2016•2017
FACULTEIT INDUSTRIËLE INGENIEURSWETENSCHAPPEN
master in de industriële wetenschappen: elektronica-ICT

Masterproef

Exploratie van functionele en declaratieve ontwikkelmethodes voor cloud programming

Promotor :
dr. Kris AERTS
ing. Leo RUTTEN

Copromotor :
De heer Thomas MACHIELS

Stijn Schildermans

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

Gezamenlijke opleiding Universiteit Hasselt en KU Leuven

2016•2017

Faculteit Industriële ingenieurswetenschappen

master in de industriële wetenschappen: elektronica-ICT

Masterproef

Exploratie van functionele en declaratieve ontwikkelmethodes voor cloud programming

Promotor :
dr. Kris AERTS
ing. Leo RUTTEN

Copromotor :
De heer Thomas MACHIELS

Stijn Schildermans

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

universiteit hasselt | KU LEUVEN

Woord vooraf

Deze thesis is de bekroning van mijn opleiding tot industrieel ingenieur elektronica-ICT. Toen ik startte met de algemene opleiding tot industrieel ingenieur was ik echter nooit van plan om deze afstudeerrichting te volgen. Mijn originele plan was immers om bouwkunde als specialisatie te kiezen en in de voetsporen van mijn vader te treden als zelfstandige in de bouwsector. Toen ik echter in het kader van deze opleiding kennismaakte met programmeren, werd ik onmiddellijk gebeten door een sterke passie hiervoor die mij tot nu toe nog niet heeft losgelaten. Op basis van deze pas gevonden passie heb ik besloten mijn ambities om te gooien en ben ik mij beginnen te verdiepen in de fascinerende wereld van elektronica en vooral ICT. Tot op de dag van vandaag heb ik absoluut geen spijt van deze beslissing. Ik heb bovendien in de loop van mijn masteropleiding met dank het aanbod van mijn faculteit aanvaard om na deze opleiding mijn verdieping in programmeren verder te zetten in de vorm van een doctoraat. Bij het schrijven van deze thesis heb ik dan ook het onderste uit de kan gehaald om enerzijds mijn geschiktheid voor deze titel te bewijzen, en anderzijds mijn interessegebied zo breed mogelijk te verkennen.

Graag zou ik een aantal mensen willen bedanken die mij steunden en/of stuurden tijdens het lange en leerrijke proces dat het opstellen van deze thesis was. In de eerste plaats zou ik graag mijn promotoren, Dr. Kris Aerts en Ing. Leo Rutten willen bedanken, alsook mijn copromotor Ing. Thomas Machiels. Dit omwille van het goede advies en de uitstekende feedback die ze mij steeds verschaften gedurende heel het verloop van deze masterproef. Ook zou ik hen in het bijzonder willen bedanken voor de grote vrijheid die ze mij hebben toevertrouwd om zelf het traject van deze masterproef grotendeels uit te stippelen en deze thesis in te vullen naar mijn eigen goeddunken, zodat ik mijn interesses ten volle kon verkennen tijdens dit onderzoek, en tegelijk mijn kennis sterk kon uitbreiden.

Naast mijn promotors en copromotor zou ik ook mijn medestudent Jef Engelen willen bedanken. Origineel was ik dit onderzoek samen met hem gestart. Hij heeft dan ook significante bijdragen geleverd aan het onderzoeksopzet en de literatuurstudie. Omdat onze ambities echter te ver uiteen liepen hebben we halverwege dit onderzoek op aanraden van onze promotoren besloten elk onze eigen weg te gaan.

Verder zou ik graag mijn ouders Georges Schildermans en Godelieve Billiau bedanken, die mij steeds steunden gedurende mijn studie, ondanks het feit dat ik een radicaal andere weg gekozen heb dan degene die zij voor mij in gedachten hadden. Mijn vader zou ik ook graag bijkomend willen bedanken om mijn volledige thesis na te lezen, en een groot aantal typfouten te helpen verbeteren.

Ten slotte zou ik graag mijn grootouders bedanken voor hun steun en luisterend oor doorheen heel mijn studie. Bij hen kon ik steeds terecht met problemen die ik met niemand anders kon delen.

Inhoudsopgave

Lijst van tabellen

Lijst van figuren

Verklarende woordenlijst

Afhankelijkheid	Externe library die gebruikt wordt in een software-project. Het project is dus afhankelijk van deze library en kan niet gebuild worden zonder dat deze library geïmporteerd is in het project.
API	**Application programming interface:** Een service die in de eerste plaats als interface dient tussen een verschillende applicaties en kan aangesproken worden om allerlei data van een bepaalde applicatie op te vragen.
Array	Eenvoudige lijst in veel programmeertalen.
Boolean	Een boolse waarde in computerprogramma's. De waarde kan enkel true of false zijn, 'waar' of 'onwaar'.
Builden	Proces in softwareontwikkeling waarbij de codebestanden worden omgevormd tot een distribueerbaar pakket, klaar om gedeployed te worden.
Cloud consumer	Gebruiker van cloud-services.
Compiled taal	Een programmeertaal die voor de uitvoering is gecompileerd tot machinetaal, ofwel een tussenliggend stadium dat kan geïnterpreteerd worden door een runtime-omgeving.
Compiler	Programma dat code in een bepaalde programmeertaal omzet naar binaire instructies of een tussenliggend stadium dat door de runtime-omgeving kan geïnterpreteerd worden. Het compileren gebeurt dus voor de uitvoering van het programma.
Compiletime	Het moment waarop een programma gecompileerd wordt door een compiler; bij de compilatie van het programma.
CPU	**Central processing unit:** Computerprocessor.
Cross-platform	Een applicatie of framework die/dat op meerdere verschillende platformen gebruikt kan worden. Dit heeft meestal betrekking op besturingssystemen.
CSP	**Cloud service provider**: Persoon of meestal organisatie die één of meerdere cloud services publiek aanbiedt.
Data member	Veld van een klasse in Java. Een data member of veld is in feite een object dat eigen is aan de klasse of objecten van die klasse. Het bevat informatie die de klasse of het object beschrijft, en meestal de toestand van de klasse of het object bijhoudt.
Debugger	Programma of meestal deel van een IDE dat toelaat om op een gestructureerde manier naar fouten in het programma te zoeken.
Deployen	Installeren van een software-service zodat deze toegankelijk is voor de gebruikers.

Double	Double-precision floating point getal. Wordt gebruikt om kommagetallen en zeer grote of kleine waarden weer te geven in computerprogramma's.
Dynamic typing	Typesysteem waarbij de types van variabelen, parameters, etc. in het programma pas bij runtime geweten zijn.
Eager evaluation	Systeem waarbij de waarde van een variabele rechtstreeks wordt berekend op het moment dat de variabele toegekend wordt.
Enum	Speciaal type in sommige programmeertalen waarmee beperkte keuzemogelijkheden gedefinieerd kunnen worden.
Event listener	Stuk code dat wordt aangeroepen wanneer een ander stuk code een bepaalde 'event' aangeeft. Deze code wordt dus uitgevoerd wanneer andere code een specifieke situatie tegenkomt.
For-lus	Controlestructuur die in veel programmeertalen wordt gebruikt om code meermaals achter elkaar uit te voeren.
Framework	Universele applicatie-specifieke basis waarop software kan gebouwd worden volgens een bepaald vooropgesteld schema.
FTP	**File Transfer Protocol**: Protocol om eenvoudig bestanden uit te wisselen tussen verschillende computers die verbonden zijn via een netwerk.
Functioneel programmeren	Programmeerparadigma waarbij een programma wordt opgebouwd aan de hand van een groot aantal toestandsloze functies, die elk stapsgewijs de input transformeren naar de output.
Gedistribueerd systeem	Computersysteem waarbij de uitvoering van code gespreid is over meerdere machines, die meestal verbonden zijn in een netwerk.
Garbage collector	Deel van de runtime-omgeving van veel programmeertalen dat verantwoordelijk is voor het vrijgeven van geheugen dat wordt bezet door delen van het programma die niet meer nodig zijn; nadat de uitvoering van hun code afgelopen is.
Http	**Hyper text transfer protocol**: Protocol voor het verzenden van data via het internet.
IDE	**Integrated Development Environment:** Programma voor het ontwikkelen van software.
If-lus	Controlestructuur die in veel programmeertalen wordt gebruikt om een keuze te maken tussen verschillende mogelijke paden in de executie van een programma, op basis van een boolse expressie.
Import	Statement bovenaan een codebestand waarmee wordt aangegeven dat extra libraries worden gebruikt in de code. De

	import statement importeert de libraryfuncties als het ware in het codebestand, waardoor niet steeds de volledige package of namespacedeclaratie moet getypt worden wanneer de library-functies worden gebruikt.
Int	**Integer:** Geheel getal in computerwetenschappen.
Internet of things (IoT)	Concept waarbij vele alledaagse objecten verbonden zijn met het internet.
Interpreted taal	Programmeertaal die niet op voorhand wordt gecompileerd, maar op runtime door een interpreter rechtstreeks wordt omgezet in machinecode.
Interpreter	Programma dat op runtime code van een programma rechtstreeks omzet in machinecode.
JSON	**Javascript object notation**: Universele notatie van data die vaak wordt gebruikt in de REST architectuur.
Keyword	Woord dat deel uitmaakt van de syntax van een bepaalde programmeertaal.
Lambda-expressie	Functioneel concept binnen object-georiënteerde programmeertalen waarbij een anonieme fyunctie kan gedeclareerd en gebruikt worden binnen de OO context.
Latency	Vaste reactietijd bij het aanroepen van een service/functie, onafhankelijk van de grootte van de gevraagde berekening.
Lazy evaluation	Systeem waarbij de waarde van een variabele pas wordt berekend op het moment dat de variabele gebruikt wordt.
Library	Pakket van klassen/functies dat een bepaalde functionaliteit biedt dat kan worden geïmporteerd in andere softwareprojecten.
Microservices	Vorm van SOA waarbij de verschillende services onafhankelijk van elkaar bestaan. Microservices hebben elk typisch een zeer kleine verantwoordelijkheid binnen het programma.
Namespace	Pakket waarin softwarefuncties/klassen met een gemeenschappelijke functionaliteit of die deel uitmaken van een gemeenschappelijke applicatie worden gegroepeerd. Deze terminologie wordt vooral in .NET gebruikt.
Neveneffect	Fenomeen waarbij het uitvoeren van code de toestand van het programma verandert. Dit kan invloed hebben op de uitvoering van de andere code in het programma.
NoSQL	**Not only SQL**: Alternatieve database-structuren, die niet of slechts gedeeltelijk van SQL gebruik maken.
Object-georiënteerd (OO) programmeren	Programmeerparadigma waarbij programma's worden voorgesteld door de interactie tussen verschillende objecten.

Package	Pakket waarin softwarefuncties/klassen met een gemeenschappelijke functionaliteit of die deel uitmaken van een gemeenschappelijke applicatie worden gegroepeerd.
Parallellisatie	De uitvoering van een programma verdelen onder meerdere threads die tegelijk uitgevoerd worden, waardoor de uitvoering van het programma versneld wordt. Deze terminologie wordt vooral in Java gebruikt.
Plugin	Software-component die een specifieke functionaliteit toevoegt aan een programma.
Pushen	Uploaden van code naar een externe opslagplaats.
Request	Aanvraag die binnenkomt bij een server.
Requestparameter	Parameter die met een http-request wordt meegegeven via de URL, die de server voorziet van noodzakelijke informatie om de request af te handelen.
Resources	In computerwetenschappen is dit een omvattende term voor alle hardware die beschikbaar is in een bepaalde context. Dit kan processorkracht, fysiek geheugen, opslagruimte, netwerk, etc. zijn.
Response	Antwoord dat een server stuurt op een request.
REST	**REpresentational State Transfer**: Architectuur die beschrijft dat applicaties als stateless services kunnen opgebouwd worden en data uitwisselen via het internet.
Return-waarde	De waarde die door een functie wordt teruggegeven nadat deze klaar is met uitvoeren.
Runtime	Het moment waarop het programma wordt uitgevoerd; tijdens de uitvoering van het programma.
Runtime-omgeving	Programma dat gecompileerde code van bepaalde programmeertalen interpreteert en omzet naar machine-instructies terwijl het programma wordt uitgevoerd.
Service level agreement	Overeenkomst tussen CSP en cloud consumer betreffende het gebruik van de door de CSP aangeboden services.
SLA	Service level agreement
SOA	**Service-oriented architecture**: Softwarearchitectuur waarbij software wordt ontwikkeld als een verzameling van services die met elkaar communiceren om zo het programma tot stand te brengen.
SOAP	Simple object access protocol: XML-gebaseerd protocol waarbij verschillende software-services via het internet met elkaar kunnen communiceren.
SQL	**Standard Query Language**: Meest gebruikte taal voor het beheren van en interageren met databases.

Static typing	Typesysteem waarbij de types van variabelen, parameters, etc. in het programma al bij compiletime geweten zijn.
String	Type voor ASCII tekst in de meeste programmeertalen.
Strong typing	Vorm van static typing waarbij het static typing systeem niet omzeild kan worden.
Syntax	Regels voor het schrijven van software in een bepaalde programmeertaal.
Thread	Sequentie van instructies die onafhankelijk kan verwerkt worden door de machine binnen een programma.
Typecasting	Techniek waarbij in statically typed programmeertalen bepaalde types van variabelen door de programmeur bewust worden getransformeerd naar andere types.
Type-inferentie	Systeem waarbij de compiler de types van de variabelen en parameters, alsook de return types van functies zelf kan afleiden, rechtstreeks op basis van de code.
Url	**Uniform resource locator**: Internetadres in de vorm van een string waarmee web-services en webapplicaties mee kunnen aangesproken worden.
VM	**Virtual Machine:** Gevirtualiseerd computersysteem dat binnen een ander computersysteem draait.
Weak typing	Vorm van static typing waarbij het static typing systeem omzeild kan worden. Dit kan bijvoorbeeld in de vorm van Geheugenherinterpretatie zoals typecasting.
WSDL	**Web service description language**: een taal die in het SOAP protocol wordt gebruikt voor het beschrijven van de functionaliteit die webservices aanbieden.
XML	**Extensible markup language**: Opmaaktaal die gebruikt wordt om allerhande eigenschappen van programma's te definiëren.

Abstract

Cloud computing en functioneel programmeren zijn twee concepten die in de software-industrie de laatste jaren sterk aan populariteit gewonnen hebben. Deze concepten worden echter nog maar zelden met elkaar gecombineerd. De onderzoeksgroep ES&S van de KU Leuven vraagt zich af in hoeverre deze combinatie toch mogelijk is, en welke de voor- en nadelen hiervan zijn. Deze masterproef beoogt om een vergelijking te maken tussen verschillende functionele programmeertalen en hun klassieke tegenhangers in een cloud-context.

Bovenstaande vergelijking gebeurt op een tweeledige wijze. Aan de hand van een diepgaande literatuurstudie en enkele codevoorbeelden zet deze masterproef eerst de eigenschappen en concepten van enkele functionele programmeertalen naast elkaar, met bijzondere aandacht voor Wolfram en F#, alsook functionele varianten van Java en C#. Daarna volgt een analyse van de performantie van de verschillende talen, aan de hand van een testapplicatie. Dit gebeurt zowel lokaal als op verschillende cloud-platformen.

Hoewel er weinig tot geen standaard cloud-platformen zijn voor de gekozen functionele talen, blijkt het eenvoudiger een functioneel programma om te vormen tot een volwaardige cloud-applicatie dan de klassieke tegenhangers. Verder valt op dat de functionele implementaties veel beter parallelliseerbaar zijn in de cloud dan object-georiënteerde varianten en dat ze veel schaalbaarder zijn. Functioneel programmeren en cloud computing blijken dus zeer goed samen te gaan.

Abstract in English

Cloud computing and functional programming are two concepts in software engineering that have gained much popularity in recent years. However, these concepts are currently rarely combined. The research group ES&S from KU Leuven is interested to see how well a combination of these concepts can be realized, and what the advantages and disadvantaged of doing so are. This master's thesis aims to compare several functional programming languages with their established counterparts in a cloud context.

This comparison is done in two ways. Firstly, the concepts and properties of several functional programming languages are compared by means of a thorough literature review and some code examples, with special attention for Wolfram, F# and functional variants of Java and C#. Furthermore, the performance of these languages is analysed, using a test application. This analysis is done both locally and on several cloud platforms.

Although there are little to no standard cloud platforms available for the chosen functional languages, functional programs turn out to be much easier to transform into cloud applications than their established counterparts. Furthermore, functional implementations are much more scalable and parallelizable in the cloud than object-oriented implementations. Thus, functional programming and cloud computing turn out to be very compatible with one another.

1 Inleiding

Het inleidende hoofdstuk van deze scriptie is gewijd aan het onderzoeksopzet. Dit hoofdstuk schetst de situering en probleemstelling van de thesis, alsook de doelstellingen van dit onderzoek. Daarnaast schetsen volgende paragrafen de globale oplossingsmethode voor de eerder aangegeven problemen, en de hiervoor aangewonden hulpmiddelen.

1.1 Situering

Vanuit de software-industrie is er al vanaf haar ontstaan vraag naar almaar efficiëntere vormen van programmeren. Aangezien de complexiteit van software en de daarbij horende budgetten steeds toenemen wordt optimalisatie van het programmeerproces voortdurend belangrijker. Indien de efficiëntie van het programmeerproces niet verhoogt, zullen de investeringen die bedrijven moeten maken om software te ontwikkelen blijven stijgen tot op het punt waar het voor de meeste bedrijven niet meer haalbaar is om te blijven innoveren.
Een eerste aanpak om een vlotter ontwikkelingsproces te bekomen, vertrekt van het hergebruik van bestaande code, en dit over bedrijven heen. Hierdoor kunnen bedrijven sneller code ontwikkelen, aangezien ze niet steeds van nul moeten beginnen. Om dit hergebruik te stimuleren delen de programmeurs hun software op in services die vlot extern aangesproken kunnen worden. Dit concept bestaat al langer in de vorm van Service Oriented Architecture (SOA) en is tegenwoordig verfijnd in de vorm van cloud computing en micro-services. Deze technieken kennen ondertussen al een zekere maturiteit, maar ook nog een aantal onopgeloste beloften.
Voor deze masterproef is van de hierboven vernoemde concepten vooral cloud computing van belang. Hierbij wordt ontwikkelde software op speciaal ingerichte infrastructuur die doorgaans door een externe service provider beheerd wordt geïnstalleerd, en betalen klanten voor het exacte gebruik van resources – “pay per use”. Hierdoor moeten bedrijven niet meer investeren in dure servers die vaak enkel op piekmomenten optimaal benut worden [1]. Dit heeft een veel efficiënter gebruik van hardware tot gevolg, en de factuur die het bedrijf zal moeten betalen voor de infrastructuur is recht evenredig met het gebruik ervan. Verder zijn cloud based toepassingen veel schaalbaarder dan hun klassieke tegenhangers.
Anderzijds wordt er op softwareniveau al geruime tijd geëxperimenteerd met een alternatieve methode om software te ontwikkelen, namelijk functioneel programmeren. Het is reeds aangetoond dat functioneel programmeren beduidend beter scoort dan klassieke imperatieve talen wat betreft het aantal regels, ontwikkeltijd, leesbaarheid van de code, foutenlast, etc. [2].
Op zich hebben cloud computing en functioneel programmeren hun weg al goed gevonden in de industrie, maar tegenwoordig worden ze nog maar zelden simultaan toegepast. Deze masterproef beoogt de mogelijkheden van een combinatie tussen enerzijds functioneel programmeren en anderzijds cloud computing te bestuderen. Op deze manier moeten programmeurs enerzijds minder code ontwikkelen (cloud computing) en anderzijds kan de code die toch nog geschreven moet worden, sneller ontwikkeld worden (functioneel programmeren).

De masterproef situeert zich binnen de onderzoeksgroep Embedded Systems & Security (ES&S). ES&S houdt zich hoofdzakelijk bezig met het ontwikkelen van embedded systemen met een focus op een efficiënte implementatie van en toepassingen voor digitale data beveiliging, naast een track rond mobiele applicaties [3]. Via het project vLambda, waar ES&S zich reeds een jaar voor engageert, heeft de onderzoeksgroep zich eveneens gericht op functioneel programmeren [4]. Hierdoor kadert deze masterproef goed binnen hun onderzoeksgebied en kan ze hopelijk een reële bijdrage leveren aan de voortgang van dit onderzoeksproject.

1.2 Probleemstelling en onderzoeksvraag

Vanuit de industrie is er vraag naar betere schaalbaarheid van software en wil men makkelijker kunnen samenwerken in grote teams. De hierboven beschreven programmeerparadigma's zijn veelbelovend op die gebieden. Er moet echter nog onderzoek gedaan worden naar hoe deze paradigma's best te combineren zijn. Momenteel moeten programmeurs de voordelen die functioneel programmeren biedt grotendeels laten vallen als ze willen genieten van die van cloud computing, en vice versa. Deze masterproef wil aan de hand van praktische cases in verschillende declaratieve frameworks (o.a. Wolfram, Google Cloud en Amazon Cloud) bestuderen welke voordelen functioneel programmeren in een cloud omgeving kan bieden. Er is nood aan een concrete vergelijkende studie die verschillende functionele talen op een rijtje zet en vergelijkt met de klassieke tegenhangers binnen het kader van cloud computing.
Vooral voor opkomende bedrijven is het interessant om te weten in hoeverre functioneel programmeren in een cloud omgeving haalbaar is, aangezien deze bedrijven vaak de budgetten niet hebben om te investeren in hun eigen hardware-infrastructuur. Zij zouden in plaats daarvan gebruik kunnen maken van een cloud gebaseerde manier van werken, zonder de voordelen van functioneel programmeren te moeten laten vallen. Indien cloud computing niet gecombineerd kan worden met de grote efficiëntiewinst van functioneel programmeren, dreigen deze kleine opkomende bedrijven aan concurrentiekracht in te moeten boeten.
Programmeertalen moeten zo eenduidig mogelijk beoordeeld worden. De performantie van de talen is zeker niet de enige maatstaf, maar ook welke de mogelijkheden en beperkingen zijn van elke taal. Hierbij is het belangrijk welke filosofieën de verschillende talen hanteren en op welke concrete manier elk van deze talen het ontwikkelproces hoopt te verbeteren. Hierbij mag niet uit het oog verloren worden dat de meeste bestaande services in een cloud omgeving geschreven zijn in klassieke imperatieve talen. Voor de industrie is het van groot belang dat nieuwe declaratieve software volledig geïntegreerd kan worden in bestaande projecten. Zo wordt er gestreefd naar een heterogene cloud omgeving waarbij declaratieve en imperatieve services informatie met elkaar kunnen uitwisselen. Dit moet zonder conflicten mogelijk zijn, zelfs binnen eenzelfde project. Dit levert volgende onderzoeksvraag:

Welke programmeertalen met functionele eigenschappen zijn het best geschikt voor cloud computing, en hoe kunnen deze het best geïntegreerd worden in heterogene cloud computing software?

1.3 Doelstellingen

De hoofddoelstelling van deze masterproef is het vergelijken van verschillende functionele programmeertalen in een cloud context. Deze talen zullen enerzijds met elkaar vergeleken worden, en anderzijds met de bestaande oplossingen. Onder de bestaande oplossingen vallen technologieën zoals .NET en andere, reeds veelvuldig in de industrie gebruikte frameworks die cloud computing ondersteunen.
De klemtoon ligt vooral op de mate waarin een functionele oplossing beter is, waarbij die beter op veel facetten kan slaan: snelheid van ontwikkeling, expressiviteit, foutenlast, herbruikbaarheid, etc. Eén van de aspecten kan ook de uitvoeringssnelheid zijn, maar enkel wanneer dit een significant effect heeft. Ook de connectiesnelheid is in deze context van belang.
Naast deze performantie-gerichte criteria zal er ook gekeken worden naar de verschillen in ontwerpstrategie en -filosofie. Dit enerzijds tussen verschillende functionele talen onderling, en anderzijds tussen functionele talen en de huidige implementaties voor cloud based software.
Verder zullen ook de mogelijkheden van functionele talen in een cloud-based omgeving bestudeerd worden. Er zal nagegaan worden wat de beperkingen van functionele oplossingen zijn, en indien mogelijk zal er een manier gezocht worden om hier stijlvol mee om te gaan. Ook worden de beperkingen van de huidige implementaties onder de loep genomen om zo opnieuw een concrete vergelijking mogelijk te maken.

1.4 Materiaal en methode

Bovenstaande probleemstelling zal benaderd worden vanuit het perspectief van de programmeur. In het algemeen is deze masterproef abstract van aard, en zal de literatuurstudie een groot deel van het onderzoek uitmaken.
In eerste instantie zal de huidige stand van zaken rond cloud computing geanalyseerd worden. Dit zal hoofdzakelijk gedaan worden aan de hand van de cursus van het vak ‘SOA en Cloud Computing’ [5]. Tegelijk zullen verschillende functionele programmeertalen bestudeerd worden die al dan niet bestaande implementatiemogelijkheden bieden in een cloud omgeving. Zowel voor het cloud- als het functionele aspect zullen aanvullende bronnen worden gezocht via de bibliotheken van de UHasselt en de KU Leuven en het internet.
Eens een algemeen beeld van de stand van zaken gevormd is, kan er geanalyseerd worden welke functionele talen het meest in aanmerking komen voor cloud gerichte toepassingen. Een aantal van deze talen worden geselecteerd en diepgaander bestudeerd om een zo compleet mogelijk beeld te kunnen vormen hierover. Deze functionele talen moeten goed worden beheerst, zodat er cloud-gebaseerde programma's in deze talen geschreven kunnen worden, teneinde een waardevolle vergelijking mogelijk te maken.
Wanneer de keuze voor de talen vastligt, zal er overgegaan worden tot het schrijven van verschillende cloud programma’s in deze talen. Het programmeerwerk zelf zal gedaan worden in de voor die talen aangeraden integrated development environments (IDE’s).

Eveneens kunnen gelijkaardige cloud programma's geschreven worden, gebruikmakend van bestaande veelgebruikte technieken om opnieuw het vergelijken zo transparant en objectief mogelijk te maken.
De gekozen functionele talen worden vervolgens op een tweeledige manier vergeleken. Enerzijds aan de hand van de in de doelstellingen beschreven criteria. Hiervoor zullen concrete testscenario's moeten worden uitgewerkt die op elk van de aspecten van de gekozen functionele talen gaat toetsen welke taal al dan niet volstaat voor dat specifieke criterium.
Anderzijds zullen de filosofie en de high-level ontwerpstrategieën die de verschillende talen hanteren bestudeerd worden. Dit zal vooral gedaan worden door de documentatie van de verschillende talen te bestuderen en te vergelijken. Er wordt getracht om zo veel mogelijk informatie te verzamelen over dit abstracte aspect van elke taal. Ook zal het schrijven van cloud-gebaseerde programma's in deze talen helpen hierover een beter zicht te vormen. Op die manier kan er een correcte en enigszins meetbare vergelijking gemaakt worden tussen de verschillende talen betreffende dit abstracte aspect.
Ten slotte zal er nauw worden vergaderd met de promotoren om te evalueren of functionele talen naar de toekomst toe effectief een plaats hebben in de wereld van cloud computing, en of het bijgevolg relevant is om deze combinatie op te nemen in het onderwijsaanbod. We zullen samen met hen uitwerken hoe dit best gebeurt en eventueel een aantal presentaties samenstellen die we aan medestudenten kunnen geven bij wijze van evaluatie of onze aanpak in de praktijk effectief is.

2 Literatuurstudie

Het is belangrijk alvorens met het eigenlijke onderzoek te beginnen een vooronderzoek te doen in de vorm van een uitgebreide literatuurstudie, teneinde een goed beeld te kunnen vormen van de huidige stand van zake omtrent cloud computing enerzijds en functioneel programmeren anderzijds.
Het doel van deze literatuurstudie is om duidelijk af te bakenen wat het begrip cloud computing juist inhoudt, wat de voordelen en nadelen ervan zijn, en welke invloed het uitoefent op de hedendaagse software-industrie. Een ander doel van deze literatuurstudie is het begrip functioneel programmeren verkennen. Hiertoe worden de eigenschappen van het functionele programmeerparadigma bestudeerd, om zo de sterktes en zwaktes van functioneel programmeren te identificeren, en na te gaan in hoeverre functioneel programmeren vanuit een conceptueel standpunt samen zou kunnen gaan met cloud computing. Van daaruit zijn in dit hoofdstuk een aantal functionele programmeertalen bestudeerd die eventueel een plaats kunnen hebben in de verdere uitwerking van deze masterproef.

2.1 Cloud Computing

Het eerste concept dat voor dit onderzoek grondig bestudeerd is, is cloud computing. Zoals eerder vermeld is dit een modern concept in de software-industrie dat de laatste jaren enorm aan populariteit gewonnen heeft. Door de jaren heen zijn er tal van hardware/software frameworks ontwikkeld die cloud-eigenschappen hebben. Hierdoor is cloud computing tegenwoordig een zeer breed begrip, en is een diepgaande studie ervan noodzakelijk voor dit onderzoek. Deze paragraaf bakent om te beginnen een definitie af van cloud computing, en geeft daarna een overzicht van de belangrijkste vormen en eigenschappen ervan, en probeert op basis hiervan te identificeren welke aspecten van cloud computing relevant zijn voor dit onderzoek en welke niet. Ten slotte geeft deze paragraaf een beknopt overzicht van de belangrijkste spelers op de moderne cloud computing-markt die voor dit onderzoek in aanmerking komen.

2.1.1 Definitie

Cloud computing is ontstaan als een veelbelovend programmeerparadigma, en de grote populariteit ervan toont aan dat de hoge verwachtingen niet onterecht waren. Langs de andere kant wordt het begrip ‘cloud’ tegenwoordig te pas en te onpas gebruikt als hip verkoopargument met verschillende betekenissen. Hierdoor is de betekenis van het begrip cloud computing de laatste jaren ietwat vertroebeld. In feite is cloud computing op zich een abstract begrip, dat een groot aantal uiteenlopende vormen van systemen omvat. Daarom is het niet evident om een voldoende omvattende definitie ervoor te formuleren. In de literatuur zijn dan ook uiteenlopende definities voor het begrip ‘cloud computing’ te vinden. Zo definieert [1] cloud computing als volgt:

''A Cloud is a type of parallel and distributed system consisting of a collection of inter-connected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resource(s) based on service-level agreements established through negotiation between the service provider and consumers.'' [1: p.3].

Deze definitie is behoorlijk abstract, en geeft zo aan dat cloud computing een ruim begrip is. De definitie haalt wel een aantal eigenschappen aan die de meeste of zelfs alle vormen van cloud computing delen, met name:

- **Parallel and distributed system**: Deze eigenschap geeft aan dat een cloud systeem normaliter op meerdere servers (gedistribueerd) tegelijk (parallel) draait;
- **Virtualisation**: De software gedeployed in de cloud is in de regel volledig gescheiden van de hardware waarop ze draait;
- **Dynamic provisioning**: Er kunnen dynamisch meer of minder hardware resources ter beschikking worden gesteld van software die in de cloud draait;
- **Presented as a unified computers resource**: De onderliggende hardware en virtualisatie is geabstraheerd van de gebruiker, die enkel weet heeft van de toplaag van de cloud service waarmee hij rechtstreeks interageert;
- **Service-level agreements**: Gebruik van cloud services is meestal onderhevig aan een service-level agreement, waarin afspraken tussen de gebruiker en de cloud service provider zijn vastgelegd betreffende betaling, beschikbaarheid van de cloud diensten, etc.

Ondanks de algemeenheid van bovenstaande definitie, zijn in de literatuur nog vele andere definities voor cloud computing terug te vinden, waarvan vele ook andere eigenschappen toewijzen aan cloud computing. Zo vermeldt [6] naast die hierboven beschreven eigenschappen ook de volgende:

- IT-resources kunnen vanop afstand ter beschikking gesteld worden aan een gebruiker of cloud consumer door een cloud service provider (CSP), of door de cloud consumer zelf worden beheerd;
- Deze IT-resources kunnen zowel virtueel zijn zoals een virtuele server of cloud programma's, of hardware-georiënteerd zijn zoals een fysieke server;
- De cloud consumer kan door een toegang vanop afstand gebruik maken van de IT-resources van de cloud provider, waarvan dit gebruik onderhevig is aan de Service level agreements (SLA's) tussen consumer en provider.

Door de eigenschappen beschreven in [1] en [6] te combineren kan de volgende definitie voor cloud computing, die in dit onderzoek gebruikt zal worden, geformuleerd worden:

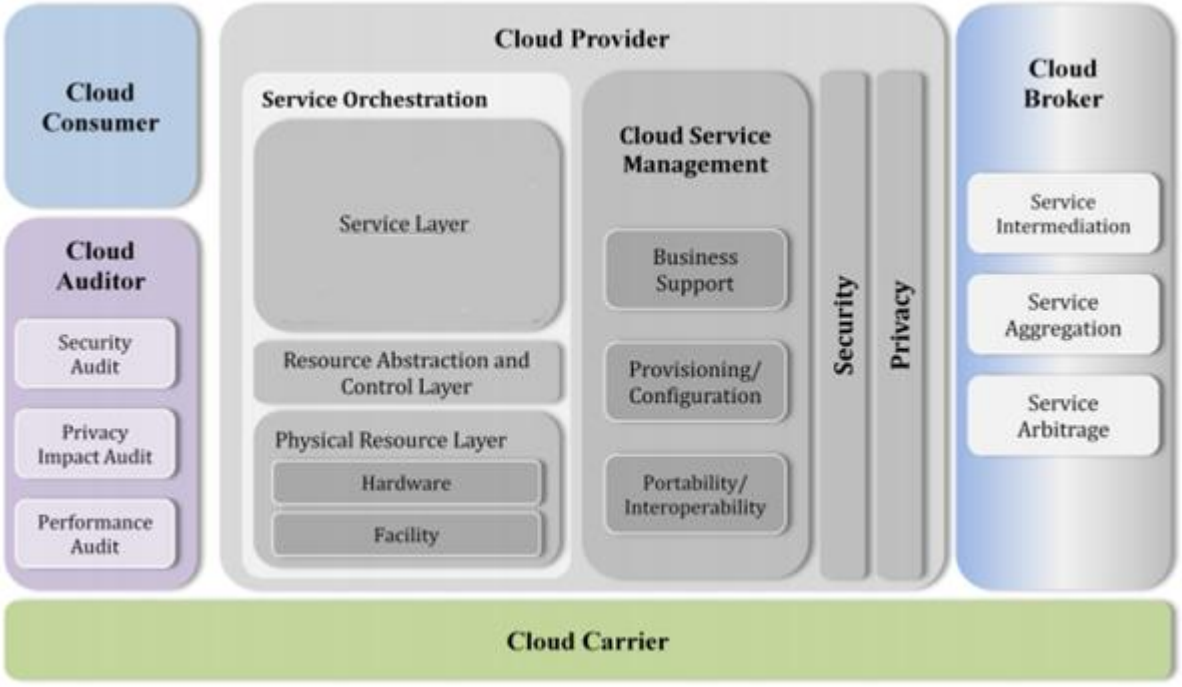
Cloud computing is een vorm van parallel en gedistribueerd systeem waarbij IT-resources vanop afstand ter beschikking gesteld worden aan cloud consumers, al dan niet via de tussenkomst van een cloud service provider, door het afspreken van service-level agreements, waarbij bepaald wordt wat de specifieke voorwaarden zijn en waarbij meestal betaald wordt voor de consumptie van de cloud resources. Deze systemen maken gebruik van virtualisatietechnieken om de services te scheiden van de hardware waarop ze draaien. Services krijgen dynamisch meer of minder hardware resources toegewezen afhankelijk van de vereisten die de services zelf stellen.

2.1.2 Architectuur

Uit de definitie van cloud computing volgt dat voor een cloud-infrastructuur een speciale architectuur noodzakelijk is. Bij het ontwerp van deze architectuur komen een aantal uitdagingen kijken, namelijk:

- De verschillende gebruikers van elkaar afschermen;
- De processen op de juiste manier schalen (o.a. meer of minder rekenkracht en opslagruimte geven) en prioriteiten geven zodat er aan de SLA voldaan kan worden;
- Dit alles registreren om de SLA's op te volgen en de juiste kosten aan te kunnen rekenen;
- En dit alles met een grote mate van veiligheid.

Er is ondertussen heel wat kennis opgebouwd rond deze uitdagingen en hoe ze het best aangepakt worden [7]. Zowat alle vormen van cloud computing hebben dan ook een gelijkaardige architectuur. Figuur 1 geeft een volledige algemene schematische weergave van de aanpak volgens het National Institute of Standardization and Technology (NIST).



Figuur 1: Algemene architectuur van cloud computing volgens het NIST [8].

Voor deze masterproef is deze architectuur niet dermate belangrijk dat alle onderdelen besproken moeten worden, maar het is wel van belang een algemene notie te hebben van hoe deze architectuur functioneert. Hieronder een korte bespreking van de belangrijkste aspecten.
Cloud services maken doorgaans gebruik van een hardware pool in plaats van op één bepaalde machine te draaien. Vanaf er een request binnenkomt van een cloud consumer zal een bovenliggende listener bepaalde IT-resources reserveren voor het afhandelen van deze request. Deze resources zijn afhankelijk van de request zelf en kunnen CPU, storage, networking, etc. inhouden. Vaak draaien cloud-applicaties in hun eigen virtual machine (VM).
A1s het volume van het binnenkomend verkeer en/of de rekenkracht die een applicatie vraagt sterk toeneemt, kan een aparte instantie van de cloud-applicatie gestart worden in een aparte VM. Sommige cloud services kunnen echter meerdere gebruikers tegelijk verwerken met één instantie, door extra CPU-kernen toe te wijzen aan deze ene instantie. Dit heet multitenacity en is beschreven in [7]. Deze applicaties zelf moeten dit wel ondersteunen. De verschillende VM's kunnen om veiligheidsredenen niet met elkaar communiceren.
Samengevat kan één cloud server meerdere instanties van dezelfde of verschillende cloud services hosten door middel van virtualisatie, en kan één cloud service op verschillende cloud servers draaien. Dit proces wordt geregeld door de automatic scaling listener die de resources beheert. Hierdoor wordt de grote schaalbaarheid en flexibiliteit van de cloud bekomen, aangezien de software volledig geabstraheerd wordt van de hardware.

2.1.3 Cloud Deployment Models

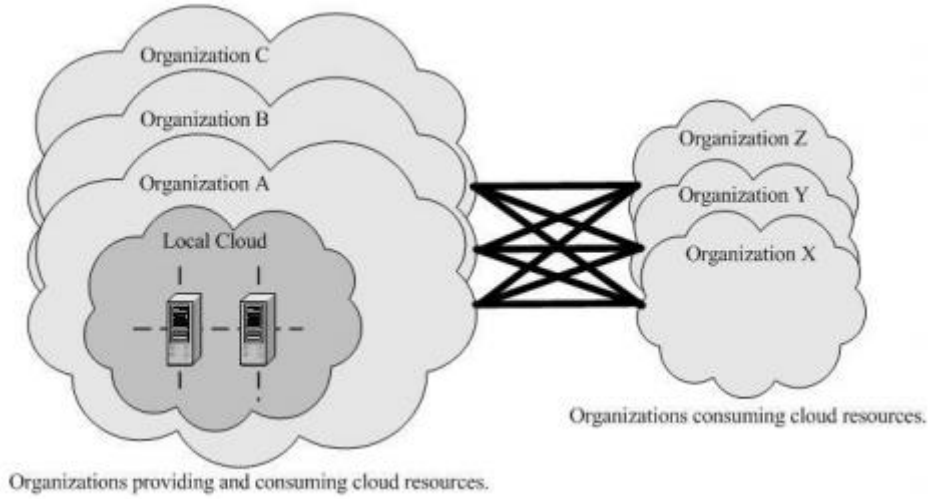
Een volgend belangrijk aspect van cloud computing is het deployment model: de manier waarop de cloud services naar de gebruiker worden gebracht. Er zijn een heel aantal verschillende cloud deployment models, elk met hun eigen voor- en nadelen [8]. Deze paragraaf geeft een overzicht van de meest gebruikte deployment models voor cloud computing, en gaat na wat de relevantie van elk van deze is voor deze masterproef.

2.1.3.1 Public Cloud

De meest gekende vorm van cloud computing is een public cloud. Hierbij is de cloud-infrastructuur door een cloud service provider beheerd en wordt deze openbaar ter beschikking gesteld via het internet. Iedereen die voldoet aan de SLA heeft toegang tot de services die door de public cloud aangeboden worden [8].
Het voordeel van een public cloud is dat het gebruik ervan eenvoudig en flexibel is. Dit is ook de goedkoopste vorm van cloud computing omdat een groot aantal personen van dezelfde infrastructuur gebruik kunnen maken. Het grootste nadeel van deze manier van werken is dat ze relatief onveilig is aangezien iedereen toegang heeft tot de infrastructuur waar soms gevoelige gegevens van gebruikers op aanwezig zijn. Goede beveiliging is dus cruciaal bij het gebruik van deze vorm van cloud computing. In paragraaf 2.1.7.2 wordt meer in detail ingegaan op het probleem van beveiliging in cloud computing en hoe deze aangepakt kunnen worden.

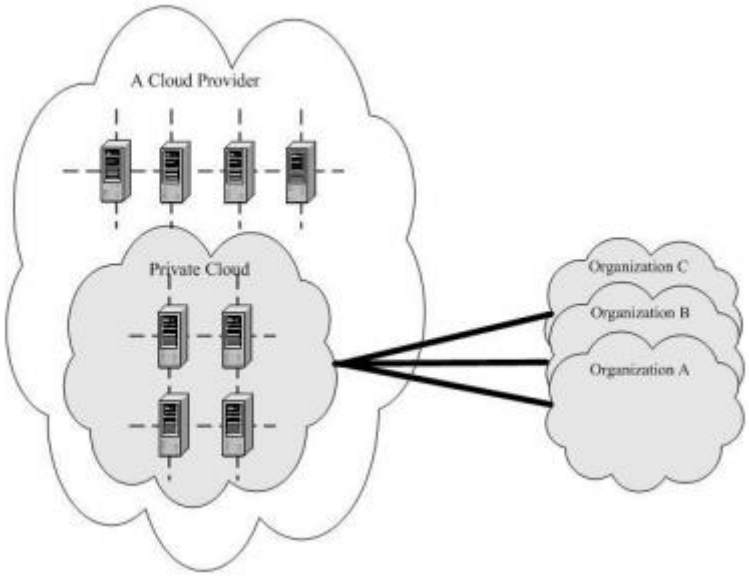
2.1.3.2 Community Cloud

Een community cloud is gelijkaardig aan een public cloud, maar de toegang is beperkt tot een 'gemeenschap'. Deze gemeenschap kan een groep van bedrijven zijn die samen services willen uitwisselen, of een groep cloud consumers in het algemeen die een netwerk van services hebben waar zij toegang toe willen, en bovendien willen dat hun gedeelde services afgeschermd zijn van de buitenwereld.
De community cloud kan nog eens opgedeeld worden in twee vormen: enerzijds de on-site community cloud, en anderzijds de outsourced community cloud [8]. Bij de on-site variant beheert elk lid van de community (in dit geval meestal grote bedrijven) zijn eigen cloud infrastructuur. Deze infrastructuur wordt dan via een netwerkverbinding gedeeld tussen de verschillende leden van de community. Figuur 2 geeft dit principe grafisch weer. De leden van de community zijn dus zowel cloud providers als cloud consumers en dragen dus allemaal de verantwoordelijkheid om hun cloud infrastructuur te beheren.



Figuur 2: On-site community cloud [8].

Anderzijds bestaat er ook een outsourced community cloud. Deze lijkt zeer sterk op een public cloud. Een externe CSP beheert de cloud-infrastructuur en de community-leden krijgen hier toegang toe via het internet. Figuur 3 laat dit principe zien. Het enige significante verschil tussen een public cloud en een outsourced community cloud is dat bij deze laatste enkel de leden van de community toegang hebben tot de cloud services.



Figuur 3: Outsourced community cloud [8].

2.1.3.3 Private Cloud

Ten slotte bestaat er ook nog de private cloud. Dit is zoals de term al doet vermoeden een afgeschermde cloud die in het bezit is van één bepaalde organisatie. Deze organisatie is tegelijk de cloud consumer en cloud service provider. De resources worden enkel aangeboden aan gekozen leden. Het nut hiervan is dat binnen een groot bedrijf centralisatie van resources bekomen kan worden, waardoor deze resources toegankelijk zijn voor verschillende delen van het bedrijf die zich al dan niet op verschillende locaties bevinden. Door deze techniek zal er efficiënter omgegaan kunnen worden met resources en zullen er minder problemen zijn omtrent beveiliging.
Ook private clouds kunnen opgedeeld worden in on-site en outsourced private clouds. Deze opdeling is analoog aan de opdeling van community clouds beschreven in vorige paragraaf, met dezelfde voor- en nadelen.

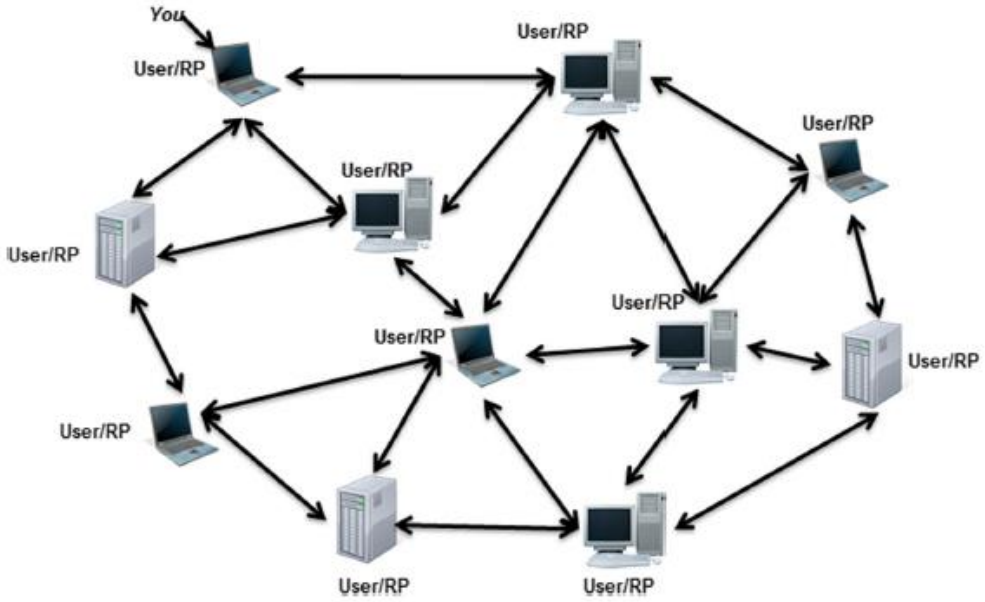
2.1.3.4 Hybrid Cloud

Bovenstaande categorieën zijn nog steeds niet voldoende om alle cloud services die tegenwoordig op de markt zijn te omvatten. Om die reden is het begrip hybrid cloud ingevoerd. Hybrid clouds bestaan uit een combinatie van meerdere soorten van de hierboven beschreven cloud deployment models. Een bedrijf kan bijvoorbeeld gebruik maken van een private cloud voor belangrijke data en een public cloud voor minder belangrijke data. Er kan ook gekozen worden voor een mengeling van on-site en outsourced cloud componenten, etc…

2.1.3.5 Distributed Cloud

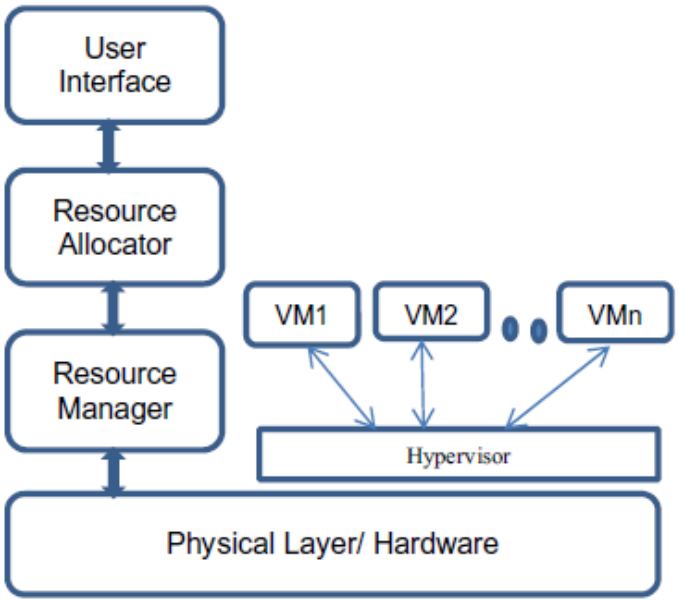
Een interessante recente ontwikkeling op het gebied van cloud deployment models is de Distributed Cloud. Voorlopig is dit deployment model nog slechts op academisch niveau gerealiseerd, maar onderzoek uit onder andere [9] geeft aan dat dit deployment model veel potentieel heeft en mogelijk op industriële schaal kan toegepast worden in de toekomst. Daarom is ook dit model opgenomen in deze literatuurstudie.

De huidige deployment models voor cloud computing bieden reeds grote voordelen, die in detail besproken worden in paragraaf 2.1.6. Er is echter nog wat ruimte voor verbetering. Tegenwoordig worden clouds immers beheerd in gecentraliseerde datacenters, of op zijn minst met één of meerdere servers die toegewijd zijn aan het voorzien van de cloud services. In [9] wordt voorgesteld om afstand te nemen hiervan en in plaats daarvan over te stappen op een Distributed Cloud waarbij de gebruikers zelf ook cloud service providers zijn, en hun eigen machines gebruiken om cloud computing te hosten, in plaats van toegewijde servers. Figuur 4 geeft schematisch weer hoe dit in zijn werk gaat.



Figuur 4: Distributed cloud model [9].

Figuur 4 toont op het eerste zicht een peer-gedistribueerd netwerk. Het verschil tussen dit gedistribueerd cloud model en een klassiek gedistribueerd netwerk is dat op elke node van het netwerk virtualisatiesoftware draait. Deze is in principe gelijkaardig aan die van de klassieke cloud implementaties, zoals weergegeven in Figuur 5: Distributed cloud architectuur .. De basiscomponenten zoals de hypervisor en de verschillende VM's zijn duidelijk nog steeds aanwezig. De hypervisor is in essentie een verzamelwoord voor de componenten uit Figuur 1 die de abstractie van de hardware voor hun rekening nemen.



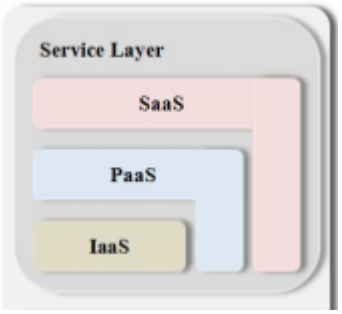
Figuur 5: Distributed cloud architectuur [9].

Bestanden die worden opgeslagen in een distributed cloud netwerk, worden steeds op meerdere nodes opgeslagen. Hierdoor voorkomt men dat bestanden onbereikbaar worden wanneer een bepaalde node wegvalt. In [9] kiest men ervoor om 10 kopies op te slaan van elk bestand verspreid over het hele netwerk. Over het exacte aantal kan gediscussieerd worden, maar het is afwegen tussen performantie of bescherming tegen het uitvallen van nodes.
Dit systeem zou na de vandaag reeds talrijk beschikbare klassieke cloud deployment models een volgende grote efficiëntiewinst kunnen betekenen in de IT-sector, aangezien veel computers een groot deel van hun resources het grootste deel van de tijd niet gebruiken. Aangezien gebruikers in dit deployment model zelf hun computer beschikbaar stellen als cloud-platform, is de CSP zoals deze nu bestaat bij de klassieke cloud deployment models hier overbodig.
Het distributed cloud model staat echter nog maar in zijn kinderschoenen. In [9] worden een aantal werkpunten aangehaald voor verder onderzoek naar distributed cloud computing. Deze zijn “security, privacy, and trust related issues” [9: p. 16]. Het spreekt voor zich dat deze zaken een grote uitdaging vormen voor de distributed cloud ten opzichte van de klassieke cloud implementaties, aangezien data van gebruikers rechtstreeks bij andere gebruikers opgeslagen wordt.

2.1.3.6 Besluit

Voor deze masterproef is de public cloud het meest voor de hand liggende platform om mee te werken, aangezien dit de meest flexibele en makkelijkst verkrijgbare vorm van cloud computing is. De verschillen met de andere deployment models hebben dan ook te maken met maturiteit, en vooral privacy- en beveiligingsimplicaties. Het is belangrijk om het bestaan van deze implicaties te kennen, maar dit is niet de focus van deze masterproef. Daarom volstaat het gebruik van een public cloud voor dit onderzoek.

2.1.4 Cloud Delivery Models



Figuur 6: Service layer van het NIST cloud architecture model [8].

Er zijn verschillende manieren om cloud services aan te bieden aan cloud consumers. In het algemeen zijn drie basisvormen te onderscheiden, namelijk Infrastructure as a Service (IaaS), Platform as a Service (PaaS), en Software as a Service (SaaS). Verder zijn er nog een heel aantal minder bekende delivery models. Deze worden allemaal samengenomen onder de noemer 'Anything as a Service' (XaaS). Hieronder een toelichting voor elk van deze vormen.
Merk op dat de IaaS-, PaaS-, en SaaS-modellen op hiërarchische wijze kunnen worden opgebouwd, maar dat hoeft niet per se. Figuur 6 geeft een detail weer van de service layer uit Figuur 1. We zien dat het mogelijk is om SaaS-services aan te bieden bovenop een PaaS cloud architectuur, maar men kan evenzeer van low level XaaS of IaaS vertrekken en een eigen implementatie gebruiken ter vervanging van de bovenliggende lagen [8].

2.1.4.1 Infrastructure as a Service (IaaS)

Figuur 7: Structuur van een IaaS cloud delivery model [5].

Het IaaS model is in principe de meest low-level manier om cloud diensten aan te bieden. IaaS is gericht op consumenten die gebruik willen maken van de cloud, maar zo veel mogelijk controle wensen over hun cloud-services [5]. Figuur 7 geeft schematisch weer hoe een IaaS cloud delivery model eruitziet. De groene delen worden door de Cloud Service Provider (CSP) voorzien. De grijze delen dienen door de Cloud Consumer zelf geïmplementeerd te worden. Merk op dat de CSP in dit geval enkel de hardware voorziet en de nodige abstractielaag om aan de eerder besproken virtualisatie te kunnen doen. De cloud consumer is volledig vrij om zijn eigen implementatie van de software te kiezen. Op deze manier is de cloud consumer niet verplicht om zich aan een bepaalde programmeertaal of een bepaald framework te houden, wat bij de andere cloud delivery models wel het geval is zoals dadelijk besproken zal worden. Het nadeel van deze manier van werken is dat de cloud consumer instaat voor het beheer van alle software op zijn cloud-ruimte. Dit kan soms ingewikkelder worden dan gewenst.

2.1.4.2 Platform as a Service (PaaS)

Figuur 8 geeft de structuur van het PaaS cloud delivery model weer. Bij dit model voorziet de CSP naast de hardware en hypervisor ook een Virtual Machine (VM) en Operating System (OS), met daarbovenop middleware en een kant-en-klare Runtime-omgeving [5]. Deze vorm van cloud delivery is gericht op gebruikers die software hebben in een bepaalde taal of een bepaald framework en die graag 'zonder zorgen' hun software willen deployen, testen, builden, …
Het voordeel van PaaS ten opzichte van IaaS is dat de verantwoordelijkheid voor het beheren van de VM en de extra software die nodig is om het platform te voorzien bij de CSP ligt in plaats van bij de gebruiker. Hierdoor is het gebruik van het PaaS model aanzienlijk eenvoudiger dan dat van het IaaS model. Het nadeel is dan weer dat de gebruiker controle verliest over de exacte implementatie van de software. Verder zijn PaaS modellen vaak beperkt tot één enkel framework of één enkele programmeertaal, waardoor het PaaS model aanzienlijk minder flexibel is dan het IaaS model.

Figuur 8: Structuur van een PaaS cloud delivery model [5].

2.1.4.3 Software as a Service (SaaS)

Figuur 9 geeft de structuur van het SaaS cloud delivery model weer. Dit is het meest complete model dat CSPs aan kunnen bieden. Het gaat hier over afgewerkte software die rechtstreeks aan de gebruiker ter beschikking wordt gesteld [5]. Meestal kan de gebruiker dan via een API interageren met de SaaS cloud software om zo data te verkrijgen uit de cloud. Deze data kan dan verwerkt worden in de software van de gebruiker, die zelf al dan niet in een cloud-omgeving draait.

Figuur 9: Structuur van een IaaS cloud delivery model [5].

2.1.4.4 Anything as a Service (XaaS)

Recent spreekt men ook van XaaS als een cloud delivery model. Geavanceerde cloud netwerken kunnen immers eender wat aanbieden als service, gaande van communicatie en data-opslag tot netwerken, monitoring, etc. [9]. Omdat de grens tussen IaaS, PaaS, en SaaS niet altijd heel scherp is, wordt hiervoor soms ook de term XaaS gebruikt [10]. XaaS is dus een meer algemene omvattende term waar naast de drie klassieke modellen ook nog andere, meestal gespecialiseerdere specifieke services mee bedoeld kunnen worden.

2.1.4.5 Besluit

Voor deze masterproef lijkt het IaaS model interessant omdat dit model zo dicht mogelijk bij de hardware staat, wat het vergelijken van de performantie tussen talen eenvoudiger maakt. Het biedt de gebruiker ook de vrijheid om frameworks te installeren die functionele talen ondersteunen.

Verder kan het PaaS model zeker ook interessant zijn, op voorwaarde dat voor de bestudeerde functionele talen een cloud service beschikbaar is volgens dit delivery model. Dergelijke talen zijn dan ook interessant voor deze masterproef, aangezien het ontwikkelen van software die uitvoerbaar is op een kant-en-klare PaaS cloud aanzienlijk eenvoudiger is dan hetzelfde doen voor een IaaS cloud.
Het SaaS-model is dan weer minder interessant omdat het doel van deze masterproef is om zelf functioneel of declaratief geprogrammeerde software te schrijven en te deployen in de cloud. Het is wel goed om te weten dat dit cloud delivery model bestaat. Eventueel kunnen er SaaS cloud services verwerkt worden in de applicaties die geschreven zullen worden voor de masterproef.
Aangezien XaaS veelal een theoretische term is en de toepassingen ervan zeer specifiek zijn en gebonden aan de specifieke CSP die XaaS-services aanbiedt is dit cloud delivery model niet interessant voor deze masterproef, en komen verdere toepassingen van dit delivery model niet aan bod in deze thesis.

2.1.5 Huidige implementaties cloud computing

Deze paragraaf heeft als doel een idee te scheppen van hoe tegenwoordig cloud-applicaties geschreven worden, en op welke manier deze in de cloud geïmplementeerd zijn.
Belangrijke PaaS cloud providers ondersteunen vaak slechts een beperkt aantal programmeertalen. De talen die het meest gebruikelijk zijn bij een PaaS service, zijn Javascript, Java, Python, PHP, Go en .NET.
Een belangrijk punt voor de masterproef is de kennis van hoe web-services functioneren. Deze services moeten via het web kunnen communiceren en dit gebeurt met aantal protocollen, architecturen en talen [5].
Een vooraanstaand protocol dat gebruikt wordt om gestructureerd informatie uit te wisselen is het Simple Object Access Protocol (SOAP). SOAP gebruikt XML (EXtensible Markup Language) om gestructureerde berichten te verzenden via onderliggende protocollen. Via XML kan data beschreven en gestructureerd worden voor de communicatie tussen systemen. Met SOAP wordt informatie verzonden van het ene naar een andere operating system met behulp van XML en een transport protocol (meestal http). SOAP is ook gemaakt met een vrije keuze van programmeertaal en ontwikkelparadigma als doel.
Web Service Description Language (WSDL) is een op XML gebaseerde taal die binnen het SOAP-protocol wordt gebruikt voor het beschrijven van de functionaliteit die SOAP-gebaseerde webservices aanbieden. Het beschrijft algemene gegevens over de service, gaande van de locatie van de services, tot de methodes aangeboden door de services.
SOAP is een krachtig, maar vrij complex protocol. De grote complexiteit zorgt ervoor dat dit niet zo eenvoudig bruikbaar is. Als antwoord hierop is REST, REpresentational State Transfer, ontstaan. Het idee hierachter is om minder complex te kunnen communiceren tussen verschillende systemen. In tegenstelling tot SOAP is dit geen protocol, maar een architectuur. REST-applicaties gebruiken http om data te posten, lezen en verwijderen. De web services zullen praktisch worden opgeroepen met een URL.

Ook is er een alternatief voor XML ontstaan, namelijk JSON. JavaScript Object Notation is ontworpen om informatie op een gestructureerde manier door te geven en is een lichter dan XML. Voor vele programmeertalen is er ondersteuning voor JSON waardoor het op een grote schaal gebruikt kan worden.
Rest en JSON zijn beide een lichter dan hun voorgangers en ook veel populairder. Veel bekende web-services maken hier dan ook gebruik van. Zo gebruikt de API van Google Maps REST en JSON, maar ook nog XML. Facebook hanteert een REST architectuur en stuurt informatie door in JSON-formaat. Ook Dropbox gebruikt JSON en REST.

2.1.6 Voordelen cloud computing

De snelgroeiende populariteit van cloud computing is te danken aan een aantal grote voordelen die dit programmeerprincipe biedt ten opzichte van klassieke methodes om software te distribueren. In [11] wordt een uitgebreid overzicht gegeven van de talrijke voordelen van cloud computing. Omdat cloud computing op zich niet de focus van deze masterproef is, maar wel het programmeren in de cloud, haalt deze paragraaf enkel de belangrijkste voordelen van cloud computing aan. Het is belangrijk om deze voordelen en de aard ervan te kennen, zodat tijdens de uitwerkingsfase van de masterproef op deze voordelen ingespeeld kan worden.

2.1.6.1 CapEx Vs OpEx

Volgens de klassieke methode van software ontwikkelen en distribueren, moet er voordat de software tot bij de klant kan worden gebracht geïnvesteerd worden in hardware om bepaalde functionaliteit aan te kunnen bieden (bvb. rekenkracht, netwerk of centrale data opslag). Deze investeringen kunnen groot zijn en ze moeten ook vaak gebeuren voordat er geld binnenkomt van de klanten. Vooral voor beginnende bedrijven is dit een probleem. Een bijkomend probleem is dat de populariteit van software soms moeilijk in te schatten is voordat ze gedistribueerd wordt. Daarom wordt vaak te veel geïnvesteerd in te krachtige infrastructuur. Deze eenmalige kapitaalsinvesteringen heten Capital Expenses (CapEx). Cloud computing lost dit probleem volledig op. Bij cloud computing betaalt men immers voor het gebruik van de software die in de cloud gedeployed is. De uitgaven zijn dan recht evenredig met de gebruikte resources. Dit wordt Operational Expenses (OpEx) genoemd [11]. Het spreekt voor zich dat OpEx wenselijk is voor de meeste beginnende bedrijven, tenzij er op voorhand exact geweten is hoe sterk de software de infrastructuur zal belasten en er voldoende kapitaal beschikbaar is.

2.1.6.2 Schaalbaarheid

Een volgend belangrijk voordeel van cloud computing is de grote schaalbaarheid ervan. Omwille van de structuur van cloud-infrastructuur is cloud computing van nature zeer schaalbaar. Het is wel belangrijk om op te merken dat de cloud applicaties zelf ook op een schaalbare manier geschreven dienen te worden. Niet alle applicaties schalen immers van nature even goed mee met de cloud waar ze in gedeployed zijn. Schaalbaarheid is vanzelfsprekend een groot voordeel wanneer de populariteit van een ontwikkelde app op

voorhand niet goed in te schatten is. Indien de app niet in de cloud gedeployed is en de vraag ernaar is veel groter dan verwacht, kan de server crashen en valt de service volledig buiten dienst. In de cloud daarentegen, kunnen er gewoon extra instances van de app gestart worden op verschillende servers van de CSP zonder dat de eindgebruiker daar last van heeft, op voorwaarde natuurlijk dat de CSP voldoende infrastructuur ter beschikking heeft. Voor de meeste public clouds is dit bijna altijd het geval.
Een nieuw opkomend veld binnen IT waar de schaalbaarheid van cloud computing zeer goed tot haar recht komt is data mining [12]. De laatste jaren is er zeer veel data verzameld en opgeslagen in enorme databanken zoals Google' s BigTable. Omdat relationele databases niet erg schaalbaar zijn, wordt voor de opslag van deze data dikwijls Not only SQL (NoSQL) gebruikt. Om data uit deze enorme databases te halen volstaan klassieke SQL-queries niet. Hiervoor is een radicaal nieuw systeem nodig gebaseerd op een nieuwe architectuur. In [12] wordt naar MapReduce gekeken. Dit is een cloud-gebaseerd alternatief voor SQL en blijkt veel beter te presteren voor grote hoeveelheden data omdat meerdere machines of 'nodes' samenwerken aan éénzelfde query. Er wordt wel aangegeven dat MapReduce aanzienlijk ingewikkelder is om te gebruiken dan SQL, maar er staat een grote snelheidswinst voor grote queries tegenover.
Voor deze masterproef is de grote schaalbaarheid van cloud computing zeer interessant omdat functionele programmeertalen vaak ook veel schaalbaarder zijn hun imperatieve tegenhangers. Paragraaf 2.2.3 gaat hier in detail op in. Zoals hierboven aangegeven beïnvloedt de schaalbaarheid van de applicatie zelf de schaalbaarheid van de cloud-implementatie ervan. Dit is dus een gebied waar functioneel programmeren zou kunnen uitblinken. MapReduce verwijst trouwens naar een techniek die dikwijls in functionele talen toegepast wordt: map om functies op reeksen data los te laten, en reduce om reeksen data tot één waarde te herleiden.

2.1.6.3 Flexibiliteit

Een laatste belangrijk voor deze masterproef interessant voordeel dat in [11] wordt aangehaald is dat cloud services veel flexibeler zijn dan klassieke deployment methodes. Indien er gebruik wordt gemaakt van een public cloud of een outsourced community of private cloud moet de gebruiker zich niet meer bezighouden met het inrichten van hardware infrastructuur om de ontwikkelde software op te installeren. Het PaaS delivery model biedt zelfs de mogelijkheid software rechtstreeks op het door de CSP voorziene platform te plaatsen, zonder dat de ontwikkelaar zich in principe zorgen moet maken over de concrete deployment en het low-level management van de servers. SaaS-services bieden ten slotte zelfs afgewerkte softwarecomponenten zodat het wiel niet steeds opnieuw moet worden uitgevonden door ontwikkelaars die gelijkaardige functionaliteit wensen. Cloud computing maakt het leven van softwareontwikkelaars dus vaak een stuk aangenamer.
Flexibiliteit is ook iets waar veel functionele talen sterk in scoren (zie paragraaf 2.2.3). Daarom is het ook de moeite waard om hier nader naar te kijken in deze masterproef.

2.1.7 Nadelen Cloud Computing

Naast de talrijke voordelen heeft cloud computing van nature ook een aantal beperkingen. Hieronder een overzicht van de belangrijkste hiervan.

2.1.7.1 Verlies van controle

Een eerste nadeel van cloud computing is dat de cloud consumer controle over zijn applicaties of de infrastrucuur waarop ze draait bij veel vormen van cloud computing uit handen moet geven aan een extern bedrijf. Dit is iets waar veel ontwikkelaars en systeembeheerders het moeilijk mee hebben [11]. Zij willen graag steeds de touwtjes strak in handen houden en onmiddellijk kunnen ingrijpen als er iets mis gaat. In de praktijk kan meestal wel via de SLA gedetailleerd worden vastgelegd hoe vaak de cloud service beschikbaar moet zijn, alsook andere zaken die te maken hebben met het uit handen geven van het beheer van de infrastructuur. Indien deze SLA niet wordt nageleefd kan de cloud consumer gerechtelijke stappen ondernemen. De consumer is dus ingedekt.
Dit neemt niet weg dat sommigen nog steeds weigerachtig staan tegenover het opgeven van controle. Dit probleem kan worden opgelost door over te stappen op een on-site private cloud, maar dan valt het voordeel van het beheer van de cloud over te laten aan een extern bedrijf weg. De flexibiliteit van cloud computing vermindert in dat geval dus. Verder kan men ook kiezen om gebruik te maken van IaaS in plaats van bijvoorbeeld SaaS of PaaS, maar dit heeft opnieuw de afweging tussen meer controle en grotere flexibiliteit tot gevolg.

2.1.7.2 Beveiliging

Een tweede belangrijk nadeel van cloud computing is beveiliging. Aangezien de cloud infrastructuur meestal beheerd wordt door een extern bedrijf kan cloud computing voor veel gebruikers minder betrouwbaar overkomen dan zelf alle hardware beheren. Daar staat wel tegenover dat de externe provider waarschijnlijk meer gespecialiseerd is in veiligheid dan de gebruiker zelf, en zou men dus evenzeer kunnen zeggen dat de beveiliging bij een externe CSP net hoger is.
Een ander probleem is dat cloud servers publiek zijn en iedereen software kan draaien op deze servers. Als iemand erin slaagt zo een server te hacken en de virtualisatie zou kunnen omzeilen, kan hij vaak aan de details van andere software die op deze server draait. Daarom is het van groot belang voorzichtig om te springen met data die in de cloud opgeslagen wordt en zo veel mogelijk encryptie toe te passen. De exacte mechanismen van de veiligheidsimplicaties van cloud computing zouden te ver leiden voor deze masterproef. Geïnteresseerde lezers worden doorverwezen naar [13].
Het grootste beveiliging-gerelateerde probleem bij cloud computing stelt zich wanneer de CSP zelf onbetrouwbaar is. In dat geval bestaat er een reëel risico dat data gelekt wordt, en andere cybercriminaliteit een bedreiging vormt voor de applicaties in de cloud. Daarom is het zorgvuldig uitkiezen van een geschikte CSP van groot belang, alsook het afsluiten van een gerechtelijk sluitende SLA met de CSP, met daarin duidelijke afspraken met betrekking tot veiligheid.

Vanzelfsprekend is het voor deze masterproef belangrijk om te weten dat er een veiligheidsrisico bestaat bij het gebruik van cloud computing. Desalniettemin is dit een breed en voorlopig nog turbulent onderzoeksgebied op zich, en behandelt deze masterproef om die reden beveiliging in de cloud niet verder.

2.1.8 Keuzecriteria cloud service providers

Door de in de laatste jaren sterk toegenomen populariteit van cloud computing is er tegenwoordig een groot aanbod aan cloud service providers. Het is dus belangrijk om op een verstandige manier de meest geschikte provider te kiezen. Deze keuze zal afhangen van de volgende diverse criteria, die te vinden zijn in [14], [15], [16] en [17]:

- **Betalingen:** Veel cloud providers hebben een gratis proefperiode of een gratis, maar sterk gelimiteerd aanbod. Dit op zich is al een belangrijke asset en laat toe om de provider te evalueren voor er overgegaan wordt naar betaald gebruik. Daarnaast is niet alleen de prijs voor de gebruikte services een belangrijk criterium, maar ook een eenduidige en duidelijke betaalmethode kan onverwachte kosten vermijden. Een duidelijke SLA speelt hier een grote rol in;
- **Snelheid en Capaciteit:** De klant moet snel een response kunnen krijgen van zijn applicatie, een wachttijd van bijvoorbeeld meerdere seconden om een antwoord te krijgen van de server is een groot nadeel. Hoeveel processing power is er beschikbaar voor een bepaalde prijs en hoe groot is de latency tussen de klant en de provider? De klant moet ook voldoende capaciteit kunnen krijgen zodat alle requests behandeld kunnen worden in een aanvaardbare tijd;
- **Reputatie:** Een betrouwbare provider die al bewezen heeft om een goede service te leveren aan klanten zal hoger in het lijstje komen te staan. De grotere, meer bekende providers hebben ook een grotere community. Hierdoor is het makkelijker om oplossingen te vinden voor vaak voorkomende problemen;
- **Veiligheid:** De cloud providers moeten de veiligheid van data kunnen garanderen. Deze veiligheid hangt van een groot aantal factoren af zoals de soort encryptie, de manier van datascheiding, autorisatie van de klanten, weerstand tegen aanvallen, fysieke beveiliging, maatregelen tegen dataverlies etc.;
- **Gebruiksgemak:** Een app deployen in de cloud moet een eenvoudige taak zijn. Het moet makkelijk zijn om in de cloud omgeving te werken, een app te managen of om een app te kunnen updaten. De leercurve voor het gebruiken van de services mag niet te groot zijn. De API van de provider is eveneens een belangrijke factor. Een uitgebreide API met gestandaardiseerde methodes is een groot voordeel;
- **Schaalbaarheid:** Cloud services zijn over het algemeen vrij schaalbaar, dit is ook een vereiste voor het kiezen van een provider. De voornaamste providers scoren hier goed op waardoor dit geen al te grote problemen mag vormen;
- **Consistentie:** De services die de provider ter beschikking stelt moeten zo veel mogelijk beschikbaar zijn. De klant moet ten alle tijden zijn applicaties kunnen gebruiken. Ook moet de provider consistent zijn op het gebied van performantie. Wanneer de services van de provider onder zware druk komen te staan moet de klant nog altijd even vlot kunnen werken met zijn applicaties.

De criteria voor het kiezen van de juiste cloud service provider zijn dus talrijk. Het komt er op aan een juiste balans te zoeken tussen al deze criteria, aangezien het onrealistisch is om te verwachten dat één cloud service provider aan al deze criteria uitstekend kan voldoen.

2.1.9 Voorbeelden Cloud Services

Er zijn tegenwoordig een heel aantal aanbieders van cloud services op de markt. Enkele voorbeelden zijn Amazon, Google, Salesforce, IBM, Microsoft, en Oracle Cloud [1]. Aangezien voor deze masterproef gebruik gemaakt zal worden van cloud services, moet er afgewogen kunnen worden welke van deze het meest toepasselijk zijn. Hieronder een overzicht van de belangrijkste cloud service providers op dit moment.

2.1.9.1 Amazon

Amazon biedt cloud services aan onder de naam van Amazon web services (AWS) via public, private en hybrid clouds. Amazon Elastic Compute Cloud levert schaalbare rekenkracht via de cloud [18]. Via de cloud kunnen klanten eenvoudig applicaties maken met behulp van beschikbare ontwikkelingstools en deze laten draaien op de virtuele servers van Amazon. Deze applicaties kunnen in de vorm van images geüpload worden naar Amazon simple storage service [19]. Hiernaast zijn andere services, voornamelijk in de vorm van Iaas en Paas beschikbaar. Amazon bezit een groot aantal services die in combinatie met elkaar gebruikt kunnen worden en richt zich op bedrijven, individuen, maar ook studenten. Samen met een gratis proeftijd van een jaar geeft deze provider heel wat voordelen in de cloud markt.

2.1.9.2 Google

Google speelt een grote rol als een public cloud provider. Google is in het bezit van een groot scala van krachtige services in de vorm van Iaas, Paas en Saas [20], onder de naam Google Cloud Platform. Met Google App Engine, wat deel is van het Google Cloud Platform, kunnen zeer schaalbare applicaties ontworpen worden en zijn verschillende API's zoals NoSQLdatastores beschikbaar [21]. Na het uploaden van een app, zorgt Google volledig voor het onderhoud hiervan.
Google Compute Engine is de IaaS-service van Google, waarmee probleemloos VM's van alle formaten kunnen worden aangemaakt en beheerd in de cloud.
Naast de App en Compute Engine is er ook BigTable wat in essentie een NoSQL database is. Dit is een krachtige database die ontworpen is voor een enorm grote schaalbaarheid [22]. Een ander product van Google is BigQuery, een service die query's kan uitvoeren voor zeer grote hoeveelheden data. Dit alles maakt Google tot een van de grootste providers.

2.1.9.3 Microsoft

Azure is het cloud platform van Microsoft voor het ontwikkelen en uitvoeren van applicaties. Naast deze App services zijn ook virtuele machines, dataopslag, en vele andere services aanwezig in de vorm van Iaas en Paas [23]. Microsoft heeft talrijke services en biedt deze aan via public en private clouds. Azure richt zich niet enkel op bedrijven en is ook toegankelijk voor studenten via Microsoft Imagine. Studenten kunnen hierdoor kosteloos gebruik maken van de cloud.

2.1.9.4 IBM

IBM cloud biedt services aan via private, public en hybrid clouds. Aan de hand van het Bluemix cloud platform zorgt IBM voor Iaas en Paas services zoals virtuele servers, dataopslag en APIs om apps te ontwikkelen [24]. De doelgroepen van IBM cloud zijn kleine tot grote bedrijven, maar studenten kunnen ook gratis gebruik maken van bepaalde services.

2.1.9.5 Oracle Cloud

Het Oracle Cloud platform levert services in de vorm van de drie belangrijkste cloud delivery models die vooral gericht zijn op bedrijven [25]. De Saas services die snelle en krachtige oplossingen bieden aan deze bedrijven, zijn vertegenwoordigd in de vorm van marketing, data-analyse, Resource planning etc. Hiernaast zijn er PaaS-services voor het eenvoudig ontwerpen van apps waaronder voor onder andere datamanagement, Security en Business analyse. Ook levert Oracle Cloud IaaS-services voor schaalbare computing en data storage. Al deze voorgaande services zijn beschikbaar via de public cloud van Oracle. Ook biedt Oracle Cloud een gratis testperiode aan van 30 dagen, en $300 aan cloud credit om een deel van hun services uit te proberen.

2.1.9.6 Salesforce

Salesforce biedt verschillende services aan in de vorm van PaaS zoals de App Cloud, IoT Cloud, Sales Cloud etc. [26]. Deze public cloud provider richt zich eerder op kleine bedrijven die schaalbare oplossingen zoeken. Deze provider heeft tevens een minder uitgebreid aanbod aan services in vergelijking met de voorgaande providers.

2.1.9.7 Besluit

De dag vandaag zijn er een groot aantal cloud service providers die elk hun troeven hebben. De meest aantrekkelijke providers zijn deze met het grootste aanbod, zowel in delivery models als aantal services voor diverse cloud gebruikers. Er zijn dus meer dan voldoende keuzemogelijkheden wat betreft IaaS en Paas cloud-platformen voor deze masterproef. Op basis van deze paragraaf lijken Amazon, Google, en Microsoft de interessantste cloud service providers. Er zijn echter nog een heel aantal andere criteria waarop je de keuze van de geschikte service providers kan baseren, zoals in vorige paragraaf beschreven. Daarnaast is ondersteuning voor de specifieke talen die in dit onderzoek bestudeerd worden een doorslaggevende factor. Hoewel deze literatuurstudie al een eerste richting aangeeft, is het voorlopig dus nog te vroeg om specifieke CSPs te kiezen of te elimineren.

2.1.10 Aanverwante Programmeerprincipes

Nu exact geweten is wat cloud computing juist inhoudt en welke vormen van cloud computing er bestaan, is het ook van belang om even stil te staan bij de aanverwante programmeerprincipes die géén cloud computing zijn. Zo zijn er immers een aantal en de lijn tussen al deze programmeerprincipes kan heel vaag zijn. Bovendien worden deze principes vaak in combinatie met cloud computing toegepast. Het is belangrijk om een onderscheid te kunnen maken tussen al deze principes, teneinde verwarring zo veel mogelijk te vermijden. Hieronder een opsomming van belangrijkste aan cloud computing verwante begrippen.

2.1.10.1 Service Oriented Architecture (SOA)

SOA omvat het gebruik en combineren van verschillende services om zo tot een applicatie te komen. Deze services kunnen web services zijn, maar dit hoeft niet per se [5]. Daarenboven zijn deze services autonoom. Als er code in een project wordt toegevoegd, mag dit geen impact hebben op de bestaande services in het project.
Services kunnen dynamisch en stateless worden aangeroepen. Dit betekent dat de services geen informatie bijhouden over gebruikers die de service aanroepen. Alle informatie over die de service nodig heeft voor het afhandelen van een aanvraag moet rechtstreeks in de aanvraag zelf zitten [27]. Hierdoor kan code zeer gemakkelijk modulair en recycleerbaar worden opgebouwd. Het zorgt er ook voor dat de services altijd op een voorspelbare manier zullen reageren op requests. Dit betekent concreet dat een service die initieel geschreven is voor een bepaalde applicatie, makkelijk kan geïntegreerd worden in andere applicaties die de functionaliteit nodig hebben die deze service biedt.
Cloud computing kan SOA-georiënteerd zijn, maar dit is niet noodzakelijk. De twee kunnen onafhankelijk van elkaar bestaan. Een applicatie kan bijvoorbeeld geschreven zijn volgens het SOA-principe, maar op een gewone server worden gedeployed. Ook kan een cloud applicatie volgens een ander ontwerppatroon zijn opgebouwd, zoals ASP.NET MVC. Het is wel zo dat een combinatie van SOA en cloud computing zeer vaak wordt toegepast, omdat de gedistribueerde eigenschappen van SOA goed aanleunen bij de flexibele aanpak van cloud computing.

2.1.10.2 Microservices

Microservices zijn een onderdeel van SOA. Het is niet eenvoudig een sluitende definitie van microservices te geven. Het onderscheid tussen microservices en standaard SOA is dan ook subtiel. Uit [28] kan afgeleid worden dat één van de verschillen eruit bestaat dat microservices niet enkel autonoom werken, maar ook onafhankelijk van elkaar kunnen gebuild en gedeployed worden. Dit is geen vereiste bij standaard SOA. Denk maar aan services in het .NET Core bijvoorbeeld. Die zitten ingebouwd in de .NET-applicatie en zijn sterk aan de applicatie zelf gekoppeld.
Nog een belangrijk verschil tussen microservices en standaard SOA is dat microservices nog minimalistischer van aard zijn. Een enkele microservice neemt slechts één zeer specifieke taak binnen een project voor zijn rekening. De opdeling van een groot project in afzonderlijke services is dus verder doorgedreven [29]. Het voordeel hiervan is dat de

applicatie nog modulairder wordt. Langs de andere kant wordt er kritiek geuit op dit model omdat deze doorgedreven opsplitsing van verantwoordelijkheden over verschillende services heen een te grote overhead met zich meebrengt en de applicatie minder overzichtelijk wordt.
Microservices kunnen net als SOA zeer goed worden toegepast in combinatie met cloud computing. Het is belangrijk goed in te zien dat cloud computing en microservices met elkaar verweven kunnen zijn, maar de technologieën een ander aspect van het programmeerproces omvatten. Voor deze masterproef kan het interessant zijn om microservices te implementeren in de cloud.

2.1.10.3 Web-services

Een laatste technologie die sterk lijkt op cloud computing, maar hier toch niet mee verward mag worden is web-services. In essentie hebben ze dezelfde functionaliteit, maar toch zijn er verschillen [30]. Met web-services worden simpelweg services die via het web worden aangesproken bedoeld. Aangezien cloud services doorgaans via het web worden aangesproken, zijn bijna alle cloud services ook web-services.
Niet alle web-services zijn echter cloud services. Wat een web-service tot een cloud service maakt is het gebruik van de typische cloud architectuur en de virtualisatie zoals beschreven in paragraaf 2.1.2. De definitie geformuleerd in 2.1.1 laat ook duidelijk zien dat het gebruik van virtualisatie een eis is om van cloud computing te kunnen spreken.
Web-services die dus géén cloud services zijn, zijn web-services die gebruik maken van hun eigen hardware infrastructuur, en dit zonder virtualisatie [30].
Reguliere web-services kunnen uiteraard ook in combinatie met cloud services gebruikt worden. Dit is zeker het geval bij het gebruik van een SOA-architectuur. In dit geval kan bijvoorbeeld interactie met een database waar gevoelige informatie op staat aan de hand van een reguliere web-service gebeuren, waarbij de hardware door de ontwikkelaars zelf beheerd wordt, terwijl de UI-services van de applicatie in de cloud gedeployed zijn.

2.2 Functioneel Programmeren

Het tweede belangrijke aspect van deze masterproef is functioneel programmeren. Het is dus essentieel om te weten wat dit begrip inhoudt en wat de voordelen en nadelen ervan zijn ten opzichte van andere programmeerparadigma's. Ook is het belangrijk kennis te vergaren over de belangrijkste functionele programmeertalen, en een idee te vormen van de mogelijkheden die deze talen bieden voor cloud computing. Deze paragraaf behandelt al deze aspecten van functioneel programmeren.

2.2.1 Definitie

Functioneel programmeren is een programmeerparadigma dat een computerprogramma voorstelt als een aaneenschakeling van verschillende functies. Functies zijn dus de elementaire bouwblokken van het programma. De voorstelling van deze functies in het functioneel programmeerparadigma is sterk gebaseerd op de λ−calculus [31]. In essentie stelt de λ−calculus een functie voor als een soort 'zwarte doos' die een bepaalde input omzet naar een bepaalde output. Deze omzetting gebeurt volledig abstract. Het is onmogelijk om de interne werking van de functie te analyseren. Functies hebben bovendien geen toestand, wat betekent dat ze steeds dezelfde output genereren op basis van een bepaalde input. Verder heeft elke functie enkel betrekking op zijn input en output, en heeft het uitvoeren van een functie dus geen enkel effect op de buitenwereld.
Functioneel programmeren maakt deel uit van het bredere declaratieve programmeerparadigma. Dit paradigma beschrijft programma's vertrekkende vanuit het probleem dat het programma moet oplossen, en de oplossing van dit probleem. De focus ligt dus op de toepassing van het programma. De interne structuur van het programma en hoe de machine het programma juist dient uit te voeren zijn bij declaratief programmeren minder van belang [32]. De declaratieve aard van functioneel programmeren is duidelijk doordat bijvoorbeeld de interne werking van functies onzichtbaar is voor de programmeur. Een functie kan dus enkel beschreven worden op basis van het resultaat dat de functie moet bekomen, en niet op basis van de manier waarop de functie tot een bepaald resultaat moet komen.
Gebaseerd op het bovenstaande kan volgende definitie voor functioneel programmeren geformuleerd worden:

Functioneel programmeren is een declaratief programmeerparadigma dat een programma beschrijft als een aaneenschakeling van elementaire functies, die ondoorzichtig en toestandsloos zijn, en elk een bepaalde input omzetten naar een bepaalde output, zonder daarbij enige andere bewerkingen uit te voeren die de toestand van de buitenwereld veranderen.

Bovenstaande definitie is behoorlijk algemeen en sluitend, en omvat goed wat deze masterproef onder functioneel programmeren verstaat. Voor de rest van dit onderzoek is deze definitie voor functioneel programmeren dus van toepassing.

2.2.2 Eigenschappen

Functioneel programmeren verschilt radicaal van de meest gebruikte huidige programmeerparadigma's. Een ander, meer gekend paradigma is object-georiënteerd (OO) programmeren dat deel uitmaakt van imperatief programmeren [33]. Omdat dit laatste beter gekend is dan functioneel programmeren, zetten volgende paragrafen beide paradigma's naast elkaar om zo beter inzicht te krijgen in de eigenschappen van functioneel programmeren, en wat functioneel programmeren juist onderscheidt van object-georiënteerd programmeren.

2.2.2.1 Voorstellingswijze van het programma

Programmeerparadigma's hebben elk hun eigen denkwijze. Bij functioneel programmeren de centrale vraag: 'wat wil ik dat de code doet', terwijl deze bij object-georiënteerd programmeren luidt: 'hoe moet de code uitgevoerd worden'.
Concreet stelt het object-georiënteerde programmeerparadigma heel het programma voor aan de hand van objecten. Uitvoering van object-georiënteerde code is in se een interactie tussen verschillende objecten, waarbij deze objecten van toestand veranderen.
Bij functioneel programmeren daarentegen is de functie het centraal element. Door verschillende functies op een georganiseerde manier te groeperen kunnen complexe programma's ontstaan. Data kan immers van functie tot functie worden doorgegeven, waarbij elke functie een bepaald deel van de verwerking van de inputdata op zich neemt, om zo uiteindelijk tot een output te komen.

2.2.2.2 Onveranderlijke data

Een belangrijke eigenschap van functionele talen is dat data onveranderlijk is. De waarde van een variabele kan niet wijzigen eens de variabele is gecreëerd. Functionele talen transformeren een set input data naar een andere set output data, terwijl de input data zelf niet verandert binnen de functie [34]. Dit impliceert dat functionele talen alternatieven moeten gebruiken voor bepaalde in object-georiënteerde elementaire zaken, zoals globale variabelen. Om de staat van het programma bij te houden maken object georiënteerde programmeertalen immers vaak gebruik hiervan. In plaats van globale variabelen gebruiken functionele talen extra parameters die de functies met elkaar uitwisselen. Zo ontstaat er toch enige notie van toestand binnen een functioneel programma.

2.2.2.3 Recursie

Een functionele programmeerstijl maakt ook vaak gebruik van recursie. Dit is een programmeertechniek waarbij een functie zichzelf opnieuw oproept, waardoor een lus ontstaat. Bij functioneel programmeren wordt recursie gebruikt in plaats van de in object-georiënteerde talen beter bekende lussen zoals for-loops. In object georiënteerde talen is recursie ook mogelijk, maar wordt het minder toegepast omdat veel programmeurs opgegroeid zijn met de for-lus en deze daarom als meer intuïtief beschouwen. Dit neemt niet weg dat recursie een stijlvol programmeerprincipe is dat bovendien meestal compacter is dan een for-loop. Daarenboven geeft recursie de betekenis van de code beter weer.

2.2.2.4 Eerste-orde functies

In functioneel programmeren zijn functies eerste-orde elementen. Dit betekent dat functies in essentie dezelfde status hebben als data. In functionele programmeertalen kunnen functies als parameter doorgegeven worden aan andere functies, en kunnen functies opgeslagen worden in (onveranderlijke) variabelen. Dit is een zeer krachtige eigenschap van functioneel programmeren waardoor allerlei bewerkingen in functionele talen vaak een veel compactere notatie hebben dan in object-georiënteerde programmeertalen. Verder kan een functie zelf de output zijn van een andere functie. De functies die functies genereren als output en/of consumeren als parameter worden hogere-orde functies genoemd.

2.2.2.5 Geen strikte control flow

Omdat functioneel programmeren deel is van het declaratief programmeren, is de control flow van een functioneel programma niet strikt gedefinieerd. Dit is in sterk contrast met object-georiënteerd programmeren. Een strikte control flow betekent dat de manier waarop de code wordt uitgevoerd vast staat en bepaald is door de programmeur. Bij imperatief programmeren moet de programmeur steeds stap voor stap in code omzetten wat hij wilt dat de machine doet, terwijl de programmeur bij declaratief programmeren eerder een beschrijving geeft van wat hij wilt dat het resultaat is van de code. De exacte implementatie ervan laat hij aan de compiler over.

2.2.2.6 Lazy evaluation

Ten slotte maken verschillende declaratieve – en dus ook functionele – talen gebruik van lazy evaluation. Dit houdt in dat de waarde van een variabele pas berekend wordt wanneer deze effectief gebruikt wordt. Dit is in tegenstelling tot eager evaluation, waar de waarde van een variabele wordt berekend op het moment waarop een waarde aan de variabele wordt gekoppeld. Imperatieve talen zijn gedwongen om met eager evaluation te werken omwille van hun eerder besproken strikt gedefinieerde control flow. Functionele talen hebben hier vanwege hun declaratieve natuur geen problemen mee en kiezen dus dikwijls voor lazy evaluation.

2.2.3 Voordelen

Functioneel programmeren biedt een aantal voordelen ten opzichte van andere programmeerparadigma's, die veelal volgen uit de hierboven gespecifieerde eigenschappen van functionele talen. Deze voordelen hebben ervoor gezorgd dat in recente jaren er een verschuiving in populariteit plaatsvindt naar het functioneel paradigma [33]. Hieronder een opsomming van de belangrijkste hiervan.

2.2.3.1 Geen neveneffecten

Dankzij de onveranderlijke data die besproken is in de vorige paragraaf heeft functioneel programmeren geen last van zogenaamde neveneffecten. Een neveneffect is in feite eender welke toestandsverandering binnen een programma [35]. Neveneffecten kunnen soms onaangename en onverwachte bijwerkingen hebben, zeker als de toestandsveranderingen asynchroon gebeuren en meerdere threads tegelijk hetzelfde element bewerken. Dit heeft de in object-georiënteerd programmeren bekende race conditions tot gevolg. Functioneel programmeren heeft hier geen last van.
Door neveneffecten kan debuggen ook ingewikkeld worden. Door neveneffecten kunnen elementen namelijk vanuit verschillende plaatsen veranderd worden, waardoor uiteindelijk het overzicht zoekraakt.

2.2.3.2 Schaalbaarheid en inherent parallelisme

Functioneel programmeren is van nature ook zeer schaalbaar. Dit komt grotendeels door de onveranderlijke data, het ontbreken van neveneffecten en het feit dat de volgorde van de uitvoering van het programma niet per sé vastligt. Hierdoor zijn programma's inherent parallel uit te voeren. Bij functioneel programmeren moet er immers niet gewacht worden op andere processen die datastructuren gelockt hebben omdat deze processen er wijzigingen in aanbrengen. Neem bijvoorbeeld het quicksort-algoritme, dat in detail wordt beschreven in [36]. In de functionele implementatie kunnen gemakkelijk de kleinste en grootste helft van de lijst in aparte processen uitgevoerd worden, terwijl dit bij imperatieve programma's moeilijker is omdat ze tegelijk dezelfde lijst willen aanpassen. De grote schaalbaarheid en parallellisme maken het heel wat eenvoudiger om functionele programma's gedistribueerd te draaien op systemen met meerdere processors.
Schaalbaarheid en parallellisme zijn belangrijke voordelen van functioneel programmeren die in deze masterproef van groot belang kunnen zijn. Cloud computing bezit van nature ook gedistribueerde eigenschappen die schaalbaarheid vooropstellen, zoals beschreven in 2.1.2.

2.2.3.3 Efficiënter ontwikkelproces

Door concepten als hogere-orde functies, recursie en het aaneenschakelen van meerdere functies, is code vaak leesbaarder en korter bij functionele talen dan bij hun object-georiënteerde tegenhangers.
Doordat functies elk vaak slechts een kleine verantwoordelijkheid hebben, is een functioneel programma van nature modulairder dan een object-georiënteerd programma. Dit is vooral een voordeel wanneer een team van meerdere ontwikkelaars samen aan eenzelfde project werkt. Zij kunnen vaak volledig autonoom werken zonder dat de code van de ene rechtstreeks invloed heeft op die van de andere. Bovendien is code op die manier ook herbruikbaarder.

2.2.3.4 Minder foutenlast

Door het ontbreken van neveneffecten en doordat functies compact zijn, wordt de code ook minder foutgevoelig. Elke functie heeft een duidelijke, makkelijk te definiëren taak. Bovendien is het ontbreken van een sterk getypeerde control flow nog een groot voordeel naar foutenlast toe. Hierdoor is de uitvoering van het programma vaak overzichtelijker. De programmeur kan tijdens het debuggen mooi de functionaliteit van de code volgen, in plaats van verstrikt te raken in meerdere geneste if- en for- structuren, waar men gemakkelijk het overzicht in verliest.
Door al de hierboven genoemde praktische voordelen van functioneel programmeren is de productiviteit van de programmeur hoger en daalt de kost van softwareontwikkeling eens de ontwikkelaar de functionele programmeerstijl beheerst.

2.2.4 Nadelen

Ondanks alle hierboven vermelde voordelen wordt functioneel programmeren nog niet grootschalig toegepast, en is het imperatieve programmeerparadigma de dag van vandaag nog steeds de standaard. De reden hiervoor is dat er uiteraard een aantal, soms grote, nadelen zijn verbonden aan het gebruik van functioneel programmeren. Het is zelfs zo dat in veel van de hierboven vermelde voordelen ook een nadeel schuilt [37]. Hieronder een opsomming van de belangrijkste nadelen van functioneel programmeren.

2.2.4.1 Moeizaam leerproces

In de eerste plaats is functioneel programmeren moeilijk om aan te leren aan programmeurs die jarenlang imperatieve talen gebruikt hebben. Dit komt doordat functioneel programmeren in essentie zo radicaal verschilt van imperatieve talen, en doordat functioneel programmeren wiskundiger van aard is. Hierdoor staan veel ervaren programmeurs weigerachtig tegenover een overstap naar functionele ontwikkelmethodes.

2.2.4.2 Financieel aspect

Naast de kosten voor de omscholing zijn er nog de kosten voor het omzetten van bestaande software. De meeste bestaande code is momenteel geschreven aan de hand van imperatieve talen. Deze projecten bieden vaak weinig tot geen integratiemogelijkheden met functionele programmeerprincipes, en ze zomaar omzetten naar een functionele programmeertaal is vaak niet haalbaar. Deze omzetting is immers bijzonder duur en er zijn vaak eenvoudigweg niet genoeg programmeurs om dat klaar te spelen [37]. De grotere programmeersnelheid en compactere code die functioneel programmeren aanbiedt is niet voldoende om investeerders over te halen om deze stap te nemen.

2.2.4.3 Minder intuïtief

Verder is functioneel programmeren voor veel toepassingen ook minder intuïtief. Dit wordt veroorzaakt door het ontbreken van zij-effecten. Veel programma's in de echte wereld willen juist dat deze zij-effecten bestaan omdat de toestand van een bestaand object veranderen intuïtiever is dan een nieuw object creëren met andere eigenschappen. Wanneer een

gebruiker bijvoorbeeld tekst typt in een tekstveld, wilt de gebruiker in eerste instantie dat de inhoud van dat tekstveld verandert. Dit is niet mogelijk in functionele programmeertalen. In plaats daarvan wordt er een nieuw tekstveld gegenereerd met een andere inhoud, en wordt het eerste tekstveld vervangen door het nieuwe. Het spreekt voor zich dat dit een extra niveau van abstractie met zich meebrengt en dit kan bepaalde problemen die heel simpel zijn in imperatief programmeren zeer complex maken in functionele talen [37].

2.2.4.4 Lagere en moeilijk in te schatten uitvoeringssnelheid

Een volgend nadeel van veel functionele talen is dat de uitvoeringssnelheid voor veel applicaties lager kan zijn. Dit nadeel ligt opnieuw in de natuur van functioneel programmeren. Bij functionele talen is er een complexere en actievere garbage collector nodig dan bij imperatieve talen, bijvoorbeeld aangezien er veel meer (onveranderlijke) data wordt gecreëerd die na de uitvoering van een (kort) proces niet meer nodig is [38].
Hierbij komt dat bij functionele talen die aan lazy evaluation doen de performantie zeer moeilijk in te schatten is [39]. Aangezien waardes van variabelen immers pas worden uitgerekend wanneer het echt nodig is en het vaak niet duidelijk is wanneer dit zo is, -denk maar aan een eenvoudige if-structuur met meerdere voorwaarden- is het zelfs voor experts zeer moeilijk in te schatten wat de performantie van een functioneel programma zal zijn. In theorie is lazy evaluation sneller dan eager evaluation omdat de waarden pas uitgerekend worden wanneer ze écht nodig zijn, maar het uitvoeringsmodel is complexer, wat ook een performantiekost heeft.
Vooral vroeger vormden bovenstaande beperkingen een reëel probleem, maar de compilers van functionele programmeertalen zijn tegenwoordig dermate gevorderd dat ze in veel gevallen zelfs sneller zijn dan object-georiënteerde talen. Zo genereert de GHC-compiler van de functionele taal Haskell (zie paragraaf 2.2.5.1) tegenwoordig zeer snelle programma's. Bovendien is de hardware tegenwoordig zo snel dat de tragere uitvoeringstijd van functionele programma's zelden merkbaar is. Feit blijft dat functionele talen hun reputatie tegen hebben.

2.2.4.5 Minder support

Een laatste nadeel van functionele talen is dat deze door hun kleinere, meer gespecialiseerde gebruikersgroepen vaak een minder uitgebreide community hebben. Dit maakt het moeilijker om via het internet aan vrijblijvende hulp te komen, of kant-en-klare bibliotheken te vinden die bepaalde complexe taken aanpakken. Hierdoor ontstaat meer overhead waar de programmeur mee geconfronteerd wordt wanneer hij een applicatie van formaat wilt schrijven in een functionele taal [38]. Voor veel bedrijven waar tijd gelijk staat aan geld is dit een significant nadeel.
Ook de tools (IDE's etc.) en vooral de debuggers die voor functionele talen bestaan zijn minder gevorderd dan degene die beschikbaar zijn voor object-georiënteerde talen. Dit levert moeilijkheden op voor minder vaardige programmeurs. De grotere programmeerefficiëntie die functioneel programmeren biedt zoals beschreven in paragraaf 2.2.3.3 is dus vooral van toepassing op reeds gevorderde ontwikkelaars.

2.2.5 Belangrijkste functionele talen

Functioneel programmeren is op zich geen nieuw concept. Er is al jaren een groot aanbod aan functionele programmeertalen op de markt. De laatste tijd heeft functioneel programmeren echter veel aan belangstelling gewonnen. Er ontstaan tegenwoordig heel wat nieuwe programmeertalen met functionele eigenschappen. Bovendien beschikken veel vooraanstaande imperatieve programmeertalen zoals Java en C# tegenwoordig over functionele eigenschappen in de vorm van lambda-expressies.
Voor deze masterproef is het van belang een overzicht te hebben van de belangrijkste functionele programmeertalen, alsook enkele recente talen met functionele eigenschappen. Bovendien is het belangrijk om na te gaan op welke manier functioneel programmeren een plaats heeft gekregen in klassieke imperatieve programmeertalen. Hieronder een overzicht van een selectie van talen met functionele eigenschappen die voor deze masterproef mogelijk interessant zijn.

2.2.5.1 Haskell

Haskell is de officieuze norm in de wereld van functionele programmeertalen. Haskell is een zuiver functionele taal die vooral in de academische wereld veel toepassing ziet. Haskell hanteert dan ook academische principes; de wiskundige en programmeertechnische puurheid van de taal is vooropgesteld ten opzichte van het gebruiksgemak. De belangrijkste aspecten van Haskell zijn zeer goed beschreven in de officiële documentatie [40]. Hieronder een overzicht van de belangrijkste van deze eigenschappen.
Haskell is een zogenaamde 'statically typed' programmeertaal. Dit betekent dat types in Haskell gekend zijn 'at compile time', wanneer het programma gecompileerd wordt. Static typing heeft zowel voor- als tegenstanders. Langs de ene kant is static typing een voordeel aangezien de compiler bugs met betrekking tot types vroegtijdig kan detecteren en de programmeur hiervan op de hoogte kan stellen voordat de code compileert.
Sommige programmeurs vinden dit soort van controle echter overbodig, en werken liever met 'dynamic typing'. Hierbij zijn de types pas gekend 'at runtime', of wanneer het programma wordt uitgevoerd. Hierdoor moeten types niet door de programmeur worden opgegeven. Dit spaart heel wat codeerwerk bij het schrijven van de code, waardoor de code bovendien compacter en leesbaarder is. Het nadeel van deze manier van werken is dat dynamic typing meer ruimte laat voor fouten, waardoor het programma kan crashen.
Sommige statically typed talen zijn ook 'strongly typed'. Dit betekent dat het static typing systeem van de taal niet omzeild kan worden [41]. De meeste statically typed talen zijn echter weakly typed. Dit betekent dat objecten van verschillende types kunnen worden samengevoegd, of dat de compiler types kan herinterpreteren indien de programmeur dat wenst. In Haskell is dit niet het geval. Haskell is dus een strongly statically typed programmeertaal [41]. Hier komt de zuiverheid van de taal naar boven.
Verder is type-inferentie sterk uitgewerkt in Haskell. Type-inferentie betekent dat de compiler zelf de type informatie van variabelen zal injecteren waar mogelijk. Op deze manier is het niet noodzakelijk dat de programmeur overal types definieert in Haskell, zonder de strongly en statically typed eigenschappen van Haskell teniet te doen. Op die manier omzeilt

Haskell één van de belangrijkste nadelen van het static typing systeem, namelijk dat de programmeur steeds de types moet opgeven in de code.
Verder is Haskell een taal die lazy evaluation toepast zoals in paragraaf 2.2.2.6 beschreven.
Haskell is ook zeer goed met parallelisme. Dit komt door de expliciete manier waarop Haskell omgaat met neveneffecten. Deze zijn immers strikt verboden in Haskell, in tegenstelling tot veel andere talen met functionele eigenschappen, die een zekere vorm van neveneffecten nog steeds toelaten. Parallellisme is in Haskell ver doorgetrokken. De zogenaamde GHC-compiler van Haskell werkt bijvoorbeeld met een parallelle garbage collector [40].
Ten slotte is Haskell een volledig open source taal en kan iedereen zijn eigen uitbreidingen creëren voor de taal. Hierdoor is Haskell zeer uitgebreid geworden voor een functionele taal, aangezien door het academisch karakter heel wat academici uitbreidingen voor de taal geschreven hebben.

2.2.5.2 Erlang/OTP

Erlang is net als Haskell een zuiver functionele programmeertaal. Erlang is ontwikkeld door Sony-Ericsson en is sinds 1998 open-source [42]. Erlang wordt meestal in één adem uitgesproken met OTP. Dit is een set van middleware libraries die het mogelijk maken grote schaalbare systemen uit te werken in Erlang. De focus van Erlang is om schaalbare software te creëren met een focus op betrouwbaarheid [43]. Hieronder volgt een opsomming van de belangrijkste eigenschappen van Erlang [44].
Fouttolerantie is een groot pluspunt van Erlang. De taal is van de grond op gebouwd om zo betrouwbaar mogelijk te zijn. De ontwikkelaars beweren dat Erlang een betrouwbaarheid heeft van 'nine nines' (99,9999999%) [45]. Dit betekent concreet dat Erlang programma's -indien correct geschreven- gemiddeld slechts 31 ms per jaar niet beschikbaar zijn. Erlang is in staat om zowel fouten in de software als in de hardware op te vangen. Deze grote betrouwbaarheid wordt gerealiseerd doordat boven elk proces in Erlang nog een supervisor draait die bij een crash van het onderliggend proces een nieuw proces opstart en zo het programma in gang houdt. Bovendien is Erlang een mature functionele taal, wat de stabiliteit ervan verder ten goede komt.
Erlang maakt gebruik van eager evaluation in tegenstelling tot Haskell.
Erlang is eveneens zeer goed in het draaien op gedistribueerde systemen. Dit komt doordat in Erlang veel functies hun eigen proces hebben. Deze processen kunnen makkelijk op verschillende processors draaien. Elk proces heeft een overhead van slechts 500 bytes. Bovendien draaien Erlang processen volledig onafhankelijk van het OS, waardoor Erlang programma's op exact dezelfde manier werken op een Windows machine als een Mac of Linux distributie.
Nog een belangrijke eigenschap van Erlang die de betrouwbaarheid ervan benadrukt is 'hot code replacement'. Dit wil zeggen dat een nieuwe versie van een programma kan worden geüpload terwijl de oude versie nog aan het draaien is. Erlang zal dan op een geschikt moment geleidelijk aan overschakelen op deze nieuwe versie zonder dat het programma ook maar één moment stilvalt. Dit is één van de functionaliteiten die in OTP ingebouwd zijn.

