**PE-taak: performance onderzoek**



Groepsleden:

* Francesco Ventura
* Muhammed Genç
* Bunyamin Eren
* Kaan Koseoglu

Inhoud

[1. Inleiding 3](#_Toc466586497)

[1.1 Wat is performance onderzoek ? 3](#_Toc466586498)

[1.2 Doel 3](#_Toc466586499)

[1.3 Referentie project 3](#_Toc466586500)

[2. Performance Analysis 4](#_Toc466586501)

[2.1 Metrieke 4](#_Toc466586502)

[2.2 Tools 4](#_Toc466586503)

[3. Oplossingsmethode Sudoku 5](#_Toc466586504)

[3.1 Referentie Algoritme 5](#_Toc466586505)

[3.2 Performance Optimalisatie 5](#_Toc466586506)

[3.2.1 Multi-Threaded 5](#_Toc466586507)

[3.2.2 Final keyword 8](#_Toc466586508)

[4. Persoonlijke Reflectie 9](#_Toc466586509)

[4.1.1 Francesco Ventura 9](#_Toc466586510)

[4.1.2 Bunyamin Eren 9](#_Toc466586511)

[4.1.3 Kaan Koseoglu 10](#_Toc466586512)

[4.1.4 Muhammed Genc 10](#_Toc466586513)

# Inleiding

## Wat is performance onderzoek ?

De letterlijke betekenis van *performance*: ‘het vermogen om veel en/of goed te presteren, prestatie’.

Een performance onderzoek is dus een onderzoek naar het prestatie van het programma. Aan de hand van een performance onderzoek weten de ontwikkelaars wat veel tijd in beslag neemt en/of er nog eventueel wijzigingen kunnen doorgevoerd worden om het programma sneller, beter te laten werken.

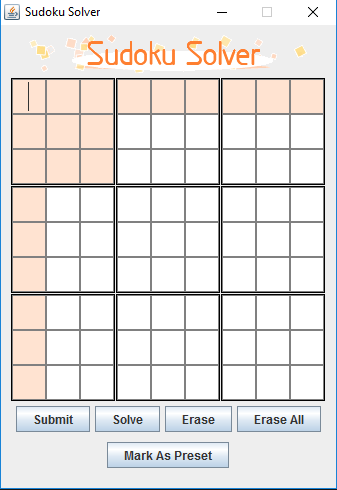
Immers is het zo dat een performance onderzoek een programma ontleed en gedetailleerd informatie weergeeft over het geheugengebruik, uitvoeringstijd,...

## Doel

Het doel van dit document is het analyseren en verbeteren van source code van een sudoku solver. Hierbij gebruiken we een set van leerrijke tools aangeleerd tijdens de les maar ook een deel van tools die zelfstudie van ons vereist.

Dit document zal gedetailleerd de performance oplossing omschrijven die wij toepassen op het project van de sudoku solver. Grafieken zullen de prestaties vergelijken van onze verbeteringen ten opzichte van de originele code. Wij hebben gekozen om Performance analysis toe te passen in de programmeertaal Java.

## Referentie project

Als referentie hebben we gekozen voor een open source GitHub project die terug te vinden is op deze link: https://github.com/miga9/sudoku-solver-java. Het project is volledig gedefinieerd in een class file met een main functie daarin. Hierbij heeft de auteur gekozen om niet in een *Object Oriented* manier te werken. De code bevat wel gedetailleerd omschreven commentaar waardoor het analyseren van de code een stuk gemakkelijker gaat verlopen.

Als *Framework*\* om de GUI aan te maken wordt er gebruik gemaakt van Java Swing. Verder nog maakt de auteur niet gebruik van Java 8 features (bv lambda’s).

# Performance Analysis

## Metrieke

Als meting voor het bepalen dat een stuk code verbeterd is maken we gebruik van enkele eigenschappen.

* Heap Size
* Snelheid (uitgedrukt in milliseconden)
* CPU cores utilisatie

De verbeterde snelheid wordt weergegeven als een percentage omdat de snelheid voor een groot deel afhankelijk is van de prestatie van uw processor. Hierbij wordt ook aangeduid wat de originele source code was en wat wij gewijzigd hebben om de prestatie te verbeteren.

