Università di Genova

MSc Computer Science
Data Science and Engineering Curriculum

Parallelization of the Mandelbrot algorithm

High Performance Computing

Andrea Bricola, Roberto Di Via, Matteo Interlando

February, 2023

Contents

1 Introduction

The Mandelbrot algorithm is a popular method for generating fractal images, which are known for their intricate and self-similar patterns. While the algorithm is relatively simple to understand and implement, it can be computationally intensive, especially when generating high-resolution images. As a result, there has been a lot of interest in finding ways to speed up the algorithm, and one promising approach is parallelization. By dividing the workload among multiple processing units, it is possible to significantly reduce the time required to generate an image. In this project, we will explore the use of three different parallelization technologies to improve the performance of the Mandelbrot algorithm: OpenMP, MPI, and CUDA.

OpenMP (Open Multi-Processing) is a set of compiler directives and library routines that enable users to write parallel programs for shared memory systems.

MPI (Message Passing Interface) is a standardized and portable message-passing system that is widely used for parallel computing on distributed memory systems.

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA for general-purpose computing on GPUs (graphics processing units).

By comparing the performance of these three approaches, we aim to find the most effective method for generating high-resolution Mandelbrot images efficiently.

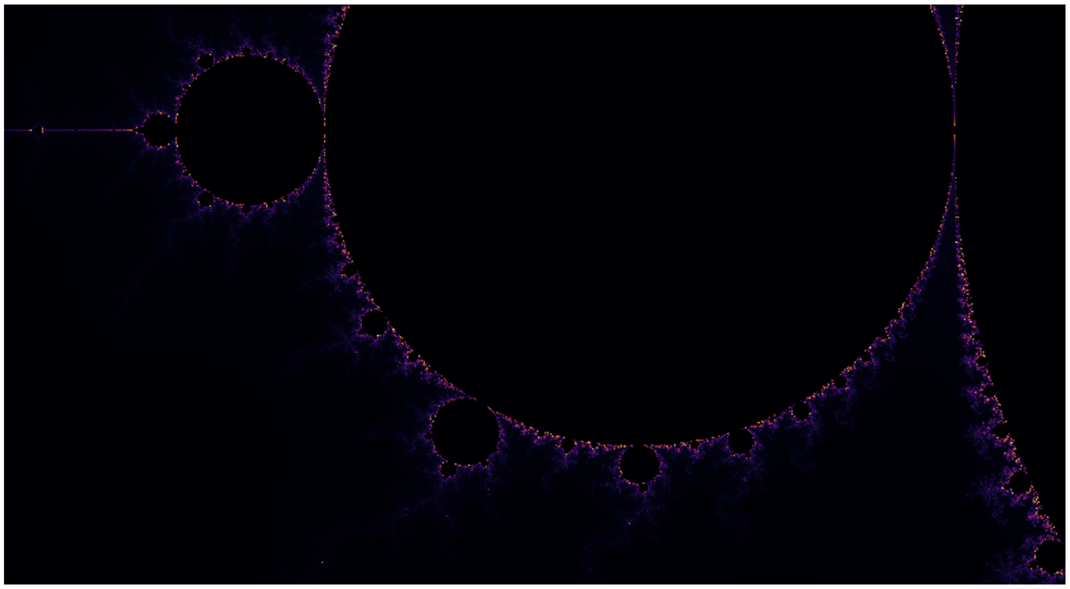


Figure 1: Visualization of Mandelbrot set computed by our program

1.1 Resources used for the experiments:

The experiments with OMP and MPI are conducted in a computer cluster owned by the INFN (Istituto Nazionale di Fisica Nucleare)). This cluster contains 8 nodes with each an Intel® Xeon Phi™ 7210 as CPU. This CPU has 64 cores and supports hyperthreading with 256 logical cores.

For the CUDA experiments we used a machine equipped with an Nvidia Geforce GTX 1650. This graphic card is built on Turing (TU117) architecture and contains 896 CUDA cores building up 14 Streaming Multiprocessors. [ref]

1.2 How to compile and execute the program

1.2.1 SERIAL version:

To compile the serial version of the program we used the Intel C++ Compiler (icpc) using 1 thread.

```
export OMP_NUM_THREADS=1; icpc -qopenmp progetto.cpp
```

To obtained the serial execution time of the program we used the ”time” command.

```
time ./a.out
```

Compiling and executing the serial version of the program we obtained:

- Resolution 360: **20.12 s**
- Resolution 720: **80.419 s**
- Resolution 1000: **155.863 s**

1.2.2 OMP version:

To compile the OMP version we used the following statement:

```
export OMP_NUM_THREADS=<n>; icpc -qopenmp progetto_omp.cpp
```

where:

<n>	Corresponds to the number of threads we want to use.
-qopenmp	Generate multi-threaded code based on OpenMP* directives.

