# Parallelizing Mandelbrot set computation
# TDDD56

Martin Söderén
marso329, 9009291098

15 november 2015

# 1 Introduction

This lab aims to implement two parallell algorithms that calculate the Mandelbrot set. For the complete definition of the Mandelbrot set see the lab description. Each complex number in the predefined set is calculated by iterating over it. If the computation reaches a maximal number of iterations the complex number is belived to be in the Mandelbrot set. If during the computation the series is detected to diverge then it is not belived to be in the set.

Each computation for the complex number is independent so these can be calculated in parallel. This is called embarassingly parallel algorithm. For each complex number $z$ a point $p$ in a image is defined where the complex numbers real part is the position on the x-axis and the imaginary part is the position on the y-axis.

# 2 Method

For the first part of the lab we are to implement a naive algorithm that just splits the image into equal part. This is not the best algorithm since calculating if a certain pixel is in the Mandelbrot requires the maximum number of iteration. Some threads will get more work than others if they get parts of the image that contains alot of pixels in the set. You can see the code for this implementation in appendix A and the resuling image can be seen in figure 1.

The second part of the lab is to the implement a better algorithm that has some kinde if loadbalancing. This was achieved by splitting the image in a 10x10 grid and having a counter protected by a mutex lock that counts up when a worker thread starts working on a part of the grid. From this number each thread can calculate which pixels should be calculated. The code for this implementation can be seen in appendix B and the resulting picture can be seen in figure 2

The reason for the funky colors are that each thread is assign a random color before it starts computing to easier see which thread has done what work. Since these colors are assign randomly there is a chance that two threads can be assign the same color. This can be fixed be checking which colors have been assign before but seemed unneccesary.

# 3 Result

As can be seen in figure 3 the running time decreases for both algorithms when more threads are spawned ,especially for the loadbalancing algorithm. The interesting things is that the naive algorithm is slower with three threads than with two threads. This can be explained by the fact that most of the pixels in the mandelbrot set is in the middle of the image and by using three threads the seconds threads started will get most of the work and the seconds two threads will get very little work. This can be seen in figure 4 where for three threads thread number two does more work than the other two combined. Compared with two threads where they do almost the same amount of work.

In figure 5 you can see that the loadbalacing algorithm is doing it work and that the work is spread out even over all threads. Compare this to figure 4 where the distribution has the form of a normal distribution where the threads spawned in the middle are doing most of the work.

It needs to be said that the loadbalacing algorithm implemented here is not the optimal since it does not take cache line size in consideration. It can be assumed that the data for the pixels are stored in a rowmajor and the best thing would be to split the image up in blocks which size is equal to the cache line size and that the computing blocks take the form of rows in the image instead.

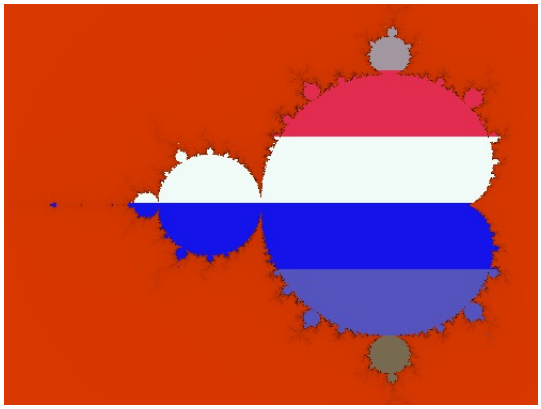

Figure 1: Naive implementation.

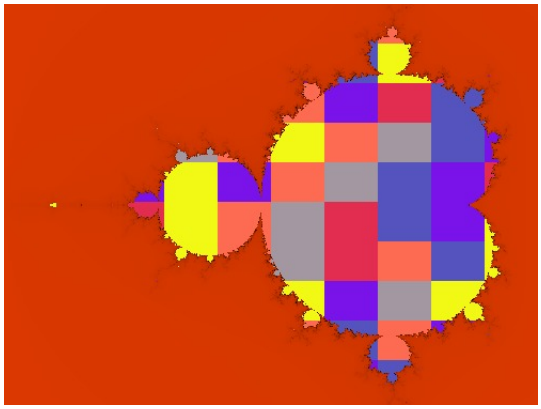

Figure 2: Loadbalanced implementation.