Erlang is speciaal ontworpen zodat het makkelijk aan te leren is, wat de overstap naar functioneel programmeren eenvoudiger makt. Bovendien zijn er in Erlang ook behoorlijk wat extra bibliotheken beschikbaar. Dit voegt extra flexibiliteit aan de taal toe waardoor Erlang geschikt is voor een groot aantal uiteenlopende toepassingen.
Erlang kan bovendien ook makkelijk communiceren met andere programmeertalen zoals Java, C, .NET, … Dit kan een groot voordeel zijn in een SOA- en cloud-georiënteerde context.
Erlang is standaard dynamically typed, maar biedt ook de mogelijkheid om van static typing gebruik te maken. Hiervoor is de Dializer plugin ontwikkeld. Deze voert type checking uit bij de compilatie. De keuze tussen static en dynamic typing aan de programmeur overlaten is op zich een mooie toevoeging, tenminste zolang de code doorheen grote projecten consistent en overzichtelijk blijft.
Een laatste groot pluspunt van Erlang, dat voortkomt uit het originele toepassingsdomein ervan, zijnde telecommunicatie, is dat omgaan met binaire data zeer eenvoudig is in Erlang. Hier zijn speciale bibliotheken voor ontwikkeld die out-of-the-box te gebruiken zijn.

2.2.5.3 F#

F# is een zeer veelzijdige taal die meerdere programmeerparadigma's ondersteunt, maar de nadruk legt op het functionele paradigma. De taal is ontwikkeld door zowel Microsoft als verschillende externe organisaties. F# is volledig ondersteund in Microsofts Visual Studio IDE en integreert eenvoudig met Microsofts .NET framework [46]. Het is echter ook mogelijk om F# stand-alone te ontwikkelen en gebruiken. De taal bevat invloeden van C#, Python, Haskell, Scala, en Erlang. Microsoft ziet F# dan ook als de culminatie van de voordelen die al deze talen te bieden hebben.
F# is zeer flexibel. Het ondersteunt alle belangrijke aspecten van functioneel programmeren, en daarnaast ook imperatieve en object-georiënteerde eigenschappen. Dit maakt F# tot een zeer aantrekkelijke optie voor programmeurs die graag van de voordelen van een functionele taal gebruik maken, maar toch nog vast willen kunnen houden aan vertrouwde object-georiënteerde principes waar zij dat wensen. Omdat F# volledig integreerbaar is in het .NET framework, is het ook evident om te switchen tussen F# en C# binnen hetzelfde programma.
F# is strongly en statically typed en maakt gebruik van type-inferentie, net als Haskell. In de regel moet de programmeur zelden zelf types declareren, maar hij mag dat wel. De compiler infereert zelf de rest. In tegenstelling tot Haskell maakt F# wel gebruik van eager evaluation.
F# kan op zowel Windows als Unix-based systemen draaien. Het kan eveneens rechtstreeks in de Azure cloud worden gedeployed. Dit maakt F# opnieuw tot een zeer programmeur-vriendelijke optie om functioneel te programmeren. In [47] wordt bovendien aangetoond dat F# zich goed leent tot gedistribueerde cloud-applicaties. Er bestaan zelfs libraries zodat concurrency en multi-core ondersteuning voor F# geoptimaliseerd kunnen worden. Al deze zaken maken van F# een zeer interessante taal om verder te bestuderen voor dit onderzoek.

2.2.5.4 Dart

Dart is een heel recente taal, en is ontworpen in opdracht van Google. Google verzorgt bovendien de verdere ontwikkeling van de taal. Enerzijds is Dart een object-georiënteerde taal, gebaseerd op concepten als klassen en enkelvormige erving [48], [49], maar anderzijds bevat het ook heel wat functionele eigenschappen. Het is een uitgebreide en veelzijdige taal en wordt voornamelijk gebruikt voor het ontwikkelen van web- en serverapplicaties, mobiele applicaties en Internet of Things devices.

Dart is gericht op het ontwerpen van makkelijk schrijfbare toepassingen met het oog op de huidige, moderne manier van app-ontwikkeling. Als nieuwe taal wil Dart vooral in dit gebied sterk overkomen ten opzichte van andere talen zoals JavaScript. Een van de belangrijke doelen van Dart is een stevige basis neerleggen door middel van uitgebreide libraries. Tevens willen de ontwikkelaars van de taal het programmeren zo eenvoudig mogelijk maken door te focussen op het voorkomen van bekende problemen en fouten. Bij Dart ligt de focus bij stabiele en betrouwbare apps, die programmeurs eenduidig en zonder verrassingen kunnen maken.

Een ander belangrijk aspect voor programmeertalen van vandaag is het maken van zeer schaalbare toepassingen. Het is dan ook geen toeval dat Dart hierin uitblinkt. Een mooi voorbeeld hiervan is Google Adwords, wat volledig in Dart geschreven is. Dit is de grote reclameservice en tevens de grootste inkomstenbron van Google, en wordt wereldwijd gebruikt door webpagina's om reclame te maken. Adwords heeft dus nood aan een enorm grote schaalbaarheid, die Dart zonder problemen kan voorzien.

Een ander sterk punt van Dart is dat het een geschikte taal is om webapplicaties mee te ontwerpen. Hier ligt de nadruk op client-side toepassingen voor de webbrowsers van mobiele apparaten. Initieel planden de ontwikkelaars bij Dart en Google om de taal te integreren in Google Chrome, en later andere browsers. Uiteindelijk werd dit idee geschrapt en is de focus verschoven naar de mogelijkheid om naar Javascript te compileren. Hierdoor kunnen toepassingen die in Dart geschreven zijn, gehercompileerd worden naar Javascript. Deze taal wordt wel door alle grote browsers ondersteund.

Ook biedt Dart enkele slimmere oplossingen aan in vergelijking met Javascript. In Dart is het mogelijk om in HTML te werken met listeners die naar functies verwijzen, terwijl er voor Javascript zelf in HTML naar de functies verwezen moeten worden. Dit zorgt ervoor dat de Dart code gescheiden kan worden van het HTML-gedeelte, wat zorgt voor een betere scheiding van de code en een betere overzichtelijkheid. Ook heeft Dart een standaard bibliotheek die de essentiële elementen van jQuery bevat. jQuery is een populaire bibliotheek waarvan Javascript gebruik maakt voor animaties, event handlers, query's te behandelen… In tegenstelling tot Javascript beschikt Dart standaard over deze bibliotheek. Het gebruik hiervan is in Dart bovendien eenvoudiger door kortere syntax.

Dart code is altijd single threaded, maar biedt ondersteuning voor concurrency. Dit wordt mogelijk gemaakt door zogenaamde Isolates die gebaseerd zijn op het actormodel van Erlang. Deze Isolates zijn volledig onafhankelijk van elkaar, hebben elk hun eigen geheugen en maken concurrency mogelijk door het doorgeven van berichten. Het voordeel hiervan is

dat als één van de isolates faalt er een nieuwe isolate kan opgestart worden die dezelfde functie kan overnemen zonder dat er crashes zijn.
De ontwikkelaars van Dart hebben gekozen voor een optionally typed taal. De programmeur kan zelf kiezen of hij types wilt declareren of niet. De programmeur weet zelf vaak welk type er nodig is in tegenstelling tot de typechecker. Zonder type annotaties loopt het ontwikkelen van de code zelf vlotter, maar wanneer er iets misgaat kunnen fouten moeilijker te vinden zijn.
Zoals reeds vermeld is Dart in de eerste plaats een object-georiënteerde taal. Alles kan omschreven worden als een object. Alle variabelen waaronder nummers, functies en null zijn gedefinieerd als objecten in Dart. Hierbij is elk object ook een instantie van een klasse, maar het is ook mogelijk om toepassingen te schrijven zonder enige klasse te definiëren indien de programmeur dat wenst.
Dart is een veelzijdige taal die niet alleen gericht is op imperatief programmeren. De taal is in het bezit van de nodige functionele aspecten zoals het mogelijke gebruik van de functies zelf. Functies kunnen als argumenten doorgegeven worden en een functie kan als variabele toegewezen worden. Ook is er ondersteuning voor lambda-expressies aanwezig en data kan onveranderlijk gemaakt worden. Kortom, Dart beschikt over een groot aantal eigenschappen die een functionele programmeerstijl toelaten.
Een laatste punt is de integratie van de Cloud in Dart. Dart heeft hier een voorsprong op andere talen met specifieke libraries die ontwikkeld zijn voor de cloud. Hierbij is het Google Cloud Platform nauw verbonden met Dart wat een groot voordeel oplevert. Het Google cloud platform en Dart zijn beide ontworpen met schaalbaarheid als een van de prioriteiten, dit geeft de mogelijkheid tot het bouwen van zeer grote toepassingen. Ook ondersteunt de Google App Engine Dart volledig. Hiernaast kan Dart code op de client-side en de server-side van de cloud geïmplementeerd worden wat helpt een uniformere cloud te bekomen. Naast het Google Cloud Platform ondersteunen ook andere cloud service providers zoals SourceVoid Dart.

2.2.5.5 Wolfram Language

Wolfram is een recente programmeertaal die meerdere paradigma's ondersteunt. De focus van Wolfram ligt op functioneel programmeren en logisch programmeren. Dit laatste is een programmeerparadigma dat een programma voorstelt als een set logische uitdrukkingen en regels [50]. De officiële Wolfram Language Reference is online te vinden op [51]. Deze is zeer compleet en vormt dan ook de basis van wat volgt.
De Wolfram programmeertaal is gebaseerd op Mathematica. Dit is een taal die al enkele decennia bestaat, en die als onderdeel van het programma Mathematica vooral toegepast wordt door wetenschappers en wiskundigen voor het analyseren van data en genereren van complexe modellen. De Wolfram programmeertaal combineert de aspecten van mathematica met die van de Wolfram|Alpha zoekmachine, die door hetzelfde moederbedrijf ontwikkeld is. Dit is een bekende zoekmachine die in tegenstelling tot Google e.d. vanuit verschillende grote databanken informatie verzamelt en die samen giet tot één enkel antwoord met alle

relevante informatie in, in plaats van een lijst met links terug te geven naar websites waar de nuttige informatie op staat.
Door de combinatie van eigenschappen van Mathematica en Wolfram|Alpha is Wolfram uitgegroeid tot een heel veelzijdige en flexibele taal, met tal van unieke mogelijkheden.
Omdat Wolfram een hoofdzakelijk functionele taal is, is Wolframcode zeer schaalbaar. De code is ook zeer compact.
Eén van de basisconcepten van Wolfram is dat het programmeerproces zo veel mogelijk geautomatiseerd moet verlopen. De ontwikkelaars van Wolfram willen de programmeur zo veel mogelijk bijstaan zodat deze laatste zich kan focussen op wat zijn code moet doen, en niet op hoe hij zijn code zodanig kan formuleren dat de machine het begrijpt en efficiënt kan uitvoeren. Het Wolfram-framework zal in de mate van het mogelijke zelf de code van de programmeur proberen te interpreteren en uit te voeren.
De taal focust ook op leesbaarheid en begrijpbaarheid van de code. Code wordt geschreven in zogenaamde Computable Document Format (CDF) bestanden. Deze kunnen code, gestructureerde commentaar, en zelfs afbeeldingen bevatten. Hierdoor is Wolfram-code aangenaam en vlot leesbaar en bovendien overzichtelijk.
De grote automatie die Wolfram biedt, is ook doorgetrokken naar het build- en deploy-proces. Als de programmeur dit wenst, hoeft hij zeer weinig tijd te besteden aan het builden en deployen van code. Wolfram biedt wel nog de vrijheid om een volledig gepersonaliseerd build- en deploy-proces uit te werken.
Wolfram is een zeer symbolische taal. Hierdoor behandelt Wolfram code net als data, waardoor de mogelijkheid ontstaat om automatisch code te genereren, en code te transformeren. Bovendien is eender welke Wolfram-code zeer snel en eenvoudig transformeerbaar tot een web-API. Dit is een belangrijk voordeel naar SOA en cloud computing toe.
Wolfram code is ook makkelijk aan te roepen vanuit andere services. Dankzij het Wolfram Symbolic Transfer Protocol (WSTP) kunnen Wolfram-programma's op een gestructureerde manier rechtstreeks communiceren met programma's in verschillende andere talen. Hierdoor is het makkelijk om Wolfram-code te gebruiken in combinatie met andere programmeertalen. Dit is zeker een pluspunt voor de taal aangezien functionele talen vaak zeer goed zijn in specifieke taken, maar minder scoren dan imperatieve talen voor andere toepassingen, en ook omdat Wolfram op zich geen bekende taal is. Door interoperabiliteit te maximaliseren, blijft Wolfram toch aantrekkelijk. Bovendien is Wolfram-code makkelijk om te zetten naar C-code, indien gewenst. Wolfram is bovendien eenvoudig te integreren in webpagina's.
De Wolfram-taal beschrijft alles als 'expressies'. 'Alles' is in dit geval zeer letterlijk te interpreteren. Zo heeft een wiskundige operator bijvoorbeeld exact dezelfde notatie als een driehoek. Op deze manier ontstaat een uniforme notatie die de taal zeer flexibel maakt. Dit impliceert ook dat een functie werkelijk eender wat als input kan krijgen, en dus op eender wat kan worden toegepast. Hoewel dit de flexibiliteit van de taal sterk ten goede komt, kunnen er problemen ontstaan bij het debuggen van complexe Wolfram-code, aangezien static typing zoals in Java niet bestaat in Wolfram. Dit kan voor verwarring zorgen.

Wolfram neemt het begrip 'dynamisch' nog een stap verder dan het typesysteem. Wolfram is namelijk geen gecompileerde taal, maar een geïnterpreteerde. Meer specifiek maakt Wolfram gebruik van een Just In Time (JIT) compiler die de code tijdens de uitvoering zelf omzet naar machine-instructies. Dit heeft als voordeel dat elke machine die beschikt over een Wolfram runtime environment eender welke Wolfram code rechtstreeks kan uitvoeren. Bovendien kan bij Wolfram net als bij Erlang code worden uitgewisseld tijdens de uitvoering van het programma zelf. Het nadeel is dat geïnterpreteerde talen in het algemeen trager zijn dan hun gecompileerde tegenhangers [52].
Ten slotte is Wolfram van de grond op geschreven met cloud computing in gedachten. Wolfram biedt zelfs een PaaS cloud service aan waar Wolfram code zeer eenvoudig naar gedeployed kan worden, namelijk de Wolfram Cloud. Iedereen die in Wolfram programmeert heeft toegang tot deze cloud. Ook kan code rechtstreeks in de cloud geschreven en opgeslagen worden. Dit alles is gratis voor beperkt gebruik. Voor grote toepassingen kan ook een betalende subscriptie genomen worden. Deze biedt ook de mogelijkheid om een desktop-IDE te gebruiken.

2.2.5.6 Imperatieve programmeertalen met functionele eigenschappen

Ook de grootste klassieke object georiënteerde programmeertalen zoals C(++), C#, Python, Java en JavaScript zijn de laatste jaren de functionele boot gesprongen en bieden steeds meer ondersteuning voor functioneel/declaratief programmeren. Dit omdat applicaties steeds groter en complexer worden en functionele talen uitblinken op gebieden waar een object-georiënteerde aanpak tekort schiet. In principe zijn al deze talen nu dan ook hybride-talen, aangezien ze meerdere programmeerparadigma's ondersteunen.
De implementatie van functioneel programmeren in deze klassieke talen gebeurt veelal aan de hand van lambda-expressies. Dit is een compacte notatie die toelaat functies als eerste-orde objecten te beschouwen. Dit kan binnen het object-georiënteerde framework. Binnen de lambda-expressie zelf gedragen variabelen zich volgens de principes van functioneel programmeren, en erbuiten kan er toch dynamisch worden omgesprongen met variabelen op de imperatieve manier. Het spreekt voor zich dat dit ook grote flexibiliteit met zich meebrengt.

2.2.6 Besluit

Functioneel programmeren is een programmeerparadigma dat radicaal verschilt van het de dag van vandaag alomtegenwoordige object-georiënteerd programmeren. Deze radicaal verschillende benadering van software brengt een heel aantal belangrijke voordelen met zich mee. In veel van deze voordelen schuilt echter ook een nadeel naar bepaalde toepassingen toe. De werkelijke kracht van functioneel programmeren komt dan ook het best tot zijn recht in specifieke situaties. Hiertoe leent de gedistribueerde aanpak van cloud computing zich zeer goed. Zeker in combinatie met SOA is het mogelijk applicaties zodanig te ontwerpen dat de voordelen van functioneel programmeren ten volle tot hun recht komen, zonder de nadelen ervan te confronteren. De aspecten van de applicatie waar functionele talen niet goed in scoren kunnen eenvoudig worden overgelaten aan klassieke imperatieve talen. Juist

daarin ligt de grote bijdrage die deze masterproef kan leveren aan het gebruik van functioneel programmeren in de cloud.
Er is tegenwoordig een groot aanbod aan functionele talen op de markt, aangevuld door imperatieve talen met functionele eigenschappen. Elk van deze talen heeft zijn eigen karakter en voor- en nadelen.
Haskell en Erlang zijn twee vooraanstaande functionele talen. Deze talen zijn echter al relatief oud, en cloud-ondersteuning is niet eenvoudig verkrijgbaar in het handige PaaS-formaat. Om deze talen in een cloud omgeving te gebruiken is dus het gebruik van een IaaS cloud model vereist. Dit heeft zo zijn voordelen, maar ook duidelijke nadelen waardoor snel software schrijven in deze talen voor een cloud-omgeving waarschijnlijk minder evident is. Bovendien zijn deze talen -in het bijzonder Haskell- al veelvuldig bestudeerd. Daarom zijn ze minder interessant voor dit onderzoek.
F#, Dart, en Wolfram zijn daarentegen alle drie moderne programmeertalen die in de laatste jaren veel aan populariteit gewonnen hebben, maar desondanks nog vrij onbekend zijn. Er is niet overdreven veel onderzoek gedaan naar deze talen, en cloud-ondersteuning ziet er veelbelovend uit voor elk van hen. Daarom zijn deze talen zeer interessant voor dit onderzoek. Omdat drie nieuwe functionele programmeertalen in detail bespreken en vergelijken een zeer omvangrijke opdracht is, zijn voor dit onderzoek de talen Wolfram en F# gekozen. Deze keuze is louter om timing-redenen genomen en betekent zeker niet dat Dart minder interessant is.
De keuze voor F# enerzijds en Wolfram anderzijds is genomen aangezien F# een bij .NET-ontwikkelaars al vrij bekende taal is die veel ondersteuning heeft binnen het .NET-framework. Wolfram langs de andere kant is een volledig gescheiden technologie die niet verweven is met klassieke programmeerframeworks. Deze combinatie van talen dekt dus een maximaal bereik binnen de wereld van functionele programmeertalen.
Naast de hoofdzakelijk functionele talen is er ook gekozen om in dit onderzoek enkele klassieke talen met functionele eigenschappen op te nemen. Zo is het niet alleen eenvoudig om de klassieke talen makkelijk te vergelijken met de hierboven vermelde functionele talen, maar kan er ook rechtstreeks een vergelijking worden gemaakt tussen functionele en klassieke implementaties binnen deze klassieke programmeertalen.
De keuze voor de klassieke talen is voor dit onderzoek gevallen op Java en C#. Dit omdat beide deze talen industrie-standaarden zijn die op grote schaal toegepast worden voor allerlei applicaties. Bovendien zijn de C#/.NET-wereld en de Java-wereld grotendeels van elkaar gescheiden, en hebben ze elk hun eigen aanpak voor veel problemen. Doordat Java en C# allebei in essentie object-georiënteerde talen zijn, komt deze verschillende aanpak vaak wel op hetzelfde neer, hetzij met enkele nuanceverschillen. Het is dus interessant om te kijken in hoeverre dit zo is voor cloud computing.
Voor zowel Java als C# is het dus interessant om zowel object-georiënterde als functionele implementaties naast elkaar te zetten in de cloud. Daarbovenop is een vergelijking van deze implementaties met de zuiver functionele implementaties in Wolfram en F# zeer interessant.

2.3 Hulpmiddelen

Deze laatste sectie vergelijkt de verschillende mogelijkheden voor het vergelijken van programmeertalen met elkaar. Het is namelijk belangrijk om een idee te hebben van de manier waarop programmeertalen in de literatuur met elkaar vergeleken worden, zodanig dat in dit onderzoek de vergelijking tussen de verschillende talen in de cloud zo uniform en ondubbelzinnig mogelijk kan gebeuren.

2.3.1 Analyse van gebruikelijke technieken om talen te vergelijken

Tot op de dag van vandaag bestaan er geen gestandaardiseerde methodes om programmeertalen te vergelijken. Programmeertalen omvatten namelijk heel wat verschillende aspecten. Veel talen kiezen ervoor om te focussen op bepaalde van die aspecten, en minder op andere. Hierdoor is het kwantitatief vergelijken van bepaalde aspecten van verschillende programmeertalen moeilijk, aangezien de ene taal sterk geoptimaliseerd kan zijn voor het geteste aspect, terwijl de andere taal dat helemaal niet is.
Een gangbare methode om de performantie van verschillende programmeertalen te vergelijken is het maken van een relatief eenvoudig testprogramma dat makkelijk schaalbaar is en veel rekenwerk vraagt. Een dergelijk programma kan dan worden geïmplementeerd in verschillende programmeertalen. Die implementatie kan dan op dezelfde machine worden getest voor de verschillende talen. Een vergelijking van de gebruikte tijd, processorbelasting, geheugengebruik, etc. is dan mogelijk. In [53] wordt dit gedaan voor een heel aantal talen.
Dit soort testprogramma's kan wel makkelijk een vertekend beeld geven. Vaak kan een bepaalde taal in één testprogramma het veel beter doen dan de andere, terwijl in een ander programma geen verschil merkbaar is. Dit hangt af van de exacte inhoud van het testprogramma. Bovendien verschillen deze testprogramma's vaak sterk van real-world toepassingen, wat het extrapoleren van performantie bevindingen aan de hand van testen nog ingewikkelder maakt. Het expliciet kwantitatief vergelijken van dit soort aspecten moet dus met een flinke korrel zout genomen worden.
Anderzijds hebben verschillende projecten verschillende eisenpakketten, en is het wel belangrijk een notie te hebben van de aspecten waar bepaalde talen goed in scoren, en de aspecten waarin ze onderdoen voor de competitie. Dit laat een goede geïnformeerde keuze toe voor een bepaalde programmeertaal om een bepaald project of deel van een project in uit te werken. Een kwalitatieve beoordeling van de capaciteiten van verschillende programmeertalen is dus een grotere meerwaarde dan exacte cijfers. Het is wel mogelijk om bepaalde criteria voorop te stellen waarop elke taal afzonderlijk kwalitatief beoordeeld kan worden. Dit is voor bepaalde talen reeds behandeld door de auteurs van [54] en [55]. Dit onderzoekt zal ook eerder een dergelijke kwalitatieve beoordeling nastreven.
Hieronder een opsomming van enkele belangrijke te evalueren criteria, gebaseerd op onder andere [54] en [55]:

- **Leesbaarheid van de code**: Dit draagt bij tot het beter en sneller begrijpen van de code door programmeurs. Dit laat een efficiëntere en betere samenwerking toe tussen programmeurs onderling;
- **Ontwikkelingstijd**: Een applicatie die op korte tijd geprogrammeerd kan worden is voordeliger voor bedrijven. Dit laat niet enkel grotere flexibiliteit toe met betrekking tot deadlines, maar de werkdruk en loonkost dalen ook;
- **Lengte van de code**: Software geschreven met meer regels code vergt meer tijd van de programmeur. Minder code kan een voordeel zijn, maar dit mag de leesbaarheid niet in het gedrang brengen;
- **Betrouwbaarheid en stabiliteit**: Een taal moet zo optimaal mogelijk blijven functioneren onder verschillende omstandigheden. Wanneer er updates van de taal zijn bestaat de mogelijkheid ook dat reeds bestaande code niet meer functioneert. Dit moet zoveel mogelijk voorkomen worden;
- **Parallellisatie**: Voor het maken van toepassingen die grote hoeveelheden data moeten verwerken is een taal vereist die zich makkelijk laat parallelliseren. Zo niet, kunnen er nadelige effecten zijn in verband met performantie en ontwerpkeuzes, of is het zelfs niet mogelijk om een dergelijke rekenintensieve toepassing te maken;
- **Schaalbaarheid**: Voor cloud-toepassingen is een schaalbare taal een groot pluspunt. Indien de vraag voor de applicatie stijgt, moet deze immers makkelijk kunnen schalen zodat alle aanvragen binnen een acceptabele tijd afgehandeld zijn.
- **Geheugengebruik**: Hoe minder geheugengebruik, hoe beter. Een taal moet zo efficiënt mogelijk omgaan met beschikbare resources. Als een toepassing kan draaien met zo min mogelijk last voor de hardware is dit voordeliger;
- **Beveiliging**: Beveiliging van gegevens is cruciaal voor bedrijven. Dit geldt zeker in een cloud-omgeving waar er enorme hoeveelheden data opgeslagen zijn;
- **Uitvoeringssnelheid**: Een taal die in vergelijking met andere sneller is, heeft een voordeel. Dit resulteert in een betere gebruikerservaring, maar kan ook kosten drukken aangezien de hardware minder belast wordt;
- **Ondersteuning voor talen**: Dit is eveneens een belangrijk aspect omdat dit de mogelijkheden kan beperken waarvoor een taal gebruikt kan worden. De aanwezigheid van bepaalde bibliotheken of de ondersteuning van een cloud provider voor een bepaalde taal kan veel voordelen opleveren;
- **Foutenlast**: Programmeertalen waarin het minder makkelijk is om fouten te maken, en het gemakkelijker is om fouten op te sporen, hebben een groot voordeel. Zeker voor complexe toepassingen en beginnende programmeurs is foutenlast een doorslaggevende factor in de keuze van een programmeertaal;
- **Gemak voor de programmeur**: Programmeurs die nog geen kennis hebben van potentieel interessante talen, hebben vaak moeite met het overschakelen en leren van een andere taal. Een taal met een kleine leercurve of vertrouwde concepten zal hierdoor ook meer aan populariteit winnen. Ook documentatie en een actieve community kunnen sterk bijdragen aan het gemak voor de programmeur bij het gebruik van een bepaalde programmeertaal.

Er zijn dus heel wat criteria waarop programmeertalen kwalitatief te vergelijken zijn. Dit onderzoek neemt zo veel mogelijk van deze criteria op. Vooral de criteria parallellisatie en schaalbaarheid zijn voor deze masterproef interessant, aangezien dit ook aspecten zijn waar cloud computing in uitblinkt. Het is dus interessant om te zien in hoeverre de bestudeerde talen op de gedistribueerde aard van cloud computing kunnen inspelen.

2.4 Besluit

Cloud computing is een breed begrip dat vele toepassingen heeft. De voordelen van cloud computing zijn talrijk. Voor deze masterproef zijn public clouds de aantrekkelijkste. Er zijn een groot aantal providers beschikbaar die uiteenlopende public cloud services aanbieden. Functioneel programmeren biedt evenzeer grote voordelen, hoewel er ook nadelen aan verbonden zijn die voor specifieke applicaties een significante impact hebben. Juist daarom is een combinatie van functioneel programmeren en cloud computing bijzonder aantrekkelijk, aangezien door gebruik te maken van een SOA-architectuur in de cloud het gebruik van functionele talen kan beperkt worden tot de aspecten waarin het uitblinkt, terwijl voor andere aspecten imperatieve talen toegepast kunnen worden, en dit alles binnen dezelfde applicatie. Er zijn een groot aantal functionele talen op de markt de dag van vandaag. In paragraaf 2.2.6 is reeds besloten dat voor dit onderzoek de hoofdzakelijk functionele talen F# en Wolfram het aantrekkelijkst zijn, alsook de imperatieve talen Java en C#, die functionele eigenschappen bezitten. Aan F# en Wolfram is verder in deze thesis een apart hoofdstuk met een gedetailleerde studie van die talen gewijd. Voor Java en C# is er een toelichting van hun functionele eigenschappen voorzien.
De meest aantrekkelijke cloud service providers voor deze masterproef zijn zoals in paragraaf 2.1.9.7 reeds besloten Amazon, Google, en Microsoft. In die paragraaf werd ook reeds aangehaald dat de uiteindelijke keuze afhankelijk is van de gekozen programmeertalen. Aangezien deze voor dit onderzoek Java, C#, F#, en Wolfram zijn, is gekozen om licht af te wijken van deze shortlist. Aangezien Java eigendom is van Oracle, en paragraaf 2.1.9.5 reeds aangaf dat Oracle Cloud ook een gratis proefperiode aanbiedt, is dit Cloud platform zeer interessant om uit te proberen in combinatie met Java, ondanks het feit dat dit cloud-platform niet tot de shortlist uit paragraaf 2.1.9.7 behoort. Langs de andere kant is Microsoft Azure gekozen als cloud platform voor C# en F#, aangezien Microsoft zelf Azure beheert en volledige ondersteuning biedt voor het .NET framework, wat eveneens eigendom is van Microsoft, en waar zowel C# als F# deel van uitmaken. Verder is voor Wolfram geopteerd om de Wolfram Cloud te gebruiken, aangezien deze ingebouwd is in de taal en er zeer veelbelovend uitziet naar implementatiegemak toe. Zo is er voor elke taal een cloud-platform gekozen dat rechtstreeks door de ontwikkelaars van de taal beheerd wordt. Daarnaast is het uiteraard ook interessant om een neutralere speler te testen als cloud-platform. Daarom is Google Cloud Platform ook gekozen voor dit onderzoek. Het is interessant om te bestuderen welke voordelen het heeft om voor de verschillende geteste programmeertalen bij de 'native' cloud-platformen te blijven, of juist naar een externe provider te gaan. In een later stadium bespreekt deze thesis al deze cloud-platformen in meer detail.

Net als bij de keuze van de programmeertalen heeft deze masterproef het interessante Amazon Web Services-cloud platform wegens timing-redenen niet opgenomen. Dit betekent niet dat Amazon een minder interessante CSP is. Verder onderzoek is nodig om ook dit platform uit te testen voor verschillende programmeertalen.
De gekozen programmeertalen worden op een kwalitatieve manier met elkaar vergeleken in de cloud. Hiervoor zijn in paragraaf 2.3.1 een heel aantal criteria afgebakend. Deze masterproef beoogt zo veel mogelijk van deze criteria op een kwalitatieve manier te beoordelen, om zo voor elke taal de sterktes en zwaktes te identificeren. De klemtoon van deze vergelijking ligt op functionele versus object-georiënteerde talen. Daarom is het van belang parallellen te vinden tussen de verschillende functionele talen in de cloud, en contrasten tussen deze functionele talen en de object-georiënteerde implementaties.

3 Functionele eigenschappen van Java en C#

Zoals in vorig hoofdstuk reeds aangehaald is het van belang elk van de gekozen te bestuderen functionele talen voor dit onderzoek eerst algemeen te bespreken, om een goed beeld te krijgen van de manier waarop functioneel programmeren deze talen beïnvloedt, en wat de beste manier is om in een functionele stijl applicaties te ontwikkelen in deze talen, alvorens het mogelijk is te beginnen met deze applicaties naar de cloud te brengen.
Zoals in paragraaf 2.2.6 reeds beschreven zijn een aantal object-georiënteerde talen met functionele eigenschappen gekozen om te bestuderen in dit onderzoek; met name Java en C#. Deze talen zijn in het bijzonder interessant omdat ze het mogelijk maken functionele en object-georiënteerde implementaties binnen eenzelfde taal rechtstreeks te vergelijken. Dit hoofdstuk beoogt bondig een overzicht te scheppen van de belangrijkste functionele eigenschappen van elk van deze programmeertalen.

3.1 Java

Sinds Java 8, wat uitgekomen is in 2014, biedt Java ondersteuning voor een aantal functionele programmeerprincipes. Hieronder een opsomming van de belangrijkste functionele eigenschappen die deze taal sindsdien bezit.

3.1.1 Anonieme functies

De basis van functioneel programmeren is het concept van anonieme functies, ofwel λ-expressies, zoals reeds beschreven in paragraaf 2.2.1. Ook Java ondersteunt deze. De syntax hiervoor is eenvoudig en komt in veel functionele programmeertalen terug. Figuur 10 geeft een voorbeeld van een eenvoudige λ-expressie in Java.

```
x -> x*2
```

Figuur 10: Lambda-expressie in Java.

Bovenstaande λ-expressie neemt een getal x als input en geeft als output het dubbele van x. Merk op dat nergens types zijn aangegeven. Bij Lambda-expressies past Java immers type-inferentie toe, zoals veel andere functionele programmeertalen.
Een anonieme functie op zich heeft uiteraard niet veel nut. Er is nood aan een manier om deze functie toe te kunnen passen op input. Hiervoor dient de functie in klassieke Java-stijl aan een variabele gebonden te zijn. Functies zijn dus sinds Java 8 ook eerste orde-elementen en Java ziet ze gewoon als eender welke andere variabele. In Java is een λ-expressie van het type Function. De input en output van de λ-expressie dienen bij de definitie van de functie meegegeven te worden [56]. Figuur 11 geeft dit principe weer toegepast op de λ-expressie uit Figuur 10.

```
Function<Integer,Integer> timesTwo = x -> x*2;
```

Figuur 11: Lambda-expressie gekoppeld aan variabele.

Nu is het mogelijk de λ-expressie toe te passen op input. Dit kan door gebruik te maken van de *apply*-methode die de klasse *Function* bevat. Figuur 12 geeft het resultaat hiervan weer voor de functie uit Figuur 11 toegepast op het getal 5.

```
int doubledValue = timesTwo.apply(5);
```

Figuur 12: Toegepaste λ-expressie als eerste-orde functie.

De output van deze functie is 10, zoals verwacht. Figuur 13 geeft dit weer.

```
run:
10
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figuur 13: Resultaat van het uitvoeren van de code uit Figuur 12.

Soms is het uiteraard wenselijk om meerdere regels code te gebruiken voor de definitie van een functie. Dit kan eenvoudig met λ-expressies in Java. Het volstaat achter de -> een set accolades te plaatsen. Binnen de accolades kan dan code ingevuld worden zoals bij eender welke andere methode in Java. De parameters van de λ-expressie gedragen zich als methodeparameters binnen de accolades. Figuur 14 geeft weer hoe dit in zijn werk gaat.

```
Function<Integer,Integer> timesTwo = x -> {
    int y = x*2;
    return 2;
};
```

Figuur 14: λ-expressie van meerdere regels code in Java.

Het is ook mogelijk om eerste-orde functies in Java 8 toe te kennen zonder gebruik te maken van λ-expressies. Hiervoor bestaat een speciale syntax die rechtstreeks verwijst naar een statische methode uit een Java-klasse. Beschouw het voorbeeld uit Figuur 15.

```
static class Multiply{
    public static int multiplyByTwo(int x){
        return x*2;
    }
}
```

Figuur 15: Statische klasse met eenvoudige methode in Java.

In dit voorbeeld is rechtstreeks in de main-klasse een inwendige statische klasse *Multiply* aangemaakt, met daarin de statische methode *multiplyByTwo*, volgens de gewone Java-conventies. Deze methode neemt een enkele int als parameter en geeft een int waarde terug die het dubbele is van de input, net zoals de hierboven besproken Lambda-expressie.
Nu is het mogelijk een eerste-orde functie te definiëren en die rechtstreeks te binden aan de MultiplyByTwo-methode uit Figuur 15. Figuur 16 geeft dit weer.

```
Function<Integer,Integer> timesTwoAlternative = Multiply::multiplyByTwo;
```

Figuur 16: Alternatieve manier om een eerste orde-functie te declareren in Java.

Deze alternatieve syntax is zoals hierboven te zien zeer compact. Dit komt de leesbaarheid sterk ten goede. Er is enkel weergegeven wat van belang is, namelijk dat timesTwoAlternative een functie is die een Integer als parameter heeft en een Integer als output teruggeeft, door het toepassen van de multiplyByTwo-methode uit de Multiply-klasse op de input. Java geeft zelf de argumenten op de correcte manier door. De werking van deze functie is identiek aan die uit Figuur 11.
Een laatste opmerking bij anonieme functies is dat functies die een boolean teruggeven, compact kunnen worden gedeclareerd als een Predicate. Zo is het niet nodig steeds het return-type te vermelden van dit speciale soort functies. De methode om een Predicate toe te passen op een input heet test in plaats van apply. Verder gedragen Predicates zich op juist dezelfde manier als functies met eender welk ander return-type. Figuur 17 geeft een voorbeeld van een Predicate in Java.

```
Predicate<Integer> isPositive = x->x>0;
```

Figuur 17: Predicate in Java.

Bovenstaande Predicate controleert of de input x een positief getal is. De iets compactere en elegantere notatie van Predicates is duidelijk. Deze predicates komen goed van pas in if-lussen of in bepaalde andere functies, die later in dit hoofdstuk toegelicht zijn.

3.1.2 Hogere-orde-functies

In vorige paragraaf is reeds aangehaald dat functies in Java 8 eerste-orde-elementen zijn. Dit betekent dat andere functies deze functies ook kunnen consumeren als input. Java 8 ondersteunt dus ook hogere-orde-functies. Het is immers mogelijk om in Java een functie te definiëren die als parameter een andere functie heeft. Figuur 18 geeft hier een voorbeeld van.

```
public static int applyToDouble(Function<Integer,Integer> f, int x){
    return f.apply(x*2);
}
```

Figuur 18: Voorbeeld van hogere-orde-functie in Java.

De functie applyToDouble past dus de doorgegeven functie f toe op het dubbele van de parameter x. Deze functie kan nu aangeroepen worden met als eerste parameter eender welke functie die een Integer omvormt tot een Integer. Toegepast op de timesTwo-functie uit Figuur 11 levert dit Figuur 19.

```
int quad = applyToDouble(timesTwo, 10);
```

Figuur 19: Hogere-orde-functie in Java.

Het resultaat van deze functie is 40, zoals Figuur 20 laat zien.

```
run:
40
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figuur 20: Resultaat van hetv uitvoeren van de code uit Figuur 19.

Met bovenstaande kennis is het mogelijk werkelijk stijlvolle functionele structuren op te bouwen in Java 8. Het is bijvoorbeeld mogelijk om anonieme functies te definiëren die meerdere parameters tegelijk verwerken door de functies op te bouwen als gedeeltelijke functies die telkens een enkele parameter verwerken. Dit concept heet in functionele talen currying. Dit is het beste te illustreren aan de hand van een voorbeeld. Beschouw hiervoor Figuur 21.

```
Function<Integer,Function<Integer,Integer>> multiply = x -> y -> x*y;
```

Figuur 21: Currying in Java.

De functie multiply is een hogere-orde-functie die een Integer als input neemt, en zelf een functie teruggeeft die een Integer als input neemt en een andere Integer als output teruggeeft. De functie multiply zelf injecteert zijn eigen input x als een deel van de uiteindelijke functie, die zelf nog een input y verwacht. De functie multiply handelt de vermenigvuldiging van de getallen x en y dus gedeeltelijk af, terwijl de anonieme λ-expressie die deze functie teruggeeft de rest van de vermenigvuldiging voor zijn rekening neemt. De vermenigvuldiging van twee getallen met behulp van deze functie ziet er nu uit als in Figuur 22:

```
int res = multiply.apply(10).apply(5);
```

Figuur 22: Uitvoeren van gecurriede functie in Java.

Het resultaat van deze functie is 50. Het spreekt voor zich dat deze vorm van functies opbouwen bijzonder elegant is. Java biedt uiteraard zoals hierboven beschreven ook alternatieve manieren om functies met meerdere argumenten te verwerken. Desondanks is currying een zeer elegant en stijlvol concept. Het feit dat dit mogelijk is in Java is dan ook een aangenaam pluspunt, ook al is de aanwezigheid ervan eerder een gevolg van de andere functionele eigenschappen van de taal dan een rechtstreeks ingebouwd concept.

3.1.3 Streams

Een vooraanstaande toepassing voor de bovenstaande functionele concepten in Java is Streams [57]. Dit is een alternatieve functionele manier om lijsten te bewerken in Java. Een Stream is vrij letterlijk te beschouwen als een 'stroom' van data, waar allerlei bewerkingen op losgelaten kunnen worden. Java voorziet een heel aantal ingebouwde functies die bewerkingen op Streams toelaten. De meeste van deze functies vragen andere functies als parameter. Heel vaak dienen λ-expressies als parameters voor deze Stream-functies.
Er bestaan een aantal soorten Streams in Java. De meest gebruikte vorm is een Stream die rechtstreeks ontstaat door het toepassen van de stream() methode op een Collection-klasse in Java. Het type van de Stream is dan het type van de collectie waarop de Stream-methode is toegepast.
Omdat Java op een speciale manier omgaat met value-types zoals int, double, en long, bestaan er speciale Streams om met deze types te werken, resp. IntStream, DoubleStream, en

LongStream. Deze beschikken over een aantal speciale methodes die niet beschikbaar zijn voor algemene Streams voor andere objecten. Figuur 23 toont 4 verschillende manieren om een stream te genereren van de getallen 1 tot en met 3.

```
//Directly generate Stream
Stream.of(1,2,3);
IntStream.range(1, 4);

//Stream from array
Integer[] ar ={1,2,3};
Arrays.stream(ar);

//Stream from List
List<Integer> l = new ArrayList();
l.add(1); l.add(2); l.add(3);
l.stream
```

Figuur 23: Verschillende manieren om Streams aan te maken in Java.

Mertk op dat het type van de Stream bij alle bovenstaande methodes Integer is, behalve bij de IntStream. Hierbij is het type het value-type int. De methode boxed() kan deze Stream eenvoudig omzetten naar een Stream<Integer>.
Eens een Stream is aangemaakt is het mogelijk een heel aantal methodes erop los te laten die functies als parameter hebben. Hieronder een opsomming van enkele van de belangrijkste van deze methodes, die heel vaak volstaan om complexe lijstbewerkingen toe te passen op Streams:

- **Filter**: Deze methode vraagt een predicaat van het type van de Stream. De functie filtert dan alle elementen uit de stream waarvoor het predicaat geldig is;
- **Map**: Deze functie vraagt een functie die het type van de Stream als parameter heeft en een ander type teruggeeft. De map-functie past deze laatste functie dan toe op alle elementen van de Stream om ze zo te transformeren tot andere elementen;
- **ForEach**: Werkt gelijkaardig aan map, maar vraagt een functie die void als return type heeft. Deze functie 'consumeert' de elementen van de Stream dus;
- **Reduce**: Reduceert de elementen van de Stream tot een enkele waarde, door het toepassen van een bepaalde functie op de elementen. De tussentijdse waarde van de reductie moeten opgeslagen worden in een accumulator. Daarom vraagt reduce een functie die een parameter van het type van de Stream neemt alsook een accumulator en deze omzet naar een bepaald return type. Als dit return type niet overeenkomt met het type van de elementen van de Stream, is er ook nood aan een combinatie-functie die elementen van het type van de Stream kan samenvoegen tot elementen van het return-type van de reduce-operatie. Dit is van belang bij parallelle Streams, waarover later meer.

Bovenstaande functies volstaan in veel gevallen om elegante lijstbewerkingen mogelijk te maken in Java. Er bestaan nog veel meer functies die toepasbaar zijn op Streams. Deze allemaal in detail bespreken zou echter te ver leiden voor deze masterproef. Daarvoor

verwijst dit onderzoek dus door naar de officiële documentatie van Java. Alle Stream-methodes kunnen eenvoudig aan elkaar gekoppeld worden tot een lange ketting van lijstbewerkingen, waardoor een compacte en elegante notatie mogelijk is. Figuur 24 laat een complexe lijstbewerking met Streams zien waarbij al de bovenstaande functies (met uitzondering van forEach) toegepast zijn.

```
public static int determineNextMove(final Disc[][] field, int difficulty){
    return IntStream.range(0, 7)
            .filter(c->field[0][c]!=null)
            .boxed()
            .map(c->new Pair<Integer,Integer>(c,determineBranchValue(
                    moveFieldNoException(field,c,Disc.RED),
                    difficulty, Disc.GREEN)))
            .reduce(new Pair<Integer, Integer>(-1,0), (ca, c)->
                    (c.getValue()> ca.getValue()
                    || (c.getValue()==ca.getValue()&&Math.random()>0.8)
                    || ca.getKey()<0)? c:ca)
            .getKey();
}
```

Figuur 24: Complexe lijstbewerking met behulp van Streams in Java.

Bovenstaande code maakt deel uit van een in Java geschreven AI die 4 op een rij kan spelen. Deze functie bepaalt de ideale kolom waarin de AI een zet dient te spelen om de hoogste winstkans te hebben. Dit gebeurt door voor elke kolom een 'gewicht' te bepalen aan de hand van de determineBranchValue-functie. Dit gewicht wordt gekoppeld aan de index van de kolom opgeslagen. Eens voor elke kolom het gewicht bepaald is wordt de lijst gereduceerd door de kolommen met het hoogste gewicht te bepalen, en uit deze resulterende lijst een willekeurige kolom te kiezen. Dit alles is dankzij Streams mogelijk in een enkele regel code. Java-technisch gezien is bovenstaande code immers een enkele regel, namelijk een enkele return-statement. Om leesbaarheidsredenen is het echter steeds aangewezen de code over meerdere regels te spreiden. Op deze manier is de complexe bewerking uit bovenstaande figuur nog steeds zeer goed leesbaar.
Voor veel toepassingen zijn Streams veel eleganter en compacter en bovendien leesbaarder dan standaard Java-code voor het bewerken van lijsten. Om dit te illustreren geeft Figuur 25 dezelfde functie weer als Figuur 24, maar dan zonder het gebruik van Streams.

```
public static int determineNextMove(Disc[][] field, int difficulty){
    int[] colWeights = new int[7];
    ArrayList<Integer> invalidMoves = new ArrayList();
    for (int col = 0; col < 7; col++){
        Disc[][]field2=null;
        boolean thrown = false;
        try {
            field2 = moveField(field, col, Disc.RED);
        } catch (MaxMovesReachedException ex) {
            invalidMoves.add(col);
            thrown = true;
        }
        if (!thrown){
            colWeights[col] = determineBranchValue(field2, difficulty,
Disc.GREEN);
            System.out.print(colWeights[col]+"\t");
        }
    }
    System.out.print("\n");
    int bestCol=0;
    while(invalidMoves.contains(bestCol) && bestCol < 7){
        if(bestCol == 6) return 10;
        else bestCol++;
    }
    for (int i=bestCol;i<7;i++){
        if(colWeights[i]>colWeights[bestCol] && !invalidMoves.contains(i))
bestCol = i;
    }
    ArrayList<Integer> bestCols = new ArrayList();
    for (int i = 0; i< 7;i++){
        if (colWeights[i] == colWeights[bestCol] &&
!invalidMoves.contains(i)) bestCols.add(i);
    }
    double i = (Math.random()*(double)bestCols.size());
    return bestCols.get((int)i);
}
```

Figuur 25: Complexe lijstbewerking met klassieke Java-code.

Bovenstaande code is duidelijk veel langer en moeilijker te begrijpen dan het voorbeeld met Streams, terwijl de werking ervan identiek is. Bovendien is door het grote aantal lussen deze code ook zeer foutgevoelig, en waren er een groot aantal pogingen nodig om de werking volledig juist te krijgen, wat bij de versie met Streams veel minder een probleem vormde. Dit illustreert zeer goed de grote voordelen die het gebruik van Streams in Java kan bieden.

3.2 C#

Ook C# bevat tegenwoordig een groot aantal functionele eigenschappen. Deze eigenschappen zijn sterk gelijkend op die die hierboven beschreven zijn voor Java. Zo maakt C# op exact dezelfde manier als Java gebruik van λ-expressies, predicaten, eerste- en hogere-orde-functies, currying, en functionele lijstbewerkingen. Enkel de syntax is bij veel van deze zaken licht afwijkend. Zo zijn λ-expressies in C# niet aangeduid met een -> maar met een => als symbool. Ook het type van functies in C# is niet Function maar Func. Ook is de :: notatie niet mogelijk in C#.
C# is in het algemeen wel flexibeler te gebruiken dan Java voor functioneel programmeren. Ze verplicht Java dat het type Function een enkele parameter als input heeft, waardoor alle functies van het type Function in Java zogenaamde 'monads' zijn [58]. C# biedt de mogelijkheid om functies met een arbitrair aantal parameters de definiëren, zoals Figuur 26 laat zien.

```
Func<int, int, int> multiply = (x, y) => x * y;
```

Figuur 26: Eerste-orde-functiedefinitie in C#.

Ook bestaat in C# een functioneel concept dat 'delegates' heet. Een delegate is in feite een sjabloon voor een functie. Een delegate legt vast wat de parameters en het return type van alle functies die tot de delegate behoren is. Zo is het mogelijk verschillende functies te groeperen en toe te passen alsof ze eenzelfde functie zouden zijn. Figuur 27 geeft een voorbeeld van een toepassing van delegates in C#.

```
class Program
{
    public delegate int del(int x, int y);

    static void Main(string[] args)
    {
        del multiplyDelegate = (x, y) => x * y;
        del sumDelegate = (x, y) => x + y;

        Func<int, int, del, int> applyToDouble = (x, y, d) => d(2 * x, 2 * y);

        applyToDouble(5, 5, multiplyDelegate);
        applyToDouble(5, 5, sumDelegate);
    }
}
```

Figuur 27: Toepassing delegates in C#.

Bovenaan is het delegate del gedefinieerd als sjabloon voor functies die 2 integers als parameter nemen en een integer als return-waarde hebben. De functies multiplyDelegate en sumDelegate zijn gedefinieerd alszijnde objecten van het type del. De functie applyToDouble ten slotte consumeert een functie van het type del en past deze toe op het dubbele van de inputparameters. Op deze manier is de functie applyToDouble bruikbaar voor zowel het toepassen van de functie multiplyDemlegate als de functie sumDelegate. Dit is een zeer elegante eigenschap die Java standaard niet bezit.

Ook C# beschikt over mogelijkheden om functioneel lijsten te bewerken. De bibliotheek die dit toelaat in C# heet LINQ. Deze bibliotheek bevat allerlei methodes om lijstbewerkingen te doen, en lijkt sterk op de Streams uit Java. LINQ in C# is echter net als de andere functionele eigenschappen net iets flexibeler en eleganter om te gebruiken. Zo is het bij LINQ niet nodig een lijst eerst expliciet om te zetten in een vorm waarop LINQ-methodes kunnen worden toegepast, en is de syntax van LINQ in het algemeen iets eleganter. In detail op alle kleine verschillen tussen Streams en LINQ ingaan zou te ver leiden voor deze masterproef. Daarom beperkt deze paragraaf zich tot het geven van een enkel voorbeeld waarbij LINQ is toegepast voor het bewerken van een lijst. Dit is weergegeven in Figuur 28.

```
int[] l = { 1,2,3,4,5,6,7,8,9,10 };
l.Where(i => i > 5)
    .Select(i => i * 5)
    .Aggregate(0, (acc, x) => acc + x);
```

Figuur 28: Lijstbewerkingen met LINQ.

In C# is het dus mogelijk om rechtstreeks LINQ los te laten op een array. Ook is in C# geen speciale behandeling nodig voor de value-types int, long, en double. De hierboven toegepaste functies zijn ook identiek aan die uit Java, maar hebben in C# een andere naam. Where komt exact overeen met filter uit de Streams, Select met map, en Aggregate met reduce. Bovenstaande code selecteert uit de lijst dus enkel de getallen groter dan 5, en vermenigvuldigt elk van deze getallen met 5. De resultaten hiervan worden uiteindelijk opgeteld.

3.3 Besluit

Zowel Java als C# beschikken over een heel aantal functionele eigenschappen, die probleemloos in combinatie met object-georiënteerde code binnen hetzelfde project kunnen voorkomen. Hierdoor zijn zeer elegante structuren uit functioneel programmeren zeer eenvoudig te combineren met de object-georiënteerde eigenschappen van deze talen. Met name voor lijstbewerkingen blinken functionele implementaties uit in Java en C#, dankzij de handige bibliotheken die in de talen geïntegreerd zijn die dit mogelijk maken. In het algemeen is C# wel nog net iets flexibeler en eleganter dan Java wat betreft de mogelijkheden voor een functionele programmeerstijl. In beide talen is het gebruik van functioneel programmeren echter geen enkel probleem en goed ondersteund.

4 F#

F# is de eerste hoofdzakelijk functionele taal die dit onderzoek nader bestudeert, zoals reeds aangegeven in paragraaf 2.2.6. F# is ontworpen als een cross-platform programmeertaal die meerdere programmeerparadigma's ondersteunt, met een focus op het functionele paradigma. F# draait op het .NET-framework van Microsoft. Hierdoor is het volledig compatibel met C#. Dezelfde IDE's kunnen gebruikt worden, en F# kan gebruik maken van zowat alle bibliotheken voor C#. Langs de andere kant is het ook eenvoudig om F#-bibliotheken te gebruiken in C#. Hioerover later in dit hoofdstuk meer.
Dit hoofdstuk heeft als doel een overzicht te scheppen van F#. Eerst geeft dit hoofdstuk een idee van de syntax van F#, waarna enkele belangrijke principes van de taal nader bestudeerd worden. Ten slotte beklijkt dit hoofdstuk de hierboven vermelde integratiemogelijkheden van F# met C# in detail.

4.1 Syntax

De eerste stap in de studie van eender welke programmeertaal is uiteraard de syntax van de taal bestuderen. Hoewel F# gebaseerd is op het .NET-framework, heeft het syntactisch gezien nagenoeg geen gelijkenissen met C#. De syntax van F# is zeer minimalistisch en declaratief. Dit weerspiegelt de functionele en dus declaratieve natuur van de taal. Figuur 29 geeft een voorbeeld van de definitie van een eenvoudige functie die twee getallen met elkaar vermenigvuldigt in F# weer.

```
let multiply x y = x * y
```

Figuur 29: Eenvoudige F#-functie.

Bovenstaande code is inderdaad zeer minimalistisch. Het let-keyword is belangrijk in C# en geeft de declaratie van een waarde aan. Dit kan zowel een functie als een gewone variabele zijn. Merk op dat alle gedeclareerde waarden in F# onveranderlijk zijn. Verder is enkel de naam van de functie te zien gevolgd door de parameters en wat de functie met de parameters doet. Ook puntkomma's zijn overbodig in F#.
Merk op dat nergens types gedefinieerd zijn, ondanks het feit dat F# een sterk getypeerde taal is. F# past dan ook zoals in paragraaf 2.2.5.3 reeds beschrevenb type-inferentie toe. Bovenstaande functie heeft dus als type int->int->int, aangezien het een functie is die twee integers als parameter heeft. Hieruit volgt onmiddellijk een andere belangrijke eigenschap die F# overneemt uit het functionele paradigma, namelijk de inherente mogelijkheid om gedeeltelijke functie-applicatie, ofwel currying, toe te passen. Het is immers automatisch mogelijk de functie multiply op te roepen met slechts 1 parameter. Het resultaat hiervan is een functie die de input vermenigvuldigt met een vast getal x. Figuur 30 geeft weer hoe dit in zijn werk gaat.

```
let multiply x y = x * y
let multiplyByTwo = multiply 2

[<EntryPoint>]
let main argv =
    printfn "%A, %A" (multiply 5 5) (multiplyByTwo 5)
    Thread.Sleep 2000
    0
```

Figuur 30: Currying in F#.

In bovenstaande code is de functie multiplyByTwo gedefinieerd die gelijk is aan de functie multiply met als eerste argument 2. Het type van deze functie is dus int->int. De functie vermenigvuldigt zijn input met twee. De output van het programma is in dit geval dus '25, 10'.
Het valt op dat de code in de main-methode geïndenteerd is, terwijl de andere functiedeclaraties dat niet zijn. Dit is niet zuiver esthetisch. F# dwingt programmeurs immers structuur in hun code te brengen door indentatie als controlestructuur te gebruiken, net zoals de meer bekende object-georiënteerde programmeertaal Python. In F# komen dus zelden accolades of andere vervelende syntactische formaliteiten voor. Ook het gebruik van parentheses is in F# tot een minimum beperkt. Parameters worden gewoon rechtstreeks achter de functiedefinitie geschreven. Dit maakt currying in deze taal ook zo elegant. Dit alles bevordert de leesbaarheid van de code aanzienlijk.
F# kent uiteraard ook lijsten. De syntax hiervoor maakt gebruik van vierkante haken. De elementen zijn gescheiden door puntkomma's. Ook is er speciale syntax voorzien voor het genereren van specifieke vaak voorkomende lmijst-patronen. Dit illustreert de minimalistische natuur van de syntax goed. Figuur 31 geeft de specifieke lijst-syntax van F# weer.

```
let list = [1;2;3;4;5]
let altList = [1..5]
let bothLists = list @ altList
```

Figuur 31: Alternatieve manieren om lijsten aan te maken en samen te voegen.

Zoals hierboven te zien is het mogelijk een lijst expliciet te definiëren als een opsomming van alle elementen in de lijst. Daarnaast is er voor lijsten van opeenvolgende gehele getallen de mogelijkheid om een compactere notatie te gebruiken. Het @-symbool dient in F# om twee lijsten samen te voegen.
F# biedt een heel aantal mogelijkheden aan om stijlvol en compact lijstbewerkingen toe te passen. Dit is één van de grote sterktes van functioneel programmeren. De lijstbewerkingen die in F# beschikbaar zijn, zijn enigszins gelijkend op de lijstbewerkingen die Streams in Java en LINQ in C# bieden, zoals beschreven in het vorige hoofdstuk. In F# zijn deze lijstbewerkingen echter nog net iets eleganter, zoals Figuur 32 illustreert.

```
let l = [-5..5]
let positive x = x > 0

let res = List.filter positive l
```

Figuur 32: Lijstbewerking in F#.

Bovenstaande code maakt eerst een, lijst aan van gehele getallen zoals eerder beschreven. De functie positive heeft als type int -> bool en bepaalt of de input al dan niet positief is. Nu is het in F# zeer compact mogelijk om de lijst l te filteren met de ingebouwde List.Filter functie, die als argumenten een lijst heeft en een functie die het type van de elementen in de lijst omzet in een bool. Dan filtert deze functie als het ware de lijst door de predicaatsfunctie op alle elementen ervan toe te passen en enkel degene te behouden waarvoor het resultaat true is. De werking ervan is dus identiek aan de filter-functie van Streams in Java.
Naast bovenstaande notatie is het in F# ook perfect mogelijk λ-expressies te gebruiken. Deze functioneren in F# op juist dezelfde manier als in Java of C#. De syntax ervan komt zelfs grotendeels overeen met die van in Java, met als enige verschil dat het in F# verplicht is om het 'fun'-keyword te gebruiken om aan te geven dat het om een λ-expressie gaat. Figuur 33 geeft de res-functie uit Figuur 32 weer, maar dan met een λ-expressie in plaats van een voorgedefinieerde functie.

```
let resLambda = List.filter(fun x-> x>0) l
```

Figuur 33: Lijstbewerking met λ-expressie.

Zoals hierboven vermeld is het gebruik van λ-expressies in F# volledig ondersteund. De werking ervan is in essentie analoog aan die van de λ-expressies in Java, die reeds beschreven zijn in paragraaf 3.1.1.
In F# is het evenzeer mogelijk meerdere lijstbewerkingen achter elkaar uit te voeren. Hiervoor beschikt F# over een zeer elegant stukje syntax dat 'piping' heet. Deze syntax leidt de output van de ene functie naar de input van de andere als een 'stroom'. Figuur 34 geeft het gebruik van deze syntax weer.

```
let resPiped = [-5..5] |>
    List.filter positive |>
    List.map(fun x-> x*2) |>
    List.reduce (fun x y-> x + y)
```

Figuur 34: Piping-syntax voor het verwerken van lijsten als 'stromen'

Bovenstaande syntax doet opnieuw sterk denken aan de Streams uit Java. Alle gebruikte functies hebben dan ook juist dezelfde betekenis. De output van de ene functie wordt rechtstreeks doorgegeven als input van de andere functie. Dit is uiteraard een zeer elegante manier om lijstbewerkingen te doen.
Het is uiteraard mogelijk nog veel dieper in te gaan op de syntax van F#. Er zijn uiteraard nog tal van andere ingebouwde lijstbewerkingsfuncties etc. Voor deze masterproef zou dit echter te ver leiden. De documentatie van F# biedt hier uiteraard veel hulp in, net zoals websites als [59], waar ook bovenstaand overzich van de F#-syntax is gebaseerd. De

volgende paragrafen van dit hoofdstuk zijn eveneens grotendeels op deze website gebaseerd en geven een overzicht van de interessantste en belangrijkste eigenschappen van F# die deze taal onderscheiden van veel andere al dan niet functionele programmeertalen. Het valt in het algemeen op dat de syntax van F# sterk lijkt op die van de academische functionele programmeertaal Haskell, die kort beschreven is in paragraaf 2.2.5.1.

4.2 Types

Zoals in paragraaf 2.2.5.3 reeds aangegeven is F# een sterk getypeerde taal. Op zich is dit niets bijzonders, maar de manier waarop F# deze sterke typering implementeert is sterk verschillend van de manier waarop dit in veel mainstream programmeertalen zoals Java en C# gebeurt. Daarom schenkt deze paragraaf even bijzondere aandacht aan de unieke manier waarop F# met types omgaat, en de vele mogelijkheden van types in deze taal.
Een eerste soort type in F# is het zogenaamde 'record type'. Dit type bestaat in feite uit een opsomming van de velden van het type, met voor elk veld op zich het type van het veld. Figuur 35 geeft een voorbeeld van een eenvoudig record type in F#.

```
type CarManufacturer = {
    Name: string;
    Country: string;
    FoundingYear: int;
}
```

Figuur 35: Record type in F#.

Het record type is zoals de naam al doet vermoeden eenvoudigweg een opsomming van een aantal velden en de types van deze velden. Dit soort van type doet sterk denken aan een struct uit de programmeertaal C. Het is eventueel ook te vergelijken met een klasse uit object-georiënteerde programmeertalen. Het record type in F# is echter veel compacter. Er bestaan ook geen concepten als erving voor dit type, en er zijn ook nooit constructors of iets dergelijks gedefinieerd. Record types worden rechtstreeks geïnitialiseerd door de waardes van elk van hun velden op te geven. Figuur 36 geeft de initialisatie van een record type in F# weer.

```
let t = {Name= "Toyota"; Country="Japan";FoundingYear=1937}
```

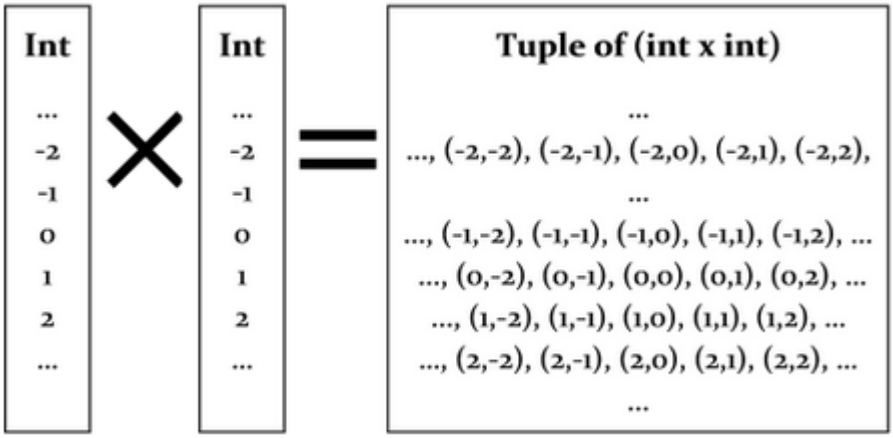
Figuur 36: Initialisatie van een record type.

Bovenstaande syntax is vanzelfsprekend en behoeft geen verdere toelichting.
Naast record types beschikt F# ook over een ander soort type, namelijk 'union types'. Dit soort van types bestaat uit een keuze tussen verschillende subtypes, die elk ook nog eens een verschillend type kunnen, en dus data met zich mee kunnen dragen. Het union type is dus vergelijkbaar met enums uit Java, met als belangrijk verschil dat union types veel eleganter zijn in gebruik en vooral hun declaratie. Figuur 37 geeft een voorbeeld van een union type in F# weer.

```
type CarBrand =
    | Toyota of CarManufacturer
    | Ford of CarManufacturer
    | Mercedes of CarManufacturer
```

Figuur 37: Union type in F#.

Het type CarBrand kan dus oftewel een Toyota, een Ford, of een Mercedes zijn. Elk van deze opties is van het type CarManufacturer. De data uit CarManufacturer is dus rechtstreeks toegankelijk vanuit het type CarBrand. Bij enums in object-georiënteerde talen is dit vaak niet mogelijk en zelfs als het mogelijk is -zoals in Java- niet gebruikelijk.
Naast gewone types is het ook mogelijk om tupels te declareren in F#. Dit is in F# zeer wiskundig opgevat. De syntax voor de declaratie van een tupel maakt immers gebruik van de wiskundige *-operator. Dit omdat de declaratie van een tupel in feite een carthesiaanse vermenigvuldiging van alle mogelijke waarden van elk van de elementen van het tupel inhoudt. Figuur 38 verduidelijkt dit concept.



Figuur 38: Tupels voorgesteld als carthesiaanse vermenigvuldiging [60].

Deze wiskundige beschouwing van tuples als een vermenigvuldiging van types kan eerst wat verwarrend zijn, maar is na enige gewenningstijd behoorlijk logisch. Dit laat ook goed de pure wiskundige en functionele natuur van F# zien. Figuur 39 toont het gebruik van tupels in F#.

```
type Car = {
    Type: CarBrand * CarModel
    Year: int;
    Options: CarOption list
}
```

Figuur 39: Declaratie van een type met een tupel in F#.

Het record type Car heeft een datamember Type. Het type van deze datamember is een tupel van CarBrand en CarModel. Bij nader inzien is de syntax voor tupels dus toch overzichtelijk en elegant.
Merk in bovenstaande code de declaratie van het veld Options op. Dit is een lijst van objecten van het type CarOption. In F# bestaan er dus ook generics zoals in C# en Java, maar

ook hiervoor is de syntax eleganter en compacter in F#. Ondanks het feit dat de definitie van het type Car verschillende ietwat complexe structuren bevat, is in één oogopslag duidelijk wat dit type juist inhoudt. Dit is dus een uitstekend voorbeeld de declaratieve natuur van F# en de voordelen die hieraan verbonden zijn.
Als afsluiter van deze paragraaf geeft Figuur 40 een methode weer die een variabele van het type Car aanmaakt.

```
let carFactory =
    let t = {Name= "Toyota"; Country="Japan";FoundingYear=1899}
    {Type=Toyota t, Prius; Year= 2012; Options=[AirCo; HeatedSeats]}
```

Figuur 40: Aanmaken van het vrij complexe type Car uit Figuur 39 in F#.

Zoals hierboven te zien is het aanmaken van complexe types in F# zeer evident, en kunnen types gewoon worden doorgegeven aan comlplexere types. Union types kunnen zelfs rechtstreeks worden aangesproken met de waardes van hun subtypes. Merk op dat bij het initialiseren van tupels een komma als syntax gebruikt dient te worden. Verder is de code zelfverklarend.
Het valt ten slotte op dat ook voor door de programmeur gedefinieerde types het type van de variabele niet dient meegegeven te worden bij de declaratie ervan. De F#-compiler leidt dit automatisch af uit de vorm van de initialisatie. In sommige gevallen kan het echter wel nodig zijn een type te specifiëren, indien de declaratie dubbelzinnig is. In F# is het aanmaken van nieuwe types echter zo eenvoudig dat het gebruikelijk is met een groot aantal types te werken die een bepaald probleem zeer strikt afbakenen. Hierdoor valt het helemaal niet vaak voor dat er dubbelzinnige typedefinities ontstaan in een goed geschreven F#-programma.
Uit bovenstaande is te besluiten dat ook in het typesysteem de gelijkenissen tussen F# en Haskell opvallen. Deze taal is dus duidelijk sterk geïnspireerd op Haskell.

4.3 Pattern matching

Zoals veel functionele programmeertalen is ook in F# pattern matching een uitstekende manier om de control flow van een programma te controleren. Pattern matching in F# is behoorlijk krachtig en bevat een aantal interessante concepten. Het gedrag van een functie kan immers volledig worden gecontroleerd door het gebruik van Pattern Matching. Figuur 41 geeft hier een uitstekend voorbeeld van weer.

```
let listMatcher aList =
    match aList with
    | [] -> printfn "the list is empty"
    | [firstElement] -> printfn "the list has one element %A " firstElement
    | [first; second] -> printfn "list is %A and %A" first second
    | _ -> printfn "the list has more than two elements"
```

Figuur 41: Pattern matching in F# [61].

De definitie van de pattern matching is zoals zo veel andere concepten in F# compact en elegant. De functie listMatcher print afhankelijk van de inhoud van het argument aList steeds een andere input terug. Het laatste opgegeven patroon is een _ en slaat op 'alle andere

gevallen'. Indien geen enkel concreet patroon matcht geeft de functie dus dit laatste scenario terug.
Pattern matching in F# is ook rechtstreeks toe te passen op complexe types. Beschouw hiervoor Figuur 42, die verder bouwt op de code uit vorige paragraaf.

```
let getManufacturer brand =
    match brand with
        | Toyota({Name=n}) -> n

let getCarType {Type=t, model} =
    getManufacturer t
```

Figuur 42: Pattern matching op complexe type sin F#.

De functie getManufacturer matcht op een CarBrand en geeft in het geval het merk 'Toyota' is de naam van de CarManufacturer gebonden aan het CarBrand-type terug. De functie getCarType heeft als argument een Car car. Deze Car kan rechtstreeks in de definitie van het argument ontleed worden in de bestanddelen ervan. Het is ook mogelijk slechts een deel van de bestanddelen van de doorgegeven types op te vragen, wat hierboven gebeurd is. De type-inferentie weet steeds de juiste types te achterhalen en zorgt dat bovenstaande code zeer compact de naam van de fabrikant van een auto kan achterhalen. Deze compacte pattern matching is uiteraard een krachtig hulpmiddel bij het bepalen van de control flow van een programma.
Een laatste belangrijke eigenschap van F# wat betreft pattern matching die vrij uniek is aan deze taal is 'active patterns'. Dit zijn patronen die eens ze gedefinieerd zijn dynamisch toepasbaar zijn in andere functies, zodat de functies zelf nog compacter en eleganter geschreven kunnen worden, zonder telkens de matching expliciet te moeten definiëren of ernaar te moeten verwijzen. Active patterns kunnen bijzonder handig zijn bij bijvoorbeeld het parsen van tekst, zoals aangegeven in [62]. Figuur 43 geeft hiervan een voorbeeld weer.

```
// create an active pattern
let (|Int|_|) str =
   match System.Int32.TryParse(str) with
   | (true,int) -> Some(int)
   | _ -> None

// create an active pattern
let (|Bool|_|) str =
   match System.Boolean.TryParse(str) with
   | (true,bool) -> Some(bool)
   | _ -> None
```

Figuur 43: Aanmaken van active patterns in F# [62].

Hierboven worden active patterns gedefinieerd voor het parsen van een string tot een Int of een Bool. De code hiervan is vrij omvangrijk. Daarom is het juist zo interessant om ze apart te kunnen definiëren.

```
// create a function to call the patterns
let testParse str =
    match str with
    | Int i -> printfn "The value is an int '%i'" i
    | Bool b -> printfn "The value is a bool '%b'" b
    | _ -> printfn "The value '%s' is something else" str
```

Figuur 44: Functie die gebruik maakt van de active patterns beschreven in Figuur 43 [62].

Bovenstaande code is dus in staat de inputstring rechtstreeks te matchen met een Int of Bool. In de functie testParse is nergens aangegeven dat de input eerst geparset dient te worden. De active patterns zorgen hier zelf voor. Dit is uiteraard een zeer krachtig principe in F# dat nog verdere vereenvoudiging van code toelaat en een nog meer declaratieve programmeerstijl mogelijk maakt.

4.4 C#-integratie

Zoals in paragraaf 2.2.5.3 reeds was aangegeven, is F# gebaseerd op het .NET-framework en laat deze taal heel wat interactie met klassieke C#-bibliotheken toe. Onrechtstreeks was dit al duidelijk uit bovenstaande paragrafen. Veel types en functies in F# zijn immers rechtstreeks dezelfde alms die uit C#. Zo is bijvoorbeeld de Thread.Sleep-functie die in Figuur 30 gebruikt is identiek dezelfde als de Thread.Sleep-functie uit C#.
Integratie van C# in en F# en omgekeerd kan echter nog een stuk verder gaan. Het is immers mogelijk in F# modules te declareren, die dienen als bibliotheken. Het is mogelijk deze modules dan te builden, en de geproduceerde assemblies toe te voegen aan een bestaand C#-project als dependencies. De syntax in F# voor het maken van een module is als weergegeven in Figuur 45.

```
namespace Library

module Multiplication =

    let multiply x y = x*y
    let multiplyByTwo = multiply 2
```

Figuur 45: Bibliotheek in F#.

Bovenstaande code declareert een module Multiplication in de namespace Library, met daarin 2 methodes. Na het builden van dit project en toevoegen van de gegenereerde executable als dependency in eender welk ander C# of F#-project is het mogelijk deze library rechtstreeks aan te spreken.
Ook in de andere richting is dit uiteraard mogelijk. De werkwijze is exact hetzelfde.
De enorm flexibele integratiemogelijkheden tussen F# en C# zijn uiteraard een groot pluspunt voor deze taal. Hierdoor is voor veel toepassingen het mogelijk om gelijdelijk F# te integreren in bestaande projecten, wat drempel voor de overstap van C# naar F# voor veel toepassingen een stuk minder hoog maakt. Deze geweldige integratiemogelijkheden laten ook toe om binnen eenzelfde project F# te gebruiken waar F# het sterkst is, en de overige zaken over te laten aan C#. Voor dit onderzoek betekent deze integratie ook dat cloud-platformen die .NET ondersteunen, hoogstwaarschijnlijk ook mogelijkheden bieden voor F#.

4.5 Besluit

F# is een bijzonder elegante functionele programmeertaal. De taal is duidelijk sterk geïnspireerd door Haskell, maar heeft ook heel wat unieke eigenschappen, en past verschillende klassieke functionele concepten op een eigenzinnige, soms vernieuwende manier toe.

Door de elegante, leesbare, en compacte syntax en bovendien het gebrek aan neveneffecten is F# een heel sterke taal om aan dataverwerking te doen, aangezien data zo eenvoudig te structureren is door het uitgebreide sterke typesysteem, en omdat lijstbewerkingen zo eenvoudig en stijlvol zijn in F#. Ook de krachtige active patterns zijn in het bijzonder voor dataverwerking heel interessant.

Ten slotte maakt de eenvoudige integratie met C# van deze taal evenzeer een krachtige uitbreiding voor het .NET-framework als een stand-alone programmeertaal. Optiomaal gebruik van deze taal bakent de taken van F# dan ook af tot de zaken waar het erg goed in is. Al bij al is F# een zeer interessante moderne functionele programmeertaal met tal van mogelijkheden.

5 Wolfram

Wolfram is van alle talen die dit onderzoek omhelst de minst conventionele. Wolfram is hoofdzakelijk functioneel van aard, maar springt op een unieke manier om met de functionele concepten. Programmeergemak staat centraal bij deze taal. Omdat uit paragraaf 2.2.5.5 blijkt dat Wolfram een heel aantal interessante eigenschappen heeft die verder gaan dan wat de meeste programmeertalen aanbieden, is besloten voor dit onderzoek Wolfram veel diepgaander te bestuderen dan de andere gekozen talen, en is er heel wat geëxperimenteerd met de taal en alle mogelijkheden ervan. Hieronder een overzicht van de belangrijkste aspecten van deze taal, en de mogelijkheden en beperkingen ervan. Dit hoofdstuk geeft ook de in dit onderzoek bepaalde bevindingen van deze taal weer wat betreft gebruiksgemak, mogelijke toepassingsgebieden, etc.
Een belangrijke opmerking bij dit hoofdstuk is dat Wolfram in het algemeen zeer goed gedocumenteerd is. Daarom steunt heel dit hoofdstuk in de eerste plaats op de officiële Wolfram-documentatie, die terug te vinden is op [63].

5.1 Syntax

De Wolfram-syntax is zeer uniform en intuïtief. Daarnaast is ze wiskundig ook heel zuiver. Hieronder een overzicht van basisbegrippen uit de Wolfram-syntax die goed de eigenschappen van de taal illustreren.

5.1.1 Expressies

Wolfram-code stelt alles op een uniforme manier voor, namelijk met behulp van expressies. De Wolfram kernel interpreteert deze expressies dan bij het uitvoeren van de code.
Een expressie bestaat uit twee delen: een head en optionele argumenten. Het head is de naam van de expressie, en de argumenten zijn een lijst van andere expressies, die als parameters dienen. Deze argumenten staan steeds tussen vierkante haken. Een voorbeeld van een eenvoudige expressie is weergegeven in Figuur 46.

```
In[5]:= Max[1, 2, 3]
Out[5]= 3
```

Figuur 46: Voorbeeld van een Wolfram-expressie.

De *Max*-functie is een ingebouwde functie in Wolfram die het maximum zoekt van alle opgegeven argumenten. In totaal zijn er meer dan 5000 ingebouwde functies die allerhande taken op zich nemen van eenvoudige zaken zoals hierboven tot het genereren van grafieken en complexe figuren in 3D. Hierover later meer. Al deze functies zijn opgebouwd als expressies, op exact dezelfde manier als in Figuur 46.

5.1.2 Lijstbewerkingen

Wolfram kent naast standaard-expressies ook lijsten. Lijsten worden genoteerd tussen accolades. Om elementen van een lijst op te vragen, moet een dubbele set vierkante haken gebruikt worden. Merk op dat in Wolfram de nummering van de elementen in de lijst start bij 1, in tegenstelling tot de meeste andere programmeertalen, waarbij het eerste element de index 0 krijgt. Figuur 47 geeft weer hoe dit eruit ziet in de praktijk.

```
In[1]:= {a, b, c, d}[[3]]
Out[1]= c
```

Figuur 47: Lijst en opvraging van een elemnt uit die lijst in Wolfram.

Zoals te zien is het derde element van de lijst c, en niet d zoals in andere programmeertalen het geval zou zijn.
Met dezelfde notatie als hierboven is het mogelijk ook sublijsten te genereren. Dit maakt van deze notatie een krachtiger hulpmiddel dan in talen als Java of C#, waar rechtstreekse interactie met lijsten beperkt is tot het opvragen van een enkel element. Voor complexere handelingen moeten controlestructuren of speciale functies gebruikt worden in deze talen, afhankelijk van de specifieke lijst-klasse die gebruikt wordt. Dit benadrukt het voordeel van de transparantie van Wolfram. Er is slechts één soort lijsten, met één set van ingebouwde eigenschappen. Deze eigenschappen blijken daarbovenop nog zeer elegant en krachtig te zijn. Figuur 48 laat een voorbeeld van een sublist-notatie in Wolfram zien.

```
In[1]:= {a, b, c, d, e, f}[[2 ;; 4]]
Out[1]= {b, c, d}
```

Figuur 48: Iets complexere lijstbewerking in Wolfram.

De notatie uit Figuur 48 vraagt dus het tweede tot en met vierde element uit de lijst op. Deze notatie heeft echter nog meer mogelijkheden, zoals Figuur 49 laat zien.

```
In[15]:= {"a", "b", "c", "d", "e", "f"}[[2 ;; 4 ;; 2]]
Out[15]= {b, d}
```

Figuur 49: Complexe lijstbewerking in Wolfram.

De code uit bovenstaande figuur maakt een sublijst van de originele lijst, en behoudt van deze sublijst enkel de elementen op de even indices. Dit is dus een vrij complexe lijstbewerking met toch een bijzonder compacte notatie.
Er valt nog heel wat te vertellen over lijstbewerkingen waaruit blijkt dat Wolfram een zeer compacte en flexibele taal is. Zoals bijvoorbeeld in Figuur 50 te zien is, is het mogelijk de grens-indices gewoon weg te laten. In dit geval vult Wolfram de grenzen voor de lijstbewerking automatisch aan met het einde of begin van de lijst.

```
In[24]:= {"a", "b", "c", "d", "e", "f"}[[2 ;; ;; 2]]
Out[24]= {b, d, f}
```

Figuur 50: Lijstbewerking met weggelaten eindindex.

Zoals verwacht is het resultaat in Figuur 50 hetzelfde als Figuur 49, maar met '*f*' toegevoegd, aangezien de lijstbewerking nu geldt voor heel de lijst vanaf index 2.
Een andere eigenschap van lijstbewerkingen volgens bovenstaande notatie die niet voorkomt in conventionele talen is dat negatieve indices probleemloos te gebruiken zijn, zoals Figuur 51 laat zien.

```
In[21]:= {"a", "b", "c", "d", "e", "f"}[[-2 ;; -1]]
Out[21]= {e, f}
```

Figuur 51: Lijstbewerking met negatieve indices in Wolfram.

Negatieve indices beginnen dus eenvoudigweg achteraan de lijst te tellen in plaats van vooraan.
Zoals uit bovenstaande figuren duidelijk te zien is, is Wolfram een zeer flexibele en krachtige taal. De notatie is elegant en compact. Complexe lijstbewerkingen zijn syntactisch gezien bijna even eenvoudig als het opvragen van een enkel element uit een lijst. Deze flexibiliteit kan naast een zege ook wel een vloek zijn. Zo is het vrij makkelijk om fouten te maken in Wolfram die moeilijk te detecteren zijn. Java zou immers onmiddellijk een foutmelding geven als een element met een negatieve index wordt opgevraagd uit een lijst. Wolfram daarentegen blijft in veel gevallen gewoon werken. Dit betekent dat verstrooidheidsfouten van de programmeur moeilijker te detecteren zijn. Er is dus nood aan aandachtige en verregaande testing om zeker te zijn dat een stuk Wolfram-code effectief doet wat het lijkt te doen en zou moeten doen.
Een eenvoudige en elegante manier om lijsten te generen in Wolfram is door gebruik te maken van de ingebouwde functie '*Table*'. Deze functie is ook een voorbeeld van de flexibiliteit van Wolfram. Figuur 52 geeft een eenvoudige toepassing van de *Table*-functie weer.

```
In[1]:= Table[i, {i, 10}]
Out[1]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Figuur 52: Table-functie voor het genereren van lijsten in Wolfram.

Bovenstaande code genereert een lijst natuurlijke getallen van 1 tot 10. De manier waarop dit gebeurt is niet zo conventioneel. De *Table*-functie heeft als eerste argument steeds een expressie die weergeeft hoe de verschillende elementen in de lijst eruit moeten zien, in functie van een inwendige variabele, in dit geval *i*. De andere argumenten zijn een opsomming van deze interne variabelen, en de grenzen waartussen deze variabelen variëren. Elke interne variabele dient samen met zijn resp. grenzen in een lijst gedeclareerd te worden. Er zijn een groot aantal mogelijkheden om specifiekere grenzen te specifiëren, die in de documentatie toegelicht zijn. Voor deze masterproef is de exacte syntax echter niet

zozeer van belang. Wat wel van belang is, is dat deze functie ook zeer flexibel is. Zo kunnen aan de hand van de *Table*-functie zeer eenvoudig geneste lijsten en complexe structuren worden opgebouwd. Figuur 53 geeft deze flexibiliteit weer.

```
In[12]:= Table[j, {i, 5}, {j, 5}]
Out[12]= {{1, 2, 3, 4, 5}, {1, 2, 3, 4, 5}, {1, 2, 3, 4, 5}, {1, 2, 3, 4, 5}, {1, 2, 3, 4, 5}}
```

Figuur 53: Nested lijsten aan de hand van de Table-functie.

Zoals hierboven te zien is volstaat het om enkel een extra interne variabele toe te voegen aan de *Table*-functie om een geneste lijst te genereren. De laatste variabele komt overeen met de binnenste nesting. Dit laat ook weer de grote flexibiliteit van Wolfram zien, aangezien het mogelijk is een willekeurig aantal argumenten toe te voegen aan de *Table*-expressie die steeds een vastgelegd effect hebben. Een nadeel van deze grote flexibiliteit is weer dat veel zaken bij conventie zijn vastgelegd en het moeilijk is als beginnende programmeur om het exacte gedrag van een expressie te voorspellen zonder eerst grondig de documentatie te lezen. Kennis van de documentatie is ook essentieel voor het werken met Wolfram.
De expressie die weergeeft hoe de *Table*-functie de lijstelementen moet genereren kan eender welke expressie zijn. Expressies kunnen dus gewoon als argumenten worden doorgegeven, en zijn dus eerste-orde elementen. De *Table*-functie is dus een hogere-orde functie. Dit is een goed voorbeeld van de functionele eigenschappen van Wolfram. Figuur 54 illustreert dit.

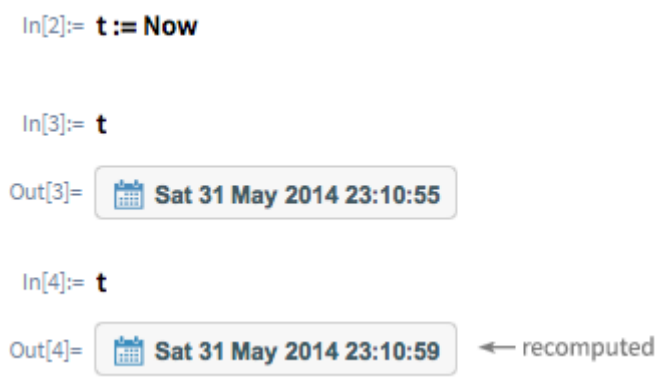
```
In[13]:= Table[Max[i, j], {i, 5}, {j, 5}]
Out[13]= {{1, 2, 3, 4, 5}, {2, 2, 3, 4, 5}, {3, 3, 3, 4, 5}, {4, 4, 4, 4, 5}, {5, 5, 5, 5, 5}}
```

Figuur 54: De Max-functie wordt rechtstreeks in de Table-functie als argument meegegeven.

5.1.3 Toekenningen

Wolfram biedt zoals zowat elke programmeertaal de mogelijkheid variabelen aan te maken. Deze variabelen kunnen eender welke expressie bevatten. Wolfram geeft de mogelijkheid om variabelen op twee verschillende manieren toe te kennen. Enerzijds bestaat er de directe toekenning, en anderzijds de indirecte toekenning.
De directe toekenning wordt aangeduid met =. In dit geval wordt de waarde van de variabele onmiddellijk berekend bij de toekenning. Dit komt overeen met eager evaluation. Deze toekenning is vooral nuttig voor constanten. De waarde van variabelen is op zich wel steeds te veranderen. Dit is een object-georiënteerde eigenschap van Wolfram, wat de grote flexibiliteit van Wolfram weergeeft. Naast deze eigenschap bestaan er in Wolfram ook controlestructuren als if- for- en while-lussen. De werking ervan is gelijkaardig aan wat te verwachten is van object-georiënteerde talen. De focus van Wolfram is echter functioneel/declaratief. In een goed geschreven Wolfram-programma komen deze controlestructuren dan ook zelden voor.

Ook is het mogelijk om door het gebruik van een ; aan het einde van eender welke regel Wolfram-code de output ervan te onderdrukken. Zonder de ; geeft Wolfram de output van elke geëvalueerde expressie weer, wat handig kan zijn voor het debuggen.
Indirecte toekenning langs de andere kant wordt aangeduid met :=. Bij het gebruik van deze toekenningsmethode herberekent Wolfram de waarde van de variabele elke keer wanneer ze aangeroepen wordt. Dit lijkt op lazy evaluation, met als belangrijk verschil dat de waarde steeds opnieuw berekend wordt als de variabele aangeroepen wordt, terwijl bij lazy evaluation de waarde slechts één keer wordt berekend, namelijk bij de eerste aanroep.
Figuur 55 illustreert hoe de indirecte toekenning in Wolfram in zijn werk gaat.



Figuur 55: Illustratie van de werking van indirecte toekenning in Wolfram aan de hand van de ingebouwde functie Now [64].

De ingebouwde functie *Now* geeft de huidige datum en tijd weer. Het is duidelijk dat bij de tweede aanroep de waarde van t herberekend is ten opzichte van de eerste keer. Merk op dat alle toekenning die de gebruiker doet -hetzij direct, hetzij indirect- bij conventie met een kleine letter dienen te beginnen. Ingebouwde functies beginnen steeds met een hoofdletter.

5.1.4 Functies

De indirecte toekenning is ook te beschouwen als een functiedefinitie. Een functie is technisch gezien immers ook niet meer dan een element dat steeds opnieuw wordt berekend wanneer het wordt aangeroepen. Een functie in Wolfram heeft dus juist dezelfde notatie als eender welke andere indirect toegekende variabele.
Natuurlijk is het gewenst om argumenten door te geven aan een functie. Aangezien een functie gewoon een expressie is die toegekend is met een indirecte toekenning, kan dit eenvoudig door een lijst van argumenten toe te voegen aan de declaratie ervan. Deze argumenten dienen gedefinieerd te zijn als interne variabelen binnen de functie. Dit kan door een _ toe te voegen achter de naam van het argument. Daarna is het mogelijk deze rechts van de declaratie gewoon te gebruiken als variabelen. Bij de uitvoering van de code vervangt Wolfram dan steeds de naam van het argument door de doorgegeven waarde bij de functie-aanroep. Figuur 56 maakt dit alles allicht duidelijker.

```
In[2]:= f[x_] := x + 1
        f[5]
Out[3]= 6
```

Figuur 56: Eenvoudige functiedefinitie in Wolfram.

5.1.5 Pattern matching

Aangezien gedeclareerde functies evenzeer expressies zijn als alle andere elementen binnen Wolfram, heeft Wolfram van nature veel potentieel voor het gebruik van pattern matching. Hierop is door de taal dan ook sterk ingespeeld. Als het mogelijk is een expressie *f[x_]* te definiëren, is het immers evenzeer mogelijk een expressie *f[10]* te definiëren die het gedrag van de meer algemene *f[x_]* niet volgt. Zo kan op zeer stijlvolle manier pattern matching worden toegepast in Wolfram, wat ook een hoeksteen is van declaratief/functioneel programmeren. Figuur 57 geeft een eenvoudig voorbeeld van pattern matching in Wolfram.

```
In[22]:= f[x_] := x + 1
         f[3] := "vier"
         Table[f[i], {i, 5}]
Out[24]= {2, 3, vier, 5, 6}
```

Figuur 57: Basis pattern-matching in Wolfram.

Het is duidelijk dat bij het argument *3* de functie f speciaal gedrag vertoont. Merk op dat de meer concrete definitie *f[3]* is toegepast op dit argument, ook al is de algemene *f[x_]* eerst gedefinieerd. Bij het uitvoeren van de code hebben meer concrete functiedefinities dus voorrang op de meer algemene bij pattern matching. Dit geeft de flexibiliteit van Wolfram opnieuw weer. Wolfram zoekt immers automatisch naar de 'beste' manier om de code uit te voeren. Deze flexibele omgang met pattern matching is daar een gevolg van.
Aangezien parameters van functies op zich ook expressies zijn, is pattern matching ook te gebruiken om een notie van types te krijgen in Wolfram. In Figuur 58 vertoont *f* een ander gedrag bij een parameter van het type *v* dan bij één van het type *u*. Dit kan een krachtige methode zijn om controlestructuren zoals if- en switch statements uit object-georiënteerde talen na te bootsen, en een notie van 'object' te krijgen in Wolfram, aangezien *u* en *v* eigenlijk gewoon expressies zijn die als 'wrapper' dienen rond de eigenlijke data.

```
In[25]:= f[u[x_]] := x + 1
         f[v[x_]] := x - 1
         Table[{f[u[i]], f[v[i]]}, {i, 5}]
Out[27]= {{2, 0}, {3, 1}, {4, 2}, {5, 3}, {6, 4}}
```

Figuur 58: Pattern matching om objecten van verschillende tyupes uit elkaar te houden.

Pattern matching kan echter nog veel uitgebreider in Wolfram. Er zijn verschillende mechanismen hiervoor. Door de _ te vervangen door een __ matcht de expressie op eender welke sequentie van expressies in plaats van op een enkele expressie. Zoals uit voorgaande

volgt staat een _ in een functie-definitie voor “eender welke expressie”. Aan de _ kan echter ook een head worden toegevoegd. Dan matcht de functie enkel op expressies met die bepaalde head. Figuur 59 geeft dit weer.

```
In[62]:= f[x_u] := "a"
         f[x_] := "b"
         {f[u[2]], f[v[2]]}
Out[64]= {a, b}
```

Figuur 59: Pattern matching in parameters in Wolfram.

Ten slotte is er nog een heel flexibele vorm van pattern matching binnen Wolfram die toelaat eender welke boolse expressie op te geven om te bepalen of een expressie matcht met het verwachte patroon of niet. De notatie voor deze vorm van pattern matching is /;. Wolfram gaat ook hier heel flexibel mee om. Waar de /;-uitrukking juist staat is niet van belang. Het effect is hetzelfde. Dit is geïllustreerd in Figuur 60.

```
In[11]:= f[x_Integer] /; x > 0 := "Groter dan 0"
         f[x_] := "Kleiner dan 0" /; x < 0 && IntegerQ[x]
         f[x_Integer] := 0
         f[x_] := "Geen getal"

         {f[-1], f[0], f[1], f["a"]}
Out[15]= {Kleiner dan 0, 0, Groter dan 0, Geen getal}
```

Figuur 60: Demonstratie complexe pattern matching in Wolfram.

De /; mag duidelijk op meerdere plaatsen staan. Ook is het mogelijk complexe boolse expressies te vormen met meerdere deelexpressies. De functie IntegerQ geeft een boolse waarde terug die aangeeft of x een Integer is of niet, en heeft dus exact dezelfde werking als de _Integer-notatie. Wolfram beidt dus tal van mogelijkheden om van Pattern matching gebruik te maken.
Pattern matching is verder niet enkel bruikbaar bij functiedefinities. Bepaalde functies zoals *Cases*, die de elementen uit een lijst filtert op basis van een opgegeven patroon, nemen patroon-definities als argument. Figuur 61 geeft dit weer.

```
In[22]:= Cases[{g[-1], g[0], g[1], g["a"]}, g[x_Integer] /; x < 0]
Out[22]= {g[-1]}
```

Figuur 61: Gebruik van pattern matching als argument voor een fucntie.

Pattern matching is in Wolfram dus meer dan enkel een controlestructuur bij het aanroepen van functies. Dit laat opnieuw de grote flexibiliteit van Wolfram zien. Een laatste opmerking hierbij is dat in Wolfram de speciale notatie ‘/.’ toelaat op basis van een patroon elementen in een lijst te vervangen door andere elementen. Eerst wordt het patroon gedefinieerd, waarna er een -> operator volgt. Na deze operator kan de programmeur een expressie definiëren die weergeeft waardoor de elementen die voldoen aan het patroon vervangen dienen te worden. Dit is ook een krachtig en elegant systeem, zoals Figuur 62 laat zien.

```
In[33]:= f[x_] := x + 1
         {g[-1], g[0], g[1], g["a"]} /. g[x_Integer] /; x < 0 → f[x]
Out[34]= {0, g[0], g[1], g[a]}
```

Figuur 62: Replace-operator in Wolfram.

5.1.6 Anonieme functies

Een andere functionele eigenschap die Wolfram kent is het concept van anonieme functies. In Wolfram heet dit soort functies 'pure functies'. Dit zijn functies die zijn gedeclareerd zonder gebonden te zijn aan een naam, en dus geen expressies zijn. Vandaar de naam 'pure functies'. Figuur 63 geeft een voorbeeld van een pure functie die 1 optelt bij zijn input.

```
In[1]:= (# + 1) &
```

Figuur 63: Pure functie in Wolfram.

Dit is de meest gangbare definitie van een pure functie. In de documentatie is ook een alternatieve definitie te vinden, die in dit onderzoek nooit is toegepast waardoor het geen zin heeft deze alternatieve methode nader toe te lichten. De syntax van een pure functie is zoals te zien in Figuur 63 zeer compact en symbolisch. Een pure functie is syntactisch een expressie die eindigt op een &. Deze expressie kan argumenten bevatten, die aangeduid zijn met #. Bovenstaande pure functie verwacht dus een enkel argument, en telt daar dan 1 bij op. Deze pure functie is nu toe te passen op een argument. Dit geeft Figuur 64.

```
In[2]:= (# + 1) &[50]
Out[2]= 51
```

Figuur 64: Pure functie toegepast op een argument.

Een pure functie kan ook meerdere argumenten hebben. In dat geval kan het eerste argument worden aangeduid als #1, het tweede als #2, enz. Figuur 65 geeft hier een voorbeeld van.

```
In[3]:= {#2, 1 + #1, #1 + #2} &[a, b]
Out[3]= {b, 1 + a, a + b}
```

Figuur 65: Pure functie met meerdere argumenten.

Het is belangrijk om het gedrag van Wolfram bij deze functie op te merken. Het valt op dat Wolfram geen foutmeldingen genereert, ondanks het feit dat letters niet bij elkaar op te tellen zijn. Dit is nogmaals een voorbeeld van de eigenzinnige interpretatie die Wolfram aan de code kan geven. Indien Wolfram een bepaalde expressie niet kan evalueren, geeft Wolfram gewoon de expressie zelf terug, in plaats van te crashen. Dit kan soms tot verwarrend gedrag van de functie leiden.

5.1.7 Functieapplicatie

Nu alles geweten is over de definitie van functies, is de volgende stap het toepassen ervan op data. Ook hier heeft Wolfram een eigen benadering voor, die veel compacter en stijlvoller is dan in veel andere talen. Uit bovenstaande is reeds duidelijk dat in de eerste plaats een functie kan toegepast worden op een set argumenten door de head van de functie te koppelen aan deze argumenten tussen vierkante haken. Een alternatieve notatie hiervoor is @. Figuur 66 laat deze beide methodes zien.

```
In[16]:= f[x_] := x + 1
         {f[1], f@1}
Out[17]= {2, 2}
```

Figuur 66: 2 methodes voor het toepassen van een functie.

Wat interessant is aan de tweede notatie, is dat door extra @-symbolen toe te voegen, het niveau waarop de functie wordt toegepast op de argumenten automatisch verhoogt. Dit is een zeer krachtig principe wat het toepassen van functies heel flexibel maakt, zoals Figuur 67 laat zien.

```
In[34]:= {g@{1, 2}, g @@ {1, 2}, g @@@ {{1}, {2, 3}}}
Out[34]= {g[{1, 2}], g[1, 2], {g[1], g[2, 3]}}
```

Figuur 67: Hoger-niveau toepassing van functies.

Door met het aantal @-symbolen te spelen is het dus mogelijk functies op eender welk niveau in een geneste lijst toe te passen. Wolfram herinterpreteert automatisch de elementen van de lijst en vormt ze automatisch om tot parameters voor de functie.
Naast gewoon toepassen is het ook eenvoudig functies te mappen op een lijst. Een mapping is de toepassing van de functie op elk element van de lijst. De notatie hiervoor is /@. Figuur 68 geeft weer hoe dit eruit ziet.

```
In[2]:= f/@{a, b, c, d}
Out[2]= {f[a], f[b], f[c], f[d]}
```

Figuur 68: Mapping van een functie op een lijst.

De functie *f* wordt zoals verwacht toegepast op alle elementen uit de lijst.
Ook mappings kunnen op verschillende niveaus worden uitgevoerd. De syntax hiervoor is wel minder elegant. Figuur 70 illustreert dit.

```
In[2]:= Map[g, {{1, 2}, {3, 4}}, {2}]
Out[2]= {{g[1], g[2]}, {g[3], g[4]}}
```

Figuur 69: Hoger-niveau-mapping in Wolfram.

Voor hoger-niveau mappings is dus de functie Map nodig die als argumenten een functie heeft en de lijst waarop de functie gemapt dient te worden, en als derde argument de lijst van niveaus waarop de functie gemapt dient te worden op de lijst. Het is dus mogelijk een functie ineens op meerdere niveaus te mappen. Figuur 70 geeft dit weer.

```
In[3]:= Map[g, {{1, 2}, {3, 4}}, {1, 2}]
Out[3]= {g[{g[1], g[2]}], g[{g[3], g[4]}]}
```

Figuur 70: Mapping op meerdere niveaus in Wolfram.

5.1.8 Operatorvormen

De culminatie van alle bovenstaande eigenschappen die de zeer stijlvolle, wiskundige, compacte, en flexibele eigenschappen van Wolfram aangeven is het gebruik van functies in hun zogenaamde 'operatorvorm'. Dit is een speciale manier waarop Wolfram toelaat bepaalde functies te gebruiken. Net als in de wiskunde staat in Wolfram de definitie van een functie in operatorvorm recht voor de data waarop ze moet toegepast worden, als een echte wiskundige operator. Dit volgt gedeeltelijk uit de expressie-syntax van Wolfram. In feite is de 'head' van elke expressie ook te zien als een operator, die is toegepast op de argumenten. Sommige Wolfram-functies gaan met dit idee echter nog een stap verder. Een goed voorbeeld hiervan is de *Select*-functie, zoals weergegeven in Figuur 71.

```
In[9]:= Select[#>7&][Table[i, {i, 10}]]
Out[9]= {8, 9, 10}
```

Figuur 71: Select-functie in operator-vorm in Wolfram.

De *Select*-functie filtert op basis van een boolse expressie alle elementen uit een lijst die voldoen aan die boolse expressie. Ondanks het feit dat de *Select*-functie al een argument heeft in de vorm van de boolse expressie, is het mogelijk de *Select*-functie inclusief boolse expressie rechtstreeks als operator toe te passen op een lijst. Ook hier komt de grote flexibiliteit en wiskundige zuiverheid van Wolfram naar boven. De programmeur kan deze stijlvolle en compacte notatie helemaal naar zijn hand zetten door expressies als weergegeven in Figuur 72 te definiëren.

```
In[35]:= filter := Select[#>7&]
         filter[Table[i, {i, 10}]]
Out[36]= {8, 9, 10}
```

Figuur 72: Operator-vorm stijlvoel toegepast in Wolfram.

In Figuur 72 is de functie *filter* gedefinieerd door gebruik te maken van de *Select*-functie. Nu is het mogelijk *filter* als eender welke andere functie toe te passen op een lijst die gewoon als argument meegegeven is, zonder dat er speciale mapping-syntax nodig is, wat in zowat alle andere programmeertalen wel het geval zou zijn. De *filter*-functie is dus een volwaardige operator. Hierin zit de kracht van deze operator-syntax. Dit is de ultieme illustratie van de enorme flexibiliteit en wiskundige stijl van Wolfram.

5.1.9 Dynamic

Wolfram bezit naast de wiskundig pure en stijlvolle functionele eigenschappen die hierboven beschreven zijn echter ook een aantal interessante object-georiënteerde eigenschappen. Zoals hierboven reeds aangehaald biedt Wolfram de mogelijkheid klassieke object-georiënteerde controlostructuren zoals if- for- en while-lussen te gebruiken, hoewel dit bij voorkeur vermeden dient te worden. Een andere object-georiënteerde eigenschap van Wolfram is dat variabelen in tegenstelling tot de meeste functionele talen van waarde kunnen wijzigen. Hierdoor zijn neveneffecten in Wolfram mogelijk. In de praktijk is het echter zelden toepasselijk dat Wolfram-code gebruik maakt van globale variabelen, en zijn data members waarvan object-georiënteerde talen veelvuldig gebruik maken onbestaande in Wolfram. Expressies dienen zo vaak mogelijk alle data die ze nodig hebben als argumenten mee te krijgen, wat het risico op ongewenste neveneffecten drastisch doet verminderen. Met deze eigenzinnige aanpak combineert Wolfram de voordelen van een functionele programmeerstijl met de voordelen van neveneffecten, wanneer die gewenst zijn.
Verder neemt Wolfram het concept van neveneffecten zoals zovele andere concepten binnen softwareontwikkeling een stap verder dan de meeste andere programmeertalen. In Wolfram is immers de zeer handige *Dynamic*-expressie ingebouwd. Door deze expressie rond eender welke andere expressie te plaatsen, kan de programmeur aangeven dat die laatste expressie steeds geherevalueerd dient te worden wanneer één van de variabelen binnen de *Dynamic*-expressie van waarde verandert. De *Dynamic*-expressie is dus een zeer stijlvolle en compacte manier om een event listener toe te voegen aan Wolfram-code. Figuur 73 geeft een voorbeeld van het gebruik van *Dynamic* in Wolfram.

```
In[53]:= x = 2;
         x + 1
         Dynamic[x + 1]
         x = 3;

Out[54]= 3

Out[55]= 4
```

Figuur 73: Toepassing van Dynamic in Wolfram.

Bovenstaande code illustreert de kracht van *Dynamic* zeer goed. Eerst is *x* gelijk aan 2. Het uitprinten van *x+1* levert dus 3. Initieel levert *Dynamic[x+1]* ook 3 als resultaat. Omdat naderhand de waarde van *x* echter wijzigt naar 3, herberekent *Dynamic* automatisch zijn inhoud op basis van de nieuwe waarde van *x*, waardoor uiteindelijk de waarde 4 te zien is als output voor *Dynamic[x+1]*. Het spreekt voor zich dat deze structuur zeer krachtig en bovendien stijlvol is. Het gebruik van *Dynamic* laat dus toe neveneffecten zeer strikt af te bakenen in Wolfram. De functie *Dynamic* heeft dan ook allerlei toepassingen naar vooral gebruikersinteractie toe. Later gaat dit hoofdstuk hier dieper op in.

5.1.10 Opties

Ten slotte heeft Wolfram ook de mogelijkheid om 'opties' te gebruiken voor heel wat functies. Een groot aantal ingebouwde functies in Wolfram is immers zeer flexibel, en kan worden gebruikt voor heel wat toepassingen. Daarom zijn in deze functies ook veel personalisatieopties ingebouwd. Deze kunnen optioneel worden meegegeven met de functie als extra argumenten. Elke optie heeft zijn eigen naam, waarna een '->' volgt. Achter dit symbool volgt de waarde die de programmeur wenst toe te kennen aan de optie. Dit is een eigenschap die Wolfram haalt uit het logische programmeerparadigma. Opties zijn immers logische regels waaraan de functie moet voldoen. De ingebouwde *Options*-functie laat toe van eender welke head van een bestaande expressie de mogelijke opties voor deze functie weer te geven. Figuur 74 geeft alle mogelijke opties weer voor de *Dynamic*-functie, verkregen door het toepassen van de *Options*-functie.

```
In[136]:= Options[Dynamic]
Out[136]= {Appearance → Automatic, AutoAction → False, Background → Automatic, BaseStyle → {}, CachedValue :→ Null,
  ContentPadding → True, ContinuousAction → True, DefaultBaseStyle → {}, Deinitialization :→ None,
  DestroyAfterEvaluation → False, DisplayFunction :→ Identity, Enabled → Automatic, Evaluator → Automatic,
  ImageSizeCache → Automatic, Initialization :→ None, BoxForm`RecursionLimit → 256, SaveDefinitions → False,
  Selectable → Automatic, ShrinkingDelay → 0., SingleEvaluation → False, SynchronousUpdating → True,
  TrackedSymbols → All, UpdateInterval → ∞}
```

Figuur 74: Opties voor de Dynamic-functie.

Zoals te zien zijn er heel wat opties voor de *Dynamic*-functie. In de documentatie is een uitgebreide toelichting te vinden over het effect van al deze opties. Tijdens dit onderzoek viel wel op dat veel opties eenvoudigweg niet werken in Wolfram, zelfs wanneer de documentatie expliciet aangeeft dat ze dat wel doen. Zo komt Figuur 75 rechtstreeks uit de documentatie.

