CP431 - Parallel Programming
Term Project

Topic: MPI Programming for Julia Sets

Aidan Traboulay, Memet Rusidovski,
Mobina Tooranisama, Nausher Rao, Zamil Bahri

Group 5

Friday 7th April, 2023

1 Introduction

The Julia Set is a fascinating mathematical object that has captured the imagination of mathematicians and computer scientists alike. It is a fractal set that arises from the iteration of a simple function in the complex plane. Generating the Julia Set is a computationally intensive task that has been the subject of much research in high-performance computing. In this report, we present two approaches to generating the Julia Set in the C programming language: using OpenCL and MPI.

OpenCL is a parallel computing framework that allows developers to write code that can execute across different devices, including GPUs, CPUs, and other accelerators. OpenCL provides a platform-independent programming model and allows for heterogeneous computing, making it an attractive option for many high-performance computing applications.

In this report, we compare the performance of these two approaches to generating the Julia Set. We implemented both approaches and ran experiments on different hardware configurations to measure their performance. We also discuss the advantages and disadvantages of each approach and provide recommendations for choosing the appropriate approach based on the hardware and problem size.

Overall, this report provides a comprehensive analysis of the generation of Julia Sets using two popular parallel computing techniques. Our findings will be of interest to researchers and practitioners in the field of high-performance computing.

2 Implementation

The project is split up into three (3) main sections: LargePNG - which handles the setup for high resolution PNG formatted files and does some calculation of the Julia sets, OpenCL which handles the rendering and output of the program, and VideoCPU which handles the creation of high resolution videos for Julia set. In an effort to modularize the project for ease of refactoring, these are each set up in their own folders, containing a main file and other relevant information, such as header files.

2.1 LargePNG Implementation (*LargePNG*)

- (*block.h*)

```c
#include <png.h>

/**
 *
 * @file block.h
 * @brief Represents a block of data in the final video output.
 *
 */
typedef struct _Block {
    int size;       // The size of the block in bytes
    int is_done;    // Whether the block has been filled with data
    int place;      // The place in the final video output
    int width;      // The width of the block in pixels
    int height;     // The height of the block in pixels
    png_byte **data; // The data of the block
} Block;
```

