava käyttöliittymä) tarjotaan valmiina. Työ vaatii siis aluksi jonkin verran olemassaolevan koodin opiskelua (noin 900 riviä ml. kommentit), mutta tässä on se hyöty, että työn varsinainen kurssin teorian kannalta tärkeä toteutus on varsin pelkistetty ja ytimekäs.

1. Fraktaalin piirto lähtee liikkeelle matalimmalla tasolla luokista
   (a) MandelbrotKernel: yleinen rajapinta Mandelbrot-kuvapisteen laskemiseksi
   (b) MandelbrotSlowKernel: toteuttaa pisteen laskennan yksinkertaisella skalaarimatematiikalla (double)
   (c) MandelbrotFastKernel: toteuttaa pisteen laskennan vektorimatematiikalla (4:n mittainen double-taulukko[1])
2. Piirron seuraava luokkien abstraktiotaso hyödyntää Javan (8+) moniperintää
   (a) MandelbrotRenderer: yleinen kehys koko fraktaalin laskemiseksi
   (b) RendererBase: yleinen kehys koko fraktaalin laskemiseksi
   (c) MandelbrotSequentialRenderer: piirtää fraktaalin peräkkäisesti
   (d) MandelbrotSlowRenderer: piirtää fraktaalin peräkkäisesti MandelbrotSlowKernel:lla
      i. SlowRenderer: yhdistää MandelbrotSlowRenderer:n laskentakoodiin

[1] Java kutsuu laitteistotasolla NEON/SSE/AVX/AltiVec-käskyjä tässä yhteydessä.

(e) MandelbrotFastRenderer: piirtää fraktaalin peräkkäisesti MandelbrotFastKernel:lla

i. FastRenderer: yhdistää MandelbrotFastRenderer:n laskentakoodiin

(f) RendererFactory: apuluokka, joka luo pyynnöstä em. piirtoluokkia

3. Seuraavat luokat tulisi ryhmän toteuttaa tehtävänannon mukaisesti

(a) MandelbrotStaticParallelRenderer: piirtää fraktaalin rinnakkaisesti MandelbrotFastKernel:lla, hyödyntäen staattista aikataulutusta

i. StaticThreadRenderer: yhdistää MandelbrotStaticParallelRenderer:n laskentakoodiin

(b) MandelbrotDynamicParallelRenderer: piirtää fraktaalin rinnakkaisesti MandelbrotFastKernel:lla, hyödyntäen dynaamista aikataulutusta

i. DynThreadRenderer: yhdistää MandelbrotDynamicParallelRenderer:n laskentakoodiin

Lisäksi käyttöliittymän irrottaminen asynkronisesti laskennan suorittamisesta vaatii ylimääräistä koodia yhteen tai useampaan projektin luokkaan (vihjeeksi luokat, joita ei tarvitse muokata, on merkitty luokkia edeltäviin kommentteihin).

Katsotaan esimerkinomaisesti luokkaa MandelbrotKernel (sivuutetaan rotaatio):

```
1 interface MandelbrotKernel {
2   int vectorSize();
3   int getMaxIterations();
4   int mandelbrot(double c_re, double c_im);
5   int[] mandelbrot(double x, double y, double inc, double rot);
6 }
```

- Metodi vectorSize() palauttaa joko 1 (slow) tai 4 (fast).
- getMaxIterations() kertoo piirtoon asetettujen iteraatioiden maksimimäärän.

- mandelbrot(c_re, c_im): laskee fraktaalin väriarvon pisteessä (c_re, c_im)
- mandelbrot(x, y, inc, rot): laskee fraktaalin väriarvot pisteissä (x, y), (x+inc, y), (x+2*inc, y), (x+3*inc, y)
- Yleisesti, em. kategorioiden 2) ja 3) luokat huolehtivat näistä matalan tason rutiineista eikä niistä tarvitse välittää (pl. se fakta, että drawTile-jaon koordinaattirajojen pitää olla vektoriprosessoinnin takia neljällä jaollisia!)

Katsotaan esimerkinomaisesti luokkaa MandelbrotSequentialRenderer:

```
1 interface MandelbrotSequentialRenderer extends MandelbrotRenderer {
2   default void drawSet(Viewport vp) {
3     drawTile(0, 0, renderWidth(), renderHeight(), vp);
4   }
5 }
```

- Metodi drawSet piirtää koko fraktaalin ja delegoi oletuksena työn metodille drawTile. Vastaava metodi voi ryhmän toteutettavaksi jäävissä luokissa käsittää rinnakkaisen strategian piirtää fraktaali.
- Metodi drawTile (luokasta MandelbrotRenderer) laskee peräkkäisesti, iteratiivisesti, rivi ja sarake kerrallaan alueen (0,0) – (renderWidth(), renderHeight()) kaikki kuvapisteet. Tätä metodia (ja sitä matalamman tason asioita) ei tarvitse muokata.

