WORDT
NIET UITGELEEND

# Booleaanse Netwerken

Een snelle implementatie van een booleaanse netwerk voor genetische netwerken die gebruik maken van de Cray SV1e bit matrix architectuur.

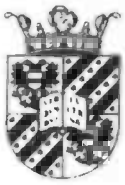
Rijksuniversiteit Groningen
Bibliotheek Wiskunde & Informatica
Postbus 800
9700 AV Groningen
Tel. 050 - 363 40 01

Brend A. Dikkers

1 december 2003

Afstudeerdocenten: Dr. M.H.F. Wilkinson en Ir. S. Achterop

WORDT
NIET UITGELEEND



# Booleaanse Netwerken

Een snelle implementatie van een booleaanse netwerk voor genetische netwerken die gebruik maken van de Cray SV1e bit matrix architectuur.

Rijksuniversiteit Groningen
Bibliotheek Wiskunde & Informatica
Postbus 800
9700 AV Groningen
Tel. 050 - 363 40 01

Brend A. Dikkers

Kootstertille, 1 December 2003

Afstudeeropdracht in het kader van de studie Informatica
aan de Rijksuniversiteit Groningen.

Afstudeerdocenten: Dr. M.H.F. Wilkinson en Ir. S. Achterop

# Voorwoord

Ter afsluiting van mijn opleiding Informatica aan de Rijksuniversiteit Groningen heb ik gekozen voor de afstudeerrichting Scientific Computing and Imaging. De afstudeeropdracht van dr. M.H.F. Wilkinson sprak mij het meest aan omdat er gebruik gemaakt moest worden van de Cray SV1e bit matrix multiplyer.

Ik heb onderzocht of booleaanse netwerken ook sneller doorgerekend kunnen worden door gebruikt te maken van de Cray SV1e bit matrix multiplyer. Tevens heb ik onderzocht of de bewering het aantal limiet cycli in een netwerk een lineaire functie was van het aantal vertices $N$, of dat het schaalde met $\sqrt{N}$, zoals ook wordt beweerd.

Het Centre for High Performance Computing and Visualization (HPC&V) gaf mij de toegang tot hun Cray SV1e. Daarvoor mijn grote dank. Hierbij bedank ik de medewerkers drs. D. Homan en dr. R. de Bruin voor hun raadgevingen. In het bijzonder bedank ik dr. M.H.F. Wilkinson en ir. S. Achterop voor de begeleiding.

# Inhoudsopgave

**Inleiding** 1

**1 De Cel** 3

**2 Modellen** 5
2.1 Grafen . . . 5
2.2 Differentiaal vergelijkingen . . . 5
2.3 Booleaanse netwerken . . . 6

**3 Bit Matrix Multiply** 9
3.1 Wat is de Bit Matrix Multiply ? . . . 9
3.2 BMM mogelijkheden . . . 10
3.3 BMM programma instructies . . . 11

**4 Fortran code** 15
4.1 Pseudocode . . . 15
4.2 De code . . . 16
4.3 Aanpassingen . . . 18
4.4 Vectorizeren . . . 19

**5 Efficientie verbeteren** 21
5.1 BMM code . . . 21
5.1.1 Functies in BMM code . . . 21
5.1.2 Meerdere regels in één keer bereken . . . 26
5.1.3 Beperkingen . . . 28
5.2 Reduceren van het netwerk . . . 29
5.2.1 De reductie . . . 29
5.2.2 Reductie resultaten . . . 36
5.3 Resultaten . . . 36

**6 Het dynamisch gedrag** 37
6.1 Aanpassingen aan de code . . . 37
6.2 Het onderzoek . . . 38
6.3 Hoeveel begintoestanden moeten er gebruikt worden ? . . . 41

**Aanbevelingen** 45

**Samenvatting** 47

**Abstract** 49

**Bibliografie** 51

**Bijlagen** 53
A-1 De gehele fortran code versie 1 . . . . . I
A-2 De gehele fortran code versie 3 . . . . . VII
A-3 De gehele fortran code versie 7 . . . . . XIII

# Inleiding

Booleaanse netwerken zijn in 1969 geïntroduceerd door Kauffman [10] als een eenvoudig model voor de complexe interactie in een regulatie systeem van levende cellen. Tevens maken booleaanse netwerken het voor onder andere biologen mogelijk om regulatie netwerken te analyseren op een efficiënte manier. Het is met name een goed "strategisch" model van echte netwerken. Met strategisch wordt hier bedoeld dat het een kwalitatief model voor een grote klasse van systemen, niet een kwantitatief ("tactisch") model voor een specifiek systeem.

Kauffman heeft, in 1969, alleen netwerken kleiner dan 10.000 elementen onderzocht. Dit omdat de computers in die tijd niet krachtig genoeg waren om grote netwerken intensief te berekenen. De huidige computers zijn zo krachtig dat netwerken van enkele tienduizenden elementen binnen enkele seconden zijn te berekenen. Dit stelt ons in staat om genetische processen nauwkeuriger na te bootsen. Een bacterie bevat tussen de 3.000 en de 6.000 genen, dus tussen de 3.000 en de 6.000 elementen in een booleaanse netwerk. Voor de mens loopt dit op tot 30.000 elementen.

Deze grote netwerken zijn te berekenen door òf het netwerk te reduceren òf door gebruik te maken van een krachtige computer. Hier is gekozen voor de krachtige computer, Cray SV1e. Omdat deze aparte hardware heeft die geschikt is om bit bewerkingen snel te kunnen uitvoeren. Tevens heeft eerder onderzoek bij de Rijksuniversiteit Groningen snelheidswinst aangetoond bij string manipulatie.

Dit verslag is als volgt ingedeeld. In hoofdstuk 1 wordt de werking van een cel beschreven. Hoofdstuk 2 worden verschillende modellen uitgelegd die een cel kunnen nabootsen. Er wordt uitgelegd hoe ieder model werkt en wat de positieve en negatieve punten zijn. De booleaanse netwerken worden wat uitvoeriger behandelt. In hoofdstuk 3 wordt de bit matrix multiplier architectuur uitgelegd. Door middel van enkele voorbeelden worden de mogelijkheden van de architectuur duidelijk gemaakt. De fortran code die gemaakt is om een booleaanse netwerk te maken wordt in detail uitgelegd in hoofdstuk 4. Alle strategieën voor verbeteringen van de efficientie die gebruikt zijn (bit matrix multiply en reductie) wordt beschreven en beoordeeld in hoofdstuk 5. In hoofdstuk 6 wordt de snelste code gebruikt om na te gaan hoe het dynamische gedrag is van booleaanse netwerken. Ook word er gekeken naar welke van twee strijdige beweringen over het dynamisch gedrag in [3] en [10] juist is. Er wordt afgesloten met enkele aanbevelingen en in de bijlagen staan verschillende source codes.

# Hoofdstuk 1

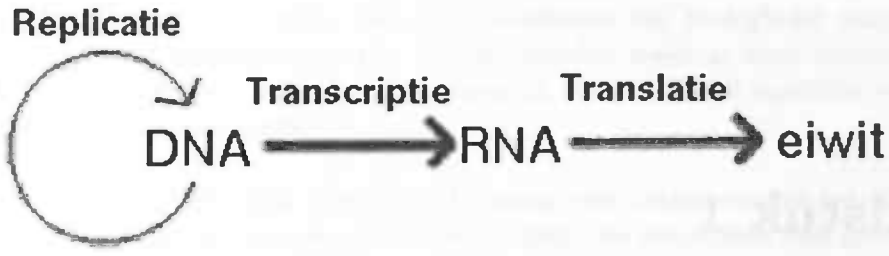
# De Cel

Alle levende organismen zijn opgebouwd uit één of meer cellen. Een cel kan het beste worden gedefinieerd als [6, 9] : de kleinste georganiseerde levende eenheid van een organisme dat in staat is tot een onafhankelijk bestaan, mits in een geschikte omgeving, dankzij een ingewikkelde metabolisme, daarbij in staat zijnde tot deling. Het gaat bij de cel dus om een eenheid van organisatie.

De cellen zijn daarbij voortdurend bezig met de vervanging van hun samenstellende bestanddelen, zoals eiwitten. Hierbij wordt een voorname rol gespeeld door de informatie afkomstig van het genetisch apparaat.
Ieder eiwit is gecodeerd in één bepaald gebeid van het DNA: het gen. Het omzetten van de DNA code naar een eiwit gebeurt volgens twee stappen: transcriptie en translatie.
Transcriptie is de synthese van een enkel-strengs RNA kopie van de informatie uit de dubbelstrengs DNA molecuul. Een specifieke volgorde van DNA, de promotor genoemd, geeft het begin voor de transcriptie aan. Translatie is de vertaling van de RNA code in een keten van aminozuren (eiwit). Deze eiwit synthese wordt uitgevoerd door ribosomen. Ribosomen zijn opgebouwd uit een grote en een kleine subunit dat samen de machine voor de eiwit synthese vormt. De kleine subunit bindt het RNA, en de grote subunit bevorderd de opbouw van het groeiende eiwit.

De instandhouding van organismen is afhankelijk van de vorming van nieuwe cellen door deling van de bestaande cellen.
De reeks van opeenvolgende gebeurtenissen, die leiden tot de vermeerdering van cellen wordt de cel cyclus genoemd. De duur van deze cyclus is het interval tussen twee gelijke toestanden uit de reeks van opeenvolgende stadia tijdens het delingsproces. Tijdens het delingsproces wordt de DNA-helix verdubbeld. Vervolgens wordt het gedupliceerde DNA afgescheiden en ontstaat er een nieuwe cel.

Cellen kunnen worden ingedeeld in twee categorieën: De prokaryotische cel (b.v. bacterie) met een eenvoudige structuur, en de eukaryotische cel (b.v. mens) met een meer complexe organisatie.




Figuur 1.1: Dit geeft precies de gebeurtenissen die in een cel kunnen voorkomen weer.

Aangezien de gen expressie en regulatie van bacteriën eenvoudiger is, wordt deze vaak als basis voor onderzoek gebruikt. Onderzoek wordt gedaan met behulp DNA microarrays waarin waargenomen wordt hoe RNA in de tijd fluctueren en ook van simulatie modellen. Deze simulatie modellen worden onder andere gebruikt voor het doorgronden van dynamische reacties op de omgeving en op interne signalen van zowel normale als gemuteerde cellen. Enkele van deze modellen zullen in de volgende hoofdstuk verder worden uitgewerkt.

# Hoofdstuk 2

# Modellen

Er zijn veel verschillende modellen om een cel, en dan in het bijzonder de genetische regulatie systemen van een cel, te modelleren [8, 11]. Hieronder worden enkele modellen uitwerkt.

## 2.1 Grafen

Een graaf is waarschijnlijke de meest eenvoudige manier om een genetisch regulatienetwerk te modelleren. Een graaf $G$ wordt gedefinieerd door $(V, E)$ waarbij $V$ een set van punten (vertices) zijn en $E$ een set lijnen (edges). Een lijn wordt gedefinieerd door $(i, j)$ waarbij $i$ het beginpunt en $j$ het eindpunt is. De punten van een graaf representeren het gen of een andere element van een regulatie systeem waar men in is geïntereseerd. De lijnen weerspiegelen de interacties tussen de genen.

De huidige databases en knowledge bases die informatie over regulatie interacties in genetische regulatie systemen bevatten kunnen worden gezien als een geannoteerde graaf.

Enkele operaties op grafen zijn mogelijk om biologische relevante voorspellingen over een regulatie systeem te kunnen doen. Bijvoorbeeld, een zoekopdracht van een pad tussen twee genen kan een missende regulatie interactie aantonen of kan aanwijzing geven over redundantie in het netwerk.

Er zijn verschillende grafen, gericht en niet gerichte grafen. Er wordt ook onderscheidt gemaakt tussen cyclische en acyclische grafen. Bayesian netwerk is een voorbeeld van een gerichte acyclische graaf.

## 2.2 Differentiaal vergelijkingen

De differentiaal vergelijkingen modelleert de concentraties van RNA's, eiwitten en andere elementen van het systeem door tijdsafhankelijke variabelen.

Gen regulatie wordt gemodelleerd door een reactie snelheidsvergelijking die de snelheid van de eiwit synthese nabootst [11]. De vergelijking is afhankelijk van de concentratie van andere elementen in het systeem.

Krachtige wiskundige programma's voor het modelleren van biologische reacties systemen door reacties snelheid vergelijkingen zijn in het verleden speciaal voor metabole processen ontwikkeld. Door deze programma's is het mogelijk om kinetische modellen van genetische regulatie processen te construeren.

De reactiesnelheidsvergelijkingen geven per tijdsstap een balans tussen het aantal moleculen die ontstaan en verdwijnen weer. Numerieke simulaties van enkele zeer goed bestudeerde regulatie systemen zijn uitgevoerd. Een probleem van het gebruik van numerieke technieken is het grote aantal onbekende parameters. Het bouwen van een complete kinetische modellen van genetische regulatie systemen vereisen een gedetailleerde kennis van reactie mechanismen.

Piecewise lineaire differentiaal vergelijkingen, Stochastische vergelijkingen, Partiële differentiaal vergelijkingen en andere ruimtelijke distributieve modellen vertonen een grote gelijkenis met de differentiaal vergelijking [8].

## 2.3 Booleaanse netwerken

Booleaanse netwerken zijn geïntroduceerd door Kauffman [10] als een eenvoudige model voor de complexe interacties in een regulatie systeem van levende cellen. Booleaanse netwerken zorgen voor inzicht in het gedrag, evolutie en zelf organiserend vermogen van grote genetische netwerken. Booleaanse netwerken geven de mogelijkheid om reverse engineering toe te passen op de data van tijdelijke patronen van gen expressies om de logische structuur van de onderliggende fysieke gen regulatie systemen te extraheren.

Booleaanse netwerken zijn gerichte grafen, waarin de $N$ vertices ieder een gen representeren, en de $K * N$ edges de interacties. De toestanden van de genen zijn Booleaanse variabelen (aan/uit). In Kauffman's model wordt de netwerk-topologie willekeurig geinitialiseerd, met de beperking dat iedere vertex slechts $K$ ingangen heeft.

De motivatie om genen door een computer te simuleren komt vanwege dat er een overeenkomst is [7, 1, 2, 12]: Een genoom bestaat uit een complex geschakeld netwerk waar niet veel over bekent is en een computer is een complex geschakeld systeem dat helemaal doorgrond is. Een voordeel om een computer te gebruiken is dat er gebruik gemaakt kan worden van de meest simpele talstelsels (binaire) om een zeer complexe gedrag te kunnen nabootsen.

Kauffman [10] heeft de booleaanse netwerken analytisch opgelost en met behulp van een computer. De netwerken die hij construeerde werden doorgerekend met 50 verschillende netwerken. Het aantal ingangen $K$ werd voor verschillende netwerken geïnitialiseerd op 1, 2, 3 en $N$. Het aantal genen $N$ werd voor verschillende netwerken geïnitialiseerd op 15, 50, 64, 100, 191, 400, 1024, 4096 en 8191. Alleen 15 tot en met 400 werden door een computer opgelost. Daarbij werden er voor enkele netwerken enkele regels verboden om te gebruiken. Bijvoorbeeld regels met alleen maar 0'en en/of 1'en. Dit werd gedaan om de verwachte cyclus lengte te verhogen.

Door de gegevens die in de biologie beschreven zijn naast de booleaanse netwerken te leggen heeft Kauffman [10] laten zien dat een willekeurige geïnitialiseerd booleaanse netwerk

met $K = 2$ het zelfde gedrag vertoont als die van een levende cel. Het gedrag van een willekeurige geïnitialiseerd booleaanse netwerk met $K = 2$ is een korte en stabiele cyclus.
Een conclusie die Kauffman trekt, nadat alle verschillende cyclus lengtes zijn uitgerekent, is dat het aantal verschillende cycli evenredig is met $\sqrt{N}$ met $N$ de netwerk-grootte. Verder laat Kauffman zien door middel van gegevens uit de biologie dat de tijd die nodig is voor cel replicatie geschat kan worden met behulp van booleaanse netwerken.

