

1. Introduction

In this project, we are going to convert the sequential program provided to parallel version with the help of MPI and Pthread. This report will give the details of the design of each version and compare their performance.

2. Methods

In the given sequential file, the program would basically calculate the points in a square area with respect to the center and scale. Throughout the program, the part requiring the most computing resources is the calculation part, which would iterate through all the points and calculate for at most iteration and record their status. Since there are not critical connection between each two points among the points for calculation, that is the result of one point would not depend on any other points. In this scenario, we can divide the calculation task into several pieces and do the computing parallel with MPI and Pthread.

a) Dividing the task into several packages

Down to the details, we need a way to split the task into packages with approximately equal size. The total number of points is $size * size$, which is one of the given parameters. In the program, we apply the following formula to divide the task. For process with rank i , it will calculate the point with from the start index to the end index (start and end index included):

$$StartIndex_i = i * \frac{size * size}{number\ of\ total\ processes} + \min(i, (size * size) \% (number\ of\ total\ processes))$$
$$EndIndex_i = (i + 1) * \frac{size * size}{number\ of\ total\ processes} + \min(i + 1, (size * size) \% (number\ of\ total\ processes)) - 1$$

For instance, with size four, three processes would proceed the points at [0,5], [6,10], [11,15] respectively. In this case, the sizes of the sub-tasks are 6, 5 and 5 separately, which are approximately same.

b) Mechanism of plotting with updated parameters in GUI

Notice that the calculation in the program is repeating endlessly since it always need to update the value of points after adjusting parameters and replot the whole diagram. Thus, we

need to make sure that other processes or threads would interact with the master process and thread correctly.

c) MPI

Since we use rank 0 process as the master program which generate GUI and fetch the updated parameters from users, we need to make sure rank 0 pass necessary information to other ranks to do the corresponding calculation work. Following procedures are designed to make sure each process do the calculation correctly.

Step 1: Rank 0 pack the parameters (size of plotting square, scale, coordinates of the center) as a struct together and then broadcast them to each process.

Step 2: All process except rank 0 unpack the data received from rank 0. Next, all processes would prepare for the calculation, which includes figure out the start and end indexes of the points they need to calculate and then convert each index to the position at a square and calculate all factors involved in calculation like the position of x, y and zoom factor.

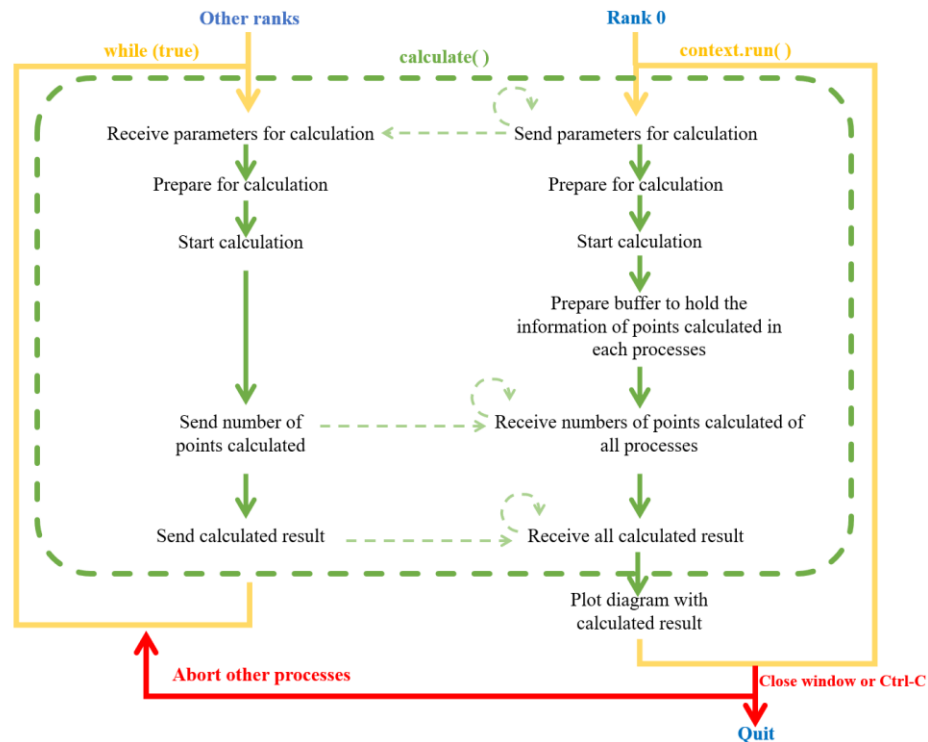
Step 3: Do the calculation of the points from the start position at the square until it finishes all of its job.

Step 4: Rank 0 gather the information of the number of points (task size) calculated by each process and store it in a vector. Then rank 0 will use the task size information to calculate the strip information. With task size and strip, rank 0 process would then gather all the data from all processes and put it to a buffer for plotting.

Step 5: Rank 0 process would then plot the diagram in GUI and all other processes would jump to step 1 and wait for rank 0. After plotting, rank 0 process would then jump back at step 1 and start a new round.

Step 6: If GUI window is closed or Ctrl-C is typed in the terminal, the rank 0 process would directly jump to step 6 and abort all other processes. In this case, you may see the abort signal prompted in the terminal which is a normal operation.

The details are also illustrated in the following figure.



d) Pthread

Besides using multi-processes, we can also use multi-threads to finish the calculation task. In this scenario, updating the diagram with new parameters would be much easier since we can directly pass parameters to threads when forking them. Waiting for all the threads finish their calculation task, the master thread would then plot the diagram directly.

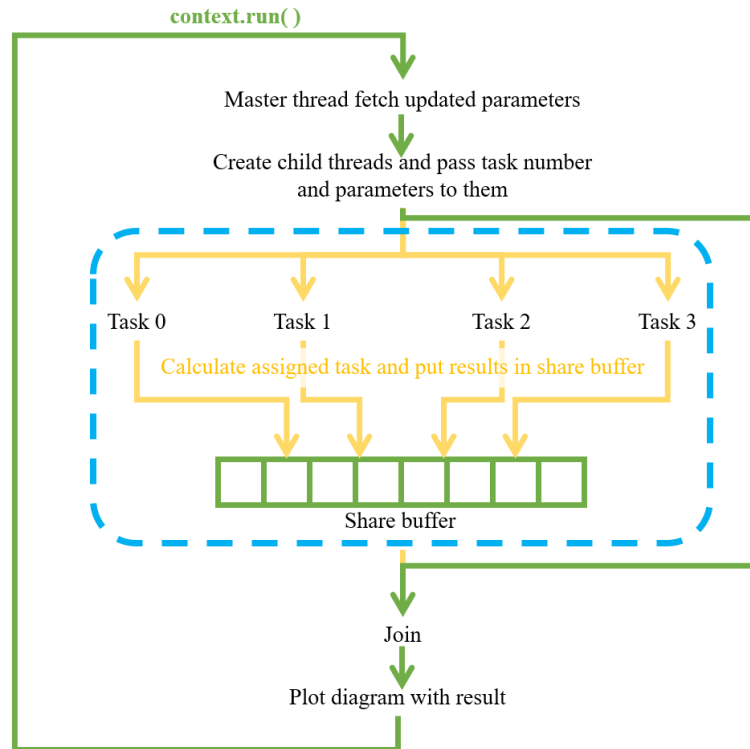
Following are the detailed steps.

Step 1: Master thread would fork several threads and passing the corresponding parameters to them.

Step 2: All child threads would then unpack the parameters passing to them and do the calculation jobs. Notice that all threads would store the calculation result to the buffer for plotting directly. Notice that no mutex is applied since regions of the buffer accessed by all threads are mutually exclusive. The main thread would wait all child threads finish their jobs by joining them. All the forked threads would then be cleared up since they are just set as local variables in the function which would be destroyed automatically after returning from the function.

Step 3: The program would then plot the results in the GUI and then jump to step 1.

The workflow is shown in the following:



3. Program execution

Besides MPI and Pthread version, the project also implements the MPI benchmark and Pthread benchmark versions which get rid of plotting and only focus on the calculation steps.

a) Compile and build

For all programs, please put them (the whole folder of each program into /pvfsmnt/(student_id)). Type 'mkdir build' to create a folder called 'build' and type 'cd build' to enter it. Type 'cmake .. -DCMAKE_BUILD_TYPE=Debug' and 'cmake --build . -j4' to build and compile the program.

Also, you need to do some authorization settings before executing the programs. In a suitable environment (all the executions of programs in this project are done in the virtual machine set up in tutorial 1), type "xhost +" first. Then connect to the server using "ssh -Y {student id}@10.26.1.30". Then type "cd /pvfsmnt/\${whoami}", "cp ~/.Xauthority /pvfsmnt/\${whoami}", "export XAUTHORITY=/pvfsmnt/\${whoami}/.Xauthority".

