**Ontwerp:**

Het basis idee is dat ik de data verdeel over de beschikbare threads, deze vervolgens lokaal sorteer via de merge sort algoritme zodat elke thread een gesorteerde lijst teruggeeft. Deze sorteer ik vervolgens met elkaar onderling door steeds i en i + 1 te sorteren waarbij i de index is van de gesorteerde lijst die we bekijken.

Het geen wat ik geparallelliseerd heb is de divide and conquer strategie van merge sort verdeeld over n stukken waarbij n het aantal threads zijn. Het sorteren van deze gesorteerde lijsten doe ik overigens ook multithreaded maar deze worden plaats van n keer parallel gedraaid, n / 2 keer parallel gedraaid en dit herhaal ik tot n <= 1.

**Analyse:**

De resultaten die we zijn te verwachten we zien dat de single threaded optie sneller is bij weinig data punten maar wanneer we dit opschalen zien we dat de multithreaded versie beter is hierin. Dit komt omdat bij weinig getallen het aanmaken van threads meer tijd verliest dan oplevert. (Dit is dezelfde conclusie als bij P4)

Multithreading zal in python weinig tot geen tijdwinst opleveren (soms zelf tijdverlies) vanwege de Global Interpreter Lock (GIL). Dit komt omdat de GIL ervoor zorgt dat alles alsnog single thread wordt omdat elke thread moet wachten tot de lock is vrij gegeven en de interpreter gebruikt kan worden. De GIL was in de eerste instantie gemaakt om fenomenen als race condities tegen te gaan.

Daar in tegen gebruik ik de multiprocessing module, deze maakt gebruikt van meerdere processen die elke zijn eigen instantie van de interpreter en memory space toegediend krijgen, dit maakt het versnellen van applicaties alsnog mogelijk.

Een reden waarom programeer talen die dichterbij de CPU liggen sneller kunnen zijn is omdat je meer controle hebt over memory allocation. Je kiest er zelf voor wanneer je een memory space vrij maakt, python daar in tegen regelt dit allemaal zelf. Een term die wordt gebruikt om dit aan te geven is ‘garbage collection’, python is dus een ‘gargbage collection’ programmeer taal het ruimt alle onnodige memory op achter zichzelf maar dit kan soms voor ongewenste situaties zorgen.

Stel je maakt een CPU call met een aantal variabelen dan zal het waarschijnlijk van de l2 cache gelezen worden en via de bus overgestuurd worden naar de CPU voor de instructies, maar het kan voorkomen dat dit meerdere cpu cycles nodig heeft omdat het van de l2 cache verwijderd was en was opgeslagen in de ram. In een programmeer taal die dichterbij de CPU werkt zoals C kan je dit zelf bepalen en kan je er dus voor zorgen dat (als de benodigde bandwith aanwezig is) alle data wordt overgestuurd naar de CPU in een/minder cycles.