Booleaanse netwerken maken het mogelijk om grote regulatie netwerken te analyseren op een efficiënte manier, door het maken van vereenvoudigingen van de structuur en dynamica van het regulatiesysteem. In een booleaanse netwerk wordt het gen alleen maar actief (aan, 1) of inactief (uit, 0) gemaakt. Een waarde er tussen in is niet mogelijk. Ook wordt er vanuit gegaan dat de transitie synchroon verloopt. Dit is in de natuur zeker niet altijd het geval. Toch worden ze vaak gebruikt [8, 11] vanwege dat het een simpel model is en makkelijk doorrekent.
De toestand van de gen wordt beschreven door een booleaanse variabele waarbij hij actief (aan, 1) en inactief (uit, 0) kan zijn. Interactie tussen de elementen worden gerepresenteerd door booleaanse functies die de toestand van de gen, door de activatie van ander genen, kunnen berekenen.

| | state | net | | rule |
|---|---|---|---|---|
| 1 | 1 | 7 | 6 | 12 (1100) |
| 2 | 0 | 5 | 6 | 12 (1100) |
| 3 | 1 | 3 | 8 | 7 (0111) |
| 4 | 1 | 7 | 4 | 1 (0001) |
| 5 | 0 | 3 | 5 | 15 (1111) |
| 6 | 0 | 6 | 3 | 0 (0000) |
| 7 | 1 | 8 | 6 | 4 (0100) |
| 8 | 0 | 1 | 2 | 14 (1110) |

De variabele `state` bevat een vector met de lengte $N$ die de toestand van een regulatie systeem van $N$ elementen in een booleaanse netwerk representeert. Elk elementen in de vector kan de waarde 1 of 0 aannemen zodat de `state` $2^N$ verschillende waarden kan aannemen. De toestand van `state` op tijdstip $t + 1$ wordt berekenen door de `rule` van $K$ elementen uit `state` van het vorige tijdstip $t$. Dus, gennummer 1 heeft toestand 1 op tijdstip $t$. Op tijdstip $t + 1$ word dit 1. De berekening gaat als volg. De toestand van de gennummers op tijdstip $t$ wordt bekeken. De ingangen zijn gennummers 7 en 6. De ingang worden beschouwd als een binair getal, zodat we met dit getal één bit kunnen nemen uit de regel. Het gennummer 7 heeft toestand 1 en gennummers 6 heeft toestand 0 dus het binaire getal is 10 en dat is 2 in het decimale stelsel. Er wordt begonnen te tellen bij 0 en van rechts naar links. De rule van gennummer 1 is (binaire) 1100 en het $2^{de}$ element is dus 1. Dit is de toestand op tijdstip $t + 1$.
De variabelen die berekent worden op tijdstip $t$ in `state` noemen we de input waarden en de variabelen op tijdstip $t + 1$ in `state` noemen we de output waarden. Voor $K$ inputs is het totaal aantal mogelijke regels $2^{2^K}$. Dit betekent dat voor $K = 2$, 16 mogelijke booleaanse functies zijn.

Een van de berekeningen die gedaan kunnen worden is het zoeken naar verschillende cycli en hun lengte. Zodra er een nieuwe toestand is berekend kan er worden nagegaan of deze

het zelfde is als die van een eerdere toestand. Als dit zo is dan is er een cyclus gevonden. Omdat er een eindig aantal toestanden zijn is het theoretisch mogelijk dat er eerst alle toestanden doorlopen moeten voordat er weer bij het begin zijn aangekomen, maar in de praktijk komt dit niet voor.
De lengte van een cyclus word bepaald door de aantal toestanden die tussen de begin en eind toestand in zitten. Er kunnen zich meerdere cycli in een netwerk bevinden. Door bij elke gevonden cyclus de eindtoestand te onthouden is het mogelijk om opzoek te gaan naar verschillende cycli. Dit wordt gedaan door bij het zoeken naar een cyclus alle gevonden eindtoestanden te vergelijken met de huidige toestand. Als een van de toestanden overeenkomt dan betekent dit dat deze cyclus al gevonden is.

Transities tussen de `state`'s in een netwerk zijn deterministisch en synchroon. Deterministisch in de zin van één enkel output voor een gegeven input en synchroon omdat alle outputs worden berekend voordat overgegaan wordt naar een andere tijdstip.

# Hoofdstuk 3

# Bit Matrix Multiply

De vector supercomputer Cray SV1e kan 64 Gflops berekening doen. Het heeft 32 Gbytes shared memory en hij heeft 32 CPU's. Elke CPU draait op 500 MHz en heeft meerdere dual pipe lines voor verschillende operatoren. Tevens heeft hij 8 vector registers die elk 64 elementen kunnen bevatten. De Cray SV1e bevat bit matrix multiply (BMM) hardware, die wordt gebruikt om de booleaanse netwerken te berekenen.

## 3.1 Wat is de Bit Matrix Multiply ?

De BMM is een functionele unit die de mogelijkheid biedt om een vector register van 64 x 64 bit matrix in te laden. De geladen vector kan dan met een scalair waarde, die zich in een register bevindt, worden vermenigvuldigd. Dit resulteert in een nieuwe vector.

De vermenigvuldiging is niet een echte vermenigvuldiging maar een AND operatie van de bit in de rij van vector A met de bit in de rij van matrix B, gevolgd door een EXCLUSIVE OR operatie op alle bits van het antwoord. Voorbeeld:

```
A = (0 1 1) B = (0 0 1)
                (1 1 0)
                (1 1 1)

* = AND, + = EXCLUSIVE OR

(0 1 1) (0 0 1)  = 0*0 + 1*0 + 1*1 = 0 + 0 + 1 = 1
        (1 1 0)  = 0*1 + 1*1 + 1*0 = 0 + 1 + 0 = 1
        (1 1 1)  = 0*1 + 1*1 + 1*1 = 0 + 1 + 1 = 0

resultaat = (1 1 0)
```

De waarde A kan ook een matrix zijn. Bijvoorbeeld:

```
A = (0 1 1) B = (0 0 1)
    (1 0 0)     (1 1 0)
    (0 1 0)     (1 1 1)
```

```
berekening       = resultaat matrix C
(0 1 1) (0 0 1) = (1 1 0)
(1 0 0) (1 1 0) = (0 1 1)
(0 1 0) (1 1 1) = (0 1 1)
```

De matrices die in bovenstaande voorbeelden gebruikt worden zijn allemaal 3 bit groot. Alle waarden die gebruikt worden door de BMM moeten echter 64 bit groot zijn.

Er zijn 4 verschillende instructies om de BMM hardware aan te roepen. De eerste laadt de inhoud van een vector register in de BMM unit. Eigenlijk laadt deze de getransponeerde matrix in de BMM unit. Vervolgens is er een vermenigvuldigings operator om de geladen matrix te vermenigvuldigen met de inhoud van een register. Er is ook een vermenigvuldigings operator die in staat is een matrix in te laden en daarna direct te vermenigvuldigen.
Daarnaast is er een instructie die de matrix ontlaadt. Deze komt er dan getransponeerd uit. Dit kan erg nuttig zijn als er een getransponeerde matrix nodig is.
De laatste instructie zet een vlag zodat de BMM unit niet meer wordt gebruikt. Er is gekozen voor een vlag omdat het tijd kost om de BMM unit leeg te maken.

Als een matrix eenmaal in het programma geladen is kan deze zo vaak gebruikt worden als nodig is. Alle meervouden van 64 kunnen gebruikt worden. Zo is het ook mogelijk om een 64 bit lange matrix te laden en te vermenigvuldigen met een 128 bits lange matrix. Het resultaat is dan een 128 bits lange matrix.

De BMM unit zorgt voor een snelle manipulatie van een veld van bits. De BMM unit is ook interessant als er operaties gedaan moeten worden op een groep van bits.

## 3.2 BMM mogelijkheden

Hier worden enkele mogelijkheden van de BMM unit weergegeven. Kijk op [13] voor meer voorbeelden.

De basis van het gebruik van de BMM unit zijn, kolom verwisselen, kolom/rij verwisselen en rij pariteit.
Bij elke berekening bevat de ene matrix de gegevens en de andere de bit map. De bit map is 64 bij 64 bits groot, niet kleiner en niet groter. Het heeft als doel om de bits, kolommen of rijen van de gegevens te selecteren die gemanipuleerd moeten worden.

**Kolom verwisselen**
De positie van de kolommen van de matrix A (gegevens) in matrix C (resultaat) wordt bepaald door matrix B (map). Kolom 1 van matrix A komt in kolom 2 van matrix C door de vermenigvuldiging van rij 2 uit matrix B.

```
gegevens    map     resultaat
  A          B          C

101010    010000    010101
101010    100000    010101
```

```
101010    000100    010101
101010 X  001000 =  010101
101010    000001    010101
101010    000010    010101
```

**Kolom/rij verwisselen**
De positie van de kolommen van de matrix B (gegevens) in matrix C (resultaat) wordt bepaald door matrix A (map). Kolom 1 van matrix B komt in rij 1 van matrix C door de vermenigvuldiging van de rij 1 uit matrix A, etc. In dit voorbeeld wordt eigenlijk de matrix B getransponeerd.

```
 map      gegevens resultaat
  A          B         C

100000    111111    100000
010000    011111    110000
001000    001111    111000
000100 X  000111 =  111100
000010    000011    111110
000001    000001    111111
```

**Rij pariteit**
Elke rij van matrix A wordt vermenigvuldig met alle elementen van matrix B. Het antwoord van deze vermenigvuldiging, de pariteit, wordt geplaatst in matrix C.

```
 map      gegevens resultaat
  A          B         C

111111    000001    101010
111110    000011    010101
111100    000111    001010
111000 X  001111 =  000101
110000    011111    000010
100000    111111    000001
```

Alle voorbeelden hierboven zorgen voor een horizontale verwerking/verplaatsing van de gegevens. Het kan ook verticaal toegepast worden.

## 3.3 BMM programma instructies

Er zijn vier fortran instructies die communicatie met de BMM unit tot stand brengen in een fortran programma.

1a. **M@LD**
Laadt een matrix in de BMM unit. Hierdoor wordt de matrix getransponeerd.

1b. **M@MX**
Vermenigvuldigt de gegeven waarde met de geladen matrix.

2. **M@LDMX**
Doet eigenlijk de operatie 1a en 1b achter elkaar aan.

3. **M@UL**
Ontlaadt de BMM unit. De gevulde matrix wordt getransponeerd terug gegeven.

4. **M@CLR**
Zet en vlag, bit matrix is leeg, zodat de bit matrix opnieuw gebruikt kan worden.

In een fortran programma hebben de BMM instructie een bepaalde volgorde en eisen. Die worden hieronder nog eens nader uitgewerkt.

**INTEGER IA(*), IB(*)**
IA, IB zijn integer vectors, het sterretje geeft de grootte aan.

**INTEGER IC(*)**
IC is een integer vector waar het resultaat in komt te staan.

**INTEGER M@B(*)**
Elke naam die begint met M@B is een integer matrix multiply memory vector. Er wordt voor deze variabele geen geheugen gereserveerd aangezien deze waarde alleen gebruikt wordt om aan de functie syntax te voldoen.

In een fortran 90 programma moeten de integer waarden die gebruikt gaan worden in de BMM unit 64 bit zijn, dus INTEGER (KIND=8).

**M@B()=M@LD(IB)**
De matrix IB wordt getransponeerd geladen. Door de M@B er voor te zetten wordt de vector niet opgeslagen.

**IC()=M@MX(IA)**
De matrix IA wordt vermenigvuldigt met de geladen matrix. Het resultaat komt in de matrix IC te staan. Als IA 128 bit lang is dan moet IC dit ook zijn en kan er volstaan worden met de gehele 128 bits lange matrix in een keer aan te roepen. De berekening houdt hier rekening mee.

**IC()=M@UL()**
In IC komt nu de getransponeerde matrix IB te staan. Dit kan erg nuttig zijn als je een getransponeerde matrix nodig hebt.

**IC()=M@CLR()**
Zorgt ervoor dat de geladen waarden in de BMM unit verwijdert of door een ander programma gebruikt mogen worden. Als een ander programma gescheduld wordt dan wordt de BMM unit gegevens niet bewaart.

Hieronder staat een klein voorbeeld om te laten zien hoe een fortran programma eruit ziet met BMM code.

```
PROGRAM MULTIPLY
  INTEGER(KIND=8) IA(1:64), IB(1:64), IC(1:64), M@B(1:64)
  IA = ISHIFT( -1_8, (/ ( I, I = 0,-63,-1 ) /) )
  IB = ISHIFT( -1_8, (/ ( I, I = 0,-63,-1 ) /) )

  PRINT *, ' Initial data:'
  PRINT '( B64 )', IA
  PRINT '( B64 )', IB

  M@B(1:64) = M@LD( IA (1:64)) ! Laadt de bit matrix IA getransponeert
  IC(1:64) = M@MX(IB(1:64)) ! vermenigvuldigt IB met getransponeerde IA
  M@B(1:64) = M@UL( ) ! ontlaadt de unit

  PRINT *, ' After multiplication:'
  PRINT '( B64 )', IC
END
```

Dit programma doet de volgende berekening.

```
  A         B         C

111111    100000    010101
011111    110000    110101
001111    111000    000101
000111 X  111100 =  111101
000011    111110    000001
000001    111111    111111
```

# Hoofdstuk 4

# Fortran code

Hier wordt de volledige Fortran code uitgewerkt. Daarna wordt de code gevectoriseerd en wordt de snelheidswinst bekeken.

## 4.1 Pseudocode

De volgende tabel geeft een volledige booleaanse netwerk weer. De eerste kolom met de getallen van 1 tot en met 8 is het gennummer. De toestand (0 of 1) van een gennummer staat in de tweede kolom, genaamd `state`. De derde en vierde kolom is het netwerk (`net`) en bevat de gennummers (1..8) die als ingangs waarde gebruikt worden. De regel voor een gennummer staat in de laatste kolom genaamd `rule`. (Voor $K = 2$ zijn dat er dus $2^{2^K} = 16$ verschillende waarden van 0 tot en met 15) De waarde voor $N$ is voor dit booleaanse netwerk 8. Er zijn 8 verschillende gennummer. $K = 2$ omdat het netwerk (`net`) twee ingangen (twee kolomen) heeft.

| | state | net | | rule |
|---|---|---|---|---|
| 1 | 1 | 7 | 6 | 12 (1100) |
| 2 | 0 | 5 | 6 | 12 (1100) |
| 3 | 1 | 3 | 8 | 7 (0111) |
| 4 | 1 | 7 | 4 | 1 (0001) |
| 5 | 0 | 3 | 5 | 15 (1111) |
| 6 | 0 | 6 | 3 | 0 (0000) |
| 7 | 1 | 8 | 6 | 4 (0100) |
| 8 | 0 | 1 | 2 | 14 (1110) |

Als alle kolomen in de bovenstaande tabel gevult zijn, dit is tijdstip $t$, dan is het mogelijk om de toestanden voor de volgende tijdstip $t + 1$ te berekenen. Dit word als volgt gedaan. Eerst worden de ingangs toestanden uitgelezen van tijdstip $t$. Dus voor gennummer 1 is dit de toestand van gennummer 7 en 6. De toestand van gennummer 7 op tijdstip $t$ is 1 en die van 6 is 0. In de volgorde van het uitlezen word er een binaire representatie van de ingangs waarde gemaakt. Dus 10 binaire word decimaal 2. Door de regel `rule` te zien als een rij biniare waarde van 0'en en 1'en is het mogelijk om er één waarde uit te halen. Namelijk die waarde die door de ingangen geconstrueerd word. Dat is in dit geval de 2 waarde. Omdat de ingangen ook beide 0 kunnen zijn moet er dus vanaf 0 getelt worden. Het is ook nog eens makkelijker programmeren als er van rechts naar links getelt wordt. De $2^{de}$ waarde is dus 1.