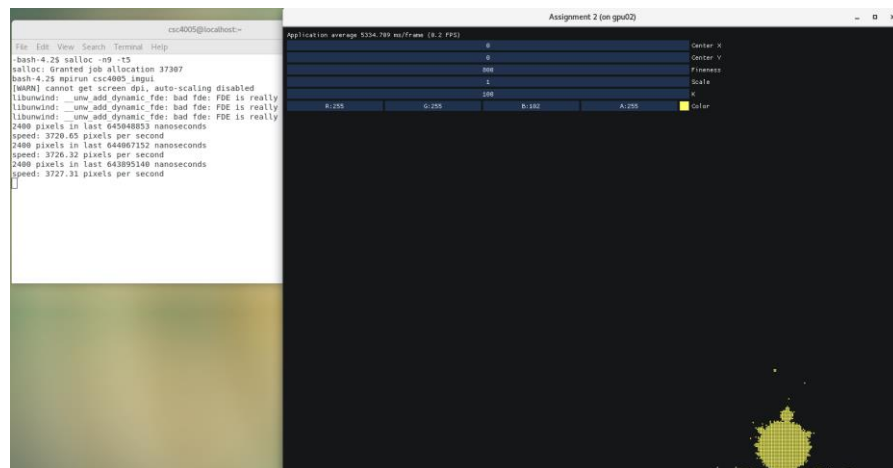
b) MPI

Since program run with multiple processes, you first need to allocate several cores.

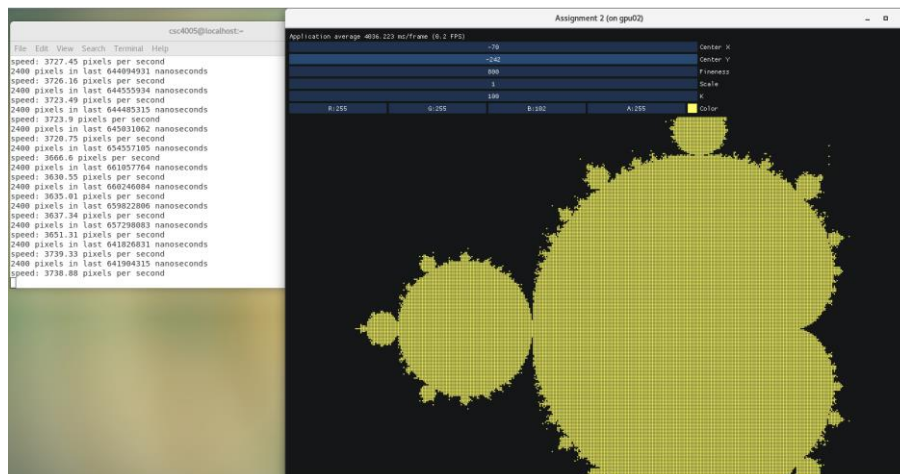
Type “salloc -n9 -t5” to request 9 cores for example. Then type “mpirun csc4005_imgui” in the build folder then it will prompt out the program.

Results:

1) Prompting GUI windows after execution



2) Adjust parameters in the GUI windows to change the plot

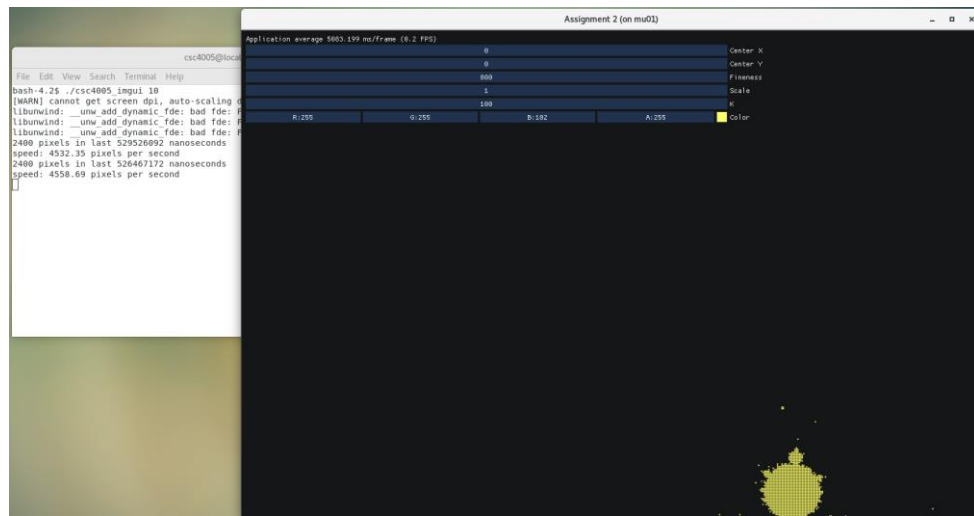


c) Pthread

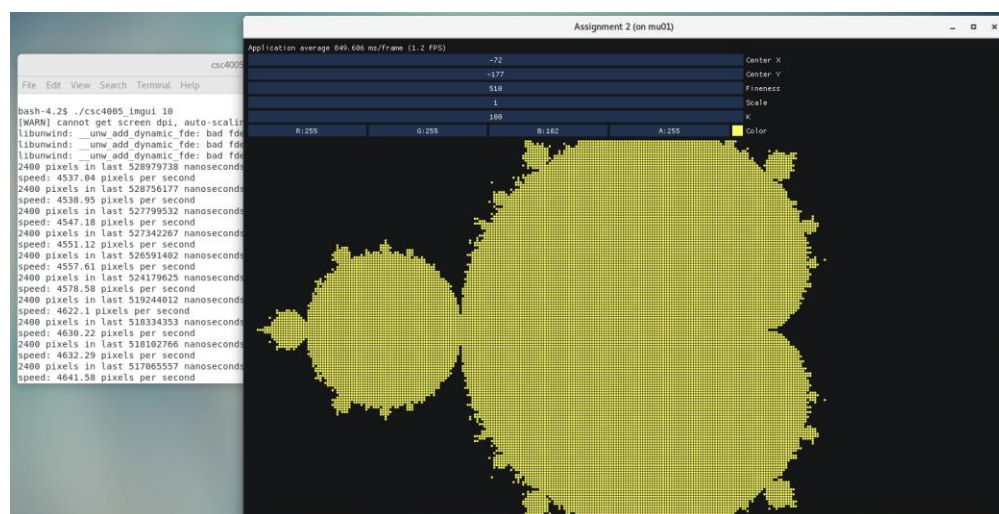
The steps to execute the pthread version program is relative easier, you just need to type “./csc4005_imgui [number of threads]” directly, for example, type “./csc4005_imgui 10” for execute program in 10 threads. The default setting is 1, which would be set up automatically if the input parameters is invalid or not given.

Results:

1) Prompting GUI windows after execution



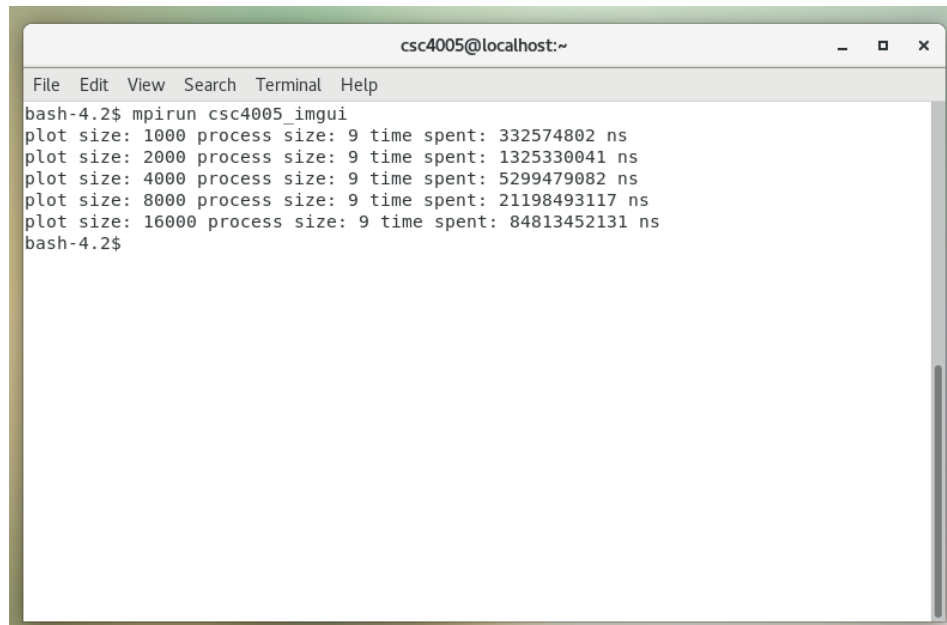
2) Adjust parameters in the GUI windows to change the plot



d) MPI benchmark

This program is specially designed to measure the running time of the algorithm in MPI version. Since program run with multiple processes, you first need to allocate several cores. Type “`salloc -n9 -t5`” to request 9 cores for example. Then type “`mpirun csc4005_imgui`” in the build folder then it will display the running results in the windows.

