Threaded Merge Sort

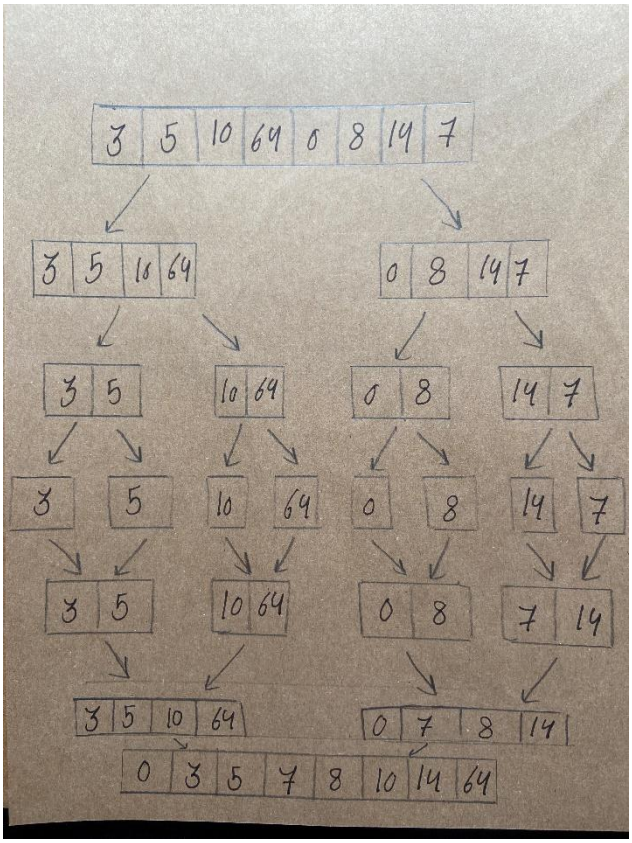
Maria Dukmak

11-3-2021

---

1. Ontwerp

Voordat we beginnen met het analyseren van de threaded merge sort lijkt me handig om eerst kort uit te leggen hoe het algoritme merge sort werkt.



Zoals we hier naast zien, de werking van een merge sort gaat als volgt:

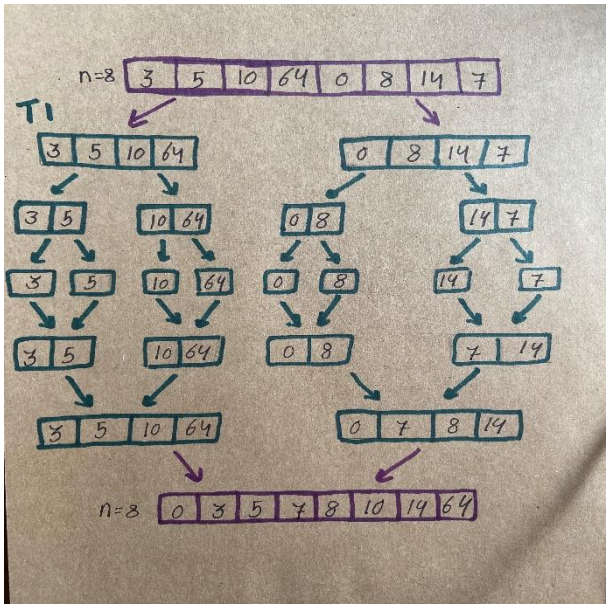
- De unsorted gegeven lijst wordt in kleinere lijsten opgesplitst.
- Vervolgens worden deze lijsten apart gesorteerd.
- En de laatste stap is het samenvoegen van de gesorteerde lijsten.

Dit algoritme kan parallel gerunt worden door het toevoegen van threads of processen.

In mijn implementatie gaan we ook het verschil van de runtime van het algoritme met threads en processen.

Hieronder heb ik er 4 ontwerpen gemaakt van hoe het algoritme gerunt zou worden als we gebruik maken van [1, 2, 4, 8] threads/processen.

