HAAGA-HELIA
ammattikorkeakoulu

Rinnakkaisohjelmointi JVM-ympäristössä

Timo Kottonen

Opinnäytetyö
Tietojenkäsittelyn koulutusohjelma
2015

HAAGA-HELIA
ammattikorkeakoulu

Tiivistelmä

9.5.2015

Tekijä(t)
Timo Kottonen

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma

Opinnäytetyön otsikko
Rinnakkaisohjelmointi JVM-ympäristössä

Sivu- ja liitesivumäärä
35

Opinnäytetyön otsikko englanniksi
Parallel programming on the JVM

Työn tarkoituksena oli selvittää, miten ohjelmoija voi rinnakkaisohjelmoinnin avulla hyödyntää moniydinprosessorien tehoa. Rinnakkaisohjelmoinnilla tarkoitetaan sellaisten sovellusten ohjelmointia, joissa useita säikeitä suorittaa tehtäviä rinnakkain. Työssä keskityttiin rinnakkaisohjelmoinnin haasteisiin, suunnitteluun ja toteutukseen. Työn taustalla oli idea koostaa rinnakkaisohjelmoinnista helposti ymmärrettävä ja havainnollistava johdatus, jota voisi käyttää esimerkiksi kurssimateriaalina. Työ rajattiin käsittelemään Java virtuaalikonetta (JVM) hyödyntäviä ohjelmointikieliä.

Työ toteutettiin analysoimalla useita sähköisiä ja kirjallisia lähteitä. Työssä käytettiin muutamia pieniä esimerkkisovelluksia, joiden tarkoitus oli havainnollistaa paremmin käsiteltäviä aiheita. Lisäksi työssä on yksi laajempi sovellusvertailu, jossa tutkittiin perinteisen ja rinnakkaisen sovelluksen eroja. Esimerkkisovellusten toteutuksessa käytettiin Java-ohjelmointikieltä.

Työn tulokset osoittivat, että JVM-ympäristössä suoritettavaan rinnakkaisohjelmointiin liittyy useita haasteita, kuten muistin eheyden ylläpitäminen ja säikeiden tehokas koordinointi. Näiden haasteiden taustalla olevien syiden ymmärtäminen vaatii ohjelmoijalta matalan tason komponenttien, kuten muistin, toimintaperiaatteiden ymmärtämistä. Työssä tehdystä sovellusvertailusta kävi ilmi, että rinnakkaisohjelmoinnin avulla sovellusten suorituskykyä voidaan parantaa, mutta vain lähdekoodin monimutkaistumisen kustannuksella.

Työn johtopäätökset osoittivat, että koska rinnakkaisohjelmointiin liittyy niin paljon haasteita, rinnakkaistusta suunniteltaessa pitää osata arvioida, onko suorituskyvyn parantuminen lähdekoodin monimutkaistumisen arvoista. Hyvien työkalujen puute on syy lähdekoodin monimutkaistumiselle ja myös sille, että ohjelmoijan pitää ymmärtää matalan tason logiikkaa. Funktionaalinen ohjelmointi on tällä hetkellä hyvä ratkaisu, koska muuttumattomuus hävittää useimmat ongelmat.

Asiasanat
rinnakkaisohjelmointi, moniydinprosessori, säie, java

HAAGA-HELIA
University of Applied Sciences

Abstract

9.5.2015

Author(s)
Timo Kottonen

Degree programme
Degree in Information Technology

Report/thesis title
Parallel programming on the JVM

Number of pages and appendix pages
35

The purpose of this thesis was to find out how a programmer can harness the power of multi-core processors by parallel programming. Parallel programming means creating a program where multiple threads process data simultaneously. The thesis concentrated on the parallel programming's challenges, design and implementation. The idea behind the thesis was to create a simple and visualized introduction to parallel programming, which could be used, for example, as course learning material. The thesis was limited to only cover the programming languages, which use the Java Virtual Machine (JVM).

The thesis was conducted by analysing multiple electronic and literary sources. Some small example programs were written to demonstrate the topics discussed. Moreover, the thesis has one larger program comparison, which deals with the differences between sequential and parallel programs. The example programs were written in Java.

The thesis indicated that parallel programming includes multiple challenges, like maintaining memory consistency and efficient thread coordination. Understanding the causes behind these challenges requires the programmer to understand, how low-level components, like the memory, work. The program comparison, conducted in the thesis, indicated that the program performance can be boosted by parallel programming, but only at the expense of increasing the code complexity.

The thesis concluded that due to parallel programming being so challenging, it's imperative to determine whether the performance boost outweighs the increasing complexity in the code or not. The lack of viable tools is the source of increasing complexity and the reason why the programmer is forced to understand the low-level logic. Functional programming is currently a good solution as it eliminates most of the challenges by immutability.

Keywords
parallel programming, multicore processor, thread, java

Sisällys

1 Johdanto 1
2 Lyhyt historia 2
3 Moniydinprosessori 3
3.1 Prosessorin muistimalli 3
3.2 Käskyn suorittaminen 4
3.3 Prosessi 4
3.4 Monisäikeistys 6
4 Java Virtual Machine 8
4.1 Toiminta 8
4.2 Muistin rakenne 9
4.3 JVM ja prosessori 11
5 Rinnakkaisohjelmointi 13
5.1 Suoritusjärjestys 13
5.2 Synkronointi 14
5.3 Lukkiutuminen 15
5.4 Nälkiintyminen 15
6 Rinnakkaisohjelmoinnin suunnittelu 17
6.1 Rinnakkaiset tehtävät 17
6.2 Tila 17
6.3 Säieallas 18
6.4 Fork-join 19
6.5 Atomiset muuttujat 19
6.6 Futuuri 21
7 Rinnakkaisohjelmoinnin toteutus 22
7.1 Peräkkäistoteutus 22
7.2 Rinnakkaistoteutus 23
7.3 Toteutustapojen vertailu 28
8 Tulokset 31
9 Pohdinta 32
9.1 Työn käyttömahdollisuus ja opinnäytetyöprosessi 32
10 Lähteet 34

1 Johdanto

Laitevalmistajien siirtyessä moniydinprosessoreihin, sovellusten suorituskyvyn optimointi ja kehittäminen on siirtynyt ohjelmoijan tehtäväksi. Nykyään markkinoilla olevissa prosessoreissa on vähintään kaksi prosessoriydintä ja tulevaisuudessa ytimien määrän on arvioitu kasvavan jopa satoihin ytimiin. Tämän työn tarkoituksena oli selvittää, miten nämä kaikki ytimet pystytään valjastamaan ohjelmoijan käyttöön.

Prosessoriytimien tehokkaaseen hyödyntämiseen tarvitaan rinnakkaisohjelmointia. Rinnakkaisohjelmoinnilla tarkoitetaan sellaisten sovellusten ohjelmointia, joissa useita säikeitä suorittaa tehtäviä rinnakkain.

Työssä keskitytään rinnakkaisohjelmoinnin haasteisiin, suunnitteluun ja toteutukseen. Työ on rajattu käsittelemään Java virtuaalikonetta (JVM) hyödyntäviä ohjelmointikieliä. Työn taustalla oli idea koostaa rinnakkaisohjelmoinnista helposti ymmärrettävä ja havainnollistava johdatus, jota voisi käyttää esimerkiksi kurssimateriaalina.

Työ toteutettiin analysoimalla sähköisiä ja kirjallisia lähteitä. Koska tarkoituksena oli saada aikaan mahdollisimman helposti ymmärrettävä kokonaisuus, käsiteltäviä asioita on pyritty havainnollistamaan kuvioilla ja esimerkkisovelluksilla. Lisäksi työ sisältää yhden laajemman sovellusvertailun, jossa peräkkäisesti toimivaa sovellusta verrataan rinnakkaisesti toimivaan sovellukseen. Esimerkkisovellusten toteutuksessa on käytetty Java-ohjelmointikieltä.

Työ koostuu loogisesta kokonaisuudesta. Rinnakkaisohjelmointia ja siihen liittyviä asiota käydään läpi kokonaisuutena, niin matalalla kuin korkealla abstraktiotasolla. Työ lähtee liikkeelle matalalta abstraktiotasolta ja abstraktiotasoa korotetaan tasaisesti läpi työn. Ensimmäiset aihealueet taustoittavat käsiteltävää aihetta ja lopuksi tietoja käytetään hyväksi suunniteltaessa ja toteuttaessa rinnakkaisohjelmointia.

2 Lyhyt historia

Tietokoneen toimintaperiaate on jo useita vuosikymmeniä perustunut transistorien toimintaan. Transistori on äärimmäisen pieni puolijohdekomponentti, joka voi joko johtaa tai olla johtamatta virtaa. Tietokone näkee yhden transistorin joko nollana tai ykkösenä. (Strickland 2012.)

Yksi tietokoneen suorituskykyyn vaikuttava tekijä on transistorien lukumäärä prosessorin mikrosirulla. 1970-luvulla George Moore, toinen Intelin perustajista ennusti, että mikrosirulle mahtuvien transistorien lukumäärä tuplaantuisi noin kahden vuoden välein. Mooren ennustus osoittautui osuvaksi ja ennustusta alettiin kutsua Mooren laiksi. (Strickland 2012.)

Mooren laki on pitänyt paikkansa useita vuosikymmeniä ja tietokoneiden suorituskyky on kasvanut tasaisesti, Mooren osoittamalla tavalla. 2000-luvun alussa kehitystä alkoivat hidastamaan fyysiset rajoitteet. Muutaman nanometrin kokoisia transistoreja oli sullottu mikrosirulle niin paljon, että tila alkoi loppua kesken. Koska transistorien määrää ei voitu enää helposti kasvattaa, siruvalmistajat päätyivät optimoimaan käytössä olevia transistoreja. Sen sijaan että mikrosirulla oli vain yksi prosessori käsittelemässä tietoa, sirulle lisättiin useita prosessoriytimiä, jotka pystyivät käsittelemään tietoa rinnakkain. (Strickland 2012.)

Moniydinprosessoreihin siirtyminen on ohjelmoijan kannalta kiinnostavaa siksi, että moniydinprosessorit eivät automaattisesti takaa kaksinkertaista tehon lisäystä parin vuoden välein. Sovellusten suorituskyvyn kehittyminen on riippuvainen siitä, osaako ohjelmoija hyödyntää kaikkien prosessoriytimien tehoa. Prosessoriytimien tehokkaaseen hyödyntämiseen tarvitaan rinnakkaisohjelmointia. (Strickland 2012.)

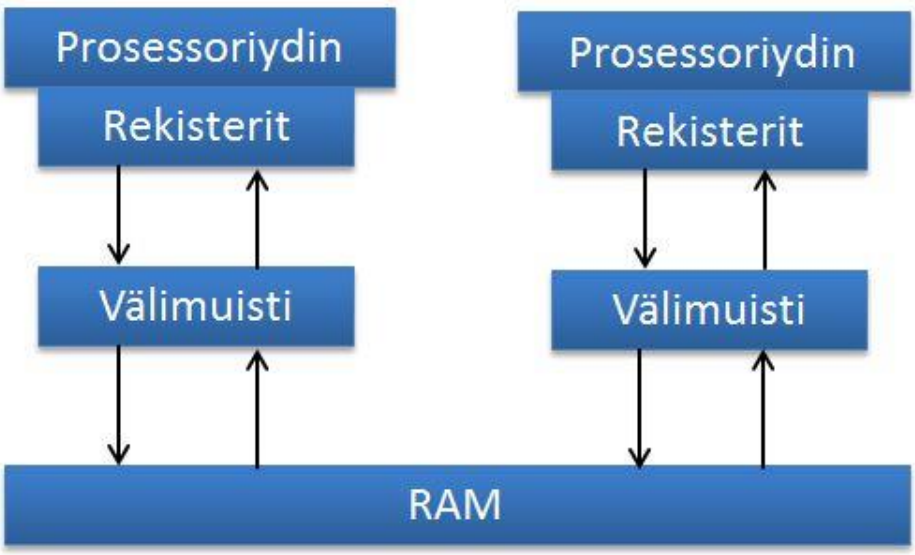
3 Moniydinprosessori

Moniydinprosessori on prosessori, joka koostuu kahdesta tai useammasta prosessoriytimestä. Useat prosessoriytimet tekevät rinnakkaiskäsittelyn mahdolliseksi, koska prosessoriytimet voivat suorittaa eri tehtäviä samanaikaisesti. Prosessorin teho määritetään kellotaajuutena, joka mitataan yleensä joko megahertseinä tai gigahertseinä. Yksi hertsi vastaa yhtä prosessorikäskyä sekunnissa eli esimerkiksi kahden gigahertsin kellotaajuudella varustettu prosessori pystyy suorittamaan 2 miljardia prosessorikäskyä sekunnissa. (Nieminen 2014, 31.)