```
In[1]:= DynamicModule[{x = "this is some text"}, Dynamic[x, BaseStyle → {FontColor → Orange}]]
Out[1]= this is some text
```

Figuur 75: BaseStyle-optie voor Dynamic zoals toegepaast in de documentatie [65].

Voor dit onderzoek is bovenstaand voorbeeld rechtstreeks gekopieerd en uitgevoerd in het Wolfram Development Platform. Dit levert Figuur 76.

```
In[2]:= DynamicModule[{x = "this is some text"}, Dynamic[x, BaseStyle :→ {FontColor :→ Orange}]]
Out[2]= this is some text
```

Figuur 76: Voorbeeld uit Figuur 75 toegepast in de eigen Wolfram-omgeving.

Vreemd genoeg laat Figuur 76 zien dat bij het letterlijk overkopiëren van het voorbeeld uit de documentatie het resultaat verschillend is van dat uit de documentatie, en de optie niet blijkt te werken. Hetzelfde probleem is voor een aantal andere opties van de *Dynamic*-functie ook vast te stellen. Dit is uiteraard een onaanvaardbaar probleem dat tot heel wat frustratie kan leiden. Het is dus aangewezen voorzichtig om te springen met opties in Wolfram. Ook is

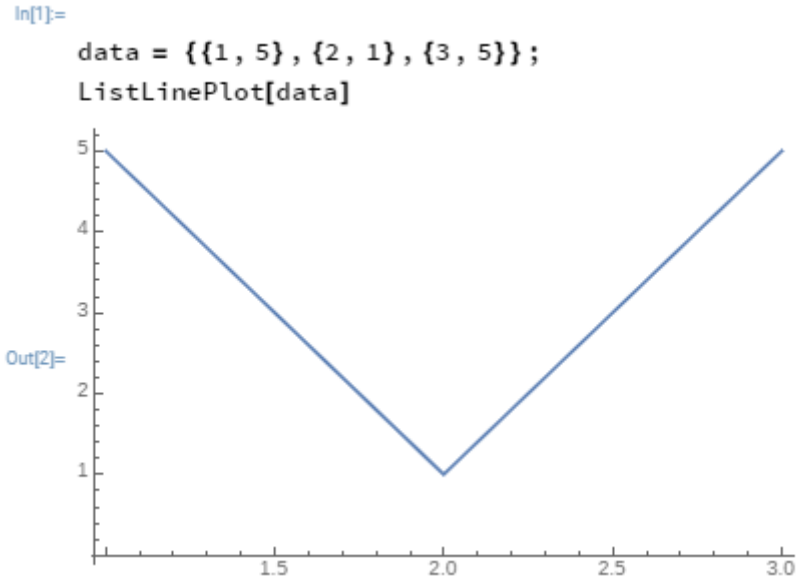
het belangrijk om te weten dat als code zich niet gedraagt zoals verwacht, de kans bestaat dat de fout niet bij de programmeur ligt, maar bij Wolfram.
Langs de andere kant moet opgemerkt worden dat Wolfram nog een vrij jonge taal is. Dit soort bugs zou in de toekomst sterk moeten afnemen.

5.2 Mogelijkheden

Zoals hierboven beschreven beschikt Wolfram over een zeer elegante en wiskundig zuivere syntax, die het mogelijk maakt een groot aantal programma's die in andere talen heel wat code zouden vergen zeer compact te schrijven. Wolfram is echter veel meer dan enkel een programmeertaal. Het is in feite de culminatie van een heel aantal technologiëen die het moederbedrijf van Wolfram ontwikkelde over de afgelopen decennia. Deze technologiëen hebben betrekking tot allerlei velden gaande van data-analyse voor wetenschappers tot de zoekmachine Wolfram|Alpha. De meeste van deze technologiëen hebben hun weg gevonden in de Wolfram-taal. Hierdoor beschikt Wolfram over een heel aantal unieke eigenschappen die geen enkele andere programmeertaal bezit. Hieronder een opsomming van de belangrijkste mogelijkheden die Wolfram biedt die verder gaan dan wat de meste andere programmeertalen kunnen aanbeiden.

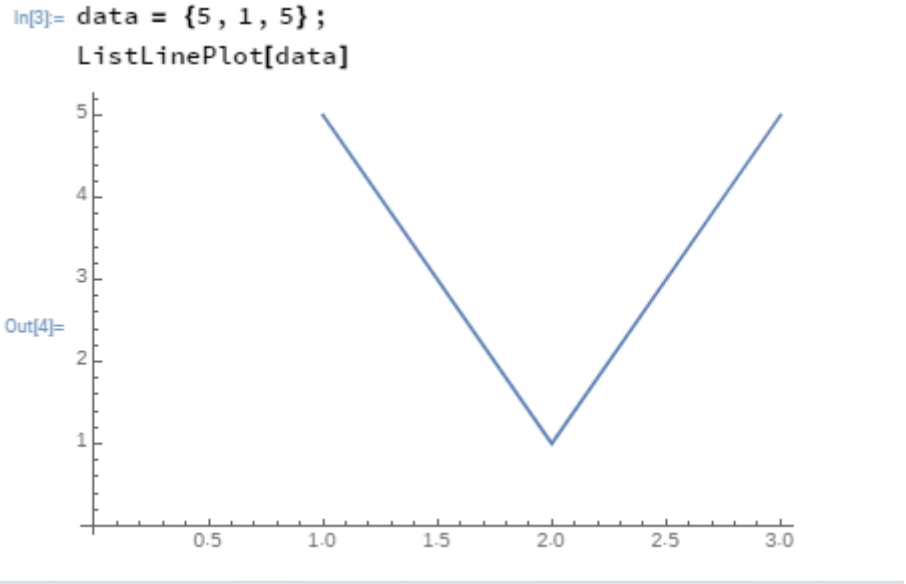
5.2.1 Datavisualisatie

Een eerste gebied waarop Wolfram uitblinkt is datavisualisatie. In Wolfram zijn een groot aantal functies ingebouwd die het zeer eenvoudig maken grafieken en zelfs 3D-figuren te genereren van een dataset. Al deze grafieken en figuren zijn binnen Wolfram nog steeds gewone expressies, waardoor ze ook heel flexibel verder verwerkt kunnen worden door andere Wolfram-code. Op al deze verschillende functies in detail ingaan zou veel te ver leiden voor deze masterproef. Deze paragraaf illustreert wel de algemene manier van werken van deze functies aan de hand van een aantal voorbeelden.
Een eerste eenvoudige functie die datavisualisatie toelaat is *ListLinePlot*. Deze functie vraagt een lijst van datapunten, en genereert op basis van die datapunten een mooie grafiek. In zijn eenvoudigste vorm is deze functie zeer compact, en is het gebruik ervan bijzonder intuïtief. Figuur 77 geeft een eenvoudig voorbeeld hiervan.



Figuur 77: Datavisualisatie aan de hand van ListLinePlot.

Het volstaat dus om data in de vorm van een lijst van x- en y-waarden mee te geven. De *ListLinePlot*-functie ondersteunt ook een heel aantal andere formaten voor de inputdata. Het is bijvoorbeeld ook mogelijk in plaats van een geneste lijst van datapunten mee te geven, gewoon een lijst van y-waardes mee te geven. De *ListLinePlot*-functie genereert voor de x-waarden dan een vector van gehele getallen, beginnende bij 1. Ook dit geeft weer de enorme flexibiliteit weer van Wolfram, die dus ook doorgedreven is in de datavisualisatie-functies. Figuur 78 geeft deze alternatieve manier om data door te geven weer.

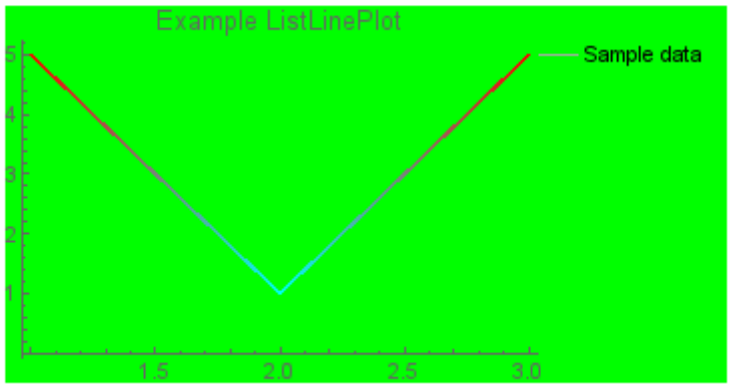


Figuur 78: Alternatieve manier om ListLinePlot te gebruiken.

De basisversie van ListLinePlot is op zich al indrukwekkend. Deze functie beschikt echter ook nog over een heel aantal opties waarmee de programmeur de weergave van de data volledig naar zijn hand kan zetten. Figuur 79 geeft een voorbeeld van enkele opties toegepast op de *ListLinePlot* uit vorige figuren.

```
In[32]:= data = {{1, 5}, {2, 1}, {3, 5}};
         ListLinePlot[data,
          PlotLabels → {"Sample data"},
          Background → Green,
          PlotLabel → "Example ListLinePlot",
          ColorFunction → Function[{x}, Hue[x]]]
```

Out[33]=



Figuur 79: Gepersonaliseerde ListLinePlot.

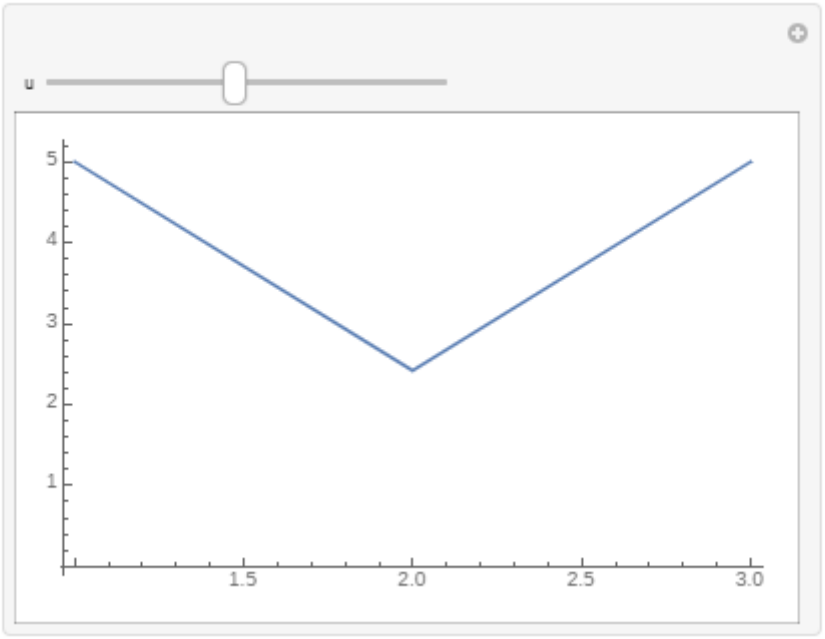
Het is dus mogelijk om op een compacte manier heel wat personalisatie toe te voegen aan de datavisualisatie-functies. Bovenstaande opties zijn uiteraard maar een kleine greep uit het totale aantal mogelijkheden. Een volledige lijst van alle opties voor de ListLinePlot-functie is te vinden op [66].

5.2.2 Data-interactie

Naast de vele datavisualisatie-functies is het ook mogelijk eenvoudig interactieve weergaves te maken van data. Hiervoor bestaan ook een aantal functies binnen en Wolfram. Deze paragraaf geeft net als de vorige een overzicht van de manier van werken van deze functies aan de hand van een aantal voorbeelden.
Een heel flexibele en veelgebruikte functie om aan data-interactie te doen in Wolfram is *Manipulate*. Deze functie is eenvoudig te gebruiken om bepaalde data te manipuleren aan de hand van een set van controls. Figuur 80 geeft een voorbeeld weer van het gebruik van *Manipulate*, in combinatie met de in vorige paragraaf besproken *ListLinePlot*.

```
In[35]:= Manipulate[ListLinePlot[{{1, 5}, {2, u}, {3, 5}}], {u, 0, 5}]
```

Out[35]=

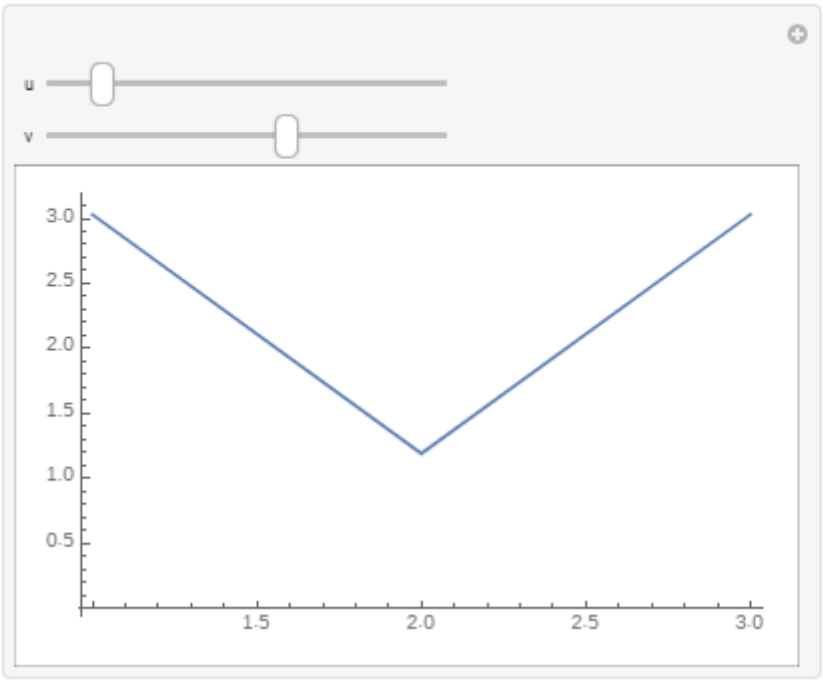


Figuur 80: Gebruik van Manipulate in combinatie met ListLinePlot.

De standaardversie van Manipulate is opnieuw zeer compact en intuïtief. De enige nodige inputs zijn hetgeen interactief weergegeven dient te worden, en de naam en het bereik van de interactieve variabelen. Deze kunnen eenvoudig rechtstreeks in de expressie voor de dynamisch weergegeven data gebruikt worden. Wolfram gebruikt onderliggend de functie *Dynamic* om de weer te geven expressie te herevalueren telkens de gebruiker iets verandert aan de interactieve variabele met behulp van de gegenereerde slider. In dit geval is de interactieve variabele *u* op ongeveer 2,5 ingesteld, waardoor de grafiek herberekend is met 2,5 als middelste y-waarde.
Het is eenvoudig mogelijk om meerdere interactieve variabelen te gebruiken in een *Manipulate*-expressie. Hiervoor volstaat dezelfde syntax als hierboven. Figuur 81 geeft hiervan een voorbeeld weer.

```
In[36]:= Manipulate[ListLinePlot[{{1, v}, {2, u}, {3, v}}], {u, 0, 5}, {v, 0, 5}]
```

Out[36]=



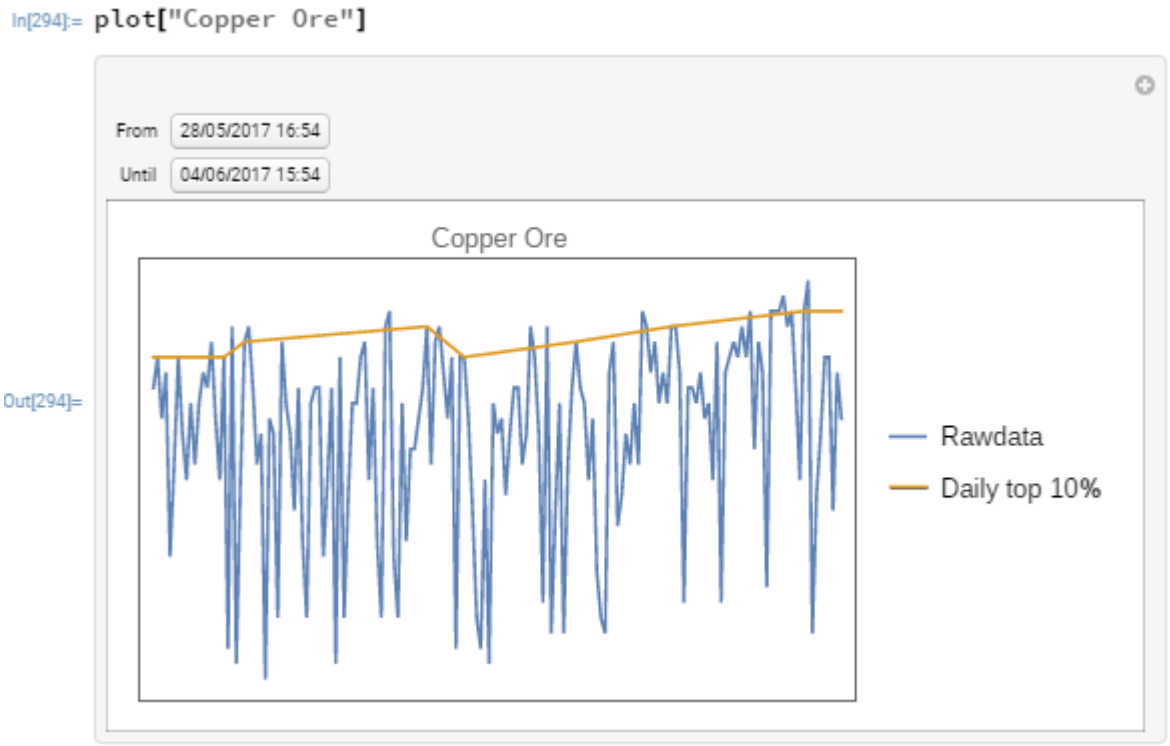
Figuur 81: Manipulate met meerdere variabelen.

De syntax in bovenstaande figuur spreekt voor zich. In dit geval is *u* ingesteld op ongeveer 1, en *v* op ongeveer 3. De *Manipulate* heeft de *ListLinePlot* mooi herberekend zoals verwacht. Merk op dat zoals eerder vermeld *Manipulate* inwendig gebruik maakt van *Dynamic* om de *ListLinePlot* steeds opnieuw te berekenen, ook al is *Dynamic* nergens in de expressie expliciet weergegeven. Het is ook mogelijk om toch expliciet *Dynamic* mee te geven bij een deel van de eerste expressie in *Manipulate*. Wolfram detecteert dit en berekent dan enkel het deel van de expressie binnen de *Dynamic*-functie opnieuw als de gebruiker de input verandert, in plaats van heel de expressie die als eerste argument met *Manipulate* meegegeven is. Voor complexe *Manipulates* kan dit heel wat rekentijd besparen.
Ook *Manipulate* heeft een heel aantal opties die toelaten zowat alles aan deze functie te personaliseren. Voor dit onderzoek is er danig geëxperimenteerd met deze opties. Zonder te veel in detail te treden over elke specifieke optie, geeft Figuur 82 een voorbeeld van een complexe *Manipulate* waarbij heel wat persoonlijke aanpassingen zijn toegevoegd.

```
fetchData := Import["http://thomasvandenabeele.no-ip.org/gw2dbapi/materials/logs", "RawJSON"]
fetchMats := Import["http://thomasvandenabeele.no-ip.org/gw2dbapi/materials", "RawJSON"]
fetchDataMat[name_String] := Import["http://thomasvandenabeele.no-ip.org/gw2dbapi/materials/logs/" <>
  URLEncode[name], "RawJSON"]
dataMat[data_, mat_String] := Flatten[#["logs"] & /@ Select[data, #["naam"] == mat &]]
listForm[data_] := ToExpression[{#["tijd"], #["prijs"]} & /@ data]
lowest[data_] := Min[data /. {t_, _} → t]
highest[data_] := Max[data /. {t_, _} → t]
split[data_, bounds_] := checkValue[data, #] & /@ bounds
data = listForm[dataMat[fetchData, "SilkScrap"]];
checkValue[data_, v_] := Select[First[#] < v && First[#] ≥ v - 24 &][data]
sort[split_] := Sort[#, (Last[#1] > Last[#2] &)] & /@ split
append[data_, top_] := Append[Prepend[top, {First[First[data]], Last[First[top]]}],
  {First[Last[data]], Last[Last[top]]}]
getTop10[data_] := append[data, #[[3]] & /@ sort[split[data, Table[i, {i, lowest[data] + 23, highest[data], 24}]]]]
interpollate[data_] := Union @@ (Table[{#1[[1]] + i, #1[[2]] + (#2[[2]] - #1[[2]]) / (#2[[1]] - #1[[1]]) * i},
   {i, 0, #2[[1]] - #1[[1]]}] & @@@ Partition[data, 2, 1])
top10[data_] := interpollate[getTop10[data]]
mapToDate[data_, min_Integer, start_] := {DatePlus[start, Quantity[(#[[1]] - min), "Hours"]], #[[2]]} & /@ data
mapToDate[data_] := mapToDate[data, lowest[data], LocalTime[ Brussels (city) ] - 1 wk + 1 h]
fetchMatNames := #["naam"] & /@ fetchMats
evaluateInput[in_String, list_] := MemberQ[list, in]
plot[name_String, data_, top10_] := Manipulate[With[{s = start, e = end},
   DateListPlot[Select[#[[1]] ≥ (s + 1 min) && #[[1]] ≤ (e + 1 min) &] /@ {data, top10},
    PlotLabel → name, PlotLegends → {"Rawdata", "Daily top 10%"}, AxesLabel → {"Tijd", "Prijs"}]],
 {{start, lowest[data], "From"}, # → DateString[#] & /@ (#[[1]] & /@ data[[;; ;; 12]]), ControlType → PopupMenu},
 {{end, highest[data], "Until"}, # → DateString[#] & /@ (#[[1]] & /@ data[[12 ;; ]][[;; ;; 12]]),
   ControlType → PopupMenu}]
plot[name_String, data_] := plot[name, mapToDate[data], mapToDate[top10[data]]]
plot[name_String /; evaluateInput[name, fetchMatNames]] := plot[name, listForm[fetchDataMat[name]]]
plot[name_String /;    !evaluateInput[name, fetchMatNames]] := "Not a valid input: " <> name
plot[mat] := "Not a valid input."
```

Figuur 82: Voorbeeld van een complexe Manipulate.

Bovenstaande code is behoorlijk uitgebreid en complex. Dit om te laten zien dat voor dit onderzoek Wolfram diepgaand bestudeerd is, en ook complexe programma's in deze taal geschreven zijn. De exacte werking van de code is niet echt van belang. Kort samengevat verzamelt deze code allerlei informatie over de prijsevolutie van een bepaald item over de afgelopen week aan de hand van een REST-API. Daarna past de Wolfram-code enkele functies toe om de data te bewerken en voor elke dag van de week de hoogste 10% van de metingen te bepalen, en in een aparte lijst te plaatsen. De lijst van meetpunten en berekende top 10%-punten worden dan aan een *Manipulate* doorgegeven, samen met de naam van het item (zie gemarkeerde functie). Deze *Manipulate* is sterk gepersonaliseerd. De start- en einddatum voor het weergeven van de data kunnen gekozen worden door de gebruiker. Dit aan de hand van popup-menu's, in plaats van de hierboven steeds gebruikte sliders. Voor het weergeven van de data zelf is opnieuw gebruik gemaakt van een *ListLinePlot*. Merk op dat pattern matching is toegepast om ervoor te zorgen dat de gebruiker enkel geldige items kan opvragen. Het resultaat van dit alles is weergegeven in Figuur 83.



Figuur 83: Resultaat uitvoeren complexe Manipulate-code uit Figuur 82.

Zoals hierboven te zien is het resultaat van deze complexe Manipulate opnieuw zeer mooi. Wolfram is dus bijzonder goed in visualisatie van data dankzij de symbolische en functionele opbouw van de taal, samen met de opties en de elegante manier waarop de taal omspringt met principes uit object-georiënteerd en logisch programmeren.
Merk op dat deze paragraaf enkel een overzicht gaf van enkele handige functies voor data-interactie binnen Wolfram. Er zijn nog veel meer functies speciaal voor dit doel ingebouwd in Wolfram, en de mogelijkheden zijn nagenoeg onbeperkt. Bovenstaande zou echter moeten volstaan om een idee te scheppen van de kracht van Wolfram voor dit soort toepassingen.

5.2.3 Natural language input

Een volgende heel interessante eigenschap van Wolfram die geen enkele andere programmeertaal bezit is de geïntegreerde zogenaamde 'natural language input'. Deze eigenschap laat programmeurs in Wolfram toe rechtstreeks in de code gewone schrijftaal te gebruiken om allerlei zaken aan te duiden. Wolfram interpreteert deze schrijftaal dan en injecteert met behulp van Wolfram|Alpha het element dat de programmeur (volgens Wolfram|Alpha) specifieerde. Deze natural language input is toepasbaar op een zeer uiteenlopend bereik aan entiteiten. Het enige wat de programmeur moet doen om van natural language input gebruik te maken in Wolfram is '=' schrijven voor een stuk schrijftaal. Bij de evaluatie van de code geeft Wolfram dan een voorstelling terug van hetgeen de programmeur waarschijnlijk bedoelde. Figuur 84 geeft een voorbeeld van het gebruik van deze natural language input.

Figuur 84: Natural language input in Wolfram.

Zoals te zien is bovenstaande syntax bijzonder stijlvol en krachtig. Helaas is deze natural language input nog geen volledige artificiële intelligentie, en is het niet moeilijk om op de beperkingen ervan te botsen. Figuur 85 illustreert dit.

Figuur 85: Beperkingen natural language input.

De input 'Red Ford Mustang' geeft net als de input 'Ford Mustang' een witte Ford Mustang terug. Dit geeft aan dat het gebruik van natural language input eerder beperkt is tot het opzoeken en analyseren van data, in plaats van het volledig dynamisch interpreteren van bepaalde input. Het is dus steeds aan te raden bij het gebruik van natural language input bij het programmeren eerst na te gaan of dit effectief het gewenste resultaat geeft, alvorens het te integreren in grote projecten.
Bovenstaande is nog maar een voorproevertje van wat mogelijk is met natural language input in Wolfram. Zo is het ook mogelijk een volledig rapport van Wolfram|Alpha te genereren over een heel aantal entiteiten, en dit alles in een enkele regel code. De syntax hiervoor is identiek aan die hierboven, maar dan met '==' in plaats van '='. Figuur 86 laat dit in actie zien.

In[1]:= Ford Mustang

Input interpretation:
2 017 Ford Mustang

Image:

Basic information:

division	Ford
type	sedan
price	$24 920 to $54 570 (MSRP) | $24 040 to $51 570 (invoice)
mileage	(14 to 22) mpg (city) | (21 to 31) mpg (highway)
passenger doors	2 doors
passenger capacity	4 people
engine type	V8 premium unleaded (3 trims) | I4 intercooled turbo premium unleaded (2 trims) | V6 regular unleaded (1 trim)
overall crash test rating	★★★★★
basic warranty	36 000 miles | 3 years
trims	GT | GT Premium | V6 | EcoBoost | EcoBoost Premium | Shelby GT350

Figuur 86: Wolfram-Alpha rapport op basis van natural language input.

Merk op dat bovenstaande figuur sterk ingekort is. Het eigenlijke rapport van Wolfram|Alpha is nog veel groter, en bevat allerhande informatie gaande tot zelfs de mogelijke leningen die kunnen worden afgesloten voor het aankopen van een Ford Mustang, en de verkrijgbare kleuren voor de carrosserie. Natural language input is dus zeer krachtig en heeft allerlei nuttige mogelijke toepassingen.

5.2.4 Real-world entities

Veel entiteiten uit de echte wereld zijn in Wolfram voorgesteld door een *Entity*-expressie. Deze entiteiten zijn binnen Wolfram dus expressies zoals alle andere. Het is mogelijk specifieke eigenschappen van deze entiteiten op te vragen. Om na te kijken of een bepaalde entiteit bestaat volstaat het in het Wolfram Development Platform ctrl+= te typen. Er ontstaat dan een inputveld waarin met natural language input een entiteit gespecifieerd kan worden. Daarna controleert Wolfram of deze entiteit bestaat met behulp van Wolfram|Alpha. Deze entiteiten blijken wel niet zo uitgebreid te zijn als de resultaten die te bekomen zijn met standaard natural language input. Zo is er voor 'Ford Mustang' geen entiteitsobject binnen Wolfram, zoals Figuur 87 weergeeft.

Figuur 87: Resultaat van het proberen omvormen van 'Ford Mustang' tot entiteit.

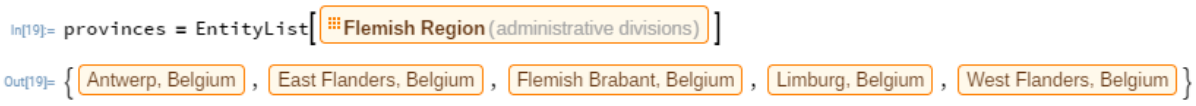
Er bestaan langs de andere kant wel entiteiten voor veel minder bekende zaken uit de echte wereld. Zo geeft de zoekterm 'USS Nimitz' wel een geldig resultaat, zoals Figuur 88 laat zien. Het lijkt dus dat de ondersteuning van deze eigenschap in Wolfram nogal arbitrair is. Ook hier is het weer aan te raden eerst apart te testen of de real-world entity wel bestaat, alvorens ze te integreren in grote projecten. Bovendien toont dit opnieuw aan dat de integratie van Wolfram|Alpha in Wolfram zeker niet onfeilbaar is.

Figuur 88: Geldige zoekterm voor real-world entities.

Zeer handige toepassingen voor real-world entiteiten zijn eenheden voor allerlei meetresultaten. Seconden, meters, inches etc. zijn namelijk ook geldige real-world entiteiten. Deze kunnen perfect geïntegreerd worden in code, waardoor berekeningen uitvoeren op data met deze eenheden een stuk eenvoudiger is dan in veel andere programmeertalen. Wolfram zorgt immers zelf dat alle low-level bewerkingen juist zijn. Een goed voorbeeld hiervan is de manier waarop Wolfram omgaat met datums. Figuur 89 illustreert dit.

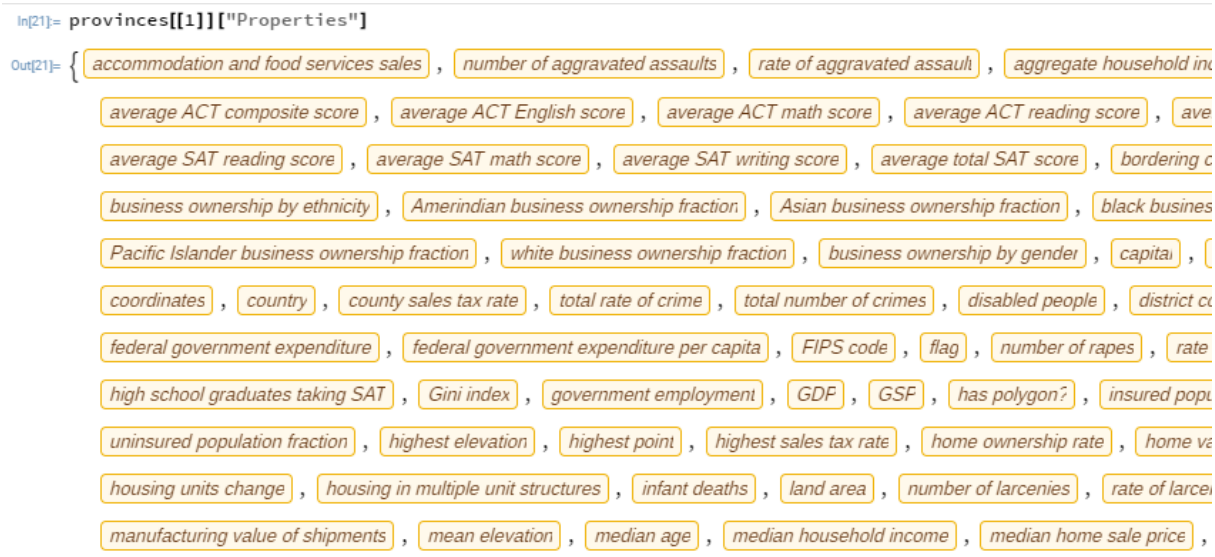
Figuur 89: Gebruik van real-world entities voor datums.

Zowel het *DateObject* als de *41 days* uit bovenstaande afbeelding zijn als real-world entiteiten opgevraagd met natural language input. Dit is veel eenvoudiger en minder foutgevoelig dan de werkwijze bij andere programmeertalen. Het resultaat van de bewerking is ook weer een handige datum-entiteit die gewoon verwerkt kan worden in de rest van de code.
Het is zelfs mogelijk een lijst van entiteiten op te vragen op basis van een zoekterm. Beschouw hiervoor Figuur 90.



Figuur 90: EntityList in Wolfram.

Als zoekterm werd 'flemish provinces' opgegeven in bovenstaande figuur. Het resultaat is een lijst van real-world entiteiten die de Vlaamse provincies voorstellen. Dit is uiteraard een zeer mooie eigenschap die heel wat toepassingen heeft.
Elke real-world entiteit heeft een aantal eigenschappen. Deze zijn allemaal rechtstreeks op te vragen van de entiteiten. Om erachter te komen welke eigenschappen een bepaalde entiteit bezit, volstaat het om het argument "Properties" aan de entiteit toe te voegen, zoals Figuur 91 demonstreert.



Figuur 91: Properties van 'Administrative Division' entiteiten.

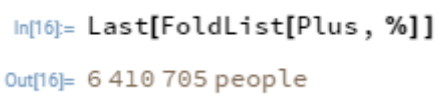
Merk op dat bovenstaande figuur slechts een klein deel van alle properties weergeeft. Het totale aantal properties van de meeste entiteiten is dus enorm. Deze properties maken allerlei interessante toepassingen mogelijk. Zo geeft Figuur 92 bijvoorbeeld de populatie van alle provincies van Vlaanderen weer. Merk wel op dat de properties uit de properties-lijst die hierboven weergegeven is niet steeds volledig overeenkomen met de werkelijke properties. Een voorbeeld hiervan is dat veel properties in de properties-lijst met een kleine letter geschreven zijn, terwijl Wolfram wel degelijk een hoofdletter verwacht bij het opvragen van de property. Met kleine letter werkt de code niet. Dit geeft aan dat de consistentie binnen Wolfram nog een stuk beter kan, en dat Wolfram nog geen mature technologie is.

Figuur 92: Populatie van Vlaanderen.

Dit alles is uiteraard indrukwekkend. Een opmerking hierbij is wel dat de properties van de real-world entiteiten niet altijd consistent of intuïtief zijn wat betreft de manier waarop Wolfram ze weergeeft. Een voorbeeld hiervan is gegeven in Figuur 93.

Figuur 93: Alternatieve manier om de populatie van Vlaanderen op te vragen.

Het opvragen van de populatie van de entiteit van Vlaanderen geeft het resultaat dus als een lijst van de populaties van alle provincies van Vlaanderen, in plaats van een enkel getal, wat logischer zou zijn. Dit bevestigt weer dat de real-world entiteiten niet zeer consistent zijn. Gelukkig is het door de zeer compacte en stijlvolle syntax van Wolfram eenvoudig om deze lijst te reduceren tot een enkel getal. Figuur 94 geeft dit weer.

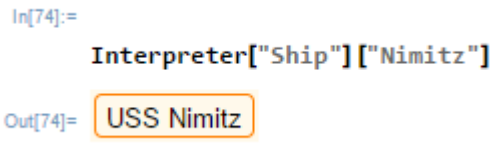


Figuur 94: Uiteindelijkke populatie van Vlaanderen.

De functie *FoldList* neemt als argumenten een functie en een lijst. De % wijst op de laatst berekende uitkomst, in dit geval de lijst van de populaties van alle provincies van Vlaanderen. *FoldList* past de functie die gespecifieerd is als eerste argument progressief toe op alle elementen van de lijst en vervangt dus elk element van de lijst door de toepassing van de gedefinieerde functie op dat element en alle voorgaande elementen. De *FoldList*-functie telt bij elk element in dit geval dus de som van alle voorgaande elementen op. Door het laatste element van de resulterende lijst te nemen is uiteindelijk de populatie van Vlaanderen gekend. Merk op dat de real-world entiteit '*people*' zich zonder problemen laat optellen door de functie *Plus*. Dit geeft weer de enorme flexibiliteit en het grote gebruiksgemak van Wolfram en de real-world entiteiten weer.

5.2.5 Interpreters

Verder bouwend op bovenstaande paragraaf bestaan er ook Interpreters in Wolfram die een zekere input kunnen omvormen tot de meest gepaste output van een bepaald type dat door de programmeur gespecifieerd is. Deze output kan eender welke expressie zijn, maar interpreters komen het beste tot hun recht in combinatie met real-world entiteiten. Een interpreter kan in operator-vorm worden toegepast op de input. Figuur 95 geeft hier een voorbeeld van weer.



Figuur 95: Voorbeeld van een Interpreter.

Een interpreter van het type "Ship" is dus in staat om de input "Nimitz" om te zetten naar de real-world entiteit 'USS Nimitz'.
Dankzij interpreters is het mogelijk om allerlei gegevens over een real-world entiteit op te vragen zonder ooit de entiteit expliciet op te vragen. Figuur 96 illustreert dit.

```
In[76]:= Entity["Ship"]["Properties"]
Out[76]= {beam overall, builder, builder country, class,
          crew, start date of operation, date commissioned,
          date laid down, date launched, depth,
          displacement, draught, historical names, image,
          length overall, maximum speed, name, operator,
          owners, total passengers, ports of registry,
          shipwreck, tonnage, total capacity, type}

In[84]:= EntityValue[Interpreter["Ship"]["Nimitz"], {"Type", "Builder",
           "BuilderCountry"}]
Out[84]= {{aircraft carrier}, Newport News Shipbuilding, United States}
```

Figuur 96: Allerlei data over USS Nimitz opgevraagd zonder ooit de entity zelf op te halen uit Wolfram|Alpha.

In bovenstaande figuur zijn eerst informatief alle eigenschappen van de real-world entiteiten van het type 'ship' in Wolfram weergegeven. Daaronder is te zien hoe door het gebruik van een interpreter verschillende eigenschappen van de real-world entiteit 'USS Nimitz' zijn weergegeven, zonder dat de entiteit 'USS Nimitz' ooit expliciet is aangemaakt. Dit is een elegante oplossing om zuiver aan de hand van Wolfram-code eigenschappen van entiteiten te achterhalen, zonder speciale input-velden te gebruiken binnen de IDE zoals dat voor alle voorgaande interactie met Wolfram|Alpha wel het geval was.

5.2.6 Forms

Bovenstaande interpreters zijn heel handig voor allerlei toepassingen. De toepassing waarbij interpreters het beste tot hun recht komen is echter gebruikersinteractie. Wolfram biedt een aantal mogelijkheden om interpreters toe te passen om intelligentie toe te voegen aan programma's. Een voor de hand liggende toepassing van interpreters voor gebruikersinteractie is in de vorm van intelligente web-forms. Dit betekent concreet dat Wolfram functies voorziet om web-forms te genereren die gekoppeld zijn aan een interpreter. Deze intelligente web-forms zijn binnen Wolfram aan te maken door de functie *FormObject* aan te roepen. Figuur 97 illustreert het gebruik van deze functie.

```
In[1]:= fo = FormObject["ship" → "Ship"]
```

Figuur 97: FormObject in Wolfram.

De code is opnieuw zeer elegant en compact. De string "*ship*" bepaalt de naam van een veld in het *FormObject*. De '->"Ship"'-syntax bepaalt dat voor het '*ship*'-veld een interpreter voor het type '*Ship*' van toepassing is. Wolfram zorgt er dan voor dat in het '*ship*'-veld enkel zaken kunnen worden ingegeven waarmee een schip geïdentificeerd kan worden.

Bovenstaand *FormObject* kan nu geïntegreerd worden in bepaalde ingebouwde Wolfram-functies die interactieve functionaliteit toevoegen aan het *FormObject*, waardoor gebruikersinteractie tot stand kan komen. Een voorbeeld van zo een functie is *FormPage*. Deze functie genereert op basis van de inhoud van het *FormObject* een pagina met inhoud die gekoppeld is aan het *FormObject*. Voor eenvoudige forms is het zelfs niet nodig om expliciet een *FormObject* te specifiëren. In dat geval kunnen de velden van het form rechtstreeks in de *FormPage*-functie gespecifieerd worden, op dezelfde manier zoals dat hierboven in de *FormObject*-functie gebeurde. Figuur 98 geeft dit weer.

```
FormPage[{"ship" → "Ship"}, ImageResize[#ship["Image"], 300]&]
```

Figuur 98: Formpage-functie in Wolfram.

Zoals hierboven te zien gebruikt de *FormPage*-functie in dit geval geen *FormObject*, maar zijn de velden rechtstreeks in de functie gespecifieerd. Het tweede argument van de *FormPage*-functie is een functie die bepaalt wat dient weergegeven te worden in de pagina onder het form, gebaseerd op de inhoud van het form. In dit geval is ervoor gekozen om een afbeelding van het door de gebruiker ingevulde schip weer te geven, en deze afbeelding te herschalen tot een breedte van 300 pixels. Dit is eenvoudig mogelijk, omdat dankzij de Interpreter '*ship*' een real-world entiteit van het type 'Ship' is, en dus alle eigenschappen uit Figuur 96 kunnen worden opgevraagd van de inhoud van het '*ship*'-veld. De velden van het form zijn vanuit deze functie eenvoudig aan te roepen als anonieme variabelen door gebruik te maken de veldnaam. Dit alles is zeer stijlvol en compact.

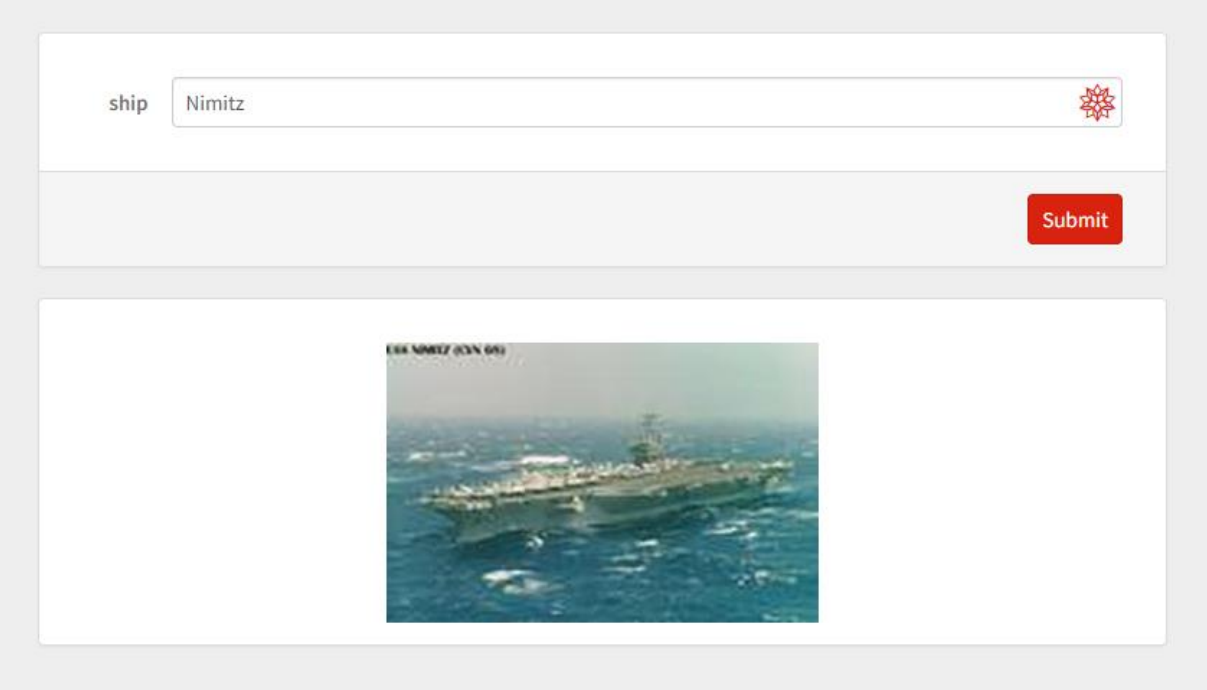
5.2.7 Cloud deployment

In vorige paragraaf is een interactieve *FormPage* aangemaakt met behulp van een interpreter. Deze dient nu naar de gebruikers gebracht te worden. Wolfram biedt de mogelijkheid om dit zeer eenvoudig te doen door de *FormPage* te deployen in de Wolfram Cloud. Ook dit is bijzonder evident in Wolfram. De enige toevoeging die nodig is aan de code uit Figuur 98, is deze code als argument mee te geven aan de ingebouwde functie *CloudDeploy*. Deze functie herkent de *FormPage*-expressie, en neemt alle nodige acties om deze code in de Wolfram Cloud te deployen, onder de naam van de gebruiker. Figuur 99 geeft dit weer.

```
In[1]:= CloudDeploy[FormPage[{"ship" → "Ship"}, ImageResize[#ship["Image"], 300]&]]
Out[1]= CloudObject[https://www.wolframcloud.com/objects/f99470c9-c9bc-4b92-88d4-b0add7b4415b]
```

Figuur 99: CloudDeploy-functie in Wolfram.

De *CloudDeploy*-functie is zoals alle andere functies in Wolfram zeer compact en stijlvol. De aangemaakte *FormPage* is nu in de cloud gedeployed, onder de naam van de programmeur. Iedereen kan gratis applicaties in de Wolfram Cloud deployen op deze manier, zolang het gebruik ervan maar beperkt is (niet meer dan 1000 calls per maand). De functie geeft een URL terug die rechtstreeks naar de gedeployde applicatie in de cloud leidt. Figuur 100 geeft het resultaat van deze functie weer.



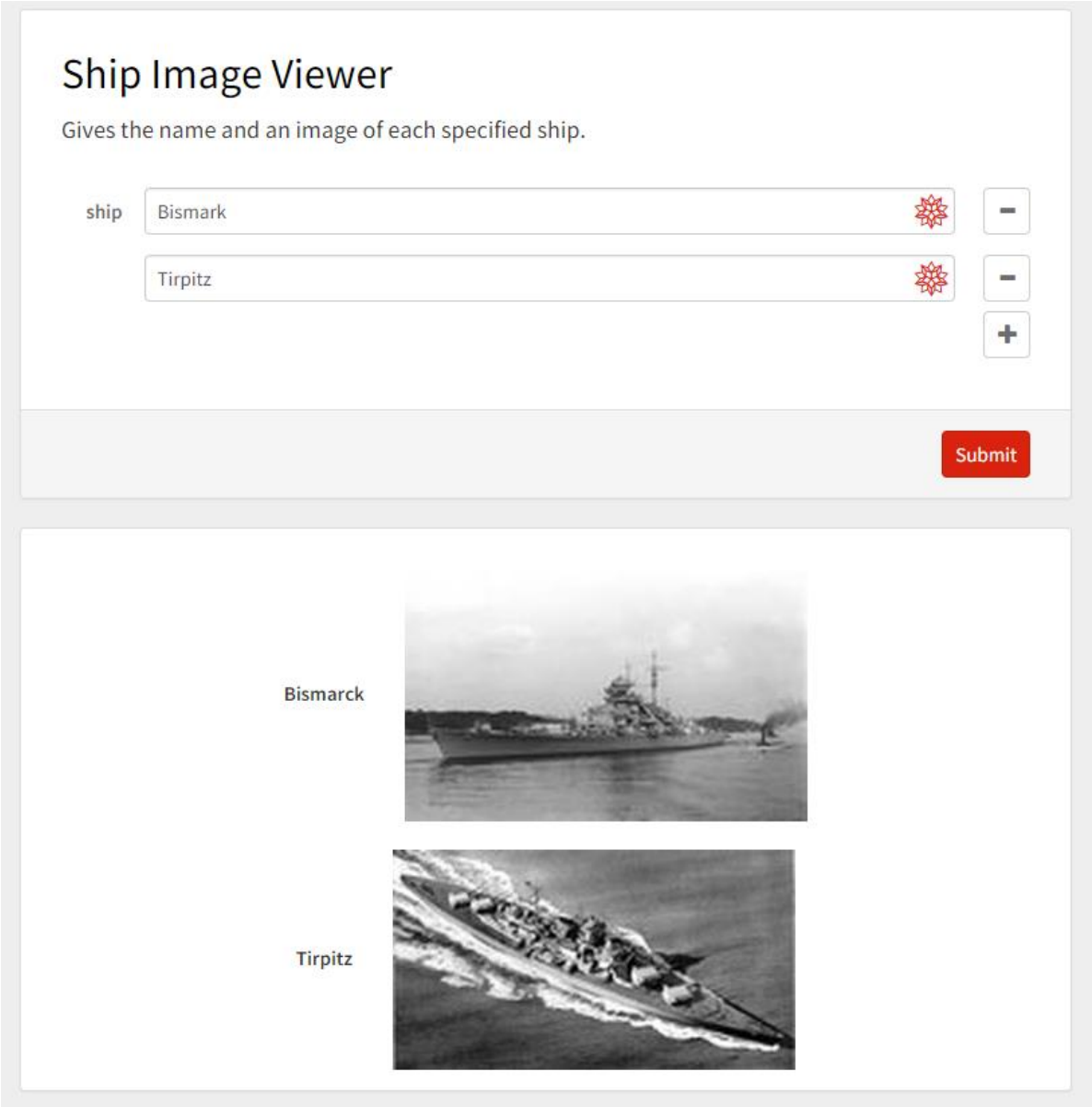
Figuur 100: Cloud-gedeployde FormPage uit Figuur 99.

Zoals hierboven te zien is een stijlvolle pagina gegenereerd met een form, en daaronder ruimte voor het resultaat van functie gedefinieerd in de *FormPage*. In Figuur 100 is reeds 'Nimitz' ingevuld door de gebruiker. De eenvoud van deze functie en het zeer mooie resultaat zijn grote pluspunten van Wolfram.
Uiteraard bieden de FormObject- en FormPage-functies heel wat mogelijkheden tot personalisatie, zodanig dat de programmeur de opmaak van de pagina volledig naar zijn hand kan zetten. Figuur 101 geeft een voorbeeld van een sterk opgemaakte *FormPage*, gebaseerd op dezelfde *FormPage* van hierboven.

```
In[2]:= fo = FormObject["ship" → RepeatingElement["Ship"],
          AppearanceRules → <|"Title" → "Ship Image Viewer",
          "Description" → "Gives the name and an image of each specified ship."|>];
        CloudDeploy[FormPage[fo, Column[(Row[{Style[#["Name"], FontWeight → "Bold"],
                ImageResize[#["Image"], 300]}, Spacer[10]]&) /@ #ship, Spacer[20]]&]]
Out[3]= CloudObject[https://www.wolframcloud.com/objects/cdadf6a2-0ee0-4ecd-a14a-65367fef2105]
```

Figuur 101: Gepersonaliseerde cloud-gedeployde FormPage-functie.

Bovenstaande code genereert een FormPage waarbij de gebruiker meerdere schepen tegelijk kan opgeven. Onder het form genereert de *FormPage*-functie voor elk opgegeven schip de naam en een afbeelding ervan. Ook zijn een titel en beschrijving toegevoegd aan het form. De syntax voor AppearanceRules van het FormObject is de Wolfram-versie van een associatieve array. Voor de rest is de code intuïtief. Ook hier valt de declaratieve natuur van Wolfram weer op. Nergens vraagt het lezen van de code een zware intellectuele inspanning. Het resultaat van bovenstaande code is gegeven in Figuur 102.



Figuur 102: Gepersonaliseerde FormPage.

Het resultaat van de code uit Figuur 101 mag er zeker zijn. In dit geval zijn de schepen 'Bismarck' en 'Tirpitz' weergegven. Merk op dat in het eerste geval de naam fout geschreven is in het form. Wolfram heeft hier blijkbaar geen problemen mee. Dit geeft aan dat de interpreters toch een zekere vorm van intelligentie hebben die verder gaat dan het zoeken in een grote database met namen van schepen.

5.2.8 Embedden van Wolfram-code in andere programma's

De Wolfram Cloud is heel flexibel en eenvoudig te gebruiken vanuit Wolfram. Het is echter mogelijk om nog een stap verder te gaan wat betreft deployment van Wolfram-code. Zo is het mogelijk alle code die gedeployed is in de Wolfram Cloud eenvoudig te embedden in andere webpagina's. Ook hier is in Wolfram een zeer compacte en intuïtieve functie voor, namelijk *EmbedCode*. Deze functie kan op elke *CloudDeploy*-expressie worden toegepast, en genereert code die toelaat de cloud-gedeployde expressie te integreren in eender welke andere html-pagina. Figuur 103 laat dit zien toegepast op de *FormPage* uit Figuur 101.

```
In[1]:= fo = FormObject["ship" → RepeatingElement["Ship"],
          AppearanceRules → <|"Title" → "Ship Image Viewer",
          "Description" → "Gives the name and an image of each specified ship."|>];
        EmbedCode[CloudDeploy[FormPage[fo, Column[(Row[{Style[#["Name"], FontWeight → "Bold"],
                ImageResize[#["Image"], 300]}, Spacer[10]]&) /@ #ship, Spacer[20]]&]]]
```

Out[2]=

Embeddable Code

Use the code below to call the Wolfram Cloud function from HTML:

Code Download Code

```
<iframe src="https://www.wolframcloud.com/objects/e5bf8fb1-2e05-4cfc-a108-d28771790a46?_embed=iframe"
 width="600" height="800" />
```

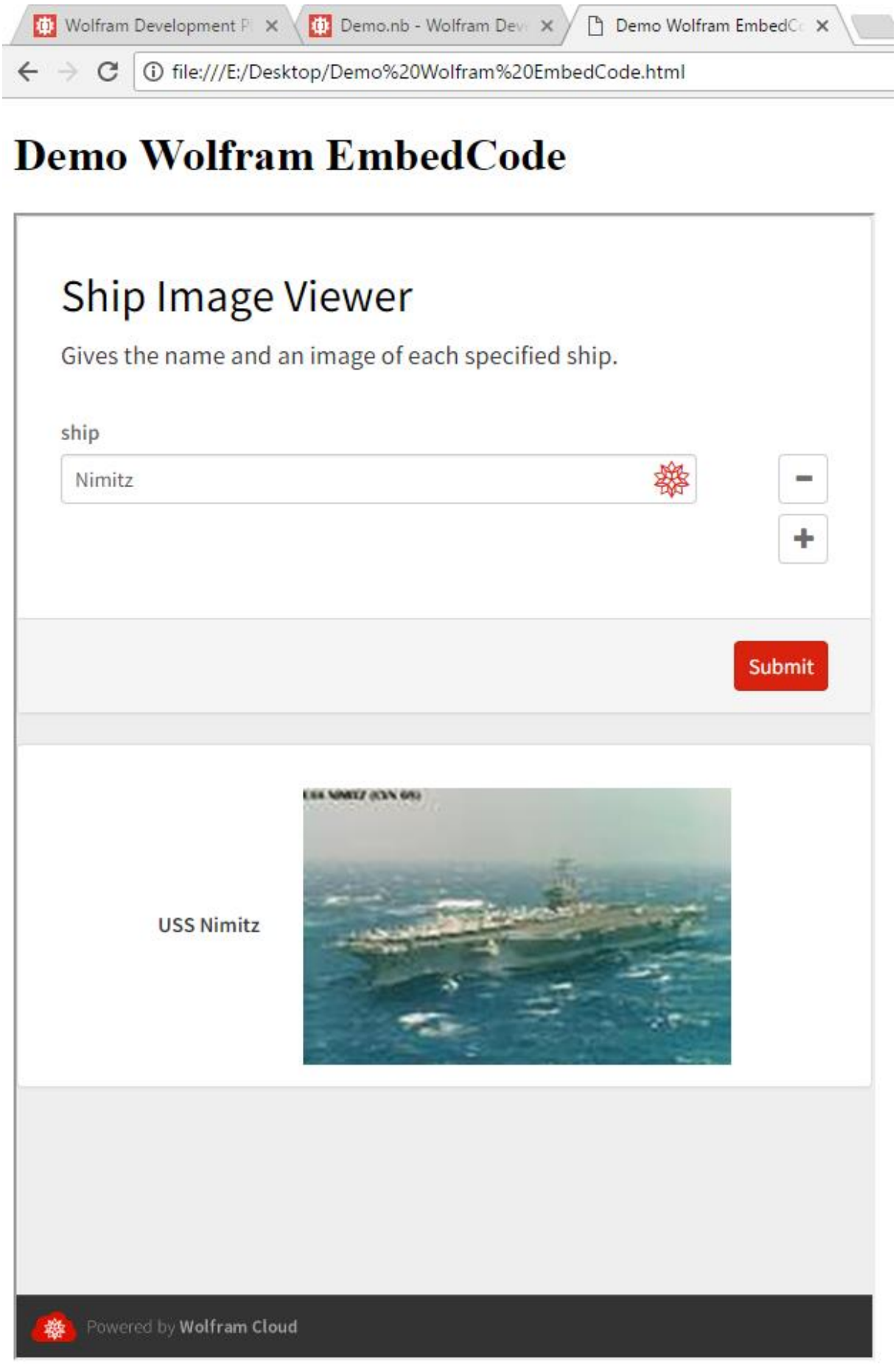
Figuur 103: EmbedCode toegepast op de code uit Figuur 101.

In Figuur 103 is te zien dat *EmbedCode* een iframe genereert dat verwijst naar de in de Wolfram Cloud gedeployde *FormPage*. Door dit iframe te integreren in eender welke html-pagina, ontstaat op deze pagina een venster met daarin de *FormPage*. Figuur 104 geeft een zeer eenvoudige html-pagina weer die gebruikt is in dit onderzoek om de werking van de EmbedCode-functie in Wolfram te testen.

```
<!DOCTYPE html>

<h1>Demo Wolfram EmbedCode</h1>
 <iframe
 src="https://www.wolframcloud.com/objects/521a2a5b-5fd1-4259-8250-c23867469319?_embed=iframe"
 width="600"
 height="800"/>
```

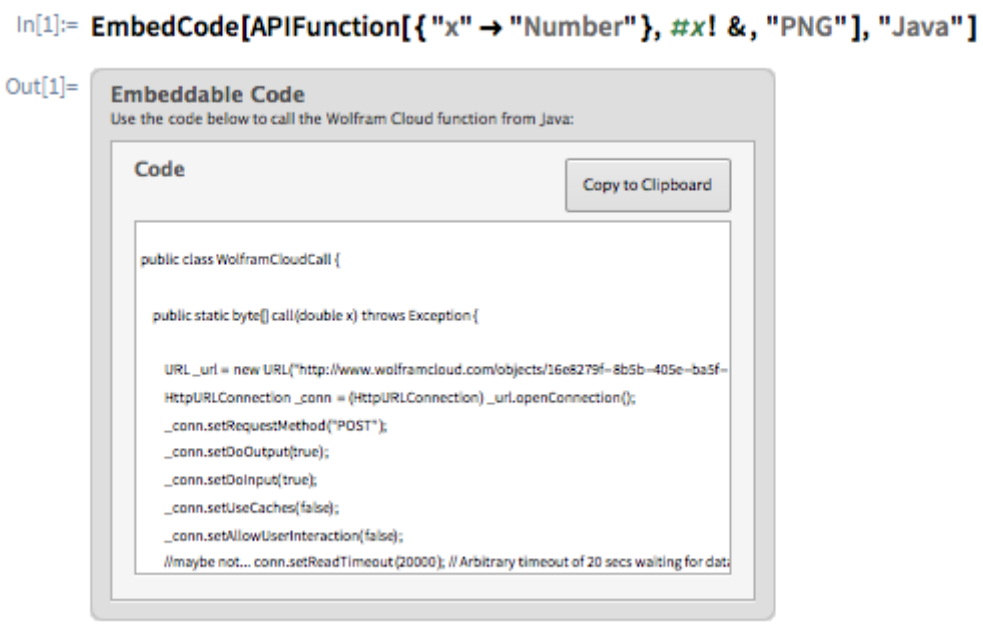
Figuur 104: Html-pagina met demo voor EmbedCode in Wolfram.

Het openen van bovenstaande html-pagina in de browser levert Figuur 105.



Figuur 105: Resultaat EmbedCode in de browser.

De EmbedCode-functie blijkt zeer goed te werken voor deze toepassing. Wel valt op dat alle Wolfram-code wordt uitgevoerd op de Wolfram Cloud, en het dus niet rechtstreeks mogelijk is om de resultaten ervan op te vragen met de hierboven beschreven werkwijze. Wel is het mogelijk om een API te schrijven die gebruik maakt van dezelfde Interpreter als bovenstaande *FormPage*, en via die API informatie op te vragen. Ook hiervoor bestaat in Wolfram een compacte en stijlvolle functie, namelijk *APIFunction*. Deze functie vraagt net als *FormPage* een aantal parameters op, waaraan types kunnen gekoppeld worden. Deze keer worden de parameters echter rechtstreeks in de URL meegegeven, zoals bij elke REST-API. Verder vraagt de *APIFunction*-functie een functie die de opgegeven parameters verwerkt tot een antwoord, opnieuw net zoals bij de *FormPage*-functie. De *APIFunction* kan het antwoord in allerlei formaten versturen, waaronder JSON, PNG, …
Behalve in html-pagina's, valt Wolfram ook eenvoudig te embedden in andere applicaties in andere programmeertalen met behulp van de *EmbedCode*-functie. Deze functie genereert steeds alle code die hiervoor nodig is, en neemt indien nodig nog extra stappen zodanig dat de integratie zo eenvoudig mogelijk is voor de programmeur. Figuur 106 geeft dit weer voor een eenvoudige API in Java.



Figuur 106: API in Java ge-embed met behulp van EmbedCode.

De EmbedCode-functie heeft als extra argument "*Java*" gekregen. Indien geen argument is opgegeven genereert EmbedCode steeds integratiecode voor html-pagina's, zoals hierboven reeds toegelicht. In het geval van Java genereert de *EmbedCode*-functie een volledige Java-klasse die kan worden toegevoegd aan een Java-project. Met deze klasse is het mogelijk de cloud-gedepoyde *APIFunction*-expressie aan te spreken. Het is duidelijk dat EmbedCode de API dus ook rechtstreeks in de Wolfram Cloud deployt, ook al is dit niet expliciet gespecifieerd in de code.
Niet zomaar elke expressie kan in elke andere omgeving worden ge-embed. EmbedCode is slechts toepasbaar op een beperkt aantal functies. Welke functies kunnen worden ge-embed

en welke niet is bovendien afhankelijk van de omgeving. Op [67] is in detail het gebruik van *EmbedCode* toegelicht, en is een link naar alle ondersteunde omgevingen te vinden. De beperkingen van *EmbedCode* zijn enerzijds begrijpelijk, maar anderzijds jammer. Dit geeft aan dat de grote flexibiliteit en het grote programmeergemak van Wolfram ten koste gaat van mogelijke toepassingsgebieden. Andere talen zouden immers meestal niet expliciet verbieden bepaalde zaken te combineren, maar eerder de programmeur verplichten zelf veel meer code te schrijven op een lager niveau om de gewenste toepassing toch mogelijk te maken. Wolfram kiest hier niet voor en houdt vast aan zijn mooie declaratieve stijl door alles wat buiten het eenvoudig haalbare bereik valt expliciet te verbieden.
Naast *EmbedCode* zijn er nog andere mogelijkheden om Wolfram in andere talen te integreren. Aangezien EmbedCode echter gebaseerd is op de Wolfram Cloud en de focus van deze masterproef cloud computing is, is *EmbedCode* in de context van deze mastertroef de te verkiezen optie.

5.2.9 Integratie van andere programmeertalen in Wolfram

Naast de mogelijkheden die Wolfram biedt om Wolfram-code te integreren in andere omgevingen, is het ook mogelijk code geschreven in andere talen rechtstreeks aan te spreken in Wolfram. Hiervoor zijn een een heel aantal bibliotheken geschreven. De talen waarbij dit mogelijk is zijn C/C++, Java, en .NET. Aan de basis van deze integratiemogelijkheden ligt het Wolfram Symbolic Transfer Protocol (WSTP), dat speciaal hiervoor ontwikkeld is [68]. Omdat dit onderzoek ook Java en .NET bestudeert, loont het de moeite om nader te bekijken hoe integratie van deze talen in Wolfram precies in zijn werk gaat.

5.2.9.1 Java

Voor de integratie van Java in Wolfram is de J/Link bibliotheek geschreven door de ontwikkelaars van Wolfram. Deze is gebaseerd op het WSTP, en is zeer eenvoudig te gebruiken. Volledige gedetailleerde documentatie van deze bibliotheek is terug te vinden op [69]. Deze paragraaf behandelt enkel de belangrijkste eigenschappen ervan die gekend moeten zijn om de bibliotheek correct te gebruiken.
De eerste stap in het proces om Java te kunnen integreren in Wolfram is het toevoegen van de J/Link-bibliotheek aan de Wolfram-code. Dit kan aan de hand van een *Needs*-functie. Dit is in feite hetzelfde als een *include* of *import*-expressie in andere programmeertalen, en behoeft verder geen toelichting.
De volgende stap is het starten van de JRE. Het Java-programma draait immers volledig gescheiden van Wolfram in de JRE. Wolfram communiceert enkel met de JRE om zo resultaten uit het Java-programma te bekomen. Alle berekeningen binnen het Java-programma zijn dus volledig door Java zelf beheerd. Het commando om de JRE op te starten is *InstallJava* of *ReinstallJava*. *InstallJava* start de JRE enkel bij de eerste aanroep van deze functie. *ReinstallJava* garandeert dat bij elke aanroep de JRE wordt afgesloten en opnieuw opgestart. Dit onderzoek maakt steeds gebruik van *ReinstallJava*.
Bij het starten van de JRE is het belangrijk dat deze weet waar ze moet zoeken naar alle Java-klassen op de pc. Indien geen argumenten zijn meegegeven met *InstallJava* of *ReinstallJava*,

weet de JRE niet waar te zoeken naar extra klassen en zijn alleen de standaard klassen uit de Java-SDK zelf bruikbaar. Voor het uitvoeren van een Java-programma is het dus belangrijk het pad naar het programma mee te geven aan de *(Re)installJava*-functie. Figuur 107 geeft een voorbeeld van hoe dit eruitziet voor een willekeurig programma op de lokale pc.

```
Needs["JLink`"]
ReinstallJava[ClassPath → "E:\\GW2\\GW2App\\dist\\GW2App.jar"]
```

Figuur 107: Opstarten van de JRE met J/Link.

Nu de JRE is opgestart met het juiste pad naar het programma, is het mogelijk klassen uit het gespecifieerde programma in te laden. Hiervoor dient de *LoadJavaClass*-functie. Deze functie verwacht als argument de naam van de in te laden klasse, inclusief de volledige naam van het package waarin deze klasse zich bevindt. Eens dit gebeurd is, is het mogelijk binnen Wolfram een overzicht op te vragen van alle methodes in de ingeladen klasse. Figuur 108 geeft hier een voorbeeld van weer.

```
In[51]:= Needs["JLink`"]
         ReinstallJava[ClassPath → "E:\\GW2\\GW2App\\dist\\GW2App.jar"];
         gw2db = LoadJavaClass["gw2app.DB", StaticsVisible → True];
         Methods[gw2db]

Out[54]//TableForm=
         boolean equals(Object)
         static java.util.ArrayList getBooleans(String) throws java.sql.SQLException
         static boolean getBoolean(String) throws java.sql.SQLException
         Class getClass()
         static java.util.ArrayList getDoubles(String) throws java.sql.SQLException
         static double getDouble(String) throws java.sql.SQLException
         static java.util.ArrayList getInts(String) throws java.sql.SQLException
         static int getInt(String)
         static java.sql.ResultSet getResultSet(String) throws java.sql.SQLException
         static java.util.ArrayList getStrings(String) throws java.sql.SQLException
         static String getString(String)
         int hashCode()
         void notify()
         void notifyAll()
         String toString()
         static void updateNoLog(String) throws java.sql.SQLException
         static void update(String) throws java.sql.SQLException
         void wait(long, int) throws InterruptedException
         void wait(long) throws InterruptedException
         void wait() throws InterruptedException
```

Figuur 108: Overzicht methodes van ingeladen Java-klasse in Wolfram.

De optie *StaticsVisible* laat toe static methodes en datamembers eenvoudiger aan te spreken. Static methodes kunnen nu de klasse is ingeladen rechtstreeks worden aangeroepen vanuit Wolfram alsof ze Wolfram-expressies zouden zijn. Parameters worden in vierkante haken meegegeven. Figuur 109 toont het aanroepen van de statische methode getInts uit bovenstaande methodenlijst.

```
in: Needs["JLink`"]
    ReinstallJava[ClassPath → "E:\\GW2\\GW2App\\dist\\GW2App.jar"];
    gw2db = LoadJavaClass["gw2app.DB", StaticsVisible → True];
    button = LoadJavaClass["java.awt.Button"];
    getInts["SELECT gw2_id FROM Items"]


out: « JavaObject[java.util.ArrayList] »
```

Figuur 109: Aanroepen statische methode Java vanuit Wolfram.

Het resultaat van deze statische methode is zoals in de methodenlijst reeds te zien is een ArrayList. Wolfram kent dit type uiteraard niet. De weergave voor alle complexe types in Wolfram is van bovenstaande vorm. Het is uiteraard mogelijk om op deze objecten nog functies toe te passen. Deze objecten dienen wel strikt beschouwd te worden als Java-objecten. Geen enkele Wolfram-functie die niet tot J/Link behoort kan iets aanvangen met de *JavaObjects* waarmee J/Link complexe Java-objecten voorstelt binnen Wolfram. Het is dus belangrijk complexe types te ontleden tot eenvoudige types die Wolfram wel kan begrijpen. Zo is het wel mogelijk om in Wolfram te werken met resultaten van berekeningen van het Java-programma. Tabel 1 geeft een overzicht van alle Java-types die Wolfram herkent, en het Wolfram-type waarmee deze Java-types overeenkomen.

Tabel 1: Overzicht van de omzetting van alle Java-types naar Wolfram-types.

Java-type	**Wolfram-type**
byte, char, short, int, long, Byte, Character, Short, Integer, Long, BigInteger	Integer
float, double, Float, Double, BigDecimal	Real
boolean	True of False
String	String
array	List
Object	JavaObject
Null	Null

Het is dus belangrijk om elk Java-object om te vormen naar één van de bovenstaande types om er bewerkingen mee te kunnen doen in Wolfram. Het is belangrijk om hier rekening mee te houden bij het schrijven van het Java-programma en functies die vanuit Wolfram aangeroepen worden steeds één van deze types (behalve Object natuurlijk) te laten teruggeven, of op zijn minst een methode te voorzien die alle informatie die Wolfram nodig heeft van een klasse/object kan omvormen tot één van de bovenstaande types.
Een laatste belangrijk aspect dat besproken dient te worden is natuurlijk het aanmaken van nieuwe Java-objecten en het aanroepen van methodes van deze objecten. Nu is het mogelijk om met de functie *JavaNew* een nieuw object aan te maken in Java. Figuur 110 geeft hiervan een voorbeeld weer.

```
JavaNew["gw2app.HoofdVenster"]
JavaShow[%]
```

Figuur 110: Aanmaken van een nieuw Java-object in Wolfram.

Bovenstaand voorbeeld maakt een JFrame aan. Om dit JFrame weer te geven moet nog extra de *JavaShow*-functie opgeroepen worden met als argument het aangemaakte *JFrame*.
Van aangemaakte Java-objecten is het uiteraard mogelijk public datamembers op te vragen, en ook public methodes op te roepen. Dit gebeurt in Wolfram met de @-operator. Een '@' in Wolfram komt dus overeen met een '.' in Java wat betreft J/Link. Ook de parameters worden in Wolfram meegegeven tussen vierkante in plaats van ronde haken, om overeen te komen met de Wolfram-syntax. Figuur 111 geeft hiervan een voorbeeld weer.

```
Needs["JLink`"]
ReinstallJava[
  ClassPath → "E:\\Dropbox\\School\\MasterProef\\Triangles\\dist\\Triangles.jar"];
t = JavaNew["triangles.PascalTriangle", 5];
t@getPoints[]
{{1}, {1, 1}, {1, 2, 1}, {1, 3, 3, 1}, {1, 4, 6, 4, 1}}
```

Figuur 111: Opvragen methode van een Java-object in J/Link.

Bovenstaand voorbeeld roept een methode op van een aangemaakt Java-object vanuit Wolfram. Deze Java-methode geeft een 2D-Array van Integers terug. Wolfram herkent dit type, zoals aangegeven in Tabel 1, en zet het automatisch om naar een geneste Wolfram-lijst van Integers. Voor methodes die de types uit Tabel 1 teruggeven is het aanroepen van Java-code vanuit Wolfram dus nagenoeg equivalent als het aanroepen van Java-code vanuit andere Java-code. De J/Link bibliotheek is dus zeer gebruiksvriendelijk.
Een belangrijke opmerking bij het gebruik van J/Link is dat deze bibliotheek enkel optimaal werkt in combinatie met een Desktop-versie van Wolfram. In de Wolfram Cloud is het immers niet toegelaten om zelf geschreven Java-programma's in te laden met J/Link. De reden hiervoor is waarschijnlijk veiligheid-gerelateerd. Iedereen zomaar toelaten zijn eigen Java-programma's uit te voeren op de servers van Wolfram is immers een groot beveiligingsrisico. J/Link is in de Wolfram Cloud wel te gebruiken voor alle klassen die Java standaard aanbiedt in de SDK. Het nut hiervan is echter uiteraard wel beperkt.

5.2.9.2 .NET

Ook voor .NET bestaat er een bibliotheek die toelaat C#-code uit te voeren op de pc, en de resultaten hiervan te communiceren met Wolfram. Het aanroepen gebeurt net als bij J/Link vanuit Wolfram. De bibliotheek die dit toelaat voor .NET heet .NET/Link. Volledige documentatie voor deze bibliotheek is te vinden op [70]. Deze bibliotheek is echter sterk gelijkend op J/Link. Het enige noemenswaardige verschil is dat .NET/Link bij het starten van de runtime niet vraagt naar het pad van de te gebruiken klassen zoals J/Link dat doet. In plaast daarvan dienen de te gebruiken .NET-assemblies één voor één ingeladen te worden nadat de runtime gestart is. Figuur 112 geeft hier een voorbeeld van weer.

```
Needs["NETLink`"]
ReinstallNET[];
LoadNETAssembly[
"E:\\Dropbox\\School\\MasterProef\\DotNet\\DotNetTriangles\\DotNetTriangles\\bin\\
Release\\DotNetTriangles.exe"];
LoadNETAssembly[
"E:\\Dropbox\\School\\MasterProef\\DotNet\\FSTriangles\\FSTriangles\\bin\\Release\\
FSTriangles.exe"];
```

Figuur 112: Opstarten van .NET runtime met .NET/Link en inladen van assemblies.

Een assembly in .NET komt in feite overeen met een programma of bibliotheek in Java. Met de functie *LoadedNETAssemblies*[] is het mogelijk alle ingeladen assemblies in Wolfram weer te geven. Verder zijn er zoals eerder vermeld maar weinig verschillen tussen .NET/Link en J/Link wat betreft gebruikswijze, zeker niet voor basistoepassingen. Daarom heeft het niet veel zin .NET/Link even gedetailleerd te bespreken als J/Link. De werkwijze beschreven voor J/Link is dus voor eenvoudige toepassingen over te nemen voor .NET/Link.

5.3 Beperkingen

Zoals uit bovenstaande duidelijk is, is Wolfram een zeer krachtig platform voor allerlei toepassingen, met tal van interessante eigenschappen. Dit alles is verpakt in een zeer intuïtieve, gebruiksvriendelijke en compacte syntax. Helaas zijn er ook een aantal nadelen verbonden aan Wolfram, waarvan sommige zelfs rechtstreeks volgen uit de voordelen van de taal. Hieronder een overzicht van de belangrijkste beperkingen van Wolfram die tijdens dit onderzoek aan het licht zijn gekomen.

5.3.1 Moeilijk in te schatten evaluatie en control flow

Doordat Wolfram declaratief is en bovendien gebruik maakt van dynamic typing is het soms niet duidelijk op welke manier Wolfram bepaalde code interpreteert. Soms is het zelfs onmogelijk om dit te voorspellen op basis van de code zelf, en is de enige manier om hierachter te komen de code uit te voeren en te kijken wat het resultaat is. Hieronder enkele voorbeelden die dit illustreren.
Een eerste voorbeeld dat bovenstaand probleem goed weergeeft is gebaseerd op de eerder in dit hoofdstuk beschreven zeer krachtige en flexibele pattern matching in Wolfram. Deze pattern-matching gebruiken als type-onderscheiding is soms gevaarlijk, is in de praktijk gebleken tijdens dit onderzoek. Figuur 113 illustreert dit goed.

```
In[28]:= u[1] = 2;
         f[u[x_]] := "a"
         f[x_] := "b"
         {f[u[2]], f[u[1]], f[2]}
Out[28]= {a, b, b}
```

Figuur 113: Gebruik types in Wolfram.

De eerste regel definieert *u[1]* als zijnde twee. Daarna is gesteld dat *f* van een *u*-expressie *'a'* moet teruggeven, en van eender welke andere expressie *'b'*. Toch is *f[u[1]]* gelijk aan b. Dit komt doordat Wolfram *u[1]* rechtstreeks interpreteert als 2, en dus *f[u[1]]* als *f[2]* ziet. Op zich lijkt dit geen probleem, aangezien bovenaan gedefinieerd is dat 2 ook een *u*-expressie is. Blijkbaar interpreteert Wolfram deze definitie echter slechts in één richting, aangezien Wolfram 2 duidelijk niet ziet als een *u*-expressie. Aldus geeft *f[2]* ook *b* terug. Dit is uiteraard een belangrijke beperking waar rekening mee gehouden dient te worden. Zowel voor onervaren als ervaren gebruikers kan deze onvoorspelbare control-flow vervelende gevolgen hebben en moeilijk te ontdekken fouten veroorzaken.
In Wolfram zijn gelukkig wel een aantal functies ingebouwd die nauwgezette controle geven over de evaluatie van expressies. Deze zijn allemaal gedocumenteerd in [71]. Deze functies kunnen er voor deze toepassing voor zorgen dat Wolfram *f[u[1]]* pas evalueert binnen de functie *f* zelf. Hiervoor is de functie *Unevaluated* gebruikt, zoals te zien in Figuur 114.

```
In[38]:= u[1] = 2;
         f[u[x_]] := "a"
         f[x_] := "b"
         {f[u[2]], f[Unevaluated[u[1]]], f[2]}
Out[41]= {a, a, b}
```

Figuur 114: Gebruik van evaluatie-controle in Wolfram.

De functie *Unevaluated* dwingt de Wolfram-interpreter om de expressie *u[1]* ongeëvalueerd door te geven aan de functie *f*. Hierdoor levert f[u[1]] nu wel het gewenste resultaat.
De flexibele interpretatie van code in Wolfram is in dit geval dus eerder een nadeel dan een voordeel, aangezien er behoefte is voor het gebruik van speciale functies voor het controleren van de interpretatie. Voor onervaren programmeurs kan deze eigenschap soms tot schijnbaar onbegrijpbaar gedrag van de code leiden. Het is dan ook makkelijk om hier fouten tegen te maken.
Het volgende voorbeeld illustreert bovenvermeld probleem nog beter. Beschouw namelijk Figuur 115.

```
In[272]:= q[of[x_], of[y_]] := "Een parameter"
          q[o[x_, x2_], o[y_, y2_]] := "Twee parameters"

          of[x_] := o[x, 1]
          of[x_, y_] := o[x, y]

          q[of[1], of[2]]

Out[276]= Een parameter
```

Figuur 115: Voorbeeld onvoorspelbare control-flow Wolfram.

Ook in bovenstaand voorbeeld is niet meteen duidelijk op welke versie van *q* de expressie *q[of[1],of[2]]* zal matchen op basis van alleen de code. Blijkbaar kiest Wolfram in dit geval de eerste versie, en worden de expressies *of[1]* en *of[2]* niet geëvalueerd tot *o[1,1]* en *o[2,1]* in dit geval. Dit is in tegenspraak met het gedrag van Wolfram uit Figuur 113, waar Wolfram de expressies wél eerst evalueerde alvorens ze als argument voor de grotere functie te gebruiken.
Wat nog merkwaardiger is, is dat aanpassingen die schijnbaar niks te maken hebben met de werking van de code toch een invloed kunnen hebben op de evaluatie van expressies. Figuur 116 geeft dit weer.

```
In[1]:= q[of[x_], of[y_]] := "Een parameter"
        q[o[x_, x2_], o[y_, y2_]] := "Twee parameters"

        of[x_] := o[x, 1]

        q[of[1], of[2]]

Out[4]= Twee parameters
```

Figuur 116: Aangepaste code uit Figuur 115.

Door het verwijderen van het ongebruikte patroon *of[x_,y_]* uit de code van Figuur 115 is het resultaat van de aanroep van de functie *q* plots anders geworden. Dit is uiteraard een zeer bizar resultaat dat tot heel wat verwarring kan leiden. Dit geeft aan dat de flexibele interpretatie van Wolfram wel degelijk volledig willekeurig is, en dus in sommige gevallen evenzeer een nadeel van de taal kan zijn als een voordeel.
Een laatste voorbeeld dat expliciet de nadelen van de flexibele interpretatie van Wolfram illustreert, los van de nadelen van het gebrek aan een sterk typesysteem, is geïllustreerd in Figuur 117.

```
In[49]:= ListQ[1]
         ListQ[{1}]
         AllTrue[{{1, 2}, {1, 2}, 2}, ListQ]
         AllTrue[{{1, 2}, {3, 4}, {5, 6}}, ListQ]
         AllTrue[1, ListQ]

Out[49]= False

Out[50]= True

Out[51]= False

Out[52]= True

Out[53]= True
```

Figuur 117: Illustratie nadelen flexibele interpretatie Wolfram.

De functie *ListQ* evalueert of de input ervan een lijst is of niet. In het eerste geval uit bovenstaande code is te verwachten dat de uitkomst van deze functie *False* is, aangezien *1* geen lijst is. Doordat *{1}* wel een lijst is, is de uitkomst van de tweede expressie *True*.
De functie *AllTrue* gaat na of alle elementen van een lijst voldoen aan de voorwaarde die wordt meegegeven als tweede argument, in dit geval de functie *ListQ*. De eerste twee aanroepen van deze functie gedragen zich zoals verwacht. De laatste aanroep van *AllTrue* heeft echter zeer bizar *True* als uitkomst. De verklaring hiervoor is vermoedelijk dat Wolfram de argumenten van *AllTrue* steeds als lijst interpreteert, en in dit geval de boolse expressie *ListQ* dan toepast op deze lijst. Wolfram zoekt dus steeds een manier om de code uit te voeren zoals ze waarschijnlijk bedoeld was door de programmeur. Dit heeft vaak uiteraard grote voordelen, aangezien Wolfram-code zelden crasht, en dit systeem heel wat verstrooidheidsfouten automatisch kan corrigeren. In de praktijk zijn hier echter ook een aantal nadelen aan verbonden waarvoor de programmeur steeds alert moet zijn voor de manier waarop Wolfram zijn code interpreteert.

5.3.2 Inconsistente syntax

Zoals uit dit hoofdstuk gebleken is zijn in Wolfram tal van manieren voorzien om functies te personaliseren, en optioneel extra parameters mee te geven. Dit maakt de code enerzijds compact en elegant, maar heeft als nadeel dat de code niet consistent is. Zo is uit een functie niet rechtstreeks af te leiden wat de opties voor deze functie zijn, en is het gebruik van de documentatie vaak noodzakelijk om een goed beeld te krijgen van alle mogelijkheden van elke functie. Gelukkig is deze documentatie wel zeer gedetailleerd en uitgebreid.
Ook is tijdens dit onderzoek opgevallen dat veel functies enkel toepasbaar zijn op specifieke expressies, zoals bijvoorbeeld de eerder beschreven *EmbedCode*-functie. Zonder de documentatie is het door de afwezigheid van een sterk typesysteem onmogelijk te achterhalen wat een functie juist als input verwacht, en welke output gegenereerd wordt op basis van een bepaalde input. Het is immers goed mogelijk dat een functie gewoon uitgevoerd wordt op een verkeerde input zonder te crashen, maar niet het gewenste resultaat levert. Dit kan zeer frustrerend zijn en leiden tot moeilijk te detecteren fouten.
Verder is het vaak zo dat functies voor bepaalde opties een specifiek geformatteerde lijst verwachten. Enkel op basis van de documentatie is het mogelijk te achterhalen wat het formaat van deze parameters juist moet zijn. Bovendien is eerder al aangehaald dat door de snelle groei en jonge leeftijd van Wolfram de documentatie soms niet overeenkomt met de effectieve waarden die geldig zijn in Wolfram voor bepaalde opties en parameters van bepaalde functies.
Ten slotte valt het soms voor dat voor een bepaalde functie een bepaalde parameter op een bepaalde manier dient geformatteerd te zijn, terwijl voor een andere functie een gelijkaardige parameter ineens compleet anders dient geformatteerd te zijn. Op dit gebied is er nog wat werk aan de Wolfram-taal op zich. Dit probleem is echter niet al te ernstig, en is meestal eenvoudig te overwinnen door aandachtige studie van de documentatie en enig experimenteerwerk. Desalniettemin is een gebrek aan consistentie steeds vervelend.

5.3.3 Beperkingen cloud-omgeving

Naast de beperkingen van de taal zelf heeft Wolfram nog een aantal problemen met de ontwikkelomgeving. Tijdens dit onderzoek is immers vastgesteld dat het in paragraaf 2.2.5.5 aangehaalde veelbelovende Wolfram Development Platform in de cloud soms zeer traag kan zijn. Op bepaalde willekeurige momenten is het zelfs nagenoeg onbruikbaar. Ook de Wolfram Cloud zelf waar code naar gedeployed kan worden is soms zodanig traag dat complexere code letterlijk meerdere minuten vraagt om te laden. Dit is uiteraard onacceptabel. Op andere momenten is de performantie van dit platform dan wel weer goed. Het uitvoeren van dezelfde code op de lokale pc daarentegen duurt maximaal enkele seconden, dus dit probleem ligt zeker niet bij de code zelf of de taal. Het is echter behoorlijk moeilijk om een Wolfram-IDE voor de lokale pc te bekomen. De enige mogelijkheid hiervoor is een dure licentie te nemen voor het Wolfram Development Platform, of een iets goedkopere licentie voor Wolfram Mathematica. Deze laatste is heel gelijkaardig aan de desktop-IDE die bij de dure licenties voor het Wolfram Development Platform zit, maar biedt minder cloud-functionaliteit aan.
Een alternatief voor een betalende licentie is het gebruik van een Raspberry Pi voor ontwikkeling van Wolfram-code. Op de Raspberry Pi is immers een gratis kopie van Wolfram Mathematica beschikbaar [72]. Dit is echter omwille van de sterke hardware-beperkingen van de Raspberry Pi ook geen ideale optie voor rekenintensieve programma's. Het is echter te verwachten dat de kwaliteit van de infrastructuur van het Wolfram Development Platform in de cloud enkel maar zal toenemen in de toekomst als Wolfram aan populariteit wint. Voorlopig is de onvoorspelbare performantie van dit cloud-platform echter wel een noemenswaardige beperking.

5.4 Besluit

Wolfram is een zeer krachtige, stijlvolle, compacte, intuïtieve en flexibele programmeertaal die de beste aspecten van verschillende programmeerparadigma's combineert, met de nadruk op het functionele paradigma. Wolfram is zeer aangenaam om te gebruiken voor allerhande toepassingen, gaande van dataverwerking en -analyse tot het snel en eenvoudig maken van user interfaces voor interactie met de verwerkte data. Ook zijn een groot aantal unieke eigenschappen geïntegreerd in Wolfram, die in geen enkele andere programmeertaal bestaan. De grote flexibiliteit en snelle groei van Wolfram hebben echter ook een aantal nadelen. Deze zullen in de toekomst echter hopelijk afnemen. Als dit zo is, is Wolfram een zeer aangename programmeertaal voor allerlei toepassingen en zeker het uitproberen waard dankzij het ongeëvenaard programmeergemak en de unieke eigenschappen die deze taal biedt.

6 Vergelijkingsmethode

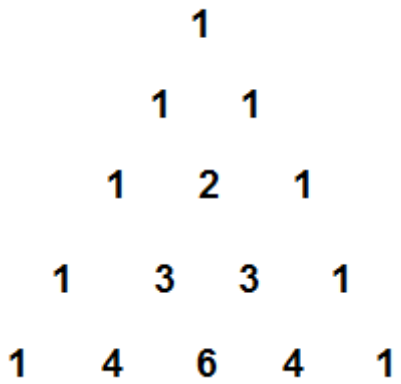
Het hoofddoel van deze masterproef is zoals beschreven in paragraaf 1.3 het vergelijken van verschillende functionele talen met hun klassieke tegenhangers in een cloud computing-context. Vergelijkingspunten zijn performantie, schaalbaarheid, connectiesnelheid, beschikbare cloud-platformen, etc. Paragraaf 2.3.1 stelde reeds dat het zwart-wit kwantitatief vergelijken van verschillende programmeertalen niet echt een meerwaarde is. Het is dus meer gepast een kwalitatief oordeel te vellen over de verschillende talen, en in het verlengde daarvan concrete toepassingsgebieden af te bakenen waarin elk van de bestudeerde talen het best tot zijn recht komt.

Alvorens het vergelijken van de verschillende talen van start kan gaan, is het belangrijk een concrete eenvoudige testapplicatie te definiëren om te implementeren in de verschillende voor dit onderzoek gekozen programmeertalen. Het is belangrijk dat de implementatie zo analoog mogelijk kan gebeuren over de verschillende talen heen. Aan de hand van deze applicatie kunnen de vooropgestelde evaluatiecriteria nagegaan worden voor elke taal. Dit hoofdstuk licht de gekozen testapplicatie toe, en illustreert de manier waarop alle performantie-gerelateerde testen in deze thesis uitgevoerd zijn. Naast de performantie-gerelateerde aspecten vergelijkt deze masterproef ook implementatie-gerichte aspecten zoals leesbaarheid, lengte van de code etc. Hieraan is hoofdstuk 7 gewijd. Nog later komen cloud-specifieke aspecten als schaalbaarheid en connectiesnelheid aan bod.

6.1 Testapplicatie

De testapplicatie voor dit onderzoek bestaat uit een kort programma dat Driehoeken van Pascal genereert. Deze paragraaf gaat dieper in op de achtergrond en eigenschappen van de Driehoek van Pascal.

De Driehoek van Pascal op zich is in feite een reeks natuurlijke getallen die gevisualiseerd zijn in de vorm van een driehoek. Helemaal aan de top staat het getal 1. Langs de twee zijden die in verbinding staan met de top staat ook overal 1. Alle andere elementen bestaan uit de som van de twee elementen schuin erboven. Hierdoor ontstaat een structuur zoals weergegeven in Figuur 118. Meer info over de Driehoek van Pascal is te vinden op [73].



Figuur 118: Driehoek van Pascal [74].

6.1.1 Implementatie

De implementatie van de Driehoek van Pascal in de verschillende talen bestaat uit een algoritme dat de driehoek opbouwt als een 2D-array van natuurlijke getallen. Een Driehoek van Pascal van vijf rijen ziet er binnen het testprogramma bijgevolg uit als bv. in Figuur 119.

[[1], [1,1], [1,2,1], [1,3,3,1], [1,4,6,4,1]]

Figuur 119: Weergave Driehoek van Pascal van vijf rijen in dit onderzoek.

Ook is er code voorzien voor het opvragen van de driehoek in de vorm van de hierboven weergegeven 2D-Array.
De Driehoek van Pascal kan op verschillende manieren geïmplementeerd worden. Deze thesis behandelt enerzijds een recursieve en anderzijds een niet-recursieve methode.
In het eerste geval berekent het algoritme elk element van de driehoek apart. Om dit te doen moeten voor elk element recursief alle bovenliggende elementen uitgerekend worden. De driehoek wordt dus element per element opgebouwd.
Bij de niet-recursieve methode daarentegen wordt de Driehoek van Pascal in plaats van element-per-element rij-per-rij opgebouwd. Er is een functie of een for-lus die een rij toevoegt aan een al bestaande driehoek. Deze bestaande driehoek kan een data member zijn in OO-talen, of een parameter in functionele talen. Op deze manier kan er bij het toevoegen van nieuwe elementen rechtstreeks beroep gedaan worden op de reeds berekende elementen. Dit spaart vanzelfsprekend enorm veel rekenwerk. Merk wel op dat de terminologie niet-recursief in dit geval enkel betrekking heeft op het berekenen van de individuele elementen. In functionele talen zal de functie die de driehoek opbouwt wel nog steeds zichzelf recursief oproepen. In essentie is dit gewoon hetzelfde als een for-lus in OO-talen. Het volgende hoofdstuk licht deze implementaties voor elke geteste taal in detail toe.

6.1.2 Tijdscomplexiteit

Om een goede benadering te kunnen geven van de performantie van de verschillende talen, is het belangrijk om te weten wat de tijdscomplexiteit is van het algoritme dat bovenstaande Driehoeken van Pascal genereert. Deze tijdscomplexiteit kan immers gebruikt worden om het gedrag van het testprogramma te voorspellen wat betreft CPU-gebruik. De tijdscomplexiteit kan ook gebruikt worden om een trendlijn te maken doorheen de waargenomen data, en zo de datavisualisatie te verbeteren. Deze paragraaf behandelt de tijdscomplexiteit voor zowel de recursieve als niet-recursieve implementatie van de Driehoek van Pascal. Als referentie wordt de Java-implementatie zonder Streams genomen, die het volgende hoofdstuk in meer detail uiteenzet. Dezelfde redenering is van toepassing op al de andere implementaties.

6.1.2.1 Niet-recursief

De tijdscomplexiteit van de niet-recursieve versie van de Driehoek van Pascal is eenvoudig te bepalen. Figuur 120 geeft weer hoe de implementatie van de niet-recursieve Driehoek van Pascal er uit ziet in Java.

```
package triangles;

import java.math.BigInteger;

public class PascalTriangle {
    private final BigInteger[][] points;

    public PascalTriangle(int size){
        points = new BigInteger[size][];
        for (int i = 0;i<size;i++){
            points[i]=new BigInteger[i+1];
            for (int j = 0; j<=i;j++){
                try{
                    points[i][j]=points[i-1][j-1].add(points[i-1][j]);
                }
                catch(ArrayIndexOutOfBoundsException ex){
                    points[i][j]=BigInteger.ONE;
                }
            }
        }
    }
    public BigInteger[][] getPoints(){
        return points;
    }
}
```

Figuur 120: Implementatie van de niet-recursieve Driehoek van Pascal in Java.

Deze implementatie bestaat uit twee geneste for-lussen, waarin telkens een coëfficiënt van de driehoek bepaald wordt op basis van de eerder berekende coëfficiënten, waarna deze nieuw berekende coëfficiënt meteen in de array die de Driehoek van Pascal voorstelt wordt opgeslagen. De enige bewerkingen die in de lussen plaatsvinden zijn optellingen, lijst-inserties, en lijst-opvragingen. Al deze bewerkingen hebben een constante tijd nodig. De tijdscomplexiteit van de gehele driehoek is dus gelijk aan het aantal keer dat de binnenste lus in het algoritme wordt bereikt.
Als n met 1 verhoogt, zal de binnenste lus in totaal 1 keer meer uitgevoerd worden. Het aantal keer dat een extra punt wordt toegevoegd zal n zijn, aangezien j van 0 tot en met i loopt voor i = n. Dit geeft volgende recursievergelijking:

$$T(n) = T(n-1) + 1 \cdot n$$

Bij een recursievergelijking van bovenstaande vorm waarbij de tijd nodig voor het uitvoeren van het algoritme met n als input gelijk is aan de tijd nodig voor het uitrekenen van het algoritme met n – 1 als input plus een polynomiale functie van n, geldt dat de tijdscomplexiteit in de limiet voor n gaande naar oneindig gelijk is aan de hoogste macht van n in de polynomiale functie plus één [75]. Hieruit volgt dat de tijdscomplexiteit van het niet-recursief Driehoek van Pascal-algoritme gelijk is aan:

$$O(n^2)$$

6.1.2.2 Recursief

De tijdscomplexiteit van het recursieve Driehoek van Pascal-algoritme is veel ingewikkelder om te bepalen. Om te beginnen geeft Figuur 121 de referentie-implementatie van dit algoritme weer in Java.

```
package triangles;

public class PascalTriangleRec {
    private final int[][] points;

    public PascalTriangleRec(int size){
        points = new int[size][];
        for (int i =0;i<size;i++){
            int[] row  = new int[i+1];
            for (int j = 0;j<=i;j++){
                row[j]=getValueAtPoint(i,j);
            }
            points[i]=row;
        }
    }
    public static int getValueAtPoint(int row, int col){
        if (col == 0 || col == row) return 1;
        else return getValueAtPoint(row-1,col-1)+getValueAtPoint(row-1,
col);
    }
    public int[][] getPoints(){
        return points;
    }
}
```

Figuur 121: Implementatie van de recursieve Driehoek van Pascal in Java.

Deze implementatie bestaat in essentie uit twee geneste for-lussen die de recursieve *getValueAtPoint* functie oproepen. De *getValueAtPoint* functie zelf roept op basis van de parameters *row* en *col* ofwel zichzelf tweemaal recursief op, ofwel geeft de functie 1 terug. Beide scenario's vragen een constante tijd. De tijdscomplexiteit van de hele Driehoek van Pascal is dus rechtstreeks afhankelijk van hoe vaak deze beide scenario's voorvallen.
Het aantal keer dat de *getValueAtPoint* functie 1 teruggeeft is vrij eenvoudig analytisch te bepalen. Er bestaat namelijk een eigenschap van de Driehoek van Pascal, namelijk Pascal's Corollary 8, die zegt dat voor elke gegeven rij r in de Driehoek van Pascal, de som van alle coëfficiënten op die rij gelijk is aan 2^r, met r beginnend bij 0 [76]. Deze stelling wordt bewezen op [77]. Bij de implementatie uit Figuur 121 is echter bij conventie aangenomen dat r begint bij 1. Dit betekent dat voor deze implementatie geldt dat de som van alle elementen op een rij r gelijk is aan 2^{r-1}. Aangezien de *getValueAtPoint* functie enkel 1 kan teruggeven ofwel zichzelf oproepen, betekent dit dat de som van alle coëfficiënten op een bepaalde rij gelijk is aan een som van éénen. Hieruit volgt dat het aantal keer dat de *getValueAtPoint* functie 1 teruggeeft voor een gegeven rij gelijk moet zijn aan 2^{r-1}.
Het aantal keer dat de *getValueAtPoint* functie zichzelf recursief oproept, is intuïtief evenzeer op een vorm van 2^r te schatten, aangezien voor elke rij de *getValueAtPoint* functie zichzelf 2 keer oproept, met voor r' = r-1. Verder geldt bij benadering dat het aantal keer dat de functie zichzelf tweevoudig recursief oproept ten opzichte van de vorige rij naast het hierboven

vastgestelde exponentiële verband ook nog evenredig is met de verhouding van het aantal coëfficiënten in rij r ten opzichte van het aantal coëfficiënten in rij r-1. Voor grote waarden van r is deze verhouding bij benadering gelijk aan 1. Voor grote r convergeert het aantal keer dat het scenario voorvalt waarbij de functie zichzelf tweemaal recursief aanroept dus naar een vorm van 2^r. Door een paar kleine aanpassingen in het recursief algoritme is eenvoudig te bepalen hoe vaak dit scenario voorvalt. Tabel 2 geeft de gevonden resultaten weer.

Tabel 2: Het aantal keer dat de getValueAtPoint zichzelf tweemaal recursief aanroept in functie van r.

Waarde voor r	**Aantal recursieve aanroepen getValueAtPoint**
1	0
2	0
3	1
4	4
5	11
6	26
7	57
8	120
9	247
10	502
11	1023

Het is duidelijk dat voor grote waarden van r het aantal recursieve oproepen convergeert naar 2^{r-1}. Voor kleine waarden van r geldt dit verband niet, aangezien de driehoek in dat geval voor een groot stuk uit 1'en bestaat. Hoe groter de driehoek echter, hoe minder impact deze éénen hebben op het resultaat. Verder valt op dat de werkelijke waardes voor het aantal recursieve oproepen niet exact overeenkomen met 2^{r-1}. De exacte waardes zijn in dit geval echter niet van belang, en deze analytisch bepalen valt buiten het bestek van deze masterproef.
Bovenstaande bevindingen resulteren in de volgende formulering voor de totale tijd nodig voor het berekenen van een enkele rij in de Driehoek van Pascal:

$$t_r = t_1 + t_{rec}$$

$$t_r = C \cdot (2^{r-1} + 2^{r-1})$$

$$t_r = C \cdot (2 \cdot 2^{r-1})$$

$$t_r = C \cdot 2^r$$

Nu de tijd voor het berekenen van een enkele rij gekend is, rest enkel de som te nemen voor alle rijen van de driehoek. Dit is de som voor r = 1 tot r = n. Dit geeft volgend resultaat:

$$t_n = \sum_{r=1}^{n} t_r$$

$$t_n = \sum_{r=1}^{n} C \cdot 2^r$$

$$t_n = C \cdot \left(\sum_{r=1}^{n} 2^r \right)$$

Deze uitdrukking is met behulp van de somformule voor meetkundige reeksen te vereenvoudigen:

$$t_n = C \cdot 2 \cdot \frac{1 - 2^n}{1 - 2}$$

$$t_n = C' \cdot (2^n - 1)$$

Met C′ = 2 · C. Voor grote waarden van n is de constante term C′ te verwaarlozen ten opzichte van de term $C' \cdot 2^n$. Daarom geldt voor grote waardes van n:

$$t_n \approx C' \cdot 2^n$$

Uit bovenstaande afleiding volgt dat de tijdscomplexiteit van het recursief Driehoek van Pascal-algoritme gelijk is aan $O(2^n)$.

6.1.2.3 Besluit

De niet-recursieve benadering is duidelijk veel efficiënter dan de recursieve, zoals intuïtief reeds aangewezen. Desalniettemin is het doel van deze testapplicatie onder andere de performantie van de verschillende programmeertalen te vergelijken. Daarom is het toch zoals reeds aangehaald aangewezen om de recursieve versie als primair uitgangspunt te nemen voor de verschillende tests.
In werkelijkheid vereisen veel applicaties echter veel minder rekenkracht. Om een volledig beeld te krijgen van de performantie van de verschillende talen over een breed applicatiedomein en een correct kwalitatief oordeel te kunnen vellen is het nuttig ook de niet-recursieve implementatie van de Driehoek van Pascal onder de loep te nemen. Door de veel betere tijdscomplexiteit kunnen in een aanvaardbare tijd veel meer coëfficiënten gegenereerd worden met dit algoritme dan met de recursieve variant. Dit is interessant voor deze masterproef, omdat hierdoor het niet-recursieve algoritme veel meer geheugenintensief is, terwijl de recursieve implementatie veel meer processorintensief is. Zo dekken deze tests een breed applicatiedomein.

Ten slotte zijn constante factoren in beide tijdscomplexiteiten die berekend zijn verwaarloosd. Het is echter wel aangewezen om deze in bepaalde omstandigheden toch op te nemen in de curvefitting van de resultaten, met name voor de cloud-gedeployde versies van de Driehoek van Pascal. Naast de zuivere performantie is de connectiesnelheid immers ook van belang. Dit is meestal een constante factor die behoorlijk groot zou kunnen zijn en van doorslaggevend belang kan zijn bij de keuze voor een bepaalde programmeertaal of een bepaald cloud-platform.

6.2 Testomgeving

Naast het definiëren en implementeren van een goede testapplicatie is het ook belangrijk om over een goede testomgeving te beschikken om de resultaten van de verschillende tests te verzamelen en te vergelijken. De grootste uitdaging in het zoeken naar een gepaste testomgeving is het vinden van een manier om zo veel mogelijk variabelen zoals reactietijd, beschikbare processorcapaciteit en geheugen, … bij de uitvoering van de code constant te houden. Verder moet er ook een manier zijn om de gegenereerde testresultaten op een overzichtelijke manier te visualiseren en te vergelijken.
Wolfram biedt voor de hierboven beschreven problemen een ideale oplossing, aangezien het standaard functionaliteit bezit om Java en .Net applicaties te integreren. Op deze manier kan op volledig uniforme wijze elke versie van de Driehoek van Pascal die in dit onderzoek aan bod komt worden uitgevoerd, en tevens kan steeds op exact dezelfde manier de gebruikte tijd achterhaald worden. Hierdoor is de relatieve performantie van elke taal eenduidig bepaald.
Bovendien maakt Wolfram het zeer eenvoudig testdata te genereren, analyseren, en visualiseren dankzij een heel aantal krachtige en flexibele ingebouwde functies. Dit maakt van Wolfram het ideale platform om alle testresultaten samen te brengen en te visualiseren.

6.2.1 Integratie verschillende talen

Het allereerste wat nodig is, is de code die toelaat de verschillende Driehoek van Pascal projecten in te laden in Wolfram. Integratie van Java en .NET in Wolfram is reeds in detail besproken in paragraaf 5.2.9. Deze paragraaf geeft enkel een beknopt overzicht van de code die gebruikt werd voor het inladen van de verschillende versies van de Driehoek van Pascal in Wolfram. Merk op dat voor elke implementatie een apart project is aangemaakt, en opgeslagen op de test-pc. De inhoud van elk van deze projecten is in detail beschreven in hoofdstuk 7.

6.2.1.1 Java

Voor de Java-versie van de recursieve Driehoek van Pascal dient het project gebuild te worden, en met behulp van J/Link geïmporteerd te worden in Wolfram. De code hiervoor is zoals weergegeven in Figuur 122.

```
Needs["JLink`"]
ReinstallJava[ClassPath → "E:\\Dropbox\\School\\MasterProef\\Triangles\\dist\\Triangles.jar"];
```

Figuur 122: Integratiecode voor de Java-testapplicatie in Wolfram.

Bij wijze van test werd er een Driehoek van Pascal van 5 rijen aangemaakt, en de punten hiervan opgevraagd. Figuur 123 geeft het resultaat hiervan weer.