1.1 (1) thread/proces



Hiernaast zien we een ontwerp op papier van het werking van het algoritme wanneer we er gebruik van een enkele thread/proces maken.

De main thread/proces wordt in het paars aangeduid en thread 1 staat in het groen.

Het gaat hier om Single Instruction Single Data. Dat betekent dat er op een moment alleen 1 instructie kan uitgevoerd worden. De instructies worden dus 1 voor 1 uitgevoerd.

De data wordt niet over meerdere/andere threads/processen verdeeld. Daarom is dit een SISD proces.

1.2 (2) threads/processen

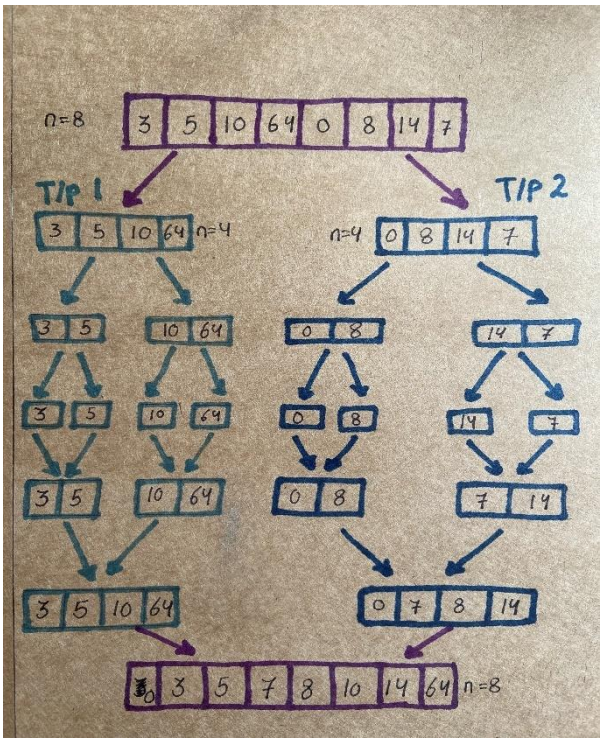
In deze afbelding kunnen we zien hoe de thread merge sort met 2 threads/processen gaat werken.

In tegenstelling tot vorig ontwerp, wordt de data hier door de main thread/proces(in het paars) opgesplitst ter voorbereiding voor de twee threads.

Hier wordt dus gebruik van Single Instruction Multiple Data gemaakt. De data(de input lijst) wordt in twee sublijsten opgesplitst. Deze sublijsten worden tegelijkertijd gesorteerd.

Wel worden de instructies 1 voor 1 uitgevoerd.

Aan het einde mergt de main thread/proces de resultaten tot 1 lijst.

