

Optimalisatie van het Bewijs ‘Sorteernetwerken van Optimale Grootte bij 9 Kanalen’

Mathias Dekempeneer

Bachelor Informatica

Katholieke Universiteit Leuven

mathias.dekempeneer@student.kuleuven.be

Vincent Derkinderen

Bachelor Informatica

Katholieke Universiteit Leuven

vincent.derkinderen@student.kuleuven.be

Abstract

Een *sorteernetwerk* is een netwerk bestaande uit *compare-and-swap* elementen. Deze netwerken kunnen gebruikt worden als algoritme om data van een vaste grootte te sorteren. Om een zo efficiënt (en goedkoop) mogelijk netwerk te bekomen wordt er gezocht naar een netwerk met zo weinig mogelijk elementen. De optimale hoeveelheid is reeds bewezen voor invoerdata tot en met grootte 9 en 10. Het algoritme voor het bewijs van invoer grootte 9 [Codish *et al.*, 2014] gebruikt een genereer- en snoei-methode dewelke deze paper zal reproduceren en optimaliseren. Voor een bewijs van invoer grootte 9 wordt de uitvoeringstijd gereduceerd van 305 uur naar 3 uur en 26 minuten en zet zo een stap dichters naar een uitvoering voor 10 en 11 kanalen.

1 Introductie

Comparatornetwerken bestaan uit zowel kanalen als comparatoren. De kanalen dienen voor invoer van data en de comparatoren, dewelke elk twee kanalen verbinden, zorgen voor de operaties op deze data. Een comparator zal namelijk de data verkregen van de twee verbonden kanalen vergelijken en gesorteerd terugplaatsen op deze twee kanalen. Dit zorgt ervoor dat een datasequentie ingevoerd bij een comparatornetwerk er partieel gesorteerd zal uitkomen. Sorteernetwerken daarentegen zijn comparatornetwerken waarvoor geldt dat bij elke mogelijke invoer de uitvoer een volledig gesorteerde sequentie is. Wanneer twee of meerdere opeenvolgende comparatoren geen gemeenschappelijk kanaal hebben, worden deze aanschouwd als een parallele laag.

Voor sorteernetwerken is er zowel onderzoek naar optimale diepte als optimale grootte. Een sorteernetwerk van optimale diepte houdt in dat er geen sorteernetwerk bestaat met even veel kanalen maar met minder parallele lagen. Dit betekent dat, mits elke comparator in een parallele laag parallel uitgevoerd wordt, men het snelst mogelijke netwerk bekomt. Een sorteernetwerk van optimale grootte daarentegen houdt in dat er geen sorteernetwerk bestaat met even veel kanalen maar met minder comparatoren. Mochten er geen comparatoren parallel uitgevoerd worden, bekomt men hierdoor het snelste en goedkoopste netwerk. Het onderzoek in deze paper spitst zich toe op het bewijzen van optimale grootte.

In 1966 bewees het werk van Floyd en Knuth de optimale groottes voor $1 \leq n \leq 8$ [Floyd and Knuth, 1973]. Voor optimale diepte vond Parberry een bewijs voor $n = 9$ en $n = 10$ waarna in 2014 Bundala en Závodný er vonden voor $11 \leq n \leq 16$ [Parberry, 1989; Bundala and Zavodny, 2013]. Het volgende kleinste open probleem bij optimale diepte was dan het bewijs voor 17 kanalen, bewezen door de onderzoeksgroep van onder meer Codish [Codish *et al.*, 2015]. Voor optimale grootte was dit voor 9 kanalen [Codish *et al.*, 2014]. Bij het bewijs voor 9 kanalen werd ook indirect het bewijs gegeven voor 10 kanalen. Bij dit bewijs maakten ze gebruik van zowel een genereer- en snoei-aanpak als een SAT-aanpak. Dit onderzoek bouwt verder op het voorgaande en tracht dichters bij een bewijs voor 11 kanalen te komen. Enkel gebruik makend van de genereer- en snoei-aanpak zal voor 9 kanalen de optimale grootte bewezen worden in 3 uur en 26 minuten op één node bestaande uit twee 12-core “Haswell” Xeon E5-2680v3 processoren (2.5GHz, 30MB level 3 cache met 64GB RAM) op de rekeninfrastructuur van het Vlaamse Supercomputer Centrum. Dit is een verbetering van 194 uur ten opzichte van de SAT-aanpak en 302 uur ten opzichte van de genereer- en snoei-aanpak. De resultaten van dit eerder werk zijn beide bekomen door een parallele uitvoer op 144 Intel E8400 cores (3.0GHz) met elk 2 threads.

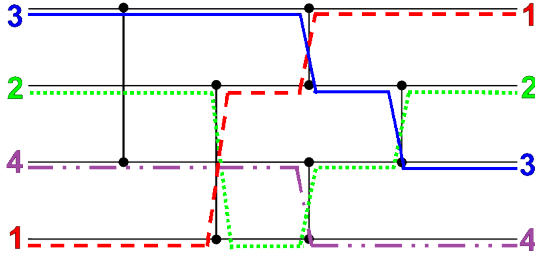
De volgende sectie licht de relevante concepten en het probleem toe. Vervolgens wordt in sectie 3 de voorgestelde oplossing beschreven, deze omvat onder meer de genereer- en snoei-aanpak, de representatie van een comparatornetwerk en de parallelisatie. Tenslotte wordt er geëindigd met een evaluatie in sectie 4.