De toestand verandert dus niet op tijdstip $t + 1$.

Een cycle word gedetecteerd door de begin toestand te onthouden en na de berekening van de volgende tijdstip te vontroleren of de twee toestanden gelijk zijn. Zo ja dan is er een cycle gevonden. Door de aantal verschillende tijdstippen bij te houden is de lengte van de cycle ook gelijk bekent.

In pseudocode ziet dit alles er als volgt uit:

```
K=2
N=8
vul alle begin toestanden in state
vul alle ingangen van het netwerk in net
vul alle regels van elk gennummer in rule
kopieer/onthoud de begin toestand
lengte cycle is 0
DO WHILE (toestanden niet gelijk aan onthoude toestand)
  bereken de volgende toestand
  verhoog de variabele lengte cycle met 1
END DO
geef de lengte van de cycle weer op het scherm
```

## 4.2 De code

De gehele code is in bijlage A-1 weergegeven. Als eerste zijn de constanten $K$ en $N$ gedefinieerd. De constanten kunnen elke willekeurige waarde aannemen als die maar groter of gelijk is aan 1.

```
INTEGER, PARAMETER :: K=2
INTEGER, PARAMETER :: N=32
```

Om de toestanden bij te houden is er de variabele `state` en een kopie er van genaamd `state2`. De regels die bij deze toestanden horen worden opgeslagen in `rule` en de verbindingen van het netwerk zijn in de variabele `net` opgeslagen. In programma code ziet dit er dan als volgt uit.

```
INTEGER :: net(N,K), state(N), state2(N), rule(N)
```

Om alles willekeurig te initialiseren, zijn er nog enkele variabelen nodig. Hier ga ik verder niet op in. Het is ook mogelijk om een andere random generator te gebruiken. Verder zijn er variabelen nodig in de DO-lussen en indexeringen. Aangezien ik deze als vanzelfsprekend acht worden deze hier verder niet besproken.

De volgende stap geeft aan hoe alle verbindingen van het netwerk tot stand komen. Dit gebeurd door alle toestanden af te gaan en deze te verbinden met een willekeurig ander toestand. Hierbij wordt er opgelet dat er geen twee dezelfde ingangen gebruikt worden van dezelfde gen.

Vervolgens worden de regels van het netwerk geïnitialiseerd. Elk toestand wordt doorlopen en willekeurig wordt er een regel voor bepaald. De regels worden altijd $2^K$ lang gemaakt.

Zo worden altijd alle regels die mogelijk zijn aangemaakt, voor $K = 2$ is dat van 0000 tot 1111.

De toestanden worden daarna willekeurig geïnitialiseerd. Met een 1 (aan) of een 0 (uit). Er wordt een kopie gemaakt van `state` in `state2` voor de *figure of 6* berekening.

Om de cycle lengte uit te rekenen wordt er gebruik gemaakt van het algoritme *figure of 6*. Dit is in onderstaande code terug te vinden. De eerste lus zorgt ervoor dat de `state` één keer berekend wordt en `state2` twee keer. Als de lus eindigt dan heeft `state` één keer en `state2` in dezelfde tijd twee keer een cyclus doorlopen. Nu is het zeker dat er een hele cyclus wordt doorlopen en niet vroegtijdig gestopt wordt omdat er toevallig eenzelfde eindtoestand is ontstaan. Dus door een nieuwe lus te starten en alleen `state` te berekenen en daarna `state` te vergelijken met `state2` wordt de cyclus lengte bepaald.

```
! Figure of 6 cycle detection
b = .FALSE.
DO WHILE (.NOT. b)
  CALL stepState(state, net, rule)
  CALL stepState(state2, net, rule)
  CALL stepState(state2, net, rule)
  CALL equaleState(state, state2, b)
END DO

! member of limit cycle found
i = 0
b = .FALSE.
DO WHILE (.NOT. b)
  CALL stepState(state, net, rule)
  i=i+1
  CALL equaleState(state, state2, b)
END DO
```

Als laatste wordt de cyclus lengte die bij dit netwerk en deze begin toestand hoort afgedrukt op het scherm.

Er zijn nog een aantal functies die worden aangeroepen die hier nog niet genoemd zijn. Dit is als eerste `equaleState`. Deze functie heeft als invoer twee toestandsvectoren en als uitvoer een boolean. De uitvoer geeft aan of de twee toestandsvectoren gelijk zijn. De vectoren worden element voor element gecontroleerd. Als deze allemaal gelijk zijn dan is de uitvoer **true** en in alle andere gevallen **false**.

De functie `kInputs` berekent het nieuwe toestand door een 1 te shiften over een aantal `input` posities. Vervolgens wordt de uitkomst geAND met de `rule`. Alleen als het resultaat ongelijk aan 0 is dan wordt het nieuwe toestand 1 anders 0.

De `stepState` heeft als doel alle nieuwe toestanden uit te rekenen. Daarvoor moeten eerst alle `input` waarden berekend worden. De `input` waarden voor $K = 2$ worden berekent door de eerste input uit het netwerk te vermenigvuldigen met 2. Vervolgens wordt de laatste `input`

er bij opgeteld. De waarde die zo ontstaat wordt doorgegeven aan `kInputs`, deze functie berekent de nieuwe waarde en plaatst deze in `state`.

Om te zien hoe snel deze code een cyclus kan vinden word er een kleine netwerk berekent. Als voorbeeld word er voor $K = 2$, $N = 1000$, één toestand en één netwerk genomen. De fortran code rekent dit booleaans netwerk in $20,0$ seconden uit.

## 4.3 Aanpassingen

Bovenstaande code heeft enkele aanpassing ondergaan [5]. Dit is allemaal te zien in de nieuwe programma code versie 3, in bijlage A-2.

Om niet altijd alle $K$ en $N$ constanten aan te hoeven passen is er een module gemaakt die deze constanten bevat. Alle constante variabelen zijn in deze module vervangen door `USE const`. Deze module is aan de programma code toegevoegd en ziet er als volgt uit.

```
MODULE const
  INTEGER, PARAMETER :: K=2
  INTEGER, PARAMETER :: N=8
END MODULE
```

Om de random generator eenvoudiger aan te roepen is er voor gekozen om alle variabelen die daarvoor gebruikt moeten worden ook in de module te plaatsen. Dus;

```
INTEGER, DIMENSION(56) :: ma
INTEGER :: inext, inextp, iff, idum
```

Hierdoor kan de functie aanroep van `CALL randomInt(iff, idum, ma, inext, inextp, out)` verandert worden naar `CALL randomInt(out)`. De initialisatie code kan dan in het hoofdprogramma weggelaten worden. Dit zijn de regels;

```
inext=1
inextp=32
```

Om de code nog enigszins overzichtelijk te houden zijn de initialisaties in aparte functies geplaatst. Alle initialisaties van het netwerk komen in de functies `initConnection`, initialisaties van de regels komen in `initRule` en de initialisaties van de toestanden komen in `initState`. Hierdoor komen de volgende variabelen te vervallen in het hoofdprogramma.

```
INTEGER :: j, out
INTEGER :: dummy, temp
INTEGER :: d
LOGICAL :: OK
```

Deze variabelen worden naar hun juiste functies verplaatst of gekopiëerd.

## 4.4 Vectorizeren

Omdat de Cray SV1e een aantal vector processoren bevat wordt de code zodanig herzien en herschrijven dat alle code gevectorizeerd wordt.
De code wordt nu met de optie -O3 gecompileerd zodat de code wordt geoptimaliseerd. Vectorizeren is een van de technieken die wordt toegepast.
De volgende statements laten zien wat er met vectoriseren bedoeld wordt.

```
DO i=1,56
  ma(i)=0
END DO
```

wordt

```
ma=0
```

en de lus

```
DO i=1,N
  state2(i) = state(i)
END DO
```

verandert in

```
state2 = state
```

Om te kunnen zien wat er geoptimaliseerd wordt door de compiler is er de optie `-r3`. Deze optie zorgt ervoor dat er een report gemaakt wordt die aangeeft wat de compiler allemaal vectorizeert, parallelizeerd en welke andere optimalisatie technieken worden toegepast. Dit geeft ons dus een goed overzicht van wat er nog gedaan kan worden om het programma nog sneller te maken. Dus door het `f90 -O3 -r3 versie3.f90 -o versie3` commando in te typen wordt er een bestand genaamd `versie3.lst` waar de volledige code in staat met de extra notities over optimalisatie technieken aangemaakt.

Het eerste wat opvalt in het report is dat de lus in `stepState`, waar de functie `kInputs` wordt aangeroepen, niet wordt gevectorizeerd. Dus de DO loop wordt ingebouwd in de procedure van kInputs, dus dit

```
DO i=1,N
  CALL kInputs(rule(i), input(i), state(i))
END DO
```

wordt dan alleen maar

```
CALL kInputs(rule, input, state)
```

Doordat de lus in de functie `kInputs` wordt ingebouwd wordt de DO lus wel gevectorizeerd. Aangezien de inhoud van de functie `kInputs` niet erg groot is word deze in zijn geheel naar de `stepState` functie verplaatst.

De code van deze `stepState` functie ziet er als volgt uit:

```
DO i=1,N
  input(i) = state(net(i,K))
  DO j=(K-1),1,-1
    input(i) = LSHIFT(input(i), 1) + state(net(i,j))
  END DO
END DO
DO i=1,N
  IF (IAND(rule(i), LSHIFT(1,input(i)))/=0) THEN
    state(i) = 1
  ELSE
    state(i) = 0
END DO
```

Het is mogelijk om van de twee DO lussen één te maken. Voor de executie tijd maakt het echter niet veel uit. Sterker nog, bij grote netwerken zal met één DO lus de executie tijd alleen maar langer worden. Dit komt doordat met één DO lus de `state` waarde wel gelezen maar niet ingeschreven kan worden, of andersom wel ingeschreven maar niet uitgelezen kan worden. Daardoor heb je een dummy waarde nodig die tijdelijke de `state` waarde opslaat en als de DO lus is afgelopen de waarden in de `state` vector zet. Dit kost dus extra tijd. De rest van de code is al zo goed als mogelijk geoptimaliseerd en wordt maar eenmalig aangeroepen. Daarom is het niet de moeite waard om deze verder te optimaliseren.

Stel $K = 2$ en $N = 60000$ dan blijkt het verschil in executie tijd erg groot te zijn (20.1136 seconden). De executietijds is voor `versie1` 26.1384 seconden en voor `versie3` is die 5.9248 seconden. De snelheid winst ontstaat voornamelijk door het verwijderen van de functie `kInputs` en de code in `stepState` te plaatsen.

Om te zien hoe snel deze code een cyclus kan vinden word het zelfde kleine netwerk berekent als bij de fortan code gedaan is. Als voorbeeld word er voor $K = 2$, $N = 1000$, één toestand en één netwerk genomen. De gevectoriseerde fortran code rekent dit booleaans netwerk in $0,013$ seconden uit.

# Hoofdstuk 5

# Efficientie verbeteren

In dit hoofdstuk wordt gewerkt met versie 3 van de fortran code. Deze versie zal geoptimaliseerd worden. Als eerste door middel van de BMM instructies [4, 13] en ten tweede door het reduceren van het netwerk.

## 5.1 BMM code

### 5.1.1 Functies in BMM code

De code in de functie `stepState` zal als eerste worden aangepast. Zoals al eerder is gezegd zijn de functies `stepState` en `equaleState` de aangewezen plaats om versnelling aan te brengen aangezien deze functies vaak worden aangeroepen.
Stap voor stap wordt diverse code aangepast. In de functie `stepState` worden de tweede DO lus iets aangepast. De code waar geshift moet worden, wordt uit de DO lus gehaalt.

```
DO i=1,N
  IF (IAND(rule(i), LSHIFT(1,input(i)))/=0) THEN
    state(i) = 1
  ELSE
    state(i) = 0
END DO
```

wordt dus

```
input = LSHIFT(1, input)
DO i=1,N
  IF (IAND(rule(i), input(i))/=0) THEN
    state(i) = 1
  ELSE
    state(i) = 0
END DO
```

De DO lus kan nu eenvoudig vervangen worden door BMM code, namelijk door een vector aan te maken die gevuld wordt met 0en en op de 64ste positie de `rule` te plaatsen. Deze vector wordt in de BMM unit geladen en vervolgens wordt de `input` er mee vermenigvuldigt zodanig dat het resultaat precies een 0 of een 1 is. Omdat het antwoord precies een 0 of een

1 is, kan deze gelijk opgeslagen worden in de nieuwe state variabele. In code ziet het er als volgt uit.

```
INTEGER :: rule, input, state
INTEGER :: i
INTEGER(KIND=8) IA(1:64), IB(1), IC(1), M@B(1:64)

DO i=1,N
  IA = 0 ! maak de matrix leeg door het met nullen te vullen
  IA(64) = rule(i) ! vul de regel op de laatste plaats in de matrix
  IB(1) = input(i) ! input waarde kopieren in IB
  M@B(1:64) = M@LD( IA (1:64)) ! laat de matrix IA in de BMM unit
  IC(1) = M@MX(IB(1)) ! vermenigvuldig IA met IB
  M@B(1:64) = M@UL( ) ! zorg ervoor dat de BMM ontladen is
  state(i) = IC(1) ! kopieer de nieuwe state op de juiste plaats
END DO
```

Ter verduidelijking van wat er gebeurd staat hieronder een voorbeeld van een kleine matrix berekening.

```
input     rule      state
  IB        IA        IC
001000    000000    000001
          000000
          000000
       X  000000 =
          000000
          001010
```

De input 1000 wordt vermenigvuldigd met de rule 1010 en heeft als resultaat state 1.
De rule moet op de laatste rij in de $IA$ matrix komen zodat het laatste bit in vector $IC$ het antwoord bevat. Het antwoord kan dan gelijk in de state variabele geplaatst worden zonder dat het eerste moet worden vertaalt. Het IF statement hoeft niet meer te worden doorlopen.

Omdat de $IB$ vector niet een meervoud is van 64 kan $M@LDMX$ niet worden gebruikt, zal moeten worden volstaan met de twee losse functie aanroepen $M@LD$ en $M@MX$.

In het voorbeeld wordt er één input waarde vermenigvuldigt per ingeladen rule. Eigenlijk zou het efficiënter zijn om alle input waarden te vermenigvuldigen per ingeladen rule. Hoe dit gedaan wordt, wordt later besproken. De verplaatste code zal eerst worden veranderd.

