LAPIN AMK
Lapland University of Applied Sciences

Rinnakkaisohjelmointi .NET-ympäristössä

Aleksi Kontkanen
Juho Lappalainen

Opinnäytetyö
Tieto- ja viestintätekniikka
Insinööri (AMK)

2014

LAPIN AMMATTIKORKEAKOULU

TEKNIIKKA JA LIIKENNE

Tieto- ja viestintätekniikka

Rinnakkaisohjelmointi .NET-ympäristössä

2014

Toimeksiantaja
Lapin ammattikorkeakoulu

Ohjaaja
Erkki Mattila

Aleksi Kontkanen
Juho Lappalainen

Hyväksytty 30.10.2014

LAPIN AMK
Lapland University of Applied Sciences

Opinnäytetyön
tiivistelmä

Tekniikan ja liikenteenala
Tieto- ja viestintätekniikka

Tekijät	Aleksi Kontkanen Juho Lappalainen	Vuosi	2014
Toimeksiantaja	Lapin ammattikorkeakoulu		
Työn nimi	Säieohjelmointi .NET-ympäristössä		
Sivu- ja liitemäärä	45 + 0		

Tämä opinnäytetyö on tehty Lapin ammattikorkeakoulussa vuonna 2014. Opinnäytetyön päätavoite oli esitellä keinoja prosessoreiden moniydinarkkitehtuurin hyödyntämiseen Microsoftin .NET-ympäristössä.

Prosessoriteknologian kehityksen trendi on ollut 2000-luvun aikana ytimien lisääminen yksittäisiin prosessoreihin. Ytimien lisäämisellä voidaan nostaa laskentatehoa ilman sellaisia ongelmia, joita syntyy keskityttäessä pelkästään yksittäisen ytimen kellotaajuuden nostamiseen. Moniydinarkkitehtuuri synnyttää kuitenkin joukon ohjelmointiin liittyviä ongelmia, jotka saattavat olla vaikeasti paikannettavia.

Opinnäytetyön tavoitteina oli selvittää säieohjelmoinnissa käytössä olevia menetelmiä ja vertailla niiden tehokkuutta toisiinsa sekä perinteseen sekventiaaliseen ohjelmointitapaan. Lisäksi pyritään nostamaan esiin näihin menetelmiin liittyvät tyypillisimmät ongelmat.

Ohjelmoijan tulee ymmärtää säieohjelmoinnin periaatteet. Oikein toteutettu useampaa säiettä käyttävä ohjelmisto antaa kuluttajalle nopean ja responsiivisen kokemuksen. Opinnäytetyön prosessissa kävi ilmi, että rinnakkaisen ohjelman toteuttaminen poikkeaa paljon perinteisestä tavasta tehdä sovelluksia. Rinnakkain toimivat algoritmit saavat sovellukset toimimaan huomattavasti nopeammin

Avainsanat: Moniydinarkkitehtuuri, säieohjelmointi, rinnakkaisohjelmointi, Microsoft .NET

LAPIN AMK
Lapland University of Applied Sciences

Abstract of Thesis

School of Technology
Technology, Communication and Transport

Authors Aleksi Kontkanen, Juho Lappalainen
Year 2014

Commissioned by Lapland University of Applied Sciences
Subject of thesis Parallel Programming in .NET
Number of pages 45 + 0

This thesis was done in Lapland University of Applied Sciences in 2014. The primary objective of this thesis was to outline and explain the methods to exploit a multi-core architecture of today's processors in the Microsoft .NET environment.

The trend of development of processor technology in the 21st century is to add multiple cores in to a single processor. By adding multiple cores, processors can increase their efficiency without having problems of increased clock-speeds. However, implementing a multi-core architecture raises a new set of hard-to-locate problems in software programming.

The primary objective of this thesis was to outline methods used in parallel programming and compare their efficiency with each other and also with traditional sequential programming method. Additionally some of the most common problems related to these methods were explained.

A programmer has to understand the principles of parallel programming in today's software industry. A parallel application gives fast and responsive experience to the customer when it is correctly implemented. This thesis revealed how different it is to implement parallel program compared to sequential one and that by adding parallel computation to the algorithms a program can perform significantly faster.

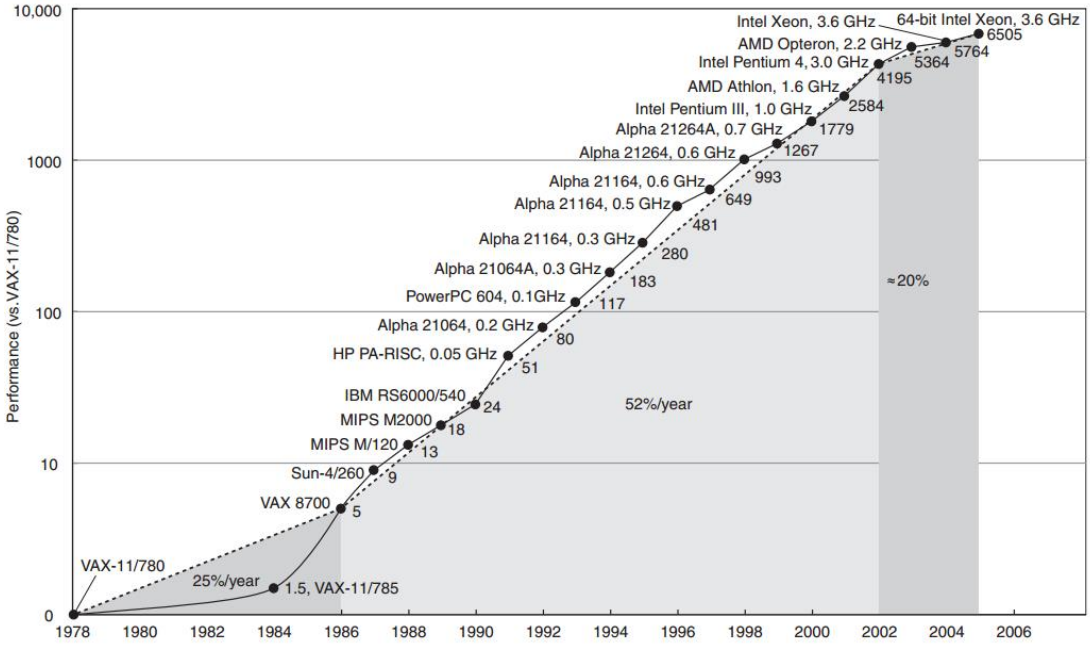
Key words: multi-core architecture, parallel programming, Microsoft .NET

SISÄLTÖ

KUVIO- JA TAULUKKOLUETTELO

1 JOHDANTO

Elektroniset tietokoneet ovat alun perin rakennettu yksiytimisillä prosessoreilla, jotka pystyvät suorittamaan peräkkäistä ohjelmakoodia oman maksimi-kellotaajuuden rajoissa. Prosessori, jonka kellotaajuus on suurempi kuin edellisen sukupolven tai kilpailevan valmistajan prosessori, pystyy siis suorittamaan sille annetut käskyt nopeammin ilman että ohjelmoijan tarvitsee tehdä muutoksia ohjelmakoodiin. Laitevalmistajat kilpailivat pitkään kellotaajuuksien kasvattamisella ja peräkkäistä koodin suorittamista tukevien tekniikoiden, kuten prosessoreiden välimuistien, kehittämisellä. Kellotaajuudet ovat kasvaneet huimasti 90-luvun alusta aina 2000-luvun puoleen väliin saakka. Tällä aikavälillä prosessoreiden suorituskyky on lähes satakertaistunut. (Columbia University 2012.)



Taulukko 1. Prosessorien kehitys (Columbia University 2012)

Tuolloin oli kaksi tapaa nopeuttaa ohjelman suorittamista: ostaa uudempia ja nopeampia laitteita tai optimoida koodia. Kolmas tapa on prosessoreiden ylikellottaminen, mutta tämä lähestymistapa laskee laitteiston luotettavuutta. Yrityksille kustannustehokkaampaa oli ostaa uutta laitteistoa. Ohjelmoijan palkkaaminen optimointiin useimmissa tapauksessa ylitti uuden tietokoneen tai komponentin hinnan selvästi.

Mooren lain mukaan (Moore 1965) transistorien määrä siruissa kaksinkertaistuu noin joka toinen vuosi. Ongelmaksi muodostuu kuitenkin lämmöntuotto kellotaajuuksien kasvaessa, joten laitteistovalmistajat alkoivat kehittää markkinoille useampiytimisiä eli moniydintekniikkaa tukevia prosessoreita suorituskyvyn kasvattamiseksi. Ytimet toimivat yksittäisinä prosessoreina, jolloin laskentatehoa saadaan kasvatettua kasvattamatta kellotaajuuksia.

Prosessorit ovat nykyään jopa 128-ytimisiä ja markkinoilta on vaikeaa löytää edes halpaa tietokonetta, jossa olisi yksiytiminen prosessori. Tietokoneiden suorituskyvyn parantamiseen käytetään nykyään usein moniydinteknologiaa ja parempaa prosessoriarkkitehtuuria.

Ytimien lisääminen ei kuitenkaan nopeuta sekventiaalisen eli perinteisen ohjelman suorittamista. Prosessori tai käyttöjärjestelmä ei voi tietää mitä osia ohjelmasta voidaan suorittaa rinnakkain eli useammalla ytimellä yhtä aikaa, joten ohjelmoijan tulee itse huomioida mahdollinen rinnakkainen toteutus ohjelmakoodissa. Käytännössä tämä tarkoittaa useampien itsenäisten säikeiden luomista ohjelman prosessien sisälle, joille annetaan ajoaikaa prosessorilta vuorotellen. Niitä voidaan suorittaa myös yhtä aikaa, jos käytössä on moniydinprosessori.

Säikeitä tukevissa ohjelmointikielissä voidaan määrätä tietty osa ohjelmasta suoritettavaksi tietylle säikeelle. Säie voidaan suorittaa halutussa vaiheessa ohjelmaa, suorittaa se tarvittaessa uudestaan tai siitä voidaan luoda useampia samanaikaisia ilmentymät. Nämä eri ilmentymät voidaan ohjata useammalle prosessoriytimelle samaan aikaan, jolloin niiden suorittaminen nopeutuu riippuen käytössä olevien resurssien määrästä. Ohjelmoijan on tiedettävä, kuinka säikeiden tulee käyttäytyä ohjelman suorituksen aikana, jotta vältyttäisiin satunnaisilta hankalasti paikannettavilta virheiltä.

Ongelma ohjelmoijan näkökulmasta on se, että ohjelma mielletään sarjaksi peräkkäisiä komentoja joita tietokone suorittaa. Tällainen ajattelutapa uusien ohjelmien luonnissa aiheuttaa sen, että ohjelma ei pysty hyödyntämään kaikkia käytettävissä olevia resursseja. Neliydinprosessorissa voi teoriassa

jäädä käyttämättä melkein 75% suorituskyvystä. Mitä enemmän prosessoreissa on ytimiä, sitä pienemmäksi tämä käyttöaste jää.

Ohjelman käyttäjä ei välttämättä tiedä kuinka ohjelma on toteutettu, mutta hidas käyttöliittymä ja hitauden tuntu toiminnallisuudessa saattaa olla riittävä peruste vaihtaa ohjelma kilpailijan tuotteeseen. Ohjelmoijan täytyy pystyä tekemään koodia, joka hyödyntää kaikki käytössä olevat resurssit, koska resurssien käyttö millä tahansa laitteella tai alustalla on kiinni ohjelmoijan ammattitaidoista.

Tämän opinnäytetyön tarkoitus on tutkia ja vertailla rinnakkaisohjelmoinnin eri tapoja Microsoftin .NET-ympäristössä C#-ohjelmointikielellä. Esimerkkisovellukset on toteutettu kolmella eri tavalla: sekventiaalisella, klassisella säikeistysmallilla ja Task Parallel Library -toteutuksella. Sekventiaalinen on perinteinen tapa tehdä ohjelmia suoraviivaisesti ilman säikeitä, klassinen säikeistysmalli on säikeiden manuaalista toteutusta sekä hallintaa ja Task Parallel Library eli TPL on Microsoftin kehittämä kirjastorajapinta rinnakkaisohjelmoinnin helpottamiseksi .NET-ympäristössä.

2 RINNAKKAISUUS .NET-YMPÄRISTÖSSÄ

2.1 Rinnakkais- ja säieohjelmointi

Perinteinen säieohjelmointi ja rinnakkaisohjelmointi eroavat toisistaan siinä, että säieohjelmoinnissa ei välttämättä tapahdu rinnakkaista suorittamista. Myös yksiytimiselle prosessorille on hyötyä ohjelman säikeistämisestä. Käyttöliittymä ja taustalla ajettavat tehtävät voivat olla eri säikeissä, jolloin käyttöjärjestelmä antaa ajoaikaa molemmille säikeille. Tällöin käyttöliittymä pysyy käyttökelpoisena sen aikaa, kun esimerkiksi halutaan ladata internetistä tiedosto ja odotetaan latauksen suoriutumista loppuun. Silloin ei kuitenkaan ole kyse todellisesta rinnakkaisuudesta, jossa molempia säikeitä suoritettaisiin yhtä aikaa.

Säieohjelmoinnin osaaminen helpottaa ohjelmoijaa omaksumaan rinnakkaisohjelmoinnin konseptit. Säieohjelmoinnin tunteminen auttaa myös hahmottamaan erilaisten rinnakkaisten tietorakenteiden toimintaperiaatteet.

