Project Algoritmen en Datastructuren 3: Genetische Algoritmen en MPI

Jan Vermeulen

27 november 2014

---

1 Sequentiele Genetische implementatie

1.1 Beschrijving

1.1.1 Configuratie

De sequentiële implementatie kan geconfigureerd worden met de volgende extra parameters:

- Het aantal iteraties
- De populatiegrootte
- De maximale stap in x en y-richting bij mutatie
- Het aantal crossovers
- De kans op mutatie

De optimale parameters worden verder in dit verslag weergegeven.

1.1.2 Initiele populatie

Om een initiele populatie te genereren wordt de n-hoek opgedeeld in $n-2$ driehoeken, waarvan vervolgens de oppervlakte van berekend wordt. Van deze oppervlakten wordt een cumulatieve array gevormd. Om een random punt te genereren op uniforme wijze wordt een getal tussen 0 en de som van de oppervlakten gegenereerd. Aan de hand van de cumulatieve array en dit random getal selecteren we een driehoek. In deze driehoek met hoekpunten $P1$, $P2$ en $P3$ kiezen we een random punt (met $R1, R2 \in [0, 1]$):

```
(1 - sqrt(R1)) * P1 + (sqrt(R1) * (1 - R2)) * P2 + (sqrt(R1) * R2) * P3
```

Dit wordt uitgevoerd voor elk punt in individu in de populatie.

1.1.3 Datatypen

De veelhoek wordt voorgesteld door een `struct` met een `integer` die het aantal hoeken bijhoudt, alsook een array van `float` types die de coördinaten van de hoeken voorstelt.

De `populatie` is slechts een array van `individu`en. Een individu wordt voorgesteld met een `struct` die de fitness en het aantal punten bijhoudt, alsook een array van `Point`s.

Een `Point` is slechts een `struct` met 2 `float`s.

1.1.4 Bepalen of punt binnen veelhoek ligt

Om te bepalen of een punt P in de convexe veelhoek ligt worden de coördinaten van de veelhoek verminderd met P in x en y-richting. Vervolgens werd gebruik gemaakt van de volgende eigenschap:

P ligt in veelhoek met punten $(x_0, y_0), (x_1, y_1), ..., (x_{n-1}, y_{n-1}) \Leftrightarrow$
$(\forall i \in [0, n-1] \; . \; x_{i+1(mod\ n)} * y_i - x_i * y_{i+1(mod\ n)} > 0) \vee$
$(\forall i \in [0, n-1] \; . \; x_{i+1(mod\ n)} * y_i - x_i * y_{i+1(mod\ n)} < 0)$

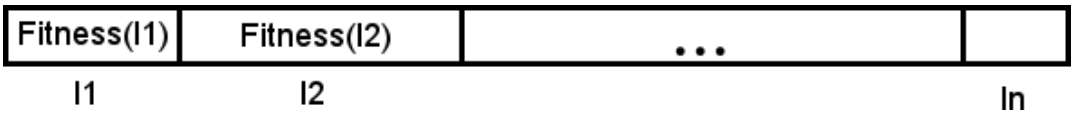
1.1.5 Crossover en mutatie

Cross-over wordt uitgevoerd aan de hand van de opgegeven parameter `num_crossovers`. Er worden dus in elke iteratie `num_crossovers*2` kinderen gegenereerd. Ook bij de mutatie wordt rekening gehouden met de parameters `max_mutation_step` (maximale stap in x en y-richting) en `mutation_prob` (mutatiekans).

1.1.6 Selectie

In elke iteratie worden 2 selecties uitgevoerd. Als eerste worden ouders gekozen om kinderen mee te vormen. Na de mutatie van deze kinderen wordt nog een selectie uitgevoerd waarin de individuen van de volgende populatie worden gekozen. Deze selectie wordt uitgevoerd met een techniek die stochastic universal sampling heet.

