5. JULIA SET

Joey Welvaadt

HOGESCHOOL UTRECHT 1734098

https://github.com/AI-S4-2023/v2hpp-herkansingsopdrachten-JoeyWelvaadt1999

1. Code

NOTITIE: DIT BETREFT VRAGEN 1, 2 EN 4

Om te beginnen, wil ik mijn uiteindelijke code weergeven. De geïmplementeerde structuur maakt gebruikt van OpenMP om het proces dat 1 enkele frame berekent te parallelliseren en maakt gebruik van MPI om alle frames te maken en verzamelen (figuur 1a en 1b). Maar om te verifieren of dit nou echt de beste manier is heb ik ook een opstelling gemaakt met enkel MPI (figuur 1a en 2a) en een met enkel OpenMP (figuur 1b en 2b). De foto's lijken misschien door elkaar te staan, maar voor enkel OpenMP moest ik enkel de main functie aanpassen en voor enkel MPI moest ik alleen de renderFrame functie aan te passen.

NOTITIE: NIET ALLE CODE PAST OP DE FOTO'S

```
double start_time = MPI_Wtime();

// TODO - parallellisation
for (combination = start; combination < stop; combination++) {
    // cout << endl << "Rendering frame " << combination << " on process [" << id << "]" << endl;
    renderFrame(local_frames, combination, start);
    count++;
}

std::vector<int> counts(nprocs);
std::vector<int> recv_counts(nprocs);
std::vector<int> displs(nprocs);
MPI_Gather(&count, 1, MPI_INT, counts.data(), 1, MPI_INT, 0, MPI_COMM_WORLD);

if (id == 0) {
    for (int i = 0; i < nprocs; i++) {
        recv_counts[i] = counts[i] * FRAME_SIZE;  // per proces, bytes
    }

    displs[0] = 0;
    for (int i = 1; i < nprocs; i++) {
        displs[i] = displs[i - 1] + recv_counts[i - 1];
    }

}

MPI_Gatherv(
    local_frames.data(),
    (stop - start) * FRAME_SIZE,
    MPI_BYTE,
    frames.data(),
    recv_counts.data(),
    displs.data(),
    MPI_BYTE,
    0,
    MPI_COMM_WORLD
);

double end_time = MPI_Wtime() - start_time;
double max_time = 0.0;

MPI_Reduce(&count, &max_count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce(&end_time, &max_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

if (id == 0) {
    cimg_library::CImg<byte> img(WIDTH,HEIGHT,FRAMES,3);
    cimg_forXYZ(img, x, y, z) {
        img(x,y,z,RED) = (frames)[z].get_channel(x,y,RED);
        img(x,y,z,GREEN) = (frames)[z].get_channel(x,y,GREEN);
        img(x,y,z,BLUE) = (frames)[z].get_channel(x,y,BLUE);
    }

    std::string filename = std::string("animation.avi");
    img.save_video(filename.c_str());

    cout << "\nTotal frames rendered: " << max_count << endl;
    cout << "Max execution time (across all processes): " << max_time << " seconds\n" << endl;
    cout << "Number of processes" << ";" << "Threads" << ";" << "Frames" << ";" << "Time" << ";" << endl;
    cout << nprocs << ";" << n_threads << ";" << max_count << ";" << max_time << ";" << endl;
}

// Also needed to send frames over MPI
MPI_Type_free(&mpi_img);
MPI_Finalize();
return 0;
```

