Threaded Merge Sort

1. Ontwerp:

Ik heb ervoor gekozen om te werken met een multiprocessing pool. Een multiprocessing pool beheert de threads en resultaten. Er hoeft dus geen eigen functie voor het terughalen van resultaten uit threads etc. geschreven te worden. Voor het toevoegen van processen wordt de functie apply_async gebruikt. Er kan gekozen worden voor andere functies maar dit is per situatie en module verschillend. De apply functie bied de mogelijkheid om met een gegeven functie steeds iets te doen met de resultaten die in de thread worden uitgevoerd. In mijn geval om de resultaten van de merge functie in de resultaten op te slaan per uitgevoerd proces.

Als eerst wordt de te sorteren lijst in stukken opgedeeld. Het aantal stukken waarin het opgedeeld wordt is gelijk aan het aantal gewenste threads. In mijn code implementatie worden de stukken gesneden tijdens het toevoegen aan de multiprocessing pool, maar om het iets duidelijker te maken heb ik het anders gedaan in de pseudocode:

Pseudocode:
Ik heb de pseudocode grotendeels in eigen woorden beschreven omdat de code half kopieren het niet veel duidelijk maakt.

Input: te sorteren lijst
Output: gesorteerde lijst

Optioneel: gebruik time module om start tijd vast te leggen
Creeer een pool manager en een resultatenlijst om de resultaten bij te houden

1e deel:
Deel eerst de input op in het gewenste aantal stukken, waarbij dit aantal het aantal threads is.

Start een nieuwe pool en stop elk stuk in een apart proces in de pool, waarbij een functie wordt meegegeven om de resultaten van het merge algoritme steeds in op te slaan in de resultatenlijst. (in het geval van multiprocessing)

Sluit de pool en wacht tot alle processen uitgevoerd zijn. Als dit het geval is, hebben we nu een lijst met gesorteerde lijsten.

2e deel:

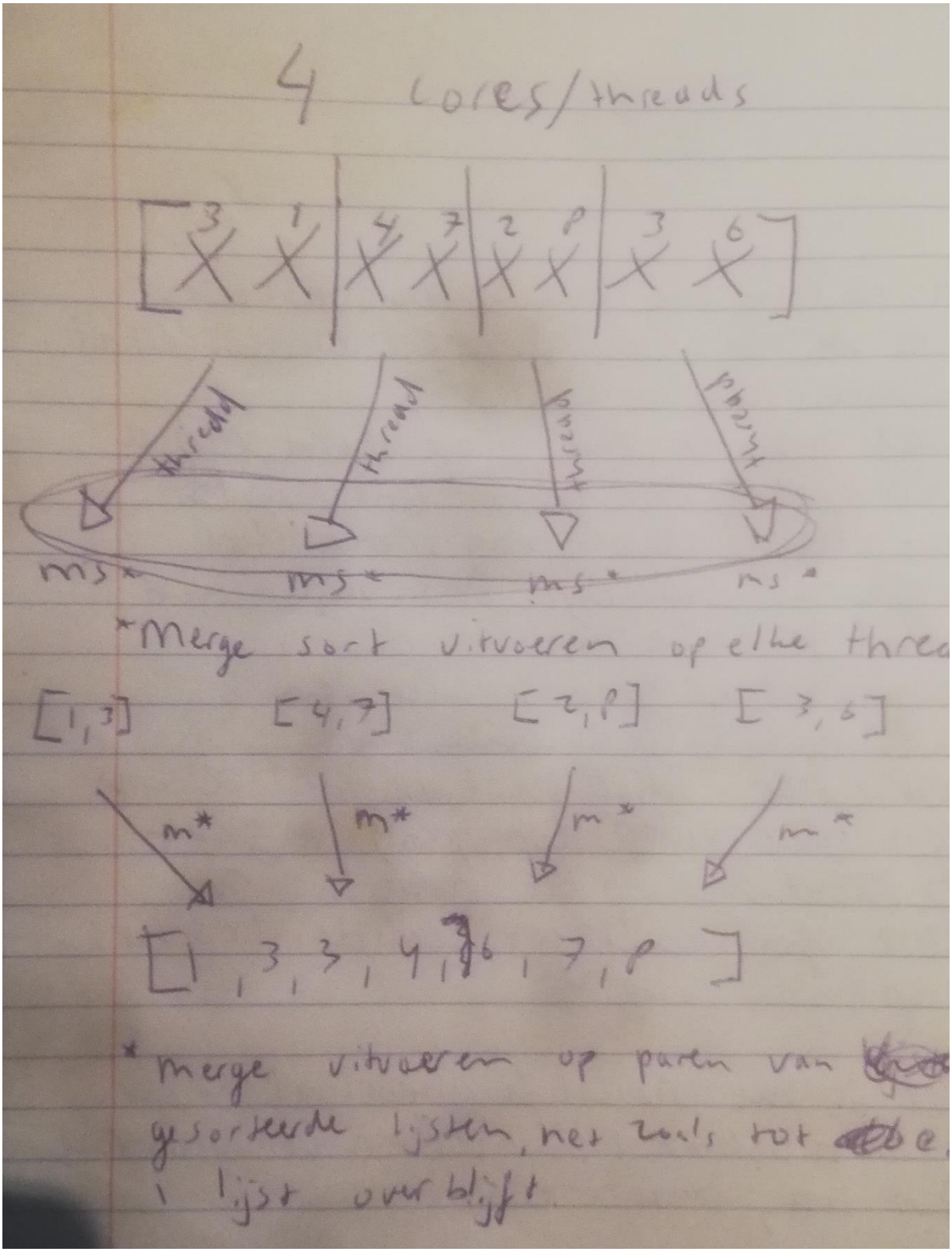
Gebruik nogmaals een pool om de gesorteerde lijsten samen te voegen tot een gesorteerde lijst. Zolang er meer dan 1 lijst in de resultatenlijst zit, start een nieuw merge proces in de pool met 2 lijsten uit deze lijst, om 2 lijsten tot 1 gesorteerde lijst samen te voegen. Voor dit merge proces wordt de merge functie van de recursieve variant van het merge algoritme gebruikt.

