

Report Parallisme

MIKHAEL SWUSTEN 0589019

1.1 Implementatie

De `ParallelSearch` klasse en zijn geneste klassen implementeren een parallelle zoekmachine voor trefwoorden met behulp van het Fork/Join framework in Java. Het zoekproces is verdeeld in twee hoofdfasen: het zoeken naar artikelen die het trefwoord bevatten en het tellen van voorkomens van het trefwoord in die artikelen om hun relevantie te berekenen.

1.1.1 Fase 1: Article Search

De klasse `SearchTask`, die `RecursiveTask<List<RRecord>>` uitbreidt, is verantwoordelijk voor het doorzoeken van een bereik van artikel-ID's in de `WikipediaData` dataset om die artikelen te identificeren die het trefwoord bevatten. Als het bereik van de te doorzoeken artikelen onder de threshold `T1` ligt, wordt het zoeken sequentieel uitgevoerd. Anders wordt de taak verdeeld in twee subtaken, die parallel worden uitgevoerd.

1.1.2 Fase 2: Counting Occurrences

De klasse `countOccurrencesTask`, die `RecursiveTask<Integer>` uitbreidt, wordt gebruikt om de occurrences van het trefwoord in de tekst van een artikel te tellen. Deze klasse volgt een soortgelijk patroon als `SearchTask`. Als het tekstsegment onder de threshold `T2` ligt, wordt de telling sequentieel uitgevoerd. Anders wordt de taak opgesplitst in twee subtaken en parallel uitgevoerd.

1.1.3 Fork/Join Patterns

- **Parallel Sorting:** Er is geen nieuwe implementatie van sorting algoritme toegevoegd. Nadat de zoekresultaten zijn verzameld, worden de resultaten gesorteerd met `Collections.sort(result)` in de zoekmethode.
- **Map Pattern:** De implementatie van de klasse `SearchTask` is gebaseerd op het map-patroon, waarbij het probleem (het doorzoeken van een reeks artikelen) wordt opgedeeld in subproblemen die elk onafhankelijk en parallel worden verwerkt.
- **Reduction Pattern:** De klasse `countOccurrencesTask` volgt het reductiepatroon. De tekst wordt opgedeeld in kleinere segmenten, de occurrences worden parallel geteld en de resultaten worden gecombineerd om de totale telling te krijgen.

2.1 Evaluatie

2.2 Experimenten en experimentele opstelling

In de rapportage zal ik niet ingaan op de Firefly-server, aangezien ik geen tests heb kunnen uitvoeren op die server. Alle benchmarks werden uitgevoerd op de lokale machine.

Lokale Machine Specificatie:

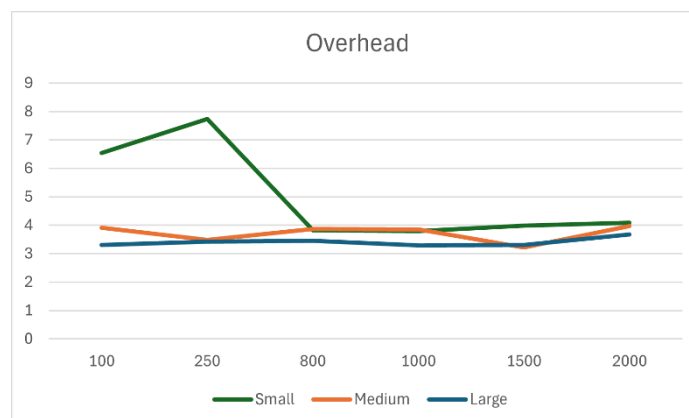
- Processor: AMD Ryzen 5 3600 (6 fysieke processors, 12 logische processors) 3,6 GHz (4,2 GHz Turbo Boost)
- BIOS: F40
- RAM: 16GB DDR4 3600MHz
- Opslag: 2TB HDD
- OS: Windows 10 pro
- Heap Size: 2048 MB
- JMH Version: 1.37
- Java Version: JDK 19.0.2, OpenJDK 64-Bit Server VM, 19.0.2+7-FR

Het doel van de benchmarks was het evalueren van de prestaties en het identificeren van de optimale waarden voor de thresholds T1 en T2 op de lokale machine. Voor de benchmarks heb ik 2 warmup iteraties, 1 fork en 3 normale iteraties van 1000ms.