3.1 Prosessorin muistimalli

Moniytimisen prosessorin muistimalli on kuvattu korkealla tasolla kuviossa 1. Muistimallilla kuvataan muistikomponenteista rakentuva kokonaisuus ja se, miten tietyt muistikomponentit kommunikoivat toistensa kanssa (Jenkov 2010b).

Prosessoriytimen muistirakenne koostuu prosessoriytimen omista rekistereistä ja välimuistista. Rekisterit sijaitsevat kaikkein lähimpänä prosessorin sisäisiä komponentteja, ja siksi tiedon lukeminen ja tallennus on kaikkein nopeinta rekistereissä. Rekistereihin tallennetaan suoritettavan käskyn tiedot. Prosessoriytimen välimuisti sijaitsee rekisterien ja keskusmuistin välissä ja toimii väliaikaisena tiedon säilytyspaikkana. Prosessori hakee käskyn suoritusta varten tiedot rekistereihin, joko väli- tai keskusmuistista. Välimuistissa säilytetään osaa keskusmuistin sisällöstä, koska tiedon hakeminen välimuistista on nopeampaa. (Nieminen 2014, 38–39.)



Kuvio 1. Prosessorin muistimalli (Jenkov 2010b)

Prosessorin muistimalli on tärkeä osa rinnakkaiskäsittelyä. Prosessoriydinten suorittamilla rinnakkaisilla tehtävillä on mahdollisuus rikkoa muistin eheys, esimerkiksi ylikirjoittamalla toisen tekemän päivityksen tai lukemalla vanhentunutta tietoa. Muistien ymmärtämisestä on myös apua sovellusten optimoinnissa. Ohjelmat, joiden käyttämät tiedot löytyvät suurimmaksi osaksi aikaa välimuistista, toimivat todella paljon nopeammin. (Nieminen 2014, 38.)

3.2 Käskyn suorittaminen

Prosessorin toimintaperiaatetta kutsutaan nouto-tulkkaus-suoritussykliksi (fetch-decode-execute cycle). Prosessori voi suorittaa tasan yhden käskyn tietyllä ajanhetkellä. Ennen kuin sykliä aletaan suorittamaan, suoritettava ohjelma ja ohjelmassa käsiteltävät tiedot ladataan keskusmuistiin. (Nieminen 2014, 32–33.)

Syklin aluksi prosessori noutaa seuraavan suoritettavan käskyn ja käskyn suorituksessa vaaditut tiedot keskusmuistista. Prosessori tallentaa suoritusohjeen tiedot omiin sisäisiin rekistereihinsä. Prosessori tulkitsee noudetun käskyn ja valmistelee operaatiot, jotka on tarkoitus suorittaa. Noudettu ja tulkattu käsky suoritetaan ja tulos tallennetaan yhteen prosessorin rekistereistä. Lopuksi tallennettu tulos siirretään prosessorin rekisteristä takaisin keskusmuistiin, jotta muut ohjelman osat voivat käyttää päivitettyä tietoa. Prosessori suorittaa tätä sykliä prosessorin kellotaajuudesta riippuen useita miljardeja kertoja sekunnissa. (Nieminen 2014, 32–33.)

3.3 Prosessi

Käyttöjärjestelmistä ja prosessoreista puhuttaessa, ajettavan ohjelman abstraktiona käytetään termiä prosessi. Jokaisella prosessilla on oma kontekstinsa, joka sisältää prosessin tilan ja prosessin käyttämän datan. Jos käytössä on yksiprosessorijärjestelmä, vain yhden prosessin konteksti voi olla muuttuvassa tilassa kerrallaan. Loput prosessit ja niiden kontekstit ovat jäädytettyinä tallessa. (Nieminen 2014, 60–61.)

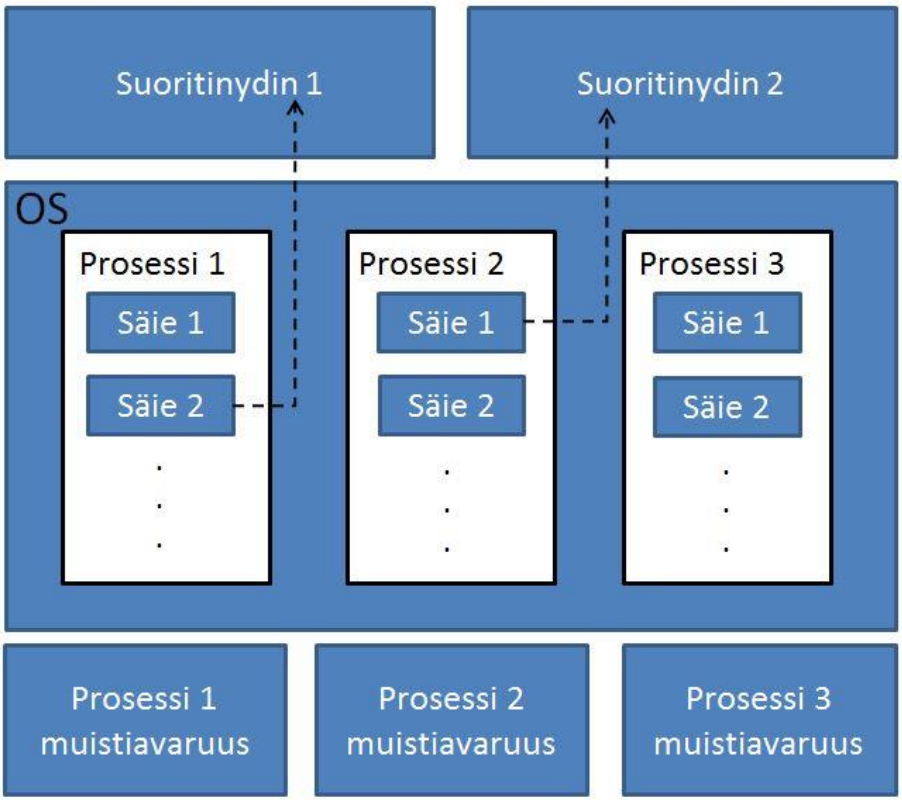
Prosessien suoritusjärjestyksen määrää tietokoneen käyttöjärjestelmän vuorontaja (scheduler). Vuorontaja jakaa prosesseille aikaviipaleita. Aikaviipale tarkoittaa käytännössä prosessin saamaa prosessoriaikaa. Vuoronnus perustuu jonomenettelyyn, jossa tehokkaat algoritmit varmistavat, että kaikki prosessit saavat prosessoriaikaa tasavertaisesti. (Nieminen 2014, 85.)

Prosessi koostuu yhdestä tai useammasta säikeestä (thread). Säie on pienin suoritettava yksikkö tietokoneessa ja säiettä kutsutaan myös kevyeksi prosessiksi. Säikeitä käytetään esimerkiksi siksi, että säikeiden luominen ja tuhoaminen on huomattavasti nopeampaa, verrattuna prosessien luomiseen ja tuhoamiseen. Koska prosessi koostuu vähintään yhdestä säikeestä, vuoronnuksessa on todellisuudessa kyse prosessien sisällä toimivien säikeiden vuoronnuksesta. (Nieminen 2014, 87–88.)

Jokaisella prosessilla on oma muistiavaruutensa, joka rakentuu muistipaikoista. Esimerkiksi yksi muistiavaruus voi olla muistipaikat nollasta tuhanteen. Prosessin käyttämät tiedot on tallennettu mustiavaruuden muistipaikkoihin. Prosessi ei pysty käyttämään muiden prosessien muistiavaruuksia. Yhden prosessin sisällä toimivat säikeet jakavat prosessin muistiavaruuden. (Nieminen 2014, 57.)

Prosessien käyttämät muistiavaruudet ovat virtuaalisia. Prosessit eivät siis näe käyttävänsä laitteen fyysistä muistia vaan muistiosoitteet ovat loogisia osoitteita. Prosessori mäppää virtuaalimuistiavaruudessa olevan muistiosoitteen fyysiseksi muistiosoitteeksi. (Nieminen 2014, 57.)

Prosessorin, käyttöjärjestelmän ja muistin yhteistyötä on havainnollistettu kuviossa 2.



Kuvio 2. Prosessorin, muistin ja käyttöjärjestelmän yhteistyö (Prokopec 2014)

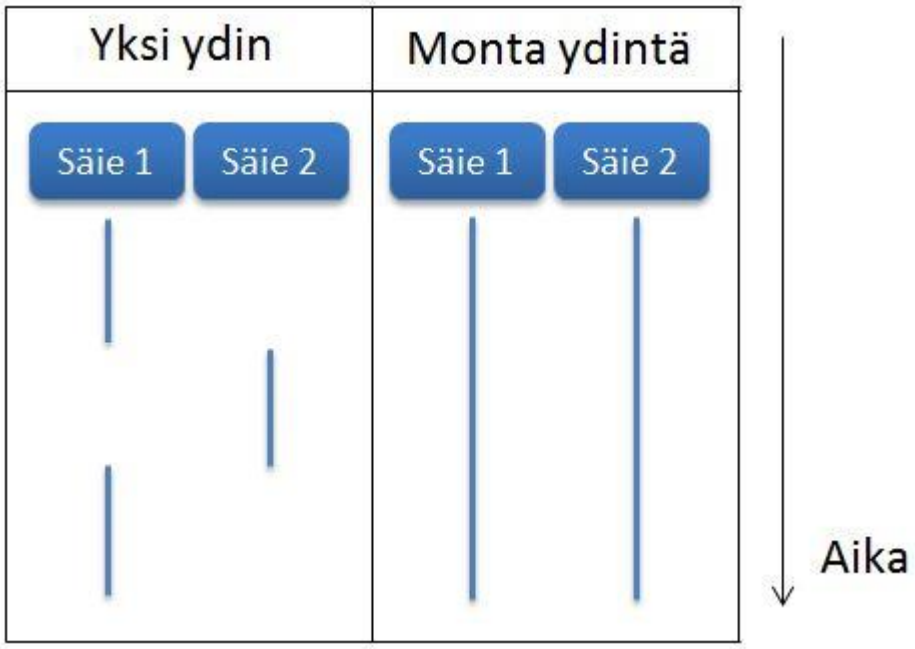
Kuviossa käyttöjärjestelmällä on käynnissä kolme prosessia. Jokaisella prosessilla on oma muistiavaruus, johon muut prosessit eivät pääse käsiksi. Yksi prosessi koostuu säikeistä. Yksi suoritinydin voi suorittaa yhden säikeen suoritusohjeen kerrallaan.

3.4 Monisäikeistys

Jos prosessi koostuu kahdesta tai useammasta säikeestä, prosessia kutsutaan monisäikeiseksi (multithreaded). Monisäikeisyyttä voi hyödyntää sekä yksi- että moniytimisissä prosessoreissa. (Bloch ym. 2006, 3-4.)

Yksiytimisessä prosessorissa, monisäikeisyyden avulla voidaan simuloida samanaikaisuutta. Esimerkiksi jos yksi prosessi koostuu kahdesta säikeestä, säikeet vuorottevat suoritusaikoja äärimmäisen nopeasti. Säikeestä vaihtamista toiseen säikeeseen kutsutaan kontekstin vaihdoksi. (Bloch ym. 2006, 3-4.)

Moniytimisissä järjestelmissä saadaan aikaan aitoa rinnakkaisuutta. Esimerkiksi kaksiytiminen prosessori voi suorittaa kahta säiettä samanaikaisesti. Kuvio 3 havainnollistaa tärkeimmän eron yksi- ja moniydinprosessorien välillä.