Neem nu het eerste en enige element uit de lijst en dit is de gesorteerde lijst.

Optioneel: sla de runtime op door de start tijd van de eindtijd af te trekken.

Tekening:

Aangezien het aantal cores/threads gewoon betekent in hoeveel stukken de lijst wordt opgedeeld het ik het proces voor 4 threads uitgetekend, het voegt weinig toe om dit voor meerdere threads uit te tekenen.



2. Complexiteit:

De tijdscomplexiteit van het algoritme blijft bijna hetzelfde, O(n log n). Er wordt immers hetzelfde algoritme uitgevoerd als bij de normale non-threaded merge sort. Het enige wat mogelijk extra tijd toevoegt, wat niet bij de non-threaded merge sort hoort is de laatste stap van het threaded merge sort algoritme. Als alle opgesneden lijsten zijn gesorteerd moeten deze worden samengevoegd tot 1 gesorteerde lijst. Dit is een extra stap en heeft dezelfde tijdscomplexiteit, O(n log n). Dit komt omdat deze stap gebruik maakt van hetzelfde merge algoritme . Deze laatste stap kan ook threaded uitgevoerd worden zoals beschreven in de code en pseudocode.

Wel zien we met deze threaded manier van sorteren een toename bij ruimte complexiteit. Er wordt meer gebruik gemaakt van geheugen door de laatste extra stap van het threaded algoritme, en daarnaast gebruikt de pool ook geheugen. Het verschil in tijd zal vooral liggen in hoe snel en goed de threads samen uitgevoerd kunnen worden, en welke andere processen op de machine al draaien.

3. Communicatie processor:

Aangezien we de lijst opsplitsen in een opgegeven aantal threads en vervolgens het algoritme x keer zovaak uitvoeren zal de communicatie ook x keer zoveel toenemen. Voor mijn gevoel is de toename van communicatie evenredig aan het aantal stukken waarin de lijst worden opgedeeld. Ik kan dit echter niet met zekerheid zeggen, vanwege het feit dat ik gebruik heb gemaakt van de multiprocessing pool module, in plaats van het zelf uitschrijven van threads en gedrag. Mogelijk heeft de module een slimme manier om communicatie overhead tegen te gaan.

4. Resultaten:

De resultaten van mijn tests waren voornamelijk afhankelijk van de grootte van de lijst. Ik kwam erachter dat met een lijst onder de 100000 items de runtime van non-threaded merge sort altijd sneller is dan een threaded variant. Het kost Python een bepaalde tijd om de pool met threads op te zetten, en dit tijd kan zomaar een kwart seconde zijn. Daarnaast kwam ik er ook achter dat Python niet echt parallel code threads kan uitvoeren. Dit zorgde ervoor dat de threaded variant vaak trager was. Toch zal de lijst sneller gesorteerd zijn met hele grote lijsten. Omdat dit veel tijd kost staat dit niet in de resultaten. Maar met een lijst vanaf 500.000 items kan het threaden in python sneller zijn. komt omdat multiprocessing slimme processor trucjes uithaalt om processen naast elkaar te kunnen uitvoeren. Hieronder is een geknipte kopie te zien van de jupyter notebook resultaten:

(0 threads betekent dat het merge sort algoritme non-threaded wordt uitgevoerd.)

