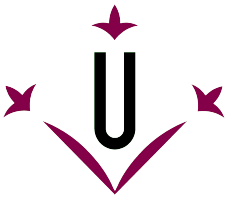
103084 - High Performance Computing

OMP Mandelbrot Fractal

Jordi Rafael Lazo Florensa

1 May 2023

Master's degree in Computer Engineering



Universitat de Lleida
Escola Politècnica Superior

1 Introduction

The file named *mandelbrot-seq.c* is a C program that generates an image of the Mandelbrot set. The Mandelbrot set is a complex mathematical set that exhibits a repeating pattern of fractals at different scales.

The program uses nested for-loops to iterate over each pixel in the image. For each pixel, it computes a corresponding complex number by mapping the pixel's coordinates to the complex plane. It then performs a series of iterations using the Mandelbrot formula to determine whether the complex number is part of the Mandelbrot set or not. If the number is part of the set, it is colored black, and if it is not part of the set, it is colored with varying shades of blue based on how many iterations were required to determine that it was not part of the set.

The program uses several variables to control the image generation, such as the width and height of the image, the zoom level, the position of the image in the complex plane, and the maximum number of iterations to perform for each pixel. It also uses the OpenMP library to parallelize the computation, which can improve the program's performance on multi-core systems.

Finally, the program outputs the image data in PPM format to standard output, along with some metadata such as the creator and dimensions of the image.

2 Scalability of the Program

The scalability of the original sequencial code refers to its ability to efficiently utilize increasing numbers of processing resources, such as additional CPU cores or threads, to improve its performance and generate the Mandelbrot set image faster.

The code is written using OpenMP, which is a standard API for parallel programming in shared-memory architectures. OpenMP allows for the automatic parallelization of the computation across multiple threads, which can take advantage of multiple CPU cores and improve performance.

The nested for-loops in the code that iterate over each pixel in the image are a natural fit for parallelization, as each pixel can be computed independently of the others. By adding the *#pragma omp parallel for* directive before the outer for loop, the computation can be automatically parallelized across multiple threads.

However, the scalability of the code is not perfect and may be limited by the overhead of parallelization and memory bandwidth limitations. For example, as the number of threads increases, the performance may begin to level off due to the overhead of synchronizing the threads and managing data dependencies. Additionally, the program may require a large amount of memory to store the image data, which may limit scalability on systems with limited memory bandwidth.

In general, the scalability of the code depends on several factors, such as the number of CPU cores, memory bandwidth, and the size of the image being generated. With the appropriate hardware and software configuration, the code can be scaled to take advantage of multiple cores and achieve significant performance improvements.

The lines of parallelized code are shown below:

```
32      for(y = 0; y < h; y++) {
33          int array[w];
34          #pragma omp parallel for private(x, pr, pi, newRe, newIm, oldRe, oldIm)
35          for(x = 0; x < w; x++) {
36              int i = 0;
```

Figure 1: Parallelization the second for loop

```
51              if(i == maxIterations) {
52                  array[x] = 0;
53              } else {
54                  double z = sqrt(newRe * newRe + newIm * newIm);
55                  int brightness = 256 * log2(1.75 + i - log2(log2(z))) / log2((double)maxIterations);
56                  array[x] = brightness;
57              }
58          }
59          int l = 0;
60          for(l=0; l < w; l++){
61              if(array[l] == 0) {
62                  color(0, 0, 0);
63              } else {
64                  int brightness = array[l];
65                  color(brightness, brightness, 255);
66              }
67          }
68      }
69
70      end = omp_get_wtime();
```

Figure 2: New version of printing pixel color

The sequencial code generates the Mandelbrot set image in a single thread, while the parallelized code uses OpenMP to parallelize the generation process and improve performance. The main difference is that in the parallelized version, the inner loop that generates the colors for each pixel is parallelized using the omp parallel for directive, which splits the iterations among multiple threads to execute them in parallel.

In addition, the parallelized code uses a private clause to declare private variables that are used within the loop, which helps to avoid race conditions and other concurrency issues that may arise when multiple threads access the same variables simultaneously.

It also uses an array to store the brightness values for each pixel in each row, and then iterates over the array to output the colors. This can help to reduce the number of I/O operations and improve performance compared to outputting each color individually in the inner loop.

Finally, this version uses *omp_get_wtime()* to measure the elapsed time for generating the image, while the first code uses *clock()*. *omp_get_wtime()* provides higher precision timing than *clock()*, which may be useful when measuring the performance of parallel programs.