Kuvio 3. Monisäikeistys yhdellä tai useammalla prosessoriytimellä

Kuviossa kuvataan se, että yhdellä ytimellä ainoastaan yksi säie voi kerrallaan olla suoritettavana. Säikeiden vuorottelu on kuitenkin niin nopeaa, että monisäikeistys pystyy simuloimaan samanaikaisuutta, vaikka oikeasti mitään ei tapahdu täysin samaan aikaan. Moniytimisellä prosessorilla kaksi tai useampaa säiettä voi olla suorituksessa rinnakkain.

Kuviosta voi päätellä, että teoriassa rinnakkain suoritettava sovellus on kaksi kertaa nopeampi kuin peräkkäin suoritettava sovellus. Todellisuudessa hyöty ei välttämättä ole tasan kaksinkertainen, koska monen säikeen rinnakkainen hallinnointi aiheuttaa lisälaskentaa. (Nieminen 2014, 82–83.)

4 Java Virtual Machine

Useissa ohjelmointikielissä ohjelmat käännetään suoraan prosessorin ymmärtämäksi konekieleksi. Javassa välissä käytetään Java virtuaalikonetta (JVM). JVM abstraktoi prosessorin toiminnan käyttämällä JVM:lle suunniteltua tavukoodia. Vennersin (1999) mukaan toimiva JVM koostuu kolmesta osasta: abstrakti määritys, konkreettinen toteutus ja ajettava instanssi. Abstrakti määritys on kirjallinen teos, jossa määritetään JVM:n vaatimukset. Konkreettinen toteutus on sovellus, joka toteuttaa abstraktin määrityksen. Ajettava instanssi on käyttöjärjestelmäprosessi, jonka avulla voidaan ajaa yksi ohjelma.

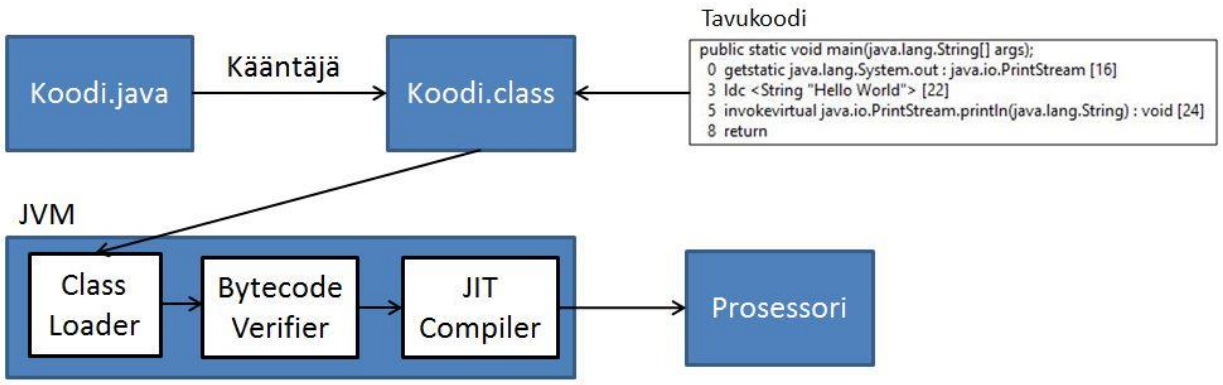
JVM-toteutuksia on moneen eri tarkoitukseen. Toteutukset eroavat toisistaan esimerkiksi ominaisuuksien tai suorituskyvyn perusteella, mutta jokaisen JVM:n pitää kuitenkin toteuttaa sama abstrakti määritys. Tässä työssä JVM:sta puhuttaessa tarkoitetaan JDK:n mukana tulevaa Hotspot-toteutusta. (Slangen 2012.)

4.1 Toiminta

Java virtuaalikonetta käyttävät ohjelmointikielet ovat korkean tason ohjelmointikieliä. Korkean tason ohjelmointikieli tarkoittaa sitä, että osa alemman tason logiikasta on abstraktoitu ohjelmoijalta. Esimerkiksi Java-ohjelmaa kirjoittava ohjelmoija muokkaa vain java-päätteistä tiedostoa ja ohjelmoijan ei tarvitse olla tietoinen, mitä tiedostolle tapahtuu, kun se suoritetaan. (Slangen 2012.)

Mitä siis kooditiedoston suorituksessa tapahtuu? Java-päätteinen tiedosto ajetaan kääntäjästä läpi, minkä seurauksena Java-tiedostosta generoidaan class-tiedosto. Class-tiedosto sisältää alkuperäisestä Java-koodista käännetyn tavukoodin. JVM sisältää komponentin nimeltä Class Loader, joka lataa kaikki class-päätteiset tiedostot JVM:lle suoritettavaksi. Kun class-tiedosto on noudettu, JVM tarkastaa tiedoston tavukoodin Bytecode Verifier -komponentilla. Tarkistettu class-tiedosto siirretään JIT (Just-In-Time) -kääntäjälle. JIT tarkoittaa sitä, että kääntäminen tehdään sovelluksen ajohetkellä. JIT-kääntäjä kääntää class-tiedoston tavukoodista prosessorin ymmärtämää konekieltä, jota prosessori voi suorittaa. (Slangen 2012.)

Korkean tason ohjelmointikielen kääntyminen konekieleksi JVM-ympäristössä on esitetty kuviossa 4.



Kuvio 4. Korkean tason ohjelmointikielestä konekieleksi

Kuviossa käytetään esimerkkinä tiedostoa Koodi.java. Koodi.java kulkee kääntäjän läpi, minkä seurauksena kääntäjä generoi tiedostosta Koodi.class-tiedoston. Koodi.class sisältää käännetyn tavukoodin. Koodi.class kulkee JVM:n läpi ja lopputuloksena tavukoodista generoidaan prosessorin ymmärtämää konekieltä.

4.2 Muistin rakenne

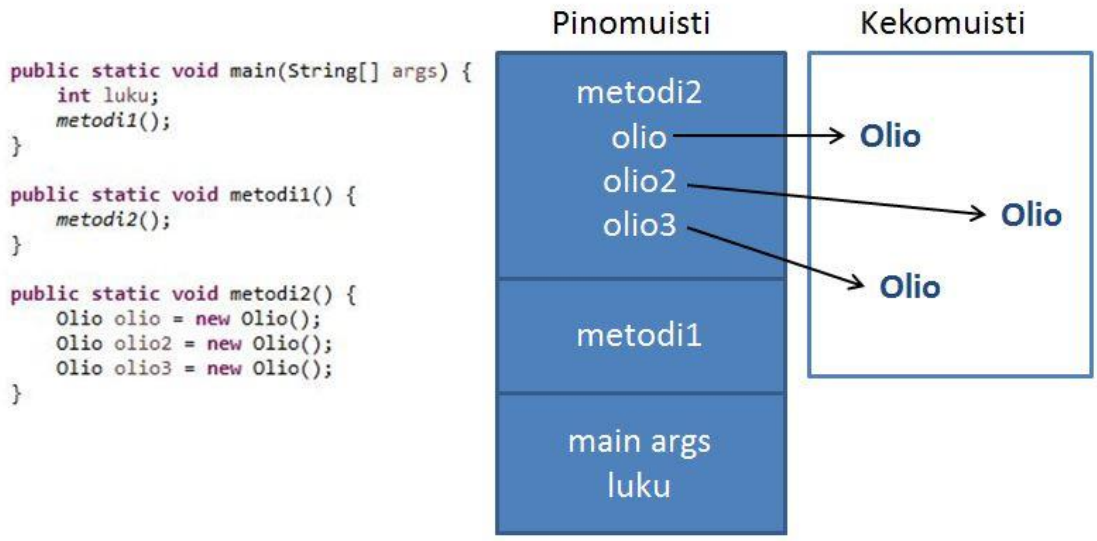
Käyttöjärjestelmä varaa JVM:lle palan muistia, jota JVM käyttää erilaisiin tarkoituksiin. 32-bittiset käyttöjärjestelmät tukevat maksimissaan neljän gigatavun muistiviipaletta. Muistista osa menee käyttöjärjestelmän vaatimiin ylläpitoresursseihin, joten muistiviipaleen todellinen koko ei ole edes neljää gigatavua. 64-bittiset käyttöjärjestelmät tukevat huomattavasti suurempaa määrää muistia. (Bailey 2012.)

JVM-instanssin käynnistäminen luo uuden prosessin, jonka sisällä useat säikeet voivat toimia. Useissa ohjelmointikielissä, kuten Pythonissa, käytetään ohjelmointikielen omia säikeitä, mutta JVM-kielissä, säie mäppäytyy suoraan käyttöjärjestelmän säikeeksi. Suora mäppäytyminen tarkoittaa sitä, että JVM-säikeet toimivat hyvin samantapaisesti kuin natiivit käyttöjärjestelmäsäikeet. (Prokopec 2014.)

JVM:n muistikokonaisuudessa on kaksi tärkeää osaa: kekomuisti (heap) ja pinomuisti (stack). Keko on muistialue, joka on jaettu ohjelman säikeiden kesken. Keossa säilytetään kaikkia ohjelman käyttämiä olioita. Pinomuistiin tallennetaan kaikki metodikutsut ja metodikutsujen sisällä käytetyt muuttujat. Jokaisella ohjelmaa suorittavalla säikeellä on oma pinonsa. Kun säie käsittelee olioita, se tallentaa pinoon viittauksen keossa olevaan olioon. Yksittäinen säie pääsee käsiksi ainoastaan omaan pinoonsa. Rinnakkaiset säikeet eivät siis pysty näkemään toistensa luomia muuttujia eli esimerkiksi jos kumpikin säie

suorittaa samaa metodia samanaikaisesti, kumpikin säie luo kaikki metodissa käytetyt muuttujat omaan pinoonsa. (Kumar 2014.)

Kuviossa 5 havainnollistetaan, miten ohjelman käyttämät metodit ja muuttujat asettuvat muistirakenteisiin.



Kuvio 5. JVM:n käyttämät muistirakenteet

Esimerkkinä käytetään ohjelmaa, joka sisältää useita metodeja, olioita ja paikallisen muuttujan. Ohjelma alkaa main-metodikutsulla, jolloin suorittavan säikeen pinoon asetetaan uusi pinokehys (stack frame). Pinokehys sisältää metodin nimen ja metodin käyttämät muuttujat. Ohjelman main-metodissa käytetään primitiivistä kokonaislukumuuttujaa nimeltä luku. Luku-muuttuja tallennetaan main-metodin pinokehykseen.

Main-metodin sisällä kutsutaan uutta metodia nimeltä metodi1. Jotta ohjelman suoritus voi jatkua, koodissa pitää siirtyä suorittamaan uutta metodia. Metodi1 asetetaan pinon päällimmäiseksi, main-metodin yläpuolelle.

Metodi1:n sisällä kutsutaan metodi2:ta, joten metodi2:n pinokehys asetetaan pinon päällimmäiseksi. Metodi2:ssa luodaan kolme uutta oliota. Luodut oliot tallennetaan kekomuistiin. Metodi2:n pinokehykseen tallennetaan viittausmuuttujat keossa oleviin olioihin.

