Inleiding & programma overzicht

Ik heb gekozen om het programma te herschrijven in Rust, vertrekkend van de code van FGS 1.31. Mijn programma kan ook de metadata en de dna outputten. Het is ontwikkeld in Rust 1.43.1, en werkt niet op <=1.42.

De file *main.rs* leest de argumenten in en roep de *from_file(s)* functies op van de hmm en de training. Hierna wordt de *run* functie opgeroepen die zich in *run.rs* bevindt. Voor de multithreading maak ik gebruik van Rayon. Ik heb ook nog een writer thread die een mpsc channel heeft om instanties van *WriteTuple* te ontvangen.

Een *WriteTuple* bestaat uit een index en een *WriteEntry*. De index dient om de volgorde van de input te blijven bewaren. In de writer thread maak ik gebruik van een binaire hoop om de binnenkomende data gesorteerd te kunnen uitschrijven. Ik heb gekozen om geen collect te doen op de Rayon iterator, omdat ik vermoed dat het nuttiger is dat de output die je al hebt al uitgeschreven wordt en niet eerst verzameld wordt (ook qua geheugengebruik zal dat schelen).
De writer thread houdt een *next_wanted* teller bij. Als de binaire hoop dan de index *next_wanted* heeft, wordt deze uitgeschreven en uit de hoop gehaald. Op deze manier is de output in de volgorde zonder dat we eerst alle output moeten bijhouden.
Een *WriteEntry* bestaat uit een naam en nul of meer *OutputEntry*. De *OutputEntry* bevat de aminozuur, DNA en metadata output (indien gewenst).

In de threading code wordt eerst de CG% bepaald. In plaats van zoals in FGS(+(+)) de regio te kopiëren van de training naar de hmm, heb ik de data anders gestructureerd in het geheugen waardoor ik een reference kan nemen naar een *TrainSingle* (*train.rs*).

De matrices in de viterbi zijn ook anders qua geheugenlayout. Dit was voornamelijk om caching redenen en ook zodat ik makkelijk een referentie kan nemen naar alle states van een positie in de sequentie. De referenties worden genomen in een struct *MatrixRefs*, gedefinieerd in *matrix.rs*. Dat laatste was nodig om een hoop bound checks te elimineren, want dat gaf anders echt slechtere performance. Ik heb mijn best gedaan om wat gemeenschappelijke code af te zonderen, en om enums te gebruiken.

Problemen in FGS(+(+))

Out of bounds

Er gebeuren heel veel out of bound accesses in FGS. Om een voorbeeld te geven kunnen we kijken naar een fragment code uit I' state update:

```
if (path[S_STATE_1][t-3]!= R_STATE && path[S_STATE_1][t-4] !=R_STATE &&
path[S_STATE_1][t-5] !=R_STATE){...}
```

FGS+(+) past dit aan door een “t >= 5” conditie toe te voegen. Volgens mij is dit incorrect omdat: als t<5, dan zal t - 5 nooit een R state kunnen zijn, en moet dat deel dus wel evalueren naar true (analoog voor t-3, t-4). Dit is slechts één voorbeeld, maar ik heb per case bekeken wat de logische optie zou zijn in plaats van overal een escape test toe te voegen.

In de functie die de aminozuren bepaald van een DNA input, wordt er in FGS ook soms buiten de grenzen gegaan, waardoor er een X staat op het einde van de output. Incomplete codons schrijf ik in mijn programma niet uit.

Threading

FGS werkt met “groepjes”. Stel dat je *p* threads hebt. De inputfile wordt ingelezen tot een veelvoud van *p* of het einde van de file is bereikt. Dan worden er *p* threads gestart en er wordt pas verder gelezen in de file wanneer alle *p* threads gedaan hebben. Dit zorgt ervoor dat je langer moet wachten, omdat je afhangt van de traagste input sequentie in de groep en als er al een thread gedaan is, dat die geen werk krijgt.

Het avontuur van de dna buffer

Ik was verbaasd om deze code te zien in FGS1.31 (analoog voor het reverse complement geval):

```
//update dna before calling get_protein, YY July 2018
dna[0] = '\0';
strncpy(dna, 0 + dna_start_t - 1, dna_end_t - dna_start_t + 1);
dna[dna_end_t - dna_start_t + 1] = '\0';
```

Dit maakt de hele reconstructie in de dna buffer ongedaan. Dus alle deletions en insertions dat je uitvoerde worden ongedaan gemaakt hierdoor. FGS+(+) doen dit niet, dus haalde ik het weg. Tot mijn verbazing daalde de sensitiviteit en precision t.o.v. FGS1.31 tot gelijke niveaus als van FGS+(+). Om dit mysterie te ontrafelen heb ik een kijkje genomen naar FGS versie 1.30 en 1.20.

Deze code stond in FGS 1.30 origineel binnen een “if(refine) {...}” blok. In 1.30 staat er nog iets extra dat ontbreekt in 1.31: een check “add complete start codon to dna” (lijnen 869 en 943 in *hmm_lib.c*) met een body die een extra codon toe indien nodig. Als ik dit ook toevoeg in mijn eigen code en die strncpy weghaal, dan scoor ik wel weer de FGS1.31 niveaus van precision en sensitivity. Dit is veel logischer dan de code van FGS1.31: je overschrijft nu de buffers niet volledig, maar voegt enkel toe. Ik ben dus er redelijk van overtuigd dat FGS1.31 enkel de juiste output geeft als je geen insertions en deletions hebt. Waarschijnlijk is deze fout erin geslopen door de onbestaande code formatting in FGS.