Initieel wordt een array opgebouwd met de cumulatieve fitness van alle individuen van de populatie. Men kan dit visualiseren als volgt:



Tussen 0 en de totale fitness wordt dan een willekeurige offset gekozen. Er wordt een afstand gedefinieerd

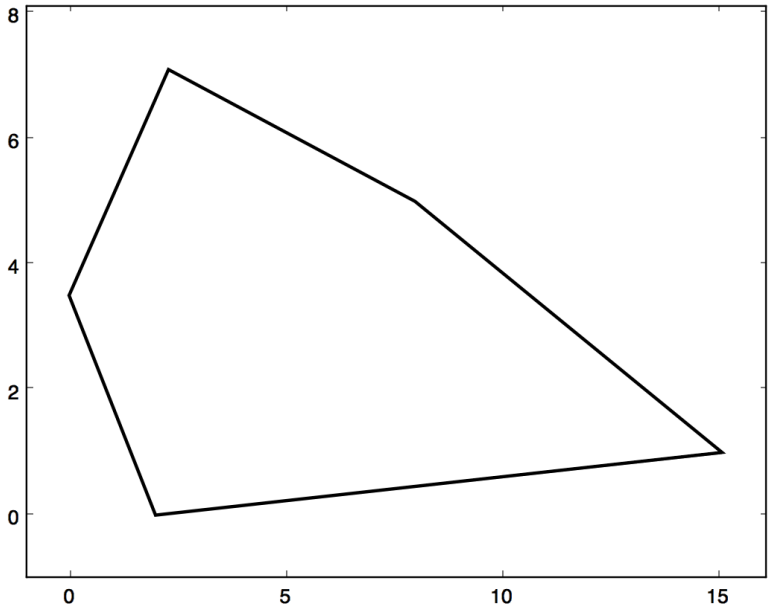
```
distance = total_fitness / n
```

waarbij n het aantal elementen is dat je wil kiezen. Op deze manier worden individuen gekozen die op punten `offset+i*distance` op deze as liggen (met `i=0,1,...n`). Punten met een grotere fitness hebben duidelijk een grotere kans om gekozen te worden.

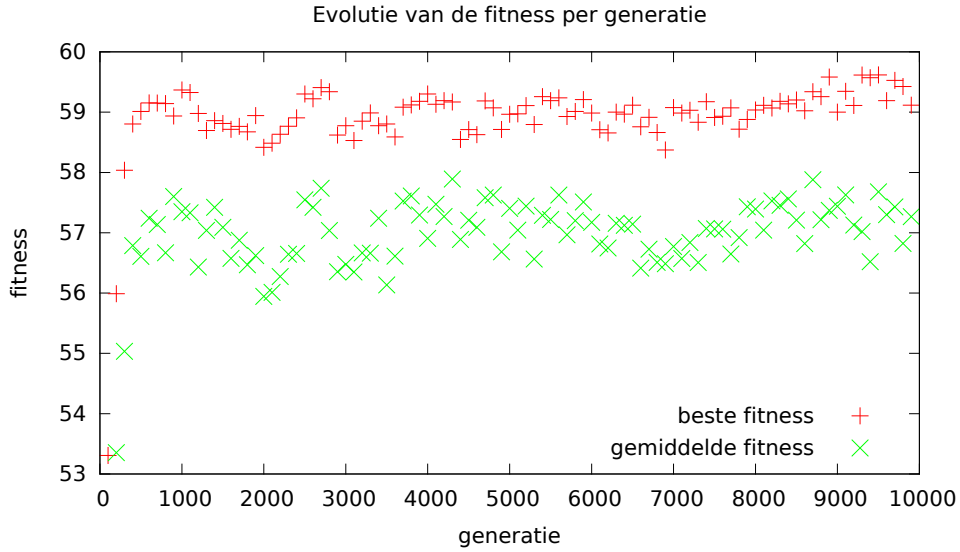
1.2 Testinput

In de hieropvolgende metingen zal steeds gebruik gemaakt worden van de volgende convexe veelhoek:

