# Threaded Merge Sort

Floris Videler

1758374

x-02-2021

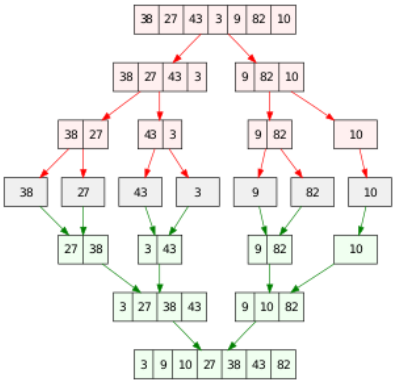
## Inleiding

Voor het vak high performance programming moesten we voor de derde inleveropdracht een threaded versie van merge sort ontwerpen, analyseren en bouwen.

## Ontwerp

Voor we verder kunnen beginnen met het ontwerpen van een threaded versie van merge sort, is het handig om te begrijpen hoe merge sort werkt. In figuur 1 is uitgetekend hoe merge sort op een kleine lijst werkt. Er zijn twee fases in merge sort:

1. Het splitsten van de lijsten, dit zijn de rode lijnen in figuur 1.
2. Het mergen en tegelijkertijd sorteren van de kleinere lijsten, dit zijn de groene lijnen in figuur 1.




*Figuur 1*

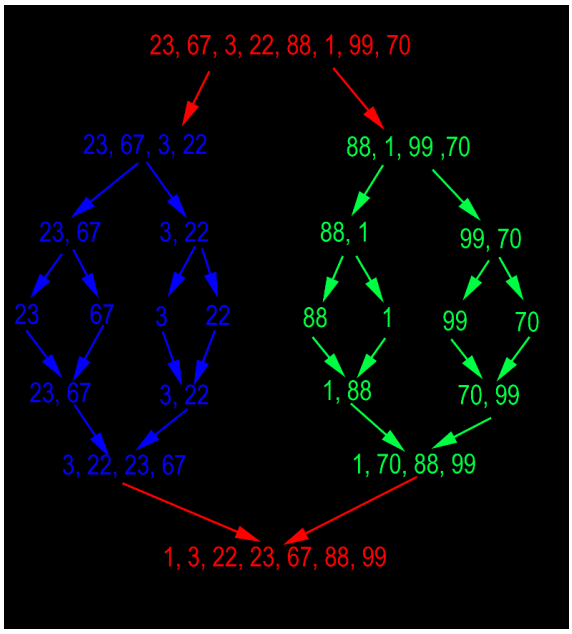
Nu we de basis van merge sort begrijpen kunnen we een stap verder en threads toevoegen. Voor ik mijn ontwerp ga toelichten is het handig om één ding te onthouden: er wordt gebruik gemaakt van een thread pool. Dit houdt in dat alle threads tegelijkertijd moeten worden gestart. Dit is dus niet zoals bij een worker thread die op ieder moment kan worden gestart. Deze aanpak zorgt voor een SIMD (Single Instruction Multiple Data) soort systeem, dit houdt in dat de merge sort functie op meerdere data items wordt toegepast.

Om mijn ontwerp duidelijk te maken, heb ik drie ontwerpen gemaakt. Eentje voor 2 threads (figuur 2), vier threads (figuur 3) en 8 threads (figuur 4). In elk van de afbeeldingen is alles wat rode is de main thread, deze wordt niet meegeteld omdat deze altijd nodig is. Alle andere kleuren representeren een eigen thread. Het ontwerp werkt als volgt:

1. De lijst wordt gesplitst tot het aantal threads (2 threads = lijst één keer splitsen, 4 threads = lijst twee keer splitsen etc.). Dit gebeurd door de main thread. De data moet van te voren worden gesplitst omdat we met een thread pool werken.
2. Elke thread heeft nu dus zijn eigen lijst om te merge sorten.

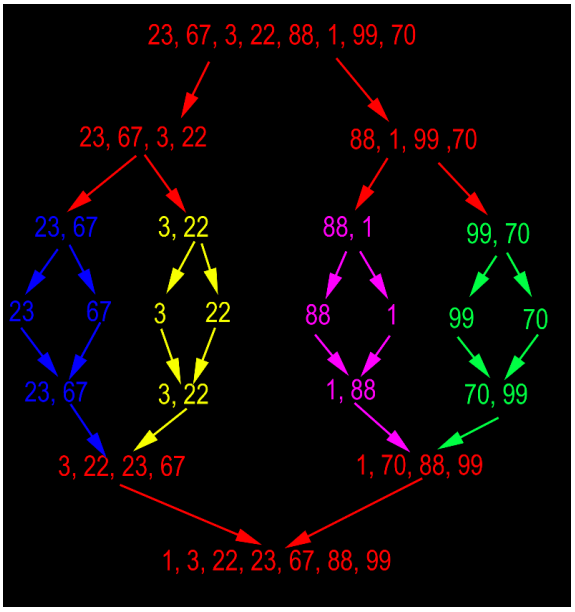
3. Als alle threads klaar zijn wordt alles door de main thread gemerged.