Result:



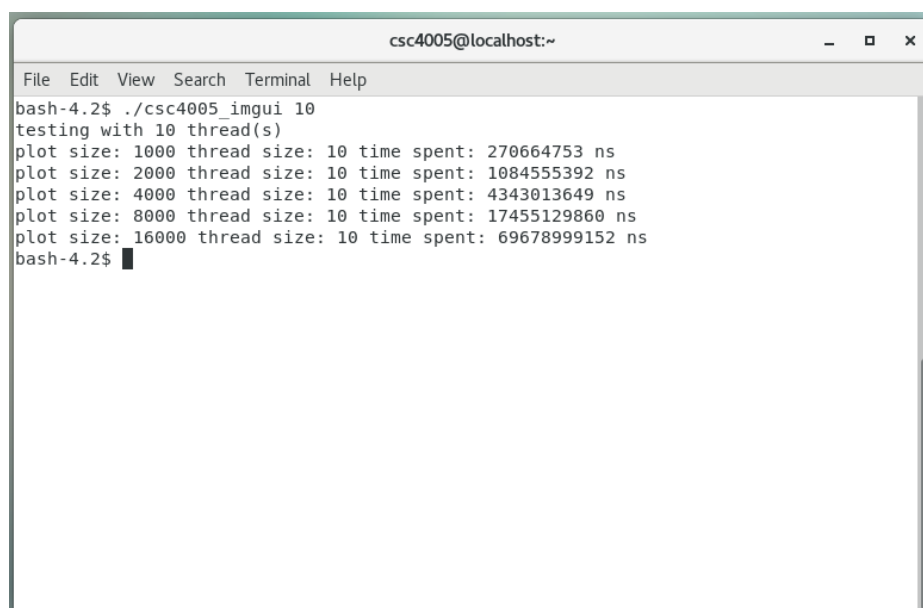
```
csc4005@localhost:~  
File Edit View Search Terminal Help  
bash-4.2$ mpirun csc4005_imgui  
plot size: 1000 process size: 9 time spent: 332574802 ns  
plot size: 2000 process size: 9 time spent: 1325330041 ns  
plot size: 4000 process size: 9 time spent: 5299479082 ns  
plot size: 8000 process size: 9 time spent: 21198493117 ns  
plot size: 16000 process size: 9 time spent: 84813452131 ns  
bash-4.2$
```

e) Pthread benchmark

This program is specially designed to measure the running time of the algorithm in Pthread version. You just need to type “./csc4005_imgui [number of threads]” directly, for example, type “./csc4005_imgui 10” for execute program in 10 threads. The default setting is 1, which would be set up automatically if the input parameters is invalid or not given.

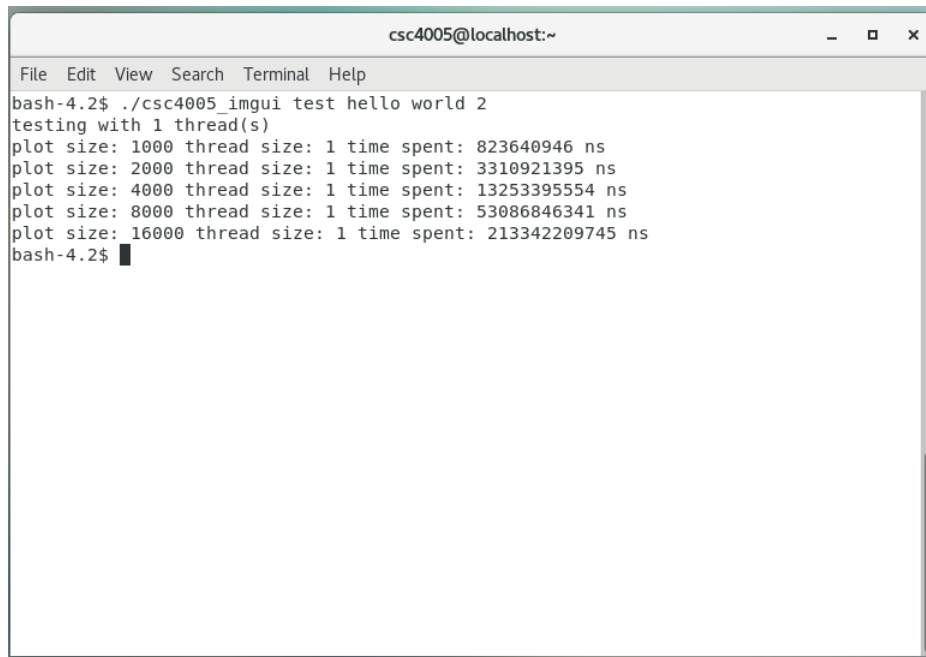
Results:

1) Run with parameters



```
csc4005@localhost:~  
File Edit View Search Terminal Help  
bash-4.2$ ./csc4005_imgui 10  
testing with 10 thread(s)  
plot size: 1000 thread size: 10 time spent: 270664753 ns  
plot size: 2000 thread size: 10 time spent: 1084555392 ns  
plot size: 4000 thread size: 10 time spent: 4343013649 ns  
plot size: 8000 thread size: 10 time spent: 17455129860 ns  
plot size: 16000 thread size: 10 time spent: 69678999152 ns  
bash-4.2$
```

2) Run with invalid parameters:



```
csc4005@localhost:~  
File Edit View Search Terminal Help  
bash-4.2$ ./csc4005_imgui test hello world 2  
testing with 1 thread(s)  
plot size: 1000 thread size: 1 time spent: 823640946 ns  
plot size: 2000 thread size: 1 time spent: 3310921395 ns  
plot size: 4000 thread size: 1 time spent: 13253395554 ns  
plot size: 8000 thread size: 1 time spent: 53086846341 ns  
plot size: 16000 thread size: 1 time spent: 213342209745 ns  
bash-4.2$
```

4. Performance analysis:

a) Time complexity:

In this analysis, we only consider the calculation task for one iteration. Also, we ignore the plotting steps.

i. Sequential version:

Assume the input size is N , basically for one iteration the program would do calculation for $N * N$ points which take $O(N^2)$. Thus, for one iteration, the time complexity would be $O(N^2)$.

ii. MPI version:

Assume the input size is N and the number of processes is p . For one iteration, the program would program would do calculation for $N * N$ points. Since we apply multi-processes to divide the task, generally each process would only need to deal with $N * N/p$ points which will take $O(\frac{N^2}{p})$. However, it will take extra $O(1)$ to pass the parameters from rank 0 process to other process and $O(N^2)$ to gather the data from all the processes. Thus,

the total time complexity would be $O(\frac{N^2}{p}) + O(1) + O(N^2) = O(N^2)$.

iii. Pthread version:

Assume the input size is N and the number of threads is t . For one iteration, the program would do calculation for $N * N$ points. Since we apply multi-threads to divide the task, generally each thread would only need to deal with $N * N/t$ points which will take $O(\frac{N^2}{t})$. In this case, the program would need extra $O(t)$ to create threads. After calculation, the results for each thread would be put directly on a shared buffer which need no extra time for collecting result. Also, since all threads do calculation on mutual exclusive regions on shared buffer which requires no mutex, all the threads would work independently and do not need to wait for others. In this case, the time complexity would be $O(\frac{N^2}{t}) + O(t) = O(\frac{N^2}{t} + t)$.

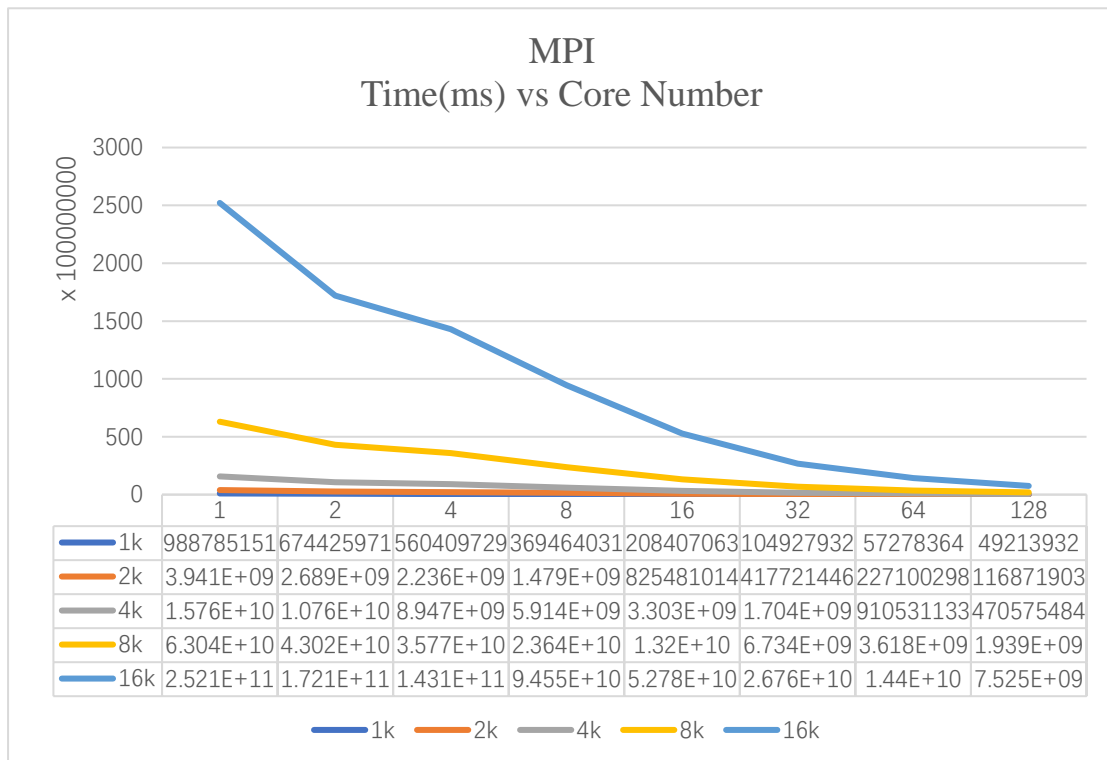
b) Test:

In this project, MPI and Pthread would work in the same way as sequential version. Thus, for testing, MPI with one process would work as the sequential version in the MPI testing group while Pthread with one thread as sequential version in the Pthread group. All the test results are generated from the MPI benchmark and Pthread benchmark version which only focus on the calculation step.