De verplaatste code `input = LSHIFT(1, input)` wordt vervangen door `input = v_power(input)`.
Wat de verplaatste code eigenlijk doet is een 1 een aantal input posities te verplaatsten naar links.
Bijvoorbeeld; als $K = 2$ dan heeft input de volgende waarden 00, 01, 10 of 11. Dat betekent dat de waarde 1 dus 0, 1, 2 of 3 posities naar links opschuift. Dit resulteert in de waarden 0001 (=1), 0010 (=2), 0100 (=4) of 1000 (=8). De vector v_power wordt gevuld met de waarden 1, 2, 4 en 8 en heeft dus een bereik van $(0:3)$.
De volgende code moet toegevoegd worden in de module const.

```
INTEGER :: v_power(0:63)
```

De initialisatie gebeurt in de functie `initbmm` die hieronder is beschreven.

```
SUBROUTINE initbmm()
  USE const
  INTEGER :: i

  v_power = 0
  DO i=0,((2**K)-1)
    v_power(i) =  2**i
  END DO
  RETURN
END
```

Nu wordt de eerste lus in vervangen door BMM code. In onderstaande code is te zien dat de waarde `input` wordt geïnitialiseerd met de eerste rij `state` waarden, daarna wordt, als er nog meer `state` waarden zijn, deze één positie naar links verschoven. Waarna de volgende `state` waarden er weer bij opgeteld worden net zolang tot alle `state` waarden doorlopen zijn.

```
DO i=1,N
  input(i) = state(net(i,K))
  DO j=(K-1),1,-1
    input(i) = LSHIFT(input(i), 1) + state(net(i,j))
  END DO
END DO
```

Om deze code om te zetten in BMM code wordt er een matrix (bit-map) gemaakt die de verschuivingen voor zijn rekening neemt, waarna alle elementen van `state` vermenigvuldigt worden met deze matrix. Het optellen van de `state` waarden blijft ongewijzigd.
De verschuivingsmatrix komt in de variabele `INTEGER(KIND=8) :: m_shift(1:64)`. Deze wordt in de module `const` geplaatst en wordt geïnitaliseerd in de functie `initbmm`. Dit gaat als volgt;

```
SUBROUTINE initbmm()
  USE const
  INTEGER :: i, bit
  LOGICAL :: b
  bit = 1
  m_shift = 0
  DO i=1,63
    m_shift(64-i) = bit
    bit = bit*2
  END DO
  v_power = 0
  DO i=0,((2**K)-1)
    v_power(i) =  2**i
  END DO
  RETURN
END
```

Als eerste wordt de matrix geladen. Dit gebeurt maar één keer, daarna kan deze steeds weer opnieuw worden gebruikt. Omdat de berekening eenvoudiger is te maken als de lengte van de te vermenigvuldigen vector een 64-voud is, wordt deze omgezet naar het kleinste een 64-voud groter dan $N$. Dat is de constante $O$. De dummy variabele dient dan als de 64-voud vector. Deze wordt gevuld met alle `state` waarden en de overige posities worden met 0 gevuld. Alle waarden in de dummy variabele worden nu één positie naar links verschoven. Vervolgens wordt elke volgende `state` waarde er bij opgeteld. Dit herhalen we zolang er nog `state` waarden zijn.

```
INTEGER, PARAMETER :: O=(((N/64)+1)*64)
INTEGER(KIND=8) :: dummy(1:O)
INTEGER(KIND=8) :: M@B(1:64)
INTEGER :: ki

M@B(1:64) = M@LD( m_shift(1:64) )
ki = K
dummy=0
DO vi=1,N
  dummy(vi) = state(net(vi,ki))
END DO
DO WHILE ( 1 < ki )
  dummy(1:O) = M@MX( dummy(1:O) )
  ki = ki - 1
  DO vi=1,N
    dummy(vi) = dummy(vi) + state(net(vi,ki))
  END DO
END DO
M@B(1:64) = M@UL()

input(1:N) = v_power(dummy(1:N))
```

Zoals u kunt zien in de laatste regel van bovenstaande code is de aanroep met de parameter dummy van de functie `v_power` verandert. Zo worden alle waarden weer in de juiste vector geplaatst. Door de functie `v_power` aan te roepen wordt de `input` waarde met de juiste macht waarde gevuld. Door de te vermenigvuldigen vector 64-voud lang te maken worden meer waarden dan nodig is berekend. En zullen in het slechtste geval 63 waarden 0 zijn. Anders moet een conditie ingebouwd worden om te testen of het einde van de te vermenigvuldigen vector al is bereikt. Beide mogelijkheden kosten evenveel tijd. De optimalisatie van de Cray compilers is zo goed dat er geen tijdsverschil tussen beide manier ontstaat, maar bovenstaande code is wel een stuk beter leesbaar zonder de DO lussen met IF condities erin.

Als laatste zal de `equaleState` functie omgezet worden in BMM code. Dit gebeurd door 64 waarden van een vector in te laden en deze vervolgens weer uit te laden zodat we een getransponeerde vector overhouden. Dit betekent dat er van 64 aparte waarden nu één integer waarde is ontstaan. Dit gebeurd voor beide vectoren. Beide integer waarden worden met elkaar vergeleken om aan het eind te kunnen constateren of deze gelijk zijn. Onderstaande code geeft dit weer.

```
SUBROUTINE bmmequaleState(source1, source2, out)
  USE const
  INTEGER :: source1(N), source2(N)
  LOGICAL :: out
  INTEGER :: index, i
  INTEGER(KIND=8) :: IC(1:64), ID(1:64), IE(1:64), IG(1:64)
  INTEGER(KIND=8) :: M@B(1:64)

  index = 0
  out = .TRUE.
  DO WHILE (index < N)
    DO i=1,64
      IF (i+index <= N) THEN
        IE(i) = source1(i+index)
      ELSE
        IE(i) = 0
      END IF
    END DO
    M@B(1:64) = M@LD(IE(1:64))
    IC(1:64) = M@UL()
    DO i=1,64
      IF (i+index <= N) THEN
        IG(i) = source2(i+index)
      ELSE
        IG(i) = 0
      END IF
    END DO
    M@B(1:64) = M@LD(IG(1:64))
    ID(1:64) = M@UL()
    IF (IC(64) /= ID(64)) THEN
      index = N
      out = .FALSE.
    END IF
    index = index + 64
  END DO
  RETURN
END
```

Net zoals bij de stepState maakt het voor de snelheid niet uit of je nu een 64-voud vector neemt of niet.

Zoals eerder gezegd wordt bij de stepState functie maar één berekening gedaan in plaats van 64. Dit wordt in de volgende paragraaf besproken.

### 5.1.2 Meerdere regels in één keer bereken

Als $K = 2$ dan zijn er $2^{2^K} = 16$ verschillende regels. Met $N = 1000$ komen er nog steeds maar 16 verschillende regels voor. Dat houd dus in dat er ongeveer $1000/16 = 62,5$ keer dezelfde regel gebruikt worden. Het idee is, door de 16 verschillende regels in te laden en niet meer alle 1000 stuks. Door na het inladen van een regel alle gennummers die die regel hebben er mee te bereken kunnen er nu meer regels tegelijk berekent worden in plaats van één per keer. Dit zou dus snelheidswinst op kunnen leveren. Het enige nadeel is dat er dus nu eerst alle gennummers die dezelfde regel hebben bijelkaar gezocht moeten worden voordat er maar een berekening gedaan kan worden.
Dit kan het beste bij de initialisatie van de `rule` worden uitgezocht. Daarvoor is de variabele `matrix` nodig. Deze wordt in de module `const` geplaatst.

```
INTEGER :: matrix_index
INTEGER :: matrix((2**(2**K)),N)
```

| | rule | ind. | gennummers | | | | |
|---|---|---|---|---|---|---|---|
| 12 (1100) | 53 | 1 | 3 | 5 | 6 | 8 | etc |
| 4 (0100) | 65 | 2 | 4 | 14 | 18 | 34 | etc |
| 1 (0001) | 61 | 9 | 15 | 21 | 33 | 53 | etc |

De variabele `matrix` bevat op de eerst plaats de `rule` en op de tweede plaats de index waarde. De index waarde geeft aan waar, in die rij, de eerste plaats is om een `rule` te plaatsen. De rij bevat, vanaf de derde plaats tot aan de waarde die op de tweede plaats staat aangegeven, de `state` index waarden die allemaal deze `rule` hebben. Om bij te houden hoeveel verschillende `rule`'s zijn ingevuld wordt er een variabele genaamd `matrix_index` in de module `const` geplaatst.
De functie `initRules` verandert zodanig dat nadat er een `rule` geplaatst is voor een `state`, deze wordt bijgehouden in de `matrix` variabele. Alles wordt zo gesorteerd dat alle `state` met dezelfde `rule` bij elkaar worden gezet in de array `matrix`.

```
SUBROUTINE bmminitRules(source)
  USE const
  INTEGER :: source(N)
  INTEGER :: i, temp, j, out, dummy, l
  INTEGER :: w, rule_index, m
  LOGICAL :: inMatrix

  matrix_index = 1
  i = 1
  do while (i < N)
    temp = 0
    DO j=1,(2**K)
      CALL randomInt(out)
      dummy = MOD(out, 2)
      temp = temp*2 + dummy
    END DO
    source(i) = temp
```

```
    j = 1
    inMatrix = .FALSE.
    rule_index = 0
    l = source(i)
    do while (j < matrix_index)
      m = matrix(j, 1)
      if (m==l) then
        inMatrix = .TRUE.
        rule_index = j
        j = matrix_index + 1
      else
        j = j + 1
      end if
    end do
    IF (.NOT. inMatrix) THEN
      matrix(matrix_index, 1) = source(i) ! regel
      matrix(matrix_index, 2) = 4 ! eerste nieuwe plaats
      matrix(matrix_index, 3) = i ! state index
      matrix_index = matrix_index + 1 ! regel aantal
    ELSE
      w = matrix(rule_index, 2) ! welke positie is vrij
      matrix(rule_index, w) = i ! state positie erin plaatsten
      matrix(rule_index, 2) = w + 1 ! vrije positie verhogen
    END IF
    i = i + 1
  end do
  RETURN
END
```

Om nu de berekening met een gevulde matrix uit te voeren in de functie stepState moet de matrix nog op de juiste manier gevuld worden. Dit gaat als volgt.
De matrix_index geeft het aantal verschillende rule's weer. Één voor één worden de verschillende rule's ingeladen. Bij een geladen rule worden alle state's met deze rule berekend.

De rule wordt op de 64'ste plaats in de matrix geplaatst en de rest van de elementen worden op 0 gezet. De state's met dezelfde rule staan in de matrix variabele. Een DO lus kan de input waarden in een tijdelijke variabele zetten totdat er 64 element ingevuld zijn. Als niet alle 64 elementen kunnen worden gevuld, worden de overige elementen op 0 gezet. Daarna wordt de nieuwe status van de state berekend. Onderstaande code geeft het gehele principe in BMM code weer.

```
  INTEGER :: m_index, r_index, r_max
  INTEGER(KIND=8) IA(1:64), IB(1:64), IC(1:64), M@B(1:64)

  m_index = 1
  DO WHILE (m_index < matrix_index)
    IB = 0
    IB(64) = matrix(m_index, 1)
```

```
    M@B(1:64) = M@LD(IB(1:64))
    r_index = 2
    r_max = matrix(m_index, 2)
    DO WHILE (r_index < r_max)
      DO i=1,64
        IF (i+r_index < r_max) THEN
          IA(i) = input( matrix(m_index, i+r_index) )
        ELSE
          IA(i) = 0
        END IF
      END DO
      IC(1:64) = M@MX(IA(1:64))
      DO i=1,64
        IF (i+r_index < r_max) THEN
  state( matrix(m_index, i+r_index) ) = IC(i)
        END IF
      END DO
      r_index = r_index + 64
    END DO
    M@B(1:64) = M@UL()
    m_index = m_index + 1
  END DO
```

Om te zien hoe snel deze code een cyclus kan vinden word het zelfde kleine netwerk berekent als bij de fortan code gedaan is. Als voorbeeld word er voor $K = 2$, $N = 1000$, één toestand en één netwerk genomen. De BMM code rekent dit booleaans netwerk in 0,026 seconden uit.

### 5.1.3 Beperkingen

De BMM code maakt gebruik van de gesorteerde `matrix` om tijdswinst te behalen. Bij een geladen `rule` worden alle `state`'s met deze `rule` berekend. Bij een grote $K$ en een bepaalde $N$ zijn er veel verschillende `rule`'s. Hierdoor zijn er maar weinig `state`'s die tegelijk berekent kunnen worden. Dit levert dus geen tijdswinst op. Bij een kleine $K$ en een bepaalde $N$ zijn er weinig verschillende `rule`'s. Hierdoor zijn er veel `state`'s die tegelijk berekent kunnen worden. Dit levert dus tijdswinst op. Bij het toenemen van $N$ neemt ook de efficiëntie toe. Er moet dus gelden $N >> 2^{2^K}$.

De totale executie tijd van het BMM programma is langer dan de gevectoriseerde versie. De BMM unit wordt gebruikt voor eenvoudige bit manipulaties, zoals shift en and. De Cray heeft hardware die al eenvoudige bit manipulaties op vectoren kan uitvoeren. Na de code te hebben gevectoriseerd, berekent de Cray deze bit manipulaties daardoor sneller dan het BMM programma.

Ook is het zo dat de BMM code iets afwijkt van de gevectoriseerde code. In de gevectoriseerde versie is in de functie `stepState` de volgende lus te vinden.

```
DO i=1,N
  input(i) = state(net(i,K))
```

```
    DO j=(K-1),1,-1
      input(i) = LSHIFT(input(i), 1) + state(net(i,j))
    END DO
  END DO
```

Deze lus is in de BMM versie als volgt herschreven.

```
dummy=0
DO vi=1,N
  dummy(vi) = state(net(vi,ki))
END DO
DO WHILE ( 1 < ki )
  dummy(1:0) = M@MX( dummy(1:0) )
  ki = ki - 1
  DO vi=1,N
    dummy(vi) = dummy(vi) + state(net(vi,ki))
  END DO
END DO
```

In de BMM code is de functie van de variabele dummy gelijk aan de variabele input in de gevectoriseerde code. In de BMM code zijn er twee DO lussen die van 1 tot $N$ lopen. De eerste DO lus plaatst alle elementen van state in dummy. De tweede DO lus zorgt er voor dat de state wordt opgeteld bij de geshifte dummy vector. Dit is niet nodig bij de gevectoriseerde versie. Daar kan worden volstaan met één DO lus van 1 tot $N$. Door de aanwezigheid van twee DO lussen in de BMM code twee keer zo langzaam dan de gevectoriseerde code.

Als men de BMM unit voor snelheidwinst wil gebruiken kan men dat alleen behalen als de berekening ingewikkelder is dan de standaard bit operators. Bijvoorbeeld de bits paarsgewijs verwisselen. Tevens moet, nadat een bepaalde bit-map is geladen, de operatie op een grote hoeveelheid gegevens uitgevoerd worden. Pas dan zal er een snelheidwinst behaald kunnen worden.

## 5.2 Reduceren van het netwerk

### 5.2.1 De reductie

We gaan uit van de gevectoriseerde code versie 3. Het principe van reductie wordt hier aan de hand van een voorbeeld uitgelegd. Stel $K = 2$ en $N = 8$ en onderstaande waarden voor net en rule worden geïnitialiseerd. De eerste kolom genummerd, 1 tot en met 10, geeft de index waarde van overige rijen weer. Bij de rule staan zowel de decimale als de binaire waarden genoteerd.