Maar er is hier nog een tweede mysterie. Eerder schreef ik dat het codeblok in een “if(refine) {...}” blok stond in FGS1.30. Het is nog steeds raar dat je in die refine case de volledige buffer zou overschrijven. Ik heb dus gekeken naar FGS 1.20, waar die refinement code nog

in Perl was in plaats van in C in *post_process.pl*. De originele Perl code overschrijft niet de gehele buffer, deze voegt enkel vanvoor toe (bij +, lijn 256) of vanachter toe (bij -, lijn 328). Lijn 256 (328 is analoog):

```
   print $faa "$tmphead\n" . getaa(substr($genome_seq, $temp[0] - 1 + $i_save, $i_save * -1)) . "$tmpseq\n";
```

We zien dus dat dit een concatenatie is in plaats van een gehele buffer te overschrijven. In de C code wordt de refinement wel nog correct berekend (de dna end of dna start worden bijgewerkt a.d.h.v. *s_save*, wat in de Perl code *i_save* was). Ik heb me voor mijn refinement code dus ook gebaseerd op de Perl code in plaats van op de (wellicht foutieve) C code.

Aminozuren output

Na al deze aanpassingen heb ik nog gemerkt dat er sommige outputs van aminozuren zijn die beginnen met een *. Ik vond het vreemd dat er een stopcodon in het begin zat, maar ik merkte dat FGS 1.30 dit ook had. Dit wordt veroorzaakt door die extra code met die check die de extra codon toevoegt in de body.

Om dit verder te onderzoeken heb ik de inputs die resulteerden in deze situatie ook eens gegeven aan GeneMark. Er waren drie situaties qua GeneMark output:

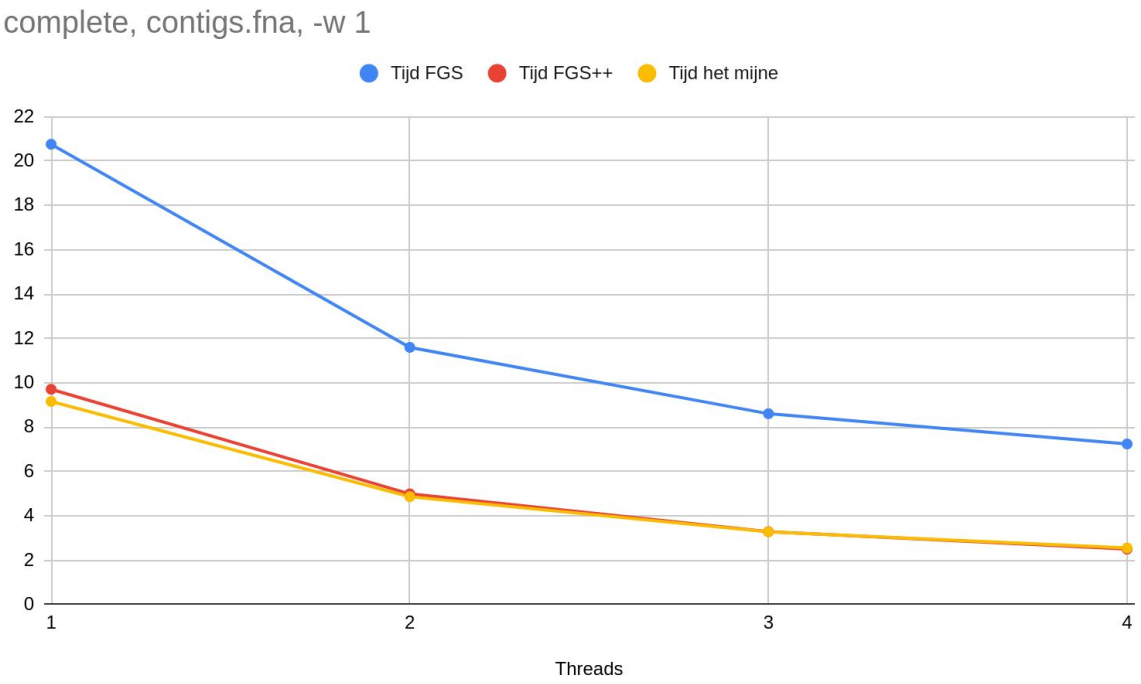
1) Geen output
2) Output, maar mijn programma en FGS 1.30 outputten meer aminozuren dan GeneMark (bv uit de 454: **>r45993.1**)
3) De * staat op het einde

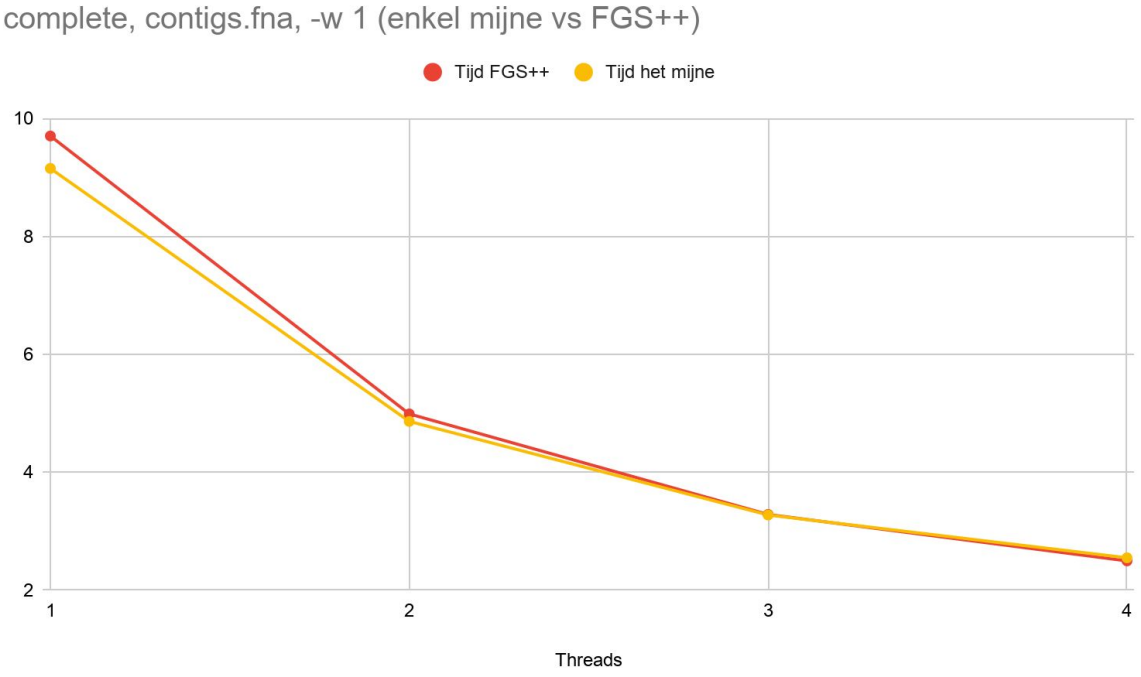
Als ik ervoor zorg dat de extra code output op het einde, dan daalt mijn sensitiviteit sterk, terwijl de precision ongeveer gelijk blijft. Dus ik leid hieruit af dat dit misschien toch niet fout is, maar ik weet hier te weinig over om dit zeker te zijn.