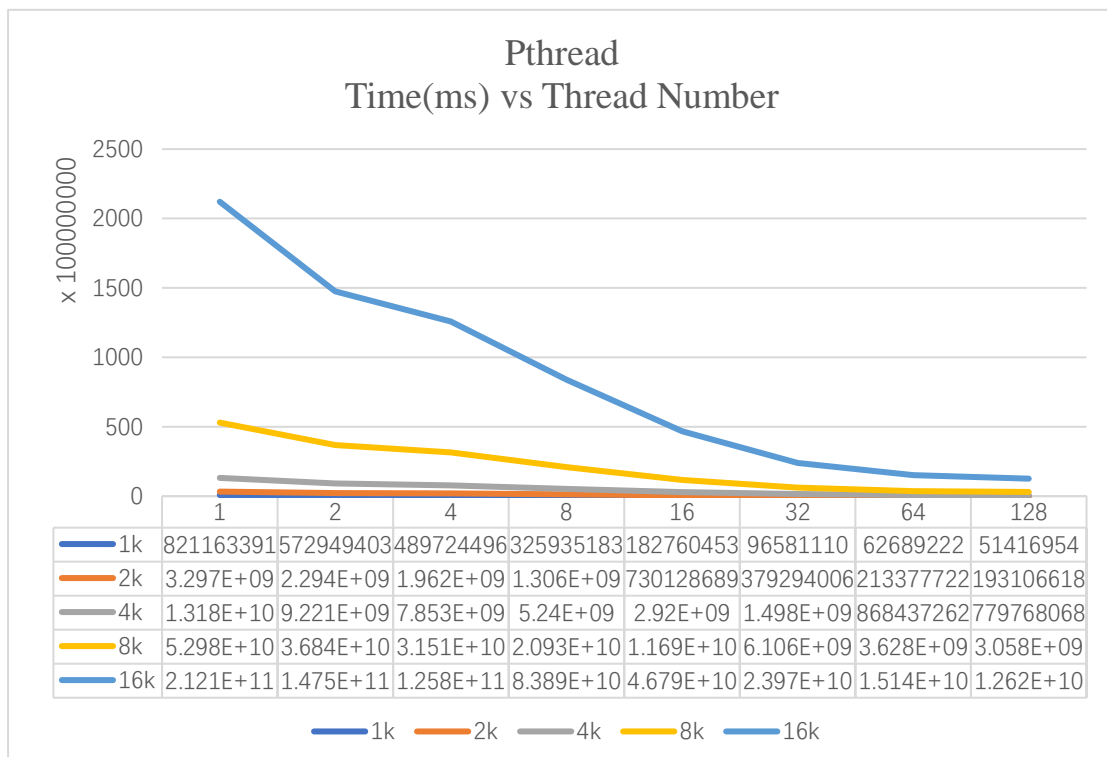
i. Time vs number of Core/Thread

For MPI, tests with core number with 1 to 32 were conducted in one node. Tests with 64 and 128 cores were conducted in 2 and 4 nodes separately.

Part 1. all test sets:



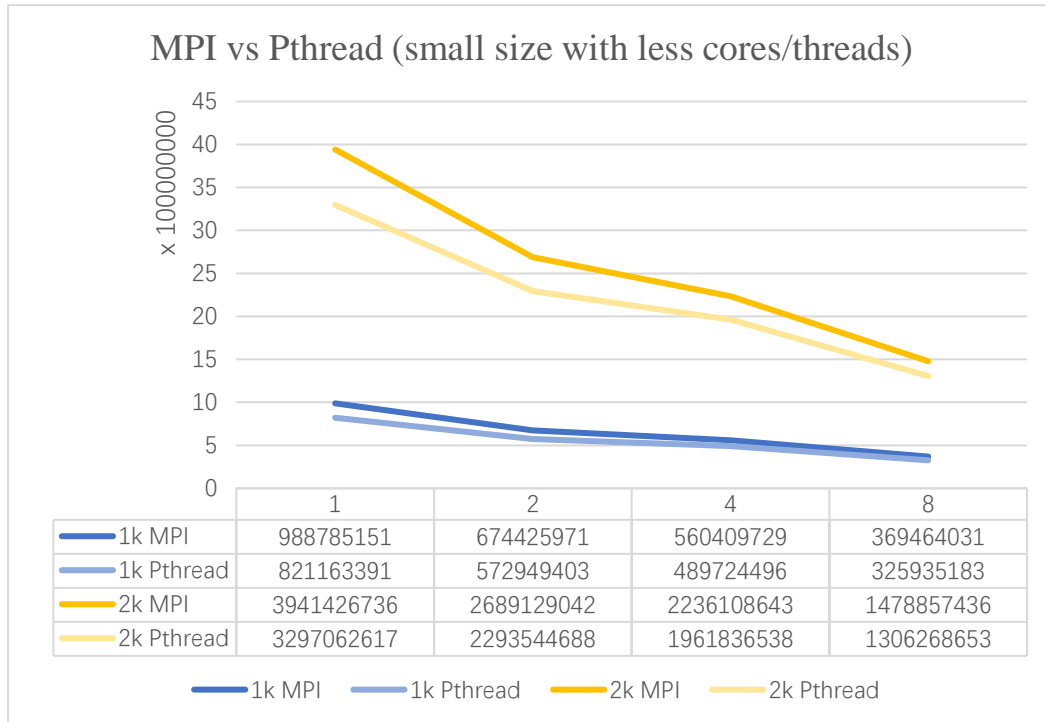
Pthread testing sets



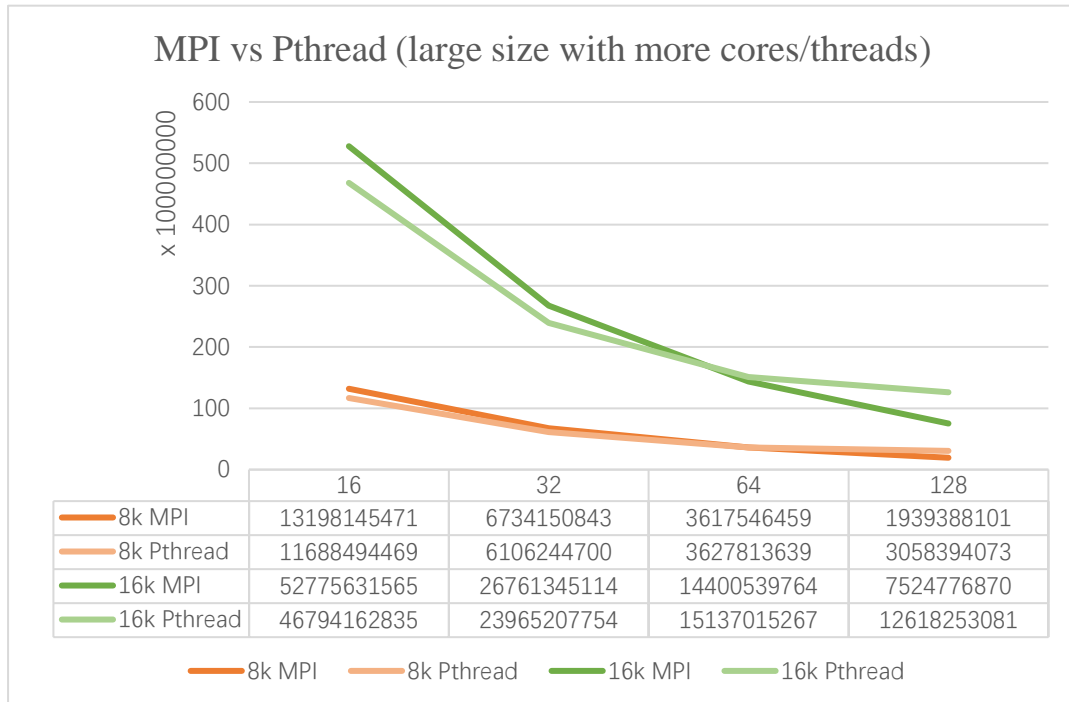
With the increase of number of processes and threads, the execution time decrease steadily for both MPI and Pthread version. The result is quite intuitive with the aid of more

processes and threads, each single process or thread would only need to deal with a smaller part of the calculation task thus results in a faster execution speed.

Part 2. MPI vs Pthread tests



Among the testing sets with small data size, Pthread generally performs slightly better than MPI. The communication overhead may be one of the possible reasons for this. In smaller test sets, it requires less time for calculation which may not offset the effect of the gathering results in MPI. This case would not happen in Pthread since all calculation are done within a shared buffer. And this smaller execution time of Pthread than MPI fix the pattern of their time complexity formulae discussed in the beginning of the section.



