Sorteernetwerken van Optimale Grootte

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

Verdergaand op eerder werk omtrent sorteernetwerken van optimale grootte wordt het algoritme beschreven door de onderzoeksgroep van onder meer Codish gereproduceerd[Codish *et al.*, 2014]. Het doel van deze reproductie bestaat eruit om bij te dragen tot een efficiëntere tijdsmeting om verder onderzoek mogelijk te maken. Net als bij TWENTY-FIVE COMPARATORS IS OPTIMAL WHEN SORTING NINE INPUTS (AND TWENTY-NINE FOR TEN) maakt de implementatie gebruik van een *genereer- en snoei-methode*. De resulterende code bewijst de optimale grootte voor 9 kanalen in 3 uur en 25 minuten en zet zo een stap dichter naar een uitvoering voor 10 en 11 kanalen.

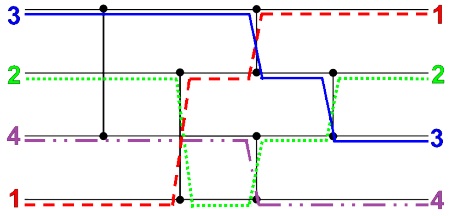
1 Introductie

Situering + bijdrage.
Sorting Network (high level), Optimal Size (high level), contributies andere papers rond deze twee, enkele getallen rond grootte orde van het probleem, wat er al geprobeerd is (SAT, generate & prune,...), hoe wij het probleem zullen aanpakken (hoe wij prunen (high level)), gebruikte hardware...

2 Probleemstelling

Een *comparator netwerk* 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. 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 comparator netwerk verwijst naar de partieel geordende vector $\vec{x} = \{x_{k+1}^1 \dots x_{k+1}^n\}$. De invoer wordt voorgesteld door $\vec{x} = \{x_0^1 \dots x_0^n\}$.

Een *sorteernetwerk* is een comparator netwerk met als eigenschap dat de uitvoer gesorteerd is ongeacht de invoer. Een sorteernetwerk C_k^n van optimale grootte houdt in dat er geen ander sorteernetwerk C_l^n bestaat waarbij $l < k$. Figuur 1 is een voorbeeld van zo een netwerk waarop ook de werking gedemonstreerd wordt. Deze figuur toont ook twee parallelle comparatoren $(1, 2)$ en $(3, 4)$, comparatoren die geen kanaal gemeenschappelijk hebben en van volgorde omgewisseld kunnen worden. Om te onderzoeken of een comparator



Figuur 1: Sorteernetwerk 4 kanalen, 5 comparatoren

netwerk een sorteernetwerk is, kunnen we gebruik maken van het *nul - één principe*. Dit principe, zoals beschreven volgens Knuth [Knuth, 1973], stelt dat wanneer een comparator netwerk met n kanalen alle 2^n mogelijke sequenties van n 0- en 1-en sorteert, het een sorteernetwerk is. De optimale grootte van een sorteernetwerk met n kanalen is reeds bewezen tot en met $n \leq 10$ (Tabel 1 [Codish *et al.*, 2014]). Voor $n > 10$

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.

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

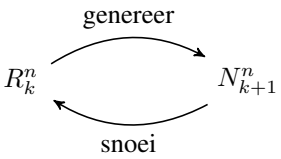
Lemma 1. $S(n+1) \geq S(n) + \lceil \log_2(n) \rceil, \forall n \geq 1$

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, dit maakt het overlopen van alle netwerken niet aantrekkelijk. Door gebruik te maken van symmetrieën willen we snoeien in het aanmaken van deze netwerken.