Lisäksi työn kannalta on olennainen seuraava luokkajoukko, joka käsittää fraktaalin säilömiseen tarkoitetun ”piirtopinnan”:

- PixelRenderer: rajapinta, joka kytkee piirron konkreettiin piirtopintaan.
- FXPixelRenderer: edellisen konkreetti toteutus JavaFX:n piirtorutiineille.
- MandelbrotCanvas: yleinen piirtopinnan ja näkymän säilövä apuluokka.

- FXMandelbrotCanvas: edellisen konkreetti toteutus JavaFX:n piirtorutiineille.
- Ticker: kytkee FXMandelbrotCanvas:n JavaFX:n ikkunaan ja tapahtumankäsittelyyn.

1.4 Käytännön toteutus

1.4.1 Työkalujen asennus

Työn tekemistä varten, asenna seuraavat työkalut seuraavassa järjestyksessä:

1. Java Development Kit (JDK): oracle.com/technetwork/java/javase/downloads/
2. Git-versionhallinta: git-scm.com/downloads
3. SBT käännöksen automatisointityökalu (build tool): www.scala-sbt.org/
4. IDEA-ohjelmointiympäristö: www.jetbrains.com/idea/download/

Työkalujen integrointi IDEAan

1. IDEA:n asetusten kautta (File -> Settings... -> Plugins > Install Jetbrains plugin...) Etsi "Scala" ja asenna.
2. Valinnainen lisäosa: Gitlab projects
3. Säädä myös File -> Settings... -> Build, execution, deployment -> Build tools -> sbt -> Project level settings -> [x] Use Auto-import
   (a) Tämä optio on valittavissa vasta kun projekti on tuotu IDEAan. Eli siis kohdan "Projektin aloittaminen" jälkeen

JavaFX

JavaFX vastaa käyttöliittymästä esimerkkiprojektissamme. Riippuu Javan versiosta, onko JavaFX mukana vai ei.

- Java 10/11 (suositus)
  - JavaFX ei ole enää JDK:n mukana, joten se joudutaan asentamaan
  - Mikäli käytät SBT:tä, on tämä vaihe valinnainen, sillä sbt osaa ladata JavaFX 11:n automaattisesti!
  - Lataa käyttöjärjestelmällesi sopiva JavaFX SDK-paketti (zip) osoitteesta https://gluonhq.com/products/javafx/
  - Luo hakemisto openjfx joko kotikansioosi tai muuhun sopivaan paikkaan, esim Windowsissa Documents (Älä lisää JavaFX:ää git-varastoosi)
  - Pura zip sinne
- JavaFX 8
  - Windows/Mac: JavaFX osana Oracle JDK 8:aa
  - Linux: OpenJFX (pitää asentaa erikseen paketinhallinnasta)

1.4.2 Konfigurointi

Ympäristömuuttujat

- Windows: Select Start -> Computer -> System Properties -> Advanced system settings -> Environment Variables -> System variables -> PATH. ...
  - MacOS: ohje vaihtelee valitettavasti versioittain
  - Linux: editoi ~/.profile tai ~/.bashrc
  - Asetukset päivittyvät kun sisäänkirjautuu uudestaan!

Ympäristömuuttuja JAVAFX_HOME (ei välttämätön)

- Mikäli käytät SBT:tä tai Java 8:aa, ei tätä tarvitse tehdä
- Aseta ympäristömuuttuja JAVAFX_HOME osoittamaan javafx-kansion juureen
  - Eli siis kansioon, johon purit JavaFX-kohdassa JavaFX-kirjaston

Ympäristömuuttuja PATH

- Testaa että komentoriviltä voi suorittaa käskyt: git, sbt, java, javac (Oracle jostain syystä unohtanut jossain Java-versiossa sisällyttää JDK:n javac:n PATH:iin)
- Jos jokin työkalu ei toimi, lisää kyseisen työkalun binäärien hakemisto PATHiin ;-eroteltuna (Linux/Mac :-eroteltuna).