Deze afbeelding laat zien hoe een threaded merge sort zou werken met 2 threads. Beide threads krijgen dan de helft van de data om te merge sorten.



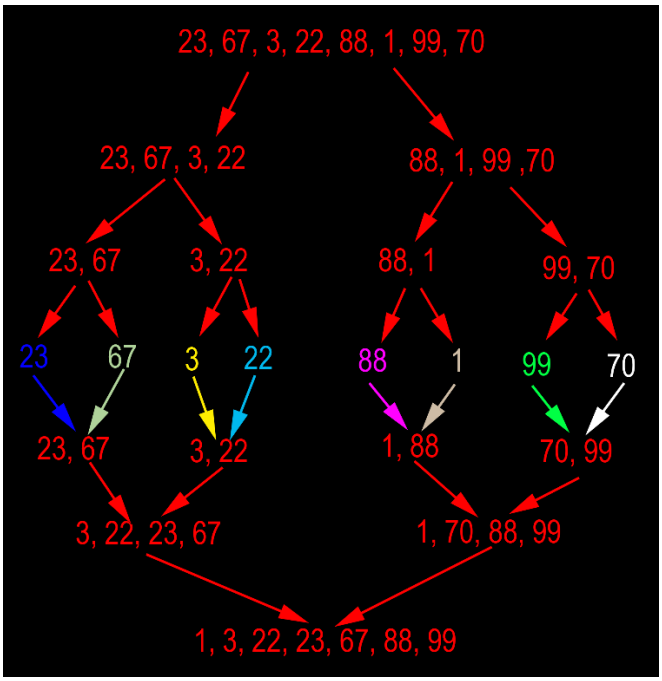

*Figuur 2*

Deze afbeelding laat zien hoe threaded merge sort zou werken met 4 threads. Zoals je kan zien deelt de main thread eerst de data in vier stukjes. En aan het einde merged de main thread de resultaten van alle threads weer. Wat al direct op valt is dat hoe meer threads er zijn, hoe meer er ook op de main thread gedaan moet worden om te zorgen dat de juiste data in de threads gaat en dat de data juist uit de threads wordt gehaald.




*Figuur 3*

Als laatste heb ik hier threaded merge sort ontworpen met 8 threads. Zoals je kan zien doet elke thread eigenlijk niets. De resultaten van de threads worden gelijk al weer overgeplaatst naar de main thread.




*Figuur 4*

## Analyse

Merge sort heeft normaal gesproken een complexiteit van *O(n log(n))*. Nu is de vraag heeft threading impact op deze complexiteit? Om dit te kunnen bepalen is het handig om te begrijpen wat *O(n log(n))* inhoudt. De lijst met data wordt telkens gehalveerd, hier komt het *log(n)* gedeelte vandaan. Het *n* gedeelte komt van een lineair zoeken op iedere laag gemaakt door *log(n)*.

Als we aannemen dat *p* threads precies *p* dingen tegelijk kunnen doen, dan zou je zeggen dat de complexiteit *O((n log(n) / p)* wordt. Deze beredenering valt echt al snel door de mand, zoals te zien is in mijn ontwerpen: hoe meer threads, hoe minder er tegelijk gedaan wordt. Dit klinkt misschien raar, maar is toch logisch, als er 8 threads zijn en een lijst met 8 items (zie figuur 4) dan doet elke thread eigenlijk niets. Terwijl als dezelfde lijst wordt gesorteerd met 2 threads (zie figuur 2), doet elke threads veel meer. Dit zou suggereren dat er een sweet spot is voor de verhouding lengte van de lijst en aantal threads.

Hoe parellel er gesorteerd kan worden hangt dus af van de hoeveelheid threads. Hoe meer threads hoe minder parallel er gesorteerd kan worden. Met een oog op het ontwerp zou er van een complexiteit uitkomen van *O(n(log(n) * p)*, dit zegt dus dat hoe meer minder threads er zijn, hoe sneller het algoritme wordt. Of dit in de praktijk ook zo werkt is echter de vraag.

## Limiet

Tot nu toe hebben we veel aannames gedaan over hoe snel het algoritme zou zijn met *p* aantal threads en wat de complexiteit is. Echter denk ik dat dit weinig waarde heeft. Dit heeft een aantal reden die we nu gaan behandelen:

### Global Interpreter Lock (GIL)

We hebben hier mee te maken omdat we in Python programmeren. Python is eigenlijk helemaal niet goed in multithreading, dit komt door deze GIL. De GIL zorgt ervoor dat maar 1 thread tegelijk gebruik kan maken van de interpreter. Dit zou betekenen dat allen alsnog sequenteel zou gebeuren, ondanks dat we gebruik maken van threading.