Figuur 1a Gathering frames: MPI

```
void renderFrame(animation &frames, unsigned int t, unsigned int offset) {
    // TODO - render frame t and store in frames[t-offset]
    int max_iter = MAX_ITER;
    int escape_radius = 2;

    double a = 2 * std::numbers::pi * t / CYCLE_FRAMES;
    double r = 0.7885;
    std::complex<double> c = r * cos(a) + static_cast<std::complex<double>>(1i) * r * sin(a);

    // Loop through all pixels in the frame
    #pragma omp parallel for collapse(2)
    for (unsigned int x = 0; x < WIDTH; x++) {
        for (unsigned int y = 0; y < HEIGHT; y++) {
            // Calculate the number of iterations for the pixel at (x,y)
            int iter = 0;
            double x_y_range = 2;

            //double scale = 1.5 - 1.45 * t / FRAMES;                          // iets simpeler
            double scale = 1.5 - 1.45 * log(1 + 9.0 * t / FRAMES) / log(10);    // iets interessanter om naar te kijken

            std::complex<double> z = 2 * x_y_range * std::complex(static_cast<double>(x)/WIDTH, static_cast<double>(y)/HEIGHT)
                - std::complex(x_y_range*3/4, x_y_range);

            z *= scale;

            while (std::abs(z) < escape_radius && iter < max_iter) {
                z = z*z + c;
                iter++;
            }

            if (iter == max_iter) {
                frames[t - offset].set_colour(x, y, {0, 0, 0});
            } else {
                pixel colour_hue = COLOURISE(static_cast<double>(iter));
                frames[t - offset].set_colour(x, y, colour_hue);
            }
        }
    }
}
```

Figuur 1b Rendering enkele frame: OpenMP

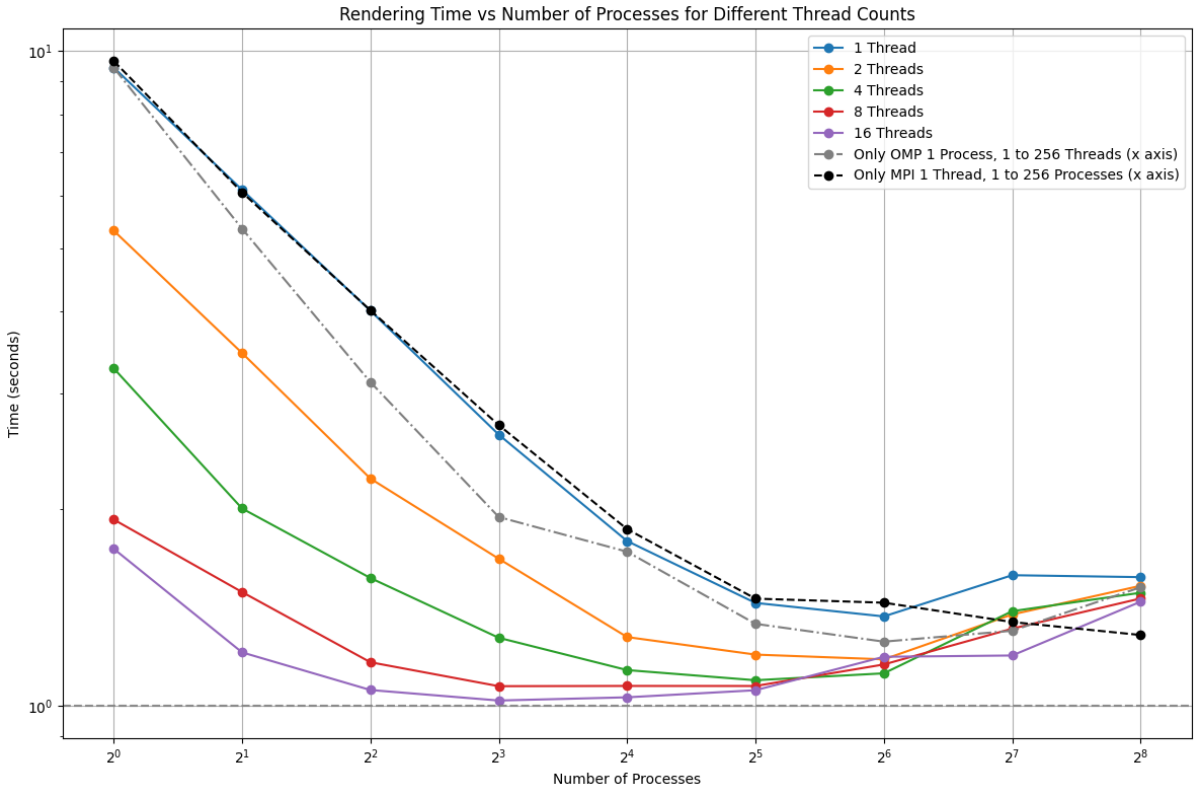
```cpp
void renderFrame(animation &frames, unsigned int t, unsigned int offset) {
    // TODO - render frame t and store in frames[t-offset]
    int max_iter = MAX_ITER;
    int escape_radius = 2;

    double a = 2 * std::numbers::pi * t / CYCLE_FRAMES;
    double r = 0.7885;
    std::complex<double> c = r * cos(a) + static_cast<std::complex<double>>(1i) * r * sin(a);

    // Loop through all pixels in the frame
    for (unsigned int x = 0; x < WIDTH; x++) {
        for (unsigned int y = 0; y < HEIGHT; y++) {
            // Calculate the number of iterations for the pixel at (x,y)
            int iter = 0;
            double x_y_range = 2;

            //double scale = 1.5 - 1.45 * t / FRAMES;                       // iets simpeler
            double scale = 1.5 - 1.45 * log(1 + 9.0 * t / FRAMES) / log(10);    // iets interessanter om naar te kijken

            std::complex<double> z = 2 * x_y_range * std::complex(static_cast<double>(x)/WIDTH, static_cast<double>(y)/HEIGHT)
                - std::complex(x_y_range*3/4, x_y_range);

            z *= scale;

            while (std::abs(z) < escape_radius && iter < max_iter) {
                z = z*z + c;
                iter++;
            }

            if (iter == max_iter) {
                frames[t - offset].set_colour(x, y, {0, 0, 0});
            } else {
                pixel colour_hue = COLOURISE(static_cast<double>(iter));
                frames[t - offset].set_colour(x, y, colour_hue);
            }
        }
    }
}
```