## Tools

**JConsole**

Komt standaard meegeleverd met de JDK, installatie is niet vereist. JConsole is uitgerust met een dashboard die doormiddel van grafieken de volgende eigenschappen meegeeft:

* Een weergaven over de total heap size ruimte dat er in beslag wordt genomen in het RAM-geheugen.
* Toont het aantal threads die het programma in beslag neemt.
* Een overzicht over hoeveel procent van de capaciteit van de in processor in beslag wordt genomen.

**YourKit**

YourKit biedt ons de mogelijk om te kijken hoe lang het duurt tot de functie gedaan heeft met uitvoeren. Dit is handig om te kijken precies in welke functie versnellingen kunnen worden toegepast. We zullen gebruik maken van de trial version van YourKit samen met IntelliJ.

**JProfiler**

JProfiler toont meer detail over de garbage collection die moet gebeuren. Hierbij toont JProfiler zelfs wat voor soort objecten zich in de Heap bevinden. Dit kan handig zijn om referenties naar een klasse te leggen in plaats van een nieuwe instantie aan te maken. Door middel van het nemen van een snapshot is het ook mogelijk om een algemeen overzicht te krijgen van de Heap.

# Oplossingsmethode Sudoku

## Referentie Algoritme

Het programma maakt gebruik van het backtracking algoritme waarbij het de bedoeling is alle mogelijke combinaties van 1 tot 9 in te vullen. Deze combinatie wordt alleen toegepast op de lege cellen die het programma kan vinden. Het programma gaat proberen alle lege cellen in te vullen tot er geen meer zijn en de sudoku is opgelost. Het programma gaat door middel van een recursieve call elke 3 op 3 box nakijken tot de rij gelijk is aan 9 (rechts onder) om uit de recursieve call te geraken. Hierbij maakt het programma niet gebruik van threads.

Meer info:

<http://www.math.cornell.edu/~mec/Summer2009/meerkamp/Site/Solving_any_Sudoku_II.html>

## Performance Optimalisatie

### Multi-Threaded

De applicatie maakt niet gebruik van threads om het algoritme van de sudoku op te lossen of te versnellen. Hierdoor lijkt het rendabel om threads bij te voegen die ervoor moeten zorgen dat de cores op een computer processor evenveel gebruikt worden.

Nadat een sudoku is opgelost kijkt het programma of alle 3x3 boxen opgevuld zijn met cijfers. Dit gebeurt per box in het raster van de sudoku. We kunnen deze operatie opsplitsen over een aantal threads die hierna samen komen en allemaal true moeten returnen. Hieronder een schema over het huidig gebruik van de threads.

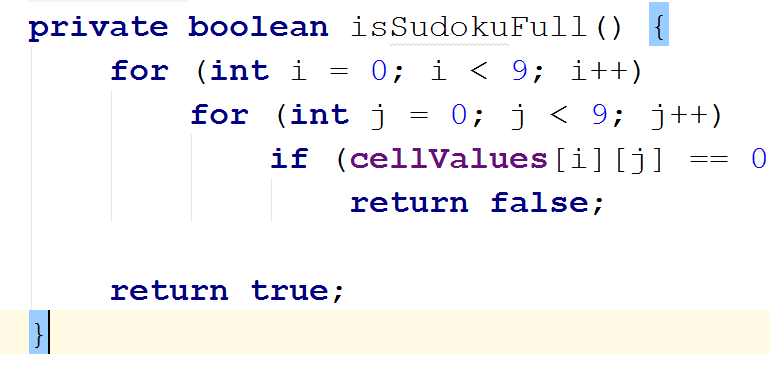


Dit is de thread usage die aantoont dat bij het oplossen van de sudoku de garbage collector intreedt en de huidige main thread blokkeert. Bij intensief gebruik kan het ervoor zorgen dat de GUI van het programma blijft hangen. De rode streepjes duiden aan dat de garbage collector de heap heeft opgeruimd. Ongeachte de verschillende keren dat een sudoku opgelost geraakt maakt de garbage collector de heap leeg na elke opgeloste sudoku.