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 permuntatie π bestaat zodat $\pi\left(outputs\left(C_a\right)\right) \subseteq outputs\left(C_b\right)$. 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 $C_{k,a}^n, C_{k,b}^n$ geldt dat $C_a \preceq C_b$ en er bestaat een sorteernetwerk $C_b; C$[1] van grootte m dan bestaat er ook een sorteernetwerk $C_a; C'$ van grootte m.*

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

Het overlopen van alle permutaties om na te gaan of er een permutatie π bestaat zodat $\pi\left(Outputs\left(C_a\right)\right) \subseteq Outputs\left(C_b\right)$, en dus $C_a \preceq C_b$, is een kostelijke bewerking. Om deze bewerkingen te vermijden en te versnellen, zullen we extra methoden moeten invoeren om snellere beslissingen te maken over het "subsumen van een ander netwerk".

3.1 Representatie van comparator netwerken

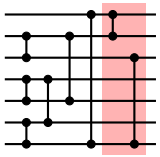
Bij de representatie van comparator netwerken moeten we rekening houden met geheugengebruik en de mogelijkheid om efficiënte bewerkingen te kunnen uitvoeren. Concreet zullen we comparatoren voorstellen door een sequentie van bits, waarbij twee bits op één staan. Bijvoorbeeld $[010010]$ stelt de comparator $(2, 5)$ voor bij een netwerk van 6 kanalen. Om de hoeveelheid overbodige bits te beperken, zullen we bij de `Java` implementatie gebruik maken van shorts[2]. Buiten de comparatoren worden ook de outputs van het netwerk bijgehouden, opgedeeld per aantal 1'en. In de `Java` implementatie kiezen we er voor om een comparator netwerk voor te stellen door een tweedimensionale array van shorts, short[][], en laten we de rij van n 1'en weg. Een voorbeeld van zo een representatie staat in tabel 2.

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

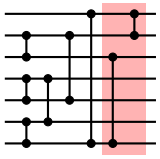
Tabel 2: Representatie C_2^4: $(1, 2)(2, 4)$

3.2 Genereren

Bij de genereer-stap lopen we over de set R_k^n en voegen we bij elk netwerk alle mogelijke comparatoren toe. Aangezien een netwerk dat wordt uitgebreid met een overbodige comparator, één waarbij de outputs ongewijzigd blijven, gesubsumed zal worden door een uitbreiding van dat netwerk met een niet overbodige comparator, kunnen we deze meteen verwijderen uit de set N_{k+1}^n. Alvorens deze beslissing te maken door alle outputs te overlopen, kunnen we ook eerst kijken of de comparator gelijk is aan de vorige in het netwerk. Wanneer 2 netwerken op de volgorde van hun parallelle comparatoren na gelijk zijn, zoals in figuur 3a en 3b, zullen deze elkaar subsumen en één van de twee verwijderd worden. Dit kan reeds bij de generatie-stap gemakkelijk opgevangen worden door bij het toevoegen van een nieuwe comparator x na te gaan of x een kanaal gemeenschappelijk heeft met de vorige comparator (Code 1).



