8 JUNI 2018

PROJECT PARALLELLISME

ANALYSEREN VAN ONLINE CONTRIBUTIES

ASMA OUALMAKRAN
0507834
BA Computerwetenschappen

Inhoudsopgave

Inleiding

De opdracht bestaat uit het implementeren en het benchmarken van parallelle algoritmes. Deze algoritmes moeten comments van de website Reddit analyseren. Reddit bestaat uit contributies die de gebruikers maken en uploaden, hierbij kan er worden gereageerd worden door de andere gebruikers. Deze reacties worden binnen het platform comments genoemd. Dit platform bevat ook SubReddits, deze gaan dan vaak over een specifiek onderwerp.
Voor de comments kan er een gevoelswaarde worden toegekend, hoe positief of negatief de inhoud is van de comment. Is de gevoelswaarde positief, dan is de inhoud ook positief. Is de gevoelswaarde negatief, dan is de inhoud ervan ook negatief. Wanneer de gevoelswaarde gelijk is aan nul, is de inhoud dan ook neutraal.

Het project bestaat uit 2 fases:

- **Fase 1**: Het algoritme moet de gemiddelde gevoelswaarde bereken van een gegeven set comments. Hiervoor wordt eerst de individuele gevoelswaarde per comment berekend. Hier gaat men ervanuit dat men comments analyseert van een SubReddit, en men het onderwerp ervan kent.
  Hierna moet het gemiddelde berekend worden van alle gevoelswaarden.
- **Fase 2**: Het algoritme moet hier ook een gevoelswaarde toekennen aan comments In tegenstelling tot fase 1 wordt hier de gegevens set comments eerst gefilterd aan de hand van een gegeven keyword, alvorens de analyse uit te voeren. Als gevolg worden er enkel comments geanalyseerd die dat specifieke keyword bevatten.
  Ook hier wordt er na de analyse een gemiddelde gevoelswaarde berekend.

Beide fases zijn geimplementeerd aan de hand van het Java fork/join framework.

Complicaties

Er waren een aantal complicaties tijdens het uitvoeren van de benchmarks zowel op Serenity als op een computer met 4 cores. De benchmarking van de sequentiële oplossing op Serenity heeft langer geduurd dan verwacht, waardoor er is geopteerd om geen benchmarks uit te voeren voor T1. Hierdoor is het niet mogelijk om de speedup en de overhead te bepalen van het algoritme. Wel zijn er voor andere waarden van P (niveau van parallelisme, bepaald door het aantal cores die worden gebruikt) benchmarks uitgevoerd, waardoor het wel mogelijk is om de trend te bepalen bij het gebruiken van de verschillende waarden voor P (2, 4, 8, 16). Ook waren er problemen tijdens het uitvoeren van de benchmarks voor dataset 2 op Serenity, hier kreeg ik steeds een NullpointerException in het begin. Het is niet opgelost geraakt waardoor er geen benchmarks zijn voor dataset 2 op Serenity.
Het zelfde probleem kwam voor bij het uitvoeren van de benchmarks op de 4 cores computer. Dit probleem is ook niet opgelost, maar omzeild door in de benchmarking geen grote sequenties van verschillende soorten oproepen uit te voeren.

Implementatie

Algemene structuur van de code

De code bestaat uit 4 klasses:

- BrandAnalyser
- DataReduce
- PrefixSumSentiment
- ParallelAnalyser

Elk van deze classes hebben een specifieke taak binnen de implementatie.
BrandAnalyser, berekend binnen een gegeven dataset, bestaande uit Reddit comments die worden geschreven door gebruikers, per Comment een gevoelswaarde. De berekende gevoelswaarde worden dat opgeslagen in een Array, zodat er later een gemiddelde van kan worden berekend, deze Array wordt teruggegeven als resultaat van de analyse. Deze is geïmplementeerd aan de hand van het fork-join framework. Hier kan ook steeds een sequential cut-off aan worden meegegeven bij het instantiëren van de klasse.

DataReduce, gaat voor een gegeven dataset en een gegeven keyword de Comments filteren. Deze creëert een nieuwe kleinere dataset met enkel de comments die dit keyword bevatten. Het maakt gebruik van een reguliere expressie om te bepalen of het woord voorkomt in de Comment. Als gevolg is de filter hooflettergevoelig. Deze is, net als BrandAnalyser, geïmplementeerd aan de hand van het fork-join framework. Hier kan ook steeds een sequential cut-off aan worden meegegeven bij het instantiëren van de klasse. Dit kon ook geïmplementeerd worden aan de hand van een Pack, dit algoritme filtert data aan de hand van een gegeven waarde, dit algoritme behoud de onderlinge volgorde van de elementen. Hier is er echter niet voor gekozen, aangezien de volgorde van de comments niet van belang zijn.