```
In[49]:= Needs["JLink`"]
         ReinstallJava[ClassPath → "E:\\Dropbox\\School\\MasterProef\\Triangles\\dist\\Triangles.jar"];
         t = JavaNew["triangles.PascalTriangleRec", 5];
         t@getPoints[]
Out[52]= {{1}, {1, 1}, {1, 2, 1}, {1, 3, 3, 1}, {1, 4, 6, 4, 1}}
```

Figuur 123: In Wolfram geïntegreerd recursief Driehoek van Pascal-algoritme, geschreven in Java.

Voor de niet-recursieve versie in Java moet er nog een wijziging gebeuren aan de integratiecode uit Figuur 122. Aangezien de tijdscomplexiteit van de niet-recursieve Driehoek van Pascal immers veel beter is dan die van de recursieve versie, moeten er veel meer rijen worden gegenereerd bij de niet-recursieve versie om een representatief resultaat te bekomen. Hiervoor is echter enorm veel werkgeheugen nodig. Het probleem hierbij is dat het maximale bruikbare geheugen voor Java geïntegreerd in Wolfram slechts ongeveer 1GB is. Dit getal is empirisch bepaald, en kan verschillen van systeem tot systeem.
Dit is op te lossen door een extra Option mee te geven bij het ReinstallJava commando in Wolfram, wat de heap die de JRE krijgt vergroot. Figuur 124 geeft de code die hiervoor nodig is weer.

```
ReinstallJava[ClassPath → "E:\\...\\dist\\Triangles.jar",JVMArguments → "-Xmx8192m"];
```

Figuur 124: Aangepaste code om meer geheugenruimte vrij te maken voor de JRE in Wolfram.

6.2.1.2 C#

Voor de integratie van C# is gebruik gemaakt van de .NET/Link bibliotheek. Het enige noemenswaardige verschil met J/Link is dat .NET/Link verplicht de gebruikte Assemblies eerst ook nog expliciet in te laden. Volgende Wolfram code uit Figuur 125 geeft dit weer.

```
Needs["NETLink`"]
ReinstallNET[];
LoadNETAssembly["E:\\...\\ DotNetTriangles\\bin\\Release\\DotNetTriangles.exe"];
```

Figuur 125: Integratiecode voor de C#-testapplicatie in Wolfram.

Voor de C# versie is een gelijkaardige test uitgevoerd aan die van Java. Figuur 126 geeft hiervan het resultaat weer.

```
In[961]:= Needs["NETLink`"]
          ReinstallNET[];
          LoadNETAssembly["E:\\Dropbox\\School\\MasterProef\\DotNet\\DotNetTriangles\\DotNetTriangles\\obj\\Release\\DotNetTriangles.exe"];
          t = NETNew["DotNetTriangles.PascalTriangle", 5];
          t@Points
Out[966]= {{1}, {1, 1}, {1, 2, 1}, {1, 3, 3, 1}, {1, 4, 6, 4, 1}}
```

Figuur 126: In Wolfram geïntegreerd recursief Driehoek van Pascal-algoritme, geschreven in C#.

6.2.1.3 F#

Er bestaat online geen documentatie over hoe F# te integreren valt in Wolfram. Aangezien F# echter ook gebruik maakt van de .NET Runtime, is het in principe mogelijk evenzeer F# als C# te gebruiken in Wolfram. Om dit te doen is het allereerst nodig om bij de code voor het integreren van de C# versie van de recursieve Driehoek van Pascal ook de Assembly van het F# project in te laden. Figuur 127 geeft hiervoor de code weer.

```
LoadNETAssembly["E:\\Dropbox\\School\\...\\ FSTriangles \\bin\\Release\\FSTriangles.exe"];
```

Figuur 127: Integratiecode voor de F#-testapplicatie in Wolfram

De grootste uitdaging voor het gebruiken van F# in Wolfram is het feit dat F# primair een functionele taal is, terwijl .NET/Link er juist op steunt om objecten te kunnen aanmaken van de ingeladen .NET types. Het is dus belangrijk om een F# type te hebben dat zich gedraagt als een C# object, en aldus vanuit Wolfram kan worden aangemaakt met .NET/Link. Dit type moet ook een data member hebben om de opgeslagen waardes van de Driehoek van Pascal bij te houden. Gelukkig heeft F# ook een aantal object-georiënteerde eigenschappen, naast de functionele. Daarom dient in de F#-code steeds een typedefinitie aanwezig te zijn die overeenkomt met de 'klasse' van het 'object' dat .NET/Link moet opvragen. Voor dit onderzoek is een type PTriangle toegevoegd in de code van de F#-versie van de recursieve Driehoek van Pascal. Figuur 128 geeft de code hiervoor weer.

```
type PTriangle(size:int) =
     member this.Points = PascalTriangle.pascalTriangle size
```

Figuur 128: Typedeclaratie in F# die nodig is voor de integratie in Wolfram.

Nu ziet .NET/Link geen onderscheid meer tussen het F# type PTriangle en een gewone C# klasse. Nu kan de F# code op dezelfde manier worden gebruikt in Wolfram als de C# versie. Ook deze code werd getest op dezelfde manier als de Java en C# implementaties. Figuur 129 geeft hiervan het resultaat weer.

```
In[19]:= Needs["NETLink`"]
         ReinstallNET[];
         LoadNETAssembly["E:\\Dropbox\\School\\MasterProef\\DotNet\\FSTriangles\\FSTriangles\\obj\\Release\\FSTriangles.exe"];
         t = NETNew["Triangles.PTriangle", 30];
         t@Points
Out[23]= « NETObject[Microsoft.FSharp.Collections.FSharpList`1[Microsoft.FSharp.Collections.FSharpList`1[System.Int32]]] »
```

Figuur 129: In Wolfram geïntegreerd recursief Driehoek van Pascal-algoritme, geschreven in F#.

Het valt op dat Wolfram niet standaard de F# lijsten omzet naar Wolfram lijsten, in tegenstelling tot de arrays uit C# en Java. Dit kan echter manueel gebeuren indien dat gewenst zou zijn. In het kader van deze testing is dit echter overbodig. Bovenstaande volstaat immers om vast te stellen dat de code werkt.

6.2.1.4 Wolfram

Voor de Wolfram code ten slotte moeten geen speciale voorbereidende stappen genomen worden voor de recursieve implementatie. Het volstaat om de Wolfram code voor de recursieve Driehoek van Pascal te plakken in het Notebook dat de testsoftware bevat.
De niet-recursieve implementatie vraagt echter nog enige speciale aandacht. In Wolfram is namelijk een veiligheid tegen oneindige lussen ingebouwd, in de vorm van een bovengrens op het aantal iteraties dat éénzelfde functie kan doorlopen. Het aantal iteraties dat nodig is om een zeer grote Driehoek van Pascal aan te maken is echter groter dan het standaard maximum van 4096. Dit is op te lossen door gebruik te maken van de functie Block. Deze functie past scoping toe waardoor binnen de Block-expressie het maximumaantal iteraties drastisch hoger kan zijn. Figuur 130 geeft de code hiervoor weer.

```
Block[{$IterationLimit = 10 000}, pascalTriangle[4400];]
```

Figuur 130: Code om het maximaal aantal iteraties in Wolfram te verhogen.

Voor de testresultaten in deze scriptie zijn geen zodanig grote driehoeken gegenereerd dat de Block functie noodzakelijk is, maar tijdens het uitvoeren van bepaalde experimenten die uiteindelijk niet in de thesis zijn opgenomen was dit wel het geval. Omwille van de informatieve waarde is bovenstaande code toch opgenomen in deze thesis.

6.2.2 Aanmaken testdata

Nu de verschillende versies van de Driehoek van Pascal geïmporteerd zijn in de testomgeving kan de eigenlijke testdata aangemaakt worden. Als performantietest wordt een *Iterator* gemaakt met de Wolfram-functie *Table* die voor waarden van i=1 tot i= iets tussen 20 en 30- afhankelijk van de gebruikte programmeertaal- een Driehoek van Pascal genereert van i rijen, en dan ook de coëfficiënten opvraagt van deze driehoek, en ze importeren in Wolfram. De *Iterator* meet de tijd die nodig is voor het genereren van elk van de driehoeken met behulp van de functie *Now*, en slaat deze op in een lijst. De maximale waarde van i moet zo uiteenlopend zijn omdat sommige talen veel sneller zijn dan andere. Deze maximale waarde is voor elke taal experimenteel bepaald.
Dit laatste in combinatie met het feit dat voor elke taal de implementatie net iets anders is, zorgen ervoor dat voor elke taal apart een expressie moet gedefinieerd worden die de testdata voor die taal genereert op een voor die taal gepaste manier. De code voor het genereren van de testdata is aldus weergegeven in Figuur 131.

```
dataWolfram := Table[{i, t1 = Now;
        pascalTriangleRec[i];
        Now - t1}, {i, 1, 20}];
dataWolframNonRecursive := Block[{$IterationLimit = 10 000}, Table[{i, t1 = Now;
        pascalTriangle[i];
        Now - t1}, {i, 10, 500, 10}]];
dataJava := Table[{i, t1 = Now;
        JavaNew["triangles.PascalTriangleRec", i]@getPoints[];
        Now - t1}, {i, 1, 30}];
dataJavaStreams := Table[{i, t1 = Now;
        JavaNew["triangles.FunctionalPascalTriangle", i]@getPoints[];
        Now - t1}, {i, 1, 30}];
dataJavaNonRecursive := Table[{i, t1 = Now;
        JavaNew["triangles.PascalTriangle", i]@getPoints[];
        Now - t1}, {i, 10, 500, 10}];
dataCS := Table[{i, t1 = Now;
        NETNew["DotNetTriangles.PascalTriangle", i]@Points;
        Now - t1}, {i, 1, 30}];
dataCSLINQ := Table[{i, t1 = Now;
        NETNew["DotNetTriangles.FunctionalPascalTriangle", i]@Points;
        Now - t1}, {i, 1, 30}];
dataFS := Table[{i, t1 = Now;
        NETNew["Triangles.PTriangle", i]@Points;
        Now - t1}, {i, 1, 30}];
```

Figuur 131: Code voor het genereren van de testdata in Wolfram.

Bovenstaande testdata moet nu ook aan een String waarde gekoppeld kunnen worden, zodat bij het uitvoeren van de tests met eenvoudige teksten kan worden gewerkt die een eenduidige beschrijving geven van wat er juist getest wordt. Dit doet denken aan een enum in Java. Hiervoor dient de data-functie, die Figuur 132 weergeeft.

```
data[name_String] := Which[
name == "Wolfram", Hold[dataWolfram],
name == "Wolfram non-recursive", Hold[dataWolframNonRecursive],
name == "Java", Hold[dataJava],
name == "Java with Streams", Hold[dataJavaStreams],
name == "Java non-recursive", Hold[dataJavaNonRecursive],
name == "C#", Hold[dataCS],
name == "C# with LINQ", Hold[dataCSLINQ],
name == "F#", Hold[dataFS],
True, Throw["Invalid input: " <> name, InvalidInputException]]
```

Figuur 132: Data-functie voor het koppelen van string-waardes aan de juiste testdata.

De functie *Which* is in feite een Switch statement, en *Hold* zorgt ervoor dat de data nog niet onmiddellijk geëvalueerd wordt (zie later). Merk ook het gebruik van de exception handling die Wolfram voorziet, als veiligheid tegen typfouten en andere potentiële oorzaken van een verkeerde input.

6.2.3 Visualisatie resultaten

De laatste stap in het testproces is het visualiseren van de testresultaten. Ten eerste is hier een functie voor nodig die bepaalt welke curvefitting-functie van toepassing is om het tijdsverloop van de evaluatie voor verschillende i te schetsen. Verschillende versies van de Driehoek van Pascal hebben immers een verschillende tijdscomplexiteit, zoals in paragraaf 6.1.2 reeds beschreven. Figuur 133 geeft de code voor deze functie weer.

```
fittingFunction[name_String] := Which[
Length[Select[name == # &][{"Wolfram non-recursive", "Java non-recursive"}]] == 1, {n^2},
Replace[Catch[data[name]], Hold[_] → True], {2^n},
True, Throw["Invalid input: " <> name, InvalidInputException]]
```

Figuur 133: FittingFunction voorhet bepalen van de gepaste curve-fitting functie voor de testdata.

De *Replace*-functie kijkt of de *Catch* een *Hold* van iets heeft teruggegeven. Als dit zo is bestaat de input. Anders is de input fout (catch geeft een String terug zoals te zien is in Figuur 132), en geeft de functie een *InvalidInputException* terug.
Ten slotte rest de code om de resultaten te visualiseren aan de hand van grafieken. Deze is beschreven in Figuur 134.

```
testPerformance[langs_List, iterations_Integer] :=testPerformance[langs, iterations, 0.5]
testPerformance[langs_List, iterations_Integer, yRange_] := Catch[
        testData = Table[ReleaseHold[#], {i, iterations}] & /@ ((data[#]) &) /@ langs;
        avg = ({First[First[#]], Mean([#[[2]] &) /@ #]} &) /@ Map[Flatten,
                Table[(Cases[#, {i, _}] &) /@ #, {i, Length[#[[1]]]}], {2}]& /@ testData;

        listPlot = ListPlot[avg,
                PlotStyle → PointSize[Large],
                PlotRange → {0, yRange},
                PlotLegends → ("Raw data " <> # &) /@ langs,
                TargetUnits → "Seconds"];

        tupels = Table[{avg[[i]], fittingFunction[langs[[i]]]}, {i, Length[langs]}];
        fit = Fit[(#[[1]])[[2;;]], #[[2]], n ] & /@ tupels;
        plot = Plot[Evaluate[fit], {n, 0, Max[Length[#[[1]]] & /@ testData]},
                PlotLabel → "Performance Recursive Pascal Triangle",
                PlotRangeClipping -> False,
                PlotRange → {0, yRange},
                AxesLabel → {"Rows", "Time (s)"},
                PlotLegends → ("Approx. " <> # &)/@ langs,
                PlotLabels → fit];

        Show[plot, listPlot,
                ImageSize -> Large],
        InvalidInputException];
```

Figuur 134: Code voor visualisatie testresultaten.

Elke test zal worden uitgevoerd door een aanroep van de *testPerformance*-functie. Eerst en vooral zijn er minstens twee parameters: enerzijds de lijst van de talen die getest dienen te worden, en anderzijds het aantal iteraties dat doorlopen dient te worden bij het uitvoeren van de test. Voor maximale nauwkeurigheid is het immers gewenst de test meerdere keren achter elkaar uit te voeren en dan het gemiddelde van alle resultaten te nemen. Voor de meeste tests in dit onderzoek is 10 iteraties gekozen. Dit getal is niet overdreven hoog, maar

geeft desalniettemin een accuraat resultaat. Verschillende malen hetzelfde experiment met 10 iteraties opnieuw uitvoeren toonde aan dat er nooit meer dan 10% afwijking is op de resultaten met dit aantal iteraties. In de context van een kwalitatief oordeel vellen is het dan ook overbodig om meer dan 10 iteraties te gebruiken. Dit bespaart ook veel tijd, aangezien sommige experimenten die later in deze thesis aan bod komen zelfs voor 10 iteraties al een half uur in beslag namen.
Naast 10 iteraties is het soms ook nuttig het resultaat van één enkele iteratie te bestuderen. Op deze manier is latency te ontdekken die bij de eerste calls ontstaat, zoals bijvoorbeeld wanneer er extra zaken moeten worden ingeladen/opgestart. De *iterations* parameter is uiteraard een integer. De *langs* parameter dient een lijst van Strings te zijn. Deze strings moeten allemaal overeenkomen met één van de Strings die in de *data* functie gedefinieerd zijn. Anders geeft de functie een *InvalidInputException* terug. Ten slotte is er nog een derde optionele parameter, namelijk *yRange*. Indien deze is opgegeven, wordt het bereik voor het plotten van de testresultaten in een grafiek op de tijds as gelijkgesteld aan de waarde van *yRange*. De standaardwaarde voor *yRange* is 0,5.
Eerst en vooral evalueert de *testPerformance* functie de *langs* lijst element per element. Voor elk element zoekt de functie eerst de juiste expressie met behulp van de *data*-functie. De code evalueert deze expressie vervolgens, en slaat het resultaat in een nieuwe lijst op. Dit proces wordt *iterations* keer herhaald.
Vervolgens neemt de functie taal per taal het gemiddelde van alle testdata, en slaat de functie dit gemiddelde op in de variabele *avg*.
Daarna maakt de code een *ListPlot* van de gegenereerde gemiddelde testresultaten, opgeslagen in de variabele *listPlot*.
Dan genereert de code een lijst van tupels die voor elke taal de testresultaten en de juiste fitting functie, gevonden aan de hand van *fittingFunction*, aan elkaar koppelt.
Vervolgens wordt voor elk van deze tupels aan de hand van de *Fit* functie een passende lineaire combinatie gegenereerd van de door *fittingFunction* bepaalde set van functies. Dit slaat de functie dan weer op in de variabele *fit*. Hierin zit nu een lijst van functievoorschriften. Merk op dat het eerste element van *avg* steeds wordt weggegooid bij het genereren van de fitting functie, omdat dit abnormaal groot zou kunnen zijn wegens latency bij het opstarten van bepaalde Kernels etc. Dit element weglaten geeft dus een accuratere trendlijn.
Hierna plot de functie elk voorschrift in *fit* tussen 0 en de grootste waarde voor het aantal rijen dat in de testdata zit. Dit is opgeslagen in de variabele *plot*.
Ten slotte worden *plot* en *listPlot* op één grafiek weergegeven aan de hand van de functie *Show*.
Voor het visualiseren van de niet-recursieve implementatie zijn kleine aanpassingen gebeurd aan de *testPerformance* functie, zodanig dat een mooie visualisatie ontstaat voor het grote bereik van de data uit deze tests. Deze aanpassingen zijn louter esthetisch van aard en zeer beperkt in omvang. Daarom heeft het geen zin deze aanpassingen in detail te bespreken.

6.3 Experimenten

Nu een concrete testapplicatie afgebakend is en een algemene testmethode gespecifieerd is, is het van belang concrete tests te definiëren om toe te passen op de verschillende in dit onderzoek opgenomen programmeertalen en cloud-platformen. Het hoofddoel van deze masterproef is immers om onder andere aan de hand van deze experimenten functionele en traditionele OO-talen in de cloud met elkaar te vergelijken. Deze paragraaf geeft een overzicht van de te testen parameters en de manier waarop dit best gebeurt. De keuze van de parameters is gebaseerd op paragraaf 2.3.1.

6.3.1 Performantie

Een eerste belangrijk aspect om de verschillende talen experimenteel mee te vergelijken is performantie. Voor rekenintensieve toepassingen is dit immers een doorslaggevende factor in de keuze van een programmeertaal voor een bepaald project, zoals paragraaf 2.3.1 reeds aangaf.

Deze parameter is vrij eenvoudig te testen aan de hand van de in vorige paragraaf beschreven testmethode. In elke taal dient zo uniform mogelijk de testapplicatie beschreven in paragraaf 6.1 geïmplementeerd te worden, waarna bovenstaande testmethode eenvoudig kan worden toegepast op elk van deze implementaties. Zoals in paragraaf 6.1.2.3 reeds aangewezen, is dit vooral zinvol voor de recursieve versie van het Driehoek van Pascal-algoritme, aangezien deze versie veel meer rekenintensief is. De niet-recursieve variant is in beperkte mate echter ook interessant, zoals diezelfde paragraaf ook aangeeft. Het uitgangspunt van de tests is dus de recursieve versie, hoewel de niet-recursieve versie ook beperkt besproken kan worden.

Om een zo transparant mogelijk beeld te kunnen vormen van de relatieve performantie van de verschillende talen is het verstandig de performantietests in eerste instantie op een lokale test-pc uit te voeren, waardoor de hardware voor elke test dezelfde is. Zo kunnen de verschillende talen transparant met elkaar vergeleken worden, alvorens de implementaties naar de cloud te deployen. Zo is later ook eenduidig vast te stellen wat de relatieve performantie is van de verschillende cloud-omgevingen ten opzichte van elkaar. Hierdoor kan ook de kwaliteit van de infrastructuur van de verschillende CSP's beoordeeld worden.

6.3.2 Parallelliseerbaarheid

Vervolgens loont het ook de moeite om te kijken in hoeverre de verschillende bestudeerde talen parallelliseerbaar zijn. Dit is voor deze masterproef een heel interessant aspect, aangezien parallellisatie wordt gepromoot voor zowel functioneel programmeren als cloud computing. Hiervoor dienen de implementaties van het Driehoek van Pascal-algoritme in elke taal eerst herschreven te worden zodat ze van parallellisatie gebruik maken. Dit kan op meerdere manieren, onder andere door bijvoorbeeld elke coëfficiënt van de Driehoek van Pascal in een aparte thread uit te rekenen. Paragrafen 9.1 en 9.2 gaan uitgebreid in op de verschillende mogelijke vormen van parallellisatie, en hoe deze geïmplementeerd zijn in de verschillende bestudeerde talen. De niet-geparallelliseerde variant van het recursief Driehoek van Pascal-algoritme kan voor deze tests steeds als referentie gebruikt worden.

Ook parallellisatie kan best voor elke taal eerst op een lokale test-pc getest worden, en daarna vergeleken worden met testresultaten voor dezelfde implementaties in de cloud. Dit om dezelfde redenen als beschreven in vorige paragraaf.
Door de vergelijking te maken tussen de lokale en cloud-gedeployde versie van de geparallelliseerde Driehoek van Pascal, is het bovendien ook mogelijk te zien in hoeverre cloud-computing kan profiteren van de grote parallellisatiemogelijkheden van functioneel programmeren. Uit paragraaf 2.1.6.2 is immers geweten dat cloud computing van nature zeer parallelliseerbaar is, door de gedistribueerde natuur ervan waarbij hardware gevirtualiseerd is. In theorie zouden de cloud-implementaties dus moeten uitblinken op dit gebied, en veel betere testresultaten moeten tonen dan de lokale test-pc, die maar een beperkt aantal CPU-kernen ter beschikking heeft.

6.3.3 Connectiesnelheid

Om verbinding te maken met een API in de cloud is een zekere tijd nodig. Indien deze connectietijd te hoog is, is een cloud-applicatie niet geschikt voor bepaalde toepassingen, zoals bijvoorbeeld SOA-toepassingen en in het bijzonder microservices, aangezien in dit soort applicaties typisch meerdere verschillende cloud-gedeployde (deel)applicaties informatie uitwisselen. Dit soort toepassingen maakt dus meerdere verschillende connecties met verschillende API's, elk met een zekere connectietijd, voor het afhandelen van een enkele request. Indien de connectiesnelheid van deze services te lang is, kan de som van de connectietijden van alle aangesproken services samen een onacceptabele vertraging opleveren.
De connectiesnelheid is in principe een constante voor elke request. Deze connectiesnelheid is afhankelijk van een heel aantal factoren. De locatie van de cloud-servers en de manier waarop de CSP binnenkomend verkeer verwerkt spelen een doorslaggevende rol in de uiteindelijke grootte van de connectietijd, maar ook de internetverbinding van de gebruiker, de belasting van het netwerk op het moment waarop de request verstuurd wordt, etc.
Connectiesnelheid is dus een belangrijk aspect waarop de verschillende talen getest dienen te worden. Dit aspect heeft rechtstreeks weinig te maken met de geteste programmeertaal. Onrechtstreeks is deze parameter toch afhankelijk van de gebruikte taal, aangezien de beschikbare cloud-providers sterk afhankelijk zijn van de gebruikte programmeertaal. Hierdoor is er een onrechtstreeks verband tussen de connectiesnelheid en de gebruikte taal. Het is dus in een ideaal scenario nodig om voor elke taal minstens één cloud provider te vinden waarbij de connectietijd laag is.
Voor het testen van de connectiesnelheid volstaat het om een cloud-implementatie voor elke taal te kiezen, en te testen met behulp van de testomgeving beschreven in paragraaf 6.2. Eén belangrijke aanpassing is wel nodig. Aangezien de connectiesnelheid constant is voor elke request, is het vanzelfsprekend dat bij de curve fitting een constante wordt bijgeteld voor de cloud-gedeployde versies van het Driehoek van Pascal-algoritme. Bijlage A geeft een overzicht van hoe de *fittingFunction*-functie uit paragraaf 6.2.3 er uiteindelijk uit ziet, rekening houdende met dit feit.

6.3.4 Schaalbaarheid

Het laatste belangrijk aspect dat onderzocht dient te worden is de schaalbaarheid van de cloud-gedeployde applicaties. Dit is het vermogen van de cloud service om zichzelf op te schalen in functie van een hoge vraag. Als er veel requests op korte tijd aankomen, mag er immers geen vertraging zijn in de dienstverlening. Dit aspect is enkel op web- en cloud-gerelateerde applicaties van toepassing. Voor niet-web-gebaseerde toepassingen is van schaalbaarheid immers geen sprake, en in de meeste gevallen is hier ook geen nood aan.
Om de schaalbaarheid te testen maakt dit onderzoek gebruik van een testprogramma dat speciaal voor hiervoor geschreven is in Java. Het principe van dit programma is om een grote reeks http-requests te kunnen sturen naar de cloud service, en te kijken of de tijd nodig voor het afhandelen van de request verandert in functie van de hoeveelheid tijd tussen de verschillende requests. Dit laat ook toe patronen te zien in de tijd nodig voor het afhandelen van elke afzonderlijke request.
Het basisonderdeel van dit Java-programma is dus code om een http-request te sturen en ontvangen. Figuur 135 geeft deze code weer.

```
void fetchData() throws MalformedURLException, IOException{
    InputStream is = new URL(url).openStream();
    BufferedReader rd = new BufferedReader(new InputStreamReader(is,
Charset.forName("UTF-8")));
    StringBuilder sb = new StringBuilder();
    int cp;
    while ((cp = rd.read()) != -1) {
        sb.append((char) cp);
    }
    is.close();
}
```

Figuur 135: Code voor het versturen van een http-request en het verwerken van de response van de server.

De *fetchData*-functie geeft niets terug, aangezien de exacte data die de server terugstuurt niet van belang is. De hele test zit in een klasse met de URL van de te testen als datamember. Vandaar het gebrek aan parameters. Voor de rest is de code zelfverklarend.
Een belangrijk probleem bij bovenstaande code is dat het hele programma steeds zal wachten bij elke request tot de server een respons stuurt. Indien geen speciale maatregelen genomen worden, zou het dus onmogelijk zijn meerdere requests op zeer korte tijd te sturen. De oplossing voor dit probleem bestaat erin voor elke request een aparte thread aan te maken, die het versturen van de request en het afhandelen van de server response voor zijn rekening neemt. Zo kunnen toch meerdere requests parallel verstuurd worden. De code hiervoor is als weergegeven in Figuur 136.

```
List<Thread> generateThreads(int iterations, int separation){
    List<Thread> threadList = new ArrayList();
    for (int i = 0; i< iterations; i++){
        final int iBuffer = i;
        Thread t = new Thread(new Runnable() {
            @Override
            public void run() {
                long t1 = System.currentTimeMillis();
                try {
                    fetchData();
                } catch (IOException ex) {

Logger.getLogger(TestRun.class.getName()).log(Level.SEVERE, null, ex);
                }
                results[iBuffer] = (double)((System.currentTimeMillis() -
t1)/1000.0);
            }
        });
        t.start();
        threadList.add(t);
        try {
            Thread.sleep(separation);
        } catch (InterruptedException ex) {
            Logger.getLogger(TestRun.class.getName()).log(Level.SEVERE,
null, ex);
        }
    }
    return threadList;
}
```

Figuur 136: Code voor het genereren van een groot aantal requests in Java.

Bovenstaande code genereert een reeks threads die elk een request sturen naar de cloud service, en de tijd die de server nodig heeft om een respons terug te sturen bijhouden. Na het terugkrijgen van een respons voegt elke thread de gepasseerde tijd toe aan de *results*-datamember. Het aantal te versturen requests kan gevarieerd worden aan de hand van de parameter *iterations*. Zo ontstaat voor de resultaten een array van doubles die *iterations* lang is.

Een laatste probleem dat verholpen dient te worden voor deze tests is het feit dat de connectietijd sterk verlaagd is als er op een gegeven moment een openstaande verbinding is met de server. Dit komt doordat de router voor een request met een bestemming die nog niet in de routing tables van de router zit opgeslagen eerst een pad moet zoeken naar de server waar de request voor bestemd is. Eenmaal dit gebeurd is voor één request, kunnen de volgende requests die worden gestuurd zolang de originele connectie open is veel sneller verstuurd worden. Dit betekent dat de eerste paar connecties veel langer duren dan de andere. Dit is afhankelijk van het tijdsinterval tussen de verschillende requests. Om ervoor te zorgen dat dit geen impact heeft op de testresultaten, is het belangrijk erop toe te zien dat er eerst een openstaande connectie aanwezig is voordat de testresultaten gegenereerd worden. Figuur 137 geeft weer hoe dit is aangepakt voor dit onderzoek.

```
public TestRun(String url,int interval){
    this.url = url;
    this.results = new double[100];
    generateThreads(25,80);
    List<Thread> threads = generateThreads(100,interval);
    for (Thread t : threads){
        try {
            t.join();
        } catch (InterruptedException ex) {

        }
    }
```

Figuur 137: Constructor voor een TestRun.

Eerst wordt de *generateThreads*-methode opgeroepen voor 25 requests met een interval van 80 ms. Deze getallen zijn zodanig gekozen dat er gegarandeerd een geldige connectie openstaat wanneer de eigenlijke test wordt uitgevoerd, zolang het afhandelen van de requests minstens 80 ms en maximaal 2 s duurt. Zo goed als alle servers vallen binnen dit interval. De volgende aanroep van *generateThreads* omvat de eigenlijke testresultaten. Deze bestaan uit 100 verschillende calls naar de te testen API. De join-functie garandeert dat de constructor van het TestRun object pas verlaten wordt wanneer alle requests afgehandeld zijn. Dit laatste aantal mag ook niet te hoog zijn, aangezien het netwerk overbelast zou kunnen raken, waardoor de testresultaten niet meer correct zouden zijn. Bovendien moet er rekening gehouden worden met de beperkte hoeveelheid krediet voor de te testen cloud-services.
Om de resultaten van bovenstaande testcode te visualiseren is er gebruik gemaakt van Wolfram, net als bij de vorige tests. Voor de schaalbaarheidstest is wel een volledige nieuwe testbench geschreven, aangezien deze test anders functioneert dan de andere.
Om te beginnen wordt er een lijst van URL's van de verschillende cloud services bijgehouden. Er is ook een *decodeUrl*-functie voorzien, die dezelfde functie en opbouw heeft als de *data*-functie uit paragraaf 6.2.2. Het is nodig om deze functionaliteit opnieuw te voorzien, omdat het van groot belang is de URL's te finetunen voor de in deze paragraaf toegelichte test. Hierover meer in hoofdstuk 0. De code voor de twee hierboven vermelde delen spreekt voor zich en is voor de volledigheid in Bijlage B te vinden.
Het laatste deel van de Wolfram-testbench bestaat uit de *testScalability*-functie. Deze is net als de *testPerformance*-functie uit paragraaf 6.2.3 gebaseerd op het genereren van een *ListPlot* van de *testData*. Figuur 138 geeft de code hiervoor weer.

```
testScalability[lang_String] := (
testData = JavaNew["scalabilitytester.TestRun", decodeUrl[lang], #]@getResults[] & /@{100, 10};
{ListPlot[testData,
PlotLabel → "Scalability test " <> lang,
PlotRange → {0, Max[Max[#] & /@ testData]},
PlotLegends → {"100 ms interval", "10 ms interval"},
ImageSize → Large,
TargetUnits → "Seconds"], {Mean[#], StandardDeviation[#]} & /@ testData})
```

Figuur 138: Code voor het genereren van schaalbaarheidstestresultaten en het weergeven hiervan in Wolfram.

Zoals te zien is voert deze code twee testruns uit. De ene met een interval van 100 ms, en de andere met een interval van 10 ms. Door de URL zodanig te kiezen dat de aangeroepen API ongeveer 100 ms nodig heeft voor het verwerken van de request, kan de schaalbaarheid van de geteste cloud-service eenduidig bepaald worden. Voor dit onderzoek is het eenvoudig om de URL zodanig te kiezen dat het afhandelen van de request ongeveer 100 ms duurt door de grootte van de opgevraagde Driehoek van Pascal juist te kiezen.

7 Implementatie

Dit hoofdstuk beschrijft in detail de implementatie van zowel de niet-recursieve als recursieve vorm van het Driehoek van Pascal-algoritme in de verschillende voor dit onderzoek gekozen talen. Bovendien vergelijkt dit hoofdstuk de implementaties op gebied van lengte van de code, leesbaarheid, programmeergemak, etc.

7.1 Recursief

Zoals in paragraaf 6.1.2.3 reeds aangegeven, is de recursieve implementatie meer geschikt voor het vergelijken van performantie tussen verschillende talen dan de niet-recursieve. Daarom is deze versie van het Driehoek van Pascal-algoritme in alle bestudeerde talen uitgewerkt. Hieronder de exacte implementatie in de verschillende talen, vergezeld van enige toelichting.

7.1.1 Java

In Java ziet de implementatie van de recursieve Driehoek van Pascal eruit als in Figuur 139.

```
package triangles;

public class PascalTriangleRec {
    private final int[][] points;

    public PascalTriangleRec(int size){
        points = new int[size][];
        for (int i =0;i<size;i++){
            int[] row  = new int[i+1];
            for (int j = 0;j<=i;j++){
                row[j]=getValueAtPoint(i,j);
            }
            points[i]=row;
        }
    }
    public static int getValueAtPoint(int row, int col){
        if (col == 0 || col == row) return 1;
        else return getValueAtPoint(row-1,col-1)+getValueAtPoint(row-1,
col);
    }
    public int[][] getPoints(){
        return points;
    }
}
```

Figuur 139: Implementatie van de recursieve Driehoek van Pascal in Java.

De Java-implementatie bestaat uit de klasse PascalTriangleRec, die als data member een 2D array van ints heeft die points heet. Hierin zit de driehoek opgeslagen. Ook is er een getter voorzien om de punten op te vragen. Inclusief getter zijn er een 20-tal regels code nodig om deze Driehoek van Pascal aan te maken.
Deze implementatie maakt gebruik van geneste for-lussen. Op zich is dit nog vrij overzichtelijk en leesbaar, maar te veel nesting kan verwarrend zijn. Verder valt op dat er een aantal verschillende declaraties nodig zijn om waardes toe te wijzen aan de juiste

objecten. Deze declaraties moeten ook op specifieke plaatsen gebeuren, of anders werkt het algoritme niet correct. Dit kan voor veel verwarring zorgen; vooral bij minder ervaren programmeurs.

7.1.2 Java met Streams

Zoals in paragraaf 2.2.5.6 beschreven beschikt Java sinds Java 8 over de mogelijkheid om functioneel te programmeren aan de hand van Lambda-expressies. In Java zijn functionele lijstbewerkingen geïmplementeerd in de vorm van Streams. In het kader van deze masterproef is het interessant om de implementatie van de recursieve Driehoek van Pascal in Java met behulp van Streams te bestuderen. Dit geeft het volgende resultaat in Figuur 140.

```
package triangles;

import java.util.List;
import static java.util.stream.Collectors.toList;
import java.util.stream.IntStream;

public class FunctionalPascalTriangle {
    private List<List<Integer>> points;

    public FunctionalPascalTriangle(int size){
        points = IntStream.range(0, size).boxed()
                .map(row->IntStream.range(0, row+1).boxed()
                        .map(col->getValueAtPoint(row, col))
                        .collect(toList()))
                .collect(toList());
    }
    public static int getValueAtPoint(int row, int col){
        if (col == 0 || col == row) return 1;
        else return getValueAtPoint(row-1,col-1) + getValueAtPoint(row-
1,col);
    }
    public List<List<Integer>> getPoints(){
        return points;
    }
}
```

Figuur 140: Implementatie van de recursieve Driehoek van Pascal in Java met Streams.

De implementatie met Streams is iets korter dan die zonder in Java. Door het gebrek aan geneste for-lussen is de code ook overzichtelijker. Bovendien is de kans op fouten tijdens het programmeren ook aanzienlijk kleiner door het gebrek aan declaraties en toewijzingen die op specifieke plaatsen moeten gebeuren.
Merk op dat deze implementatie gebruik maakt van nested Lists van ints, in plaats van een 2D-Array. De reden hiervoor is dat in Arrays in Java blijkbaar een oude glitch aanwezig is, waardoor deze implementatie met Streams niet evident is aan de hand van een 2D-Array. Deze bug wordt in detail beschreven in [78]. Uiteindelijk heeft deze kleine afwijking minimale gevolgen voor de performantie, aangezien het berekenen van de verschillende coëfficiënten van de Driehoek van Pascal vele malen meer tijd vergt dan het plaatsen ervan in een Lijst.

7.1.3 C#

Naast de Java-implementatie is er ook één in C# gemaakt. Figuur 141 geeft deze weer.

```
namespace DotNetTriangles
{
    class PascalTriangle
    {
        public int[][] Points { get; set; }

        public PascalTriangle(int size)
        {
            Points = new int[size][];
            for (int i = 0; i < size; i++)
            {
                int[] row = new int[i + 1];
                for (int j = 0; j <= i; j++)
                {
                    row[j] = GetValueAtPoint(i, j);
                }
                Points[i] = row;
            }
        }
        public static int GetValueAtPoint(int row, int col)
        {
            if (col == 0 || col == row) return 1;
            else return GetValueAtPoint(row-1, col-1) + GetValueAtPoint(row-1, col);
        }
    }
}
```

Figuur 141: Implementatie van de recursieve Driehoek van Pascal in C#.

De C# implementatie is grotendeels identiek aan de Java-versie. De C#-implementatie is wel iets eleganter aangezien er geen expliciete getter voorzien moet worden voor de Points.

7.1.4 C# met LINQ

Ook C# bevat zoals reeds in paragraaf 2.2.5.6 aangegeven functionele eigenschappen. In .NET spreekt men van Language Integrated Query (LINQ) in plaats van Streams wanneer men het heeft over functionele lijstbewerkingen. De C# implementatie met LINQ is ook gerealiseerd en getest voor deze masterproef. Figuur 142 geeft het resultaat van deze implementatie weer.

```
using System.Collections.Generic;
using System.Linq;

namespace DotNetTriangles
{
    class FunctionalPascalTriangle
    {
        public List<List<int>> Points { get; set; }

        public FunctionalPascalTriangle(int size)
        {
            Points=Enumerable.Range(0, size)
                .Select(row => Enumerable.Range(0, row + 1)
                    .Select(col => GetValueAtPoint(row, col))
                    .ToList())
                .ToList();
        }
        public static int GetValueAtPoint(int row, int col)
        {
            if (col == 0 || col == row) return 1;
            else return GetValueAtPoint(row-1,col-1) + GetValueAtPoint(row-1,col);
        }
    }
}
```

Figuur 142: Implementatie van de recursieve Driehoek van Pascal in C# met LINQ.

Deze implementatie is effectief korter dan de zuiver OO-implementatie, vooral omdat de LINQ libraries minder breedsprakerig en nog iets eleganter zijn dan de Java Streams. Ook hier is gebruik gemaakt van nested Lists in plaats van een 2D-Array, in de eerste plaats om de Java-implementatie zo getrouw mogelijk na te bootsen. De functionele implementatie in C# heeft voldoende aan slechts 5 korte regels code om de Driehoek van Pascal te genereren. Inclusief encapsulatie zijn dit een 20-tal regels exclusief witruimte, wat toch een verbetering is ten opzichte van de OO-versie in deze taal.

7.1.5 F#

Verder is er ook nog een zuiver functionele implementatie voorzien in F#. Figuur 143 laat de code hiervan zien.

```
namespace Triangles

module PascalTriangle =

    let rec pascalTriangleValue row col =
        match col with
            | 1 -> 1
            | col when col = row -> 1
            | _->pascalTriangleValue (row-1) (col-1) + pascalTriangleValue (row-1) col

    let pascalTriangle size = [1..size] |>List.map (fun r->[1..r] |> List.map (fun c-
>pascalTriangleValue r c))
```

Figuur 143: Implementatie van de recursieve Driehoek van Pascal in F#.

Hier is aanzienlijk minder code nodig om tot het juiste resultaat te komen. De code is dus veel compacter en overzichtelijker dan de alternatieven in Java en C#. Toch is een zeer duidelijke structuur zichtbaar aan de hand van de namespace declaratie, de duidelijke match-clause, het gebruik van de let en rec *keywords* en de piping syntax. Deze structuur is deels opgelegd en maakt deel uit van de syntax, zoals reeds beschreven in paragraaf 4.1. Hierdoor is de code bijzonder leesbaar en overzichtelijk, ondanks de compacte schrijfwijze. Ook valt op dat accolades en andere syntactische formaliteiten zo goed als volledig afwezig zijn in deze implementatie.
Wat wel opviel tijdens het schrijven van deze implementatie is dat de syntax radicaal anders is dan voor de meeste programmeertalen, en daardoor de manier van werken compleet anders is dan bij de andere implementaties. Online was er voor F# beduidend minder documentatie te vinden die snel overzichtelijk kon aangeven hoe een dergelijk probleem in F# best aangepakt wordt. De abstracte functionele denkwijze die in F# sterk aanwezig is creëert daarentegen juist een grote behoefte voor duidelijke en concrete hulpkanalen voor dergelijke praktische problemen. Zonder een grondige kennis van de syntax en documentatie van F# is het dan ook moeilijk om een implementatie in F# te realiseren, terwijl dit voor andere talen veel minder een probleem vormt. Voor onervaren programmeurs is dit een significant nadeel van F#.

7.1.6 Wolfram

Ten slotte is er nog een implementatie voorzien in de Wolfram Language. Figuur 144 toont hiervan het resultaat.

```
pascalTriangleRec[size_Integer] /; size == 1 := {{1}}
pascalTriangleRec[size_Integer] :=Table[pascalTriangleValue[i, j], {i, 1, size}, {j, 1, i}]
pascalTriangleValue[row_Integer, col_Integer] /; col == 1 || col == row := 1
pascalTriangleValue[row_Integer, col_Integer] :=
pascalTriangleValue[row - 1, col - 1] + pascalTriangleValue[row - 1, col]
```

Figuur 144: Implementatie van de recursieve Driehoek van Pascal in de Wolfram Language.

Deze implementatie telt slechts een vijftal regels en is dus de kortste van allemaal. De functionaliteit van de code is toch nog steeds overzichtelijk. Bovendien zijn er geen namespace, package, of klasse definities nodig en kan deze code rechtstreeks in eender welk ander Wolfram code document geïntegreerd worden, terwijl voor alle andere implementaties eerst een omvattend project aangemaakt moet worden, en zekere stappen moeten ondernomen worden om de implementatie van de code zelf te integreren in dit project zodanig dat de code op een zinvolle en eenvoudige manier te gebruiken is. Dit alles is niet nodig in Wolfram. Dit is enerzijds een voordeel aangezien de code zeer compact is en nergens rekening mee gehouden moet worden tijdens het schrijven ervan, maar anderzijds kan het ook als een nadeel gezien worden aangezien er geen structurerende elementen aanwezig zijn in de code. Het is dan ook volledig aan de programmeur om structuur te brengen in een groot Wolfram programma. Zoals beschreven in paragraaf 2.2.5.5 reikt Wolfram hier wel een groot aantal hulpmiddelen voor aan. Wolfram geeft de programmeur dus de vrijheid om naar eigen believen structuur aan te brengen in zijn code.
Door de grote flexibiliteit van Wolfram is wel gebleken dat voorzichtig dient omgesprongen te worden met zaken als pattern matching. Wolfram erkent immers snel een patroon dat niet bedoeld was door de programmeur door de krachtige interpretatie-eigenschappen die het bezit. Gelukkig is debuggen in Wolfram wel zeer eenvoudig omdat een enkele regel code rechtstreeks kan worden uitgevoerd, en het probleem zeer snel en eenvoudig geïsoleerd kan worden op die manier.

7.2 Niet-recursief

Zoals in paragraaf 6.1.2.3 reeds aangegeven is het interessant om naast de recursieve implementatie ook de niet-recursieve variant te bestuderen.
Deze versie van het Driehoek van Pascal-algoritme is enkel in Java en Wolfram geïmplementeerd. Dit is immers voldoende om te besluiten of er een verschil bestaat in relatieve performantie van de verschillende talen tussen de recursieve en niet-recursieve implementaties. Hieruit kunnen voldoende kwalitatieve conclusies getrokken worden, zonder te diep in te gaan op de exacte kwantitatieve resultaten voor elke taal. Zoals beschreven in paragraaf 2.3.1 is een exacte kwantitatieve beoordeling van de performantie van de verschillende talen immers niet echt relevant.

7.2.1 Java

Allereerst de Java-implementatie van de niet-recursieve Driehoek van Pascal, weergegeven in Figuur 145.

```
package triangles;

import java.math.BigInteger;

public class PascalTriangle {
    private final BigInteger[][] points;

    public PascalTriangle(int size){
        points = new BigInteger[size][];
        for (int i = 0;i<size;i++){
            points[i]=new BigInteger[i+1];
            for (int j = 0; j<=i;j++){
                try{
                    points[i][j]=points[i-1][j-1].add(points[i-1][j]);
                }
                catch(ArrayIndexOutOfBoundsException ex){
                    points[i][j]=BigInteger.ONE;
                }
            }
        }
    }
    public BigInteger[][] getPoints(){
        return points;
    }
}
```

Figuur 145: Implementatie van de niet-recursieve Driehoek van Pascal in Java.

De code is ongeveer even lang als de recursieve versie, maar wel minder overzichtelijk door het gebruik van try-catch en nested for-loops. Verder valt een overstap van een 2D array van ints naar een 2D array van BigIntegers op. Deze is nodig omdat de getallen in de niet-recursieve Driehoek van Pascal veel groter kunnen worden dan in de recursieve variant, aangezien de niet-recursieve variant veel sneller is dan de recursieve. Het aantal rijen en de bijhorende coëfficiënten die berekend moeten worden om een representatieve test te kunnen uitvoeren zijn dus veel groter.

7.2.2 Wolfram

Figuur 146 toont de implementatie van de niet-recursieve Driehoek van Pascal in Wolfram.

```
pascalTriangle[size_Integer] /; size == 1 := {{1}}
pascalTriangle[size_Integer] := buildTriangle[size, pascalTriangle[1]]
buildTriangle[size_Integer, t_] :=If[size == Length[t] + 1, addRow[t], buildTriangle[size, addRow[t]]]
addRow[t_] := Append[t,Flatten[{1, Flatten [t[[-1 ;;]]][[2 ;;]] + Flatten[t[[-1 ;;]]][[ ;; -2]], 1}]]
```

Figuur 146: Implementatie van de niet-recursieve Driehoek van Pascal in de Wolfram Language.

De Wolfram-code is opnieuw slechts een vijftal regels groot. Ze ziet er op het eerste zicht wel complex uit, maar als de betekenis van de List-operations goed gekend is in Wolfram is de code nog steeds goed leesbaar. Er zijn geen speciale aanpassingen nodig aan types, aangezien Wolfram met dynamic typing werkt. Wel was voor deze implementatie niet meteen duidelijk hoe de functie Flatten gebruikt diende te worden. Het was niet evident om de volledig correcte notatie te vinden. Opnieuw is dit uiteindelijk geen al te groot probleem gebleken aangezien debuggen in Wolfram zeer flexibel en eenvoudig is.

7.3 Algemeen

Nu de implementatie in elke taal gekend is, is het nuttig om alle vergelijkingscriteria uit paragraaf 2.3.1 die betrekking hebben op implementatie naast elkaar te zetten voor alle bestudeerde talen. Dit gebeurt vooral op basis van de recursieve implementaties, hoewel dezelfde conclusies ook van toepassing zijn op de niet-recursieve implementaties. Tabel 3 geeft een overzicht weer van dit alles.

Tabel 3: Vergelijking implementatie verschillende talen.

Taal	Leesbaarheid	Ontwikkelingstijd	Lengte	Foutenlast	Programmeer-gemak
Java	OK	Lang	± 20 regels	Groot	Goed
Java met Streams	Goed	Gemiddeld	± 20 regels	Klein	OK
C#	OK	Lang	± 25 regels	Groot	Goed
C# met LINQ	Goed	Kort	± 20 regels	Klein	Goed
F#	Uitstekend	Kort	± 10 regels	Klein	Slecht
Wolfram	Goed	Zeer kort	± 5 regels	Gemiddeld	Uitstekend

De object-georiënteerde implementaties in Java en C# zijn zeer gelijkaardig over heel de lijn. Naar implementatie toe blinken ze op geen enkel gebied uit. Enkel voor programmeergemak scoren ze goed vanwege de grote hoeveelheid ondersteuning die voor deze talen online te vinden is door het grote aantal gebruikers ervan. Ook is de syntax intuïtief. De ontwikkelingstijd en foutenlast zijn als slecht gecategoriseerd, aangezien door de imperatieve aard van de talen en inherente neveneffecten het moeilijk kan zijn de verschillende variabelen in de controlestructuren op exact de juiste manier te gebruiken zodat het gewenste resultaat ontstaat.
De functionele implementaties in Java en C# zijn ook vrij gelijkaardig. Ze scoren bovendien beter dan de OO-versies op zowat elk gebied. Vooral naar foutenlast toe bieden deze implementaties grote voordelen, aangezien de declaratieve schrijfwijze ervoor zorgt dat de programmeur geen ingewikkelde controlestructuren meer moet uitschrijven, wat veel mogelijke fouten elimineert. De implementatie in C# met LINQ is wel de betere van de twee, doordat de syntax natuurlijker is. Dit spaart code en ontwikkelingstijd. Voor programmeergemak scoort Java met Streams dus net iets minder dan de OO-versie in Java en beide versies in C#. Ondersteuning is voor zowel Streams als LINQ wel talrijk aanwezig.
De zuiver functionele implementaties in F# en Wolfram scoren naar implementatie toe in het algemeen het beste van alle bestudeerde talen. Zowel lengte van de code als ontwikkelingstijd tonen zeer goede resultaten voor F# en in het bijzonder Wolfram. De grootste verschillen tussen de twee zijn zichtbaar in de foutenlast en het programmeergemak, alsook leesbaarheid. Wolfram heeft een iets grotere foutenlast doordat het dynamically typed is, en de taal zodanig flexibel is dat de flexibiliteit soms de programmeur verrast. Wolfram compenseert wel gedeeltelijk voor het vrij grote risico op het maken van fouten doordat het zo eenvoudig te debuggen is. Daarnaast is Wolfram zodanig compact dat de leesbaarheid soms ietwat vertroebelt. Deze is wel nog steeds beter dan bij de OO-implementaties vanwege de declaratieve natuur van Wolfram.
F# scoort langs de andere kant topscores voor leesbaarheid en foutenlast door de overzichtelijke, minimalistische syntax met opgelegde regels die de leesbaarheid bevorderen, en de combinatie van het strikte typesysteem en type-inferentie. Op het gebied van programmeergemak moet F# echter onderdoen voor alle andere talen, aangezien de syntax radicaal verschilt van andere programmeertalen, waardoor het moeilijk is om F# te leren. Daarenboven is de online ondersteuning voor F# beperkt, en is het niet evident om snel een oplossing te vinden voor een probleem in F# via de gebruikelijke wegen zoals fora en andere online hulpmiddelen. Voor beginnende programmeurs is F# alleen al om die reden dan ook niet aan te raden.

7.4 Besluit

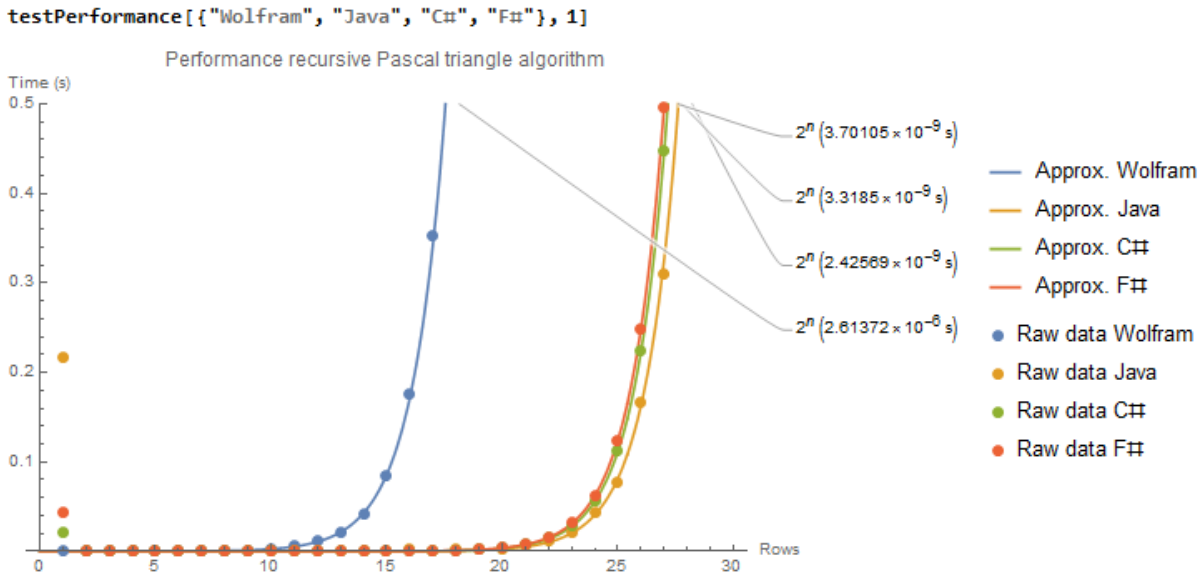
Functionele implementaties scoren over heel de lijn in het algemeen veel beter dan hun object-georiënteerde tegenhangers op het gebied van implementatie. Verschillende functionele talen hebben hun specifieke nadelen tegenover de andere talen, maar in het algemeen is er naar implementatie toe geen reden om niet van functioneel programmeren gebruik te maken. Voor F# is het echter aangeraden om eerst grondig de documentatie te bestuderen, alvorens aan de slag te gaan met de taal. Wolfram en zeker Streams en LINQ zijn voldoende intuïtief om gaandeweg op te pikken. Voor eenvoudige toepassingen is Wolfram de taal bij uitstek vanwege de zeer compacte syntax en grote flexibiliteit ervan.

8 Performantie

Dit hoofdstuk vergelijkt de performantie van de verschillende talen aan de hand van de hierboven beschreven Driehoek van Pascal-algoritmes. In eerste instantie vergelijkt dit hoofdstuk de recursieve implementatie in elke bestudeerde taal. Daarna volgt een vergelijking van de niet-recursieve implementatie in Java en Wolfram. De redenen voor de keuze van deze werkwijze zijn reeds eerder toegelicht. Alle resultaten uit dit hoofdstuk zijn bekomen op basis van de tests beschreven in paragraaf 6.2.
Merk op dat al onderstaande resultaten op de lokale pc zijn verkregen. Zoals eerder vermeld is deze stap essentieel om een waardevol oordeel te kunnen vellen over de performantie van de verschillende implementaties in de cloud, aangezien bij deze lokale tests de hardware voor alle implementaties gegarandeerd identiek is. Deze hardware bestaat uit een Intel Core i5 4690K quad-core processor met een klokfrequentie 4,4 GHz en 16GB RAM-geheugen met een klokfrequentie van 1666 MHz. In hoofdstuk 0 volgt een vergelijking tussen de in dit hoofdstuk verkregen resultaten en die in de verschillende geteste cloud-platformen.

8.1 Recursieve Driehoek van Pascal

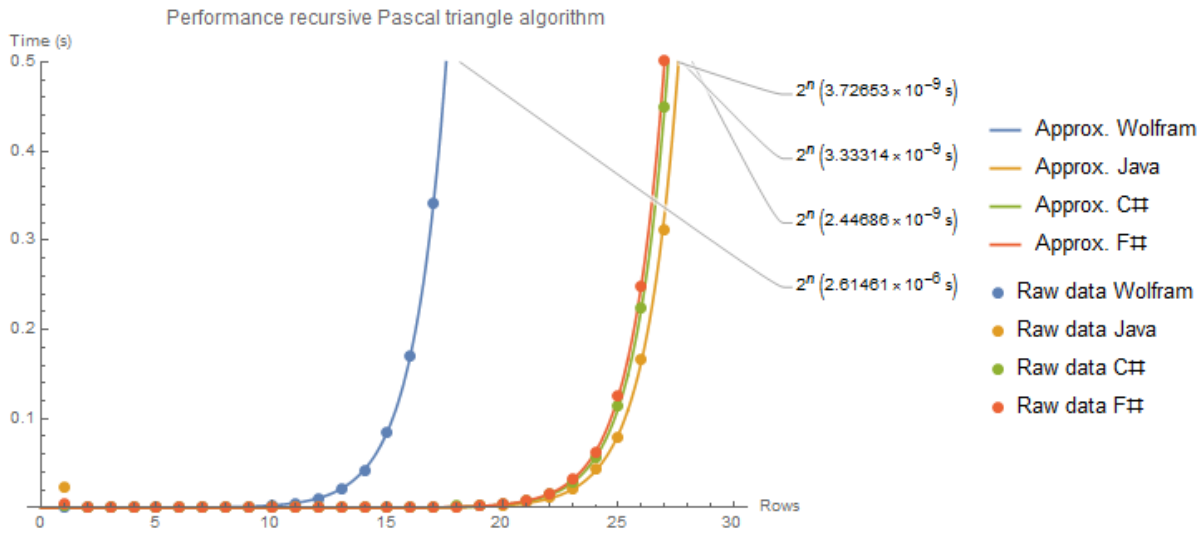
Allereerst is het nuttig een vergelijking te maken tussen de performantie van Wolfram, Java, C#, en F# voor de recursieve versie van de Driehoek van Pascal. Eerst en vooral is deze test uitgevoerd voor één iteratie. Figuur 147 geeft de resultaten van dit experiment weer.



Figuur 147: Vergelijking performantie recursieve Driehoek van Pascal in Wolfram, Java, C#, en F#.

Er zijn een aantal vaststellingen te maken op basis van bovenstaand resultaat. Om te beginnen schijnt Wolfram veel trager te zijn dan alle andere opties. Dit is waarschijnlijk te wijten aan het feit dat Wolfram functioneel en geïnterpreteerd is, wat de uitvoeringssnelheid van de code niet ten goede komt.
Verder is er bij de eerste meting voor elke optie behalve Wolfram een zekere latency. Dit heeft vermoedelijk te maken met het opstarten van de JRE/.NET runtime en het inladen van de nodige klassen wanneer die voor de eerste keer worden aangeroepen. Bij F# blijkt de

latency iets groter te zijn dan bij C#. Voor beide van deze talen is de latency echter verwaarloosbaar. Voor Java daarentegen is er een vrij grote latency van 0,25 seconden. Dit is toch een significante hoeveelheid, waarmee rekening gehouden dient te worden bij het grootschalig gebruik van J/Link in Wolfram.
Verder blijkt de benaderingsfunctie op basis van de berekende tijdscomplexiteit zeer goed te passen voor deze implementaties.
Om een meer correct beeld te krijgen over de performantie van de talen is bovenstaande test 10 keer achter elkaar uitgevoerd, zoals reeds toegelicht in paragraaf 6.3.1. Figuur 148 geeft het gemiddelde resultaat hiervan weer.



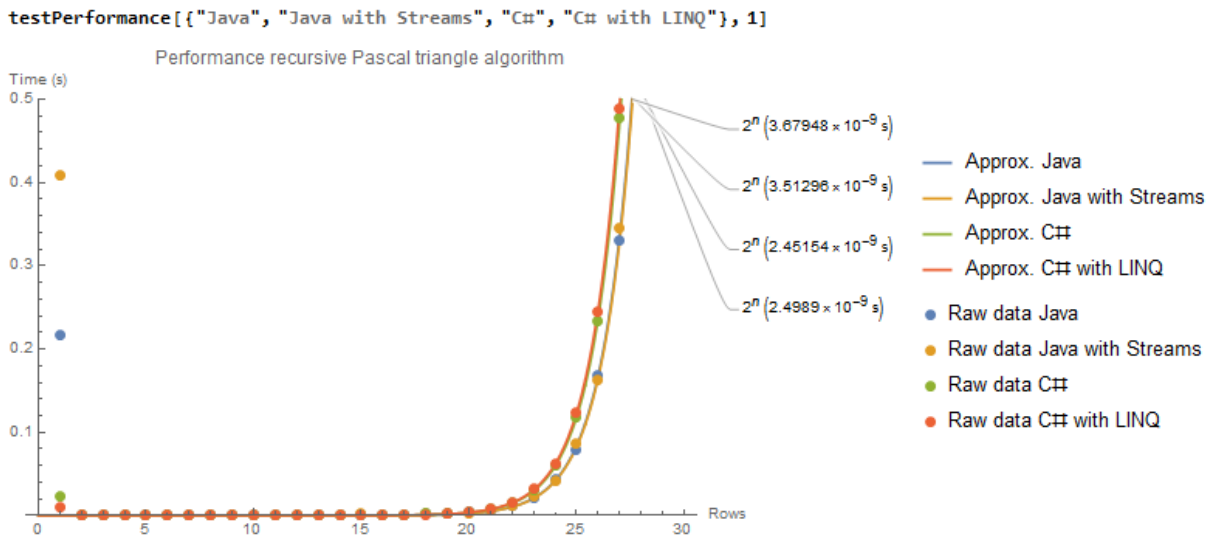
Figuur 148: Vergelijking performantie recursieve Driehoek van Pascal in Wolfram, Java, C#, en F# voor 10 iteraties.

De latency bij de eerste call is gereduceerd naar gemiddeld ongeveer 1/10 van de latency bij 1 call. Dit kan enkel als de gemiddelde latency van alle volgende calls ongeveer 0 is. Dit bevestigt het vermoeden dat de latency bij de eerste call enkel te maken heeft met het inladen van de nodige klasses/types.
Bij dit gemiddeld resultaat over 10 iteraties kan ook een betrouwbaar oordeel gevormd worden over de performantie van elk van de talen. Tabel 4 vergelijkt de coëfficiënten van de verschillende benaderingsfuncties.

Tabel 4: Relatieve performantie recursieve Driehoek van Pascal aan de hand van de coëfficiënten van de benaderingsfuncties.

	Wolfram	Java	C#	F#
Wolfram	1	$\frac{2{,}4 \cdot 10^{-9}}{2{,}6 \cdot 10^{-6}} \approx 0{,}0009$	$\frac{3{,}3 \cdot 10^{-9}}{2{,}6 \cdot 10^{-6}} \approx 0{,}0013$	$\frac{3{,}7 \cdot 10^{-9}}{2{,}6 \cdot 10^{-6}} \approx 0{,}0014$
Java	$\frac{2{,}6 \cdot 10^{-6}}{2{,}4 \cdot 10^{-9}} \approx 1070$	1	$\frac{3{,}3 \cdot 10^{-9}}{2{,}4 \cdot 10^{-9}} \approx 1{,}36$	$\frac{3{,}7 \cdot 10^{-9}}{2{,}4 \cdot 10^{-9}} \approx 1{,}52$
C#	$\frac{2{,}6 \cdot 10^{-6}}{3{,}3 \cdot 10^{-9}} \approx 784$	$\frac{2{,}4 \cdot 10^{-9}}{3{,}3 \cdot 10^{-9}} \approx 0{,}73$	1	$\frac{3{,}7 \cdot 10^{-9}}{3{,}3 \cdot 10^{-9}} \approx 1{,}12$
F#	$\frac{2{,}6 \cdot 10^{-6}}{3{,}7 \cdot 10^{-9}} \approx 702$	$\frac{2{,}4 \cdot 10^{-9}}{3{,}7 \cdot 10^{-9}} \approx 0{,}66$	$\frac{3{,}3 \cdot 10^{-9}}{3{,}7 \cdot 10^{-9}} \approx 0{,}89$	1

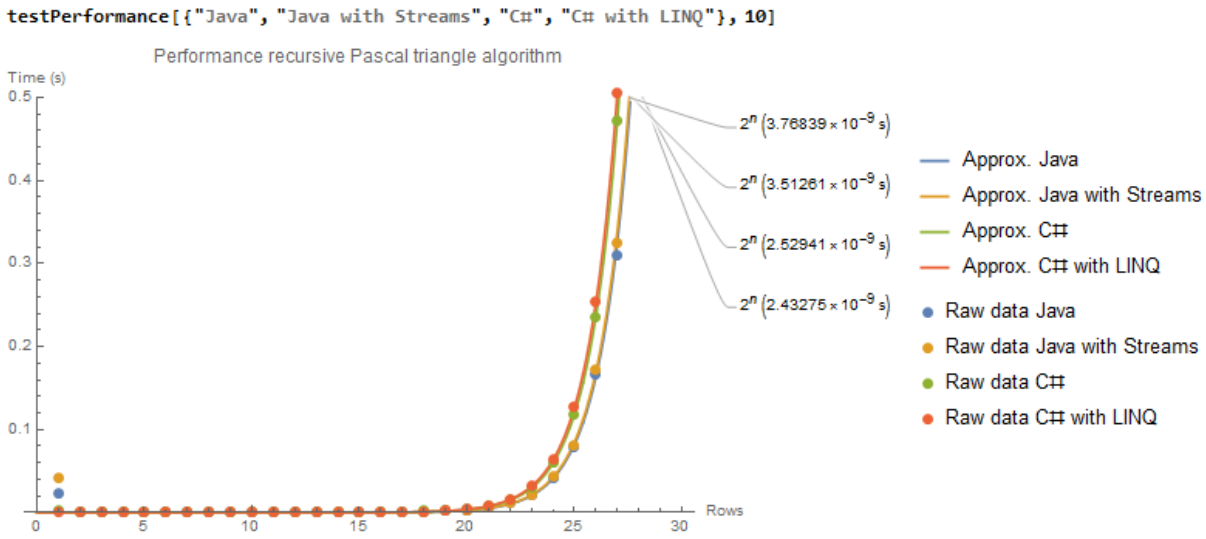
Het valt meteen op dat Wolfram het bijzonder slecht doet wat performantie betreft in vergelijking met de andere talen voor deze implementatie. Het is ongeveer 1000 keer trager dan Java en ongeveer 700 en 800 keer trager dan resp. F# en C#. Dit is toch vrij onacceptabel en heeft ernstige implicaties naar eender welke CPU-intensieve toepassing in Wolfram toe. Java blijkt voor deze implementatie qua performantie uit te blinken. Het is ongeveer 25% sneller dan C# en 35% sneller dan F#.
Ten slotte is vast te stellen dat F# het zeker niet slecht doet. Het is slechts ongeveer 10% trager dan C#. Dit past in het plaatje dat F# een gecompileerde functionele taal is. Van de meeste functionele programmeertalen is in de literatuur immers geweten dat ze in het algemeen de neiging hebben trager te zijn dan hun OO-tegenhangers, zoals beschreven in paragraaf 2.2.4.4. Anderzijds is van gecompileerde talen geweten dat ze sneller zijn dan hun geïnterpreteerde tegenhangers (zie paragraaf 2.2.5.5), wat het grote snelheidsverschil tussen F# en Wolfram kan verklaren.
De volgende test omvat het vergelijken van de implementaties in Java en C# met hun resp. functionele uitwerkingen aan de hand van Streams en LINQ. Figuur 149 geeft het resultaat hiervan weer voor één iteratie.



Figuur 149: Vergelijking performantie recursieve Driehoek van Pascal tussen OO en functionele implementaties in Java en C# voor één iteratie.

De latency bij de eerste aanroep voor Java met Streams ligt toch op een behoorlijk hoge 0,4 seconden. Dit komt allicht doordat in de Java Streams versie de List klasse ook moet worden ingeladen. Bij C# komt dit patroon echter niet terug. Beide implementaties in C# hebben een zeer lage latency bij de eerste call. Dit komt overeen met de vaststellingen uit Figuur 147, wat aangeeft dat .NET/Link toch aanzienlijk sneller is dan J/Link bij het inladen van klasses. Waarschijnlijk heeft dit te maken met het feit dan .NET/Link expliciet verplicht de in te laden assemblies op voorhand op te geven, waardoor er minder overhead ontstaat bij de uitvoering van het programma.

Hetzelfde experiment is ook uitgevoerd voor 10 iteraties. Zo ontstaat Figuur 150.

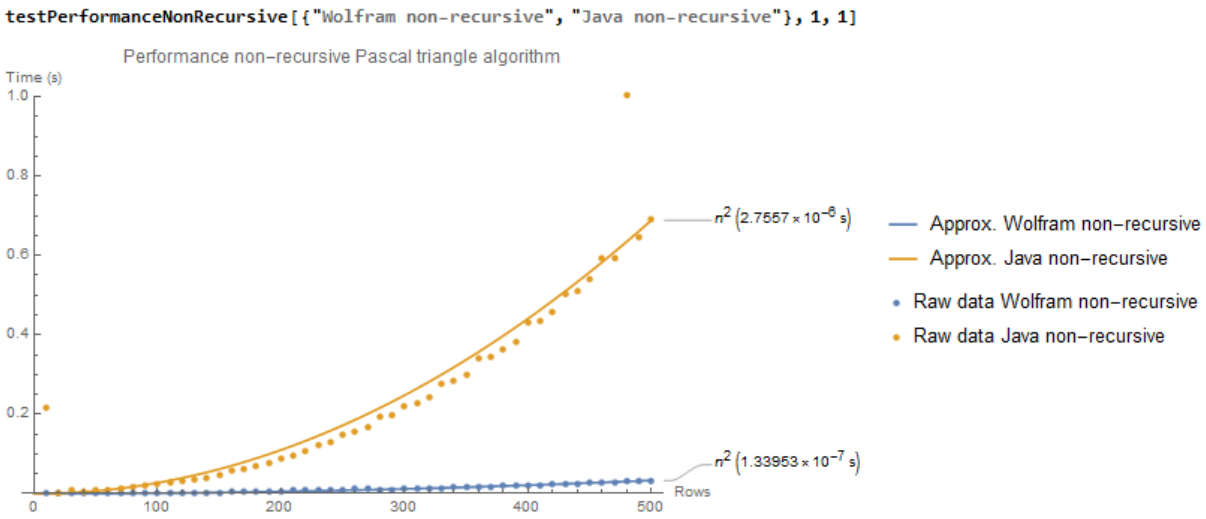


Figuur 150: Vergelijking performantie recursieve Driehoek van Pascal tussen OO en functionele implementaties in Java en C# voor 10 iteraties.

Opnieuw is duidelijk dat de latency enkel geldt voor de eerste cyclus volgens dezelfde redenering als voorheen. Bovendien blijkt dat de functionele implementaties in Java en C# nagenoeg identiek zijn qua performantie aan de zuivere OO-varianten. Er is een minimale vertraging merkbaar voor de functionele varianten ten opzichte van de object-georiënteerde, wat opnieuw de verwachtingen uit paragraaf 2.2.4.4 bevestigt. Deze vertraging is echter verwaarloosbaar en is voor nagenoeg alle toepassingsgebieden zeker geen argument tegen Streams of LINQ.

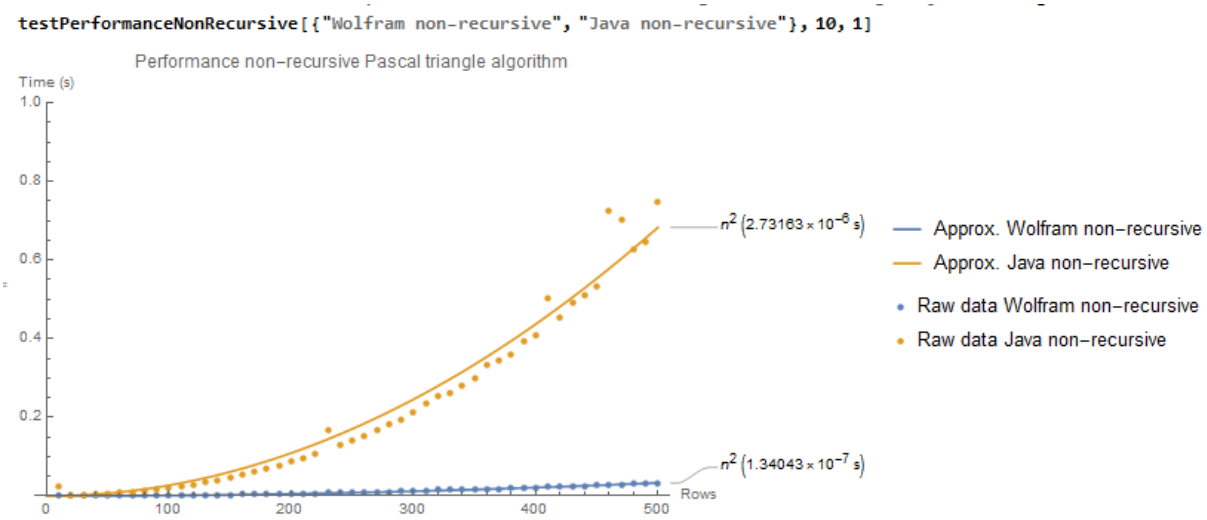
8.2 Niet-recursieve Driehoek van Pascal

De resultaten van de performantie voor het recursief algoritme laten zien dat Wolfram een grote achterstand heeft op de andere talen voor rekenintensieve applicaties. Om een volledig beeld te krijgen van de relatieve performantie van Wolfram ten opzichte van andere talen, zet deze sectie Wolfram tegenover Java voor het genereren van een Driehoek van Pascal op de niet-recursieve manier, zoals reeds eerder vermeld.
Figuur 151 geeft het resultaat weer van het vergelijken van de Java- en Wolfram-implementatie voor de niet-recursieve Driehoek van Pascal, en dit voor één iteratie.



Figuur 151: Performantie niet-recursieve Driehoek van Pascal-algoritme in Wolfram en Java voor 1 iteratie met aangepast tijdsvenster.

Uit deze figuur is opnieuw de reeds besproken latency bij de eerste aanroep van Java op te maken. De geschatte tijdscomplexiteit blijkt ook zeer goed overeen te komen met de testresultaten, in het bijzonder voor grote waardes van n. Dit is te verklaren door het feit dat de berekening van de tijdscomplexiteit enkel rekening houdt met het gedrag in de limiet voor n gaande naar oneindig. Hierdoor kunnen voor kleine waarden voor n soms afwijkingen ontstaan tussen de testresultaten en de theoretische verwachtingen voor de tijdscomplexiteit.
Bovendien valt ook op dat er een duidelijke uitschieter aanwezig is bij de resultaten voor Java. Dit kan te maken hebben met het feit dat er grote geheugenblokken op korte tijd moeten worden toegewezen en vrijgemaakt bij de uitvoering van deze test. Deze uitschieter heeft uiteraard een grote impact op de trendlijn doorheen de testdata en is ook gedeeltelijk een verklaring voor de licht afwijkende resultaten ten opzichte van de theoretische verwachtingen voor kleine n. Met een experiment over 10 iteraties is te zien in hoeverre deze uitschieters standhouden. Figuur 152 geeft hier het resultaat van weer.



Figuur 152: Performantie niet-recursieve Driehoek van Pascal-algoritme in Wolfram en Java voor 10 iteraties.

Figuur 152 bevestigt dat er bij grote waarden voor n best wat uitschieters aanwezig zijn in de testdata. Bovendien zijn deze uitschieters 'gedempt' in vergelijking met de uitschieter in de testdata bij Figuur 151. Dit betekent dat de uitschieters (pseudo-) willekeurig zijn, en dus geen systematische afwijking van de testdata ten opzichte van de verwachte resultaten aangeeft. Bovendien bevestigen deze resultaten het vermoeden dat op sommige momenten grote hoeveelheden geheugen moeten worden toegewezen en vrijgemaakt voor de berekeningen voor grote n.
Verder valt uit Figuur 152 af te leiden dat Java voor deze test veel trager is dan Wolfram. De exacte hoeveelheid vertraging is:

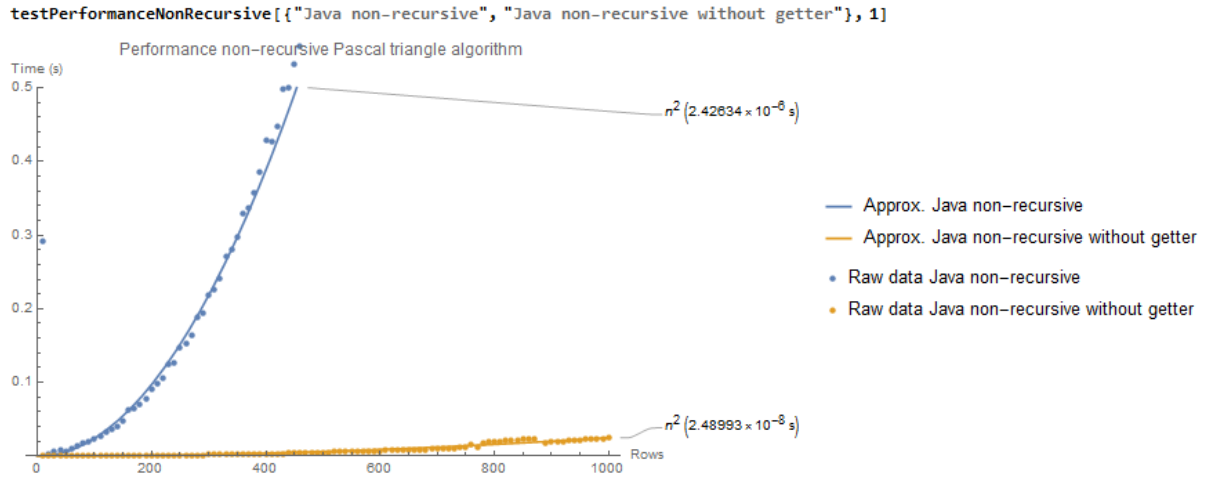
$$\frac{2{,}73163 \cdot 10^{-6}}{1{,}34043 \cdot 10^{-7}} \approx 20$$

Ineens is de relatieve performantie compleet anders. In paragraaf 8.1 was de performantie van Java immers ongeveer 1000 keer beter dan die van Wolfram. Door een ander algoritme te gebruiken, is Wolfram ineens 20 keer sneller dan Java. Java is dus plots relatief gezien ongeveer 20 000 keer trager geworden. Dit is bijzonder vreemd en vergt dus extra onderzoek.
Aangezien het hoogst onwaarschijnlijk is dat Java plots 20 000 keer trager wordt door het veranderen van algoritme, ligt de oorzaak van dit bizar resultaat waarschijnlijk niet bij Java zelf. Een andere mogelijkheid is dat het probleem zich stelt bij de communicatie tussen Wolfram en Java. Bij elke aanroep naar de JRE wordt immers aan de hand van de getPoints[] methode de gehele gegenereerde Driehoek van Pascal opgevraagd uit Java en geïmporteerd in Wolfram. Bij de niet-recursieve Driehoek van Pascal zou dit heel wat overhead kunnen creëren, aangezien de gegenereerde driehoek voor n in de grootte-orde van 100 tot 1000 ettelijke MegaBytes groot kan zijn.
Om deze hypothese te bevestigen wordt er een nieuwe dataset toegevoegd, namelijk een data set die de driehoek volgens het niet-recursief algoritme in Java genereert, maar de gegenereerde 2D-Array daarna niet importeert. Figuur 153 geeft de code hiervoor weer.

```
dataJavaNonRecursiveNoGetter := Table[{i, t1 = Now;
        JavaNew["triangles.PascalTriangle", i];
        Now - t1}, {i, 10, 1000, 10}];
```

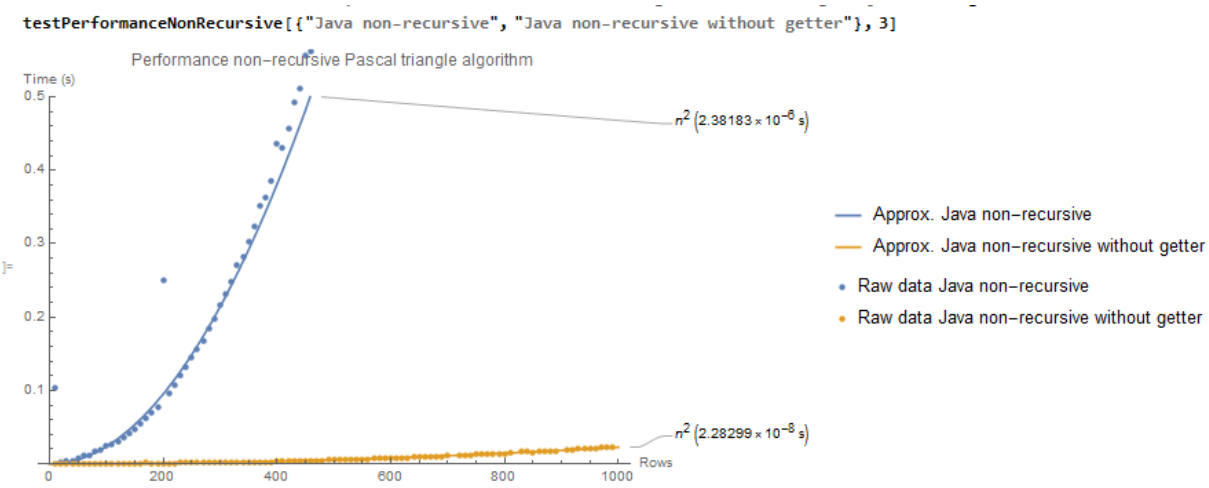
Figuur 153: Testdataset voor de niet-recursieve Driehoek van Pascal in Java die de gegenereerde driehoek niet doorgeeft aan Wolfram.

Zoals hierboven te zien is de maximale waarde voor i ook gestegen van 500 naar 1000.
Deze nieuwe versie van de niet-recursieve dataset in Java is in Figuur 154 naast de oude versie gezet.



Figuur 154: Vergelijking tussen Java niet-recursief met en zonder getter voor 1 iteratie.

Hieruit valt te zien dat er een zeer groot verschil bestaat in performantie tussen de versies van de niet-recursieve dataset in Java. Dit bevestigt de hypothese dat er zeer veel overhead komt kijken bij het communiceren van grote hoeveelheden data tussen de JRE en de Wolfram Kernel.
Om bovenstaande resultaten te bevestigen werd weer het gemiddelde genomen over meerdere iteraties. Figuur 155 geeft dit weer.



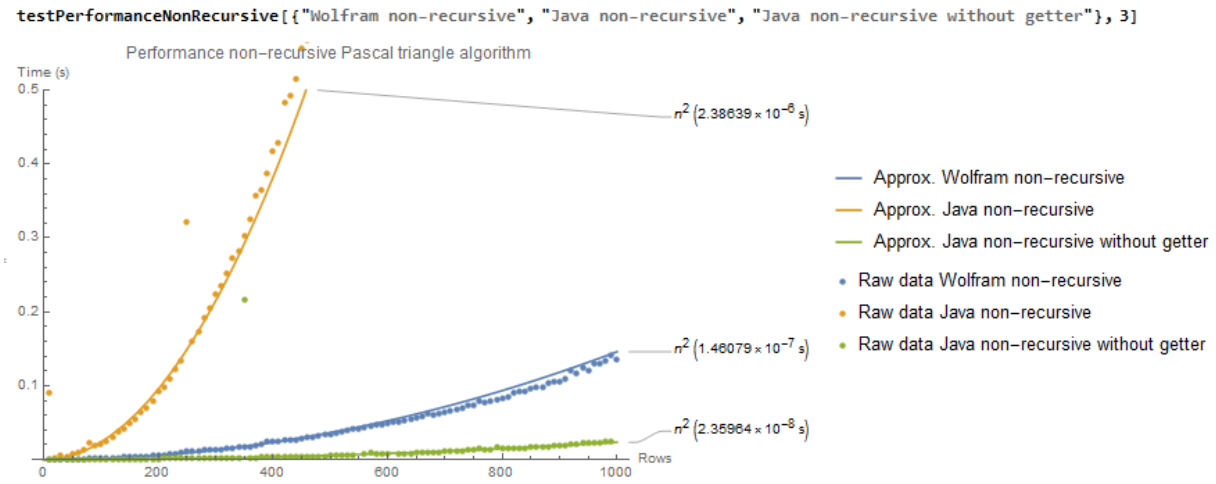
Figuur 155: Vergelijking tussen Java niet-recursief met en zonder getter voor 3 iteraties.

Merk op dat het aantal iteraties beperkt is tot drie. Dit wegens een tekort aan beschikbaar geheugen op de test-pc. Uit deze drie iteraties zijn echter al voldoende conclusies te trekken. Allereerst valt op dat er opnieuw uitschieters aanwezig zijn, waarschijnlijk om dezelfde reden als voorheen.
Vervolgens is te zien dat de trendlijn voor de data met getter voor grote n afwijkt van de testdata. Dit wordt waarschijnlijk gedeeltelijk veroorzaakt door eerder vermelde uitschieters. Een andere oorzaak kan liggen in het feit dat blijkbaar het overgrote merendeel van de tijd nodig voor dit algoritme gebruikt wordt voor de communicatie tussen Wolfram en Java. De tijdscomplexiteit van deze communicatie hoeft niet per se kwadratisch te zijn in functie van n. Hierdoor kan de testdata afwijken van de verwachtingen. Daarnaast is er uiteraard ook nog de al eerder gestelde verklaring dat het gedrag in de limiet voor n gaande naar oneindig

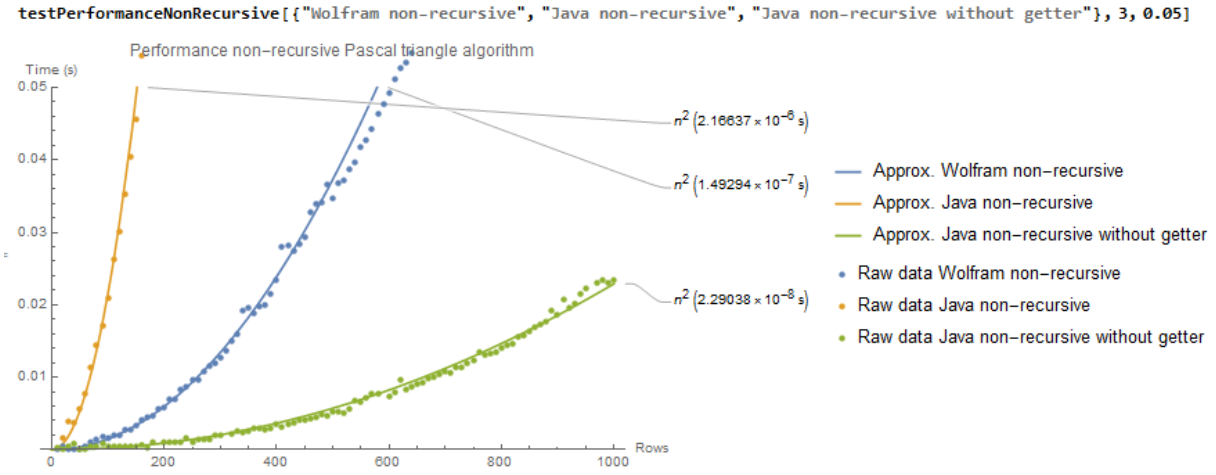
niet per se overeenkomt met het gedrag voor kleinere waarden voor n. Dit wordt echter hoogstwaarschijnlijk nog beïnvloed door andere factoren, aangezien voor geen enkele andere test met dezelfde dataset de afwijking van de trendlijn zo groot is. De meest voor de hand liggende hiervan is dat het aantal testpunten voor grote n klein is in verhouding met het aantal voor kleine n. Hierdoor wegen de resultaten voor kleine n zwaar door in de berekening van de trendlijn. Indien meer testdata verzameld zou worden, zou de trendlijn waarschijnlijk voor de kleine n meer afwijken, en voor grote n beter passen. Praktisch is het verzamelen van zo veel data echter moeilijk, aangezien het beschikbaar geheugen beperkt is en ook de rekentijd snel zeer hoog zou oplopen.
De belangrijkste conclusie is echter dat de performantie van de dataset zonder getter verschillende ordegroottes beter is dan die van de dataset met getter. Dit verklaart het eerder opgedoken probleem van het enorme verschil in relatieve performantie tussen Java en Wolfram voor het recursieve en niet-recursieve algoritme. Het verschil in performantie tussen de twee varianten is:

$$\frac{2{,}38183 \cdot 10^{-6}}{2{,}28299 \cdot 10^{-8}} \approx 104$$

De versie zonder getter is dus ongeveer 100 keer sneller dan de versie met getter. Dit betekent dat 99% van de tijd gebruikt door de versie met getter gaat naar de communicatie tussen Wolfram en Java. Op een andere manier bekeken zorgt één enkele schijnbaar triviale toevoeging van een getter ervoor dat het algoritme ineens 100 keer trager wordt.
Hieruit zou geconcludeerd kunnen worden dat het beter is de getter ook te verwijderen uit de testdata van de recursieve varianten van de Driehoek van Pascal. Dit is echter niet zo. In de eerste plaats omdat deze getter noodzakelijk is voor de implementatie in F#. Als de F# dataset wordt uitgevoerd zonder getter, wordt er helemaal niets berekend. De oorzaak hiervan ligt waarschijnlijk in het feit dat .NET/Link achter de schermen F#-types op een iets andere manier voorstelt dan C#-klassen, waardoor ietwat afwijkend gedrag ontstaat. Om F# geen structurele achterstand te geven ten opzichte van de andere programmeertalen, werd de getter aan alle datasets toegevoegd. Bovendien heeft deze getter een verwaarloosbare impact voor kleine n. Merk immers op dat bij de recursieve varianten van het Driehoek van Pascal-algoritme de maximale waarde voor n slechts 30 is. Figuur 155 laat zien dat de vertraging veroorzaakt door de getter voor waardes van n kleiner dan 50 verwaarloosbaar is. De getter heeft dus in het geval van de recursieve Driehoek van Pascal nagenoeg geen effect op de resultaten.
Nu kan deze nieuwe versie van het niet-recursieve algoritme in Java naast de recursieve variant en Wolfram gezet worden, eveneens voor drie iteraties. Figuur 156 toont het resultaat hiervan voor een globaal beeld, en Figuur 157 voor een veel kleiner tijdsvenster van 0,05 seconden.



Figuur 156: Vergelijking niet-recursief Driehoek van Pascal-algoritme voor Wolfram, Java, en Java zonder getter voor 3 iteraties.



Figuur 157: Detail van vergelijking niet-recursief Driehoek van Pascal-algoritme voor Wolfram, Java, en Java zonder getter voor 3 iteraties.

In Figuur 156 is te zien dat Wolfram wat betreft performantie ergens in het midden zit tussen Java zonder getter en Java met getter. Ruw geschat zou het verschil tussen Wolfram en Java zonder getter ongeveer een factor 5 tot 10 zijn. Figuur 157 illustreert goed hoe veel trager de implementatie in Java met getter juist is. In deze detailweergave is ook goed zichtbaar dat de trendlijn voor Java zonder getter exact juist is. In Figuur 156 is wel een enkele uitschieter te zien voor deze dataset.
Wat de performantie betreft kan de Java-versie zonder getter nu naast de Wolfram implementatie gezet worden:

$$\frac{1{,}49294 \cdot 10^{-7}}{2{,}29038 \cdot 10^{-8}} \approx 6{,}5$$

Ruwweg is hieruit te besluiten dat Wolfram 'slechts' een 5-tal keer trager is voor deze implementatie dan Java. Java verliest dus sterk aan voordeel wat betreft uitvoeringssnelheid ten opzichte van Wolfram in vergelijking met de recursieve implementatie. Met behulp van Tabel 4 is dit verlies concreet gelijk is aan:

$$\frac{1070}{6{,}5} \approx 165$$

Hieruit volgt dat Java relatief ten opzichte van Wolfram ongeveer 150 keer trager is voor de niet-recursieve implementatie dan voor de recursieve. De verklaring hiervoor zit in het typesysteem van Wolfram. Wolfram gebruikt achter de schermen immers steeds hetzelfde type voor gehele getallen. In Java is er echter een sterke opdeling van de types die gehele getallen weergeven in Integers, Longs, … Door de opdeling kunnen bewerkingen met deze types sterk worden geoptimaliseerd. Voor de recursieve versie van het Driehoek van Pascal-algoritme kon zonder problemen in Java het zeer efficiënte Integer type worden gebruikt, aangezien de waardes van de coëfficiënten in de driehoek beperkt bleven doordat het algoritme op zich zo traag is. In de niet-recursieve versie van het Driehoek van Pascal-algoritme echter zijn de waardes van de coëfficiënten veel groter, waardoor het enige Java type dat groot genoeg is om met dergelijke waarden overweg te kunnen het BigInteger type is, zoals reeds toegelicht in paragraaf 7.2.1. Van dit type is gekend dat het veel trager is dan de andere types voor het weergeven van getallen in Java. Wolfram kan daarentegen voor beide implementaties gewoon van hetzelfde type gebruikmaken. Dit verklaart het grote relatieve performantieverschil voor de twee structureel gezien anders identieke implementaties. Wel blijft Wolfram nog steeds aanzienlijk trager dan Java. De verklaring hiervoor ligt allicht op zijn minst gedeeltelijk in de functionele en geïnterpreteerde natuur van Wolfram. Deze resultaten illustreren dat de relatieve performantie van verschillende programmeertalen zeer applicatie-specifiek kan zijn.

8.3 Besluit

De resultaten uit dit hoofdstuk zijn opmerkelijk. Er zijn een heel aantal interessante vaststellingen gedaan. Ten eerste blijkt Wolfram voor CPU-intensieve toepassingen beduidend trager te zijn dan alle andere opties. Dit illustreert dat de grote flexibiliteit en programmeergemak die Wolfram biedt een prijs hebben naar performantie toe.
Verder is in dit hoofdstuk opgevallen dat bij de eerste aanroep van een methode in Java of C# binnen Wolfram een zekere latency ontstaat, die te wijten is aan het feit dat de nodige klassen eerst dienen ingeladen te worden. Bij C# en F# is deze latency verwaarloosbaar. Bij Java daarentegen is deze zodanig hoog dat er rekening mee gehouden dient te worden bij het gebruik van J/Link in Wolfram.
Verder is gebleken dat zoals uit de literatuurstudie reeds te verwachten was, F# trager is dan C#. Het verschil is echter niet significant voor de meeste toepassingen. Voor de functionele implementaties in Java en C# tegenover de object-georiënteerde gelden exact dezelfde vaststellingen en conclusies.
Uit de tests met de niet-recursieve variant van het Driehoek van Pascal-algoritme blijkt enerzijds dat er heel wat overhead komt kijken bij de communicatie tussen Wolfram en Java via J/Link. Dezelfde resultaten zijn te verwachten voor .NET/Link. Voor geheugenintensieve applicaties is het belangrijk om hier rekening mee te houden.

Verder is uit de niet-recursieve variant gebleken dat de relatieve performantie van verschillende talen sterk afhankelijk is van de exacte testapplicatie. Door het toevoegen van een schijnbaar onbelangrijk stukje code kunnen sterk verschillende resultaten naar boven komen. Dit benadrukt het belang van een kwalitatieve beoordeling van de performantie in plaats van een kwantitatieve.
Wolfram blijkt door zijn flexibiliteit sterk in te moeten boeten aan optimalisatie. Andere talen bieden in veel gevallen meer mogelijkheden om het uitvoeringsproces te optimaliseren. Voor geheugenintensieve applicaties blijkt Wolfram relatief gezien veel beter te scoren dan bij CPU-intensieve toepassingen, onder meer omdat in de concrete implementatie uit dit onderzoek de andere talen ook gedwongen zijn om met meer algemene, minder efficiënte datatypes te werken. Ook voor dit soort toepassingen loopt de performantie van Wolfram echter nog sterk achter op de andere talen uit dit onderzoek, hoewel de achterstand acceptabeler is. Ook dit geeft aan dat performantie een zeer applicatie-specifieke parameter is, hoewel de conclusies uit dit hoofdstuk zeker wel een algemene trend aangeven.

9 Parallellisatie

Naast de single-threaded performantie is een ander belangrijk aspect van een programmeertaal de mate waarin deze parallellisatie ondersteunt. Parallellisatie betekent in deze context de mogelijkheid om een rekenintensieve taak te spreiden over meerdere threads, waardoor er meerdere CPU-kernen tegelijk kunnen werken aan het programma. Voor deze masterproef is parallellisatie een zeer belangrijk aspect, aangezien zowel functioneel programmeren als cloud computing volgens de literatuur uitblinken op dit gebied. Dit hoofdstuk bestudeert dus in welke mate de geteste programmeertalen parallelliseerbaar zijn. Dit gebeurt in eerste instantie op dezelfde lokale test-pc als bij de performantietests, opnieuw aangezien op deze manier de hardware constant is voor alle geteste implementaties. In hoofdstuk 0 komt parallellisatie in de cloud aan bod. Alle tests in dit hoofdstuk maken enkel gebruik van de recursieve implementatie van het Driehoek van Pascal-algoritme in elke bestudeerde taal.

9.1 Vormen

De Driehoek van Pascal is op verschillende manieren te parallelliseren. Deze paragraaf gaat in op de verschillende mogelijkheden en de theoretische verwachtingen hiervan.

9.1.1 Rijen

Een eerste aanpak om de recursieve Driehoek van Pascal te parallelliseren is door het parallelliseren van de verschillende rijen. Elke rij krijgt op deze manier zijn eigen thread, en kan dan door een andere CPU-kern worden uitgewerkt dan de andere rijen.
Deze aanpak ligt het meest voor de hand en is ook het eenvoudigst te implementeren.
Het is belangrijk om te weten welke performantiewinst er te verwachten valt door het toepassen van deze parallellisatiemethode. Deze performantiewinst is vrij eenvoudig intuïtief en analytisch te bepalen aan de hand van volgende redenering:
De tijdscomplexiteit van het recursief Driehoek van Pascal-algoritme is zoals in paragraaf 6.1.2.2 bepaald $O(2^n)$, waarbij n het aantal rijen van de driehoek voorstelt. Aangezien elke rij een aparte thread krijgt bij deze eerste parallellisatiemethode, en het verschil in tijdsduur voor het berekenen van de coëfficiënten tussen de verschillende rijen zo groot is, is de totale tijd die nodig is voor het oplossen van het algoritme vermoedelijk ongeveer gelijk aan de tijd nodig voor het uitrekenen van de langste rij. Dit is uiteraard de laatste. Uit de tijdscomplexiteit volgt dat de tijd nodig voor de laatste rij ongeveer gelijk is aan de helft van de totale tijd die nodig is voor het algoritme. Onderstaande afleiding bevestigt dit:

$$t = \sum_{i=1}^{n} t_i$$

Met t de totale tijd nodig voor het algoritme en t_i gelijk aan de tijd die nodig is voor het berekenen van een enkele rij. Deze laatste is makkelijk te bepalen uit de tijdscomplexiteit van het volledige algoritme. Uit paragraaf 6.1.2.2 is immers geweten dat de tijdscomplexiteit van het geheel niet meer is dan de som van de tijdscomplexiteiten van de afzonderlijke rijen. In dezelfde paragraaf werd reeds bepaald dat de tijd nodig voor het berekenen van een bepaalde rij gelijk is aan:

$$t_r = C \cdot 2^r$$

Met r gaande van 1 tot n. Dit ingevuld in de originele formule geeft:

$$t = C \cdot \left(\sum_{r=1}^{n} 2^r \right)$$

Deze vergelijking is opnieuw eenvoudig uit te werken met behulp van de somformule voor rekenkundige reeksen:

$$t = C \cdot 2 \cdot \frac{1 - 2^n}{1 - 2}$$

$$t = C \cdot 2 \cdot (2^n - 1)$$

Met $2 \cdot C$ een constante die verwaarloosbaar is voor grote waardes van n ten opzichte van de eerste term. Bij benadering geldt dus dat:

$$t \approx 2 \cdot C \cdot 2^n$$

Met $C \cdot 2^n$ de tijdscomplexiteit van de laatste rij. De waarde van r is immers gelijk aan de waarde van n voor de laatste rij. Hieruit volgt dat de totale tijd nodig voor het uitrekenen van het recursief Driehoek van Pascal-algoritme bij benadering gelijk is aan 2 maal de tijd nodig voor het berekenen van de laatste rij. Het parallelliseren van de rijen zou dus een verdubbeling van de performantie moeten creëren, indien er minstens twee CPU-kernen ter beschikking zijn. Ook dit laatste volgt uit bovenstaande afleiding. Aangezien de helft van de tijd nodig is voor één enkele thread, betekent dit dat zelfs indien alle andere threads door één enkele core sequentieel worden uitgevoerd, deze ene andere core daar slechts even veel tijd voor nodig heeft als de andere core met het uitrekenen van enkel de laatste rij; althans in theorie. Een andere interessante vaststelling op basis van deze afleiding is dat nooit meer dan een verdubbeling van de snelheid te halen ten opzichte van gewoon sequentieel uitvoeren van het algoritme, ongeacht het aantal beschikbare CPU-kernen.

9.1.2 Kolommen

Een andere methode om de recursieve Driehoek van Pascal te parallelliseren bestaat eruit om alle rijen sequentieel te doorlopen, maar voor elke rij de kolommen te parallelliseren. Deze methode omzeilt het probleem dat de laatste rij de uitvoering van heel het algoritme ophoudt voor processoren met meer dan twee kernen.
Figuur 158 geeft een Driehoek van Pascal van 20 rijen weer. Hieruit is eenvoudig te zien waarom de versie met geparallelliseerde kolommen veel grotere performantiewinsten toelaat dan de versie met enkel geparallelliseerde rijen.

```
                                                  1
                                                1 1
                                              1 2 1
                                            1 3 3 1
                                          1 4 6 4 1
                                        1 5 10 10 5 1
                                      1 6 15 20 15 6 1
                                    1 7 21 35 35 21 7 1
                                  1 8 28 56 70 56 28 8 1
                                1 9 36 84 126 126 84 36 9 1
                             1 10 45 120 210 252 210 120 45 10 1
                           1 11 55 165 330 462 462 330 165 55 11 1
                        1 12 66 220 495 792 924 792 495 220 66 12 1
                     1 13 78 286 715 1287 1716 1716 1287 715 286 78 13 1
                   1 14 91 364 1001 2002 3003 3432 3003 2002 1001 364 91 14 1
                1 15 105 455 1365 3003 5005 6435 6435 5005 3003 1365 455 105 15 1
             1 16 120 560 1820 4368 8008 11440 12870 11440 8008 4368 1820 560 120 16 1
          1 17 136 680 2380 6188 12376 19448 24310 24310 19448 12376 6188 2380 680 136 17 1
       1 18 153 816 3060 8568 18564 31824 43758 48620 43758 31824 18564 8568 3060 816 153 18 1
    1 19 171 969 3876 11628 27132 50388 75582 92378 92378 75582 50388 27132 11628 3876 969 171 19 1
```

Figuur 158: Driehoek van Pascal van 20 rijen.

Deze figuur illustreert dat het verschil in rekenkracht tussen de verschillende kolommen op eenzelfde rij veel kleiner is dan het verschil in rekenkracht tussen verschillende rijen. Door de structuur van het algoritme dat in dit onderzoek is toegepast voor het genereren van Driehoek van Pascal is het immers zo dat elke coëfficiënt uit de driehoek tot stand komt door een optelling van een aantal éénen, zoals reeds beschreven in paragraaf 6.1.2.2. De waarde van elk element is dus rechtstreeks een weergave van het aantal optellingen dat is moeten gebeuren om dit element te bepalen. Door de waarde van de elementen op eenzelfde rij met elkaar te vergelijken kan dus een betrouwbaar oordeel worden geveld over de relatieve tijd die nodig is voor het berekenen van elk van die elementen.
De verhouding tussen de middelste elementen van de onderste rij uit Figuur 158 en de elementen er direct naast is gelijk aan:

$$\frac{92378}{75582} \approx 1{,}2$$

Dit betekent dat de middelste elementen slechts ongeveer 20% meer rekentijd vragen dan de elementen erlangs. Dit relatief verschil neemt wel toe naar de randen toe, maar het is duidelijk dat het verschil in rekentijd tussen de meest rekenintensieve coëfficiënten

aanzienlijk kleiner is dan bij de vorige parallellisatiemethode. Hierdoor kunnen bij deze parallellisatiemethode veel meer CPU-kernen ten volle benut worden, aangezien geen enkele individuele coëfficiënt de hele berekening ophoudt. Deze methode van parallellisatie kan dus alle 4 de kernen op de test-pc ten volle benutten.

9.1.3 Rijen en kolommen

Door bovenstaande twee parallellisatiemethodes te combineren is het in principe mogelijk nog een grotere performantiewinst te bekomen dan voorheen. Voor de tests op de lokale pc maakt dit echter geen verschil, aangezien vorige paragraaf reeds aantoonde dat enkel parallellisatie van de kolommen voldoende is om de vier processorkernen van de test-pc volledig te satureren voor het grootste deel van de executietijd.
Toch is het parallelliseren van zowel rijen als kolommen een goed idee voor deze masterproef, aangezien één van de eigenschappen van Cloud computing is dat de hardware schaalbaar is, naar gelang de vraag van de gebruiker, zoals beschreven in paragraaf 2.1.6.2. In principe zou het dus mogelijk moeten zijn dat in de cloud voor korte tijd een groot aantal CPU-kernen wordt toegewezen aan het testprogramma. Als dit zo is zou er wel een merkbare performantiewinst moeten zijn voor deze methode ten opzichte van enkel het parallelliseren van de kolommen. In het ideale geval zou deze methode twee keer zo snel moeten zijn als de methode voor enkel het parallelliseren van de kolommen, aangezien nu ook de rijen geparallelliseerd zijn, waardoor elke coëfficiënt in de hele driehoek parallel door zijn eigen thread wordt berekend. Daarom kan de maximale performantiewinst voor rijen, waarvan eerder in dit hoofdstuk bepaald is dat dit een factor twee is, gesuperponeerd worden op de performantiewinst die het parallelliseren van enkel de kolommen oplevert.

9.2 Implementatie

Uiteraard dient voor elke parallellisatiemethode besproken in dit hoofdstuk de code van het recursieve Driehoek van Pascal-algoritme in de verschillende bestudeerde talen aangepast te worden. Deze paragraaf gaat dieper in op deze aanpassingen, en toont voor elke taal de implementatie van alle drie de hierboven parallellisatiemethoden met enige toelichting.

9.2.1 Java

Om te beginnen geven Figuur 159, Figuur 160, en Figuur 161 de code weer die nodig is voor het parallelliseren van resp. rijen, kolommen, en rijen en kolommen. Voor elke parallellisatiemethode is een aparte klasse aangemaakt. Voor de overzichtelijkheid geven deze figuren enkel de constructor van elk van deze klassen weer. Dit geldt trouwens voor alle implementaties in Java en C# besproken in deze sectie.

```
public PascalTriangleRecParallelRows(int size){
    points = new int[size][];
    final List<Thread> threads = new ArrayList();

    for (int i =0;i<size;i++){
        final int iBuffer = i;
        Thread rowThread = new Thread(new Runnable() {
            @Override
            public void run() {
                final int[] row  = new int[iBuffer+1];
                for (int j = 0;j<=iBuffer;j++) {
                    row[j]=getValueAtPoint(iBuffer,j);
                }
                points[iBuffer]=row;
            }
        });
        rowThread.start();
        threads.add(rowThread);
    }
    for(int i = 0; i<threads.size(); i++) {
        try {
            threads.get(i).join();
        } catch (InterruptedException ex) {
            System.err.println("Something went wrong");
        }
    }
}
```

Figuur 159: Implementatie van de recursieve Driehoek van Pascal geparallelliseerd voor rijen in Java.