- (*main.c*)

```c
#include <complex.h>
#include <math.h>
#include <mpi.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include "block.h"
```

```
#define FRAME_COUNT 100
#define MAX_ITER 50
#define PNG_DEBUG 3
#define SIZE 1000
#define OUTPUT_FILE_NAME "a.png"

/**
 *
 * @brief Global variables.
 *
 */
int x;
int y;

int WIDTH;
int HEIGHT;
int NUM_PASSES;

char *FILENAME = "a.png";

png_byte COLOR_TYPE;
png_byte BIT_DEPTH;
png_byte **IMAGE_DATA;

png_structp PNG_PTR;
png_inis thfop INFO_PTR;
png_bytep *ROW_PTR;

Block KNAPSACK;

/**
 *
 * @brief Function prototypes.
 *
 */
void write_img();
void calculate(int index);
void allocate(int size);

/**
 *
 * @brief Write the image to a file.
 * @param void
 * @return void
 *
```

```
 */
void write_img() {
    COLOR_TYPE = PNG_COLOR_TYPE_RGBA;
    BIT_DEPTH = 8;

    NUM_PASSES = 1;

    FILE *fp = fopen(FILENAME, "wb");

    if (!fp) {
        fprintf(stderr, "[write_png_file] File %s could not be opened for
            writing", FILENAME);
    }

    PNG_PTR = png_create_write_struct(PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);

    if (!PNG_PTR) {
        fprintf(stderr, "[write_png_file] png_create_write_struct failed");
    }

    INFO_PTR = png_create_info_struct(PNG_PTR);

    if (!INFO_PTR) {
        fprintf(stderr, "[write_png_file] png_create_info_struct failed");
    }

    if (setjmp(png_jmpbuf(PNG_PTR))) {
        fprintf(stderr, "[write_png_file] Error during init_io");
    }

    png_init_io(PNG_PTR, fp);

    if (setjmp(png_jmpbuf(PNG_PTR))) {
        fprintf(stderr, "[write_png_file] Error during writing header");
    }

    png_set_IHDR(PNG_PTR, INFO_PTR, WIDTH, HEIGHT,
                 BIT_DEPTH, COLOR_TYPE, PNG_INTERLACE_NONE,
                 PNG_COMPRESSION_TYPE_BASE, PNG_FILTER_TYPE_BASE);

    png_write_info(PNG_PTR, INFO_PTR);

    if (setjmp(png_jmpbuf(PNG_PTR))) {
        fprintf(stderr, "[write_png_file] Error during writing bytes");
    }
```

```
    png_set_compression_level(PNG_PTR, 6);
    png_set_filter(PNG_PTR, 0, 0);

    fprintf(stdout, "HERE<<<<\n");
    png_write_image(PNG_PTR, KNAPSACK.data);
    fprintf(stdout, "THERE>>>>>\n");

    if (setjmp(png_jmpbuf(PNG_PTR))) {
        fprintf(stderr, "[write_png_file] Error during end of write");
    }

    png_write_end(PNG_PTR, NULL);

    for (y = 0; y < HEIGHT; y++) {
        free(KNAPSACK.data[y]);
    }

    free(KNAPSACK.data);

    fclose(fp);

    return;
}

/**
 *
 * @brief Calculate the mandelbrot set.
 * @param index The index of the block.
 * @return void
 *
 */
void calculate(int index) {
    int bit = 0;
    float scale = 1.5;

    float cx = -0.8;
    float cy = 0.156;

    float zx = (scale * (float)(WIDTH / 2 - x) / (WIDTH / 2));
    float zy = (scale * (float)(HEIGHT / 2 - y) / (HEIGHT / 2));

    int i = 0;
    int c = 0;

    float temp = (zx * zx - zy * zy + cx);
```

```
for (y = index; y < SIZE + index; y++, c++) {
    png_byte *row = KNAPSACK.data[c];

    for (x = 0; x <= WIDTH; x++) {
        png_byte *ptr = &(row[x * 4]);

        bit = 0;
        zx = (scale * (float)(WIDTH / 2 - x) / (WIDTH / 2));
        zy = (scale * (float)(HEIGHT / 2 - y) / (HEIGHT / 2));

        for (i = 0; i < MAX_ITER; i++) {
            temp = (zx * zx - zy * zy + cx);
            zy = (2.0 * zx * zy + cy);
            zx = temp;
            if (zx * zx + zy * zy > 4.0) {
                bit = i % 5;
                if (bit == 0) {
                    bit = 5;
                }
                break;
            }
        }

        if (bit == 0) {
            ptr[0] = 0;
            ptr[1] = 0;
            ptr[2] = 0;
        } else if (bit == 1) {
            ptr[0] = 204;
            ptr[1] = 107;
            ptr[2] = 73;
        } else if (bit == 2) {
            ptr[0] = 210;
            ptr[1] = 162;
            ptr[2] = 76;
        } else if (bit == 3) {
            ptr[0] = 236;
            ptr[1] = 230;
            ptr[2] = 194;
        } else if (bit == 4) {
            ptr[0] = 115;
            ptr[1] = 189;
            ptr[2] = 168;
        } else {
            ptr[0] = 153;
            ptr[1] = 190;
```

```
                ptr[2] = 183;
            }
            ptr[3] = 255;
        }
    }

    return;
}

/**
 *
 * @brief Allocate the image data.
 * @param size The size of the image.
 * @return void
 *
 */
void allocate(int size) {
    IMAGE_DATA = (png_byte **) malloc(sizeof(png_byte *) * size);

    for (y = 0; y < size; y++) {
        IMAGE_DATA[y] = (png_byte *) malloc(sizeof(png_bytep) * WIDTH);
    }

    KNAPSACK.height = HEIGHT;
    KNAPSACK.width = WIDTH;
    KNAPSACK.size = HEIGHT * WIDTH;
    KNAPSACK.place = 0;
    KNAPSACK.is_done = 0;

    KNAPSACK.data = IMAGE_DATA;

    return;
}

/**
 *
 * @brief The main function.
 * @param argc The number of arguments.
 * @param argv The arguments.
 * @return 0
 *
 */
int main(int argc, char **argv) {
    int process_rank, cluster_size;

    MPI_Init(&argc, &argv);
```

```
MPI_Comm_size(MPI_COMM_WORLD, &cluster_size);
MPI_Comm_rank(MPI_COMM_WORLD, &process_rank);
MPI_Status status;

WIDTH = 10840;
HEIGHT = 10160;

int cluster = cluster_size - 1;

int num_blocks = (HEIGHT / SIZE);

if (process_rank == 0) {
    allocate(HEIGHT);
} else {
    allocate(SIZE);
}

if (process_rank == 0) {
    int next = 1;

    for (int i = 0; i < cluster; i++) {
        KNAPSACK.place = (i * SIZE);

        MPI_Recv(&next, 1, MPI_INT, MPI_ANY_SOURCE, 3, MPI_COMM_WORLD,
            &status);
        MPI_Send(&KNAPSACK, 4, MPI_INT, next, 1, MPI_COMM_WORLD);
    }

    for (int i = 0; i < num_blocks - cluster; i++) {
        MPI_Recv(&next, 1, MPI_INT, MPI_ANY_SOURCE, 5, MPI_COMM_WORLD,
            &status);
        MPI_Recv(&KNAPSACK, 4, MPI_INT, next, 8, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);

        for (int k = 0; k < SIZE; k++) {
            MPI_Recv(KNAPSACK.data[k + KNAPSACK.place], KNAPSACK.width * 4,
                MPI_BYTE, next, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }

        KNAPSACK.place = i * SIZE + (cluster * SIZE);
        MPI_Send(&KNAPSACK, 4, MPI_INT, next, 7, MPI_COMM_WORLD);
    }

    for (int i = 0; i < cluster; i++) {
        MPI_Recv(&next, 1, MPI_INT, MPI_ANY_SOURCE, 5, MPI_COMM_WORLD,
            &status);
```

```
            MPI_Recv(&KNAPSACK, 4, MPI_INT, next, 8, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);

            for (int k = 0; k < SIZE; k++) {
                MPI_Recv(KNAPSACK.data[k + KNAPSACK.place], KNAPSACK.width * 4,
                    MPI_BYTE, next, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            }

            KNAPSACK.is_done = 1;
            MPI_Send(&KNAPSACK, 4, MPI_INT, next, 7, MPI_COMM_WORLD);
        }

        clock_t begin = clock();
        write_img();
        clock_t end = clock();

        double time_spent = ((double)(end - begin) / CLOCKS_PER_SEC);

        fprintf(stdout, "Write time: %f %d\n", time_spent, KNAPSACK.width);

    } else {
        MPI_Send(&process_rank, 1, MPI_INT, 0, 3, MPI_COMM_WORLD);
        MPI_Recv(&KNAPSACK, 4, MPI_INT, 0, 1, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);

        calculate(KNAPSACK.place);

        fprintf(stdout, "Recieving %d\n", KNAPSACK.place);

        while (KNAPSACK.is_done == 0) {
            MPI_Send(&process_rank, 1, MPI_INT, 0, 5, MPI_COMM_WORLD);
            MPI_Send(&KNAPSACK, 4, MPI_INT, 0, 8, MPI_COMM_WORLD);

            for (int k = 0; k < SIZE; k++) {
                MPI_Send(KNAPSACK.data[k], KNAPSACK.WIDTH * 4, MPI_BYTE, 0, 2,
                    MPI_COMM_WORLD);
            }

            fprintf(stdout, "Sending %d\n", KNAPSACK.place);
            MPI_Recv(&KNAPSACK, 4, MPI_INT, 0, 7, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);

            if (KNAPSACK.is_done != -1) {
                calculate(KNAPSACK.place);
            }
```

```
            fprintf(stdout, "Recieving %d\n", KNAPSACK.place);
        }
    }

    MPI_Finalize();

    return 0;
}
```