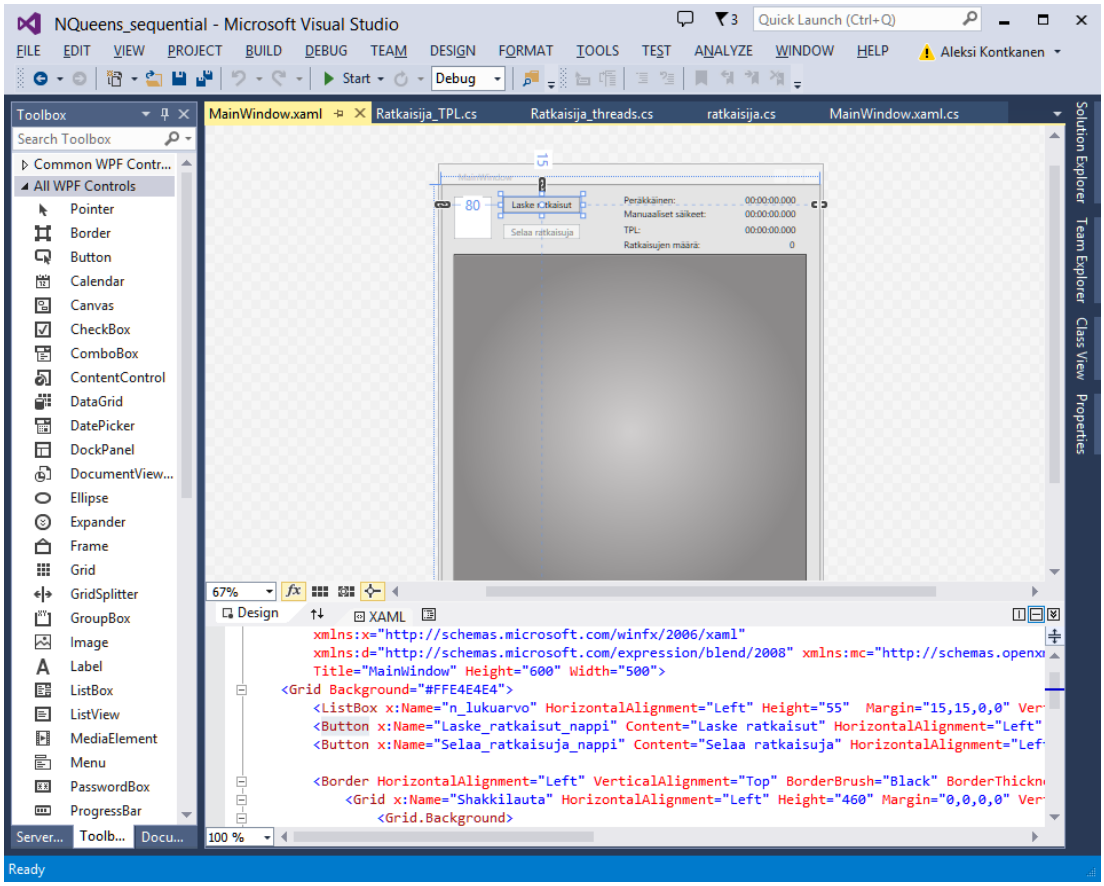
2.2 C#-ohjelmointikieli

C# (lausutaan ”C sharp”) on ohjelmointikieli erilaisten sovellusten kehittämiseen .NET-ympäristössä. C# on yksinkertainen ja tehokas olio-ohjelmointikieli. Kieleen liitetyt innovaatiot mahdollistavat nopean ohjelmistokehityksen, ja se on ulkoasultaan C-kielen kaltainen (katso koodiesimerkki 1).

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Esimerkin_nimiavaruus
{
    class Esimerkki
    {
        public static void Main()
        {
            Task TPL_task = new Task(() =>
            {
                Console.WriteLine("Hello Task Parallel Library!");
            });

            TPL_task.Start();

            // Ohjelma kirjoittaa konsoliin:
            // Hello Task Parallel Library!
        }
    }
}
```

Koodiesimerkki 1. C#-syntaksi

Microsoftin kehittämä Visual Studio (kuvio 1.) sisältää koodieditorin, kääntäjän, projektipohjia, suunnittelutyökaluja, apuohjelmia ohjelmoimiseen, helppokäyttöisen debuggerin (vianpaikannus työkalu) sekä muita työkaluja C#-ohjelmointiin. .NET-ympäristö tarjoaa valmiita luokkakirjastoja, joilla saa käyttöönsä käyttöjärjestelmän tarjoamat palvelut. Hyvin suunnitellut valmiit luokat nopeuttavat ohjelman kehityskaarta huomattavasti. (Microsoft Developer Network 2014a.)



Kuvio 1. Visual Studio-ohjelmointiympäristö

Visual Studiossa voi käyttää graafisen käyttöliittymän luomiseen kahta eri tapaa. Ensimmäinen tapa on luoda käyttöliittymän elementit C#-ohjelmakoodissa. Toinen ja suositellumpi tapa on käyttää käyttöliittymäeditoria. Tämä graafinen käyttöliittymäeditori luo ohjelman ulkoasusta XAML-koodia (Extensible Application Markup Language). XAML-koodia voi myös muokata käyttöliittymäeditorin alla näkyvästä XAML-editorista (koodiesimerkki 2).

```
<Window
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d" x:Class="NQueens_sequential.MainWindow"
        Title="Nqueens ratkaisija" Height="600" Width="500">
    <Grid Background="#E4E4E4">
        <ListBox x:Name="n_lukuarvo" HorizontalAlignment="Left" Height="55"  Margin="15,15,0,0" VerticalAlignment="Top" Width="50"
                 SelectionChanged="n_lukuarvo_SelectionChanged" SelectedIndex="7"/>
        <Button x:Name="Laske_ratkaisut_nappi" Content="Laske ratkaisut" HorizontalAlignment="Left" Margin="80,15,0,0"
                VerticalAlignment="Top" Width="100" Height="21" Click="Laske_ratkaisut_nappi_Click"/>
        <Button x:Name="Selaa_ratkaisuja_nappi" Content="Selaa ratkaisuja" HorizontalAlignment="Left" Margin="80,50,0,0"
                VerticalAlignment="Top" Width="100" IsEnabled="False"
                Click="Selaa_ratkaisuja_nappi_Click" Visibility="Hidden"/>

        <Border HorizontalAlignment="Left" VerticalAlignment="Top" BorderBrush="Black" BorderThickness="1"
                Width="460" Height="460" Margin="15,90,0,0">
            <Grid x:Name="Shakkilauta" HorizontalAlignment="Left" Height="460" Margin="0,0,0,0" VerticalAlignment="Top"
                  Width="460" RenderTransformOrigin="0,0" >
                <Grid.Background>
                    <RadialGradientBrush>
                        <GradientStop Color="#FF8B8989" Offset="1"/>
                        <GradientStop Color="#FFD1CFCF"/>
                    </RadialGradientBrush>
                </Grid.Background>
            </Grid>
        </Border>
        <Label Content="Peräkkäinen:" HorizontalAlignment="Left" Margin="232,6,0,0" VerticalAlignment="Top"/>
        <Label Content="Manuaaliset säikeet:" HorizontalAlignment="Left" Margin="232,24,0,0" VerticalAlignment="Top"/>
        <Label Content="TPL:" HorizontalAlignment="Left" Margin="232,44,0,0" VerticalAlignment="Top"
               RenderTransformOrigin="0.684,0.077"/>
        <Label x:Name="perakkainen_tulokset" Content="00:00:00.000" HorizontalAlignment="Right"
               Margin="0,6,-0.4,0" VerticalAlignment="Top" Width="102"/>
        <Label x:Name="threads_tulokset" Content="00:00:00.000" HorizontalAlignment="Right" Margin="0,24,-0.4,0"
               VerticalAlignment="Top" RenderTransformOrigin="-0.553,0.577" Width="102"/>
        <Label x:Name="TPL_tulokset" Content="00:00:00.000" HorizontalAlignment="Right" Margin="0,44,-0.4,0"
               VerticalAlignment="Top" Width="102"/>
        <Label x:Name="lasketaan_ratkaisuja" Content="LASKETAAN RATKAISUJA..." HorizontalAlignment="Left" Margin="15,70,0,0"
               VerticalAlignment="Top" Foreground="#FF45A040" Visibility="Hidden"/>
        <Label Content="Ratkaisujen määrä:" HorizontalAlignment="Left" Margin="232,64,0,0" VerticalAlignment="Top"/>
        <Label x:Name="iteraatio" Content="0" HorizontalAlignment="Right" Margin="0,64,-0.4,0" VerticalAlignment="Top"
               Width="44"/>

    </Grid>
</Window>
```

Koodiesimerkki 2. XAML-esimerkki

2.3 Termistö ja lyhenteet

Tässä opinnäytetyössä käytetään tietotekniikan vakiintunutta termistöä käsiteltäessä rinnakkaisohjelmointia. Termeille, joille ei löydy vakiintunutta suomenkielistä vastinetta, on selitetty tässä kappaleessa. Lähdekoodissa käytetyt sanat on kirjoitettu `Courier New` -fontilla tekstin selkeyttämiseksi.

TPL (TPL): Task Parallel Library.

Klassinen säikeistys (Classic Threading Model): Säikeistetty ohjelma, joka on toteutettu ilman TPL:n kaltaista rajapintaa.

Rinnakkaisohjelmointi (Parallel Programming): Ohjelmointitapa, joka mahdollistaa ohjelman suorittamisen useammalla prosessorilla tai prosessoriytimellä samanaikaisesti.

Moniydinarkkitehtuuri (Multicore Architecture): Tapa, jolla prosessori on rakennettu. Prosessorissa on useampi kuin yksi ydin.

Kontekstin vaihdos (Context Switch): Prosessori vaihtaa suoritettavaa tehtävää toiseen, vaikka edellistä tehtävää ei ole suoritettu loppuun.

Instanssi (Instance): Luokan ilmentymä, luokasta luotu olio.

Sekventiaalinen (Sequential): Perinteinen ohjelmointitapa, johon ei ole toteutettu rinnakkaisuutta.

Ylimääräinen tiedonkäsittely (Overhead): Rinnakkaisessa toteutuksessa syntyvää ylimääräistä tiedonkäsittelyä, esim. säikeiden koordinointi ytimeltä toiselle.

Säieturvallinen (Thread Safe): Koodi, jota kahden säikeen on turvallista käsitellä yhtäaikaa.

Yhdistyminen (Join): Odotetaan säikeen suorituksen loppumista, jonka jälkeen sen tuottamia tuloksia voidaan käyttää turvallisesti.

Nälkiintyminen (Starvation): Säie odottaa prosessorilta suoritusaikaa päästäkseen jatkamaan suorittamista.

ThreadPool (ThreadPool): `ThreadPool`-luokka, jota käytetään säikeiden koordinointiin.

2.4 Parallel Framework

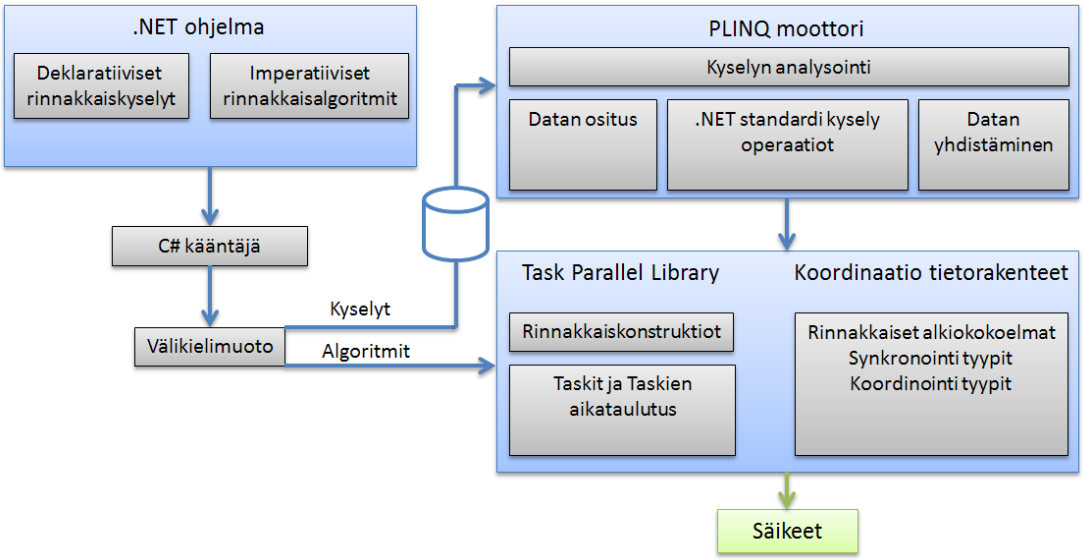
.NET 4.0:ssa esiteltyjä rinnakkaisohjelmointirajapintoja kutsutaan Parallel Frameworkiksi tai PFX:ksi. Näitä rajapintoja ovat:

- Parallel LINQ tai PLINQ
- `Parallel`-luokka
- Task Parallelism -rakenteet
- Rinnakkaiset alkiokokoelmat (Concurrent Collections)
- `SpinLock`- ja `Spinwait`-tietueet

Parallel-luokka sekä Task Parallelism-rakenteet muodostavat yhdessä Task Parallel Libraryn.

2.5 .NET ympäristö

Microsoftin .NET-ympäristöä ja Visual Studio -ohjelmointiympäristöä käytetään pääasiassa Windows-ohjelmistojen kehitykseen. Ympäristöstä löytyy myös muun muassa työkalut ASP.NET web- sekä Windows Phone-ohjelmien kehitykseen. Ympäristö tukee useita eri ohjelmointikieliä, jotka esikäännetään Microsoft Intermediate Language eli MSIL-välikielimuotoon. MSIL-koodi käännetään Common Language Runtimessa eli CLR:ssä binäärimuotoiseksi suoritettavaksi ohjelmaksi. (Microsoft Developer Network 2014c.)



Kuvio 2. Rinnakkaisohjelmoinnin arkkitehtuuri .NET-ympäristössä (Microsoft Developer Network 2014d.)

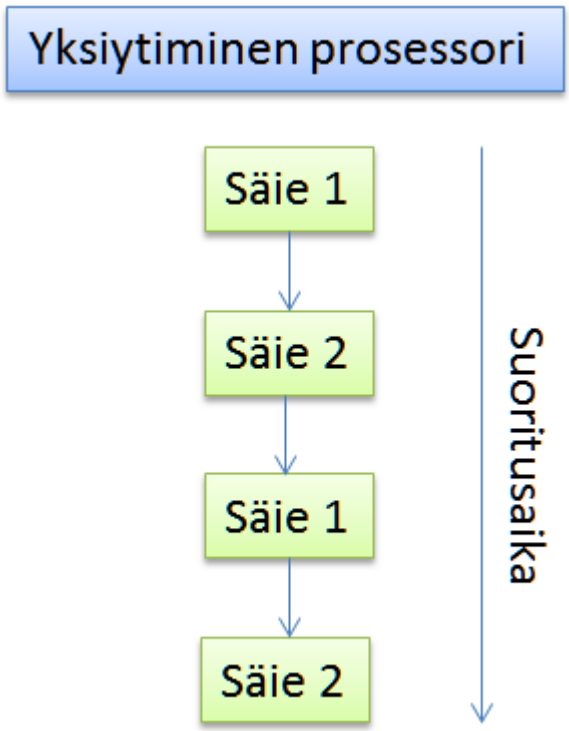
.NET-ympäristön 4.0-versioon on lisätty työkaluja ja rajapintakirjastoja rinnakkaisohjelmoinnin helpottamiseksi (katso kuvio 2). Nämä työkalut helpottavat tehokkaan ja skaalautuvan rinnakkaisesti suoritettavan ohjelman ohjelmointia. Kehitysympäristö myös huolehtii säikeiden käsittelystä ja turvallisuudesta siten, että ohjelmoijan ei itse tarvitse suoranaisesti niihin puuttua.

Näillä työkaluilla tuotetut ohjelmat osaavat skaalautua niitä suorittavan laitteiston mukaisesti. Kehitysvaiheessa ei siis ole tarpeen tietää, kuinka monta säiettä ohjelman tulee luoda saadakseen kaiken mahdollisen laskentatehon käyttöönsä. TPL huolehtii säikeiden optimaalisesta määrästä ajon aikana.

2.6 Säikeet

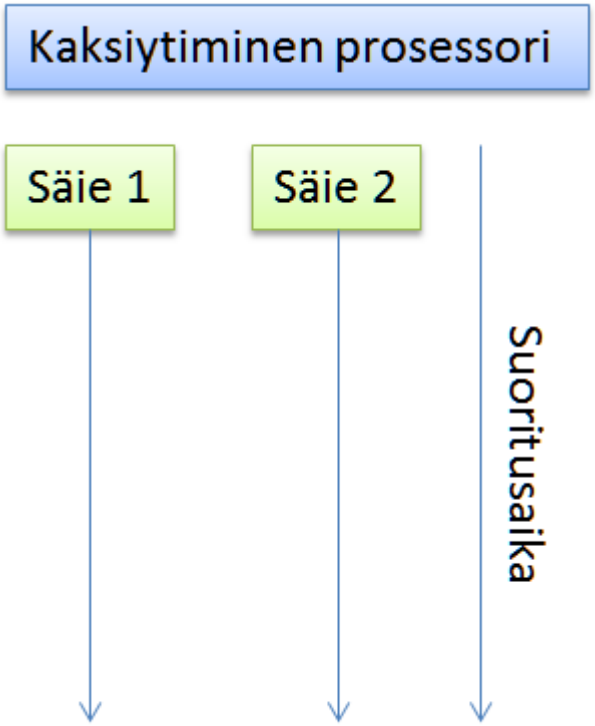
Säikeet eli Threadit ovat ajonaikaisia itsenäisesti suoritettavia ohjelmakoodikokonaisuuksia. Ohjelmoija määrittelee säikeissä suoritettavan toiminnallisuuden ja luo säikeestä ilmentymän eli instanssin halutussa kohtaa ohjelmaa ja suorittaa sen. Näitä ilmentymiä samasta tai useammasta eri säikeestä voi olla useampi yhtäaikaisesti suoritettavana, jolloin käyttöjärjestelmä jakaa niille suoritusaikaa prosessorilta. Kun käytössä on useampi prosessori tai prosessoriydin, voidaan säikeitä suorittaa rinnakkain.

Ongelma yksiytimisten prosessoreiden suoritustavassa on kontekstin vaihdos. Suoritustehoa kuluu tehtävien tai säikeiden vaihdoin välillä eli syntyy ylimääräistä tiedonkäsittelyä. Kontekstin vaihdos on kuitenkin välttämätöntä, mikäli ohjelman käyttöliittymä halutaan säilyttää responsiivisena. Mikäli käyttäjä haluaa ohjelman suorittavan jonkin muun tehtävän ja jatkaa käyttöliittymän käyttöä samanaikaisesti, on suoritettava toiminnallisuus ja käyttöliittymä ohjelmoitava eri säikeille. Käyttöjärjestelmä jakaa suoritusaikaa näille säikeille (kuvio 3), mutta tehtävien suorittaminen hidastuu.



Kuvio 3.Yksiytiminen prosessori

Kaksiytiminen prosessori pystyy suorittamaan tehtäviä rinnakkain (kuvio 4), mikäli säikeet on ohjelmoitu suoritettavaksi yhtä aikaa. Tällöin käyttäjä ei huomaa ohjelman käyttöliittymässä hidastumista.



Kuvio 4. Kaksiytiminen prosessori

Säikeet ovat ikään kuin prosesseja. Käyttöjärjestelmässä jokaisella aukinaisella sovelluksella on käynnissä oma prosessinsa, joita käyttöjärjestelmä suorittaa rinnakkain. Säikeet ovat prosessin sisällä suoritettavia kevyempiä kokonaisuuksia.

2.7 Task Parallel Library

Task Parallel Library on luokkakirjasto .NET-ympäristössä `System.Threading` ja `System.Threading.Tasks` -nimiavaruuksissa. TPL:n tarkoitus on auttaa ja helpottaa sovelluskehittäjiä yksinkertaistamaan rinnakkaisuuden toteuttaminen sovelluksissaan. TPL sovittaa ohjelman rinnakkaisuuden sitä suorittavan tietokoneen mukaiseksi. Lisäksi TPL hoitaa tehtävien jakamisen, säikeiden aikataulutuksen ThreadPool:ssa, säikeiden peruuttamisen ja tilan hallinnan ja muita matalan tason tehtäviä. TPL:n avulla kehittäjä pystyy keskittymään halutun toiminnallisuuden toteuttamiseen ohjelmassa puuttumatta säikeistykseen samalla maksimoiden ohjelman suorituskyvyn. (Microsoft Developer Network 2014b.)

Kehittäjää suositellaan kirjoittamaan säikeistettyä ohjelmaa TPL:n avulla. Kaikkea koodia ei kuitenkaan ole suotavaa säikeistää. Esimerkiksi vain

vähän tietoa käsittelevä silmukka on parempi jättää säikeistämättä, koska TPL-toteutus vaatii prosessorilta suoritusaikaa. Lisäksi TPL, kuten mikä tahansa säikeistetty koodi, monimutkaistaa ohjelman suorittamista. (Microsoft Developer Network 2014b.)

Task Parallel Library pohjautuu taskeihin (Task), jotka ovat toisistaan riippumattomia itsenäisiä operaatioita. Taskit muistuttavat klassisia tai ThreadPool:in tehtäviä (work item). TPL:n taskit ovat tehokkaita ja järjestelmän mukaan skaalautuvia. Ohjelmoijalla on taskien hallintaan enemmän mahdollisuuksia kuin klassisiin säikeisiin. Task-luokka helpottaa muun muassa odottamista, peruuttamista ja virheiden käsittelyä. (Microsoft Developer Network 2014a.)

2.8 PLINQ

Parallel LINQ tai PLINQ on .NET 4.0:ssa esitelty tapa toteuttaa LINQ kyselyitä rinnakkain. PLINQ on helppokäyttöinen ja se hoitaa taustalla laskentatyön osittamisen ja tuloksien muodostamisen. PLINQ:n käyttöä on kuitenkin syytä rajoittaa, mikäli LINQ-kysely ei vaadi suurta laskentatyötä. Tällöin automaattiseen säikeistykseen ja työn jakamiseen kuluva ylimääräinen tiedonkäsittely voi jopa hidastaa ohjelmaa. PLINQ otetaan käyttöön kutsumalla kyselyssä metodia `.AsParallel()` (katso koodiesimerkit 3 ja 4). (Freeman 2012, 219-220.)