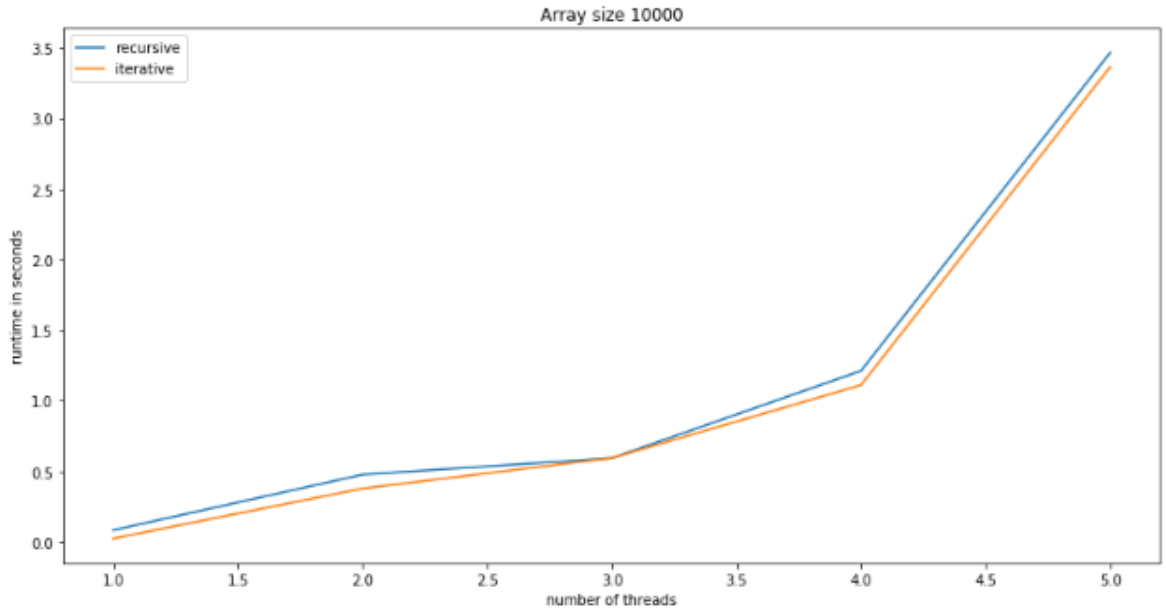
```
0 thread(s):
                Timing in seconds (iterative): 0.02324709999999186 | Verify: True
                Timing in seconds (recursive): 0.08263010000000293 | Verify: True

1 thread(s):
                Timing in seconds (iterative): 0.37805843353271484 | Verify: True
                Timing in seconds (recursive): 0.4763798713684082 | Verify: True

2 thread(s):
                Timing in seconds (iterative): 0.5940005779266357 | Verify: True
                Timing in seconds (recursive): 0.5929794311523438 | Verify: True

4 thread(s):
                Timing in seconds (iterative): 1.110999584197998 | Verify: True
                Timing in seconds (recursive): 1.2119994163513184 | Verify: True

8 thread(s):
                Timing in seconds (iterative): 3.3661444187164307 | Verify: True
                Timing in seconds (recursive): 3.4677205085754395 | Verify: True
```



```
0 thread(s):
                Timing in seconds (iterative): 0.284585600000014 | Verify: True
                Timing in seconds (recursive): 1.0182525999999825 | Verify: True

1 thread(s):
                Timing in seconds (iterative): 0.7873365879058838 | Verify: True
                Timing in seconds (recursive): 1.0789270401000977 | Verify: True

2 thread(s):
                Timing in seconds (iterative): 0.9119818210601807 | Verify: True
                Timing in seconds (recursive): 1.0316827297210693 | Verify: True

4 thread(s):
                Timing in seconds (iterative): 1.5860052108764648 | Verify: True
                Timing in seconds (recursive): 1.7765417098999023 | Verify: True

8 thread(s):
                Timing in seconds (iterative): 4.069053411483765 | Verify: True
                Timing in seconds (recursive): 3.8426828384399414 | Verify: True
```