2.2 OpenCL Implementation (*LargePNG*)

- (*julia.cl*)

```
// Useful: https://www.informit.com/articles/article.aspx?p=1732873&seqNum=3
__kernel void julia(__global uchar3 *A, __global float *B){ //, const unsigned
    int count){ //(__global int *A, __global int *B, __global int *C) {
    // Get the index of the current element
    int i = get_global_id(0);

    int SIZE = 128 * 16;

    int width = SIZE;
    int height = SIZE;
    int MAX_ITER = 256;

    int x = (i % SIZE);
    int y = (int) i / SIZE;


    // Do the operation
    int bit = 0;
    float scale = 1.5;
    float cx = B[0];// -0.8;
    float cy = B[1];// 0.153;
    float zx = scale * (float)(width / 2 - x) / (width / 2);
    float zy = scale * (float)(height / 2 - y) / (height / 2);
    int j = 0;
    int c = 0;
    float tmp = zx * zx - zy * zy + cx;


        for (j = 0; j < MAX_ITER; j++)
            {
                tmp = zx * zx - zy * zy + cx;
                zy = 2.0 * zx * zy + cy;
```

```
            zx = tmp;
            if (zx * zx + zy * zy > 4.0)
            {
                //bit = 1;
                bit = j % 5;
                if(bit == 0) bit = 5;
                break;
            }
        }

// if(i % 1024 == 0)
// A[i] = '\n';
// else{
    // A[i].x = x;
    // A[i].y = y;
    // A[i].z = 255;
// }

if (bit == 0){
    A[i].x = 0;
    A[i].y = 0;
    A[i].z = 0;
}
else if (bit == 1){
    A[i].x = 20;
    A[i].y = 42;
    A[i].z = 231;
}
else if (bit == 2){
    A[i].x = 132;
    A[i].y = 196;
    A[i].z = 32;
}
else if (bit == 3){
    A[i].x = 93;
    A[i].y = 30;
    A[i].z = 105;
}
else if (bit == 4){
    A[i].x = 43;
    A[i].y = 211;
    A[i].z = 50;
}
else{
    A[i].x = 53;
    A[i].y = 181;
```

```
        A[i].z = 204;
    }
}
```

- (*main.c*)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <string.h>
#include <stdarg.h>
#include <stddef.h>

// Without this gl.h gets included instead of gl3.h
// #define GLFW_INCLUDE_NONE
#include <GLFW/glfw3.h>
// For includes related to OpenGL, make sure their are included after glfw3.h
// #include <OpenGL/gl3.h>

#ifdef __APPLE__
    #include <OpenCL/opencl.h>
#else
    #include <CL/cl.h>
#endif

#define GL_SILENCE_DEPRECATION
#define MAX_SOURCE_SIZE (0x100000)
#define SIZE 1024 * 2
#define LIST_SIZE SIZE *SIZE

/**
 *
 * @brief Function prototypes.
 *
 */
void error_callback(int, const char *);
void key_callback(GLFWwindow *, int, int, int, int);
void frame_buffer_resize_callback(GLFWwindow *, int, int);
void renderer(unsigned char *, unsigned char *);

/**
 *
 * @brief Error callback function.
 * @param error The error code.
```

```
 * @param description The error description.
 * @return void
 *
 */
void error_callback(int error, const char *description) {
    fputs(description, stderr);

    return;
}

/**
 *
 * @brief Key callback function.
 * @param window The window.
 * @param key The key.
 * @param scancode The scancode.
 * @param action The action.
 * @param mods The mods.
 * @return void
 *
 */
void key_callback(GLFWwindow *window, int key, int scancode, int action, int
    mods) {
    if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS) {
        glfwSetWindowShouldClose(window, GLFW_TRUE);
    }

    return;
}

/**
 *
 * @brief Frame buffer resize callback function.
 * @param window The window.
 * @param width The width.
 * @param height The height.
 * @return void
 *
 */
void frame_buffer_resize_callback(GLFWwindow *window, int width, int height) {
    glViewport(0, 0, width, height);
}