Sitä mukaa kun metodit saadaan suoritettua, metodien pinokehykset poistetaan säikeen pinosta ja säie siirtyy suorittamaan jäljelle jäävän pinon päällimmäistä pinokehystä. Kun

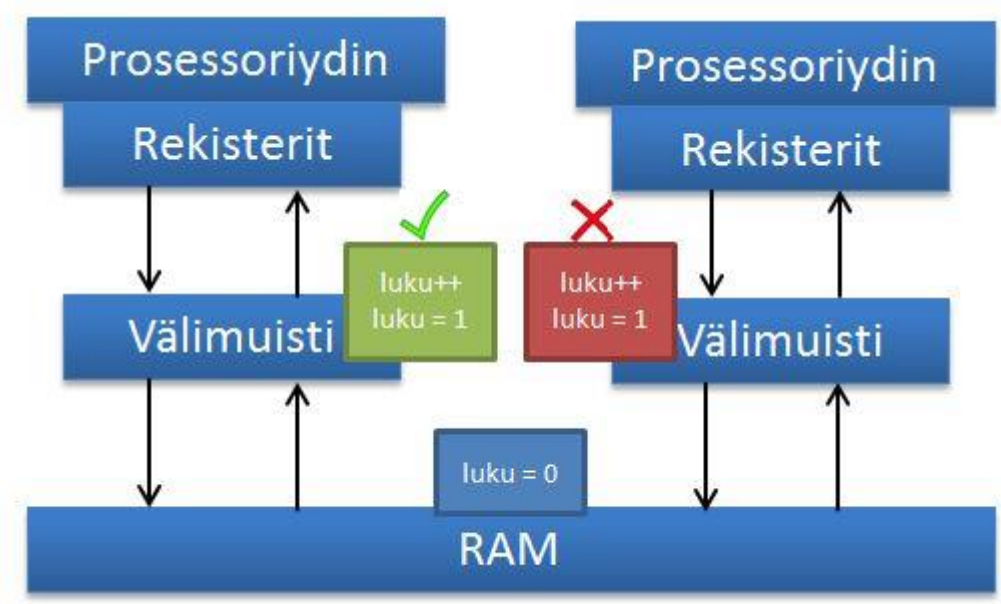
pinokehys poistetaan, pinokehyksessä olevat muuttujat häviävät. Esimerkiksi kun metodi2:n pinokehys otetaan pois, viittaukset keossa oleviin oliohin häviävät. Kun keossa olevaan olioon ei enää viitata mistään pinosta, olio voidaan poistaa keosta. (Bailey 2012.)

4.3 JVM ja prosessori

JVM on kuitenkin vain prosessorin abstraktio, eikä prosessori erottele keossa tai säikeiden pinoissa olevia asioita. Kaikki tiedot tallennetaan samaan paikkaan eli keskusmuistiin. Osa tallennetusta tiedosta voi olla väliaikaisesti prosessorin välimuistissa tai prosessorin sisäisissä rekistereissä. Ongelmaksi muodostuu muuttujien näkyvyys eri säikeille.

On useita eri syitä, miksi säie ei välttämättä voi nähdä heti toisen säikeen laskemaa tulosta. Kääntäjä saattaa generoida suoritettavat ohjeet eri järjestyksessä lähdekoodiin verrattuna, muuttujia saatetaan tallentaa prosessorin rekistereihin keskusmuistin sijaan tai prosessoriytimet saattavat suorittaa suoritusohjeita rinnakkain tai eri järjestyksessä. Lisäksi prosessoriytimien välimuistit saattavat muuttaa järjestystä, missä järjestyksessä muutokset kirjoitetaan keskusmuistiin. (Bloch ym. 2006, 207.)

Muuttujien näkyvyyteen liittyvää ongelmaa on kuvattu kuviossa 6.



Kuvio 6. Muuttujien näkyvyys (Jenkov 2010b)

Kuviossa käytetään esimerkkinä kokonaislukumuuttujaa luku. Luku-muuttujan alkuarvoksi asetetaan nolla, ja muuttujan alkuarvo tallennetaan keskusmuistiin. Ohjelmassa luodaan kaksi säiettä, jotka kumpikin korottavat luku-muuttujan arvoa yhdellä. Ensimmäisen

säikeen pinossa oleva luku-muuttuja ladataan prosessoriytimen välimuistiin. Luku-muuttujan arvoa korotetaan yhdellä (luku++), säikeen suoritusohjeen mukaisesti. Luku-muuttujan oikea arvo on siis nyt yksi, mutta muuttujan päivitetty arvo on näkyvissä ainoastaan prosessoriytimen välimuistissa. Jos rinnakkainen säie lukee luku-muuttujan arvon keskusmuistista, ennen kuin päivitetty arvo on ehditty kirjoittaa keskusmuistiin, rinnakkainen säie näkee edelleen muuttujan arvona nollan. Tällaisissa tilanteissa rinnakkaiset säikeet rikkovat muistin eheyden.

5 Rinnakkaisohjelmointi

Rinnakkaisohjelmointi tarkoittaa sitä, että ohjelmoidaan sovellus, jossa kaksi tai useampia säikeitä suorittaa ohjelmaa rinnakkain. Aitoa rinnakkaisuutta saadaan aikaan vain moniydinprosessoreilla. Rinnakkaisuutta ohjelmoitaessa pitää osata huomioida sovelluksen suoritusjärjestys ja siitä aiheutuvat ongelmat muistin kanssa.

5.1 Suoritusjärjestys

Rinnakkaisella sovelluksessa ei ole yhtä määritettyä suoritusjärjestystä vaan ohjelman suoritusjärjestys on jokaisella suorituskerralla satunnainen. Satunnaisuus aiheuttaa ongelmia sovelluksen tilan hallinnan kanssa. Esimerkiksi jos useita säikeitä käsittelee samaa muuttujaa, suoritusjärjestys määrää muuttujan lopullisen arvon. Kuviossa 7 on esitetty yksinkertainen esimerkki sovelluksesta, jossa useita säikeitä käsittelee samaa muuttujaa. (Bloch ym. 2006, 5.)

```
public class Lasku implements Runnable {

    private int luku = 0;

    public void run() {
        luku = luku + 1;
        System.out.print(luku+ " ");
    }

    public static void main(String[] args) {
        Lasku lasku = new Lasku();
        for (int i = 0; i < 10; i++) {
            new Thread(lasku).start();
        }
    }
}
```

Kuvio 7. Sovelluksen tilan hallinta useilla säikeillä

Sovelluksessa määritetään luku-niminen muuttuja, jonka alkuarvoksi asetetaan nolla. Main-metodissa luodaan uusi instanssi ajettavasta luokasta Lasku, joka toteuttaa Runnable-rajapinnan. Runnable-rajapinnan avulla Lasku-luokan voi antaa Thread-luokan konstruktorille parametriksi. For-loopissa luodaan kymmenen säiettä, jotka kaikki suorittavat Lasku-luokan run-metodin. Tavoitteena on kasvattaa luku-muuttujaa jokaisella kierroksella yhdellä, joten tulokseksi halutaan luku-muuttujan printtaus yhdestä kymmeneen.

Sovelluksen antama tulos ei vastaa haluttua tulosta. Oheisessa taulukossa 1 on sovelluksen antamia tuloksia eri suorituskerroilta.

Taulukko 1. Esimerkkisovelluksen tuloksia eri suorituskerroilta

Suorituskerta	Sovelluksen antama tulos
1	1 2 4 3 5 6 7 8 9 10
2	2 4 3 2 5 6 7 8 9 10
3	1 2 3 4 5 6 7 8 9 10
4	1 2 2 3 4 3 5 6 7 8
5	1 2 4 5 5 6 7 8 9 10

Sovelluksen suorituskerrasta riippuen, tietyt säikeet pääsevät suorittamaan oman run-metodinsa aiemmin kun toiset. Suoritusjärjestyksen satunnaisuuteen vaikuttaa kääntäjän tekemä optimointi ja käyttöjärjestelmän vuorontajan tekemä säikeiden vuoronnus. Lisäksi prosessorin muistinhallinta on selitys useille sovelluksen antamille oudoille tuloksille. Esimerkiksi joidenkin säikeiden pinoissa olevat muuttujat haetaan prosessorin välimuistiin ja pidetään siellä niin kauan, että useita muita säikeitä on ehtinyt jo kirjoittaa oman tuloksensa keskusmuistiin. (Bloch ym. 2006, 207.)

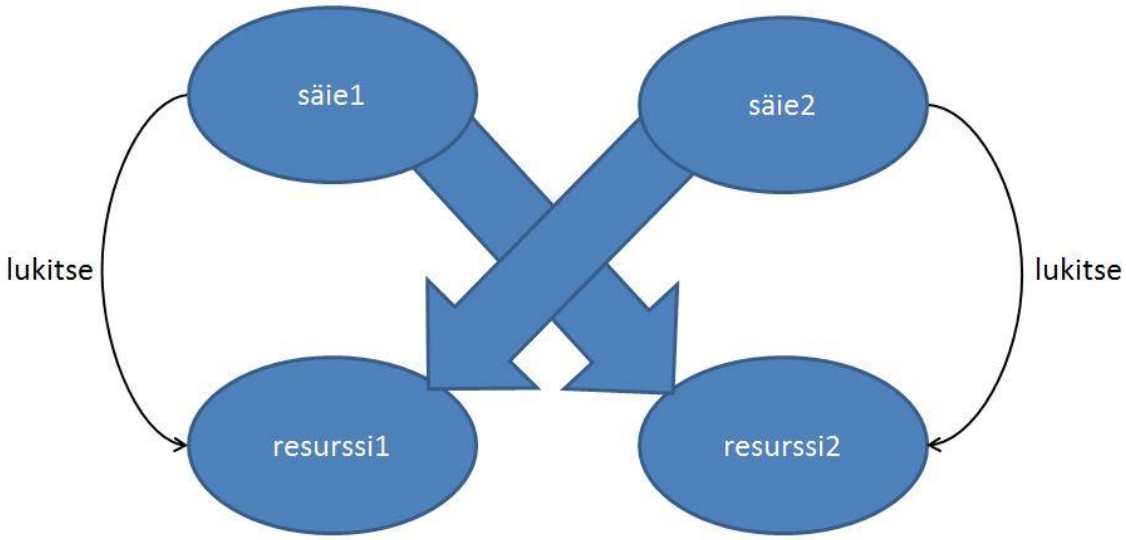
5.2 Synkronointi

Sovelluskoodissa olevia alueita, joissa suoritusjärjestys määrää lopullisen tuloksen, kutsutaan kriittisiksi alueiksi. Kriittisen alueen voi tunnistaa siitä, että useat säikeet muokkaavat sovelluksen tilaa. Jotta kriittisen alueen suorittaminen olisi turvallista, kriittinen alue pitää synkronoida. Synkronointi perustuu siihen, että kun koodi on synkronoitu, vain yksi säie voi suorittaa kerrallaan synkronoitua koodialuetta. Suorittava säie hankkii synkronoidun resurssin lukon itselleen ja pitää sen itsellään niin kauan että suoritus on päättynyt ja muutokset on tallennettu jaettuun muistiin. (Jenkov 2010c.)

Lasku-esimerkin tapauksessa luku-muuttuja ja muuttujaa päivittävä metodi lukitaan kerrallaan vain yhden säikeen käyttöön. Muuttuja vapautuu seuraavan säikeen käyttöön vasta kun edellinen säie on korottanut luku-muuttujan arvoa yhdellä ja tallentanut muuttamansa arvon keskusmuistiin.

5.3 Lukkiutuminen

Synkronointiin liittyy myös ongelmia. Jos kaksi samanaikaista säiettä lukitsee itselleen jonkin resurssin, jota toisen säikeen pitäisi päästä käyttämään, päädytään lukkiutumiseen. Kuviossa 8 esitetään lukkiutumisen perus idea. (Bloch ym. 2006, 128.)



Kuvio 8. Lukkiutuminen

Ensimmäinen säie lukitsee resurssi1:n itselleen ja tarvitsee jatkaakseen resurssi2:ta. Toinen säie lukitsee samanaikaisesti itselleen resurssi2:n ja tarvitsee jatkaakseen resurssi 1:tä. Sovellus ei voi mitenkään jatkaa suoritusta, koska säikeet odottavat loputtomiin toistensa lukitsemien resurssien aukeamista.

Esimerkiksi tietokantaympäristössä, tietokannanhallintajärjestelmä huolehtii siitä, että tietokanta pystyy toipumaan lukkiutumisesta, mutta sovelluksen tapauksessa, sovellus ei pysty itse toipumaan lukkiutumisesta. Ainoa tapa päästä eroon lukkiutumisesta on käynnistää sovellus uudelleen. Tämän takia synkronointi pitää suunnitella huolellisesti. (Bloch ym. 2006, 128.)

5.4 Nälkiintyminen

Jos säikeelle ei voida antaa prosessoriaikaa, koska kaikki muut säikeet käyttävät kaiken prosessoriajan, säikeen pitkää odottamista kutsutaan nälkiintymiseksi (starvation). Nälkiintymistä aiheuttaa säikeiden priorisointi ja synkronointi. (Jenkov 2010d.)