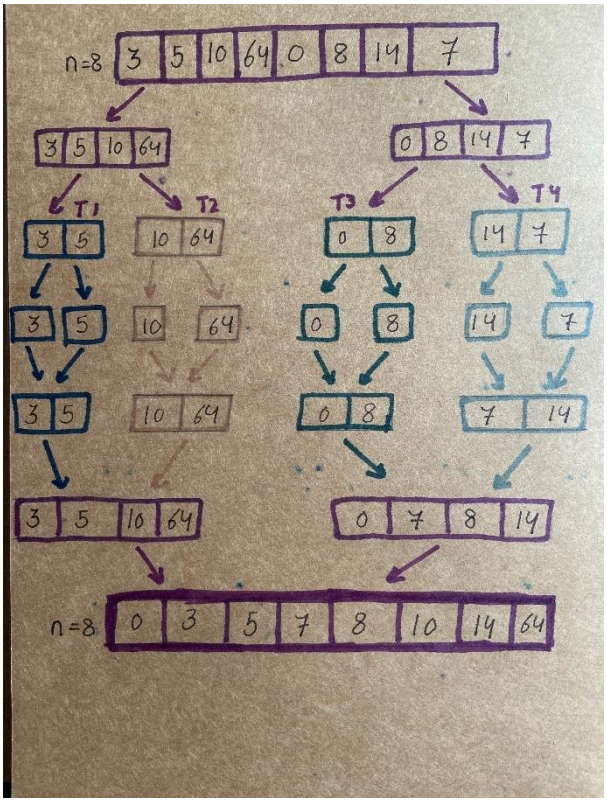


1.3 (4) threads/processen

Hier wordt de data door de main thread/proces tot 4 sublijsten verdeeld.

Elke thread krijgt dus een lijst mee die er gestorteed moet worden. Dit proces is dus ook van Single Instruction Multiple.

Aan het einde zien we dat de main thread weer alle sublijsten tot 1 lijst mergt.

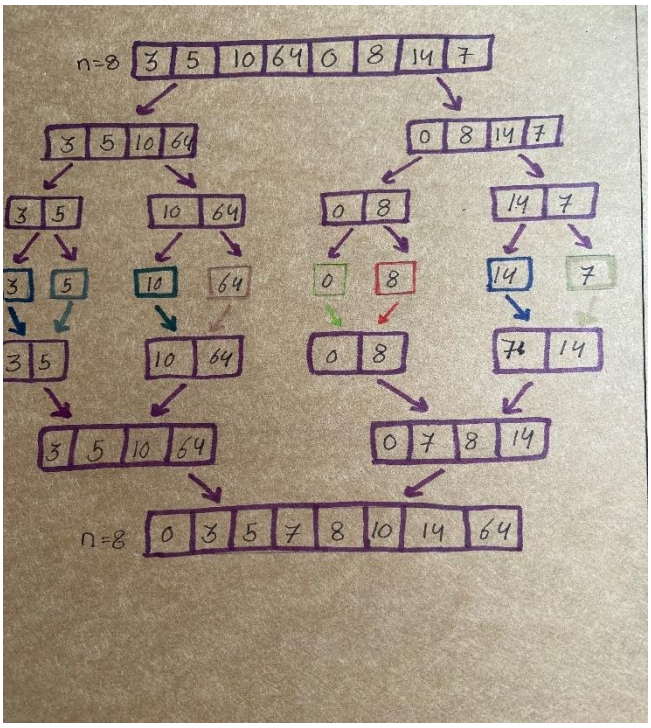


1.4 (8) threads/processen

Hier naast zien we het ontwerp voor 8 threads/processen. Zoals we kunnen zien elke thread/proces krijgt maar een lijst van lengte 1 om te sorteren. Dat heeft eigenlijk geen zin. Aangezien de resultaten van de threads alsnog door de main thread moeten gemerged worden.

Dit blijft wel Single Instruction Multiple architectuur.

De meeste werkt wordt hier door de main thread gedaan. Als we ook naar (4) threads/processen gaan kijken zien we dat de main thread meer moeten doen dan bij (1)of (2) threads. Dat komt omdat de main thread moet zorgen dat alle threads de juiste lijsten mee moeten krijgen.



---

2. Analyse

Voordat we de complexiteit van de threaded merge sort gaan bepalen. Laat we een kijkje nemen naar de complexiteit van een normale merge sort die is gelijk aan O(n log(n)).

Stel je voor dat het aantal threads/processen die gemaakt worden is *p* dan zouden we kunnen zeggen dat de complexiteit O(n log(n)/p) is. Omdat we bijna alle werk parallel kunnen maken.

Echter is het soms niet het geval, neem het aannamen van 8 threads/processen voor een lijst van 8 elementen als voorbeeld. Het gebruiken van 8 threads heeft daar geen zin. Aangezien elke thread daar niets doet.

Deze complexiteit hangt dus ook van de hoeveelheid threads/processen en de lengte van de lijst af. Dus de complexiteit kan in sommige gevallen O(n log(n) * p) worden.