/**
 *
 * @brief Render the image.
```

```
 * @param data The image data.
 * @param A The image data.
 * @return void
 *
 */
void renderer(unsigned char *data, unsigned char *A) {
    for (int y = 0; y < SIZE; ++y) {
        for (int x = 0; x < SIZE; ++x) {
            data[y * SIZE * 3 + x * 3] = A[y * SIZE * 4 + x * 4];
            data[y * SIZE * 3 + x * 3 + 1] = A[y * SIZE * 4 + x * 4 + 1];
            data[y * SIZE * 3 + x * 3 + 2] = A[y * SIZE * 4 + x * 4 + 2];
        }
    }
}

/**
 *
 * @brief Main function.
 * @param void
 * @return void
 *
 */
int main(void) {
    clock_t begin = clock();

    if (!glfwInit()) {
        exit(1);
    }

    GLFWwindow *window = glfwCreateWindow(1024, 1024, "CP431: Julia Sets",
        NULL, NULL);

    if (!window) {
        glfwTerminate();
        exit(1);
    }

    unsigned char *A = (unsigned char *) malloc(sizeof(unsigned char) *
        LIST_SIZE * 4);
    float B[2] = {-0.8, 0.143};

    FILE *filename = fopen("julia.cl", "r");
    if (!filename) {
        fprintf(stderr, "Failed to load kernel.\n");
        exit(1);
    }
```

```
char *source_str = (char *) malloc(MAX_SOURCE_SIZE);
size_t source_size = fread(source_str, 1, MAX_SOURCE_SIZE, filename);

fclose(filename);

cl_platform_id platform_id = NULL;
cl_device_id device_id = NULL;
cl_uint ret_num_devices;
cl_uint ret_num_platforms;
cl_int ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id,
    &ret_num_devices);

cl_context context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);

cl_command_queue command_queue = clCreateCommandQueue(context, device_id,
    0, &ret);

cl_mem a_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY, LIST_SIZE *
    sizeof(unsigned char) * 4, NULL, &ret);
cl_mem b_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float)
    * 2, NULL, &ret);

ret = clEnqueueWriteBuffer(command_queue, a_mem_obj, CL_TRUE, 0, LIST_SIZE
    * sizeof(unsigned char) * 4, A, 0, NULL, NULL);
ret = clEnqueueWriteBuffer(command_queue, b_mem_obj, CL_TRUE, 0,
    sizeof(float) * 2, B, 0, NULL, NULL);

cl_program program = clCreateProgramWithSource(context, 1, (const char
    **)&source_str, (const size_t *)&source_size, &ret);

ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

cl_kernel kernel = clCreateKernel(program, "julia", &ret);

ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&a_mem_obj);
ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&b_mem_obj);

size_t global_item_size = LIST_SIZE;
size_t local_item_size = 64;

ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
    &global_item_size, &local_item_size, 0, NULL, NULL);

ret = clEnqueueReadBuffer(command_queue, a_mem_obj, CL_TRUE, 0, LIST_SIZE *
```

```
    sizeof(unsigned char) * 4, A, 0, NULL, NULL);
unsigned char *data = (unsigned char *) malloc(sizeof(unsigned char) *
    LIST_SIZE * 3);

for (int y = 0; y < SIZE; ++y) {
    for (int x = 0; x < SIZE; ++x) {
        data[y * SIZE * 3 + x * 3] = A[y * SIZE * 4 + x * 4];
        data[y * SIZE * 3 + x * 3 + 1] = A[y * SIZE * 4 + x * 4 + 1];
        data[y * SIZE * 3 + x * 3 + 2] = A[y * SIZE * 4 + x * 4 + 2];
    }
}

glfwMakeContextCurrent(window);

GLuint texture_handle;
glGenTextures(1, &texture_handle);
glBindTexture(GL_TEXTURE_2D, texture_handle);
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, SIZE, SIZE, 0, GL_RGB,
    GL_UNSIGNED_BYTE, data);

clock_t end = clock();
double time_spent = ((double)(end - begin) / CLOCKS_PER_SEC);
printf("Pack time: %f < %c >\n", time_spent, A[4]);

while (!glfwWindowShouldClose(window)) {
    int width;
    int height;

    begin = clock();

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    B[1] += 0.0001;
    ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *) &a_mem_obj);
    ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *) &b_mem_obj);
    ret = clEnqueueWriteBuffer(command_queue, b_mem_obj, CL_TRUE, 0,
        sizeof(float) * 2, B, 0, NULL, NULL);
    ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
        &global_item_size, &local_item_size, 0, NULL, NULL);
    ret = clEnqueueReadBuffer(command_queue, a_mem_obj, CL_TRUE, 0,
```

```
            LIST_SIZE * sizeof(unsigned char) * 4, A, 0, NULL, NULL);
        renderer(data, A);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, SIZE, SIZE, 0, GL_RGB,
            GL_UNSIGNED_BYTE, data);

        glfwGetFramebufferSize(window, &width, &height);
        glViewport(0, 0, width, height);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(0, width, 0, height, -1, 1);
        glMatrixMode(GL_MODELVIEW);
        glEnable(GL_TEXTURE_2D);
        glBindTexture(GL_TEXTURE_2D, texture_handle);
        glBegin(GL_QUADS);
        glTexCoord2d(0, 0);
        glVertex2i(0, 0);
        glTexCoord2d(1, 0);
        glVertex2i(SIZE, 0);
        glTexCoord2d(1, 1);
        glVertex2i(SIZE, SIZE);
        glTexCoord2d(0, 1);
        glVertex2i(0, SIZE);
        glEnd();
        glDisable(GL_TEXTURE_2D);

        glfwSwapBuffers(window);
        glfwWaitEvents();

        glfwPollEvents();

        end = clock();
        time_spent = ((double)(end - begin) / CLOCKS_PER_SEC);
    }

    glfwDestroyWindow(window);
    glfwTerminate();

    return 0;
}
```

2.3 VideoCPU Implementation (*LargePNG*)

- (*block.h*)