Priorisoinnilla tarkoitetaan sitä, että säikeelle annetaan prioriteetti yhdestä kymmeneen. Korkeammalla prioriteetilla olevat säikeet saavat enemmän prosessoriaikaa kuin

alhaisemalla prioriteetilla olevat säikeet. Prioriteettien tulkitseminen riippuu käyttöjärjestelmästä. (Jenkov 2010d.)

Synkronointi aiheuttaa nälkiintymistä, kun säikeet yrittävät päästä käsiksi synkronoituun osaan koodista. Synkronoitu koodi voi olla esimerkiksi jokin laskennallisesti raskas toimenpide. Kun yksi säie suorittaa toimenpidettä kaikki muut säikeet joutuvat odottamaan suoritusvuoroaan. (Jenkov 2010d.)

6 Rinnakkaisohjelmoinnin suunnittelu

Rinnakkaisuuden hyödyntämiseen liittyy paljon haasteita, jotka vaikeuttavat rinnakkaisohjelmoinnin suunnittelua. Ohjelmoijan pitää pystyä koordinoimaan säikeiden suoritusta niin, että rinnakkaiset säikeet eivät pysty rikkomaan muistin eheyttä. Säikeiden hallinnointia varten on onneksi kehitetty erilaisia lähetymistapoja, suunnittelumalleja ja hyödynnettäviä tietorakenteita.

6.1 Rinnakkaiset tehtävät

Ennen kuin rinnakkaisia tehtäviä kannattaa etsiä, pitää ymmärtää ratkaistava ongelma: onko ongelma mahdollista jakaa pienempiin osaongelmiin, ja onko näillä osilla riippuvuuksia toistensa kanssa. Kaikkia ongelmia ei ole mahdollista ratkaista rinnakkaisesti, koska jotkin ongelmat vaativat selvän suoritusjärjestyksen. Esimerkiksi Fibonaccin lukujonon generointi rinnakkaisesti ei ole käytännöllistä, koska jokainen luku on riippuvainen edellisistä laskuista. (Barney 2014.)

Rinnakkaisten tehtävien löytäminen perustuu siihen, että suurempi toiminnallinen kokonaisuus pilkotaan pienemmiksi itsenäisiksi osiksi. Esimerkiksi jos ajatellaan suoritettavana kokonaisuutena iteroimista listan läpi, joka sisältää miljoona elementtiä. Jos käytössä olisi neljä säiettä, olisi kätevää, että jokainen säie voisi suorittaa oman neljäsosansa miljoonasta rinnakkain. Listan läpikäyminen olisi teoriassa neljä kertaa nopeampaa.

Ratkaistavaa ongelmaa analysoidessa on otettava huomioon myös se, että onko ongelmaa edes järkevä rinnakkaistaa. Jos kyseessä on hyvin kevyt tehtävä, rinnakkaistus saattaa jopa hidastaa ohjelman suoritusta, koska useiden säikeiden vuoronnuksesta aiheutuu lisälaskentaa. (Bloch ym. 2006, 7.)

6.2 Tila

Kaikki rinnakkaissuoritukseen liittyvät ongelmat liittyvät sovelluksen tilaan, joten sovelluksen tilan huomiointi on äärimmäisen tärkeää. Subramaniamin (2011, 35–39) mukaan tilaa voi lähestyä kolmella eri tavalla.

Ensimmäinen tapa on käyttää jaettua muuttuvuutta eli annetaan minkä tahansa säikeen muuttaa mitä tahansa muuttujaa. Jaetun muuttuvuuden käyttäminen on hyvin tavallinen ohjelmointityyli, mutta jaettu muuttuvuus asettaa ongelman muistin eheyden ylläpitämisestä. Sovelluksessa pitää varmistaa, että muuttujien tila pysyy eheänä ja ainoa

ratkaisu tähän ongelmaan on synkronointi. Syy, miksi tämä ei ole kovin hyvä lähestymistapa on se, että synkronointi hävittää suurimman osan rinnakkaistuksen tehosta.

Toinen lähestymistapa on käyttää eristettyä muuttuvuutta. Eristetyn muuttuvuuden idea on se, että muuttujia voi muuttaa, mutta muuttujat eivät näy kuin yhdelle säikeelle kerrallaan. Tilan muuttaminen on siis eristetty. Kaikille säikeille näkyvät muuttujat pitää suunnitella muuttumattomiksi.

Viimeinen vaihtoehto on täysi muuttumattomuus. Kun käytetään muuttumattomuutta, sovelluksella ei ole tilaa ollenkaan, vaan jokaiselle muuttujalle annetaan lopullinen arvo initialisoinnin yhteydessä. Muuttumattomuutta varten suunnittelu ei ole helppoa, koska ongelmia ei yleensä pysty pilkkomaan sopivan kokoisiksi, pysyviksi paloiksi. Mutta jos muuttumattomuus onnistutaan saavuttamaan, rinnakkaistus on helpompi toteuttaa ja sovelluksella ei ole vaaraa rikkoa muistin eheyttä.

6.3 Säieallas

Jos sovelluksen pitää suorittaa suuri määrä raskaita samanaikaisia tehtäviä, uuden säikeen luominen jokaista tehtävää varten on todella hidasta. Uuden säikeen luominen vaatii muistin varaamista säikeen pinolle ja kontekstin vaihdon säikeestä toiseen. Pelkästään muistin varaaminen ja kontekstin vaihto voi olla hitaampaa kuin itse suoritettava tehtävä. Tämän takia on hyvä käyttää säieallasta. (Prokopec 2014.)

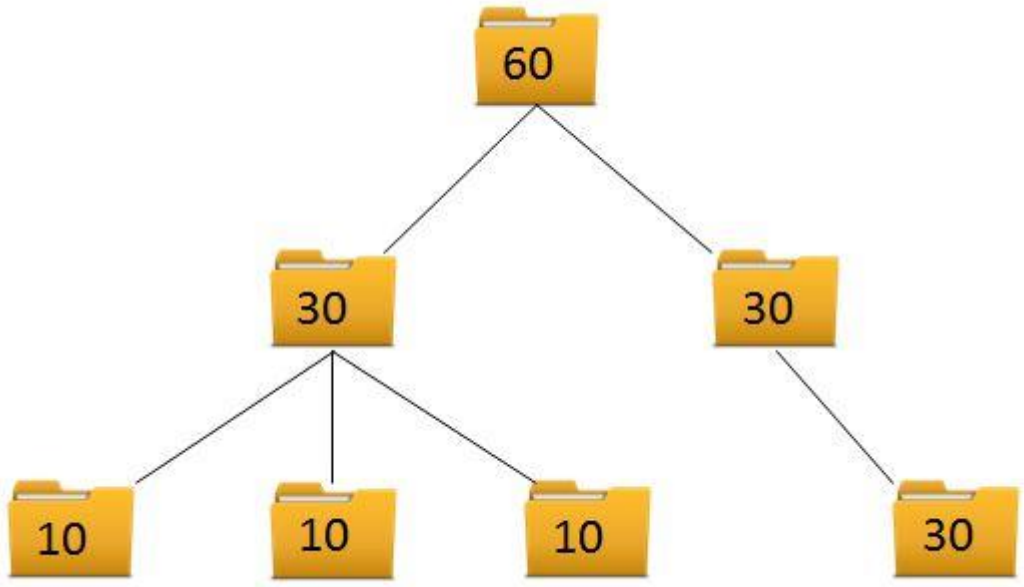
Säieallas (thread pool) on sovelluksessa käytettävien säikeiden säilytyspaikka. Sovelluksen käynnistyessä voidaan ennaltamäärätä käytettävien säikeiden määrä ja luoda säieallas. Aina kun sovelluksen pitää suorittaa jokin tehtävä, tehtävä mäpätään yhteen altaassa odottavaan säikeeseen. Tehtävän suorituksen jälkeen säie palaa altaaseen odottamaan uutta tehtävää. (Bloch ym. 2006, 105–106.)

Säiealtaan kokoa suunniteltaessa, pitää ottaa huomioon käytettävä laitteisto, muistin määrä ja minkä tyyppisiä tehtäviä ohjelmassa suoritetaan. Blochin ym. (2006, 106) mukaan laskennallisissa sovelluksissa altaan optimaalinen koko vastaa prosessoriytimien määrää, koska laskennallisissa sovelluksissa kaikki säikeet ovat koko ajan käytössä. Ulkopuolista interaktiota vaativissa sovelluksissa kannattaa käyttää suurempaa säieallasta, koska tällaisissa sovelluksissa kaikki säikeet eivät ole vuoronnettavissa samanaikaisesti, esimerkiksi synkronoitujen koodialueiden takia.

Säiealtaan kokoa suunniteltaessa, pitää myös huomioida muut sovelluksen käyttämät rajatut resurssit. Esimerkiksi kun sovelluksessa hyödynnetään tietokantaa, sovellukselle määritetään tietokantayhteyksiä sisältävän altaan maksimi koko. Säiealtaan koko ei kannata olla suurempi kuin yhteysaltaan koko, jos sovelluksen jokainen tehtävä vaatii oman tietokantayhteytensä. (Bloch ym. 2006, 106.)

6.4 Fork-join

Fork-join on yksi sovelluksen rinnakkaistuksen toteutusmalli. Fork-joinin idea on se, että ratkaistava ongelma haarautetaan (fork) itsenäisiksi osiksi. Haarautettu osa annetaan säikeelle suoritettavaksi, jolloin säie laskee oman osatuloksensa kokonaisuudesta. Lopuksi kaikkien säikeiden osatulokset yhdistetään (join) muodostamaan lopullinen tulos. (Lea 2006.)



Kuvio 9. Fork-join

Kuviossa 9 on annettu esimerkki fork-joinista. Esimerkissä halutaan tietää tietyn kansion sisältämien tiedostojen käyttämä tallennustila. Kansiohierarkia jakautuu yleensä useisiin eri alikansioihin, joten laskentaa voidaan rinnakkaistaa niin, että säikeille annetaan omia alikansiohierarkioita laskettavaksi. Säikeet laskevat alikansioiden käyttämät tallennustilat ja lopuksi tulokset yhdistetään kertomaan lopullinen tulos.

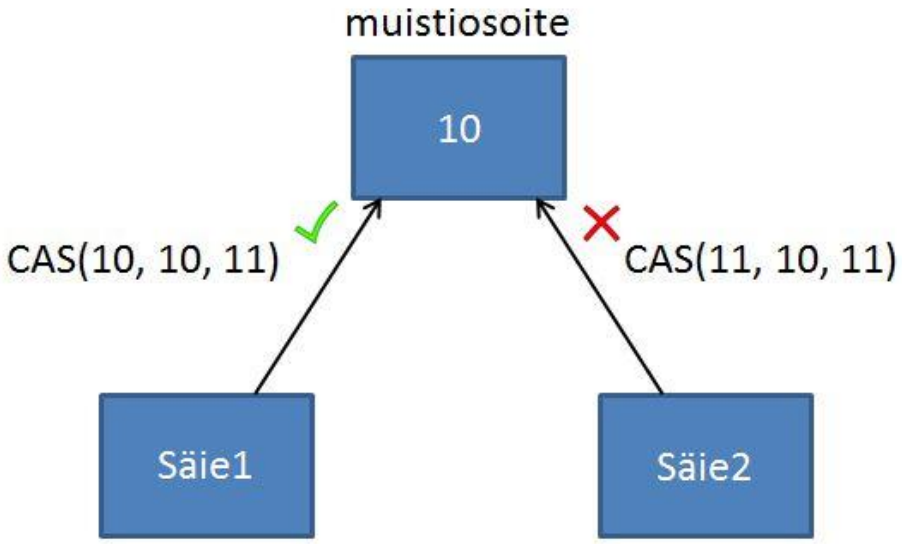
6.5 Atomiset muuttujat

Kun useita säikeitä käyttää samaa muuttujaa, muuttujaan käsiksi pääsy pitää synkronoida. Muuttujan lukitsemiseen liittyy ongelmia, kuten säikeiden pysähtyminen ja

säikeiden lukitsemiseen liittyvä ylimääräinen laskenta. Synkronoinnin voi välttää käyttämällä atomista muuttujaa. Atominen muuttuja on muuttuja, joka käyttää atomista operaatiota eli operaatiota, joka näkyy järjestelmälle ainoana suoritettavana operaationa tietyllä ajanhetkellä. (Wicht 2010.)