PrefixSumSentiment, gaat voor een gegeven Float array de som bepalen voor deze getallen. En geeft als waarde de som terug. Dit is een aangepaste implementatie van het prefix-sum algoritme gebruikt in WPO4.

ParallelAnalyser, deze klasse bevat de functie main, hierin worden voor Fase 1 en Fase 2 de juiste klasses geïnstantieerd en hun bijbehorende functies in de correcte volgorde opgeroepen om de gemiddelde gevoelswaarde te berekenen.

Er is voor deze aanpak gekozen aangezien dit de mogelijkheid geeft om elk algoritme makkelijk te vervangen. Ook zorgt dit ervoor dat elk deel van de berekeningen parallel kunnen worden uitgevoerd. Momenteel wordt elk deel sequentieel opgeroepen, maar het geeft wel de mogelijkheid om elk deel ook nog eens te parallelliseren. Hierbij moet men wel rekening houden met concurrency, en moet elk deel nog op elkaar worden afgestemd. Dit is momenteel niet het geval wegens tijdsgebrek.

Fase 1

Code voor deze fase kan worden opgeroepen door *ParallelAnalyser.SentimentSubreddit (List<Comment> comments, int P, int T)* op te roepen. Hierbij is **comments** de dataset die men wenst te analyseren, *P* het aantal threads die men wilt gebruiken en *T* de sequentiële cut-off. Er wordt gebruik gemaakt van een thread pool om de threads bij te houden, en het mogelijk te maken deze te hergebruiken.

Voor deze fase wordt er gebruik gemaakt van de Klasses *BrandAnalyser* en *PrefixSumSentiment. BrandAnalyser* gaat in eerste instantie elke comment een gevoelswaarde toekennen waarna *PrefixSumSentiment* aan de hand van de individuele scores, de totale score berekend.

Fase 2

Code voor deze fase kan worden opgeroepen door *ParallelAnalyser SentimentBrand(String brand,List<Comment> comments, int P, int T)* op te roepen. Hierbij is brand de string waarop er zal worden gefilterd, comments de dataset die men wenst te analyseren, P het aantal threads die men wilt gebruiken en T de sequentiële cut-off. Er wordt gebruik gemaakt van een thread pool om de threads bij te houden, en het mogelijk te maken deze te hergebruiken.

Hier wordt *DataReduce* eerst opgeroepen om een gefilterde dataset te creëren, deze wordt dan op zijn beurt opgeroepen met *BrandAnalyser* die dan voor elke comment een gevoelswaarde zal berekenen. Hierna wordt er zoals in fase 1 gebruik gemaakt van *PrefixSumSentiment* om de volledige score te bepalen, waarna het dan gedeeld wordt door het aantal geanalyseerde comments om de gemiddelde score te bepalen.

Benchmarking

De benchmarks voor fase 1 en fase2 zijn uitgevoerd op een intel i5-6600 CPU op 3,30 GHz. Deze processor heeft 4 cores en 4 threads, het beschikt niet over hyperthreading dat mogelijks de performantie kan beïnvloeden.

Berekeningen overhead, computationele speedup en applicatie speedup

P: Het aantal processoren die beschikbaar zijn.

Tp: De runningtime wanneer er P processoren beschikbaar zijn.

Fase 1

Berekend voor dataset 1

overhead (T1/Tseq): 594822/295868 = 2,01

Computationlele speedup (T1/Tp):

#P		T1	Tp	(T1/Tp)
P=2	(T1/T2)	594822	331168	1,796134
P=4	(T1/T4)	594822	221697	2,683047

Fase 2

Berekend voor dataset 2.

overhead (T1/Tseq): 492/ 14021 = 0,03515926

Computationlele speedup (T1/Tp):

Sequential cut-off = 2

#P		T1	Tp	(T1/Tp)
P=2	(T1/T2)	492,9779	254,2015	1,93932
P=4	(T1/T4)	492,9779	401,2717	1,228539

Sequential cut-off = 1000

#P		T1	Tp	(T1/Tp)
P=2	(T1/T2)	1054,554	994,3663	1,060529
P=4	(T1/T4)	1054,554	328,3706	3,211476

Sequential cut-off = 50 000

#P		T1	Tp	(T1/Tp)