```c
#include <png.h>

/**
 *
 * @file block.h
 * @brief Represents a block of data in the final video output.
 *
 */
typedef struct _Block {
    int size;       // The size of the block
    int place;      // The place of the block
    int width;      // The width of the block
    int height;     //  The height of the block
    png_byte **data; // The location of the image data of the block
} Block;
```

- (*frame.h*)

```c
/**
 *
 * @file frame.h
 * @brief Represents a frame in the video.
 *
 */
typedef struct _Frame {
    int index;      // The position of this frame in the video.
    float real;     // The real value of the point of this frame
    float imaginary; // The imaginary value of the point of this frame
} Frame;
```

- (*main.c*)

```c
#include <complex.h>
#include <math.h>
#include <mpi.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include "block.h"
```

```
#include "frame.h"

#define FRAME_COUNT 100
#define MAX_ITER 50
#define PNG_DEBUG 3
#define SIZE 1000
#define OUTPUT_FILE_NAME "a.png"

/**
 *
 * @brief Global variables.
 *
 */
int x;
int y;

int WIDTH;
int HEIGHT;
int NUM_PASSES;

png_byte COLOR_TYPE;
png_byte BIT_DEPTH;
png_byte **IMAGE_DATA;

png_structp IMAGE;
png_infop IMAGE_INFO;
png_bytep *ROW_PTR;

Frame FRAME;
Block KNAPSACK;

/**
 *
 * @brief Function prototypes.
 *
 */
void write_image();
void calculate(int index);
void allocate();
void root_process();
void child_process(int rank, int processor_count);

/**
 *
 * @brief Write the image to a file.
 * @param void
```

```
 * @return void
 *
 */
void write_image() {
    char buffer[12];
    snprintf(buffer, 12, "%d.png", FRAME.index);
    file = buffer;
    COLOR_TYPE = PNG_COLOR_TYPE_RGBA;
    BIT_DEPTH = 8;
    NUM_PASSES = 1;

    FILE *file = fopen(file, "wb");
    if (!file) {
        fprintf(stderr, "[write_png_file] File %s could not be opened for
            writing", file);
        exit(1);
    }

    IMAGE = png_create_write_struct(PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);
    if (!IMAGE) {
        fprintf(stderr, "[write_png_file] png_create_write_struct failed");
        exit(1);
    }

    IMAGE_INFO = png_create_info_struct(image);
    if (!IMAGE_INFO) {
        fprintf(stderr, "[write_png_file] png_create_info_struct failed");
        exit(1);
    }

    if (setjmp(png_jmpbuf(IMAGE))) {
        fprintf(stderr, "[write_png_file] Error during init_io");
        exit(1);
    }

    png_init_io(IMAGE, file);
    if (setjmp(png_jmpbuf(IMAGE))) {
        fprintf(stderr, "[write_png_file] Error during writing header");
        exit(1);
    }

    png_set_IHDR(IMAGE, IMAGE_INFO, WIDTH, HEIGHT, BIT_DEPTH, COLOR_TYPE,
                 PNG_INTERLACE_NONE, PNG_COMPRESSION_TYPE_BASE,
                     PNG_FILTER_TYPE_BASE);

    png_write_info(IMAGE, IMAGE_INFO);
```

```
    if (setjmp(png_jmpbuf(IMAGE))) {
        fprintf(stderr, "[write_png_file] Error during writing bytes");
        exit(1);
    }

    png_set_compression_level(IMAGE, 6);
    png_set_filter(IMAGE, 0, 0);
    png_write_image(IMAGE, KNAPSACK.data);

    if (setjmp(png_jmpbuf(IMAGE))) {
        fprintf(stderr, "[write_png_file] Error during end of write");
        exit(1);
    }

    png_write_end(IMAGE, NULL);
    fclose(file);

    return;
}

/**
 *
 * @brief Calculate the mandelbrot set for a given frame.
 * @param index The index of the frame.
 * @return void
 *
 */
void calculate(int index) {
    float scale = 1.5;
    float cx = FRAME.real;
    float cy = FRAME.imaginary;
    float zx = (scale * (float)(WIDTH / 2 - x) / (WIDTH / 2));
    float zy = (scale * (float)(HEIGHT / 2 - y) / (HEIGHT / 2));
    int i = 0;
    int c = 0;
    float temp = (zx * zx - zy * zy + cx);

    for (int y = index; y < HEIGHT; y++, c++) {
        png_byte *row = KNAPSACK.data[y];
        for (int x = 0; x <= WIDTH; x++) {
            png_byte *ptr = &(row[x * 4]);
            int bit = 0;
            zx = (scale * (float)(WIDTH / 2 - x) / (WIDTH / 2));
            zy =( scale * (float)(HEIGHT / 2 - y) / (HEIGHT / 2));
            for (int i = 0; i < MAX_ITERATIONS; i++) {
```