Figuur 2 RenderFrame veranderd naar enkel MPI

```cpp
int main (int argc, char *argv[]) {
    int n_threads = 1;
    if (argc >= 2) {
        n_threads = std::stoi(argv[1]);
    }
    omp_set_num_threads(n_threads);

    int id = -1, nprocs = 1;

    cout << "Number of threads: " << n_threads << endl;

    animation frames;

    frames.initialise(FRAMES);

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

    double start_time = omp_get_wtime();
    int max_count = 0;


    // TODO - parallellisation
    #pragma omp parallel for
    for (int combination = 0; combination < FRAMES; combination++) {
        // cout << endl << "Rendering frame " << combination << " on process [" << id << "]" << endl;
        renderFrame(frames, combination, 0);
        max_count++;
    }

    double end_time = omp_get_wtime() - start_time;

    cimg_library::CImg<byte> img(WIDTH,HEIGHT,FRAMES,3);
    cimg_forXYZ(img, x, y, z) {
        img(x,y,z,RED) = (frames)[z].get_channel(x,y,RED);
        img(x,y,z,GREEN) = (frames)[z].get_channel(x,y,GREEN);
        img(x,y,z,BLUE) = (frames)[z].get_channel(x,y,BLUE);
    }

    std::string filename = std::string("animation.avi");
    img.save_video(filename.c_str());

    cout << "\nTotal frames rendered: " << max_count << endl;
    cout << "Max execution time (across all processes): " << end_time << " seconds\n" << endl;
    cout << "Number of processes" << ";" << "Threads" << ";" << "Frames" << ";" << "Time" << ";" << endl;
    cout << 1 << ";" << n_threads << ";" << max_count << ";" << end_time << ";" << endl;

    MPI_Finalize();
    return 0;
}
```

