interview

Een jaar geleden leverde Intel nog maar twintig procent single core CPU's uit. Ondertussen is het vrijwel onmogelijk geworden om een CPU met één kern te kopen. Toch wordt er behalve voor heel grote systemen, nog maar weinig software geschreven die echt ingericht is op multicore processoren. Brian Goetz – bekend van zijn boek *Java concurrency in practice* – gaf op Devoxx een drukbezochte presentatie over Java-concurrency.

'Anders denken, andere aanpak'

Brian Goetz over Java-concurrency

Dré de Man
Tekst en foto's

Concurrency is een relatief nieuw onderwerp. Het heeft echter vanaf het allereerste begin deel uitgemaakt van Java. Tot voor kort was het echt een onderwerp voor experts, zeker niet voor de doorsnee programmeur. Gezien de grote belangstelling is dat aan het veranderen.
Goetz: "Het is een heel ingewikkeld onderwerp en steeds meer programmeurs zien zich gedwongen het te begrijpen. Een van de consequenties daarvan is, dat zonder betere tools mensen fouten zullen gaan maken. Het is een weinig vergevingsgezinde hoek van de taal. In veel ander gevallen wijst de compiler op fouten. Of je voert het programma uit en je krijgt een exception en je weet meteen dat je een fout gemaakt hebt. Met concurrent programmeren is het heel goed mogelijk om iets echt verkeerd te doen, zonder een foutmelding van de compiler en zonder een exception van de runtime. Je krijgt alleen af en toe het verkeerde antwoord en dat is veel erger. Dat vormde de motivatie om de libraries te verbeteren. Het boek ontstond uit het werk dat we aan de libraries deden. Je kunt het zien als de gebruiksaanwijzing voor de concurrency libraries. Het bevat echter ook al de dingen die niet in de API-documentatie staan."

De hardwareontwikkeling gaat nu heel snel. U vertelde zelfs in uw presentatie dat we binnenkort honderd of zelfs duizend cores mogen verwachten.
"Historisch zie je dat de processorsnelheid in termen van kloksnelheid exponentieel is toegenomen. We zijn daaraan gewend geraakt en het heeft ontwikkelaars ook lui

Brian Goetz is bekend van zijn boek 'Java concurrency in practice'.

gemaakt. Als je wilde dat je programma sneller werkte, dan hoefde je alleen op vakantie te gaan voor een paar maanden of zo. Nu zien we dat er nog steeds een exponentiële toename is, maar die zit hem niet in de processorsnelheid. Nu zit het in het aantal processoren dat je voor hetzelfde geld kunt kopen. We kopen dus meer processorkracht, maar het is moeilijker om er voordeel van te hebben, om het echt te gebruiken."

Als je niets doet, dan is het goed mogelijk dat dezelfde regels code op een snellere processor niet sneller zullen draaien.

"Precies. De toename in aantallen kernen zal doorgaan en dat betekent dat we de manier waarop we software schrijven moeten veranderen om er voordeel van te hebben. Vaak is het niet duidelijk hoe je het moet doen. Sommige taken zijn heel gemakkelijk te verdelen, andere niet.

Ontwikkelaars moeten leren om een opgave te analyseren en te zeggen: 'waar is er exploiteerbaar parallellisme in deze taak die ik - wanneer ik meer processoren of kernen heb - sneller kan uitvoeren. Als ik er niet meer heb, dan kan ik het sequentieel doen in plaats van tegelijkertijd en dan werkt het ook.' Dat is een nieuwe manier van denken. We hebben niet geleerd zo te denken. In feite hebben we het omgekeerde geleerd: dingen beperken. Eerst doe je dit, dan dat. Dat is niet zoals de echte wereld werkt. Alles gebeurt voorturend tegelijkertijd. Daarom is er enige heroriëntatie nodig. We moeten identificeren wat de interacties zouden kunnen zijn. Moet dit gebeuren vóór dat, of maakt het niet uit? Dat is een aanpassing voor ons."

Sommige taken zijn gemakkelijk te parallelliseren, vooral data-intensieve taken als sorteren en zoeken. Er zijn ook taken die heel moeilijk in verschillende threads onder te brengen zijn, maar u heeft wel technieken om het parallelliseren ervan te vergemakkelijken.

"Ja, één daarvan is de divide and conquer-techniek. In feite is het een heel oude techniek om een groot probleem in meerdere kleine problemen te verdelen. Dat blijf je doen, totdat je heel kleine problemen hebt die sequentieel heel gemakkelijk op te lossen zijn. Vervolgens werk je terug en combineer je de resultaten. Het aardige daarvan is, dat het onafhankelijk is van de grootte van het probleem en van het aantal uitvoerenden. We kennen ook de afhankelijkheden, dus het kan gemakkelijk geautomatiseerd worden als je eenmaal de beslissing hebt genomen hoe de dingen te verdelen. Je moet dus nog steeds weten hoe je een groot probleem kunt verdelen in twee kleine problemen. Het sorteren van een grote lijst met getallen is makkelijk, maar gecompliceerdere taken hebben meer aandacht nodig."

