High Performance Programming Project
Simulating the Mandelbrot Set

Oscar Jacobson*

July 2022

1 Introduction/Background

When coding, the lower bound of time taken for a task to complete is always going to be a predetermined attribute of the the task itself. Depending on the amount, and quality, of resources dedicated to that task then determines what level of performance is achieved compared to the theoretical maximum. Dedicated resources refers to both computational resources and time put in to produce high quality solutions. When seeking peak performance, computational resources are important. But the quality of written code, its structure and how it's interpreted by the machine is also of great importance.

Using multiple processors to execute code in parallel, multithreading, is one of the best ways to increase performance of code. This does however require the task to be paralleisable and for the threads to be able to use the multiple processors effectively. When using mulltithreading the code is separated into two parts. A serial part, which is run by one single thread, and a parallel part that can be divided into smaller tasks split up between threads.

When optimising a code there are a lot of factors to consider. Firstly, in what way is the code to be optimised? Secondly, how? This project is mainly going to consider the optimisation of time taken for a predetermined task to complete as well as discussing the efficiency of that optimisation.

Amdahl's law gives the theoretical maximum speedup given what portion of an algorithm is actually parallelizable.[1] In practice there are always a serial part to any algorithm, meaning there is always going to be a lower bound of possible speedup. Amdahl's law also describes the diminishing returns of adding resources to already parallelized sections. The speedup should therefore in theory continuously increase with added resources but also have an exponential decrease in efficiency. This is detrimental to computer scientists as resources always come at a cost of either money or time wasted. What then, is the most effective way of executing a algorithm given the allocated computational resources with regards to efficiency?[1]

*Uppsala University, High Performance Programming course, Teacher: Jarmo Rantakokko

2 Problem description

The Mandelbrot set is the set of complex numbers defined by the function:

$$f(z) = z^2 + c \tag{1}$$

Where $f(z) \in C, Z_0 = 0$ and repeating the function never diverges to infinity[1].

The Mandelbrot set is therefore a set of fractal values, derived from different starting points c, in the complex plane where the continuous iterating of function 1 is bounded and never diverges to infinity.[2] These points make up a set with infinite complexity and is by definition not computable in a finite amount of time.

Assuming a large amount of computational power has been used and being able to store all the complexity in a hypothetical figure, zooming into any of the edges between blue and white of figure 1 below would reveal more and more smaller, self-similar, patterns in infinity.

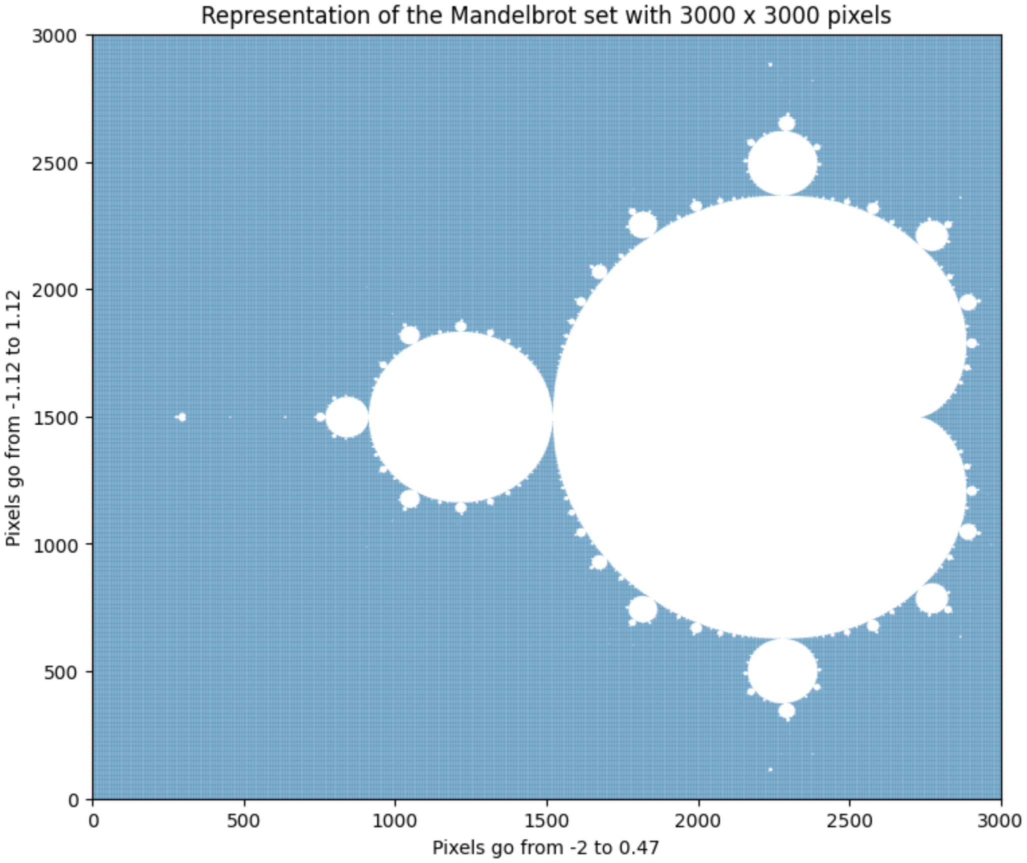


Figure 1: Proof of accuracy, Mandelbrot simulation using matplotlib and a csv file generated from OptParallelMandel.c. (V3)

[1] Example of implementation in Equation appendix A

Because of the simplicity of function 1, generating a sufficiently complex figure of the Mandelbrot set is achievable using a moderate amount of computing power as demonstrated above. The problem setting of computing the Mandelbrot set is predetermined by the desired complexity of the outcome and is therefore malleable. Using this variable problem setting, together with time taken and computational resources[2] used, one can determine the performance of the algorithms and code used to produce the figure.

The following project will use the computation of the Mandelbrot set with set levels of complexity to determine the efficiency and speed of code executed serially versus in parallel. To examine the benefits and possible drawbacks of parallel implementation.

The time complexity of computing the Mandelbrot is hard to define exactly. Assuming computations are done locally though, the paper *The Computational Complexity of Some Julia Sets*[3] argues that a Julia set, which is closely related to the Mandelbrot set and uses the same base-equation 1, can be computed locally in time with complexity O(k^2M(k)).

Where k^2 is the total number of pixels and M(k) is the function executed at each pixel. The paper uses the logic of a Touring machine time to determine a binary problem at each pixel. In this project the function M(k) on each pixel is a loop that either goes on for MAX_ITER iterations or is exited earlier. As this function is hard to determine we will assume the function to be homogeneous throughout the set which then will cause time complexity of M(k) to be a constant and give the final time complexity C*k^2 locally, or approximately O(n) where n = k^2 is the amount of pixels computed.

3 Solution method

The Mandelbrot set is relatively simple to compute. To determine if a point is in the Mandelbrot set Algorithm 1 is used. This checks if the repeating of equation 1 has a euclidean distance that would end up outside of a set border $2 * 2 = 4$. If the algorithm manages to reach $MAX\_ITER = 1000$ iterations without ending up outside the border the point is assumed to belong to the Mandelbrot set. If the loop at any point finds itself outside the border the loop exits early and the point is assumed to not belong to the set. The pseudo code for the described algorithm can be seen in Algorithm 1.

The computation of the Mandelbrot set is brute force. The algorithm would therefore benefit greatly from optimisations. Firstly, the for loop is highly parallelizable as the individual pixels can be evaluated independently. Secondly, the simple algorithm used to determine if each point is in the Mandelbrot set is not optimal for computations.

Multiplications have time complexity $O(\mathrm{n}^2)$ whereas standard addition and subtraction have complexity between $O(\mathrm{n})$ and $O(\log(\mathrm{n}))$.[4] Multiplications are therefore very unattractive when doing large computations. In Algorithm 2 the inner while-loop is optimised with algebraic simplifications of complex number multiplications. This simplification reduces the amount of multiplications from five to three as well as reducing the amount of simple computations being made.[5]. The reduction of multiplications and reduction of simple computations were made in two steps and can be seen in code appendix 3 as C functions optmandelbrotV1 and optmandel-

[2] Computational resources refers to the hardware and/or software of the computer used.