3 Obtained Results

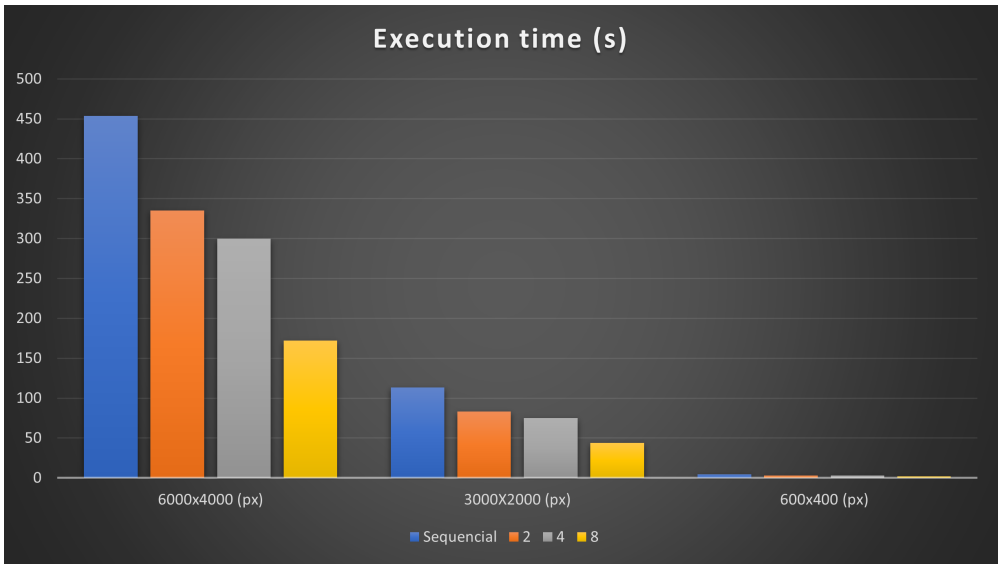


Figure 3: Execution time of all mandelbrot versions

The table shows the execution time (in seconds) for the sequential version and different numbers of threads (2, 4, and 8) of the parallelized version of the mandelbrot set generation program for three different image sizes: 6000x4000, 3000x2000, and 600x400 and 10,000 iterations.

As we can see, the parallelized version is faster than the sequential version for all image sizes and number of threads, which is expected because the parallelized version distributes the workload among multiple threads, allowing the program to make use of multiple processors/cores simultaneously.

Moreover, the execution time decreases as the number of threads increases for all image sizes, except for the 600x400 image size, where the execution time with 2,4 and 8 threads are almost the same. This behavior is expected because increasing the number of threads can only improve performance up to a certain point, after which the overhead of managing the threads becomes a bottleneck and starts to degrade performance.

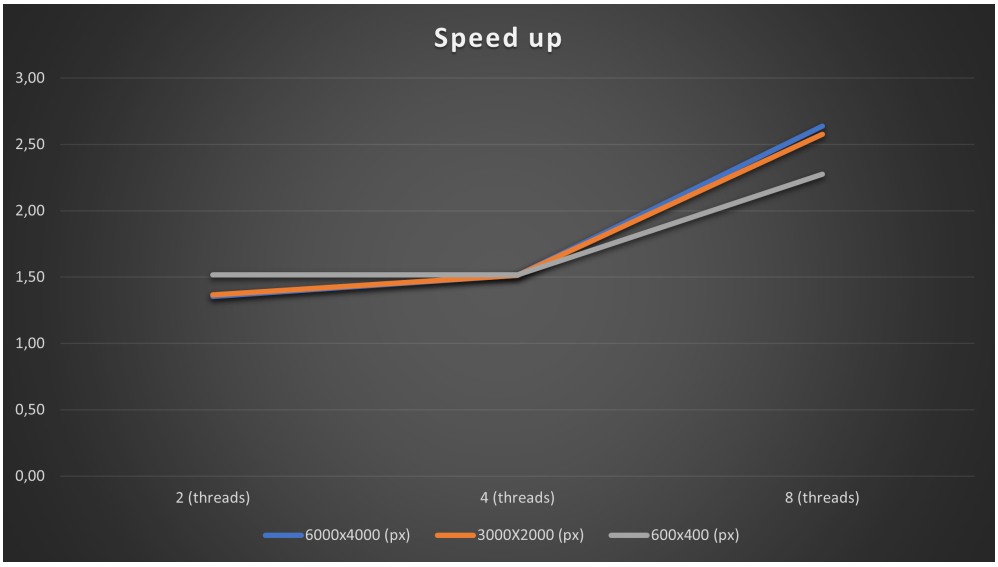


Figure 4: Speed up of the different executions

Speedup is calculated as the ratio of the sequential execution time to the parallel execution time. A speedup greater than 1 means that the parallel version is faster than the sequential version.

Looking at the table, we can see that as the number of threads increases, the speedup generally improves. This is because with more threads, the workload is divided among more processors, allowing the computation to be done more quickly.

For the largest image size (6000x4000), we can see that using 8 threads results in the highest speedup (2.64), followed by 4 threads (1.51), and finally 2 threads (1.35). This suggests that for very large problems, using more threads can be more beneficial.

