Het functionele paradigma is de toekomst.

Sjors Sparreboom

Hogeschool Rotterdam
0890040@hr.nl

Johan Bos

Hogeschool Rotterdam
0878090@hr.nl

11 januari 2018

Samenvatting

Met de komst van het internet, big data en de toenemende capaciteit van computers is er vernieuwde interesse in de functionele programmeertalen. Dit rapport bekijkt samen met de lezer de voor en nadelen bij de stelling "Het functionele paradigma is de toekomst." Dit gebeurt op basis van een literatuuronderzoek naar de relaties en verschillen van de heersende programmeerparadigma's. Het onderzoek beperkt zich tot het imperatieve en functionele paradigma. Om het functionele paradigma en zijn plaats binnen de informatica beter te kunnen begrijpen is er binnen dit onderzoek van een minimale kennisgeving van de heersende programmeerparadigma's uitgegaan. Ook is kennis van programmeren wenselijk, maar het onderzoek kan ook prima zonder deze kennis worden gevolgd; terminologie (cursief) waar geen verwijzing voor is kan worden gevonden in paragraaf A.

I. Inleiding

Dit document is een poging om de gemeenschap te overtuigen van de significantie van functioneel programmeren als ook te informeren wat de voordelen en nadelen zijn van dit paradigma.

Functioneel programmeren dankt zijn naam aan zijn fundamentele werking, namelijk de toepassing van functies op argumenten. Dit beschrijft heel mooi wat programmeren inhoudt, maar wat is nu eigenlijk een functioneel programma? Een mooie beschrijving van een functioneel programma is die van Hughes [8]: „Een functioneel programma is een functie waar de gegeven input voor dit programma wordt behandeld als argument van een functie, welk vervolgens het resultaat hiervan terugeeft als output".

Het functioneel programmeren is niet nieuw en bestaat al sinds de jaren 40. De fundamentele werking van deze stijl van programmeren is gebasseerd op het *lambda calculus*[2, 16]. Typerend voor deze stijl is dat functies opgebouwd zijn uit andere functies. Een mooi voorbeeld is de hoofd functie (het programma zelf), deze is opgebouwd uit functies, welke op hun beurt ook weer opgebouwd zijn uit functies; dit heet modulariteit een concept dat wordt behandeld in paragraaf ii. Ook zijn functionele talen inherent parallel [15, 22] en hebben functionele programma's geen *assignments*, waardoor variabelen met een waarde niet meer kunnen veranderen, meer hierover in paragraaf i.

In de rest van dit document zal er worden gediscussieerd over de voor- en nadelen van deze vorm van programmeren en zal er in paragraaf V af worden gesloten met een conclusie op de stelling *"Het functionele paradigma is de toekomst"*.

II. Middenstuk

In dit middenstuk wordt er eerst korte uitleg gegeven over imperatief programmeren. De geïformeerde lezer kan ervoor kiezen deze paragraaf over te slaan, maar moet onthouden dat kennis van imperatief programmeren benodigd is in de volgende paragrafen.

Opvolgend zal er worden gekeken naar de voordelen en nadelen van de stelling *"Het functioneel programmeren is de toekomst"*.

i. Imperatief programmeren

In paragraaf I werd de functionele vorm van programmeren behandeld. In dit paragraaf wordt een nog oudere vorm van programmeren behandeld; imperatief programmeren.

De term imperatief programmeren is een letterlijke beschrijving van het gedrag dat deze programma's vertonen, namelijk het sequentieel uitvoeren van een lijst van instructies. Echter is het vanuit deze context niet duidelijk wat een imperatief programma is. Een duidelijke beschrijving van imperatief programmeren wordt gegeven door Chailloux, Manoury en Pagano [3, hfdst. 3] en luidt als volgt:

> Imperatief programmeren staat in tegenstelling tot functioneel programmeren dichtbij de machine representatie. Het introduceert een *memory state* die door acties van het programma kunnen worden gemodificeerd. We noemen deze acties van een programma *instructies*; een imperatief programma is dan ook een lijst van deze instructies.