```
                temp = (zx * zx - zy * zy + cx);
                zy = (2.0 * zx * zy + cy);
                zx = temp;
                if (zx * zx + zy * zy > 4.0) {
                    bit = i % 5;
                    if (bit == 0) {
                        bit = 5;
                    }

                    break;
                }
            }

            if (bit == 0) {
                ptr[0] = 0;
                ptr[1] = 0;
                ptr[2] = 0;
            } else if (bit == 1) {
                ptr[0] = 204;
                ptr[1] = 107;
                ptr[2] = 73;
            } else if (bit == 2) {
                ptr[0] = 210;
                ptr[1] = 162;
                ptr[2] = 76;
            } else if (bit == 3) {
                ptr[0] = 236;
                ptr[1] = 230;
                ptr[2] = 194;
            } else if (bit == 4) {
                ptr[0] = 115;
                ptr[1] = 189;
                ptr[2] = 168;
            } else {
                ptr[0] = 153;
                ptr[1] = 190;
                ptr[2] = 183;
            }

            ptr[3] = 255;
        }
    }

    return;
}
```

```
/**
 *
 * @brief Allocate memory for the image data.
 * @param void
 * @return void
 *
 */
void allocate() {
    IMAGE_DATA = (png_byte **) malloc(sizeof(png_byte *) * HEIGHT);
    for (int y = 0; y < HEIGHT; y++) {
        IMAGE_DATA[y] = (png_byte *) malloc(sizeof(png_bytep) * WIDTH);
    }

    KNAPSACK.height = HEIGHT;
    KNAPSACK.width = WIDTH;
    KNAPSACK.size = HEIGHT * WIDTH;
    KNAPSACK.place = 0;
    KNAPSACK.data = IMAGE_DATA;

    return;
}

/**
 *
 * @brief Code ran by process rank 0 to handle results from all child processes.
 * @param void
 * @return void
 *
 */
void root_process() {
    bool next = true;

    for (int i = 0; i < FRAME_COUNT; i++) {
        FRAME.imaginary = 0.136 + (0.001 * i);
        FRAME.index = i;
        FRAME.real = -0.8 + (0.001 * i);

        MPI_Recv(&next, 1, MPI_INT, MPI_ANY_SOURCE, 3, MPI_COMM_WORLD, &status);
        fprintf(stdout, "%d <<<\n", next);
        MPI_Send(&FRAME, 4, MPI_INT, next, 1, MPI_COMM_WORLD);
    }

    FRAME.index = -1;

    for (int i = 1; i < processor_count; i++) {
        MPI_Send(&FRAME, 4, MPI_INT, i, 1, MPI_COMM_WORLD);
```

```
    }

    return;
}

/**
 *
 * @brief Code ran by all child processes to handle work from the root process.
 * @param rank The rank of the child process.
 * @param processor_count The total number of processes.
 * @return void
 *
 */
void child_process(int rank, int processor_count) {
    while (FRAME.index >= 0) {
        MPI_Send(&rank, 1, MPI_INT, 0, 3, MPI_COMM_WORLD);
        MPI_Recv(&FRAME, 4, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        if (FRAME.index != -1) {
            allocate();
            calculate(0);
            write_image();
        }
    }

    fprintf(stdout, "Recieving %f - %d - %f \n", FRAME.imaginary, FRAME.index,
        FRAME.real);

    for (int y = 0; y < HEIGHT; y++) {
        free(KNAPSACK.data[y]);
    }

    free(KNAPSACK.data);

    return;
}

/**
 *
 * @brief Main function.
 * @param argc The number of arguments.
 * @param argv The arguments.
 * @return 0
 *
 */
int main(int argc, char **argv) {
    int rank;
```

```
    int processor_count;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &processor_count);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    WIDTH = 3840;
    HEIGHT = 2160;

    FRAME.imaginary = 0.0;
    FRAME.index = 0;
    FRAME.real = 0.0;

    allocate();

    if (rank == 0) {
        root_process();
    } else {
        child_process(rank, processor_count);
    }

    MPI_Finalize();