Hier zijn een aantal uitzonderingen op, deze uitzonderingen zijn vooral op het gebied van IO en data verplaatsen. Deze uitzonderingen hebben soms effect zoals later te zien zal zijn.

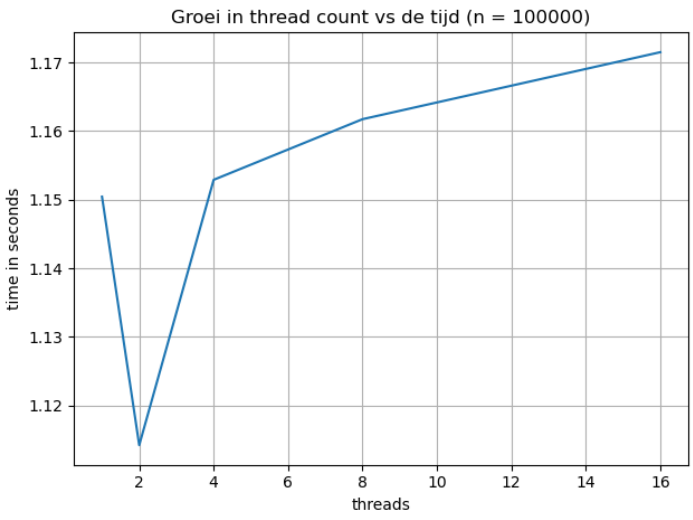
### Overhead

Naast de GIL hebben we ook te maken met overhead. Threads aanmaken kost ook tijd en ruimte. Dit heeft ook weer te maken met de hoeveelheid threads. Als we weer hetzelfde voorbeeld nemen als eerst met 8 threads en 8 elementen in een lijst, is er weinig ruimte nodig voor de threads om te communiceren met de main thread. Elke thread heeft toch maar een lijst van één lang. Als we de zelfde lijst zouden sorteren met 2 threads is er meer overhead nodig voor ruimte, de main thread moet opeens vier elementen naar elke thread verplaatsen en er ook weer uithalen. Wel moeten er een stuk minder threads aangemaakt worden, dat bespaard weer tijd.

## POC

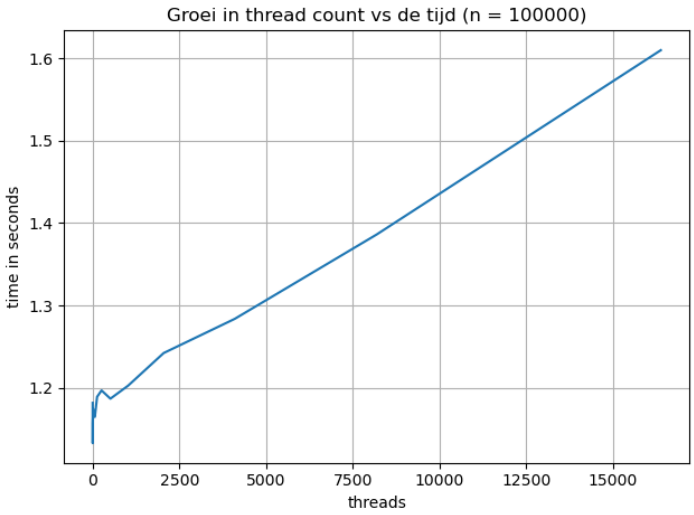
Nu we het ontwerp hebben geanalyseerd is het tijd om het daadwerkelijk te maken. De code is te vinden op GitHub (https://github.com/FlorisVideler/HPP/tree/main/Threaded%20Merge%20Sort), voor alle data is gebruik gemaakt van een aantal threads dat *2^p* is (1 alleen main thread, 2, 4, 8 etc.). **macht van 2**

In figuur 5 is te zien hoe mijn threaded merge soort presteert. Wat direct op valt is dat met 2 threads merge sort wat sneller is. Mijn vermoeden is dat komt door dat er met 2 threads het meeste werk uit handen wordt genomen van de main thread. Na 2 threads gaat de prestatie snel achteruit en is het verstandiger om maar 1 thread te gebruiken.




*Figuur 5*

Dit wordt pas echt duidelijk als we naar het extreme toe gaan. In figuur 6 is geplot tot *16.384* threads. Het resultaat is een bijna lineair verband tussen het aantal threads en hoelang het duurt om de lijst te sorteren.




*Figuur 6*

## Conclusie

Al met al een hele interessante opdracht waar ik veel nieuwe dingen heb geleerd over de mogelijkheden multi threading en de limieten ervan. In dit specefieke voorbeeld worden vooral de limieten van multithreading duidelijk, dit heeft vooral te maken met de GIL. Ook hier is wel een weg omheen, door bijvoorbeeld gebruik te maken van de multiprocessing python module, deze omzeilt de GIL door ieder process een eigen interpreter te geven.

Zonder deze module is mijn implementatie niet heel nuttig. Wel was het nuttig om na te denken over hoe mijn ontwerp invloed heeft op tijd en ruimte en hoe threads hier invloed op hebben.