5
2.0 0.0
0.0 3.5
2.3 7.1
8.0 5.0
15.1 1.0



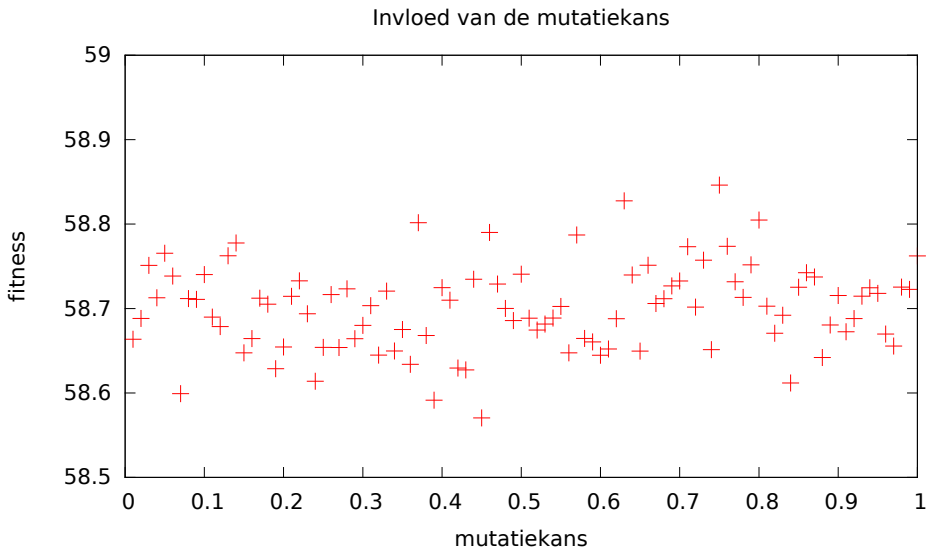
1.3 Evolutie van de fitness per generatie



Bij de eerste 300 iteraties stijgt de gemiddelde en beste fitness zeer snel. Naarmate de fitness het optimum nadert vertraagt deze stijging. Vervolgens zien we een stagnatie met een licht stijgende trend. Voor deze meting werden de volgende parameters gebruikt:

- population_size=200
- max_mutation_step=0.9
- num_crossovers=100
- mutation_prob=0.5

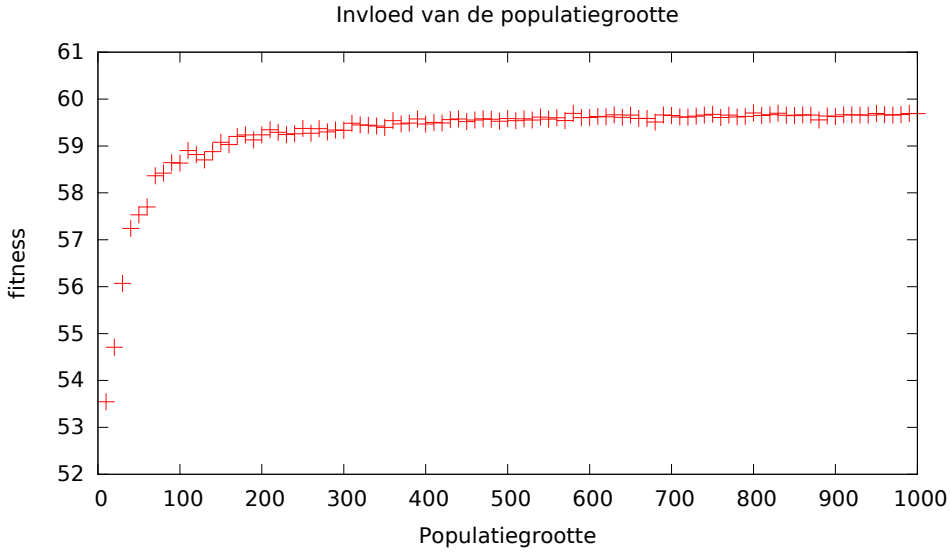
1.4 Invloed van de mutatiekans



Hoewel voor deze meting gemiddelden van 30 uitvoeringen werden beschouwd, is er niet echt een algemene trend af te lezen uit deze data. De hoogste fitness werd in deze metingen bereikt met een mutatiekans van 0,75. Er lijkt echter geen sterk verband te zijn tussen de mutatiekans en de fitness van het beste individu na n iteraties voor deze implementatie. Voor deze meting werden de volgende parameters gebruikt:

- iterations=3000
- population_size=200
- max_mutation_step=0.9
- num_crossovers=100