```
public PascalTriangleRecParallelCols(int size){
    points = new int[size][];
    final List<Thread> threads = new ArrayList();
    for (int i =0;i<size;i++){
        final int iBuffer = i;
        final int[] row  = new int[i+1];
        for (int j = 0;j<=i;j++){
            final int jBuffer = j;
            Thread colThread = new Thread(new Runnable() {
                @Override
                public void run() {
                    row[jBuffer]=getValueAtPoint(iBuffer,jBuffer);
                }
            });
            colThread.start();
            threads.add(colThread);
        }
        points[i]=row;
    }
    for(int i = 0; i<threads.size(); i++) {
        try {
            threads.get(i).join();
        } catch (InterruptedException ex) {
            System.err.println("Something went wrong");
        }
    }
}
```

Figuur 160: Implementatie van de recursieve Driehoek van Pascal geparallelliseerd voor kolommen in Java.

```
public PascalTriangleRecParallelRowsCols(int size){
    points = new int[size][];
    final List<Thread> rowThreads = new ArrayList();

    for (int i =0;i<size;i++){
        final int iBuffer = i;
        Thread rowThread = new Thread(new Runnable() {
            @Override
            public void run() {
                final int[] row  = new int[iBuffer+1];
                List<Thread> colThreads = new ArrayList();
                for (int j = 0;j<=iBuffer;j++){
                    final int jBuffer = j;
                    Thread colThread = new Thread(new Runnable() {
                        @Override
                        public void run() {
                            row[jBuffer]=getValueAtPoint(iBuffer,jBuffer);
                        }
                    });
                    colThreads.add(colThread);
                    colThread.start();
                }
                for (int i = 0; i< colThreads.size();i++){
                    try {
                        colThreads.get(i).join();
                    } catch (InterruptedException ex) {}
                }
                points[iBuffer]=row;
            }
        });
        rowThreads.add(rowThread);
        rowThread.start();
    }
    for (int i = 0; i<rowThreads.size();i++){
        try {
            rowThreads.get(i).join();
        } catch (InterruptedException ex) {}
    }
}
```

Figuur 161: Implementatie van de recursieve Driehoek van Pascal geparallelliseerd voor rijen en kolommen in Java.

Het valt onmiddellijk op dat deze constructors veel langer zijn dan de niet-geparallelliseerde versie. In de verschillende loops is een nieuw Runnable object toegevoegd. Deze notatie is moeilijk leesbaar. Bovendien is de implementatie vrij complex en is het makkelijk om fouten te maken. Er is ook veel boilerplate code bijgekomen. Verder moet er nog een extra for-lus worden toegevoegd op het einde om ervoor te zorgen dat de main Thread wacht totdat alle Threads van de driehoek klaar zijn, zodat de constructor pas afloopt als alle coëfficiënten effectief berekend zijn. In de versie voor geparallelliseerde rijen en kolommen dient deze lus bovendien nog eens herhaald te worden binnen de lus die de colThreads aanmaakt, zodat elke rowThread wacht tot alle colThreads klaar zijn, waarna de main thread nog eens wacht tot alle rowThreads klaar zijn. Het enige positieve is wel dat de code consistent is: voor alle drie de implementaties geldt dezelfde strategie. Dit alles leidt echter tot behoorlijk lange, ingewikkelde, en moeilijk leesbare code; die bovendien erg vatbaar is voor fouten.

9.2.2 Java met Streams

Naast de standaard OO-implementatie, is de implementatie in Java met Streams ook geparallelliseerd. Dit is op alle drie de mogelijke manieren gedaan. Figuur 162, Figuur 163, en Figuur 164 geven de implementatie voor resp. geparallelliseerde rijen, geparallelliseerde kolommen, en geparallelliseerde rijen én kolommen weer:

```
public FunctionalPascalTriangleParallelRows(int size){
    points = IntStream.range(0, size).boxed()
            .parallel()
            .map(row->IntStream.range(0, row+1).boxed()
                    .map(col->getValueAtPoint(row, col))
                    .collect(toList()))
            .collect(toList());
}
```

Figuur 162: Implementatie van de recursieve Driehoek van Pascal geparallelliseerd voor rijen in Java met Streams.

```
public FunctionalPascalTriangleParallelCols(int size){
    points = IntStream.range(0, size).boxed()
            .map(row->IntStream.range(0, row+1).boxed()
                    .parallel()
                    .map(col->getValueAtPoint(row, col))
                    .collect(toList()))
            .collect(toList());
}
```

Figuur 163: Implementatie van de recursieve Driehoek van Pascal geparallelliseerd voor kolommen in Java met Streams.

```
public FunctionalPascalTriangleParallelRowsCols(int size){
    points = IntStream.range(0, size).boxed()
            .parallel()
            .map(row->IntStream.range(0, row+1).boxed()
                    .parallel()
                    .map(col->getValueAtPoint(row, col))
                    .collect(toList()))
            .collect(toList());
}
```

Figuur 164: Implementatie van de recursieve Driehoek van Pascal geparallelliseerd voor rijen en kolommen in Java met Streams.

Deze implementaties zijn duidelijk veel korter dan de zuiver OO-implementaties. Door het gebruik van Streams kan een Stream geparallelliseerd worden door eenvoudigweg de *parallel* functie op te roepen die in Java Streams ingebouwd zit. Dit maakt de code veel compacter en leesbaarder. Bovendien is de code ook weer mooi consistent, en veel minder vatbaar voor fouten, aangezien de *parallel*-functie het aanmaken van de threads en wachten tot ze allemaal klaar zijn allemaal zelf regelt. Hierin komt de declaratieve natuur van Streams in Java sterk naar boven, wat in dit geval een groot voordeel is voor alle parallellisatiemethoden.

9.2.3 C#

Analoog aan de OO-implementatie in Java is er ook een OO-implementatie voor alle eerder in dit hoofdstuk beschreven parallellisatiemethoden gemaakt in C#. Deze lijken zeer sterk op de Java-implementaties. Figuur 165, Figuur 166, en Figuur 167 tonen de code voor elk van de drie implementaties, in dezelfde volgorde als voorheen:

```
public PascalTriangleRecParallelRows(int size)
{
    Points = new int[size][];
    List<Thread> threads = new List<Thread>();

    for (int i = 0; i < size; i++)
    {
        int iBuffer = i;
        Thread rowThread = new Thread(delegate () {
            int[] row = new int[iBuffer + 1];
            for (int j = 0; j <= iBuffer; j++)
            {
                row[j] = GetValueAtPoint(iBuffer, j);
            }
            Points[iBuffer] = row;
        });
        rowThread.Start();
        threads.Add(rowThread);
    }
    for (int i = 0; i < threads.Count; i++) threads[i].Join();
}
```

Figuur 165: Implementatie van de recursieve Driehoek van Pascal geparallelliseerd voor rijen in C#.

```
public PascalTriangleRecParallelCols(int size)
{
    Points = new int[size][];
    List<Thread> threads = new List<Thread>();

    for (int i = 0; i < size; i++)
    {
        int iBuffer = i;
        int[] row = new int[iBuffer + 1];
        for (int j = 0; j <= iBuffer; j++)
        {
            int jBuffer = j;
            Thread colThread = new Thread(delegate () {
                row[jBuffer] = GetValueAtPoint(iBuffer, jBuffer);
            });
            colThread.Start();
            threads.Add(colThread);
        }
        Points[iBuffer] = row;
    }
    for (int i = 0; i < threads.Count; i++) threads[i].Join();
}
```

Figuur 166: Implementatie van de recursieve Driehoek van Pascal geparallelliseerd voor kolommen in C#.

```
public PascalTriangleRecParallelRowsCols(int size)
{
    Points = new int[size][];
    List< Thread > rowThreads = new List<Thread>();

    for (int i = 0; i < size; i++)
    {
        int iBuffer = i;
        Thread rowThread = new Thread(delegate() {
            int[] row = new int[iBuffer + 1];
            List<Thread> colThreads = new List<Thread>();
            for (int j = 0; j <= iBuffer; j++)
            {
                int jBuffer = j;
                Thread colThread = new Thread(delegate() {
                    row[jBuffer] = GetValueAtPoint(iBuffer, jBuffer);
                });
                colThreads.Add(colThread);
                colThread.Start();
            }
            foreach (Thread t in colThreads) t.Join();
            Points[iBuffer]=row;
        });
        rowThreads.Add(rowThread);
        rowThread.Start();
    }
    foreach (Thread t in rowThreads) t.Join();
}
```

Figuur 167: Implementatie van de recursieve Driehoek van Pascal geparallelliseerd voor rijen en kolommen in C#.

Bovenstaande code lijkt zeer sterk op de OO-implementatie in Java. De syntax is toch al iets compacter omdat C# flexibeler omspringt met Exception Handling etc. Juist dezelfde opmerkingen blijven wel van kracht als bij de OO-implementatie in Java.

9.2.4 C# met LINQ

Ook in C# is een functionele implementatie gemaakt van de geparallelliseerde recursieve Driehoek van Pascal. In Figuur 168, Figuur 169, en Figuur 170 opnieuw de code, op dezelfde manier als in de vorige paragrafen:

```
public FunctionalPascalTriangleParallelRows(int size)
{
    Points = Enumerable.Range(0, size)
        .AsParallel().AsOrdered()
        .Select(row => Enumerable.Range(0, row + 1)
            .Select(col => GetValueAtPoint(row, col))
            .ToList())
        .ToList();
}
```

Figuur 168: Implementatie van de recursieve Driehoek van Pascal geparallelliseerd voor rijen in C# met LINQ.

```
public FunctionalPascalTriangleParallelCols(int size)
{
    Points = Enumerable.Range(0, size)
        .Select(row => Enumerable.Range(0, row + 1)
            .AsParallel().AsOrdered()
            .Select(col => GetValueAtPoint(row, col))
            .ToList())
        .ToList();
}
```

Figuur 169: Implementatie van de recursieve Driehoek van Pascal geparallelliseerd voor kolommen in C# met LINQ.

```
public FunctionalPascalTriangleParallelRowsCols(int size)
{
    Points = Enumerable.Range(0, size)
        .AsParallel().AsOrdered()
        .Select(row => Enumerable.Range(0, row + 1)
            .AsParallel().AsOrdered()
            .Select(col => GetValueAtPoint(row, col))
            .ToList())
        .ToList();
}
```

Figuur 170: Implementatie van de recursieve Driehoek van Pascal geparallelliseerd voor rijen en kolommen in C# met LINQ.

Deze code lijkt zeer sterk op de Java-versies met Streams. Het enige verschil is dat in C# met LINQ expliciet moet worden meegegeven dat uitvoering van de threads op een geordende manier dient te gebeuren. Hier gelden dan ook dezelfde opmerkingen als voor de Java-versie met Streams.

9.2.5 F#

Verder zijn alle parallellisatiemethoden ook in F# geïntegreerd. In Figuur 171, Figuur 172, en Figuur 173 de code hiervoor.

```
let parallelPascalTriangleRows size =
    Async.Parallel [ for r in 1..size->
        async{return [1..r] |>
        List.map (fun c->pascalTriangleValue r c)}] |>
    Async.RunSynchronously
```

Figuur 171: Implementatie van de recursieve Driehoek van Pascal geparallelliseerd voor rijen in F#.

```
let parallelPascalTriangleCols size =
    [1..size] |>
    List.map(fun r-> Async.Parallel[for c in 1..r ->
        async {return pascalTriangleValue r c }] |>
        Async.RunSynchronously)
```

Figuur 172: Implementatie van de recursieve Driehoek van Pascal geparallelliseerd voor kolommen in F#.

```
let parallelPascalTriangleRowsCols size =
    Async.Parallel [ for r in 1..size->
        async{return Async.Parallel[for c in 1..r ->
            async{return pascalTriangleValue r c}] |>
        Async.RunSynchronously}] |>
    Async.RunSynchronously
```

Figuur 173: Implementatie van de recursieve Driehoek van Pascal geparallelliseerd voor rijen en kolommen in F#.

De F#-implementatie is compacter dan die in alle andere hierboven besproken talen. Bovendien is de F#-implementatie nog steeds consistent. Dezelfde syntax is gebruikt voor alle drie de parallellisatiemethoden. De leesbaarheid is ook goed, hoewel de implementaties met Streams en LINQ op dit gebied nog iets beter scoren, aangezien de parallellisatiesyntax uit twee delen bestaat in F#, namelijk het async.Parallel-gedeelte voor het aanmaken van de Threads, en het Async.Runsynchronously-gedeelte, waar de output van async.Parallel naar gepiped is. Voor beginnende programmeurs kan dit verwarrend overkomen; in het bijzonder de versie met parallelle rijen en kolommen waar deze syntax tweemaal in enkele regels code voorkomt.

9.2.6 Wolfram

In Wolfram werden eveneens alle 3 de parallellisatiemethodes toegepast. De resulterende code is hieronder weergegeven in Figuur 174.

```
pascalTriangleRecParallelRows[size_Integer] /; size == 1 := {{1}}
pascalTriangleRecParallelRows[size_Integer] :=
Parallelize[Table[pascalTriangleValue[i, j], {i, 1, size}, {j, 1, i}]]

pascalTriangleRecParallelCols[size_Integer] /; size == 1 := {{1}}
pascalTriangleRecParallelCols[size_Integer] :=
Table[Parallelize[Table[pascalTriangleValue[i, j], {j, 1, i}]], {i, 1, size}]

pascalTriangleRecParallelRowsCols[size_Integer] /; size == 1 := {{1}}
pascalTriangleRecParallelRowsCols[size_Integer] := Parallelize[
Table[pascalTriangleValue[i, j], {i, 1, size}, {j, 1, i}], Method → "FinestGrained"]
```

Figuur 174: Implementatie van de verschillende geparallelliseerde versies van de recursieve Driehoek van Pascal in Wolfram.

Wolfram heeft opnieuw de elegantste implementatie. Er zijn slechts enkele regels nodig om de volledige geparallelliseerde Driehoek van Pascal te definiëren in alle gevallen. Bovendien is de code zeer goed leesbaar. Een nadeel is wel dat de schrijfwijze bij Wolfram niet consistent is. Hoewel steeds de Parallellize functie gebruikt is, is voor alle drie de parallellisatiemethoden is er een andere benaderingswijze nodig, en als de syntax voor één methode gekend is, kan van daaruit niet zonder enig zoekwerk de juiste syntax voor de andere methodes bepaald worden. Voor programmeurs die weinig ervaring hebben met Wolfram is dit een probleem. Bij het programmeren is men hier dus sterk afhankelijk van de Wolfram documentatie. Anders is het simpelweg niet mogelijk om te weten welke syntax nu juist van toepassing is. Gelukkig is Wolfram wel zeer goed gedocumenteerd. Een laatste opmerking hierbij is dat de Parallelize functie op een zeer hoog niveau werkt, waardoor de programmeur zelf heel weinig controle heeft over de exacte manier waarop de parallelisatie geïmplementeerd is. Voor grote en complexe toepassingen zou dit een probleem kunnen vormen. Parallellizes mogen bijvoorbeeld niet genest worden in Wolfram. Alle andere implementaties geven ten minste deze vrijheid en laten de programmeur ook op zijn minst toe exact te bepalen waar de parallellisatie start.

9.3 Integratie

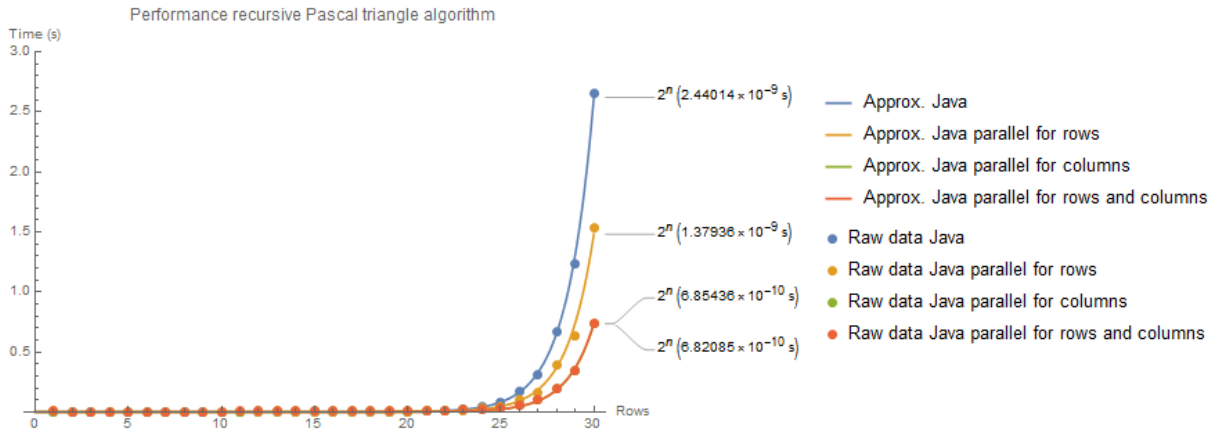
De integratie van de verschillende geparalleliseerde algoritmes is op dezelfde manier gebeurd als voor de gewone performantietesten, beschreven in paragraaf 6.2.1. Zie Bijlage C voor een volledig overzicht van alle code die gebruikt is voor het genereren van alle testdata voor heel deze masterproef.
Uiteraard is de data-functie ook aangevuld op gepaste wijze. Bijlage D geeft een overzicht van de volledige data-functie die namen koppelt aan de testdata.
Ten slotte zijn om de geparallelliseerde F#-implementaties te kunnen integreren in Wolfram uiteraard ook een aantal extra types aangemaakt, op dezelfde manier als beschreven in paragraaf 6.2.1.3.

9.4 Resultaten

Deze paragraaf bespreekt de resultaten van het uitvoeren van de tests beschreven in paragraaf 6.2 voor al bovenstaande parallellisatiemethoden. Eerst en vooral is voor elke taal voor 10 iteraties elke parallellisatiemethode vergeleken met de niet-geparallelliseerde variant in deze taal. Daarna maakt deze paragraaf een globale vergelijking tussen de resultaten van alle uitgevoerde tests over de verschillende talen heen.

9.4.1 Java

Om te beginnen geeft Figuur 175 de testresultaten voor de OO-implementatie in Java, weergegeven voor een globaal tijdsvenster.



Figuur 175: Vergelijking parallellisatiemethoden voor Java voor 10 iteraties.

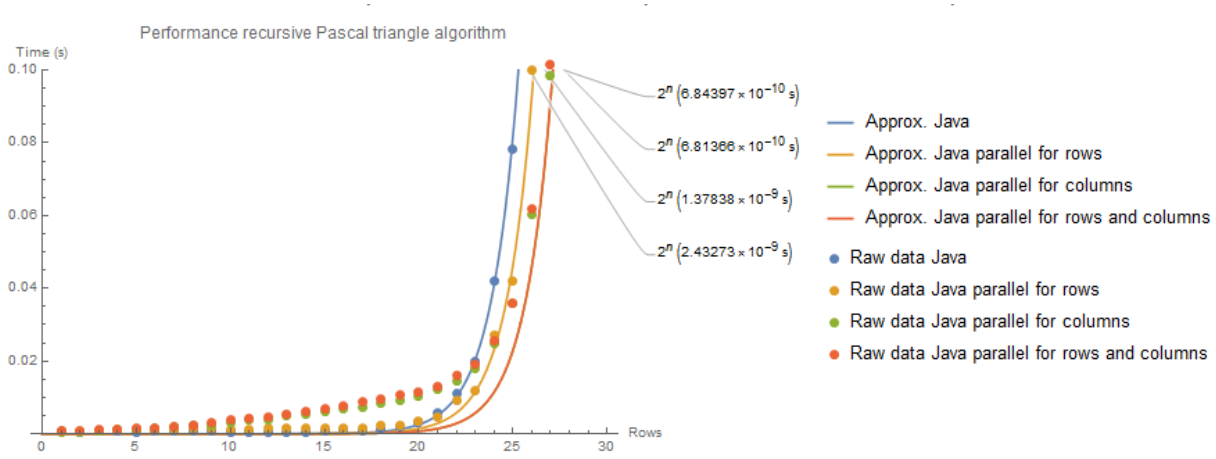
Een aantal zaken springen in het oog. Allereerst valt op dat alle parallellisatiemethodes een grote performantiewinst leveren. Dit is uiteraard de bedoeling van parallellisatie. Ook valt op dat voor geparallelliseerde kolommen en geparallelliseerde rijen en kolommen de resultaten nagenoeg samenvallen. Dit was eveneens te verwachten, aangezien zoals beschreven in paragraaf 9.1 alle vier de beschikbare kernen van de test-pc volledig in gebruik zijn door enkel de kolommen te parallelliseren. Voor lokale tests heeft het parallelliseren van zowel rijen als kolommen zoals verwacht dus geen zin.

Door de coëfficiënten van de benaderingscurves van elk van de parallellisatiemethoden uit Figuur 175 naast elkaar te zetten ontstaat Tabel 5. Deze tabel verschaft een beter inzicht in de relatieve performanties van de verschillende parallellisatiemethodes.

Tabel 5: Vergelijking coëfficiënten van de verschillende parallellisatiemethoden voor Java.

Data	**Parallelle rijen**	**Parallelle kolommen**	**Parallelle rijen en kolommen**
Relatieve nodige tijd	$\frac{1{,}4 \cdot 10^{-9}}{2{,}4 \cdot 10^{-9}} \approx 0{,}58$	$\frac{6{,}8 \cdot 10^{-10}}{2{,}4 \cdot 10^{-9}} \approx 0{,}28$	$\frac{6{,}9 \cdot 10^{-10}}{2{,}4 \cdot 10^{-9}} \approx 0{,}29$
Relatieve performantiewinst	$\left(\frac{1}{0{,}58} - 1\right) \cdot 100 = 72\%$	$\left(\frac{1}{0{,}28} - 1\right) \cdot 100 = 257\%$	$\left(\frac{1}{0{,}29} - 1\right) \cdot 100 = 245\%$

Uit Tabel 5 zijn interessante vaststellingen te doen. Allereerst is de performantiewinst van de geparallelliseerde rijen slechts ongeveer 72%, in tegenstelling tot de verwachte 100%. Dit geeft aan dat parallellisatie in deze Java-implementatie extra overhead veroorzaakt. Dit is in feite logisch. De verschillende aangemaakte threads moeten immers met elkaar gecoördineerd worden om het eindresultaat te bepalen. Dezelfde redenering verklaart waarom de performantiewinst van de andere parallellisatiemethodes slechts rond de 250% ligt in plaats van de verwachte 300% voor de quad-core processor van de test-pc.
Naast het globaal beeld uit Figuur 175 is het ook zinnig om een gedetailleerde weergave te bekijken voor kleine n, in een kleiner tijdsvenster. Deze gedetailleerde weergave is te zien in Figuur 176.



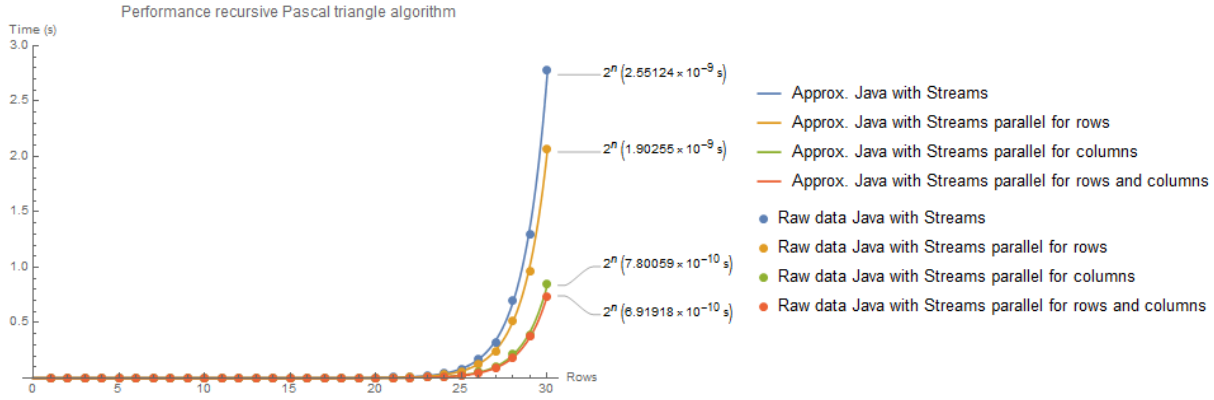
Figuur 176: Vergelijking parallellisatiemethoden voor Java voor 10 iteraties met een verkleind tijdsvenster.

Op Figuur 176 zijn een aantal interessante zaken af te lezen. Het valt op dat de niet-geparallelliseerde versie van de recursieve Driehoek van Pascal mooi de verwachte tijdscomplexiteit volgt voor kleine n. De eerste parallellisatiemethode volgt eveneens goed het verwachte patroon. De tweede en derde methode doen dit echter veel minder. Dit kan niet de fout zijn van de trendlijnen zelf, aangezien ze perfect passen voor de eerste twee methodes, en enkel voor de laatste twee sterk afwijken voor kleine n. Bovendien is voor grote waardes van n zelfs geen afwijking ten opzichte van de trendlijnen te zien voor geen

enkele van de parallellisatiemethodes. De afwijking van de trendlijn voor parallellisatiemethode twee en drie bij kleine n moet dus aan de testresultaten zelf te wijten zijn. Het is duidelijk dat er een verband bestaat tussen het aantal threads dat een bepaalde methode gebruikt, en de mate waarin die methode afwijkt van het verwachte resultaat. Dit betekent dat er een grote overhead is bij het creëren van de threads. Dit betekent dat in Java goed moet worden nagedacht bij het parallelliseren van processen, want voor minder rekenintensieve processen kan parallelliseren immers volgens bovenstaande testresultaten een sterke negatieve impact hebben op de performantie in Java, in plaats van een verbetering te bieden op dit vlak. Dit in combinatie met het feit dat parallellisatie behoorlijk moeilijk te implementeren is in Java betekent dat parallellisatie voorzichtig benaderd dient te worden in Java met een zuiver OO-implementatie, en vaak zelfs meer nadelen heeft dan voordelen.

9.4.2 Java met Streams

Naast de OO-implementatie is ook de performantie van Java met Streams getest voor alle parallellisatiemethodes. Figuur 177 geeft hiervan de resultaten weer.



Figuur 177: Vergelijking parallellisatiemethoden voor Java met Streams voor 10 iteraties.

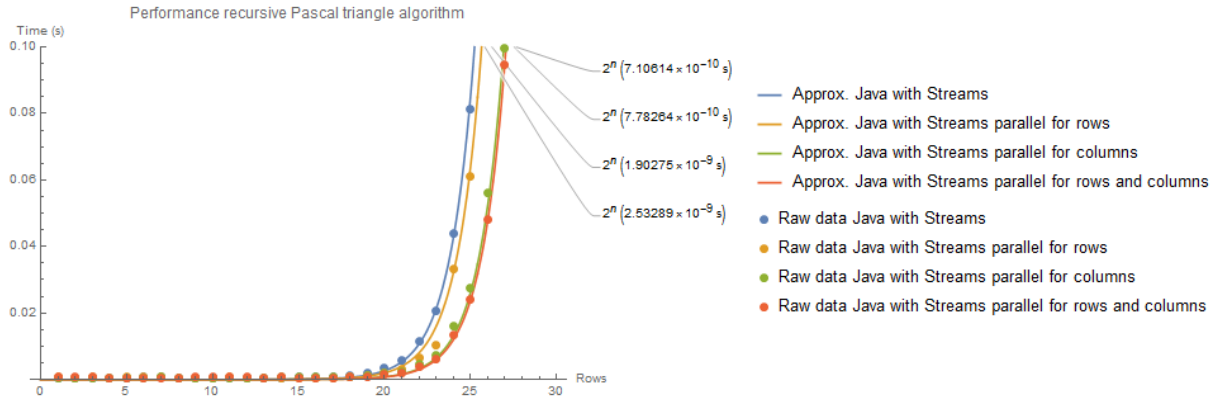
Het valt op dat de performantiewinst voor de geparallelliseerde rijen toch opvallend veel kleiner is dan verwacht. De andere parallellisatiemethodes presteren dan weer wel naar behoren.

Daarnaast is te zien dat in deze implementatie de geparallelliseerde rijen en kolommen het iets beter doen dan de geparallelliseerde kolommen. Dit is in eerste zin in tegenspraak met de conclusies uit de vorige paragraaf. Dit kan toch enigszins worden verklaard door in te zien dat de verschillende parallelle threads als een soort van puzzel moeten worden uitgevoerd door de PC. Hoe meer threads er zijn, hoe beter deze puzzel in mekaar kan passen. Hoe goed de puzzel past, is afhankelijk van de exacte grootte van de individuele stukken. Dit kan worden beïnvloed door allerlei factoren zoals de exacte hoeveelheid overhead de threads met zich meebrengen, de andere processen die draaien op de pc, etc. Dit kan verklaren waarom een methode met meer threads en overhead toch betere resultaten zou kunnen scoren dan een andere methode. Tabel 6 geeft een gedetailleerd overzicht van de relatieve performanties.

Tabel 6: Vergelijking coëfficiënten van de verschillende parallellisatiemethoden voor Java met Streams.

Data	**Parallelle rijen**	**Parallelle kolommen**	**Parallelle rijen en kolommen**
Relatieve performantie	$\frac{1{,}9 \cdot 10^{-9}}{2{,}5 \cdot 10^{-9}} \approx 0{,}76$	$\frac{7{,}8 \cdot 10^{-10}}{2{,}5 \cdot 10^{-9}} \approx 0{,}31$	$\frac{6{,}9 \cdot 10^{-10}}{2{,}5 \cdot 10^{-9}} \approx 0{,}28$
Relatieve performantiewinst	$\left(\frac{1}{0{,}76} - 1\right) \cdot 100 = 32\%$	$\left(\frac{1}{0{,}31} - 1\right) \cdot 100 = 223\%$	$\left(\frac{1}{0{,}28} - 1\right) \cdot 100 = 257\%$

Uit Tabel 6 blijkt dat de performantie voor geparallelliseerde rijen zoals eerder al vastgesteld zeer slecht is. Voor de andere 2 methodes is de performantie echter beter. De methode voor rijen en kolommen is zelfs iets beter dan die voor enkel kolommen. Deze resultaten zijn eerder tegenstrijdig en kunnen alleen maar verklaard worden door te concluderen dat de efficiëntie van parallellisatie voor Java met Streams sterk van de exacte aard van het algoritme afhankelijk is. Door minieme aanpassingen aan de code kan de relatieve performantie blijkbaar sterk veranderen.
Ook voor Java met Streams werd het experiment herhaald voor een kleiner tijdsvenster om zo de overhead bij het aanmaken van extra threads beter in te kunnen schatten.

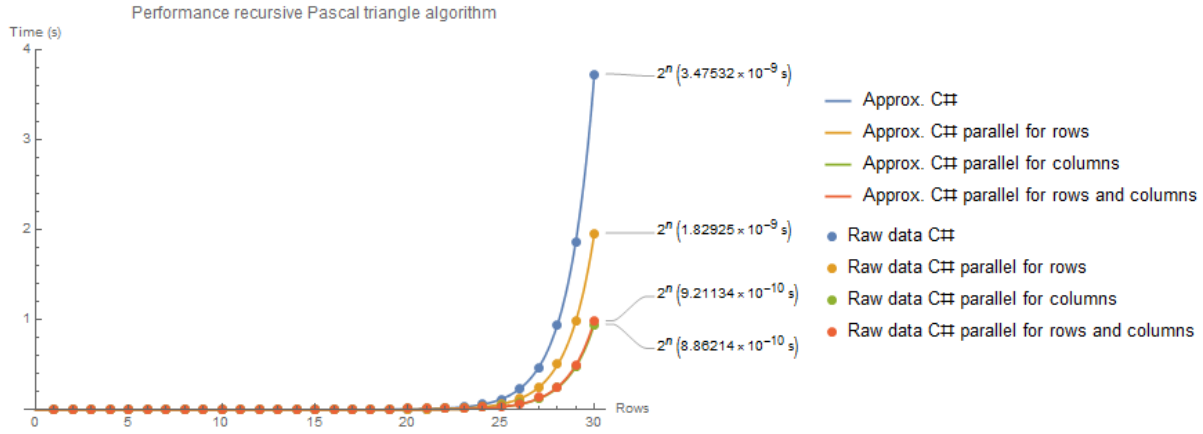


Figuur 178: Vergelijking parallellisatiemethoden voor Java met Streams voor 10 iteraties met een verkleind tijdsvenster.

Figuur 178 laat zien dat voor Java met Streams helemaal geen overhead bestaat bij het aanmaken van threads. Hierdoor is parallellisatie voor niet-rekenintensieve taken geen risico. Hiermee wordt een belangrijke drempel voor het toepassen van parallellisatie weggenomen. Dit komt de algemene performantie van geparallelliseerde toepassingen zeker ten goede. Aangezien de implementatie van parallellisatie in Java met Streams ook zeer eenvoudig is, is er geen enkele reden om geen parallellisatie toe te passen waar mogelijk, in het bijzonder voor rekenintensieve toepassingen waar dit een significante verbetering van de uitvoeringssnelheid kan betekenen.

9.4.3 C#

De volgende geteste taal is C#, te beginnen met de OO-implementatie. De testresultaten hiervoor zijn weergegeven in Figuur 179.



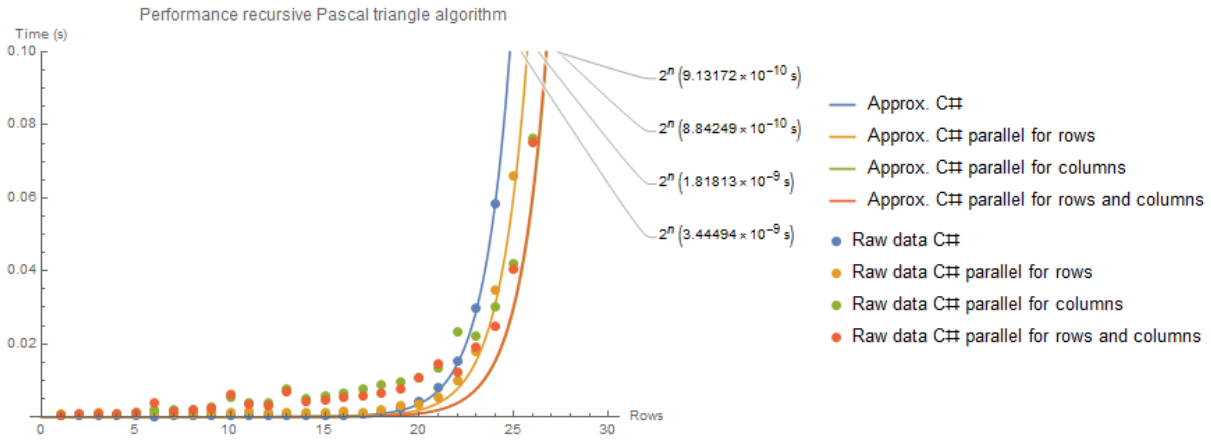
Figuur 179: Vergelijking parallellisatiemethoden voor C# voor 10 iteraties.

C# blijkt op het eerste zicht zeer gelijkaardig aan Java wat de performantie van de verschillende parallellisatiemethodes betreft. De vaststellingen voor Java zijn dus hier ook geldig. Tabel 7 geeft hierover uitsluitsel.

Tabel 7: Vergelijking coëfficiënten van de verschillende parallellisatiemethoden voor C#.

Data	**Parallelle rijen**	**Parallelle kolommen**	**Parallelle rijen en kolommen**
Relatieve performantie	$\frac{1{,}8 \cdot 10^{-9}}{3{,}5 \cdot 10^{-9}} \approx 0{,}51$	$\frac{8{,}9 \cdot 10^{-10}}{3{,}5 \cdot 10^{-9}} \approx 0{,}25$	$\frac{9{,}2 \cdot 10^{-10}}{3{,}5 \cdot 10^{-9}} \approx 0{,}26$
Relatieve performantiewinst	$\left(\frac{1}{0{,}51} - 1\right) \cdot 100 = 96\%$	$\left(\frac{1}{0{,}25} - 1\right) \cdot 100 = 300\%$	$\left(\frac{1}{0{,}26} - 1\right) \cdot 100 = 285\%$

Tabel 7 laat zien dat de performantie van C# voor parallellisatie zeer goed is. Er is bijna geen overhead in geen van de gevallen.
Naast het algemeen overzicht is ook voor C# een detailweergave voor kleine n gemaakt. Figuur 180 geeft het resultaat hiervan weer.

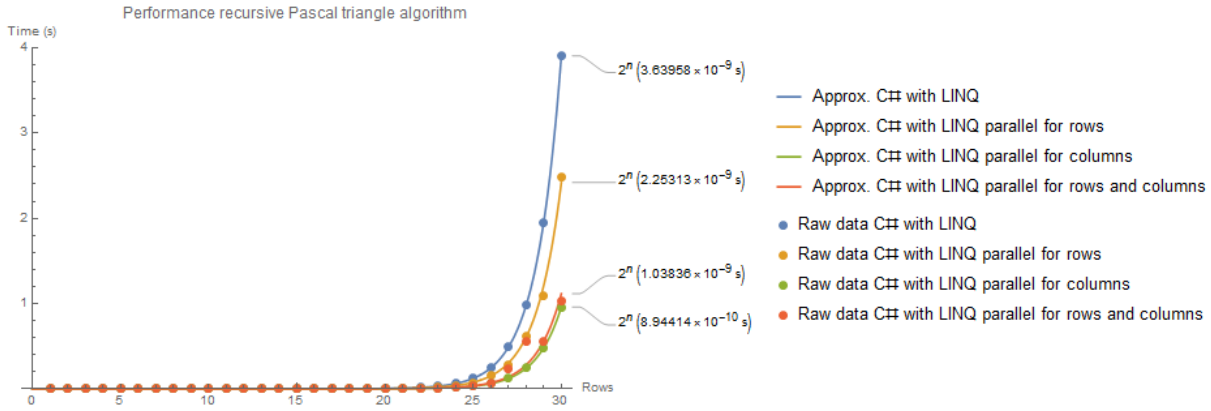


Figuur 180: Vergelijking parallellisatiemethoden voor C# voor 10 iteraties met een verkleind tijdsvenster.

Ook deze resultaten zijn zeer gelijkaardig aan die in Java. Alle conclusies voor Java zijn hier dan ook toepasbaar. Wel valt op dat de parallellisatiemethode voor rijen en kolommen hier net iets sneller is dan die voor enkel kolommen. Op het eerste zicht is dit vreemd aangezien ook hier weer veel meer threads en dus overhead zouden moeten zijn voor de methode met parallelle rijen en kolommen. Dit is echter opnieuw te verklaren zoals in de vorige paragraaf; aangezien meer threads beter op elkaar zouden kunnen aansluiten, en aldus meer efficiënt van de beschikbare CPU-kernen gebruik kunnen maken.

9.4.4 C# met LINQ

De variant van de C#-implementatie met LINQ is uiteraard ook bestudeerd. Figuur 181 geeft de testresultaten voor deze versie weer voor een globaal tijdsvenster.



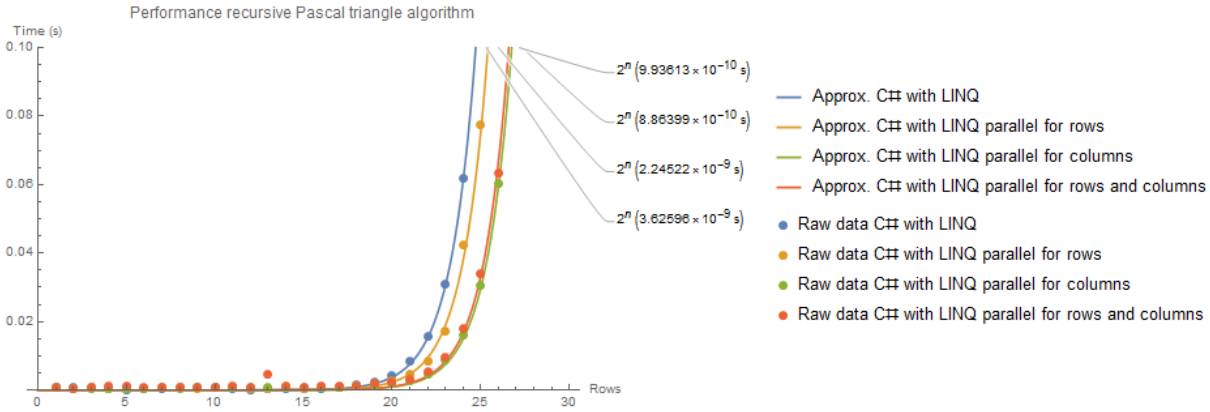
Figuur 181: Vergelijking parallellisatiemethoden voor C# met LINQ voor 10 iteraties.

Voor deze implementatie zijn opnieuw te verwachten resultaten zichtbaar. Er is te zien dat de versie met geparallelliseerde rijen en kolommen iets trager is dan de versie met enkel de kolommen geparallelliseerd, waarschijnlijk opnieuw door de extra overhead die ontstaat bij het aanmaken en coördineren van extra threads. De performantiewinst voor enkel geparallelliseerde rijen lijkt opnieuw kleiner dan verwacht. Tabel 8 geeft ook voor deze implementatie de relatieve performantiewinst voor elk van de geteste parallellisatiemethodes.

Tabel 8 : Vergelijking coëfficiënten van de verschillende parallellisatiemethoden voor C# met LINQ.

Data	**Parallelle rijen**	**Parallelle kolommen**	**Parallelle rijen en kolommen**
Relatieve performantie	$\frac{2{,}3 \cdot 10^{-9}}{3{,}6 \cdot 10^{-9}} \approx 0{,}64$	$\frac{8{,}9 \cdot 10^{-10}}{3{,}6 \cdot 10^{-9}} \approx 0{,}25$	$\frac{10^{-9}}{3{,}6 \cdot 10^{-9}} \approx 0{,}28$
Relatieve performantiewinst	$\left(\frac{1}{0{,}64} - 1\right) \cdot 100 = 56\%$	$\left(\frac{1}{0{,}25} - 1\right) \cdot 100 = 300\%$	$\left(\frac{1}{0{,}28} - 1\right) \cdot 100 = 257\%$

De performantiewinst voor geparallelliseerde rijen is slechts rond de 55%. Dit suggereert opnieuw een grote overhead bij het parallelliseren. Dit wordt echter net als bij Java met Streams niet ondersteund door de resultaten van de twee andere parallellisatiemethodes. Een gelijkaardige verklaring aan die van bij Java met Streams is dan ook hier van toepassing. Ook voor C# met LINQ is er een detailopname gemaakt voor kleine n. Figuur 182 geeft hier de resultaten van weer.

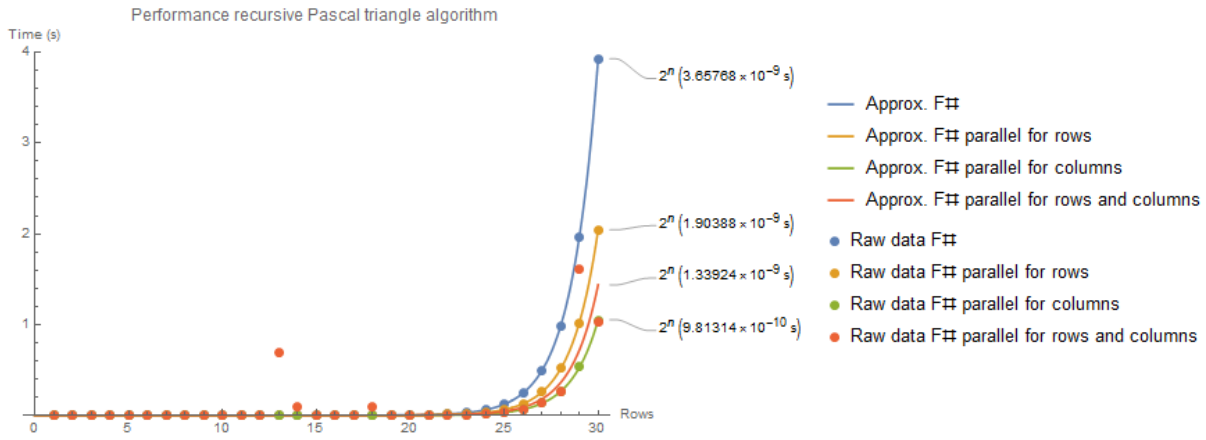


Figuur 182: Vergelijking parallellisatiemethoden voor C# met LINQ voor 10 iteraties met een verkleind tijdsvenster.

Opnieuw is de parallel met Java duidelijk: deze resultaten komen zeer sterk overeen met die van Java met Streams. Ook hier zijn dan ook dezelfde conclusies vanuit Java met Streams overdraagbaar.

9.4.5 F#

Vervolgens is F# aan de beurt. De testresultaten voor deze taal zijn weergegeven in Figuur 183.



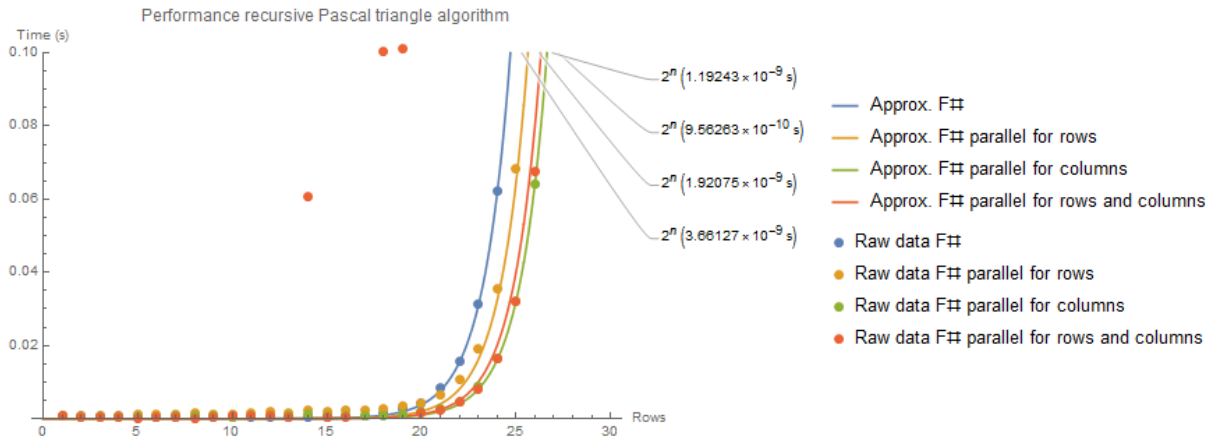
Figuur 183: Vergelijking parallellisatiemethoden voor F# voor 10 iteraties.

Een aantal zaken vallen op. Allereerst is te zien dat de performantie van de geparallelliseerde rijen-methode zeer goed is. Dit wijst erop dat in F# slechts een kleine overhead ontstaat bij het managen van meerdere threads.
Daarnaast lijken de parallellisatiemethodes voor enkel kolommen en zowel rijen als kolommen grotendeels samen te vallen. Bij de methode waarbij zowel rijen als kolommen geparallelliseerd zijn valt echter op dat er een heel aantal uitschieters zijn. Het beheren van een groot aantal threads heeft dus in F# een kost in de vorm van onvoorspelbaar gedrag. De curve-fitting werd zo goed mogelijk aangepast om de uitschieters weg te filteren, maar dit heeft toch duidelijk een impact gehad op de gegenereerde coëfficiënt. Voor F# is de coëfficiënt voor de derde parallellisatiemethode dus niet betrouwbaar. De andere coëfficiënten zijn dat daarentegen wel. Deze uitschieters kunnen zowel door F# zelf veroorzaakt zijn als door het testsysteem. Aangezien geen enkele andere implementatie hier last van lijkt te hebben, is F# waarschijnlijk de primaire oorzaak. Deze oorzaak kan liggen bij het feit dat het aantal threads bij de laatste parallellisatiemethode zodanig groot en de levensduur ervan zodanig kort is dat er problemen ontstaan bij de garbage collection, die de uitvoering van het programma vertragen. Een andere mogelijkheid is dat het probleem ontstaat doordat de derde parallellisatiemethode gebruik maakt van geneste Async.Parallel-statements. Het zou kunnen dat hierdoor het programma meer moeite heeft met het coördineren van alle aangemaakte threads. Er is verder onderzoek nodig om de exacte oorzaak van dit probleem te achterhalen. Dit valt echter buiten het bestek van deze masterproef.
Tabel 9 geeft de relatieve performantiewinst voor elk van de parallellisatiemethoden in F# weer. Aangezien op de uitschieters na de parallellisatiemethode voor zowel rijen als kolommen nagenoeg perfect lijkt samen te vallen met die voor enkel kolommen, is voor deze parallellisatiemethode dan ook de coëfficiënt van die voor enkel kolommen overgenomen.

Tabel 9: Vergelijking coëfficiënten van de verschillende parallellisatiemethoden voor F#.

Data	**Parallelle rijen**	**Parallelle kolommen**	**Parallelle rijen en kolommen**
Relatieve performantie	$\frac{1{,}9 \cdot 10^{-9}}{3{,}7 \cdot 10^{-9}} \approx 0{,}51$	$\frac{9{,}8 \cdot 10^{-10}}{3{,}7 \cdot 10^{-9}} \approx 0{,}26$	$\frac{9{,}8 \cdot 10^{-10}}{3{,}7 \cdot 10^{-9}} \approx 0{,}26$
Relatieve performantiewinst	$\left(\frac{1}{0{,}51} - 1\right) \cdot 100 = 96\%$	$\left(\frac{1}{0{,}26} - 1\right) \cdot 100 = 285\%$	$\left(\frac{1}{0{,}26} - 1\right) \cdot 100 = 285\%$

Tabel 9 toont aan dat alle parallellisatiemethoden zeer goed scoren in F#. Uiteraard moet dit resultaat voor de derde parallellisatiemethode wel genuanceerd worden vanwege de talrijke uitschieters.
Figuur 184 geeft de detailopname voor een klein tijdsvenster voor F# weer.

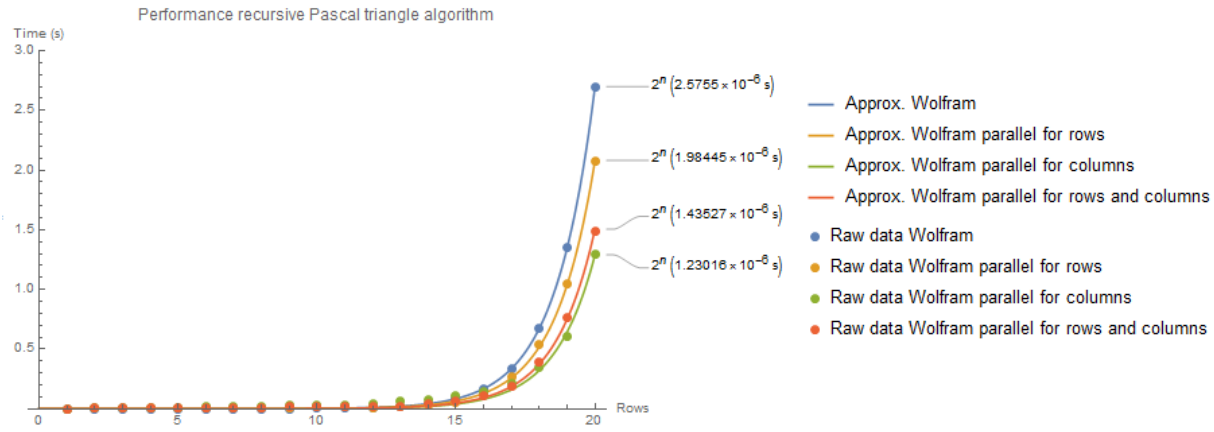


Figuur 184: Vergelijking parallellisatiemethoden voor F# voor 10 iteraties met een verkleind tijdsvenster.

Uit Figuur 184 zijn enkele interessante vaststellingen af te leiden. Allereerst is de performantie voor kleine n voor alle methoden nagenoeg identiek. Er is dus geen overhead bij het aanmaken van threads. Wel is ook hier te zien dat bij het parallellelliseren van rijen en kolommen willekeurige uitschieters ontstaan in de testdata, terwijl dit voor geen enkele andere methode in F# zo is. Parallellisatie heeft dus in F# geen overhead bij het aanmaken van threads. F# blijkt dus zeer goed parallelliseerbaar te zijn, met de opmerking dat bij een groot aantal threads die op een geneste manier aangemaakt zijn de performantie onvoorspelbaar kan zijn.

9.4.6 Wolfram

Ten slotte geeft Figuur 185 de testresultaten voor het parallelliseren van Wolfram weer.

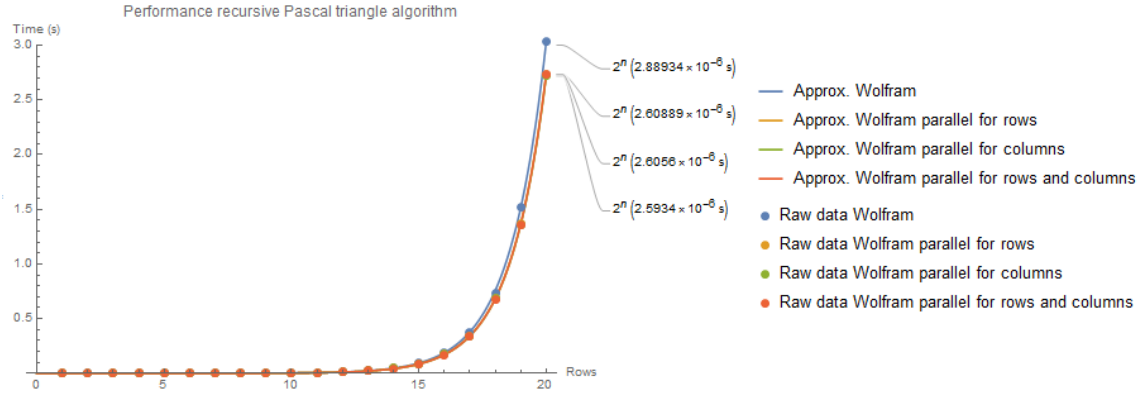


Figuur 185: Vergelijking parallellisatiemethoden voor Wolfram voor 10 iteraties.

Het valt op dat alle parallellisatiemethodes het slecht lijken te doen in Wolfram. De verklaring hiervoor ligt in de manier waarop Wolfram omgaat met parallellisatie. Voor elk parallel element start Wolfram immers een nieuwe Wolfram Kernel op. Wat in de andere talen dus een thread is, is een volledig nieuwe runtime in Wolfram. Het spreekt voor zich dat dit heel wat overhead met zich meebrengt.
Het is bovendien belangrijk om op te merken dat het aantal parallelle subkernels dat kan worden opgestart sterk afhankelijk is van de Wolfram-licentie van de gebruiker. Een vrij absurde vaststelling is dat de full-size Developer licentie van 160 euro per maand voor het Wolfram Development Platform [79] geen toegang geeft tot parallelle subkernels. Dit is enkel mogelijk met een geldige licentie voor Wolfram Mathematica, die apart aan te kopen is. Bovendien is dit feit zeer slecht gedocumenteerd. Dit is toch een ernstige tekortkoming voor een taal die zichzelf adverteert als zeer parallelliseerbaar, zoals op de officiële Wolfram-website te lezen valt [80]. Figuur 186 geeft weer wat er gebeurt als de geparallelliseerde versie van het recursieve Driehoek van Pascal-algoritme in Wolfram wordt uitgevoerd binnen het Wolfram development Platform zonder gebruik te maken van Wolfram Mathematica.

```
testPerformance[{"Wolfram", "Wolfram parallel for rows", "Wolfram parallel for columns", "Wolfram parallel for rows and columns"}, 1, 3]
```

```
SubKernels`LocalKernels`LaunchLocal: Could not provide a subkernel license.
SubKernels`LocalKernels`LaunchLocal: Could not provide a subkernel license.
SubKernels`LocalKernels`LaunchLocal: Could not provide a subkernel license.
General: Further output of SubKernels`LocalKernels`LaunchLocal::nolic2 will be suppressed during this calculation.
SubKernels`LocalKernels`LaunchLocal: 4 of 4 kernels failed to launch.
ParallelTable: No parallel kernels available; proceeding with sequential evaluation.
ParallelTable: No parallel kernels available; proceeding with sequential evaluation.
ParallelTable: No parallel kernels available; proceeding with sequential evaluation.
General: Further output of ParallelTable::nopar will be suppressed during this calculation.
```



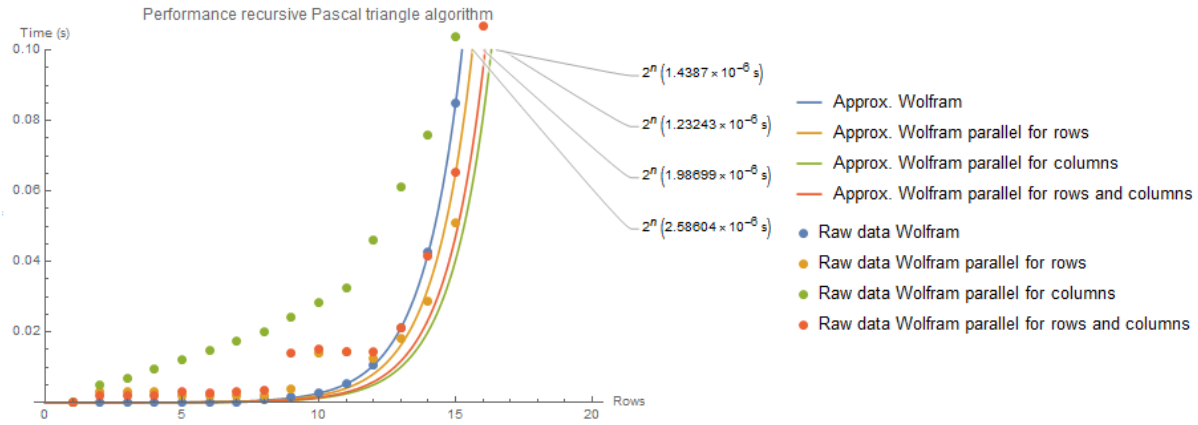
Figuur 186: Resultaat van parallellisatie van Wolfram toe te passen in het Wolfram Development Platform met een geldige Developer licentie.

Figuur 186 laat zien dat er binnen het Wolfram Development Platform een foutmelding wordt gegenereerd indien geprobeerd wordt om Wolfram te parallelliseren. Alle berekeningen worden vervolgens sequentieel afgehandeld. De performantie van alle parallellisatie methoden is dan ook gelijk aan die van de niet-geparallelliseerde versie. Tabel 10 geeft een overzicht van de performantiewinst van de verschillende parallellisatiemethodes in Wolfram, gebaseerd op de coëfficiënten uit Figuur 185. Deze figuur is bekomen door gebruik te maken van een proeflicentie voor Wolfram Mathematica.

Tabel 10: Vergelijking coëfficiënten van de verschillende parallellisatiemethoden voor Wolfram.

Data	**Parallelle rijen**	**Parallelle kolommen**	**Parallelle rijen en kolommen**
Relatieve performantie	$\frac{2 \cdot 10^{-6}}{2{,}6 \cdot 10^{-6}} \approx 0{,}77$	$\frac{1{,}2 \cdot 10^{-6}}{2{,}6 \cdot 10^{-6}} \approx 0{,}46$	$\frac{1{,}4 \cdot 10^{-9}}{2{,}6 \cdot 10^{-6}} \approx 0{,}54$
Relatieve performantiewinst	$\left(\frac{1}{0{,}77} - 1\right) \cdot 100 = 30\%$	$\left(\frac{1}{0{,}46} - 1\right) \cdot 100 = 117\%$	$\left(\frac{1}{0{,}54} - 1\right) \cdot 100 = 85\%$

In bovenstaande concrete resultaten valt op dat de performantiewinst van de geparallelliseerde versies van het recursief Driehoek van Pascal-algoritme in Wolfram veel kleiner is dan verwacht. Vast staat dat Wolfram zeker geen goed parallelliseerbare taal is, ondanks de beweringen van de ontwikkelaars ervan.
Ten slotte rest nog de performantie voor de geparallelliseerde recursieve Driehoek van Pascal in Wolfram te bestuderen voor een klein tijdsvenster. Dit is weergegeven in Figuur 187.



Figuur 187: Vergelijking parallellisatiemethoden voor Wolfram voor 10 iteraties met een verkleind tijdsvenster.

Uit Figuur 187 is af te leiden dat naast de algemene performantie ook de overhead bij kleine waardes van n voor Wolfram zeer slecht is. Opvallend is ook dat blijkbaar voor de geparallelliseerde kolommen de overhead het ergste is. De grote overhead is in het algemeen opnieuw te verklaren doordat Wolfram voor elke parallelle thread een volledig nieuwe Kernel opstart. Deze eigenaardige omgang met parallellisatie maakt dat parallellisatie in Wolfram in bijna alle gevallen te vermijden is.

9.4.7 Algemeen

Na elke taal op zich bestudeerd te hebben in bovenstaande paragrafen geeft deze paragraaf een overzicht van de belangrijkste conclusies over de verschillende talen heen. Tabel 11 geeft een weergave van de relatieve performantiewinst voor elke parallellisatiemethode in elke taal. De zeer goede en zeer slechte resultaten zijn ingekleurd om een beter overzicht te geven.

Tabel 11: Relatieve performantie verschillende parallellisatie methodes voor de verschillende talen.

Taal	**Parallelle rijen**	**Parallelle kolommen**	**Parallelle rijen en kolommen**
Java	72%	257%	245%
Java met Streams	32%	223%	257%
C#	96%	300%	285%
C# met LINQ	56%	300%	257%
F#	96%	285%	285%
Wolfram	30%	117%	85%

Naast de algemene performantie is ook de overhead voor kleine waardes van n in een tabel gegoten, namelijk Tabel 12. Deze tabel geeft een kwalitatief oordeel over deze parameter.

Tabel 12: Kwalitatieve beoordeling overhead bij kleine n.

Taal	**Parallelle rijen**	**Parallelle kolommen**	**Parallelle rijen en kolommen**
Java	*Weinig*	*Veel*	*Veel*
Java met Streams	*Geen*	*Geen*	*Geen*
C#	*Weinig*	*Veel*	*Veel*
C# met LINQ	*Geen*	*Geen*	*Geen*
F#	*Geen*	*Geen*	*Geen*
Wolfram	*Weinig*	*Zeer veel*	*Veel*

Bovenstaande tabellen maken het zien van verbanden tussen verschillende talen aanzienlijk eenvoudiger. Hieronder een opsomming van de belangrijkste vaststellingen:

- Er bestaan duidelijk parallellen tussen het soort programmeertaal, en de prestaties ervan wat parallellisatie betreft;
- De OO-implementaties in zowel Java als C# blijken goede tot zeer goede parallellisatiekarakteristieken te bezitten. De overhead voor het aanmaken van threads blijkt bij deze talen wel groot te zijn;
- De functionele implementaties in zowel Java als C# scoren qua parallellisatie performantie slechter dan hun respectievelijke OO-implementaties. Langs de andere kant is de overhead die komt kijken bij het aanmaken van threads veel kleiner voor de functionele implementaties dan de OO-varianten;
- F# scoort over het algemeen wat parallellisatie betreft het beste, met als enige nadeel dat er uitschieters ontstaan in de performantie bij geparallelliseerde rijen en kolommen. Daardoor is de meting hiervoor moeilijk te doen. In het algemeen kan gezegd worden dat F# zeer parallelliseerbaar is in vergelijking met de rest, zolang het aantal threads niet overdreven groot wordt, en de threads niet op een geneste manier aangemaakt zijn;
- Ten slotte blijkt Wolfram ook op het gebied van parallellisatie het onderspit te delven ten opzichte van de andere implementaties. Op elk gebied scoort Wolfram het slechtste resultaat van alle geteste talen in het parallellisatie-departement.

9.5 Besluit

Dit hoofdstuk toont aan dat parallellisatie in functionele talen veel makkelijker te implementeren is dan in OO-talen. Waar bij OO-implementaties een complexe, langdradige, en onoverzichtelijke structuur dient uitgeschreven te worden die bovendien zeer foutgevoelig is voor het aanmaken en beheren van de threads, volstaat het in alle functionele implementaties om een enkele functie aan te roepen of een eenvoudige en compacte structuur te creëren die de parallellisatie volledig op zich neemt. Dit levert grote voordelen op voor parallellisatie in alle functionele implementaties naar leesbaarheid, ontwikkelingstijd, lengte van de code, foutenlast, en programmeergemak toe.
De performantie van functionele talen blijkt voor parallellisatie afhankelijk van de specifieke taal en implementatie -met uitzondering van Wolfram- iets lager of vergelijkbaar met die van OO-talen. Een groot voordeel van functionele talen is wel dat de overhead bij het aanmaken van threads veel lager is. Dit in combinatie met de veel eenvoudigere implementatie van de functionele varianten maakt parallellisatie in de functionele varianten van Java en C# veel aangenamer te implementeren en stabieler vanuit performantiestandpunt. De keerzijde is dat de globale performantie iets lager ligt dan de OO-varianten, ondanks het feit dat de overhead bij het aanmaken van threads duidelijk veel kleiner is. Voor een programmeur die gebruik wilt maken van de voordelen van parallellisatie -al is het niet optimaal-, zonder zich veel zorgen te moeten maken over de nadelen ervan -zoals inconsistente performantie door grote overhead bij veel kortstondige threads- zijn de functionele varianten in Java en C# veel aantrekkelijkere opties dan de OO-implementaties. Indien performantie echter de belangrijkste bezorgdheid is, is het verstandiger om voor toepassingen waarvan alle threads gegarandeerd veel rekentijd vragen de OO-implementaties te kiezen binnen Java of C#.
De zuiver functionele F# variant scoort in het algemeen de beste resultaten voor performantie, in combinatie met een korte en stijlvolle implementatie. Dit gaat in sommige gevallen wel ten koste van stabiliteit, waardoor uitschieters in de performantie waarneembaar zijn bij het aanmaken een groot aantal threads op een geneste manier. Er is meer onderzoek nodig om te bepalen waar deze uitschieters juist aan te wijten zijn, maar het is veilig om te zeggen dat F# zeer goed parallelliseerbaar is, zolang het aantal threads niet overdreven groot is, en nested Async.Parallel-uitdrukkingen vermeden worden.
Ten slotte is vastgesteld dat Wolfram op alle gebied ondermaats presteert ten opzichte van de andere opties wat parallellisatie betreft vanuit een performantie-perspectief. Bovendien is de enige optie om aan parallellisatie te doen het aankopen van een dure licentie van een specifiek Wolfram-product. Dit alles maakt Wolfram zeer onaantrekkelijk voor alle soorten parallelle toepassingen.

10 Cloud Service Providers

Bovenstaande hoofdstukken focusten zich enkel op de bestudeerde programmeertalen. De focus van deze masterproef is echter functioneel programmeren in een cloud-context. Vanaf dit hoofdstuk behandelt deze scriptie dan ook alle cloud-gerelateerde aspecten van de bestudeerde talen.
Alvorens de eerder beschreven Driehoek van Pascal applicaties in de cloud te deployen en testresultaten te verzamelen is het van belang een grondige kennis te hebben van de verschillende cloud service providers die gekozen zijn voor dit onderzoek. Dit hoofdstuk neemt elk van de cloud service providers die in paragraaf 2.4 uitgekozen zijn voor dit onderzoek onder de loep. Voor elk van de gekozen cloud-platformen gaat dit hoofdstuk dieper in op de bevindingen uit paragraaf 2.1.9.

10.1 Oracle Cloud

Oracle Cloud is de eerste cloud service provider die voor dit onderzoek nader bestudeerd is. Deze provider lijkt in het bijzonder interessant voor de Java-implementaties uit dit onderzoek, aangezien Oracle de eigenaar is van Java. Oracle Cloud biedt immers een groot aantal cloud-based producten aan die gecentreerd zijn rond deze taal. Deze producten gaan van data-opslag (DaaS), over volledige PaaS app-hosting voor Java EE webapplicaties en zelfs Java SE-applicaties, tot IaaS en SaaS-oplossingen [81]. De aangeboden services zijn er in alle soorten en maten, met prijzen van enkele honderden tot vele duizenden euro's per maand. Oracle Cloud biedt de mogelijkheid om aan de hand van een gratis proefversie verschillende van hun Cloud Services uit te proberen. Deze proefversie is 30 dagen geldig, en een bedrag van $300 aan gratis krediet wordt ter beschikking gesteld van de gebruiker. Deze proefversie is echter enkel van toepassing op selecte delen van het enorme aanbod aan services binnen de Oracle Cloud. Het aanbod aan beschikbare trials binnen de Oracle Cloud is echter nog steeds meer dan uitgebreid genoeg om de smaak van hun technologieën te pakken te krijgen [82].
Voor deze thesis is dan ook van deze gratis proefversie gebruik gemaakt. Het proces om een gratis proefaccount aan te maken is echter omslachtig en onvoldoende gedocumenteerd om een aangename gebruikservaring te garanderen. Het opgeven van een geldig creditkaartnummer is verplicht, en na het aanmaken van de account kan het een tijd duren voordat een bevestigingsmail aankomt met een zeer lange lijst van instructies en allerlei credentials en urls die voor elke specifieke service binnen het Oracle Cloud-platform gebruikt dient te worden. Er zijn verschillende sets van credentials nodig om van verschillende delen van het cloud-systeem gebruik te maken, waarvan sommige door de gebruiker zelf aangemaakt dienen te worden, en sommige door Oracle. Dit alles maakt het van start gaan met Oracle Cloud verwarrend en omslachtig voor nieuwe gebruikers. Daarbovenop bleken de door Oracle toegewezen credentials voor de proefaccount die voor dit onderzoek werd aangemaakt niet te kloppen. Het is voor dit onderzoek onmogelijk gebleken om in te loggen op eender welke service, hoewel de instructies in de welkom-email duidelijk leken en nauwgezet gevolgd werden. Contact opnemen met de support van Oracle

bleek geen oplossing te bieden voor het probleem. De support verwees tot twee keer toe door naar een andere sectie van de support, en niemand bleek in staat dit probleem op te lossen, of zelfs meer inzicht te verschaffen in de oorzaak ervan. Uiteindelijk werd gevraagd om nog een extra account aan te maken op nog een andere support-omgeving in de vorm van een website van Oracle, en een langdradig proces te doorlopen om een ticket-claim te maken bij de technische support. Op dit punt is besloten de inspanningen om gebruik te maken van de Oracle cloud stop te zetten, aangezien dit platform al heel wat kostbare onderzoekstijd had verslonden zonder deze masterproef vooruit te helpen.
Ondanks het gebrek aan gedetailleerde testing van het Oracle Cloud Platform staat al zeker vast dat dit voor nieuwe gebruikers geen aantrekkelijk cloud-platform is. Verdere studie van dit platform is dus overbodig, aangezien er een grote hoeveelheid alternatieve cloud-platformen op de markt zijn, zoals duidelijk aangegeven in paragraaf 2.1.9.

10.2 Google Cloud Platform

Een andere in paragraaf 2.4 uitgekozen cloud service provider is het Google Cloud Platform. Dit omdat het Google Cloud Platform enorm veel mogelijkheden biedt, zeer flexibel is, en transparant is over welke diensten betalend zijn en welke niet. Er is tevens een gratis trial beschikbaar van 12 maanden, ofwel $300 in cloud-krediet, afhankelijk van welk van beide eerst opgebruikt is. Deze trial is van toepassing op alle diensten van het Google Cloud Platform, hetzij met enige beperkingen. Zo is bijvoorbeeld het maximale aantal CPU-kernen waarvan gebruik gemaakt mag worden beperkt tot 8 in totaal over alle applicaties gespreid, terwijl met een betalende subscriptie tot wel 64 kernen kunnen worden toegewezen aan elke applicatie.
Origineel was gepland om met dit platform te werken als een neutrale partij, en op die manier een vergelijking te kunnen maken tussen de echte 'in-house' cloud-platformen voor elke taal, en een externe provider. Door de complicaties met het Oracle Cloud Platform, is voor deze masterproef echter gekozen Google Cloud Platform te gebruiken als platform voor alle Java-implementaties. Dit neemt echter niet weg dat andere talen ook in dit platform geïmplementeerd kunnen worden. Daarover meer in het volgende hoofdstuk. Hieronder een toelichting over de manier waarop het Google Cloud Platform functioneert.
Google Cloud Platform bestaat uit een groot aantal diensten van Google die allemaal gericht zijn op softwareontwikkeling en -deployment in de cloud. De aangeboden diensten variëren van versiebeheer en dataopslag over SaaS-services voor onder andere machine learning, tot IaaS en PaaS oplossingen voor app hosting in een groot aantal verschillende talen. De ondersteunde talen zijn onder andere Go, Java, .NET, Node.js, PHP, Python, en Ruby [83]. Afhankelijk van de taal is de manier van werken soms wel sterk verschillend, en moet er gebruik worden gemaakt van verschillende technologieën binnen het Google Cloud Platform. Google zelf biedt in het algemeen goede documentatie voor het gebruik van Google Cloud Platform, onder andere op [83]. Hieronder enige toelichting bij de diensten binnen het Google Cloud Platform die voor deze masterproef van belang zijn.

10.2.1 Google App engine

De Google App engine is een PaaS cloud service die Google aanbiedt om eenvoudig apps te hosten in de cloud. De diensten van Google App Engine zijn in twee grote delen opgedeeld: het Standard Environment en het Flexible Environment. Het verschil tussen deze environments is groot en belangrijk [84]. Onderstaande paragrafen lichten elke van deze environments in detail toe.

10.2.1.1 Standard Environment

Het Standard environment is in de eerste plaats bedoeld om goedkoop schaalbare apps te kunnen hosten in de cloud. In het Standard Environment van de Google App Engine kunnen apps in bepaalde programmeertalen relatief eenvoudig worden gehost door een aantal stappen te volgen. Deze stappen verschillen wel van taal tot taal.
Binnen het Standard Environment is er de zogenaamde Always Free Tier [85]. Dit zijn een aantal diensten van het Standard Environment van Google App Engine die zoals de naam al zegt steeds gratis zijn. Er zijn uiteraard wel gebruikslimieten opgelegd voor deze gratis diensten. De Always Free Tier volstaat in het algemeen om op kleine schaal van bepaalde Google App Engine Standard Environment-diensten gebruik te maken.
Naast de Always Free Tier is er binnen het Standard Environment van de Google App Engine ook de mogelijkheid om betalend apps te deployen. Voor deze apps moet enkel betaald worden in verhouding tot de hoeveelheid resources die ze gebruiken. De Always Free Tier wordt steeds afgetrokken van de factuur voor betalende apps. Er moet dus enkel betaald worden voor het verbruik dat boven de quota van de Always Free tier gaat. De uiteindelijke factuur is dus meestal bescheiden.
Bij het Standard Environment van de Google App Engine komen jammer genoeg wel een aantal belangrijke beperkingen kijken met betrekking tot de ondersteunde programmeertalen. Deze zijn immers beperkt tot specifieke versies van slechts enkele mainstream programmeertalen. Meer specifiek zijn de enige ondersteunde talen in het Google App Engine Standard Environment:

- Python 2.7,
- Java 7,
- PHP 5.5,
- Go 1.6.

Zoals in bovenstaande opsomming te zien is zijn deze talen niet enkel beperkt in aantal, maar de ondersteunde versies zijn ook verouderd. Voor dit onderzoek kan dus enkel Java in het Standard Environment van het Google Cloud Platform gedeployed worden. Hierbij komt dan de belangrijke beperking dat de enige ondersteunde versie Java 7 is. Voor dit onderzoek vormt dit een probleem, aangezien Streams en andere functionele eigenschappen van Java pas sinds Java 8 ondersteund worden. Voor deze masterproef volstaat het Google App Engine Standard Environment dus slechts gedeeltelijk.

10.2.1.2 Flexible Environment

Het Flexible Environment van de Google App engine biedt voor bovenstaand compatibiliteitsprobleem een oplossing. Zoals de naam al doet vermoeden is het Flexible Environment immers veel flexibeler dan het Standard Environment. Er zijn in het Flexible Environment immers nagenoeg geen beperkingen op de programmeertalen en versies ervan die kunnen gebruikt worden. Bovendien is er meer controle over de infrastructuur waar de applicaties in gedeployed worden.
Binnen het Flexible Environment van de Google App Engine zijn er geen gratis diensten, maar de betaling is enkel in verhouding tot de gebruikte resources, wat de prijzen meestal schappelijk houdt.
De technologie achter het Flexible Environment van de Google App Engine verschilt volledig van die achter het Standard Environment. Binnen het Flexible Environment worden apps immers gehost in Docker containers voor eender welke versie van eender welke programmeertaal. Docker is een opkomende technologie van Google die toelaat applicaties te draaien binnen containers. Deze containers kunnen gezien worden als een minimalistische VM, hoewel de achterliggende technologie anders is. Bij een container gebeurt de abstractie op het OS-niveau in plaats van het hardware-niveau. De OS-kernel is dus gedeeld met het host-systeem. Hierdoor hebben containers veel minder overhead, en nemen ze veel minder resources in beslag, wat containers veel aantrekkelijker maakt dan VM's voor het deployen van applicaties in de cloud [86] en [87]. Docker geeft een groot aantal mogelijkheden en flexibiliteit aan de programmeur, maar is desondanks niet noodzakelijk moeilijk om te gebruiken indien van de voor de meeste mainstream programmeertalen door Google zelf voorziene standaardconfiguraties gebruik wordt gemaakt. De talen waarvoor een standaardconfiguratie beschikbaar is in de App Engine Flexible Environment zijn [88]:

- Go,
- Java 8,
- PHP 5/7,
- Python 2.7/3.5,
- .NET,
- Node.JS,
- Ruby.

Deze masterproef gaat niet in detail in op de werking en het gebruik van Docker, maar maakt wel gebruik van een standaardconfiguratie van Google om met behulp van Docker een Java 8 applicatie te hosten in het Flexible Environment van de Google App Engine. Hierover meer in paragraaf 11.1.

10.2.2 Google Compute Engine

Google Compute Engine is de IaaS-service die het Google Cloud Platform aanbiedt [89]. Google Compute engine biedt de mogelijkheid om VM's aan te maken en te configureren zoals de gebruiker dit zelf wenst. Er zijn allerlei configureerbare opties zoals networking, hardware, etc.
Google Compute Engine is net als het Flexible Environment van de Google App Engine een zuiver betalende service. Er wordt aangerekend in verhouding tot het aantal resources dat gereserveerd is voor de VM, en het gebruik ervan. Betalende accounts kunnen tot 64 cores reserveren voor hun applicaties, terwijl ook hier de voor de thesis gebruikte trial account beperkt is tot 8 cores.
De VM kan in de cloud grotendeels worden beheerd alsof het gewoon een lokale VM is. Veel zaken zoals CPU-gebruik zijn rechtstreeks in de browser te monitoren. Google biedt ook standaard-images aan voor deze VM's zodat de configuratie ervan eenvoudig is.
Een belangrijke opmerking is dat het Flexible Environment van de Google App Engine onderliggend gebruik maakt van Compute Engine VM's om apps te hosten [88]. Dit betekent dat alle configuratie die het Flexible Environment van de Google App Engine biedt ook beschikbaar is in de Compute Engine, plus nog een aantal extra's. Voor deze masterproef is het interessant om ook de Google Compute Engine te bestuderen, aangezien dit een IaaS-service is waar toch heel wat gebruiksgemak aan gekoppeld is. Zo dekt deze masterproef naast een aantal PaaS-services ook een IaaS-platform.

10.3 Azure

Microsoft Azure, of kortweg Azure, is een door Microsoft beheerd cloud-platform dat erg bekend is bij .NET-ontwikkelaars. Azure biedt allerlei uiteenlopende services aan, gaande van data-opslag over versiebeheer geïntegreerd met Visual Studio tot IaaS en PaaS-oplossingen voor het creëren van virtuele machines en hosten van apps [90]. Het aanbod van Azure lijkt ongeveer even uitgebreid als dat van het Google Cloud Platform, maar de opdeling van de verschillende services is in Azure veel duidelijker, waardoor Azure transparanter en eenvoudiger is om te begrijpen. Dit is vooral voor beginnende gebruikers een groot voordeel. Voor deze masterproef is enkel de App Service dienst van Azure van belang. Over deze service in de volgende paragraaf een algemeen overzicht.

10.3.1 Azure App Service

App Service is het PaaS platform waarop eenvoudig apps in verschillende programmeertalen kunnen worden gedeployed. Hoewel Azure vooral bekend is vanuit de .NET wereld, ondersteunt Azure ook een heel aantal andere talen. Deze talen zijn [91]:

- .NET,
- Node.JS,
- PHP,
- Java,
- Python.

Er zijn voor de Azure App Service een groot aantal hulpmiddelen beschikbaar om het integreren van applicaties beschreven in bovenstaande talen in het Azure cloud platform eenvoudig te maken. Deze verschillen van taal tot taal en zijn gedocumenteerd in [91]. Voor deze masterproef is Azure gebruikt voor C# met .NET Core en F#.
Bij het hosten van een applicatie in de Azure App Service moet een App Service Plan worden aangemaakt. Binnen eenzelfde App Service plan kunnen meerdere verschillende applicaties huizen. Er zijn tal van configuratiemogelijkheden voor de applicaties, waaronder de locatie waar de app gehost moet worden. Voor dit onderzoek werd steeds Europe-West gekozen.
In de Azure Cloud gebeurt de facturering op basis van de lopende App Service Plans, en dus niet de applicaties zelf. Het kan dus zijn dat er wordt aangerekend zonder dat er een actieve applicatie in de cloud staat. De kosten voor Azure blijken vrij hoog te zijn. Voor dit onderzoek ongeveer 250 euro per maand voor een eenvoudige REST-API die amper verkeer krijgt in de cloud. Gelukkig biedt Azure net zoals de meeste andere cloud service providers een gratis trial versie met $200 aan gratis cloud-krediet, die maximaal een maand lang geldig is. Merk op dat afhankelijk van het type account en de prijs van de subscriptie meer mogelijkheden tot personalisatie van het App Service Plan verkrijgbaar zijn, en dat meer resources kunnen worden toegewezen aan applicaties, alsook meer parallelle instanties kunnen aangemaakt worden van eenzelfde applicatie. Op [92] is een lijst van alle beperkingen voor de gratis account te zien. Ondanks de vele beperkingen, is de gratis versie voldoende gebleken voor de tests uit dit onderzoek.

10.4 Wolfram Cloud

Ten slotte is voor Wolfram gebruik gemaakt van de geïntegreerde Wolfram Cloud. Deze werd in detail reeds toegelicht in paragraaf 5.2.7.
Wolfram Cloud biedt de mogelijkheid om gratis Wolfram-applicaties in hun PaaS-cloud te deployen. Deze applicaties zijn echter gebonden aan een maximaal verbruik van 1000 cloud credits per maand. Cloud credits zijn een virtueel betaalmiddel dat door Wolfram in het leven is geroepen om het gebruik van Wolfram Cloud-diensten mee te betalen [93]. Eén credit staat in essentie gelijk aan een enkele call naar een in de cloud gedeployde API of ander object, zolang het afhandelen van deze call niet meer dan 100 ms duurt. Indien dit wel het geval is, wordt één cloud credit aangerekend per 100 ms processortijd die het afhandelen van de request vraagt. Bepaalde specifieke taken zoals een email sturen kosten ook meerdere credits. Aangezien de tests in dit onderzoek heel wat rekentijd vragen, volstaat een gratis proefversie voor de Wolfram Cloud dus niet. Voor dit onderzoek is dan ook een Developer licentie genomen voor het Wolfram Development Platform [79]. Dit is een behoorlijk dure licentie van 160 euro per maand inclusief btw. Deze geeft echter toegang tot 40 000 cloud credits per maand, en daarbovenop nog extra voordelen zoals de mogelijkheid om een desktop-IDE te gebruiken, die veel sneller is dan het Development Platform in de Cloud. Bovendien biedt deze licentie de mogelijkheid om voor een goedkope prijs extra cloud-credits aan te kopen. Deze licentie is dus zeker voldoende voor dit onderzoek, hoewel het belangrijk is op te merken dat de gratis proefversie voor Wolfram Cloud dus veel beperkter is dan die voor alle andere geteste cloud-platformen, wat uiteraard een nadeel is.

11 Cloud Deployment

Dit hoofdstuk handelt over de implementatie van de verschillende talen in de cloud. Meer concreet hoe voor elke taal en elk gekozen cloud-platform de bestaande recursieve Driehoek van Pascal-applicaties uit vorige hoofdstukken omgevormd worden tot een RESTful web-API, en deze API in de cloud wordt gedeployed. De meeste cloud service providers bieden immers enkel de mogelijkheid om een applicatie in de cloud te deployen in de vorm van een webapplicatie.
De bedoeling van dit hoofdstuk is niet de documentatie van de verschillende gebruikte cloud-platformen en frameworks na te bootsen, maar eerder om een overzicht te scheppen van wat er allemaal komt kijken bij het deployen van een bestaande desktop-applicatie in de verschillende geteste cloud-omgevingen. Voor een allesomvattende handleiding van de cloud-platformen en al hun mogelijkheden verwijst dit onderzoek dan ook door naar de officiële documentatie van elk van de gebruikte cloud-platformen en frameworks.

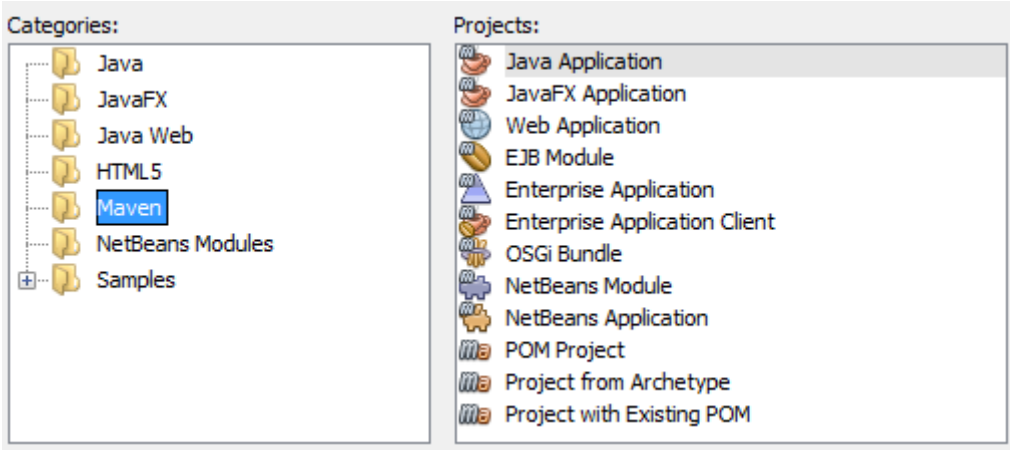
11.1 Java

Voor Java zijn er een heel aantal Cloud-platformen beschikbaar [94]. Zoals eerder toegelicht is het gekozen cloud-platform voor dit onderzoek het Google Cloud Platform. Meer bepaald de Google App Engine; zowel het Standard als Flexible Environment. Deze paragraaf licht eerst alle aanpassingen toe die aan de code gedaan dienen te worden om de recursieve Driehoek van Pascal om te vormen tot een REST-API. Vervolgs licht deze paragraaf toe hoe de API naar elk van de omgevingen binnen de Google App Engine gedeployed kan worden.

11.1.1 REST-API

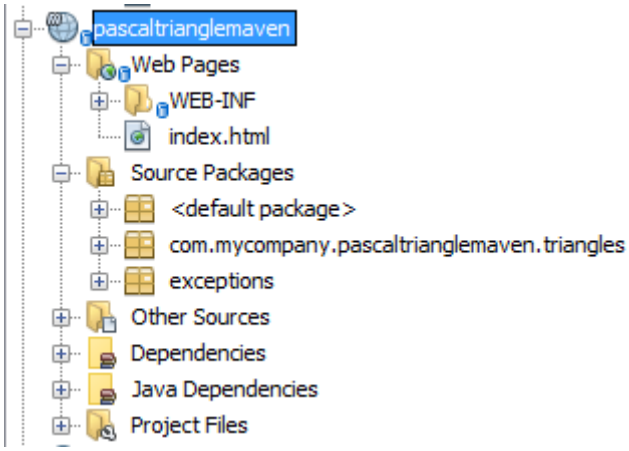
Om de Java-versie van de recursieve Driehoek van Pascal om te vormen tot een REST-API zijn heel wat aanpassingen en toevoegingen nodig aan de code. De manier waarop dit juist dient te gebeuren, is afhankelijk van de gebruikte ontwikkelingstools. In dit onderzoek is er voor alle Java-based projecten gebruik gemaakt van de NetBeans 8.0.2 IDE [95]. Onderstaande toelichting is dan ook enkel geldig voor deze IDE. De te volgen werkwijze is voor andere tools in grote lijnen allicht gelijkaardig, maar de details kunnen afwijken.
De eerste stap in het proces om de Java-versie van de recursieve Driehoek van Pascal om te vormen tot een REST-API is het aanmaken van een nieuwe Java EE-applicatie. In de NetBeans IDE moet hiervoor eerst de Java EE-component geïnstalleerd worden. Online zijn hier goede handleidingen voor te vinden.
De volgende stap is een degelijk framework aanwenden dat het schrijven van REST-API's vereenvoudigt. Voor Java is dit het Jersey framework [96]. Dit framework biedt alle mogelijkheden die te verwachten zijn van een REST-API framework, zoals RESTful routing, low-level afhandeling van http requests, etc. Dit alles is uitgebreid gedocumenteerd in [97]. Aangezien Jersey slechts een detail is voor dit onderzoek, worden geïnteresseerde lezers dan ook naar de documentatie doorverwezen voor gedetailleerde toelichting over alle mogelijkheden van Jersey.

Bij het gebruik van Jersey is het sterk aan te raden dit te combineren met Maven [98] en [99]. Maven is een onder Java-ontwikkelaars erg bekende tool die toelaat eenvoudig code te delen en het build-proces sterk te vereenvoudigen. Bovendien legt Maven een aantal standaarden op waardoor alle Maven-projecten een uniforme en duidelijke structuur hebben wat betreft documentatie, build-proces, en projectstructuur. Centrale Maven-repositories dienen als een knowledgebase waarvan allerlei project templates kunnen worden afgehaald om uit te werken tot volwaardige applicaties. Dit alles is open-source.
Van een centrale Maven-repository kan een template voor een RESTful Java EE Web API afgehaald worden, gebaseerd op het Jersey framework. De manier waarop dit best gebeurt, is afhankelijk van de gebruikte ontwikkelingstools. NetBeans biedt de optie om aan de hand van de plugin manager binnen de IDE een Maven-plugin te installeren, die de mogelijkheid geeft allerlei Maven archetypes te gebruiken voor het aanmaken van een nieuw project. Een Maven archetype is in essentie een template voor een project die voldoet aan de Maven-richtlijnen [100]. Figuur 188 geeft een overzicht van de verschillende beschikbare Maven archetypes binnen NetBeans na het installeren van de Maven-plugin.



Figuur 188: Verschillende Maven archetypes beschikbaar in NetBeans na het installeren van de Maven plugin.

Het Web Application archetype uit Figuur 188 is voor dit project het meest geschikt. Door deze optie te kiezen genereert NetBeans met behulp van Maven een volledige template voor een Java EE webapplicatie met de functionaliteit van Jersey ingebouwd. Figuur 189 geeft een overzicht van de structuur van het gegenereerde project. Merk op dat deze figuur is gemaakt op basis van het afgewerkte Java-project. De packages *com.mycompany.pascaltrianglemaven.triangles* en *exceptions* zijn specifiek aan dit project. De rest hoort bij het archetype. Daarnaast is het belangrijk om op te merken dat dit niet de eigenlijke mappenstructuur is van het project zelf, maar een gestructureerde interactieve weergave van het project binnen NetBeans. Alle bestanden van het project zijn wel vrij eenvoudig en intuïtief terug te vinden in de eigenlijke mappenstructuur van het project. Voor deze masterproef zou een gedetailleerde bespreking van deze mappenstructuur echter te ver leiden. Om die reden worden hieronder enkel de nodige aanpassingen aan het project besproken om de recursieve Driehoek van Pascal in Java om te vormen tot een REST-API.



Figuur 189: Structuur van het Web Application Maven archetype.

De eerste map binnen het project is meteen van belang voor het maken van een REST-API. In de Web pages map kunnen de verschillende html-pagina's van de webapplicatie zitten. Standaard wordt een index.html file gegenereerd met generische inhoud. Voor de REST-API is deze html-pagina onbelangrijk en kan ze genegeerd worden. Verder bevat de Web Pages map ook de WEB-INF submap. Deze is voorlopig nog onbelangrijk, maar zal in een later stadium essentieel blijken.
De tweede map in het Web Application archetype is Source Packages. Deze map bevat alle Java back-end source code van het project. Voor een REST-API is dit tevens zo goed als alle code van het hele project. Zoals te zien is in Figuur 189 zijn er voor dit project drie submappen gemaakt in de Source Packages, met elk hun eigen functie. In het default package moeten bij conventie alle Servlets van de applicatie zitten. Servlets zijn speciale Java-klassen die verantwoordelijk zijn voor het afhandelen van binnenkomende http-requests. Voor deze API is er slechts één Servlet nodig. Figuur 190 geeft de code voor deze Java-Servlet weer.

```
WebServlet(urlPatterns = {"/pascaltriangle"})
public class PascalTriangle extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {

        try{
            int size = Integer.parseInt(request.getParameter("size"));
            String version = request.getParameter("version");

            String json;

            if(version.equals("oo")){
                int[][] points;
                switch(request.getParameter("type")){
                    case "sequential": points =
                    new PascalTriangleRec(size).getPoints(); break;
                    case "parallelrows": points =
                    new PascalTriangleRecParallelRows(size).getPoints();
                    break;
                    case "parallelcols": points =
                    new PascalTriangleRecParallelCols(size).getPoints();
                    break;
                    case "parallelrowscols": points =
                    new PascalTriangleRecParallelRowsCols(size)
                    .getPoints(); break;
                    default: throw new InvalidInputException();
                }
                json = new JSONArray(points).toString();
            }
            else if (version.equals("functional")){
                List<List<Integer>> points;
                switch(request.getParameter("type")){
                    case "sequential": points =
                    new FunctionalPascalTriangle(size).getPoints(); break;
                    case "parallelrows": points =
                    new FunctionalPascalTriangleParallelRows(size)
                    .getPoints(); break;
                    case "parallelcols": points =
                    new FunctionalPascalTriangleParallelCols(size)
                    .getPoints(); break;
                    case "parallelrowscols": points =
                    new FunctionalPascalTriangleParallelRowsCols(size)
                    .getPoints(); break;
                    default: throw new InvalidInputException();
                }
                json = new JSONArray(points).toString();
            }
            else throw new InvalidInputException();

            response.getWriter().write(json);
        }
        catch(InvalidInputException ex){
            response.getWriter().write("Invalid input!");
        }
    }
```

Figuur 190: Java Servlet voor het verwerken van http-requests en het genereren van de gepaste Driehoek van Pascal.

Merk op dat er een groot aantal import statements nodig zijn, die voor de overzichtelijkheid hier zijn weggelaten.
De allereerste regel code geeft aan dat de bovenliggende routing-laag het routing-patroon '/pascaltriangle' naar deze Servlet dient te leiden. Bij de definitie van de klasse wordt verder meegegeven dat ze van de grondklasse *HttpServlet* erft. Hierin zit alle functionaliteit voor het verwerken van http-requests ingebouwd. Om een http GET-request te verwerken, dient de methode *doGet* uit de *HttpServlet* klasse te worden overschreven. Deze methode heeft als parameters een object van de klasse *HttpServletRequest*, en een object van de klasse *HttpServletResponse*. Deze objecten bevatten alle nodige informatie betreffende respectievelijk de binnengekomen request, en de te versturen response.
De parameters van de request kunnen aan de hand van de functie *getParameter* in de *HttpServletRequest* klasse worden opgevraagd. Voor deze API zijn de parameters *version*, *type*, en *size* gekozen. *Version* laat toe te kiezen tussen de OO- en Streams-implementatie van de recursieve Driehoek van Pascal. *Type* dient om te kiezen tussen de verschillende parallellisatie-opties, en *size* bepaalt het aantal rijen van de te genereren Driehoek van Pascal. Aan de hand van if- en switch-statements kan dan op basis van de waarde van de parameters *version* en *type* de juiste versie van de Driehoek van Pascal met *size* rijen gegenereerd worden, waarna de gegenereerde punten kunnen worden opgevraagd.
De parameters *version* en *size* hebben slechts enkele geldige waardes. Daarom is het geheel omvat in een try-catch-controlestructuur. Indien *version* of *size* geen geldige waarde hebben, wordt een *InvalidInputException* gegenereerd. Dit is een custom exception die in een andere package binnen de Source Packages van deze applicatie gedefinieerd is. Indien de code deze exception oproept, wordt tekst 'Invalid input!' gegenereerd als response.
Alvorens de gegenereerde driehoek als response kan worden teruggegeven, moet de 2D array van Integers die de driehoek voorstelt nog worden omgezet naar een 2D JSON-array. Ook dit is in Java niet evident. Hiervoor moet gebruik gemaakt worden van een third-party bibliotheek die het genereren van JSON-strings op basis van Java-objecten toelaat. Voor dit project is gekozen voor de *org.json*-bibliotheek. Deze kan met behulp van Maven als afhankelijkheid worden ingevoegd in het project. Hiervoor moet in de pom.xml file, die dienst doet als een soort configuratiebestand voor het project, en die huist in de Project Files directory in Figuur 189, een klein volgens de Maven-conventies geformatteerd blokje XML-code worden bijgevoegd dat aangeeft dat het project een afhankelijkheid heeft, en waar deze afhankelijkheid te vinden is. Hiervoor bestaat een speciaal 'dependencies' XML-object in de pom.xml file, weergegeven in Figuur 191.

```
<dependencies>
    <dependency>
        <groupId>javax</groupId>
        <artifactId>javaee-web-api</artifactId>
        <version>7.0</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.json</groupId>
        <artifactId>json</artifactId>
        <version>20160810</version>
    </dependency>
</dependencies>
```

Figuur 191: Dependencies in de pom.xml file van de Java Web Application.

Zoals hierboven te zien is, is de org.json afhankelijkheid toegevoegd. Om dit te doen is het nodig om een *groupId* mee te geven, een *artifactId*, en een versienummer. Dit is vastgelegd bij conventie. De exacte waardes hiervoor zijn online te vinden. Na het toevoegen van de extra afhankelijkheid is het belangrijk om heel het project opnieuw te builden met Maven. Uiteraard dient de optie 'build with dependencies' gekozen te worden. Dit zal de nieuwe *org.json* afhankelijkheid downloaden en bijvoegen in het project onder de map 'Dependencies' uit Figuur 189, in de vorm van een jar-bestand.
Na dit alles gedaan te hebben is het uiteindelijk mogelijk een JSON-Array te vormen van de coëfficiënten van de Driehoek van Pascal. Hiervoor bestaat er in de *org.json*-bibliotheek de *JSONArray* klasse. Door een nieuwe *JSONArray* te maken van de 2D-Array points en daar de *toString*-methode van te gebruiken, ontstaat een JSON-string met daarin de coëfficiënten van de opgevraagde Driehoek van Pascal. Deze kan ten slotte worden meegegeven naar het HttpResponse object aan de hand van de *getWriter* en *write* methodes van de *HttpResponse*-klasse.
De andere packages in de Source Packages map uit Figuur 189 bevatten respectievelijk de verschillende *PascalTriangle*-klassen die beschreven zijn in paragrafen 7.1.1 en 9.2.1, en de *InvalidInputException* die erft van de klasse *Exception* maar verder leeg is.
Verder zijn er nog de mappen Dependencies en Java Dependencies. Dependencies bevat zoals de naam al zegt de afhankelijkheden die het project nodig heeft, gedeclareerd in de pom.xml file zoals hierboven reeds aangegeven. De Java Dependencies map bevat de JDK en de nodige bibliotheken om een Java EE webapplicatie te maken. Maven beheert beide Dependencies mappen, en de programmeur moet hier niet rechtstreeks wijzigingen in aanbrengen.
De laatste map in het Java Web Application archetype is de Project Files map. Deze is van groot belang voor de algemene configuratie van de REST-API. De belangrijke pom.xml file die hierboven al aangehaald is huist hierin. In de pom.xml file worden allerhande algemene projectparameters ingesteld, zoals de Java-versie, afhankelijkheden, etc.
Naast de pom.xml file bevat de Project Files directory ook het nb-configuration.xml bestand. Deze is specifiek voor het configureren van de modules in de NetBeans IDE, en is dan ook relatief onbelangrijk voor dit onderzoek, aangezien deze configuratie standaard gegenereerd is en geen aanpassingen van de programmeur vraagt voor dit onderzoek.

11.1.2 Google App Engine Standard Environment

Na alle bovenstaande stappen ondernomen te hebben is de Java-versie van de recursieve Driehoek van Pascal omgevormd tot een REST-API. Dit is slechts de eerste stap. Nu moet de applicatie nog in de cloud gedeployed worden. Deze paragraaf bespreekt dit proces voor het Standard Environment van de Google App Engine. Online biedt Google reeds een gedetailleerde handleiding aan voor dit proces op [101]. Hieronder de belangrijkste stappen van dit proces, met enige toelichting.

De eerste stap in het deployment-proces voor eender welke Google Cloud Platform-applicatie is het aanmaken van een nieuw Google Cloud Platform-project. Dit gebeurt rechtstreeks in de console van Google Cloud Platform in de browser. Daarna moet er in het Google Cloud Platform-project nog expliciet een Google App Engine applicatie worden aangemaakt. Bij het aanmaken van de applicatie dient enerzijds de programmeertaal gespecifieerd te worden van de app, en anderzijds de locatie waar de app gedeployed dient te worden. Deze keuze is achteraf niet meer te wijzigen. Voor dit onderzoek is voor alle apps in het Google Cloud Platform Europe-West als regio gekozen.

Na bovenstaande voorbereidingen in het cloud platform zijn er op de lokale pc ook een aantal zaken te doen. Eerst en vooral moet de commandolijn-versie van Maven worden geïnstalleerd. Dit is niet hetzelfde als de Maven-Plugin binnen NetBeans. Google heeft immers een handige plugin ontwikkeld voor Maven die een aantal commando's bevat die het cloud-deployen van een Java Web App sterk vereenvoudigen. Het is belangrijk dat bij het installeren van Maven een JAVA_HOME-omgevingsvariabele is ingesteld en dat die wijst naar de map die de Java-SDK bevat op de lokale pc. Ook kan Maven best worden toegevoegd aan de PATH-omgevingsvariabele van het systeem voor gebruiksgemak.

Nadat Maven geïnstalleerd is moet de Google App Engine Maven plugin worden toegevoegd aan het project. Dit laat toe gebruik te maken van de eerder vermelde door Google gecreëerde Maven-commando's die het deployen van een Java Web App naar de Google App Engine sterk vereenvoudigen. De plugin dient gedeclareerd te worden in de pom.xml file van het project. Figuur 192 geeft dit weer.

```
<plugin>
    <groupId>com.google.appengine</groupId>
    <artifactId>appengine-maven-plugin</artifactId>
    <version>1.9.51</version>
</plugin>
```

Figuur 192: Declaratie Google appengine Maven plugin in pom.xml.

Aan de code van de Java Web App zelf moeten ook enkele aanpassingen gebeuren. Ten eerste gaf paragraaf 10.2.1.1 reeds aan dat het Standard Environment van de Google App engine enkel Java 7 ondersteunt, wat betekent dat de functionele varianten van de Driehoek van Pascal verwijderd moeten worden uit het project. Streams zijn immers pas ondersteund sinds Java 8. De code voor de PascalTriangle-Servlet uit Figuur 190 ziet er nu uit als in Figuur 193.

```
public class PascalTriangle extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        try{
            int size = Integer.parseInt(request.getParameter("size"));
            String json;

                int[][] points;
                switch(request.getParameter("type")){
                    case "sequential": points =
                    new PascalTriangleRec(size).getPoints(); break;
                    case "parallelrows": points =
                    new PascalTriangleRecParallelRows(size).getPoints();
                    break;
                    case "parallelcols": points =
                    new PascalTriangleRecParallelCols(size).getPoints();
                    break;
                    case "parallelrowscols": points =
                    new PascalTriangleRecParallelRowsCols(size)
                    .getPoints(); break;
                    default: throw new InvalidInputException();
                }
                json = new JSONArray(points).toString();

            response.getWriter().write(json);
        }
        catch(InvalidInputException ex){
            response.getWriter().write("Invalid input!");
        }
    }
}
```

Figuur 193: Java 7-versie van de Servlet voor de Driehoek van Pascal API.

Ook moet er in de pom.xml file specifiek gedefinieerd worden dat het compilatietarget Java 1.7 is. Figuur 194 geeft dit weer.

```
<properties>
    <endorsed.dir>${project.build.directory}/endorsed</endorsed.dir>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.target>1.7</maven.compiler.target>
    <maven.compiler.source>1.7</maven.compiler.source>
</properties>
```

Figuur 194: Properties voor de Driehoek van Pascal API in Java in de pom.xml-file.

Naast deze aanpassingen moeten in de WEB-INF-directory van Figuur 189 twee bestanden worden bijgevoegd. Het eerste bestand heet bij conventie appengine-web.xml. Hierin staat de informatie die Google nodig heeft om het project aan de net aangemaakte App Engine applicatie te linken. Figuur 195 geeft de inhoud van dit bestand weer.

```
<?xml version="1.0" encoding="utf-8"?>
<appengine-web-app xmlns="http://appengine.google.com/ns/1.0">
    <application>pascaltriangleapi</application>
    <version>1</version>
    <threadsafe>true</threadsafe>
</appengine-web-app>
```

Figuur 195: Inhoud appengine-web.xml.

Bovenstaande inhoud is minimaal en opgelegd door Google. Application verwijst naar de pas aangemaakt applicatie, en version is een verplicht versienummer. Threadsafe geeft aan dat de applicatie probleemloos mag schalen als de vraag groot is. Optioneel kunnen er nog extra elementen worden toegevoegd in dit configuratiebestand die een meer gedetailleerde controle toelaten over hoe de applicatie in de cloud wordt gedeployed. Dit zou echter te ver gaan voor dit onderzoek, en geïnteresseerde lezers worden doorverwezen naar [83].
Een volgende belangrijke aanpassing die moet gedaan worden in het Java-project, die niet goed gedocumenteerd is door Google, is de routing van de Web App. Voorheen volstond het om bovenaan de servlet-klasse een routing annotatie toe te voegen. Om te deployen naar de Google App Engine is dit echter niet meer voldoende. Een alternatieve manier om aan routing te doen binnen Java Web Apps is door een web.xml bestand toe te voegen in de WEB-INF-directory. Dit bestand heeft niet rechtstreeks iets te maken met het Google Cloud Platform, maar is dus noodzakelijk om de applicatie hierin te laten werken. De voornaamste functie van dit bestand is routing informatie bieden voor de Java Web App op een hoog niveau, in plaats van in elke servlet de routes te specifiëren. Voor deze API werden volgende routes uit Figuur 196 gespecifieerd.