Algorithm 1 Serial algorithm to determine if every point is in the Mandelbrot set

```
for Every pixel within border do
    Determine $[x_0, y_0]$ position for pixel
    Set $x = 0$
    Set $y = 0$
    while $x^2 + y^2 \leq x_{Max}^2 + y_{Max}^2$ AND Iteration <= MAX iterations do
        xtemp $= x * x - y * y - x_0$
        $y = 2 * x * y + y_0$
        $x =$ xtemp
        Iteraration++
    end while
    if Iteration == MAX iteration then
        Starting point is in the Mandelbrot set!
    else
        Starting point is not in the Mandelbrot set!
    end if
end for
```

brotV2.

Algorithm 2 Optimized serial algorithm to determine if every point is in the Mandelbrot set

```
for Every pixel within border do
    Determine $[x_0, y_0]$ position for pixel
    Set $x = 0$
    Set $y = 0$
    Set $x_2 = 0$
    Set $y_2 = 0$
    while $x_2 + y_2 \leq 4$ AND Iteration <= MAX iterations do
        $y = (x + x) * y + y_0$
        $x = x_2 - y_2 + x_0$
        $x_2 = x * x$
        $y_2 = y * y$
        Iteraration++
    end while
    if Iteration == MAX iteration then
        Starting point is in the Mandelbrot set!
    else
        Starting point is not in the Mandelbrot set!
    end if
end for
```

The algorithms was all implemented in C code. C is a low level programming language giving good control over how the code is interpreted by the computer and is able to give very precise instructions. Compared to a higher level programming language like python which was used to produce most of the graphs in this project C allows for more flexibility but also requires more complex and precise writing of code. When parallelizing the code the OMP package for C was used. It is written to be accessible to most C programmers and easy to use. Including the $< omp.h >$ header file allows for the Mandelbrot calculations to be parallelized very easily. In this case only two lines of code was used to run the outer for loops in parallel. The library then handles most of the parallelization by itself.

By specifying the parallel section using *#pragma omp parallel* and number of threads used with *num_threads()* (On the same line). One is able to run a for loop in parallel by preceding the line of the loop with *#pragma omp for*. This also allows for the decision of either *static* or *dynamic* scheduling. *Static* is the preset option which splits the for loop evenly between all the available threads, with the first ones receiving the first part iterations and second thread receiving the second part of iterations and so on. This is simple and works very well for most purposes. The dynamic scheduling makes adjustments to the amount of tasks any given thread is given depending on the time taken of each individual task. This can be great when having varying amounts of work being done by different iterations of the for loop.

The final and arguably most significant benefit of C code is the use of compiler optimizations. The compiler is a program that interprets the written C code and makes an executable file in machine code with the same instructions. The compiler optimizations flags are built in functions in the GCC compiler that changes the way the machine code is written. The optimization flags makes the usage of optimization easy and intuitive to use. There are four main optimization flags in the GCC compiler which incorporate around 100 different optimizations into singular flags. The optimization flags used in this project are: None (No optimization flag at all), -O, -O2, -O3 and -Ofast. The -O.. flags tell the compiler to optimize code with different levels of compilation time and code size whereas -Ofast incorporates all of the optimizations of the -O flags and adds some speed optimizations to mathematical operations. -Ofast can be a dangerous flag to use as it can reduce accuracy for the sake of speed in some situations. See Appendix 9 for a quick and easy overview.

4 Analysis/Experiments

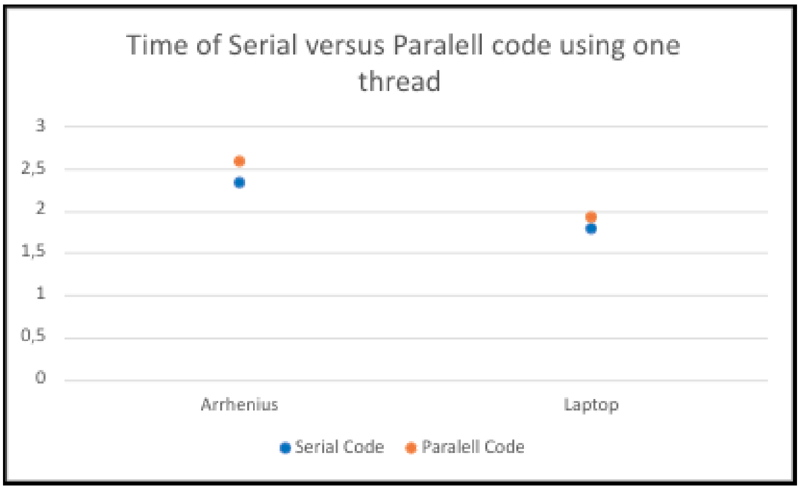


Figure 2: Serial vs Parallel code using only one thread

Experiments was carried out on two different computer systems. Firstly on a personal Acer laptop with 4 cores and limited processing power. Secondly on a large computer provided by Uppsala Universitet called Arrhenius which is a massive system with a large amount of cores. By remotely accessing this computer code could be executed using up to 1000 threads. Times were recurded using the *OMP Get Time* functions.

Figure 2 shows the result of running the serial code and then the parallelized versions using only one thread on both computers. There is a clear difference and by using OMP we introduce a small negative offset in the execution time. When using parallelized code with one single thread we see a slow down of 5 to 10 percent. This is of course a net positive trade off when using more than one thread, but the overhead of OMP is noticeable.

To examine the computational complexity of the algorithms used to compute the Mandelbrot set time was plotted against pixels evaluated. Figure 3 shows the exponential growth of execution time versus increasing complexity. Here the horizontal axis represents the amount of pixels evaluated in one dimension. As pixels evaluated are N^2 the problem is two-dimensional. Figure 4 shows the complexity over n^2 using a logarithmic scale on both axes. A logarithmic regression was used to determine the actual exponential growth of time versus complexity and was fitted to about 1.96 for n. This means that the actual computational complexity is somewhat smaller than the previously assumed $O(k)$ and is instead approximately $O(k^{1.96})$. The $O(k)$ complexity assumed that the function at each pixel was homogeneous. Revisiting algorithms 1 and 2 this is obviously not true. Every loop with starting points outside of the Mandelbrot set will exit early because of the escape algorithm and will not require as many computations to be made.

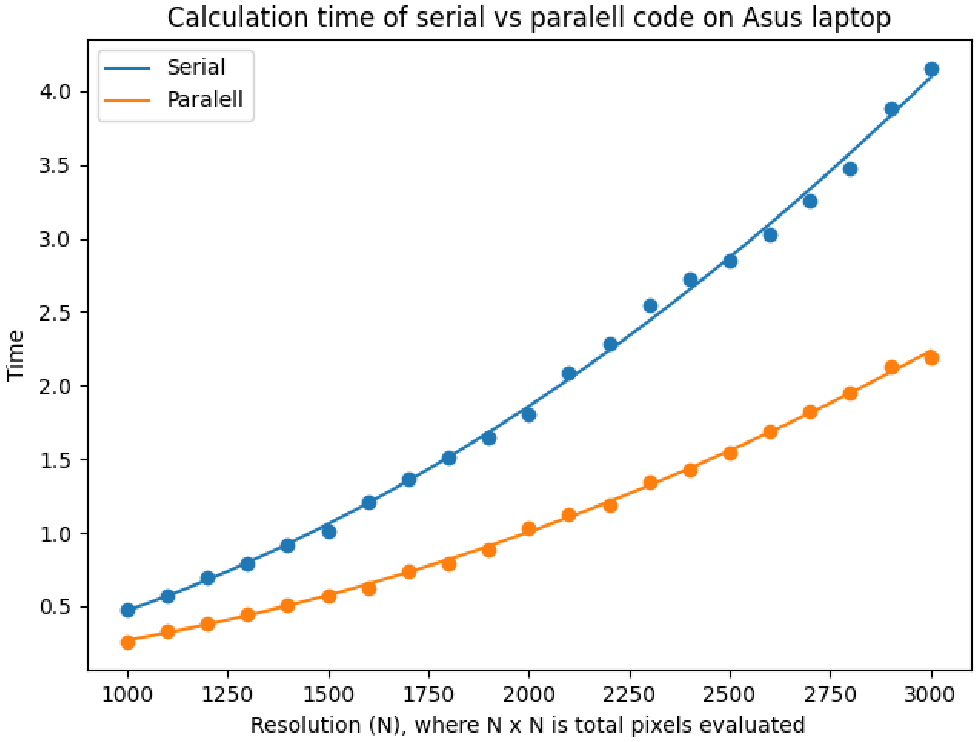


Figure 3: Comparison of serial and parallel code on Acer laptop when calculating the Mandelbrot with varying amount of pixels

This results in the computational complexity to be slightly less than quadratic when increasing complexity (Pixels) quadratically.