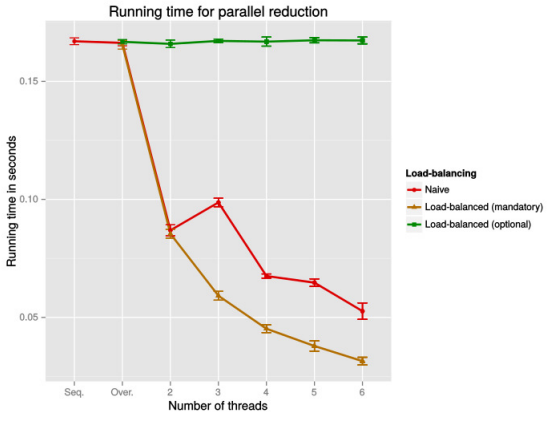


Figure 3: Running time for both implementations.

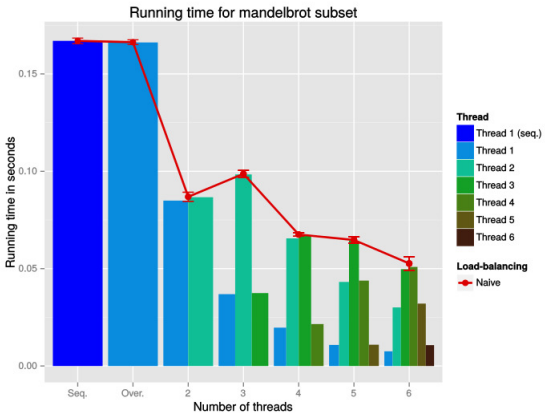


Figure 4: Naive work distribution.

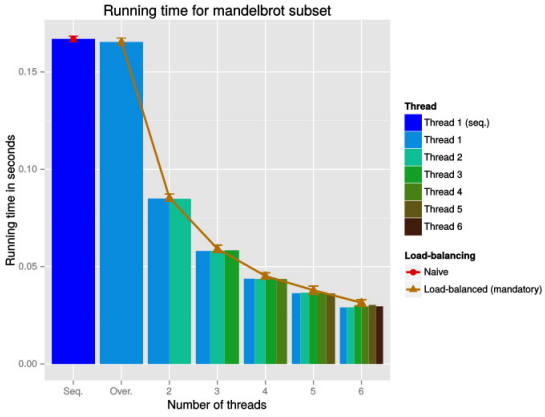


Figure 5: Loadbalanced work distribution.

# A Naive implementation

```
parameters->mandelbrot_color.red=randint(255);
parameters->mandelbrot_color.green=randint(255);
parameters->mandelbrot_color.blue=randint(255);
parameters->begin_h = (args->id)*((parameters->height)/(NB_THREADS));
if (args->id!=(NB_THREADS-1)){
        parameters->end_h = ((args->id)+1)*((parameters->height)/(NB_THREADS));
}
else{
        parameters->end_h=parameters->height;
}
        parameters->begin_w = 0;
        parameters->end_w = parameters->width;
        compute_chunk(parameters);
```

# B Loadbalanced implementation

```
parameters->mandelbrot_color.red=randint(255);
parameters->mandelbrot_color.green=randint(255);
parameters->mandelbrot_color.blue=randint(255);
while(1){
        int myblocknumber;
        pthread_mutex_lock(&block_lock);
        myblocknumber = block_counter;
        if (myblocknumber!=100){
                block_counter+=1;
                pthread_mutex_unlock(&block_lock);
        }
        else{
                pthread_mutex_unlock(&block_lock);
                break;
        }
        parameters->begin_w=(myblocknumber%10)*parameters->width/10;
        if (myblocknumber%10!=9){
                parameters->end_w=(myblocknumber%10+1)*parameters->width/10;
        }
        else{
                parameters->end_w=parameters->width;
        }
        parameters->begin_h=(myblocknumber/10)*(parameters->height/10);
        if(myblocknumber/10+1!=10){
                parameters->end_h=(myblocknumber/10+1)*(parameters->height/10);
        }
        else{
                parameters->end_h=parameters->height;
        }
        compute_chunk(parameters);
}
```