```
namespace Esimerkin_nimiavaruus
{
    class Esimerkki
    {
        public static void Main()
        {
            int[] numeroita = new int[10] {1,2,3,4,5,6,7,8,9,10};

            var LINQ_tulokset = numeroita.Where(n => n > 5).Sum();

            Console.WriteLine("LINQ kyselyn tulokset {0}", LINQ_tulokset);

            // Tulostaa konsoliin numeroista 1-10 summan, johon lasketaan vain
            // 5 suuremmat luvut.
        }
    }
}
```

Koodiesimerkki 3. Esimerkki LINQ-kyselystä

```
class Example
{
    static void Main()
    {
        var numeroita = Enumerable.Range(0, int.MaxValue());

        // Lajitellaan numeroita rinnakkain.
        // Suodatetaan kaikki 10:llä jaolliset numerot

        var PLINQ_kysely = from num in numeroita.AsParallel()
                           where num % 10 == 0
                           select num;

        // Tulostetaan tulokset
        PLINQ_kysely.ForAll((e) => {
            Console.WriteLine(e);
        });

    }
}
```

Koodiesimerkki 4. Esimerkki PLINQ-kyselystä

PLINQ:n tuottamat tulokset saattavat olla eri järjestyksessä kuin missä elementit on esitelty kyselylle. Tämän ongelman voi korjata kutsumalla metodia `.AsOrdered()` kyselyssä, mutta sen kutsuminen vaikuttaa suorituskykyyn. Suorituskyky laskee, mikäli PLINQ:n tulee pitää kirjaa siitä mikä oli kunkin elementin alkuperäinen sijainti (Freeman 2012, 226–229).

PLINQ:n käyttö sopii hyvin taustalla suoritettaviin tehtäviin. Sillä on kuitenkin samat rajoitukset kuin TPL:llä toteutetussa koodissa: PLINQ:n suorittamat tehtävät tulee olla säieturvallisia.

3 SOVELLUKSET

3.1 Yleistä sovelluksista

Sovelluksia on kaksi, joista molemmat tuovat esille rinnakkaisohjelmointiin liittyviä etuja ja mahdollisia haittoja. Molemmille sovellukselle on kirjoitettu lyhyt tiivistelmä toteutuksesta.

Testit on toteutettu AMD:n valmistamalla 6-ytimisellä FX-6350-prosessorilla. Sovellukset on toteutettu Visual Studio 2013 -ohjelmointiympäristössä ja sovellusten testituloskaaviot on tehty sovellusliitännäisellä, jonka nimi on Visual Studion Concurrency Analyzer.

3.2 Sovellus 1: esimerkki rinnakkaisuudesta

3.2.1 Sovelluksen kuvaus ja toteutus

Ensimmäinen sovellus on yksinkertainen esimerkki rinnakkaisesta ohjelmoinnista. Ohjelma luo miljoona `Kayntikortti`-oliota (katso koodiesimerkki 5) ja täyttää näiden olioiden `etuNimi` ja `sukuNimi` merkkijono -tyyppiset kentät satunnaisella merkkijonolla. Lisäksi oliot lisätään `Kayntikortti`-tyyppiseen listaan. Olioiden käsittelystä mitataan siihen kuluva suoritusaika. Ohjelmiin on lisätty `Thread`-luokan staattisen metodin `.Sleep()` kutsuminen ennen varsinaista tietojenkäsittelyosuutta ja sen jälkeen. Metodi `.Sleep()` nimensä mukaisesti pysäyttää kaiken suorittamien. Sulkujen sisälle annettu parametri on pysähdyksen kesto millisekunteina.

```
namespace sovellus1
{
    public class KayntiKortti
    {
        string etuNimi       = "";
        string sukuNimi      = "";

        public KayntiKortti(string p_etuNimi, string p_sukuNimi){
            etuNimi       = p_etuNimi;
            sukuNimi      = p_sukuNimi;
        }

        public void TaytaKayntiKortti()
        {
            etuNimi  = arvoMerkkeja();
            sukuNimi = arvoMerkkeja();
        }
```

Koodiesimerkki 5. Sovelluksen jäsenkentät

Ohjelman alussa luodaan ilmentymä listasta, johon kortit laitetaan sekä yksi `Kayntikortti`-tyyppinen olio, `tempKayntiKortti`, jota käytetään silmukassa. Lisäksi luodaan `Stopwatch`-tyyppinen olio `System.Diagnostics`-nimiavaruudesta ajan mittaamista varten. `Kayntikortti`-oliot luodaan käsittelemällä niitä miljoonan iteraation `for`-silmukassa. Aluksi täytetään käynnistämisen yhteydessä luotu `tempKayntiKortti` uusilla arvoilla, jonka jälkeen se lisätään listaan (katso koodiesimerkki 6).

Miljoonan iteraation jälkeen mitataan suoritukseen kulunut aika `Stopwatch`-oliosta. `Stopwatch`-olion antama aika tulostetaan tämän jälkeen käyttäjälle.

```
class Program
{
    static void Main(string[] args)
    {
        Random r = new Random();
        List<KayntiKortti> kayntiKorttiLista = new List<KayntiKortti>();
        KayntiKortti tempKayntiKortti = new KayntiKortti("", "");
        Stopwatch stopwatch = new Stopwatch();

        System.Threading.Thread.Sleep(4000);
        stopwatch.Start();
        for (int i = 0; i < 1000000; i++)
        {
            tempKayntiKortti.TaytaKayntiKortti();
            kayntiKorttiLista.Add(tempKayntiKortti);
        }
        stopwatch.Stop();

        Console.WriteLine("Valmis: " + stopwatch.Elapsed);
        Console.ReadLine();
    }
}
```

Koodiesimerkki 6. Sovellus 1: n pääohjelma sekventiaalisesti

Koodiesimerkki 8:ssa sovellus on toteutettu käyttäen perinteistä säikeistysmallia. Säikeitä luodaan kuusi kappaletta ja jokaiselle niille annetaan parametriksi metodi `luoKayntiKortteja()`. Tämän metodin toiminta on esitelty koodiesimerkissä 7.

```
public void luoKayntiKortteja(){
    // Jokainen säie luo oman KayntiKortti olion, joka lisätään yhteiseen listaan
    KayntiKortti tempKayntiKortti = new KayntiKortti("", "");

    while (this.i < 1000000) // käytetään jaettua resurssia.
    {
        // Täytetään tempKayntiKortti satunnaisilla merkkijonoilla
        tempKayntiKortti.taytaKayntiKortti(arvoMerkkeja(), arvoMerkkeja());

        lock (this.lukko) // Lukitaan listaan lisääminen vain yhdelle säikeelle
        {
            kayntiKorttiLista.Add(tempKayntiKortti); // Lisätään yhteiseen listaan
            this.i++;
        }
    }
}
```

Koodiesimerkki 7. Käyntikorttien luominen

Säikeet jakavat kokonaislukutyyppisen jäsenmuuttujan nimeltä `i`, joka laskee kuinka monta käyntikorttia on jo lisätty listaan. Jäsenmuuttuja `i` on esitelty luokan alussa ja siihen viitataan luokan sisäisistä metodeista `this`-avainsanalla.

Säikeet käynnistetään `Thread`-luokan metodilla `.Start()` ja niiden päättymistä odotetaan metodilla `.Join()`. On huomioitava, että `.Join()`-metodi lukitsee käyttöliittymän odottamisen ajaksi. Se ei sovi käytettäväksi ohjelmissa, jotka vaativat jonkinlaisen käyttöliittymän.