(a) Netwerk 1



(b) Netwerk 2

Figuur 3

Code 1: Test op parallelle comparatoren

```
x & 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 comparator.

Tenslotte, na het toevoegen van de comparator, kunnen we de nieuwe outputs berekenen door de huidig bijgehouden outputs te gebruiken als invoer voor de nieuwe comparator.

[1] $C_b; C$ is een concatenatie van netwerk C_b en C.

[2] In `Java` bestaat een short uit 16 bits.

3.3 Snoeien

Bij de snoei-stap lopen we over de set N_{k+1}^n en verwijderen we alle netwerken die gesubsumed worden door een ander netwerk 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 comparator netwerken, voeren we enkele methoden in. Zo gebruiken we onder meer lemma 3, beschreven in de paper van Codish *et al.*[Codish *et al.*, 2014]. Bij 9 kanalen wordt de methode 1.07666×10^{13} keer uitgevoerd waarbij 1.05438×10^{13} keer een beslissing genomen wordt.

Lemma 3. *Wanneer het aantal outputs bij C_a met x 1'en ($1 \leq x \leq n$) groter is dan bij C_b weten we dat $C_a \not\preceq C_b$ met C_a en C_b twee comparator netwerken.*

Voor lemma 4 van Codish ([Codish *et al.*, 2014]) introduceren we extra informatie over het comparator netwerk, namelijk $w(C_a, x, k)$ waarbij $x \in \{0,1\}$ en $0 \leq k \leq n$. Dit representeert de set van posities i waarvoor er een output bestaat in C_a met k 1'en waarvoor geldt dat op de i^{de} positie van deze output een x voorkomt. Om efficiënt operaties te kunnen uitvoeren zullen we de posities voorstellen door middel van een bit representatie. Zo zal bijvoorbeeld $w(C_a, 1, 2) = 0110$ inhouden dat er bij de outputs met twee 1'en minstens één output bestaat met een 1 op de 2^{de} positie, één met een 1 op de $3de$ positie en geen enkel met een 1 op positie 1 of 4. Deze informatie voegen we bij elk netwerk toe in de vorm van een array van shorts, w. Elk kanaal k van het netwerk C_a vereist dan 4 opeenvolgende indices in w, zoals te zien in tabel 3. Deze informatie slaan we voor elk kanaal k op vanaf index[3] $(k-1) \times 4$.

$w(C_a, 0, 1)$	$\lvert w(C_a, 0, 1) \rvert$	$w(C_a, 1, 1)$	$\lvert w(C_a, 1, 1) \rvert$

Tabel 3: De inhoud van w op indices $0-3$ voor kanaal 1.

Lemma 4. *Wanneer voor een comparator netwerk C_a en C_b met n kanalen geldt dat $\lvert w(C_a, x, k) \rvert > \lvert w(C_b, x, k) \rvert$ voor $x \in \{0,1\}$ en $0 \leq k \leq n$ dan $C_a \not\preceq C_b$.*

De methode van lemma 4 wordt bij 9 kanalen 2.22803×10^{11} keer uitgevoerd waarbij 2.05631×10^{11} keer een beslissing genomen wordt.