Benchmarks & ufora test

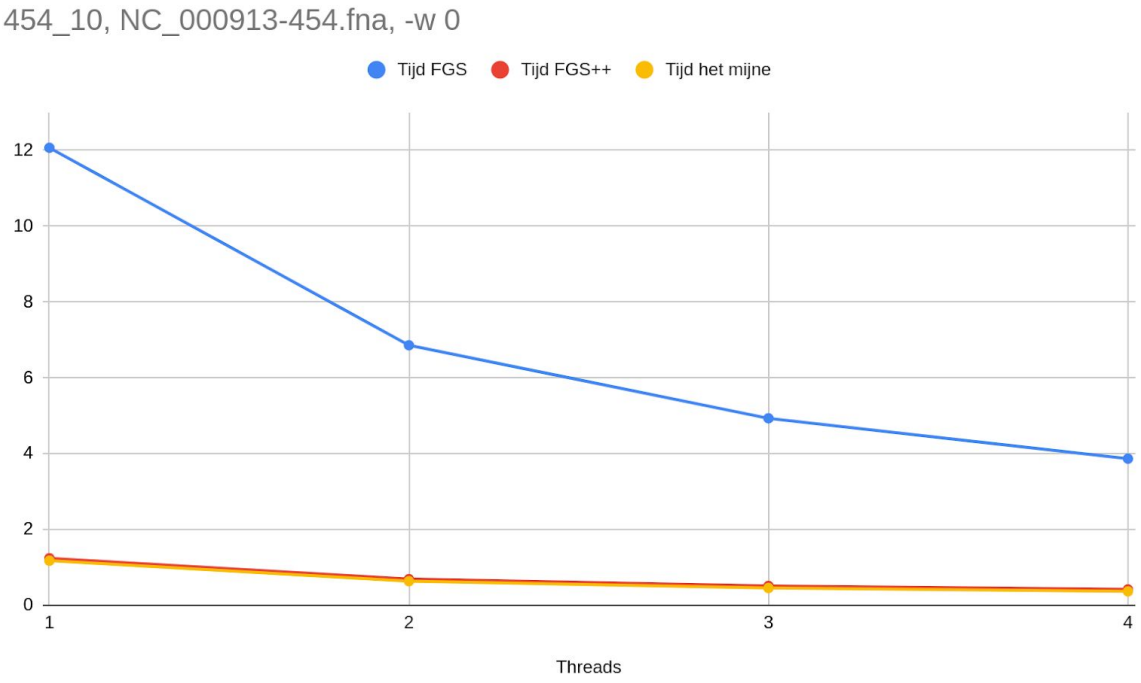
Gecompileerd met RUSTFLAGS='-C target-cpu=native', geeft dit een kleine winst aan performance. Ik heb dit niet gedaan bij mijn benchmarks en in mijn ingediende versie.