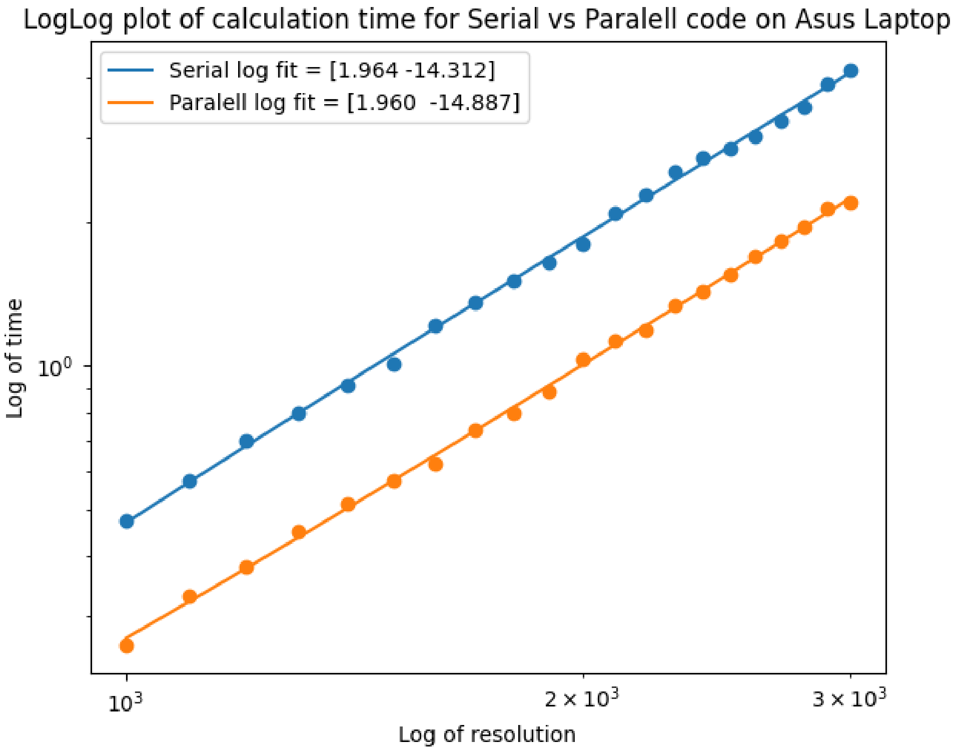
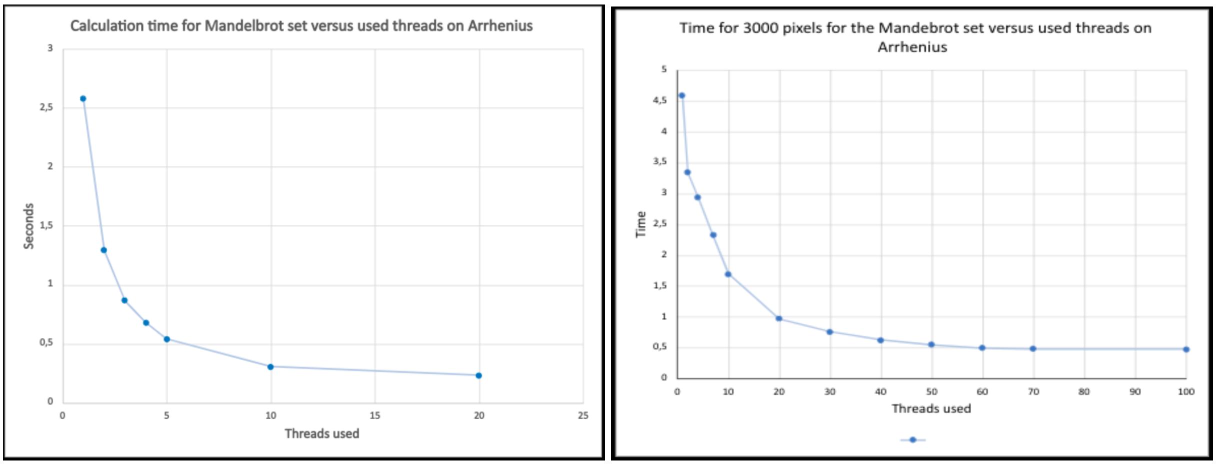


Figure 4: LogLog plot of time for serial and parallel code on Acer laptop when calculating the Mandelbrot with varying amount of pixels

To examine the impact of running the tasks in parallel the time for computing two set complexities was plotted against the number of used threads in figure 5. The times for computing 2000 and 3000 pixels of the Mandelbrot set respectively show similar results. The calculation times seems to drop off exponentially and converge on some lower bound. The efficiency of adding more threads to run tasks in parallel can be seen in figure 6 where the proportional speedup is plotted against the number of threads used.

Figure 6 clearly displays the diminishing returns of efficiency when adding more resources to a parallelized problem. As the speedup is plotted against added resources the gradient of the plot can be seen as efficiency of threads added. The gradient is steep at first and then drops off significantly for some amount of threads used. There is a slight difference in the two tails of the graphs 6a and 6b. The graph for the larger problem seems to still benefit slightly from added threads whereas the smaller problem seems to have hit a upper bound of speedup. They both seem to have ended up at 11 times speedup though and using Amdahl's law we can assume that the upper bound of speedup is the same for both problems as the amount of serial code should be identical.

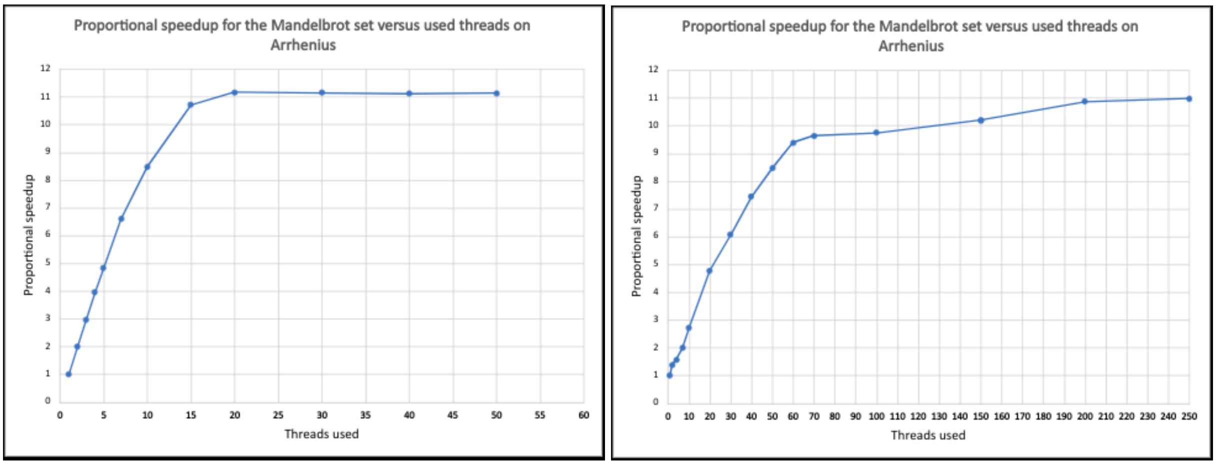
Figure 5: Time for computing the Mandelbrot set on Arrhenius.it.uu.se with varying amount of threads.



(a) Calculating 2000 pixels of the Mandelbrot set on Arrhenius with varying amount of threads

(b) Calculating 3000 pixels of the Mandelbrot set on Arrhenius with varying amount of threads

Figure 6: Proportional speedup of the Mandelbrot set calculations with varying amount of threads on Arrhenius.



(a) Proportional speedup of the Mandelbrot set calculations of 2000 pixels.

(b) Proportional speedup of the Mandelbrot set calculations of 3000 pixels.

Disregarding the fact that using hundreds of threads causes significant inefficiencies the graphs show a significant reduction in calculation time. Using only two lines of code the same calculations could be made more than ten times faster using minimal time spent coding and a relatively low usage of computational resources.