Aan de oppervlakte lijkt het alsof concurrent programmeren heel simpel is. Je zet gewoon 'synchronized' in de declaratie van een methode en je bent klaar.

"Ja, dat lijkt zo. De realiteit is echter anders. De vragen die je stelt om te beoordelen of een programma thread-safe is, zijn verschillend van de vragen die je gewend bent te stellen. De meeste tijd dat we nadenken over onze code, letten we op de control flow van de code. Wanneer je je programma analyseert voor concurrency, wordt je geacht aandacht te geven aan de data. Deze data worden geset door deze thread en dan gelezen. Je heb iets gedaan om ervoor te zorgen dat het netjes overgegeven wordt van de één naar de ander en dat is een manier van analyseren die onbekend is. We zijn opgeleid om dit soort dingen als onbelangrijke details te zien. We zijn getraind om details te verwaarlozen en dat is nu juist wat niet moet gebeuren. Je moet jezelf dus opnieuw trainen om op dingen te letten die je vroeger verwaarloosde."

We spraken net al over programma's die met Multi threading goed compileren, die zelfs bij testen goed lijken en die dan uiteindelijk na twee jaar ineens een fout antwoord opleveren. Zijn er al manieren om dit toch bij testen te ondervangen?

"Er zijn technieken die het mogelijk maken concurrent programma's te testen, maar dat is moeilijk. De reden daarvoor is vrij eenvoudig. Als je een sequentieel programma hebt en je voert hetzelfde programma met dezelfde data honderd keer uit, dan krijg je iedere keer hetzelfde antwoord. Concurrent programma's daarentegen zijn onvoorspelbaar. Er zijn zoveel dingen die je niet kunt controleren als je zegt: 'ik ga die tien dingen tegelijk doen en dan de antwoorden combineren'. In werkelijkheid zal het steeds in een andere volgorde worden uitgevoerd. Je kunt dat soort kleine dingen niet echt voorspellen, dus fouten die verbonden zijn met subtiele timing-afhankelijkheden zullen mogelijk niet optreden op je testmachine. Misschien treden ze zelfs maar één in de honderd miljoen keer op en als je het maar tien miljoen keer test, zal het waarschijnlijk niet gebeuren. Dus deploy je het. Dan is het ineens onder een veel zwaardere belasting dan in de testomgeving en dat is het moment waarop vreemde dingen gebeuren. Het is fundamenteel een ander probleem, omdat er veel meer variabelen zijn. Dus testen is een deel van het verhaal, design een ander belangrijk deel."

We moeten de manier van software schrijven veranderen om voordeel van snellere processoren te hebben

"Het blijkt dat de meeste programma's vol zitten met onschuldige fouten en daarmee kunnen we vaak wegkomen. De waarschijnlijkheid dat deze dingen tot problemen leiden, is echter groter in een concurrent programma dan in een sequentieel programma. Lang voordat we aan testen toekomen, wordt design en analyse veel belangrijker, omdat ieder van die kleine dingen die onschuldig zijn in een sequentieel programma in een concurrent programma niet meer onschuldig kunnen zijn. Het begint dus bij design en analyse, testen maakt er zeker deel van uit, statische analyse ook."

"Er bestaan tools die code kunnen analyseren op patronen die dubieus zijn. Dat is ook geen tovermiddel, maar zie het als een extra persoon die je code naloopt. Dat is misschien soms duur, maar je vindt op die manier meer bugs dan op welke andere manier ook. In werkelijkheid zul je gewoon alles moeten doen: design, statische analyse, testen, code review en samen geven ze je een zekere mate van vertrouwen."

In het licht van wat u eerder zei over het feit dat we op weg zijn naar een toekomst waarin je geen andere keus hebt dan te programmeren voor multi-threading, gaan

we een gevaarlijke toekomst tegemoet.
“De andere optie – en men begint zich daarvan bewust te worden – is kiezen voor een aanpak die minder riskant is. De basisregel is dat je moet synchroniseren. Het is makkelijk om je gedeelde data te vinden en te synchroniseren. Dat is één manier om het programma veilig te maken. Een andere is door geen data te delen. Ik heb mijn data en niet de jouwe. Wanneer we data willen delen, dan stuur ik jou een bericht en jij stuurt mij een bericht terug.”