```
class Program
{
    static Random random = new Random();
    List<KayntiKortti> kayntiKorttiLista = new List<KayntiKortti>();
    int i = 0;
    Object lukko = new Object();

    static void Main(string[] args)
    {
        Program p = new Program();
        Stopwatch stopwatch = new Stopwatch();

        int saikeiden_maara = 6;
        Thread[] saikeet = new Thread[saikeiden_maara];

        System.Threading.Thread.Sleep(3000);
        stopwatch.Reset();
        stopwatch.Start();

        for (int i = 0; i < saikeiden_maara; i++)
        {
            Thread t = new Thread(new ThreadStart(p.luoKayntiKortteja));
            saikeet[i] = t;
        }
        for (int i = 0; i < saikeiden_maara; i++)
        {
            saikeet[i].Start();
        }
        for (int i = 0; i < saikeiden_maara; i++)
        {
            saikeet[i].Join();
            Console.WriteLine("Saie " + i + "lopettaa: " + stopwatch.Elapsed);
        }
        stopwatch.Stop();
    }
```

Koodiesimerkki 8. Sovellus 1:n versio, jossa säikeet on tehty käsin

Task Parallel Library-toteutus on kuvattu koodiesimerkissä 9. Ohjelma on paljon selkeämpi luettavuudeltaan kuin koodiesimerkissä 8 esitelty klassinen säikeistysmalli. `Parallel`-luokan staattinen metodi `.For()` mukailee perinteistä `for`-silmukkaa. Tässä tapauksessa sillä on kolme parametria. Ensimmäinen on iteraation alkuarvo eli minkä kokonaisluku arvon muuttuja `i` saa, kun silmukkaa suoritetaan ensimmäistä kertaa. Toinen parametri on viimeisen suoritettavan iteraation luku. Kolmas parametri on delegaatti, jonka

arvoksi annetaan lambda-lause. Lambda-lause puolestaan sisältää kahden tavallisen metodin kutsut.

```
class Program
{
    static void Main(string[] args)
    {
        Random r = new Random();
        List<KayntiKortti> kayntiKorttiLista = new List<KayntiKortti>();
        KayntiKortti tempKayntiKortti = new KayntiKortti("", "");
        Stopwatch stopwatch = new Stopwatch();

        System.Threading.Thread.Sleep(3000);
        stopwatch.Start();
        Parallel.For (0, 1000000, i =>
        {
            tempKayntiKortti.TaytaKayntiKortti();
            kayntiKorttiLista.Add(tempKayntiKortti);
        });
        stopwatch.Stop();

        Console.WriteLine("Valmis: " + stopwatch.Elapsed);
        System.Threading.Thread.Sleep(3000);
        Console.Read();
    }
}
```

Koodiesimerkki 9. Sovellus 1:n versio TPL-toteutuksena

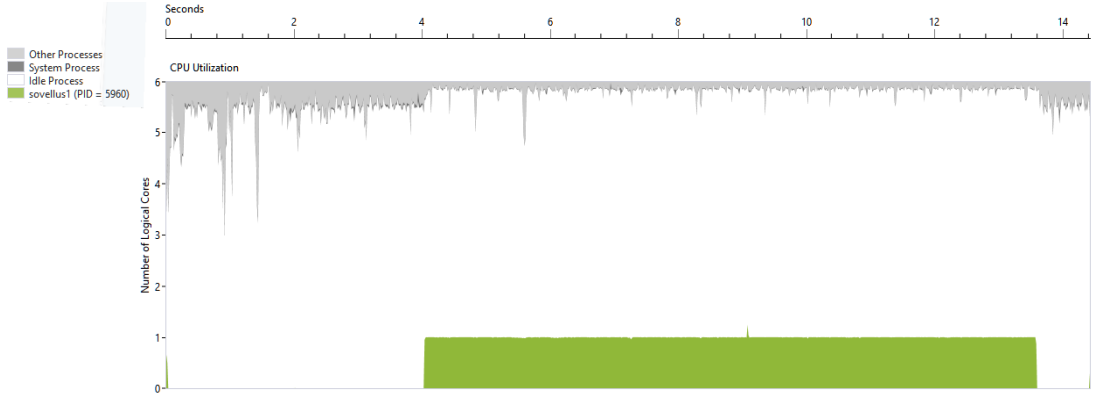
3.2.2 Sovelluksen testaaminen

Ohjelmat sisältävät suorituksen mittaamiseen tarvittavan `Stopwatch`-olion, joka kertoo ohjelman suorittamiseen tarvittavan ajan. Visual Studion Concurrency Analyzer-sovelluslaajennus tuottaa kaaviokuvia prosessoriytimien käyttöasteesta sekä erottelee jokaisen säikeen saaman suoritusajan eri ytimissä.

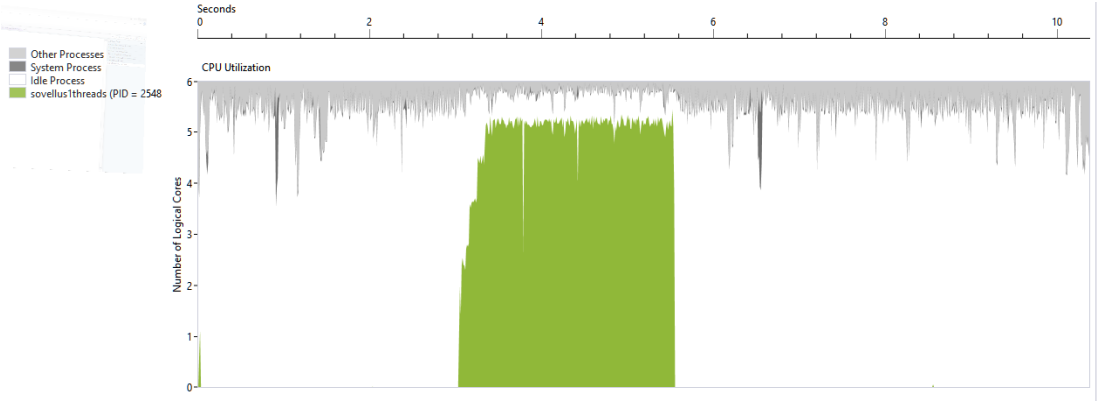
Tärkein mittausarvo on suoritukseen kulunut aika sekä Concurrency Analyzerin tuottamat mittaustiedot. Mittaustiedot ovat esiteltynä kahdessa muodossa. Ensimmäisessä muodossa kuvataan prosessoreiden kokonaiskäyttöastetta (katso kuviot 5, 6 ja 7).

3.2.3 Testitulosten analysointi

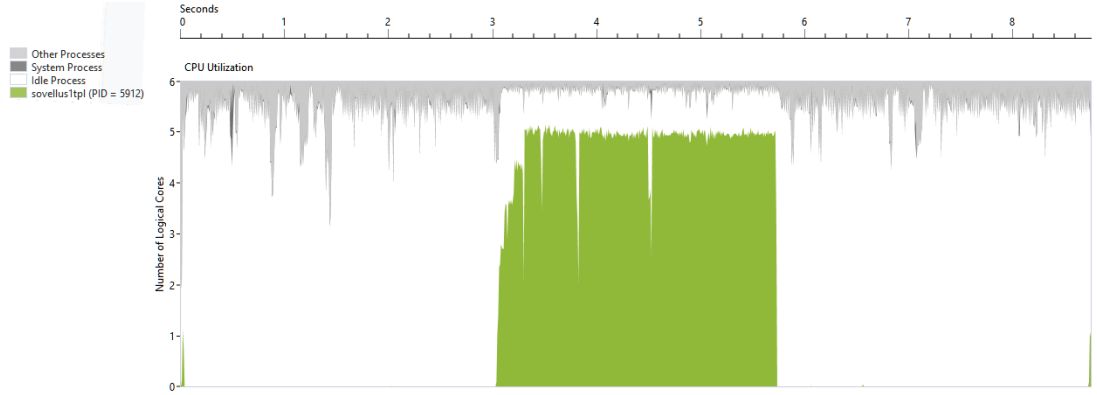
Analysoinnissa vertaillaan eri ohjelmointitapojen toteutuksista saatuja arvoja. Näistä mainittakoon tärkein eli kokonaisaika, jonka prosessi vaatii tehtävien suorittamiseen. Huomioitavaa on myös kuvaajat joissa näkyvät kuorman jakautuminen eri ytimille.



Kuvio 5. Prosessoriytimien käyttöaste sekventiaalisella toteutustavalla



Kuvio 6. Prosessoriytimien käyttöaste klassisella säikeistysmallilla

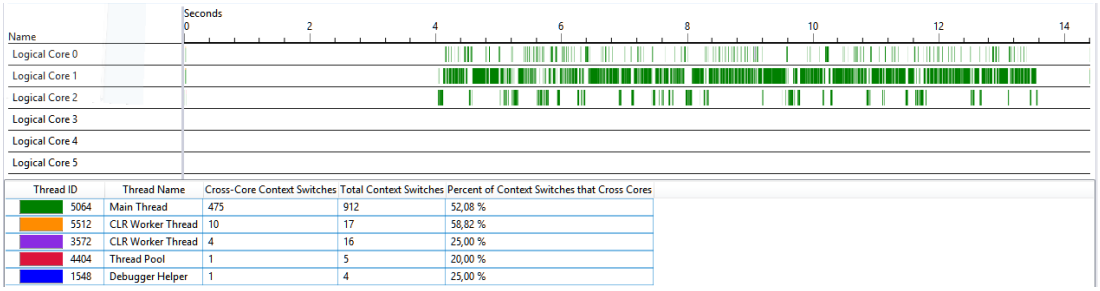


Kuvio 7. Prosessoriytimien käyttöaste TPL-toteutuksena

Ensimmäisellä sekventiaalisella ohjelmointitavalla huomataan, että kokonaisaika on noin 9,6174 sekuntia. (kuvio 5), kun se muilla implementaatio-tavoilla on noin 2,5 - 2,7 sekunnin välillä (kuviot 6 ja 7), joissa kuormaa jaetaan useammalle ytimelle yhden sijasta. Klassisen säikeistysmallin kokonaisaika on noin 2,5735 sekuntia (kuvio 6) ja Task Parallel Library-toteutuksella noin 2,7621 sekuntia.

Kaaviokuva prosessiytimien käytöstä myös puoltaa tätä, koska sekventiaalisella tavalla ohjelmoidusta prosessista melkein kaikki kuorma

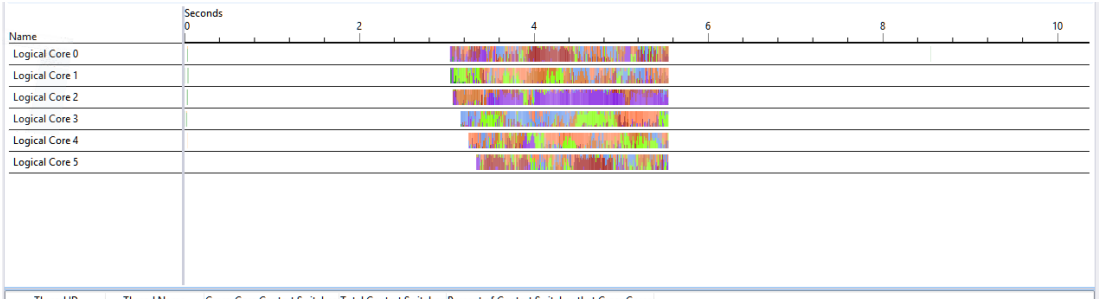
menee yhdelle ytimelle (kuvio 5). Kahdessa muussa esimerkissä kuorman jakautuminen menee kaikille kuudelle loogiselle ytimelle tasaisesti kuten nähdään kuvajaasta (kuviot 6 ja 7).



Thread ID	Thread Name	Cross-Core Context Switches	Total Context Switches	Percent of Context Switches that Cross Cores
5064	Main Thread	475	912	52,08 %
5512	CLR Worker Thread	10	17	58,82 %
3572	CLR Worker Thread	4	16	25,00 %
4404	Thread Pool	1	5	20,00 %
1548	Debugger Helper	1	4	25,00 %

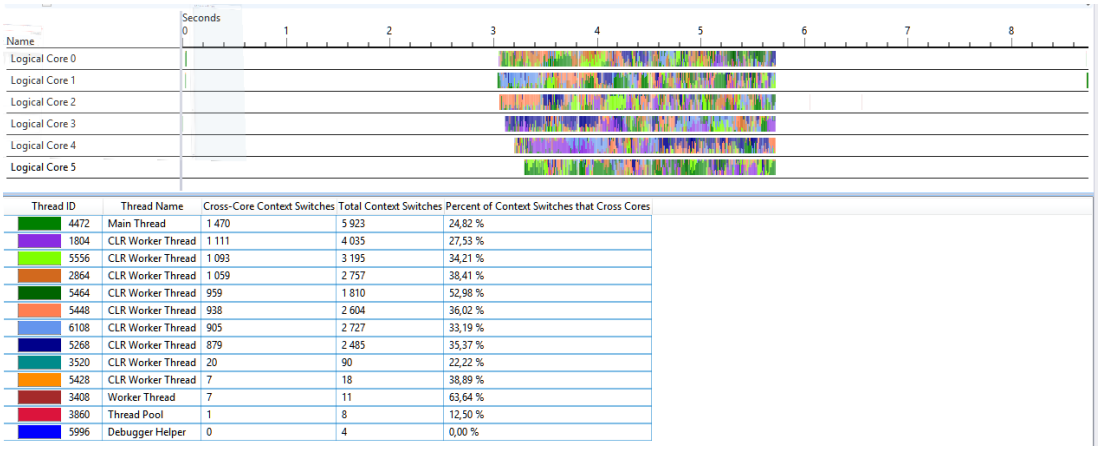
Kuvio 8. Säikeiden suorittaminen eri ytimillä sekventiaalisella toteutustavalla

Ensimmäisellä ohjelmalla on vain yksi säie käytössä kerrallaan (kuvio 8). Yksi säie on jakautunut vuorotellen kolmelle eri ytimelle. Kuorman siirtyessä ytimeltä toiselle on vain yksi ydin kerrallaan aktiivisena toisten ytimien ollessa idle-tilassa. Ydin ei idle-tilassa suorita mitään muutakaan sovellusta, vaikka onkin käynnissä.



Thread ID	Thread Name	Cross-Core Context Switches	Total Context Switches	Percent of Context Switches that Cross Cores
2936	CLR Worker Thread	922	6 279	14,68 %
7552	CLR Worker Thread	690	3 688	18,71 %
7416	CLR Worker Thread	668	2 656	25,15 %
6500	CLR Worker Thread	627	2 437	25,73 %
5912	CLR Worker Thread	626	3 042	20,58 %
7708	CLR Worker Thread	589	2 379	24,76 %
6904	Main Thread	32	129	24,81 %
3680	CLR Worker Thread	6	12	50,00 %
4236	CLR Worker Thread	0	11	0,00 %
8092	Thread Pool	0	6	0,00 %
3060	Debugger Helper	0	3	0,00 %

Kuvio 9. Säikeiden suorittaminen eri ytimillä käsin tehdyillä säikeillä



Thread ID	Thread Name	Cross-Core Context Switches	Total Context Switches	Percent of Context Switches that Cross Cores
4472	Main Thread	1 470	5 923	24,82 %
1804	CLR Worker Thread	1 111	4 035	27,53 %
5556	CLR Worker Thread	1 093	3 195	34,21 %
2864	CLR Worker Thread	1 059	2 757	38,41 %
5464	CLR Worker Thread	959	1 810	52,98 %
5448	CLR Worker Thread	938	2 604	36,02 %
6108	CLR Worker Thread	905	2 727	33,19 %
5268	CLR Worker Thread	879	2 485	35,37 %
3520	CLR Worker Thread	20	90	22,22 %
5428	CLR Worker Thread	7	18	38,89 %
3408	Worker Thread	7	11	63,64 %
3860	Thread Pool	1	8	12,50 %
5996	Debugger Helper	0	4	0,00 %

Kuvio 10. Säikeiden suorittaminen eri ytimillä TPL-toteutuksena

Kahdessa toisessa esimerkissä huomataan, että säikeitä on useita, joita myös siirrellään ytimeltä toiselle ja samanaikaisesti toinen säie alkaa tekemään työtään ytimellä josta aikaisempi säie on lähtenyt. Säikeen siirtymistä toiselle ytimelle suorittettavaksi kutsutaan kontekstin vaihdoksi. Kuviossa 9 on esitelty klassisen säikeistysmallin tulokset. Ohjelma luo kuusi säiettä ja suorittaa niitä jokaisella ytimellä.

Jos vertaillaan TPL-toteutusta (kuvio 9) ja manuaalisesti syötettyjen säikeiden määrää (kuvio 10) niin on huomattavaa, että ainakin tässä tapauksessa manuaalinen tapa on noin 0,2 sekuntia nopeampi. Tuloksista käy myös ilmi, että manuaalisesti syötetyissä säikeiden määrässä säikeitä on pääsäikeen lisäksi kuusi, kun taas TPL:ssä niitä on pääsäikeen lisäksi yhdeksän.

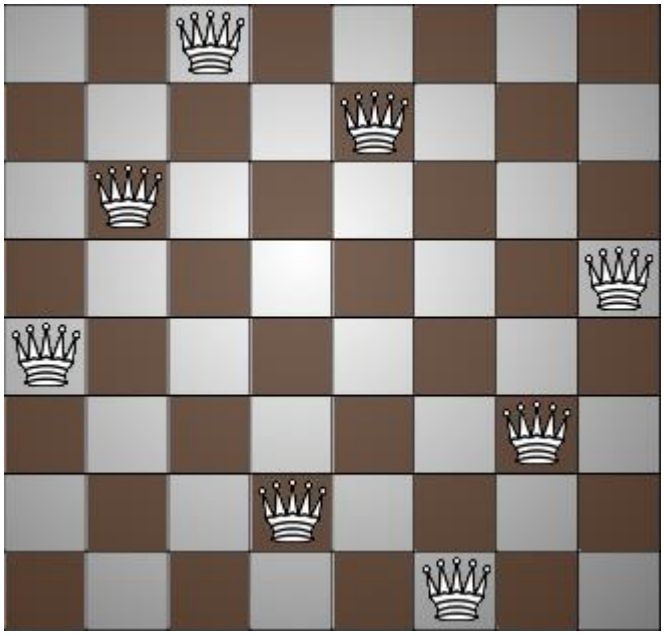
Pääsäikeen käytössä on myös eroja näiden kahden tavan välillä. TPL:ssä pääsäiettä käytetään eniten, kun taas klassisessa säikeistysmallissa pääsäikeen käyttö on lähes olematonta.

3.3 Sovellus 2: Nqueens-ongelman ratkaisija

3.3.1 Sovelluksen kuvaus ja toteutus

Toinen sovellus on ratkaisijasovellus matemaattiseen ongelmaan nimeltään ”Nqueens”. Ongelma on seuraavanlainen: Annetaan N:lle jokin kokonaislukuarvo, joka edustaa shakkilaudan leveyttä ja korkeutta sekä laudalle asetettavien kuningattarien lukumäärää. Kuningatar voi liikkua shakkilaudalle rajattomasti joko viistosti, vaakatasossa tai pystytasossa.

Tehtävänä on järjestää kuningattaret shakkilaudalle siten, että ne eivät pysty syömään eli liikkumaan toistensa yli ensimmäisellä siirrolla ja selvittää kuinka monta tällaista ratkaisua on olemassa (katso esimerkkiratkaisu kuviosta 11).



Kuvio 11. Esimerkkiratkaisu kun N on 8

Sovellukseen tehdään graafinen käyttöliittymä, jossa käyttäjä voi valita N:n arvon. Ohjelma laskee ratkaisut kaikilla kolmella suoritustavalla ja tulostaa laskemiseen kuluneet ajat käyttäjälle vertailtavaksi. Käyttöliittymään tulostetaan myös kokonaisratkaisujen määrä.

Shakkilauta toteutetaan kokoelmana `Ruutu`-olioita, joiden maksimimäärä on käyttäjän antama N-arvo kertaa kaksi. Tälle shakkilaudalle ohjelma piirtää oikean ratkaisun aina kun oikea ratkaisu syntyy. Ohjelmaan luodaan N-kappaletta `Kuningatar`-olioita, jotka sisältävät kokonaislukumuuttujat `i`,`x` ja `y` sekä totuusarvomuuttujan `isMoved`. Muuttuja `i` edustaa kyseisen olion indeksiä, `x` ja `y` koordinaatteja shakkilaudalla ja `isMoved` asetetaan todeksi, kun kyseistä oliota on käsitelty kyseisellä vuorolla.