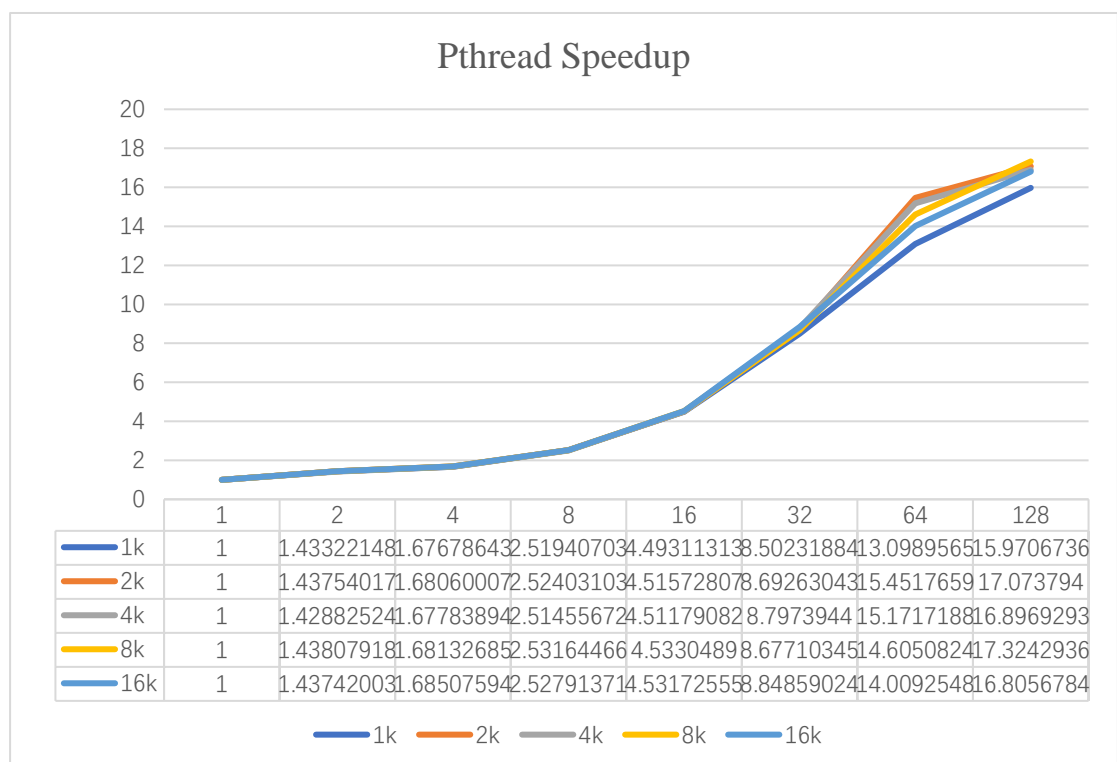
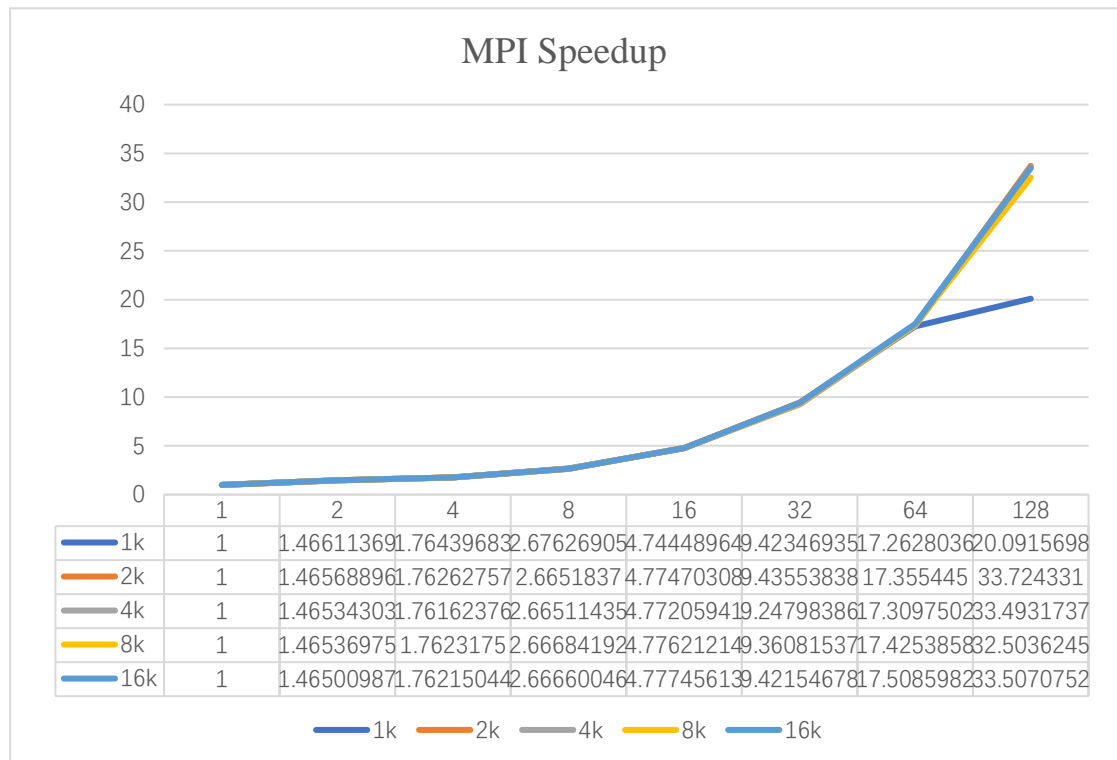
The result in within large data size and large cores shows a different pattern. With more cores/threads, the execution speed of MPI gradually surpass that of Pthread. In this case, the number of software thread (created in the program) may be far larger than the compartment of physical threads. To make sure all threads work concurrently, the processor may need to divide the execution into small time slice and swap threads who finish their job out and restore another new one from memory. When there are too many threads, the swap and restore overhead become large while this situation won't happen in the MPI. In this case, the communication overhead for MPI may be relative smaller compared to swapping overhead in the Pthread.

Part 3. Speedup

We can also calculate the Speedup of the program.

The formula of the Speedup is given as:

$$\text{Speedup} = \frac{\text{running time of sequentail version}}{\text{running time of parallel version}}$$



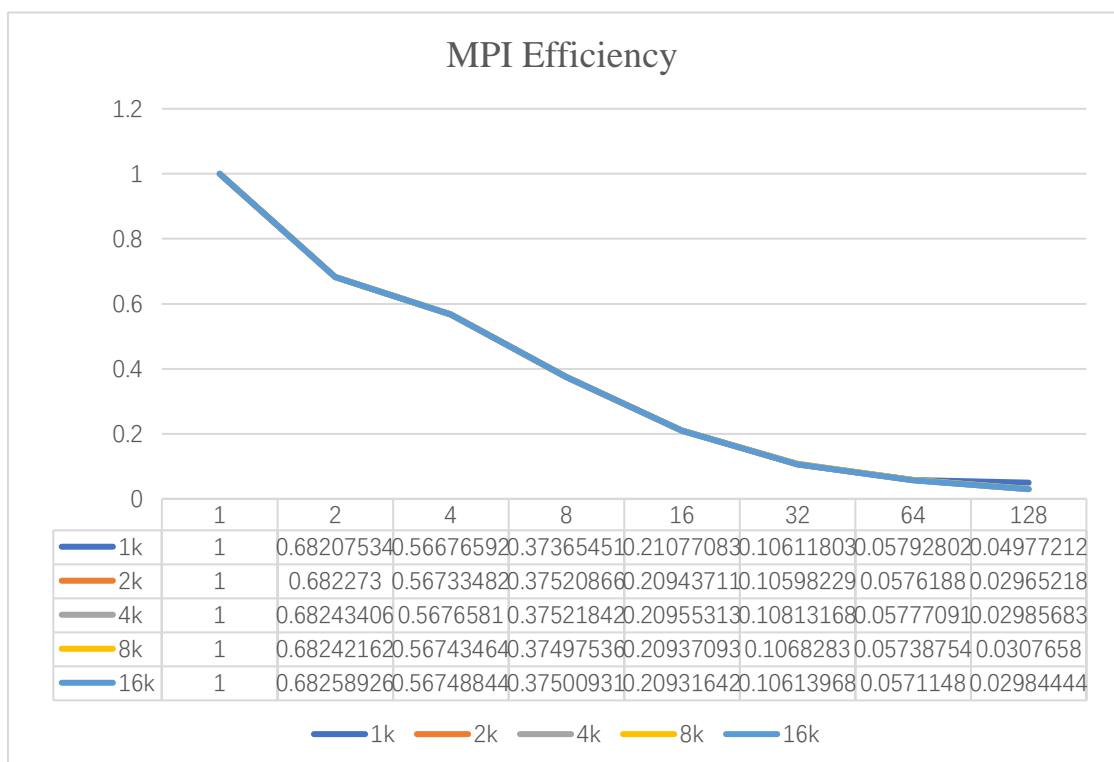
From the figures, it is easy to see that with more cores/threads, the speedup would be larger which is quite intuitive since more cores/threads would deal with smaller tasks requiring less execution time. Notice that smaller sizes tend to have a relatively small speedup, and it is reasonable. For MPI (Pthread), the ratio of communication time (time