Maar kost dat geen performance?
“Mmm, dat valt wel mee, het is een techniek die populair is bij functioneel programmeren en veel minder bij objectgeoriënteerd programmeren. Een andere techniek is het niet gebruiken van veranderbare data. Weer een techniek die populair is bij functioneel programmeren. Als je een activiteit hebt die riskant is, dan kun je bescherming zoeken of die riskante activiteit vermijden. Nu concurrency steeds alledaagser wordt, hoop ik dat men zich realiseert dat die technieken uit de wereld van het functionele programmeren aantrekkelijker worden. Ze vergen meer discipline bij het programmeren, maar niemand heeft ooit gezegd dat dit gemakkelijk zou gaan worden. Als je de discipline hebt om te kiezen voor een meer rigide manier, reduceert het de interactie en zeg je makkelijker: ‘jij schrijft deze component en ik die en we krijgen geen onverwachte interacties’. Tot nu toe is dit niet populair geweest, maar ik denk dat mensen van mening veranderen wanneer ze een keer op de blaren hebben moeten zitten.”

Er zijn dus technieken om thread safety te bereiken, maar aan de andere kant kunnen die ook negatieve effecten op de performance hebben, die zo groot kunnen zijn dat een programma helemaal tot stilstand komt. Dan krijg je deadlocks en meer in het algemeen liveness hazards.

“Dit is een gebied waarbij je moet zeggen dat het soms heel moeilijk te voorspellen wordt wanneer een systeem zich zal gaan gedragen op onverwachte manieren. Liveness hazards en met name deadlocks komen vooral voor wanneer je meerdere activiteiten hebt die op een ongecoördineerde manier toegrijpen op dezelfde data. Er zijn technieken om dit te voorkomen maar die zijn moeilijk, omdat ze of een zeer duidelijk discipline van tevoren vragen of een globale analyse noodzakelijk maken. In het ideale geval zou je de correctheid van de code daardoor bepaald willen zien. Maar bij lock analyse moet je om deadlocks te vermijden misschien wel naar het programma als geheel kijken. Hoe groter het is, hoe duurder dat wordt. Je zit dus met een keuze tussen twee kwaden: erg gedisciplineerd werken of je hebt een hoop programma-analyse.”
“Er zijn tools die je indicaties geven van de plaatsen binnen je code waar je extra aandacht aan moet geven. Dat helpt echt. Die tools zijn echter nog in de prille ontwikkelfase en ook erg duur. Een voorbeeld is Flashlight van SureLogic. Het neemt je code, voegt extra code toe aan iedere instructie en wijst vervolgens op mogelijke deadlock-situaties, ook al zijn die in werkelijkheid nog niet opgetreden. Het is een heel bruikbare techniek, maar wel duur. Ik verwacht dat na verloop van tijd de prijs van dit soort programma’s zal gaan dalen.”

In hoeverre hebben we speciale aanpassingen gezien van de JVM aan de eisen van concurrency?
“Veel van de interne JVM-veranderingen hadden van doen met parallellisme. De garbage collector was tot 1.3 een seriële single threaded collector. In 1.4 was er een

Brian Goetz gaf op Devoxx een presentatie over Java-concurrency.

parallelle collector aan toegevoegd, in 1.5 een concurrent collector en in Java 6 waren er een parallelle en een concurrent collector toegevoegd. Dus de garbage collector werd steeds meer sophisticated, terwijl hij probeerde te profiteren van de multiple processoren om de pauzetijd korter te maken. Dat was veel werk, maar het zou alleen moeten blijken als een verbetering van de performance en dat wilden we ook. Dat zal ook verdergaan. Hetzelfde is waar voor de libraries. Ze waren op een bepaalde manier geschreven in een tijd dat de meeste systemen één of hooguit twee processoren hadden. Nu zou je andere implementatie-aanpakken kiezen voor de libraries, dus in de loop van de tijd zullen de libraries aangepast worden, zodat ze beter presteren in parallelle omgevingen."

Zitten er niet te veel beperkingen aan het parallel uitvoeren van computerinstructies?