While to execute and takes the total execution time of the program we used:

```
time ./a.out image.data
```

Disclaimer: We kept the default -O2 optimization level since it achieves already good performances.

1.2.3 MPI version:

To compile the MPI version we used the following statement:

```
mpiicpc progetto_mpi.c
```

While to takes the total execution time and to execute the program we used:

```
time mpirun -np <n> ./a.out img.data
```

In this way we execute the different processes on a single machine, to execute them on multiple machines we have to specify other flags:

-np	Specify the number of processes to run.
-hostfile	Specify a file containing a list of hostnames where MPI processes should be run.
-perhost	Specify the number of processes to run on each host.

So, in our case, the command line to distribute the processes is:

```
time mpirun -hostfile ~/mpi_hosts.txt -np <n> -perhost <n> ./a.out img.data
```

1.2.4 MPI + OMP version:

For the parallelized MPI + OMP program, it's like the MPI version but first we have to define the number of threads for the OMP directives. In our case the command line used is:

```
export OMP_NUM_THREADS = <n>
```

```
mpiicpc -qopenmp progetto_mpi+omp.cpp
```

```
time mpirun -hostfile ~/mpi_hosts.txt -np <n> -perhost <n> ./a.out img.data
```

1.2.5 CUDA version:

To compile the CUDA version we simply do:

```
nvcc progetto_cuda.cu
```

While to execute:

```
time ./a.out img.data
```

2 Analysis of the program

The program concerns computing the Mandelbrot set, that is a fractal created by iterating a function of a complex number and checking if the magnitude of the result is less than or equal to 2. If it is, the number is considered to be part of the Mandelbrot set. The set is represented graphically by coloring the points in the complex plane that are in the set black. Meanwhile the points that are not in the set are drawn with various colors based on how quickly they escape the Mandelbrot set, i.e. when the inner loop figures out they do not belong to the Mandelbrot set.

First, the program defines the boundaries of the complex plane to be used in the image, as well as the ratio and resolution of the image. In the main function, it allocates an array of size HEIGHT x WIDTH to store the results of the computation.

For each pixel in the image, the program determines if the corresponding point in the complex plane belongs to the Mandelbrot set. To do this, it calculates the starting number c and then repeatedly updates $z = z^2 + c$ until the absolute value of z reaches 2. If this happens, the point is determined to be outside the Mandelbrot set, and the number of iterations needed to reach 2 is recorded. This number of iterations is stored for each point as it will be used to color the image in order to visualize the fractal.

```cpp
int *const image = new int[HEIGHT * WIDTH];
for (int pos = 0; pos < HEIGHT * WIDTH; pos++) {
    image[pos] = 0;

    const int row = pos / WIDTH;
    const int col = pos % WIDTH;
    const complex<double> c(col * STEP + MIN_X, row * STEP + MIN_Y);

    complex<double> z(0, 0);
    for (int i = 1; i <= ITERATIONS; i++) {
        z = pow(z, 2) + c;   // z = z^2 + c

        if (abs(z) >= 2) { // If it is convergent
            image[pos] = i;
            break;
        }
    }
}
```

After the core computation, the program stores the image as a list of numbers in a file.

```cpp
ofstream matrix_out;     // Write the result to a file
matrix_out.open(argv[1], ios::trunc);
for (int row = 0; row < HEIGHT; row++) {
    for (int col = 0; col < WIDTH; col++) {
        matrix_out << image[row * WIDTH + col];
        if (col < WIDTH - 1) matrix_out << ',';
    }
    if (row < HEIGHT - 1) matrix_out << endl;
}
```

```
    matrix_out.close();
```

3 Parallelization with OMP

The goal of the problem is to determine if a pixel is part of the Mandelbrot set. As each pixel's computation is independent, it can be parallelized using multiple processing units. The OpenMP library allows us to create a multi-threaded program and utilize all the cores of our computing machine.