Kort samengevat is het een stijl van programmeren dat direct geïnspireerd is door assembleer, en net als in assembleer zijn imperatieve programma's in staat de memory state van een programma aan te passen. Input-output acties zijn in imperatieve programma's dan ook modificaties van het geheugen.

III. Algemene voordelen functioneel programmeren

Uit het onderzoek van TIOBE [21] en IEEE [9] blijkt dat de imperatieve stijl hedendaags nog steeds de populairste vorm van programeren is. Wat echter opvalt is dat er enkele functionele talen in deze resultaten (Erlang, Scala, F, Haskell, Clojure) staan, en dit is niet toevallig. Na Backus [1] Turing Award lezing in 1977 was er een verandering binnen de academische wereld. Veel academici en wetenschappers waren genïnspireerd en begonnen te investeren in deze stijl van programmeren; erg verassend aangezien deze vorm van programmeren na de jaren 60 als een een onconventionele, onwerkbare vorm van software ontwikkeling werd gezien. Het moest echter nog een kleine 30 jaar duren voordat er buiten de academische en wetenschappelijke hoek nieuwe interesse was voor deze vergeten stijl van programmeren. In deze paragraaf wordt de aanleiding van deze nieuw gevonden interesse onderzocht.

i. Parallel en Gedistribueerd programmeren

Sinds de uitvinding van het internet en *big data* is er buiten de academische wereld hernieuwde interesse ontstaan in de functionele talen. Zoals beschreven door Wu e.a. [25] zijn er met komst van big data nieuwe manieren van dataverwerking nodig, aangezien de traditionele manieren niet langer toereikend zijn. Deze nieuwe manieren van dataverwerking baseren zich op twee vormen van computatie die door de goedkopere en verbeterede hardware componenten en de komst van de *multi-core processor* een geheel nieuw domein aan mogelijkheden openen; namelijk *parallel* en *gedistrubeerd* computeren [5, 25].

Het is dan ook om deze redenen dat er hedendaags nieuwe interesse is in functioneel programmeren. Voordat we echter kunnen vertellen waarom fuctionele programma's zo geschikt zijn voor parallel en gedistrubeerd programmeren, zullen we eerst kijken naar wat de valkuilen zijn van deze vormen van pro-

grammeren en waarom de imperatieve stijl van programmeren hiervoor minder geschikt wordt geacht.

Volgens Jones e.a. [11] is gedistribueerd en parallel computeren complex en niet zonder enige risico's. Het is complex in de zin dat voor het uitvoeren van een programma op een gedistrubeerde of parallele manier het curciaal is dat de evaluatie volgende van het programma klopt. Dit omdat de proccessoren niet op dezelfde snelheid een programma evalueren en executeren; ze hebben dus niet dezelfde *throughput*. In een imperatief programma is dit ëën van de grootste oorzaken van bugs [8]. Zoals te zien is in paragraaf i maken deze programma's gebruik van een memory state, welke aangepast kan worden door de instructies van het programma. Als deze aanpassingen niet in de bedoelde volgorde gebeurt wordt het een bug en is er sprake van een *race conditie*.

In *Programming in the Age of Concurrency: Software Transactional Memory* vertellen Peyton-Jones en Harris dat in imperatieve programma's een concept genaamd *locks* (ook wel *mutexes* genoemd) is geïntroduceerd om deze race condities tegen te gaan. Deze oplossing werkt, maar introduceert op zijn beurt weer een nieuw probleem, namelijk *deadlock*. Deze toestand genaamd deadlock is het gevolg van de mutaties op data die mogelijk zijn in een imperatief progamma. Deze situatie van locks en deadlocks an ook een klassiek voorbeeld van een *side-effect*.

Het is dus wenselijk een side-effect vrij programma te hebben, maar hoe is dit mogelijk? Volgens Peyton-Jones en Harris [18, 19] ligt de oplossing in het gebruik van een functionele programmeertaal. Functionele programmeertalen hebben geen van de bovengenoemde problemen. Zo zijn ze inherent parallel en is data in deze programma's niet muteerbaar, omdat er geen *assigment* declaratie is. Dit elimineert gelijk een belangrijke bron van bugs, zoals de deadlock en race conditie en maakt de executie volgorde van deze programma's irrelevant.

ii. Schaalbaar en Modulair

In paragraaf I werd al kort aangehaald dat functionele programma's modulair zijn doordat functies opgebouwd zijn uit andere functies, een concept genaamd *currying* [14, hfdst. 5].