```
stopwatch.Start(); // käynnistetään ajanotto
while (siirtojen_maara <= siirrot_yhteensa -1) // toistetaan kunnes kaikki mahdolliset siirrot on tehty
{
    foreach (Kuningatar k in kuningattaret) // Laudan kuningatar-olioiden kokoelma
    {
        if (k.liiku == true) // Jos kuningatar saa liikkua
        {
            var r1 = shakkiLauta.First(s => s.x == k.x && s.y == k.y);
            r1.varattu = false; // asetetaan edellinen ruutu vapaaksi
            k.y++;
            if (k.y == n) // Tarkistetaan voidaanko liikkua eteenpäin vai palataanko alkuun
            {
                k.y = 0;
                // jos palataan alkuun, annetaan seuraavalle kuningattarelle liikkumismahdollisuus
                if (k.i != n - 1) { kuningattaret.ElementAt(k.i + 1).liiku = true; }
            }
            var r2 = shakkiLauta.First(s => s.x == k.x && s.y == k.y);
            r2.varattu = true; // asetetaan uusi ruutu varatuksi
            if (k.i != 0)
            {
                k.liiku = false; // kuningatar pysyy seuraavan kierroksen paikoillaan
            }
        }
    }

    if (tarkista_ratkaisu(kuningattaret))
    {
        // d1 on delegaatti käyttöliittymään
        d1(shakkiLauta); // tarkistetaan onko laudalle syntynyt kuvio oikea ratkaisu
    }
    siirtojen_maara++;
}
stopwatch.Stop(); // pysäytetään ajanotto
// d2 on delegaatti käyttöliittymään
d2(stopwatch.Elapsed);
```

Koodiesimerkki 10. Sekventiaalisen ratkaisualgoritmin kuningattarien sijoittelulogiikka

Kuningattaret asetellaan taulukon 0,0-indeksistä alkaen siten, että jokainen uusi kuningatar sijoitetaan seuraavaan vapaaseen ruutuun, jolloin x-koordinaatti kasvaa (koodiesimerkki 11).

```
while (i < n) // kuningattaria on n-kappaletta
{
    if (i == 0) // ensimmäinen kuningatar
    {
        Kuningatar k = new Kuningatar(i, i, 0, true); // ensimmäinen kuningatar liikkuu
        kuningattaret.Add(k); // lisätään kuningatar-kokoelmaan
    }
    else
    {
        Kuningatar k = new Kuningatar(i, i, 0, false); // muut eivät liikkuu
        kuningattaret.Add(k); // lisätään kuningatar-kokoelmaan
    }
    shakkiLauta.ElementAt(i).varattu = true; // Varataan shakkilaudan ruutu

    i++;
}
```

Koodiesimerkki 11. Kuningattarien sijoittelu shakkilaudalle

Ongelmaa tarkastellessa huomataan, että ratkaisu ongelmaan on tilanne, jossa millään `Kuningatar`-oliolla ei ole sama x- tai y-koordinaatti. Toisin sanoen vain yksi kuningatar voi olla kerrallaan yhdellä x- tai y-koordinaatilla.

Sovellus alkaa ratkaista ongelmaa siten, että `Kuningatar`-olion, jonka indeksi-muuttuja on 0 (0-kuningatar), y-koordinaattia kasvatetaan yhdellä. Tämän jälkeen tarkastetaan onko ratkaisu syntynyt. Jos ratkaisu on syntynyt, sovellus lisää ratkaisujen määrään yhden. Tätä toistetaan, kunnes saavutetaan y-koordinaatti, joka on sama kuin N. Kun jonkin `kuningatar`-olion y-koordinaatti on N, annetaan siirron tapahtua myös kyseisen kuningattaren indeksistä seuraavalle kuningattarelle. Tätä toistetaan, kunnes saavutetaan N:nen kuningattaren y-koordinaatin arvo N. Ongelman ratkaiseminen on esitelty kuviossa 10.

Tarkastusalgoritmi tarkastaa ensin, onko kahdella kuningattarella sama Y-indeksin arvo. Seuraavaksi tarkastetaan viistosti oikealla ylöspäin kasvavat indeksit ja lopuksi viistosti alaspäin kasvavat indeksit. Erilliselle X-akselin tarkastelulle ei ole tarvetta, koska kuningattaret liikkuvat vain oman Y-akselin suuntaisesti. Tarkastusalgoritmi on esitelty koodiesimerkissä 12.

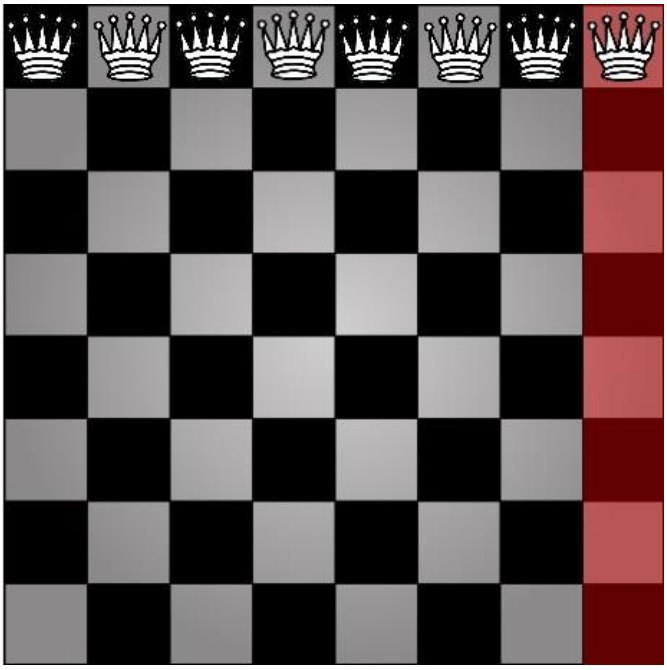
```
private bool tarkista_ratkaisu(List<Kuningatar> k) {
    int kuningatar_indeksi = 0;
    int i = 0;
    int j = 0;
    int ki = 0;

    // käydään läpi kaikkien kuningattarien koordinaatit
    while(kuningatar_indeksi < k.Count - 1){

        i = kuningatar_indeksi + 1;

        // tarkastetaan onko silmukassa oleva kuningatar
        // samalla Y-akselin koordinaatilla.
        // HUOM. X-akselia ei tarvitse tarkastaa, koska
        // jokainen kuningatar on omassa X-koordinaatissa.
        while (i < k.Count) {
            if (k.ElementAt(kuningatar_indeksi).y == k.ElementAt(i).y) {
                return false; // ei oikea ratkaisu
            }
            i++;
        }

        // tarkastetaan onko silmukassa oleva kuningatar
        // samalla koordinaatilla yläviistosti
        j = k.ElementAt(kuningatar_indeksi).y - 1;
        ki = k.ElementAt(kuningatar_indeksi).x + 1;
        while (j >= 0 && ki < n) {
            var loydetty_ruutu = k.FirstOrDefault(s => s.x == ki && s.y == j);
            if (loydetty_ruutu != null) {
                return false;  // ei oikea ratkaisu
            }
            ki++;
            j--;
        }

        // tarkastetaan onko silmukassa oleva kuningatar
        // samalla koordinaatilla alaviistosti
        j = k.ElementAt(kuningatar_indeksi).y + 1;
        ki = k.ElementAt(kuningatar_indeksi).x + 1;
        while (j < n && ki < n)
        {
            var loydetty_ruutu = k.FirstOrDefault(s => s.x == ki && s.y == j);
            if (loydetty_ruutu != null)
            {
                return false;  // ei oikea ratkaisu
            }
            ki++;
            j++;
        }

        kuningatar_indeksi++;
    } // while-loppu

    return true;  // oikea ratkaisu
}
```

Koodiesimerkki 12. Ratkaisualgoritmin kuvion tarkistuslogiikka

Sovelluksen käyttämä ns. brute force-algoritmi ei ole nopein tapa ratkaista Nqueens:n kaltaista ongelmaa. Brute force-menetelmässä sovellus käy läpi

kaikki mahdolliset kuningataryhdistelmät, joita on N potenssiin N kappaletta. N:n kasvaessa algoritmi osoittautuu lähes hyödyttämäksi, koska ratkaisujen laskeminen suurella N:n arvolla voi kestää useita vuosia. Sovelluksen tarkoitus ei kuitenkaan ole esittää parasta mahdollista algoritmia ongelman ratkaisemiseksi vaan toteuttaa algoritmi sekventiaalisesti sekä rinnakkain, jotta saadaan selville rinnakkaistoteutuksen edut.

Tehtävän toteuttaminen rinnakkain edellyttää tehtävän jakamista osiin. TPL:n käyttäminen algoritmissa sellaisenaan hidastaa ohjelman suorittamista ylimääräisen laskennan vuoksi. Algoritmi jakautuu helposti käyttäjän antaman N:n arvon mukaisesti (katso koodiesimerkki 13). Tehtävästä luodaan N kappaletta alitehtäviä, joissa jokaisessa shakkilaudan viimeinen `kuningatar` on staattinen eli sen sijainti ei muutu ratkaisujen laskemisen aikana (katso kuvio 12). Jokaisella alitehtävällä kuningattaren Y-indeksi kasvaa yhdellä.



Kuvio 12. Staattisten kuningatar-olioiden rivi väritettynä

Alitehtävät voidaan suorittaa toisista alitehtävistä lähes täysin riippumattomasti, joten säikeiden synkronointi jää vähäiseksi. Ainoa yhteinen alitehtävien jakama resurssi on synkronoitu delegaatin kutsu, joka piirtää oikean ratkaisun käyttöliittymään. Piirtämisen yhteydessä kasvatetaan oikeiden ratkaisujen määrää, joten alitehtävien tuloksien yhdistämiselle ei ole tarvetta.