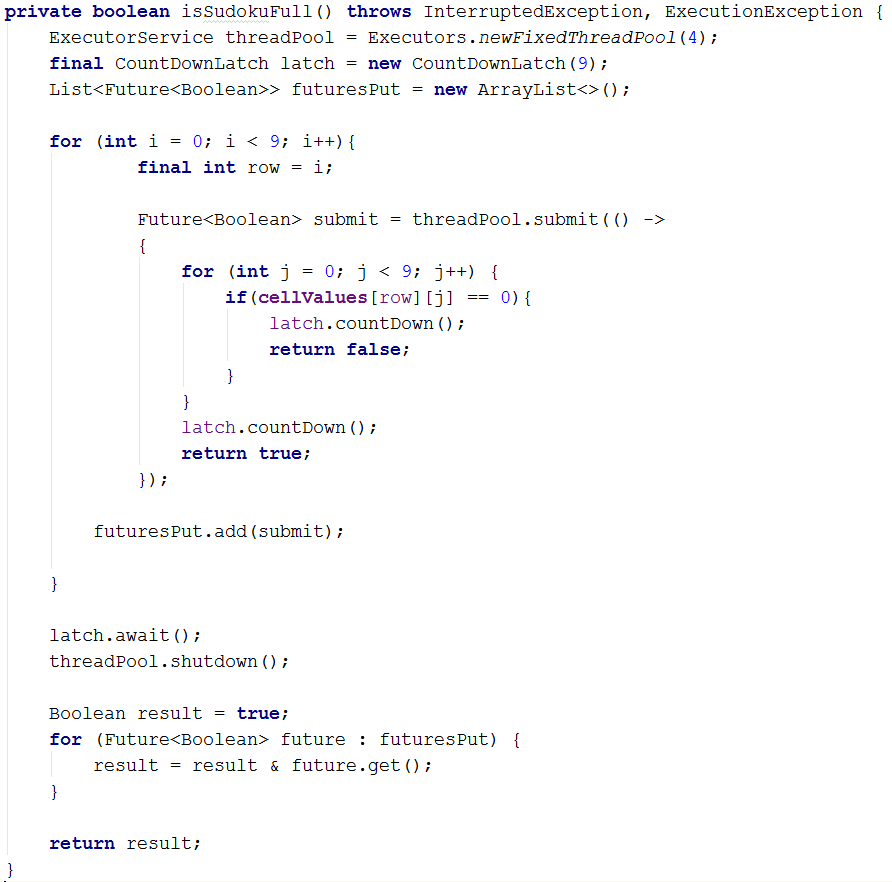
We zullen nu dat probleem verhelpen door gebruik te maken van een threadpool die threads alloceert. Zie hieronder de code die wij hebben verbeterd.

**Voorbeeld oplossing**

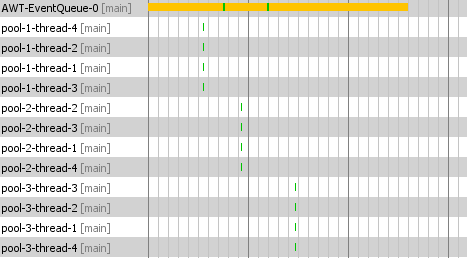
*Originele code*



*Onze verbetering*



Als we een overzicht maken van het gebruik over de threads dan ziet dat er zo uit.

**

**

Geen threads die worden geblokkeerd, wat goed rendement is voor een programma die altijd moet reageren op acties ondernomen door de gebruiker. Maar wat levert dit in verband met performance? We hebben JProfiler gebruikt om de snelheid te meten van de functie die we verbeterd hebben. De tijd die nodig om een thread op te starten is rond de 1000 µs. Hierbij tonen wij onderaan met een grafiek hoeveel impact dit heeft ten opzichte van de originele oplossing.