2.1 Runtime analyse

We gaan het algoritme met multithreading en multiprocessing runnen. We laten het algoritme lijst van de volgende lengtes runnen [10000, 100000, 300000] .

We zien dat de runtime van multiprocessing toeneemt bij het toename van het aantal processen. Dat is namelijk niet helemaal te verwachten. Echter kunnen deze resultaten door veel dingen beïnvloed worden. Zoals het I/O bound instructies die ook parallel uitgevoerd worden. Maar dit alleen een aanname.

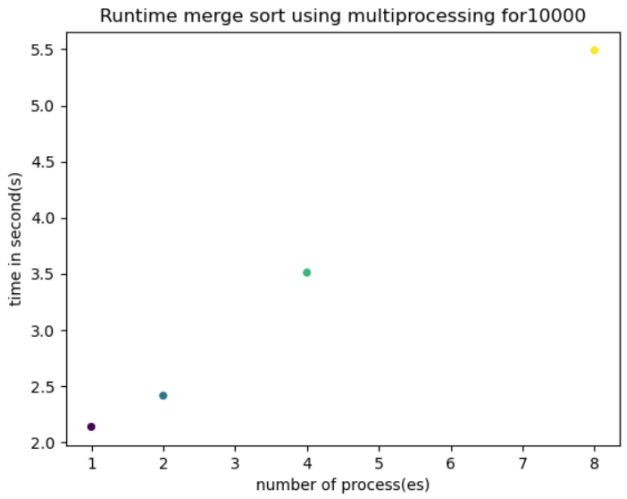
Bij multithreading gaat het ook ongeveer hetzelfde. Bij 8 threads neemt de tijd een stuk toe. Soms is de runtime met multithreading korter dan multiprocessing. Wat niet te verwachten is.

De reden waarom dit het geval is weet ik niet zeker, maar het kan door meerdere oorzaken komen zoals:

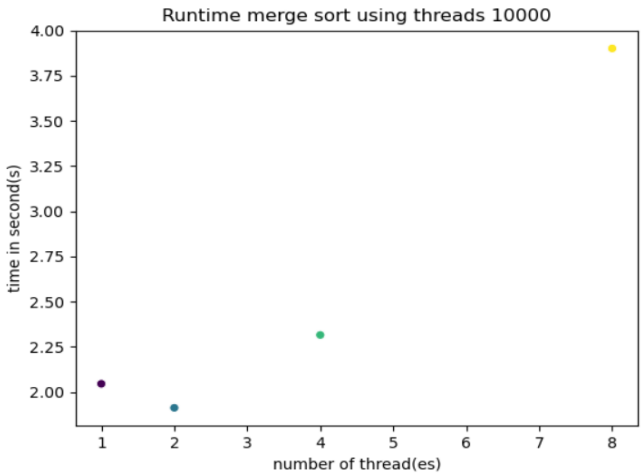
- Het aanmaken van de processen en het joinen/mappen daarvan kost tijd.

- Het opsplitsen van de lijst in sublijsten kost ook tijd.

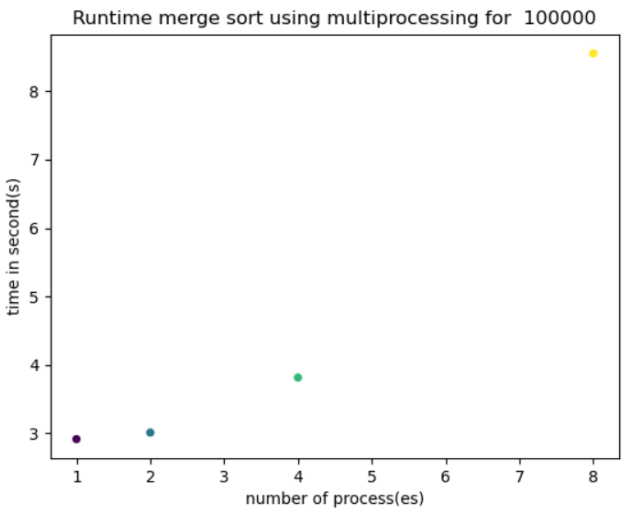
- Of dat het mergen van de gesorteerde sublijsten tijd kost aangezien die op de andere processen moet wachten.