```
private void ratkaise()
{

    stopwatch.Start(); // ajanoton aloitus

    Task[] tasks = new Task[n]; // n-kappaletta alitehtäviä
    for (int i = 0; i < n; i++)
    {
        // annetaan alitehtävälle lambda-lausekkeena
        // ratkaise_osa-metodi. Metodissa on laskuri, joka
        // kertoo monesko alitehtävä käynnistetään.
        // Alitehtävä saa sen mukaan n:nen rivin kuningattaren
        // staattisen arvon.
        tasks[i] = (Task.Factory.StartNew(() => ratkaise_osa()));
    }

    // odotetaan kunnes alitehtävät on
    // suoritettu loppuun
    Task.WaitAll(tasks);

    d2(stopwatch.Elapsed); //delegaatin kutsu käyttöliittymään
}
```

Koodiesimerkki 13. Algoritmin jakaminen TPL:n task-olioille

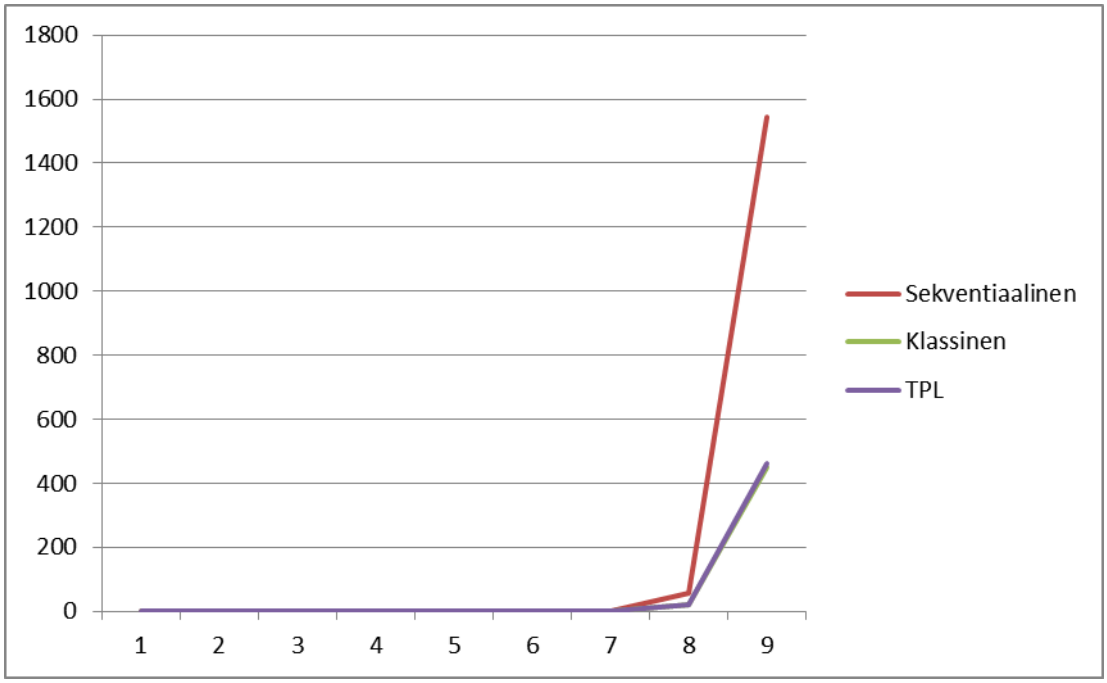
3.3.2 Sovelluksen testaaminen

Sovelluksen ajan mittaamiseen käytetään `stopwatch`-luokan ilmentymää kuten ensimmäisessäkin sovelluksessa. Lisäksi Concurrency Analyzer kerää mittadataa prosessoriytimien käyttöasteesta. Sovelluksesta valitaan N:n arvo jolloin ratkaisut lasketaan vuorotellen kaikilla toteutustavoilla.

Sovelluksesta vertaillaan myös suoritusaikaa eri N:n arvojen välillä. Suurin mitattu N:n arvo on yhdeksän. Sovelluksen käyttämästä algoritmista johtuen tätä suuremmat N:n arvot on jätetty pois, koska suoritusajat kasvavat suhteettoman suuriksi.

3.3.3 Testitulosten analysointi

Sovelluksen käyttämä brute force-algoritmi on eksponentiaalisesti kasvava algoritmi, joka kasvattaa ratkaisemiseen tarvittavaa työmäärää räjähdysmäisesti arvon N kasvaessa. Sekventiaalisella menetelmällä tehtävän ratkaisemiseksi kuluu yli 10 tuntia N:n ollessa 10. Klassisella sekä TPL-menetelmällä suoritukseen kuluva aika on lähes sama N:n arvosta riippumatta.



Kuvio 13. Suoritusaika kullakin toteutustavalla eri N:n arvoilla sekunteina.

Tarkoituksena on tuoda esille säikeistyksen tuoma etu. Sekä klassisella säikeistys- että TPL-menetelmällä tehokkuus kasvaa jopa 3,4 kertaiseksi N:n ollessa 9. Tehokkuus kasvaa mitä suuremmalla N:n arvolla tehtävää ratkaistaan.

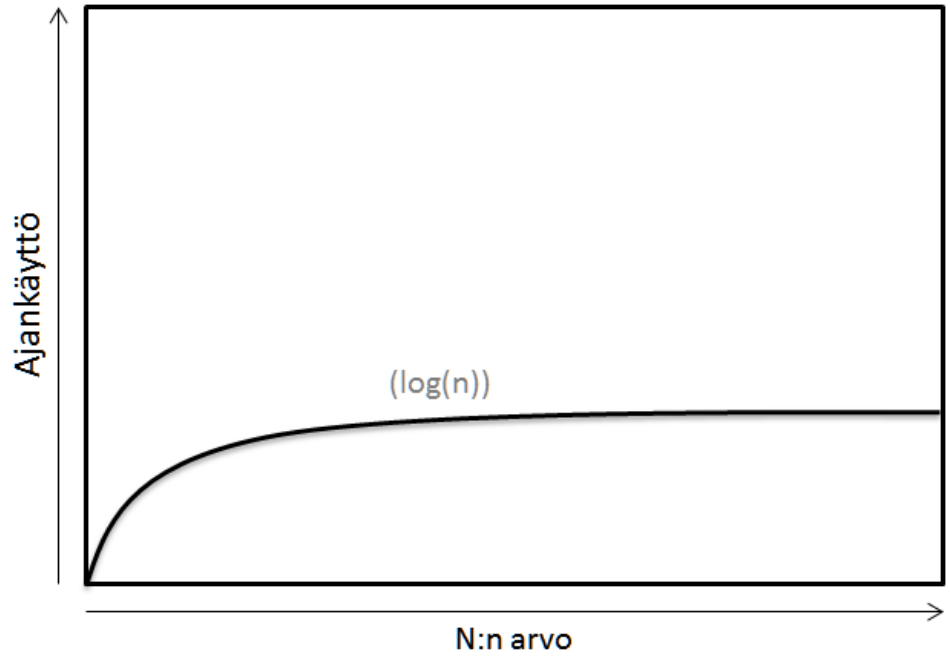
Hyödyt rinnakkaisesta toteutuksesta tulevat esille vasta N:n ollessa 6. Sekventiaalinen toteutustapa on suoraviivaisempi eikä vaadi ylimääräistä laskentaa säikeiden koordinoimiseen ja tämän takia toteutustapa on muita tapoja nopeampi. Mahdollisten ratkaisujen kasvaessa jokaiselle alitehtävälle syntyy riittävästi laskettavaa, joten säikeiden koordinointia on suhteessa työmäärään vähän.

Pienillä N:n arvoilla suoritusajat saattavat vaihdella suhteellisen rajusti. Tämä johtuu käyttöjärjestelmän sen hetkisestä kuormasta, joka saattaa estää suoritusajan jakamisen välittömästi sovellukselle.

N	Sekventiaalinen	Klassinen	TPL	Ratkaisujen määrä	Oikeita ratkaisuja
1	0,0005182	0,0007821	0,0005966	1	1
2	0,0000142	0,0013740	0,0000732	4	0
3	0,0000400	0,0076687	0,0019714	27	0
4	0,0013150	0,0022095	0,0031178	256	2
5	0,0087760	0,0316121	0,0134750	3125	10
6	0,1111266	0,0627937	0,0496512	46656	4
7	2,3258726	1,0866566	0,9790771	823543	40
8	57,0544051	20,6922287	19,2722624	16777216	92
9	1545,0104133	450,9356188	461,8928243	387420489	352

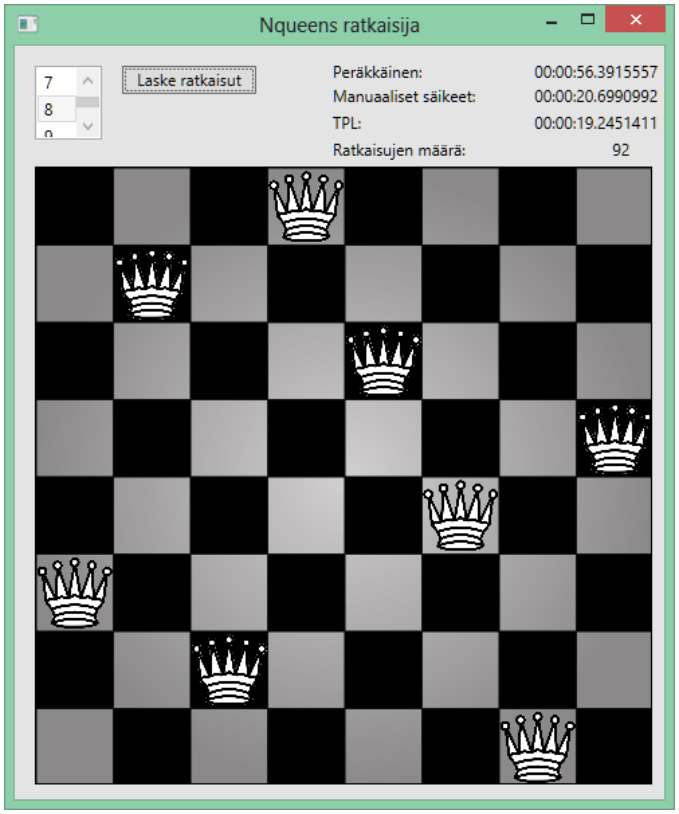
Taulukko 2. Tulokset sekunteina Nqueens-ongelman ratkaisemisesta eri N:n arvoilla.

Rinnakaisen toteutuksen nopeus on selkeästi parempi kuin sekventiaalisella toteutuksella, mutta siitä ei ole juurikaan apua mikäli N:n arvo on suuri. Algoritmin ollessa esimerkiksi logaritminen rinnakkaisesta toteutuksella on paljon hyötyä. Logaritmisen algoritmin ajankäyttö kasvaa vain hieman N:n arvon kasvaessa suureksi (Hilfiger 2006).



Kuvio 14. Logaritminen algoritmi

Logaritmisessa algoritmissa aikaa ei koskaan kulu niin paljon, että se olisi hyödytön (kuvio 14).



Kuvio 15. Kuvakaappaus sovelluksesta

Kuviossa 15 on esitelty sovelluksen graafinen käyttöliittymä. Ohjelma päivittää aina ratkaisun löytyessä kyseisen ratkaisun käyttöliittymän shakkilaudalle. Suoritusajat ja ratkaisujen kokonaismäärä tulostuvat oikeaan yläkulmaan suorituksen päätyttyä.

4 SUUNNITTELUMALLIT

4.1 Yleisesti suunnittelumalleista

Useampaa ydintä hyödyntävän ohjelman suunnittelu poikkeaa paljon normaalista sekventiaalisen ohjelman suunnittelusta, jossa yhteisiä jäsenmuuttujia voidaan käyttää suhteellisen riskittömästi ja ohjelman toteutus on lineaarista. Rinnakkaisohjelmoinnissa viittauksia luokkien julkisiin muuttujiin tulee välttää ja eri toiminnot erotella niin, ettei niillä ole yhteisiä resursseja käytettävissä.

Mitä suuremman osan ohjelmasta saa toimimaan rinnakkain, sitä enemmän hyötyä siitä on. Pieniä osia, kuten silmukoita, joiden sisällä ei ole paljoa laskentatehoa vaativia toimintoja, ei kannata ohjelmoida suoriutumaan rinnakkain. Pienien osien ohjelmoiminen suoriutumaan rinnakkain aiheuttaa säikeiden koordinointiin tarvittavaa ylimääräistä laskentaa, jolloin prosessin käyttämä aika voi pienenemisen sijaan kasvaa (Freeman 2012, 3-4).

Valmiin sekventiaalisen ohjelman muuntaminen rinnakkaiseksi on erittäin työlästä suunnittelutavasta riippuen. Mikäli jaettuja resursseja on käytetty paljon, niin paras ratkaisu on todennäköisesti suunnittella ja toteuttaa koko ohjelma uudestaan.

4.2 Rinnakkaissilmukka

Rinnakkaissilmukka on yksinkertaisuudessaan silmukassa tapahtuvan toiminnan jakamista eri ytimille. Yksi säie ottaa yhden indeksin arvon itselleen ja suorittaa tällä arvolla tapahtuvan toiminnan silmukassa ennen kuin se siirtyy seuraavaan vapaaseen indeksiin. Koodiesimerkki 14 esittää kuinka `Parallel.For()`-silmukka muodostetaan.

```
int n = jokin_kokonaisluku;
Parallel.For(0, n, i =>
{
    // For-loopin sisältö
});
```

Koodiesimerkki 14. Esimerkki Parallel.For() silmukasta (Microsoft Developer Network 2014j.)

Toteutustavan haaste on koordinoida yhteisten resurssien käyttöä, kuten yhteisiä jäsenmuuttujia. Koordinoimaton yhteisten resurssien käyttö voi aiheuttaa tilanteen, jossa kaksi säiettä kirjoittavat samaan muuttujaan samaan aikaan tietoa.

Ohjelmoija ei voi myöskään luottaa suoritettavan silmukan indeksiin eli iteraatiolukuun. Ongelma rinnakkain suoritettavassa sovelluksessa on, että silmuikoita ei välttämättä suoriteta numerojärjestyksessä, vaan 8. iteraatio voidaan suorittaa ennen 5. iteraatiota. (Microsoft Developer Network 2014g.)

4.3 Rinnakkaistehtävät

Rinnakkaistehtävillä tarkoitetaan toisistaan riippumattomien tehtävien suorittamista eri ytimillä. Rinnakkaissilmukoiden kohdalla suoritetaan samaa toiminnallisuutta usealla eri ytimellä, kun taas rinnakkaistehtävät ovat toisistaan riippumattomia eikä niillä ole samoja rajoituksia. Koordinointia kuitenkin tarvitaan mikäli rinnakkaistehtävät käyttävät toistensa metodeja. Koodiesimerkissä 15 on esimerkki kuinka kaksi toisistaan riippumatonta tehtävää käynnistyy yhtäaikaa ja ohjelma jää odottamaan niiden päättymistä. (Microsoft Developer Network 2014h.)