Figure 7 and 8 investigates the effect of the different compiler optimization flags previously mentionaed by plotting the average runtime for the different versions of code using the different optimization flags. The effect of adding any optimization flag is visible in the first figure 7 where the code clearly struggles to compute the problem in a reasonable amount of time. The simpler *Serial* code, which was the first iteration of code written is also outperforming the later versions by a large margin. The GCC compiler therefore seems to handle the simple code significantly better than the more complex later versions.

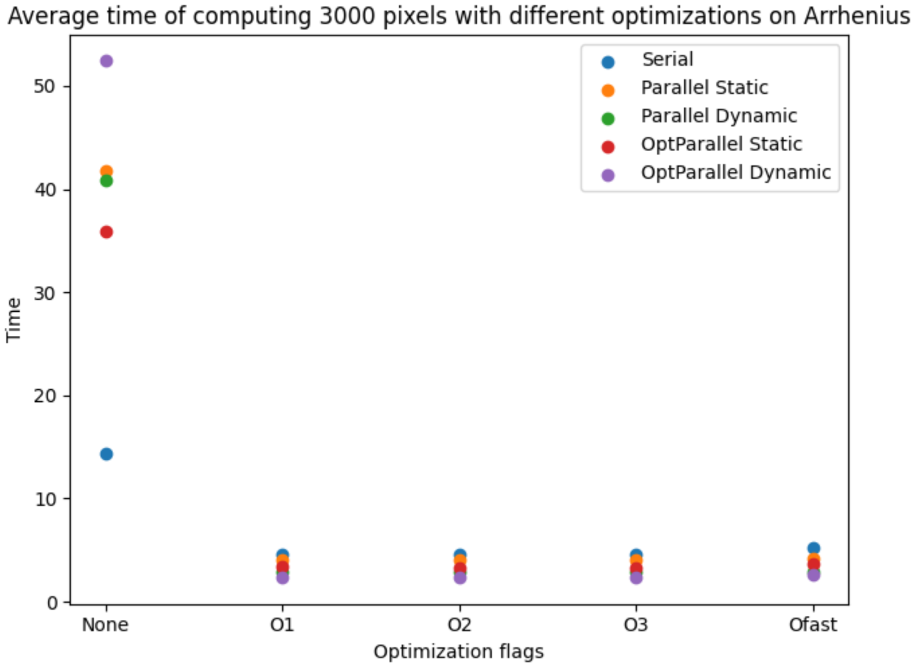


Figure 7: Average time of computing 3000 pixels of the Mandelbrot set using 2 threads (for parallel code only) and using different optimization flags (including no optimization)

Figure 8 shows the average runtime for different compiler optimizations. The thre -O flags seem to perform identically on every version of code while the -Ofast flag seems to slow all versions down slightly. As the -Ofast flag includes all the -O flag optimizations this slowdown has to come from any of the additionally added optimizations. It is hard to say exactly from where but it is possible that the loss of accuracy in float numbers from *-ffast-math* can cause the edge cases to iterate for longer before exiting.

Additionally the *Serial* version of the code who performed the best at no optimizations is now performing the worst at full optimization. This is obviously due to the fact that the other versions of code are executed using two threads. Even more interesting is the fact that the fully optimized version who performed the worst with no optimization is now performing the best. This would also be a case of the compiler having a hard time of the added complexity of the later, optimized versions of code.

Important to notice in figure 8 is also the order of the *static* and *dynamic* scheduling. The *dynamic* option for the unoptimized parallel code have a significantly lower runtime than the optimized *static* version. The simple choice of *dynamic* scheduling is therefore more potent in this case than the serial optimizations performed to reduce computations.

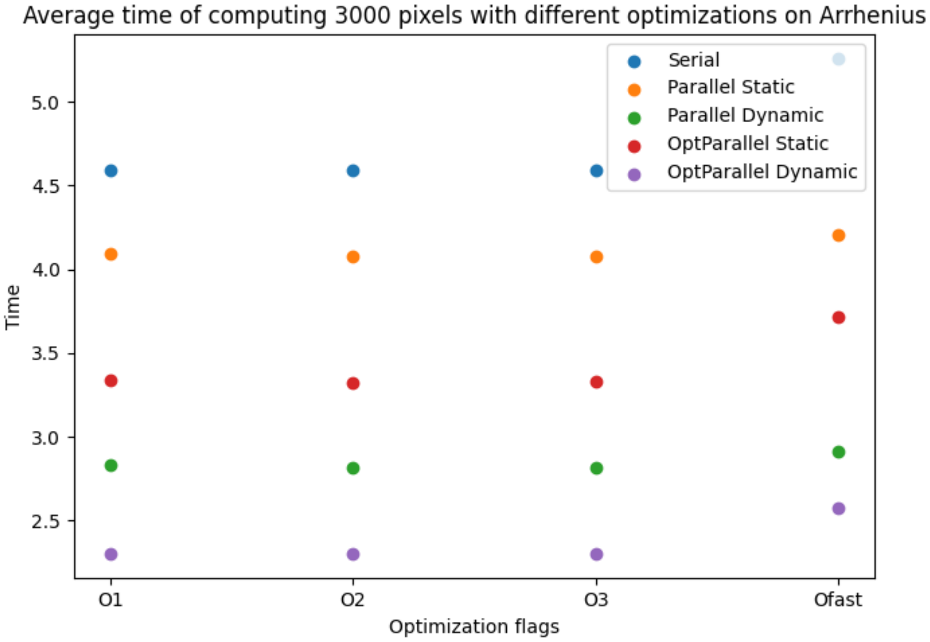


Figure 8: Average time of computing 3000 pixels of the Mandelbrot set using 2 threads (for parallel code only) and using different optimization flags (excluding no optimization)

Code version	Threads used	Time	Speedup	Ratio (Approximately)
Serial	1	4.586940	-	-
Parallel	500	0.538923	8.5	0.85 : 50
OptParallel	50	0.467398	10	1 : 5
OptParallel	500	0.433938	10.5	1 : 50

Table 1: Table over final values and respective speedup for set amount of used threads when computing 3000 pixels on Arrhenius. (Using dynamic scheduling and the -O3 flag)

Table 1 displays the final values for some important cutoff numbers of threads. It also shows the speedup of the optimization and the given ratio of efficiency with respect to used resources. Importantly the *OptParallel* optimization manages to reach the highest level of speedup of about 10.5. Reassuring that the optimizations made has given a significant increase compared to the original *Parallel* code who only managed a speedup of about 8.5 for the same amount of threads. The *OptParallel* code also manages to produce a 10 times speedup when using only 50 threads. This is only a slight loss of speedup for 10 times less resources used.

5 Conclusions

When optimizing code to solve a computationally expensive problem like Computing the Mandelbrot set, C is a great tool to use. The programming language supports all kinds of options for optimization and gives high flexibility because of its low level.

Using knowledge about the complexity of different kinds of operations, the task itself could be optimized and simplified to require less computations for the same result. The tools available in C then allowed for a big improvement in execution time by firstly parallelizing the highly parallelizable problem as well as using the GCC compiler optimizations to achieve the maximum amount of speedup using very little extra work in the form of writing code. OMP allowed for 10 times the speedup in two lines of code and some tweaking in the compilation optimizations.

The code is, in my opinion, very well optimized. Using an optimal, serial inner while-loop, parallelizing the outer for-loop as well as using the dynamic scheduling. On top of that, Amdahl's law was used to visualize the trade off between efficiency and added resources. Making sure that no computations would be ran with excessive waste in the event of future large scale implementations. (Not likely!)

As a final note: During my course in High Performance Programming I have been introduced to pthreads which is a library similar to OMP to make parallelization of code possible. The difference of the two libraries is that pthreads is significantly more low-level and allows for almost full controll over the creation and tasking of individual threads. Where OMP could create threads and parallelize tasks in two lines pthreads allow for more precise instructions. The drawback of OMP comes with the previously mentioned overhead start up time. The trade off between OMP and pthreads is therefore simplicity versus controll and less overhead. The negatives of pthreads is the difficoulty of writing low-level parallelized code. A drawback which is significantly reduced with coding experience.

Using OMP this is as far as I could optimize the code with limited previous coding experience in C. There is however no doubt in my mind that this could be optimized further with regards to execution time using pthreads and additional coding work!