2 Probleemstelling

Een *comparatornetwerk* C_k^n bestaat uit n kanalen en k comparatoren. Een comparator (i, j) verbindt twee verschillende kanalen i en j waarbij $0 < i < j \leq n$. We nemen x_l^m als waarde op kanaal m net voor comparator l . Deze waarde is een element uit een totaal geordende set en is deel van de oorspronkelijke invoer. De l^{de} comparator vergelijkt de huidige waarden van beide kanalen en plaatst de kleinste waarde op kanaal i en de grootste waarde op kanaal j zodat $x_{l+1}^i = \min(x_l^i, x_l^j)$ en $x_{l+1}^j = \max(x_l^i, x_l^j)$. De uitvoer van een comparatornetwerk verwijst naar de partieel geordende vector $\vec{x} = \{x_{k+1}^1 \dots x_{k+1}^n\}$. De invoer wordt voorgesteld

door $\vec{x} = \{x_1^1 \dots x_1^n\}$.

Een *sorteernetwerk* is een comparatornetwerk met als eigenschap dat de uitvoer gesorteerd is ongeacht de invoer. De grootte van een comparatornetwerk verwijst naar het aantal comparatoren. Een sorteernetwerk C_k^n van optimale grootte houdt dan in dat er geen ander sorteernetwerk C_l^n bestaat waarbij $l < k$. Figuur 1 is een voorbeeld van een sorteernetwerk waarop ook de werking gedemonstreerd wordt. Deze figuur toont ook twee parallelle lagen, bestaande uit enerzijds de eerste en de tweede comparator en anderzijds uit de derde en de vierde. Deze zijn respectievelijk $(1, 3)$, $(2, 4)$ en $(1, 2)$, $(3, 4)$. De parallelle lagen bestaan uit comparatoren die geen kanaal gemeenschappelijk hebben en waarbij de volgorde van uitvoer omgewisseld kan worden.



Figuur 1: Voorbeeld van een sorteernetwerk bestaande uit 4 kanalen en 5 comparatoren.

Om te onderzoeken of een comparatornetwerk een sorteernetwerk is, kunnen we gebruik maken van het *nul - één principe*. Dit principe, zoals beschreven volgens Knuth [Knuth, 1973], stelt dat om na te gaan of een comparatornetwerk met n kanalen een sorteernetwerk is, er enkel nagegaan moet worden dat alle 2^n mogelijke sequenties van n nullen en enen gesorteerd worden. De optimale grootte van een sorteernetwerk met n kanalen is reeds bewezen tot en met $n \leq 10$ (Tabel 1 [Codish et al., 2014]).

n	6	7	8	9	10	11	12
bovengrens	12	16	19	25	29	35	39
ondergrens	12	16	19	25	29	33	37

Tabel 1: Minimaal aantal comparatoren bij $6 \leq n \leq 12$ kanalen.

Voor $n > 10$ zijn er bovengrenzen gekend door zowel concrete voorbeelden als de systematische constructie van Batcher [Batcher, 1968]. De ondergrenzen werden gevonden door zowel bewijzen als lemma 1 [Voorhis, 1972].

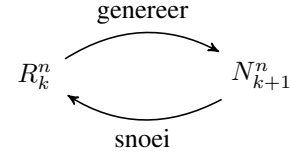
Lemma 1. $S(n+1) \geq S(n) + \lceil \log_2(n) \rceil, \forall n \geq 1$ waarbij geldt dat $S(n)$ gelijk is aan het aantal comparatoren in een sorteernetwerk van optimale grootte met n kanalen.

3 Voorgestelde oplossing

Om te bewijzen dat een sorteernetwerk C_k^n een sorteernetwerk is van optimale grootte, moeten we bewijzen dat er geen sorteernetwerk C_{k-1}^n bestaat. Aangezien n kanalen zorgen voor $\frac{n(n-1)}{2}$ verschillende comparatoren kunnen er

$\binom{n(n-1)/2}{k}$ verschillende netwerken gevormd worden met k comparatoren. Voor 9 kanalen en 24 comparatoren betekent dit 2.245×10^{37} verschillende netwerken. Deze grote hoeveelheid maakt het overlopen van alle netwerken niet aantrekkelijk. Om dit aantal te reduceren zullen we gebruik maken van symmetrieën waardoor we bepaalde netwerken reeds kunnen verwijderen bij het aanmaken.

We gebruiken de *genereer- en snoei-methode* zoals beschreven door Codish et al. (sectie 3, [Codish et al., 2014]). Deze methode heeft een cyclisch verloop waarbij men bij elke cyclus de set R_k^n uitbreidt naar N_{k+1}^n om vervolgens te snoeien en de set R_{k+1}^n te bekomen (Figuur 2). Specifiek



Figuur 2: Genereer- en snoei-principe

zullen we vertrekken van een netwerk zonder comparatoren om te eindigen bij R_k^n bestaande uit één sorteernetwerk van optimale grootte. Bij de genereer-stap zullen we aan elk netwerk van R_k^n alle mogelijke comparatoren toevoegen zodat $|N_{k+1}^n| = |R_k^n| \times \frac{n(n-1)}{2}$. Bij de snoei-stap zullen we dan netwerken verwijderen volgens het subsumes principe beschreven in definitie 1.

