



Turun yliopisto
University of Turku

Hajautetut ohj.järjestelmät ja pilvipalvelut

Harjoitustyö 1

Laskennan nopeuttaminen säieohjelmoinnilla

Ryhmä:

Mikko Mallikas (123456)

Harjoitustyö 1

Laskennan nopeuttaminen
säieohjelmoinnilla

16. marraskuuta 2018

11 s., 1 liites.

Turun yliopisto

Tulevaisuuden teknologioiden laitos

Tietotekniikka

Hajautetut ohj.järjestelmät ja
pilvipalvelut

Sisältö

1	Johdanto	1
1.1	Mandelbrot-fraktaali	1
1.2	Säikeistystehtävät	2
1.3	Alustava toteutus	4
1.4	Käytännön toteutus	7
1.4.1	Työkalut	7
1.4.2	Konfigurointi	8
1.4.3	Projektin haku	9
2	Tehtävän kuvaus	10
2.1	Ratkaisu <i>staattisella aikataulutuksella</i>	10
2.2	Ratkaisu <i>dynaamisella aikataulutuksella</i>	10
2.3	Käyttöliittymän irrottaminen laskennasta asynkronisesti	11
Liitteet		
A	Liitedokumentti	A-1

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 koordinaoida 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, \dots, p_n saa tehtäväksi kiinteän määrän tehtäviä t_1, \dots, 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, \dots, p_n saa tehtäväksi vaihtelevan määrän tehtävistä t_1, \dots, 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 $\frac{\text{leveys}}{8} \times \frac{\text{korkeus}}{8}$. 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 fraktaalien 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-fraktaalien 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. Fraktaalien 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-tilausta¹)
2. Piirron seuraava luokkien abstraktiotaso hyödyntää Javan (8+) moniperintää
 - (a) MandelbrotRenderer: yleinen kehys koko fraktaalien laskemiseksi
 - (b) RendererBase: yleinen kehys koko fraktaalien laskemiseksi
 - (c) MandelbrotSequentialRenderer: piirtää fraktaalien peräkkäisesti
 - (d) MandelbrotSlowRenderer: piirtää fraktaalien peräkkäisesti MandelbrotSlowKernel:lla
 - i. SlowRenderer: yhdistää MandelbrotSlowRenderer:n laskentakoodiin

¹Java kutsuu laitteistotasolla NEON/SSE/AVX/Altivec-käskyjä tässä yhteydessä.

- (e) MandelbrotFastRenderer: piirtää fraktaalien 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 fraktaalien rinnakkaisesti MandelbrotFastKernel:lla, hyödyntäen staattista aikataulutusta
 - i. StaticThreadRenderer: yhdistää MandelbrotStaticParallelRenderer:n laskentakoodiin
 - (b) MandelbrotDynamicParallelRenderer: piirtää fraktaalien 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 (sivutetaan 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 fraktaalien väriarvon pisteessä `(c_re, c_im)`
- `mandelbrot(x, y, inc, rot)`: laskee fraktaalien 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 fraktaalien ja delegoi oletuksena työn metodille `drawTile`. Vastaava metodi voi ryhmän toteutettavaksi jäävissä luokissa käsitellä rinnakkaisen strategian piirtää fraktaali.
- Metodi `drawTile` (luokasta `MandelbrotRenderer`) laskee peräkkäisesti, iteratiivisesti, rivi ja sarake kerrallaan alueen `(0,0) – (renderWidth(), renderHeight())` kaikki kuvapistet. Tätä metodia (ja sitä matalamman tason asioita) ei tarvitse muokata.

Lisäksi työn kannalta on olennainen seuraava luokkajoukko, joka käsittelee fraktaalien säilömiseen tarkoitettua ”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 Työkalut

- Työn tekemistä varten, asenna seuraavat työkalut seuraavassa järjestyksessä:

1. Java: [oracle.com/technetwork/java/javase/downloads/](https://www.oracle.com/technetwork/java/javase/downloads/)

2. Git: git-scm.com/downloads

3. SBT: www.scala-sbt.org/

4. IDEA: www.jetbrains.com/idea/download/

(a) IDEA:n asetusten kautta (File -> Configure -> Plugins + Install JetBrains plugin) Scala-plugin.