For the smaller image sizes (3000x2000 and 600x400), the speedup is generally lower, and there is not much difference between using 2, 4, or 8 threads. This suggests that for smaller problems, the benefits of parallelization may not be as significant.

Overall, the speedup achieved by parallelization is also more significant for larger image sizes, as shown by the fact that the ratio between the sequential and parallel execution times is higher for the 6000x4000 image size than for the other two image sizes. This is because larger image sizes require more computations and take longer to generate, so the benefit of parallelization is more pronounced.

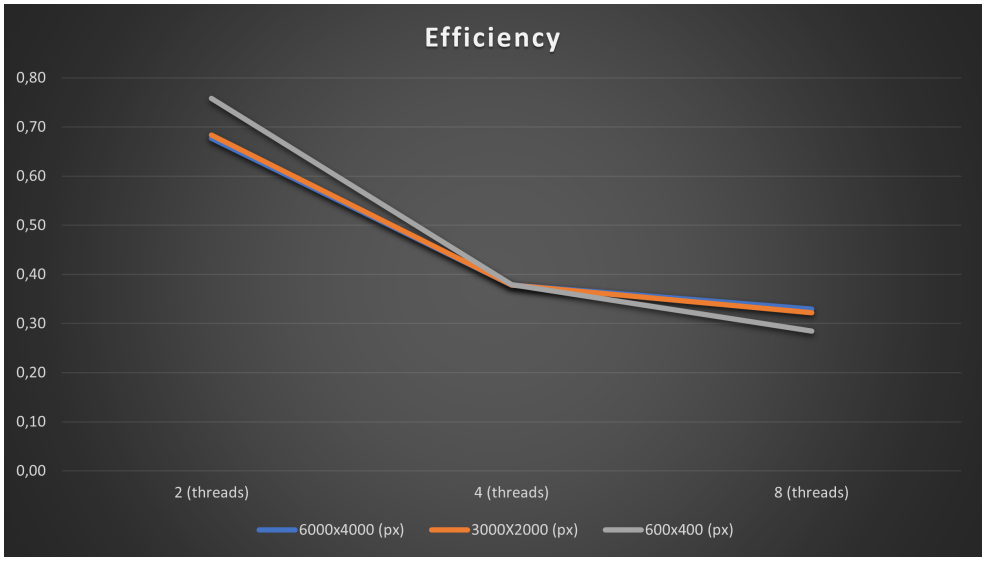


Figure 5: Efficiency of the different executions

This table shows the efficiency of the parallelization of the mandelbrot code indicates how well the code scales as the number of threads increases. In general, an efficient parallel implementation should have an efficiency close to 1.0, indicating that it is utilizing the available resources effectively. However, in practice, it is rare to achieve perfect scalability due to various factors such as the problem size, data dependencies, and resource limitations.

As we can see, the efficiency decreases as the number of threads increases for all image sizes. This is due to the overhead associated with parallelizing the algorithm, such as thread synchronization and communication, becoming more significant as the number of threads increases.

In addition, the efficiency is influenced by the size of the problem. For the larger problem sizes (6000x4000 and 3000x2000), the efficiency is generally lower than for the smaller problem size (600x400). This is because larger problem sizes require more computations, and the overhead of managing the threads becomes more significant.

Overall, the table suggests that the parallelization of the mandelbrot code is not very efficient, especially for larger problem sizes and higher numbers of threads. Improving the parallelization efficiency may require reducing the overhead of thread management, optimizing the algorithm for parallelism, or using more advanced parallelization techniques.

4 Conclusions

Generating a mandelbrot set image can be computationally intensive, so parallelization is often used to speed up the process.

To evaluate the effectiveness of parallelization, three versions of the Mandelbrot set code were tested: a sequential version and three parallel versions using OpenMP with two, four and eight threads respectively. Three different image sizes were used to test the code: 6000x4000, 3000x2000, and 600x400 pixels and 10,000 iterations.

The results of the tests were recorded in three tables: one for execution time, one for speedup, and one for efficiency. The execution time table showed that the parallel versions of the code were faster than the sequential version, with the speedup increasing as the number of threads increased. The largest image size (6000x4000 pixels) had the greatest speedup, while the smallest image size (600x400 pixels) had the smallest speedup.

The speedup table showed that increasing the number of threads increased the speedup, with the largest image size having the highest speedup. The efficiency table showed that the efficiency of the parallel versions of the code decreased as the number of threads increased, with the largest image size having the lowest efficiency.

In conclusion, parallelization can significantly reduce the execution time of the mandelbrot set code, with the speedup increasing as the number of threads increases. However, increasing the number of threads beyond a certain point can lead to decreased efficiency due to overhead and contention for resources. The optimal number of threads may depend on the specific hardware and problem size being used. Therefore, careful consideration and testing are necessary to determine the most efficient parallelization strategy for generating mandelbrot set images.