    return 0;
}
```

3 Benchmarks

3.1 Local System Benchmarks

The program was also run on a Macbook Pro 16" (2019) Core i7-9750H, which consists of six cores and 16GB of 2666 MHz DDR4 SDRAM. It also has an AMD Radeon Pro 5300M GPU with 4GB of GDDR6 memory. Here are the results:

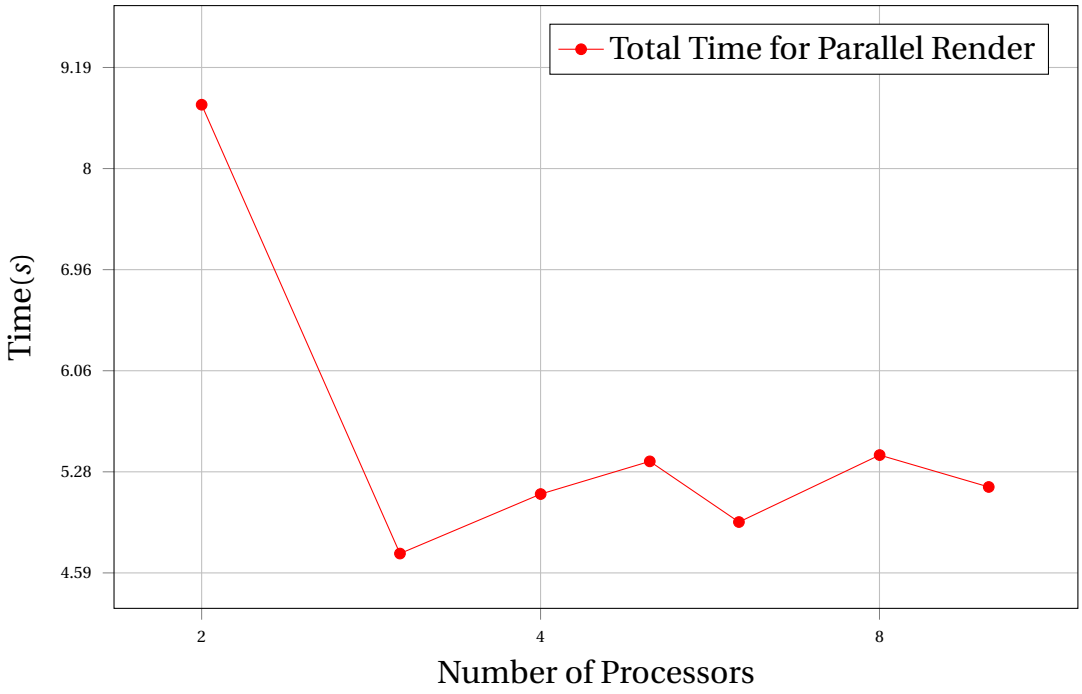
- OpenCL version provides real-time display of julia sets with frame generation taking 10-15 ms at 1080p resolution.
- MPI to generate video runs at approximately 300 ms per frame at 4k resolution.
- PNG lib saves image of 50,000x50,000 pixels in approximately 45s with filters and compression lowered modestly.
- Passing messages of block size 1000 is usually under 10ms.

3.2 Teach Cluster System Information

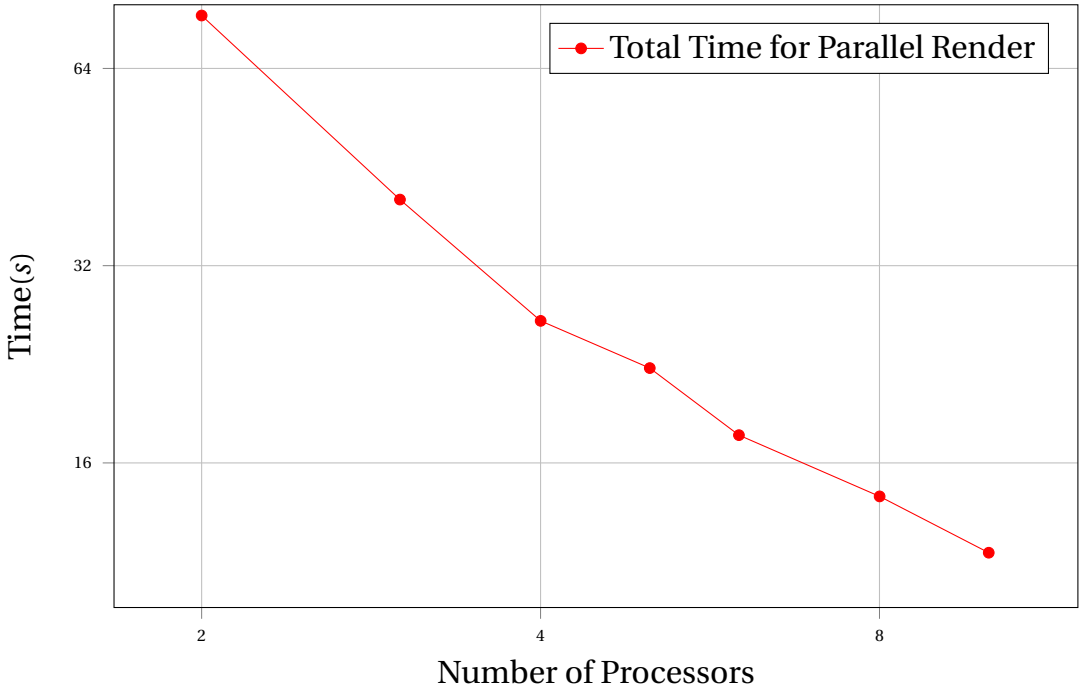
These benchmarks were run on the Teach SciNet cluster. The cluster consists of 42 repurposed x86_64 nodes each with 16 cores (from two octal core Intel XeonSandybridge E5-2650 CPUs) running at 2.0GHz with 64GB of RAM per node.

3.3 Graphs

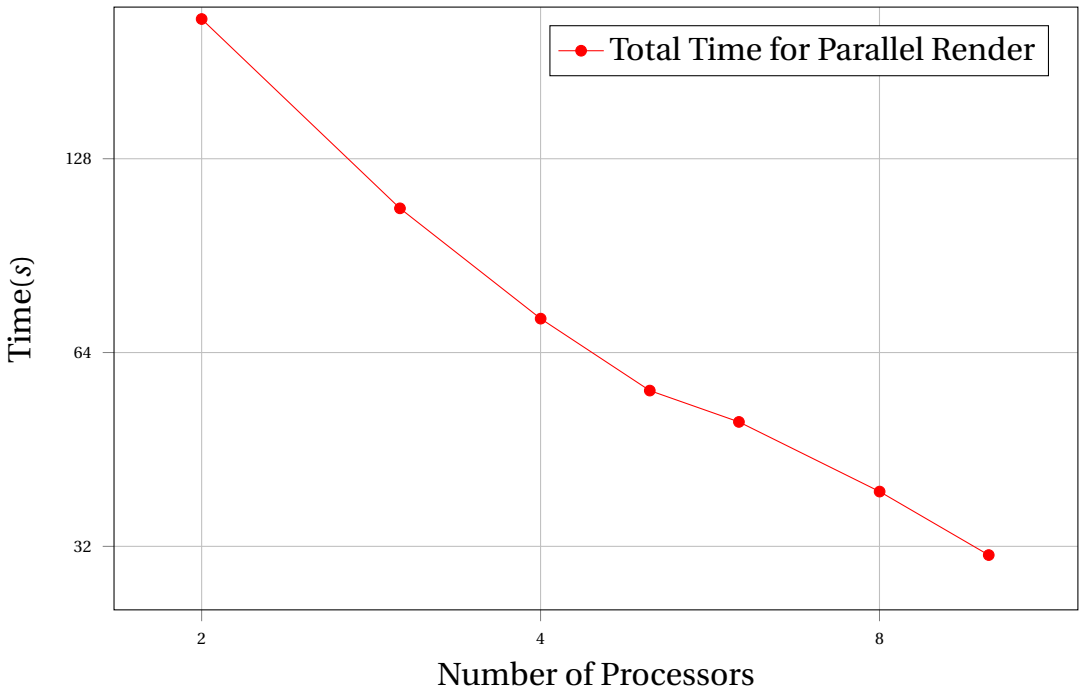
3.3.1 10,000 x 10,000 Pixels



3.3.2 30,000 x 30,000 Pixels



3.3.3 50,000 x 50,000 Pixels



4 Analysis

4.1 Code Analysis

4.1.1 MPI - Passing Structs

When using the MPI library for parallel computing in C, passing arrays of pointers to structures can be problematic since the memory of the struct may not be adjacent. This often requires the use of a buffer or the individual sending of pointers. Passing structs instead can be a more straightforward approach to organizing data. However, it's important to note that structs may contain padding, which needs to be handled appropriately. While using buffers like Flatbuffers or Protobuffers can be a solution, they require the use of external frameworks and can be challenging to implement.

4.1.2 PNG Library

The PNG file format was chosen due to its lossless nature and ability to compress image files without compromising image quality. Saving images in PNG format can significantly reduce the file size compared to other formats. The PNG format allows for varying levels of compression, which impacts both the size of the output file and the time required to compute it. While higher compression can produce smaller files, the returns diminish at the highest compression levels. Standard PNG libraries are available that can make code portable and quick to develop, making PNG a viable option for image compression in a variety of applications.

4.1.3 Load Balancing

For load balancing, we initially distributed work to all processes from the main process. All data was passed using a 'Block' with a variable name 'knapsack.' This knapsack contains important values such as the size of the block, index, is_done flag, and others. Once the data was distributed, the main process went into the waiting state and waited for a completed process to send back the computed data. The message passing time was significantly shorter than the compute time, so the processes did not spend much time idle while waiting to send back the computed arrays.

4.1.4 Load Balancing for Frame Generation