Het voordeel van currying is door deze manier van functie applicatie, functies gedeeltelijk kunnen worden toepast [8]. Door het gedeeltelijk toepassen van deze functies, kunnen er nieuwe functies van oude afgeleid worden, functies zijn dus erg schaalbaar. Dit maakt dat functionele programma's gemakkelijk modulair kunnen worden opgebouwd; wat volgens Jones [10] wordt beschouwt als één van de belangrijkste punten in het succesvol construeren van herbruikbare en onderhoudbare software. Ook spreken Chowdhury en Iqbal [4] hun voorkeur uit over de modulaire aanpak van het ontwerpen van software, omdat er hierdoor toekomstige legacy problemen, zoals wordt besproken in paragraaf ii kunnen worden voorkomen.

Dat modulair programmeren een pre is blijkt uit de bovengenoemde bronnen, maar wat is een module nu precies? Lipvača [14, hdfst. 6] beschrijft een module als: „Een module een collectie van gerelateerde functies, types en typeclasses". Wat types en typeclasses zijn is in dit document niet relevant. Wat wel relevant is dat door de modulariteit en schaalbaarheid van functionele programma's veel algoritmes in vaak korter zijn dan in een imperatief programma. Zo zien we hieronder een twee versies van een quicksort algoritme; een imperatieve versie in C en een functionele versie in Haskell.

```
void Sort(int input[],int left,int right)
{
  int L = left;
  int R = right;
  int M = input[(left + right) / 2];

  do
  {
    while(input[L] < M)
    {
      L++;
    }
    while(M < input[R])
    {
     R−−;
    }
```

```
    if(L <= R)
    {
      int inputLeft = input[L];
      input[L] = input[R];
      input[R] = inputLeft ;
      L++;
      R−−;
    }
  } while(L < R);

  if(left < R)
  {
    Sort(input, left, R);
  }
  if(L < right)
  {
    Sort(input, L, right);
  }
 }

 void QuickSort(int input[], int length)
 {
  Sort(input, 0, length − 1);
 }
}
```

```
quicksort :: Ord a => [a] → [a]
quicksort []     = []
quicksort (p:xs) =
  (quicksort lesser) ++ [p] ++ (quicksort greater)
   where
     lesser  = filter (< p) xs
     greater = filter (>= p) xs
```

Wat we in het bovengenoemde voorbeeld zien is dat de tweede versie korter en leesbaarder is dan de eerste versie. Ook valt op dat de tweede versie verklarend en expressief is. Dit in tegenstelling tot de de eerste versie, waar we duidelijk de imperatieve aard terug zien in de sequentieel instructies die door het algortime dienen te worden uitgevoerd.

IV. Algemene nadelen functioneel programmeren

Dat de functionele stijl van programmeren tegenwoording steeds vaker wordt overwogen, zagen we in paragraaf III. Er zijn echter ook gebieden en problemen die zich niet schikken voor een functionele aanpak van het probleem. In paragraaf i en paragraaf ii wordt er dan ook onderzocht waar deze desinteresse vandaan komt.

i. Portabiliteit en Beschikaarheid

Beschikbaarheid en portabiliteit van functionele talen op verschillende architecturen en platformen is een grote reden waarom het soms beter is te besluiten tegen het gebruik van deze talen en te kiezen voor een imperatieve tegenhanger. Er zijn dan ook genoeg projecten waarvan de ontwikkelaars liever het project hadden gebouwd in een functionele taal, maar vervolgens toch besloten te kiezen voor C of C++. Eén van deze projecten is de taal voor de PRL database. Volgens Wadler [23] besloten onderzoekers van Lucent om deze taal te bouwen in SML, om ultimatum te kiezen C++, omdat SML niet beschikbaar was op het Amdahl mainframe.