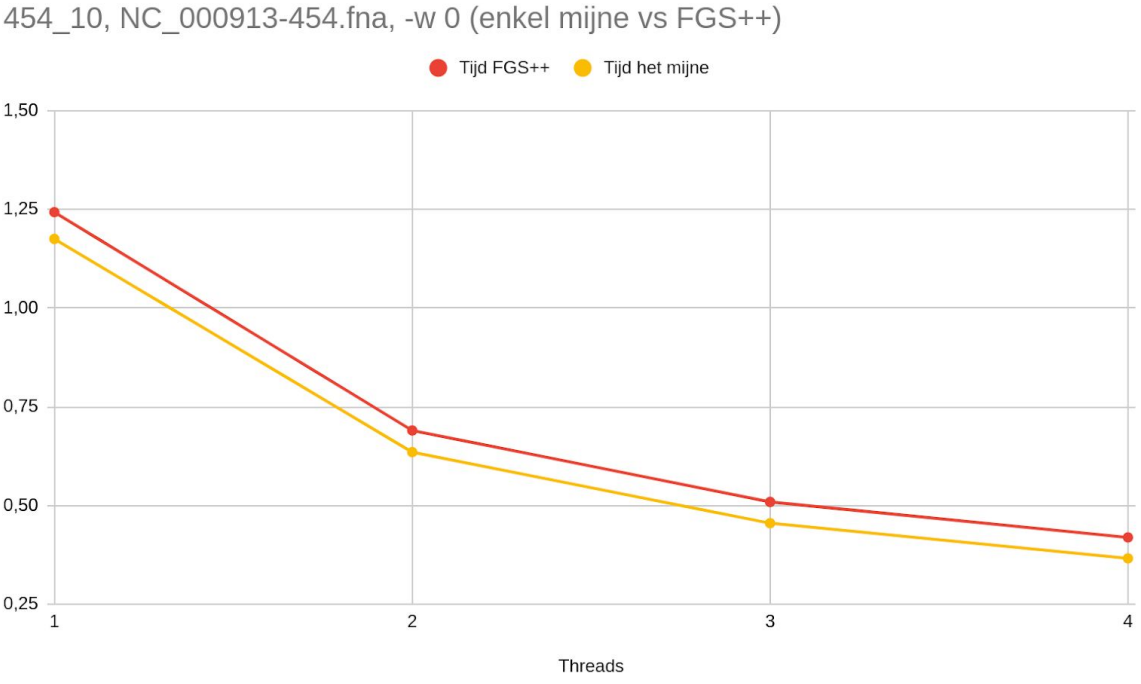
Tijd benchmarks



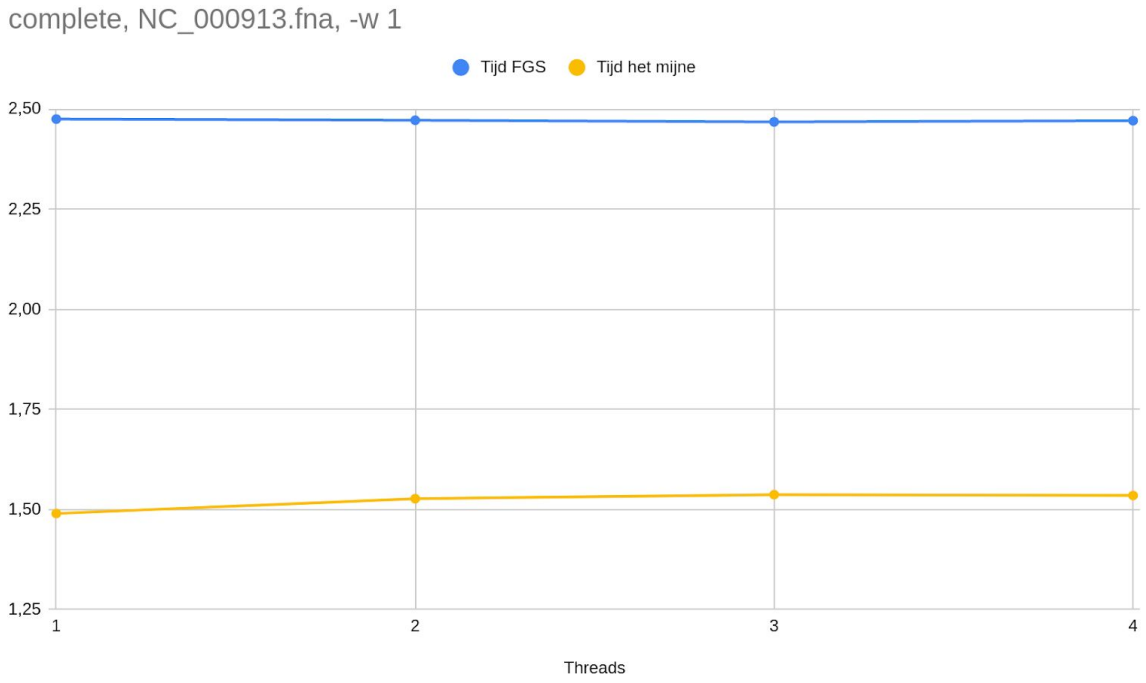


Dezelfde grafiek, ingezoomd, maar enkel het mijne vs FGS++

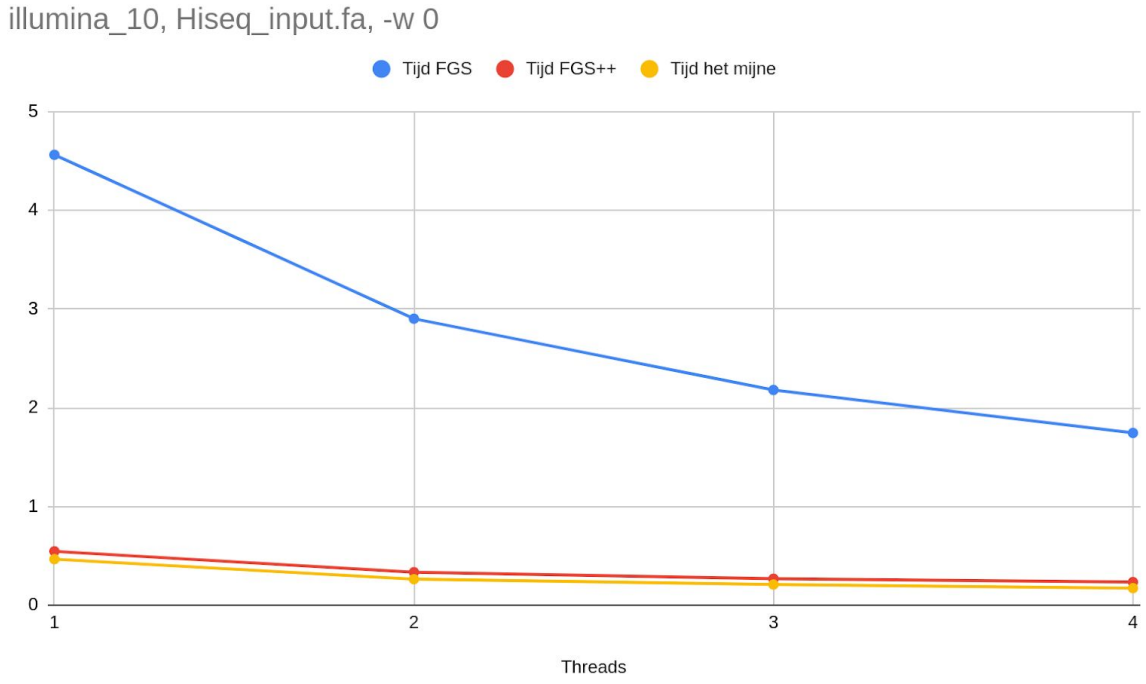