Specifically to our program, we decided to parallelize the outer loop in which every iteration corresponds to a separate pixel. In order to do this we inserted **#pragma omp parallel for**.

```
#pragma omp parallel for
for (int pos = 0; pos < HEIGHT * WIDTH; pos++) {
    image[pos] = 0;
    const int row = pos / WIDTH;
    const int col = pos % WIDTH;
    const complex<double> c(col * STEP + MIN_X, row * STEP + MIN_Y);
    complex<double> z(0, 0);

    for (int i = 1; i <= ITERATIONS; i++) {
        z = pow(z, 2) + c;

        if (abs(z) >= 2) // If it is convergent
        {
            image[pos] = i;
            break;
        }
    }
}
```

In this way the computation is divided into N threads, each working on a specific number of pixels, as determined by the variable 'pos'. However, the inner loop, which involves the repeated update of a complex number, cannot be parallelized as it requires the output from the previous iteration before proceeding. Since the variables being modified (pos, row, col, c, and z) are all private to each iteration of the loop, there is no need to use the private clause in the "#pragma omp parallel" directive.

3.1 Experiments

To gain a deeper understanding of the impact of parallelization, we conducted experiments with three variations of the program. In particular, using a resolution of 360, 720, and 1000, respectively.

3.1.1 Execution time

First, we collected the execution time of the parallel program with the 3 different resolutions.

Execution time			
Threads / Resolution	360	720	1000
1	25.05	97.69	186.58
2	14.88	50.27	96.01
4	11.83	42.76	80.82
8	7.61	26.9	50.96
16	4.77	16.15	29.02
32	3.88	9.82	17.68
64	3.36	7.83	13.35
128	3.36	5.94	10.96
256	3.13	5.64	8.69

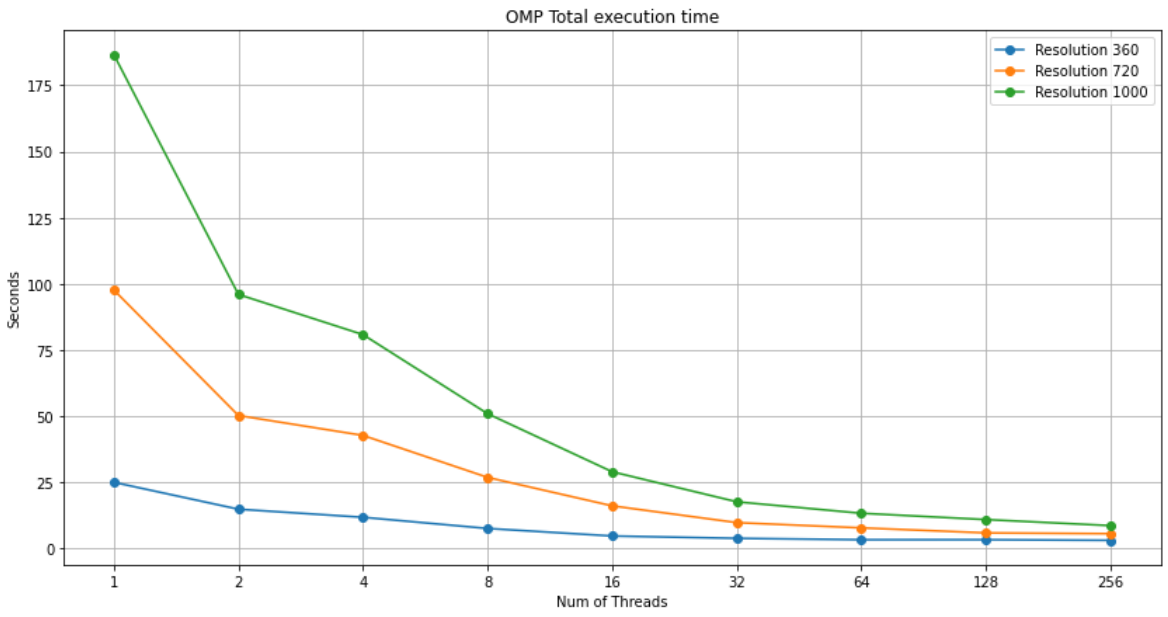


Figure 2: Time execution

3.1.2 Dynamic scheduling VS Static scheduling

Dynamic scheduling in the OpenMP library is useful when the workload of different iterations of the loop is not known and/or varies from iteration to iteration.