De state wordt niet geïnitialseerd. De niet geïnitialseerd state wordt aangegeven door middel van een X. Een X betekent don't care (het maakt niet uit of er een 0 of een 1 wordt ingevuld). Door het invullen van verschillende state variabelen is het netwerk meerdere malen te gebruiken. Voor dit onderzoek wordt alleen state twee elementen langer gemaakt. Het eerste element, $N + 1$, krijgt de waarde 0 en het tweede element, $N + 2$, krijgt de waarde 1. Dit ziet er dan als volgt uit.

| | state | net | | rule |
|---|---|---|---|---|
| 1 | X | 7 | 6 | 12 (1100) |
| 2 | X | 5 | 6 | 12 (1100) |
| 3 | X | 3 | 8 | 7 (0111) |
| 4 | X | 7 | 4 | 1 (0001) |
| 5 | X | 3 | 5 | 15 (1111) |
| 6 | X | 6 | 3 | 0 (0000) |
| 7 | X | 8 | 6 | 4 (0100) |
| 8 | X | 1 | 2 | 14 (1110) |
| 9 | 0 | | | |
| 10 | 1 | | | |

Nu wordt de lijst van rule's doorlopen en zodra daar uitsluitend 0'en staan wordt de index waarde, die bij deze rule hoort, in net veranderd in $N + 1$. Voor de rule's die uitsluitend 1'en bevat veranderd de bijbehorende index waarde in net in $N + 2$.
Voor index waarde 5 bevat rule uitsluitend 1'en. De verwijzing van 5 in net wordt verandert van 5 naar $N + 2$. Dat is dus $net(2, 1)$ en $net(5, 2)$.

| | state | net | | rule |
|---|---|---|---|---|
| 1 | X | 7 | 6 | 12 (1100) |
| 2 | X | 10(N+2) | 6 | 12 (1100) |
| 3 | X | 3 | 8 | 7 (0111) |
| 4 | X | 7 | 4 | 1 (0001) |
| 5 | X | 3 | 10(N+2) | 15 (1111) |
| 6 | X | 6 | 3 | 0 (0000) |
| 7 | X | 8 | 6 | 4 (0100) |
| 8 | X | 1 | 2 | 14 (1110) |
| 9 | 0 | | | |
| 10 | 1 | | | |

Nu wordt de rule met index waarde 5 verwijdert. Dit doen we door alle state's, net'en en rule's één positie naar boven op te schuiven vanaf index waarde 6.

| | state | net | | rule |
|---|---|---|---|---|
| 1 | X | 7 | 6 | 12 (1100) |
| 2 | X | 10(N+2) | 6 | 12 (1100) |
| 3 | X | 3 | 8 | 7 (0111) |
| 4 | X | 7 | 4 | 1 (0001) |
| 5 | X | 6 | 3 | 0 (0000) |
| 6 | X | 8 | 6 | 4 (0100) |
| 7 | X | 1 | 2 | 14 (1110) |
| 8 | 0 | | | |
| 9 | 1 | | | |
| 10 | | | | |

In net worden alle waarden die groter zijn dan de verwijderde index waarde (5) met 1 verlaagt. Om de verwijzing naar de juiste index waarde weer kloppend te maken. Het nieuwe netwerk ziet er dan als volgt uit.

| | state | net | | rule |
|---|---|---|---|---|
| 1 | X | 6 | 5 | 12 (1100) |
| 2 | X | 9(N+2) | 5 | 12 (1100) |
| 3 | X | 3 | 7 | 7 (0111) |
| 4 | X | 6 | 4 | 1 (0001) |
| 5 | X | 5 | 3 | 0 (0000) |
| 6 | X | 7 | 5 | 4 (0100) |
| 7 | X | 1 | 2 | 14 (1110) |
| 8 | 0 | | | |
| 9 | 1 | | | |
| 10 | | | | |

Dezelfde stappen worden doorlopen om alle 0'en in `rule` met index waarde 5 te verwijderen.

| | state | net | | rule |
|---|---|---|---|---|
| 1 | X | 5 | 7(N+1) | 12 (1100) |
| 2 | X | 8(N+2) | 7(N+1) | 12 (1100) |
| 3 | X | 3 | 6 | 7 (0111) |
| 4 | X | 5 | 4 | 1 (0001) |
| 5 | X | 6 | 7(N+1) | 4 (0100) |
| 6 | X | 1 | 2 | 14 (1110) |
| 7 | 0 | | | |
| 8 | 1 | | | |
| 9 | | | | |
| 10 | | | | |

Om dit netwerk verder te reduceren wordt in `net` gezocht naar een rij met alleen maar verwijzingen naar $N+1$ en/of $N+2$. Deze waarden zijn namelijk constant en blijft de bijbehorende `state` dus ook constant. Deze $X$ waarde is nu te berekenen. Afhankelijk van de uitkomst van $X$ wordt de verwijzing in `net` vervangen door $N+1$ ($X=0$) of $N+2$ ($X=1$). `state` wordt vervolgens verwijdert door de `state`'s, `net`'en en `rule`'s één positie naar boven op te schuiven. In `net` worden alle verwijzingen weer kloppend gemaakt zoals eerder is beschreven.

In het voorbeeld heeft index waarde 2 als enige een rij met uitsluitend $N+1$ en/of $N+2$ verwijzingen. $X$ kan worden berekend en heeft als uitkomst 0. In `net` wordt index waarde 2 vervangen door $N+1$. Hierna wordt `state` verwijdert en `net` kloppend gemaakt.

| | state | net | | rule |
|---|---|---|---|---|
| 1 | X | 4 | 6(N+1) | 12 (1100) |
| 2 | X | 2 | 5 | 7 (0111) |
| 3 | X | 4 | 3 | 1 (0001) |
| 4 | X | 5 | 6(N+1) | 4 (0100) |
| 5 | X | 1 | 6(N+1) | 14 (1110) |
| 6 | 0 | | | |
| 7 | 1 | | | |
| 8 | | | | |
| 9 | | | | |
| 10 | | | | |

De `state` $(X)$ kan ook uitgerekend worden, voor de rijen in `net`, waar maar één $N+1$ of $N+2$ verwijzing staat. Dit wordt gedaan door de berekening tweemaal uit te voeren. Eenmaal met de aanname dat de onbekende `state` waarde 0 is en eenmaal met de aanname dat de onbekende `state` waarde 1 is. Wanneer het antwoord van beide berekeningen hetzelfde is, dan is de $X$ constant. Dus wordt de bijbehorende index waarde vervangen in `net` door $N+1$ of $N+2$. Waarna de index waarde wordt verwijdert. Als het antwoord van beide berekeningen verschillend is, kan hier geen reductie toegepast worden.

In het voorbeeld bevat de eerst rij in net een $N+1$ verwijzing. Met de aanname dat de onbekende waarde $(net(1,1) = 4, state(4) = X)$ achtereenvolgens 0 en 1 is wordt de berekening gedaan. Met de aanname 0 is het antwoord 0 en met aanname 1 wordt het antwoord ook 0. Aangezien beide antwoorden gelijk zijn kan de verwijzing naar index waarde 1 in net vervangen worden door $N+1$.

| | state | net | | rule |
|---|---|---|---|---|
| 1 | X | 1 | 4 | 7 (0111) |
| 2 | X | 3 | 2 | 1 (0001) |
| 3 | X | 4 | 5(N+1) | 4 (0100) |
| 4 | X | 5(N+1) | 5(N+1) | 14 (1110) |
| 5 | 0 | | | |
| 6 | 1 | | | |
| 7 | | | | |
| 8 | | | | |
| 9 | | | | |
| 10 | | | | |

Deze stappen worden net zolang herhaalt tot het netwerk niet verder gereduceerd kan worden. Bij kleine netwerken is het mogelijk het netwerk tot 0 state's te reduceren. Het voorbeeld heeft het volgende eindresultaat.

| | state | net | | rule |
|---|---|---|---|---|
| 1 | X | 2 (N+1) | 1 | 1 (0001) |
| 2 | 0 | | | |
| 3 | 1 | | | |
| 4 | | | | |
| 5 | | | | |
| 6 | | | | |
| 7 | | | | |
| 8 | | | | |
| 9 | | | | |
| 10 | | | | |

In pseudocode ziet de hierboven beschreven methode er als volgt uit.

```
SUBROUTINE opschuiven ( nieuwe waarde )
  DO i=1,nieuwe N
    DO j=1,K
    IF (net(i,j) == i) THEN
      net(i,j) = nieuwe waarde
    END DO
  END DO
  schuif de rule's van i t/m nieuwe N een positie op
  schuif de net's van i t/m nieuwe N een positie op
  verlaag alle waarden in net met 1
  schuif de state's van i t/m N+2 een positie op
```

```
  nieuwe N = nieuwe N - 1
END


SUBROUTINE verbouw netwerk

  nieuwe N = N

  ! verwijder alle rule's met alleen maar nullen
  DO i=1, nieuwe N
    IF (rule == 0) THEN
      opschuiven( N+1 )
    END IF
  END DO

  ! verwijder alle rule's met alleen maar enen
  DO i=1, nieuwe N
    IF (rule == (2**(2**K)-1)) THEN
      opschuiven( N+2 )
    END IF
  END DO


  DO WHILE (niet verandert)

    !  beide netwerk elementen zijn constant
    DO i=1,nieuwe N
      IF (net(i,1)==N+1 OR net(i,1)==N+2) AND
         (net(i,2)==N+1 OR net(i,2)==N+2) THEN
        aangeven dat het netwerk verandert wordt
        constante uitrekenen
        onthoudt welke waarde die het moet worden
        opschuiven( constante die het moet worden )
      END IF
    END DO

    !  een van beide netwerk ingangselementen is constant
    DO i=1,nieuwe N
      IF (net(i,1)==N+1 OR net(i,1)==N+2) THEN

        constante met onbekende waarde is 0 uitrekenen
        onthoudt uitkomst

        constante met onbekende waarde is 1 uitrekenen
        onthoudt uitkomst

        IF (beide uitkomsten gelijk zijn) THEN
```

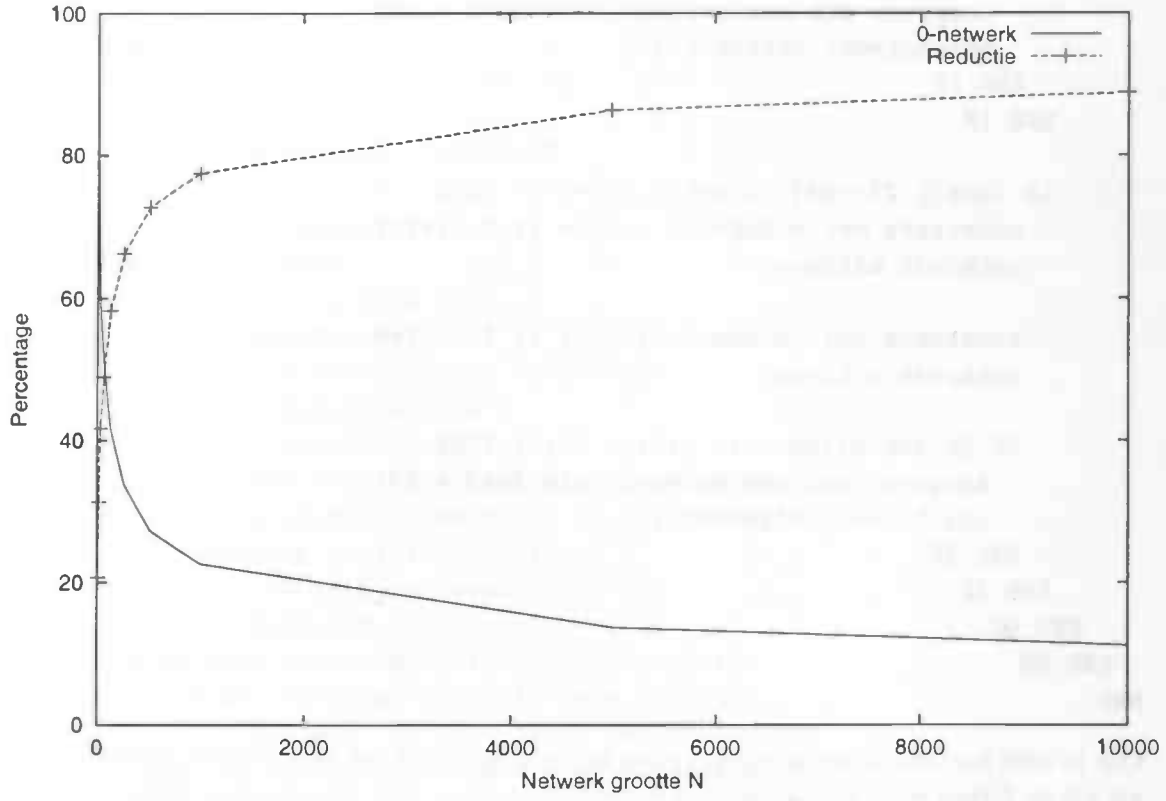
```
          aangeven dat het netwerk veranderd wordt
          opschuiven( uitkomst )
        END IF
      END IF

      IF (net(i,2)==N+1 OR net(i,2)==N+2) THEN
        constante met onbekende waarde is 0 uitrekenen
        onthoudt uitkomst

        constante met onbekende waarde is 1 uitrekenen
        onthoudt uitkomst

        IF (beide uitkomsten gelijk zijn) THEN
          aangeven dat het netwerk veranderd wordt
          opschuiven( uitkomst )
        END IF
      END IF
    END DO
  END DO
END
```

Om te zien hoe snel deze code een cyclus kan vinden word het zelfde kleine netwerk berekent als bij de fortan code gedaan is. Als voorbeeld word er voor $K = 2$, $N = 1000$, één toestand en één netwerk genomen. De gevectoriseerde fortran code met de reductie van het netwerk rekent dit booleaans netwerk in $0,005$ seconden uit. Let op, de reductie van het netwerk is niet meegerekent.




Figuur 5.1: Percentage verwijderde `state`'s en het percentage 0-netwerken die voorkomen bij verschillende netwerk groottes.

### 5.2.2 Reductie resultaten

Na bovenstaande reductie methode te hebben geïmplementeerd is deze toegepast op verschillende netwerken. Voor 1000 verschillende netwerken is nagegaan hoeveel `state`'s er berekend moesten worden voor en na de reductie. Het percentage dat als nog berekend moest worden is voor diverse netwerken in figuur 6.1 weergegeven. Hierin is te zien dat bij grote netwerken zo'n 90% minder `state`'s berekend hoeven te worden. In de grafiek is ook weergegeven het percentage netwerken dat tot 0 `state`'s gereduceerd kan worden. Bij kleine netwerken is het percentage dat tot 0 `state`'s gereduceerd kan worden hoger dan bij grote netwerken. Bij grote netwerken is het overigens nog altijd mogelijk netwerken tot 0 `state`'s te reduceren.

## 5.3 Resultaten

Het voorbeeld netwerk dat door alle verschillende code is berekent is er goed te zien welke code het snelste is. Het voorbeeld netwerk is $K = 2$, $N = 1000$, één toestand en één netwerk genomen. De snelheden zijn:

| | |
|---|---|
| fortran code | 20,000 sec. |
| BMM code | 0,026 sec. |
| gevec. fortran code | 0,013 sec. |
| gevec. fortran code met reductie | 0,005 sec. |

# Hoofdstuk 6

# Het dynamisch gedrag

In het artikel [3] wordt beweerd dat het aantal limiet cycli evenredig is met het aantal genen $N$, maar dit is alleen voor kleine netwerken ($< 128$) aangetoond. Of dit voor alle netwerken geldt is niet onderzocht. Kauffman [10] vond een voor grotere netwerken een evenredigheid met $\sqrt{N}$. We vermoeden dat dit komt door het gebruik van minder krachtige computers waarop de toestands-ruimte niet goed kon worden doorzocht. Om hierover een uitspraak te kunnen doen moet gekeken worden hoe het aantal cycli zich gedraagt als functie van $N$ in grote netwerken. Om dit te onderzoeken werd het netwerk eerst gereduceerd, zoals beschreven in het vorige hoofdstuk.

## 6.1 Aanpassingen aan de code

Om de berekeningen na te doen zoals in het artikel [3] is beschreven, zal er eerst een nieuwe versie van de code gemaakt moeten worden. De aanpassingen worden hier in pseudocode beschreven en de werkelijke code staat in de bijlage A-3.

Er komen twee lussen om de oude code heen. Één om het een aantal netwerken te kunnen bereken en één om het een aantal nieuwe begintoestanden te berkenen. Binnen de lus van de aantal verschillende begin toestanden word er ook nog eens gebruik gemaakt van een lijst. Die lijst bevat ten eerste de lengte van de cycle en ten tweede hoevaak deze gevonden is en als laatste de eindtoestand. Zodra er een cycle is berekent word er bij het zoeken naar andere cycle lengtes van het zelfde netwerk gebruik van deze eindtoestand. Zodra er een toestand gelijk is aan een eindtoestand van een gevonden cycle wordt er gestopt met zoeken naar de cycle lengte. Omdat deze al bekent is, namelijk die van de eindtoestand in de lijst. Er hoeft dus alleen nog maar het aantal van die eindtoestand in de lijst verhoogt te worden en er kan verder gegaan worden met het zoeken. Aan het eind van van alle gevonden cycle lengtes word er gekeken hoeveel verschillende waarden we hebben gevonden. Deze worden opgetelt bij de waarden die we al gevonden hadden. Er wordt vervolgens door gegaan met een nieuwe netwerk. Mocht er nu toevallig een netwerk gereduceerd worden tot nul toestanden dan word er een nieuw netwerk gemaakt en de tot nul gereduceerd netwerk word niet meegeteld.