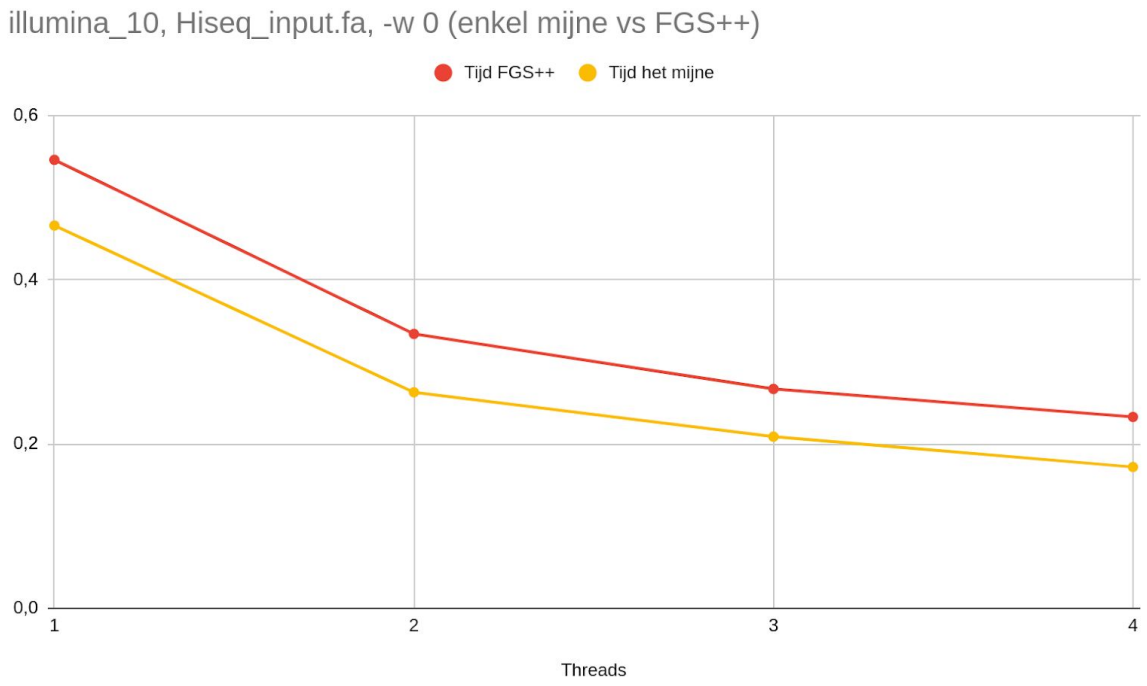


Dezelfde grafiek, ingezoomd, maar enkel het mijne vs FGS++



FGS++ staat hier niet op wegens segfault
Dit is een input die slechts 1 reeks bevat, dus dit test enkel de threading overhead