*Intel i5 4690K Devil Canyon OC @ 4.7 GHZ – 4 Cores*

De functie wordt in de huidige code 6 keer aangeroepen waarbij de schaalbaarheid te klein is en het verschil enorm uitloopt. Trekken wij de for lus verder uit naar 10000 keer dan is er wel een performance winst

**Conclusie**

De code belast de processor minimaal waardoor de executie van de code te snel verloopt om te kunnen verholpen worden door meerdere threads.

### Final keyword

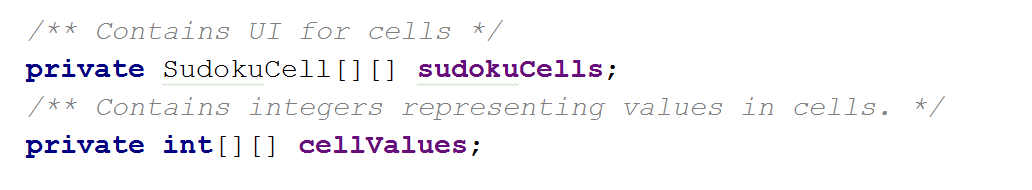
Het final keyword definieert een constante binnen de Java syntax. Dit zorgt voor een snellere compileertijd. Performance winst door gebruik te maken van final is minimaal op zijn best maar kan een verschil maken binnen het gebruik van een Boolean.

Zie deze stackoverflow post voor meer informatie: <http://stackoverflow.com/questions/4279420/does-use-of-final-keyword-in-java-improve-the-performance>

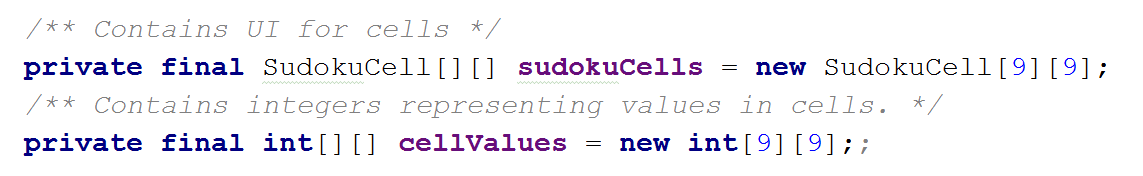
Als voorbeeld nemen we de sudoku cells die in de broncode gedefinieerd zijn als een tweedimensionale array van Integers. Door deze als final te definiëren versnellen we het compileren. We kunnen ook potentiele memory leaks vermijden omdat er geen referenties kunnen zijn naar de variabele.

**Voorbeeld oplossing**

*Originele code*

**

*Onze verbetering*

**

***Conclusie***

Verschil is bijna onmeetbaar aangezien dankzij de JIT compiler de functie verbeterd wordt per iteratie. Daarom is deze test genomen na 25x keer uitvoeren waaruit blijkt dat met de final statement toch enkele µs winst is.

# Persoonlijke Reflectie

### Francesco Ventura

In het begin had ik moeite om mezelf te integreren met de nieuwe tools omdat de informatie die de tools gaven niet altijd duidelijk was wat ermee werd bedoeld. Hiermee had ik verschillende videos op pluralsight bekeken om een duidelijker beeld te krijgen naar wat ik effectief moest kijken.

Java code is moeilijk om te benchmarken aangezien na elke iteratie van de code de JIT compiler aanpassingen doet aan de bytecode om optimalisaties te doen waardoor de tijd gemeten na 1 keer uitvoeren totaal anders is dan 100 keer. Daarom zijn veel van de testen bovenaan meerdere keren gedaan zodat ik een resultaat kan weergeven die binnen de “foutmarge” zit.

Ik had liever dit project uitgevoerd binnen een taal die meer controle gaf over het alloceren van Objecten op de heap. Talen zoals C of C++ laten zelf toe om native x86 ASM te schrijven waardoor prestatie echt vooruit kan schieten terwijl je bij Java amper een “struct” kunt schrijven.

Veel concepten waarvan ik wist dat die winst gingen ophalen in verband met prestatie kon ik niet toepassen op die project. Zoals bv. :

* List sorteren
* Meerdere threads
* IsEmpty() oproepen inplaats van == 0
* …

Het project was opzicht niet complex genoeg en was zo goed geschreven dat de auteur zelf gebruik maakte van binary operaties.