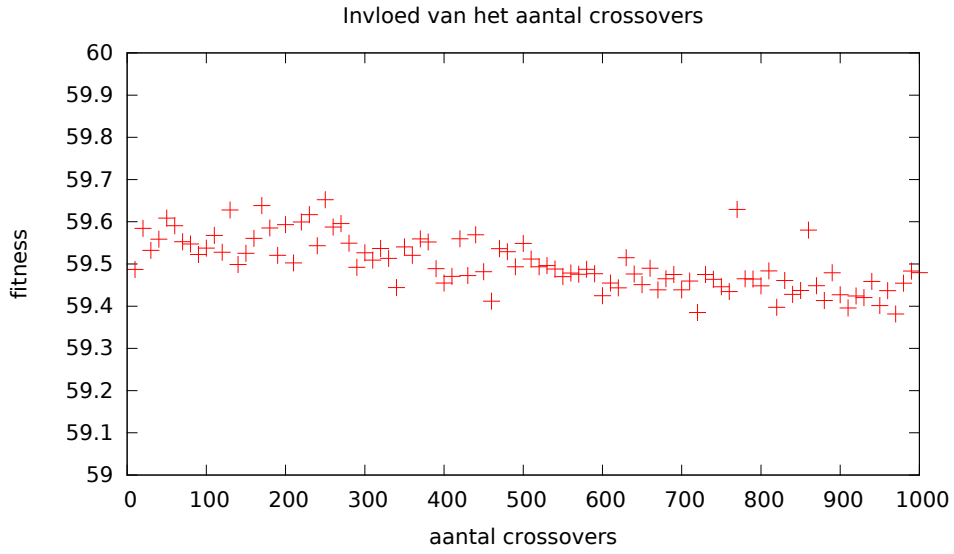
1.5 Invloed van de populatiegrootte



Het is duidelijk dat de beste fitness van de populatie zal stijgen naarmate de populatie groter wordt. De fitness stijgt initieel zeer snel, maar de stijging zwakt zeer snel af. De grafiek heeft een verloop gelijkaardig aan een logaritmische functie. Voor deze meting werden de volgende parameters gebruikt:

- iterations=3000
- max_mutation_step=0.9
- num_crossovers=100
- mutation_prob=0.3

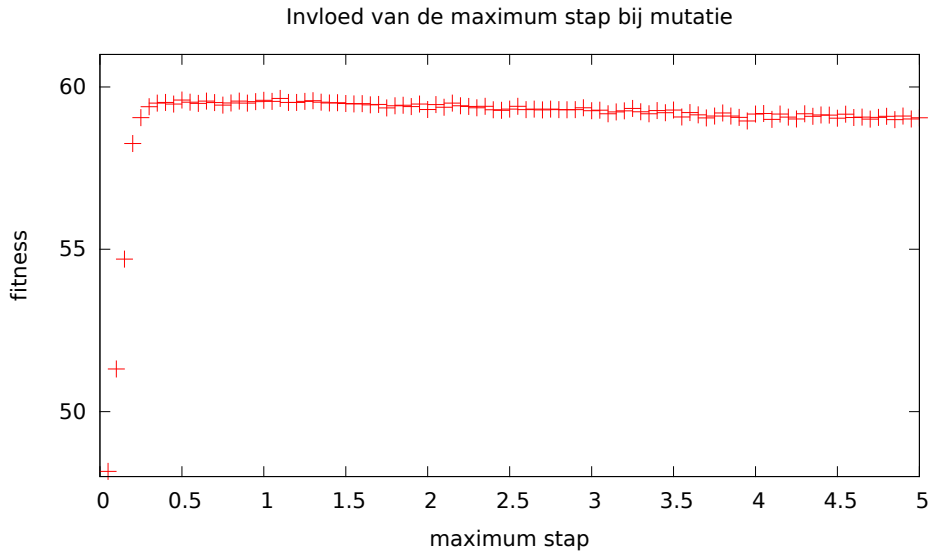
1.6 Invloed van het aantal crossovers



Tegen de intuïtieve verwachting in, zien ze een daling van de beste fitness naarmate het aantal crossovers (equivalent met het aantal kinderen maal 2 in elke iteratie) te groter wordt. Het aantal crossovers mag dus niet te groot gekozen worden. Deze meting werd uitgevoerd met volgende parameters:

- iterations=3000
- population_size=500
- max_mutation_step=0.9
- mutation_prob=0.3