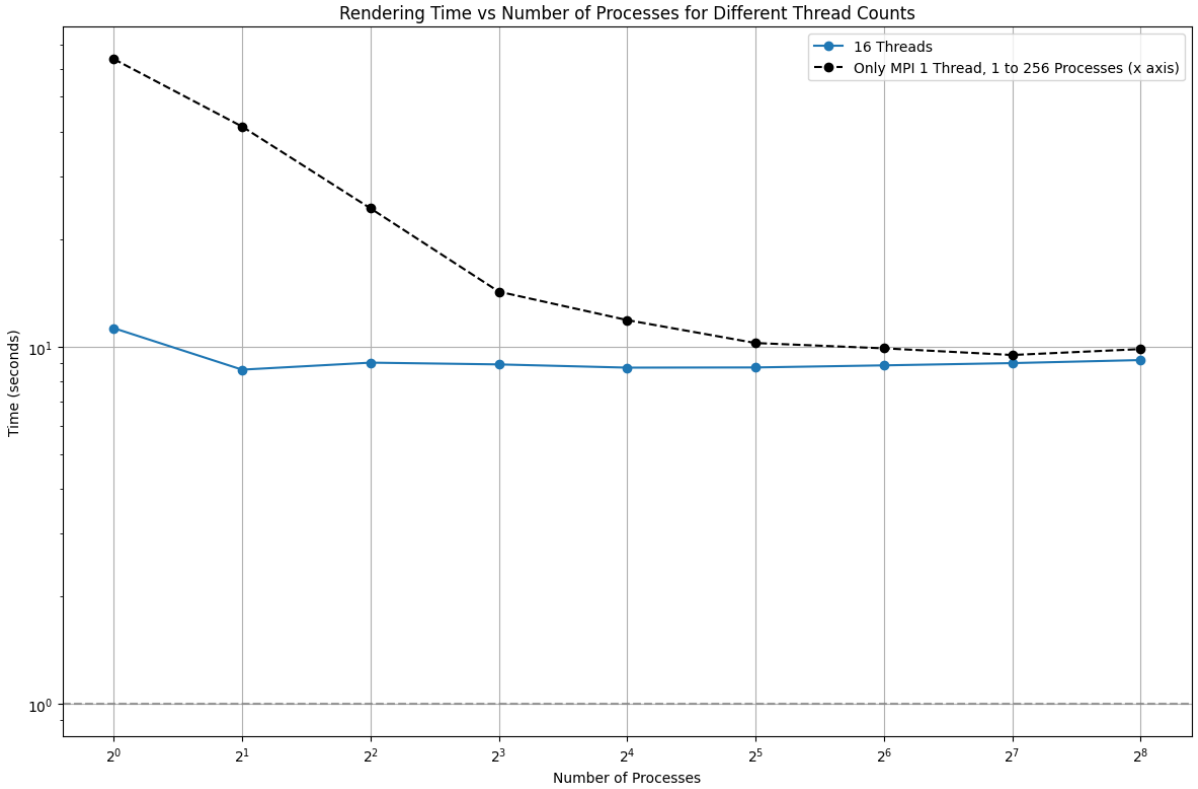
Figuur 2b Main veranderd naar enkel OpenMP

Nu de code duidelijk gemaakt is kunnen we kijken naar de resultaten van elk. Deze zijn samen in een grafiek weergegeven in figuur 3. Wat blijkt is dat het gebruik maken van de combinatie van OpenMP en MPI voor een groot deel zorgt voor de minste tijd, overigens gaat een implementatie van enkel MPI het beter sneller doen vanaf 256 processen komen. Dit kan komen omdat er op een gegeven moment 256 processen en 16 threads worden aangemaakt, wat zorgt voor veel communicatieoverhead terwijl er weinig werk is voor alle processen en threads.



Figuur 3 Benchmark verschillende aantallen processen en threads, ook OpenMP en MPI apart

Om daadwerkelijk aan te tonen dat bij meer werk de gecombineerde versie van beide MPI en OpenMP beter presteert is in figuur 4 de benchmark te zien van enkel MPI en de combinatie met OpenMP waar ze 5000 frames aanmaken in plaats van 750.



Figuur 4 Enkel MPI vs combinatie met OpenMP 5000 frames

NOTITIE: HET VOLGENDE BETREFT VRAAG 3

In de implementatie van beide OpenMP en MPI worden de frames in de root node gegathered. Ik heb voor de makkelijke optie gekozen. Overigens heb ik wel nagedacht over het verschil in chunking tegenover striping. Ik maak gebruik van chunking, ik deel eerst alle frame indexes op en ieder proces gaat dan met zijn stuk aan de slag. Het gebruik maken van striping, zodat het om en om wordt verdeeld over de processen, leidt tot de volgende dingen. Bij gahering zorgt het er voor dat de frames worden gegathered op willekeurige volgorde.

Dit komt doordat niet elke frame even lang duurt. In het ergste geval doet een frame er 256 * 256 * 81 stappen over tot deze klaar is. Als er 1 proces is die veel van dit soort frames moet doen, kan er voor zorgen dat 1 proces veel later klaar is dan andere processen. Hier kan striping een handje bij helpen door de frames te geven aan processen die daar ruimte voor hebben.