The process of generating frames involved a master-slave approach where processes were assigned a frame number and a starting value to generate. Upon completion of a frame, the process, P_n, communicated with the main process and requested the generation of another frame. The generated frames were saved to disk by each process with its respective frame number as the filename. To combine all frames into a video, an FFMpeg script was used, which added compression based on user-defined parameters such as Bitrate, Codec, Frame Rate, and Resolution, among others. This

approach facilitated the efficient generation and storage of frames for video production.

4.2 Benchmark Analysis

A clear trend can be noted as the resolution size increases, that is - from $10,000 \times 10,000$ pixels to $30,000 \times 30,000$ pixels, it can be seen that parallelism has a great effect on the rendition. This implies that since creating Julia Sets require a great amount of computational power, as the number of cores increases, it takes significantly less time to create the render of the image. This trend continues with proportionality to the resolution size. While not shown in the graphs, a test was conducted on the resolution size of $100,000 \times 100,000$ pixels, and it was found that it takes approximately $110.3s$ for 10 cores; Thus, it can be extrapolated to verify that the results found are indeed accurate.

4.3 Problems / Issues Encountered & Solutions

- **Passing Double and Triple pointers to Processes**
  One of our initial issues was allocating memory to a triple pointer and then deconstructing this in order to be passed to mpi processes. This can be difficult to synchronize as multiple send and receives are required, in a loop. This is because a pointer of pointers contains locations in memory that are likely scattered across ram. since these elements are not in a predictable sequence they need to be deconstructed before being sent. From our testing this did not impact performance in a meanful way, and did not justify the use of a third party buffer system like Google FlatBuffers.

- **Sharing memory between the GPU and OpenGL**
  Memory needs to be pasted from cpu to gpu and back. The memory also needs to be divided into 3 byte chucks since each coordinate need red, green, and blue pixel data. This has a solution in OpenCL using a special variable naming scheme. This became a problem however since the documentation doesn't clearly state that memory can only be divide in powers of two and lead to results that were skewed and not coloured properly. This was difficult to debug because of the limited capability of the gpu, espcially in making system calls for logging data. Through some debugging using very small images we were able to find this issue and the properly pass memory to the gpu for computation.

- **Communication Error**
  In the initial design of the system, segmentation faults would occur after an allocation of 11 cores was reached. A segmentation fault typically occurs when the program tries to access memory that is outside of its reach and does not have access to it. In this particular case, we noticed memory exceeding the virtual machines limit. When using Windows subsystem for linux errors can occur

with allocating large blocks of memory and the VM is limited in the amount of swap space that can be used. By fixing our settings and testing outside the VM we were able to remedy this issue.

References

[1] "Teach SciNet Documentation." Teach. (28 January 2023). Retrieved February 28, 2023, *https://docs.scinet.utoronto.ca/index.php/Teach*