```
Task t1 = Task.Factory.StartNew(teeEnsimmainenAsia);
Task t2 = Task.Factory.StartNew(teetoinenAsia);

Task.WaitAll(t1, t2);
```

Koodiesimerkki 15. Esimerkki rinnakkaisista tehtävistä

Rinnakkain suoritettavia silmukoita käyttäessä tulee yleensä jossain vaiheessa vastaan tilanne, jossa tietoa pitää saada luettua tai kirjoitettua yhteiseen muistipaikkaan. Tähän ongelmaan ratkaisuna toimivat lukot, joilla

saadaan tietty muuttuja tai funktio lukittua tietylle säikeelle käyttöön. Muut säikeet odottavat tämän lukon avautumista, ennen kuin ottavat lukon itselleen ja suorittavat lukitun toiminnallisuuden.

Lukkojen käytön ongelmana on, että muut säikeet joutuvat odottamaan lukittua elementtiä. Odottava säie kuluttaa järjestelmän resursseja, vaikka ei suorita mitään toiminnallisuutta.

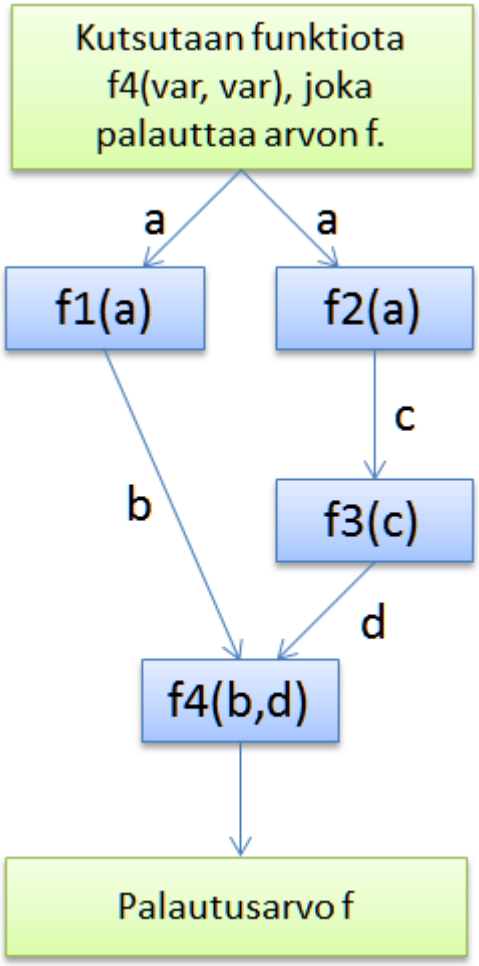
4.4 Rinnakkaisaggregaatti

Rinnakkaisaggregaatti käyttää paikallisia muuttujia silmukoissa, jotka yhdistetään silmukoiden päätyttyä yhteiseksi palautusarvoksi. Tällaiset osittaiset ja paikallisesti suoritettavat tehtävät ovat toisistaan riippumattomia. (Microsoft Developer Network 2014i.)

Rinnakkaisaggregaatti ei vaadi synkronointia, koska silmukat eivät jaa mitään yhteistä resurssia. Algoritmi on joskus helpompi ohjelmoida käyttämään rinnakkaisaggregaattia kuin lisätä siihen synkronisointia. Helppo tapa käyttää rinnakkaisaggregaattia on lisätä PLINQ:n metodi `.AsParallel()` normaaliin LINQ-kyselyyn. (Microsoft Developer Network 2014i.)

4.5 Futuurit

Futuurit liittyvät aikasemmin mainittujen rinnakkaistehtävien suoritukseen, joissa toisistaan riippumattomat tehtävät suoritetaan rinnakkain. Futuurilla voidaan jaksottaa koodia niin, että osa tehtävistä suoritetaan ennen kuin jotain tiettyä tehtävää aletaan suorittamaan. Esimerkiksi tehtävä C voidaan suorittaa vasta kun tehtävät A ja B on suoritettu, koska näistä palautuvia arvoja tarvitaan tehtävän C suorittamiseen. (Microsoft Developer Network 2014j.)



Kuvio 16. Fuuturi-kaavio (Microsoft Developer Network 2014j.)

Kuviossa 16 on esitelty kuinka funktion `f4` parametrit ovat riippuvaisia toisten funktioiden `f1`, `f2` ja `f3` palautusarvoista. Funktiot haluataan suorittaa rinnakkain. Kaaviosta huomaa, että funktiot `f1` ja `f2` voidaan suorittaa rinnakkain, koska niillä ei ole riippuvuussuhteita muihin funktioihin.

4.6 Dynaamiset rinnakkaistehtävät

Tehtäviä voidaan dynaamisesti lisätä suoritettavien tehtävien jonoon ohjelman suorituksen edetessä, jolloin puhutaan dynaamisista rinnakkaistehtävistä. Dynaamisuus tapahtuu esimerkiksi tilanteessa, jossa sekventiaalisessa koodissa tapathuu rekursio, eli metodi kutsuu itseään. (Microsoft Developer Network 2014k.)

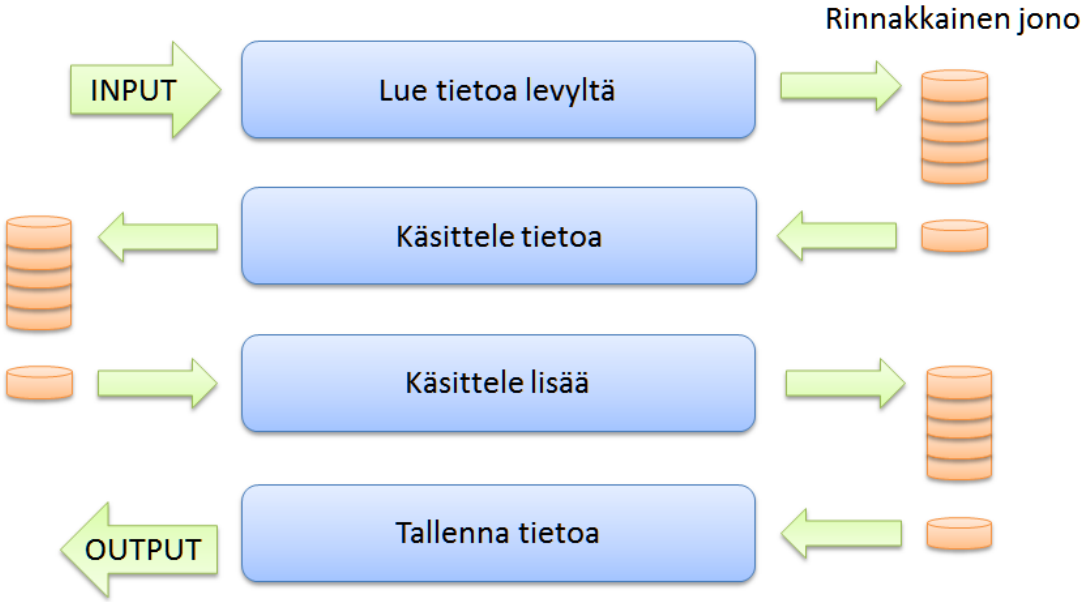
Dynaamisuutta voidaan ohjelmoida myös algoritmeihin, jotka saavat tuntemattoman määrän tietoa käsiteltäväksi. Ohjelmassa voidaan ennalta määrätä, kuinka paljon tietoa kukin tehtävä saa. Toteutettavien tehtävien

määrä riippuu käsiteltävän tiedon määrästä. (Microsoft Developer Network 2014k.)

4.7 Tiedonsiirtokanavat

Algoritmin eri osat voivat olla sidoksissa toisiinsa niin, että sen suorittaminen yhdessä säikeessä useita kertoja rinnakkaisessa silmukassa ei ole järkevää. Algoritmi voi myös olla käytössä reaaliaikaisessa järjestelmässä, jolloin käsiteltävien tietojen määrä vaihtuu ajankohdasta riippuen. Tällöin ei ole tehokasta käyttää rinnakaista silmukkaa, koska tietoa voidaan käsitellä varsin vähän. (Microsoft Developer Network 2014f.)

Tiedonsiirtokanavat ovat rinnakkain suoritettavia sekventiaalisia tehtäviä, jotka ovat riippuvaisia toisistaan ja niiden välille on rakennettu rinnakkain toimiva jono (`System.Collections.Concurrent.ConcurrentQueue`) tietorakenne. (Microsoft Developer Network 2014f.)



Kuvio 17. Tiedonsiirtokanavat (Microsoft Developer Network 2014f.)

Tiedonsiirtokanavista muodostuu linjasto, jossa toiminnallisuutta kuvaavat kuvion 17 siniset laatikot voidaan suorittaa rinnakkain niin itseensä kuin toisiinsakin nähden.

5 RINNAKKAISUUDEN ONGELMAT

5.1 Yleisesti rinnakkaisuuden ongelmista

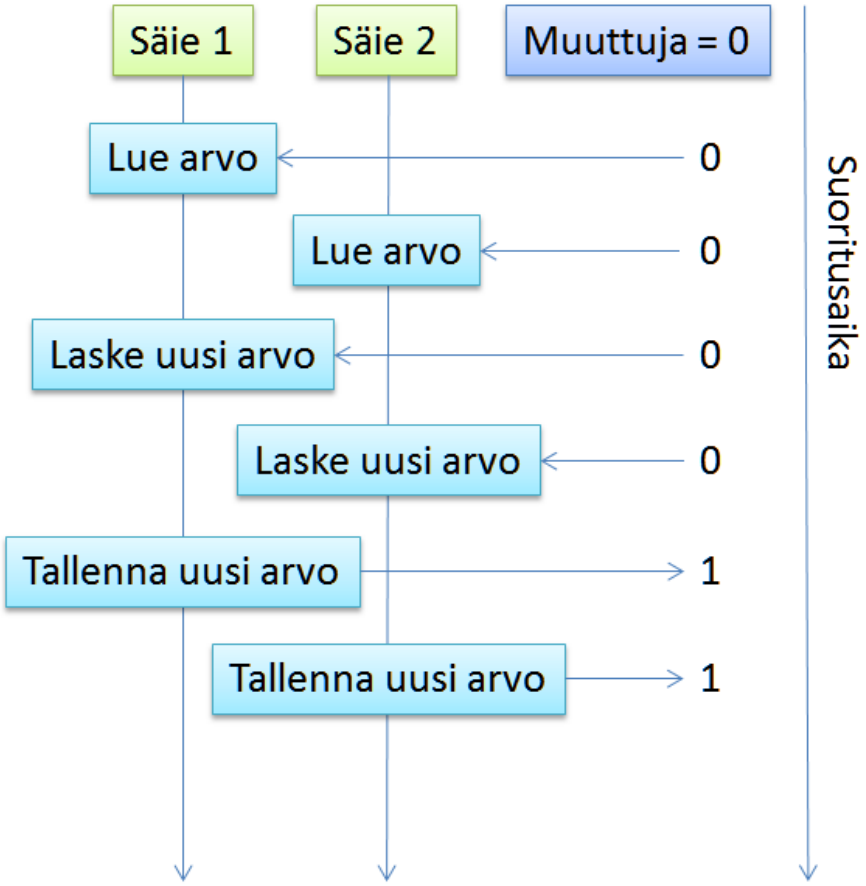
TPL:n käytössä tulee esiin samoja ongelmia kuin klassisessa säikeistyksessäkin. TPL:n luokkien ja metodien toimintaa on syytä tutkia, vaikka niiden toiminnan voi näennäisesti päätellä nimeämisestä tai syntaksista. Ohjelmoijan tulee kiinnittää erityistä huomioita tiedon koordinointiin ja synkronointiin säikeiden välillä, kun ne suorittavat ohjelmaa rinnakkain. Säikeistä johtuvat virheet ovat usein hankalasti paikannettavia tai vaikeasti toistettavia. Tässä kappaleessa esitellään yleisimpiä virheitä ja niiden syitä.

5.2 Synkronointivirheet

Synkronisointivirheet ovat yleisimpiä virhetyyppejä rinnakkaisohjelmoinnissa. Useamman yhtäaikaisesti käynnistyneen säikeen on jossain vaiheessa loputtava ja yhdistyttävä pääsäikeeseen, jotta niiden tuottamia tuloksia voidaan käyttää muualla ohjelmassa. Synkronisoinnilla varmistetaan, että säikeen tuottamaa tietoa ei käytetä, ennen kuin säie on suoritettu loppuun. Synkronisoinnilla estetään myös säikeitä häiritsemästä toisiaan, kun ne käyttävät esimerkiksi samaa muuttujaa tai metodia suorituksen aikana. (Freeman 2012, 59-60.)

5.3 Säikeiden välinen kilpailu

Synkronoinnin puuttuminen aiheuttaa kilpailutilanteen säikeiden välillä. Kaksi säiettä kilpailee keskenään jaetusta resurssista, kuten muuttujasta. Tarkoituksena voi olla esimerkiksi kasvattaa lukua silmukan sisällä. Tällöin koodi näyttää loogiselta, mutta se tuottaa epätoivottuja tuloksia. Synkronoinnin puuttuessa kaksi säiettä voivat lukea ja kirjoittaa muuttujaa itsenäisesti jolloin toisen säikeen laskentatyö ei ole ehtinyt tallentua muuttujaan, ennen kuin toinen säie on lukenut sen omaan muistiinsa. (Freeman 2012, 50-52.)



Kuvio 18. Säikeiden välinen kilpailutilanne (Freeman 2012, 51.)

Molemmat säikeet lukevat alkuarvon 0 ja lisäävät siihen yhden. Säie 1 tallentaa laskutuloksen muuttujaan jonka jälkeen säie 2 tallentaa samaan muuttujaan oman tuloksensa.

Säieturvallisesta luokasta voidaan kutsua sen tarjoamia metodeja ilman, että syntyy synkronointivirheitä. .NET sisältää useita säieturvallisia luokkia. TPL:n yhteydessä on esitelty muun muuassa erilaisten kokoelmien (`System.Collections.Concurrent`) säieturvalliset versiot, kuten `ConcurrentStack` (Freeman 2012, 87-96). Suuriimpaan osaan valmiita luokkia ei ole kuitenkaan ohjelmoitu säieturvallisuutta. Säikeden suorittamien operaatioiden tulee olla ns. atomisia operaatioita, jotka suoritetaan aina kokonaan ennen kuin prosessori antaa ajoaikaa toiselle säikeelle. Säieturvallisuuden puuttuessa ohjelmoijan täytyy estää kahta säiettä käyttämästä samaa koodia yhtäaikaisesti. Helpoin tapa luoda säieturvallisuutta on käyttää lukkoja. Koodia suorittava säie lukitsee muuttujan tai metodin käytön ajaksi, jolloin muut säikeet odottavat lukon avautumista.

Lukkojen käyttö voi aiheuttaa sovelluksen hidastumista, jos usea säie ”nälkiintyy” eli ei saa prosessorilta suoritusaikaa odottaessaan jonkin lukon avautumista. Lukkojen tarpeetonta käyttöä on pyrittävä välttämään. TPL:ssä on klassisessa säikeistysmallissa käytetyn lukon (`Lock`) lisäksi esitelty useita muita hieman käyttötavaltaan poikkeavia kevyempiä synkronointimenetelmiä. Ohjelmoijan tulee pyrkiä käyttämään keveintä synkronointimenetelmää ylimääräisen laskentatyön välttämiseksi. (Freeman 2012, 61.)

Lukkojen käyttäminen itsessään ei välttämättä hidasta ohjelmaa. Alla olevassa taulukossa esitellään .NET 4.0:ssa käytössä olevien lukkojen aiheuttama ylimääräinen laskenta.

Lukko tyyppi	**Käyttötarkoitus**	**Prosessien välinen**	**Ylimääräinen laskenta**
lock (Monitor.Enter /Monitor.Exit)	Varmistaa, että vain yksi säie voi kerrallaan käsitellä resursseja tai suorittaa koodia.	-	20ns
Mutex		Kyllä	1000ns
SemaphoreSlim	Varmistaa, että vain ennalta määrätty määrä säikeitä voi kerrallaan käsitellä resursseja tai suorittaa koodia.	-	200ns
Semaphore		Kyllä	1000ns
ReaderWriterLockSlim	Antaa lukuoikeuden useammalle säikeelle, mutta kirjoitusoikeuden vain yhdelle.	-	40ns
ReaderWriterLock		-	100ns

Taulukko 3. Erilaisia lukkorakenteita .NET 4.0:ssa (Albahari 2011.)

Lukkojen käyttö synkronoimisessa hidastaa ohjelmaa, mikäli useampi säie joutuu odottamaan lukon vapautumista ja syntyy kontekstin vaihdos.

5.4 Lukkiutuminen

Lukkiutumisongelma syntyy yksinkertaisimmillaan, kun vähintään kaksi säiettä odottaa toistensa vapautumista. Kumpikaan säie ei tällöin jatka ohjelman suorittamista ennen kuin toinen säie vapauttaa oman lukkonsa. Säikeet jäävät siis ikuisesti lukkoon. Lukkiutuminen tapahtuu esimerkiksi alla olevassa tilanteessa (koodiesimerkki 16). (Freeman 2012, 45-47.)

