Threaded Merge Sort

Kenan de Vries

2VA

27/02/2023

Inleiding

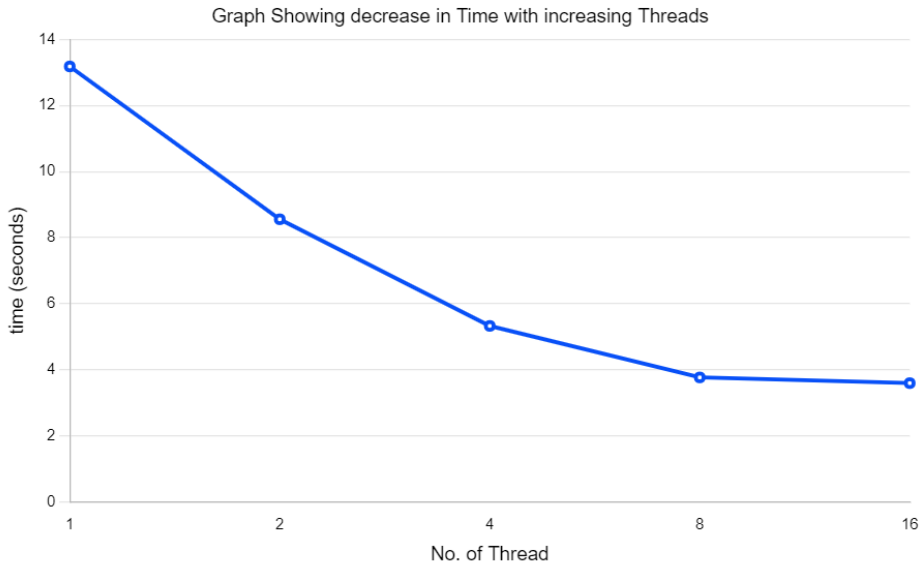
Afgelopen weken hebben wij tijdens het vak High peformance programming geleerd over threads en hoe we deze kunnen benutten om bepaalde algoritmes sneller te kunnen laten uitvoeren. Aan ons is nu de opdracht om van een voorgaande opdracht, de merge sort, een nieuwe versie te maken: de threaded merge sort. Deze versie gebruikt threads om de merge sort te parallelliseren. Dit verslag omvat mijn implementatie van het algoritmen.

Merge sort vs Threaded merge sort

Een basis merge sort is heel simpel in werking. Een array wordt eerst onderverdeeld in kleinere delen, dus 1,2,3,4,8,7,6,5. Wordt bijvoorbeeld een array van 1,2,3,4 en 8,7,6,5. Hierna worden beide arrays apart gesorteerd en aan het einde weer bij elkaar gevoegd. Het enige verschil met een threaded merge sort, is dat meerdere threads (minimaal 2, anders krijg je een normale merge sort) de lijsten opzich nemen. 1,2,3,4 zou dan worden gedaan door thread 1 en 8,7,6,5 door thread 2. Deze worden tegelijkertijd gesorteerd. Als deze beide klaar zijn worden ze aan het einde samengevoegd. Er kan dus overhead zijn omdat de threads op elkaar moeten wachten. In de bijlage wordt dit proces beter uitgelegd en geïllustreerd.