P=2	(T1/T2)	336,6626	254,0845	1,325003
P=4	(T1/T4)	336,6626	249,3887	1,349952

Overhead voor P=4 (T1/T4):

Sequential cut-off = 2

(T1/T4) = 492,9779 / 401,2717 = 1,228539

Sequential cut-off = 1000

(T1/T4) = 1054,554 / 328,3706 = 3,211476

Sequential cut-off = 50 000

(T1/T4) = 336,6626 / 249,3887 = 1,349952

Application speedup voor P=4 (Tseq/T4):

Sequential cut-off = 2

(Tseq/T4) = 14021 / 401,2717 = 34,94212

Sequential cut-off = 1000

(Tseq/T4) = 14021 / 328,3706 = 42,69957

Sequential cut-off = 50 000

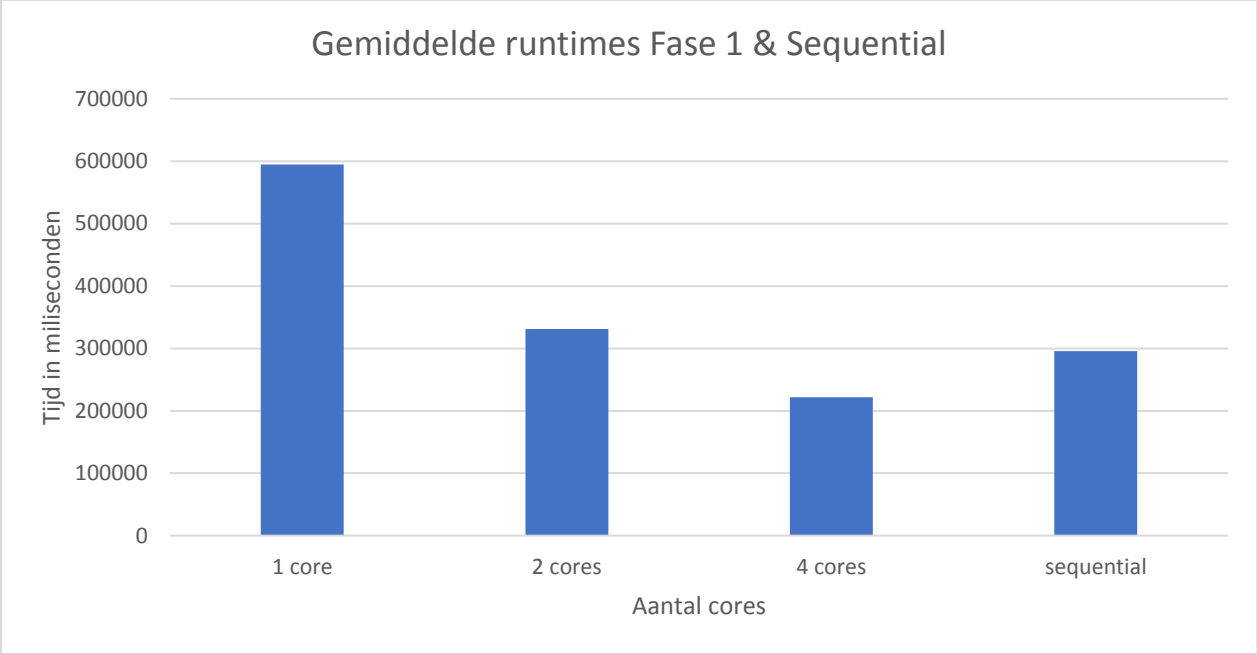
(Tseq/T4) = 14021 / 249,3887 = 56,22261

Serenity

Hiervoor is de berekening voor de overhead en de speedup niet mogelijk. Wegens tijdsgebrek zijn er geen bechmarks gemaakt voor T1.