Dezelfde problemen zien we in *embedded systems* [7]. Bij het programmeren van deze systemen moet er rekening worden gehouden dat de architectuur van deze veel kan verschillen van de algemene computer en dat er weinig intern geheugen beschikbaar is [17, 24]. Om toch goede programma's te kunnen schrijven voor deze systemen is het dus noodzakelijk voor de programmeur om direct het geheugen te kunnen controleren.

Functionele programma's alloceren echter in korte tijd veel geheugen. Dit is geen probleem op systemen met genoeg geheugen, maar wordt een probleem in een embedded omgeving. Om ervoor te zorgen dat voldoende geheugen beschikbaar is, worden zogenaamde cells na gebruik vrijgegeven door een *garbage collector* [20]. Deze afhankelijkheid van garbage collectie zorgt ervoor dat er beperkte controle is over het geheugen. Dit gebrek over controle van het geheugen maakt de meeste moderne functionele talen niet geschikt voor embedded systemen, zeker de systemen met weinig geheugen of harde *real-time* beperkingen [12].

ii. Legacy code en Infrastructuur

Er is veel commericiële software in de omloop dat is gedateerd en kan profiteren van de voordelen van functioneel programmeren (zie paragraaf III). Om deze voordelen te benutten moet de huidige infrastructuur van deze systemen worden veranderd, wat in de praktijk een nogal lastig en gevoelig onderwerp blijkt te zijn. Er is namelijk hevig geïnvesteerd in deze systemen

en is de werking van ervan vaak verwoven met de gebruiksdoeleinde [4, 13].

Vervanging is dus lastig, zeker als er nagegaan wordt dat na decennia lang schrijven van software, systemen vaak samengesteld zijn uit componenten. Veel van deze componenten zijn geschreven in een imperatieve programmeertaal zoals C of C++, waardoor een *foreign function interface* met C, en andere interfaces naar andere talen vaak noodzakelijk is om de achterwaartse comptabiliteit te garanderen[23]. Echter door de isolationistische natuur van functionele talen zijn deze foreign function interface moeilijk implementeerbaar is, zeker voor functionele talen die gebruik maken van *lazy evaluation* [6, p. 149].

V. Conclusie

In dit document is er gediscussieerd over de voordelen en nadelen van het functioneel programmeren. In paragraaf III werd duidelijk dat met de komst van het internet en de ontwikkelingen op hardware gebied er hernieuwde interesse is voor functioneel programmeren. De schaalbaarheid, modulariteit, niet muteerbare datastructuren en het inherent vermogen om parallel expressies te evalueren maken functionele talen een uitermate geschikt stuk gereedschap om software te schrijven voor de 21^e eeuw.

Het is echter misschien iets te vroeg om te stellen dat de functionele stijl van programmeren de toekomst is. Zo zien we nog steeds geen functionele opmars in de embedded systems hoek en worden legacy problemen in grote commericiële software systemen nog altijd imperatief opgelost.

Er is een glansrijke toekomst voor de functionele stijl van programmer, maar vooralsnog moet er goed gekeken worden naar de probleemstelling en het doel voor er paradigmatisch besluit kan worden genomen.

Referenties

[1] John Backus. „Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs". In: *Commun. ACM* 21.8 (aug 1978), p. 613–641. ISSN: 0001-0782. DOI: 10.1145/359576.359579.

[2] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics. Elsevier Science, 2013. ISBN: 978-0-08-093375-7.

[3] Emmanuel Chailloux, Pascal Manoury en Bruno Pagano. *Developing Applications with Objective Caml*. OREILLY, 2000, p. 1–742. ISBN: 2-84177-121-0.

[4] Maria Wahid Chowdhury en Muhammad Zafar Iqbal. *Integration of Legacy Systems in Software Architecture*. Tech. rap. University of Victoria, Department of Computer Science e.a.

[5] Raul Castro Fernandez e.a. *Making State Explicit for Imperative Big Data Processing*. Tech. rap. Imperial College London, University of Kent en City University London.