```
<servlet>
    <servlet-name>pascaltriangle</servlet-name>
    <servlet-class>PascalTriangle</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>pascaltriangle</servlet-name>
    <url-pattern>/pascaltriangle</url-pattern>
</servlet-mapping>
```

Figuur 196: Routing in de Web.xml-file in de Java web-API voor de Driehoek van Pascal.

De werking van bovenstaande code spreekt voor zich.
Voor de geparallelliseerde versies van de Driehoek van Pascal stellen zich nog enkele problemen. Een aantal zaken zijn immers niet toegestaan of mogelijk in Google App Engine Standard Environment. Een volledig overzicht van al deze zaken is te vinden in [102]. Enkele van deze beperkingen zijn van belang voor de hierboven gecreëerde API. Ten eerste laat Google niet toe dat gebruikers zomaar zelf threads aanmaken en beheren aan de hand van de in Java ingebouwde Thread-klasse in het Google App Engine Standard Environment. In de plaats hiervan schrijft Google voor gebruik te maken van een *ThreadFactory*-object voor het genereren van threads. Dit *ThreadFactory*-object moet aangemaakt worden aan de hand van de *currentRequestThreadFactory*-methode van de *ThreadManager*-klasse, die deel is van het *com.google.appengine.api* package. Dit package moet als een afhankelijkheid aan het project worden toegevoegd. Dit gebeurt net zoals voorheen door de afhankelijkheid te declareren in

de pom.xml file, en het project opnieuw te builden met alle afhankelijkheden. Figuur 197 geeft de declaratie van deze afhankelijkheid weer.

```
<dependency>
    <groupId>com.google.appengine</groupId>
    <artifactId>appengine-api-1.0-sdk</artifactId>
    <version>1.9.51</version>
</dependency>
```

Figuur 197: Afhankelijkheid die het gebruik van multithreading in Google App Engine Standard Environment toelaat.

Vervolgens moeten de klassen *com.google.appengine.api.ThreadManager* en *java.util.concurrent.ThreadFactory* geïmporteerd worden in elk van de klassen die parallellisatie toepassen. Hierna kan aan de hand van de *currentRequestThreadFactory*-methode een *ThreadFactory*-object worden aangemaakt. Dit object biedt de mogelijkheid aan de hand van de methode *newThread* nieuwe threads aan te maken. Figuur 198 geeft weer hoe de constructor van de *PascalTriangleRecParallelRows*-klasse er uit ziet na de implementatie van de hierboven beschreven aanpassingen. Deze aanpassingen zijn analoog voor de *PascalTriangleRecParallelCols*- en *PascalTriangleRecParallelRowsCols*-klassen.

```
public PascalTriangleRecParallelRows(int size){
    points = new int[size][];
    final List<Thread> threads = new ArrayList();
    ThreadFactory factory = ThreadManager.currentRequestThreadFactory();

    for (int i =0;i<size;i++){
        final int iBuffer = i;
        Thread rowThread = factory.newThread(new Runnable() {
            @Override
            public void run() {
                final int[] row  = new int[iBuffer+1];
                for (int j = 0;j<=iBuffer;j++)
                  row[j]=getValueAtPoint(iBuffer,j);
                points[iBuffer]=row;
            }
        });
        rowThread.start();
        threads.add(rowThread);
    }
    for(int i = 0; i<threads.size(); i++) {
        try {
            threads.get(i).join();
        } catch (InterruptedException ex) {
            System.err.println("Something went wrong");
        }
    }
}
```

Figuur 198: Constructor van de PascalTriangleParallelRows-klasse gebruik makend van de ThreadManager-klasse zoals voorgeschreven door Google voor gebruik in het App Engine Standard Environment.

Een laatste obstakel om de Java-versie van de recursieve Driehoek van Pascal in de cloud te deployen is dat het maximumaantal threads dat een applicatie in het Standard Environment van de Google App Engine mag aanmaken voor het afhandelen van een request door Google beperkt is tot 50. Als een applicatie toch meer threads probeert aan te maken, wordt een

IllegalStateException gegenereerd in de applicatie. Voor dit onderzoek is dit een probleem, aangezien de geparallelliseerde versie voor rijen en kolommen veel meer dan 50 threads kan aanmaken. Deze versie maakt immers een thread aan voor elke coëfficiënt van de driehoek. In theorie is dit ook een probleem voor de twee andere parallellisatiemethoden. Deze ondervinden in realiteit echter geen hinder door deze beperking. Het algoritme is immers zodanig traag dat een Pascal-driehoek met meer dan 50 rijen (en dus ook kolommen in minstens één rij) onrealistisch veel rekentijd zou vragen. Ook deze rekentijd is in het Google App Engine Standard Environment trouwens beperkt tot enkele seconden voor het afhandelen van een request.
De code van de *PascalTriangleRecParallelRowsCols*-klasse dient dus aangepast te worden om rekening te houden met deze opgelegde beperking door Google. Figuur 199 geeft de uiteindelijke code van deze klasse weer. Merk op dat enkel de relevante code is getoond om zo veel mogelijk overzichtelijkheid te behouden.

```
public PascalTriangleRecParallelRowsCols(int size){
    points = new int[size][];
    final ThreadFactory factory =
ThreadManager.currentRequestThreadFactory();
    final List<Thread> threads = new ArrayList();
    final List<Thread> rowThreads = new ArrayList();

    for (int i =0;i<size;i++){
        final int iBuffer = i;
        while(countRunningThreads(threads)>=49){
            try {
                Thread.sleep(10);
            } catch (InterruptedException ex) {}
        }
        Thread rowThread = factory.newThread(new Runnable() {
            @Override
            public void run() {
                final int[] row  = new int[iBuffer+1];
                List<Thread> colThreads = new ArrayList();
                for (int j = 0;j<=iBuffer;j++){
                    final int jBuffer = j;
                    Thread colThread = factory.newThread(new Runnable() {
                    @Override
                    public void run() {
                        row[jBuffer]=getValueAtPoint(iBuffer,jBuffer);
                    }
                    });
                    while(countRunningThreads(threads)>=49){
                        try {
                            Thread.sleep(10);
                        } catch (InterruptedException ex) {}
                    }
                    threads.add(colThread);
                    colThreads.add(colThread);
                    colThread.start();
                }
```

```
                for(Thread t : colThreads){
                    try {
                        t.join();
                    } catch (InterruptedException ex) {}
                }
                points[iBuffer]=row;
            }
        });
        while(countRunningThreads(threads)>=49){
            try {
                Thread.sleep(10);
            } catch (InterruptedException ex) {}
        }
        threads.add(rowThread);
        rowThreads.add(rowThread);
        rowThread.start();
    }
    for(Thread t : rowThreads){
        try {
            t.join();
        } catch (InterruptedException ex) {}
    }
}
int countRunningThreads(List<Thread> threads){
    int activeThreads = 0;
    int i = 0;
    while(i<threads.size()){
        try{
            Thread t = threads.get(i);
            if (t!=null && t.isAlive()) activeThreads++;
            i++;
        }catch(ArrayIndexOutOfBoundsException ex){
            return activeThreads;
        }
    }
    return activeThreads;
}
```

Figuur 199: Code constructor PascalTriangleRecParallelRowsCols-klasse na aanpassingen om nooit meer dan 50 threads tegelijk aan te maken.

Zoals te zien op bovenstaande figuur zijn de aanpassingen om niet meer dan 50 threads aan te maken niet erg evident, en is er heel wat extra code nodig om dit te bereiken. Ten eerste is een *countRunningThreads*-methode toegevoegd. Deze methode telt het aantal actieve threads dat de constructor aanmaakte. De constructor zelf controleert voor het aanmaken van een nieuwe thread steeds of het aantal threads nog niet te hoog is aan de hand van deze methode. Indien dit wel zo is wacht de constructor 10 ms alvorens een extra thread aan te maken. Zoals Figuur 199 illustreert is de code voor de *PascalTriangleRecParallelRowsCols*-klasse nu behoorlijk lang en onoverzichtelijk door alle beperkingen die zowel Java als Google stellen voor deze applicatie. Dit is uiteraard een belangrijk argument tegen het toepassen van Java in deze cloud-omgeving, alvast voor sterk geparallelliseerde toepassingen.
Nadat alle bovenstaande aanpassingen doorgevoerd zijn kan uiteindelijk de Java Web App gedeployed worden naar het Google App Engine Standard Environment. Hiervoor volstaat het een nieuw commandolijn-venster te openen, en te navigeren naar de top-level directory van het Java Web App-project. Daarna kan aan de hand van het commando *'mvn appengine:update'* de Web App naar de Google App Engine gepushed worden.

Dit commando haalt alle nodige afhankelijkheden af en verwerkt deze in het project, buildt het geheel, en deployt het naar de Google App Engine-applicatie gedefinieerd in het appengine-web.xml bestand. Indien er nog geen Google-account geassocieerd is met deze app, opent het commando eerst een browser-venster met de vraag om in te loggen. Indien dit succesvol verloopt, geeft Google een code terug in de browser die geplakt dient te worden in de commandolijn console. Uiteindelijk wordt de app dan gedeployed in de cloud en is de API via een URL aan te spreken. Deze URL is te zien in de Google App Engine console in de browser.
Het proces om een Java EE webapplicatie in het Standard Environment van de Google App Engine te deployen is dus behoorlijk langdradig. Na alle hierboven beschreven voorbereidingen is het updaten van de app echter zeer eenvoudig. Dit kan gewoon door het '*mvn appengine:update*'-commando opnieuw uit te voeren vanuit de top-level map van het project. Versiebeheer is eenvoudig mogelijk door het versienummer in de appengine-web.xml van het project aan te passen. Ook is er de mogelijkheid een app op te delen in verschillende services, om zo nog beter code van grote projecten te kunnen beheren. Dit laatste valt echter buiten het bestek van dit onderzoek.

11.1.3 Google App Engine Flexible Environment

Het grootste nadeel van de hierboven beschreven implementatiemethode in het Standard Environment van de Google App Engine is dat deze omgeving enkel Java 7 ondersteunt, waardoor de implementaties van de Driehoek van Pascal met Streams hierin niet deploybaar zijn. Het Flexible Environment biedt een oplossing voor dit probleem, aangezien deze omgeving ook Java 8 ondersteunt. De achterliggende technologie voor het Flexible Environment is immers compleet anders dan voor het Standard Environment. Zoals beschreven in paragraaf 10.2.1.2 werkt het Flexible Environment met Docker containers. De Java web-app die eerder in dit hoofdstuk werd aangemaakt moet dus voorbereid worden voor encapsulatie in een Docker-container, die dan in de cloud gedeployed wordt. Deze paragraaf beschrijft de belangrijkste stappen van dit proces. Deze zijn immers ook voor het Flexible Environment talrijk, en wijken bovendien sterk af van de werkwijze voor deployment naar Standard Environment. De meeste van deze stappen zijn terug te vinden in de documentatie van het Google Cloud Platform, in de eerste plaats op [103]. Gelukkig bestaat er zoals in paragraaf 10.2.1.2 beschreven voor Java 8 een standaardconfiguratie voor de Docker container die door Google zelf ter beschikking gesteld is. Dit vereenvoudigt het proces sterk.
Aangezien de aanpassingen die aan een Java Web App moeten gebeuren om deze in het Flexible Environment van de Google App engine te hosten geheel anders zijn dan de aanpassingen nodig voor een deployment naar het Standard Environment, kan er dus voor dit proces best vertrokken worden van een nieuwe Java Web App, waar nog geen enkele aanpassing aan gebeurd is betreffende het Google Cloud Platform. Dit onderzoek vertrekt dus van een kopie van de Java Web App-versie van het recursieve Driehoek van Pascal-algoritme in Java, die in paragraaf 11.1.1 beschreven is.

Alvorens kan begonnen worden met de eigenlijke implementatie van de Java Web App dient eerst de Google Cloud SDK geïnstalleerd te zijn op de lokale pc. Dit is een set command-line tools die het beheren van projecten en apps in het Google Cloud Platform aanzienlijk vereenvoudigen, alsook het deployen en updaten van apps, etc. [104].
Bij de installatie van de Google Cloud SDK op zich komen echter enkele complicaties kijken. Voor de Cloud SDK kan geïnstalleerd worden, moet immers Python geïnstalleerd zijn op de machine. Enkel zeer specifieke versies van Python volstaan, namelijk versies van Python 2.7. Dit belangrijk detail is slecht gedocumenteerd en kan tot veel frustratie leiden bij nieuwe gebruikers. Merk op dat Python ook aan het systeem PATH moet worden toegevoegd. Indien dit niet gebeurt crasht de Google Cloud SDK.
Nadat voorgaande stappen ondernomen zijn kan de Google Cloud SDK zelf geïnstalleerd worden. Deze installatie spreekt voor zich. Het is wel aangewezen onmiddellijk na de installatie het commando '*gcloud init*' uit te voeren. Hiermee kan ingelogd worden op een Google-account, en kan een project geselecteerd worden. Bovendien kan een locatie ingesteld worden waar de apps in Google Compute Engine gehost zullen worden. Deze configuratie is eenmalig, maar is altijd aan te passen achteraf. Alle andere gcloud- en Maven- commando's die Google beschikbaar stelt via de App Engine SDK gaan uit van de ingestelde parameters aan de hand van *gcloud init*. Best kunnen eerst een Google Cloud Platform-project en een Google App Engine-applicatie aangemaakt worden in de console van het Google Cloud Platform in de browser op dezelfde manier als bij de vorige implementatie in het Standard Environment, alvorens *gcloud init* wordt uitgevoerd. Op die manier is het doorlopen van *gcloud init* vanzelfsprekend.
Na dit alles dienen nog enkele aanpassingen aan het project zelf te gebeuren. Ten eerste moet ook voor het Flexible Environment de Google Cloud Maven-plugin worden toegevoegd aan de pom.xml file, zoals in vorige paragraaf beschreven. Deze plugin is echter niét dezelfde als die voor het Standard Environment. Figuur 200 geeft de exacte plugin-declaratie weer die dient gebruikt te worden voor het Flexible Environment.

```
<plugin>
    <groupId>com.google.cloud.tools</groupId>
    <artifactId>appengine-maven-plugin</artifactId>
    <version>1.3.0</version>
</plugin>
```

Figuur 200: Appengine Maven plugin voor het Google App Engine Flexible Environment.

Daarnaast moet er ook een Jetty-plugin worden toegevoegd aan het project in de pom.xml. Jetty zal binnen de Docker Container een Java Servlet web-server-infrastructuur inrichten, waardoor de container effectief een webserver wordt die luistert op een bepaalde poort en het verkeer doorverbindt met het Java-project [105]. Figuur 201 geeft de code weer voor het toevoegen van de Jetty-plugin aan het project.

```
<plugin>
    <groupId>org.eclipse.jetty</groupId>
    <artifactId>jetty-maven-plugin</artifactId>
    <version>9.4.2.v20170220</version>
</plugin>
```

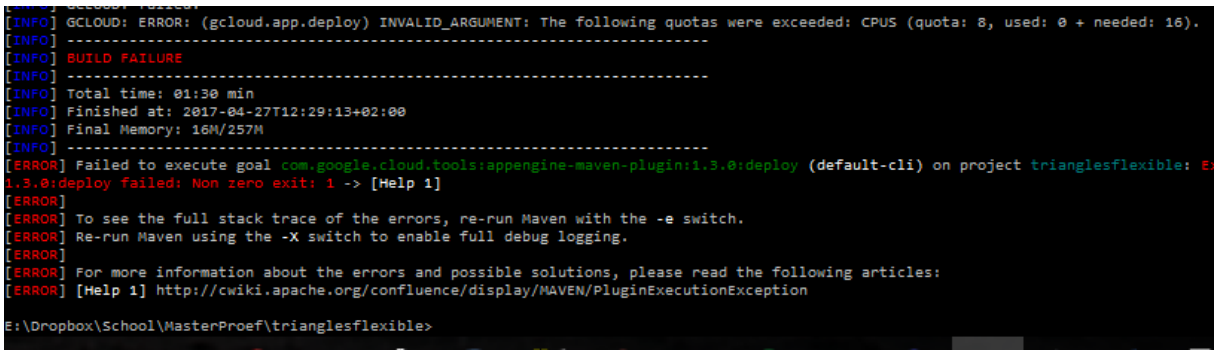
Figuur 201: Jetty plugin.

Voor de zekerheid kan men best ook de routing voor de web-app specifiëren aan de hand van een web.xml file in de WEB-INF-map van de Java Web App, op dezelfde manier als eerder uitgelegd voor het Standard Environment.
De algemene configuratie voor het Google App Engine Flexible Environment op zich gebeurt aan de hand van een app.yaml bestand. Deze dient manueel te worden toegevoegd aan het project, in de map /src/main/appengine, die uiteraard ook aangemaakt dient te worden. De standaard app.yaml is zeer beperkt, maar hier kunnen allerlei optionele extra's aan worden toegevoegd. Deze allemaal in detail bespreken zou te ver gaan voor dit onderzoek. De documentatie van Google biedt bovendien uitgebreide info hierover. Figuur 202 geeft de inhoud van de app.yaml file weer voor dit project.

```
runtime: java
env: flex

handlers:
- url: /.*
  script: this field is required, but ignored

resources:
  cpu: 4
  memory_gb: 8
```

Figuur 202: App.yaml voor dit project.

De runtime, env, en handlers velden zijn verplicht en zijn voor dit project niet gewijzigd van de standaardversie. Het optionele veld resources is echter wel toegevoegd. Dit om zo veel mogelijk CPU-kernen toe te wijzen aan de applicatie. Het maximumaantal kernen dat toegewezen kan worden aan de applicatie met een gratis account is 8. Om één of andere reden wordt echter nog steeds een foutmelding gegeven indien 8 kernen worden aangegeven in de app.yaml. Bij het initialiseren lijkt het erop dat het aantal kernen verdubbelt ten opzichte van wat is aangegeven in de app.yaml file. Figuur 203 geeft de gegenereerde foutmelding weer wanneer een app met 8 kernen in de app.yaml gedeployed wordt naar de cloud. Uiteindelijk is voor dit project dan maar noodgedwongen gebruik gemaakt van slechts 4 CPU-kernen.



Figuur 203: Output wanneer het deployment proces wordt gestart met 8 cpu kernen in de app.yaml file.

Ten slotte is het nog belangrijk om op te merken dat het toegewezen geheugen voor de applicatie verplicht tussen de 0,9 en 6,5 GB moet liggen per toegewezen CPU-kern. Om die reden is zoals in Figuur 202 te zien is het geheugen voor de applicatie vastgelegd op 8GB, terwijl dit in de praktijk veel te veel is voor deze toepassing.
Naast app.yaml kunnen er nog een heel aantal optionele configuratiebestanden worden toegevoegd aan het Java project, waaronder een gepersonaliseerde Dockerfile. Voor dit project is dit echter niet noodzakelijk gebleken, en voor extra uitleg over alle configuratie-opties verwijst dit onderzoek geïnteresseerde lezers dan ook door naar [103].
Na alle hierboven beschreven stappen is het deployen van het project op zich zeer eenvoudig. Het volstaat om in de commandolijn-terminal te navigeren naar de top-level map van het project, en het commando '*mvn appengine:deploy*' uit te voeren. Maven zal het project bouwen, en aan de hand van de Google App Engine-plugin het project deployen in het Flexible Environment van de Google App Engine. De URL van de gedeployde service is in de commandolijn-terminal terug te vinden, alsook in de console van de Google App Engine in de browser. Net zoals bij het Standard Environment is het deployen van nieuwe versies eenvoudig mogelijk door het commando '*mvn appengine:deploy*' opnieuw uit te voeren vanuit de top-level map van het project.

11.2 C#

Ook de C#-versies van de recursieve Driehoek van Pascal-applicatie zijn in de cloud gedeployed. Dit is net zoals voor Java in de vorige paragraaf gedaan in de vorm van een RESTful web-API. Voor C# is zoals eerder vermeld Azure het geprefereerde cloud-platform, aangezien Microsoft zowel eigenaar is van .NET/C# als Azure, waardoor dit onderzoek verwacht dat ondersteuning voor C# in Azure zeer goed is. De Azure technologie die voor dit onderzoek gekozen is, is de Azure App Service, die zoals in paragraaf 10.3 toegelicht het PaaS-platform is binnen Azure. Langs de andere kant is de C#-applicatie ook in het Google Cloud Platform gedeployed. Meer specifiek in de Google Compute Engine. Dit heeft meerdere interessante implicaties naar vergelijking toe. Niet enkel kan de 'in-house' Azure-technologie vergeleken worden met een externe CSP in de vorm van het Google Cloud Platform, maar ook kan hierdoor een PaaS-platform rechtstreeks vergeleken worden met een IaaS platform. Deze paragraaf beschrijft eerst hoe de C#-versie van de recursieve Driehoek van Pascal om te vormen is tot een REST-API. Daarna beschrijft deze paragraaf alle stappen om deze API te deployen in de hierboven gespecifieerde cloud-platformen.

11.2.1 REST-API

Net als bij Java is voor C# de eerste stap om de recursieve Driehoek van Pascal in de cloud te deployen de desktop-applicatie omvormen tot een RESTful web-API. Om dit te doen zijn er meerdere mogelijkheden. C# biedt immers verschillende frameworks om webapplicaties in te onwtikkelen, namelijk .NET en het recentere .NET Core. Deze laatste is een verdere ontwikkeling van .NET, maar is op zoveel vlakken verschillend van .NET dat ze in de praktijk als verschillende frameworks beschouwd kunnen worden. .NET Core is bijvoorbeeld cross-platform, terwijl .NET enkel Windows ondersteunt. Om die reden zijn - althans voorlopig- niet alle features van .NET verkrijgbaar in .NET Core, .NET Core is dus niet volledig 'backwards compatible' met .NET, waardoor het technisch gezien niet helemaal als een opvolger van .NET beschouwd kan worden. Voor bijna alle nieuwe toepassingen, met uitzondering degene die expliciet zwaar steunen op features uit .NET, is het aan te raden om van .NET Core gebruik te maken [106].

Hierin komt meteen een zwakte van de Google Compute Engine naar boven. Google voorziet namelijk kant-en-klare images voor de VM's can Compute Engine. Eén van die images is voor .NET, maar de meest recente versie ten tijde van dit onderzoek ondersteunt enkel .NET 4.5.2, en dus geen .NET Core. Het gebruik van .NET Core in de Google Compute Engine is hoogstwaarschijnlijk ook mogelijk, maar dit vergt een veel grotere inspanning. Vanwege de reeds zeer brede scope van dit onderzoek beperkt deze masterproef zich tot de .NET 4.5.2-versie in Google Compute Engine. Een alternatieve manier om .NET Core in het Google Cloud Platform te deployen kan het Google App Engine Flexible Environment zijn. Deze optie is voor dit onderzoek ook onderzocht, maar de plugins die nodig zijn hiervoor in Visual Studio leken niet goed te werken. De Google App engine plugin die verder in deze paragraaf wordt beschreven en die nodig is voor het deployen van een .NET Core applicatie naar het Flexible Environment van de Google App engine vanuit Visual Studio werd immers ten tijde van dit onderzoek enkel ondersteund in Visual Studio 2015, terwijl .NET Core enkel in Visual Studio 2017 goed wordt ondersteund. Op het moment dat deze thesis wordt geschreven is de Google Cloud Plugin net beschikbaar geworden voor Visual Studio 2017, waardoor .NET Core applicaties ondertussen wel zonder problemen vanuit Visual Studio kunnen ontwikkeld en gedeployed worden naar de Google App Engine Flexible Environment. Een alternatief is werken met de .NET Core SDK en de Google App Engine SDK vanuit command-line, net zoals in vorige paragraaf beschreven voor Java. Dit proces is beschreven op [107]. Voor deze masterproef is echter een andere optie gekozen die gebruik maakt van Google Compute Engine, en die wel makkelijk vanuit Visual Studio uit te voeren is. Bovendien is voor Java al gebruik gemaakt van de Google App Engine Flexible Environment. Het gebruik van Compute Engine voor C# laat dus toe een bredere waaier aan technologieën te dekken.

Azure is op dit gebied veel flexibeler en laat zonder problemen het gebruik van .NET Core toe. Om die reden is besloten om voor dit onderzoek zowel een implementatie in .NET als .NET Core te voorzien. Volgende paragrafen lichten elk van deze implementaties toe.

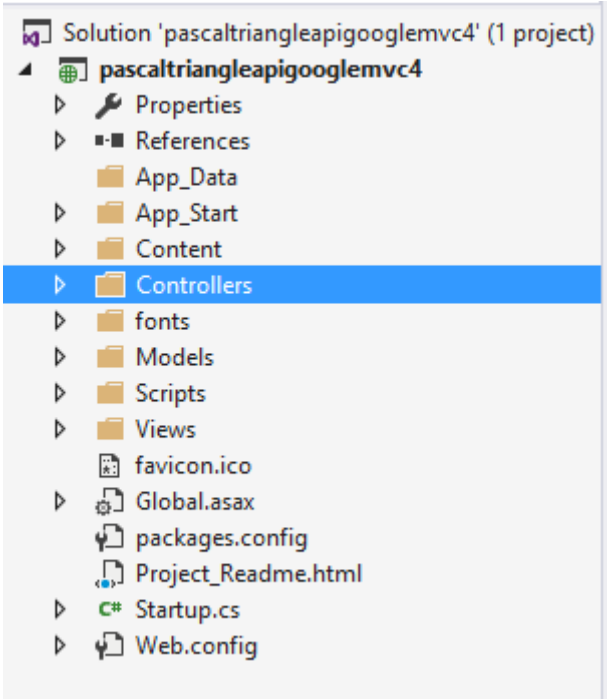
11.2.1.1 .NET 4.5

Deze paragraaf licht toe hoe het bestaande recursieve Driehoek van Pascal-project in C# kan worden omgevormd tot een RESTful web-API in .NET 4.5, voor gebruik in combinatie met de Google Compute Engine. Doordat google verplicht specifieke door hen ontwikkelde tools te gebruiken om het project aan te maken, is deze implementatie specifiek gericht op deployment in de Google Compute Engine. De grote lijnen zijn echter geldig voor elke .NET 4.5 webapplicaties en eender welk cloud-platform. Voor de implementatie is in dit onderzoek gebruik gemaakt van Visual Studio 2015. Sinds zeer recent is Visual Studio 2017 echter een even goede optie. Tijdens het voeren van dit onderzoek waren de hierboven vermelde tools nog niet beschikbaar voor Visual Studio 2017. Ten tijde van het schrijven van deze thesis is dit wel het geval. De werkwijze echter voor beide versies van Visual Studio is allicht minstens voor het grootste deel analoog. Bij andere ontwikkelingstools kan de werkwijze echter sterk afwijken, aangezien de hieronder beschreven werkwijze sterk steunt op het gebruik van de Google Cloud Tools voor Visual Studio.

De eerste stap voor het omzetten van de recursieve Driehoek van Pascal-applicatie naar een RSTful web-API voor deployment in de Google Compute Engine is het installeren van de hierboven reeds vermelde Cloud Tools voor Visual Studio-plugin. Dit kan rechtsreeks via het 'Extensions and Updates' venster in Visual Studio.

Eens de tools geïnstalleerd zijn moeten ze nog geconfigureerd worden. In het Tools menu in Visual Studio is een optie 'Google Cloud Tools' bijgekomen. Deze optie aanklikken opent een Google Cloud Explorer-venster. Hiermee kan een Google-account worden gelinkt aan Visual Studio.

Eens de nodige tools geïnstalleerd en geïnitialiseerd zijn kan een nieuw project worden aangemaakt in Visual Studio volgens een template die voorzien is door het Google Cloud Platform, speciaal om apps te ontwikkelen die naar Google Cloud Services gedeployed moeten worden. In de New Project dialoog binnen Visual Studio is een optie Google Cloud Platform bijgekomen bij de templates. De template gekozen voor dit project is de ASP.NET 4 MVC-template. Deze vormt de basis voor de Driehoek van Pascal-API. Deze template is zeer omvangrijk en bevat verschillende controllers en configuratiebestanden. Figuur 204 geeft de inhoud van het gegenereerde project weer.



Figuur 204: Structuur van het ASP.NET 4 MVC Template van Google Cloud Platform.

Er zijn een groot aantal mappen in dit project, waarvan de meeste al verschillende door Google gegenereerde bestanden bevatten. Hier in detail op in gaan zou te ver leiden voor dit onderzoek, en gelukkig moeten voor dit project enkel in de Controllers map en de Models map toevoegingen gedaan worden. Dit is veel eenvoudiger dan in Java, waar een groot aantal aanpassingen aan het project moesten gebeuren, en geen kant-en-klare template beschikbaar is.
Om te beginnen moeten in de Models map vanzelfsprekend alle Driehoek van Pascal-klassen die gemaakt zijn in C# geïmporteerd worden (zie hoofdstuk 7 en 9), zodat de API de verschillende versies van de Driehoek van Pascal kan genereren.
Verder is er een extra controller toegevoegd in de Controllers map, die de afhandeling van alle requests met de /pascaltriangle routing op zich neemt. In .NET is dit zeer eenvoudig. In de controllers map moet immers gewoon een klasse met de naam *PascalTriangleController* aangemaakt worden, die de *IController*-interface die ingebouwd zit in .NET implementeert. Deze interface bevat alle gewenste functionaliteit voor het afhandelen van de requests. Voor routing moeten geen extra stappen genomen worden. In .NET geldt immers de conventie dat requests met als routing pad /[controllernaam] rechtstreeks naar de controller worden geleid met die naam.
De *IController*-interface bevat een methode *Execute*, die een parameter van de klasse *RequestContext* krijgt. Deze methode wordt opgeroepen bij elke binnenkomende request met de juiste routing. De *RequestContext* bevat een property van het type *HttpContext*, wat op zich een property van de klasse *Response* bezit. Dit laatste object beschikt over een methode *Write*, waarmee eenvoudig een respons als string geschreven kan worden. De parameters van de request kunnen worden opgevraagd als een property van *requestContext.HttpContext.Request*. Nu rest enkel op een gelijkaardige manier als in Java aan de hand van de parameters van de

binnenkomende request de juiste Driehoek van Pascal te genereren, en de gegenereerde punten om te zetten naar JSON. Ook dit laatste is zeer eenvoudig in .NET. In het framework zit immers de klasse *JavaScriptSerializer* ingebouwd. De methode *Serialize* van deze klasse zet de punten van de gegenereerde driehoeken rechtstreeks om in JSON-strings. Nu zijn alle nodige gegevens beschikbaar om de request af te handelen. Zo kan eenvoudig en snel in .NET 4.5 een REST-API gemaakt worden. Dit alles in de praktijk omgezet levert Figuur 205.

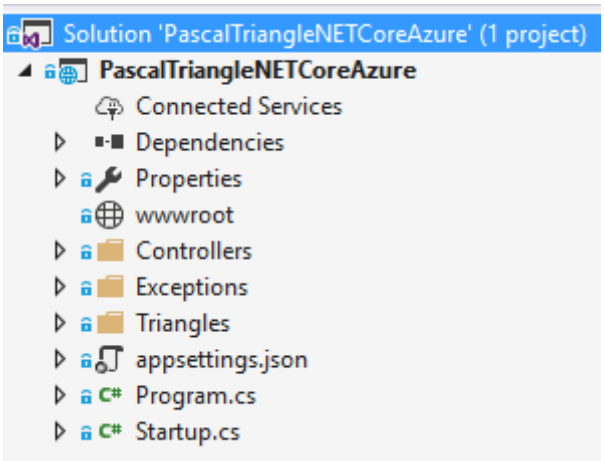
```
public void Execute(RequestContext requestContext)
{
    try
    {
        NameValueCollection parameters = requestContext.HttpContext.Request.Params;
        int size = Int32.Parse(parameters.Get("size"));
        String version = parameters.Get("version");
        String json;

        if (version.Equals("oo"))
        {
            int[][] points;
            switch (parameters.Get("type"))
            {
                case "sequential": points = new PascalTriangle(size).Points; break;
                case "parallelrows": points = new
                PascalTriangleRecParallelRows(size).Points; break;
                case "parallelcols": points = new
                PascalTriangleRecParallelCols(size).Points; break;
                case "parallelrowscols": points = new
                PascalTriangleRecParallelRowsCols(size).Points; break;
                default: throw new InvalidInputException();
            }
            json = new JavaScriptSerializer().Serialize(points);
        }
        else if (version.Equals("functional"))
        {
            List<List<int>> points;
            switch (parameters.Get("type"))
            {
                case "sequential": points = new
                FunctionalPascalTriangle(size).Points; break;
                case "parallelrows": points = new
                FunctionalPascalTriangleParallelRows(size).Points; break;
                case "parallelcols": points = new
                FunctionalPascalTriangleParallelCols(size).Points; break;
                case "parallelrowscols": points = new
                FunctionalPascalTriangleParallelRowsCols(size).Points; break;
                default: throw new InvalidInputException();
            }
            json = new JavaScriptSerializer().Serialize(points);
        }
        else throw new InvalidInputException();

        requestContext.HttpContext.Response.Write(json);
    }
    catch (InvalidInputException)
    {
        requestContext.HttpContext.Response.Write("Invalid input!");
    }
}
```

Figuur 205: Code voor het verwerken van een HttpRequest voor de Driehoek van Pascal-API in .NET 4.5.

Bovenstaande code is sterk gelijkend op de Java-implementatie uit Figuur 190. Merk opnieuw het gebruik van de custom *InvalidInputException* op.
Nu is de Driehoek van Pascal-API reeds klaar voor deployment in de Google Compute Engine. Het maken van de API zelf is dus een stuk sneller en evidenter dan in Java. Voor andere cloud-platformen kan gewoon vertrokken worden van een lege ASP.NET-applicatie, in plaats van bovenstaande template te gebruiken. Afhankelijk van het beoogde cloud-platform is het wel mogelijk dat extra configuratie noodzakelijk is.

11.2.1.2 .NET Core

Zoals eerder vermeld is naast de .NET 4.5-versie voor Google Compute Engine ook een .NET Core versie voorzien van de REST-API van de C# versie van de Driehoek van Pascal, met als doel deze te deployen in de Azure App Service. Dit proces is beduidend eenvoudiger en flexibeler dan dat voor de Google Compute Engine. Azure vereist immers niet het gebruik van speciale templates of tools. Volgende .NET Core-implementatie is dus ook geldig voor andere cloud-platformen of web-services in het algemeen. Deze implementatie liet ook probleemloos toe om gebruik te maken van Visual Studio 2017, wat dit onderzoek dan ook gedaan heeft.
De eerste stap in het omvormen van de C# versie van de Driehoek van Pascal tot een RESTful web-API is het aanmaken van een nieuw .NET Core-project. Dit kan binnen Visual Studio 2017 eenvoudig via de File > New > Project…-dialoog. Tussen de talrijke opties bevindt zich ASP.NET Core Web Application, wat voor dit onderzoek gekozen is. Er opent een nieuw venster om een template te kiezen. Hier is de gekozen optie de Web API-template. Merk op dat de Authentication-optie af moet staan. Visual Studio genereert nu een nieuw leeg project volgens deze template. Figuur 206 geeft de structuur weer van dit nieuw project, nadat de nodige toevoegingen zijn gebeurd voor het maken van de Driehoek van Pascal-API.



Figuur 206: Structuur .NET Core project voor de Driehoek van Pascal-API.

Het valt op dat dit project veel compacter is dan de .NET 4.5-template voor de Google Compute Engine. Merk op dat de Triangles-map is toegevoegd voor dit onderzoek. Alle verschillende Driehoek van Pascal-klassen zitten hierin. Ook de Exceptions map is toegevoegd met daarin de custom *InvalidInputException*, die in vorige paragrafen al

toegelicht is. Ook in dit project zijn verder maar weinig aanpassingen nodig om een volledig functionele Driehoek van Pascal-API te bekomen. Enkel in de Controllers-map moet zoals bij het project uit vorige paragraaf een *PascalTriangleController* worden aangemaakt. De code hiervan is wel sterk verschillend van het .NET 4.5 project, en is weergegeven in Figuur 207.

```
[Route("[controller]")]
    public class PascalTriangleController : Controller
    {
        [HttpGet("{version}/{type}/{size}")]
        public JsonResult Triangle(String version, String type, int size)
        {
            try
            {
                if (version.Equals("oo"))
                {
                    int[][] points;
                    switch (type)
                    {
                        case "sequential": points = new
                        PascalTriangle(size).Points; break;
                        case "parallelrows": points = new
                        PascalTriangleRecParallelRows(size).Points; break;
                        case "parallelcols": points = new
                        PascalTriangleRecParallelCols(size).Points; break;
                        case "parallelrowscols": points = new
                        PascalTriangleRecParallelRowsCols(size).Points; break;
                        default: throw new InvalidInputException();
                    }
                    return Json(points);
                }
                else if (version.Equals("functional"))
                {
                    List<List<int>> points;
                    switch (type)
                    {
                        case "sequential": points = new
                        FunctionalPascalTriangle(size).Points; break;
                        case "parallelrows": points = new
                        FunctionalPascalTriangleParallelRows(size).Points; break;
                        case "parallelcols": points = new
                        FunctionalPascalTriangleParallelCols(size).Points; break;
                        case "parallelrowscols": points = new
                        FunctionalPascalTriangleParallelRowsCols(size).Points; break;
                        default: throw new InvalidInputException();
                    }
                    return Json(points);
                }
                else throw new InvalidInputException();
            }
            catch (InvalidInputException)
            {
                return Json("Invalid input!");
            }
        }
    }
```

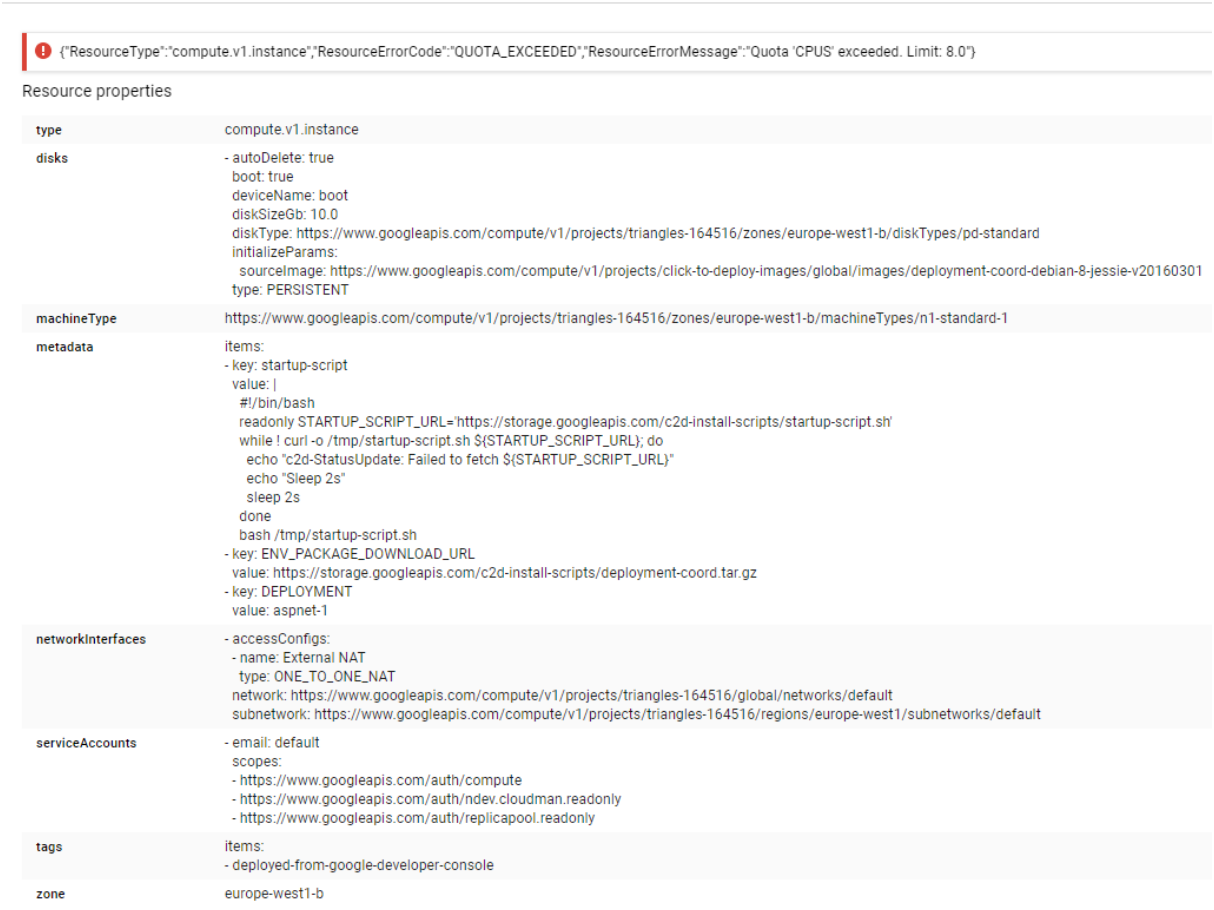
Figuur 207: PascalTriangleController-klasse in .NET Core.

Merk op dat bovenstaande code een weergave is van de volledige *PascalTriangleController*-klasse. De code in .NET Core is dus een stuk compacter, aangezien geen interfaces moeten geïmplementeerd worden. In .NET Core erft de *PascalTriangleController* immers van de *Controller*-klasse, waardoor niet alle methodes uit die klasse expliciet geïmplementeerd dienen te worden, wat bij de *IController* interface uit .NET 4.5 wel het geval was. Ook is de code eleganter. Routing kan rechtstreeks als een attribuut boven de gebruikte methodes en controller worden gedeclareerd. Hierdoor kan makkelijk een stijlvoller routing-patroon bekomen worden dan in .NET 4.5, en zijn geen requestparameters nodig, wat altijd stijlvoller is.
Ook kan rechtstreeks een methode worden gedefinieerd die kan worden aangesproken met een route, in tegenstelling tot .NET 4, waar een algemene Execute methode moest worden aangeroepen. De methode in .NET Core kan rechtstreeks JSON-strings teruggeven, wat het werken met JSON uiteraard nog eenvoudiger maakt. De code spreekt verder voor zich. Opnieuw is de *InvalidInputException* gebruikt.
Dit zijn alle aanpassingen om in .NET Core een RESTful Web-API te maken die klaar is om in de cloud gedeployed te worden. Volgende paragrafen beschrijven dit deployment-proces voor .NET 4 in Google Compute Engine en .NET Core in Azure in detail.

11.2.2 Google Compute Engine

Google Compute Engine is binnen het Google Cloud Platform één van de aangeraden opties om .NET-applicaties in te deployen. Dit proces is jammer genoeg nogal omslachtig en langdradig. Een uitgebreide quickstart-handleiding voor dit proces is te vinden op [108]. Hierin is ook het aanmaken van een ASP.NET 4 webapplicatie vervat. Deze paragraaf vertrekt vanaf de in paragraaf 11.2.1.1 aangemaakte REST-API voor de recursieve Driehoek van Pascal in .NET 4.5. Hieronder een overzicht van de belangrijkste stappen die moeten genomen worden om dit project in de Google Compute Engine te deployen.
De eerste stap is net zoals bij Java een nieuw project aanmaken in het Google Cloud Platform. Ook moet facturatie verplicht ingeschakeld zijn voor dit project, aangezien Google Compute Engine een betalende service is zonder gratis opties, in tegenstelling tot de App Engine (althans het Standard Environment hiervan).
Voor deze implementatie is gebruik gemaakt van Visual Studio 2015. Zoals hierboven beschreven is Visual Studio 2017 sinds zeer recent een even goede optie. Verder moet ook de Google Cloud SDK geïnstalleerd zijn. Paragraaf 11.1.3 beschreef dit niet zo heel evidente proces reeds in detail.
De volgende stap is het aanmaken van een VM in Google Compute Engine, onder het pas aangemaakte project. Google heeft dit proces voor de gebruikers gemakkelijk gemaakt. Op [109] is immers een kant-en-klare image te vinden voor een VM met het ASP.NET 4.5.2 framework op geïnstalleerd. De programmeur hoeft enkel op 'starten in compute engine' te klikken om deze Image te gebruiken. Hiermee wordt een nieuwe VM-instantie aangemaakt in Compute Engine. Allerlei eigenschappen kunnen worden geselecteerd voor de VM, zoals de locatie, die op Western Europe is ingesteld voor dit onderzoek. Het is belangrijk dat de opties 'Allow HTTP Traffic' en 'Allow WebDeploy Traffic' aangeduid zijn. Ook het aantal

CPU-kernen, geheugen, etc. van de VM kunnen worden ingesteld. Betalende accounts kunnen tot 64 cores kiezen voor de applicatie, terwijl de gratis proefversie waar in dit onderzoek van gebruik is gemaakt beperkt is tot 8. Vreemd genoeg wordt wel een foutmelding gegenereerd indien 8 kernen gekozen worden voor de VM. Figuur 208 geeft deze foutmelding weer.



Figuur 208: Gegenereerde foutmelding bij het aanmaken van een VM met 8 kernen in de gratis proefversie.

Deze foutmelding doet sterk denken aan de foutmelding uit Figuur 203 voor Java 8 in het Google App engine Flexible Environment. Dit is dus hoogstwaarschijnlijk een fout binnen Google. De enige manier om rond dit probleem te raken is slechts 4 CPU-kernen aangeven bij het aanmaken van de VM. Dit is op zich een jammere zaak.
Nadat de VM is aangemaakt, kunnen in de Google Cloud Platform Console in de browser allerlei gegevens van de VM gemonitord worden, zoals bijvoorbeeld CPU-gebruik. Het opstarten van de VM gaat snel en zonder verdere complicaties.
Het deployen zelf is dankzij de geïnstalleerde Cloud Tools voor Visual Studio, die beschreven zijn in paragraaf 11.2.1.1, eenvoudig. Om dit te doen moet gewoon de Google Cloud Explorer worden geopend. Hierin zou vanzelf het project te zien moeten zijn dat in het begin van deze paragraaf in het Google Cloud Platform is aangemaakt. Binnen dit project is normaal ook de aangemaakte VM in de Google Compute Engine zichtbaar. Alvorens gedeployed kan worden moeten eerst nog een set credentials worden aangemaakt. Dit kan door rechts te klikken op de weergave van de VM en de optie Manage Windows Credentials

te kiezen. Na een aantal stappen te doorlopen die nader zijn toegelicht in [108], kunnen de nieuw gegenereerde credentials worden opgeslagen.
Nu kan de API rechtstreeks vanuit Visual Studio naar de Compute Engine gedeployed worden. Hiervoor dient enkel met de rechtermuisknop geklikt te worden op het project in de Solution Explorer. Bij de opties verschijnt 'Publish to Google Cloud'. Door deze optie te kiezen verschijnt een dialoog waarin Compute Engine kan aangeduid worden als gewenste platform om de app op te deployen. Daarna rest enkel op publish te klikken en de .NET 4.5 versie van de Pascal Triangle API is gedeployed in Google Compute Engine.

11.2.3 Azure

Naast de Google Compute Engine is de Driehoek van Pascal-API ook gedeployed in Azure. Hiervoor is gebruik gemaakt van de .NET Core-versie van de recursieve Driehoek van Pascal, die beschreven is in paragraaf 11.2.1.2.
Om dit project naar Azure te deployen volstaat het met de rechtermuisknop te klikken op het project in de Solution Explorer, en dan de Publish optie te kiezen. Een venster verschijnt, met daarin de optie 'Microsoft Azure App Service'. Deze moet aangeduid worden, samen met 'create new'. Nu springt er een dialoogvenster open dat eerst vraagt om in te loggen. Merk dat na het activeren van de gratis trial van Azure, eerst opnieuw dient ingelogd te worden in Visual Studio. Dit kan door eerst uit te loggen ui Visual Studio, daarna Visual Studio te herstarten, en ten slotte opnieuw in te loggen. Dit is nodig omwille van een bekende caching-bug in Visual Studio die anders niet detecteert dat de gratis trial voor Azure is geactiveerd voor de gebruikte Microsoft-account -die uiteraard gelinkt is aan Visual Studio- en dus geen toegang tot Azure zal verschaffen.
Eens bovenstaand relog-proces is doorlopen, is de gratis trial-subscriptie voor Azure selecteerbaar. Daarnaast is het ook belangrijk een naam voor de applicatie op te geven. Vervolgens dient er ook een zogenaamde resource group en App Service plan aangemaakt worden. De resource group is slechts een logische opdeling van services voor resource-management. Het App Service Plan is belangrijker. Dit is de groep van fysieke resources waarop de applicatie uiteindelijk terecht zal komen in de cloud. Meerdere apps kunnen hetzelfde service plan delen. Hiermee kan de regio van de app worden gekozen, in dit geval West-Europa.
Ook moet de grootte van het App Service Plan worden gedefinieerd, uitgedrukt in de vorm van een betalingsplan. Voor de gratis trial is enkel de Free optie mogelijk. Deze bepaalt allerlei instellingen naar schaalbaarheid en rekenkracht van de resources toe. In Azure is het maximale aantal CPU-kernen voor een App Service Plan steeds beperkt tot 4. Voor de gratis account maakt dit geen verschil met Google, maar betalende accounts kunnen in het Google Cloud Platform wel beduidend meer resources aan hun applicaties toewijzen.
Nu rest enkel de app een unieke naam te geven en te publishen naar Azure. Dit kan door het overeenkomstige veld in te vullen en in het dialoogvenster op 'Create' te klikken. Dit deployt de .NET Core applicatie naar Azure. Na enkele minuten kan de app bezocht worden via de url [appnaam].azurewebsites.net. Ook biedt Azure de mogelijkheid om via het Portal in de browser allerlei instellingen van de applicatie aan te passen, en het gebruik ervan te

monitoren. De belangrijkste optie voor dit onderzoek is Scaling. Deze is op autoscaling gezet voor dit onderzoek.
Het hele Deployment-proces in Azure is veel korter, soepeler en logischer dan eender welke van de in de vorige paragrafen beschreven werkwijzen voor het Google Cloud Platform. Azure is dus zeer eenvoudig en intuïtief te gebruiken, en bovendien flexibel, wat vooral naar nieuwe gebruikers toe een groot voordeel is tegenover het Google Cloud Platform.

11.3 F#

Ook voor F# werd het Driehoek van Pascal-programma omgevormd tot een RESTful web-API en in de cloud gedeployed. Voor F# is voor Azure gekozen als cloud-platform. Dit omdat F# draait binnen het .NET framework, en dus Azure als de meest geschikte cloud-omgeving als eerste in de gedachten springt. Net als C# is ook de F#-versie van de API gedeployed in het PaaS-platform van Azure, namelijk de Azure App Service. Deze paragraaf licht eerst het omvormen van de desktop-versie van de recursieve Driehoek van Pascal naar een RESTful web-API toe. Daarna zijn de stappen beschreven die nodig zijn om deze API in Azure te deployen.

11.3.1 REST-API

Voordat de F#-applicatie naar Azure kan gedeployed worden, moet ze uiteraard eerst worden omgevormd tot een RESTful Web-API. Zeer recent zijn er voor Visual Studio 2017 een aantal tools uitgekomen die het ontwikkelen in F# veel aangenamer maken [110], en dus wordt Visual Studio 2017 gebruikt voor de ontwikkeling van de F# Web-API. De werkwijze toegelicht in deze paragraaf is op zijn minst gedeeltelijk overdraagbaar naar andere ontwikkelingsomgevingen, maar bepaalde details zullen ongetwijfeld afwijken.
Er bestaat een zeer compacte bibliotheek die in staat is van eender welke F# console-applicatie in een paar regels code een volledig functionele web-API te maken, namelijk Suave. Deze bibliotheek kan met de NuGet Package Manager eenvoudig worden toegevoegd aan het bestaande F#-project. Voor F# moet er dus in tegenstelling tot C# en Java geen volledig nieuw project worden aangemaakt, en ook niet gewerkt worden met een groot framework met allerlei opties die voor dit onderzoek niet van belang zijn. Dit is uiteraard een groot voordeel. De enige opmerking hierbij is dat het belangrijk is dat het F#-project is aangemaakt als een .NET 4.6 project. Anders werkt de Suave-bibliotheek niet naar behoren. Er moeten geen nieuwe mappen of bestanden worden toegevoegd aan het project. Voor de eenvoud is zelfs alle code van heel het F#-project verplaatst naar het hoofdbestand van het project, waarin de main-methode huist, namelijk program.fs. De hele Driehoek van Pascal-REST-API is dus een enkel bestand van nog geen 70 regels code. Figuur 209 geeft alle code weer die nodig is om van de bestaande F#-applicatie een RESTful Web-API te maken. De code die nodig is voor het genereren van de verschillende versies van de Driehoek van Pascal in F# is voor de overzichtelijkheid weggelaten, aangezien die code al uitgebreid is beschreven in vorige hoofdstukken.

```
let triangle t=
    request (fun r ->
        match r.queryParam "size" with
        | Choice1Of2 size ->
                OK (Json.serialize (t (size|>int))|> Json.formatWith
Chiron.Formatting.JsonFormattingOptions.Pretty)
        | Choice2Of2 msg -> BAD_REQUEST msg)

let routes =
    GET >=> choose
        [path "/pascaltriangle/sequential" >=> triangle pascalTriangle
         path "/pascaltriangle/parallelrows" >=> triangle parallelPascalTriangleRows
         path "/pascaltriangle/parallelcols" >=> triangle parallelPascalTriangleCols
         path "/pascaltriangle/parallelrowscols" >=> triangle
         parallelPascalTriangleRowsCols]

[<EntryPoint>]
let main args =
    startWebServer defaultConfig routes
    0
```

Figuur 209: Code voor het maken van een REST-API van het Driehoek van Pascal-programma in F#.

Deze code is duidelijk zeer compact en efficiënt. Best kan de code van onder naar boven geïnterpreteerd worden, dus beginnende bij de main-methode. Het enige wat deze methode bevat is een enkele regel code die een webserver start met een standaardconfiguratie, en met de gespecifieerde routes. *StartWebServer* en *defaultConfig* zijn allebei deel van de Suave-bibliotheek. De routes zijn net boven de main-methode gedefinieerd. Hier is duidelijk te zien dat bepaalde paden gemapt zijn naar de functie *pascalTriangle* met als argument steeds één van de Driehoek van Pascal-functies uit de vorige hoofdstukken. Dit is een mooi voorbeeld van de stijl van F#. De pascaldriehoek-functies zelf zijn rechtstreeks als argument meegegeven met de *triangle*-functie.
De *triangle*-functie neemt de afhandeling van de requests op zich. Om deze functie om te vormen tot een functie die requests afhandelt volstaat het om een request-object terug te geven. Dit request object heeft als parameter een functie. Deze functie heeft dan weer als parameter de binnenkomende request. Suave herkent deze structuur en zal voor r de binnenkomende request injecteren. Met de Suave-functie *queryParam* kan eenvoudig een parameter van de request worden opgevraagd. In dit geval is dit de parameter *size*. De functie *queryParam* geeft een optie met 2 keuzes terug: één voor als de parameter bestaat, en één voor wanneer dit niet het geval is. In het laatste geval geeft de *triangle*-functie de http-response 'Bad Request', ook wel code 400 genoemd, terug. Indien *size* echter wel een geldige parameter is, geeft de functie *triangle* een OK-respons terug, of code 200. Dit gebeurt door het aanroepen van de in Suave aanwezige OK-functie. Deze functie neemt als argument de gegenereerde respons. In dit geval zijn dit de punten van de door de routes functie gekozen Driehoek van Pascal-functie met grootte *size*. De punten moeten nog wel naar JSON omgezet worden. Dit kan syntactisch gezien eenvoudig met de Chiron-bibliotheek voor F#. Deze kan in Visual Studio 2017 aan het project worden toegevoegd met behulp van de ingebouwde NuGet package manager. In de Chiron-bibliotheek zitten de nodige functies om JSON-strings te genereren van een lijst met integers in F#. Hoewel dit eenvoudig lijkt, is dit de grootste moeilijkheid gebleken voor het omzetten van de Driehoek van Pascal in een REST-

API in F#. Er was heel wat zoekwerk nodig om een werkende bibliotheek te vinden met de nodige functionaliteit. Documentatie en codevoorbeelden van de Chiron-bibliotheek zijn ook zeer schaars en moeilijk te vinden. Dit laat zien dat F# nog een taal in ontwikkeling is. Dit soort ongemakkelijkheden zal in de toekomst allicht verbeteren, als F# aan populariteit wint. Desalniettemin is de implementatie van een RESTful API in F# zoals hierboven beschreven veel compacter en eleganter dan in de andere onderzochte talen. Dit op zich is een groot voordeel voor F#.

11.3.2 Azure

Nu de F# applicatie is omgezet naar een RESTful web-API met behulp van de Suave-bibliotheek, rest enkel nog de nodige aanpassingen te doen om het project op Azure te deployen. Deze aanpassingen zijn opvallend beperkt in omvang en vrij goed gedocumenteerd, zoals in [111]. Desondanks is voor dit onderzoek gebleken dat de documentatie van F# gespreid is over verschillende websites, en meestal lijken de beschikbare handleidingen een grondige voorkennis van (functioneel) programmeren te verwachten. Dit is een goede illustratie van het feit dat F# tegenwoordig nog een niche-taal is, met een sterk gespecialiseerde gebruikersgroep. In de toekomst zal ondersteuning voor F# waarschijnlijk echter sterk verbeteren, indien de taal aan populariteit blijft winnen.
In [111] wordt volgende benadering voor het deployen van de REST-API in F# in de Azure App Service voorgesteld: De Azure App Service is een PaaS cloud-platform waarop IIS draait om .NET aplicaties te hosten. IIS is het programma dat Windows gebruikt om web-verkeer van en naar .NET-applicaties te beheren. Omdat Suave dit web-verkeer voor de F#-applicatie regelt, moet ervoor gezorgd worden dat IIS het web-verkeer rechtstreeks aan Suave kan doorgeven. Hiervoor kan een web.config bestand toegevoegd worden aan het F#-project. Figuur 210 geeft de inhoud van dit web.config bestand weer.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <remove name="httpplatformhandler" />
      <add name="httpplatformhandler"
      path="*"
      verb="*"
      modules="httpPlatformHandler"
      resourceType="Unspecified"/>
    </handlers>
    <httpPlatform stdoutLogEnabled="true"
    stdoutLogFile=".\suave.log"
    startupTimeLimit="20"
    processPath="%HOME%\site\wwwroot\FSTrianglesAzure.exe"
    arguments="%HTTP_PLATFORM_PORT%"/>
  </system.webServer>
</configuration>
```

Figuur 210: Web.config-file om het F# project in Azure te kunnen deployen.

Om te beginnen wordt een nieuwe *httphandler* aangemaakt. Daarna stelt de code in het bestand een *httpPlatform* object in voor deze handler. Dit object bevat enerzijds specificaties voor een logging-file die niet echt van belang is voor dit onderzoek, en anderzijds de applicatie die gestart dient te worden aan de hand van het *processPath*-attribuut. Dit attribuut is voor dit project ingesteld op de executable van het F# project. Het *httpPlatform* -object specifieert ook dat bij het aanroepen van deze executable de omgevingsvariabele %HTTP_PLATFORM_PORT% als argument dient meegegeven te worden. Dit is de interne poort waarlangs IIS al het web-verkeer voor deze applicatie stuurt. De main-methode van de F#-applicatie moet deze poort dus opvangen en koppelen aan de poort waar Suave op luistert. Hiervoor moet de main-methode in het F#-project herschreven worden. Figuur 211 toont deze aangepaste main-methode.

```
[<EntryPoint>]
let main [| port |] =
    let config =
        { defaultConfig with
              bindings = [ HttpBinding.create HTTP IPAddress.Loopback (uint16 port) ]
              listenTimeout = TimeSpan.FromMilliseconds 3000. }
    startWebServer config routes
    0
```

Figuur 211: Aangepaste main-methode in de F# versie van het Driehoek van Pascal-API-project om op Azure gedeployed te kunnen worden.

De main-methode vangt nu het argument *port* op. Deze methode start nog steeds een webserver zoals in Figuur 209, maar deze keer met een licht aangepaste versie van *defaultConfig*, zodat de Suave-webserver luistert op de poort die door IIS is meegegeven. Dit is de enige aanpassing die in het project zelf dient te gebeuren. De laatste stap is uiteraard het project te builden voor een Release. In de /Obj/Release map van het F#-project is nu een lijst met bestanden bijgekomen, waaronder de executable van de API. Deze bestanden zijn later van belang.
Nu is het project klaar om naar Azure gedeployed te worden. Dit is het eenvoudigst door te werken met FTP (File Transfer Protocol). Hiervoor zijn echter eerst een aantal configuratiestappen vereist. Eerst en vooral is het belangrijk een FTP client-programma te installeren op de lokale pc. Voor dit onderzoek is van FileZilla gebruik gemaakt. Eens FileZilla geïnstalleerd is moet op Azure een nieuwe applicatie worden aangemaakt voor de F#-versie van de Driehoek van Pascal-API. Dit kan binnen hetzelfde App Service Plan dat eerder aangemaakt is voor de C#-versie. Eens dit gebeurd is, is het vereist credentials aan te maken voor FTP-communicatie tussen FileZilla en het pas aangemaakte project op de Azure App Service-server. Dit kan via het Azure Portal in de browser, zoals beschreven in [112]. Ook de FTP-hostname waarmee verbinding dient gemaakt te worden in de FTP-client is terug te vinden in het Azure Portal voor de applicatie.
Eens al deze stappen doorlopen zijn kan in de lokale FTP-client de pas aangemaakte Azure-applicatie worden toegevoegd als een nieuwe FTP-host, uiteraard gebruikmakend van de gegevens die net zijn gegenereerd in het Azure Portal.

Nu is het mogelijk verbinding te maken met de root directory van de applicatie in Azure. Nu rest enkel het web.config bestand uit Figuur 210 toe te voegen aan deze root-directory, en in de site\wwwroot map in Azure alle hierboven gegenereerde build-bestanden te kopiëren via FTP. Het project op Azure bestaat nu uit een root map met daarin het web.config bestand en de map 'site'. Deze laatste map bevat de map 'wwwroot', waar alle build-bestanden van de RESTful web-API in F# inzitten, waaronder uiteraard de executable van de Driehoek van Pascal-API zelf. Eens de bestanden geüpload zijn is binnen enkele minuten via de url [applicatienaam].azurewebservices.net het project toegankelijk. Op die manier is de F#-versie van de Driehoek van Pascal in de Azure cloud gedeployed.

11.4 Wolfram

Wolfram is de laatste taal die voor dit onderzoek geïntegreerd is in de cloud. Het cloud-platform dat hiervoor gekozen is, is vanzelfsprekend de Wolfram cloud, aangezien dit platform zeer goed ondersteund is binnen Wolfram zelf, zoals beschreven in paragraaf 5.2.7. Deze integratie is dan ook zeer eenvoudig gebleken. Het is mogelijk om aan de hand van een handvol functies eender welke expressie in de cloud te deployen in slechts enkele minuten. Onderstaande code in Figuur 212 laat dit principe zien, toegepast op het recursieve Driehoek van Pascal-algoritme.

```
pascalTriangleDecoder[type_String, size_Integer] := Which[
type == "sequential", pascalTriangleRec[size],
type == "parallelrows", pascalTriangleRecParallelRows[size],
type == "parallelcols", pascalTriangleRecParallelCols[size],
type == "parallelrowscols", pascalTriangleRecParallelRowsCols[size],
True, Throw["Invalid input!", invalidInputException]]

CloudDeploy[APIFunction[{"type" → "String", "size" → "Integer"},
ExportForm[Catch[pascalTriangleDecoder[#type, #size], invalidInputException],"JSON"] &],
Permissions -> "Public"]
```

Figuur 212: Code voor de Driehoek van Pascal API in Wolfram

De *pascalTriangleDecoder*-functie werkt als een eenvoudige switch statement om het type van Driehoek van Pascal te bepalen. Verder is er opnieuw basic exception handling toegevoegd. Om van de Wolfram-expressie die wordt teruggegeven door de *pascalTriangleDecoder*-functie een cloud-based REST API te maken, volstaat het drie eenvoudige stappen te doorlopen. Allereerst moet ervoor gezorgd worden dat de Wolfram expressie JSON teruggeeft in plaats van geneste Wolfram-lijsten. Vervolgens moet de *pascalTriangleDecoder*-expressie omgevormd worden tot een RESTful web-API. Ten slotte moet deze API in de cloud gedeployed worden. Voor al deze zaken is een kant-en-klare functie beschikbaar in Wolfram. Bovenstaande code is verder dan ook zelfverklarend. Deze code geeft een *CloudObject* terug, zoals in Figuur 213 te zien. De gegenereerde API kan dan rechtstreeks worden aangeroepen in de Wolfram Cloud aan de hand van de URL die de *CloudObject*-expressie bevat.

```
CloudObject[https://www.wolframcloud.com/objects/77e96b98-fdf0-4dae-8562-0da371206a30]
```

Figuur 213: CloudObject dat gebruikt kan worden om de url van de gegenereerde cloud-API op de vragen.

Het is belangrijk om op te merken dat al de gebruikte functies in bovenstaande code rechtstreeks in Wolfram ingebouwd zijn. Er is dan ook geen behoefte om gebruik te maken van extra bibliotheken, frameworks, of iets dergelijks. Het valt op dat de Wolfram-code ook hier bijzonder compact is. Ook moet er niks veranderd worden aan de bestaande Wolfram-code voor de Driehoek van Pascal. Bovenstaande code kan gewoon in eender welk ander Wolfram-project worden toegevoegd, zonder de structuur of werking van de andere code in het project te verstoren. Dit illustreert opnieuw de grote flexibiliteit en het nog grotere gebruiksgemak van Wolfram. Dit is een groot pluspunt ten opzichte van de andere bestudeerde programmeertalen.

11.5 Besluit

In dit hoofdstuk zijn een heel aantal programmeertalen geïntegreerd in een heel aantal cloud-omgevingen, gaande van IaaS tot PaaS-platformen. De wijze waarop dit dient te gebeuren is opvallend uiteenlopend, zelfs binnen hetzelfde cloud-platform.
Java is van alle geteste talen het moeilijkst om te zetten naar een REST-API. Er zijn een heel aantal aanpassingen nodig aan de code, en deze moeten op diverse plaatsen gebeuren. Daarbovenop is het niet evident om de Java-API in de Google App Engine te deployen. Java is dus geen aantrekkelijke taal voor cloud computing vanuit een implementatie-standpunt, en al zeker niet in combinatie met de Google App Engine.
C#-code laat zich eenvoudiger omvormen tot een REST-API. De .NET en .NET Core frameworks zijn opvallend veel eenvoudiger te gebruiken. Ook het deployen van C# naar de cloud is relatief eenvoudig in vergelijking met Java. Voor het Google Cloud Platform is dit proces wel beduidend omslachtiger dan voor Azure. Dit laatste platform is bijzonder snel, eenvoudig, en aangenaam om te gebruiken in combinatie met C#; zeker omdat het standaard geïntegreerd is in Visual Studio.
De functionele implementaties binnen object-georiënteerde talen kunnen in dit hoofdstuk niet apart besproken worden aangezien ze gebonden zijn aan de object-georiënteerde taal waartoe ze behoren, en dus ook aan de frameworks die ze gebruiken. Alle opmerkingen voor object-georiënteerde talen zijn dan ook van toepassing op de functionele implementaties binnen deze talen. De enige opmerking hierbij is dat voor de functionele implementaties in het geval van Java iets minder ondersteuning is in de cloud. Dit is echter meer te wijten aan het geteste cloud-platform dan aan de functionele implementatie op zich.
De hoofdzakelijke functionele talen zijn opvallend veel eenvoudiger om te vormen tot REST-API's dan hun object-georiënteerde tegenhangers. De code is veel compacter en overzichtelijker dan bij de hoofdzakelijk object-georiënteerde talen. Hierin komt opnieuw de declaratieve natuur van deze talen naar boven.
Het proces om F# in de cloud te deployen is wel iets langdradiger dan bij C#, maar nog steeds eenvoudiger dan de geteste Java-varianten. Wel viel op dat online handleidingen voor zowel het maken van een REST-API als het deployen ervan naar Azure veel schaarser zijn

dan bij de andere talen. Deze handleidingen veronderstellen ook allemaal dat de programmeur reeds een grondige voorkennis heeft van hetgeen hij wilt bereiken. F# is dan ook zeer elegant en fijn om te gebruiken, zolang de gebruiker zelf bekwaam is. Nieuwe gebruikers met weinig programmeerervaring zullen ongetwijfeld veel last hebben met het toepassen van deze taal.
Van alle implementaties blinkt Wolfram uit als zijnde de meest compacte en eenvoudig te gebruiken taal voor zowel het omvormen van de recursieve Driehoek van Pascal tot een REST-API, als het deployen ervan in de cloud. De code is zeer compact, leesbaar, intuïtief, flexibel en bovendien goed gedocumenteerd. Wolfram is wat implementatie betreft dan ook veruit de aangenaamste taal voor cloud computing.

12 Cloud Resultaten

Dit hoofdstuk bespreekt in detail de resultaten van het uitvoeren van de tests beschreven in paragraaf 6.3, toegepast op de cloud-implementaties van het recursief Driehoek van Pascal-algoritme in de verschillende voor dit onderzoek bestudeerde talen, beschreven in het vorige hoofdstuk, en is dus de essentie van deze masterproef.
De bevindingen uit hoofdstuk 8 en hoofdstuk 9 komen hierbij zeer goed van pas, aangezien nu een volledig transparante vergelijking kan ontstaan tussen de verschillende talen en cloud-platformen.
In dit hoofdstuk is de vergelijking tussen de verschillende cloud-implementaties opnieuw kwalitatief van aard. Dit neemt niet weg dat voor sommige tests accurate kwantitatieve resultaten opgemeten zijn. De interpretatie van deze resultaten gebeurt echter steeds op een genuanceerde, kwalitatieve manier, zoals in alle voorgaande hoofdstukken.
In eerste instantie bespreekt dit hoofdstuk de resultaten voor elke taal in detail. Daarna is een algemene paragraaf voorzien die alle resultaten van de verschillende talen naast elkaar zet en met elkaar vergelijkt. Zo ontstaan betere inzichten over de relatieve verhoudingen van de kwaliteit van de resultaten over verschillende talen heen. Ten slotte geeft dit hoofdstuk een algemene conclusie over het gebruik van de verschillende geteste talen in de cloud. Als afsluiter bakent dit hoofdstuk voor elke in dit onderzoek geteste combinatie van programmeertaal en cloud-platform af in welke situaties deze combinatie het best tot zijn recht komt.

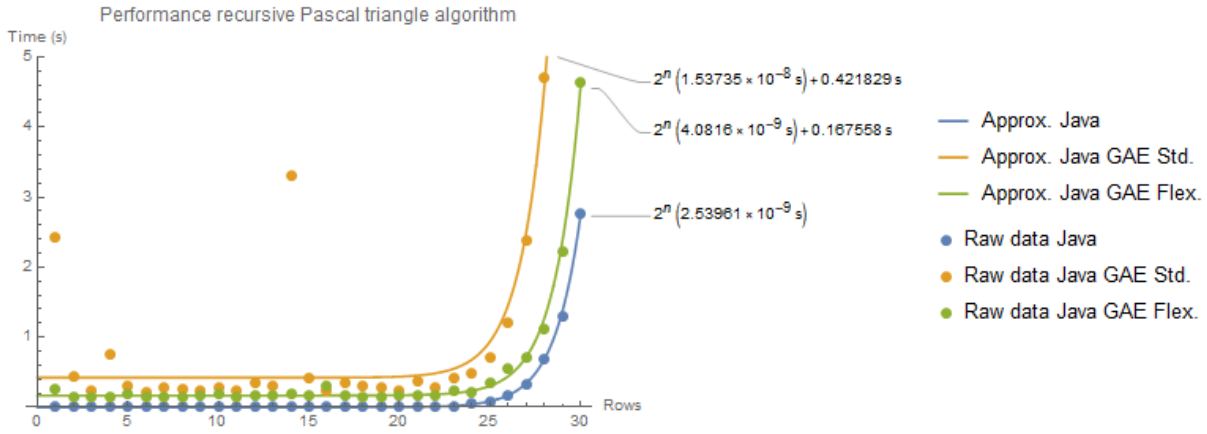
12.1 Java

Zoals in alle voorgaande hoofdstukken is ook hier Java de eerste taal die besproken wordt. Paragraaf 11.1 beschreef reeds dat deze taal zowel in het Standard als het Flexible Environment van de Google App Engine is geïmplementeerd voor dit onderzoek. Deze paragraaf beschrijft de testresultaten voor deze beide omgevingen in detail.

12.1.1 Connectiesnelheid en schaalbaarheid

Zowel de performantie als de connectiesnelheid voor beide implementaties van de Java-versie van de recursieve Driehoek van Pascal-API in het Google Cloud Platform zijn eenvoudig gelijktijdig te testen. De resultaten van deze test zijn dan ook in eenzelfde figuur weergegeven. Figuur 214 geeft deze resultaten weer voor één iteratie.

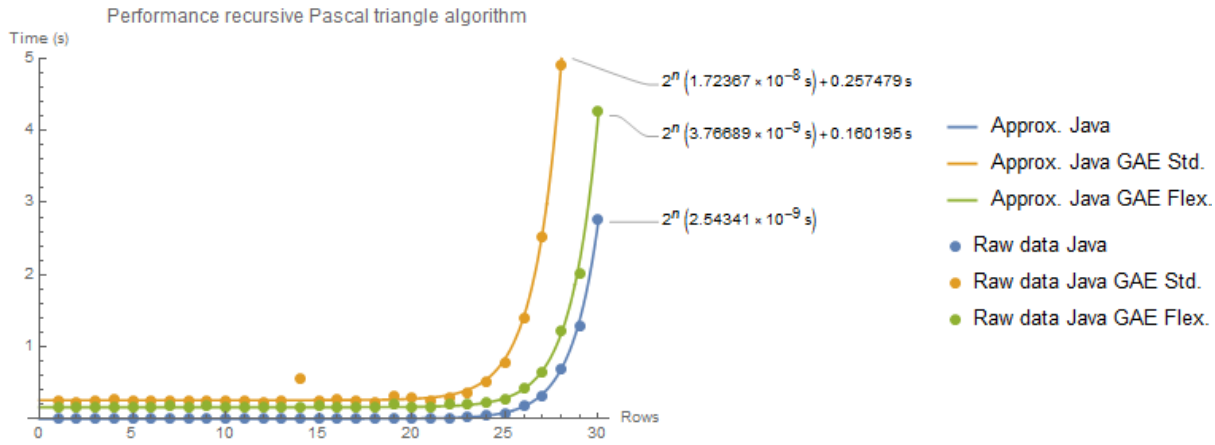
```
testPerformance[{"Java", "Java GAE Std.", "Java GAE Flex."}, 1, 5]
```



Figuur 214: Performantie Java in GAE Standard en Flexible Environment voor één iteratie.

Het valt meteen op dat er nogal wat uitschieters aanwezig zijn in dit testresultaat, vooral voor het Standard Environment. Voor het Flexible Environment lijken de uitschieters minder sterk, maar aangezien de resultaten zo klein zijn voor deze implementatie is de eigenlijke grootte moeilijk zichtbaar. Relatief ten opzichte van het gemiddelde kunnen ze nog steeds 50% bedragen. Deze uitschieters zijn allicht te wijten aan allerlei oorzaken die te maken hebben met de internetverbinding tussen de test-pc en de aangesproken webserver waar de API zich op bevindt. Het is normaal dat hier af en toe vrij grote vertragingen bij komen kijken. De uitschieters mogen echter niet zomaar buiten beschouwing gelaten worden, aangezien Figuur 214 duidelijk laat zien dat er een zeker verband is tussen de grootte en frequentie van de uitschieters en het gebruikte cloud-platform. Daarom is het bij deze tests in het bijzonder belangrijk om een gemiddelde te nemen over meerdere iteraties. Aan dit gemiddelde is nog steeds te zien in welke mate er uitschieters aanwezig zijn in de testdata, maar de impact van deze uitschieters op het algemeen resultaat is op die manier beperkt. Voor al de komende tests in dit hoofdstuk is opnieuw 10 iteraties gekozen, omdat anders de tijd nodig voor het uitvoeren van de verschillende tests te hoog zou oplopen.
Het opnieuw uitvoeren van bovenstaande test, maar dan voor 10 iteraties levert Figuur 215.

```
testPerformance[{"Java", "Java GAE Std.", "Java GAE Flex."}, 10, 5]
```



Figuur 215: Performantie Java in GAE Standard en Flexible Environment voor 10 iteraties.

Uit Figuur 215 zijn een heel aantal interessante resultaten af te leiden. Wat betreft de connectiesnelheid valt op dat het Flexible Environment het goed doet met een kleine 0,2 seconden. Het Standard Environment doet het iets minder goed. 0,3 seconden is echter nog steeds acceptabel. Ook valt opnieuw op dat er vrij veel uitschieters aanwezig zijn bij het Standard Environment. Deze omgeving is dus niet alleen gemiddeld iets trager, maar heeft ook een grotere variatie op de meetresultaten en is daardoor ook minder consistent.
Naar performantie toe is er echter een veel groter verschil merkbaar tussen de verschillende geteste omgevingen. Tabel 13 geeft de exacte verhoudingen weer.

Tabel 13: Vergelijking relatieve performantie verschillende cloud-omgevingen voor Java.

Cloud-omgeving	**Relatieve performantie t.o.v. desktop-versie**
Google App Engine Standard Environment	$\frac{1{,}7 \cdot 10^{-8}}{2{,}5 \cdot 10^{-9}} \approx 7$
Google App Engine Flexible Environment	$\frac{3{,}8 \cdot 10^{-9}}{2{,}5 \cdot 10^{-9}} \approx 1{,}5$

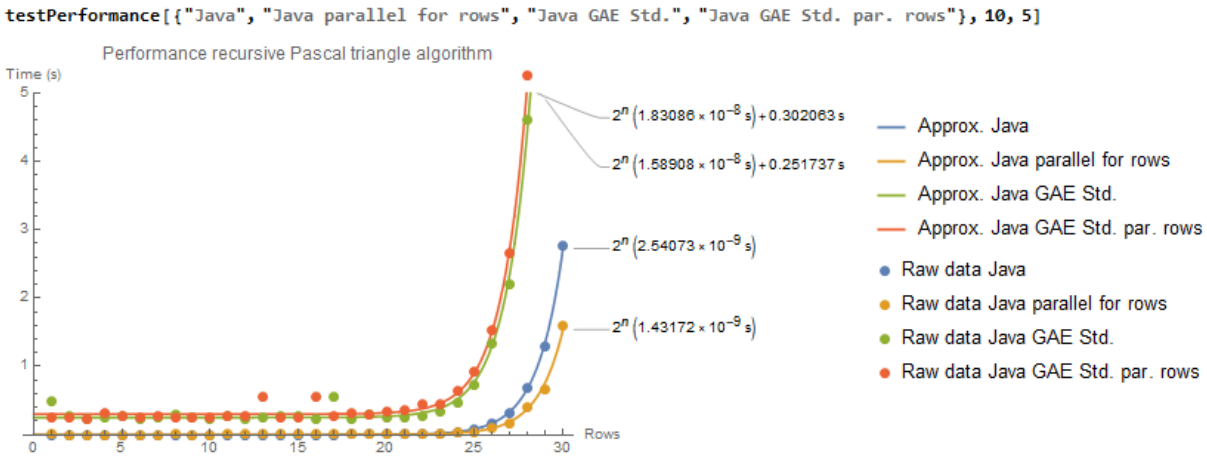
Het Standard Environment van de Google App Engine is wel 7 keer trager dan de desktop-versie. Dit is vrij onacceptabel voor toepassingen die veel rekenkracht vragen. Het Flexible Environment scoort op dit gebied veel beter. Het is slechts 50% trager dan de test-pc, wat behoorlijk goed is aangezien de test-pc een hoge single-threaded performantie heeft. Het Flexible Environment is dus duidelijk beter geschikt voor rekenintensieve applicaties in Java dan het Standard Environment. Dit laatste dient zelfs vermeden te worden voor toepassingen die expliciet veel rekenkracht vereisen.

12.1.2 Parallellisatie

Het derde belangrijke te bestuderen aspect van de cloud-implementaties is parallellisatie. Deze paragraaf bespreekt de resultaten hiervan voor Java. Het Standard en Flexible Environment van de Google App Engine worden elk afzonderlijk besproken.

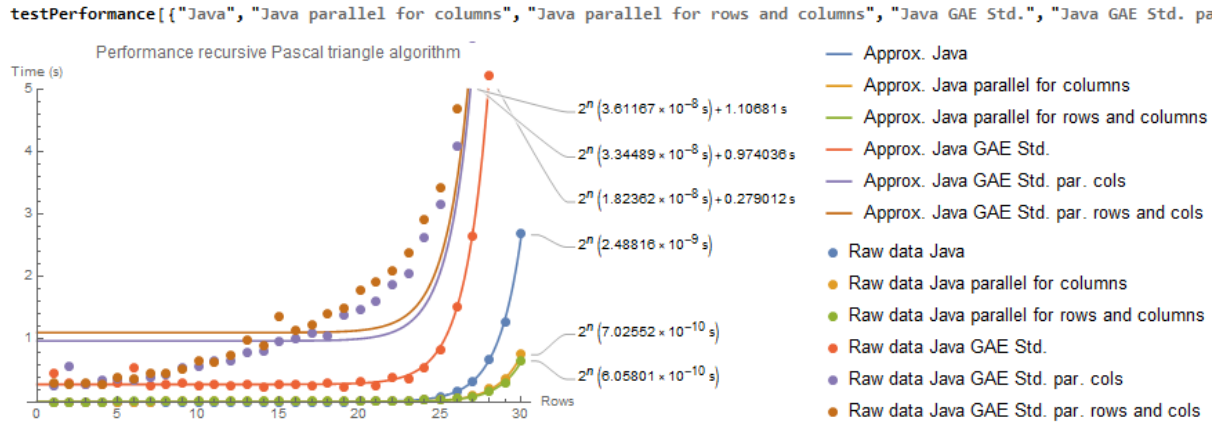
12.1.2.1 Google App engine Standard Environment

Omdat parallellisatie in het Standard Environment van de Google App Engine op een aangepaste manier moet gebeuren in vergelijking met de desktop-versie van de testapplicatie, zoals beschreven in paragraaf 11.1.2, loont het de moeite om eerst de parallellisatiemethode voor enkel rijen onder de loep te nemen voor deze implementatie en te vergelijken met de desktop-versie. Op deze manier is te bepalen of bij het gebruik van de nieuwe parallellisatiemethode extra overhead komt kijken. Figuur 216 geeft de resultaten van deze test weer.



Figuur 216: Vergelijking parallellisatiemethode voor enkel rijen in het Standard Environment van Google App Engine met de desktop-variant.

De resultaten uit Figuur 216 zijn opmerkelijk. De geparallelliseerde variant blijkt in het Standard Environment van Google App Engine trager te zijn dan de niet-geparallelliseerde versie. Dit gaat uiteraard lijnrecht in tegen één van de hoofddoelen van parallellisatie in het algemeen. Ter verificatie van deze vreemde resultaten zijn de methodes voor geparallelliseerde kolommen en rijen en kolommen ook getest. Figuur 217 geeft hiervan de resultaten weer.

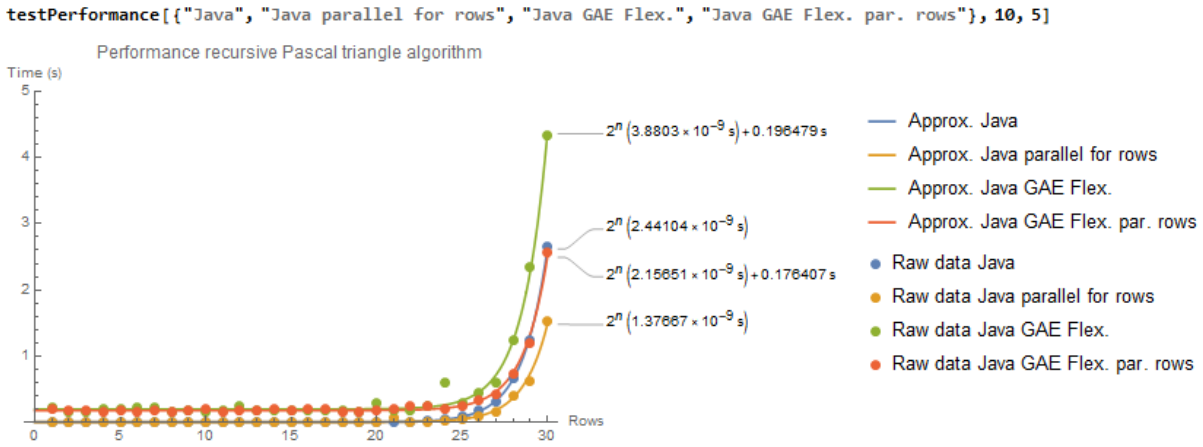


Figuur 217: Vergelijking parallellisatiemethode voor enkel kolommen en voor rijen en kolommen in het Standard Environment van Google App Engine met de desktop-variant.

De andere parallellisatiemethoden laten in het Standard Environment van de Google App engine nog triestigere resultaten zien. De performantie ligt veel lager dan die van de niet-geparallelliseerde versie. Bovendien lijkt de tijdscomplexiteit van het recursieve Driehoek van Pascal-algoritme bij deze parallellisatiemethodes sterk af te wijken van de in paragraaf 6.1.2.2 bepaalde $O(2^n)$. Dit wijst op zeer grote overhead bij het aanmaken en uitvoeren van de threads. Verder parallellisatie bestuderen in het Standard Environment van de Google App Engine voor Java heeft gebaseerd op deze resultaten dan ook geen zin. Parallellisatie in deze cloud-omgeving is dus enkel nuttig om kleine dingen asynchroon bij te houden, zoals bijvoorbeeld een timer. Naar performantiewinst toe is parallellisatie in het Google App Engine Standard Environment voor Java op het eerste zicht nutteloos of zelfs regelrecht zwaar contraproductief.

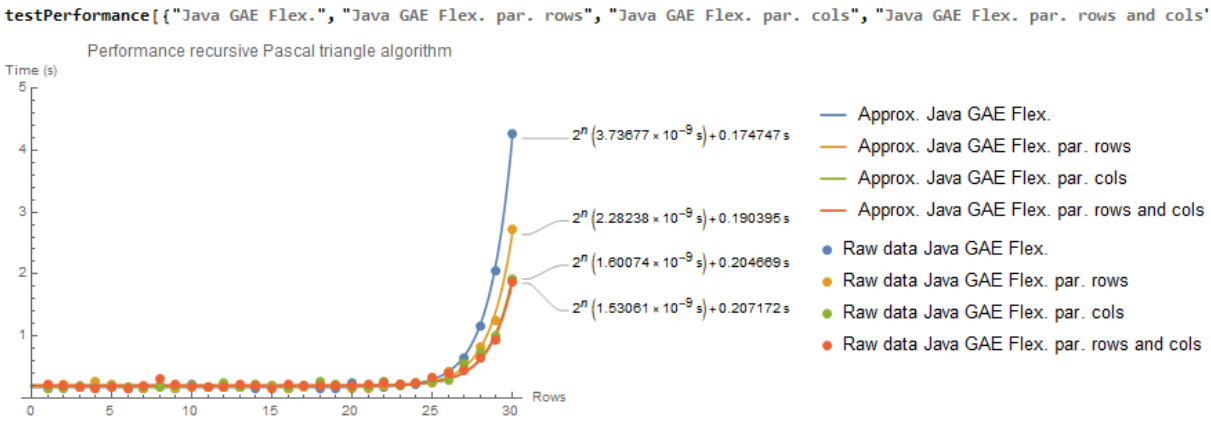
12.1.2.2 Google App Engine Flexible Environment

In tegenstelling tot het Standard Evironment vraagt het Flexivble Environment van de Google App Engine geen speciale aanpassingen aan de code om de geparallelliseerde versies van de Driehoek van Pascal in Java in de cloud te deployen. Omwille van de verontrustende resultaten voor het Standard Environment is parallellisatie in het Flexible Environment van de Google App engine ook eerst voor enkel geparallelliseerde rijen getest. Figuur 218 geeft de resultaten hiervan weer.



Figuur 218: Vergelijking parallellisatiemethode voor enkel rijen in het Flexible Environment van Google App Engine met de desktop-variant.

Figuur 218 toont veel betere resultaten voor het Flexible Environment van de Google App Engine in vergelijking met het Standard Environment. De relatieve performantiewinst is in het Flexible Environment nagenoeg identiek aan die van de desktop-versie. Parallellisatie in het Flexible Environment van de Google App Engine schijnt dus wél zin te hebben. Om uitsluitsel te krijgen moeten de andere parallellisatiemethodes natuurlijk ook getest worden. Het resultaat hiervan is te zien in Figuur 219.



Figuur 219: Vergelijking parallellisatiemethode voor enkel kolommen en zowel rijen als kolommen in het Flexible Environment van Google App Engine met de desktop-variant.

Zoals uit paragraaf 11.1 te verwachten is de performantie van de versie met geparallelliseerde kolommen ruwweg gelijk aan die van zowel parallelle rijen als kolommen, omwille van het beperkte aantal CPU-kernen dat beschikbaar is. Verder valt op dat in

vergelijking met de desktop-versie de parallellisatie eerder beperkt is. Betalende accounts die tot 64 kernen kunnen toewijzen aan de applicatie zouden voor beide parallellisatiemethodes uit Figuur 219 allicht nog veel betere resultaten kunnen halen. Zoals eerder aangehaald is de gratis versie in theorie beperkt tot 8, maar zoals paragraaf 11.1.3 al aangaf is het maximum in de praktijk 4.
Tabel 14 geeft de exacte relatieve performantiewinst weer voor de verschillende methodes. De resultaten voor de desktop-versie zijn overgenomen uit paragraaf 9.4.1. De coëfficiënten voor Google App Engine Flexible Environment zijn berekend volgens de methode uit paragraaf 9.4.1, gebaseerd op de resultaten uit Figuur 219.

Tabel 14: Vergelijking relatieve performantiewinst Java Google App Engine Flexible Environment met de desktop-versie.

Implementatie	**Performantiewinst geparallelliseerde rijen**	**Performantiewinst geparallelliseerde kolommen**	**Performantiewinst geparallelliseerde rijen en kolommen**
Desktop	72%	257%	245%
Google App Engine Flexible Environment	61%	131%	146%

Uit Tabel 14 is makkelijk af te leiden dat parallellisatie ook in het Flexible Environment een veel minder grote performantiewinst oplevert dan gehoopt. Volgens de documentatie voorzien door Google zou immers theorethisch een performantiewinst van 700% mogelijk moeten zijn. Uit de complicaties beschreven in paragraaf 11.1.3 was echter al te verwachten dat de performantiewinst niet meer dan 300% zou zijn. In de praktijk liggen deze resultaten dus nog een stuk lager. De parallellellisatiemethode voor enkel rijen doet het iets slechter in de cloud dan op de lokale pc, maart dit verschil ligt nog binnen de foutenmarge. Bovendien is de performantiewinst voor enkel rijen gebaseerd uit de coëfficiënten uit Figuur 218 zelfs 77%, wat zelfs iets meer is dan de Desktop-versie. Hieruit valt te besluiten dat de parallellisatie voor enkel rijen even goed is in Flexible Environment van Google App Engine als voor de desktop-toepassing.
De andere parallellisatiemethoden daarentegen leveren niet veel meer dan een verdubbeling van de performantie op, wat toch aanzienlijk minder is dan wat op de lokale test-pc te realiseren is, ondanks het feit dat beide platformen over hetzelfde aantal CPU-kernen beschikken. Hieruit valt de besluiten dat voor een beperkt aantal threads parallellisatie goed mogelijk is in Google App Engine Flexible Environment, maar dat voor een groot aantal threads de performantie onvoldoende is om dit platform aan te bevelen voor sterk parallelle object-georiënteerde toepassingen in Java.

12.1.3 Schaalbaarheid

De laatste test voor de Java-implementaties in de cloud is schaalbaarheid. Dit aspect is getest aan de hand van de methode beschreven in paragraaf 6.3.4. Deze paragraaf zet de resultaten van deze test uiteen voor de Java-implementaties van het recursieve Driehoek van Pascal-algoritme voor zowel het Standard als Flexible Environment van Google App engine, in die volgorde.

12.1.3.1 Google App Engine Standard Environment

Deze paragraaf bespreekt de resultaten van de schaalbaarheidstest voor het Standard Environment van de Google App engine. Het idee van deze test is een groot aantal requests te sturen op een interval van 100ms, gevolgd door een groot aantal requests met een interval van 10 ms, zoals toegelicht in paragraaf 6.3.4. Deze paragraaf licht ook toe dat het van belang is de request zodanig te kiezen dat de uitvoeringstijd ongeveer gelijk is aan 100 ms. Op deze manier is het de schaalbaarheid van het geteste platform zo goed mogelijk zichtbaar. Dan overlapt immers geen enkele request bij een tijdsinterval van 100 ms, terwijl voor een 10 ms interval er steeds 10 overlappende requests zijn. Op deze manier is dus een duidelijk beeld te vormen van de schaalbaarheid van de geteste cloud-platformen.
De driehoek van Pascal-API biedt de mogelijkheid om de uitvoeringstijd van de request zeer sterk te controleren, wat deze API ideaal maakt voor het testen van schaalbaarheid. De tijd nodig voor het afhandelen van de requests is immers eenvoudig te controleren door te spelen met de grootte van de opgevraagde Driehoek van Pascal.
Het is dus van belang de ideale grootte van de opgevraagde driehoek te bepalen, zodat het afhandelen van de request in het Standard Environment van Google App Engine ongeveer 100 ms rekentijd vraagt. Dit is voor dit onderzoek gedaan aan de hand van onderstaande redenering:
Figuur 215 geeft weer dat de nodige tijd voor het berekenen van een Driehoek van Pascal van n rijen in het Standard Environment van de google App engine ongeveer gelijk is aan:

$$t_n \approx 1{,}7 \cdot 10^{-8} \cdot 2^n$$

De constante uit Figuur 215 speelt uiteraard geen rol aangezien die enkel te maken heeft met networking en niet met de rekentijd nodig voor het berekenen van de Driehoek van Pascal. 100ms invullen voor t en omvormen geeft:

$$n = \log 2 \frac{0{,}1}{1{,}7 \cdot 10^{-8}}$$

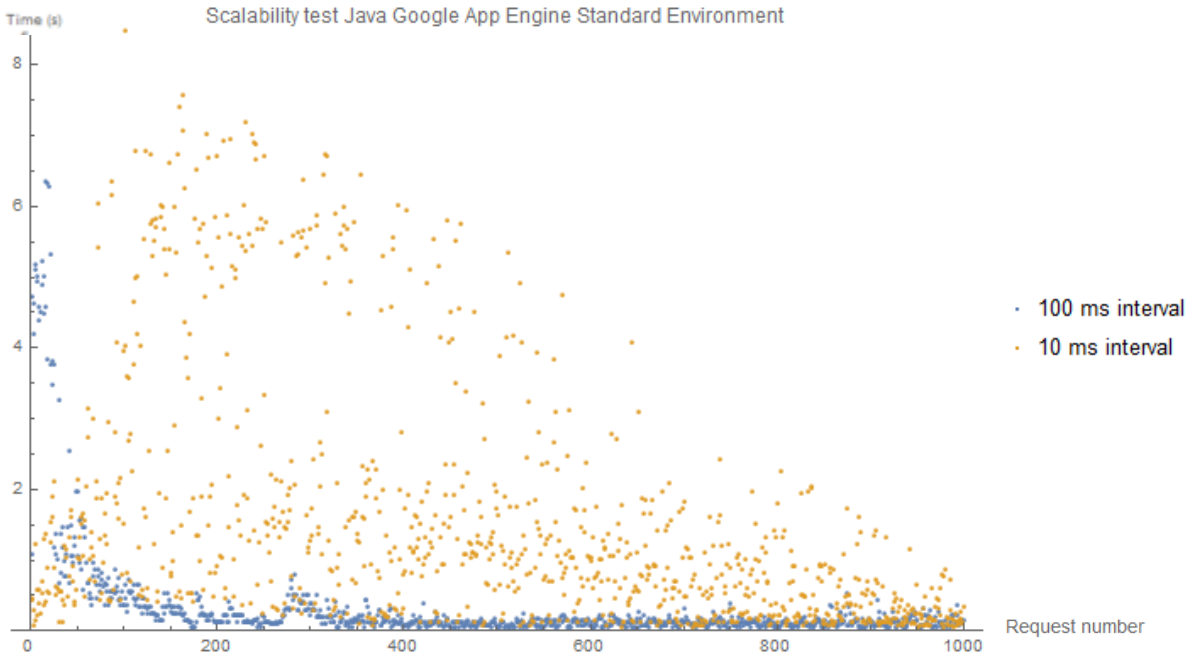
$$n = 22{,}48$$

Om een rekentijd van maximaal 100ms uit te komen moet n naar beneden worden afgerond. Dit betekent dus dat de ideale waarde voor n 22 is. De rekentijd voor n = 22 bedraagt ongeveer:

$$t \approx 1{,}7 \cdot 10^{-8} \cdot 2^{22}$$

$$t \approx 0{,}07s$$

Voor 22 rijen is dus bij een 100 ms interval zeker geen overlapping tussen de verschillende requests, en bij een 10 ms interval overlappen gemiddeld ongeveer 7 requests. Dit is dus zeker een goede keuze voor de schaalbaarheidstests voor Java in het Google App Engine Standard Environment.
De eerste schaalbaarheidstest voor 1000 requests geeft de resultaten uit Figuur 220.



Figuur 220: Resultaat schaalbaarheidstest Google App Engine Standard Environment voor Java voor 1000 requests.

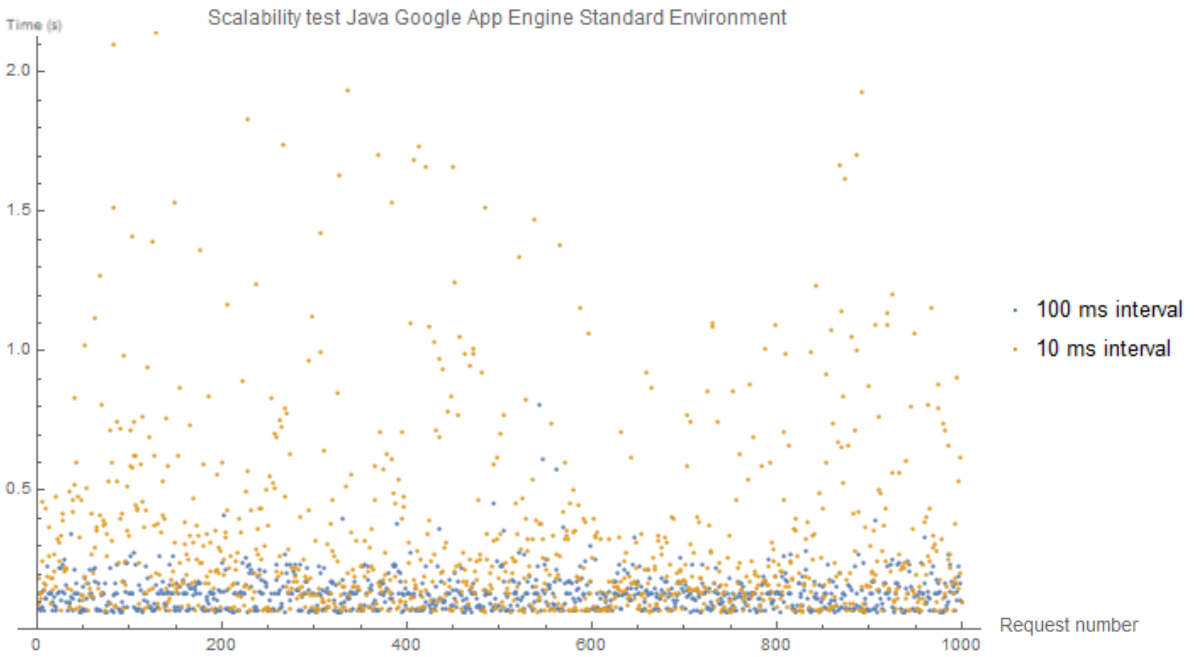
Deze test geeft interessante resultaten. Eerst en vooral valt op dat voor 100 ms de eerste requests tot 6 seconden nodig hebben om afgehandeld te worden. De allereerste request heeft wel slechts 1 seconde nodig, wat overeenkomt met de resultaten uit Figuur 214. Daar heeft de eerste request zelfs meer dan 2 seconden nodig. Dit betekent dat de eerste 10-20 requests al verzonden zijn voordat de eerste afgehandeld is. Hierdoor komen deze requests waarschijnlijk in een wachtrij terecht, omwille van de plotse piek in verkeer. Eens de API is 'wakker geschud', heeft ze echter veel minder tijd nodig om de volgende requests af te handelen, wat ook in Figuur 214 te zien is. Na de eerste 50 requests heeft geen enkele request nog meer dan een seconde nodig om afgehandeld te worden. Tegen het einde van de test met 100 ms interval is de nodige tijd zelfs stabiel en laag.
Voor een interval van 10ms zijn ook heel interessante resultaten te zien. Aangezien deze requests rechtstreeks na de test van 100 ms verstuurd zijn, is er geen 'opstartperiode' zichtbaar. Het is wel duidelijk dat zich opnieuw een soort wachtrij vormt waardoor de afhandelingstijd snel toeneemt voor de eerste 200 requests. Daarna daalt de wachttijd wel snel. Rond request nummer 500 wordt een aanzienlijk deel van de requests ongeveer even snel afgehandeld als de requests met een 100 ms interval. Rond request nummer 1000 wordt

elke request zelfs binnen de seconde verwerkt, ondanks de grote hoeveelheid verkeer. Dit alles wijst op een grote schaalbaarheid van het Google App Engine Standard Environment, al heeft die schaalbaarheid even nodig om op gang te komen.
Door de resultaten uit Figuur 220 statistisch te beschouwen zijn meer sluitende resultaten te bekomen. Tabel 15 geeft enkele statistische gegevens over de testdata weer.

Tabel 15: Gemiddelde en standaardafwijking van de resultaten uit Figuur 220.

Interval	**Gemiddelde**	**Standaardafwijking**
100 ms	0,32 s	0,76 s
10 ms	1,6 s	1,8 s

De resultaten zijn zoals reeds verwacht interessant en sterk verschillend voor het 100 ms en 10 ms interval. Eerst en vooral is de gemiddelde verwerkingstijd ongeveer 5 keer groter bij 10 ms dan bij 100 ms. De veel grotere standaardafwijking bij de test voor 10 ms illustreert wel dat de resultaten veel variabeler zijn voor het 10 ms interval. Dit in combinatie met het feit dat in Figuur 220 duidelijk te zien is dat de responstijd steeds daalt, suggereert dat het interessant is een tweede identieke test uit te voeren, onmiddellijk na de eerste, in de hoop dat de resultaten van deze test stabieler zijn. Hierdoor is een duidelijker en completer beeld te scheppen van de schaalbaarheid van het Google App Engine Standard Environment. Figuur 221 geeft het resultaat van deze test weer.



Figuur 221: Resultaat tweede schaalbaarheidstest Google App Engine Standard Environment voor Java voor 1000 requests, uitgevoerd onmiddellijk na de eerste identieke test.

Deze resultaten zijn een stuk stabieler, wat de theorie bevestigt dat de schaling een tijdje nodig heeft om tot stand te komen. Verder valt op dat geen enkele request meer dan 2 seconden nodig heeft om afgehandeld te worden, wat een grote verbetering is tegenover de vorige test en op zich zeer acceptabel is. Tabel 16 geeft de statistische data weer van dit testresultaat.

Tabel 16: Gemiddelde en standaardafwijking van de resultaten uit Figuur 221.

Interval	**Gemiddelde**	**Standaardafwijking**
100 ms	0,13 s	0,06 s
10 ms	0,33 s	0,33 s

Voor het 100ms interval is slechts een gemiddelde van 0,13 seconden gemeten. Dit is bijzonder weinig gegeven zijnde dat de rekentijd 0,07s bedraagt. Dit betekent dat alle requests voor 100 ms zeer snel verwerkt worden. De reden dat de verwerkingstijd hier lager is dan de connectietijd uit de tests in de vorige paragrafen, is te wijten aan het feit dat voor deze test er steeds een openstaande connectie is met de Google App Engine Service, waardoor de routing die normaal veel tijd kost van de test-pc tot de server niet voor elke request moet herhaald worden, wat bij de tests uit de vorige paragrafen wel zo is. Dit levert een veel snellere connectie op met de server. Verder is te zien dat voor 100ms de standaardafwijking zeer laag is, wat erop wijst dat alle requests zonder enig probleem kunnen worden afgehandeld, en er op geen enkel moment significante wachtrijen of iets dergelijks ontstaan.
Voor het 10ms interval zijn ook veel stabielere en betere resultaten te zien dan bij de eerste schaalbaarheidstest uit Figuur 220. Met een gemiddelde van 0,33s duurt de afhandeling van een request gemiddeld ongeveer 2,5 keer zo lang voor het 10 ms interval dan voor het 100 ms interval. Dit zijn behoorlijk goede resultaten, aangezien eerder in deze paragraaf bepaald is dat er steeds ongeveer 7 overlappende requests dienen afgehandeld te worden bij een interval van 10 ms. Verder valt op dat de standaardafwijking opnieuw veel hoger is voor 10 ms dan voor 100 ms. Deze keer zijn de resultaten wel veel stabieler dan bij de eerste schaalbaarheidstest, zoals Figuur 221 laat zien. Deze hoge standaardafwijking is dus ongetwijfeld het gevolg van de schaling die in werking treedt en probeert een evenwicht te zoeken tussen snel afhandelen van requests en spaarzaam omspringen met hardware-resources.
Ten slotte is het ook belangrijk om op te merken dat veel normale servers zouden crashen bij dergelijk grote hoeveelheid inkomend verkeer. Het feit dat de Google App Engine al het verkeer probleemloos afhandelt en daarbovenop nog eens behoorlijk snel is -weliswaar na enige opstarttijd-, is indrukwekkend. De resultaten zijn dus algemeen zeer goed voor dit platform wat schaalbaarheid betreft. Voor het Google App Engine Standard Environment is schaalbaarheid dus een pluspunt.

