Ghent University

Computiationele biologie

Verslag project "FragGeneScan"

Tibo Vanheule

Academiejaar 2019-2020

Contents

1 Introductie

Er werd in de opgave drie keuzes voorgesteld:

- FragScanGene optimaliseren en interface aanpassen.
- Bestaande optimalisaties van fragScanGene verbeteren.
- FragSceneScan zelf schrijven.

Ik heb voor de eerste aanpak gekozen. Om deze code dus zo goed mogelijk te begrijpen, is Doxygen gebruikt [2]. Doxygen is vrij gemakkelijk om code te documenteren en die documentatie te genereren als pdf of html. Het allereerste pdf document van deze generator is te vinden in het project [1]. Via Javadoc kon ik makkelijk verder heel het project documenteren. De uiteindelijke documentatie is te vinden op https://tibovanheule.space/fraggenescan/ [4] en in het pdf document [3].

2 Main

Als eerste werd de interface herschreven. Er werd voor gezorgd dat er gelezen werd van Standard input, dit was relatief makkelijk na het samenvoegen van de loops (Zie sectie "Optimalisaties"). Als volgt werd alles naar Standard Output geschreven, dit was ook relatief makkelijk, omdat fprintf ook naar stdout kan schrijven. Later werden de twee opties "-d" en "-e" toegvoegd. Om dit nog werkende te houden op windows heb ik gekozen voor met iffen te werken. Op linux kan men alles wegschrijven naar /dev/null.

3 Optimalisaties

In de code waren een heleboel optimalisatie mogelijkheden. In deze sectie worden de belangrijkste optimalisaties besproken.

3.1 Samevoegen van loops

In de main leest men het sequentie bestand 3 keer uit. Dit is niet zo efficiënt en zorgt voor een vertraging. Zeker met systemen waarop de toegang tot de harde schijf traag is. Met enige creativiteit lukt het om deze 3 loops samen te voegen.

Het algorithme werkt als volgt:

1. alloceer geheugen met variable size voor de sequentie.
2. lees de lijn met max grootte STRINGLEN
3. controleer als de grootte van lijn + lengte sequentie niet groter is dan size.
4. indien ja, realloc om het gealloceerd geheugen te vergroten.
5. append de lijn aan de sequentie
6. ga terug naar stap 2

3.2 Calloc

Op verschillende plaatsen in de code, maar vooral in util_lib. Wordt malloc gebruikt om geheugen te alloceren en wordt het gealloceerd geheugen manueel op nul gezet. Calloc heeft dezelfde werking en uitkomst en vervangt deze 2 stappen door één instructie. Het is dus ook een mooiere stijl van programmeren.

Na een test [7, 8] ,bestaande uit 20000 iteraties, krijgen we volgende resultaten in figuur 1 en 2 (grafieken gemaakt in Power Bi [5]).

In het aller slechtste geval is de calloc-instructie veel sneller is dan de malloc-instructie. Er kan gesteld worden dat de calloc-instructie ofwel sneller is of even efficiënt is dan malloc. Sinds calloc de code ook korter maakt en leesbaarder maakt, is op de mogelijke plaatsen malloc vervangen door calloc.

3.3 Vermijden van geheugen operaties

Er werden soms onnodige geheugen operaties uitgevoerd. In viterbi waren er grote memsets, die het gealloceerd geheugen terug op 0 zetten. In get_prob_from_cg() kopieerde men onnodig geheugen. Het vermijden van deze operaties zorgde voor de grootste toename in performance.

3.4 Threadpool

Aan de hand van een threadpool kan het programma ook nog versnelt worden. Een thread aanmaken is duur, dus hergebruiken van een thread kan het programma versnellen.

Sinds hiervoor een aparte niet-eigen bibliotheek gebruikt en het niet cross-platform is, is dit een optioneel optie gemaakt. Gebruik make pool om een threadpool te gebruiken.

3.5 Guards

Bij het openen van de header files bleek dat de conventies niet gevolgd waren. Een header file kon dus meerdere keren geincludeerd worden. Om problemen te vermijden werden guards toegevoegd [6].

3.6 Stack vs Heap

De grootste fout in de inititiele code was plaatsen van te grote variablen in het stack geheugen. Windows en Mac os hebben een veel kleinere stack dan linux. Hierdoor crashte het programma op beide systemen als gevolg van een Stack overflow. De oplossing was om deze variabelen over te brengen naar het dynamisch geheugen (heap). Echter gaf linux dan fouten bij het freeen van dit geheugen in hmm_lib. Dit is opgelost door het gebruik van define's.

4 Conclusie

	windows (gnu make)	debian VPS	mac os
orginele code	stack overflow, seg fault	0.40 mins	stack overflow, seg fault
na error-fix	0.37	n.v.t.	0.18
mijn code	0.14	0.15 mins	0.11

Table 1: timings van een uitvoering met 1 thread

De initiële code is duidelijk niet geschreven door een informaticus. Optimaliseren was hierdoor moeilijker dan verwacht. Dit komt omdat er veel conventies al dan niet bewust genegeerd waren. Als gevolg vertraagde de code bij het invoegen van deze conventies. Er waren veel optimalisatie mogelijkheden bij memory operaties. Daarom is er een flinke daling bij de VPS te zien in tabel 1. Die draait op een gevirtualiseerd systeem en heeft dus een grotere latency naar het geheugen.

Windows in tegenstelling profiteerde vooral van threading (tabel 2).

	windows (gnu make)	debian VPS	mac os
orginele code	stack overflow, seg fault	0.43 mins	stack overflow, seg fault
mijn code	0.06	0.14 mins	0.17

Table 2: timings van een uitvoering met 4 threads

References

[1] *Documentatie voor enige aanpasingen.pdf*. URL: `Project%20root`.

[2] *Doxygen: Main Page*. URL: `http://www.doxygen.nl/`.

[3] *fraggenescan.pdf*. URL: `Project%20root`.

[4] *FragScanTibo: Hoofd Pagina*. URL: `https://tibovanheule.space/fraggenescan/`.

[5] *Gegevensvisualisatie? | Microsoft Power BI*. URL: `https://powerbi.microsoft.com/nl-nl/`.

[6] *Include guard*. en. Page Version ID: 956136327. May 2020. URL: `https://en.wikipedia.org/w/index.php?title=Include_guard&oldid=956136327`.

[7] *Test: Malloc + setter vs Calloc Matrix*. URL: `project%20root/test/malloc%20vs%20calloc%20for%20matrixs`.

[8] *Test: Malloc + setter vs Calloc Vector*. URL: `project%20root/test/malloc%20vs%20calloc%20for%20vector`.

5 Figuren

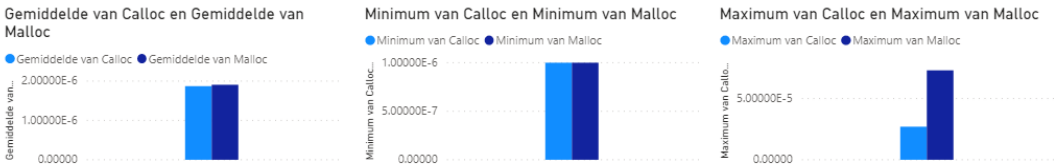

Figure 1: malloc vs calloc bij matrixen

Figure 2: malloc vs calloc bij vectoren