[6] Hagiya, Masami en Philip Wadler. *Functional and Logic Programming: 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings (Lecture Notes in Computer Science)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. ISBN: 3-540-33438-6.

[7] Thomas A. Henzinger en Joseph Sifakis. „The Embedded Systems Design Challenge". In: *Proceedings of the 14th International Symposium on Formal Methods (FM), Lecture Notes in Computer Science*. Springer, 2006, p. 1–15.

[8] John Hughes. *Why functional programming matters*. Tech. rap. Glasgow University, 1989.

[9] IEEE. *The 2017 top programming languages*. 2017. URL: https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages.

[10] Simon Peyton Jones. „An exploration of modular programs". In: *In The Glasgow Workshop on Functional Programming*. 1996, p. 96.

[11] Simon Peyton Jones e.a. *Harnessing the Multicores: Nested Data Parallelism in Haskell*. Tech. rap. University of New South Wales, School of Computer Science, Engineering en Microsoft Research Ltd, 2008.

[12] Martin Kero en Simon Aittamaa. *Scheduling garbage collection in real-time systems*. Tech. rap. Jan 2010, p. 51–60.

[13] B. P. Lientz, E. B. Swanson en G. E. Tompkins. „Characteristics of Application Software Maintenance". In: *Commun. ACM* 21.6 (jun 1978), p. 466–471. ISSN: 0001-0782.

[14] Miran Lipvača. *Learn You a Haskell for Great Good!* 1ste ed. No Starch Press, apr 2011, p. 400. ISBN: 1-59327-283-9.

[15] Björn Lisper. *Functional Programming and Parallel Computing*. Tech. rap. Mälardalen University, School of Innovation, Design en Engineering, 2013.

[16] Greg Michaelson. *An Introduction to Functional Programming Through Lambda Calculus*. Dover books on mathematics. Dover Publications, 2011. ISBN: 978-0-486-47883-8.

[17] Gergely Patai. *Functional Programming and Embedded Systems*. Tech. rap. Budapest University of Technology, Economics, Faculy of Electrical Engineering en Informatics, 2006.

[18] *The Future is Parallel, and the Future of Parallel is Declarative*. 2012.

[19] Simon Peyton-Jones en Tim Harris. *Programming in the Age of Concurrency: Software Transactional Memory*. 2017. URL: https://channel9.msdn.com/shows/Going+Deep/Programming-in-the-Age-of-Concurrency-Software-Transactional-Memory/.

[20] Patrick M. Sansom en Simon L. Peyton Jones. „Generational garbage collection for Haskell". In: *In Functional Programming Languages and Computer Architecture*. ACM Press, 1993, p. 106–116.

[21] TIOBE. *TIOBE Index for January 2018*. Jan 2018. URL: http://www.tiobe.com/tiobe-index.

[22] G. Tremblay en G. R. Gao. „The Impact of Laziness on Parallelism and the Limits of Strictness Analysis". In: *PROCEEDINGS HIGH PERFORMANCE FUNCTIONAL COMPUTING*. 1995, p. 119–133.

[23] Philip Wadler. „Why No One Uses Functional Languages". In: *SIGPLAN Not.* 33.8 (aug 1998), p. 23–27. ISSN: 0362-1340.

[24] Malcolm Wallace. „Functional Programming and Embedded Systems". Proefschrift. University of York, 1995.

[25] Xindong Wu e.a. *Data Mining with Big Data*. Tech. rap. Hefei University of Technology, School of Computer Science en Information Engineering.

A. Terminologie

Assignment:

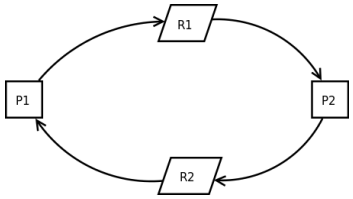
Een *assignment* stelt de waarde in die is opgeslagen in de opslaglocatie die wordt aangeduid met een variabelenaam.

Big Data:

Data sets die zo enorm en complex zijn dat traditionele dataverwerkingssoftware niet langer toereikend zijn.