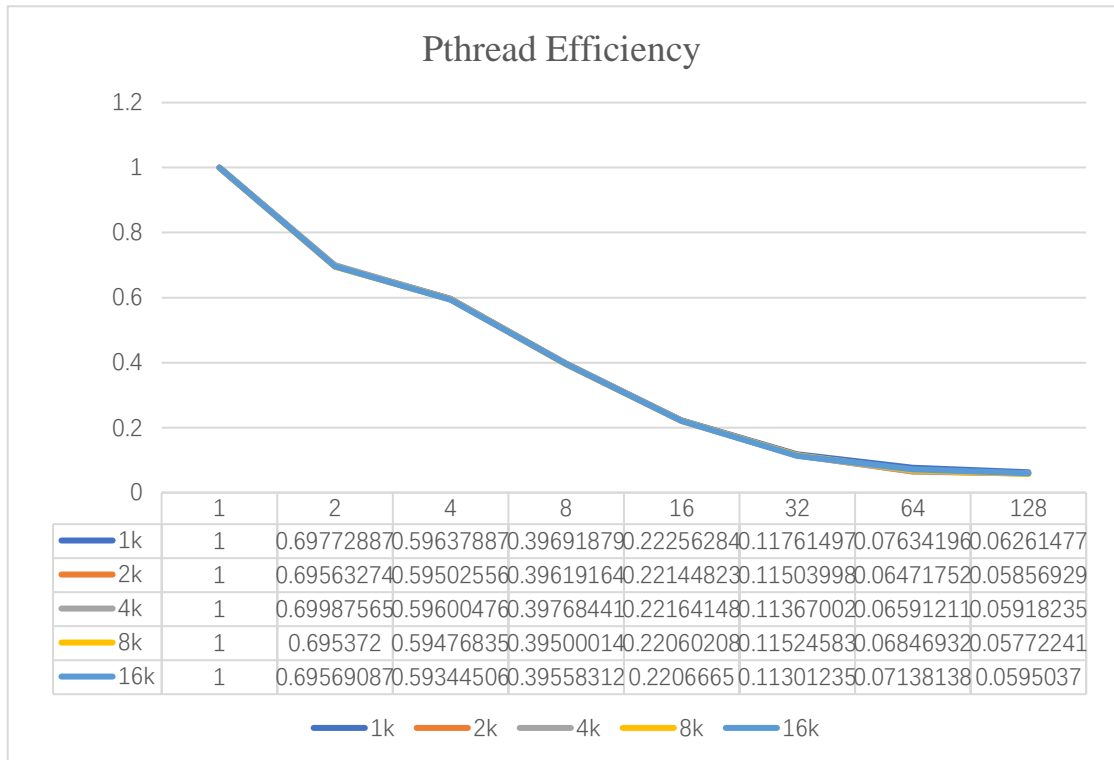
for swapping threads) over calculation time may be larger, in other words, calculation time could not fully offset the effect of communication time (time for swapping threads).

Part 4. Efficiency

The efficiency formula is given as:

$$E = \frac{\text{Speedup}}{\text{number of processes}}$$





In general, the efficiency will generally decrease with the increase of number of cores (threads). One possible reason is that with more cores (threads) running, the data size assigned to each process will decrease a lot, which result in the ratio of calculation to communication time (time for swapping threads) for each process (thread) increase. Since each core (thread) will spend generally more time for waiting to pass data between processes (waiting before other threads swap out), their running efficiency decrease.

5. Conclusion and Limitations

This report mainly discusses two parallel versions (MPI and Pthread) of the given Mandelbrot Set Computation sequential program. Tested with different number of cores and threads and data size, the report shows the improvement of parallel calculation and also discuss the communication overhead and thread switching may offset the performance improvement brought by parallel calculation.

To further improve the computation efficiency, we can separate the tasks into equal time-consuming pieces instead of equal size pieces. Notice that the calculations for each point in the required squares may be different (some points may require more calculation iteration while some may not). If these points requiring heavy calculation tasks lay in a region in a high-density

way, the process or thread assigned with points of this regions may takes more time for calculation while other threads could do nothing but waiting. In this case, we can actually divide task into smaller pieces and assign these pieces to each process and thread first. After finishing their jobs, some processes or threads would then require new pieces from the remaining. In this case, the execution time will be much approximately equal for all processes and threads.

Codes (main.cpp for different versions)

MPI:

```
#include <chrono>
#include <iostream>
#include <graphic/graphic.hpp>
#include <imgui_impl_sdl.h>
#include <vector>
#include <complex>
#include <mpi.h>
#include <cstring>

struct Square {
    std::vector<int> buffer;
    size_t length;

    explicit Square(size_t length) : buffer(length*length), length(length) {}

    void resize(size_t new_length) {
        buffer.assign(new_length * new_length, false);
        length = new_length;
    }

    auto& operator[](std::pair<size_t, size_t> pos) {
        return buffer[pos.second * length + pos.first];
    }
};

// define struct to pass information
typedef struct Arguments_S {
    int size;
    int scale;
    int k_value;
    double x_center;
    double y_center;
} Arguments;

static MPI_Datatype Arguments_TYPE;

void calculate(Square &buffer, int size, int rank, int processSize, int scale,
double x_center, double y_center, int k_value) {
    // rank 0 broadcast parameters
    Arguments arguments;
```

```

if(rank==0){
    arguments.size = size;
    arguments.scale = scale;
    arguments.k_value = k_value;
    arguments.x_center = x_center;
    arguments.y_center = y_center;
}

MPI_Bcast(&arguments, 1, Arguments_TYPE, 0, MPI_COMM_WORLD);

// retrieve data
if(rank!=0){
    size = arguments.size;
    scale = arguments.scale;
    k_value = arguments.k_value;
    x_center = arguments.x_center;
    y_center = arguments.y_center;
}

// allcate buffer
std::vector<int> calculationResult;

int piecesNum = size*size/processSize;
int remainder = size*size%processSize;
int startPosi = rank*piecesNum+std::min(rank, remainder);
int endPosi = (rank+1)*piecesNum+std::min(rank+1,remainder)-1;
int taskSize = endPosi-startPosi+1;

// prepare for calculation task size
int startI = startPosi/size;
int i = startI;
int j = startPosi%size;
int count = 0;

double cx = static_cast<double>(size) / 2 + x_center;
double cy = static_cast<double>(size) / 2 + y_center;
double zoom_factor = static_cast<double>(size) / 4 * scale;
for (; i < size; ++i) {
    if(i!=startI){
        j=0;
    }
    for (; j < size; ++j) {
        // avoid extra calculation
        if(count>=taskSize){

```

```

        // break outside for loop
        i = size;
        break;
    }
    count++;
    double x = (static_cast<double>(i) - cx) / zoom_factor;
    double y = (static_cast<double>(j) - cy) / zoom_factor;
    std::complex<double> z{0, 0};
    std::complex<double> c{x, y};
    int k = 0;
    do {
        z = z * z + c;
        k++;
    } while (norm(z) < 2.0 && k < k_value);
    calculationResult.push_back(k);
}

// gather calculation result
// first gather the sizes of each subtask
std::vector<int> taskSizeList;
if(rank==0){
    taskSizeList.resize(processSize);
}

MPI_Gather(&taskSize, 1, MPI_INT, taskSizeList.data(), 1, MPI_INT, 0,
MPI_COMM_WORLD);

std::vector<int> strips;
// rank 0 calculate strips of received data packages
if(rank==0){
    strips.push_back(0);
    for(int i=1;i<processSize;i++){
        strips.push_back(strips[i-1]+taskSizeList[i-1]);
    }
}

// rank 0 gather calculated result
MPI_Gatherv(calculationResult.data(), taskSize, MPI_INT, buffer.buffer.data(),
taskSizeList.data(), strips.data(), MPI_INT, 0, MPI_COMM_WORLD);
}

static constexpr float MARGIN = 4.0f;
static constexpr float BASE_SPACING = 2000.0f;

```

```

static constexpr size_t SHOW_THRESHOLD = 500000000ULL;

int main(int argc, char **argv) {
    int rank;
    int processSize;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &processSize);

    // initialize MPI struct
    const int nitems=5;
    int blocklengths[5] = {1,1,1,2,2};
    MPI_Datatype types[5] = {MPI_INT, MPI_INT, MPI_INT, MPI_DOUBLE, MPI_DOUBLE};
    MPI_Aint offsets[5];

    offsets[0] = offsetof(Arguments, size);
    offsets[1] = offsetof(Arguments, scale);
    offsets[2] = offsetof(Arguments, k_value);
    offsets[3] = offsetof(Arguments, x_center);
    offsets[4] = offsetof(Arguments, y_center);

    MPI_Type_create_struct(nitems, blocklengths, offsets, types, &Arguments_TYPE);
    MPI_Type_commit(&Arguments_TYPE);

    // rank 0 plot calculation result
    if (0 == rank) {
        graphic::GraphicContext context{"Assignment 2"};
        Square canvas(100);
        size_t duration = 0;
        size_t pixels = 0;
        context.run([&](graphic::GraphicContext *context [[maybe_unused]],
SDL_Window *) {
            {
                auto io = ImGui::GetIO();
                ImGui::SetNextWindowPos(ImVec2(0.0f, 0.0f));
                ImGui::SetNextWindowSize(io.DisplaySize);
                ImGui::Begin("Assignment 2", nullptr,
                    ImGuiWindowFlags_NoMove
                    | ImGuiWindowFlags_NoCollapse
                    | ImGuiWindowFlags_NoTitleBar
                    | ImGuiWindowFlags_NoResize);
                ImDrawList *draw_list = ImGui::GetWindowDrawList();
                ImGui::Text("Application average %.3f ms/frame (%.1f FPS)", 1000.0f /
ImGui::GetIO().Framerate,