Fase 1

Vergelijking runtimes van de sequentiële oplossing en fase 1

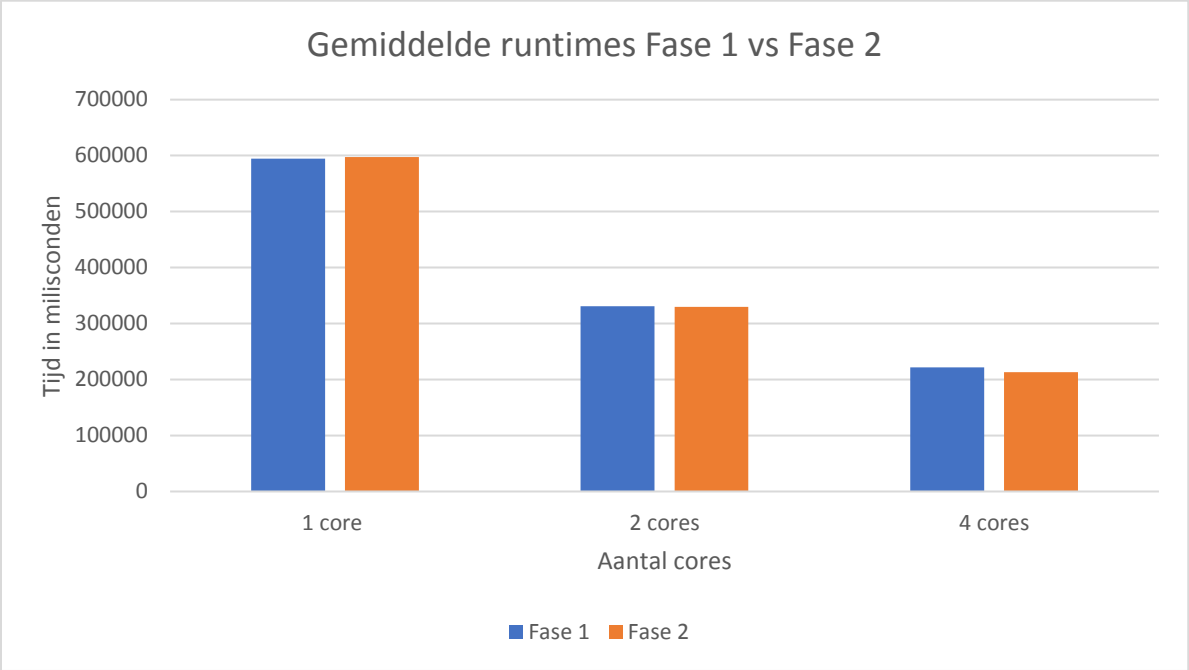


De benchmarks zijn uitgevoerd op dataset 1 en voor de parallelle oplossingen is er gebruik gemaakt van een sequential cut-off van 1000.

Uit deze grafiek kan concluderen dat de runtime voor 1 core hoger ligt dan voor de sequentiële oplossing. De oorzaak kan te wijten zijn aan de overhead die er is voor T1 deze is namelijk overhead (T1/Tseq) = 594822/295868 = 2,01.

Wel is er voor T2 een speedup van (T1/T2) =594822 / 331168 = 1,796134, maar is dan nog steeds niet sneller dan de sequentiële oplossing, dit kan mogelijks liggen aan de overhead of aan een slecht gekozen sequential cut-off waardoor de maximale snelheid niet wordt bereikt. Wel is er een significante versnelling voor T4, hier is de computationele speedup
(T1/T4) = 594822/ 221697 = 2,683047.
De meest optimale resultaat kan bereikt worden door gebruik te maken van T4, wel moeten er verder nog metingen gebeuren met verschillende sequential cut-offs om hiervoor de optimale parameter te kunnen benaderen of te bepalen.

Fase 1 vs Fase 2



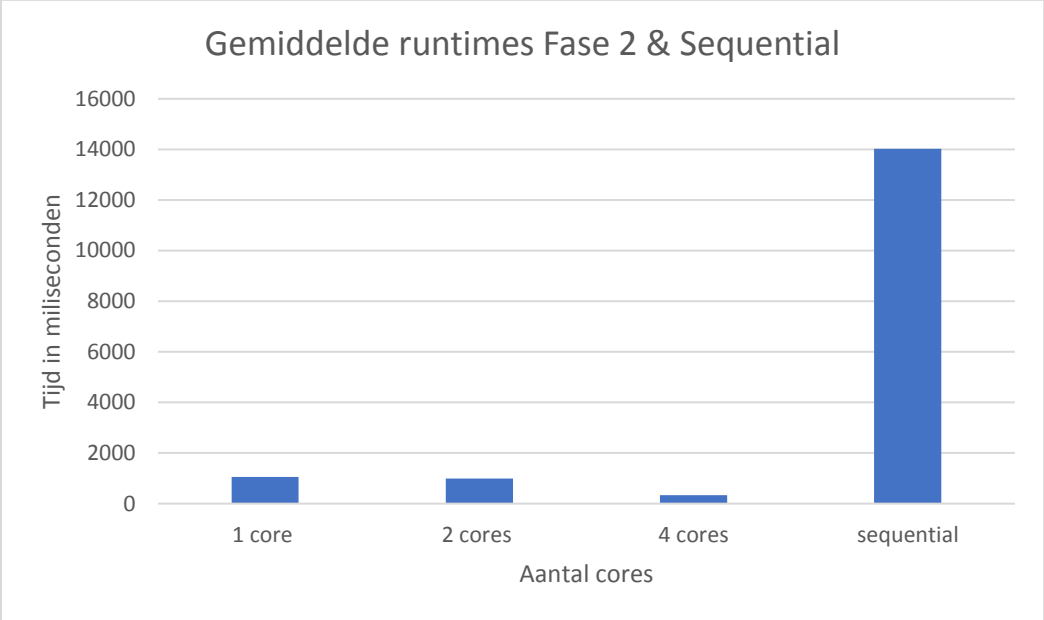
Voor deze benchmarks is ervoor gezorgd geweest dat er geen gefilterde data wordt gebruikt, maar dat de filter nog steeds gebruikt werd. Hierbij werd de data wel gefilterd, maar voor de analyse is er gebruik gemaakt van de originele dataset.