Analyse

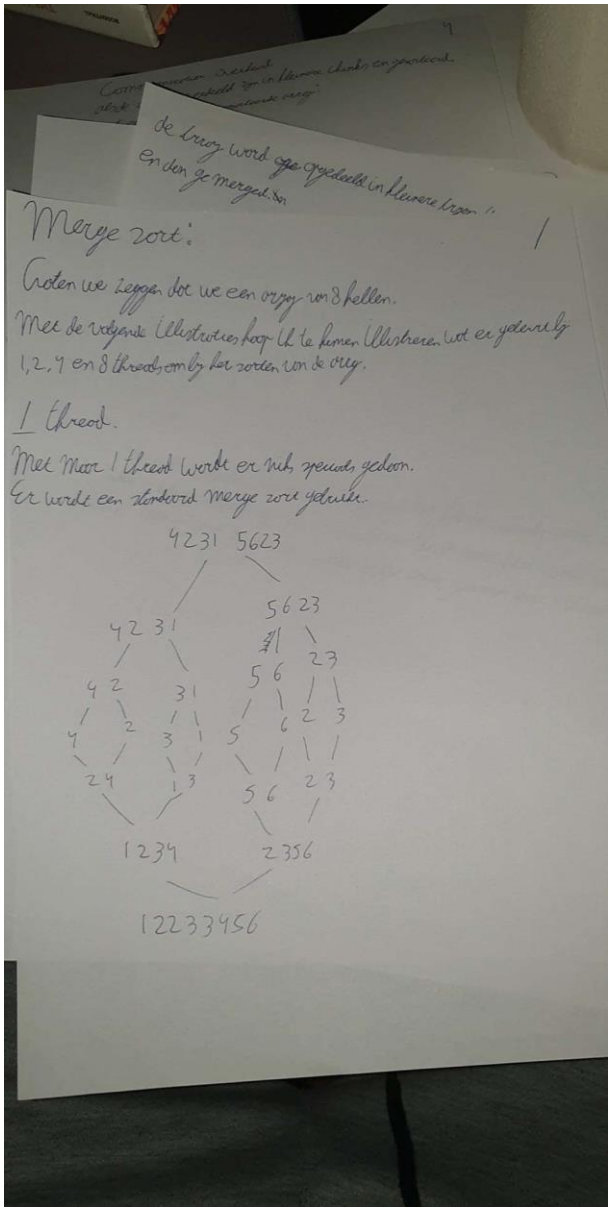
Zoals verwacht presteert het algoritme met gebruik van threads over het algemeen aanzienlijk beter dan de basis merge sort. De enige uitzondering hierop is met kleine arrays. Bij een array van 100 200 300 etc. Is het merge sorten met 1 thread aanzienlijk sneller dan bij gebruik van meerdere threads. Dit komt waarschijnlijk omdat het mergen van de gesorteerde lijst aan het einde langer duurt door de hoeveelheid threads. Rond een array van de 900 is de benodigde tijd vrijwel gelijk van alle arrays, al is de single threaded merge sort enkele microseconden slomer dan de rest en is 2 threads optimaal. Bij een array van 3000 en hoger is past echt duidelijk verschil te zien. 1 thread doet er 2x zolang over om te sorteren dan bij gebruik van 2 of meer threads en de hoeveelheid threads vermindert de benodigde tijd. Het interessantste voor mij was een enorme array, namelijk een van 50 miljoen testen. Hieronder zijn daarvan de resultaten te zien die de ware kracht van parallel merge sorten demonstreert:



Conclusie

Zoals in de analyse eigelijk al werd gezegd is de standaard merge soort beter voor ‘kleinere’ arrays. Het middenpunt ligt bij de ongeveer 900/1000 elementen, en bij sprake over duizende elementen is de standaard merge soort erg traag vergeleken met de threaded. Als er echter meer dan 8 threads worden gebruikt lijkt er weinig verbetering te zitten in de runtime. Mijn conclusie is dus dat parallel merge soort beter is bij arrays boven de 1000, en dat het eigenlijk geen nut heeft om meer dan 8 threads te gebruiken voor dit algoritmen omdat de performance nauwelijks omhooggaat.

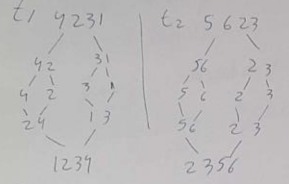
Bijlages:


de lijst word opgedeeld in kleinere lijsten
en dan gemerged.
Merge sort:
Straten we zeggen dat we een array van 8 cellen.
Met de volgende illustraties hoop ik te laten illustreren wat er gebeurt bij
1, 2, 4 en 8 threads om bij het sorten van de array.
1 thread.
Met maar 1 thread wordt er niks speciaals gedaan.
Er wordt een standaard merge sort gebruikt.
4231 5623
4231
5623
42
31
56
23
4
2
3
1
5
6
2
3
24
13
56
23
1234
2356
12233456

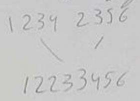
2

de array word opgedeeld in kleinere arrays, gesorteerd en dan ge merged.

2 threads.



De threads werken (aan het begin) tegelijk aan kleinere chunks van de array. Ze zullen ongeveer dezelfde tijd klaar zijn. Indien de ene thread eerder klaar is wacht hij op de andere. Nadat ze goed zijn gesorteerd worden ze bij elkaar gevoegd door 1 thread.



3

time complexity met 1 thread

Zonder threading is de time complexity

$O(N \log N)$

Met threading

$O(\frac{1}{t} N \log N)$

Waarbij t uitleverend de threads zijn.

De time complexity blijft hetzelfde voor de big O

$O=(N \log N)$

Maar de benodigde tijd voor het sorteren van de array wordt verminderd t× bij het negeren van de communicatie overhead. geproduceerd door de laatste merge stap.

4

Communication overhead

als de arrays verdeeld zijn in kleinere chunks en gesorteerd. geeft elke thread een gesorteerde array.

T=4 8 elementen

4231 5623

T1	T2	T3	T4
42	31	56	32
↓	↓	↓	↓
24	13	56	23

Nu de chunks gesorteerd zijn voegt 1 thread alles samen.

T_1 en T_2 output : 1234
T_3 en T_4 output = 2356 } 12233456

dus de daadwerkelijke besparing in tijd wordt verminderd door de overhead loos.