References

[1] Wikipedia contributors. Amdahl's law — Wikipedia, The Free Encyclopedia; 2022. [Online; accessed accessed 3-September-2022]. https://en.wikipedia.org/w/index.php?title=Amdahl%27s_law&oldid=1107025481.

[2] Wikipedia contributors. Mandelbrot set — Wikipedia, The Free Encyclopedia; 2022. [Online; accessed accessed 3-September-2022]. https://en.wikipedia.org/w/index.php?title=Mandelbrot_set&oldid=1108139023.

[3] Robert Rettinger, Klaus Weihrauch. The computational complexity of some Julia sets; July 2002. [Online accesed 29th December 2022]. https://www.sciencedirect.com/science/article/pii/S1571066104803864.

[4] Wikipedia contributors. Computational complexity of mathematical operations — Wikipedia, The Free Encyclopedia; 2022. [Online; accessed 11-September-2022]. https://en.wikipedia.org/w/index.php?title=Computational_complexity_of_mathematical_operations&oldid=1108780149.

[5] Wikipedia contributors. Plotting algorithms for the Mandelbrot set — Wikipedia, The Free Encyclopedia; 2022. [Online; accessed 3-September-2022]. https://en.wikipedia.org/w/index.php?title=Plotting_algorithms_for_the_Mandelbrot_set&oldid=1101564767.

[6] Juan Carlos Ponce Campuzano. The Mandelbrot set – Complex Analysis; 2022. [Online; accessed 29th December 2022]. https://complex-analysis.com/content/mandelbrot_set.html.

[7] Links to resources used when writing code; 2022. [Online; accessed 11-September-2022].
omp_get_time() https://www.ibm.com/docs/en/xl-fortran-aix/15.1.0?topic=openmp-omp-get-wtime
,
OpenMP clauses, https://docs.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-clauses?view=msvc-170
,
Dynamic scheduling, http://www.inf.ufsc.br/~bosco.sobral/ensino/ine5645/OpenMP_Dynamic_Scheduling.pdf
,
Copying results to a .txt file, https://social.msdn.microsoft.com/Forums/vstudio/en-US/6a12570b-df26-4736-9f32-60c8b0ccc281/how-to-write-into-a-csv-file-in-c?forum=vcgeneral
,
Plotting the mandelbrot set, https://stackoverflow.com/questions/68041664/plot-a-2d-binary-matrix-as-a-line-in-matplotlib-using-plot
,
Regression in python, https://rowannicholls.github.io/python/curve_fitting/exponential.html.

A Equation Appendix

Example of equation 1 with starting point $c = 1$:

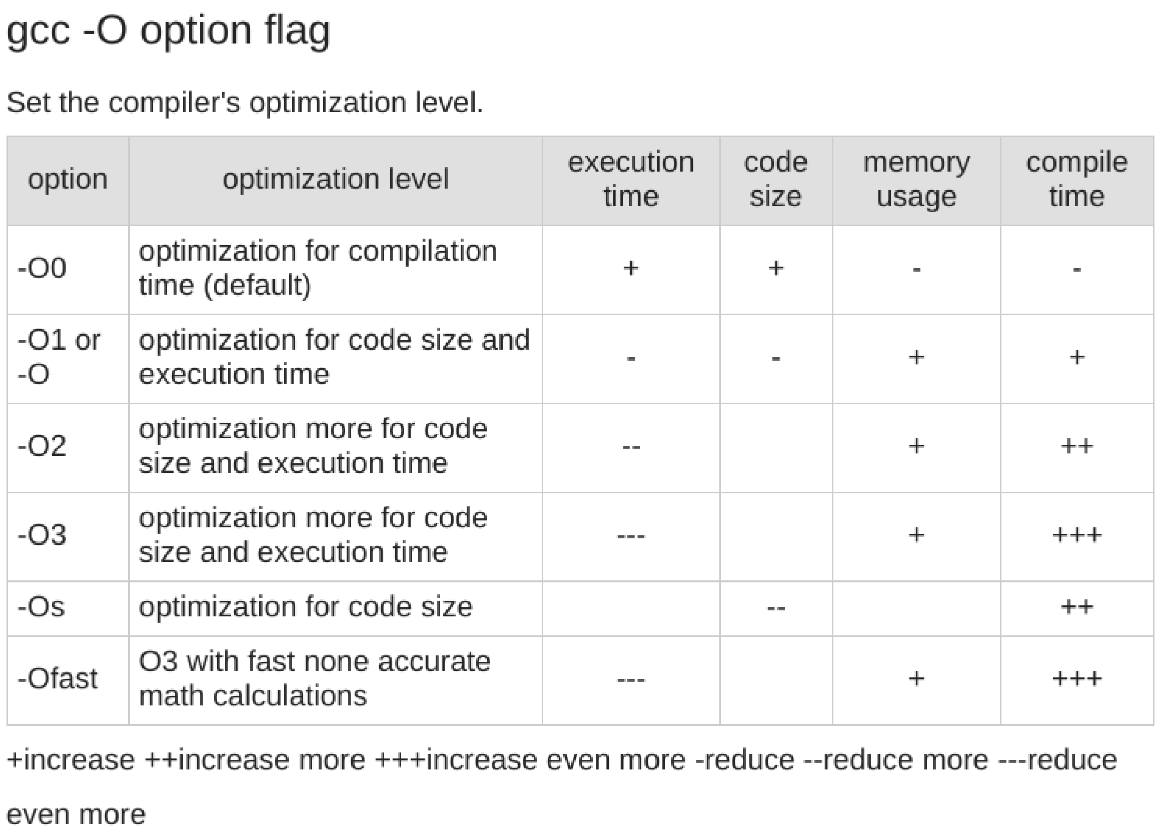
$0^2 + 1 = 1$
$1^2 + 1 = 2$
$2^2 + 1 = 5$
$5^2 + 1 = 26$
Goes on to infinity and is therefore not a part of the Mandelbrot set![6]

Example of equation 1 with starting point $c = -1$:

$0^2 - 1 = -1$
$-1^2 - 1 = 0$
$0^2 - 1 = -1$
$-1^2 - 1 = 0$
Never leaves the loop $-1 \Rightarrow 0 \Rightarrow -1$ and is therefore a part of the Mandelbrot set![6]

For visual representation [6]

B Figure Appendix



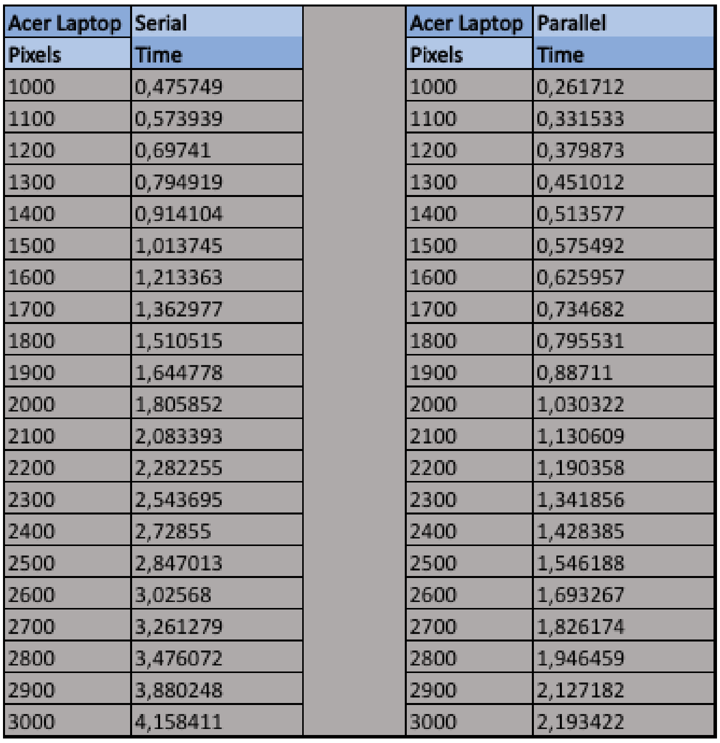
gcc -O option flag

Set the compiler's optimization level.