De score van de benchmark is de gemiddelde uitvoeringstijd van de methode per bewerking, uitgedrukt in milliseconden. Een lagere score geeft betere prestaties aan.

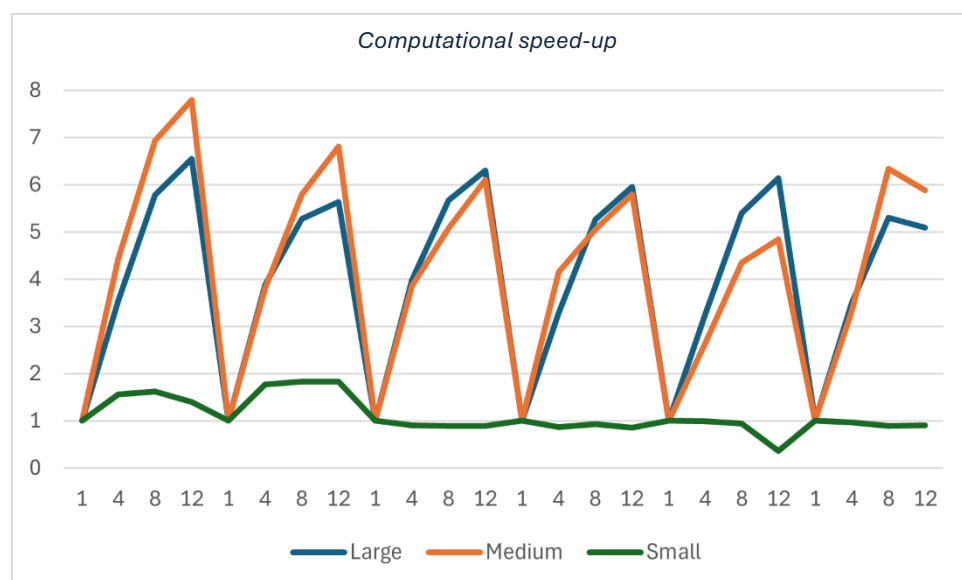
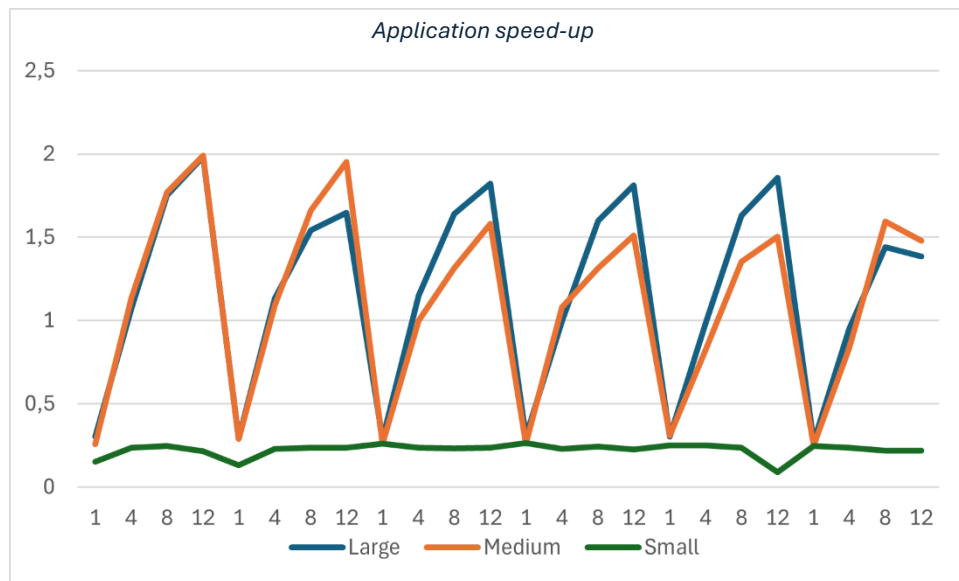
2.3 Overhead en speed-up