Men kan zien dat beide algoritmes dezelfde trends volgen en ook sneller worden naarmate dat het parallelisme niveau wordt verhoogd. Ook zijn de runtimes van beide fases ongeveer even hoog, wel is er voor fase 2 op T4 een kleine versnelling tegenover de runtime van fase 1 voor T4. Deze versnelling is niet significant, maar kan verder worden geoptimaliseerd voor beide algoritmes door de sequentiele cut-off te bepalen waarbij het resultaat optimaal is.

Dit laat ook zien dat het parallel filteren van de data voor het analyseren, geen tot weinig invloed heeft op de snelheid van het algoritme.

Fase 2

Vergelijking runtimes van de sequentiële oplossing en fase 2

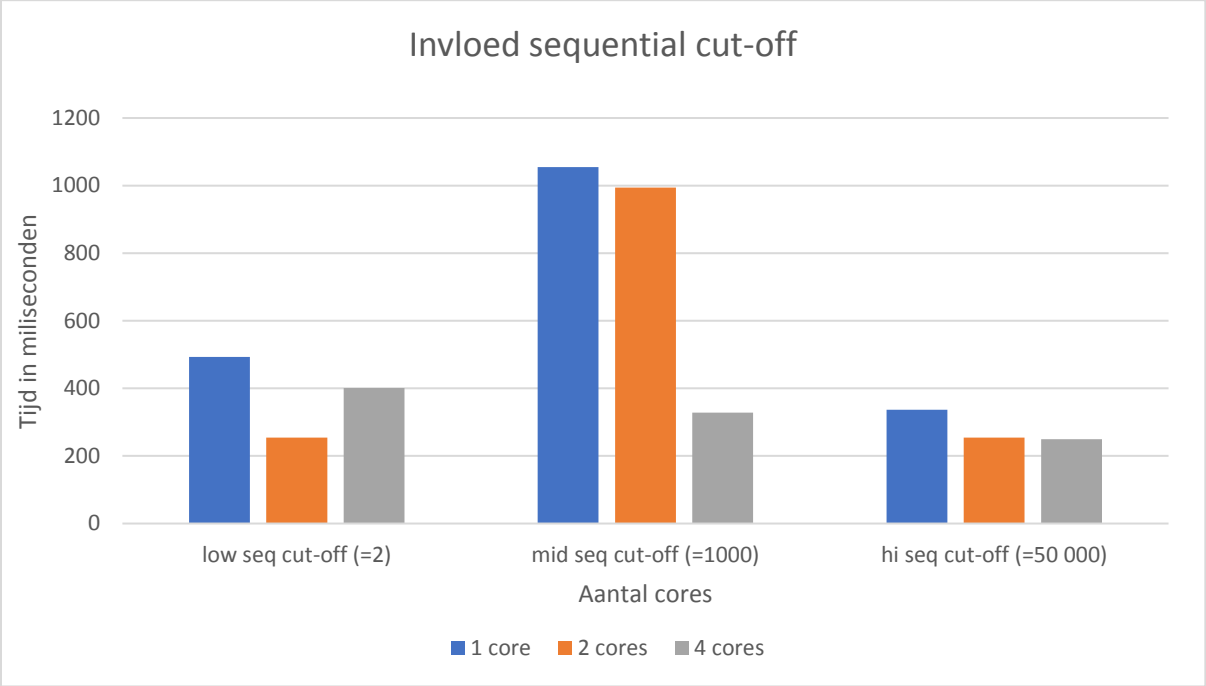


De benchmarks voor de parallelelle code is uitgevoerd met een sequentiële cut-off van 1000. Hier is het verschil tussen de parallelle oplossingen en de sequentiële oplossing duidelijker aanwezig dan bij de code voor fase 1. Dit is te wijten aan de grote verschillen tussen de overhead, deze bedraagt voor fase 1 namelijk (T1/Tseq) = 594822/295868 = 2,01en voor fase 2 (T1/Tseq) = 492/ 14021 = 0,03515926.

De parallelle implementatie van fase 2 is efficiënter dan die voor fase 1. Dit ligt aan het feit dat het algoritme voor fase 2 in dit geval wel gefilterde data gebruikt en hierdoor een analyse uitvoert op een veel kleinere dataset. Dit maakt het algoritme heel snel veel performanter. Hierdoor is het ook sneller dan fase 1.

Wel zijn de overheads van beide algoritmes heel verschillend, en verklaart dit het verschil in performantie tegenover hun sequentiële tegenhanger.