option	optimization level	execution time	code size	memory usage	compile time
-O0	optimization for compilation time (default)	+	+	-	-
-O1 or -O	optimization for code size and execution time	-	-	+	+
-O2	optimization more for code size and execution time	--		+	++
-O3	optimization more for code size and execution time	---		+	+++
-Os	optimization for code size		--		++
-Ofast	O3 with fast none accurate math calculations	---		+	+++

+increase ++increase more +++increase even more -reduce --reduce more ---reduce even more

Figure 9: Table of simple optimization flags for the GCC compiler. Accessed online from: *Rapid Tables*, `https://www.rapidtables.com/code/linux/gcc/gcc-o.html` [12th September 2022]



Acer Laptop	Serial		Acer Laptop	Parallel
Pixels	Time		Pixels	Time
1000	0,475749		1000	0,261712
1100	0,573939		1100	0,331533
1200	0,69741		1200	0,379873
1300	0,794919		1300	0,451012
1400	0,914104		1400	0,513577
1500	1,013745		1500	0,575492
1600	1,213363		1600	0,625957
1700	1,362977		1700	0,734682
1800	1,510515		1800	0,795531
1900	1,644778		1900	0,88711
2000	1,805852		2000	1,030322
2100	2,083393		2100	1,130609
2200	2,282255		2200	1,190358
2300	2,543695		2300	1,341856
2400	2,72855		2400	1,428385
2500	2,847013		2500	1,546188
2600	3,02568		2600	1,693267
2700	3,261279		2700	1,826174
2800	3,476072		2800	1,946459
2900	3,880248		2900	2,127182
3000	4,158411		3000	2,193422

Figure 10: Table of execution times for different amount of pixels computed in serial and in parallel on Acer laptop.

C Code Appendix

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>

#define max_iter 1000
#define resolution 1000
#define border 2.0

int* grid;



void check(int* grid, int N){
    //Only a visual check, prints ones and zeros to terminal
    for (int i = 0; i < N; i++){
        for (int j = 0; j < N; j++){
            printf("%d", grid[j*N + i]);
        }
        printf("\n");
    }
    return;
}




// Ref: https://complex-analysis.com/content/mandelbrot_set.html
void mandelbrot(){
    int i, j, iter = 0;
    long double x, y, xx = 0, yy = 0, temp;
    long double pixel = (border+border)/resolution;  //(Positive border + Negative border) over amount of /
      splits

    for (i = 0; i < resolution; i++){
        x = -border + i * pixel;
        for (j = 0; j < resolution; j++){   //Makes a square border, not necessary because of while loop
            y = -border + j * pixel;
            iter = 0;
            xx = 0;
            yy = 0;
            while (xx*xx + yy*yy <= border*border && iter < max_iter){
                temp = xx*xx - yy*yy + x;
                yy = 2*xx*yy + y;
                xx = temp;
                iter++;
            }
            if (iter == max_iter){
                grid[i*resolution + j] = 0;
            }
            else{
                //Can maybe use some faster operator? Bitshift? This operation probably doesnt take up that /
      much time
                grid[i*resolution + j] = 1;
            }
        }
    }
}
```

```
int main(int argc, char *argv[]) {
    if(argc != 2) {
        printf("Please give 1 argument(s): Print = 1\n");
        return -1;
    }
    int prt;
    long double start = 0, stop = 0;
    double pixelsz = (border+border)/resolution;
    printf("Pixelsize = %f, Pixels = %d\n", pixelsz, resolution);

    prt = atoi(argv[1]);
    grid = (int *)malloc(resolution * resolution * sizeof(int));
    if (resolution > 1){
        start = omp_get_wtime();
        mandelbrot();
        stop = omp_get_wtime();
    }
    else if (resolution <= 0){
        printf("Set resolution cannot be negative!?\n");
    }
    else{
        printf("No need to test one pixel...\n");
    }
    if (prt){
        check(grid, resolution);
        printf("To test, make sure resolution will match about one row of 1:s printed in your terminal\n");
    }

    printf("Calculations took %Lf seconds (In Wtime)\n", stop-start);
    free(grid);
    return 0;
}
```

Listing 1: Serial implementation.[7]

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>

//What is a "necessary" large number, when do you loose accuracy, probably can be linked back to pixel size
#define max_iter 1000
#define resolution 3000
#define border 2.0
#define border2 border*border //Used so the loop doesnt have to calculate at every iteration
int* grid;


void check(int* grid, int N){
    //Only a visual check, prints ones and zeros to terminal
    for (int i = 0; i < N; i++){
        for (int j = 0; j < N; j++){
            printf(”%d”, grid[j*N + i]);
        }
        printf(”\n”);
    }
    return;
}

//https://slaystudy.com/c−program−to−read−write−linked−list−to−a−file/
void write(char filename[], int* array, int N){
    remove(”result.csv”);
    FILE* file;
    file = fopen (filename, ”w+”);
    if (file == NULL)
    {
        fprintf(stderr, ”\nCouldn’t Open File’\n”);
        exit (1);
    }

    for (int i = 0; i < N; i++){
        // fprintf(file, ”[”);
        for (int j = 0; j < N; j++){
            if (i == N−1 && j == N−1){
                fprintf(file, ”%d”, array[i*N + j]);
            }
            else{
                fprintf(file, ”%d,”, array[i*N + j]);
            }
        }
    }

    printf(”List stored in the file successfully\n”);

    fclose(file);
}

//Ref for serial code: https://complex−analysis.com/content/mandelbrot_set.html
void mandelbrot(int Thr){
    int i, j, iter = 0;
    long double x, y, xx = 0, yy = 0, temp;
    long double pixel = (border+border)/resolution;  //(Positive border + Negative border) / amount of splits

    //Very simple OMP implementation, only added 2 lines.
    #pragma omp parallel private(i, j, iter) shared(grid) num_threads(Thr)
    {
        #pragma omp for schedule(dynamic)            //Dynamic because of load balancing, threads executed in/the edges will be alot faster than ones that do 1000 iterations every loop.
        for (i = 0; i < resolution; i++){
            x = −border + i * pixel;
            for (j = 0; j < resolution; j++){   //Makes a square border, not necessary because of while loop
                y = −border + j * pixel;
                iter = 0;
                xx = 0;
                yy = 0;
                //Any pixel outside of the circular radius of length 2 will be terminated after the first loop.
                while (xx*xx + yy*yy <= border2 && iter < max_iter){
                    temp = xx*xx − yy*yy + x;
                    yy = 2*xx*yy + y;
                    xx = temp;
                    iter++;
                }
                if (iter == max_iter){
                    grid[i*resolution + j] = 0;
                }
                else{
                    grid[i*resolution + j] = 1;
                }
            }
        }
    }
}






int main(int argc, char *argv[]) {
    if(argc != 4) {
        printf("Please give 3 argument(s): Print = 1 or 0, Thr = Number of threads, Write = 1 or 0\n");
        return -1;
    }
    int prt, Thr, wrt;
    long double start = 0, stop = 0;
    double pixelsz = (border+border)/resolution;
    printf("Pixelsize = %f, Pixels = %d\n", pixelsz, resolution);

    prt = atoi(argv[1]);
    Thr = atoi(argv[2]);
    wrt = atoi(argv[3]);

    grid = (int *)malloc(resolution * resolution * sizeof(int));

    if (resolution > 1){
        start = omp_get_wtime();
        mandelbrot(Thr);
        stop = omp_get_wtime();
    }
    else if (resolution <= 0){
        printf("Set resolution cannot be negative!?\n");
    }
    else{
        printf("No need to test one pixel...\n");
    }
    if (prt){
        check(grid, resolution);
        printf("To test, make sure resolution will match about one row of 1:s printed in your terminal\n");
    }
    if (wrt){
        write("result.csv", grid, resolution);
    }
    printf("Calculations took %Lf seconds (In Wtime)\n", stop-start);
    free(grid);
    return 0;
}
```

Listing 2: Parallel implementation.[7]

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>

//What is a "necessary" large number, when do you loose accuracy, probably can be linked back to pixel size
#define max_iter 1000
#define resolution 3000
#define border 2.0
#define Yborder 1.12
#define Xborder 0.47
#define border2 border*border //Used so the loop doesnt have to calculate at every iteration
int* grid;


void check(int* grid, int N){
    //Only a visual check, prints ones and zeros to terminal
    for (int i = 0; i < N; i++){
        for (int j = 0; j < N; j++){
            printf("%d", grid[j*N + i]);
        }
        printf("\n");
    }
    return;
}