Tenslotte komen we aan het nagaan van de permutaties, een naïeve methode zou zijn om alle $n!$ permutaties te overlopen. In de plaats daarvan zullen we enkel permutaties afgaan die voldoen aan lemma 5.

Lemma 5. $C_a \preceq C_b \Rightarrow \pi(Outputs(C_a)) \subseteq Outputs(C_b) \Rightarrow \pi(w(C_a, x, k)) \subseteq w(C_b, x, k), \forall x \in \{0,1\}, \forall k \in \{1..n\}$.

Om de mogelijke permutaties bij te houden zullen we gebruik maken van een voorstelling die te zien is in tabel 4. De waarden in een kolom stellen alle mogelijke posities voor die op die plaats kunnen voorkomen. Wanneer we tabel 4 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 5, waarna we lemma 5 gebruiken om posities te verwijderen.

[3] In `Java` begint een array met index 0.

1	2	2	1
2	3		
4			

Tabel 4: 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 5: Een permutatietabel in het begin van het algoritme voor 4 kanalen.

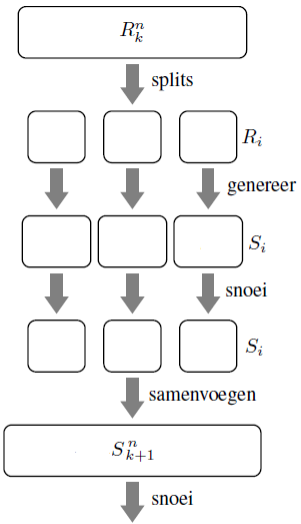
We weten namelijk dat als $\pi(outputs(C_a)) \subseteq outputs(C_b)$ er bij de gepermuteerde outputs enkel een 1 kan komen op de plaats waar dit bij C_b ook het geval is. Op de plaats waar C_b een 0 heeft, kunnen dus enkel de posities komen waar C_a een 0 heeft. Nemen we bijvoorbeeld $w(C_a, 1, 1) = 0101$ en $w(C_b, 1, 1) = 0111$ dan kunnen we tabel 5 reduceren tot tabel 6. 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	1
4	2	2	2
	3	3	3
	4	4	4

Tabel 6: 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 1 element hebben, mogen we dit element uit alle andere kolommen verwijderen. We kunnen nadien ook 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.

3.4 Parallellisatie



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 genereerstap uitvoeren en vervolgens binnen de resterende 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 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 lijst[4]. 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 de thread het aantal opeenvolgende lege indices tellen en dit aantal opslaan op de eerste lege plaats in deze reeks. Wanneer een thread dit getal tegenkomt, kan hij het opgeslagen aantal indices overslaan.

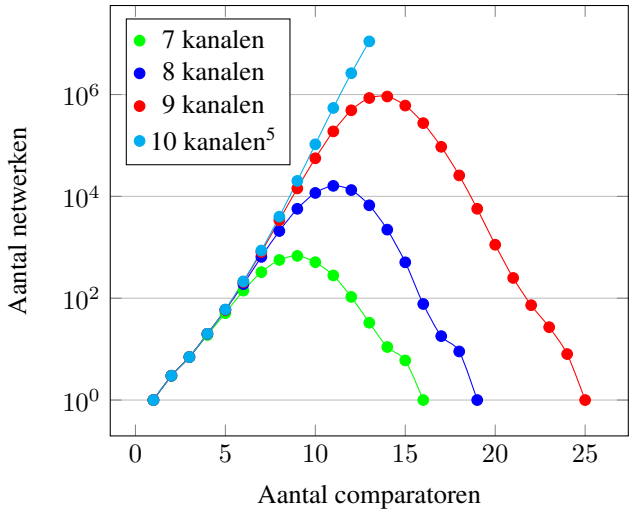
3.5 Geheugen

Door het grote aantal netwerken is het testen op subsumes en de efficiëntie van de datastructuren van groot belang. In figuur 5 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.

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 outputs met x 1'en 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 2-dimensionale lijst wordt aanzien als een lijst van referenties naar andere lijsten kan de oude referentie gemakkelijk herbruikt worden.