Invloed van sequentiële cut-off



In bovenstaande grafiek kan men de invloeden afleiden van sequentiele cut-offs voor 1,2 en 4 cores. Er is gekozen geweest om het te testen voor een sequentiale cut-off van 2, 1000 en 50 000. Dit is bepaald geweest aan de hand van de grootte van de dataset (het aantal comments dat het bevat). Voor 2 is er gekozen om de invloed van een mimimale sequential cut-off te bepalen dit betekend dat het werk wordt verdeelt over een grote hoeveelheid threads, die zelf bestaan uit een kleine hoeveelheid werk. Voor 1000 is er gekozen om de invloed dat het creëren van meerdere sub-threads heeft te meten. Als laatste is er gekozen voor een sequential cut-off van 50 000 is gekozen om te meten wat de invloed is wanneer het werk in grote stukken wordt verdeeld. Dit zorgt er ook voor dat het aantal te analyseren comments per thread groter is dan bij de vorige twee sequential cut-off's.

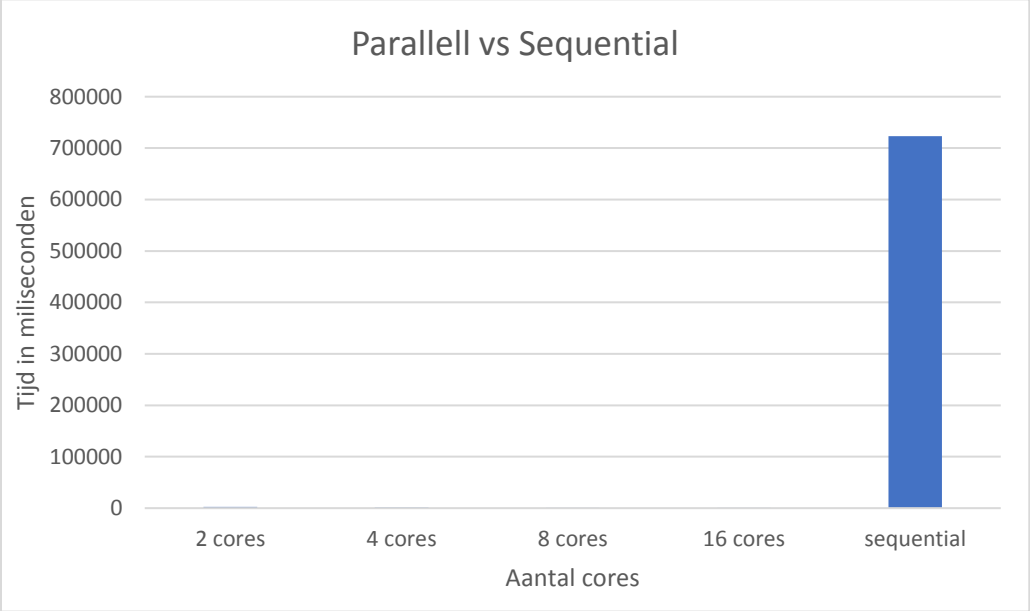
Bij 1 en 2 cores kan men zien dat de runtime voor een sequentiele cut-off van 1000 hoger is dan bij 2 en 5 000. En dat bij 4 cores de runtimes daalt naarmate dat de sequential cut-off stijgt.
Dat de runtimes voor 2 cores en een sequential cut-off van 1000 zo hoog zijn was een onverwacht resultaat, men zou verwachten dat deze even laag zou liggen voor een sequential cut-off van 2. Aangezien er meerdere subtaken worden, minder dan voor cut-off = 2 maar wel mee dan voor cut-off = 50 000. Dit zou voor een betere load balance zorgen.