Static scheduling in the OpenMP library is useful when the workload of the iterations is known and does not vary from iteration to iteration. Static scheduling is also more efficient than dynamic scheduling, as it can allow the threads to be scheduled in advance so that the threads do not need to wait for the work to be distributed at runtime.

In our problem, the double loop has iterations whoose execution time is different. Some points may be determined not to belong to the Mandelbrot set after just one iteration of the inner loop, while other points may take many more iterations to escape. Some points may not escape even after 1000 iterations. Given this variability, using a **dynamic scheduling** approach may be more appropriate than static scheduling. So we implemented it in this way:

```
#pragma omp parallel for schedule(dynamic)
for (int pos = 0; pos < HEIGHT * WIDTH; pos++) {
    ...
    for (int i = 1; i <= ITERATIONS; i++) {
        ...
    }
}
```

We can see from the following results that doing these changes we get a better performance.

Execution time with Dynamic scheduling			
Threads / Resolution	360	720	1000
1	25.45	98.3	185.35
2	13.38	51.19	96.66
4	7.94	29.56	51.89
8	4.63	16.38	28.63
16	3.26	10.19	16.3
32	2.9	7.61	12.51
64	2.63	**6.03**	9.5
128	**2.42**	6.22	8.67
256	2.7	7.92	**8.42**