Definitie 1 (Subsumes). We zeggen “Comparator netwerk $C_{k,a}^n$ subsumes comparator netwerk $C_{k,b}^n$ ” wanneer een permutatie π bestaat zodat $\pi(\text{outputs}(C_a)) \subseteq \text{outputs}(C_b)$. Dit wordt genoteerd als $C_a \preceq C_b$ om aan te duiden dat er een permutatie π bestaat zodat $C_a \leq_\pi C_b$.

Lemma 2. Wanneer voor comparator netwerk $C_{k,a}^n, C_{k,b}^n$ en C geldt dat $C_a \preceq C_b$ en er bestaat een sorteernetwerk gevormd door de concatenatie van C_b en C van grootte m dan bestaat er ook een sorteernetwerk gevormd door de concatenatie van C_a en C van grootte m .

Concreet kunnen we de definitie van subsumes en lemma 2 beschreven door Codish et al. gebruiken om in te zien dat we netwerken die gesubsumed worden door andere netwerken kunnen verwijderen [Codish et al., 2014]. Wanneer een set van netwerken een sorteernetwerk bevat, zal het snoeien van deze set resulteren in het bekomen van het sorteernetwerk. Dit gegeven kan gebruikt worden om de eindigheid van het algoritme in te zien.

Om na te gaan dat er een permutatie π bestaat zodat $\pi(\text{outputs}(C_a)) \subseteq \text{outputs}(C_b)$, en dus $C_a \preceq C_b$, zouden we alle permutaties kunnen overlopen. Om deze kostelijke bewerkingen te vermijden en te versnellen, zullen we extra methoden moeten invoeren om sneller beslissingen te maken over het “subsumen van een ander netwerk”. Deze beslissingen kunnen zowel tijdens de genereer-stap als de snoei-stap plaats vinden.

In de volgende subsecties zal de voorgestelde oplossing specifiekere beschreven worden. In subsectie 3.1 wordt beschreven hoe efficiënt de informatie van een comparatornet-

werk kan worden opgeslagen. Vervolgens zal in subsectie 3.2 en 3.3 zowel de opbouw van de genereer- en snoei-methode als de extra beslismethoden toegelicht worden. Bijkomend zullen we een manier geven in subsectie 3.4 om het proces te paralleliseren zonder al te veel locks. Tenslotte wordt het geheugengebruik van de implementatie besproken en wordt er een manier aangeboden om deze nog verder te beperken.

3.1 Representatie van comparatornetwerken

Bij de representatie van comparatornetwerken moeten we rekening houden met het geheugengebruik en de mogelijkheid om efficiënte bewerkingen te kunnen uitvoeren. Concreet zullen we comparatoren voorstellen door een sequentie van bits waarbij twee bits op één staan. Bijvoorbeeld [010001] stelt de comparator (1, 5) voor bij een netwerk van 6 kanalen. Doordat kanaal 1 rechts geplaatst wordt, komt elk kanaal beter overeen met zijn bitshift¹. Om de hoeveelheid overbodige bits te beperken, zullen we bij de Java implementatie gebruik maken van shorts². Dit is voldoende voor een bewijs tot en met 16 kanalen. Buiten de comparatoren worden ook de mogelijke uitvoer van het netwerk bijgehouden. Aangezien het nul-één-principe stelt dat enkel sequenties van nullen en enen getest moeten worden, kunnen we de uitvoer opdelen per aantal enen. Voor een netwerk zonder comparatoren betekent dit 2^n mogelijke uitvoer sequenties. Dit aantal zal echter dalen naarmate er meer en meer comparatoren worden toegevoegd om uiteindelijk te eindigen met slechts n uitvoer sequenties. Door de gekozen representatie van invoer en comparatoren kunnen we Code 1 gebruiken om de uitvoer van een comparator te bepalen gegeven een bepaalde invoer.

Code 1: swapCompare

```
/**
 * input - De invoer voor de comparator.
 * comp - De comparator (bv. 00101)
 * return - Het resultaat bekomen door de bits
 *         van de input om te wisselen afhankelijk
 *         van de comparator.
 */
short swapCompare(short input, short comp) {
    int channel1 = 31 - Integer.numberOfLeadingZeros(comp);
    int channel2 = Integer.numberOfTrailingZeros(comp);

    int firstBit = (input >> channel1) & 1;
    int secondBit = (input >> channel2) & 1;

    return (firstBit <= secondBit) ? input :
        (input ^ comp);
}
```

In de Java implementatie kiezen we er voor om een comparatornetwerk voor te stellen door een tweedimensionale array van shorts, short[], en laten we de rij van n nullen en n enen weg. Een voorbeeld van zo een representatie staat in tabel 2.

¹Bijvoorbeeld een bitshift naar rechts, \gg , zal achteraan een aantal nullen verwijderen.

²In Java bestaat een short uit 16 bits.