In pseudocode ziet het er als volgt uit;

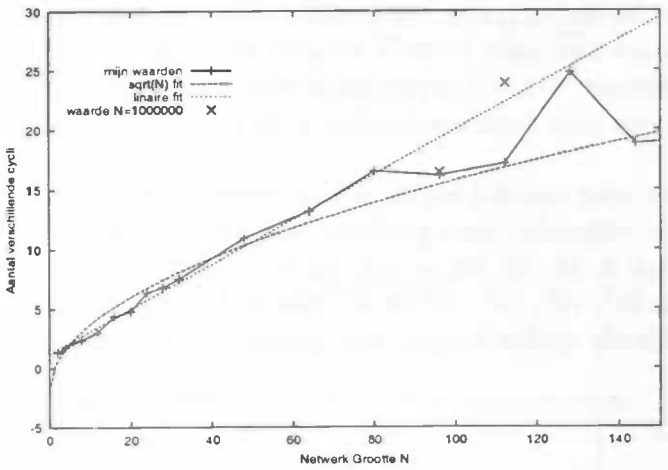
```
K = 2
N = 8
```

```
NETWEKEN = 50
TOESTANDEN = 10000

aantal netwerken = 0
DO WHILE aantal netwerken < NETWEKEN
  maak een nieuwe netwerk
  maak een nieuwe rij regels
  reduceer het netwerk
  IF netwerk niet leeg THEN
    aantal toestanden = 0
    DO WHILE aantal toestanden < TOESTANDEN
      maak nieuwe begintoestand
      onthoud begintoestand
      zorg voor een lege lijst van begintoestanden
       met de aantal erbij hoe groot de cycle lengte
       is en hoevaak ze gevonden zijn
      DO WHILE niet gelijke toestanden
        doe een tijdstap
        zoek deze toestand op in de lijst van al
         gevonden toestanden en als deze bestaat
         noteer dat dan. We zijn klaar met het zoeken
         naar de cycle lengte neem een volgende nieuwe
         begintoestand
        stop met zoeken als de cycle lengte groter is geworden
         dan 100000.
      END DO
      houd bij in een lijst
       hoeveel cycles er gevonden zijn en met welke lengte
      aantal toestanden = aantal toestanden + 1
    END DO
    aantal netwerken = aantal netwerken + 1
  END IF
END DO
schrijf de gevonden waarden (cycle lengte en hoevaak
 deze gevonden zijn) in een bestand weg
```

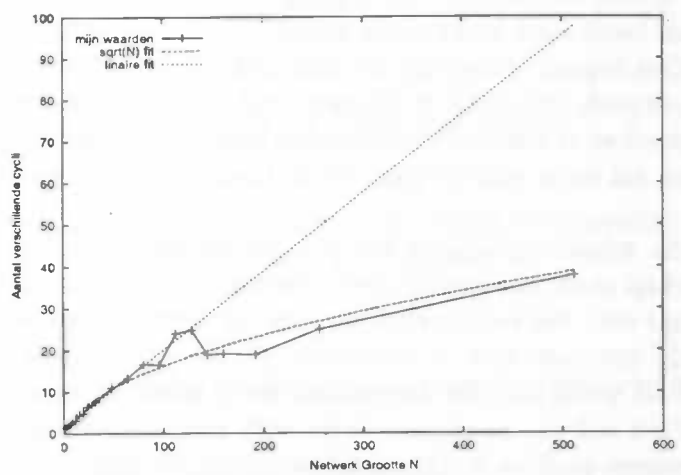
## 6.2 Het onderzoek

Als eerste word de grafiek die in het artikel [3] staat nagemaakt. Dan is te zien of de code correct is. Om de grafiek te maken is er gekozen om 50 gereduceerde netwerken te gebruiken en die elk $10^5$ verschillende begintoestanden door te rekenen. De variabele $K$ word voor elk netwerk op 2 gezet en de variabele $N$ word gevarieerd met de volgende waarden 2, 4, 6, 8, 12, 16, 20, 24, 28, 32, 48, 64, 80, 96, 112, 128 en 144. De lage $N$ waarde 2 tot en met 8 zijn gekozen om tekijken of de lijn echt recht loopt of niet. Om te kijken of de lijn ook na 128 recht blijft lopen is er alvast één waarde groter dan 128 in de grafiek mee genomen. In de figuur 6.1 zijn de bevindingen te zien. Ook zijn er nog een antal andere lijnen bij getekend,




Figuur 6.1: Verschilende $N$ waarde, de nagemaakte grafiek zoals in het artikel [3] is beschreven.

een lineaire fit en een $\sqrt{N}$ fit curve. Zoals verwacht zijn de gevonden waarden dicht bij de rechte lijn, behalve bij de $N = 96$, $N = 112$ en $N = 144$. De waarden 96 en 112 zijn nog een keer berekent en dan met $10^6$ verschillende begin toestanden. De antwoorden die daar bij horen zijn als kruisje in figuur 6.1 getekent. De lineaire bevinding is dus afhankelijk van het aantal genomen begintoestanden.
In het begin zijn er veel kleine waarden genomen om beter te kijken of de lijn echt recht loopt en het antwoord is ja. Zelfs bij de waarde 2, 4 en 6 loopt de lijn recht. Alleen naarmate $N$ verhoogd wordt gaat deze alle kanten op.
Dit is goed te zien als er met dezelfde instellingen die gebruikt zijn voor figuur 6.1 de volgende $N$ waarden er bij getekent worden, 160, 192 , 256 en 512. Hier is goed te zien dat de lineaire

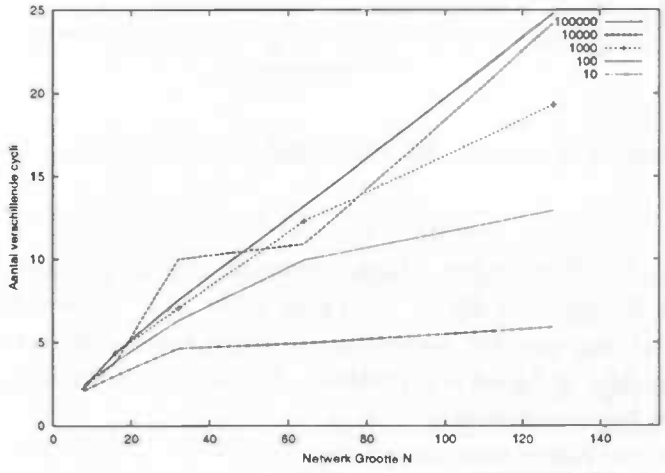



Figuur 6.2: Verschilende $N$ waarde, de nagemaakte grafiek zoals in het artikel [3] is beschreven maar dan met een grotere $N$ waarden.

fit het beste is in het begin maar naarmate de $N$ waarden verhoogt worden deze niet meer

in de buurt is. Mijn bewering is dat het aantal begintoestanden dat gekozen moeten worden verhoogd moet worden naarmate je de $N$ waarde verhoogt. Alleen dan krijg wordt een lineaire verhouding zichtbaar. Dit is dus precies de reden waarom Kauffman een $\sqrt{N}$ verhouding zag. Hij nam maar een paar begintoestanden voor verschillende netwerken.

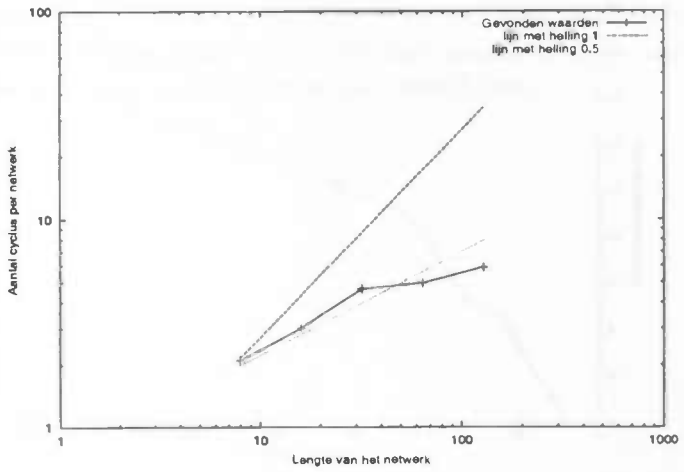
Om te laten zien wat voor verschil het uit maakt hoeveel begintoestanden er gebruikt worden. Is er in figuur 6.3 de volgende lijnen getekent. Het figuur is gemaakt met de volgende instellingen, $K = 2$, $N$ van 8, 16, 32, 64 en 128, 50 verschillende gereduceerde netwerken en voor elk netwerk neem ik $10^1$, $10^2$, $10^3$, $10^4$ en $10^5$ nieuwe begin toestanden om ze door te rekenen. Het aantal verschillende cyclus lengtes zijn gedeelt door 50 en in figuur 6.3 geplaatst. In



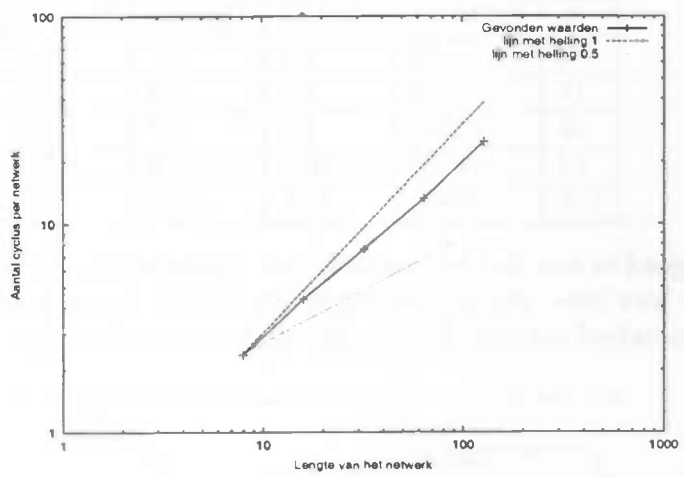

Figuur 6.3: Verschillende netwerken uitgerekent d.m.v. verschillend aantalen begintoestanden.

figuur 6.3 is goed te zien dat het dus wel degelijk uit maakt hoeveel begintoestanden gebruikt worden. Kauffman heeft deze grafiek ook gemaakt. De onderste lijn komt dan ook overeen met zijn grafiek. Ook is goed te zien dat het nauwlijks uit maakt of je nu $10^1$ of $10^6$ toestanden berekent als het netwerk ($N$) klein is, bijvoorbeeld $N = 8$. Maar zodra het netwerk groter word ($N = 128$) moet er al veel meer toestanden berekent worden om nog een goed resultaat te kunnen behalen, als we er van uit gaan de de lineaire eigenschap de juiste is.

In de logaritmische schaalverdeling is het ook duidelijke te zien dat de cycle lengtes naar de lineaire eigenschap gaan naarmate je meer toestanden berekent. Figuur 6.4 geeft de lijn in logaritmische schaal weer van een berekening voor $10^1$ begintoestanden waarbij $N$ de waarden 8, 16, 32, 64 en 128 genomen zijn. $K = 2$ en er zijn 50 verschillende gereduceerde netwerken gebruikt. Figuur 6.5 geeft dezelfde berekening weer maar dan voor $10^5$ begintoestanden. Om na te gaan of het ook uit maakt om de netwerk grootte te veranderen is er voor $N$ 8, 16, 32, 64 en 128 genomen en $K = 2$. De aantal verschillende begin toestanden wordt een vaste waarde voor genomen, namelijk 1000. De aantalenl netwerken die doorzocht worden zijn 50, daanrna 100 en als laatste 150 stuks. Elk netwerk word 1000 keer onderzocht dus de laatste netwerk heeft meer cycle lengtes gevonden dan de eerste. Tevens is het zo dat naarmate de $N$ waarde groter wordt de lijn onnauwkeuriger word. Zoals in de grafiek te zien is maakt het dus niets uit hoeveel netwerken we gebruikten. De lijnen blijven aan het begin allemaal gelijk




Figuur 6.4: $10^1$ begin toestanden.

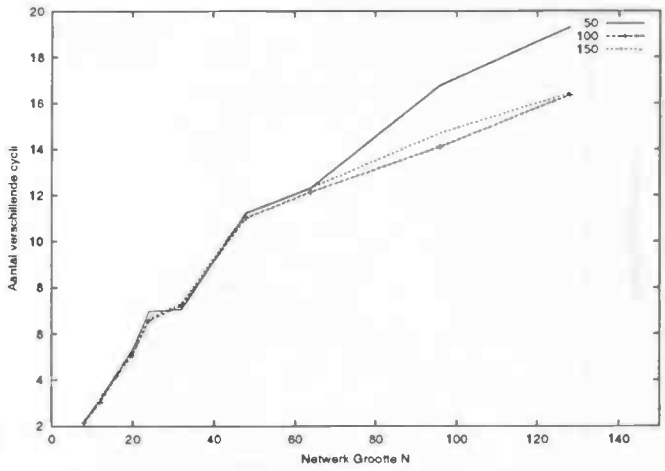



Figuur 6.5: $10^5$ begin toestanden.

alleen aan het einde wijken ze iets van elkaar af. Dit komt voornamelijk doordat er voor het aantal begintoestanden een te kleine waarde voor gekozen is.

## 6.3 Hoeveel begintoestanden moeten er gebruikt worden ?

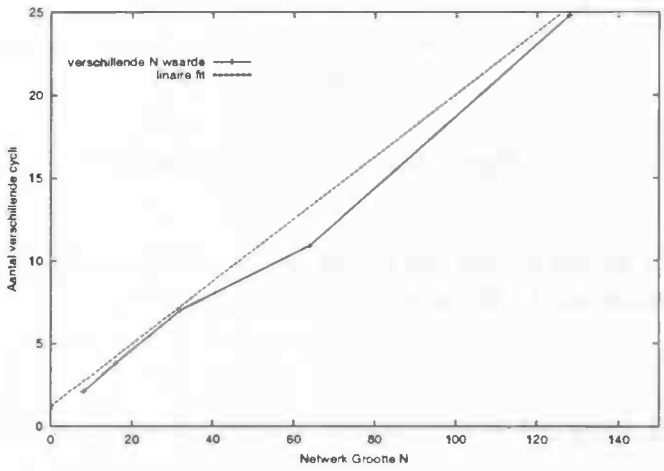
Aan de hand van de gevonden waarden word geprobeer om een antwoord te krijgen op de vraag hoeveel beginteostanden er nu gekozen moeten worden. De waarden die er gevonden zijn staan in de hieronder beschreven tabel. Kolom 1 geeft de waarde van $N$ aan. Kolom 2 toont de aantal begin toestanden die er genomen zijn. Kolom 2 tootn de gemiddelde cycle lengte die berekent is. De gemiddelde cycle lengte die volgens de lineaire functie bereknt is staat in kolom 4. Het verschil tussen de kolom 3 en 4 word in de laatste kolom genoteert.




Figuur 6.6: .

| $N$ | NrOfCyc | CycLen | CycLen | Verschil |
|---|---|---|---|---|
| 8 | 10 | 2.1 | 2.7 | 0.6 |
| 16 | 100 | 3.8 | 4.2 | 0.4 |
| 32 | 1000 | 7.0 | 7.2 | 0.2 |
| 64 | 10000 | 10.9 | 13.3 | 2.4 |
| 128 | 100000 | 24.8 | 25.3 | 0.5 |

In de tabel is nu goed te zien dat het verschil, met uitzondering van $N = 64$, steeds ongeveer een $0,5$ is. Door met deze waarden waarden een nieuwe figuur te tekenen is goed te zien dat de nieuwe lijn inderdaad heel er dicht bij de lineaire fit lijn ligt. Om voor $N = 64$ een




Figuur 6.7: Verschillende aantal begintoestanden.

waarde te krijgen die in de buurt ligt van de lineaire fit lijn moet er $10^5$ begin toestanden berekent worden. Dan komt de cycle lengte op 13.2. Dat enkele $N$ waarden een ander aantal begintoestanden nodig heeft om in de buurt te komen van de lineaire fit lijn is al eerder

geconstateerd. Bij figuur 6.1 komen de waarde 96 en 112 niet in de buurt. Daarvoor is nog eens 10 keer zoveel berekeningen voor nodig. De hier geconstateerde aantal begintoestanden is dus een richtlijn en kan voor enkele $N$ waarden verschillen.