```
object lukko1 = new object();
object lukko2 = new object();

new Thread(() =>
{
    lock (lukko1)
    {
        Thread.Sleep(1000);
        lock (lukko2) { };      //Lukkiutuminen
    }
}).Start();
lock(lukko2)
{
    Thread.Sleep(1000);
    lock (lukko1) { };          //Lukkiutuminen
}
```

Koodiesimerkki 16. Säikeiden lukkiutuminen

Uusi säie ottaa lukon (`lukko1`) ja jää suorittamaan tehtävää sekunnin ajaksi, jonka jälkeen säie yrittää saada lukon (`lukko2`) itselleen. Pääsäie on kuitenkin ottanut lukon (`lukko2`) itselleen ja odottaa lukon (`lukko1`) vapautumista. `Lukko1` ei kuitenkaan vapaudu ennen kuin uusi säie on saanut lukon `lukko2` itselleen. Molemmat odottavat lukkojen vapautumista ikuisesti.

5.5 Muut ongelmat

Säieohjelmointiin liittyy paljon muita teknisiä ongelmia, joita ei esitellä tässä opinnäytetyössä. Virheet muodostuvat yleensä siitä lähtökohdasta, että ohjelmoija mieltää koodin sekventiaaliseksi eikä huomioi taustalla olevien säikeiden käyttäytymistä. Ohjelmoijan tulee tutustua kunkin tekniikan käyttötarkoituksiin ja siihen liittyviin tyypillisiin ohjelmointivirheisiin. Tämän opinnäytetyön lähteistä löytyy aiheeseen liittyvää kirjallisuutta, joista saa tietoa niin klassisten `System.Threading`-nimiavaruuden sekä TPL:n tekniikoista.

6 OHJELMOINTITAPOJEN VERTAILU

TPL on monipuolinen ja helposti lähestyttävä ratkaisu klassisen säikeistyksen lisänä. Ohjelmoija voi käyttää .NET-ympäristössä molempia tekniikoita. Aiemmat tekniikat ovat osittain vanhentuneita eikä niiden käyttöä suositella virhealttiuden tai suorituskyvyn vuoksi. TPL tarjoaa muun muassa suorituskykyisempiä synkronointiprimitiivejä.

Tehokkuuden mittaaminen menetelmien välillä voi olla työlästä ellei ohjelmoija tunne niiden toimintatapoja. Ohjelmoija helposti pitäytyy niissä menetelmissä, jotka hän tuntee parhaiten. Eri menetelmien kokeileminen voi vaatia koodin uudelleen ohjelmoimisen menetelmälle sopivaksi. Klassinen säikeistysmalli vaatii paljon aihepiirin tuntemusta. Koodista tulee helppolukuisempaa, kun eri menetelmät ovat ikään kuin koottu pienistä osista ja ovat näkyvillä koodissa. TPL:n `Parallel`-luokasta kutsut metodit saattavat ajaa saman asian kuin perinteisillä menetelmillä tehty ratkaisu, mutta toiminnallisuus ei välttämättä ole helposti pääteltävissä koodista.

TPL helpottaa säikeiden koordinointia koodista. Perinteisen säikeen suoriutumisen loppumista voi koodista odottaa esimerkiksi staattisella metodilla `Thread.Join()`. Tämä kuitenkin estää UI säikeen toiminnan. Ongelman välttämiseksi voidaan luoda `WaitHandle`-olio ja antaa se parametriksi luotavalle säikeelle, jonka avulla saadaan säikeen odottaminen asynkroniseksi UI-säikeen kanssa. Ongelmaan ei klassisessa säikeistysmallissa ole yhtä hyväksi havaittua keinoa vaan se usein toteutetaan hieman eri tavoin. TPL:n `Task`-luokka sen sijaan tarjoaa useita staattisia metodeja ongelmaan. Esimerkiksi `Task.WaitAll(Task[])` osaa hoitaa edellä mainitun ongelman ohjelmoijalle erittäin ymmärrettävässä muodossa. Metodi ottaa vastaan joukon `Task`-olioita, joita tulee odottaa ennen kuin pääsäiettä jatketaan.

Toinen säikeiden koordinointiin liittyvä asia on säikeiden keskeyttäminen. Usea säie voi esimerkiksi käsitellä saman tietokannan eri osia haettavan asian löytämiseksi. Kun yksi säie on löytänyt etsityn tiedon, muiden säikeiden ei enää tarvitse jatkaa etsintöjä vaan voivat vapauttaa käyttämänsä

laskentaresurssit. Klassisessa mallissa tämä toteutetaan käsin. Säikeisiin on luotava toiminnallisuus, joka osaa lopettaa toiminnan ja vapauttaa resurssit. TPL tarjoaa tähän ongelmaan `CancellationToken`-luokan, jonka ilmentymän voi liittää yhteen tai useampaan säikeeseen. Luokan ilmentymän `.Cancel()`-metodia kutsuessa säikeet saavat tiedon peruuttamisesta (Freeman 2012, 15-18).

TPL helpottaa erityisesti tiedonkäsittelyä, jossa säikeiden ei tarvitse jakaa tietoa keskenään. Ilman TPL:n kaltaista rajapintaa säikeiden määrittely ja luominen monimutkaistaa koodia tällaisissa tilanteissa.

7 POHDINTA

Rinnakkaisohjelmointi ja klassinen säikeistys ovat pohjimmiltaan tehokkuuden lisäämistä ohjelmaan. Ohjelman suorittama työ paloitellaan osiin, ja näitä paloja voidaan suorittaa rinnakkain. Ohjelmoinnin ydin on kuitenkin tuottaa haluttuja toiminnallisuuksia ohjelmalle ja TPL:n kaltainen rajapinta ei helpota itse toiminnallisuuden tekemistä. Toiminnallisuus saattaa kuitenkin sisältää raskaita tietokantalukuja, tiedon indeksointia tai muuta suurta laskentatehoa vaativia tehtäviä. Näitä tehtäviä on niiden luonteen rajoissa hyvä muuttaa suoritettavaksi useammalla ytimellä yhtäaikaisesti, jotta saadaan käyttöön kaikki tietokoneesta löytyvä laskentateho.

Sekventiaalinen ohjelmointi on läsnä myös missä tahansa muussa säikeistetyssä ohjelmoinnissa. Toiminnallisuus koostuu sarjasta peräkkäisiä komentoja joita ei voida eikä ole edes järkevää suorittaa useammassa kuin yhdessä säikeessä.

Ohjelmoidessa rinnakkaisuutta tulee ohjelmoijan ottaa huomioon tiedon koordinointi, kun sovellus käyttää yhteisiä resursseja, josta tietoa haetaan. Ohjelmoijan vastuulle jää lisätä sovellukseen oikea määrä koordinointia, jotta sen tuottama ylimääräinen tiedonkäsittely ei tule liian raskaaksi tai jopa hidasta ohjelmaa sekventiaaliseen toteutustapaan verrattuna. Liian vähäinen koordinointi taas puolestaan voi johtaa hankalasti paikannettaviin virheisiin ohjelmassa.

TPL:n kaltaisen rajapinnan käyttö rinnakkaisuuden ohjelmoinnissa on helppo tapa lisätä tehokkuutta omiin sovelluksiinsa. Klassinen säikeistysmalli vaatii aihepiiriin perehtymistä huomattavasti enemmän, mutta ohjelmoijan on tunnettava säikeistyksen periaatteet ymmärtääkseen, kuinka TPL:n kaltainen rajapinta toimii. Rajapinnan ja säikeiden ymmärtäminen auttaa ohjelmoijaa tiedon koordinoimisessa rinnakkaistoteutuksissa.

Tulevaisuuden trendi on prosessoriytimien lisääminen sekä muu hajautettu tietojenkäsittely. Vaativat laskentatehtävät pyritään pilkkomaan ja jakamaan eri ytimille tai tietokoneille. Hajautettu tietojenkäsittelyä hyödyntävät muun

muuassa SETI@home-projekti (University of California, 2014), joka hajauttaa avaruudesta tulevan radiosäteilyn käsittelyn aiheesta kiinnostuneiden tietokoneille. Samaa ideaa toteuttaa myös Folding@home-projekti (Stanford University, 2014), joka hajauttaa proteiinien molekyylidynamiikkaan vaadittavaa laskentaa.

LÄHTEET

Moore, G. 1965. Intel-yhtiön historiaa. Osoitteessa: http://www.intel.com/content/www/us/en/history/museum-gordon-moore-law.html.

Carr, C. 2011. Luentosarja rinnakkaisohjelmoinnista .NET-ympäristössä. Osoitteessa: http://www.blackwasp.co.uk/ParallelProgramming.aspx. 8.8.2011.

Denis, C. 2010. Artikkeli Concurrency Visualizerin sovelluslaajennuksen käytöstä. Osoitteessa: http://blogs.lessthandot.com/index.php/Architect/EnterpriseArchitecture/visual-studio-2010-concurrency-profiling/. 14.3.2010.

Albahari, J. 2011. E-kirja ”Threading in C#”. Osoitteessa: http://www.albahari.com/threading/. 27.4.2011.

Freeman, A. 2010. Pro .NET Parallel Programming in C#. New York: Apress.

Campbell, C. – Johnson, R. – Miller, A. – Toub, S. 2010. Parallel Programming with Microsoft .NET: Design Patterns for Decomposition and Coordination on Multicore Architectures. Redmond: Microsoft Press.

Microsoft Developer Network 2014a. Visual C#-ohjelmointikielen kuvaus Osoitteessa: http://msdn.microsoft.com/en-us/library/kx37x362.aspx.

– 2014b. Oppimateriaalia Task Parallel Library:sta. Osoitteessa: http://msdn.microsoft.com/en-us/library/dd460717(v=vs.110).aspx.

– 2014c. Kuvaus .NET-ympäristöstä. Osoitteessa: http://msdn.microsoft.com/en-us/library/hh425099(v=vs.110).aspx.

– 2014d. Parallel Programming in the .NET Framework Osoitteessa: http://msdn.microsoft.com/en-us/library/dd460693(v=vs.110).aspx.

– 2014e. Task Parallelism (Task Parallel Library). Osoitteessa: http://msdn.microsoft.com/en-us/library/dd537609(v=vs.110).aspx.

– 2014f. Pipelines-suunnittelumalli. Osoitteessa: http://msdn.microsoft.com/en-us/library/ff963548.aspx

– 2014g. Parallel Loops. Osoitteessa: http://msdn.microsoft.com/en-us/library/ff963552.aspx

– 2014h. Parallel Tasks-suunnittelumalli. Osoitteessa: http://msdn.microsoft.com/en-us/library/ff963549.aspx

– 2014i. Parallel Aggregation-suunnittelumalli. Osoitteessa: http://msdn.microsoft.com/en-us/library/ff963547.aspx

– 2014j. Futures-suunnittelumalli. Osoitteessa: http://msdn.microsoft.com/en-us/library/ff963556.aspx

– 2014k. Dynamic Task Parallelism-suunnittelumalli. Osoitteessa: http://msdn.microsoft.com/en-us/library/ff963551.aspx

Hilfinger, P. 2006. Data Structures and Advanced Programming, kappale 7.2.11 Osoitteessa: https://inst.eecs.berkeley.edu/~cs61b/sp06/labs/s7-2-11

University of California 2014. SETI@home-projektin kotisivut. Osoitteessa: http://setiathome.berkeley.edu/.

Stanford University 2014. Folding@home-projektin kotisivut. Osoitteessa: http://folding.stanford.edu/home.

Columbia University 2012. Advanced Microarchitectures luentosarja. Osoitteessa: http://www.cs.columbia.edu/~sedwards/classes/2012/3827-spring/advanced-arch-2011.pdf.