De reden voor de tragere runtimes voor T1 en T2 met een sequential cut-off van 1000, kan liggen aan de overhead waarbij deze meer zou bedragen dan voor T4, waardoor ook de speedup lager ligt van voor T4. De speedup voor T2 bedraagd (T1/T2) = 1054,554 / 994,3663 = 1,060529 en voor T4 bedraagt dit (T1/T4) = 1054,554 / 328,3706 = 3,211476. Wat een significant verschil maakt. Bovendien ligt de overhead voor T4 op (T1/T4) = 1054,554 / 328,3706 = 3,211476, dit zou dan nog hoger liggen voor T2.

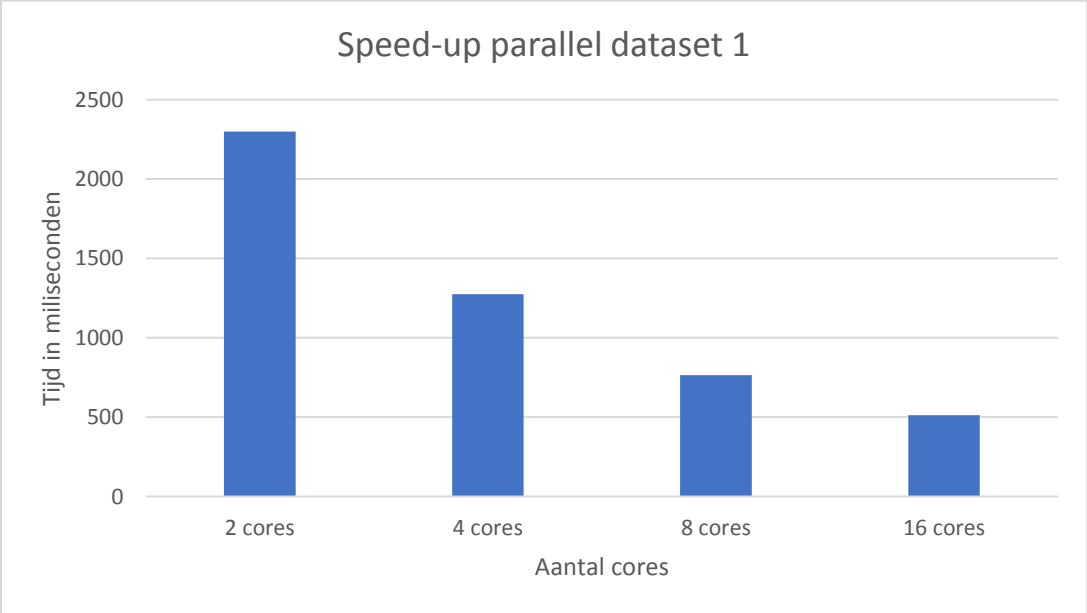
Om een zo goed mogelijke performantie te krijgen, is het dus aangeraden om een zo hoog mogelijke cut-off te nemen, maar nog klein genoeg zodat het niet sequentieel zou zijn. Dit zou het geval zijn als de sequential cut-off even groot zou zijn al de dataset.

Serenity

De benchmarking is gebeurd met het algoritme van fase 2. Hierbij is er een sequentiele cut-off genomen van 1000 comments.