1.4.3 Projektin aloittaminen ja projektirungon lataus

Forkkaamalla projekti (suositeltu)

1. Ryhmän perustaja forkkaa (eli tekee itsellensä kopion projektista gitlabiin) osoitteessa https://gitlab.utu.fi/tech/education/distributed-systems/fractalexplorer/forks/new
   (a) Suositeltavaa pitää näkyvyys forkilla Privatena, etteivät muut pääse kopioimaan koodia
2. Ryhmäläisille tulee antaa tämän jälkeen tarvittavat oikeudet forkattuun projektiin kohdasta Settings > Members
3. Ryhmän jäsenet voivat nyt kloonata forkkisi harjoitustyöstä joko komentorivillä (git clone) tai IDEA:lla (Check out from Version Control -> Git)

```
$ git clone https://gitlab.utu.fi/ututunnuksesi/fractalexplorer.git
$ cd fractalexplorer
```

4. Joka kerta kun joku lähettää (push) commitit gitlabiin, ajetaan CI/CD-pipeline, joka tarkistaa, kääntyykö harjoitustyö ja onko se ratkaistu oikein. Ei siis kannata pelästyä ilmoitusta siitä, että testaus epäonnistui.

Manuaalisesti

1. Lataa (kloonaa) git-komentorivityökalulla projekti osoitteesta https://gitlab.utu.fi/tech/education/distributed-systems/fractalexplorer.git

```
$ git clone \
https://gitlab.utu.fi/tech/education/distributed-systems \
/fractalexplorer.git
$ cd fractalexplorer
```

   (a) Huom. Et pysty tällöin lähettämään muutoksia gitlabiin ilman lisäkonfiguraatiota, sillä sinulla ei ole kirjoitusoikeuksia projektipohjaan

   (b) Mikäli git tuottaa liiaksi hankaluuksia tälläkin asteella, voit ladata uusimman version harjoitustyöpohjasta suoraan osoitteesta https://gitlab.utu.fi/tech/education/distributed-systems/fractalexplorer/-/archive/master/fractalexplorer-master.zip

2. Voit aloittaa työskentelyn. Huomaa, että tällä tavoin et pysty hyödyntämään gitin tarjoamia etuja ryhmätyöskentelyssä, etkä saa myöskään automaattitarkastajaa käyttöön.

3. Lopuksi kuitenkin automaattitarkastajaa käytetään toimivuuden tarkistamiseksi. Tällöin voit luoda gitlabiin uuden puhtaan repositoryn ja lähettää koodit suoraan sinne.

1.4.4 Käyttö

Komentoriviltä projekti käynnistyy komennolla:

```
1 $ sbt run
```

Mikäli käynnistät suoraan IDEA:sta, käynnistä main() luokasta MandelbrotMain.

Huomioita

- Erikseen ladatun projektin voi importata myöhemminkin IDEAan
- SBT integroituna IDEA:n sisällä – ei tarvita komentoriviä
- Java-versio valitaan projektia luotaessa. Voi vaihtaa myöhemmin. Suositus: 11

2 Tehtävän kuvaus

Toteuta luvun 1.2 kolme säikeistystehtävää:

2.1 Ratkaisu *staattisella aikataulutuksella*

- Jaa kuva yhtäsuuriin vaakasuoriin siivuihin ja laske ne rinnakkain.
- Toteuta laskenta Thread/Runnable-luokkia käyttäen
- Selvitä kokeellisesti ohjelman käyttöliittymän ja/tai benchmark-rutiinien avulla konekokoonpanollesi optimaalinen määrä siivuja ja säikeitä.
- Mikäli koneesi ei sisällä moniydinprosessoria (ts. > 8v vanha), säikeiden käyttö ei ole järkevää tehtävässä alunperinkään, mutta voit lisätä piirtorutiinin yhteyteen joka säikeelle Thread.sleep(5)-viiveen, jolloin tehtävä on jälleen mielekäs.
- Vinkki: tehtävä ei vaadi juurikaan koodin uudelleenjärjestelyä

2.2 Ratkaisu *dynaamisella aikataulutuksella*

- Jaa kuva yhtäsuuriin ”tiiliin” laske ne rinnakkain.
- Toteuta laskenta joko Thread/Runnable-luokkia käyttäen tai käytä apuna Javan java.util.concurrent-luokkia

- Selvitä kokeellisesti ohjelman käyttöliittymän ja/tai benchmark-rutiinien avulla konekokoonpanollesi optimaalinen määrä tiiliä ja säikeitä.
- Mikäli koneesi ei sisällä moniydinprosessoria (ts. > 8v vanha), säikeiden käyttö ei ole järkevää tehtävässä alunperinkään, mutta voit lisätä piirtorutiinin yhteyteen joka säikeelle Thread.sleep(5)-viiveen, jolloin tehtävä on jälleen mielekäs.
- Vinkki: tehtävä ei vaadi juurikaan koodin uudelleenjärjestelyä