"We hebben in ieder geval te maken met de Wet van Amdahl, (zie kader) een wiskundige vergelijking die een limiet oplegt aan het versnellen van een proces als je meer werkers hebt. Een van de kernkwantiteiten is de vraag of iets is sequentieel uitgevoerd moet worden. Sommige activiteiten kunnen heel goed geparallelliseerd worden. Amdahl zegt in feite dat wanneer je een deel hebt dat niet te versnellen is en dat deel twintig procent uitmaakt van de totale tijd, je het geheel maximaal met een factor vijf kunt versnellen. Aan de andere kant geeft dat je ook een indicatie van waar je kunt beginnen je programma sneller te maken: zijn er artefacten in mijn implementatie die het parallelliseren van deze taak in de weg staan? Kan ik een ander algoritme gebruiken, andere datastructuren, kan ik mijn aanname veranderen zodat dit deel niet geserialiseerd wordt? Neem een pagina op Amazone.com. Die pagina heeft allerlei onderdelen, die voor de gebruiker samen een pagina vormen. Je zou het ook zo kunnen implementeren: pas als alle onderdelen ingevuld zijn, verstuur je de pagina. Je kunt ook na 800 ms zeggen: ik lever de pagina desnoods zonder reviews, of met een klein berichtje: 'review niet beschikbaar'. Door een beetje analyse kun je vaak veel bottlenecks wegnemen en dat blijf je doen totdat je op het punt komt waar het snel genoeg is. Dus er zijn beperkingen, maar het is verbazingwekkend wat een beetje slimheid kan doen, als je weet waar je moet kijken."

Je kunt het beste voorzichtig zijn door je eigen kunnen heel erg te wantrouwen

Tot nu toe was de bottleneck bij programma's altijd I/O, en dat is nu snel aan het veranderen. Wat betekent dat voor concurrent programmeren?

"Het verandert de vergelijking dramatisch. Allerlei afwegingen die in programma's gemaakt zijn, in de zin van geheugen versus responsetijd, veranderen nu. Nu de kosten van geheugen veranderen, worden de afwegingen ook anders. In de toekomst zal er dus veel werk ontstaan om dit soort dingen aan te passen."

Wat zijn de concurrency-veranderingen in Java 7?

"Het belangrijkste is het Fork/Join framework, het framework voor parallel decomposition. Het appelleert aan een kleine set van problemen, meestal grote datasets waarmee je grofweg dezelfde dingen doet, voor ieder element ervan. Als je dat soort wetenschappelijk werk of business analyse doet is het erg handig, want je krijgt sneller antwoord. Als je andere dingen doet, is het niet van belang. Sommige ontwikkelaars zullen het nooit gebruiken en sommigen zullen het een extreem handig framework vinden om de manier waarop ze dit soort problemen aanpakken te organiseren. Maar de komst van Java 7 zal nog wel even op zich laten wachten, begin 2010, al zal er wel een preview zijn op JavaOne."

Zou u een paar adviezen kunnen geven voor beginners op dit gebied – afgezien van het lezen van uw boek?

"Wat ik ontdekt heb, is dat veel van de technieken die je specifiek kunt gebruiken om je concurrent programma veiliger te maken, in feite speciale gevallen zijn van heel oude wijsheden over programmeren. Een voorbeeld: het is een goed idee om data in te sluiten. Verberg je data, zodat alleen dit kleine beetje code die data kan zien in plaats van de gehele wereld. Dat simplificeert je programma, omdat je nu alleen dat hele kleine stukje code hoeft te analyseren om er zeker van te zijn dat dit dataitem nooit gecorrumpeerd wordt. Dat is een algemeen principe, en het blijkt dat veel van die algemene principes dezelfde dingen zijn die het gemakkelijker maken om concurrent code te schrijven. Om terug te komen op wat we net bespraken: vele programma's bevatten onschuldige fouten. De fouten komen niet tot het niveau waarop dingen fout gaan, maar in deze meer vijandige omgeving worden zulke fouten gevaarlijker. Veel concurrent programmeren is alleen maar zorgvuldig programmeren: volg de regels, overtuig jezelf er niet van dat je de regels kunt overtreden, omdat er niets verkeerds zal gebeuren. Beperk je tot het volgen van de regels en veel van de slechte dingen die zouden kunnen gebeuren, zullen zeer waarschijnlijk niet gebeuren."

"Het omgekeerde is dat je jezelf ervan overtuigt: 'ik overtreed de regels hier en ik heb over alle dingen die mis kunnen gaan nagedacht en ik kan daarmee leven'. Dat maakt het noodzakelijk dat je alle dingen kunt bedenken die fout kunnen gaan. Maar als je voorstellingsvermogen onvoldoende is, dan gaat het mis. Je moet dus voorzichtig zijn. Ik ben van mening dat je het beste voorzichtig kunt zijn door je eigen kunnen heel erg te wantrouwen. Hoe meer je denkt: 'ik ben geweldig, ik kom overal mee weg', des te waarschijnlijker is het er iets verschrikkelijk mis zal gaan." «

Wet van Amdahl

In het geval van parallellisering zegt de Wet van Amdahl dat wanneer P het deel van het programma is dat kan worden geparallelliseerd (en dus voordeel heeft van de parallellisering) en (1 – P) het deel is dat serieel moet worden uitgevoerd, de maximale versnelling die kan worden bereikt bij een N-aantal processoren is:

$$\frac{1}{(1-P)+\frac{P}{N}}$$