Eindopdracht: Julia Set met OpenMP en MPI

Leerling: Sietse Neve

Leerlingnummer: 1810364

Github link: https://github.com/DrZisnotavailable/HPP.git

Inleiding

In deze opdracht is een programma ontworpen om een animatie van de Julia Set te genereren. Elke frame is een Julia-fractal met veranderende parameters per frame. De rendering van de pixels is rekenintensief, dus is gekozen voor een hybride parallelisatie-aanpak:

- **MPI** verdeelt de frames over meerdere processen/nodes
- **OpenMP** parallelliseert de rendering binnen elk frame

Deze aanpak benut effectief de kracht van moderne multicore- en multiprocessor-systemen.

Waarom OpenMP en MPI?

Het voorgestelde ontwerp is geschikt voor deze opdracht omdat:

- Elke frame onafhankelijk is van andere frames → ideaal voor **MPI**, die deze frames over processen kan verdelen zonder onderlinge afhankelijkheden.
- Binnen een frame is er veel herhalend rekenwerk per pixel → ideaal voor **OpenMP**, die dit via multithreading versnelt.

Andere ontwerpen, zoals dynamische taakverdeling tussen nodes of master-worker patronen, kunnen theoretisch beter schalen, maar zijn voor deze toepassing onnodig complex gezien de gelijke werklast per frame.

OpenMP-parallelisatie binnen een frame

Geoptimaliseerde loops

De dubbele for-loop over alle pixels is geparallelliseerd:

```
#pragma omp parallel for collapse(2)
for (int y = 0; y < HEIGHT; ++y) {
    for (int x = 0; x < WIDTH; ++x) {
        image[y * WIDTH + x] = computeJuliaPixel(x, y, frameIndex);
    }
}
```

Niet-geoptimaliseerde delen

- Initialisatie van de pixel-array is licht en hoeft niet geparallelliseerd.
- Het schrijven van PNG-bestanden moet enkel door één thread gebeuren om bestandsconflicten te voorkomen.

MPI voor verdeling van frames

Elke process krijgt een aaneengesloten blok van frames (chunking). Dit is eenvoudig te implementeren en goed schaalbaar.

- Geen synchronisatie nodig
- Elke process schrijft eigen resultaten weg
- Gathering is niet nodig

Striping vs. Chunking

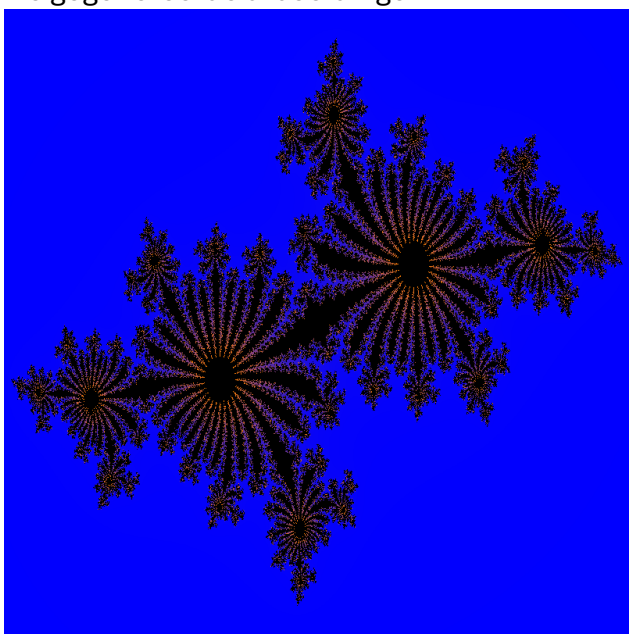
- **Striping** (om en om) is niet nodig omdat alle frames ongeveer evenveel rekenwerk vereisen.
- **Chunking** geeft minder communicatie- en I/O-overhead.

Geheugengebruik

Elke afbeelding is ~1.8MB (800x800x3). Een process rendert bv. 30 frames → ~54MB. Geheugengebruik blijft beperkt omdat frames niet gedeeld of opgeslagen hoeven te worden buiten het eigen proces. Hieronder een gegenereerde foto:

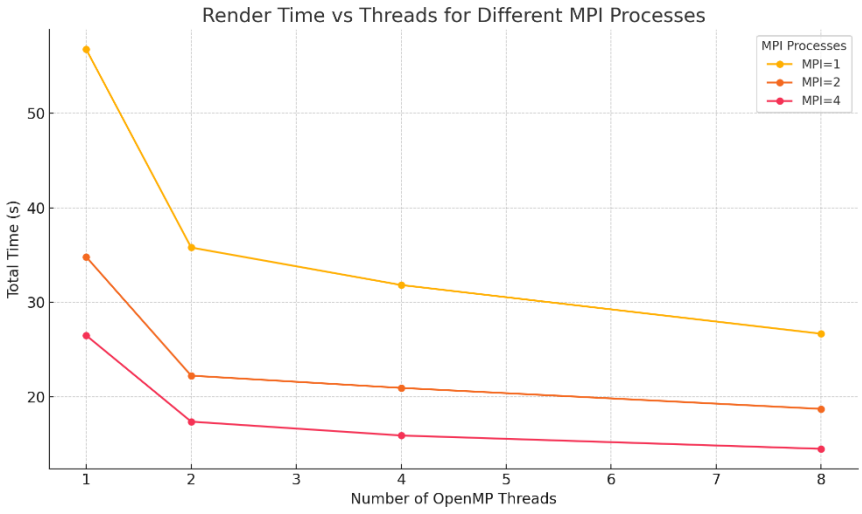
Resultaten

De gegenereerde afbeeldingen:



worden opgeslagen in mappen als `frames_<thread_count>`, waardoor prestaties eenvoudig te vergelijken zijn.

Onderstaande grafiek toont de totale uitvoertijd in seconden bij verschillende aantallen MPI-processen en OpenMP-threads. De afbeelding visualiseert duidelijk het effect van toenemende parallellisatie:



Zoals te zien is, daalt de totale tijd significant wanneer het aantal threads en processen toeneemt. Wel is er sprake van afnemende meeropbrengst bij hogere waarden, wat overeenkomt met de verwachtingen volgens Amdahl's Law.

Conclusie

De combinatie van OpenMP voor binnen-frame parallelisatie en MPI voor frameverdeling is:

- Efficiënt
- Simpel te implementeren
- Goed schaalbaar

Er is geen nood aan complexere dynamische verdelingen door de gelijkmatige belasting per frame.