Een andere plaats is bij de parallellisatie, hier daalt de hoeveelheid geheugen doordat een thread enerzijds zal snoeien binnen zijn set alvorens de set toe te voegen. 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.



Figuur 5: Aantal resterende netwerken na het uitvoeren van genereer en snoei bij toevoegen van de k^{de} comparator.

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 7. Dit bevestigt wat reeds geweten was door Codish *et al.* De tijdsmetingen voor het vinden van een sorteernetwerk van optimale grootte van 5 tot en met 9 kanalen zijn te zien op figuur 6. 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.

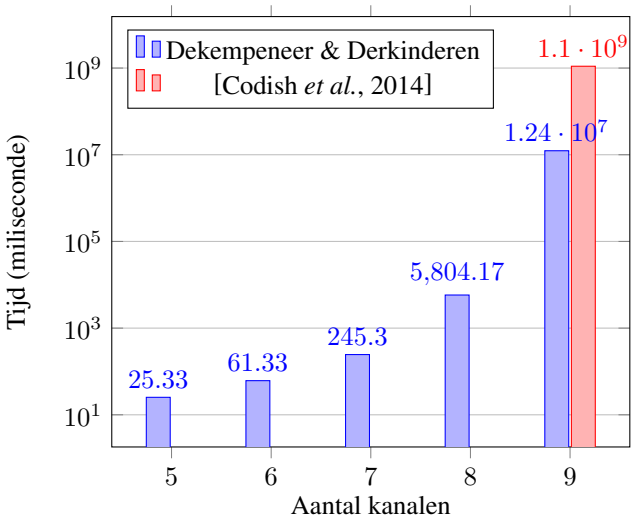
4.1 Benadering 10 en 11 kanalen

De uitvoering van het programma voor 10 kanalen is na 299 uur[6] stopgezet. De tussentijdse resultaten met betrekking tot de uitvoeringstijd zijn te zien in figuur 8 en met betrekking tot het aantal netwerken zijn te zien in figuur 5. Voor 10, en

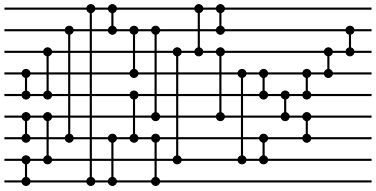
[4] Bij de `Java` implementatie zullen we verwijderen via "$= null$".

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

[6] 12 dagen en 11 uur



Figuur 6: Tijdsmetingen voor uitvoer bij 5 tot en met 9 kanalen.



Figuur 7: Sorteernetwerk 9 kanalen, 25 comparatoren

dus ook voor 11, kanalen is de `Java` implementatie met de gebruikte hardware onvoldoende om resultaten binnen een redelijk tijdsbestek te bekomen. Gebaseerd op figuur 8 schatten we dat voor 10 kanalen meer dan 1500 dagen[7] vereist zijn.

Tabel geven van hoeveel beslissingen er op welke plaats genomen worden.

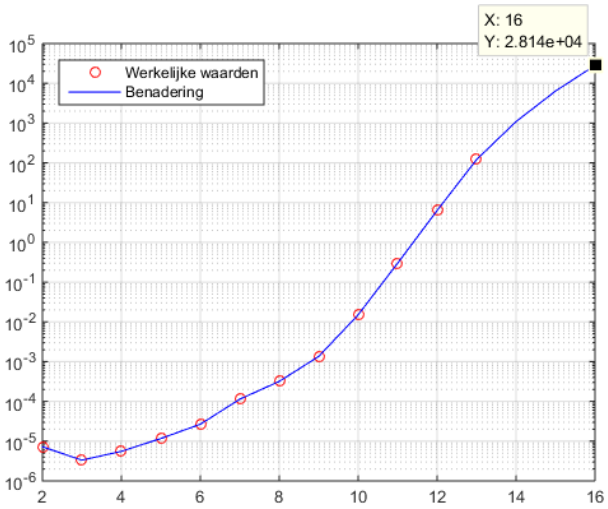
5 Conclusies

Conclusie[Codish *et al.*, 2014]
Conclusie van wat er bereikt is en hoe er verder aan gewerkt kan worden.[Codish *et al.*, 2015]

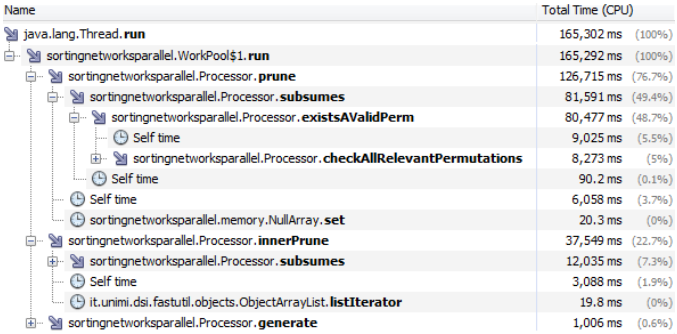
Erkenning

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.
Professor Dr. Ir. Tom Schrijvers, Katholieke Universiteit Leuven.

[7] Berekend op basis van polynomiaal verloop van graad 11.



Figuur 8: Tijdsverloop 10 kanalen tot en met de 13^{de} comparator en benadering van tijdsverloop tot en met de 16^{de} comparator.



Figuur 9: Profile time 8

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.

[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, Luis Cruz-Filipe, and Peter Schneider-Kamp. Sorting networks: the end game. In *Proceedings of the 9th International Conference on Language and Automata Theory and Applications, LATA*, LNCS, 2015.

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

[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.