//https://slaystudy.com/c-program-to-read-write-linked-list-to-a-file/
void write(char filename[], int* array, int N){
    remove("result.csv");
    FILE* file;
    file = fopen (filename, "w+");
    if (file == NULL)
    {
        fprintf(stderr, "\nCouldn't Open File'\n");
        exit (1);
    }

    for (int i = 0; i < N; i++){
        // fprintf(file, "[");
        for (int j = 0; j < N; j++){
            if (i == N-1 && j == N-1){
                fprintf(file, "%d", array[i*N + j]);
            }
            else{
                fprintf(file, "%d,", array[i*N + j]);
            }
        }
    }

    printf("List stored in the file successfully\n");

    fclose(file);
}

//Ref for serial code: https://complex-analysis.com/content/mandelbrot_set.html
void mandelbrot(int Thr){
    int i, j, iter = 0;
    long double x, y, xx = 0, yy = 0, temp;
    long double pixel = (border+border)/resolution;  //(Positive border + Negative border) / amount of splits

    //Very simple OMP implementation, only added 2 lines.
    #pragma omp parallel private(i, j, iter) shared(grid) num_threads(Thr)
    {
        #pragma omp for schedule(dynamic)            //Dynamic because of load balancing, threads executed in/ the edges will be alot faster than ones that do 1000 iterations every loop.
        for (i = 0; i < resolution; i++){
            x = -border + i * pixel;
            for (j = 0; j < resolution; j++){   //Makes a square border, not necessary because of while loop
                y = -border + j * pixel;
                iter = 0;
                xx = 0;
                yy = 0;
                //Any pixel outside of the circular radius of length 2 will be terminated after the first loop.
                while (xx*xx + yy*yy <= border2 && iter < max_iter){
                    temp = xx*xx - yy*yy + x;
                    yy = 2*xx*yy + y;
                    xx = temp;
                    iter++;
                }
                if (iter == max_iter){
                    grid[i*resolution + j] = 0;
                }
                else{
                    grid[i*resolution + j] = 1;
                }
            }
        }
    }
}


//Algebraic simplification 5 mult => 3 mult
void optmandelbrotV1(int Thr){
    int i, j, iter = 0;
    long double x0, y0, x, y, z, xx = 0, yy = 0;
    long double pixel = (border+border)/resolution;

    #pragma omp parallel private(i, j, iter) shared(grid) num_threads(Thr)
    {
        #pragma omp for schedule(dynamic)
        for (i = 0; i < resolution; i++){
            x0 = -border + i * pixel;
            for (j = 0; j < resolution; j++){
                y0 = -border + j * pixel;
                iter = 0;
                x = 0;
                y = 0;
                z = 0;
                xx = 0;
                yy = 0;
                //Opt in while loop
                while (xx + yy <= border2 && iter < max_iter){
                    x = xx - yy + x0;
                    y = z - xx - yy + y0;
                    xx = x*x;
                    yy = y*y;
                    z = (x+y)*(x+y);
                    iter++;
                }
                //Opt above
                if (iter == max_iter){
                    grid[i*resolution + j] = 0;
                }
                else{
                    grid[i*resolution + j] = 1;
                }
            }
        }
    }
}



//Algebraic simplification reduced computations, still 3 multiplications
void optmandelbrotV2(int Thr){
    int i, j, iter = 0;
    long double x0, y0, x, y, xx = 0, yy = 0;
    long double pixel = (border+border)/resolution;


    #pragma omp parallel private(i, j, iter) shared(grid) num_threads(Thr)
    {
        #pragma omp for schedule(static)
        for (i = 0; i < resolution; i++){
            x0 = -border + i * pixel;
            for (j = 0; j < resolution; j++){
                y0 = -border + j * pixel;
                iter = 0;
                x = 0;
                y = 0;
                xx = 0;
                yy = 0;
                //Opt in while loop
                while (xx + yy <= border2 && iter < max_iter){
                    y = (x + x)*y + y0;
                    x = xx - yy + x0;
                    xx = x*x;
                    yy = y*y;
                    iter++;
                }
                //Opt above
                if (iter == max_iter){
                    grid[i*resolution + j] = 0;
                }
                else{
                    grid[i*resolution + j] = 1;
                }
            }
        }
    }
}




//Algorithm simplification, reduced area of computation
//Will probably slow the execution time down as more pixels will go towards max_iter.
//On the other hand more complexity is gained (Gained complexity is positive in a visual sense)
//for the same amount of iterations and iterations made outside of this border are in almost
//all cases going to be unneccessary.
```

```
//This is therefore more of a quality optimisation rather than a computational optimization.
void optmandelbrotV3(int Thr){
    int i, j, iter = 0;
    long double x0, y0, x, y, xx = 0, yy = 0;
    long double Xpixel = (border+Xborder)/resolution;
    long double Ypixel = (Yborder+Yborder)/resolution;


    #pragma omp parallel private(i, j, iter) shared(grid) num_threads(Thr)
    {
        #pragma omp for schedule(dynamic)
        for (i = 0; i < resolution; i++){
            x0 = -border + i * Xpixel;
            for (j = 0; j < resolution; j++){
                y0 = -Yborder + j * Ypixel;
                iter = 0;
                x = 0;
                y = 0;
                xx = 0;
                yy = 0;
                while (xx + yy <= border2 && iter < max_iter){
                    y = (x + x)*y + y0;
                    x = xx - yy + x0;
                    xx = x*x;
                    yy = y*y;
                    iter++;
                }
                if (iter == max_iter){
                    grid[i*resolution + j] = 0;
                }
                else{
                    grid[i*resolution + j] = 1;
                }
            }
        }
    }
}





int main(int argc, char *argv[]) {
    if(argc != 4) {
        printf("Please give 3 argument(s): Print = 1 or 0, Thr = Number of threads, Write = 1 or 0\n");
        return -1;
    }
    int prt, Thr, wrt;
    long double start = 0, stop = 0;
    double pixelsz = (border+border)/resolution;
    printf("Pixelsize = %f, Pixels = %d\n", pixelsz, resolution);

    prt = atoi(argv[1]);
    Thr = atoi(argv[2]);
    wrt = atoi(argv[3]);

    grid = (int *)malloc(resolution * resolution * sizeof(int));

    if (resolution > 1){
        start = omp_get_wtime();
        optmandelbrotV2(Thr);
        stop = omp_get_wtime();
    }
    else if (resolution <= 0){
        printf(”Set resolution cannot be negative!?\n”);
    }
    else{
        printf(”No need to test one pixel...\n”);
    }
    if (prt){
        check(grid, resolution);
        printf(”To test, make sure resolution will match about one row of 1:s printed in your terminal\n”);
    }
    if (wrt){
        write(”result.csv”, grid, resolution);
    }
    printf(”Calculations took %Lf seconds (In Wtime)\n”, stop−start);
    free(grid);
    return 0;
}
```

Listing 3: Optimized parallel implementations.[7]

```
import matplotlib.pyplot as plt
import numpy as np
import math

filename = ”result.csv”



with open(filename) as file_name:
    array = np.loadtxt(file_name, delimiter=”,”)


N = len(array)
N2 = int(math.sqrt(N))
array2 = [[0 for i in range(N2)] for j in range(N2)]

for i in range(N2):
    j = 0
    for j in range(N2):
        array2[i][j] = int(array[i∗N2+j])

# print(array2)
coords = np.argwhere(array2)
# print(coords)
plt.xlim(−1,N2)
plt.xlabel(”Pixels go from −2 to 0.47”)
plt.ylim(−1,N2)
plt.ylabel(”Pixels go from −1.12 to 1.12”)
plt.title(”Representation of the Mandelbrot set with 3000 x 3000 pixels”)
plt.scatter(coords[:,0],coords[:,1], marker=”o”, s = 0.01, linewidths=0)
plt.show()
```

Listing 4: Python code to print the Mandelbrot Set figure.[7]