1. What is the overhead of your implementations? Is this overhead acceptable? What causes this overhead?



De gemiddelde overhead voor de small dataset is 5 met een piek van 7,7. De gemiddelde overhead voor de medium dataset is 3,7 en voor de large dataset 3,41. Voor grote datasets lijkt de overhead relatief efficiënt, maar voor kleine datasets kan het een probleem zijn. Mogelijke oorzaken van deze overhead zijn dat de implementatie niet geoptimaliseerd is voor kleine datasets en slecht geoptimaliseerd geheugenbeheer.

- What are the speed-ups of your implementations? Are these speed-ups higher or lower than expected? Why?



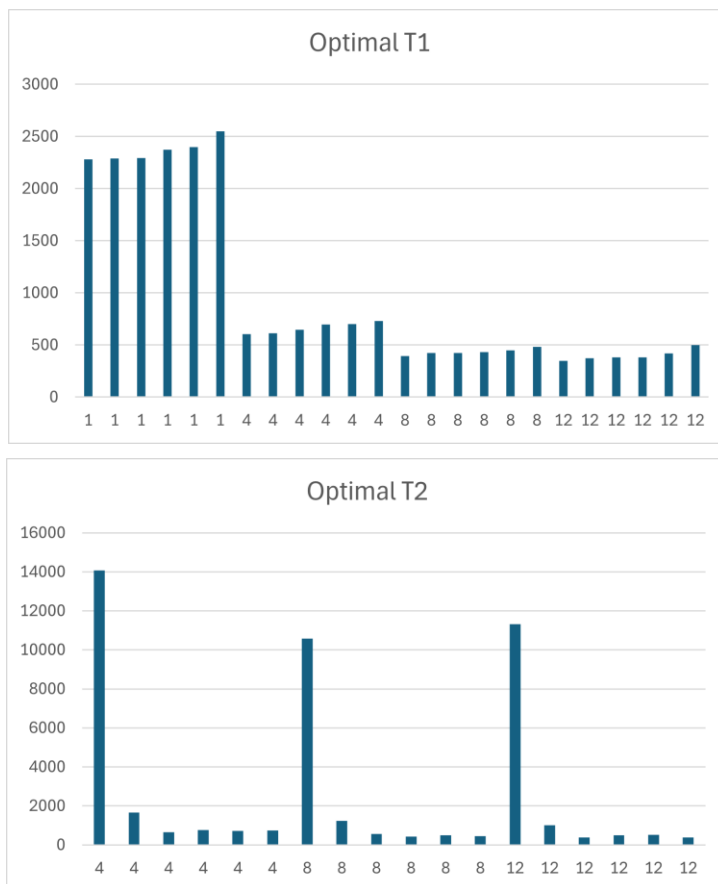
De snelheid neemt toe naarmate het aantal threads (x-as) voor grote datasets. Dit is verwacht gedrag voor parallele implementaties, omdat het verhogen van het aantal threads over het algemeen leidt tot een beter gebruik van resources en dus betere prestaties.

- What speed-up is larger: the computational or application speed-up? Is this what you expect? Why?

De maximale computational speed-up is ongeveer 6,30 en application speed-up is ongeveer 1,98 voor de Large dataset.

4. How well do your implementations scale? Why do they (not) scale well?
De implementaties toont een behoorlijke schaalbaarheid, vooral in termen van computational speed-up. Als het aantal threads (p) toeneemt, verbeteren zowel de computational als de application speed-ups over het algemeen, wat aangeeft dat de implementaties effectief extra resources gebruiken om de prestaties te verbeteren.
5. Based on your experiments, which of both implementations is the best? Why?
In dit geval is de implementatie van fase 1 beter omdat deze een iets hogere score oplevert dan fase 2. De implementatie van fase 2 had niet veel bijdrage.