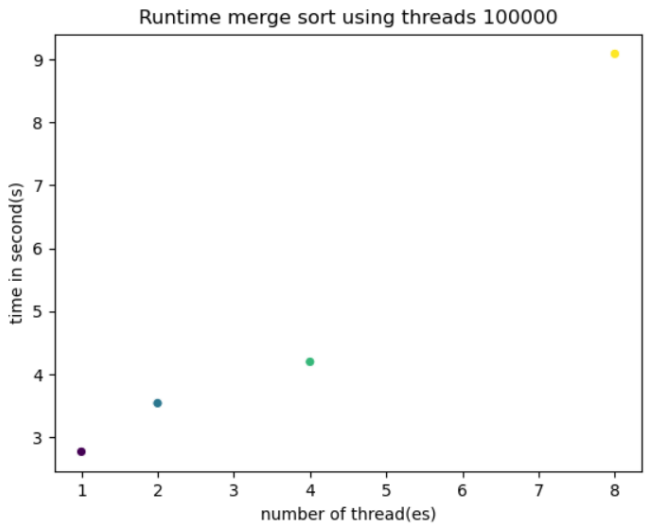
Figuur 1



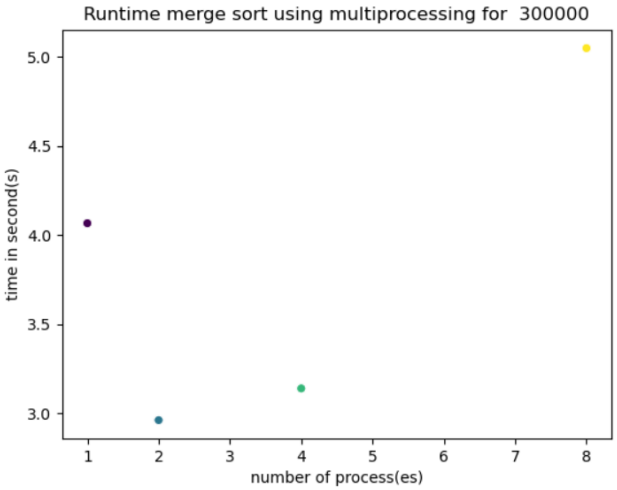
Figuur 2



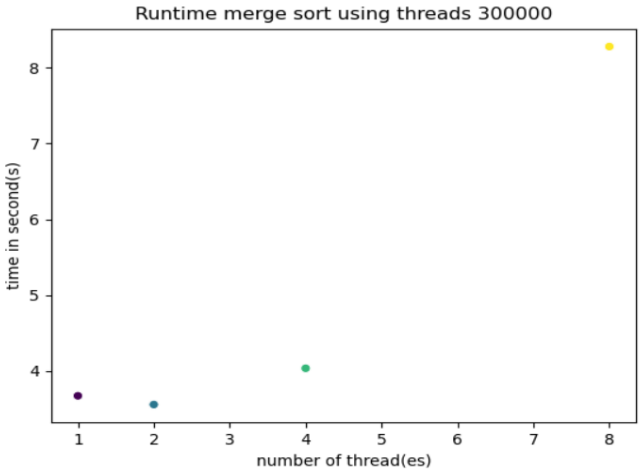
Figuur 3



Figuur 4



Figuur 5



Figuur 6

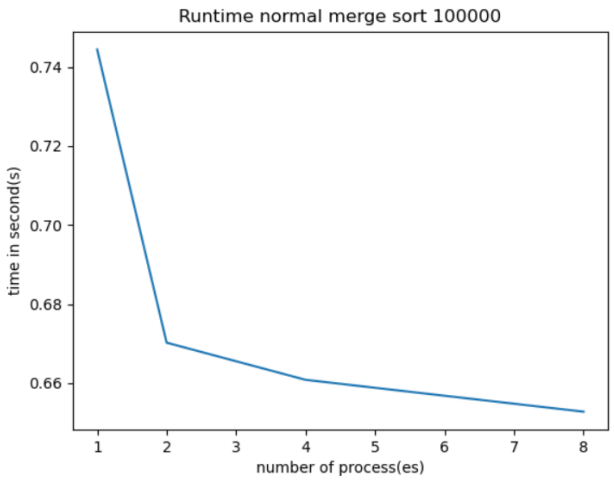
2.3 Communicatie overhead

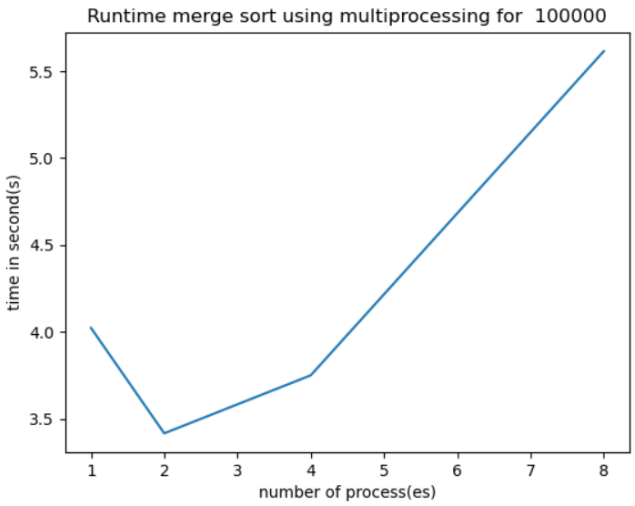
Het aanmaken van threads en processen kost zowel ruimte als tijd. Dit is afhankelijk van het aantal threads/processen die aangemaakt moeten worden. De communicatie stappen komen tussen de main en threads/processen. Bij elke extra aangemaakte thread kommen er twee communicatie stappen. Dat komt omdat de main thread de lijst moet meegeven aan de thread/proces en de thread dat terug moet geven aan de main om de lijsten te mergen. Kortom bij elke thread vermenigvuldigen we met 2 voor het bereken van de communicatie stappen.

2.4 Verschillen met normale merge sort

Hieronder gaan we de runtime vergelijken van een normale merge sort vergelijken met een multiprocessing versie.

Zoals we zien, het verschil is duidelijk. Het toevoegen van threads/processen heeft een groote invloed op de runtime van het algoritme.





Extra: GIL

Python is een taal die niet goed multithreading kan runnen. Dat komt door de GIL. Die zorgt ervoor dat alleen maar 1 thread tegelijk gebruik kan maken van de global interpeter. Dat betekent dus dat bijna alles sequenteel verloopt bij multithreading. Dit wordt ook soms gezien als een slechte feature van Python.

Een GIL is handig is handig voor het management van de programmer taal, memory leaks te voorkomen of andere mogelijke bugs. Alleen dat heeft in het geval van multithreading een slechte invloed op het parallel runnen van de threads.

Echter is het gebruik maken van multi-processing een oplossing hiervoor. Aangezien elke Python process krijgt zijn eigen interpreter and memory space.

Extra: worker thread

Hier worden er eerst twee threads aangemaakt vervolgens kan er een nieuwe/extra thread aangemaakt worden wanneer er meerdere peocessen uitgevoerd moeten worden in 1 thread. Op deze manier kan er voor gezorgd worden dat de threads sneller en efficienter kunnen werken. Zoals je in het plaatje hieronder kunt zien.