12.1.3.2 Google App Engine Flexible Environment

Naast het Standard Environment zijn de hierboven beschreven schaalbaarheidstests ook uitgevoerd voor het Flexible Environment van de Google App Engine. Opnieuw is de eerste stap in dit proces het berekenen van het optimale aantal rijen van de op te vragen Driehoek van Pascal, zodat de rekentijd ongeveer 100 ms bedraagt. Voor het Flexible Environment geeft dit, opnieuw gebaseerd op Figuur 215:

$$t_n \approx 3{,}8 \cdot 10^{-9} \cdot 2^n$$

$$n = \log 2 \frac{0{,}1}{3{,}8 \cdot 10^{-9}}$$

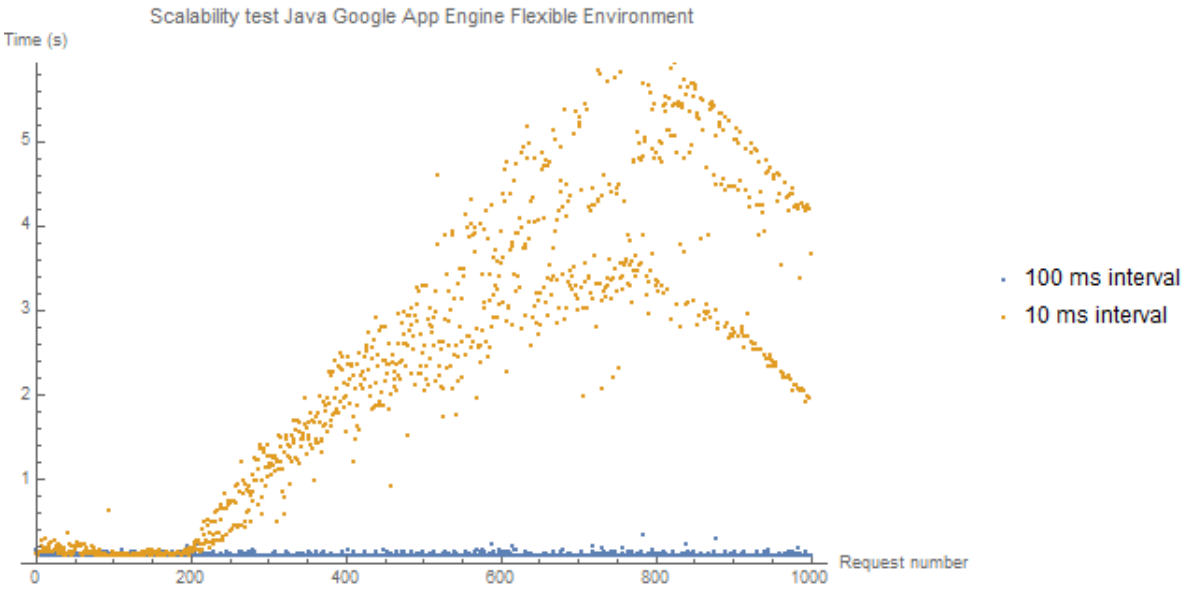
$$n = 24{,}64$$

$$=> n = 24$$

$$t \approx 3{,}8 \cdot 10^{-9} \cdot 2^{24}$$

$$t \approx 0{,}06s$$

Het optimale aantal rijen is dus 24. De geschatte rekentijd voor het bepalen van de Driehoek van Pascal is 0,06 seconden. Bij het 10 ms interval zijn er dus gedurende de hele test 6 overlappende requests. Figuur 222geeft het resultaat weer van het uitvoeren van de schaalbaarheidstest uit paragraaf 6.3.4 voor de Java-implementatie van de recursieve Driehoek van Pascal-API in het Flexible Environment van de Google App Engine voor 1000 requests.



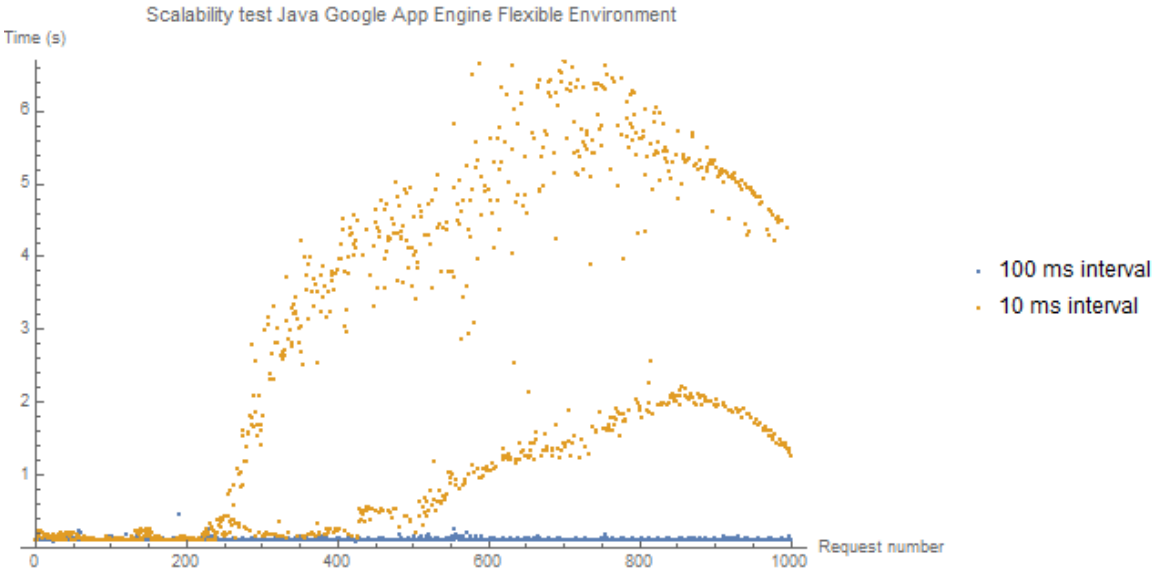
Figuur 222: Resultaat schaalbaarheidstest Google App Engine Flexible Environment voor Java voor 1000 requests.

De resultaten van deze test zijn opmerkelijk. Ze lijken wel het inverse te zijn van de resultaten voor het Standard Environment. Om te beginnen worden alle requests met een interval van 100 ms probleemloos afgehandeld. Voor de requests met een 10 ms interval geldt dit initieel ook. Dit betekent dat de Flexible Environment-API veel meer stand-by is dan de Standard Environment-versie, die duidelijk wat tijd nodig had om in gang te komen om een grote hoeveelheid verkeer af te handelen.
Daarmee houdt het goede nieuws echter op. Er is duidelijk te zien dat na een 200-tal requests plots de nodige tijd drastisch toeneemt bij het 10 ms interval. Dit wijst erop dat de draaiende instanties van de API in de cloud verzadigd raken. De vertraging is echter enorm en kan zowel door het netwerk als de instanties zelf worden veroorzaakt. Het is immers goed mogelijk dat Google het maximale aantal requests dat in korte tijd van een bepaalde afzender naar een bepaalde service gestuurd wordt beperkt om veiligheidsredenen. Tabel 17 geeft de statistische gegevens van deze laatste test weer.

Tabel 17: Gemiddelde en standaardafwijking van de resultaten uit Figuur 222.

Interval	**Gemiddelde**	**Standaardafwijking**
100 ms	0,11 s	0,02 s
10 ms	2,4 s	1,7 s

Het is duidelijk dat voor 100 ms de requests prima worden verwerkt, maar voor 10 ms de resultaten te wensen over laten. Net zoals bij het Standard Environment is hier ook direct na de eerste test een tweede uitgevoerd om te zien of er een verschil in resultaten waar te nemen is. De resultaten van deze tweede test zijn te zien in Figuur 223.



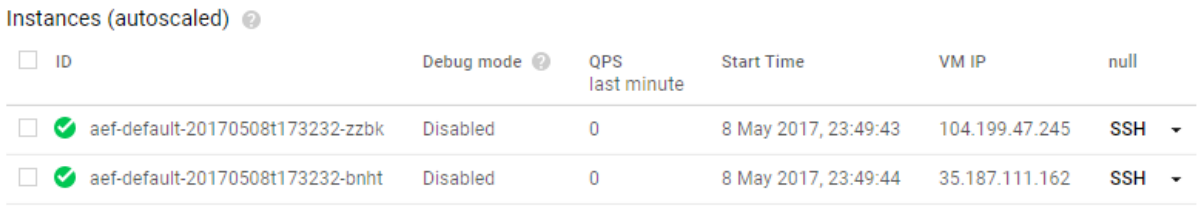
Figuur 223: Resultaat tweede schaalbaarheidstest Google App Engine Flexible Environment voor Java voor 1000 requests, uitgevoerd onmiddellijk na de eerste identieke test.

Figuur 223 laat zeer interessante resultaten zien. Het lijkt wel alsof de afhandeling van de requests gebeurt door twee afzonderlijke instanties, waarvan de ene veel performanter is dan de andere. Het is duidelijk dat één van de instanties veel minder snel verzadigd raakt, en dat bovendien de maximale verwerkingstijd bij deze instantie een stuk lager ligt. De algemene trend bij van de resultaten verandert echter niet veel ten opzichte van Figuur 222. Tabel 18 geeft de statistische gegevens van deze test weer.

Tabel 18: Gemiddelde en standaardafwijking van de resultaten uit Figuur 223.

Interval	**Gemiddelde**	**Standaardafwijking**
100 ms	0,11 s	0,02 s
10 ms	2,3 s	2,1 s

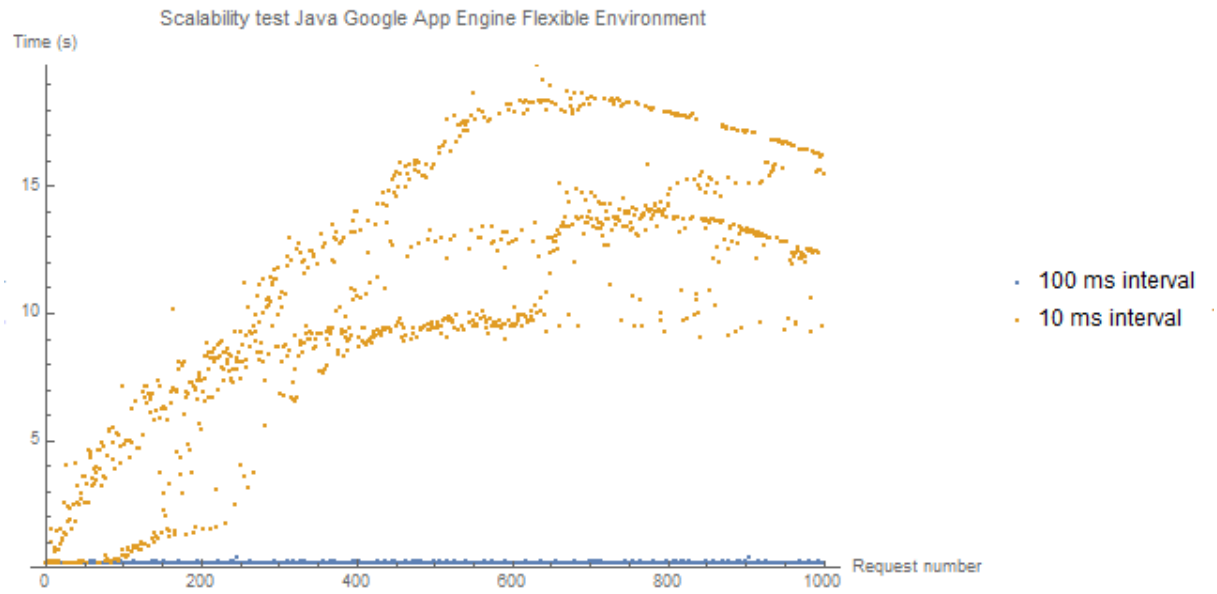
Voor het 100ms interval verandert er niets. Ook de gemiddelde tijd voor het 10ms interval is bijna identiek ten opzichte van de eerste schaalbaarheidstest voor dit cloud-platform. Enkel de standaardafwijking voor het 10ms interval is sterk toegenomen, wat duidelijk wijst op de twee verschillende instanties die elk een deel van de requests afhandelen en een groot onderling performantieverschil hebben. Dit vereist uiteraard extra onderzoek.
In de console van de Google App Engine in de browser is er een tab te vinden die alle actieve instanties van de applicatie weergeeft. Hier is inderdaad vast te stellen dat er standaard twee instanties van de applicaties aanwezig zijn, zoals Figuur 224 laat zien.



Figuur 224: Instanties van de Driehoek van Pascal-API in het Google App Engine Flexible Environment.

Dit verklaart waarom er twee duidelijk verschillende trends zichtbaar zijn in Figuur 223. Het is bovendien interessant en belangrijk om te weten dat het performantieverschil tussen verschillende instanties van de applicatie groot kan zijn in Google App Engine Flexible Environment. De autoscaling-optie die te zien is in bovenstaande figuur zorgt ervoor dat dit aantal instanties automatisch toeneemt indien de API lange tijd veel verkeer te verwerken krijgt. De hierboven uitgevoerde tests lijken dus te kort te zijn om dit te bewerkstelligen. Verder valt op dat na een tijd de tijd nodig voor het afhandelen van de requests afneemt. Dit kan op twee zaken wijzen. Enerzijds zou het kunnen dat Google detecteert dat er geen nieuw verkeer binnenkomt en daardoor de laatste requests sneller doorlaat, of gewoon dat de networking hardware door de afwezigheid van nieuwe requests minder verzadigd is.
Een andere verklaring kan zijn dat het Flexible Environment wel schaalbaar is, maar dat de schaalbaarheid toch een tijdje nodig heeft om op gang te komen. Dit kan best getest worden door dezelfde test nog eens uit te voeren, maar voor 25 rijen. Hierdoor verdubbelt de tijd nodig voor het afhandelen van de requests, waardoor er meer tijd ontstaat voor de Google

App Engine om de nodige schaling te implementeren. Deze methode is ook beter dan louter meer requests sturen, aangezien die tweede methode ineffectief is als Google de maximale hoeveelheid verkeer inderdaad bewust beperkt. Figuur 225 geeft de resultaten van deze test weer.



Figuur 225: Herhaling van de test uit Figuur 223 voor 25 rijen.

Ook deze test heeft interessante resultaten. Om te beginnen is de nodige tijd voor het afhandelen van de requests drastisch toegenomen. Voor beide instanties is meer dan een verdubbeling waarneembaar. Verder valt op dat er duidelijk nog steeds twee verschillende instanties of alleszins instantiegroepen zijn, elk met hun eigen karakteristieken. Het is immers ook mogelijk dat er meer dan twee instanties zijn, maar dat die elk in één van de twee patronen weergegeven in bovenstaande figuur vallen. De snelste instantie(groep) kan nog enige tijd het verkeer verwerken, maar vanaf 100 requests begint de tijd toe te nemen. Verder valt op dat er verschillende plateaus waarneembaar zijn in de nodige tijd voor de snelste instantie(groep). Dit weist erop dat de snelste instantie(groep) zichzelf wel degelijk schaalt, maar dat er enige tijd nodig is vooraleer deze schaling tot stand komt. Dit is te verklaren aan de hand van de achterliggende technologie van het Google App Engine Flexible Environment. Zoals beschreven in paragraaf 10.2.1.2 maakt het Flexible Environment van de Google App engine achter de schermen immers gebruik van Docker-containers. Om de applicatie te schalen moet het Flexible Environment dus nieuwe Docker-containers opstarten. Dit kan enkele seconden duren, afhankelijk van de onderliggende hardware en grootte van de container. Verder is het goed mogelijk dat Google moedwillig de schaling van de Docker-containers zodanig controleert dat alle requests kunnen afgehandeld worden binnen een acceptabele tijd, en tegelijkertijd het aantal nodige containers minimaal probeert te houden, om zo de overhead en de kosten te drukken. Het zou immers kunnen dat Google bij de verschillende plateaus in bovenstaande figuur detecteert dat de actieve instanties al het binnenkomend verkeer probleemloos kunnen afhandelen, waardoor Google onmiddellijk een aantal containers stopzet om kosten te besparen, waardoor de wachttijd

voor de requests opnieuw stijgt. Google pobeert zo een ideale balans te zoeken tussen kwalitatieve dienstverlening en efficiëntie. Naar het einde van de test toe daalt de wachttijd voor de verschillende requests gevoelig, wat aan zou kunnen geven dat Google het bovenvermeld evenwicht heeft bereikt. Dit alles is echter speculatie en moeilijk in te schatten op basis van bovenstaande testresultaten. Verder onderzoek is dus aangewezen om tot een sluitende conclusie te komen hieromtrent.
De traagste instantie presteert daarentegen ondermaats. Er is weinig schaling merkbaar, en de nodige tijd voor het afhandelen van de requests stijgt vrij lineair tot wel 20seconden. Tabel 19 geeft de statistische resultaten van de laatste test weer.

Tabel 19: Gemiddelde en standaardafwijking van de resultaten uit Figuur 225.

Interval	**Gemiddelde**	**Standaardafwijking**
100 ms	0,16 s	0,02 s
10 ms	10,6 s	4,9 s

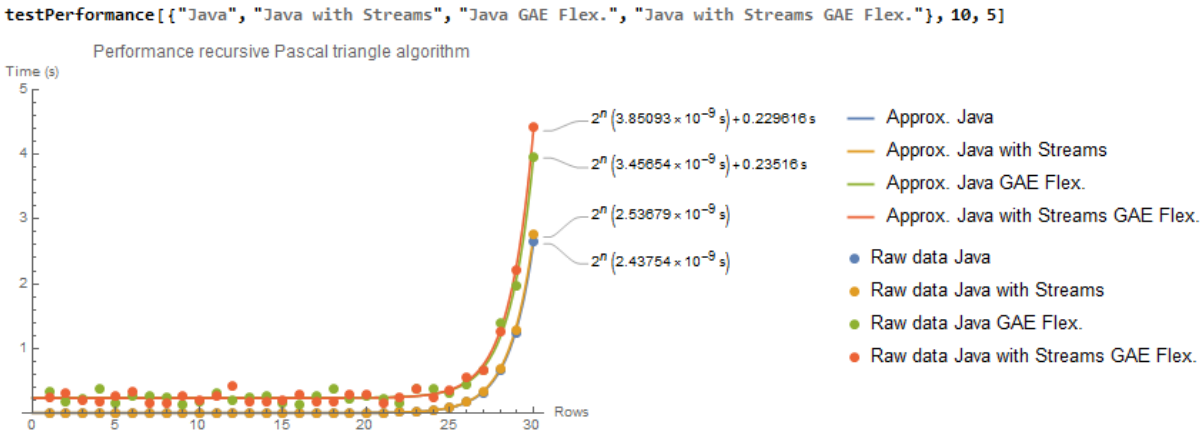
De gemiddelde tijd voor het afhandelen van de requests voor 100ms is met 50ms toegenomen, wat te verwachten is aangezien een enkele request afhandelen nu 120ms aan rekentijd vraagt in plaats van 60. De standaardafwijking is nog steeds zeer laag, wat betekent dat het Flexible Environment geen enkel probleem heeft met de requests met een 100ms interval af te handelen.
Voor de requests met een 10ms interval tekent zich echter een heel ander beeld af. De gemiddelde tijd is verviervoudigd, en de standaardafwijking is ook enorm. Dit wordt hoogstwaarschijnlijk veroorzaakt door de verzadiging van vooral de traagste instantie(groep), en het grote verschil in schaalbaarheid tussen de twee instanties/instantiegroepen. De schaalbaarheid van het Google App Engine Flexible Environment is dus moeilijk voorspelbaar en afhankelijk van de exacte instanties die de gebruiker krijgt toegewezen van Google. Het is wel een trend dat er verschillende instanties zijn met hun eigen karakteristieken. Zolang het verkeer beperkt blijft zijn deze instanties zelfs sneller dan het Standard Environment. Desalniettemin is schaalbaarheid geen sterkte van Java in het Google App Engine Flexible Environment, toch zeker niet wat betreft voorspelbaarheid.

12.2 Java met Streams

Naast de OO-versie in Java is ook de versie met Streams getest in de Google App Engine. Zoals eerder vermeld is deze versie enkel gedeployed naar het Flexible Environment van de Google App Engine omwille van het feit dat het Standard Environment enkel Java 7 ondersteunt, zoals toegelicht in paragraaf 11.1.2, en Streams pas ondersteund worden sinds Java 8. Deze paragraaf beschrijft dus de resulatateen van de tests beschreven in paragraaf 6.3, toegepast op de Java-implementatie van de recursieve Driehoek van Pascal-API met Streams in het Flexible Environment van de Google App Engine, zoals beschreven in paragraaf 11.1.3.

12.2.1 Connectiesnelheid en performantie

De eerste test die is uitgevoerd voor de Java-implementatie met Streams in het Flexible Environment van de Google App Engine is opnieuw de combinate van connectiesnelheid en performantie. Figuur 226 vergelijkt Java met Streams en de OO-versie in deze taal met elkaar, alsook met de desktop-implementatie van beide versies, en dit op dezelfde manier als alle performantietesten met 10 iteraties in de rest van dit hoofdstuk.

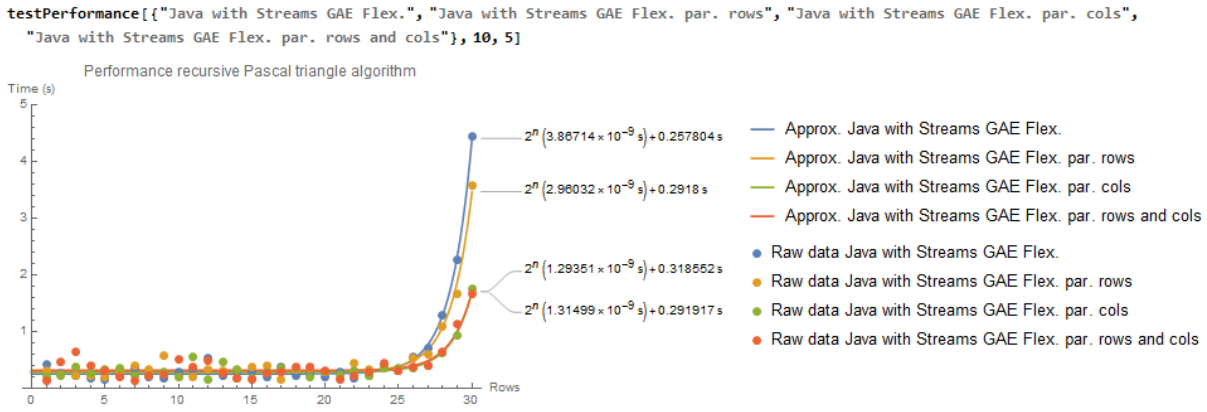


Figuur 226: Vergelijking connectiesnelheid en performantie voor Java met Streams in het Google App Engine Flexible Environment met de desktop versie en de OO-versie.

De connectiesnelheid is zoals verwacht identiek voor beide impelmentaties in de Google App Engine. Dit is op zich zeer logisch, aangezien deze beide versies binnen hetzelfde project zijn uitgewerkt, en zich dus op dezelfde server bevinden.
Het performantieverschil tussen de functionele en OO-versie in de cloud is ook zeer vergelijkbaar met de resultaten voor de desktop-versie. Dit is ook te verklaren door het feit dat in essentie de implementaties sterk op elkaar lijken, en de functionele implementatie gewoon een ander deel van hetzelfde project is. Hierdoor is de functionele implementatie relatief ten opzichte van de OO-implementatie in de Google App engine nagenoeg identiek aan die voor de desktop relatief ten opzichte van de OO-implementatie op dit platform. De resultaten uit Figuur 226 voldoen dus volledig aan de verwachtingen.

12.2.2 Parallellisatie

Ook voor deze implementatie zijn de parallellisatiekarakteristieken in de cloud bestudeerd, en vergeleken met de resultaten die in paragraaf 9.4.2 bekomen zijn voor de lokale test-pc. Verwacht wordt dat deze opnieuw sterk gelijkend zijn op die van de desktop-versie in verhouding tot de OO-implementatie. Figuur 227 geeft de resultaten weer van dit experiment voor alle parallellisatiemethoden in de cloud.



Figuur 227: Performantie verschillende parallellisatiemethoden voor Java met Streams in het Google App Engine Flexible Environment.

Eerst en vooral zijn er vrij veel uitschieters te zien in de testdata. Deze kunnen allerlei oorzaken hebben. Aangezien het gemiddelde van deze uitschieters over 10 iteraties nooit meer dan een seconde bedraagt, is het echter veilig om te stellen dat deze uitschieters waarschijnlijk niet veroorzaakt worden door de implementatie, maar eerder door factoren zoals internetverkeer etc. op het moment waarop de test is uitgevoerd. Het valt op dat de parallellisatie voor enkel rijen zeer slecht presteert. De andere parallellisatiemethodes geven ook ongeveer een verdrievoudiging van de performantie aan, wat niet zo veel is. Dit was echter wel te verwachten wegens de in paragraaf 12.1.3.2 reeds vastgestelde teleurstellende parallellisatiekarakteristieken van het Google App Engine Flexible Environment. Tabel 20 geeft een allesomvattende vergelijking van de resultaten uit Figuur 227 met die van de OO-implementatie in het Flexible Environment van de Google App Engine, en zowel de functionele als de OO-implementatie voor de desktop, die in paragraaf 9.4.2 reeds bepaald zijn.

Tabel 20: Vergelijking relatieve performantiewinst Java Google App Engine Flexible Environment met de desktop-versie.

Implementatie	**Performantiewinst geparallelliseerde rijen**	**Performantiewinst geparallelliseerde kolommen**	**Performantiewinst geparallelliseerde rijen en kolommen**
Desktop OO	72%	257%	245%
GAE Flex. OO	61%	131%	146%
Desktop Streams	32%	223%	257%
GAE Flex. Streams	30%	200%	200%

Tabel 20 laat zien dat de parallellisatie in het Google App Engine Flexible Environment voor Java met Streams zeer goed overeenkomt met de vastgestelde resultaten voor de desktop-versie met Streams. Enkel de versie met geparallelliseerde rijen en kolommen is significant trager. In het algemeen valt wel op dat parallellisatie voor Java met Streams betere resultaten laat zien dan de OO-implementatie in het Flexible Environment van de Google App Engine. De parallellisatiemethode voor enkel rijen bij de versie met Streams valt ook in de cloud uit de toon. Deze vreemde vaststelling komt wel overeen met de performantie van de versie met

Streams voor de lokale test-pc. Deze eigenaardigheid is dus op dezelfde manier te verklaren als in paragraaf 9.4.2.
Voor de andere parallellisatiemethodes scoort de versie met Streams echter veel beter dan de OO-versie in het Flexible Environment van de Google App engine. Dit kan liggen aan de manier waarop Streams omgaan met multithreading. Uit paragraaf 11.1.2 is immers reeds duidelijk dat parallellisatie voor OO-talen in de cloud niet per se evident is. Het zou dus zeker kunnen dat de manier waarop de OO-versie van het recursieve Driehoek van Pascal-algoritme aan parallellisatie doet ook in het Flexible Environment van de Google App Engine problemen met zich meebrengt naar performantie toe. Bovenstaande resultaten onderstrepen dit. Hieruit is duidelijk dat functionele implementaties in Java zich beter laten parallelliseren in de cloud dan hun object-georiënteerde tegenhangers.

12.2.3 Schaalbaarheid

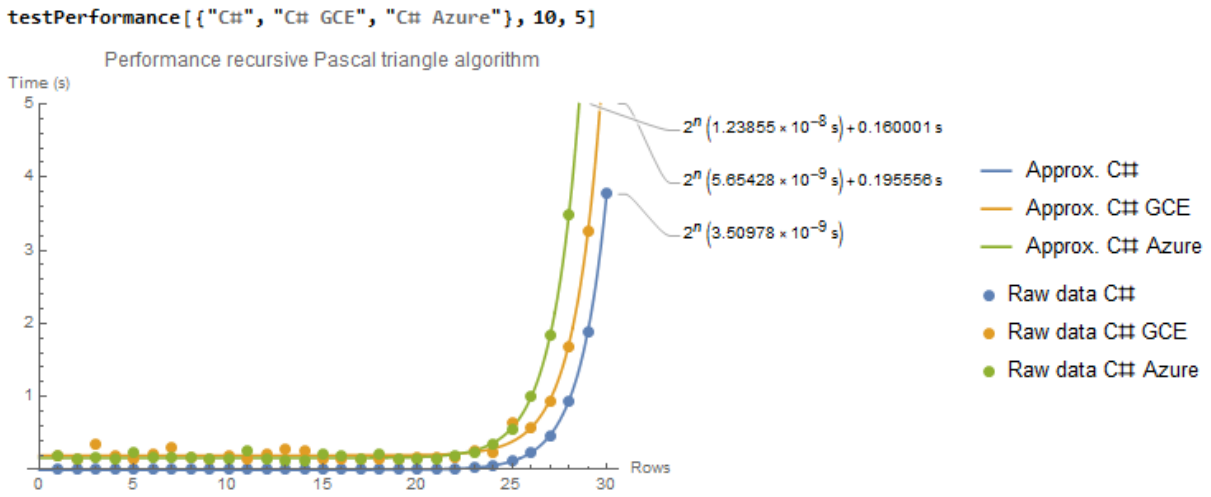
Het laatste te testen aspect is ook voor deze versie van het recursieve Driehoek van Pascal-algoritme de schaalbaarheid. Deze is echter identiek gebleken aan de OO-versie in Java van dit algoritme in het Flexible Environment van de Google App Engine. Het heeft dan ook geen zin de testresultaten hier opnieuw te bespreken. Alle conclusies voor deze implementaties zijn dan ook dezelfde als deze voor het Flexible Environment uit paragraaf 12.1.3.2. Dit is ook logisch, aangezien de functionele versie zich in hetzelfde project bevindt als de OO-versie. Het hele project moet immers geschaald worden bij een piek in verkeer, waardoor er geen onderscheid is in schaalbaarheid tussen de OO- en functionele variant.

12.3 C#

Ook voor de OO-implementatie in C# zijn talrijke testresultaten verzameld, gebaseerd op de tests beschreven in paragraaf 6.3. Deze taal is zoals beschreven in paragraaf 11.2 zowel in Google Compute Engine als in Microsoft Azure geïmplementeerd. Het eerste platform maakt gebruik van het .NET 4.5.2 framework, terwijl in het tweede platform het meer recente .NET Core-framework toegepast is. Deze paragraaf zet al de testresultaten voor C# uiteen voor deze beide platformen, volgens dezelfde structuur als de vorige paragrafen.

12.3.1 Connectiesnelheid en performantie

Zoals bij de vorige implementaties zijn de eerste bestudeerde parameters de connectiesnelheid en performantie. Deze zijn beide voor zowel de Google Compute Engine als Azure vergeleken met de desktop-versie in Figuur 228.



Figuur 228: Performantie en connectiesnelheid van C# in zowel Google Compute Engine als Azure vergeleken met de Desktop-versie.

De connectiesnelheid is voor beide cloud-platformen zeer acceptabel rond de 0,2 seconden. Ook de uitschieters zijn voor beide platformen beperkt in omvang en frequentie.
De performantie is echter drastisch verschillend tussen beide versies. De implementatie in Google Compute Engine presteert vrij goed, maar de implementatie in Azure is aanzienlijk trager. Tabel 21 geeft de exacte numerieke relatieve performantie van de verschillende implementaties weer.

Tabel 21: Vergelijking relatieve performantie verschillende cloud-omgevingen voor C#.

Cloud-omgeving	**Relatieve performantie t.o.v. desktop-versie**
Google Compute Engine	$\frac{5{,}7\cdot10^{-9}}{3{,}5\cdot10^{-9}}\approx 1{,}6$
Azure	$\frac{1{,}2\cdot10^{-8}}{3{,}5\cdot10^{-9}}\approx 3{,}4$

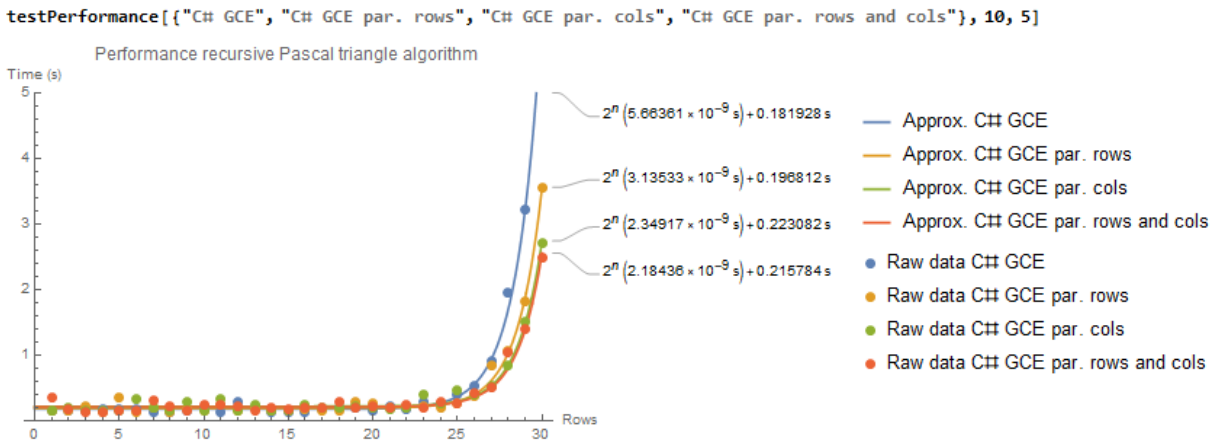
Google Compute Engine blijkt acceptabel te presteren. 60% trager dan de desktop versie is zeer acceptabel, aangezien de test-pc over behoorlijk krachtige hardware beschikt voor single-threaded taken.
Azure daarentegen doet het een stuk slechter. Het heeft namelijk gemiddeld 3,4 keer zoveel tijd nodig voor het genereren van de gevraagde Driehoek van Pascal als de desktop-versie. Dit is een aanzienlijke vertraging. Dit is echter nog binnen de grenzen van wat acceptabel te noemen is voor de meeste toepassingen. Voor rekenintensieve taken is dit platform echter verre van ideaal.

12.3.2 Parallellisatie

Ook de parallellisatiekarakteristieken zijn getest voor C# in zowel Google Compute Engine als Azure, op dezelfde manier als alle andere hierboven besproken implementaties. Deze paragraaf bespreekt voor elk van deze cloud-platformen de verkregen resultaten.

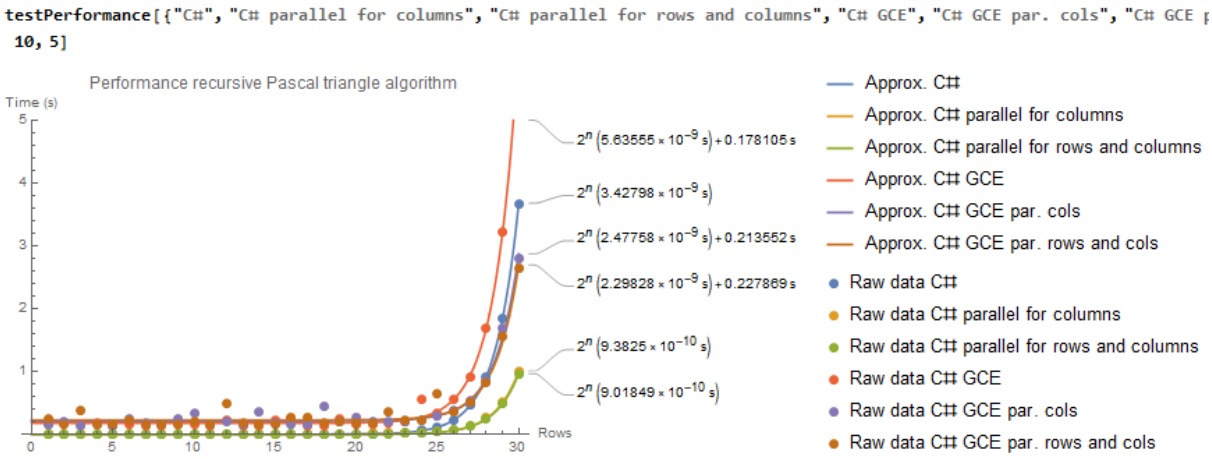
12.3.2.1 Google Compute Engine

Het eerste bestudeerde platform voor de parallellisatie van C# is Google Compute Engine. Om te beginnen geeft Figuur 229 de resultaten weer voor de verschillende parallellisatiemethoden in dit cloud-platform.



Figuur 229: Testresultaten verschillende parallellisatiemethoden voor de OO-implementatie in C# in Google Compute Engine.

De parallellisatiemethode voor enkel rijen lijkt het vrij goed te doen in Google Compute Engine. Het valt echter op dat de methodes voor kolommen en rijen en kolommen veel slechter presteren dan verwacht op basis van het aantal toegewezen CPU-kernen aan de VM, zoals beschreven in paragraaf 11.2.2. Net zoals bij het Flexible Environment van de Google App Engine, is er slechts ongeveer een verdubbeling van de performantie merkbaar voor geparallelliseerde kolommen en rijen en kolommen ten opzichte van de sequentiële versie, terwijl er 4 CPU-kernen zijn toegewezen aan de VM. Op de lokale pc was een veel grotere performantiewinst waar te nemen. Ter vergelijking geeft Figuur 230 de vergelijking weer tussen de twee laatst vernoemde parallellisatiemethodes in Google Compute Engine en de lokale pc.



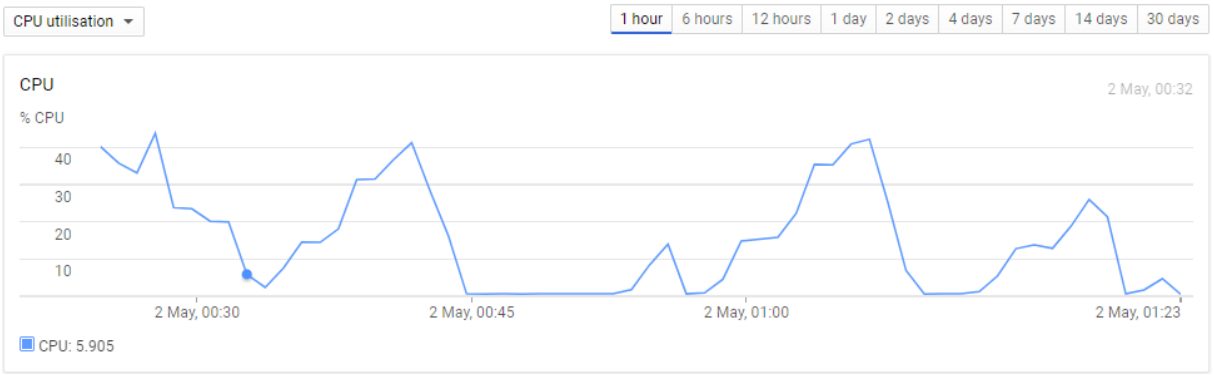
Figuur 230: Vergelijking parallellisatiemethode voor kolommen en rijen en kolommen in Google Compute Engine en de desktop-versie.

Ook de andere parallellisatiemethodes lijken teleurstellend te presteren voor de C#-implementatie van de recursieve Driehoek van Pascal-API in de Google Compute Engine. Allicht geeft een gedetailleerde vergelijking tussen de coëfficiënten van de desktop-versie en de versie in Google Compute Engine gedeployde versie van dit algoritme een beter inzicht in de mate waarin parallellisatie voor de cloud-gedeployde variant tekortschiet.
Tabel 23 verschaft zo een vergelijking aan de hand van de coëfficiënten van de benaderingskrommes voor elk van de parallellisatiemethodes uit Figuur 229 en de in paragraaf 9.4.3 verzamelde resultaten voor de Desktop-versie. Alle berekeningen zijn uitgevoerd op dezelfde manier als in paragraaf 9.4.3.

Tabel 22: Vergelijking relatieve performantiewinst bij de verschillende parallellisatiemethodes voor de OO-implementatie in C#in Google Compute Engine en Azure met de desktop-versie.

Implementatie	**Performantiewinst geparallelliseerde rijen**	**Performantiewinst geparallelliseerde kolommen**	**Performantiewinst geparallelliseerde rijen en kolommen**
Desktop	96%	300%	285%
Google Compute Engine	84%	148%	159%

Het is duidelijk dat het performantieverschil tussen de verschillende parallellisatiemethodes en de sequentiële versie in de desktop-versie van het recursieve Driehoek van Pascal-algoritme in C# veel groter is dan in de Google Compute Engine. Enkel de methode voor geparallelliseerde rijen doet het goed. De performantie van de andere parallellisatiemethodes is ronduit ondermaats. In de Google Compute Engine zijn er wel enkele mogelijkheden ter beschikking om na te gaan waar dit performantieverschil vandaan komt. Zo kan steeds het CPU-gebruik worden gemonitord van de applicatie in Compute Engine. Figuur 231 geeft dit CPU-gebruik weer.

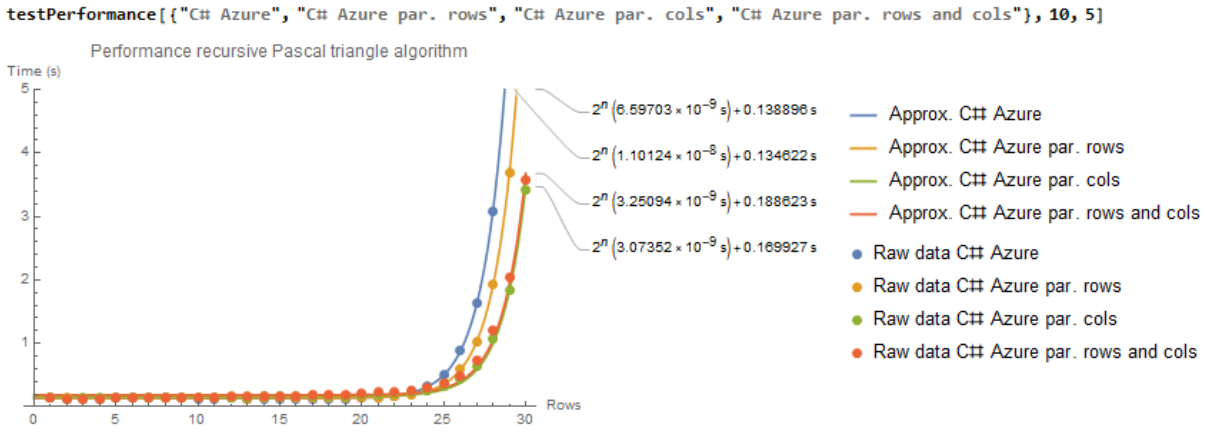


Figuur 231: CPU-gebruik van de OO-implementatie in C# gedeployed in de Google Compute Engine tijdens het uitvoeren van het experiment uit Figuur 229.

Figuur 231 is opmerkelijk. Blijkbaar gaat het CPU-gebruik van de VM nooit boven de 50% tijdens het uitvoeren van de applicatie. Dit verklaart waarom er slechts iets meer dan een verdubbeling in performantie te meten is. Rekening houdend met de overhead bij het afhandelen van eender welke request is het logisch dat de performantie in dat geval iets meer dan het dubbel van de performantie voor de sequentiële versie is.
Nu rest enkel de vraag waarom Google Compute Engine slechts 50% CPU-gebruik toewijst aan de applicatie. Dit blijkt bij nader inzien een zeer goede vraag te zijn. Google Compute Engine biedt nergens een verklaring hiervoor of een instelling om meer CPU toe te wijzen aan de applicatie. Eventueel zou het .NET 4-Framework of de voorbeeldapplicatie waarop deze implementatie is gebaseerd ook een rol kunnen spelen, maar dit is onwaarschijnlijk, zeker aangezien het Flexible Environment van de Google App Engine voor Java vergelijkbare resultaten liet zien, zoals beschreven in paragraaf 12.1.2.2. Bovendien is uit paragraaf 10.2.2 geweten dat het Flexible Environment van de Google App Engine onderliggend gebruik maakt van de Compute Engine, wat nog sterker doet vermoeden dat de oorzaak ligt bij het cloud-platform en niet het gebruikte framework. Dit is dus een eigenaardigheid die geen voor de hand liggende verklaring heeft, en niet eenvoudig opgelost kan worden. Aangezien de OO-implementatie in C# geen hoofddoel van deze masterproef is, wordt hier geen verder onderzoek naar verricht in dit werk. Verder onderzoek is dus aangewezen om tot op de bodem van dit probleem te raken. Google Compute Engine blijkt voorlopig dus niet goed parallelliseerbaar te zijn voor deze implementatie.

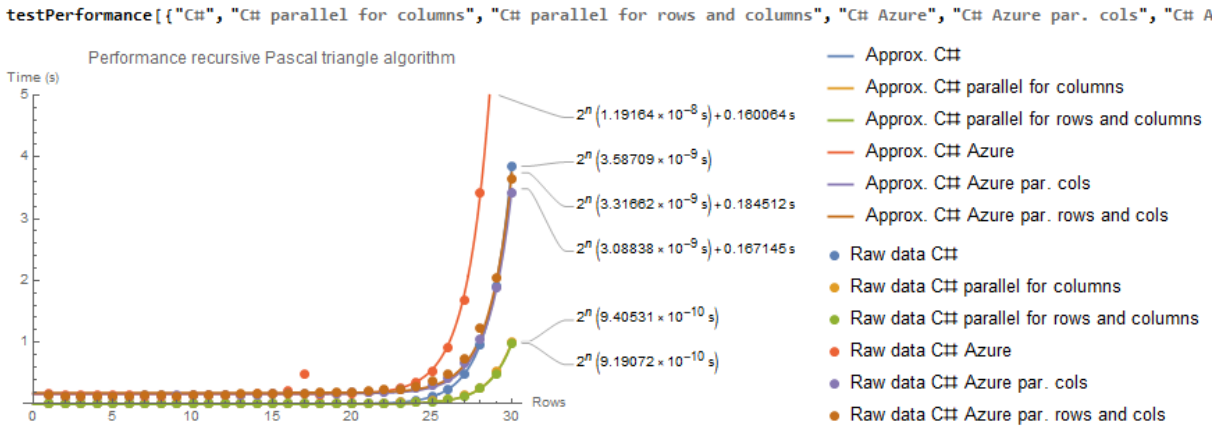
12.3.2.2 Azure

Naast Google Compute Engine is uiteraard ook de parallellisatie voor de OO-implementatie van de recursieve Driehoek van Pascal-API in C# in Azure getest. Figuur 232 geeft de resultaten hiervan weer.



Figuur 232: Testresultaten verschillende parallellisatiemethoden voor de OO-implementatie in C# in Azure.

De Azure-implementatie met .NET Core blijkt veel beter parallelliseerbaar te zijn dan de .NET 4-variant in Google Compute Engine. Deze implementatie bereikt ruim een verdrievoudiging van de uitvoeringssnelheid van de sequentiële versie. Ook voor Azure zijn de versies voor geparallelliserde kolommen en geparallelliseerde rijen en kolommen naast elkaar geplot in een aparte figuur, en dit naast de desktop-versie. Dit levert Figuur 233 op.



Figuur 233: Vergelijking parallellisatiemethode voor kolommen en rijen en kolommen in Azure en de desktop-versie.

Deze figuur laat mooi zien dat parallellisatie in de Azure cloud ongeveer gelijk is aan die op de lokale pc. Tabel 23 geeft ten slotte ook voor dit cloud-platform een vergelijkend overzicht tussen de verschillende cloud-gedeployde parallellisatiemethodes en de in paragraaf 9.4.3 verzamelde resultaten voor de Desktop-versie. Opnieuw zijn alle berekeningen analoog gebeurd aan die in paragraaf 9.4.3.

Tabel 23: Vergelijking relatieve performantiewinst bij de verschillende parallellisatiemethodes voor de OO-implementatie in C#in Google Compute Engine en Azure met de desktop-versie.

Implementatie	**Performantiewinst geparallelliseerde rijen**	**Performantiewinst geparallelliseerde kolommen**	**Performantiewinst geparallelliseerde rijen en kolommen**
Desktop	96%	300%	285%
Azure	67%	255%	233%

Na gedetailleerderd de resultaten vergeleken te hebben aan de hand van Tabel 23 zijn er opnieuw een aantal interessante vaststellingen te doen. In Azure is juist de omgekeerde trend te zien ten opzichte van in Google Compute Engine. Parallelle rijen presteren niet optimaal, terwijl de andere parallellisatiemethoden vrij dicht in de buurt komen van de desktop-versie. Zoals in paragraaf 10.3.1 al aangegeven is ook in Azure het aantal CPU-kernen dat kan worden toegewezen aan de applicatie beperkt tot 4. Een belangrijk verschil met het Google Cloud Platform is dat zelfs betalende accounts in Azure niet de mogelijkheid hebben om van meer dan 4 CPU-kernen gebruik te maken voor hun applicaties. Dit is op zich wel jammer. Uiteindelijk volgt hieruit als besluit dat wat parallellisatie betreft in Azure zeer acceptabele resultaten te verkrijgen zijn met .NET Core, maar dat er geen mogelijkheid is om deze resultaten verder op te drijven door up te graden naar een duurdere licentie.

12.3.3 Schaalbaarheid

Ten slotte is ook de schaalbaarheid getest voor C# in zowel Google Compute Engine als Azure. Opnieuw is dit gebeurd aan de hand van de schaalbaarheidstest beschreven in paragraaf 6.3.4. Net zoals in vorige paragraaf zijn hieronder eerst de resultaten van deze test voor Google Compute Engine besproken, en daarna die voor Azure.

12.3.3.1 Google Compute Engine

Zoals hierboven reeds aangegeven is Google Compute Engine het eerste behandelde cloud-platform voor C#. De eerste stap is opnieuw de grootte van de te genereren Driehoek van Pascal zodanig te bepalen dat de uitvoeringstijd van de request ongeveer 100 ms is, om zo de beste testresultaten te kunnen bekomen. Dit gebeurt op dezelfde manier als in paragraaf 12.1.3 voor Java. De coëfficiënt waar deze berekening mee start is bekomen uit Figuur 228.

$$t_n \approx 5{,}7 \cdot 10^{-9} \cdot 2^n$$

$$n = \log 2 \frac{0{,}1}{5{,}7 \cdot 10^{-9}}$$

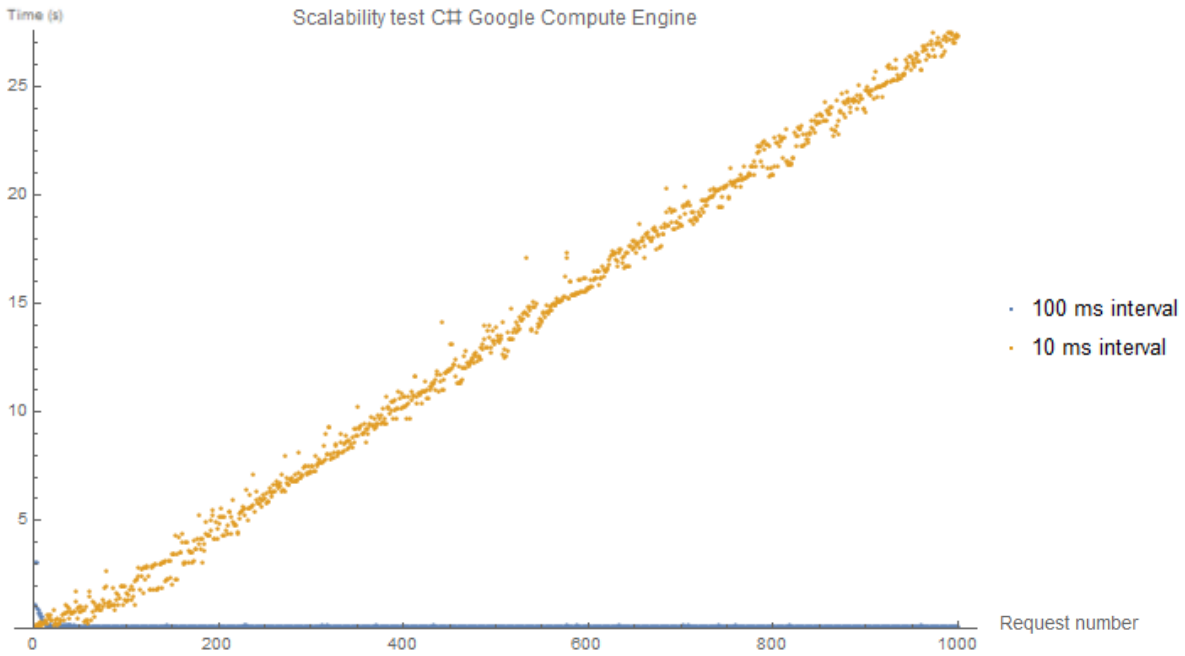
$$n = 24{,}06$$

$$=> n = 24$$

$$t \approx 5{,}7 \cdot 10^{-9} \cdot 2^{24}$$

$$t \approx 0{,}1s$$

Voor Google Compute Engine is het ideale aantal rijen dus 24. Voor dit aantal is de tijd nodig voor het berekenen van een enkele driehoek ongeveer exact 100 ms, wat ideaal is. Voor de requests met een 100 ms interval zal dus zo goed als nooit een request overlappen, terwijl voor een 10ms interval steeds 10 overlappende requests zullen zijn. Figuur 234 geeft het resultaat van deze test weer.



Figuur 234: Schaalbaarheidstest voor de OO-implementatie in C# met .NET 4 in Google Compute Engine voor 1000 requests.

De resultaten van dit experiment zijn even interessant als teleurstellend, maar ook enigszins te verwachten. Om te beginnen hebben de allereerste requests met een 100 ms interval veel meer tijd nodig om verwerkt te worden dan de andere. Voor de requests met een 10 ms interval is deze verhoging bij de eerste requests afwezig. Dit wijst erop dat de Google Compute Engine net als het Standard Environment van de App Engine even 'wakker geschud' moet worden. Alle andere requests met een 100 ms interval kunnen perfect verwerkt worden.
De requests met een 10 ms interval langs de andere kant geven veel slechtere resultaten. De wachttijd stapelt zich enkel maar lineair op. Dit wijst op de afwezigheid van eender welke schaling. Dit is eigenlijk wel te verwachten, aangezien Google Compute Engine een VM voorziet waarop de .NET 4-applicatie draait. Dit is uiteraard een enkele VM, en deze is niet schaalbaar. Voor de volledigheid geeft Tabel 24 de statistische gegevens van bovenstaande test weer.

Tabel 24: Gemiddelde en standaardafwijking van de resultaten uit Figuur 234.

Interval	Gemiddelde	Standaardafwijking
100 ms	0,14 s	0,14 s
10 ms	13 s	8,2 s

Zoals uit Figuur 234 reeds te vermoeden viel is de performantie voor het 100 ms interval zeer goed. De vrij hoge standaardafwijking is vooral te wijten aan de eerste requests die veel langer duren dan de rest. Voor het 10ms interval is een zeer slecht gemiddelde van 13s meetbaar en een zeer hoge standaardafwijking omwille van de lineaire stijging van de wachttijd over het gehele request-bereik. Dit betekent dus dat per definitie Compute Engine de schaalbaarheidsboot compleet mist. Aangezien dit een IaaS-service is waarbij een sterke focus ligt op het aanmaken en beheren van een enkele VM, is dit resultaat wel te verwachten.

12.3.3.2 Azure

Deze paragraaf bespreekt hetzelfde experiment als hierboven weergegeven, maar dan voor de .NET Core-implementatie van de recursieve Driehoek van Pascal-API in Azure. De eerste stap is opnieuw het berekenen van het optimale aantal rijen voor de op te vragen driehoek tijdens de test. De coëfficiënt waarop deze berekening zich baseert is opnieuw bekomen uit Figuur 228:

$$t_n \approx 1{,}2 \cdot 10^{-8} \cdot 2^n$$

$$n = \log 2 \frac{0{,}1}{1{,}2 \cdot 10^{-8}}$$

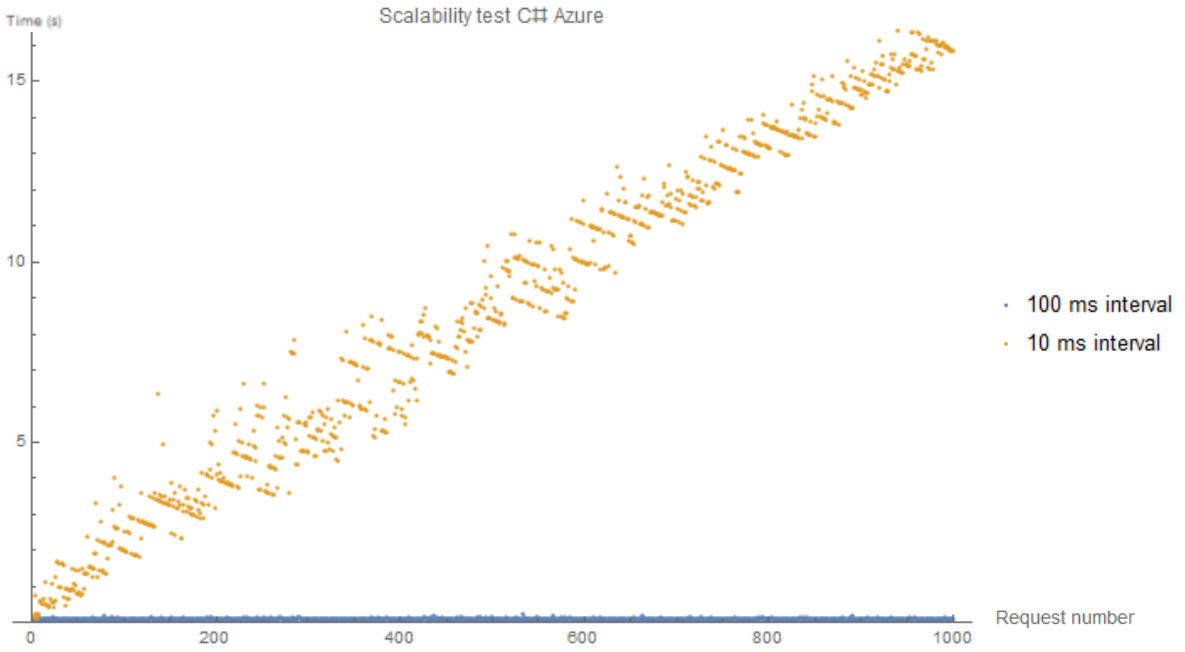
$$n = 22{,}99$$

$$=> n = 23$$

$$t \approx 1{,}2 \cdot 10^{-8} \cdot 2^{23}$$

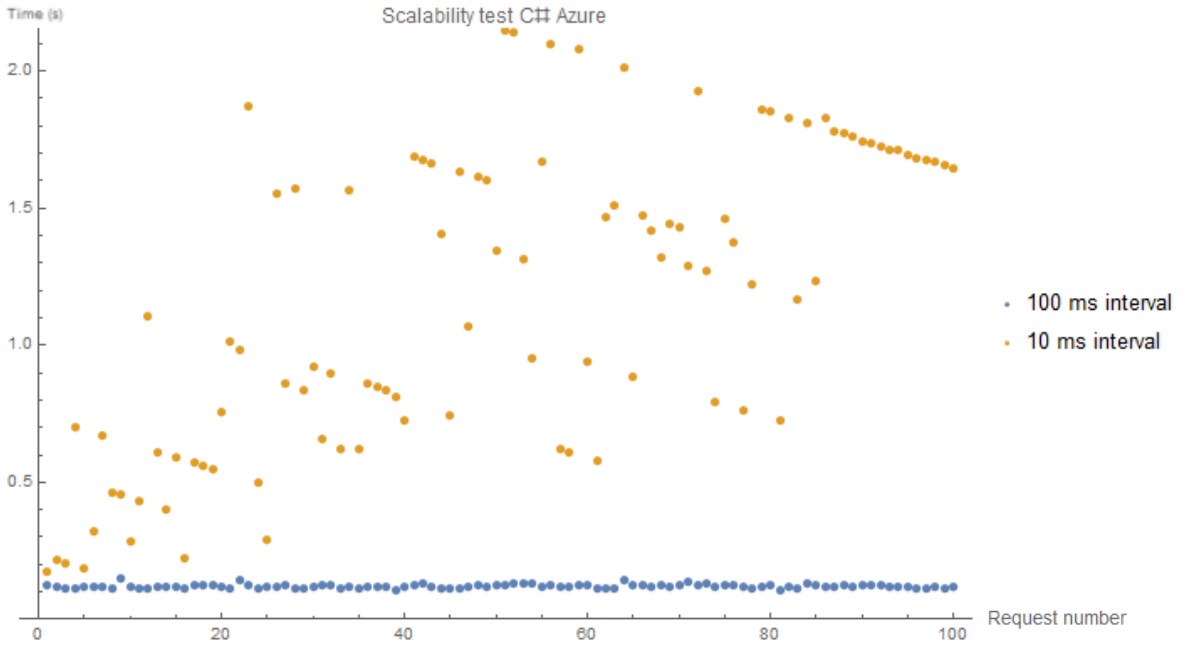
$$t \approx 0{,}1s$$

Normaal gezien dient er zoals eerder vermeld naar beneden afgerond te worden, maar aangezien in dit geval het resultaat zodanig dicht bij 23 ligt, is het toch beter om naar boven af te ronden. Dit omdat het verschil binnen de foutenmarge ligt, en omdat bij een afronding naar boven opnieuw 10 overlappende requests ontstaan bij een 10 ms interval, terwijl dit er bij een afronding naar beneden slechts 5 zouden zijn, wat minder interessante resultaten zou opleveren, die tevens moeilijker vergelijkbaar zijn met de resultaten voor de andere geteste implementaties cloud-omgevingen. In dit geval is het dus zeker verantwoord om toch naar boven af te ronden.
Figuur 235 geeft de resultaten weer van de schaalbaarheidstest voor in Azure met 23 rijen.



Figuur 235: Schaalbaarheidstest voor de OO-implementatie in C# met .NET 4 in Azure voor 1000 requests.

Een aantal zaken vallen op uit deze test. Om te beginnen is er geen vertraging voor de eesrte requests in Azure, wat erop wijst dat de app meer stand-by is dan in Google Compute Engine. Verder valt voor het 10 ms interval op dat de schaalbaarheid niet indrukwekkend is. De resultaten lijken op het eerste zicht op die van de Google Compute Engine, maar bij Azure is er wel degelijk een zekere schaalbaarheid aanwezig. De lineaire stijging van de tijd nodig voor het afhandelen van de requests is immers meer 'uitgesmeerd'. Bovendien is deze tijd nooit veel meer dan 15 seconden, terwijl dit bij Google Compute engine bijna 30 seconden kan worden. De schaalbaarheid van Azure is allicht duidelijker zichtbaar in Figuur 236, waar hetzelfde experiment is uitgevoerd, maar ingezoomd is op de eerste 100 requests.



Figuur 236: Zelfde experiment als in Figuur 235, maar voor slechts 100 requests.

In Figuur 236 is duidelijk zichtbaar dat er wel degelijk sprake is van beperkte schaalbaarheid in Azure. Deze beperkte schaalbaarheid is voor veel applicaties waarschijnlijk reeds voldoende om een aangename gebruikerservaring te bieden. Ten slotte geeft Tabel 25 de statistische resultaten van het experiment uit Figuur 235 weer.

Tabel 25: Gemiddelde en standaardafwijking van de resultaten uit Figuur 235.

Interval	**Gemiddelde**	**Standaardafwijking**
100 ms	0,12 s	0,01 s
10 ms	8,6 s	4,6 s

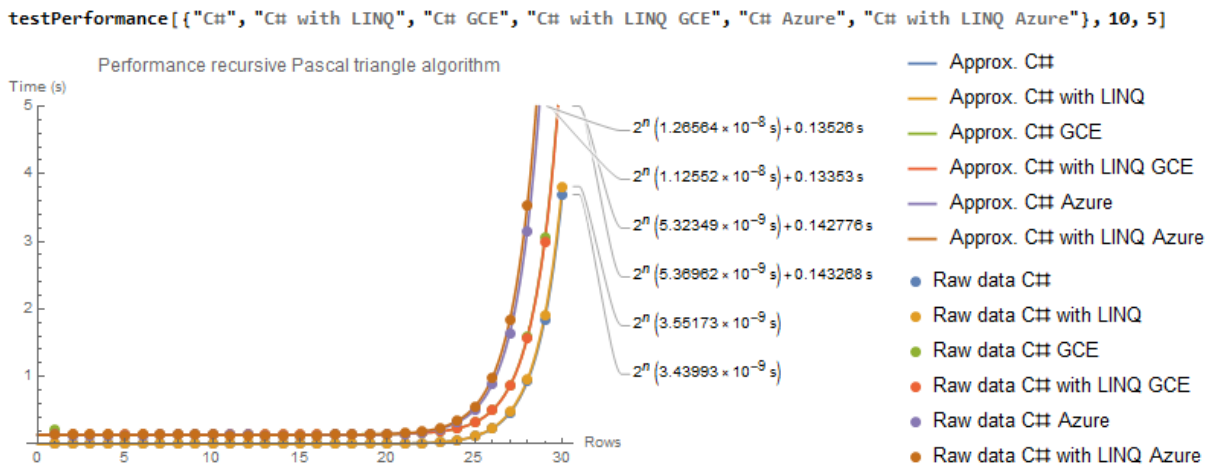
Uit Tabel 25 volgt zoals verwacht dat Azure schaalbaarder is dan Google Compute Engine. Voor het 100 ms interval zijn zelfs zeer goede resultaten te noteren wegens de afwezigheid van een grotere latency bij de eerste requests. Voor het 10ms interval is de vertraging ook ongeveer 5 seconden minder groot gemiddeld, en de standaardafwijking ook aanzienlijk kleiner. Azure met .NET Core is dus beperkt schaalbaar en presteert op dit gebied beter dan Google App engine met .NET 4.5 voor de C#-versie van de recursieve Driehoek van Pascal-API.

12.4 C# met LINQ

Net als voor Java is ook voor C# naast de klassieke OO-implementatie een functionele implementatie in de cloud gedeployed. Voor C# is dit gedaan aan de hand van LINQ. Deze versie is zowel in de Google Compute Engine als in Azure gedeployed, zoals beschreven in paragraaf 11.2. Deze paragraaf bespreekt de testrestultaten van deze implementaties, en vergelijkt ze tevens met de OO-implementaties in C# voor alle geteste platformen.

12.4.1 Connectiesnelheid en performantie

Net zoals bij Streams in paragraaf 12.2.1, zijn voor C# met LINQ de connectiesnelheid en performantie rechtstreeks vergeleken met de OO-implementatie in elk cloud-platform. Dit is tevens op een enkele figuur geplot met de performantie van de functionele en object-georiënteerde implementatie op de lokale pc.



Figuur 237: Vergelijking connectiesnelheid en performantie voor C# met LINQ in de Google Compute Engine en Azure met de desktop-versie en de OO-versie.

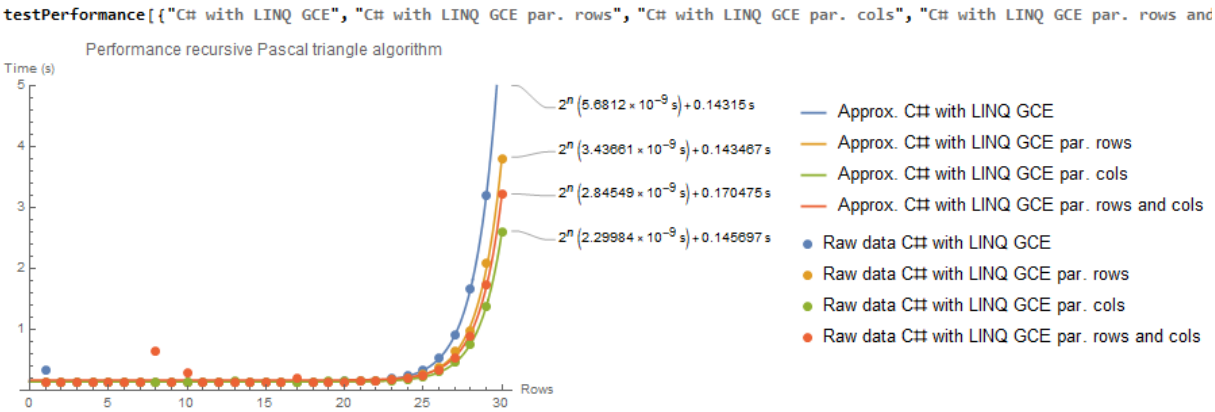
Net zoals bij Streams is er geen beduidend performantieverschil tussen de OO-implementatie en de implementatie met LINQ te zien. Dit volgt mooi het eerder uitgezette patroon voor functionele implementaties in klassieke OO-talen. De connectiesnelheid is vanzelfsprekend ook gelijk voor de functionele talen ten opzichte van hun klassieke tegenhangers in elk van de geteste cloud-platformen.

12.4.2 Parallellisatie

Ook voor C# met LINQ is in dit onderzoek in zowel Google Compute Engine als Azure gekeken naar de parallellisatiekarakteristieken. Deze paragraaf bespreekt de resultaten voor elk van deze cloud-platformen.

12.4.2.1 Google Compute Engine

Allereerst geeft Figuur 238 de resultaten weer voor alle parallellisatiemethoden in de Google Compute Engine weer.



Figuur 238: Performantie verschillende parallellisatiemethoden voor C# met LINQ in Google Compute Engine.

C# met LINQ blijkt last te hebben van dezelfde beperkingen als de OO-implementatie in C# in de Google Compute Engine, hetzij in mindere mate. Er kan niet meer dan ruwweg een verdubbeling van de performantie bereikt worden bij de parallellisatie voor zowel rijen als kolommen. Daarentegen is de peformantie voor enkel parallelle kolommen wel behoorlijk

goed, en bovendien beduidend beter dan bij zowel parallelle rijen als kolommen. Dit wijst toch op een inefficiënte afhandeling van de overhead die komt kijken bij een groot aantal threads in Google Compute Engine. Tabel 26 geeft een overzicht van de relatieve performantiewinst die elk van deze parallellisatiemethodes bereiken ten opzichte van de sequentiële versie in de Google Compute Engine. Als referentie geeft deze tabel ook de resultaten voor de desktop-versie weer uit paragraaf 9.4.4. Bovendien zijn ook de resultaten voor de OO-versie voor zowel de desktop-versie (zie paragraaf 9.4.3) als de versie in Google Compute Engine (zie paragraaf 12.3.2.1) weergegeven. De resultaten voor de implementatie met LINQ in de Google Compute Engine zijn gebaseerd op de bovenstaande figuur. De berekeningen zijn analoog aan die uit de vorige paragrafen betreffende parallellisatie.

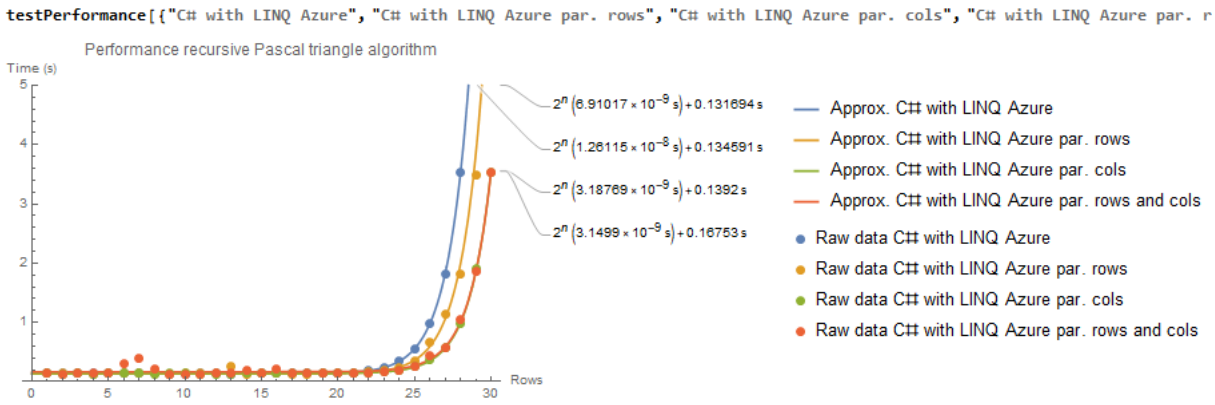
Tabel 26: Vergelijking relatieve performantiewinst C# en C# met LINQ in Google Compute Engine met de desktop-versie.

Implementatie	**Performantiewinst geparallelliseerde rijen**	**Performantiewinst geparallelliseerde kolommen**	**Performantiewinst geparallelliseerde rijen en kolommen**
Desktop OO	96%	300%	285%
GCE OO	84%	148%	159%
Desktop LINQ	56%	300%	257%
GCE LINQ	68%	248%	204%

Deze resultaten laten zien dat ook voor C# in deze cloud-omgeving de functionele implementatie beter scoort dan de OO-versie. Enkel de versie voor parallelle rijen scoort iets minder. Voor de andere parallellisatiemethoden scoort LINQ beduidend beter dan de OO-implementatie. Ook valt op dat in de Google Compute Engine LINQ 'gematigder' lijkt te zijn dan bij de desktop-versie. De methode voor geparallelliseerde rijen scoort beter, terwijl de andere methodes iets slechter scoren. Bij deze eerste parallellisatiemethode is het verschil echter behoorlijk klein. Dit is dus waarschijnlijk te wijten aan toevalligheden en is afhankelijk van de exacte manier waarop het besturingssysteem de uitvoering van de verschillende threads plant. Het relatieve performantieverschil voor de andere methodes illustreert weer dat er waarschijnlijk extra overhead komt kijken bij het beheren van threads in de cloud. Ook het vreemde gedrag van de Google Compute Engine wat betreft processorgebruik voor de geparallelliseerde versies, zoals toegelicht in paragraaf 12.3.2.1, heeft allicht te maken met de lagere performantie voor C# met LINQ in de cloud dan op de lokale test-pc. Desalniettemin levert ook hier het gebruik van een functionele implementatie een beduidende performantiewinst op ten opzichte van de OO-versie.

12.4.2.2 Azure

Figuur 239 geeft de resultaten weer voor de verschillende parallellisatiemethoden in Azure.



Figuur 239: Performantie verschillende parallellisatiemethoden voor C# met LINQ in Azure.

Opnieuw blijkt Azure veel beter te presteren in het parallellisatie-departement dan Google Compute Engine. De parallellisatiemethodes voor kolommen en rijen en kolommen geven een veel grotere performantiewinst dan in Google Compute Engine, net zoals bij de OO-implementatie in C#. Bovendien is de performantie voor deze twee parallellisatiemethodes nagenoeg identiek, wat aangeeft dat Azure toch op een veel efficiëntere manier omgaat met multithreading dan Google Compute Engine.
Tabel 27 zet de relatieve performantiewinst die te bereiken valt door parallellisatie voor elke methode naast elkaar, gebaseerd op bovenstaande figuur, en dit naast de resultaten die eerder in deze thesis bekomen werden voor de Desktop-versie en de OO-implementatie in Azure, op net dezelfde manier als Tabel 26 uit vorige paragraaf.

Tabel 27: Vergelijking relatieve performantiewinst C# met LINQ in Azure met de desktop-versie.

Implementatie	**Performantiewinst geparallelliseerde rijen**	**Performantiewinst geparallelliseerde kolommen**	**Performantiewinst geparallelliseerde rijen en kolommen**
Desktop OO	96%	300%	285%
Azure OO	67%	255%	233%
Desktop LINQ	56%	300%	257%
Azure LINQ	88%	319%	307%

Het eerste wat opvalt is dat de resultaten voor geparallelliseerde rijen sterk variëren voor de verschillende implementaties, zonder een duidelijk patroon. Hieruit is te concluderen dat de uiteindelijke efficiëntie van deze parallellisatiemethode sterk afhankelijk is van hoe het OS op het moment van de uitvoering de threads beheert. Als de thread van de laatste rij toevallig een lage prioriteit krijgt bvb., kan dit de uitvoering van heel het algoritme sterk vertragen. Daarom is voorzichtigheid geboden bij het vergelijken van de performantie van deze parallellisatiemethode over zo veel implementaties en verschillende platformen heen.

Verder valt op dat Azure in het algemeen veel beter scoort voor parallelisatie dan Googcle Compute Engine. Deze trend was al duidelijk bij de OO-implementaties, maar bij de functionele implementaties is dit nog duidelijker. In Azure is zelfs een performantiewinst van meer dan 300% te meten, wat onbereikbaar is op de lokale pc. Dit kan wel aan benaderingsfouten liggen, maar het is toch veilig om te stellen dat de implementatie in C# met LINQ in Azure zeker even performant is als op de lokale pc. Dit is het beste resultaat van alle cloud-implementaties in de klassieke programmeertalen Java en C#. Merk ook op dat de OO-implementatie in Azure opmerkelijk minder performant is. Azure is dus wat C# betreft een goed parallelliseerbaar cloud-platform, met als enige jammere beperking dat slechts 4 CPU-kernen kunnen worden toegewezen aan de applicatie. Bovendien blijkt de functionele implementatie ook hier weer veel beter te scoren dan de object-georiënteerde.

12.4.3 Schaalbaarheid

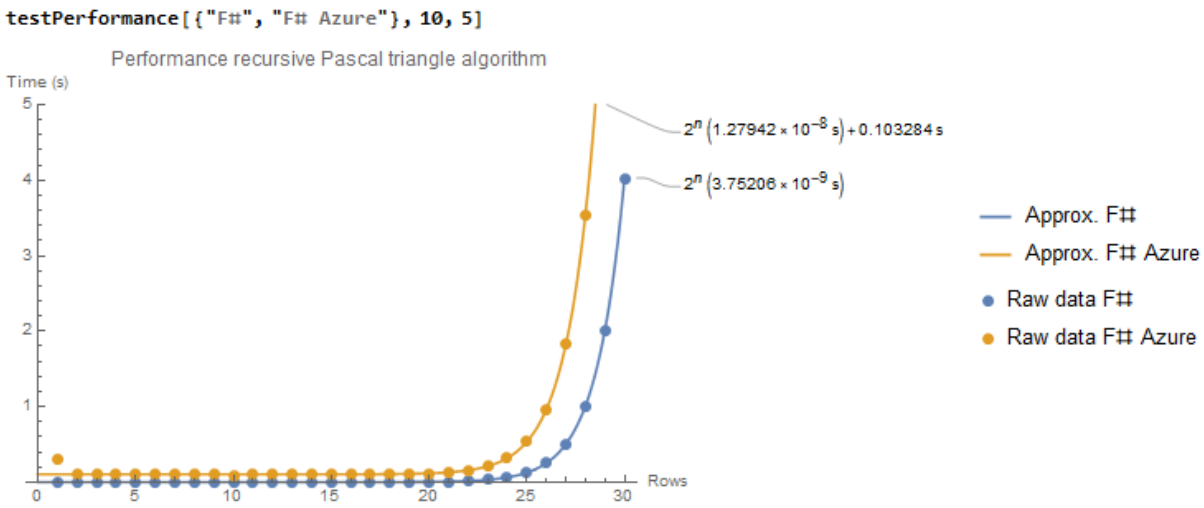
Net zoals bij Java met Streams is ook voor C# met LINQ de schaalbaarheid getest, maar blijkt deze identiek te zijn aan die van de OO-implementaties, opnieuw allicht omwille van het feit dat de functionele implementaties zich in hetzelfde project bevinden als de OO-varianten, waardoor de volledige infrastructuur identiek is, en het hele project steeds geschaald moet worden, onafhankelijk van de aangesproken implementatie. Alle conclusies uit paragraaf 12.3.3 zijn hier dus ook van toepassing.

12.5 F#

F# is de eerste hoofdzakelijk functionele taal die in dit onderzoek behandeld is. Zoals beschreven in paragraaf 11.3 is deze taal geïmplementeerd in het Azure cloud-platform van Microsoft. Deze paragraaf bespreekt de resultaten van de verschillende tests uit paragraaf 6.3 voor F# in de Azure cloud, op dezelfde manier als in de vorige paragrafen.

12.5.1 Connectiesnelheid en schaalbaarheid

Zoals bij alle voorgaande geteste talen is de eerste test ook hier een eenvoudige plot van de sequentiële variant van het Driehoek van Pascal-algoritme, zowel in de Azure cloud als de desktop-variant. Hieruit zijn opnieuw over een gemiddelde van 10 iteraties makkelijk de connectiesnelheid en performantie van de Azure cloud af te leiden. Figuur 240 geeft de resultaten van deze test weer.



Figuur 240: Testresultaten verschillende parallellisatiemethoden voor F# in Azure en de Desktop-versie voor 10 iteraties.

De connectiesnelheid blijkt zeer goed te zijn met 0,1 seconde gemiddeld. Dit is vreemd genoeg een noemenswaardige verbetering ten opzichte van C#. Dit kan meerdere oorzaken hebben. Een eerste mogelijke oorzaak is dat de server waarop de F#-applicatie is geïnstalleerd een aanzienlijk betere verbinding heeft dan die waar de C#-applicatie op staat. Een andere verklaring zou kunnen zijn dat het Suave framework veel sneller is in het afhandelen van de requests dan .NET Core. Het is echter onwaarschijnlijk dat dit zo een groot verschil in connectietijd oplevert. Er is een enkele uitschieter te zien bij de allereerste request, maar de andere requests worden perfect binnen een korte tijd afgehandeld. Dit betekent dat de F#-app niet zo stand-by is in Azure, maar wel heel betrouwbaar en snel de andere requests afhandelt.
Tabel 28 geeft de vergelijking in performantie tussen Azure en de dektop-versie voor F# weer.

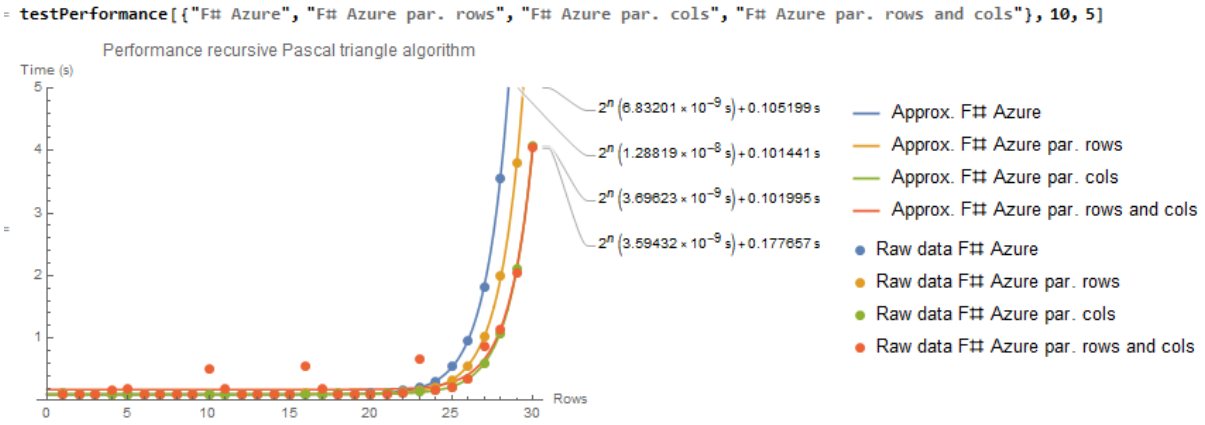
Tabel 28: Relatieve performantie Azure voor F#.

Cloud-omgeving	**Relatieve performantie t.o.v. desktop-versie**
F# Azure	$\frac{1{,}3 \cdot 10^{-8}}{3{,}8 \cdot 10^{-9}} \approx 3{,}4$

Azure blijkt voor F# ongeveer 3,4 keer trager te zijn dan de Desktop-variant. Dit resultaat komt exact overeen met het resultaat dat bepaald werd voor C# in paragraaf 12.3.1. Er is dus geen extra performantiekost voor het gebruiken van de overwegend functionele taal F# met de Suave-bibliotheek in de cloud, althans niet voor Azure.

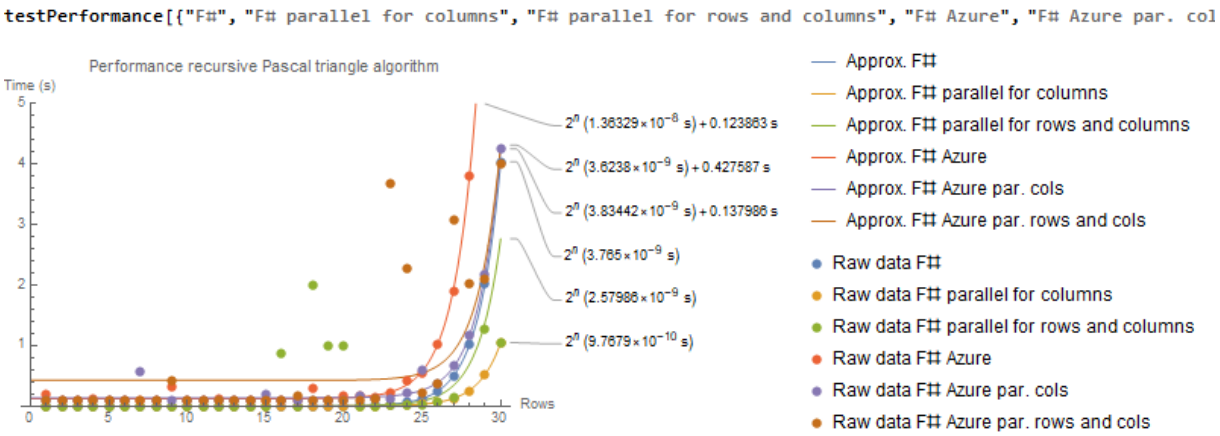
12.5.2 Parallellisatie

Ook voor F# zijn alle parallellisatiemethodes beschreven in paragraaf 9.1 geïmplementeerd en getest in de Azure cloud. Figuur 241 geeft hiervan de resultaten weer.



Figuur 241: Performantie verschillende parallellisatiemethoden voor F# in Azure.

Uit deze figuur zijn een aantal interessante resultaten af te leiden. Eerst en vooral lijken de parallellisatiekarakteristieken voor F# in Azure goed te zijn. Verder is te zien dat net als bij de desktop-variant bestudeerd in paragraaf 9.4.5, de methode voor parallelle rijen en kolommen uitschieters vertoont.
Ter verificatie en illustratie van het probleem met de uitschieters geeft Figuur 242 de resultaten weer van een test waarin de parallellisatiemethode voor kolommen en rijen en kolommen in Azure naast de desktop-versie geplot zijn. Dit is gedaan voor één iteratie, zodat de uitschieters het best tot hun recht komen. Merk op dat de gevonden coëfficiënten kunnen afwijken van de vorige testen, maar dit is uiteraard omwille van het feit dat slechts één iteratie beschouwd is in deze test en de willekeurige uitschieters hierdoor een grote impact op de resultaten hebben.



Figuur 242: Parallellisatie voor kolommen en rijen en kolommen in Azure en lokaal voor één iteratie.

Op bovenstaande resultaten is duidelijk te zien dat er ongeveer evenveel uitschieters aanwezig zijn in de Azure cloud als bij de lokale versie. Het voorkomen van deze uitschieters en de grootte ervan lijken ook totaal willekeurig. De uitschieters in Azure hebben

in het algemeen wel een grotere omvang dan die voor de lokale versie. Rekening houdend met het performantieverschil tussen de cloud en de lokale versie is dit echter als normaal te beschouwen. Aangezien zowel in Azure als op de test-pc de uitschieters een gelijkaardig patroon vertonen, zijn ze dus zeker niet te wijten aan de hardware van de test-pc, maar aan F# zelf. De exacte oorzaak hiervan uitzoeken zou te ver leiden voor dit onderzoek. Verder onderzoek is dus nodig op dit gebied. Voorlopig is wel te concluderen dat er voorzichtig omgesprongen dient te worden met overmatig parallelliseren in F# en met parallellisaties te nesten, zoals ook beschreven in paragraaf 9.4.5.
Tabel 29 geeft de relatieve performantiewinst weer van Azure in de cloud in vergelijking met de desktop-versie. Merk op dat de uitschieters bij rijen en kolommen amper invloed schijnen gehad te hebben op de trendlijn. De in Figuur 241 bekomen coëfficiënt voor geparallelliseerde rijen en kolommen voor de implementatie in Azure is hierdoor wel representatief en kan aldus probleemloos opgenomen worden in de tabel, in tegenstelling tot de coëfficiënt die in paragraaf 9.4.5 bepaald is voor de desktop-versie. Tabel 29 is gebaseerd op de gegevens uit Figuur 241 en paragraaf 9.4.5. De performantiewinst is berekend zoals in paragraaf 9.4.5.

Tabel 29: Vergelijking relatieve performantiewinst bij de verschillende parallellisatiemethodes voor F# in Google Compute Engine en Azure met de desktop-versie.

Implementatie	**Performantiewinst geparallelliseerde rijen**	**Performantiewinst geparallelliseerde kolommen**	**Performantiewinst geparallelliseerde rijen en kolommen**
Desktop	96%	285%	285%
Azure	91%	251%	261%

F# blijkt goed parallelliseerbaar te zijn in Azure. Alleen blijkt ook voor F# het aantal beschikbare CPU-kernen in de cloud beperkt te zijn, wat jammer is. Uit vorige paragrafen was dit echter al af te leiden. De performantie komt zeer dicht in de buurt van de desktop-versie. Bovendien is deze stelling consistent geldig voor alle parallellisatiemethoden, in tegenstelling tot veel van de eerder besproken talen. In het algemeen is F# in de cloud dus beter parallelliseerbaar dan de verschillende geteste OO-tegenhangers, in vergelijking met wat lokaal te bereiken valt, en vooral beter voorspelbaar is, wat toch een groot voordeel is. De enige opmerking hierbij zijn de uitschieters bij de versie voor geparallelliseerde rijen en kolommen.

12.5.3 Schaalbaarheid

Zoals bij alle voorgaande geteste implementaties is de laatste uit te voeren test ook voor F# schaalbaarheid in de cloud. Ook hier is de eerste stap het bepalen van de ideale grootte van de op te vragen driehoek van Pascal, zodat de rekentijd ongeveer 100 ms bedraagt. Hieronder de berekening voor F# in Azure, analoog aan die voor de andere implementaties. De coëfficiënt waarop deze berekening gebaseerd is komt uit Figuur 240:

$$t_n \approx 1{,}3 \cdot 10^{-8} \cdot 2^n$$

$$n = \log 2 \frac{0{,}1}{1{,}3 \cdot 10^{-8}}$$

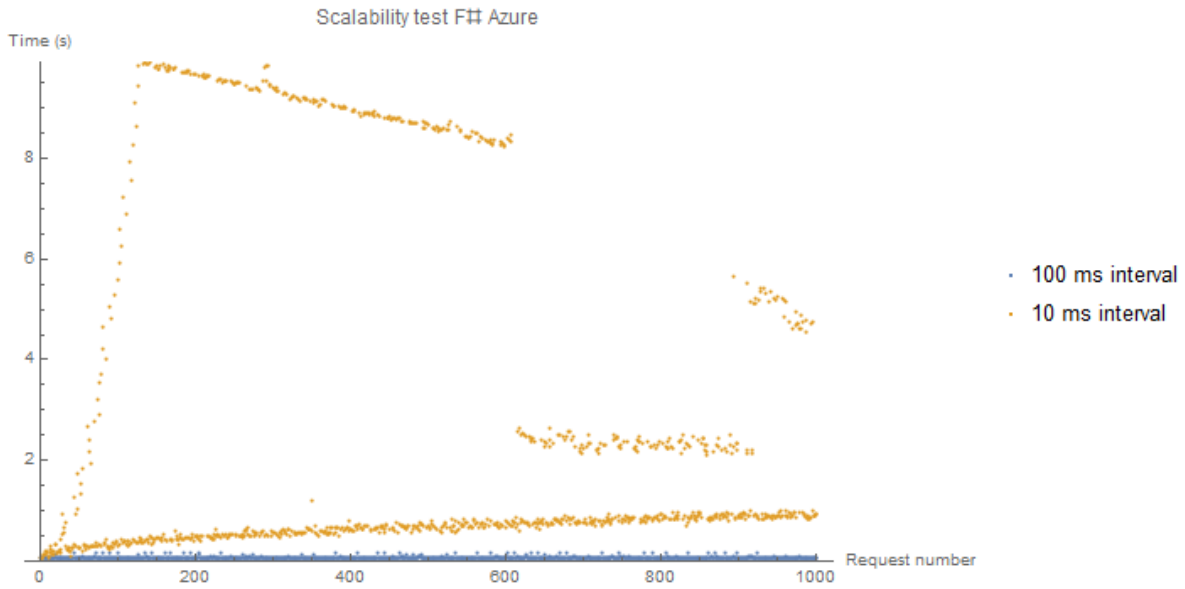
$$n = 22{,}87$$

$$=> n = 22$$

$$t \approx 1{,}3 \cdot 10^{-8} \cdot 2^{22}$$

$$t \approx 0{,}05s$$

Het resultaat voor n is 22,87. Dit betekent in principe dat er naar beneden afgerond moet worden, en n gelijkgesteld moet worden aan 22. Hierdoor is de tijd voor het afhandelen van de requests ongeveer 0,05 seconden. Dit resultaat ligt niet dicht genoeg bij 23 om toch naar boven af te ronden, maar is ook zodanig hoog dat de rekentijd minimaal is bij een afronding naar beneden. Aangezien dit resultaat zodanig op de grens ligt tussen naar boven en naar beneden afronden, is er voor geopteerd deze implementatie zowel te testen voor n = 22 als n = 23. De rekentijd voor n = 23 is immers gelijk aan 0,11 seconden. Bij n = 22 zijn er bij het 100 ms interval dus zeker geen overlappingen tussen de verschillende requests, terwijl er voor het 10 ms interval steeds ongeveer 5 requests overlappen. Voor n = 23 overlappen de requests voor een 100 ms interval licht, terwijl er voor het 10 ms interval maar liefst 11 overlappende requests zijn op elk moment. Door beide resultaten te combineren is een allesomvattend beeld te verkrijgen dat zeker evenwaardig is aan de andere testresultaten. Indien slechts één van de opties gekozen wordt, wijken de resultaten voor F# onvermijdelijk te fel af van de andere geteste implementaties.
Om te beginnen geeft Figuur 243 de resultaten weer voor 1000 requests voor 22 rijen.



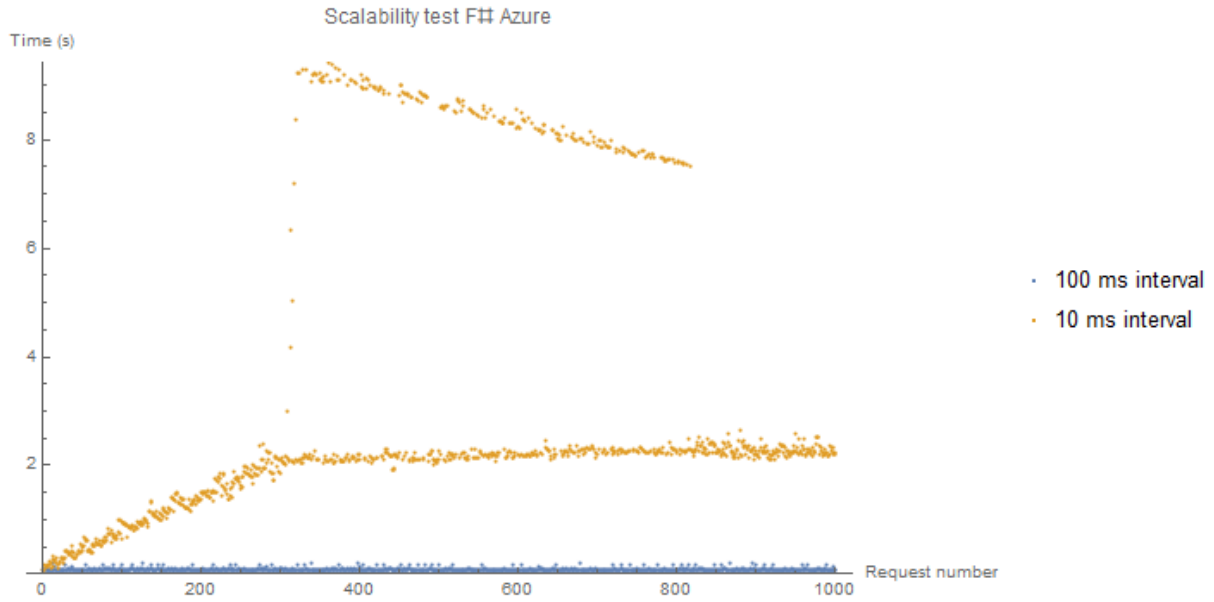
Figuur 243: Schaalbaarheid F# Azure voor 1000 requests en 22 rijen.

Figuur 243 laat eigenaardige resultaten zien. De schaalbaarheid voor F# lijkt zich compleet anders te gedragen dan die van C# in Azure. Zoals te verwachten worden de requests met een 100ms interval probleemloos afgehandeld. De request met een 10 ms interval laten echter opmerkelijke resultaten zien. Om te beginnen wordt een groot deel van de requests in minder dan een seconde afgehandeld. Het valt op dat over het gehele testinterval dit voor een groot deel van de requests zo is. Wat ook opvalt is dat de wachttijd voor een deel van de requests in het begin pijlsnel omhoog schiet, alsof ze in een lange wachtrij terechtkomen. Dit is echter maar voor een deel van de requests het geval. Rond request nummer 600 daalt de wachttijd ineens enorm tot ongeveer 2 tot 3 seconden. Tegen request nummer 900 stijgt de wachttijd ineens weer tot ongeveer 5 seconden. Merk op dat dit steeds slechts voor een deel van de requests is, en dat de pieken telkens een dalende trend vertonen. Dit alles wijst erop dat F# in Azure goed schaalbaar is, en dat de binnenkomende requests worden verdeeld over verschillende instanties. Het duurt even voor deze verdeling optimaal is, waardoor in het begin een deel van de requests een lange wachttijd heeft. Naar het midden van de test toe lijkt het verkeer veel beter gebalanceerd te zijn, waardoor geen enkele request een overdreven lange responstijd heeft. Op het einde kan het inkomende verkeer toch weer te veel worden voor de aanwezige instanties, waardoor er terug een iets grotere wachttijd ontstaat voor een deel van de requests. Tabel 30 geeft de statistische gegevens weer van deze test.

Tabel 30: Gemiddelde en standaardafwijking van de resultaten uit Figuur 243.

Interval	**Gemiddelde**	**Standaardafwijking**
100 ms	0,09 s	0,02 s
10 ms	2,8 s	3,3 s

Het gemiddelde voor het 100 ms interval is bijzonder laag, zelfs onder de ideale rekenduur van 100ms. Dit benadrukt de nood om deze test nog eens te doen met 23 rijen. Het gemiddelde voor het 10ms interval is een respectabele 2,8s. De standaardafwijking daarentegen is zeer hoog. Daarom is het interessant deze test onmiddellijk na de eerste iteratie opnieuw te doen, om te zien of Azure het verkeer in een volgende test beter kan balanceren, wat consistentere resultaten zou moeten opleveren. Figuur 244 geeft het resultaat van deze tweede iteratie weer.



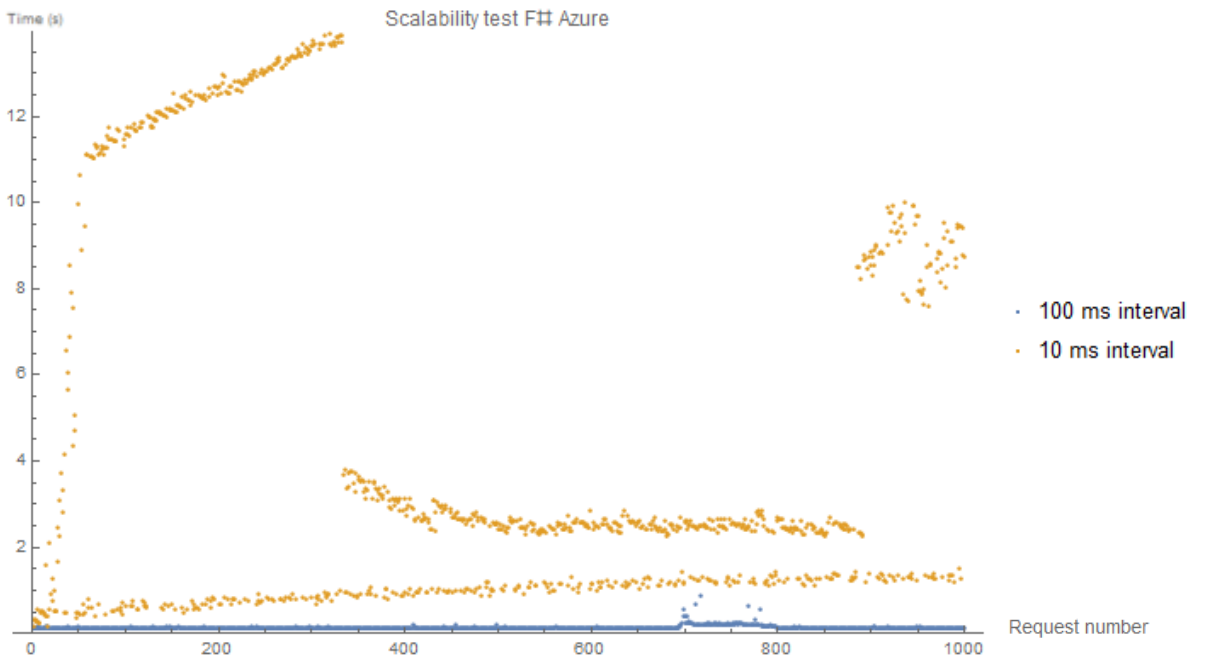
Figuur 244: Schaalbaarheid F# Azure voor 1000 requests en 22 rijen, direct uitgevoerd na de eerste test.

De tweede iteratie ziet er veel minder chaotisch uit. Het vermoeden van hierboven dat Azure naar de ideale verdeling van het verkeer over de verschillende instanties zoekt, lijkt dus te kloppen. De responstijd voor 10ms gaat lineair omhoog voor de eerste 300 requests, wat aangeeft dat de aanwezige instanties al het binnenkomend verkeer tot op dat punt net niet kunnen afhandelen. Bij request nummer 300 ontstaat opnieuw een splitsing van de requests in een deel dat snel kan worden afgehandeld, en een deel dat veel langer moet wachten, namelijk opnieuw ongeveer 8 seconden. Dit laatste deel daalt wel snel in wachttijd, en tegen request nummer 800 lijkt de schaling klaar te zijn, waardoor alle requests in ongeveer 2 seconden kunnen worden afgehandeld. Dit geeft aan dat F# in Azure op zich zeer schaalbaar is, maar dat Azure zelf de instanties van de applicatie zodanig probeert te balanceren, dat er nooit meer dan 2 seconden nodig zijn voor het afhandelen van de request. Tabel 31 geeft de statistische gegevens weer van deze tweede iteratie.

Tabel 31: Gemiddelde en standaardafwijking van de resultaten uit Figuur 244

Interval	**Gemiddelde**	**Standaardafwijking**
100 ms	0,09 s	0,02 s
10 ms	3,1 s	2,7 s

De statistische gegevens ondersteunen de hierboven uiteengezette theorie. De resultaten voor 100ms interval zijn identiek aan die van de eerste iteratie. Die van 10ms interval zijn wel anders. Het gemiddelde is met 10% gestegen van 2,8 naar 3,1. De standaardafwijking is echter gedaald van 3,3 naar 2,7 seconden. Zeker in verhouding tot het gemiddelde geeft dit aan dat de resultaten veel minder gespreid zijn. Dit alles wijst sterk in de richting van het feit dat F# zeer schaalbaar is, maar dat Azure zelf de requests in ongeveer 2seconden probeert af te handelen en op die manier de schaalbarheid controleert. De tests voor 23 rijen kunnen hierin uitsluitsel brengen, aangezien deze veel meer rekenintensief zijn. Figuur 245 geeft het resultaat van een eerste schaalbaarheidstest voor F# in Azure voor een driehoek van 23 rijen.



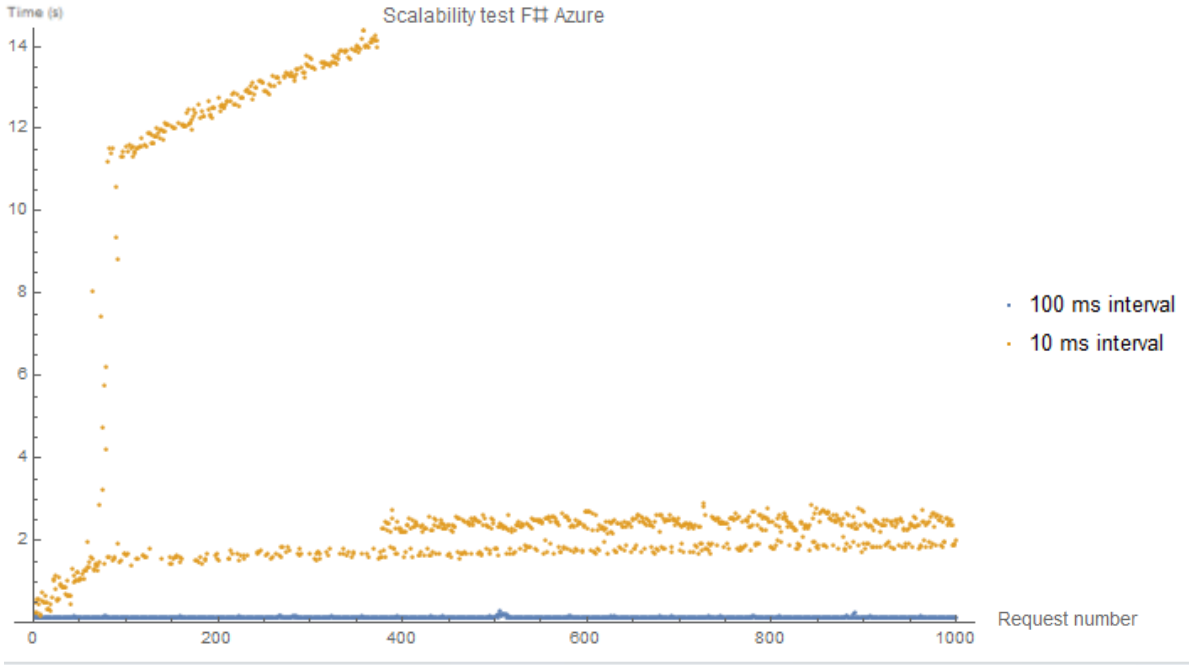
Figuur 245: Schaalbaarheid F# Azure voor 1000 requests en 23 rijen.

De resultaten uit Figuur 245 zijn behoorlijk interessant. Om te beginnen blijkt Azure voor het 100 ms interval nagenoeg volledig stabiel te zijn. Op een enkele plaats stijgt de wachttijd merkbaar, waarschijnlijk omdat de rekentijd nu net iets groter is dan het tijdsinterval tussen de verschillende requests. Deze lichte stijging is echter zeer snel gecorrigeerd.
De resultaten voor 10 ms lijken zeer sterk op die uit Figuur 243. Het belangrijkste verschil is dat nu de wachttijd tot 14 seconden stijgt, wat nog geen verdubbeling is. Dit wijst ook weer op goede schaalbaarheid. Ook gaat de wachttijd voor de ongelukkige requests in stijgende in plaats van dalende lijn. Dit heeft allicht vooral te maken met de dubbel zo grote rekenkracht die nodig is voor het afhandelen van deze requests ten opzichte van de tests voor een driehoek van 22 rijen. Verder valt op dat over de hele testrun opnieuw een deel requests een responstijd van slechts ongeveer een seconde heeft. Ook wordt nu na 300 requests al opgeschaald, in plaats van na 600 requests. Dit betekent opnieuw hoogstwaarschijnlijk dat Azure de schaling balanceert, en dat de F#-implementatie van de recursieve Driehoek van Pascal-API helemaal geen beperkingen heeft die de schaalbaarheid belemmeren. De reden dat de schaling hier sneller gebeurt is hoogstwaarschijnlijk dat Azure nu detecteert dat de gevraagde rekenkracht veel groter is, en dat er requests zijn die meer dan 10 seconden moeten wachten om afgehandeld te worden. Na de 300ste request is een zeer gelijkaardig patroon waarneembaar als in Figuur 243. De wachttijden zijn ongeveer gelijk, wat er opnieuw op wijst dat Azure het aantal instanties voor deze testrun hoger heeft genomen dan voor de vorige om zo een ideale compromis te sluiten tussen toegewezen resources en wachttijd voor de requests, en dat Suave en F# perfect kunnen meegaan hierin, in tegenstelling tot de bekomen resultaten voor C# in hetzelfde cloud-platform. Tabel 32 geeft de statistische resultaten voor deze test weer.

Tabel 32: Gemiddelde en standaardafwijking van de resultaten uit Figuur 245.