### Bunyamin Eren

In het begin leek het alsof dit programma maar een simpele was, dat geen gebruik maakte van OO omdat het één enkele java file was. Daarom gingen we er natuurlijk van uit dat als we gebruik zouden maken van threads en enkele variabelen zouden verbeteren, dit veel sneller zou verlopen.

We stelden echter vast dat hoewel alles in een enkel bestand zit, het toch goed geschreven is. Het multithreaden van deze applicatie bood geen verbeteringen, het was zelfs veel trager dan de originele versie.

Dit komt omdat de bewerkingen die worden uitgevoerd veel te klein zijn en niet genoeg resources gebruiken. Het maken van de threadpool en het opstarten ervan neemt veel meer resources in beslag, waardoor de applicatie na deze “verbetering” veel trager is.

We dachten ook dat we het programma overzichtelijker konden maken door het OO te maken of lambda’s te gebruiken. Hierdoor werkte het niet meer en het heeft uiteindelijk ook niet veel effect op de snelheid ervan.

Ik heb ook enkele concepten bijgeleerd/verbeterd:

* Lambda’s
* Threading

Van de leerstof die we in de les hebben gezien konden we niet alles toepassen om de performantie te verbeteren. Daarom hebben we geen gebruik gemaakt van lambda’s, delegates of logging.

Zelf heb ik in de praktische uitvoering van de PE niet heel veel gedaan, maar was wel altijd up to date met de vorderingen en hielp mee met het schrijven van het verslag. Het was een leuke uitdaging en een goede manier om zelfstandig bij te leren.

### Kaan Koseoglu

Toen we de opdracht hebben gehoord, waren we aan het twijfelen of we een java applicatie gingen zoeken of een .net. Uiteindelijk hebben we toch voor Java gekozen. Eerder had ik nog nooit een applicatie ‘ontleed’. Daardoor vond ik deze PE toch wel interessant.

Het ontleed-proces ging vrij goed, we hebben uiteindelijk alle aspecten kunnen testen. Maar niet alle aspecten zijn gebruikt geweest aangezien het niet altijd een verbetering opleverde.

Multithreading is niet van toepassing geweest. Aangezien het niet veel opleverde hebben we dat niet gebruikt. In logging zijn we ook niet dieper ingegaan in het project dan de lessen.

De final statement toevoegen heeft ons een kleine verbetering opgeleverd.

Ik zie een applicatie als een ijsberg in het water. Wat de users zien, ervaren is het topje. Maar wij hebben met dit project onder de waterlijn gedoken en het programma ‘op een andere manier bekeken’. Het was noch gemakkelijk, noch moeilijk als het coderen ervan. Maar dit soort opdrachten helpen ons om later eventueel beter code review te doen, om code van anderen op een andere perspectief te bekijken. Meestal is het zo dat als je aan het coderen bent dat je in één perspectief een programma zit te bekijken waardoor je misschien niet beseft dat je applicatie sneller kan werken als je andere programmeerstijlen gebruikt.

Over het algemeen vond ik de PE niet heel moeilijk. Het viel best wel mee. Ik vond dat de leerstof parallel ging met de PE waardoor we constant geïnspireerd werden om verschillende tools te gebruiken voor verschillende modules.

### Muhammed Genc

Het was een zeer interessante opdracht, omdat we niet zelf een programma moesten coderen. Maar anders om te werk moesten gaan en het programma ontleden en verbeteren.

We dachten dat door gebruik te maken van threads en door enkele verbeteringen een groot verschil zou zijn. Maar we stelden vast dat het geen verbetering bood voor de applicatie. Dit kwam omdat het niet veel resources gebruikte.

De opdracht zelf viel mee, maar er was wel veel opzoek werk naar mogelijke verbeteringen.  
Het heeft ons bijgeleerd dat je toch je applicatie met tools eens moet nakijken en dat er altijd verbeteringen zijn.