```
import matplotlib.pyplot as plt
import numpy as np
import math


# https://www.rapidtables.com/code/linux/gcc/gcc−o.html

# 3000 pixels on Arrhenius, 5 run average

# Squares are static, circles are dynamic, triangles are serial

# Colour code O−flags

# Size code threads used

# Black border optimized code

x = [”None”, ”O1”, ”O2”, ”O3”, ”Ofast”]
x2 = [”O1”, ”O2”, ”O3”, ”Ofast”]


firstN = np.average([14.405045,14.439531,14.405501])
first1 = np.average([4.580848,4.580092,4.599721,4.579987,4.601426])
first2 = np.average([4.578698,4.596516,4.590520,4.595324,4.595570])
first3 = np.average([4.579547,4.607347,4.588174,4.618052,4.585199])
firstF = np.average([5.255005, 5.261210, 5.254551, 5.255460, 5.254170])




Par2StaticN = np.average([32.746605,88.251384,35.528149,18.806589,33.282322])
Par2Static1 = np.average([4.075797,4.116952,4.076162,4.083337,4.098357])
Par2Static2 = np.average([4.090253,4.082025,4.075259,4.084499,4.069975])
Par2Static3 = np.average([4.070566,4.070891,4.068264,4.088786,4.085888])
Par2StaticF = np.average([4.202978,4.226665,4.204412,4.208907,4.197604])

Par2DynN = np.average([38.756843,29.882664,52.398546,18.744674,64.301041])
Par2Dyn1 = np.average([2.829258,2.827959,2.846629,2.834829,2.827751])
Par2Dyn2 = np.average([2.821151,2.815847,2.816474,2.818326,2.818084])
Par2Dyn3 = np.average([2.820088,2.816029,2.817033])
Par2DynF = np.average([2.913535,2.908084,2.912552,2.907866,2.909157])




Opt2StaticN = np.average([15.066247,53.878906,35.433578,36.996117,38.371362])
Opt2Static1 = np.average([3.325122,3.375224,3.337185,3.324473,3.323666])
Opt2Static2 = np.average([3.322618,3.324472,3.321764,3.327861,3.330071])
Opt2Static3 = np.average([3.322269,3.345679,3.332575,3.321705,3.328425])
Opt2StaticF = np.average([3.703461,3.704810,3.703531,3.704464,3.753005])

Opt2DynamicN = np.average([57.859485,59.471619,35.662638,58.891779,50.400638])
Opt2Dynamic1 = np.average([2.304138,2.303191,2.301097,2.306293,2.302371])
Opt2Dynamic2 = np.average([2.300893,2.308918,2.299493,2.301866,2.307504])
Opt2Dynamic3 = np.average([2.300082,2.300530,2.302222,2.300849,2.323820])
Opt2DynamicF = np.average([2.562179,2.626138,2.567246,2.565704,2.563030])

# plt.scatter(x,[firstN,first1,first2,first3,firstF], label=”Serial”)
# plt.scatter(x,[Par2StaticN,Par2Static1,Par2Static2,Par2Static3,Par2StaticF], label=”Parallel Static”)
# plt.scatter(x,[Par2DynN,Par2Dyn1,Par2Dyn2,Par2Dyn3,Par2DynF], label=”Parallel Dynamic”)
# plt.scatter(x,[Opt2StaticN,Opt2Static1,Opt2Static2,Opt2Static3,Opt2StaticF], label=”OptParallel Static”)
# plt.scatter(x,[Opt2DynamicN,Opt2Dynamic1,Opt2Dynamic2,Opt2Dynamic3,Opt2DynamicF], label=”/
       OptParallel Dynamic”)


plt.scatter(x2,[first1,first2,first3,firstF], label=”Serial”)
plt.scatter(x2,[Par2Static1,Par2Static2,Par2Static3,Par2StaticF], label=”Parallel Static”)
plt.scatter(x2,[Par2Dyn1,Par2Dyn2,Par2Dyn3,Par2DynF], label=”Parallel Dynamic”)
plt.scatter(x2,[Opt2Static1,Opt2Static2,Opt2Static3,Opt2StaticF], label=”OptParallel Static”)
plt.scatter(x2,[Opt2Dynamic1,Opt2Dynamic2,Opt2Dynamic3,Opt2DynamicF], label=”OptParallel Dynamic”)


plt.legend()
plt.title(”Average time of computing 3000 pixels with different optimizations on Arrhenius”)
plt.ylabel(”Time”)
plt.xlabel(”Optimization flags”)
plt.show()
```

Listing 5: Python code to print optimized results.[7][7]

```
import matplotlib.pyplot as plt
import numpy as np


y = /
       [0.475749,0.573939,0.69741,0.794919,0.914104,1.013745,1.213363,1.362977,1.510515,1.644778,1.805852,2.083393,2.282255,2.543

# y = /
       [0.475749,0.573939,0.69741,0.794919,0.914104,1.013745,1.213363,1.362977,1.510515,1.644778,1.805852,2.083393,2.282255,2.543


x = /
       [1000,1100,1200,1300,1400,1500,1600,1700,1800,1900,2000,2100,2200,2300,2400,2500,2600,2700,2800,2900,3000]/

# x = /
       [1000,1100,1200,1300,1400,1500,1600,1700,1800,1900,2000,2100,2200,2300,2400,2500,2600,2700,2800,2900,3000,5000,/
        10000]

yy = /
       [0.261712,0.331533,0.379873,0.451012,0.513577,0.575492,0.625957,0.734682,0.795531,0.88711,1.030322,1.130609,1.190358,1.341

# yy = /
       [0.261712,0.331533,0.379873,0.451012,0.513577,0.575492,0.625957,0.734682,0.795531,0.88711,1.030322,1.130609,1.190358,1.341
        5.979748,25.07875]

y = np.array(y)
ly = np.log(y)
yy = np.array(yy)
lyy = np.log(yy)
x = np.array(x)
lx = np.log(x)


fit = np.polyfit(x, y, 2)
fit2 = np.polyfit(x, yy, 2)

```

```
lfit = np.polyfit(np.log(x), np.log(y), 1)
lfit2 = np.polyfit(np.log(x), np.log(yy), 1)

print(f’Fit 1: {fit}’)
print(f’Fit 2: {fit2}’)

x_fitted = np.linspace(np.min(x), np.max(x), 100)
y_fitted = fit[2] + fit[1]∗x_fitted + fit[0]∗x_fitted∗x_fitted
y2_fitted = fit2[2] + fit2[1]∗x_fitted + fit2[0]∗x_fitted∗x_fitted
# y_fitted = fit[0]∗x_fitted + fit[1]
# print(x_fitted∗x_fitted)

plt.scatter(x,y)
plt.plot(x_fitted,y_fitted, label=”Serial”)
plt.plot(x_fitted,y2_fitted, label=”Paralell”)
plt.scatter(x,yy)
plt.ylabel(”Time”)
plt.xlabel(”Resolution (N), where N x N is total pixels evaluated”)
plt.title(”Calculation time of serial vs paralell code on Asus laptop”)
plt.legend()



plt.figure()
plt.scatter(x,y)
plt.plot(x_fitted,y_fitted, label=”Serial log fit = [1.964 −14.312]”)
plt.plot(x_fitted,y2_fitted, label=”Paralell log fit = [1.960  −14.887]”)
plt.scatter(x,yy)
plt.yscale(”log”)
plt.xscale(”log”)
plt.ylabel(”Log of time”)
plt.xlabel(”Log of resolution”)
plt.legend()
plt.title(”LogLog plot of calculation time for Serial vs Paralell code on Asus Laptop”)
print(f’Log fit 1: {lfit}’)
print(f’Log fit 2: {lfit2}’)

plt.show()
```

Listing 6: Python code to print results and the LogLog Regression of thread-increase.[7]

```
CC = gcc
LD = gcc
CFLAGS = −Wall −Werror −fopenmp
LDFLAGS = −fopenmp
EXECUTABLE = MandelbrotSerial MandelbrotParalell OptParallelMandel

all:(EXECUTABLE)MandelbrotSerial : SerialMandel.c(LD) (LDFLAGS)(CFLAGS) SerialMandel.c −o /
     MandelbrotSerial

MandelbrotParalell: ParalellMandel.c
  (LD)(LDFLAGS) /
     (CFLAGS)ParalellMandel.c − oMandelbrotParalellOptParalellMandel : OptParallelMandel.c(LD) /
     (LDFLAGS)(CFLAGS) OptParallelMandel.c −o OptMandelbrotParalell

clean:
```

```
rm -f
```

Listing 7: Makefile to compile the C code using GCC[7]