Comparators	[0011]	[1010]	
Uitvoer één 1	[0001]	[0100]	[0010]
Uitvoer twee 1'en	[0011]	[0101]	[0110]
Uitvoer drie 1'en	[0111]	[1011]	

Tabel 2: Representatie C_2^4 : (1, 2)(2, 4)

3.2 Genereren

Bij de genereer-stap itereren we over de set R_k^n en voegen we bij elk netwerk alle mogelijke comparatoren toe. Een overbodige comparator voor een bepaald comparatornetwerk is een comparator die, wanneer toegevoegd aan het comparatornetwerk, geen wijziging veroorzaakt in de mogelijke uitvoer. Wanneer een overbodige comparator wordt toegevoegd aan een comparatornetwerk, zal deze gesubsumed worden door een uitbreiding van dat netwerk met een niet overbodige comparator. Bijgevolg kunnen we deze meteen uit de set N_{k+1}^n verwijderen. Om te beslissen of een comparator al dan niet overbodig is, moeten we voor elke mogelijke uitvoer nagaan of deze een wijziging teweegbrengt. We kunnen dit sneller laten verlopen door eerst te kijken of de comparator al dan niet identiek is met de vorige comparator in het netwerk.

Wanneer twee netwerken, op de volgorde van hun parallelle comparatoren na, gelijk zijn, zoals in Figuur 3a en 3b, zullen deze elkaar subsumen en zal één van de twee verwijderd worden. Dit kan opgevangen worden in de generatie-stap om zo extra werk in de snoei-stap te vermijden. Bij het toevoegen van een nieuwe comparator (a, b) moet er dan gecontroleerd worden of deze comparator een kanaal gemeenschappelijk heeft met de vorige (Code 2).

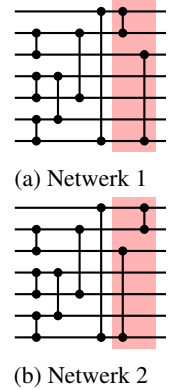
Code 2: Test op parallelle comparatoren.

```
(a,b) & vorigeComp != 0
```

Wanneer dit niet het geval is en het dus parallelle comparatoren zijn, kunnen we bijvoorbeeld kiezen om het netwerk weg te gooien waarbij de nieuwe comparator kleiner is dan de vorige.

Tenslotte kunnen we, na het toevoegen van de comparator, de nieuwe uitvoer berekenen door de huidig bijgehouden uitvoer als invoer te gebruiken voor de nieuwe comparator. Om de efficiëntie te verhogen kunnen we bij de omzetting van de uitvoer eerder verkregen informatie gebruiken. Een comparator is namelijk niet overbodig wanneer er een uitvoer bestaat met een s aantal enen waarvoor geldt dat de comparator deze uitvoer wijzigt. Deze informatie kan gebruikt worden om bij de omzetting slechts te beginnen bij uitvoer met het aantal enen gelijk aan s . Een algemene structuur van deze code wordt geïllustreerd in Code 3.

In het ergste geval moeten er aan alle elementen in de set R_k^n alle mogelijke comparatoren worden toegevoegd. Dit be-



Figuur 3




tekent $\mathcal{O}\left(|R_k^n| \times \frac{n(n-1)}{2}\right)$.

Code 3: Pseudocode - optimalisatie via s

```
for(short nieuweComp : comparatoren) {
  int s = vanafAantalEnenWijzigd(netwerk,
    nieuweComp);
  if(s != -1) {
    voegToeAanNetwerk(netwerk, nieuweComp, s)
  }
}
```

3.3 Snoeien

Bij de snoei-stap lopen we over de set N_{k+1}^n en verwijderen we alle comparatornetwerken die gesubsumed worden door een ander comparatornetwerk in de resterende set. Om het aflopen van alle permutaties te vermijden, en sneller te beslissen of $C_a \preceq C_b$ met C_a en C_b twee comparatornetwerken, voeren we enkele methoden in. Zo gebruiken we onder meer lemma 3 en lemma 4, beschreven in TWENTY-FIVE COMPARATORS IS OPTIMAL WHEN SORTING NINE INPUTS (AND TWENTY-NINE FOR TEN) [Codish *et al.*, 2014].

Aantal enen	1	2	3	4	5
C_1 	00001 00010	00011 00110 01010	00111 01011 01110	01111 11110	11111
C_2 	00001 00010	00011 00101 00110 01001	00111 01011 01101	01111 10111	11111
C_3 	00001 00010 00100	00011 00101 00110	00111 01110 10110	01111 10111 11110	11111

De waarden in een kolom stellen alle mogelijke posities voor die op die plaats kunnen voorkomen. Wanneer we tabel 3 gebruiken om de mogelijke permutaties weer te geven dan zullen we permutatie 4321 en 1324 bekomen, waarbij 4321 een eenheidspermutatie zal voorstellen. Bij het begin van het algoritme zullen we starten met tabel 4, deze laat alle $n!$ permutaties toe. Hierna gebruiken we lemma 5 om posities uit de kolommen te verwijderen.

1	2	2	1
2	3		4
4			

Tabel 3: Een voorbeeld van een permutatietabel voor 4 kanalen.

1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4

Tabel 4: Een permutatietabel in het begin van het algoritme voor 4 kanalen.

We weten namelijk dat als

$$\pi(\text{outputs}(C_a)) \subseteq \text{outputs}(C_b)$$

er bij de gepermuteerde uitvoer enkel een 1 kan komen op de plaats waar dit bij C_b ook het geval is. Op de plaats waar C_b een 0 heeft, kunnen dus enkel de posities komen waar C_a een 0 heeft. Nemen we bijvoorbeeld $w(C_a, 1, 1) = 0101$ en $w(C_b, 1, 1) = 0110$ dan kunnen we tabel 4 reduceren tot tabel 5. We kunnen voor $w(C_a, x, k)$ deze methode doortrekken voor elke $1 \leq k \leq n$ en voor zowel $x = 0$ als $x = 1$. Wanneer we elke kolom bijhouden door een bit representatie kunnen we gemakkelijk de doorsnede van de mogelijke posities nemen na elke berekening voor een bepaalde k en x door middel van de $\&$ -operatie.