Atomisuus perustuu prosessorin tarjoamaan CAS-operaatioon (compare-and-swap). CAS ottaa kolme parametria: muistiosoitteen arvon, odotetun arvon ja uuden arvon. Muistiosoitteen arvo vaihdetaan uuteen arvoon ainoastaan, jos muistiosoitteen nykyinen arvo vastaa odotettua arvoa. Kun useat säikeet yrittävät päivittää atomista muuttujaa, vain yksi säie pystyy muuttamaan sitä kerrallaan. Tärkein ero synkronointiin verrattuna on se, että rinnakkaiset säikeet eivät ole missään vaiheessa pysäytettyinä. Kun toinen säie epäonnistuu muuttujan päivitysyrityksessä, säie voi yrittää tehdä päivityksen uudelleen tai tehdä jotain muuta. (Wicht 2010.)

Kuviossa 10 on esitetty atomisen muuttujan toimintaperiaate.



Kuvio 10. CAS

Esimerkissä muistiosoitteen alkuperäinen arvo on 10 ja kaksi säiettä haluaa päivittää samanaikaisesti arvoksi 11. Säie1 suorittaa CAS:n ensimmäisenä, jolloin muistiosoitteen nykyinen arvo ja odotettu arvo ovat kumpikin 10. Koska nykyinen ja odotettu arvo ovat samat, muistiosoitteen arvo päivitetään uuteen arvoon eli 11. Toinen säie suorittaa oman CAS-operaationsa, mutta nykyinen arvo (11) ei enää vastaa odotettua arvoa (10). Toisen säikeen tekemä päivitysyritys epäonnistuu.

6.6 Futuuri

Säikeiden pysähtyminen odottamaan muiden säikeiden käyttämiä tai ulkopuolisia resursseja, heikentää sovelluksen suorituskykyä. Esimerkiksi tilanne, jolloin sovelluksessa tehdään tietokantakysely. Kyselyyn liittyy paljon sovelluksen ulkopuolisia resursseja, kuten tietokantayhteyden luominen ja tietokantakäyttäjän todentaminen. Säikeen on pakko pysähtyä odottamaan sovelluksen ulkopuolisia resursseja. Koska säikeiden ei haluta olevan toimettomina, tällaisissa tilanteissa säikeiden pysähtymisen voi välttää, käyttämällä futuuria. (Prokopec 2014.)

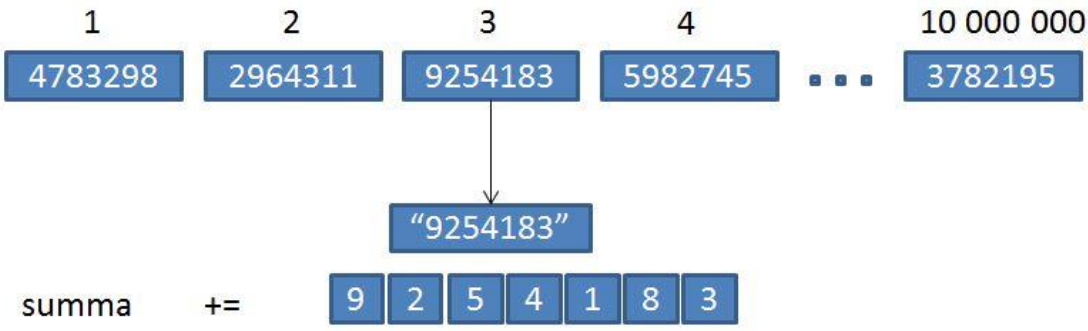
Futuuri on tietorakenne, joka nimensä mukaan kuvaa tulevaa arvoa. Kun futuuri luodaan, futuurin ei tarvitse sisältää arvoa. Futuuriin sijoitetaan arvo, vasta kun arvo on saatavilla. Kun futuuriin sijoitetaan arvo, arvoa ei voi enää muuttaa. Tietokantakyselyn tuloksen hakeminen voidaan toteuttaa futuurin avulla niin, että kyselyn tulos sijoitetaan futuurin arvoksi, vasta kun tulos on saatavilla. Kun tulosta odotetaan saataville, säikeet voivat suorittaa sillä aikaa muita tehtäviä. (Prokopec 2014.)

7 Rinnakkaisohjelmoinnin toteutus

Tässä kappalessa hyödynnetään edellisten kappaleiden sisältöä, rinnakkaisesti toimivan sovelluksen luomisessa. Tarkoituksena on antaa konkreettinen esimerkki, miten sovelluksen toteutus muuttuu, kun tehtävän suoritus rinnakkaistetaan. Sovellus toteutetaan myös perinteisellä mallilla eli peräkkäistoteutuksena, josta saadaan hyvä vertailukohta. Kahden toteutustavan hyviä ja huonoja puolia analysoidaan. Sovellusten testaus suoritetaan kaksi gigahertsisellä dual-core prosessorilla eli käytössä on maksimissaan kaksi säiettä kerrallaan.

7.1 Peräkkäistoteutus

Käytetään esimerkkinä sovellusta, joka saa ulkoisesta lähteestä taulukon lukuja ja tekee luvuilla laskennallisesti raskaita tehtäviä. Laskennallisesti raskaana tehtävänä käytetään kaikkien lukujen numeroiden yhteenlaskemista (esim. luku 123 lasketaan 1+2+3). Taulukossa on kymmenen miljoonaa lukua, jotka ovat kaikki seitsemän numeroisia. Toteutetaan sovellus ensin ilman rinnakkaistusta.



Kuvio 11. Peräkkäistoteutuksen kuvaus

Sovelluksessa on siis yksi taulukko, joka sisältää kymmenen miljoonaa lukua. Yhden luvun voi jakaa numeroihin muuttamalla sen ensin merkkijonoksi ja sen jälkeen iteroimalla merkkijonon läpi, numero kerrallaan. Numerot lisätään yksi kerrallaan summa-muuttujaan.

Kuvion pohjalta voidaan kirjoittaa sovelluksen lähdekoodi (kuvio 12).

```
public class PerakkaisToteutus {

    private static final int TAULUKON_KOKO = 10000000;

    public static void main(String[] args) {
        final int[] lukuTaulukko = UlkoinenLahde.generoiLukuTaulukko(TAULUKON_KOKO);
        final long aloitusAika = System.nanoTime();
        long summa = 0;
        for (int luku : lukuTaulukko) {
            final String lukuStr = String.valueOf(luku);
            for(int i = 0; i < lukuStr.length(); i++) {
                final int numero = Character.getNumericValue(lukuStr.charAt(i));
                summa += numero;
            }
        }
        final double kesto = (System.nanoTime() - aloitusAika) / 1.0e9;
        System.out.println("Suoritus kesti: " + kesto);
        System.out.println("summa: " + summa);
    }
}
```

Kuvio 12. Peräkkäistoteutuksen lähdekoodi

Peräkkäisesti toteutettuna, sovellus on hyvin yksinkertainen. Ensimmäiseksi generoidaan lukutaulukko ja initialisoidaan summa-muuttuja. Lukutaulukko käydään läpi for-silmukassa ja jokaisella kierroksella käsiteltävä luku muutetaan merkkijonoksi. Merkkijonoksi muutettu luku käydään läpi uuden sisäkkäisen silmukan avulla ja summa-muuttujaan lisätään jokaisella kierroksella yksi numero.

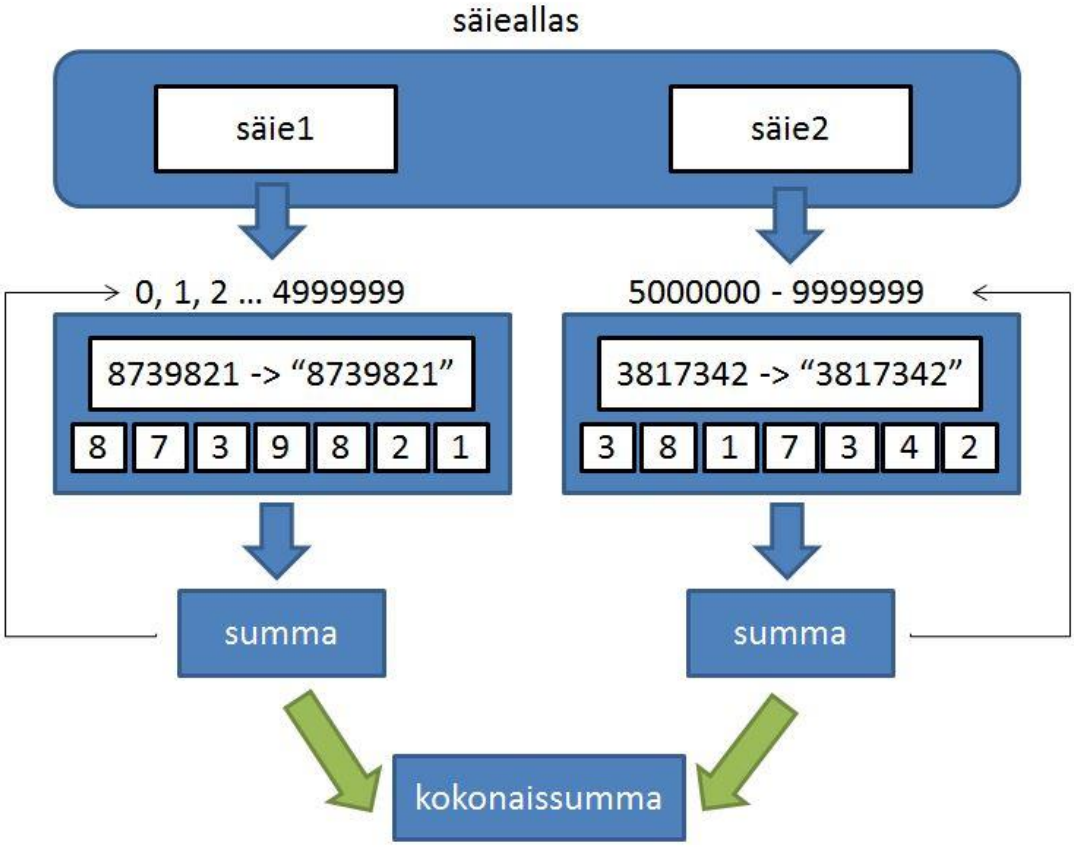
Suoritusaikaa aletaan mittaamaan vasta, kun lukutaulukko on jo generoitu eli tarkoituksena on ottaa aikaa ainoastaan lukutaulukon käsittelystä. Peräkkäisesti etenevä sovellus selviytyy suorituksesta noin viidessä sekunnissa, joka on käyttäjälle jo ihan huomattava viive. Sovelluksen suorituskykyä voisi olla hyvä parantaa.

7.2 Rinnakkaistoteutus

Kun sovelluksen suorituskykyä halutaan parantaa rinnakkaistuksella, pitää ottaa selvää, mitkä sovelluksen tehtävät vievät kaikkein eniten resursseja. Laskennallisesti kevyitä tehtäviä ei ole järkevää rinnakkaistaa, koska hyöty on pieni verrattuna rinnakkaistuksen aiheuttamaan lähdekoodin monimutkaistumiseen ja virhealttiuteen. Lukutaulukko-esimerkin tapauksessa, ainoa suuri kokonaisuus on taulukon läpikäyminen, joten rinnakkaisille säikeille voisi olla hyödyllistä jakaa suuresta kokonaisuudesta pienempiä osia käsiteltäväksi.

Rinnakkaistuksen ongelmat liittyvät kaikki sovelluksen tilaan, joten on tärkeää selvittää, muuttaako rinnakkaistettava tehtävä sovelluksen tilaa. Peräkkäistoteutuksen kuviosta ja lähdekoodista on helppo löytää yksi muuttuva muuttuja eli summa. Jos useita säikeitä

pääsee käsiksi samaan summa-muuttujaan, muistin eheys vaarantuu. Koska halutaan välttää synkronointia ja saada rinnakkaistuksesta mahdollisimman paljon hyötyä, summa pitää eristää jokaiselle säikeelle omaksi muuttujaksi. Kokonaisuuden rinnakkaistaminen on kuvattu kuviossa 13.