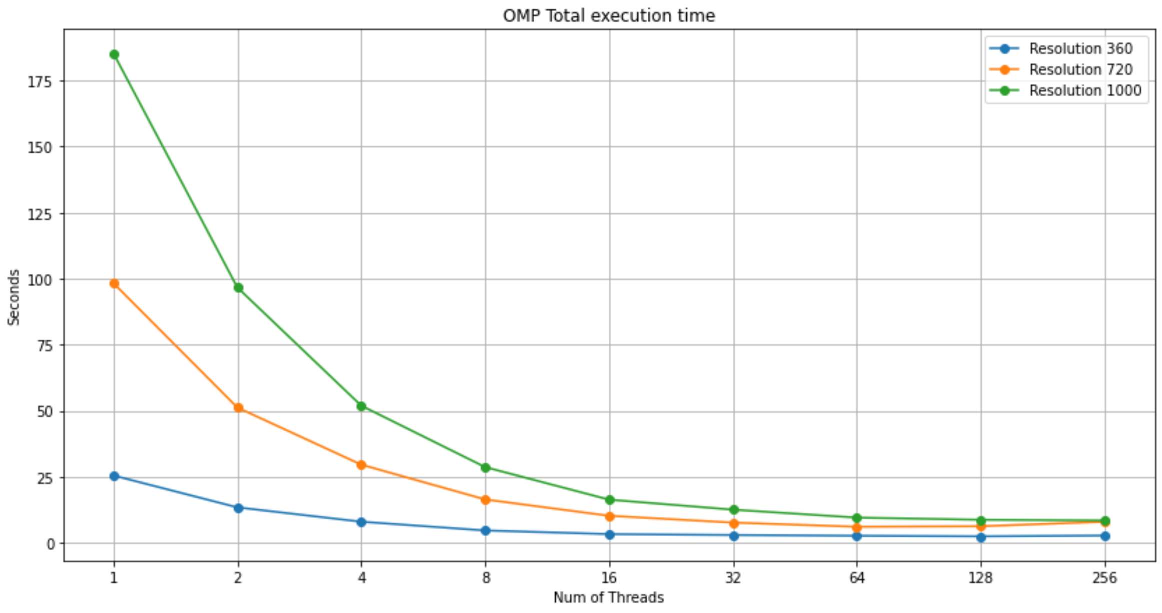


Figure 3: Time execution with dynamic schedule

3.1.3 Percentange of the parallel component

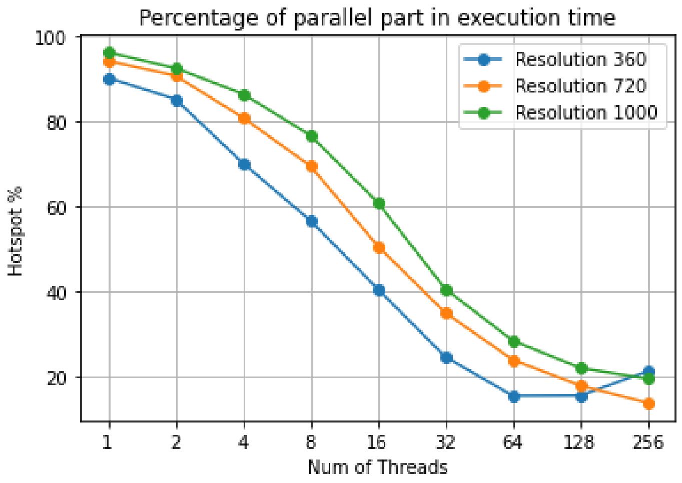


Figure 4: Percentage of parallel part on total with regards to exec time

Regarding the hotspot, the execution time decreased proportionally as the number of threads increased.

3.1.4 The Speedup

There are several reasons why it is useful to calculate the speedup of a parallel computing system. By calculating the speedup, we can compare the relative performance of different parallel computing systems. We can determine the optimal configuration under different configurations (e.g., using different numbers of threads), or we can determine how well the system is utilizing its resources and identify potential bottlenecks or inefficiencies.

The speedup in parallel computing can be straightforwardly defined as:

$$Speedup = \frac{Time\ on\ single\ processor}{Time\ on\ parallel\ system}$$

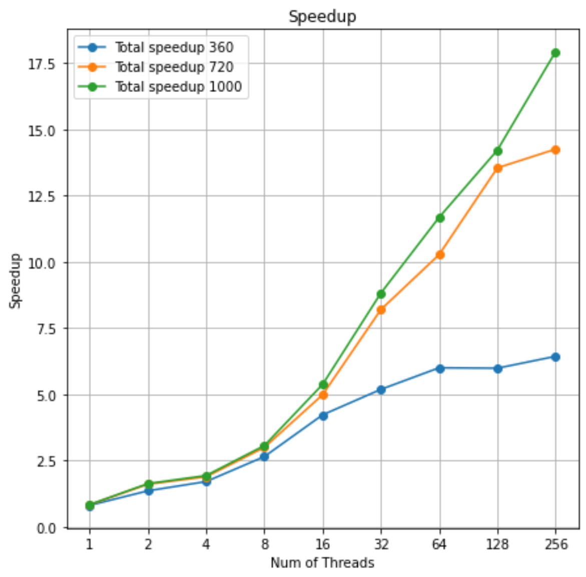


Figure 5: Speedup with static schedule

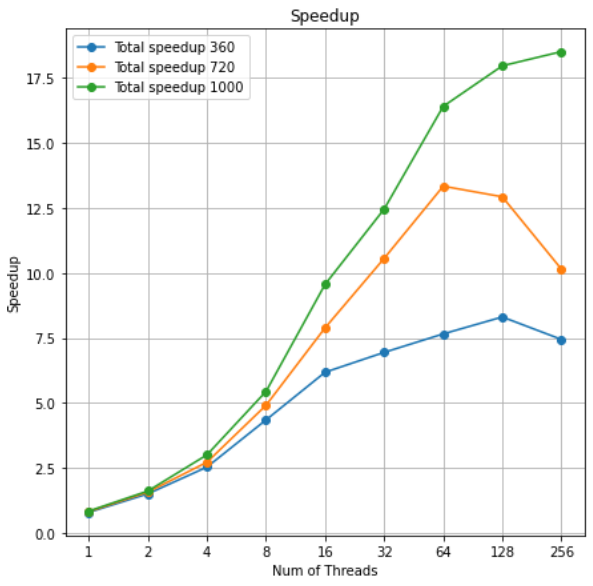


Figure 6: Speedup with dynamic schedule

For the case of resolution 1000 results indicate that utilizing dynamic scheduling in OpenMP resulted in improved computation speed, as predicted.
Meanwhile in resolution 720 static scheduling showed better performance at 128 and 256 threads .

3.1.5 The Efficiency

The efficiency of a parallelized program is a measure of how well the program utilizes the available resources, it ranges between 0 and 1 (where 1 corresponds to the ideal speedup) and it is calculated as the ratio of the speedup to the number of threads used.

$$E = \frac{Speedup}{Number\ of\ Threads}$$

Where *Speedup* is the ratio of the execution time of the program on a single processor to the execution time of the program on multiple processors.