Interval	Gemiddelde	Standaardafwijking
100 ms	0,15 s	0,05 s
10 ms	4,5 s	4,4 s

Voor het 100 ms interval is voor 23 rijen ongeveer 0,06 seconden toegevoegd aan de rekentijd, wat exact te verwachten was. Ook de standaardafwijking is in orde. Deze resultaten zijn meer in lijn met de andere geteste cloud-platformen, in tegenstelling tot de resultateen voor 22 rijen en 100 ms. Voor het 10ms interval valt op dat het gemiddelde niet eens verdubbeld is ten opzichte van 22 rijen, wat ook weer op grote schaalbaarheid wijst. De standaardafwijking is wel weer zeer hoog in vergelijking met het gemiddelde. Daarom is het ook voor deze test aangewezen om opnieuw een identieke schaalbaarheidstest uit te voeren, direct na de eerste iteratie. Het resultaat hiervan is te zien in Figuur 246.



Figuur 246: Schaalbaarheid F# Azure voor 1000 requests en 23 rijen, direct uitgevoerd na de eerste test.

De resultaten van deze tweede iteratie zijn bijzonder gelijkaardig aan die uit Figuur 244. Opnieuw is er een lineaire stijging te zien, die nu wel reeds bij 100 requests afvlakt, in plaats van 300. Dit is logisch gezien de verdubbeling in rekenkracht die nodig is. De stijging in wachttijd is wel opnieuw groter, tot 14 seconden. Na 400 requests is de applicatie echter perfect geschaald, wat toch aangeeft dat het vermoeden uit de vorige tests juist moet zijn. De wachttijd voor de latere requests is opnieuw rond de 2 seconden. Hier is wel een onderscheid te zien in verschillende instanties of instantiegroepen, waarbij de ene iets sneller blijkt te zijn dan de andere. Dit performantieverschil is echter beperkt. Opnieuw is de schaling veel sneller voltrokken dan bij 22 rijen, wat erop wijst dat Azure de schaling volledig controleert, en F# en Suave op zich zeer goede schaalbaarheidskarakteristieken bezitten. Tabel 33 geeft ten slotte de statistische gegevens weer van deze test.

Tabel 33: Gemiddelde en standaardafwijking van de resultaten uit Figuur 246.

Interval	Gemiddelde	Standaardafwijking
100 ms	0,14 s	0,01 s
10 ms	4,2 s	4,3 s

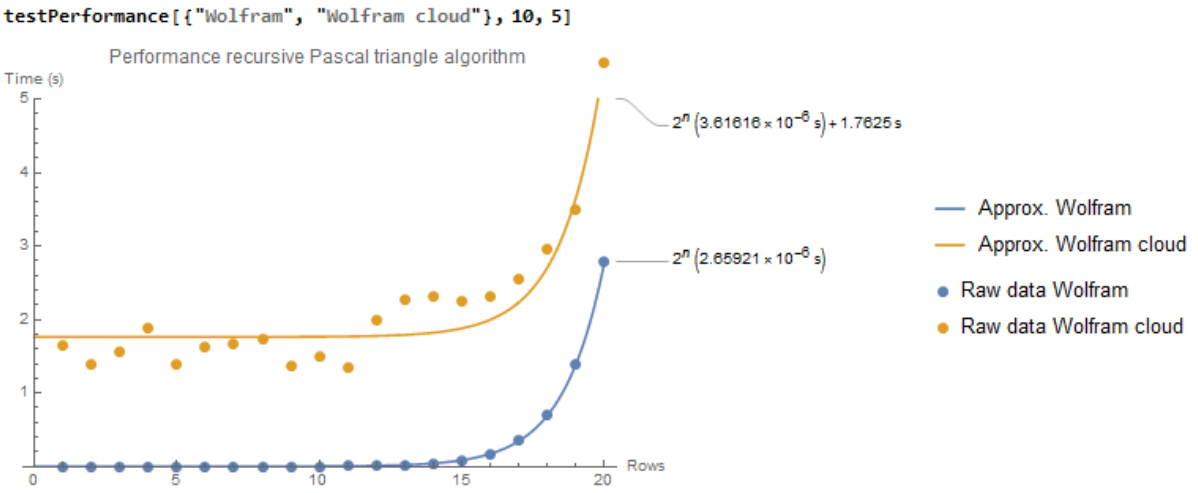
Voor het 100 ms interval is de performantie nagenoeg perfect. Over het 10 ms interal zijn echter weer een aantal dingen op te merken. Voor deze test is het gemiddelde ten opzichte van de eerste iteratie licht gedaald, wat opnieuw wijst op een betere distributie van het inkomend verkeer en betere schaling dan bij de eerste iteratie. De standaardafwijking is nu echter slechts licht gedaald. Dit lijkt op het eerste zicht vreemd, maar is uiteindelijk een bevestiging voor de zeer grote schaalbaarheid van F#. In Figuur 246 is immers te zien dat de requests voor deze test tot 14 seconden moeten wachten voordat de schaling volledig voltrokken is, terwijl dit voor de tests met 22 rijen slechts 8 seconden was. Na de schaling compleet is worden alle requests echter in ongeveer 2 seconden afgehandeld voor beide tests. Dit trekt het gemiddelde sterk naar beneden voor beide tests. In verhouding voor de tweede test echter nog sterker dan voor de eerste test omdat de uitschieters veel groter zijn. De standaardafwijking blijft echter behoorlijk hoog voor 23 rijen door het grote verschil tussen de uitschieters en de evenwichtssituatie. Uiteindelijk betekent dit dus dat de slechte verhouding tussen gemiddelde en standaardafwijking een bevestiging is van de grote schaalbaarheid van F# in Azure. Voor toepassingen waarbij grote pieken in de vraag ernaar kunnen ontstaan en schaalbaarheid in het algemeen van belang is, is de combinatie van F#, Suave, en Azure dan ook ten zeerste aan te raden.

12.6 Wolfram

Ten slotte rest het testen van de Wolfram cloud-implementatie. Deze paragraaf bespreekt de resultaten voor deze implementatie op dezelfde manier als alle andere implementaties in dit hoofdstuk. Voor Wolfram is er enkel een cloud-implementatie voorzien in de Wolfram Cloud, aangezien dit het enige ondersteunde cloud-platform is voor deze taal, zoals eerder beschreven in paragraaf 11.4.

12.6.1 Connectiesnelheid en performantie

Eerst en vooral is er ook voor Wolfram een vergelijking gemaakt tussen de performantie van de standaard sequentiële implementatie van het recursieve Driehoek van Pascal-algoritme in de cloud, en lokale variant. Het resultaat hiervan is te zien in Figuur 247.



Figuur 247: Vergelijking connectiesnelheid en performantie voor Wolfram in de Wolfram Cloud en de Desktop-versie voor 10 iteraties.

De resultaten van bovenstaand experiment zijn niet al te rooskleurig. Ten eerste is de connectiesnelheid voor de Wolfram Cloud zeer groot. Bij dit experiment zelfs 1,8 seconden gemiddeld. Dit is onacceptabel voor het bouwen van serieuze applicaties, zeker bij het gebruik van microservices of een andere SOA-architectuur die steunt op het combineren van meerdere onafhankelijk gedeployde services voor het verwerken van een request. In dat geval zou de laadtijd van een applicatie immers al snel meerdere seconden kunnen bedragen, wat de dag van vandaag veel te veel is, zeker gezien de vele andere in deze thesis reeds bestudeerde cloud-omgevingen die het veel beter doen op dit vlak.
Verder valt ook op dat de connectiesnelheid nogal variabel is. In dit ene experiment zijn waardes tusen de 1 en 2,5 seconden te zien. Dit geeft aan dat de Wolfram Cloud geen al te betrouwbaar platform is wat connectiesnelheid betreft.
Naar performantie toe scoort de Wolfram Cloud echter aanzienlijk beter. Er is niet al te veel verschil tussen de Wolfram Cloud en de lokale test-pc naar pure rekenkracht toe. Tabel 34 geeft de exacte relatieve peformantie van de Wolfram Cloud weer ten opzichte van de desktop-versie.

Tabel 34: Vergelijking relatieve performantie Wolfram Cloud.

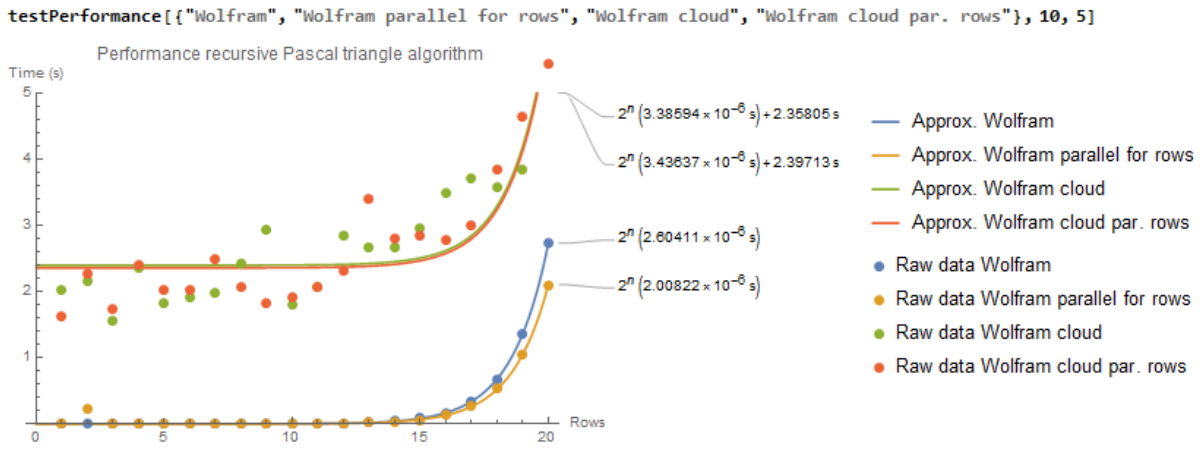
Cloud-omgeving	**Relatieve performantie t.o.v. desktop-versie**
Wolfram Cloud	$\frac{3{,}6 \cdot 10^{-6}}{2{,}7 \cdot 10^{-6}} \approx 1{,}3$

De Wolfram Cloud blijkt slechts een 30% trager te zijn dan de desktop-implementatie. Gezien de test-pc over een behoorlijk krachtige processor beschikt voor single-threaded applicaties zoals eerder vermeld, is dit een behoorlijk indrukwekkend resultaat. De hoge connectietijd is dan toch wel echt jammer aangezien deze het potentieel van de anders sterke cloud-infrastructuur toch echt belemmert. Merk wel op dat deze connectiesnelheid sterk afhankelijk is van de locatie van de server en client. Aangezien Wolfram een vrij nieuw fenomeen is, is de infrastructuur van de Wolfram Cloud allicht nog maar op zeer beperkte

locaties verkrijgbaar, waardoor voor deze tests die vanuit West-Europa zijn uitgevoerd het aannemelijk is dat de connectietijd zeer hoog is. Voor requests verstuurd vanuit bijvoorbeeld Noord-Amerika is connectiesnelheid voor dit cloud-platform misschien geen enkel probleem. Meer onderzoek is nodig om te bepalen of dit effectief zo is. Als Wolfram echter in populariteit toeneemt de komende jaren, is te verwachten dat de connectiesnelheid in de toekomst sterk kan verbeteren.

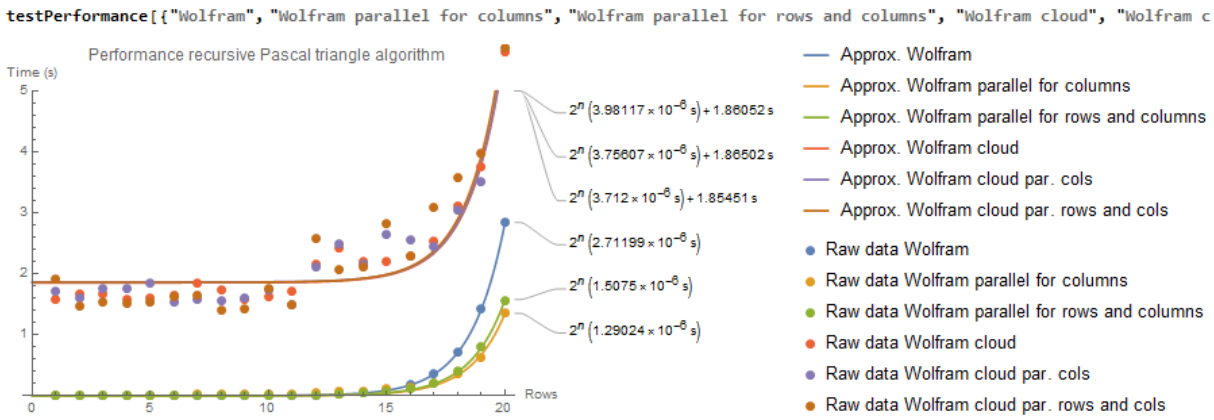
12.6.2 Parallellisatie

Ook voor Wolfram zijn alle geparallelliseerde varianten van de Driehoek van Pascal in de cloud getest en vergeleken met de desktop-toepassing. Uit paragraaf 1969.4.6 is geweten dat parallellisatie met een Wolfram Cloud Platform-licentie niet evident is. Daarom is het aangewezen om voorzichtig te werk te gaan bij het testen en dus te beginnen met enkel de versie met geparallelliseerde rijen in de cloud te vergelijken met de desktop-versie, in plaats van zoals voor de meeste andere geteste implementaties rechtstreeks alle cloud-gedeployde parallellisatiemethodes naast elkaar te zetten. De resultaten van deze test zijn te zien in Figuur 248.



Figuur 248: Vergelijking parallelle rijen met de sequentiële versie in de Wolfram cloud en voor Wolfram op de lokale pc.

De resultaten uit Figuur 248 zijn bedroevend. Zoals gevreesd lijkt parallellisatie in de Wolfram Cloud helemaal niet ondersteund te zijn. De resultaten zijn heel variabel, en bovendien identiek aan die van de sequentiële versie. Dit is een gemiste kans voor de Wolfram Cloud, zeker gezien de single-threaded performantie zo goed is. Voor de volledigheid zijn ook de andere parallellisatiemethoden getest in de Wolfram Cloud, hoewel de verwachtingen hiervan gebaseerd op bovenstaande figuur minimaal zijn. Figuur 249 geeft de resultaten weer.



Figuur 249: Vergelijking parallelle kolommen en rijen en kolommen met de sequentiële versie in de Wolfram cloud en voor Wolfram op de lokale pc.

Ook de andere parallellisatiemethoden geven dezelfde resultaten als de parallellellisatiemethode voor enkel rijen, zoals verwacht. Van parallellisatie in de Wolfram Cloud is dus jammer genoeg geen sprake. Uit paragraaf 9.4.6 valt af te leiden dat dit een bewuste beperking is die de ontwikkelaars van Wolfram hebben gelegd op de Wolfram Cloud, wat ten zeerste te betreuren is. Het is wel belangrijk om op te merken dat Wolfram wel toelaat functies die gebruik maken van parallellisatie te deployen in de cloud. Bij de uitvoering negeert de Wolfram Kernel deze functies echter gewoon in de Wolfram Cloud. Op geen enkele manier wordt de programmeur op de hoogte gesteld van deze onbestaande parallellisatie in de Wolfram Cloud, zelfs niet als hij expliciet geparallelliseerde functies in de Wolfram Cloud deployt. Dit is uiteraard vervelend en kan in veel gevallen een echte anticlimax zijn, zeker aangezien Wolfram zichzelf adverteert als 'zeer parallelliseerbaar'. Langs de andere kant zijn bovenstaande vaststellingen ook min of meer goed nieuws, aangezien de expliciet ingevoerde afwezigheid van ondersteuning voor parallellisatie in de Wolfram Cloud en het gebrek aan communicatie hierover aan zouden kunnen geven dat paralllelisatie technisch wel mogelijk is in de Wolfram Cloud, maar voorlopig om economische redenen nog niet toegelaten is. Zoals in vorige paragraaf reeds aangehaald is Wolfram –en zeker de Wolfram Cloud- nog relatief nieuw. Hierdoor staat de infrastructuur van dit cloud-platform waarschijnlijk nog niet op punt, en is ze waarschijnlijk (nog) niet uitgebreid genoeg om een groot aantal CPU-kernen aan een enkele applicatie toe te kunnen wijzen, zonder dat andere applicaties in de Wolfram Cloud met een tekort aan resources te kampen krijgen. De in vorige paragraaf vastgestelde hoge connectietijd voor de Wolfram Cloud ondersteunt deze theorie verder.
Bovenstaande is echter hoofdzakelijk speculatie gebaseerd op de in dit onderzoek gemaakte vaststellingen. Het is in het slechtste geval natuurlijk toch mogelijk dat een diep gewortelde technische beperking aan de basis ligt van het verbod op parallellisatie in de Wolfram Cloud, zeker aangezien ook op de lokale test-pc Wolfram slecht scoort voor parallellisatie, zoals paragraaf 9.4.6 reeds aantoonde. Dit is vanuit het perspectief van dit onderzoek moeilijk te zeggen. Het is echter aannemelijk dat indien de Wolfram Cloud ooit parallellisatie zou toelaten, de performantiewinst in de cloud niet indrukwekkend zou zijn.

12.6.3 Schaalbaarheid

Ten slotte is ook voor Wolfram de schaalbaarheid van de Driehoek van Pascal-API in de Wolfram Cloud onder de loep genomen. Net zoals voor de andere implementaties moet eerst het ideale aantal rijen voor de request worden uitgerekend, opnieuw volgens de vertrouwde procedure. Hieronder is de berekening voor Wolfram in de Wolfram Cloud weergegeven, gebaseerd op Figuur 247.

$$t_n \approx 3{,}6 \cdot 10^{-6} \cdot 2^n$$

$$n = \log 2 \frac{0{,}1}{3{,}6 \cdot 10^{-6}}$$

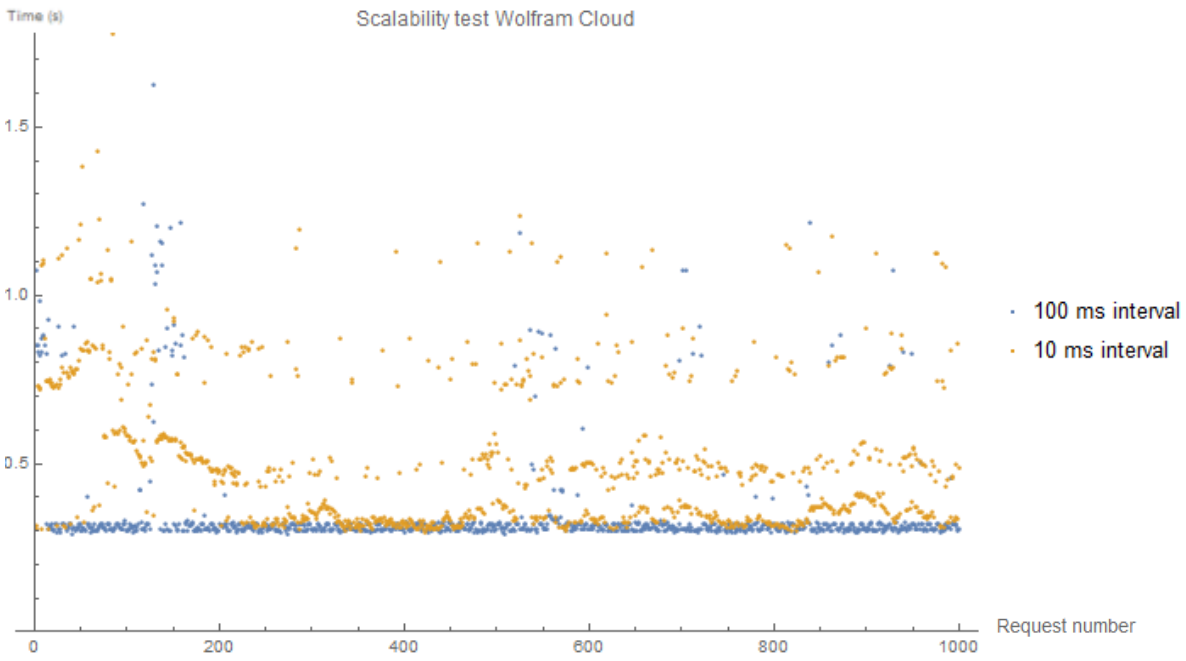
$$n = 14{,}76$$

$$=> n = 14$$

$$t \approx 3{,}6 \cdot 10^{-6} \cdot 2^{14}$$

$$t \approx 0{,}06s$$

Het ideale aantal rijen is dus 14. Dit geeft een tijd van 0,06 seconden per berekening, wat overeenkoment met 6 overlappende requests bij een 10 ms interval.
Figuur 250 geeft het resultaat van een eerste schaalbaarheidstest voor 1000 requests weer.



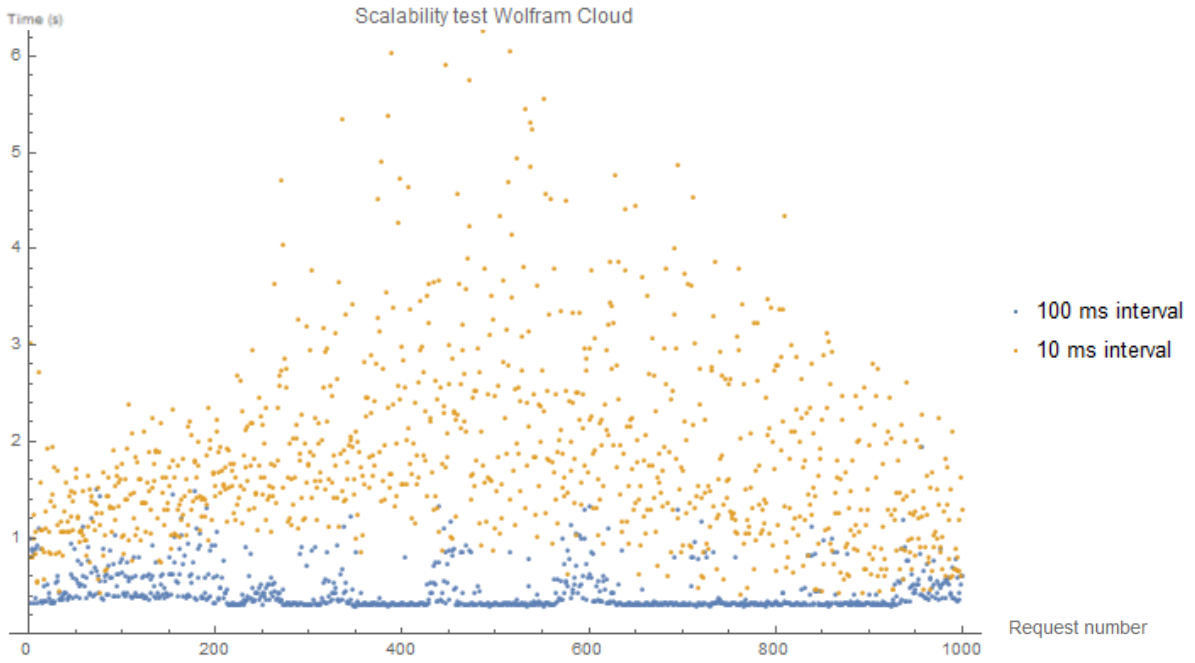
Figuur 250: Schaalbaarheid Wolfram cloud voor 1000 requests.

Deze resultaten blijken zeer positief te zijn. De connectietijd is voor een openstaande connectie om te beginnen zeer lang, maar dit is reeds geweten uit paragraaf 12.6.1. Verder valt op dat een groot deel van de requests voor een 10 ms interval juist even snel afgehandeld is als de requests voor een 100 ms interval. Ook worden nagenoeg alle requests binnen de seconde afgehandeld, en zijn er geen grote pieken te zien in de afhandelingstijd. Dit alles wijst op een zeer goede schaalbaarheid van Wolfram in de Wolfram Cloud. Dit is te verklaren doordat Wolfram een geïnterpreteerde taal is in plaats van een gecompileerde. Er is dus geen enkel omvattend framework of iets dergelijks nodig om de Wolfram code in de cloud op te slaan, wat ook al duidelijk te zien was aan de implementatie van de Wolfram-versie van de recursieve Driehoek van Pascal-API, zoals beschreven in paragraaf 11.4. Omwille van dit feit kan eender welke resource waar een Wolfram Kernel op draait het blokje cloud-gedeployde Wolfram-code dus rechtstreeks uitvoeren, zonder enige extra opstarttijd of initialisatie. Dat is één van de grote voordelen van de geïnterpreteerde natuur van Wolfram. Ondanks het feit dat Wolfram dus vrij traag is, maakt Wolfram dit in de Wolfram Cloud goed door de zeer goede schalingskarakteristieken. Tabel 35 geeft de statistische resultaten weer van bovenstaand experiment.

Tabel 35: Gemiddelde en standaardafwijking van de resultaten uit Figuur 250.

Interval	**Gemiddelde**	**Standaardafwijking**
100 ms	0,35 s	0,16 s
10 ms	0,51 s	0,22 s

De gemiddelde tijd nodig voor het afhandelen van de requests met een 100 ms interval is zoals eerder toegelicht behoorlijk hoog. De standaardafwijking is ook aan de hoge kant. Dit wijst op een inconsistente performantie van de cloud. Het opmerkelijkste resultaat is dat de tijd voor het afhandelen van de requests met een 10 ms interval slechts iets hoger is dan die voor het 100 ms interval. Ook de standaardafwijking blijft goed binnen de perken. De Wolfram Cloud is statistisch gezien dus ook zeer schaalbaar.
Omwille van de eerder vastgestelde sterk wisselende performantie van de Wolfram cloud, en de uit bovenstaande testresultaten af te leiden inconsistente afhandeling van de requests met een 100 ms interval is geopteerd om het schaalbaarheidsexperiment nog enkele keren uit te voeren op andere tijdstippen, om te zien of er een variatie in performantie te zien is. Ook is geëxperimenteerd met het aantal rijen te verhogen tot 15, waardoor de rekentijd per request 120ms wordt. Opvallend is dat dan nog steeds grotendeels identieke resultaten zijn bekomen voor de meeste van deze tests als in Figuur 250. Uiteindelijk blijkt toch ook dat soms de testresultaten minder positief zijn, maar nog steeds zeer goed. Een voorbeeld van één van de slechtst verkregen resultaten is Figuur 251. Deze resultaten zijn verkregen voor een schaalbaarheidstest van 1000 requests voor een driehoek van 15 rijen.



Figuur 251: Slechtst bekomen resultaat voor de schaalbaarheidstest in de Wolfram Cloud voor 15 rijen.

Het valt op dat de resultaten voor 10 ms interval hier veel meer gespreid zijn dan bij de vorige resultaten. Toch blijft de totale wachttijd voor het overgrote merendeel van de requests onder de 4 seconden, wat zeer indrukwekkend is gezien de 120 ms rekentijd per request, wat leidt tot maar liefst 12 overlappende requests bij een 10 ms tijdsinterval tussen het versturen ervan. Ten slotte valt op dat voor het 100 ms interval ook enkele pieken te zien zijn. Dit kan een combinatie zijn van het feit dat de berekening nu langer duurt, waardoor schaalbaarheid ook voor het afhandelen van de requests met een 100 ms interval nodig is, en het feit dat er op dat specifieke moment onvoldoende schaalbaarheid ter beschikking was. Uiteindelijk gaat de nodige tijd voor het 10 ms interval ook terug richting die voor het 100ms interval, wat erop wijst dat er wel degelijk voldoende schaling in het Wolfram Cloud systeem zit, en dat er waarschijnlijk juist op dat moment meer resources vrijkomen doordat andere applicaties bvb. toevallig op dat moment minder resources vragen van de Wolfram Cloud. Tabel 36 geeft de statistische resultaten weer van deze laatste test.

Tabel 36: Gemiddelde en standaardafwijking van de resultaten uit Figuur 251.

Interval	**Gemiddelde**	**Standaardafwijking**
100 ms	0,44 s	0,22 s
10 ms	1,9 s	0,95 s

Het gemiddelde voor vooral het 10ms interval ligt een stuk hoger dan bij de vorige resultaten. Toch is dit nog zeer acceptabel, zeker in vergelijking met de andere cloud-platformen. Het besluit is dus dat Wolfram een zeer schaalbare programmeertaal is in de cloud, en door deze indrukwekkende schaalbaarheid voor specifieke applicaties waar schaalbaarheid essentieel is grotendeels goedmaakt voor de eerder in dit onderzoek vastgestelde gebreken van deze taal en dit cloud-platform.

12.7 Algemeen

In vorige paragrafen is opgevallen dat de resultaten van de verschillende cloud-testen uit dit onderzoek even talrijk als uiteenlopend zijn. Daarom geeft deze paragraaf een algemeen overzicht waarbij alle geteste implementaties met elkaar vergeleken worden voor elk van de geteste criteria, om zo een beter inzicht te krijgen in de relatieve scores van elke taal en elk cloud-platform voor alle tests uit dit onderzoek ten opzichte van de andere geteste talen en platformen, en een algemeen besluit te kunnen vormen.

12.7.1 Connectietijd

De connectietijd is één van de belangrijkste parameters voor eender welke webapplicatie. Deze is echter afhankelijk van een groot aantal factoren. De relatieve connectietijd van de ene applicatie ten opzichte van de andere is bijna uitsluitend afhankelijk van het geteste cloud-platform, indien de tests vanop eenzelfde pc op een gelijkaardig tijdstip uitgevoerd zijn, wat voor dit onderzoek het geval is. Ook de locatie van de test-pc is uiteraard een belangrijke factor. Onrechtstreeks is deze factor echter ook te linken aan de kwaliteit van het geteste cloud-platform, aangezien een goed platform steeds de mogelijkheid zou moeten bieden applicaties geografisch zo dicht mogelijk bij de beoogde gebruikers ervan te deployen.
Om bovenstaande redenen zegt een kwalitatieve vergelijking van connectietijden voor de verschillende implementaties meer dan enkel naar de kwantitatieve data voor elke implementatie op zich te kijken. Tabel 37 geeft een overzicht van alle connectietijden voor alle geteste cloud-implementaties.

Tabel 37: Vergelijking connectietijd voor alle geteste cloud-implementaties.

Cloud platform	**Programmeertaal / Framework**	**Connectietijd (s)**
Google App Engine std. Env.	Java 7 OO	0,25 - 0,3
Google App Engine flex. Env.	Java 8 OO	0,2
	Java 8 met Streams	0,2
Google compute engine	C# .NET 4.5	0,2
	C# met LINQ .NET 4.5	0,15
Azure	C# .NET Core	0,15 - 0,2
	C# met LINQ .NET Core	0,15
	F# Suave	0,1
Wolfram Cloud	Wolfram	1,3 – 2,4

De meetse cloud-implementaties hebben een aanvaardbare connectietijd van 0,15 tot 0,2 seconden gemiddeld. Enkel de Google App engine Standard Envrionment doet het gevoelig slechter, maar de prestaties zijn nog acceptabel. Wolfram is de enige taal die echt uit de boot valt, met een connectietijd van 1,3 tot een absurd hoge 2,4 seconden. Het spreekt voor zich dat dit een grote handicap is voor het Wolfram Cloud platform. Zoals in vorige paragraaf beschreven is het wel zeer goed mogelijk dat de connectietijd in de toekomst nog sterk zal verbeteren, aangezien Wolfram nog volop aan het groeien is. Ook moet opnieuw opgemerkt

worden dat zoals hierboven beschreven de connectietijd sterk afhankelijk is van veel factoren die niet rechtstreeks aan de toegepaste programmeertaal of zelfs het gebruikte cloud-platform gebonden kunnen worden. Desondanks is er wel een indirect verband tussen programmeertaal en connectiesnelheid, aangezien de beschikbare cloud-platformen en de kwaliteit ervan wel van de taal afhankelijk zijn. Bij Wolfram is er geen alternatief voor de Wolfram cloud, waardoor de conclusies van deze test wel degelijk een doorslaggevende impact hebben op de kwaliteit van Wolfram als programmeertaal voor cloud computing.
Ten slotte is het nog belangrijk op te merken dat F# de snelste taal blijkt te zijn wat betreft connectiesnelheid, ondanks het feit dat deze taal in dezelfde cloud-omgeving is gebruikt als de C# varianten, namelijk Azure. Het gebruikte Framework zou dus ook een vrij belangrijke impact kunnen hebben op de connectietijd, hoewel ook toevalligheden zoals een geografisch meer nabijgelegen server een belangrijke rol kunnen gespeeld hebben bij het bekomen van dit resultaat. Dit is moeilijk te zeggen vanuit het perspectief van dit onderzoek. Verder onderzoek is dus nodig om hier uitsluitsel over te geven.
De tests uit dit onderzoek duiden F# in de Azure cloud dus aan als beste geteste implementatie wat betreft connectiesnelheid, hoewel nuancering van dit resultaat noodzakelijk is. Nagenoeg alle geteste implementaties zijn vergelijkbaar wat betreft dit aspect. Enkel Java 7 in de Google App Engine Standard Environment is iets trager, en Wolfram valt volledig uit de boot.

12.7.2 Performantie

Ook voor de performantie van elke implementatie in de cloud is het interessant een algemeen overzicht te scheppen van de resultaten uit de vorige paragrafen. Dit aspect is veel eenduidiger te vergelijken over de verschillende cloud-implementaties heen dan de connectiesnelheid. Aangezien de relatieve performantie van elke programmeertaal op zich al uitvoerig besproken is in hoofdstuk 8, focust deze paragraaf zich op de relatieve performantie van elke cloud-implementatie in vergelijking met de desktop-versie voor elke taal. Hierdoor zijn de bekomen resultaten in deze paragraaf enkel en alleen afhankelijk van het gebruikte cloud-platform. Tabel 38 geeft hier de resultaten van weer.

Tabel 38: Vergelijking performantie voor alle geteste cloud-implementaties.

Cloud platform	**Programmeertaal / Framework**	**Performantie rel. t.o.v. desktop-versie**
Google App Engine std. Env.	Java 7 OO	14%
Google App Engine flex. Env.	Java 8 OO	67%
	Java 8 met Streams	64%
Google compute engine	C# .NET 4.5	63%
	C# met LINQ .NET 4.5	68%
Azure	C# .NET Core	29%
	C# met LINQ .NET Core	28%
	F# Suave	29%
Wolfram Cloud	Wolfram	77%

De resultaten uit Tabel 38 zijn uiteenlopend en interessant. Om te beginnen scoort het Google App Engine Standard Environment opnieuw slecht. Ditmaal het slechtst van allemaal. Met slechts een zevende van de uitvoerinsgssnelheid van de Desktop-versie van het recursieve Driehoek van Pascal-algoritme in Java is deze cloud-omgeving dan ook niet aan te raden voor applicaties die rekenintensief zijn. Alle andere implementaties in het Google Cloud Platform scoren daarentegen wel goed. 70% van de uitvoeringssnelheid van de lokale versie is respectabel, aangezien de test-pc over een zeer krachtige processor beschikt voor singe-threaded rekentaken.
De performantie van Azure valt verder nogal tegen. Om en bij de 30% van de lokale versie is nog net acceptabel te noemen, maar meer performantie is zeker wenselijk. Zoals te verwachten zijn de resultaten wat performantie betreft voor alle implementaties in de Azure App Service op een kleine afwijking binnen de foutenmarge na identiek. De Wolfram Cloud komt verrassend als snelste Cloud-platform uit de hoek, met een zeer respectabele 77% van de desktop-versie. Dit compenseert gedeeltelijk voor de vreselijke connectiesnelheid, althans voor rekenintensieve applicaties. Toch is Wolfram in de Wolfram Cloud voor dit soort applicaties zeker niet aan te bevelen, aangezien uit paragraaf 8.1 geweten is dat de performantie voor Wolfram als taal zeer laag ligt ten opzichte van de andere talen. De echte winnaars van deze test zijn dus het Flexible Environment van de Google App Engine voor Java 8 en de Google Compute Engine voor C# met .NET 4.5. Deze talen en cloud-platformen zijn dus aan te raden voor applicaties waar performantie een doorslaggevende rol speelt. Merk op dat ook in de cloud-omgevingen de performantie van functionele implementaties in Java en C# ongeveer even perforamant zijn als de OO-versies.
Ten slotte valt op dat geen enkel cloud-platform wat betreft performantie het beter doet dan de lokale test-pc. Dit geeft aan dat voor rekenintensieve applicaties native implementaties die rechtstreeks op de pc van de gebruiker draaien nog steeds de voorkeur hebben, behalve wanneer veel gebruikers enkel beperkte hardware ter beschikking hebben, zoals smartphones of chromebooks.

12.7.3 Parallellisatie

Ook voor de resultaten van de eerder in dit hoofdstuk beschreven parallellisatietesten in de cloud is een handig overzicht gemaakt voor alle cloud-implementaties. Dit aspect is zowel afhankelijk van de toegepaste programmeertaal als van het gebruikte cloud-platform, en is makkelijk transparant te testen voor alle in dit onderzoek beschouwde implementaties..
Tabel 39 geeft hiervan de resultaten weer voor alle geteste parallellisatiemethodes als de relatieve performantiewinst van de geparallelliseerde versie ten opzichte van de sequentiële versie, uitgedrukt in procent.

Tabel 39: Vergelijking parallellisatie voor alle geteste cloud-implementaties.

Cloud platform	Programmeertaal / Framework	Parallelle rijen	Parallelle kolommen	Parallelle rijen en kolommen
Google App Engine std. Env.	Java 7 OO	-11%	<< 0%	<< 0%
Google App Engine flex. Env.	Java 8 OO	61%	131%	146%
	Java 8 met Streams	30%	200%	200%
Google compute engine	C# .NET 4.5	84%	148%	159%
	C# met LINQ .NET 4.5	68%	248%	204%
Azure	C# .NET Core	67%	255%	233%
	C# met LINQ .NET Core	88%	319%	307%
	F# Suave	91%	251%	261%
Wolfram Cloud	Wolfram	0%	0%	0%

De resultaten uit Tabel 39 zijn vrij uiteenlopend. Zoals eerder vastgesteld is parallellisatie in het Google App Engine Flexible Environment voor Java 7 wel mogelijk, maar verslechtert dit de performantie van de applicatie. Indien een groot aantal threads wordt aangemaakt is de impact op de performantie zelfs drastisch. Dit maakt van deze cloud-omgeving de slechtst parallelliseerbare versie van ze allemaal.
Net zoals het Standard Evironment van de Google App Engine voor Java 7 scoort ook Wolfram Cloud zeer slecht wat betreft het parallelliseren van je toepassing. Voor deze taal is parallellisatie door de ontwikkelaars bewust uitgeschakeld in de cloud, zoals in paragraaf 12.6.2 in detail beschreven is.
Een positief, opvallend patroon is dat alle andere functionele implementaties in de cloud beduidend beter parallelliseerbaar blijken dan hun object-georiënteerde tegenhangers. De enige uitzondering hierop is Java met Streams in Google App Engine Flexible Environment voor geparallelliseerde rijen. De performantie van dit specifieke resultaat is veel lager dan van de andere. Voor de desktop-versie was dit echter ook al zo, zoals beschreven in paragraaf 9.4.2. Dit betekent dus dat dit resultaat zeker geldig is. Verder onderzoek is nodig om de exacte oorzaak hiervan te bepalen, aangezien dit buiten het bestek van deze masterproef valt.
F# blijkt in de Azure cloud de best parallelliseerbare implementatie te zijn, samen met C# met LINQ in Azure. Deze laatste is trouwens de enige implementatie die in de cloud een grotere relatieve performantiewinst weet te bereiken ten opzichte van de desktop-implementatie, hoewel het verschil binnen de foutenmarge ligt.
Ook moet opgemerkt worden dat in het algemeen de parallellisatie in de cloud tegenvalt. Uit de literatuurstudie leek het immers zo dat in de cloud grote hoeveelheden resources voor korte tijd aan een enkele applicatie zouden kunnen toegewezen worden, waardoor in theorie parallellisatie een veel grotere performantiewinst zou kunnen opleveren in de cloud dan op de meeste pc's van gebruikers haalbaar is. Dit blijkt jammer genoeg in de praktijk niet zo.
Wel is het belangrijk om op te merken dat zoals eerder beschreven het aanmaken van een gepaste betalende account het aantal toekenbare resources voor applicaties in sommige

cloud-platformen drastisch kan verhogen. Dit onderzoek is beperkt tot de gratis proefversies voor alle cloud-platformen, behalve Wolfram. Met een gepast subscriptieplan voor andere platformen zouden de resultaten uit Tabel 39 dus drastisch beter kunnen zijn voor met name alle geteste implementaties in het Flexible Environment van de Google App Engine en de Google Compute Engine.

12.7.4 Schaalbaarheid

Het laatste geteste aspect van de verschillende cloud-implementaties uit dit onderzoek is schaalbaarheid. Dit aspect is zowel afhankelijk van de toegepaste programmeertaal als van het gebruikte cloud-platform, en is voor dit onderzoek bijzonder interessant aangezien zowel functioneel programmeren als cloud-computing geprezen worden voor hun grote schaalbaarheid. Hierover geeft Tabel 40 uitsluitsel. Deze tabel geeft een kwalitatieve beoordeling van de schaalbaarheid voor elk cloud-platform en elke programmeertaal behandeld in dit onderzoek, gebaseerd op de bevindingen van de schaalbaarheidstesten uit vorige paragrafen weer. Zo is dit aspect makkelijk te vergelijken voor de verschillende geteste implementaties.

Tabel 40: Vergelijking schaalbaarheid voor alle geteste cloud-implementaties.

Cloud platform	**Programmeertaal / Framework**	**Schaalbaarheid**
Google App Engine std. Env.	Java 7 OO	Goed
Google App Engine flex. Env.	Java 8 OO	Acceptabel
	Java 8 met Streams	Acceptabel
Google compute engine	C# .NET 4.5	Geen
	C# met LINQ .NET 4.5	Geen
Azure	C# .NET Core	Weinig
	C# met LINQ .NET Core	Weinig
	F# Suave	Goed
Wolfram Cloud	Wolfram	Zeer goed

Deze laatste test geeft opnieuw zeer uiteenlopende resultaten voor de verschilledne cloud-platformen en implementaties. Om te beginnen valt op dat de verschillende omgevingen binnen het Google Cloud Platform elk sterk verschillende schaalbaarheidseigenschappen hebben. Dit is allicht te wijten aan de sterk verschillende technologiën die achter deze omgevingen schuilgaan, zoals beschreven in paragraaf 10.2. Verrassend genoeg blijkt het Google App Engine Standard Environment voor Java 7 het best schaalbare platform te zijn binnen het Google Cloud Platform, ondanks het feit dat het voor alle andere geteste parameters achterop hinkte. Dit maakt van dit platform plots een keuze die het overwegen waard is voor single-threaded applicaties die geen grote rekenkracht vragen, maar wel schaalbar moeten zijn. Voor het Flexible Environment is acceptabel als beoordeling gegeven in Tabel 40, maar er moet opgemerkt worden dat dit gebeurd is op basis van een test die slechts 10 seconden duurt. Zoals in paragraaf 12.1.3.2 en de documentatie van Google zelf

reeds aangegeven, is het Flexible Environment waarschijnlijk behoorlijk schaalbaar, maar heeft de schaling vrij lang nodig om tot stand te komen. Hierdoor kan dit cloud-platform voor applicaties die geen pieken hebben in verkeer maar wel een min of meer constante grote stroom van inkomend verkeer ook een goede optie zijn. Verder onderzoek is nodig om hier uitsluitsel over te krijgen.
Geen van de geteste implementaties voor C# toont goede schaalbaarheidskarakteristieken. Dit is gedeeltelijk te wijten aan de cloud-platformen, maar kan ook iets te maken hebben met het .NET framework, aangezien verrassend genoeg F# in Azure veel betere parallellisatie laat zien dan C#. Hieruit blijkt dat F# opnieuw een zeer sterke programmeertaal is voor cloud computing. Dat kan aan de taal F# liggen maar ook aan het feit dat de Suave-biblotheek voor het maken van een REST-API veel meer lightweight is dan het .NET Core framework, wat betere schaalbaarheid toelaat aangezien het opstarten van nieuwe instanties veel sneller gaat voor minimalistische lightweight-toepassingen. Een nadeel van deze minimalistische aanpak is dat het gebruik van Suave meer kennis en toewijding vraagt van de programmeur dan het gebruik van .NET (Core), zoals reeds beschreven in paragraaf 11.3.1.
De grootste verrassing op het gebied van schaalbaarheid komt echter van Wolfram in de Wolfram Cloud. Ondanks de onbestaande parallellisatie in dit platform, blijkt de Wolfram Cloud de beste schaalbaarheid van alle geteste cloud-platformen te laten zien, met amper verschil in uitvoeringstijd tussen rustige momenten en plotse pieken in verkeer. Jammer genoeg is de performantie van het Wolfram Cloud Platform wat betreft schaalbaarheid zoals in paragraaf 12.6.3 aangegeven nogal instabiel, maar desalniettemin is na uitvoerig testen geen enkele keer een ronduit slecht resultaat bekomen voor schaalbaarheid in de Wolfram Cloud tijdens dit onderzoek. Dit maakt van Wolfram ondanks de eerder vastgestelde zwaktes toch een sterke taal in de cloud, hetzij vooral voor specifieke toepassingen waar schaalbaarheid essentieel is.

12.8 Besluit

In het algemeen zijn een aantal interessante besluiten te trekken uit dit omvangrijke en gevarieerde hoofdstuk. Hieronder een samenvatting van de belangrijkste bevindingen.
Alle geteste functionele implementaties blijken het behoorlijk goed te doen in de bestudeerde cloud-platformen in vergelijking met de onderzochte object-georiënteerde alternatieven. Enkel Wolfram is een dubbelzinnig verhaal.
In het algemeen is te stellen dat de functionele implementaties in Java en C# op elk gebied minstens even goed scoren als hun object-georiënteerde tegenhangers, en naar parallellisatie toe in de cloud veel beter scoren dan de geteste OO-implementaties. Voor applicaties waar parallellisatie een belangrijke rol speelt, is het zeker aan te raden een functionele implementatie te gebruiken. F# scoort ook zeer goed voor parallellisatie. De Wolfram-implementatie in de Wolfram Cloud de enige functionele implementatie die teleurstellend presteert wat betreft parallellisatie. In de toekomst zou dit mogelijk kunnen verbeteren, net als de lage connectiesnelheid die de Wolfram Cloud plaagt.
Waar functionele talen echter het sterkst in uit blijken te blinken is schaalbaarheid. Zowel F# als Wolfram laten goede tot zeer goede schaalbaarheidskarakteristieken zien. Dit maakt van

deze talen zeer aantrekkelijke opties voor services waar een goede schaalbaarheid gewenst is. Hetzelfde geldt voor het Standard Environment van Google App Engine met Java.
Voor Wolfram is verder een goede performantie in de cloud vast te stellen. Deze vaststelling dient echter sterk genuanceerd te worden, aangezien Wolfram als taal veel trager is dan alle andere bestudeerde opties. Ook de zeer slechte connectiesnelheid en onbestaande parallellisatie leiden tot de conclusie dat deze taal zeker potentieel bevat voor cloud-ontwikkeling. Indien de in dit hoofdstuk vastgestelde plooien kunnen worden gladgestreken in de toekomst, is Wolfram ook een zeer valabele optie voor heel wat toepassingen.
Nog een belangrijke opmerking is dat Google Compute Engine voor C# op alle gebied behalve performantie geen goede oplossing is voor C# in vergelijking met de Azure App Service. Enkel de kostprijs en de flexibiliteit die een IaaS-platform biedt ten opzichte van een PaaS-platform kunnen redenen zijn om dit platform te gebruiken.
In het algemeen is F# dus de beste taal om aan cloud-ontwikkeling te doen, en dit in combinatie met de Azure App Service, omwille van de zeer goede schaalbaarheid, parallellisatie, en connectiesnelheid. Ook is de eenvoudige implementatie van F# in dit platform zeker niet te vergeten. Enkel de performantie is niet optimaal in Azure, maar nog steeds acceptabel voor de meeste toepassingen. Indien in de toekomst meer performante en makkelijk toegankelijke cloud-platformen beschikbaar raken voor F#, is deze taal veruit de beste voor cloud-ontwikkeling, op basis van de in deze masterproef beschouwde karakteristieken.
Om dit hoofdstuk af te sluiten volgt hieronder een opsomming voor elk cloud-platform en elke taal, met de op basis van de in dit onderzoek bestudeerde karakteristieken bepaalde situaties waarin elk van de bestudeerde platformen het meest geschikt is:

- **Google App engine Standard Environment met Java 7 Object-georiënteerd**: Voor applicaties die volledig in Java 7 geschreven zijn en geen behoefte hebben aan de verbeteringen die Java 8 biedt, zoals bijvoorbeeld bestaande projecten, die zeer schaalbaar moeten zijn, maar waarbij performantie geen belangrijke rol speelt, en parallellistie niet wordt gebruikt om peformantiewinst te bekomen. Voor nieuwe projecten is dit cloud-platform niet aantrekkelijk, behalve door de lage kostprijs en grote schaalbaarheid. Voor hobbyprogrammeurs is dit platform eventueel toch de beste optie door de mogelijkheid om voor onbeperkte tijd gratis gebruik te maken van deze omgeving voor toepassingen die niet veel verkeer binnenkrijgen;
- **Google App engine Flexible Environment met Java 8 Object-georiënteerd:** Bestaande Java OO-applicaties waarbij performantie eventueel belangrijk is, maar onmiddellijke schaalbaarheid op de tweede plaats komt, net als parallellisatie;
- **Google App engine Flexible Environment met Java 8 met Streams:** Bestaande en nieuwe Java-projecten waarbij performantie eventueel belangrijk is, maar onmiddellijke schaalbaarheid op de tweede plaats komt. Voor nieuwe projecten is er geen reden om niet van Streams gebruik te maken in de cloud, aangezien deze implementatie op elk gebied minstens even goed scoort als de OO-versie. Voor parallellisatie zelfs beter naar zowel implementatie als performantie toe;

- **Google Compute Engine met C# in .NET 4.5 object-georiënteerd:** Er is weinig reden om van deze implementatie-optie gebruik te maken, behalve als de flexibiliteit van een IaaS-platform specifiek gewenst is. Enkel in het specifieke geval van een .NET-applicatie waarbij performantie en de flexibiliteit van een IaaS-platform belangrijk zijn, maar parallellisatie en vooral schaalbaarheid geen vereisten zijn is dit platform een competitieve oplossing. Voor nieuwe projecten is het om allerlei redenen desalniettemin beter om van .NET Core gebruik te maken, wat veel eenvoudiger is in de Azure App Service;
- **Google Compute Engine met C# in .NET 4.5 met Streams:** Alle opmerkingen voor de OO-implementatie in C# in dit cloud-platform blijven van toepassing. Omwille van de betere parallellisatie heeft LINQ wel de voorkeur ten opzichte van de OO-versie. Er is dus ook hier geen enkele reden om voor nieuwe projecten in dit platform geen LINQ te gebruiken;
- **Azure App Service met C# in .NET Core object-georiënteerd:** Bestaande .NET Core-projecten die volledig OO-geschreven zijn, zolang performantie geen opperste prioriteit is, en schaalbaarheid ook niet heel belangrijk is. De kostprijs van dit platform is relatief hoog. Dit platform is op alle andere gebieden zeer sterk voor .NET Core, en hoewel het in deze masterproef niet expliciet getest is ongetwijfeld ook .NET 4 en eerder;
- **Azure App Service met C# in .NET Core met LINQ:** Voor alle .NET-gebaseerde projecten, zolang performantie en schaalbaarheid geen grote prioriteit vormen. Net zoals bij Java heeft een functionele stijl in C# geen nadelen en voor parallellisatie zelfs grote voordelen. Opnieuw is enkel kostprijs eventueel een zorg;
- **Azure App Service met F#:** Voor applicaties die compact en snel geschreven moeten zijn is F# een betere oplossing dan C# of Java. De eenvoudige cloud-deployment in de Azure App Service maakt dit platform ideaal voor SOA-architecturen, waarbij voor bepaalde delen van het project de compacte F#-notatie echt een zege kan zijn. Performantie is ongeveer hetzelfde als bij C# in Azure, wat dus het enige echte nadeel van F# in Azure is in vergelijking met de andere cloud-platformen. Voor applicaties die zeer parallelliseerbaar en vooral schaalbaar moeten zijn en performantie geen opperste prioriteit is, is deze optie ideaal. De veel betere schaalbaarheid en nog compactere code zijn de belangrijkste eigenschappen die F# onderscheiden van C# met LINQ in .NET Core voor Azure. De kostprijs is opnieuw vrij hoog, maar gelijk aan die van C#. Al bij al is F# in de Azure App Service een zeer sterke implementatie gebleken.
- **Wolfram Cloud met Wolfram:** Wolfram blinkt uit voor het zeer eenvoudig en compact schrijven van cloud-applicaties. Voor bepaalde specifieke toepassingen kan het de moeite zeker lonen om Wolfram te gebruiken. Datavisualisatie en -verwerking in de vorm van API's zijn hier voorbeelden van, zolang de benodigde rekenkracht beperkt blijft. Ook de mogelijkheden van de geïntegreerde Wolfram|Alpha-functionaliteit, zoals beschreven in paragraaf 5.2 mogen niet over het hoofd gezien worden. Dit maakt Wolfram nu al geschikt voor bepaalde niche-toepassingen in de

industrie, en dit dan vooral in een SOA-context, zolang het gebruik van Wolfram beperkt kan worden tot de zaken waar Wolfram in uitblinkt, en voor de andere delen van de applicatie meer geschikte programmeertalen en cloud-platformen kunnen gebruikt worden.
De zeer hoge connectietijd voor de Wolfram Cloud zet dit alles echter weer sterk in perspectief. Indien die beter zou zijn zou Wolfram zeker op bredere schaal inzetbaar zijn in de industrie. De hoge kostprijs van een degelijke ontwikkelomgeving en cloud-ondersteuning vormt ook een belangrijk nadeel. Enkel voor de hierboven beschreven niche-toepassingen waar programmeergemak een zeer grote prioriteit is en performantie en parallellisatie geen bezorgdheid zijn is Wolfram in de Wolfram Cloud voor industriële toepassingen dus aan te raden, in het bijzonder wanneer zeer grote schaalbaarheid gewenst is.
Een andere grote doelgroep waar Wolfram een zeer aantrekkelijke optie voor is bestaat uit hobbyprogrammeurs voor wie de beperkte gratis aspecten van dit platform volstaan, en die vooral geïnteresseerd zijn in het snel ontwikkelen van een eenvoudige applicatie met beperkte performantie- en geen parallellisatievereisten.
Voorlopig is het in het algemeen verstandig om nog enkele jaren de evolutie van Wolfram en de bijhorende Wolfram Cloud te volgen alvorens op grote schaal volledige projecten te ontwikkelen in de Wolfram Cloud met Wolfram, maar voor bepaalde niche-services binnen een project is Wolfram nu reeds aan te raden.

13 Conclusie

Deze thesis heeft een gedetailleerde vergelijking gemaakt tussen verschillende object-georiënteerde en functionele programmeertalen, met de nadruk op cloud computing. Een groot aantal aspecten van de verschillende talen is vergeleken. De resultaten hiervan zijn vrij uiteenlopend. Op sommige gebieden is het moeilijk een algemeen geldende conclusie te formuleren. Op andere gebieden is er wel een duidelijke tendens vast te stellen.

Zo blijken functionele implementaties door hun declaratieve natuur veel eleganter en compacter te zijn dan hun object-georiënteerde tegenhangers voor de in deze thesis bestudeerde toepassingen. Bovendien bleken de functionele varianten steeds leesbaarder en minder foutgevoelig. Vooral bij geparallelliseerde toepassingen komen de grote voordelen van functioneel programmeren op dit gebied tot hun recht. Ook naar performantie toe is parallellisatie in functionele talen veiliger toe te passen dan in object-georiënteerde, aangezien geparallelliseerde functionele implementaties duidelijk minder overhead veroorzaken dan object-georiënteerde. De algemene performantie van functionele talen is bij zowel sequentiële als parallelle toepassingen nagenoeg gelijk aan die van object-georiënteerde toepassingen, met als enige uitzondering op deze regel Wolfram, wat veel trager is dan alle andere geteste talen. De oorzaak hiervan ligt echter waarschijnlijk eerder bij de geïnterpreteerde dan de functionele natuur van deze taal.

Er bestaat tegenwoordig een groot aantal cloud-platformen voor mainstream programmeertalen. Doordat deze talen over talrijke functionele eigenschappen beschikken, is het impliciet zeer goed mogelijk om functioneel programmeren toe te passen in de cloud. Voor de hoofdzakelijk functionele programmeertalen is cloud-ondersteuning echter beperkter. De weinige beschikbare cloud-platformen voor deze talen bleken wel gebruiksvriendelijker te zijn dan veel van de geteste omgevingen voor de object-georiënteerde talen.

De compactere en elegantere implementaties die functionele programmeertalen doorgaans bieden komen in het bijzonder tot hun recht bij cloud computing. Het is in de geteste functionele talen veel eenvoudiger gebleken om een bestaande desktop-applicatie om te vormen tot een cloud-deploybare REST-API dan voor de object-georiënteerde talen. Vooral Wolfram blinkt hierin uit, doordat cloud-ondersteuning rechtstreeks is ingebouwd in de taal zelf. Het beperkte aantal beschikbare cloud-platformen is dus niet onmiddellijk een belemmering voor deze talen in de cloud. Enkel de Wolfram Cloud toonde significante performantiebeperkingen in de vorm van een zeer hoge connectietijd en het gebrek aan parallellisatiemogelijkheden. Beide kunnen in de toekomst wel verbeteren als Wolfram meer aanslaat en de kwaliteit van cloud-infrastructuur verbetert.

In het algemeen valt op dat de betere parallellisatiekarakteristieken van de geteste functionele implementaties in de cloud nog beter tot hun recht komen dan op de lokale pc, met uitzondering van Wolfram. Alle andere geteste functionele implementaties toonden voor dit aspect beduidend betere resultaten in de cloud dan hun object-georiënteerde tegenhangers, terwijl op de lokale pc de verschillen eerder subtiel bleken. In combinatie met de veel eenvoudigere implementatie van parallellisatie in functionele talen is het ten zeerste

aan te raden voor cloud-toepassingen waar parallellisatie een significante rol speelt gebruik te maken van een functionele implementatie.
Verder blinken de geteste hoofdzakelijk functionele talen in de cloud uit wat betreft schaalbaarheid. De implementaties van Wolfram en F# in de cloud bleken bijzonder schaalbaar en waren in staat een grote hoeveelheid binnenkomend verkeer in korte tijd af te handelen. Voor de andere implementaties die gebruik maakten van object-georiënteerde frameworks was dit met uitzondering van het Google App Engine Standard Environment voor Java nooit het geval.
De functionele implementaties in Java en C# blijken op elk getest aspect op zijn minst even goed te scoren als de object-georiënteerde implementaties binnen dezelfde taal. Naar implementatie en parallellisatie toe scoren ze zelfs beduidend beter dan de object-georiënteerde alternatieven, in het bijzonder in de cloud. Er is dus geen reden om niet van functioneel programmeren gebruik te maken in deze talen.
Na analyse van alle bestudeerde aspecten blijkt F# in het algemeen de best presterende programmeertaal te zijn voor cloud computing. Op elk gebied scoort deze taal heel goed. F# weet een ideale compromis te sluiten tussen implementatiegemak en performantie.
Naar implementatiegemak toe scoort Wolfram nog beter dan F#, maar het gebruik van Wolfram brengt grote beperkingen met zich mee naar performantie, parallellisatie, en connectietijd in de cloud toe. Door het grote aantal unieke eigenschappen dat Wolfram bezit is deze taal voor bepaalde specifieke toepassingen echter nog steeds aan te raden, hoewel het verstandiger is om nog even te wachten met het ontwikkelen van grote applicaties in Wolfram en in het bijzonder in de Wolfram Cloud. Deze technologie heeft nog enkele jaren nodig om volledig matuur te worden.
Deze masterproef focuste zich op de vergelijking tussen een groot aantal programmeertalen en een groot aantal cloud-platformen. Bijgevolg zijn tijdens dit onderzoek hier en daar een aantal geïsoleerde vreemde vaststellingen gedaan, zoals bijvoorbeeld het ontstaan van uitschieters in de performantie van sterk geparallelliseerde F#-toepassingen. Door de zeer brede scope van deze thesis is door timing-gerelateerde beperkingen niet diep ingegaan op deze afwijkende resultaten, en enkel speculatie gegeven over de mogelijke oorzaken van deze vreemde resultaten, aangezien ze slechts een beperkte impact hebben op de algemene conclusies van deze masterproef. Verder onderzoek is nodig om de exacte oorzaak van deze vreemde resultaten te achterhalen.
Zo kort mogelijk samengevat is de conclusie van deze masterproef dus dat functioneeel programmeren in de cloud zeer goed mogelijk is, en dat functionele implementaties in het bijzonder voor applicaties waar parallellisme een significante rol speelt zelfs sterk te verkiezen zijn boven object-georiënteerde oplossingen. Dit kunnen zowel functionele implementaties binnen object-georiënteerde programmeertalen zijn als onafhankelijke functionele programmeertalen. Indien daarnaast schaalbaarheid een primaire vereiste voor de applicatie is, is het zelfs aan te raden gebruik te maken van een hoofdzakelijk functionele programmeertaal en object-georiënteerde programmeertalen volledig links te laten liggen.

Bibliografie

[1] R. Buyya, C. Shin Yeo, S. Venugopal, J. Broberg en I. Brandic, „Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *The International Journal of eScience,* vol. 25, nr. 6, pp. 599-616, 2009.

[2] S. Mukherjee, Thinking in Linq, Apress, 2014.

[3] „Embedded systems & security," KULeuven, [Online]. Available: http://iiw.kuleuven.be/onderzoek/ess. [Geopend 16 October 2016].

[4] K. Aerts en T. Schrijvers, „vLambda: Functioneel programmeren voor de Vlaamse Software-Industrie," [Online]. Available: http://cmdstud.khlim.be/~kris/vlambda/. [Geopend 16 October 2016].

[5] K. Aerts, Service Oriented Architecture (SOA) en Cloud Computing, Diepenbeek: UHasselt, 2014.

[6] Arcitura Education, „WhatIsCloud," 11 11 2016. [Online]. Available: http://whatiscloud.com/origins_and_influences/index.

[7] Arcitura Education TM, „CloudPatterns," Arcitura Education TM, ondekend. [Online]. Available: http://cloudpatterns.org/design_patterns/overview. [Geopend 11 november 2016].

[8] F. Liu, J. Tong, J. Mao, R. Bohn, J. Messina, L. Badger en D. Leaf, „NIST Cloud Computing Reference Architecture," Information Technology Laboratory, National Institute of Standards and Technology, U.S. Department of Commerce, Verenigde Staten van Amerika, 2011.

[9] K. Praveenkumar, T. Johnson P. en E. Chan-tin, „Towards an efficient distributed cloud computing architecture," *Peer-to-Peer Networking and Applications,* vol. 9, nr. 3, pp. 1-17, 2016.

[10] M. Rouse, „XaaS (anything as a service)," TechTarget, augustus 2010. [Online]. Available: http://searchcloudcomputing.techtarget.com/definition/XaaS-anything-as-a-service. [Geopend 13 november 2016].

[11] Anoniem, „A summary of important cloud-computing issues distilled from ACM CTO Roundtables," *ACMQueue,* vol. 7, nr. 5, p. 2, 2009.

[12] A. Fernandez, S. del Rio, V. Lopez, A. Bawakid, J. M. del Jesus, M. J. Benitez en F. Herrera, „Big Data with Cloud Computing: an insight on the computing environment, MapReduce, and programming frameworks," *WIREs,* vol. 4, nr. 5, pp. 380-409, 2014.

[13] K. M. Srinivassan, K. Sarukesi, P. Rodrigues, S. M. Manoj en P. Revathy, „State-of-the-art cloud computing security taxonomies: a classification of security challenges in the present cloud computing environment," in *ICACCI '12 Proceedings of the International Conference on Advances in Computing, Communications and Informatics*, Chennai, India, 2012.

[14] Microsoft Azure, „How do I choose a cloud service provider?," Microsoft, 2017. [Online]. Available: https://azure.microsoft.com/en-us/overview/choosing-a-cloud-service-provider/. [Geopend 21 april 2017].

[15] Cloudorado, „Cloud Computing Comparisson Engine," Cloudorado, [Online]. Available: https://www.cloudorado.com/. [Geopend 21 april 2017].

[16] S. Schildermans, „Autorisatie & security in een SOA- & Cloud-Omgeving," UHasselt, Diepenbeek, 2017.

[17] A. Li, X. Yang, S. Kandula en M. Zhang, „Comparing Public-Cloud Providers," *IEEE Computer Society*, pp. 50-53, 2011.

[18] „Amazon EC2 - Virtual Server Hosting," Amazon, [Online]. Available: https://aws.amazon.com/ec2/. [Geopend 13 November 2016].

[19] „Amazon s3," [Online]. Available: https://aws.amazon.com/s3/. [Geopend 13 November 2016].

[20] „Products & Services," Google, [Online]. Available: https://cloud.google.com/products/. [Geopend 13 November 2016].

[21] „Google App Engine," Google, [Online]. Available: https://cloud.google.com/appengine/. [Geopend 13 November 2016].

[22] „Cloud Bigtable," Google, [Online]. Available: https://cloud.google.com/bigtable/. [Geopend 13 November 2016].

[23] „Microsoft Azure," Microsoft, [Online]. Available: https://azure.microsoft.com/nl-nl/. [Geopend 13 November 2016].

[24] „IBM Cloud," IBM, [Online]. Available: https://www.ibm.com/cloud-computing/. [Geopend 13 November 2016].

[25] „Oracle Cloud," Oracle, [Online]. Available: https://cloud.oracle.com/home. [Geopend 14 November 2016].

[26] „Salesforce," Salesforce, [Online]. Available: https://www.salesforce.com/eu/. [Geopend 13 November 2016].

[27] Arcitura Education, „stateless (primary state condition)," Arcitura Education, Onbekend. [Online]. Available: http://serviceorientation.com/soaglossary/stateless. [Geopend 17 november 2016].

[28] T. Redfern, „Difference between Microservices Architecture and SOA," stackoverflow, 4 10 2016. [Online]. Available: http://stackoverflow.com/questions/25501098/difference-between-microservices-architecture-and-soa. [Geopend 12 11 2016].

[29] M. Fowler, „GOTO 2014 • Microservices • Martin Fowler," GOTO Conferences, 15 januari 2015. [Online]. Available: https://www.youtube.com/watch?v=wgdBVIX9ifA&feature=youtu.be&t=13m10s. [Geopend 12 november 2016].

[30] A. Reichert, „Web services vs. cloud services: Are they the same?," TechTarget, november 2014. [Online]. Available: http://searchsoa.techtarget.com/tip/Web-services-vs-cloud-services-Are-they-the-same. [Geopend 12 november 2016].

[31] M. Esponda, „λ−Kalkül," 2013. [Online]. Available: http://www.inf.fu-berlin.de/lehre/WS12/ALP1/lectures/V19_ALPI_Lambda_Kalkuel_2013.pdf. [Geopend 16 april 2017].

[32] S. I. Moore, „Functional, Declarative, and Imperative Programming [closed]," Stack Overflow, 2 december 2011. [Online]. Available: https://stackoverflow.com/questions/602444/functional-declarative-and-imperative-programming. [Geopend 25 mei 2017].

[33] M. C. Benton en N. M. Radziwill, „Improving Testability and Reuse by Transitioning to Functional Programming," *Software Quality Professional*, vol. 18, nr. 3, pp. 30-38, 2016.

[34] S. Mukherjee, Thinking in LINQ, New York: Springer Science+Business Media, 2014.

[35] Aaronaught, „What is a "side effect?"," 5 augustus 2011. [Online]. Available: http://softwareengineering.stackexchange.com/questions/40297/what-is-a-side-effect. [Geopend 16 november 2016].

[36] Khan Acadamy, „Overview of quicksort," Khan Acadamy, 2016. [Online]. Available: https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/overview-of-quicksort. [Geopend 18 november 2016].

[37] E. Lippert, „Why hasn't functional programming taken over yet?," Stack Exchange Inc., 14 mei 2010. [Online]. Available: http://stackoverflow.com/questions/2835801/why-hasnt-functional-programming-taken-over-yet. [Geopend 17 november 2016].

[38] T. Jelvis, „What are some limitations/disadvantages of functional programming?," Quora, 1 april 2014. [Online]. Available: https://www.quora.com/What-are-some-limitations-disadvantages-of-functional-programming. [Geopend 17 november 2016].

[39] N. Ramsey, „Pitfalls/Disadvantages of Functional Programming," Stack Exchange Inc., 24 november 2009. [Online]. Available: http://stackoverflow.com/questions/1786969/pitfalls-disadvantages-of-functional-programming. [Geopend 17 november 2016].

[40] Haskell, „Haskell," Haskell, 2014-2016. [Online]. Available: https://www.haskell.org/. [Geopend 16 november 2016].

[41] E. Allik, „Is Haskell a strongly typed programming language?," 15 december 2015. [Online]. Available: http://stackoverflow.com/questions/34287668/is-haskell-a-strongly-typed-programming-language. [Geopend 16 november 2016].

[42] Wikipedia, „Erlang (programming language)," 14 november 2016. [Online]. Available: https://en.wikipedia.org/wiki/Erlang_(programming_language). [Geopend 16 november 2016].

[43] Erlang, „Build massively scalable soft real-time systems,” 2016. [Online]. Available: https://www.erlang.org/. [Geopend 16 november 2016].

[44] H. Veldstra, „Why Erlang Is Awesome,” [Online]. Available: http://whyerlang.com/. [Geopend 16 november 2016].

[45] darvids0n, „Erlang's 99.9999999% (nine nines) reliability,” 8 december 2011. [Online]. Available: http://stackoverflow.com/questions/8426897/erlangs-99-9999999-nine-nines-reliability. [Geopend 16 november 2016].

[46] F# Software Foundation and individual contributors, „About F#,” F# Software Foundation, 2015. [Online]. Available: http://fsharp.org/about/index.html. [Geopend 16 november 2016].

[47] D. Soshnikov, „Using Functional Programming for Development of Distributed, Cloud and Web Applications in F#,” in *CEE-SECR 2011*, Moskou, 2011.

[48] ECMA, „Dart Programming Language specification,” ECMA, 2015.

[49] „Dart Language,” [Online]. Available: https://www.dartlang.org/. [Geopend 16 November 2016].

[50] Computer Hope, „Logic Programming,” Computer Hope, 26 april 2017. [Online]. Available: https://www.computerhope.com/jargon/l/logic-programming.htm. [Geopend 25 mei 2017].

[51] Wolfram, „Wolfram Language,” 2016. [Online]. Available: http://www.wolfram.com/language/?source=nav. [Geopend 10 oktober 2016].

[52] IBM, „Compiled versus interpreted languages,” IBM Corporation, 2010. [Online]. Available: https://www.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos.zappldev/zappldev_85.htm. [Geopend 25 mei 2017].

[53] „The Computer Language,” Benchmarksgame, [Online]. Available: http://benchmarksgame.alioth.debian.org/u32/performance.php?test=mandelbrot. [Geopend 21 april 2017].

[54] R. Naim, S. Hanamasagar, M. Miladinova, M. F. Nizam en J. Noureddine, „Comparative Studies of 10 Programming Languages within 10 Diverse Criteria,” 2010.

[55] L. Jiang, L. Mingzhi, R. Sleiman en L. Yuanwei, Comparative Studies of 10 Programming Languages, 2010.

[56] E. Delarzo, „Functional Programming with Java 8 Functions,” DZone, 20 oktober 2014. [Online]. Available: https://dzone.com/articles/functional-programming-java-8. [Geopend 5 juni 2017].

[57] Benjamin, „Java 8 Stream Tutorial,” winterbe, 31 juli 2014. [Online]. Available: http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/. [Geopend 5 juni 2017].

[58] J. Popovic, „Functional programming in C#," Code Project, 10 juni 2012. [Online]. Available: https://www.codeproject.com/Articles/375166/Functional-programming-in-Csharp. [Geopend 5 juni 2017].

[59] F# for fun and profit, „The "Why use F#?" series," F# for fun and profit, 2016. [Online]. Available: http://fsharpforfunandprofit.com/series/why-use-fsharp.html. [Geopend 6 juni 2017].

[60] F# for fun and profit, „Tuples," F# for fun and profit, 2016. [Online]. Available: http://fsharpforfunandprofit.com/posts/tuples/. [Geopend 6 juni 2017].

[61] F# for fun and profit, „Pattern matching for conciseness," F# for fun and profit, 2016. [Online]. Available: http://fsharpforfunandprofit.com/posts/conciseness-pattern-matching/. [Geopend 6 juni 2017].

[62] F# for fun and profit, „Active patterns," F# for fun and profit, 2016. [Online]. Available: http://fsharpforfunandprofit.com/posts/convenience-active-patterns/. [Geopend 6 juni 2017].

[63] Wolfram, „Wolfram Language & System Documentation Center," Wolfram, 2017. [Online]. Available: http://reference.wolfram.com/language/. [Geopend 15 mei 2017].

[64] Wolfram, „ The Wolfram Language: FAST INTRODUCTION FOR PROGRAMMERS," Wolfram, 2017. [Online]. Available: http://www.wolfram.com/language/fast-introduction-for-programmers/en/interactive-usage/. [Geopend 15 mei 2017].

[65] Wolfram, „Dynamic," Wolfram, 2017. [Online]. Available: https://reference.wolfram.com/language/ref/Dynamic.html. [Geopend 3 juni 2017].

[66] Wolfram, „ListLinePlot," Wolfram, 2017. [Online]. Available: https://reference.wolfram.com/language/ref/ListLinePlot.html. [Geopend 4 juni 2017].

[67] Wolfram, „EmbedCode," Wolfram, 2017. [Online]. Available: https://reference.wolfram.com/language/ref/EmbedCode.html. [Geopend 4 juni 2017].

[68] Wolfram, „WSTP (Wolfram Symbolic Transfer Protocol)," Wolfram, 2017. [Online]. Available: https://www.wolfram.com/wstp/. [Geopend 4 juni 2017].

[69] Wolfram, „Writing Java Programs That Use the Wolfram Language," Wolfram, 2017. [Online]. Available: http://reference.wolfram.com/language/JLink/tutorial/WritingJavaProgramsThatUseTheWolframLanguage.html#15141. [Geopend 4 juni 2017].

[70] Wolfram, „.NET/Link User Guide," Wolfram, 2017. [Online]. Available: http://reference.wolfram.com/language/NETLink/tutorial/Overview.html. [Geopend 4 juni 2017].

[71] Wolfram, „Evaluation Control," Wolfram, 2017. [Online]. Available: https://reference.wolfram.com/language/guide/EvaluationControl.html. [Geopend 15 mei 2017].

[72] Wolfram, „Wolfram Language & Mathematica free on every Raspberry Pi," Wolfram, 2017. [Online]. Available: http://www.wolfram.com/raspberry-pi/. [Geopend 5 juni 2017].

[73] „Pascal's Triangle," Math Is Fun, 2014. [Online]. Available: http://www.mathsisfun.com/pascals-triangle.html. [Geopend 3 april 2017].

[74] Anoniem, „Pascal Triangle Program in Java," Sitesbay, [Online]. Available: http://www.sitesbay.com/java-program/java-pascal-triangle. [Geopend 3 april 2017].

[75] K. Dekimpe en B. Demoen, „Analyse van algoritmen," in *Fundamenten voor de Informatica*, Kortrijk, K.U.Leuven, 2005, pp. 30-41.

[76] A. Bogomolny, „Patterns in Pascal's Triangle," Cut The Knot, 2017. [Online]. Available: http://www.cut-the-knot.org/arithmetic/combinatorics/PascalTriangleProperties.shtml. [Geopend 7 april 2017].

[77] user99403, „Proving by induction that ∑nk=0(nk)=2n," StackExchange, 10 oktober 2013. [Online]. Available: http://math.stackexchange.com/questions/519832/proving-by-induction-that-sum-k-0nn-choose-k-2n. [Geopend 7 april 2017].

[78] V. Búr, „Dealing with an ArrayStoreException," Stack Overflow, 11 september 2012. [Online]. Available: http://stackoverflow.com/questions/12369957/dealing-with-an-arraystoreexception. [Geopend 3 april 2017].

[79] Wolfram, „WOLFRAM DEVELOPMENT PLATFORM pricing," Wolfram, 2017. [Online]. Available: http://www.wolfram.com/development-platform/pricing/. [Geopend 9 april 2017].

[80] Wolfram, „Parallel Computing," Wolfram, 2017. [Online]. Available: https://reference.wolfram.com/language/guide/ParallelComputing.html. [Geopend 27 mei 2017].

[81] Oracle, „Oracle Cloud Predictions," Oracle, 2017. [Online]. Available: https://www.oracle.com/cloud/index.html. [Geopend 12 mei 2017].

[82] Oracle, „Experience Oracle Cloud with $300 in free credit," Oracle, [Online]. Available: https://cloud.oracle.com/tryit. [Geopend 12 mei 2017].

[83] Google, „Google Cloud Platform Documentation," Google, 17 april 2017. [Online]. Available: https://cloud.google.com/docs/. [Geopend 20 april 2017].

[84] Google, „Choosing an App Engine Environment," Google, 30 maart 2017. [Online]. Available: https://cloud.google.com/appengine/docs/the-appengine-environments. [Geopend 20 april 2017].

[85] Google, „Google Cloud Platform Free Tier," Google, [Online]. Available: https://cloud.google.com/free/. [Geopend 20 april 2017].

[86] sdxcentral, „Containers vs VMs: Which is better in the Data Center?," sdxcentral, 2017. [Online]. Available: https://www.sdxcentral.com/cloud/containers/definitions/containers-vs-vms/. [Geopend 12 mei 2017].

[87] M. Otey, „Containers vs. Virtual Machines," Redmond Magazine, 9 juni 2016. [Online]. Available: https://redmondmag.com/articles/2016/09/01/containers-vs-virtual-machines.aspx. [Geopend 12 mei 2017].

[88] Google, „App Engine Flexible Environment," Google, 9 maart 2017. [Online]. Available: https://cloud.google.com/appengine/docs/flexible/. [Geopend 12 mei 2017].

[89] Google, „COMPUTE ENGINE," Google, [Online]. Available: https://cloud.google.com/compute/. [Geopend 12 mei 2017].

[90] Azure, „Azure-producten," Microsoft, 2017. [Online]. Available: https://azure.microsoft.com/nl-nl/services/. [Geopend 12 mei 2017].

[91] Azure, „Aan de slag met Azure," Microsoft, [Online]. Available: https://docs.microsoft.com/nl-nl/azure/#pivot=services. [Geopend 12 mei 2017].

[92] J. Roth en e. al., „Azure subscription and service limits, quotas, and constraints," Microsoft Azure, 19 april 2017. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-subscription-service-limits. [Geopend 27 mei 2017].

[93] Wolfram, „Wolfram Cloud Credits," Wolfram, 2017. [Online]. Available: https://www.wolfram.com/cloud-credits/. [Geopend 27 mei 2017].

[94] G. García, „Cloud Platforms where simple Java web application can be deployed for free [closed]," StackOverflow, 13 oktober 2011. [Online]. Available: http://stackoverflow.com/questions/7750266/cloud-platforms-where-simple-java-web-application-can-be-deployed-for-free. [Geopend 9 april 2017].

[95] NetBeans, „NetBeans IDE 8.0.2 Download," NetBeans, 2015. [Online]. Available: https://netbeans.org/downloads/8.0.2/. [Geopend 19 april 2017].

[96] Oracle Corporation, „Jersey; About," Oracle Corporation, 2017. [Online]. Available: https://jersey.java.net/. [Geopend 19 april 2017].

[97] „Jersey 2.25.1 User Guide," Oracle Corporation, 2017. [Online]. Available: https://jersey.java.net/documentation/latest/index.html. [Geopend 19 april 2017].

[98] Maven, „Welcome to Apache Maven," The Apache Software Foundation, 13 april 2017. [Online]. Available: https://maven.apache.org/. [Geopend 19 april 2017].

[99] Maven, „Introduction," The Apache Software Foundation, 13 april 2017. [Online]. Available: https://maven.apache.org/what-is-maven.html. [Geopend 19 april 2017].

[100] Maven, „Introduction to Archetypes," The Apache Software Foundation, 13 april 2017. [Online]. Available: https://maven.apache.org/guides/introduction/introduction-to-archetypes.html. [Geopend 19 april 2017].

[101] Google, „Deploying a Java App," Google, 20 maart 2017. [Online]. Available: https://cloud.google.com/appengine/docs/standard/java/tools/uploadinganapp. [Geopend 20 april 2017].

[102] Google Cloud Platform, „Java Runtime Environment," Google, 17 maart 2017. [Online]. Available:

https://cloud.google.com/appengine/docs/standard/java/runtime?csw=1#The_Sandbox. [Geopend 27 april 2017].

[103] Google Cloud Platform, „The Java 8/ Jetty 9 Runtime," Google, 21 maart 2017. [Online]. Available: https://cloud.google.com/appengine/docs/flexible/java/dev-jetty9. [Geopend 27 april 2017].

[104] Google Cloud Platform, „Google Cloud SDK Documentation," Google, [Online]. Available: https://cloud.google.com/sdk/docs/. [Geopend 27 april 2017].

[105] Eclipse, „Jetty://," Eclipse, 2016. [Online]. Available: http://www.eclipse.org/jetty/. [Geopend 27 april 2017].

[106] P. e. a. Carter, „Choosing between .NET Core and .NET Framework for server apps," Microsoft, 16 november 2016. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/choosing-core-framework-server. [Geopend 28 mei 2017].

[107] Google Cloud Platform, „Quickstart for .NET in the App Engine Flexible Environment," Google, 9 mei 2017. [Online]. Available: https://cloud.google.com/appengine/docs/flexible/dotnet/quickstart. [Geopend 13 mei 2017].

[108] Google, „Quickstart," Google, 27 april 2017. [Online]. Available: https://cloud.google.com/tools/visual-studio/docs/quickstart?authuser=2. [Geopend 13 mei 2017].

[109] Google Cloud Platform, „ASP.NET Framework," Google, 16 maart 2017. [Online]. Available: https://console.cloud.google.com/launcher/details/click-to-deploy-images/aspnet?authuser=2&project=pascaltriangleapi. [Geopend 13 mei 2017].

[110] Microsoft Visual Studio, „Visual F# Tools," YouTube, 12 april 2017. [Online]. Available: https://www.youtube.com/watch?v=I6xZyGEofvo&index=17&list=WL&t=1273s. [Geopend 13 mei 2017].

[111] H. Ademar, „Deploying Suave to Azure App Service," Suave, [Online]. Available: https://suave.io/azure-app-service.html. [Geopend 13 mei 2017].

[112] A. ChauChan, „Windows Azure Website: Uploading/Downloading files over FTP and collecting Diagnostics logs," Microsoft, 19 jinu 2012. [Online]. Available: https://blogs.msdn.microsoft.com/avkashchauhan/2012/06/19/windows-azure-website-uploadingdownloading-files-over-ftp-and-collecting-diagnostics-logs/. [Geopend 13 mei 2017].

[113] „STIRLING APPROXIMATION FORMULA," [Online]. Available: http://cs.pwr.edu.pl/cichon/Math/StirlingApp.pdf. [Geopend 3 april 2017].

[114] AbcAeffchen, „Time-complexity of recursive algorithm for calculating binomial coefficient," Stack Overflow, 7 oktober 2014. [Online]. Available: http://stackoverflow.com/questions/26228385/time-complexity-of-recursive-algorithm-for-calculating-binomial-coefficient. [Geopend 3 april 2017].

[115] The Computer Language Benchmarks Game, „mandelbrot,” http://benchmarksgame.alioth.debian.org, [Online]. Available: http://benchmarksgame.alioth.debian.org/u32/performance.php?test=mandelbrot. [Geopend 21 april 2017].

[116] Google Cloud Platform, „Installing Cloud SDK,” Google, 26 april 2017. [Online]. Available: https://cloud.google.com/sdk/downloads. [Geopend 27 april 2017].

Bijlagen

Bijlage A: Functie voor het bepalen van de juiste curve-fitting kromme voor elk van de geteste datasets

```
fittingFunction[name_String] := Which[
  Length[Select[name == # &][{"Wolfram cloud",
    "Wolfram cloud par. rows",
    "Wolfram cloud par. cols",
    "Wolfram cloud par. rows and cols",
    "Java GAE Std.",
    "Java GAE Std. par. rows",
    "Java GAE Std. par. cols",
    "Java GAE Std. par. rows and cols",
    "Java GAE Flex.",
    "Java GAE Flex. par. rows",
    "Java GAE Flex. par. cols",
    "Java GAE Flex. par. rows and cols",
    "Java with Streams GAE Flex.",
    "Java with Streams GAE Flex. par. rows",
    "Java with Streams GAE Flex. par. cols",
    "Java with Streams GAE Flex. par. rows and cols",
    "C# GCE",
    "C# GCE par. rows",
    "C# GCE par. cols",
    "C# GCE par. rows and cols",
    "C# Azure",
    "C# Azure par. rows",
    "C# Azure par. cols",
    "C# Azure par. rows and cols",
    "C# with LINQ GCE",
    "C# with LINQ GCE par. rows",
    "C# with LINQ GCE par. cols",
    "C# with LINQ GCE par. rows and cols",
    "C# with LINQ Azure",
    "C# with LINQ Azure par. rows",
    "C# with LINQ Azure par. cols",
    "C# with LINQ Azure par. rows and cols",
    "F# Azure",
    "F# Azure par. rows",
    "F# Azure par. cols",
    "F# Azure par. rows and cols"}]] == 1, {2^n, 1},
  Length[Select[name == # &][{"Wolfram non-recursive",
    "Java non-recursive", "Java non-recursive without getter"}]] == 1, {n^2},
  Replace[Catch[data[name]], Hold[_] → True], {2^n},
  True, Throw["Invalid input: " <> name, invalidInputException]]
```

Bijlage B: URL's van de verschillende cloud services van dit onderzoek en functie om deze aan een string-waarde te koppelen

```
urlJavaGAEstd =
"https://pascaltriangleapi.appspot.com/pascaltriangle?type=sequential&size=22";
urlJavaGAEflex =
"http://trianglesjava8v2.appspot.com/pascaltriangle?version=functional&type=
sequential&size=24";
urlWolfram =
"https://www.wolframcloud.com/objects/a874ae7b-a428-46be-978d-6b28aac9e3fb?type=
sequential&size=14";
urlCSGCE =
"http://146.148.19.212/pascaltriangle?version=oo&type=sequential&size=24";
urlCSAzure =
"http://pascaltrianglenetcoreazure20170420090449.azurewebsites.net/pascaltriangle/oo
/sequential/23";
urlCSLINQAzure =
"http://pascaltrianglenetcoreazure20170420090449.azurewebsites.net/pascaltriangle/
functional/sequential/23";
urlFSAzure =
"http://fstrianglesazure.azurewebsites.net/pascaltriangle/sequential?size=22";

decodeUrl[cloud_String] := Which[
  cloud == "Java Google App Engine Standard Environment", urlJavaGAEstd,
  cloud == "Java Google App Engine Flexible Environment", urlJavaGAEflex,
  cloud == "C# Google Compute Engine", urlCSGCE,
  cloud == "C# Azure", urlCSAzure,
  cloud == "C# with LINQ Azure", urlCSLINQAzure,
  cloud == "F# Azure", urlFSAzure,
  cloud == "Wolfram Cloud", urlWolfram,
  True, ""]
```

Bijlage C: Overzicht van alle testdata-functies

```
dataWolfram := Table[{i, t1 = Now; pascalTriangleRec[i]; Now - t1}, {i, 1, 20}];
dataWolframNonRecursive := Block[{$IterationLimit = 10 000}, Table[{i, t1 = Now;
pascalTriangle[i];
  Now - t1}, {i, 10, 1000, 10}]];
dataWolframParallelRows := Table[{i, t1 = Now; pascalTriangleRecParallelRows[i];
  Now - t1}, {i, 1, 20}];
dataWolframParallelCols := Table[{i, t1 = Now; pascalTriangleRecParallelCols[i]; Now - t1},
{i, 1, 20}];
dataWolframParallelRowsCols := Table[{i, t1 = Now; pascalTriangleRecParallelRowsCols[i];
  Now - t1}, {i, 1, 20}];
dataWolframCloud := Table[{i, t1 = Now; Import[
  "https://www.wolframcloud.com/objects/a874ae7b-a428-46be-978d-6b28aac9e3fb?type=
  sequential&size=" <> ToString[i]]; Now - t1}, {i, 1, 20}];
dataWolframParallelRowsCloud := Table[{i, t1 = Now; Import[
  "https://www.wolframcloud.com/objects/a874ae7b-a428-46be-978d-6b28aac9e3fb?type=
  parallelrows&size=" <> ToString[i]]; Now - t1}, {i, 1, 20}];
dataWolframParallelColsCloud := Table[{i, t1 = Now; Import[
  "https://www.wolframcloud.com/objects/a874ae7b-a428-46be-978d-6b28aac9e3fb?type=
  parallelcols&size=" <> ToString[i]]; Now - t1}, {i, 1, 20}];
dataWolframParallelRowsColsCloud := Table[{i, t1 = Now; Import[
  "https://www.wolframcloud.com/objects/a874ae7b-a428-46be-978d-6b28aac9e3fb?type=
  parallelrowscols&size=" <> ToString[i]]; Now - t1}, {i, 1, 20}];
dataJava := Table[{i, t1 = Now; JavaNew["triangles.PascalTriangleRec", i]@getPoints[];
  Now - t1}, {i, 1, 30}];
dataJavaNonRecursive := Table[{i, t1 = Now; JavaNew["triangles.PascalTriangle",
i]@getPoints[];
  Now - t1}, {i, 10, 500, 10}];
dataJavaNonRecursiveNoGetter := Table[{i, t1 = Now; JavaNew["triangles.PascalTriangle", i];
  Now - t1}, {i, 10, 1000, 10}];
dataJavaParallelRows := Table[{i, t1 = Now;
  JavaNew["triangles.PascalTriangleRecParallelRows", i]@getPoints[]; Now - t1}, {i, 1, 30}];
dataJavaParallelCols := Table[{i, t1 = Now;
  JavaNew["triangles.PascalTriangleRecParallelCols", i]@getPoints[]; Now - t1}, {i, 1, 30}];
dataJavaParallelRowsCols := Table[{i, t1 = Now;
  JavaNew["triangles.PascalTriangleRecParallelRowsCols", i]@getPoints[]; Now - t1}, {i, 1,
30}];
dataJavaGAEstd := Table[{i, t1 = Now; Import[
  "https://pascaltriangleapi.appspot.com/pascaltriangle?type=sequential&size=" <>
  ToString[i]]; Now - t1}, {i, 1, 28}];
dataJavaGAEstdParallelRows := Table[{i, t1 = Now; Import[
  "https://pascaltriangleapi.appspot.com/pascaltriangle?type=parallelrows&size="
  <> ToString[i]];Now - t1}, {i, 1, 28}];
```

```
dataJavaGAEstdParallelCols := Table[{i, t1 = Now; Import[
  "https://pascaltriangleapi.appspot.com/pascaltriangle?type=parallelcols&size="
  <> ToString[i]]; Now - t1}, {i, 1, 28}];
dataJavaGAEstdParallelRowsCols := Table[{i, t1 = Now; Import[
  "https://pascaltriangleapi.appspot.com/pascaltriangle?type=parallelrowscols&size
  =" <> ToString[i]]; Now - t1}, {i, 1, 28}];
dataJavaGAEflex := Table[{i, t1 = Now;Import[
  "http://trianglesjava8v2.appspot.com/pascaltriangle?version=oo&type=sequential&
  size=" <> ToString[i]]; Now - t1}, {i, 1, 30}];
dataJavaGAEflexParallelRows := Table[{i, t1 = Now; Import[
  "http://trianglesjava8v2.appspot.com/pascaltriangle?version=oo&type=parallelrows
  &size=" <> ToString[i]]; Now - t1}, {i, 1, 30}];
dataJavaGAEflexParallelCols := Table[{i, t1 = Now; Import[
  "http://trianglesjava8v2.appspot.com/pascaltriangle?version=oo&type=parallelcols
  &size=" <> ToString[i]]; Now - t1}, {i, 1, 30}];
dataJavaGAEflexParallelRowsCols := Table[{i, t1 = Now;
  Import["http://trianglesjava8v2.appspot.com/pascaltriangle?version=oo&type=
  parallelrowscols&size=" <> ToString[i]]; Now - t1}, {i, 1, 30}];
dataJavaStreams := Table[{i, t1 = Now;
  JavaNew["triangles.FunctionalPascalTriangle", i]@getPoints[]; Now - t1}, {i, 1, 30}];
dataJavaStreamsParallelCols := Table[{i, t1 = Now;
  JavaNew["triangles.FunctionalPascalTriangleParallelCols", i]@getPoints[]; Now - t1}, {i, 1, 30}];
dataJavaStreamsParallelRows := Table[{i, t1 = Now;
  JavaNew["triangles.FunctionalPascalTriangleParallelRows", i]@getPoints[]; Now - t1}, {i, 1, 30}];
dataJavaStreamsParallelRowsCols := Table[{i, t1 = Now;
  JavaNew["triangles.FunctionalPascalTriangleParallelRowsCols", i]@getPoints[];Now - t1}, {i, 1, 30}];
dataJavaStreamsGAEflex := Table[{i, t1 = Now; Import[
  "http://trianglesjava8v2.appspot.com/pascaltriangle?version=functional&type=
  sequential&size=" <> ToString[i]]; Now - t1}, {i, 1, 30}];
dataJavaStreamsGAEflexParallelRows := Table[{i, t1 = Now; Import[
  "http://trianglesjava8v2.appspot.com/pascaltriangle?version=functional&type=
  parallelrows&size=" <> ToString[i]]; Now - t1}, {i, 1, 30}];
dataJavaStreamsGAEflexParallelCols := Table[{i, t1 = Now; Import[
  "http://trianglesjava8v2.appspot.com/pascaltriangle?version=functional&type=
  parallelcols&size=" <> ToString[i]]; Now - t1}, {i, 1, 30}];
dataJavaStreamsGAEflexParallelRowsCols := Table[{i, t1 = Now; Import[
"http://trianglesjava8v2.appspot.com/pascaltriangle?version=functional&type=
parallelrowscols&size=" <> ToString[i]]; Now - t1}, {i, 1, 30}];
dataCS := Table[{i, t1 = Now;
  NETNew["DotNetTriangles.PascalTriangle", i]@Points; Now - t1}, {i, 1, 30}];
dataCSParallelRows := Table[{i, t1 = Now;
  NETNew["DotNetTriangles.PascalTriangleRecParallelRows", i]@Points; Now - t1}, {i, 1, 30}];
```

```
dataCSParallelCols := Table[{i, t1 = Now;
  NETNew["DotNetTriangles.PascalTriangleRecParallelCols", i]@Points; Now - t1}, {i, 1, 30}];
dataCSParallelRowsCols := Table[{i, t1 = Now;
  NETNew["DotNetTriangles.PascalTriangleRecParallelRowsCols", i]@Points; Now - t1}, {i, 1, 30}];
dataCSGCE := Table[{i, t1 = Now;
  Import["http://35.187.172.246/pascaltriangle?version=oo&type=sequential&size=" <>
  ToString[i]]; Now - t1}, {i, 1, 30}];
dataCSGCEParallelRows := Table[{i, t1 = Now; Import[
  "http://35.187.172.246/pascaltriangle?version=oo&type=parallelrows&size=" <>
  ToString[i]]; Now - t1}, {i, 1, 30}];
dataCSGCEParallelCols := Table[{i, t1 = Now; Import[
  "http://35.187.172.246/pascaltriangle?version=oo&type=parallelcols&size=" <>
  ToString[i]]; Now - t1}, {i, 1, 30}];
dataCSGCEParallelRowsCols := Table[{i, t1 = Now; Import[
  "http://35.187.172.246/pascaltriangle?version=oo&type=parallelrowscols&size=" <>
  ToString[i]]; Now - t1}, {i, 1, 30}];
dataCSAzure := Table[{i, t1 = Now;
  Import[ "http://pascaltrianglenetcoreazure20170420090449.azurewebsites.net/
pascaltriangle/oo/sequential/" <> ToString[i]]; Now - t1}, {i, 1, 28}];
dataCSAzureParallelRows := Table[{i, t1 = Now;
Import["http://pascaltrianglenetcoreazure20170420090449.azurewebsites.net/pascaltriangle/oo/paralle
lrows/" <> ToString[i]]; Now - t1}, {i, 1, 30}];
  dataCSAzureParallelCols := Table[{i, t1 = Now; Import[
"http://pascaltrianglenetcoreazure20170420090449.azurewebsites.net/pascaltriangle/oo/parallelcols/
" <> ToString[i]]; Now - t1}, {i, 1, 30}];
dataCSAzureParallelRowsCols := Table[{i, t1 = Now;
  Import["http://pascaltrianglenetcoreazure20170420090449.azurewebsites.net/
  pascaltriangle/oo/parallelrowscols/" <> ToString[i]];Now - t1}, {i, 1, 30}];
dataCSLINQ := Table[{i, t1 = Now;
  NETNew["DotNetTriangles.FunctionalPascalTriangle", i]@Points; Now - t1}, {i, 1, 30}];
dataCSLINQParallelRows := Table[{i, t1 = Now;
  NETNew["DotNetTriangles.FunctionalPascalTriangleParallelRows", i]@Points; Now - t1}, {i, 1, 30}];
dataCSLINQParallelCols := Table[{i, t1 = Now;
  NETNew["DotNetTriangles.FunctionalPascalTriangleParallelCols", i]@Points; Now - t1}, {i, 1, 30}];
dataCSLINQParallelRowsCols := Table[{i, t1 = Now; NETNew[
  "DotNetTriangles.FunctionalPascalTriangleParallelRowsCols", i]@Points; Now - t1}, {i, 1, 30}];
dataCSLINQGCE := Table[{i, t1 = Now; Import[
  "http://35.187.172.246/pascaltriangle?version=functional&type=sequential&size="
  <> ToString[i]]; Now - t1}, {i, 1, 30}];
dataCSLINQGCEParallelRows := Table[{i, t1 = Now; Import[
  "http://35.187.172.246/pascaltriangle?version=functional&type=parallelrows&size=
  " <> ToString[i]]; Now - t1}, {i, 1, 30}];
```

```
dataCSLINQGCEParallelCols := Table[{i, t1 = Now;
  Import["http://35.187.172.246/pascaltriangle?version=functional&type=parallelcols&size=
  " <> ToString[i]]; Now - t1}, {i, 1, 30}];
dataCSLINQGCEParallelRowsCols := Table[{i, t1 = Now; Import[
  "http://35.187.172.246/pascaltriangle?version=functional&type=parallelrowscols&
  size=" <> ToString[i]]; Now - t1}, {i, 1, 30}];
dataCSLINQAzure := Table[{i, t1 = Now;
  Import["http://pascaltrianglenetcoreazure20170420090449.azurewebsites.net/
  pascaltriangle/functional/sequential/" <>ToString[i]]; Now - t1}, {i, 1, 28}];
dataCSLINQAzureParallelRows := Table[{i, t1 = Now;
  Import["http://pascaltrianglenetcoreazure20170420090449.azurewebsites.net/
  pascaltriangle/functional/parallelrows/" <>ToString[i]]; Now - t1}, {i, 1, 30}];
dataCSLINQAzureParallelCols := Table[{i, t1 = Now;
  Import["http://pascaltrianglenetcoreazure20170420090449.azurewebsites.net/
pascaltriangle/functional/parallelcols/" <>ToString[i]]; Now - t1}, {i, 1, 30}];
dataCSLINQAzureParallelRowsCols := Table[{i, t1 = Now;
  Import["http://pascaltrianglenetcoreazure20170420090449.azurewebsites.net/
  pascaltriangle/functional/parallelrowscols/" <>ToString[i]]; Now - t1}, {i, 1, 30}];
dataFS := Table[{i, t1 = Now; NETNew["Triangles.PTriangle", i]@Points; Now - t1}, {i, 1, 30}];
dataFSParallelRows := Table[{i, t1 = Now;
  NETNew["Triangles.PascalTriangleParallelRows", i]@Points; Now - t1}, {i, 1, 30}];
dataFSParallelCols := Table[{i, t1 = Now;
  NETNew["Triangles.PascalTriangleParallelCols", i]@Points;Now - t1}, {i, 1, 30}];
dataFSParallelRowsCols := Table[{i, t1 = Now;
  NETNew["Triangles.PascalTriangleParallelRowsCols", i]@Points; Now - t1}, {i, 1, 30}];
dataFSAzure := Table[{i, t1 = Now; Import[
  "http://fstrianglesazure.azurewebsites.net/pascaltriangle/sequential?size=" <>
  ToString[i]]; Now - t1}, {i, 1, 28}];
dataFSAzureParallelRows := Table[{i, t1 = Now; Import[
  "http://fstrianglesazure.azurewebsites.net/pascaltriangle/parallelrows?size=" <>
  ToString[i]]; Now - t1}, {i, 1, 30}];
dataFSAzureParallelCols := Table[{i, t1 = Now; Import[
  "http://fstrianglesazure.azurewebsites.net/pascaltriangle/parallelcols?size=" <>
  ToString[i]]; Now - t1}, {i, 1, 30}];
dataFSAzureParallelRowsCols := Table[{i, t1 = Now; Import[
  "http://fstrianglesazure.azurewebsites.net/pascaltriangle/parallelrowscols?size=
  " <> ToString[i]]; Now - t1}, {i, 1, 30}];
```

Bijlage D: Volledige data-functie voor het linken van een string-waarde aan testdata

```
data[name_String] := Which[
  name == "Wolfram", Hold[dataWolfram],
  name == "Wolfram non-recursive", Hold[dataWolframNonRecursive],
  name == "Wolfram parallel for rows", Hold[dataWolframParallelRows],
  name == "Wolfram parallel for columns", Hold[dataWolframParallelCols],
  name == "Wolfram parallel for rows and columns", Hold[dataWolframParallelRowsCols],
  name == "Wolfram cloud", Hold[dataWolframCloud],
  name == "Wolfram cloud par. rows", Hold[dataWolframParallelRowsCloud],
  name == "Wolfram cloud par. cols", Hold[dataWolframParallelColsCloud],
  name == "Wolfram cloud par. rows and cols", Hold[dataWolframParallelRowsColsCloud],
  name == "Java", Hold[dataJava],
  name == "Java non-recursive", Hold[dataJavaNonRecursive],
  name == "Java non-recursive without getter", Hold[dataJavaNonRecursiveNoGetter],
  name == "Java parallel for rows", Hold[dataJavaParallelRows],
  name == "Java parallel for columns", Hold[dataJavaParallelCols],
  name == "Java parallel for rows and columns", Hold[dataJavaParallelRowsCols],
  name == "Java GAE Std.", Hold[dataJavaGAEstd],
  name == "Java GAE Std. par. rows", Hold[dataJavaGAEstdParallelRows],
  name == "Java GAE Std. par. cols", Hold[dataJavaGAEstdParallelCols],
  name == "Java GAE Std. par. rows and cols", Hold[dataJavaGAEstdParallelRowsCols],
  name == "Java GAE Flex.", Hold[dataJavaGAEflex],
  name == "Java GAE Flex. par. rows", Hold[dataJavaGAEflexParallelRows],
  name == "Java GAE Flex. par. cols", Hold[dataJavaGAEflexParallelCols],
  name == "Java GAE Flex. par. rows and cols", Hold[dataJavaGAEflexParallelRowsCols],
  name == "Java with Streams", Hold[dataJavaStreams],
  name == "Java with Streams parallel for columns", Hold[dataJavaStreamsParallelCols],
  name == "Java with Streams parallel for rows", Hold[dataJavaStreamsParallelRows],
  name == "Java with Streams parallel for rows and columns",
   Hold[dataJavaStreamsParallelRowsCols],
  name == "Java with Streams GAE Flex.", Hold[dataJavaStreamsGAEflex],
  name == "Java with Streams GAE Flex. par. rows", Hold[dataJavaStreamsGAEflexParallelRows],
  name == "Java with Streams GAE Flex. par. cols", Hold[dataJavaStreamsGAEflexParallelCols],
  name == "Java with Streams GAE Flex. par. rows and cols",
   Hold[dataJavaStreamsGAEflexParallelRowsCols],
  name == "C#", Hold[dataCS],
  name == "C# parallel for rows", Hold[dataCSParallelRows],
  name == "C# parallel for columns", Hold[dataCSParallelCols],
  name == "C# parallel for rows and columns", Hold[dataCSParallelRowsCols],
  name == "C# GCE", Hold[dataCSGCE],
```

```
name == "C# GCE par. rows", Hold[dataCSGCEParallelRows],
name == "C# GCE par. cols", Hold[dataCSGCEParallelCols],
name == "C# GCE par. rows and cols", Hold[dataCSGCEParallelRowsCols],
name == "C# Azure", Hold[dataCSAzure],
name == "C# Azure par. rows", Hold[dataCSAzureParallelRows],
name == "C# Azure par. cols", Hold[dataCSAzureParallelCols],
name == "C# Azure par. rows and cols", Hold[dataCSAzureParallelRowsCols],
name == "C# with LINQ", Hold[dataCSLINQ],
name == "C# with LINQ parallel for rows", Hold[dataCSLINQParallelRows],
name == "C# with LINQ parallel for columns", Hold[dataCSLINQParallelCols],
name == "C# with LINQ parallel for rows and columns", Hold[dataCSLINQParallelRowsCols],
name == "C# with LINQ GCE", Hold[dataCSLINQGCE],
name == "C# with LINQ GCE par. rows", Hold[dataCSLINQGCEParallelRows],
name == "C# with LINQ GCE par. cols", Hold[dataCSLINQGCEParallelCols],
name == "C# with LINQ GCE par. rows and cols", Hold[dataCSLINQGCEParallelRowsCols],
name == "C# with LINQ Azure", Hold[dataCSLINQAzure],
name == "C# with LINQ Azure par. rows", Hold[dataCSLINQAzureParallelRows],
name == "C# with LINQ Azure par. cols", Hold[dataCSLINQAzureParallelCols],
name == "C# with LINQ Azure par. rows and cols", Hold[dataCSLINQAzureParallelRowsCols],
name == "F#", Hold[dataFS],
name == "F# parallel for rows", Hold[dataFSParallelRows],
name == "F# parallel for columns", Hold[dataFSParallelCols],
name == "F# parallel for rows and columns", Hold[dataFSParallelRowsCols],
name == "F# Azure", Hold[dataFSAzure],
name == "F# Azure par. rows", Hold[dataFSAzureParallelRows],
name == "F# Azure par. cols", Hold[dataFSAzureParallelCols],
name == "F# Azure par. rows and cols", Hold[dataFSAzureParallelRowsCols],
True, Throw["Invalid input: " <> name, invalidInputException]]
```

Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
Exploratie van functionele en declaratieve ontwikkelmethodes voor cloud programming

Richting: **master in de industriële wetenschappen: elektronica-ICT**
Jaar: **2017**

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Voor akkoord,

Schildermans, Stijn

Datum: **6/06/2017**