Kuvio 13. Rinnakkaistoteutuksen kuvaus

Kuviossa toteutus esitetään käyttämällä kahta säiettä. 10 miljoonaa taulukon indeksiä jaetaan käytössä oleville säikeille tasaisesti, jotta kaikilla säikeillä on yhtä suuri työmäärä. Säikeet laskevat itsenäisesti oman osuutensa taulukosta ja päivittävät omaa summa-muuttujaansa jokaisella kierroksella. Kun säie on saanut oman suorituksensa tehtyä, säikeen laskema summa lisätään kokonaissummaan. Teoriassa tämän toteutuksen avulla sovelluksen suorituskyvyn pitäisi kaksinkertaistua.

Haluttu lopputulos käy hyvin selväksi kuviosta, mutta toteutustapa lähdekoodissa voi olla tässä vaiheessa vielä epäselvä. Yksi hyvä lähestymistapa on ensin eristää säikeen tehtävä omaksi luokaksi. Tehtävä-luokalle pitää pystyä antamaan parametrina käsiteltävä taulukko ja säikeen osuuden ensimmäinen ja viimeinen indeksi taulukosta. Tehtävä-luokka tarvitsee lisäksi summa-muuttujan, jota päivitetään ja tietysti itse toiminnallisuuden

eli metodin, jossa käydään silmukkaa läpi ja päivitetään summaa. Tehtävä-luokan lähdekoodi on esitettynä kuviossa 14.

```
public class Tehtava implements Runnable {
    private final int[] lukuTaulukko;
    private final int ensimmainen;
    private final int viimeinen;
    private long summa;

    public Tehtava(final int[] lukuTaulukko, final int ensimmainen,
            final int viimeinen) {
        this.lukuTaulukko = lukuTaulukko;
        this.ensimmainen = ensimmainen;
        this.viimeinen = viimeinen;
    }

    public long yhdenSaikeenLaskemaSumma() {
        return summa;
    }

    public void run() {
        for (int i = ensimmainen; i <= viimeinen; i++) {
            final int luku = lukuTaulukko[i];
            final String lukuStr = String.valueOf(luku);
            for (int j = 0; j < lukuStr.length(); j++) {
                final int numero = Character.getNumericValue(lukuStr.charAt(j));
                summa += numero;
            }
        }
    }
}
```

Kuvio 14. Tehtävä-luokan lähdekoodi

Tehtävän pitää toteuttaa Runnable-rajapinta, jotta tehtävä-olion voi antaa säikeelle suoritettavaksi. Kun säikeelle annetaan tehtävä-olio suoritettavaksi, säie suorittaa tehtävä-olion run-metodin. Tehtävä-olio luodaan antamalla sille parametreina taulukko ja taulukon osuuden ensimmäinen ja viimeinen indeksi. Ensimmäistä ja viimeistä indeksiä käytetään run-metodin silmukassa rajaamaan säikeen työmäärä tiettyyn osuuteen taulukosta. Tehtävä-olion summa-muuttujaa päivitetään run-metodin silmukassa kierros kierrokselta.

Tehtävän eristämisen jälkeen voidaan kirjoittaa sovelluksen logiikka. Kyseessä on laskennallinen sovellus, joten säiealtaan koon on hyvä vastata käytettävän laitteen prosessoriydinten määrää. Säikeiden suorittamille tehtäville tarvitaan oma taulukkonsa, jossa tehtävien laskemia summia voidaan pitää tallessa. Lisäksi tarvitaan muuttujat ensimmäiselle ja viimeiselle indeksille ja lasketulle säikeen osuudelle taulukosta. Rinnakkaistoteutuksen muuttujien initialisointi esitetään kuviossa 15.

```
public class RinnakkaisToteutus {

    private static final int TAULUKON_KOKO = 10000000;
    private static final int YDINTEN_MAARA = Runtime.getRuntime().availableProcessors();

    public static void main(String[] args) {
        final int[] lukuTaulukko = UlkoinenLahde.generoiLukuTaulukko(TAULUKON_KOKO);
        final long aloitusAika = System.nanoTime();
        long kokonaissumma = 0;
        final ExecutorService saieallas = Executors.newFixedThreadPool(YDINTEN_MAARA);
        Tehtava[] tehtavat = new Tehtava[YDINTEN_MAARA];
        int ensimmainen = 0;
        int viimeinen;
        final int saikeenOsuus = TAULUKON_KOKO / YDINTEN_MAARA;
```

Kuvio 15. Muuttujien initialisointi

Prosessoriydinten määrä selvitetään Runtimen availableProcessors-metodilla. Selvitettyä ydinten määrää käytetään säiealtaan ja tehtävätaulukon luonnissa. Lukutaulukko generoidaan ja kokonaissumma-muuttuja initialisoidaan samalla tavalla kuin peräkkäistoteutuksessa. Muuttuja ”ensimmäinen” initialisoidaan nollanteen indeksiin ja viimeinen arvo jätetään vielä tyhjäksi. Säikeen osuus lasketaan jakamalla taulukon koko ydinten määrällä. Koska esimerkissä taulukon koko on kymmenen miljoonaa, ja käytössä on kaksi ydintä, säikeen osuudeksi jää viisi miljoonaa indeksiä.

Nyt voidaan kirjoittaa silmukka, jossa luodaan ydinten määrän verran tehtäviä ja mäpätään tehtävät säikeiden suoritettavaksi (kuvio 16).

```
for (int i = 0; i < YDINTEN_MAARA; i++) {
    viimeinen = ensimmainen + saikeenOsuus - 1;
    tehtavat[i] = new Tehtava(lukuTaulukko, ensimmainen, viimeinen);
    saieallas.execute(tehtavat[i]);
    ensimmainen = viimeinen + 1;
}
```

Kuvio 16. Tehtävien mäppääminen säikeille

Ensimmäiseksi pitää laskea viimeisen indeksin arvo. Ensimmäisellä kierroksella lasku menisi 0 + 5 000 000 – 1. Päädytään arvoon 4 999 999 eli ensimmäinen säie käy läpi taulukon ensimmäiset viisi miljoonaa indeksiä, nollasta 4 999 999 indeksiin. Kun ensimmäinen ja viimeinen indeksi on selvitetty, voidaan luoda uusi tehtävä-olio määritetyillä parametreillä. Tehtävä tallennetaan tehtävät-taulukkoon. Tehtävä annetaan säiealtaalle suoritettavaksi, jolloin säieallas mäppää yhden kahdesta säikeestä suorittamaan tehtävää. Silmukan lopuksi selvitetään seuraavan kierroksen ensimmäinen indeksi, joka vastaa nykyisen kierroksen viimeistä indeksiä plus yksi. Seuraavan tehtävän ensimmäinen indeksi olisi siis tässä tapauksessa 5 000 000.

Nyt kun säikeet ovat suorittaneet tehtävänsä ja tallentaneet laskemansa summat, ainoa jäljelle jäävä asia on käydä tehtävät-taulukkoon tallennetut summat läpi ja lisätä ne kokonaissumma-muuttujaan.

```
for (int i = 0; i < tehtavat.length; i++) {
    kokonaissumma += tehtavat[i].yhdenSaikeenLaskemaSumma();
}

final double kesto = (System.nanoTime() - aloitusAika) / 1.0e9;
System.out.println("Suoritus kesti: " + kesto);
System.out.println("Kokonaissumma: " + kokonaissumma);
```

Kuvio 17. Kokonaissumman laskeminen

Tehtävät-taulukko käydään läpi for-silmukan avulla (kuvio 17). Jokaiselta tehtävä-oliolta haetaan sen laskema summa yhdenSaikeenLaskemaSumma-metodilla ja haettu summa lisätään kokonaissumma-muuttujaan. Kaikki näyttää päällisin puolin olevan kunnossa, mutta sovelluksen printtaamat tulokset eivät ole oikeita. Kokonaissumma vaihtelee nollasta muutamaan sataan.

Tässä tulee hyvin ilmi rinnakkaistoteutuksen tärkein ero peräkkäistoteutukseen verrattuna, ja myös se, kuinka vaikeaa rinnakkaisesti suoritettavien sovellusten debuggaus voi olla. Kun rinnakkaiset säikeet luodaan, ne suorittavat omia tehtäviään täysin irrallisina pääohjelman suoritusjärjestyksestä. Koodia ei voi siis lukea järjestyksessä vaan pitää tiedostaa, että jotkin tehtävät tapahtuvat taustalla samanaikaisesti.

Sovelluksen palauttamat vaihtelevat tulokset johtuvat siitä, että säikeet eivät ehdi laskea omien tehtäviensä tuloksia loppuun, ennen kuin pääohjelmassa on aloitettu laskemaan säikeiden laskemia summia yhteen. Sovelluksessa pitää siis määrittää, että säikeiden laskemia summia lasketaan yhteen vasta, kun säikeet ovat saaneet omat suorituksensa tehtyä.

```
for (int i = 0; i < YDINTEN_MAARA; i++) {
    viimeinen = ensimmainen + saikeenOsuus - 1;
    tehtavat[i] = new Tehtava(lukuTaulukko, ensimmainen, viimeinen);
    saieallas.execute(tehtavat[i]);
    ensimmainen = viimeinen + 1;
}

saieallas.shutdown();
saieallas.awaitTermination(30, TimeUnit.SECONDS);

for (int i = 0; i < tehtavat.length; i++) {
    kokonaissumma += tehtavat[i].yhdenSaikeenLaskemaSumma();
}

final double kesto = (System.nanoTime() - aloitusAika) / 1.0e9;
System.out.println("Suoritus kesti: " + kesto);
System.out.println("Kokonaissumma: " + kokonaissumma);
```

Kuvio 18. Rinnakkaistoteutuksen toiminnallisen osan lopullinen versio

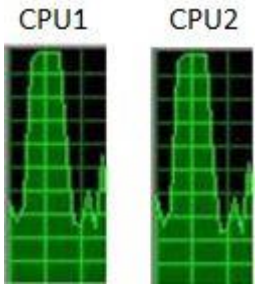
Kuviossa 18 on sovelluksen toiminnallisen osuuden lopullinen versio. Ainoa muuttunut asia, on se, että säieallas suorittaa pari metodia ennen summien yhteenlaskemista. Shutdown-metodi estää uusien tehtävien lisäyksen säiealtaalle ja sammuttaa altaan, kun kaikki käynnissä olevat tehtävät on suoritettu. AwaitTermination-metodi odottaa parametreissa määritetyn maksimi ajan tehtävien suorittamista. Tässä tapauksessa odotetaan 30 sekuntia. Jos tehtävät eivät ole suoritettu 30 sekunnissa, sovellus heittää keskeytyspoikkeuksen.

Nyt sovellus toimii oikein ja palauttaa oikean tuloksen. Sovelluksen suoritusaika on myös pudonnut kahteen ja puoleen sekuntiin, joten teoreettinen suorituskyvyn kaksinkertaistuminen saavutettiin.

7.3 Toteutustapojen vertailu

Arvioinnin kohteena oli yksi sovellus, joka toteutettiin kahdella eri tavalla. Taulukossa 2 toteutustapoja on vertailtu suoritusajan, koodirivien ja prosessoriydinten käyttöasteen perusteella.

Taulukko 2. Toteutustapojen vertailu

	Suoritusaika	Koodirivit	Prosessoriydinten hyödyntäminen
Peräkkäistoteutus	5s	20	
Rinnakkaistoteutus	2,5s	60	

Tavoitteena oli parantaa sovelluksen suorituskykyä rinnakkaistuksen avulla ja lopulta suorituskyky parani kaksinkertaisesti. Keskimäärin peräkkäisesti etenevä sovellus suoriutui tehtävästä viidessä sekunnissa, kun taas rinnakkaistoteutus selvityi laskennasta kahdessa ja puolessa sekunnissa.