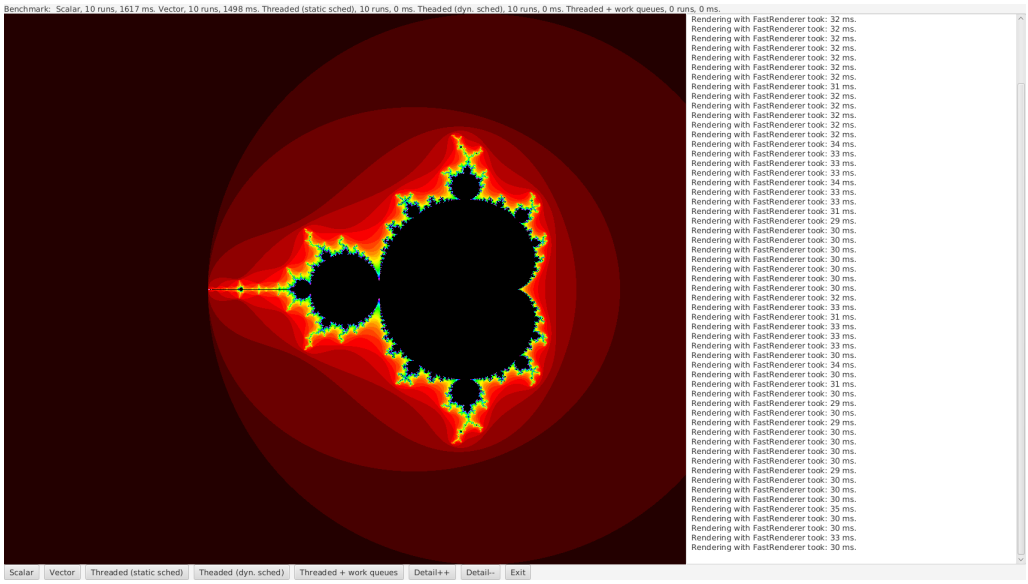
2.3 Käyttöliittymän irrottaminen laskennasta asynkronisesti

- Toteuta piirron irrottaminen käyttöliittymäsäikeestä
- Toteutuksessa kannattaa käyttää työmäärän vähentämiseksi vähintään Javan tarjoamaa jonoluokkaa ja/tai java.util.concurrent-luokkia.
- Totea asynkronisuuden toiminta lisäämällä laskennan kuormaa Detail++ -napista ja etsimällä kuvasta laskennallisesti vaativa kohta. Alkup. toteutus jumiutuu pahasti. Ratkaisun tulisi pystyä vastaamaan käyttäjän klikkailuun laskennan hidastumisesta huolimatta.
- Vinkki: tehtävä vaatii koodin uudelleenjärjestelyä usean kuvan jonottamiseen
- Mikäli koneesi sisältää vähän RAM-muistia, kytke päälle swap/page file, pienennä koodista JavaFX-ikkunan kokoa ja/tai rajaa jonon pituus 2 kuvaan.
- JavaFX:n ikkunan koko säädetään MandelbrotApp-luokan rivillä:

```
1  primaryStage.setScene(new Scene(
2    root, screen.getWidth() * 2 / 3, screen.getHeight() * 2 / 3
3  ));
```

Liite A Liitedokumentti

Mandelbrotin fraktaali perustuu kompleksilukufunktioon $x_{n+1} = x_n^2 + c$, jossa x ja c ovat kompleksilukuja. C on vakio ja x:lle annetaan alkuarvoksi $x_0 = (0, 0)$, tällöin yhtälöstä saadaan $x1 = c$ ja edelleen $x_2 = x_1^2 + c$. Iterointia jatketaan kunnes x:n itseisarvo ylittää arvon 2. Jos c:n itseisarvo on lähellä nollaa, niin x ei milloinkaan saavuta arvoa 2. Tätä vastaa fraktaalin kuvaajan keskellä oleva musta alue. Jos c:n itseisarvo on suuri, esim 2, niin heti ensimmäinen iteraatio saa x:n ylittämän arvon 2. Tätä vastaa kuvan reunoilla olevat tummimmat alueet. Tällä välillä on epämääräisen muotoinen alue, jossa tarvittavien iteraatiokierrosten määrä on vaikeasti ennustettavissa. Kuvassa kukin piste vastaa yhtä c:n arvoa ja kyseisen pisteen väri kertoo tarvittujen iterointikierrosten lukumäärän kyseisellä c:n arvolla.



Kuva A.1: Mandelbrot-fraktaali työn ohjelmarungolla piirrettynä.