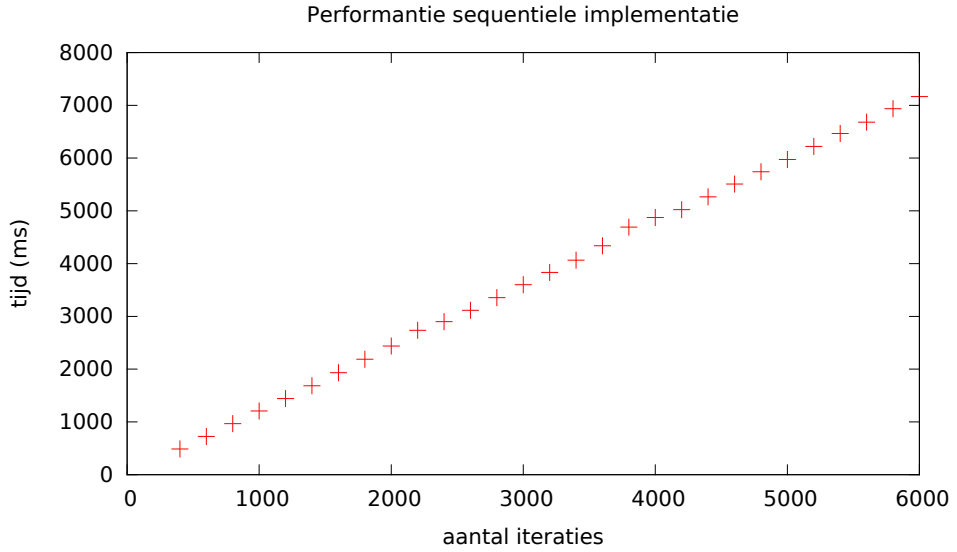
1.7 Invloed van de maximum stap bij mutatie



Bij een heel kleine maximum stap, zien we een zeer sterke stijging in de grafiek. Nadat deze curve een lokaal optimum bereikt, zien we een zeer kleine daling. De maximum stap kiezen we dus best niet te klein. Het optimum wordt hier ongeveer bereikt in 0,6. Deze meting werd uitgevoerd met volgende parameters:

- iterations=3000
- population_size=500
- num_crossovers=200
- mutation_prob=0.3

1.8 Performantie en geschiktheid

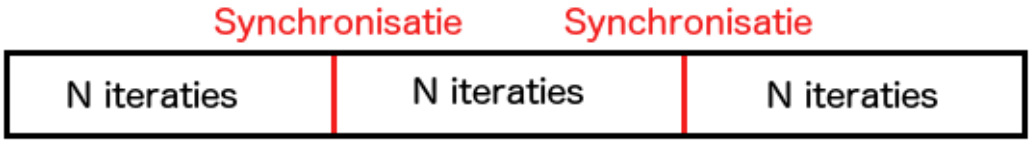


Aangezien dit algoritme iteratief is, heeft dit logischerwijs een lineaire complexiteit. Een genetisch algoritme lijkt zeer geschikt om dit probleem op te lossen, aangezien een optimale plaatsing slechts kan bekomen worden door trial and error. Een genetisch algoritme lijkt de enige mogelijkheid om dit probleem op te lossen in lineaire tijd. Het is mogelijk om de huidige parameter van het aantal iteraties te vervangen door een nauwkeurigheid en vervolgens het algoritme stop te zetten wanneer de verbetering van de fitness kleiner is dan de nauwkeurigheid.

2 Gedistribueerd algoritme

2.1 Beschrijving

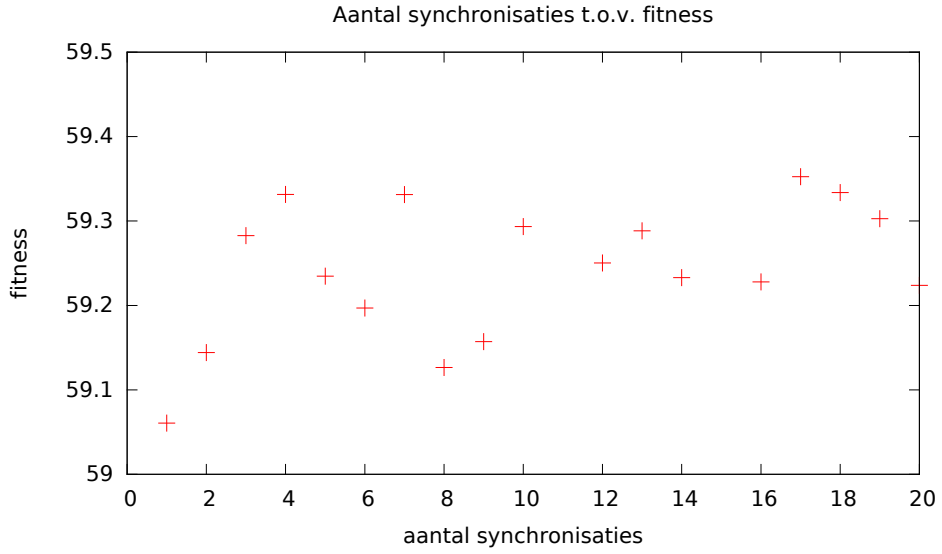
Het parallel algoritme heeft nog 1 extra parameter, namelijk het aantal synchronisaties. Bovenop een aantal iteraties zal dit algoritme om de n iteraties synchroniseren en zal elke thread zijn individuen uitwisselen met andere threads. Hieronder staat een voorbeeld met N iteraties en 2 synchronisaties.