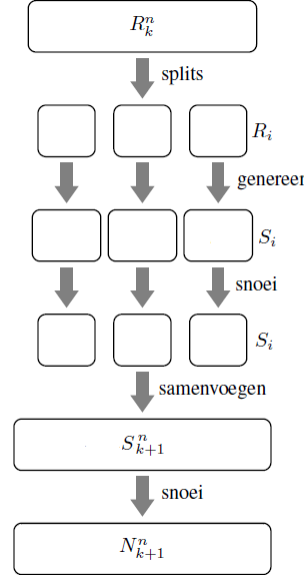
2	1	1	2
4	2	2	4
	3	3	
	4	4	

Tabel 5: Een permutatietabel voor 4 kanalen.

Wanneer tijdens het algoritme een kolom leeg zou komen te staan, kunnen we het algoritme stopzetten. Dit betekent namelijk dat er geen enkele permutatie bestaat die niet door lemma 5 wordt afgekeurd. Mocht op het einde een kolom één element hebben, mogen we dit element uit alle andere kolommen verwijderen. We kunnen nadien ook nog nagaan of alle elementen minstens éénmaal voorkomen in de hele tabel. Tot slot gebruiken we de overblijvende permutatietabel om onze mogelijke permutaties, die aan lemma 5 voldoen, na te gaan. Dit kan onder andere door middel van een recursieve methode.

In het ergste geval moeten we voor elk element in N_{k+1}^n subsumes nagaan met elk ander element in N_{k+1}^n . Nemen we voor de kost van een subsumes-test $f(n)$ dan betekent dit $\mathcal{O}(|N_{k+1}^n| \times (|N_{k+1}^n| - 1) \times f(n))$.

3.4 Parallellisatie



Figuur 6: Opbouw parallelle genereer & snoei

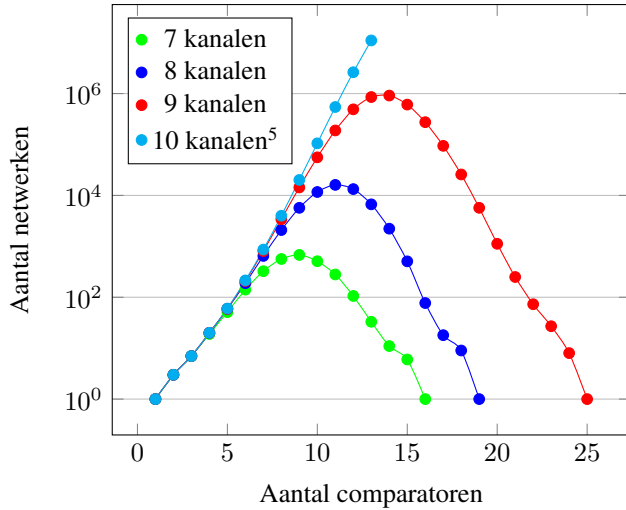
Om het algoritme te laten functioneren met meerdere processoren, zullen we enkele aanpassingen doorvoeren. Bij de overgang van R_k^n naar N_{k+1}^n zal elke thread een aantal netwerken uit R_k^n nemen, in ons geval 256, hierop de genereer-stap uitvoeren en vervolgens binnen de resulterende set de snoei-stap uitvoeren. Op dat moment beschikt elke thread over een set van netwerken met $k + 1$ comparatoren waarop de snoei-stap nog moet worden uitgevoerd ten opzichte van alle andere sets. Elke thread zal vervolgens zijn set in een gedeelde lijst S_{k+1}^n in het centraal geheugen plaatsen. Om deze operatie zo efficiënt mogelijk te maken, vermijden we zowel locks als het moeten vergroten van de lijst. Daarom zullen we in het begin van de cyclus zorgen dat deze lijst groot genoeg is en gebruik maken van een variabele die bijhoudt op welke index een volgend netwerk moet worden bijgevoegd. In onze Java implementatie zullen we voor deze variabele een `AtomicInteger` gebruiken, deze variabele garandeert een atomische `getAndIncrement(int)` functie. Een thread kan deze functie gebruiken om voldoende plaats in de lijst op te eisen voor zijn set door de grootte van zijn set mee te geven als parameter.

Na het toevoegen aan de gedeelde lijst volgt de snoei-stap. Hier zal elke thread subsumes moeten nagaan tussen alle netwerken in zijn set en elk ander reeds toegevoegd netwerk. Door het (hopelijk) vele verwijderen van netwerken ontstaan er veel opeenvolgende lege plaatsen in de lijst⁴. Hierdoor zal het algoritme vaak overbodig het netwerk opvragen. Om dit aantal, dat groter wordt naarmate het aantal kanalen stijgt, te verminderen, introduceren we een manier om deze opeenvolgende lege indices over te slaan. Telkens wanneer een lege index wordt gedetecteerd door een thread zal deze het aantal opeenvolgende lege indices tellen en dit aantal opslaan op de eerste lege plaats in deze reeks. Dit getal stelt dus het aantal opeenvolgende lege plaatsen voor na die index. Wanneer een thread dan dit getal tegenkomt, weten we exact hoeveel indices de thread kan overslaan.