```

```

        ImGui::GetIO().Framerate);
static int center_x = 0;
static int center_y = 0;
static int size = 800;
static int scale = 1;
static ImVec4 col = ImVec4(1.0f, 1.0f, 0.4f, 1.0f);
static int k_value = 100;
ImGui::DragInt("Center X", &center_x, 1, -4 * size, 4 * size, "%d");
ImGui::DragInt("Center Y", &center_y, 1, -4 * size, 4 * size, "%d");
ImGui::DragInt("Fineness", &size, 10, 100, 1000, "%d");
ImGui::DragInt("Scale", &scale, 1, 1, 100, "%.01f");
ImGui::DragInt("K", &k_value, 1, 100, 1000, "%d");
ImGui::ColorEdit4("Color", &col.x);
{
    using namespace std::chrono;
    auto spacing = BASE_SPACING / static_cast<float>(size);
    auto radius = spacing / 2;
    const ImVec2 p = ImGui::GetCursorScreenPos();
    const ImU32 col32 = ImColor(col);
    float x = p.x + MARGIN, y = p.y + MARGIN;
    canvas.resize(size);
    auto begin = high_resolution_clock::now();
    calculate(canvas, size, rank, processSize, scale, center_x,
center_y, k_value);
    auto end = high_resolution_clock::now();
    pixels += size;
    duration += duration_cast<nanoseconds>(end - begin).count();
    if (duration > SHOW_THRESHOLD) {
        std::cout << pixels << " pixels in last " << duration << "
nanoseconds\n";
        auto speed = static_cast<double>(pixels) /
static_cast<double>(duration) * 1e9;
        std::cout << "speed: " << speed << " pixels per second" <<
std::endl;
        pixels = 0;
        duration = 0;
    }
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            if (canvas[{i, j}] == k_value) {
                draw_list->AddCircleFilled(ImVec2(x, y), radius,
col32);
            }
            x += spacing;

```

```

        }
        y += spacing;
        x = p.x + MARGIN;
    }
}
ImGui::End();
});
}
else{
    // parameter placeholder
    Square canvas(0);
    while(true){
        // calculate in while loop to interact with rank0
        calculate(canvas, 0, rank, processSize, 0, 0, 0, 0);
    }
}

// if plotting windows closed, rank 0 would then stop all other processes
if(rank==0){
    std::cout << "aborting all processes" << std::endl;
    MPI_Abort(MPI_COMM_WORLD, 0);
}

MPI_Type_free(&Arguments_TYPE);
MPI_Finalize();
return 0;
}

```

Pthread

```

#include <chrono>
#include <iostream>
#include <graphic/graphic.hpp>
#include <imgui_impl_sdl.h>
#include <vector>
#include <complex>
#include <mpi.h>
#include <cstring>

struct Square {
    std::vector<int> buffer;
    size_t length;
}

```

```

explicit Square(size_t length) : buffer(length), length(length * length) {}

void resize(size_t new_length) {
    buffer.assign(new_length * new_length, false);
    length = new_length;
}

auto& operator[](std::pair<size_t, size_t> pos) {
    return buffer[pos.second * length + pos.first];
}
};

// struct to hold parameters
struct Arguments {
    int* buffer;
    int size;
    int scale;
    double x_center;
    double y_center;
    int k_value;
    int taskNum;
    int threadSize;
};

void *subTask(void *arg_ptr) {
    // retrieve information from passed parameters
    auto arguments = static_cast<Arguments *>(arg_ptr);
    int piecesNum = arguments->size*arguments->size/arguments->threadSize;
    int remainder = arguments->size*arguments->size%arguments->threadSize;
    int startPosi = arguments->taskNum*piecesNum+std::min(arguments->taskNum,
remainder);
    int endPosi =
(arguments->taskNum+1)*piecesNum+std::min(arguments->taskNum+1,remainder)-1;
    int taskSize = endPosi-startPosi+1;

    // prepare for calculation task size
    int startI = startPosi/arguments->size;
    int i = startI;
    int j = startPosi%arguments->size;
    int count = 0;

    double cx = static_cast<double>(arguments->size) / 2 + arguments->x_center;
    double cy = static_cast<double>(arguments->size) / 2 + arguments->y_center;

```

```

    double zoom_factor = static_cast<double>(arguments->size) / 4 *
arguments->scale;
    for (; i < arguments->size; ++i) {
        if(i!=startI){
            j=0;
        }
        for (; j < arguments->size; ++j) {
            // if finish task then return
            if(count>=taskSize){
                return nullptr;
            }
            count++;
            double x = (static_cast<double>(i) - cx) / zoom_factor;
            double y = (static_cast<double>(j) - cy) / zoom_factor;
            std::complex<double> z{0, 0};
            std::complex<double> c{x, y};
            int k = 0;
            do {
                z = z * z + c;
                k++;
            } while (norm(z) < 2.0 && k < arguments->k_value);
            arguments->buffer[i*arguments->size+j] = k;
        }
    }

    return nullptr;
}

void calculate(Square &buffer, int size, int scale, double x_center, double
y_center, int k_value, int threadSize) {
    // fork threads
    std::vector<pthread_t> threads(threadSize);

    // initialize each thread with task information
    for(int i=0;i<threadSize;i++){
        pthread_create(&threads[i], nullptr, subTask, new Arguments{
            .buffer = buffer.buffer.data(),
            .size = size,
            .scale = scale,
            .x_center = x_center,
            .y_center = y_center,
            .k_value = k_value,
            .taskNum = i,
            .threadSize = threadSize

```



```

    });
}

// wait until all the threads finish
for (auto & i : threads) {
    pthread_join(i, nullptr);
}
}

static constexpr float MARGIN = 4.0f;
static constexpr float BASE_SPACING = 2000.0f;
static constexpr size_t SHOW_THRESHOLD = 500000000ULL;

int main(int argc, char **argv) {
    // use 4 threadsd as default
    int threadNum = 4;

    // fetch thread size from command line
    if(argc>=2){
        threadNum = atoi(argv[1]);
    }

    // avoid invalid input
    if(threadNum==0){
        threadNum = 4;
    }

    graphic::GraphicContext context{"Assignment 2"};
    Square canvas(100);
    size_t duration = 0;
    size_t pixels = 0;
    context.run([&](graphic::GraphicContext *context [[maybe_unused]], SDL_Window *)
{
    {
        auto io = ImGui::GetIO();
        ImGui::SetNextWindowPos(ImVec2(0.0f, 0.0f));
        ImGui::SetNextWindowSize(io.DisplaySize);
        ImGui::Begin("Assignment 2", nullptr,
                    ImGuiWindowFlags_NoMove
                    | ImGuiWindowFlags_NoCollapse
                    | ImGuiWindowFlags_NoTitleBar
                    | ImGuiWindowFlags_NoResize);
        ImDrawList *draw_list = ImGui::GetWindowDrawList();

```

```

        ImGui::Text("Application average %.3f ms/frame (%.1f FPS)", 1000.0f /
ImGui::GetIO().Framerate,
        ImGui::GetIO().Framerate);
    static int center_x = 0;
    static int center_y = 0;
    static int size = 800;
    static int scale = 1;
    static ImVec4 col = ImVec4(1.0f, 1.0f, 0.4f, 1.0f);
    static int k_value = 100;
    ImGui::DragInt("Center X", &center_x, 1, -4 * size, 4 * size, "%d");
    ImGui::DragInt("Center Y", &center_y, 1, -4 * size, 4 * size, "%d");
    ImGui::DragInt("Fineness", &size, 10, 100, 1000, "%d");
    ImGui::DragInt("Scale", &scale, 1, 1, 100, "%.01f");
    ImGui::DragInt("K", &k_value, 1, 100, 1000, "%d");
    ImGui::ColorEdit4("Color", &col.x);
    {
        using namespace std::chrono;
        auto spacing = BASE_SPACING / static_cast<float>(size);
        auto radius = spacing / 2;
        const ImVec2 p = ImGui::GetCursorScreenPos();
        const ImU32 col32 = ImColor(col);
        float x = p.x + MARGIN, y = p.y + MARGIN;
        canvas.resize(size);
        auto begin = high_resolution_clock::now();
        // calculate points for the graph with updated parameters
        calculate(canvas, size, scale, center_x, center_y, k_value,
threadNum);
        auto end = high_resolution_clock::now();
        pixels += size;
        duration += duration_cast<nanoseconds>(end - begin).count();
        if (duration > SHOW_THRESHOLD) {
            std::cout << pixels << " pixels in last " << duration << "
nanoseconds\n";
            auto speed = static_cast<double>(pixels) /
static_cast<double>(duration) * 1e9;
            std::cout << "speed: " << speed << " pixels per second" <<
std::endl;
            pixels = 0;
            duration = 0;
        }
        for (int i = 0; i < size; ++i) {
            for (int j = 0; j < size; ++j) {
                if (canvas[{i, j}] == k_value) {
                    draw_list->AddCircleFilled(ImVec2(x, y), radius, col32);

```

```

        }
        x += spacing;
    }
    y += spacing;
    x = p.x + MARGIN;
}
}
ImGui::End();
}
});

return 0;
}

```

MPI benchmark

```

#include <chrono>
#include <iostream>
#include <graphic/graphic.hpp>
#include <imgui_impl_sdl.h>
#include <vector>
#include <complex>
#include <mpi.h>
#include <cstring>

struct Square {
    std::vector<int> buffer;
    size_t length;

    explicit Square(size_t length) : buffer(length*length), length(length) {}

    void resize(size_t new_length) {
        buffer.assign(new_length * new_length, false);
        length = new_length;
    }

    auto& operator[](std::pair<size_t, size_t> pos) {
        return buffer[pos.second * length + pos.first];
    }
};

// define struct to pass information
typedef struct Arguments_S {