Deadlock:

Deadlock, zoals te zien in fig. 1 is een toestand waarin elk proces wacht op een ander proces. Deze toestand kan voorkomen wanneer de locks worden verkregen in de verkeerde volgorde.



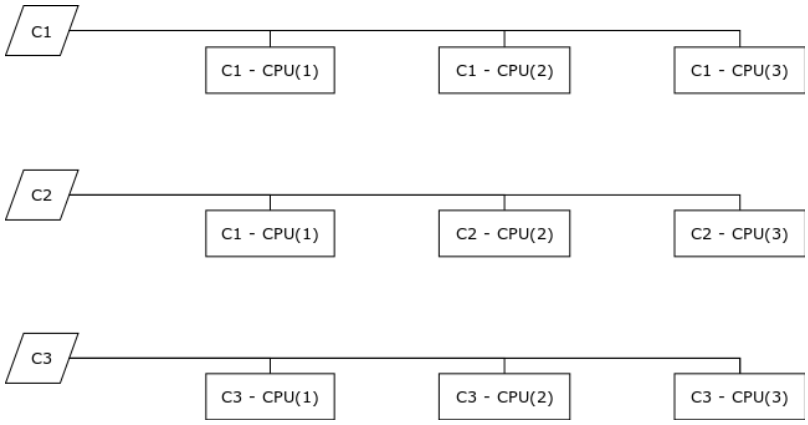
Figuur 1: *Deadlock (source: wikipedia.org/wiki/Deadlock)*

Foreign function interface:

Een *foreign function interface* (FFI) is een mechanisme waarmee een programma dat in een programmeertaal is geschreven, routines kan noemen of gebruik kan maken van services die in een andere taal zijn geschreven.

Gedistribueerde computatie:

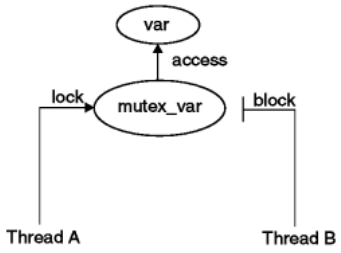
Een *gedistribueerd computatie* is een computatie waar een enkele taak wordt opgesplitst in meerdere subtaken. Deze subtaken worden vervolgens gedistribueerd over verschillende nodes in een netwerk en onafhankelijk berekent. Als al deze computaties gedaan zijn zullen deze onafhankelijke berekeningen worden samengevoegd tot één resultaat, zoals te zien in fig. 2.



Figuur 2: *Gedistribueerd proces*

Lock/Mutex:

Lock, zoals te zien in fig. 3 is een mechanisme om het benaderen van een bron in een omgeving te beperken.



Figuur 3: *Lock/Mutex*

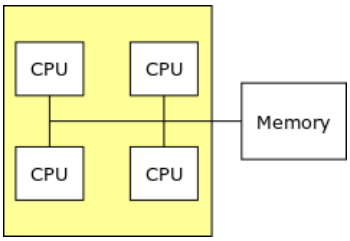
Memory leak:

Een *Memory leak* betekent dat een programma meer geheugen toewijst dan nodig voor de executie. Dit kan over tijd erg oplopen aangezien dit niet gebruikte geheugen nooit opnieuw toegewezen kan worden, zonder het wordt gedealloceerd.

Multi-core processor:

Multi-core processoren, bestaan uit verschillende processoren en gedeeld geheugen, zie fig. 4. Deze processoren zijn sneller en hebben een lager energieverbruik dan de single-core processor, zie fig. 7, echter is het programmeren voor deze architectuur een stuk lastiger.

Memory state:



Figuur 4: *Multi-core processor*

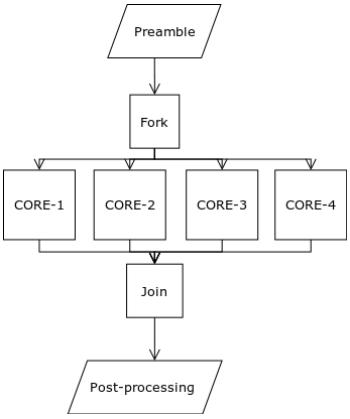
Memory state alle opgeslagen ingegeven waardes van een programma als variabelen of constanten. Bij het analyseren van de status van dit programma kunnen programmeurs deze opgeslagen ingegeven waardes bekijken.