⁴Bij de Java implementatie zullen we verwijderen via “= null”.

3.5 Geheugen

Door het grote aantal netwerken is het testen op subsumes en de efficiëntie van de datastructuren van groot belang. In Figuur 7 zien we dat dit aantal voor 9 kanalen kan oplopen tot meer dan 900000 netwerken. Bij het stijgen van het aantal kanalen wordt dan ook het geheugenbeheer des te belangrijker.



Figuur 7: Aantal resterende netwerken na het uitvoeren van genereer en snoei bij toevoegen van de k^{de} comparator.

In Java hebben we het voordeel dat we niet expliciet aan geheugenbeheer moeten doen. We gaan anderzijds wel enkele maatregelen nemen om de vereiste hoeveelheid geheugen te verlagen. Eén van de mogelijke plaatsen waar we dit kunnen doen, is bij de representatie van een netwerk. Wanneer een comparator aan een netwerk wordt toegevoegd, is het mogelijk dat een lijst van uitvoer sequenties met s enen ongewijzigd blijft. Om te vermijden dat we hierdoor meerdere malen dezelfde lijst in het geheugen hebben, zullen we een referentie doorgeven van deze lijst en slechts een nieuwe lijst gebruiken wanneer de lijst gewijzigd wordt. Aangezien bij Java een tweedimensionale lijst wordt aanzien als een lijst van referenties naar andere lijsten kan de oude referentie gemakkelijk hergebruikt worden.

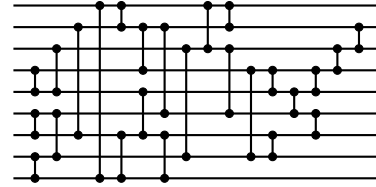
Een andere plaats is bij de parallelisatie, hier daalt de hoeveelheid geheugen doordat een thread enerzijds zal snoeien binnen zijn set alvorens de set toe te voegen aan de gedeelde lijst en anderzijds doordat de genereer- en snoei-stap door elkaar worden uitgevoerd.

Onze Java implementatie gebruikte bij de uitvoering voor 8 en 9 kanalen respectievelijk 208MB en 3951MB. Deze hoeveelheid kan verschillen naargelang de frequentie waarbij de Java Virtual Machine de *Garbage Collection* uitvoert. Voor minder dan 8 kanalen was het geheugen verbruik verwaarloosbaar klein.

⁵Voor 10 kanalen is het aantal netwerken berekend tot en met comparator 13.

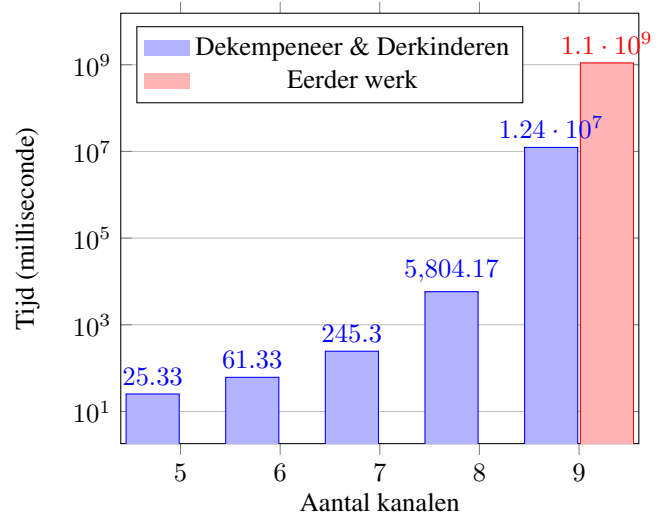
4 Evaluatie

Het beschreven algoritme, geïmplementeerd in Java, vindt voor 9 kanalen reeds na 3 uur en 25 minuten een oplossing met 25 comparatoren die gevisualiseerd wordt in Figuur 8. Dit bevestigt wat reeds geweten was [Codish *et al.*, 2014].



Figuur 8: Het gevonden sorteernetwerk van optimale grootte bij 9 kanalen met 25 comparatoren.

De tijdsmetingen voor het vinden van een sorteernetwerk van optimale grootte van 5 tot en met 9 kanalen zijn te zien op Figuur 9. Op deze figuur is ook de tijdsmeting van eerder werk te zien, ongeveer 12 dagen 17 uur en 58 minuten [Codish *et al.*, 2014]. De bekomen resultaten van dit werk zijn afkomstig van het uitvoeren op één node bestaande uit twee 12-core “Haswell” Xeon E5-2680v3 processoren (2.5GHz, 30MB level 3 cache met 64GB RAM) op de rekeninfrastructuur van het Vlaamse Supercomputer Centrum.