(b) Valinnaisesti plugin: Gitlab projects

(c) Sääda myös File -> Settings -> Build, execution, deployment -> Build tools -> sbt -> Project level settings -> [x] use auto-import

5. JavaFX

- Java 10/11 (suositus)
 - lataa käyttöjärjestelmäsi SDK-paketti (zip) osoitteesta <https://gluonhq.com/products/javafx/>
 - tee kotikansioon hakemisto openjfx
 - pura zip sinne

- tämä vaihe valinnainen, jos käytät SBT:tä
- JavaFX 8
 - Windows/Mac: Oracle JDK (tulee mukana)
 - 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. ...
 - Mac: ohje vaihtelee valitettavasti versioittain
 - Linux: editoi ~/.profile tai ~/.bashrc
 - Aetukset päivittyvät kun sisäänkirjautuu uudestaan!
- Ympäristömuuttuja **JAVAFX_HOME** (ei välttämätön)
 - viittaa yllä valittuun javafx-kansion juureen
 - esim. /home/minä/openjfx
- 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 haku

1. Lataa git-ohjelmalla projekti **TODO**

```
1 $ git clone TODO
2 $ cd TODO
```

2. tai gitlab-sivun pilvikuvakkeen takaa (jos et hallitse gitiä)

3. tai IDEA:lla (Check out from Version Control -> Git)

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

- Komentoriviltä projektin pitäisi käynnistyä käskyllä:

```
1 $ sbt run
```

2 Tehtävän kuvaus

Toteuta luvun 1.2 kolme säikeistystehtävää:

2.1 Ratkaisu *staattisella aikataulutusella*

- 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 aikataulutusella*

- 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

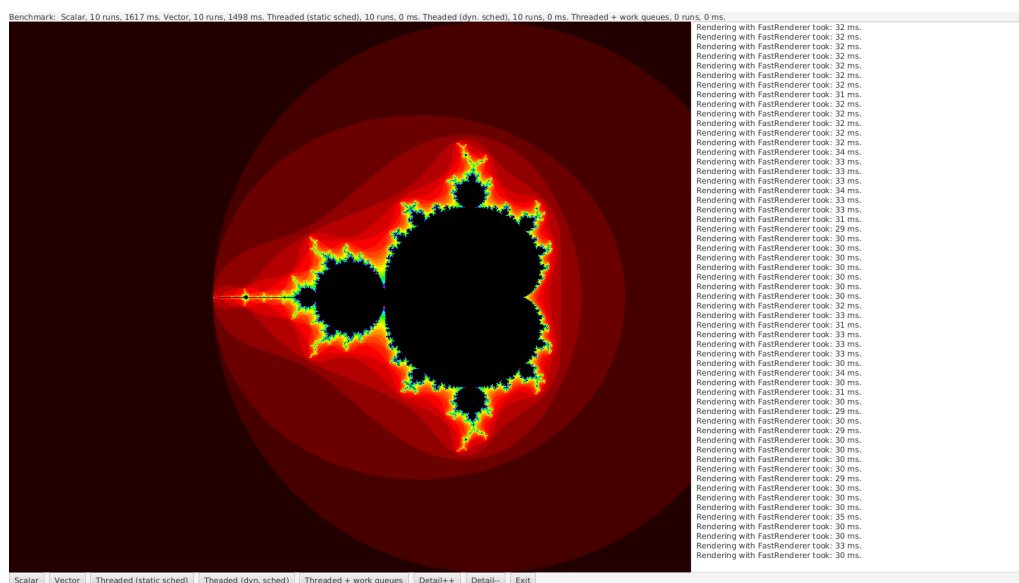
2.3 Käyttöliittymän irrottaminen laskennasta asynkronisesti

- 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 $x_1 = c$ 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 fraktaalın 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.