Prosessoriydinten käyttöastekuvauksesta käy ilmi, miksi suorituskyky parani. Peräkkäistoteutusta analysoidessa pitää huomioida se, että sovellusta suorittaa kerrallaan tasan yksi ydin, joten käyttöastekuvauksista on mahdoton sanoa, että mitkä osuudet kummastakin kuvaajasta ovat ohjelman suoritusta. Kuvaajista voi kuitenkin nähdä sen, että prosessoriydinten tehoa ei missään vaiheessa hyödynnetä kokonaan, vaan tehoa hyödynnetään satunnaisina piikkeinä. Kuvaajan leveys kertoo myös sen, että peräkkäistoteutuksessa tehoa piti hyödyntää pidempään, ennen kuin ohjelma saatiin suoritettua. Rinnakkaistoteutuksessa kumpikin ydin hyödyntää käytettävissä olevan tehon tasaisesti.

Suorituskyvyn paraneminen ei kuitenkaan tapahtunut kovin helposti. Rinnakkaistuksen toteutus vaati paljon työtä ja koodirivien määrästä huomataankin, että rinnakkaistoteutus vaati kolminkertaisen määrän lähdekoodia peräkkäistoteutukseen verrattuna. Sitä mukaa kun lähdekoodia tuli lisää, koodi myös monimutkaistui. Tämä onkin varmaan se tärkein

asia, mikä pitää pystyä arvioimaan ennen tehtävän rinnakkaistamista: onko suorituskyvyn parantuminen koodin monimutkaistumisen arvoista?

Kyseessä olevan esimerkin tapauksessa peräkkäistoteutuksen suoritusaika on vielä niin lyhyt, että rinnakkaistaminen ei ollut koodin monimutkaistumisen arvoista. Huomattavasti raskaammat tehtävät, jotka vaativat useita kymmeniä sekunteja, ovat huomattavasti parempia kohteita rinnakkaistamiselle.

8 Tulokset

Työssä kävi ilmi, että rinnakkaisohjelmoinnin ymmärtäminen vaatii syvällisempää ymmärrystä tietokoneen ja JVM:n toiminnasta. Syvällisempää ymmärrystä vaaditaan sen takia, että rinnakkaisohjelmoinnin haasteet liittyvät konkreettisesti alemman tason komponentteihin, kuten muistiin.

Muistirakenteet ja niiden väliset yhteydet ilmenivät kaikkein tärkeimmiksi asioiksi kokonaisuuden hahmottamista varten. JVM:n luomat abstraktiot keko- ja pinomuistista ovat käteviä ohjelmoijan työn helpottamiseksi, mutta pitää silti ymmärtää, että taustalla kaikki tallentuu keskusmuistiin. Lisäksi prosessoriydinten välimuistit ja niihin liittyvät muuttujien näkyvyysongelmat on hyvä ymmärtää.

Rinnakkaisohjelmointiin liittyi useita haasteita, kuten muistin eheyden ylläpitäminen ja säikeiden tehokas koordinointi. Haasteena on myös muuttaa ohjelman suorittamiseen liittyvää ajatusmallia: lähdekoodia ei voi enää ajatella kronologisena toteutusketjuna vaan pitää tiedostaa, että laskentaa suoritetaan taustalla samanaikaisesti.

Työssä kävi ilmi, että rinnakkaisohjelmoinnin toteuttamista varten on kehitetty useita eri suunnittelumalleja ja tietorakenteita, joiden tarkoitus on helpottaa rinnakkaisohjelmointia. Esimerkiksi säikeiden koordinointia voi helpottaa käyttämällä säieallasta tai synkronointia voi välttää käyttämällä atomista muuttujaa. Lisäksi fork-join on melko yksinkertainen rinnakkaistuksen toteutustapa.

Työssä tehdyssä sovellusvertailussa rinnakkaisohjelmoinnin toteuttaminen osoittautui melko haastavaksi. Ongelman jakaminen rinnakkaisiksi tehtäviksi voi ongelmasta riippuen olla todella hankalaa tai jopa mahdotonta. Tehtävien jakamisen jälkeenkin, lähdekooditoteutus on hankala hahmottaa. Ohjelman suoritukseen liittyvän ajatusmallin muuttaminen aiheutti ongelmia toteutusvaiheessa.

Rinnakkaisohjelmoinnin avulla sovelluksen suorituskyky pystyttiin kaksinkertaistamaan. Suorituskyvyn paraneminen johtui siitä, että sovellus hyödynsi paremmin kaikkia käytössä olevia prosessoriytimiä. Ongelmaksi muodostui se, että rinnakkaistoteutus vaati kolme kertaa enemmän lähdekoodia.

9 Pohdinta

Rinnakkaisohjelmointiin liittyy olennaisesti alemman tason fyysiset komponentit, joiden toimintaa pitää ymmärtää. Tämän takia rinnakkaisohjelmointia ei voida tehdä kovin korkealla abstraktiotasolla ja tämän takia rinnakkaisohjelmointi eroaa niin paljon tavanomaisesta ohjelmoinnista. Tavanomaisessa ohjelmoinnissa pärjää ilman alemman tason ymmärrystä.

Rinnakkaisohjelmointia ei tueta tarpeeksi hyvin JVM-kielissä. Rinnakkaisohjelmointia olisi todennäköisesti mahdollista tehdä korkealla abstraktiotasolla, jos käytössä olisi tarpeeksi hyvät työkalut. Perinteinen ohjelmointi toimii korkealla abstraktiotasolla, koska ohjelmoija saa kuvaavia virhe- ja poikkeusilmoituksia ongelmatilanteissa. Rinnakkaisohjelmoinnissa taustalla tapahtuvasta virheestä ei ilmoiteta mitenkään vaan vika pitää löytää toimivasta ohjelmasta.

Rinnakkaisohjelmoinnissa pystyttäisiin saamaan virhe- ja poikkeusilmoituksia muistin eheyden rikkovista muistitoimenpiteistä, jos muistin eheyttä pystyttäisiin valvomaan. Esimerkiksi tietokantaympäristössä, samanaikaisille transaktioille voidaan asettaa selvät toimintasäännöt, jotka pitävät huolen tietokannan eheydestä. Samaa ideaa voisi hyödyntää sovelluksissa. Tällaista komponenttia itseasiassa käytetäänkin muutamissa ohjelmointikielissä. Esimerkiksi uudemmassa JVM-kielessä, Clojuressa käytetään kirjastoa, jonka avulla ohjelmoija saa virheilmoituksia päällekkäisistä muistitoimenpiteistä (Subramaniam 2011, 142).

Ohjelmointikieli määrää käytettävissä olevat ohjelmointiparadigmat. Ohjelmointiparadigmat rajaavat käytettävissä olevat ongelmanratkaisutekniikat. Useiden ohjelmointiparadigmojen käyttö tekee rinnakkaisohjelmoinnin toteuttamisen helpommaksi. Esimerkiksi funktionaalinen ohjelmointiparadigma ei käsittele tilaa lainkaan. Kun tilaa ei ole, kaikille muuttujille annetaan lopullinen arvo muuttujan initialisoinnin yhteydessä. Muuttumattomuus tekee rinnakkaistuksen toteuttamisen helpommaksi. Ohjelmointikielen pitää kuitenkin vahvasti tukea funktionaalista ohjelmointia, jotta muuttumattomuus on käytännöllistä toteuttaa.

9.1 Työn käyttömahdollisuus ja opinnäytetyöprosessi

Työn lopulliseksi tulokseksi muodostui kurssimateriaaliksi sopiva johdantomateriaali rinnakkaisohjelmoinnista. Työssä läpikäytyjä asioita ei käsitellä koulutusohjelman kursseilla tarpeeksi, joten se vastaa mielestäni tarvetta hyvin.

Työn toteutusvaiheessa tehdyt päätökset ovat vahvasti kytköksissä työn tavoitteeseen koostaa mahdollisimman helposti ymmärrettävä ja havainnollistava kokonaisuus. Työ etenee matalta abstraktiotasolta ylöspäin, koska arvioin sen olevan kokonaisuuden hahmottamisen kannalta paras etenemisjärjestys. Työssä käsitellään pääasiassa abstrakteja kokonaisuuksia, joten työssä on pyritty käyttämään paljon esimerkkejä ja kuvioita, joiden tarkoitus on havainnollistaa käsiteltävää asiaa paremmin. Lisäksi joitakin käsiteltäviä asioita on tarkoituksenmukaisesti yksinkertaistettu, koska kokonaisuuden hahmottamisen kannalta tarkempi käsittely ei ole tarpeellista.

Kokonaisuutena, opinnäytetyöprosessi oli onnistunut. Työn lopullinen tulos vastaa hyvin alkuvaiheessa asetettua tavoitetta ja työ pysyi hyvin aikataulussa. Aihevalinta oli myös hyvä, koska aiheen käsittely syvensi hyvin oman opintopolun opintoja. Työ oli rajattu ainoastaan JVM-kieliin, joten jatkotutkimuksena voisi olla mielenkiintoista vertailla rinnakkaisohjelmointia JVM-kielten ja skriptauskielten, kuten Pythonin ja Rubyn välillä. Työssä tehdään myös vain pintaraapaisu ohjelmointiparadigmojen vaikutuksesta rinnakkaisohjelmointiin, joka voisi olla mielenkiintoinen tutkimuksen aihe.

Oman oppimisen kannalta, työn tekeminen oli todella hyödyllistä. Työn avulla pääsin syventymään paremmin tietokoneen ja JVM:n toimintaperiaatteisiin ja soveltamaan näitä toimintaperiaatteita rinnakkaisohjelmoinnissa. Tällä hetkellä vaikuttaa siltä, että moniydinprosessorit ovat siruvalmistajien pysyvä ratkaisu prosessoritehon kehittämisessä, joten rinnakkaisohjelmoinnin ymmärtämisestä on tulevaisuudessa varmasti hyötyä.

10 Lähteet

Bailey, C. 2012. From Java code to Java heap: Understanding the memory usage of your application. URL: https://www.youtube.com/watch?v=FLcXf9pO27w. Katsottu: 2.3.2015.

Barney, B. 2014. Introduction to Parallel Computing. Luettavissa: https://computing.llnl.gov/tutorials/parallel_comp/. Luettu 15.3.2015.

Bloch, J., Bowbeer, J., Göetz, B., Holmes, D., Lea, D. & Peierls, T. 2006. Java Concurrency In Practice. Addison-Wesley-Professional.

Jenkov, J. 2010a. Race Conditions and Critical Sections. Luettavissa: http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html. Luettu: 10.3.2015.

Jenkov, J. 2010b. Java Memory Model. Luettavissa: http://tutorials.jenkov.com/java-concurrency/java-memory-model.html. Luettu 2.3.2015.

Jenkov, J. 2010c. Java Synchonized Blocks. Luettavissa: http://tutorials.jenkov.com/java-concurrency/synchronized.html. Luettu 10.3.2015.

Jenkov, J. 2010d. Starvation and Fairness. Luettavissa: http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html. Luettu 10.3.2015.

Kumar, P. 2014. Java Heap Memory vs. Stack Memory Difference. Luettavissa: http://www.journaldev.com/4098/java-heap-memory-vs-stack-memory-difference. Luettu: 3.3.2015.

Lea, D. 2006. A Java Fork/Join Framework. Luettavissa: http://gee.cs.oswego.edu/dl/papers/fj.pdf. Luettu: 24.3.2015.

Nieminen, P. 2014. Käyttöjärjestelmät kurssimateriaalia. Luettavissa: http://users.jyu.fi/~nieminen/kj14/moniste_uusin.pdf. Luettu: 26.2.2015

Prokopec, A. 2014. Learning Concurrent Programming in Scala. Packt Publishing.

Slangen, S. 2012. What Is The Java Virtual Machine & How Does It Work? Luettavissa: http://www.makeuseof.com/tag/java-virtual-machine-work-makeuseof-explains/. Luettu: 3.3.2015

Strickland, J. 2012. How Moore's Law Works. Luettavissa: http://computer.howstuffworks.com/moores-law1.htm. Luettu: 24.2.2015.

Subramaniam, V. 2011. Programming Concurrency on the JVM. Pragmatic Bookshelf.

Venners, B. 1999. The Java Virtual Machine. Luettavissa: http://www.artima.com/insidejvm/ed2/jvm.html. Luettu 1.3.2015.

Wicht, B. 2010. Java Concurrency – Part 7: Atomic Variables. Luettavissa: http://java.dzone.com/articles/java-concurrency-%E2%80%93-part-6. Luettu: 26.3.2015.