Deze synchronisatie gebeurt met `MPI_Alltoall`. Indien er n processen zijn, worden de individuen van elke thread opgedeeld in groepen van `population_size/n`. De i-de groep individuen van het elk proces zal vervolgens terechtkomen bij het in i-de proces. Om de uitwisseling van individuen tussen threads mogelijk te maken, wordt de populatie individuen omgezet naar een array van `floats`.
Na afloop van alle iteraties en synchronisaties wordt het individu met beste fitness gekozen in elk proces. Dit wordt dan vergeleken met de beste individuen van alle andere processen met behulp van `MPI_Allreduce` met `MPI_MAXLOC`.

2.2 Verhouding aantal synchronisaties en aantal iteraties

Het totale aantal iteraties is equivalent met het product van de parameters `synchs` en `iterations`. Bij volgende testen werd dus steeds `iterations` verlaagd wanneer `synchs` verhoogd werd, zodat we hetzelfde aantal effectieve iteraties behouden, met meer synchronisatiestappen tussen de iteraties.

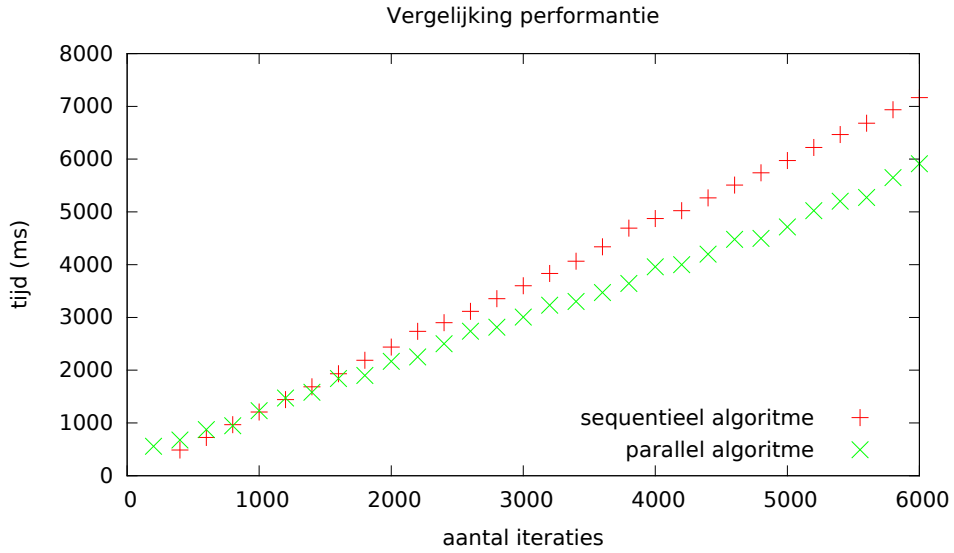


Ook hier kan geen sterke correlatie aangetoond worden, maar we zien wel een stijging naarmate het aantal synchronisaties verhoogd wordt. Deze meting werd uitgevoerd met volgende parameters:

- population_size=200
- max_mutation_step=0.5
- num_crossovers=100
- mutation_prob=0.75

3 Performantievergelijking

Om een vergelijking te maken tussen het sequentieel en het parallel algoritme werd de parameter `iterations` van de parallelle implemetatie constant gehouden. Na 50 iteraties werd dus steeds een synchronisatie uitgevoerd. Het aantal iteraties uit de sequentiële versie kwam dus steeds overeen met `synchs*iterations` van de parallelle versie.



Tijdens de eerste 1500 iteraties loopt het parallel algoritme ongeveer gelijk aan het sequentieel algoritme. Dit is te wijten aan de overhead van MPI. Vervolgens zien we dat het parallel algoritme steeds een kleinere uitvoeringstijd kent vergeleken met het sequentiële algoritme.
Deze meting werd uitgevoerd met volgende parameters:

- population_size=50
- max_mutation_step=0.5
- num_crossovers=200
- mutation_prob=0.3