Parallellisme - Project report

Hans Van der Ougstraete

July 16, 2020

Contents

1 Implementatie

1.1 Fase 1

Fase 1 is opgedeeld in twee delen, het eerste deel maakt een lijst met relevante businesses door in de base case de evaluaterevalence methode uit SequentialSearch aan te roepen. De Yelpdata wordt dus in stukken opgedeeld tot het kleinste deel een business is. Deze lijst wordt in het tweede deel aan de hand van een parallele mergesort gesorteerd. Deze code is terug te vinden in de ParallelSearch klasse.

1.2 Fase 2

Het doorzoeken van de revieuw om het aantal voorkomens van een woord te bepalen kan op verschillende manieren. Er bestaat een java klasse split welke een text opdeelt en per woord in een vector stopt. Echter heb ik moeilijkheden om de reguliere expressie die aangeeft waar een woord eindigt (bv spatie of leesteken) zo in te stellen dat de unit tests slagen. Ik had net iets meer of minder voorkomens dan de sequentiële search afhangende van de gebruikte reguliere expresie. Zonder controle over de input van de revieuws zou een reguliere expressie die voor een preset werkt misschien in een andere preset iets over het hoofd kijken. Deze code kan u vinden in CountOccurencesP en wordt aangeroepen in de functie countOccurencesP beide staan in ParallelSearch.java

Een ander oplossing is het gebruik van hashmaps, hierbij wordt het voorkomen van ieder woord geteld. Deze hash opslagen zou bij een volgende zoektocht snel het aantal voorkomens uit de hash kunnen halen. In deze hash zijn de woorden de key's en het aantal voorkomens de value.

Als derde oplossing heb ik geprobeerd de sequentiële manier van voorkomens tellen te parallelliseren. Het is me niet gelukt hier alle bugs uit te krijgen. De code werkt omdat de tweede if-test in de compute functie altijd true is en het sequentiële algoritme start. Deze code kan u vinden in CountOccurencesALT en wordt aangeroepen in de functie countOccurencesP beide staan in ParallelSearch.java

2 Evaluatie

De benchmarks op eigen computer (quad core i5-2300 @2.80GHz) zijn 4 maal uitgevoerd. Voor SequentialSearch, ParallelSearch met 1, 2 workers en ParallelSearch met 4 workers. Dit telkens 1500 keer. De resultaten worden per preset weergegeven.

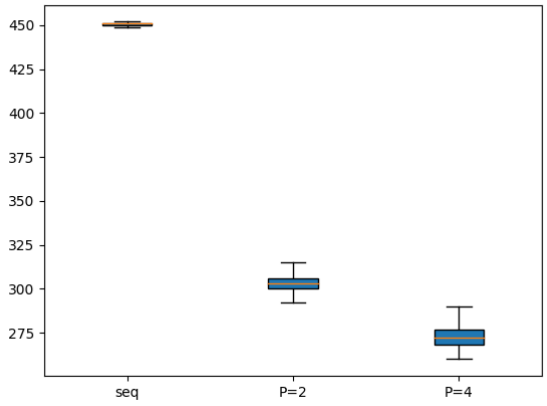

Figure 1: Preset1

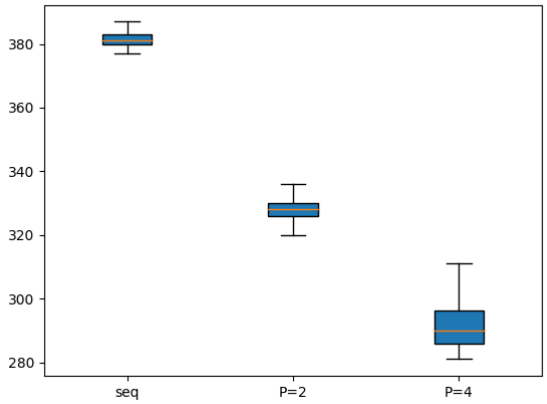

Figure 2: Preset2

Runtime ms	Preset1	Preset2	Preset3
Parallel P=1	466	514	145
Parallel P=2	304	328	111
Parallel P=4	273	291	108
Sequential	451	381	134

Table 1: Gemiddele runtime in miliseconden

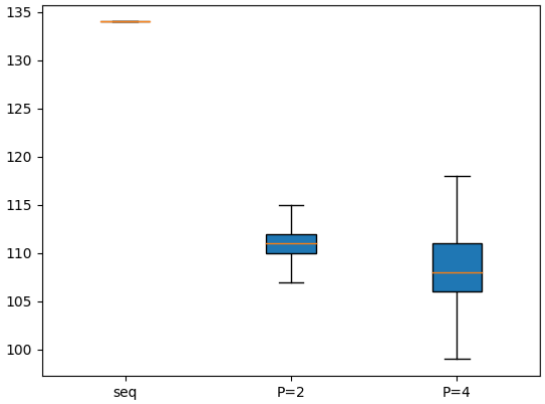

Figure 3: Preset3

Overhead	Preset1	Preset2	Preset3
T1/Tseq	1.03	1.35	1.08

Table 2: Overhead waarden voor de 3 presets.

2.1 Fase 1

2.1.1 Overhead

Bovenstaande tabel toont de overhead, deze overhead van het verdelen in threads is niet aanwezig bij de sequentiële versie omdat de sequentiële versie niet opdeelt in alsmaar kleinere taken. De overhead is verschillend per preset omdat het opdelen van het werk afhangt van de respectievelijke data. De overhead kan kleiner gemaakt worden door een goede cut-off waarde te gebruiken, hierdoor wordt er sneller sequentieel gewerkt en daardoor minder opgedeeld in subtaken.

2.1.2 Speed-up

Een perfecte speed-up zou gelijk zijn aan het aantal extra processors, dit is hier niet het geval. De speed-up is ook verschillend per preset, het aantal businesses en revieuws beïnvloeden dit volgens mij. Bij preset 3 is er nauwelijks verschil

Speed-up	Preset1	Preset2	Preset3
Tseq/T2	1.48	1.16	1.21
Tseq/T4	1.65	1.31	1.24

Table 3: Speed-up values for all 3 presets.

tussen T=2 en T=4, ik veronderstel dat dit is omdat de data klein is. het meeste winst wordt geboekt bij preset 1, de preset met het meest aantal businesses.

2.2 Fase 2

Testing..