```

```

        int size;
        int scale;
        int k_value;
        double x_center;
        double y_center;
    } Arguments;

static MPI_Datatype Arguments_TYPE;

void calculate(Square &buffer, int size, int rank, int processSize, int scale,
double x_center, double y_center, int k_value) {
    // rank 0 broadcast parameters
    Arguments arguments;

    if(rank==0){
        arguments.size = size;
        arguments.scale = scale;
        arguments.k_value = k_value;
        arguments.x_center = x_center;
        arguments.y_center = y_center;
    }

    MPI_Bcast(&arguments, 1, Arguments_TYPE, 0, MPI_COMM_WORLD);

    // retrieve data
    if(rank!=0){
        size = arguments.size;
        scale = arguments.scale;
        k_value = arguments.k_value;
        x_center = arguments.x_center;
        y_center = arguments.y_center;
    }

    // allcate buffer
    std::vector<int> calculationResult;

    int piecesNum = size*size/processSize;
    int remainder = size*size%processSize;
    int startPosi = rank*piecesNum+std::min(rank, remainder);
    int endPosi = (rank+1)*piecesNum+std::min(rank+1,remainder)-1;
    int taskSize = endPosi-startPosi+1;

    // prepare for calculation task size
    int startI = startPosi/size;

```

```

int i = startI;
int j = startPosi%size;
int count = 0;

double cx = static_cast<double>(size) / 2 + x_center;
double cy = static_cast<double>(size) / 2 + y_center;
double zoom_factor = static_cast<double>(size) / 4 * scale;
for (; i < size; ++i) {
    if(i!=startI){
        j=0;
    }
    for (; j < size; ++j) {
        // avoid extra calculation
        if(count>=taskSize){
            // break outside for loop
            i = size;
            break;
        }
        count++;
        double x = (static_cast<double>(i) - cx) / zoom_factor;
        double y = (static_cast<double>(j) - cy) / zoom_factor;
        std::complex<double> z{0, 0};
        std::complex<double> c{x, y};
        int k = 0;
        do {
            z = z * z + c;
            k++;
        } while (norm(z) < 2.0 && k < k_value);
        calculationResult.push_back(k);
    }
}

// gather calculation result
// first gather the sizes of ech subtask
std::vector<int> taskSizeList;
if(rank==0){
    taskSizeList.resize(processSize);
}

MPI_Gather(&taskSize, 1, MPI_INT, taskSizeList.data(), 1, MPI_INT, 0,
MPI_COMM_WORLD);

std::vector<int> strips;
// rank 0 calculate strips of received data packages

```

```

    if(rank==0){
        strips.push_back(0);
        for(int i=1;i<processSize;i++){
            strips.push_back(strips[i-1]+taskSizeList[i-1]);
        }
    }

    MPI_Gatherv(calculationResult.data(), taskSize, MPI_INT, buffer.buffer.data(),
taskSizeList.data(), strips.data(), MPI_INT, 0, MPI_COMM_WORLD);
}

int main(int argc, char **argv) {
    int rank;
    int processSize;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &processSize);

    // initialize MPI struct
    const int nitems=5;
    int          blocklengths[5] = {1,1,1,2,2};
    MPI_Datatype types[5] = {MPI_INT, MPI_INT, MPI_INT, MPI_DOUBLE, MPI_DOUBLE};
    MPI_Aint      offsets[5];

    offsets[0] = offsetof(Arguments, size);
    offsets[1] = offsetof(Arguments, scale);
    offsets[2] = offsetof(Arguments, k_value);
    offsets[3] = offsetof(Arguments, x_center);
    offsets[4] = offsetof(Arguments, y_center);

    MPI_Type_create_struct(nitems, blocklengths, offsets, types, &Arguments_TYPE);
    MPI_Type_commit(&Arguments_TYPE);

    // holding the test size
    int testSize[5] = {1000, 2000, 4000, 8000, 16000};

    for(int i=0;i<5;i++){
        if (0 == rank) {
            Square canvas(0);
            size_t duration = 0;
            size_t pixels = 0;
            static int center_x = 0;
            static int center_y = 0;
            int size = testSize[i];

```

```

        static int scale = 1;
        static ImVec4 col = ImVec4(1.0f, 1.0f, 0.4f, 1.0f);
        static int k_value = 100;
        {
            using namespace std::chrono;
            canvas.resize(size);
            // rank 0 start timer
            auto begin = high_resolution_clock::now();
            calculate(canvas, size, rank, processSize, scale, center_x, center_y,
k_value);

            auto end = high_resolution_clock::now();
            pixels += size;
            duration += duration_cast<nanoseconds>(end - begin).count();
            // rank 0 report result for one iteration
            std::cout << "plot size: " << size <<
                " process size: " << processSize << " time spent: " << duration << "
ns" << std::endl;
        }
    }
    else{
        Square canvas(0);
        calculate(canvas, 0, rank, processSize, 0, 0, 0, 0);
    }
}

MPI_Type_free(&Arguments_TYPE);
MPI_Finalize();
return 0;
}

```

Pthread benchmark

```

#include <chrono>
#include <iostream>
#include <graphic/graphic.hpp>
#include <imgui_impl_sdl.h>
#include <vector>
#include <complex>
#include <mpi.h>
#include <cstring>

```

```

struct Square {
    std::vector<int> buffer;
    size_t length;

    explicit Square(size_t length) : buffer(length), length(length * length) {}

    void resize(size_t new_length) {
        buffer.assign(new_length * new_length, false);
        length = new_length;
    }

    auto& operator[](std::pair<size_t, size_t> pos) {
        return buffer[pos.second * length + pos.first];
    }
};

// struct to hold parameters
struct Arguments {
    int* buffer;
    int size;
    int scale;
    double x_center;
    double y_center;
    int k_value;
    int taskNum;
    int threadSize;
};

void *subTask(void *arg_ptr) {
    // retrieve information from passed parameters
    auto arguments = static_cast<Arguments*>(arg_ptr);
    int piecesNum = arguments->size*arguments->size/arguments->threadSize;
    int remainder = arguments->size*arguments->size%arguments->threadSize;
    int startPosi = arguments->taskNum*piecesNum+std::min(arguments->taskNum,
remainder);
    int endPosi =
(arguments->taskNum+1)*piecesNum+std::min(arguments->taskNum+1,remainder)-1;
    int taskSize = endPosi-startPosi+1;

    // prepare for calculation task size
    int startI = startPosi/arguments->size;
    int i = startI;
    int j = startPosi%arguments->size;
    int count = 0;

```



```

double cx = static_cast<double>(arguments->size) / 2 + arguments->x_center;
double cy = static_cast<double>(arguments->size) / 2 + arguments->y_center;
double zoom_factor = static_cast<double>(arguments->size) / 4 *
arguments->scale;
for (; i < arguments->size; ++i) {
    if(i!=startI){
        j=0;
    }
    for (; j < arguments->size; ++j) {
        // if finish task then return
        if(count>=taskSize){
            return nullptr;
        }
        count++;
        double x = (static_cast<double>(i) - cx) / zoom_factor;
        double y = (static_cast<double>(j) - cy) / zoom_factor;
        std::complex<double> z{0, 0};
        std::complex<double> c{x, y};
        int k = 0;
        do {
            z = z * z + c;
            k++;
        } while (norm(z) < 2.0 && k < arguments->k_value);
        arguments->buffer[i*arguments->size+j] = k;
    }
}

return nullptr;
}

```

```

void calculate(Square &buffer, int size, int scale, double x_center, double
y_center, int k_value, int threadSize) {
    // fork threads
    std::vector<pthread_t> threads(threadSize);

    // initialize each thread with task information
    for(int i=0;i<threadSize;i++){
        pthread_create(&threads[i], nullptr, subTask, new Arguments{
            .buffer = buffer.buffer.data(),
            .size = size,
            .scale = scale,
            .x_center = x_center,
            .y_center = y_center,

```

```

        .k_value = k_value,
        .taskNum = i,
        .threadSize = threadSize
    });
}

// wait until all the threads finish
for (auto & i : threads) {
    pthread_join(i, nullptr);
}
}

int main(int argc, char **argv) {
    // use 1 threadsd as default
    int threadSize = 1;

    // fetch thread size from command line
    if(argc>=2){
        threadSize = atoi(argv[1]);
    }

    // avoid invalid input
    if(threadSize==0){
        threadSize = 1;
    }

    std::cout << "testing with " << threadSize << " thread(s)" << std::endl;

    int testSize[5] = {1000, 2000, 4000, 8000, 16000};

    Square canvas(100);
    size_t duration = 0;
    static int center_x = 0;
    static int center_y = 0;
    static int size = 800;
    static int scale = 1;
    static int k_value = 100;
    {
        using namespace std::chrono;
        for(int i=0;i<5;i++){
            duration = 0;
            size = testSize[i];
            canvas.resize(size);
            auto begin = high_resolution_clock::now();

```

```
        calculate(canvas, size, scale, center_x, center_y, k_value, threadSize);
        auto end = high_resolution_clock::now();
        duration += duration_cast<nanoseconds>(end - begin).count();
        std::cout << "plot size: " << testSize[i] <<
            " thread size: " << threadSize << " time spent: " << duration << " ns"
<< std::endl;
    }
}

return 0;
}
```