Parallele computatie:

Een *parallele computatie* is een vorm van computatie waarin meerdere calculaties simultaan kunnen worden uitgevoerd. Deze vorm van executie is sinds de komst van de multi-core processor, zie fig. 4 het dominante paradigma in computer architectuur.

Een parallelle executie van een proces wordt gedaan door gebruik te maken van threads, zie fig. 6. In fig. 5 zien we een parallel executie op een multi-core processor (fig. 4).

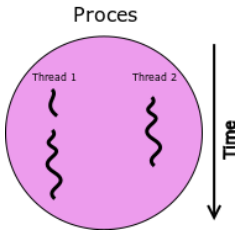
Wat opvalt zijn de extra kernen van deze architectuur, die een single-core processor, zoals te zien in fig. 7 niet heeft. Als deze verschillende kernen zinnig werk tegelijkertijd doen kan dit het gehele proces versnellen.



Figuur 5: *Parall proces*

Process & Thread:

Een *thread* is de kleinste sequentie van een programma dat onafhankelijk kan opereren. Hoe een proces en thread opereren hangt af van de implementatie van het operatie systeem. De meeste voorkomende implementatie is dat een thread een onderdeel is van het proces, zoals te zien in fig. 6.



Figuur 6: *Thread*

Race conditie:

Twee processen P1 en P2, kunnen beide schrijven naar de variabel tmp. De intentie van de programmeur is dat zowel proces P1 als P2 tmp van een waarde voorziet, deze vervolgens gelijk te gebruiken en een uiteindelijke waarde voor y en z te presenteren. De executie van algoritme algoritme 1 levert de volgende waardes voor y en z:

$$y = 24, z = 2055$$

Echter doordat de processen op verschillende snelheden kunnen draaien, doet zich een situatie voor, zoals te zien in algoritme algoritme 2 waar de volgorde van executie niet wordt uitgevoerd zoals de programmeur had bedoeld. De executie van dit algoritme levert de volgende waardes voor y en z:

$$y = 4096, z = 2055$$

Zoals te zien is dit niet de juiste waarde voor y. Deze situatie wordt ook wel een race conditie genoemd in de informatica.

Algorithm 1 Programmeurs intentie

Shared Variable
tmp
end Shared Variable

procedure P1	**procedure** P2
.	$tmp = 2048$
.	$z = tmp + 7$
.	.
$tmp = 12$	.
$y = 2 \times tmp$	.
end procedure	**end procedure**

Algorithm 2 Race conditie

Shared Variable
tmp
end Shared Variable

procedure P1	**procedure** P2
$tmp = 12$	.
.	$tmp = 2048$
$y = 2 \times tmp$	.
.	$z = tmp + 7$
.	.
end procedure	**end procedure**

Real-time proces:

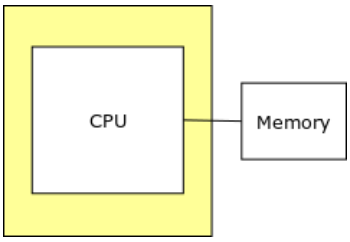
Een *real-time proces* is een proces waarin een programma het resultaat moet garanderen binnen een bepaalde tijd.

Side-effect:

Binnen de informatica heeft een programma, functie of expressie een *side-effect* wanneer het een waarde buiten zijn scope aanpast of een interactie heeft met zijn aanroep functies of de rest van het programma zonder een waarde te retourneren.

Single-core Processor:

Single-core processoren hebben maar 1 processor direct verbonden aan het gerheugen. Deze processoren zijn gemakkelijk te programmeren, maar hebben echter een lagere hoog energieverbruik en een lage prestatie, in tegenstelling tot de multi-core processor, zie fig. 4.



Figuur 7: *Single-core processor*

Throughput:

Throughput is de maximale snelheid waarop *iets* kan worden verwerkt.