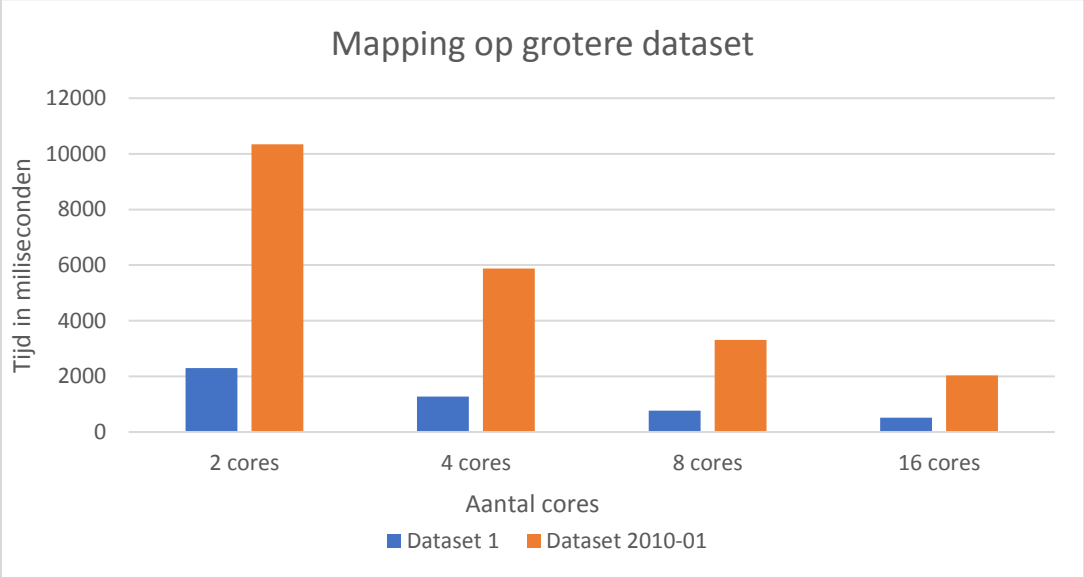


In deze histogram kan je het grote verschil van runtimes tussen de parallelle oplossing en de sequentiële oplossing. Dit verschil is veel groter dan bij de benchmarks met de 4 core computer. Dit kan liggen aan het feit dat de kloksnelheid van Serenity's cores lager ligt, deze is namelijk 2,3Ghz. Dit vertraagt de sequentiële oplossing.



De histogram Speed-up parallel dataset 1 laat de versnelling zien bij het toevoegen van extra cores. Hierbij is de sequential cut-off gehouden op 1000. Er is een duidelijke versnelling te zien bij het

verdubbelen van het aantal cores die beschikbaar worden gemaakt. Er is niet gemeten wat de runtime is voor 36 cores (alle beschikbare cores gebruiken), dus weten we niet of het hier dan nog steeds versneld.
Omdat er geen metingen zijn gebeurd voor T1, weten we niets over de overhead, of over de speedup, we kunnen ze enkel afleiden vanuit de runtimes. Wel kunnen we zien dat naarmate dat het aantal cores worden verhoogt, de versnelling minder groot wordt. En de grafiek zich logaritmisch gedraagt, op een bepaald punt ook een asymptoot zal bereiken waarna er geen versnelling meer aanwezig is bij het verhogen van het parallelisme niveau.
Dit laat wel zien dat het algoritme zich goed gedraagt bij het toevoegen van extra cores.



Uit bovenstaande grafiek kan men de runtimes aflezen voor het algoritme van fase 2 met een sequential cut-off van 1000. Hier werd er gebruik gemaakt van dataset 1 wat een kleine dataset is, en dataset 2010-01 die alle Reddit comments bevatten voor januari 2010 en ook meer comments bevat dan dataset 1. Dit is te merken aan het feit dat er meer tijd nodig is om deze te analyseren.
Bovendien kan men ook zien dat zowel met de grote als met de kleine dataset dezelfde trend aanwezig is, namelijk dat de runtimes lager zijn naarmate dat het aantal beschikbare cores stijgen.
Dit laat zien dat het algoritme om kan gaan met grotere stukken data, en de versnellingen in dezelfde trend zijn als bij een kleinere dataset.
En dat er zeker meer snelheid is tot en met 16 cores, en betekend dat we niet zeker kunnen zijn of deze trend nog zichtbaar is wanneer er nog meer cores worden gebruikt. Er zal wel zeker een limiet zijn tot waar men versnelling krijgt, uit deze grafiek is niet af te leiden wanneer men de maximum snelheid bereikt heeft.

Conclusie

De sequential cut-off alsook de overhead kunnen een grote invloed hebben op de performantie van het algoritmes. Hieruit kunnen we besluiten dat een goede/optimale sequential cut-off belangrijk is om de runtime in te korten. Ook is het belangrijk om de overhead zo laag mogelijk te houden, bij een te hoge overhead is een parallelle oplossing niet sneller dan de sequentiële.
Ook het ervoor zorgen dat elk deel van het algoritme parallel gebeurt heeft weinig invloed op de performantie wanneer er een extra stap wordt toegevoegd dat parallel is.

We kunnen ook besluiten dat het algoritme zich goed aanpast aan grotere taken, en aan een hoger parallelisme niveau.