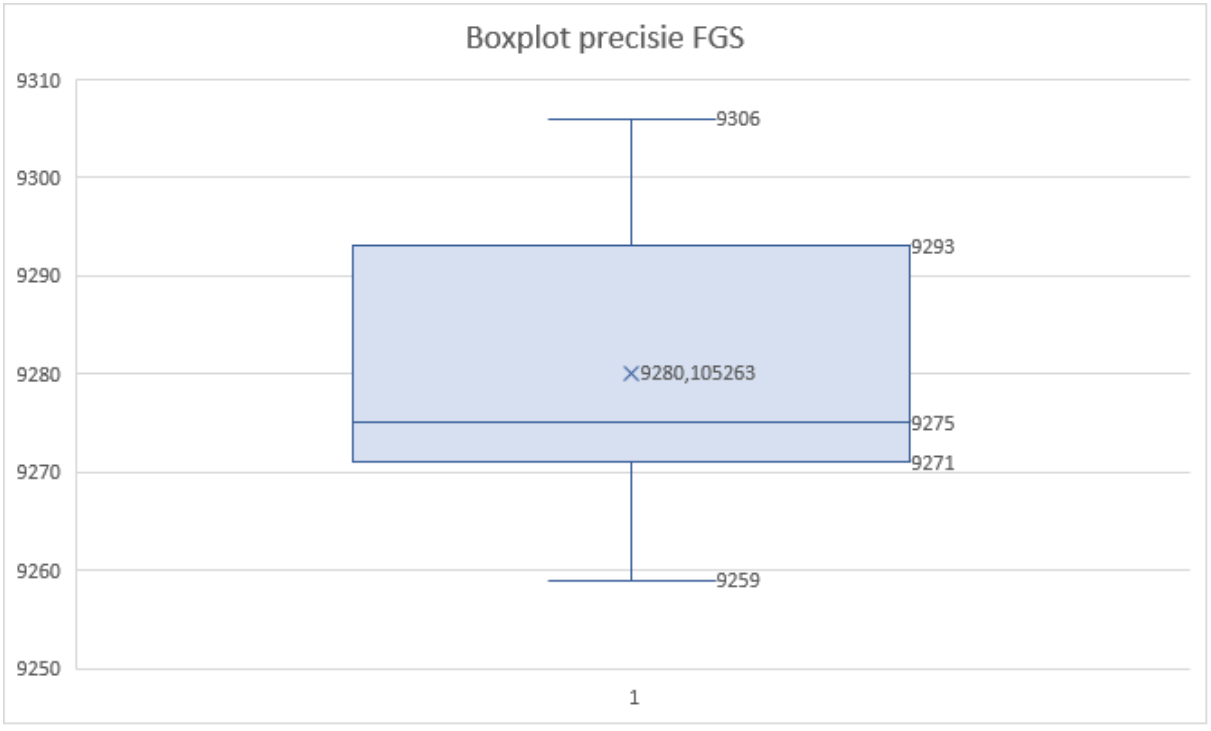


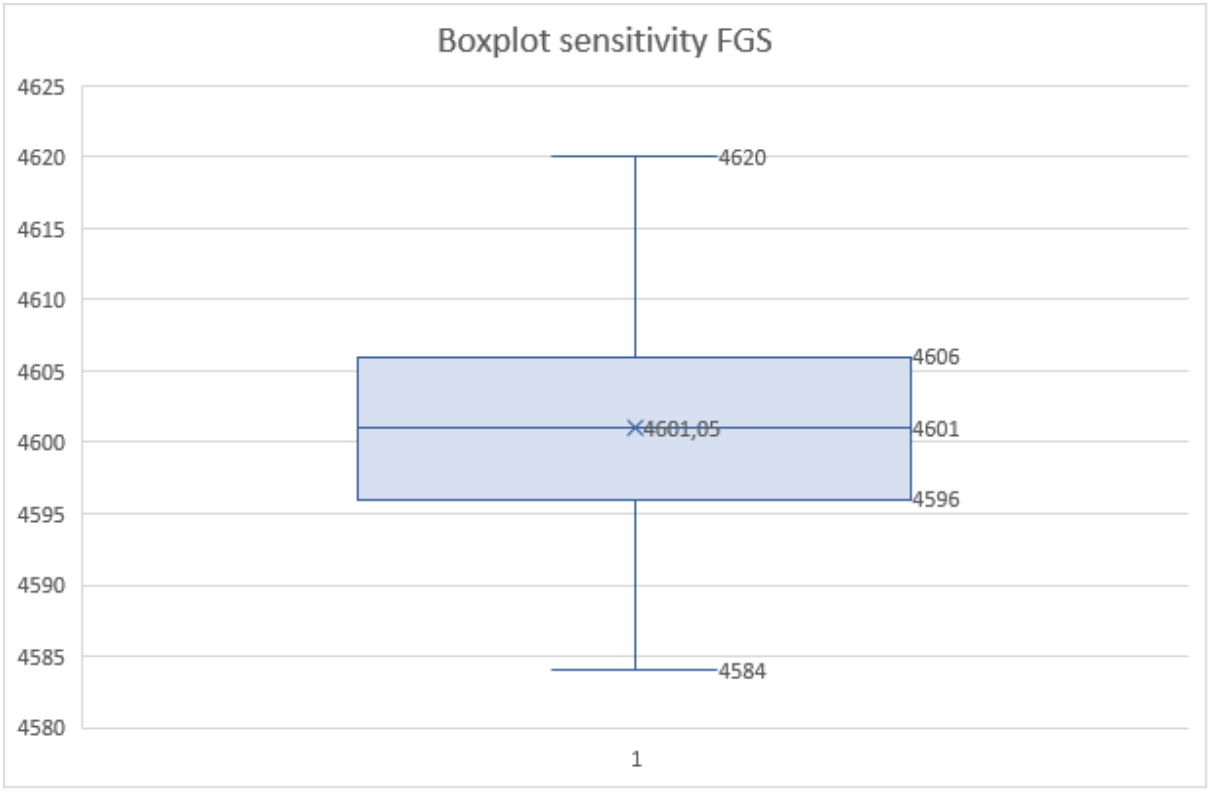
Dezelfde grafiek, maar enkel het mijne vs FGS++