2.4 Optimale threshold



1. How does the threshold influence the overhead and speed-up of your implementations? What happens when you make the threshold larger or smaller? Explain.
Om de beste T1 te bepalen, werd T2 op de maximumwaarde gezet en T1 op 100, 250, 800, 1000, 1500, 2000. Hieruit kan ik concluderen dat een lagere T1 beter is dan een hogere T1. Om de optimale T2 te bepalen, werd T1 op 1 gezet en T2 op de waarden 1000, 10000, 100000, 1000000, 10000000. Hieruit kan ik concluderen dat een hogere T2 beter is dan een kleinere T2.
2. Suppose you would need to determine a threshold naively (i.e., without benchmarking). How would you do this? What would you consider?
Voor T1 zou ik een kleine threshold kiezen en voor T2 zou ik een grote threshold nemen omdat tekst verwerken sneller gaat in grote stukken.

3. Based on your measurements, what is the best threshold? Why?

Voor de large dataset zijn dit de beste waarden: $P=12$, $T1=100$, $T2=100.000$ met score 337,533.

Voor de medium dataset zijn dit de beste waarden: $P=12$, $T1=100$, $T2=1.000.000$ met score 118,853.

Voor de small dataset zijn dit de beste waarden: $P=1$, $T1=1000$, $T2=1.000.000$ met score 0,45911 (slechter dan sequential search).

Een kleine $T1$ maakt beter gebruik van de geparallelleerde implementatie en een grote $T2$ is beter voor het verwerken van teksten.

2.5 Implementation characteristics

1. What is the work of the computation?

Het rekenwerk omvat:

- Zoeken naar artikelen die een specifiek trefwoord bevatten. $\rightarrow O(n)$ (n is aantal artikels)
- Het tellen van de voorkomens van het trefwoord in elk artikel.
- Het evalueren van de relevantie van elk artikel op basis van het voorkomen van het trefwoord. $\rightarrow O(l)$ (l is gemiddelde lengte van tekst)

Dit werk is verdeeld in twee hoofdfasen: het zoeken naar artikelen ($O(n * l)$) en het tellen van voorkomens ($O(k * m)$) waarbij m de lengte van een tekstsegment is en k het aantal segments is). Samen is de tijdscomplexiteit $O(n * l) + O(k * m)$.

2. What is the span of the computation?

In de eerste fase vind je het langste pad van afhankelijkheden wanneer de search task wordt opgedeeld in twee subtaken. Elke subtaak verdeelt zich vervolgens recursief in andere subtaken totdat de threshold $T1$ is bereikt. Op elk recursieniveau splitsen de taken zich in tweeën, wat leidt tot een binaire boomstructuur. De maximale diepte van deze boom komt overeen met het aantal keren dat de reeks artikelen wordt gehalveerd voordat de threshold $T1$ wordt bereikt. Daarom kan de duur van de artikel zoek fase worden berekend als $O(\log n)$, waarbij n het aantal artikelen is.

In de tweede fase ontstaat de langste pad van afhankelijkheden wanneer de tekst van een artikel wordt opgedeeld in segmenten voor gelijktijdige telling. Elk tekstsegment wordt geanalyseerd om het aantal voorkomens van het trefwoord te tellen, en deze aantallen van verschillende segmenten worden samengevoegd. Daarom kan de duur deze fase voor het tellen van voorkomens dus worden geschat op $O(\log m)$, waarbij m staat voor de lengte van de tekst van het artikel.

Samen is dit $O(\log n) + O(\log m)$.

3. How many tasks are created by your implementation?

Initieel wordt er 1 task gemaakt om de volledige reeks artikel-ID's te doorzoeken. Als het bereik de threshold $T1$ overschrijdt, splitst deze taak zich in twee subtaken (left and right). Elk van deze subtaken worden verder opgesplitst in twee tasks als hun bereik groter is dan de threshold $T1$, en dit proces gaat recursief verder tot de threshold is bereikt. Het tellen van voorkomens wordt soortgelijk gedaan.