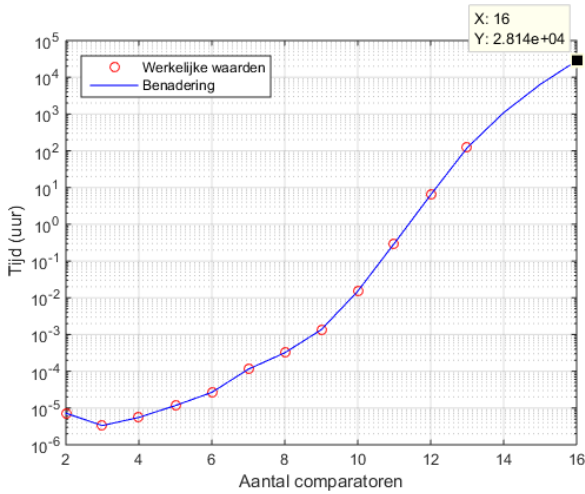
Figuur 9: Tijdsmetingen voor uitvoer bij 5 tot en met 9 kanalen.

4.1 Benadering 10 en 11 kanalen

De uitvoering van het programma voor 10 kanalen is na 299 uur⁶ stopgezet. De tussentijdse resultaten met betrekking tot de uitvoeringstijd zijn te zien in Figuur 10 en met betrekking tot het aantal netwerken zijn te zien in Figuur 7. Voor 10, en dus ook voor 11, kanalen is de Java implementatie met de gebruikte hardware onvoldoende om resultaten binnen een redelijk tijdsbestek te bekomen. Gebaseerd op

⁶12 dagen en 11 uur

Figuur 10 schatten we dat voor 10 kanalen meer dan 1500 dagen⁷ vereist zijn.



Figuur 10: Tijdsverloop 10 kanalen tot en met de 13^{de} comparator en benadering van tijdsverloop tot en met de 16^{de} comparator, de vermoedelijke piek.

4.2 Profilering

Via een profilering, zoals in Figuur 11, kunnen we de bottleneck vaststellen met als doel de uitvoeringstijd te verbeteren. In dit profiel zien we dat de snoei-methode, de prune-methode in de figuur, duidelijk de bottleneck is. In deze methode wordt er door een thread voor elk netwerk in de gedeelde lijst subsumes uitgevoerd met elk netwerk in zijn eigen lijst, zoals beschreven in sectie 3.4. Wanneer we de uitvoeringstijd willen verbeteren, kunnen we enerzijds proberen deze prune methode te voorkomen zoals bijvoorbeeld in de genereer-stap en anderzijds door deze prune-methode efficiënter te maken. Hier kan bijvoorbeeld onderzoek gedaan worden naar of men al dan niet onder een bepaalde voorwaarde netwerken van de eigen lijst kan overslaan.

Name	Self Time (CPU)	Total Time (CPU)
⌚ prune	1,925,071 ms (80%)	2,257,121 ms (17.1%)
⌚ existsAValidPerm	155,080 ms (6.4%)	376,525 ms (2.9%)
⌚ isValidPermutation	150,901 ms (6.3%)	150,901 ms (1.1%)
⌚ innerPrune	89,898 ms (3.7%)	145,503 ms (1.1%)
⌚ checkAllRelevantPermutations	70,522 ms (2.9%)	221,423 ms (1.7%)
⌚ subsumes	11,008 ms (0.5%)	387,534 ms (2.9%)
⌚ processData	3,391 ms (0.1%)	3,402 ms (0%)
⌚ generate	1,006 ms (0%)	4,997 ms (0%)
⌚ java.lang.Object.clone[native]	588 ms (0%)	588 ms (0%)

Figuur 11: Profiel van een partiële uitvoering voor 9 kanalen.

4.3 Beslissingen

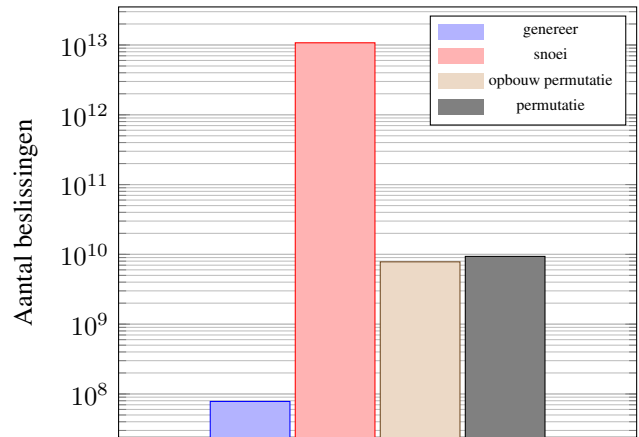
Door de genereer- en snoei-aanpak mag het totaal aantal te overlopen netwerken dan wel gedaald zijn, de hoeveelheid

⁷Berekend op basis van een polynomiaal verloop van graad 11.

beslissingen dat er genomen moeten worden blijft nog steeds hoog zoals te zien in Figuur 12. In deze figuur zien we dat de meeste beslissingen genomen worden tijdens het snoeien. Er moet echter wel een onderscheid gemaakt worden in het soort van beslissingen. De beslissingen genomen in de genereer-stap zijn beslissingen die het genereren van een netwerk voorkomen en zijn gebaseerd op redundante comparatoren. Deze beslissingen zorgen voor minder netwerken in de gedeelde lijst en dus voor zowel minder snoei-operaties als geheugen-verbruik.

De beslissingen genomen in de snoei-stap daarentegen, zijn beslissingen die enkel voorkomen dat er permutaties moeten worden opgebouwd om na te gaan of $C_a \preceq C_b$ geldt. Deze zorgen er dus niet voor dat een netwerk uit de gedeelde lijst verdwijnt, enkel dat er niet al te veel tijd gependend wordt aan het onnodig nagaan van permutaties. Hier wordt onder meer lemma 3 en lemma 4 gebruikt.