# Aanbevelingen

## Fortran

De gevectoriseerde Fortran code is de code van dit onderzoek geworden. Omdat de berekening snel uitgevoerd kan worden door gebruik te maken van gevectoriseerde bit manipulaties.

## BMM

Om een booleaanse netwerk sneller uit te kunnen rekenen, met behulp van de BMM hardware van de Cray SV1e, zal een andere representatie van een booleaanse netwerk gemaakt moeten worden. De representatie die nu gebruikt word zorgt er voor ijle matrixen (matrix bevat veel 0'en). Bij een nieuwe of andere representatie van het booleaanse netwerk is, bij gebruik van de BMM, het aan te bevelen om geen standaard bit operaties te gebruiken (AND, OR, etc.), omdat die sneller door gevectoriseerde code berekend wordt. Wanneer hieraan is voldaan, is met de BMM, alleen bij grote netwerken, snelheidswinst ten opzichte van Fortran code te behalen. Dit komt omdat bij grote netwerken vele berekeningen met een geladen matrix gedaan kunnen worden. De efficiënte berekeningen compenseren de lange tijd om een matrix in te laden.

Om netwerken sneller uit te rekenen wordt er netwerk reductie toegepast [3]. Speciale hardware wordt nauwelijks toegepast. Parallelle rekeneenheden hebben als voordeel dat elk netwerk op een aparte processor tegelijkertijd berekend kan worden. In de toekomst moet deze mogelijkheid verder uitgediept worden.

## Het dynamische gedrag

Er is geconcludeerd dat voor grote netwerken het aantal verschillende cyclus lengtes als functie van de netwerk grootte een lineaire verband heeft, zoals in het artikel [3] beweerd wordt. De bewering kan alleen gevonden worden als er ook voldoende aantal begintoestanden genomen wordt, anders zal het $\sqrt{N}$ verband al weer snel gevonden worden.
Door meer begintoestanden te nemen is er een beter resultaat te behalen door meer netwerken door te rekenen niet.
Er kan alleen een schatting gedaan worden voor de hoeveelheid begintoestanden die een bepaalde netwerk grootte $N$ nodig heeft om een juiste berekening te krijgen.

# Samenvatting

Boolean netwerken zijn geïntroduceerd als eenvoudig model voor de complexe interactie in een regulatie systeem van levende cellen. Graven en differentiaal vergelijkingen zijn andere simulatie methoden om regulatie systemen van levende cellen na te bootsen. Boolean netwerken worden hier als simulatie model gebruikt. Door gebruik te maken van de Cray SV1e Bit Matrix Multiplyer (BMM) wordt geprobeerd om boolean netwerken sneller te bereken. De huidige representatie van de boolean netwerk maakt gebruik van eenvoudige bit manipulaties. De Cray SV1e kan eenvoudige bit manipulaties sneller met vectoren berekenen dan met de BMM. Hierdoor wordt er geen tijdwinst met de BMM code behaald. Daarnaast is onderzocht of het verband tussen netwerk grootte $N$ en aantal limiet cycles lineair is voor alle netwerken. Voor grotere netwerken blijkt er een ook een lineaire verband te bestaan als er maar genoeg toestanden berekend wordt.

# Summary

Boolean networks were introduced as simplified models of the complex interactions in the regulatory networks of living cells. Graphs and differential equations are other simulation models to model regulatory networks of living cells. Boolean networks have been used here. The Cray SV1e Bit Matrix Multiply (BMM) was used to try to speed up the boolean network calculation. The representation of the boolean network uses simple bit manipulations. The Cray SV1e is able to calculate simple bit manipulations faster using vector instructions instead of the BMM. Therefore no speed gain was encounted. Furthermore the question whether the relationship between network size $N$ and number of limit cycles is lineair was investigated for all networks. For large networks we also see a lineaire behavior.

# Bibliografie


[1] Apter, M. J., *Cybernetics and Development*, Oxford: Pergamon Press, 1966

[2] Bonner, J. F., *The Molecular Biology of Development*, London: Oxford University Press, 1965

[3] Bilke, S., Sjunnesson F., *Stability of the Kauffman model* Physical review E, vol 65, 016129, 2001.

[4] Bruin, R. de, Homan, D., *Local Cray Guide* `http://www.rug.nl/hpc`, folder_cray.pdf, 2002

[5] *Cray Manuals - Fortran Language Reference Manual* `http://www.cray.nl/craydoc/`, vol 1,2 en 3, S-3692-35, S-3693-35 en S-3694-35

[6] Goodman, S.R., *Medical cell biology* ISBN 0-397-51254-6, Lippincott - Raven publishers, 1994

[7] Jacob, F. & Monod, J., 21st Symp. Soc. Study of Development and Growth, London: Academic Press, 1963

[8] Jong, H. de, *Modeling and Simulation of Genetic Regulatory Systems: A literature Review* `http://www.inria.fr/rrrt/rr-4032.html`, ISSN 0249-6399, 2000

[9] Junqueira, L.C., Carneiro, J., e.a., *Functionele histologie* ISBN 90-6348-379-1, Bunge, zevende geheel herziende druk, 1996

[10] Kauffman, S. A., *Metabolic stability and epigenesis in randomly constructed genetic nets.* J. Theoret. Biol., 22:437-467, 1969.

[11] McAdams, H. H. en Arkin, A., *Simulation of prokaryotic genetic circuits.* Annu. Rev. Biophys. Biomol. Struct., 27:199 224, 1998.

[12] Sugita, M., J. Theoret. Biol. 4, 179, 1963

[13] Voorbeelden en uitleg over de BMM,
`http://casaccia.enea.it/hermes/bmm/practical.html`

# Bijlagen

## A-1 De gehele fortran code versie 1

```
!
! Brend A. Dikkers, versie 1.0.0 van de Boolean netwerk fortran code
!

      PROGRAM versie1
        INTEGER, PARAMETER :: K=2
        INTEGER, PARAMETER :: N=32

        INTEGER :: iff, idum, ma(56), inext, inextp
        INTEGER :: net(N,K), state(N), state2(N), rule(N)
        INTEGER :: i, j, out
        INTEGER :: dummy, temp
        INTEGER :: d
        LOGICAL :: OK, b

        ! initialisatie
        iff=0
        idum=42
        DO i=1,56
          ma(i)=0
        END DO
        inext=1
        inextp=32

        ! CALL initConnections(net)
        DO i=1, N
          CALL randomInt(iff, idum, ma, inext, inextp, out)
          net(i, 1) = MOD(out, N) + 1
          DO j=2, K
            OK = .FALSE.
            DO WHILE (.NOT. OK)
              CALL randomInt(iff, idum, ma, inext, inextp, out)
              net(i, j) = MOD(out, N) + 1
              d=1
              DO WHILE (d<j)
                OK = (net(i, d) /= net(i, j))
                d=d+1
                IF (OK) d=j
              END DO
            END DO
          END DO
        END DO

        ! CALL initRules(rule)
        DO i=1,N
```

```
      temp = 0
      DO j=1,(2**K)
        CALL randomInt(iff, idum, ma, inext, inextp, out)
        dummy = MOD(out, 2)
        temp = temp*2 + dummy
      END DO
      rule(i) = temp
    END DO

    ! CALL initState(state)
    DO i=1,N
      CALL randomInt(iff, idum, ma, inext, inextp, out)
      state(i) = MOD(out, 2)
    END DO

    ! copyState
    DO i=1,N
      state2(i) = state(i)
    END DO

    ! Figure of 6 cycle detection
    b = .FALSE.
    DO WHILE (.NOT. b)
      CALL stepState(state, net, rule)
      CALL stepState(state2, net, rule)
      CALL stepState(state2, net, rule)
      CALL equaleState(state, state2, b)
    END DO

    ! member of limit cycle found
    i = 0
    b = .FALSE.
    DO WHILE (.NOT. b)
      CALL stepState(state, net, rule)
      i=i+1
      CALL equaleState(state, state2, b)
    END DO

    ! length of limit cycle found
    PRINT *, 'cycle length = ', i
  END

SUBROUTINE randomInt(iff, idum, ma, inext, inextp, output)
  INTEGER :: output
  INTEGER :: iff, idum, ma(56), inext, inextp
  PARAMETER (MBIG=1000000000)
  PARAMETER (MSEED=161803398)
```

```
  PARAMETER (MZ=0)
  INTEGER :: mj, mk, i, ii, kk


  IF (iff == 0) THEN
    iff=1
    IF (idum < 0) THEN
      mj=MSEED+idum
    ELSE
      mj=MSEED-idum
    END IF
    mj = MOD(mj, MBIG)
    ma(56)=mj
    mk=1
    DO i=1,54
      ii=MOD((21*i), 55) + 1
      ma(ii)=mk
      mk=mj-mk
      IF (mk < MZ) mk = mk + MBIG
        mj=ma(ii)
    END DO
    DO kk=1,4
      DO i=2,56
        ma(i) = ma(i) - ma(MOD(((i-1)+30), 55)+2)
        IF (ma(i) < MZ) THEN
          ma(i) = ma(i) + MBIG
        END IF
      END DO
    END DO
    inext=1
    inextp=32
  END IF
  inext = inext + 1
  IF (inext == 57) inext=2
  inextp = inextp + 1
  IF (inextp == 57) inextp=2
  mj=ma(inext)-ma(inextp)
  IF (mj < MZ) mj = mj + MBIG
  ma(inext)=mj
  output = mj
  RETURN
END

SUBROUTINE equaleState(source1, source2, out)
  INTEGER, PARAMETER :: N=32
  INTEGER :: source1(N), source2(N)
  INTEGER :: i
```

```
  LOGICAL :: out

  out = .TRUE.
  i = 1
  DO WHILE (i < N)
    IF (source1(i) /= source2(i)) THEN
      out = .FALSE.
      i = N
    END IF
    i = i+1
  END DO

  RETURN
END

SUBROUTINE kInputs(rule, input, state)
  INTEGER :: rule, input, state
  INTEGER :: x

    x = IAND(rule, LSHIFT(1,input))
    IF (x /= 0) THEN
      state = 1
    ELSE
      state = 0
    END IF

  RETURN
END

SUBROUTINE stepState(state, net, rule)
  INTEGER, PARAMETER :: N=32
  INTEGER, PARAMETER :: K=2

  INTEGER :: state(N), rule(N), input(N)
  INTEGER :: net(N,K)
  INTEGER :: i, j

  DO i=1,N
    input(i) = state(net(i,K))
    DO j=(K-1),1,-1
      input(i) = LSHIFT(input(i), 1) + state(net(i,j))
    END DO
  END DO

  DO i=1,N
    CALL kInputs(rule(i), input(i), state(i))
  END DO
```

```
      RETURN
    END
```

## A-2 De gehele fortran code versie 3

```
!
! Brend A. Dikkers, versie 3.0.0 van de Boolean netwerk fortran code
!

      MODULE const
        INTEGER, PARAMETER :: K=2
        INTEGER, PARAMETER :: N=32
        INTEGER, DIMENSION(56) :: ma
        INTEGER :: inext, inextp, iff, idum
      END MODULE


      PROGRAM progB
        USE const
        INTEGER :: net(N,K), state(N), state2(N), rule(N)
        INTEGER :: i
        LOGICAL :: b

        ! initialisatie
        iff=0
        idum=42
        ma=0

        CALL initConnections(net)

        CALL initRules(rule)

        CALL initState(state)

        state2 = state

        ! Figure of 6 cycle detection
        b = .FALSE.
        DO WHILE (.NOT. b)
          CALL stepState(state, net, rule)
          CALL stepState(state2, net, rule)
          CALL stepState(state2, net, rule)
          CALL equaleState(state, state2, b)
        END DO

        ! member of limit cycle found
        i = 0
        b = .FALSE.
        DO WHILE (.NOT. b)
          CALL stepState(state, net, rule)
```

```
      i=i+1
      CALL equaleState(state, state2, b)
    END DO

    ! length of limit cycle found
    PRINT *, 'cycle length = ', i
  END

SUBROUTINE randomInt(output)
  USE const
  INTEGER :: output
  PARAMETER (MBIG=1000000000)
  PARAMETER (MSEED=161803398)
  PARAMETER (MZ=0)
  INTEGER :: mj, mk, i, ii, kk


  IF (iff == 0) THEN
    iff=1
    IF (idum < 0) THEN
      mj=MSEED+idum
    ELSE
      mj=MSEED-idum
    END IF
    mj = MOD(mj, MBIG)
    ma(56)=mj
    mk=1
    DO i=1,54
      ii=MOD((21*i), 55) + 1
      ma(ii)=mk
      mk=mj-mk
      IF (mk < MZ) mk = mk + MBIG
        mj=ma(ii)
    END DO
    DO kk=1,4
      DO i=2,56
        ma(i) = ma(i) - ma(MOD(((i-1)+30), 55)+2)
        IF (ma(i) < MZ) THEN
          ma(i) = ma(i) + MBIG
        END IF
      END DO
    END DO
    inext=1
    inextp=32
  END IF
  inext = inext + 1
  IF (inext == 57) inext=2
```

```
  inextp = inextp + 1
  IF (inextp == 57) inextp=2
  mj=ma(inext)-ma(inextp)
  IF (mj < MZ) mj = mj + MBIG
  ma(inext)=mj
  output = mj
  RETURN
END

SUBROUTINE equaleState(source1, source2, out)
  USE const
  INTEGER :: source1(N), source2(N)
  INTEGER :: i
  LOGICAL :: out

  out = .TRUE.
  i = 1
  DO WHILE (i < N)
    IF (source1(i) /= source2(i)) THEN
      out = .FALSE.
      i = N
    END IF
    i = i+1
  END DO

  RETURN
END

SUBROUTINE stepState(state, net, rule)
  USE const
  INTEGER :: state(N), rule(N), input(N)
  INTEGER :: net(N,K)
  INTEGER :: i, j, x

  DO i=1,N
    input(i) = state(net(i,K))
    DO j=(K-1),1,-1
      input(i) = LSHIFT(input(i), 1) + state(net(i,j))
    END DO
  END DO

  DO i=1,N
    x = IAND(rule(i), LSHIFT(1,input(i)))
    IF (x /= 0) THEN
      state(i) = 1
    ELSE
      state(i) = 0
```

```
    END IF
  END DO

  RETURN
END

  SUBROUTINE initConnections(source)
    USE const
    INTEGER, DIMENSION(N,K) :: source
    INTEGER :: i, out, j, d
    LOGICAL :: OK
  DO i=1, N
    CALL randomInt(out)
    source(i, 1) = MOD(out, N) + 1
    DO j=2, K
      OK = .FALSE.
      DO WHILE (.NOT. OK)
        CALL randomInt(out)
        source(i, j) = MOD(out, N) + 1
        d=1
        DO WHILE (d<j)
          OK = (source(i, d) /= source(i, j))
          d=d+1
        IF (OK) d=j
        END DO
      END DO
    END DO
  END DO
    RETURN
  END

  SUBROUTINE initRules(source)
    USE const
    INTEGER, DIMENSION(N) :: source
    INTEGER :: dummy, out, temp
    DO i=1,N
      temp = 0
      DO j=1,(2**K)
        CALL randomInt(out)
        dummy = MOD(out, 2)
        temp = temp*2 + dummy
      END DO
      source(i) = temp
    END DO
    RETURN
  END
```

```
SUBROUTINE initState(source)
  USE const
  INTEGER, DIMENSION(N) :: source
  INTEGER :: i, out
DO i=1,N
  CALL randomInt(out)
  source(i) = MOD(out, 2)
END DO

  RETURN
END
```