Test op ufora met umgap

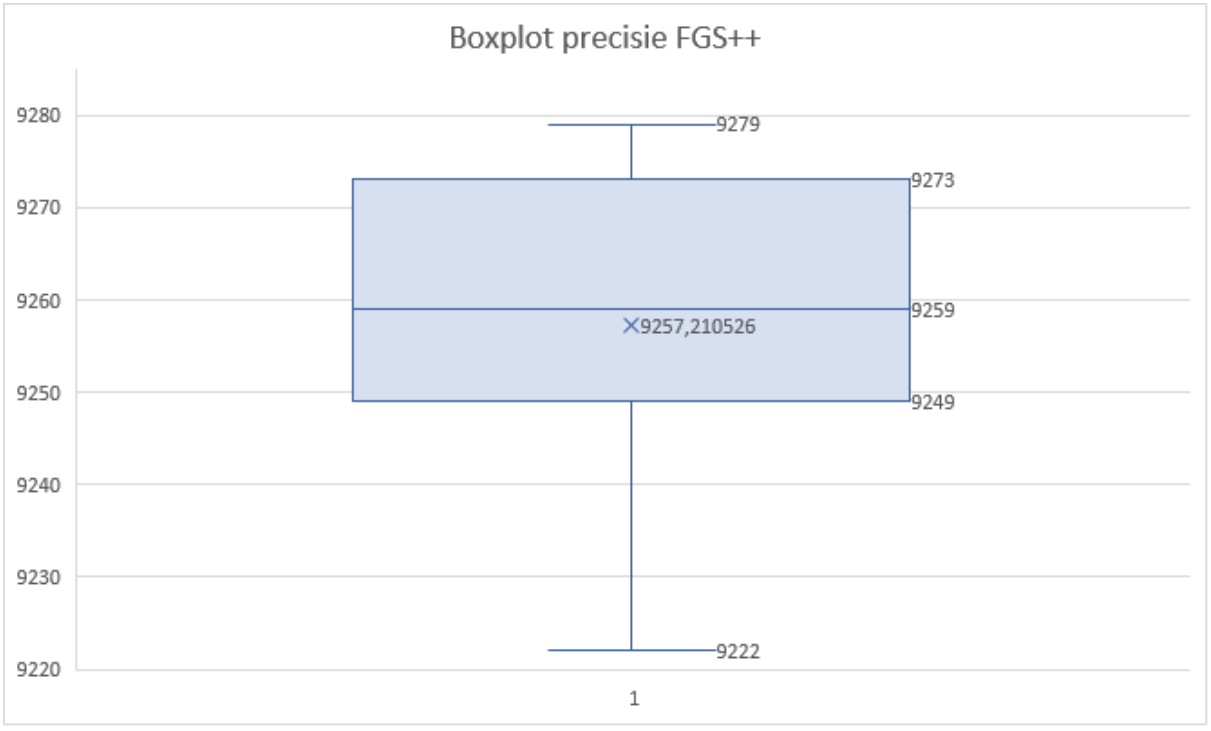
30 runs voor elk programma

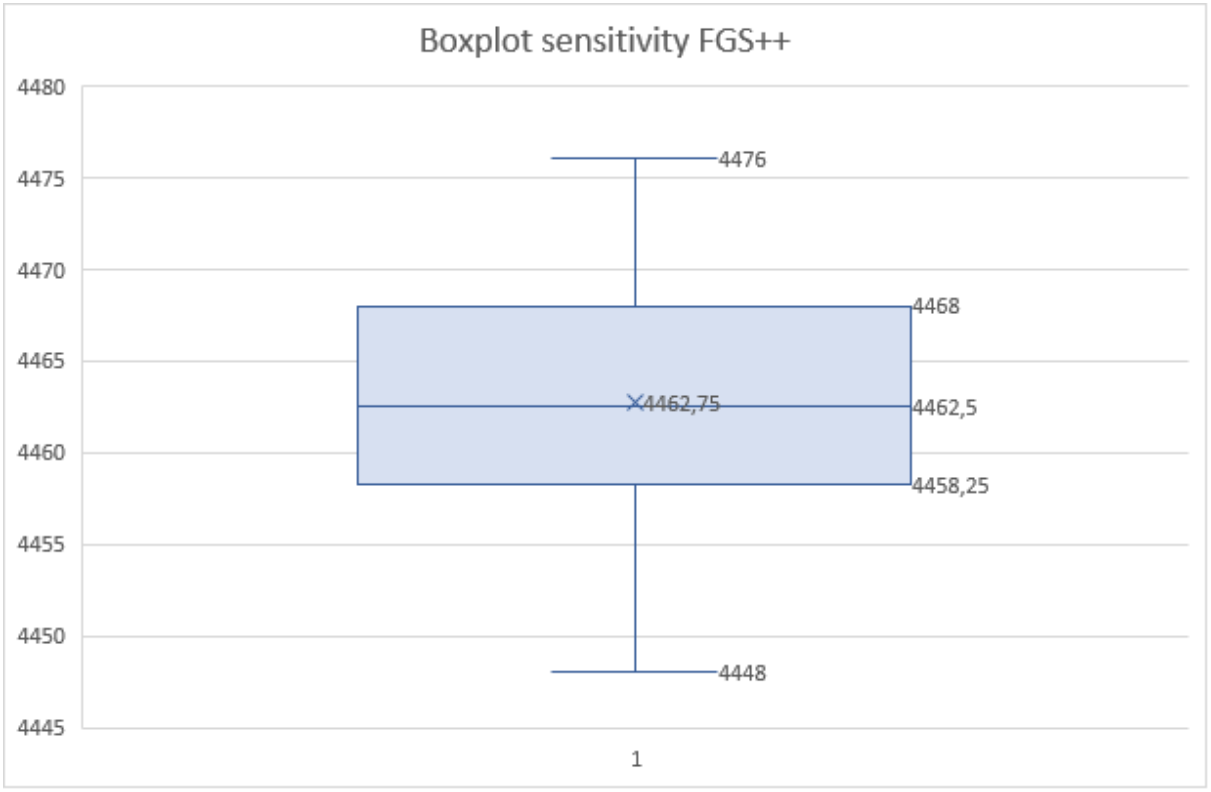
Runs op FGS



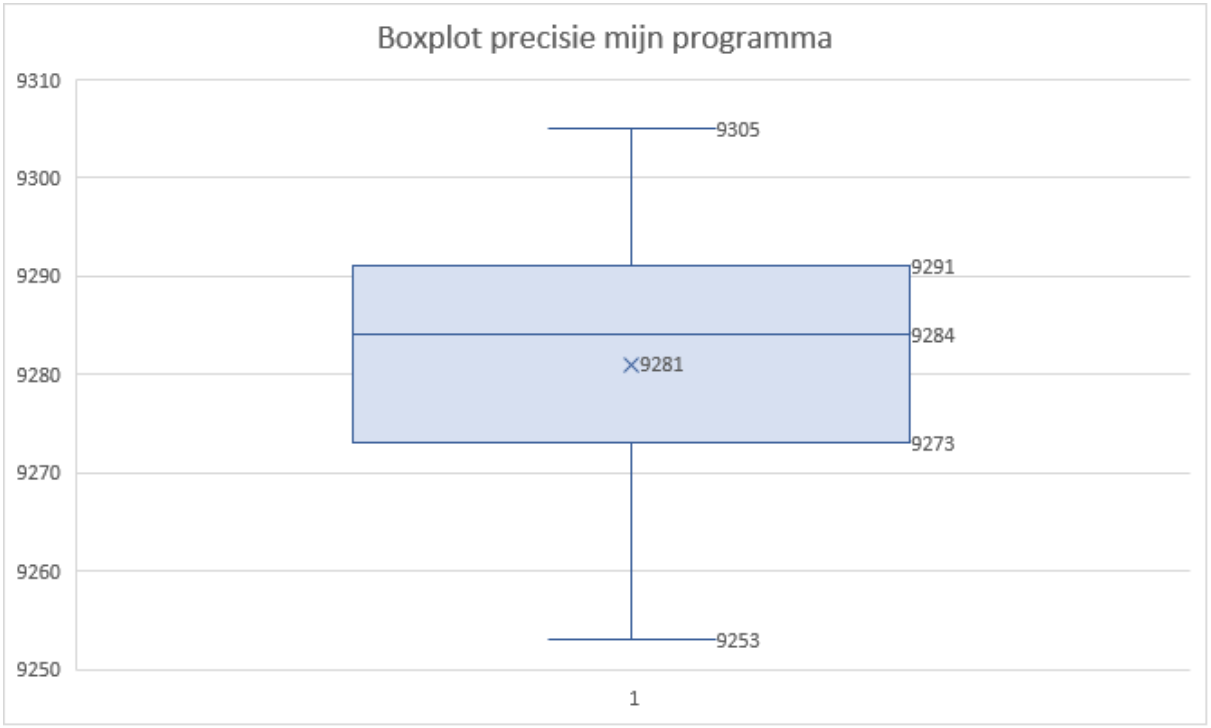

Boxplot sensitivity FGS
4625
4620
4615
4610
4605
4600
4595
4590
4585
4580
4620
4606
4601,05
4601
4596
4584
1

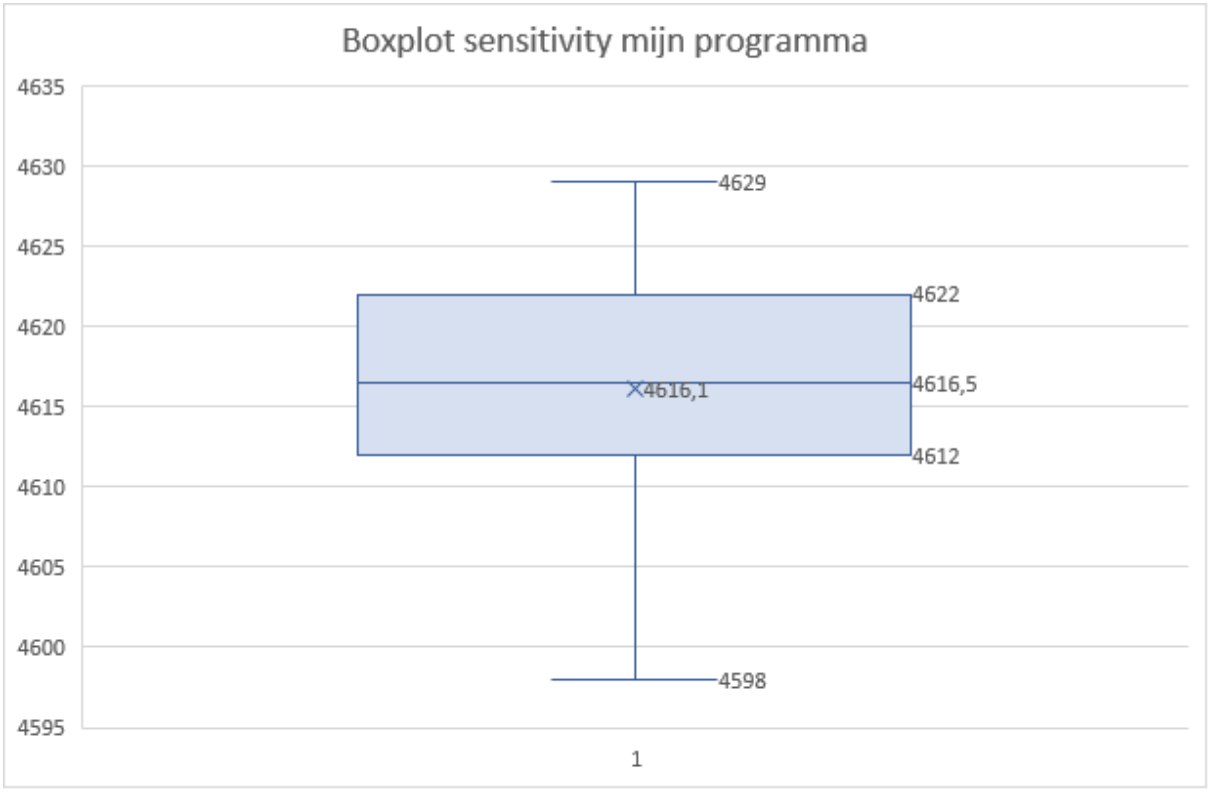
Runs op FGS++


Boxplot precisie FGS++
9280
9270
9260
9250
9240
9230
9220
9279
9273
9257,210526
9259
9249
9222
1



Runs op mijn programma





Cache performance

Getest met cachegrind voor de dataset van de ufora test, op mijn cpu: L1d van 128 KiB

	Data refs	**Data 1 misses**	**Instruction refs**	**Instruction 1 misses**
FGS	15,298,773,665	240,834,306	16,846,638,596	7,672,016
FGS++	4,068,074,609	62,358,983	5,233,331,321	3,330,983
Mijn programma	1,346,918,722	15,608,572	5,233,037,439	1,126,941