Vervolgens zijn er nog de beslissingen die genomen worden tijdens de opbouw van de mogelijke permutaties en tijdens het aflopen van deze permutaties. Tijdens de opbouw gebeurt deze beslissing wanneer er geen geschikte permutatie bestaat doordat de permutatietabel een lege kolom bevat.

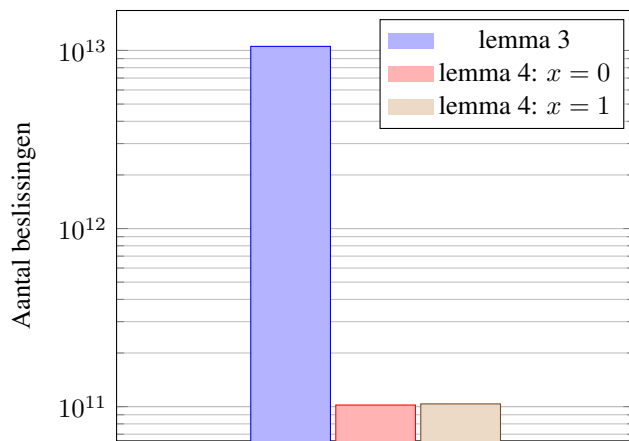


Figuur 12: Aantal beslissingen bij een uitvoer voor 9 kanalen.

In Figuur 13 wordt er specifiek gekeken naar de beslissingen in de snoei-stap. Hier zien we dat lemma 3 het grootste aantal beslissingen maakt. Dit is deels te wijten aan het feit dat deze als eerste wordt nagegaan in de implementatie. Het is dan ook belangrijk dat dit lemma efficiënt wordt geïmplementeerd. Er zou nog onderzoek gedaan kunnen worden naar of we dit lemma al dan niet kunnen gebruiken om netwerken in de gedeelde lijst over te slaan, zoals reeds vermeld in sectie 4.2. Zo zou bijvoorbeeld de set van elke thread na het intern snoeien gesorteerd kunnen worden volgens een bepaald argument alvorens deze toe te voegen aan de gedeelde lijst. Vervolgens zou men dan via dit argument kunnen beslissen om al dan niet een bepaald aantal netwerken in de gedeelde lijst over te slaan.

5 Conclusies

Onze implementatie van de genereer- en snoei-aanpak van Codish *et al* zorgde voor een bewijs voor 9 kanalen na



Figuur 13: Aantal beslissingen tijdens het snoeien bij een uitvoer voor 9 kanalen.

3 uur en 26 min, een versnelling ten opzichte van de eerdere 305 uur. Het is echter niet snel genoeg voor een bewijs van 10 kanalen, deze zou bij benadering meer dan 1500 dagen duren. Bijgevolg blijft het bewijs voor het sorteernetwerk van optimale grootte voor 11 kanalen het volgende open probleem. Er zou nog extra werk geleverd kunnen worden omtrent het efficiënt overlopen van alle comparator-netwerken alsook het bruikbaar maken voor meerdere nodes.

Erkenning

Graag willen we Professor Dr. Ir. Tom Schrijvers bedanken voor zijn begeleiding doorheen dit onderzoek.

De rekeninfrastructuur en dienstverlening gebruikt in dit werk, werd voorzien door het VSC (Vlaams Supercomputer Centrum), gefinancierd door het FWO en de Vlaamse regering - departement EWI. Bijgevolg willen we de onderzoeksgroep DTAI bedanken voor de aangeboden credits voor deze rekeninfrastructuur.

Referenties

- [Batcher, 1968] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, AFIPS '68 (Spring)*, pages 307–314, New York, NY, USA, 1968. ACM.
- [Bundala and Zavodny, 2013] Daniel Bundala and Jakub Zavodny. Optimal sorting networks. *CoRR*, abs/1310.6271, 2013.
- [Codish *et al.*, 2014] Michael Codish, Luis Cruz-Filipe, Michael Frank, and Peter Schneider-Kamp. Twenty-five comparators is optimal when sorting nine inputs (and twenty-nine for ten). Technical report, IEEE International Conference on Tools with Artificial Intelligence (ICTAI), November 2014.
- [Codish *et al.*, 2015] Michael Codish, Luís Cruz-Filipe, Thorsten Ehlers, Mike Müller, and Peter Schneider-Kamp. Sorting networks: to the end and back again. *CoRR*, abs/1507.01428, 2015.

[Floyd and Knuth, 1973] R. W. Floyd and D. E. Knuth. In *A survey of combinatorial theory*, chapter The Bose-Nelson sorting problem, pages 163–172. North-Holland, 1973.

[Knuth, 1973] D. E. Knuth. *The art of computer programming. Vol.3: Sorting and searching*. Addison-Wesley, 1973.

[Parberry, 1989] I. Parberry. A computer assisted optimal depth lower bound for sorting networks with nine inputs. In *Supercomputing, 1989. Supercomputing '89. Proceedings of the 1989 ACM/IEEE Conference on*, pages 152–161, Nov 1989.

[Voorhis, 1972] David C. Voorhis. *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*, chapter Toward a Lower Bound for Sorting Networks, pages 119–129. Springer US, Boston, MA, 1972.