## A-3 De gehele fortran code versie 7

```
!
! Brend A. Dikkers, versie 7.0.0 van de Boolean netwerk fortran code
!
      MODULE const
        INTEGER, PARAMETER :: K=2
        INTEGER, PARAMETER :: N=32
        INTEGER, PARAMETER :: NR_OF_CYCLE_LENGTHS=1000000
        INTEGER, PARAMETER :: NR_OF_NETWORKS=50
        INTEGER, PARAMETER :: Nplus2=N+2
        INTEGER, DIMENSION(56) :: ma
        INTEGER :: inext, inextp, iff, idum
        INTEGER :: newN
      END MODULE


      PROGRAM Boolean_netwerk
        USE const
        IMPLICIT NONE
        INTEGER :: net(N,K), state(Nplus2), state2(Nplus2), rule(N)
        INTEGER :: i
        LOGICAL :: b


        INTEGER :: net_i, state_i
        INTEGER :: ii
        INTEGER :: diff_index_2, diff_2(NR_OF_CYCLE_LENGTHS)
        INTEGER :: diff_index_1, diff_1(NR_OF_CYCLE_LENGTHS)
        LOGICAL :: containts, new_netwerk, calc

        INTEGER :: states(NR_OF_CYCLE_LENGTHS,1:N+2)
        INTEGER :: s(1:N), a, aantal_gevonden
        LOGICAL :: gevonden, state_bestaat

        ! initialisatie
        iff=0
        idum=42
        ma=0

        diff_index_2 = 1
        diff_index_1 = 1
        net_i = 1
        DO WHILE (net_i<=NR_OF_NETWORKS )
          aantal_gevonden = 0
          new_netwerk = .TRUE.
          DO WHILE ( new_netwerk )
```

```
  new_netwerk = .FALSE.
  CALL initConnections(net)
  CALL initRules(rule)
  CALL extraState(state)
  CALL rebuild(state, net, rule)
  IF (newN<=0) new_netwerk = .TRUE.
END DO

state_i = 1
DO WHILE (state_i<=NR_OF_CYCLE_LENGTHS)
  CALL initState(state)

  state2 = state

  ! Figure of 6 cycle detection
  b = .FALSE.
  calc = .TRUE.
  i = 0
  DO WHILE (.NOT. b)
    CALL stepState(state, net, rule)
    CALL stepState(state2, net, rule)
    CALL stepState(state2, net, rule)
    CALL equaleState(state, state2, b)
    i = i + 1
    IF (MOD(i,100000)==0) THEN
      calc = .FALSE.
      state_i = 100000
      b = .TRUE.
    END IF
  END DO

  IF (calc) THEN
    b = .FALSE.
    state_bestaat = .FALSE.

      a = 1
      do while (a <= aantal_gevonden)  ! door alle states door lopen
        s(1:N) = states(a,3:N+2) ! kijk of we in dezelfde cycle zitten
        gevonden = .FALSE.
        CALL equaleState(state,s,gevonden)
        if (gevonden) then  ! zo ja
         state_bestaat = .TRUE.
         states(a,2) =  states(a,2) + 1 ! tel een op bij de gevonden woo
         a = aantal_gevonden+2 ! en eindig het doorlopen
         b = .TRUE.
        end if  ! zo nee
        a = a + 1 !  kijk naar de volgende
```

```
        end do

      ! member of limit cycle found

      i = 0
      DO WHILE (.NOT. b)
        CALL stepState(state, net, rule)
        i=i+1
        CALL equaleState(state, state2, b)

        ! voor de tweede keer
        a = 1
        do while (a <= aantal_gevonden)  ! door alle states door lopen
          s(1:N) = states(a,3:N+2) ! kijk of we in dezelfde cycle zitten
          gevonden = .FALSE.
          CALL equaleState(state,s,gevonden)
          if (gevonden) then  ! zo ja
           state_bestaat = .TRUE.
           states(a,2) =  states(a,2) + 1 ! tel een op bij de gevonden woorde
           a = aantal_gevonden+2 ! en eindig het doorlopen
           b = .TRUE.
          end if  ! zo nee
          a = a + 1 !  kijk naar de volgende
        end do
      END DO

        if (aantal_gevonden==0) then
          aantal_gevonden = 1
          states(aantal_gevonden,1) = i ! cycle_length
          states(aantal_gevonden,2) = 1 ! aantal
          states(aantal_gevonden,3:N+2) = state(1:N) ! toestand
          state_bestaat = .TRUE.
        end if

        ! toevoegen
        if (.NOT. state_bestaat) then
          aantal_gevonden = aantal_gevonden + 1
          states(aantal_gevonden,1) = i ! cycle_length
          states(aantal_gevonden,2) = 1 ! aantal
          states(aantal_gevonden,3:N+2) = state(1:N) ! toestand
      end if
      state_i = state_i + 1

    END IF
  END DO

! states bevat alle gevonden states
```

```
! diff_2 bevat alle gevonden lengtes
! diff_1 bevat aantal gevonden lengtes in diff_2
  DO i=1,aantal_gevonden
    ii=1
    containts = .FALSE.
    DO WHILE (ii<diff_index_2)
      IF ( diff_2(ii) == states(i,1) ) THEN
        diff_1(ii) = diff_1(ii) + 1
        ii = diff_index_2
        containts = .TRUE.
      END IF
      ii = ii + 1
    END DO

    IF (.NOT. containts) THEN
      diff_2(diff_index_2) = states(i,1)
      diff_index_2 = diff_index_2 + 1
      diff_1(ii) = 1
      diff_index_1 = diff_index_1 + 1
    END IF
  END DO

  net_i = net_i + 1
 END DO

 ! schrijf de gevonden waarden we in een bestand
 OPEN(1, FILE='N12.dat')

  a = 0
  DO i=1,diff_index_2-1
    a = a + diff_1(i)
    WRITE (1, '(II)') diff_2(i), diff_1(i)
  END DO
  WRITE (1, '(IF)') a, (a/50.0)

  CLOSE(1)


END

SUBROUTINE randomInt(output)
  USE const
  IMPLICIT NONE
  INTEGER :: output
  INTEGER, PARAMETER :: MBIG=1000000000
  INTEGER, PARAMETER :: MSEED=161803398
  INTEGER, PARAMETER :: MZ=0
```

```
  INTEGER :: mj, mk, i, ii, kk


  IF (iff == 0) THEN
    iff=1
    IF (idum < 0) THEN
      mj=MSEED+idum
    ELSE
      mj=MSEED-idum
    END IF
    mj = MOD(mj, MBIG)
    ma(56)=mj
    mk=1
    DO i=1,54
      ii=MOD((21*i), 55) + 1
      ma(ii)=mk
      mk=mj-mk
      IF (mk < MZ) mk = mk + MBIG
        mj=ma(ii)
    END DO
    DO kk=1,4
      DO i=2,56
        ma(i) = ma(i) - ma(MOD(((i-1)+30), 55)+2)
        IF (ma(i) < MZ) THEN
          ma(i) = ma(i) + MBIG
        END IF
      END DO
    END DO
    inext=1
    inextp=32
  END IF
  inext = inext + 1
  IF (inext == 57) inext=2
  inextp = inextp + 1
  IF (inextp == 57) inextp=2
  mj=ma(inext)-ma(inextp)
  IF (mj < MZ) mj = mj + MBIG
  ma(inext)=mj
  output = mj
  RETURN
END

SUBROUTINE equaleState(source1, source2, out)
  USE const
  IMPLICIT NONE
  INTEGER :: source1(Nplus2), source2(Nplus2)
  INTEGER :: i
```

```
  LOGICAL :: out

  out = .TRUE.
  i = 1
  DO WHILE (i < newN)
    IF (source1(i) /= source2(i)) THEN
      out = .FALSE.
      i = newN
    END IF
    i = i + 1
  END DO

  RETURN
END

SUBROUTINE stepState(state, net, rule)
  USE const
  IMPLICIT NONE
  INTEGER :: state(Nplus2), rule(N), input(N)
  INTEGER :: net(N,K)
  INTEGER :: i, j !, x

  DO i=1,newN
    input(i) = state(net(i,K))
    DO j=(K-1),1,-1
      input(i) = ISHFT(input(i), 1) + state(net(i,j))
    END DO
  !END DO

  !DO i=1,N
  !  x = IAND(rule(i), ISHFT(1,input(i)))
  !  IF (x /= 0) THEN
  IF ( (IAND(rule(i), ISHFT(1,input(i)))) /= 0 ) THEN
      state(i) = 1
    ELSE
      state(i) = 0
    END IF
  END DO

  RETURN
END

  SUBROUTINE initConnections(source)
    USE const
  IMPLICIT NONE
    INTEGER, DIMENSION(N,K) :: source
    INTEGER :: i, out, j, d
```

```
  LOGICAL :: OK
DO i=1, N
  CALL randomInt(out)
  source(i, 1) = MOD(out, N) + 1
  DO j=2, K
    OK = .FALSE.
    DO WHILE (.NOT. OK)
      CALL randomInt(out)
      source(i, j) = MOD(out, N) + 1
      d=1
      DO WHILE (d<j)
        OK = (source(i, d) /= source(i, j))
        d=d+1
      IF (OK) d=j
      END DO
    END DO
  END DO
END DO
  RETURN
END

SUBROUTINE initRules(source)
  USE const
IMPLICIT NONE
  INTEGER, DIMENSION(N) :: source
  INTEGER :: dummy, out, temp, i, j
  DO i=1,N
    temp = 0
    DO j=1,(2**K)
      CALL randomInt(out)
      dummy = MOD(out, 2)
      temp = temp*2 + dummy
    END DO
    source(i) = temp
  END DO
  RETURN
END

SUBROUTINE initState(source)
  USE const
IMPLICIT NONE
  INTEGER, DIMENSION(Nplus2) :: source
  INTEGER :: i, out
DO i=1,N
  CALL randomInt(out)
  source(i) = MOD(out, 2)
END DO
```

```
  CALL extraState(source)
  RETURN
END

SUBROUTINE extraState(source)
  USE const
IMPLICIT NONE
  INTEGER, DIMENSION(Nplus2) :: source

  source(N+1) = 0
  source(N+2) = 1

  RETURN
END

SUBROUTINE rebuild(state, net, rule)
  USE const
IMPLICIT NONE
  INTEGER :: state(Nplus2), net(N,K), rule(N)
  INTEGER :: i, j, m, answer1, answer2, replace, input
  LOGICAL :: rebuilded

  newN = N
  i = 1
  DO WHILE (i<newN)
    IF (rule(i)==0) THEN  ! 0000
       !PRINT *,'new network 0', i
      CALL rebuildNetwork(i, net, (newN+1), state, rule)
       !CALL printNetwerk(state, net, rule)
    END IF

    IF (rule(i)==15) THEN  ! 1111
       !PRINT *,'new network 15', i
      CALL rebuildNetwork(i, net, (newN+2), state, rule)
       !CALL printNetwerk(state, net, rule)
      i = i - 1
    END IF

    i = i + 1
  END DO

  rebuilded = .TRUE.
  DO WHILE (rebuilded)
    rebuilded = .FALSE.
     !PRINT *,'',rebuilded
    i = 1
    DO WHILE (i<=newN)
```

```
   !PRINT *,i,'',newN
  IF ( ( ( net(i,1)==(newN+1) ) .OR. ( net(i,1)==(newN+2) ) ) .AND.
      ( ( net(i,2)==(newN+1) ) .OR. ( net(i,2)==(newN+2) ) ) ) THEN
    ! uit rekenen de state
    input = ISHFT(state(net(i,2)), 1) + state(net(i,1))
    IF ( (IAND(rule(i), ISHFT(1,input))) /= 0 ) THEN
      replace = newN+2 !state(i) = 1
    ELSE
      replace = newN+1 !state(i) = 0
    END IF

    CALL rebuildNetwork(i, net, replace, state, rule)
     !PRINT *,'i',i,' ',replace
     !CALL printNetwerk(state, net, rule)
    rebuilded = .TRUE.
  END IF
  i = i + 1
END DO
 !PRINT *,'',rebuilded

i = 1
DO WHILE (i<=newN)

  ! is het eerste element gelijk aan de stabiele waarde
  IF ( ( net(i,1)==(newN+1) ) .OR. ( net(i,1)==(newN+2) ) ) THEN
    ! uit rekenen de state
    input = ISHFT(0, 1) + state(net(i,1))
    IF ( (IAND(rule(i), ISHFT(1,input))) /= 0 ) THEN
      answer1 = newN+2 !state(i) = 1
    ELSE
      answer1 = newN+1 !state(i) = 0
    END IF

    input = ISHFT(1, 1) + state(net(i,1)) ! anders
    IF ( (IAND(rule(i), ISHFT(1,input))) /= 0 ) THEN
      answer2 = newN+2 !state(i) = 1
    ELSE
      answer2 = newN+1 !state(i) = 0
    END IF

    ! als beide antwoorden het gelijke aantwoord geven
    ! dan is deze ook stabiel
    IF (answer1==answer2) THEN
      CALL rebuildNetwork(i, net, answer1, state, rule)
       !PRINT *,'nieuw i',i,'',answer1,'net1',net(i,1)
       !CALL printNetwerk(state, net, rule)
      rebuilded = .TRUE.
```

```
          END IF
        END IF
        IF (i<=newN) THEN
          IF ( ( net(i,2)==(newN+1) ) .OR. ( net(i,2)==(newN+2) ) ) THEN
            ! uit rekenen de state
            input = ISHFT(state(net(i,2)), 1) + 0 ! anders
            IF ( (IAND(rule(i), ISHFT(1,input))) /= 0 ) THEN
              answer1 = newN+2 !state(i) = 1
            ELSE
              answer1 = newN+1 !state(i) = 0
            END IF

            input = ISHFT(state(net(i,2)), 1) + 1 ! anders
            IF ( (IAND(rule(i), ISHFT(1,input))) /= 0 ) THEN
              answer2 = newN+2 !state(i) = 1
            ELSE
              answer2 = newN+1 !state(i) = 0
            END IF

            ! als beide antwoorden het gelijke aantwoord geven
            ! dan is deze ook stabiel
            IF (answer1==answer2) THEN
              CALL rebuildNetwork(i, net, answer1, state, rule)
               !PRINT *,'nieuw i',i,'',answer1,'net2',net(i,2)
               !CALL printNetwerk(state, net, rule)
              rebuilded = .TRUE.
            END IF
          END IF
        END IF
        i = i + 1
      END DO
       !PRINT *,'',rebuilded
    END DO

     !PRINT *,'N', N ,' newN', newN
END

SUBROUTINE rebuildNetwork(i, net, answer1, state, rule)
  USE const
IMPLICIT NONE
  INTEGER :: i, net(N,K), answer1, state(Nplus2), rule(N)
  INTEGER :: j, m
  ! vervangen
  DO j=1,newN
    DO m=1,K
      IF (net(j,m) == i) THEN
        net(j,m) = answer1
```

```
      END IF
    END DO
  END DO

  ! opschuiven
  newN = newN - 1
  DO j=i,newN
    state(j) = state(j+1)
    net(j,1) = net(j+1,1)
    net(j,2) = net(j+1,2)
    rule(j) = rule(j+1)
  END DO
  state(newN+1) = 0
  state(newN+2) = 1

  DO j=1,N
    DO m=1,K
      IF (net(j,m) > i) THEN
        net(j,m) = net(j,m) - 1
      END IF
    END DO
  END DO
END

SUBROUTINE printNetwerk(state, net, rule)
  USE const
IMPLICIT NONE
  INTEGER :: state(Nplus2), net(N,K), rule(N)
  INTEGER :: i

  DO i=1,newN
    PRINT *,i,'state', state(i), ' net ',net(i,1), net(i,2),' rule',rule(i)
  END DO
  PRINT *,(newN+1),'state', state(newN+1)
  PRINT *,(newN+2),'state', state(newN+2)
END
```