The efficiency obtained considering the dynamic speedup (represented in figure 8) is as follows:

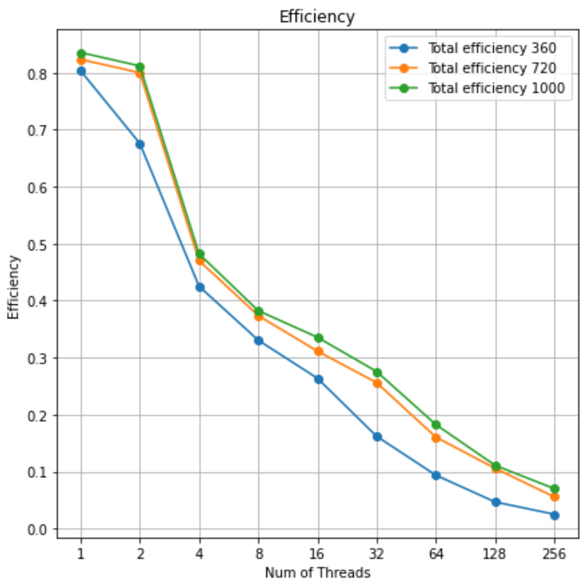


Figure 7: Efficiency for static schedule

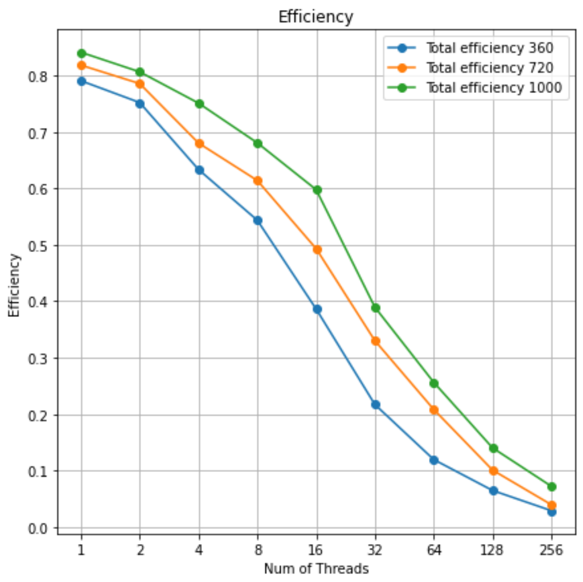


Figure 8: Efficiency for dynamic schedule

The efficiency chart indicates that the computation is highly efficient until the use of 8 threads. Beyond that, the efficiency drops significantly, indicating that a faster execution time is obtained at the cost of great electricity consumption due to having all cores busy. This highlights that efficiency is a useful metric to determine when excessive energy consumption results in only minimal performance gains.

4 Parallelization with MPI

The Message Passing Interface (MPI) is a standardized and portable system for message-passing between parallel processes that communicate over a network.

In the Mandelbrot program, we parallelized the calculation of each pixel's membership in the Mandelbrot set over multiple Linux processes. Each process computes a portion of the image and then sends it to the root process, which combines all the pieces to form the complete image. The workload was divided among the processes by dividing the image into equal chunks, with the exception of the last process, which may have a slightly larger chunk if the image size is not evenly divisible by the number of processes.

```
    int chunk_size = size / world_size;
    int pos_start = rank * chunk_size;
    int pos_end = pos_start + chunk_size - 1;

    int pos_start_last = (world_size - 1) * chunk_size;
    int chunk_size_last = size - pos_start_last;

    if(rank == world_size - 1 && pos_end < size - 1){
        pos_end = size - 1;
        chunk_size = chunk_size_last;
    }
```

Each process calculates if pixels of the chuck belong to the Mandelbrot set, with knowledge of the start and end position of the chunk. The results are stored in a send buffer, which will be used for communication at the end of the computation.

```
    for (int pos = pos_start; pos <= pos_end; pos++) {
        buf_send[j] = 0;
        ...
        for (int i = 1; i <= ITERATIONS; i++) {
            z = pow(z, 2) + c;
            if (abs(z) >= 2) {
                buf_send[j] = i;
                break;
            }
        }
        j++;
    }
```

Then, each process will communicate the result to the root process, through **MPI_Gatherv** function.

```
int* counts;
int* displacements;
if(rank == root){
    counts = new int[world_size];
    displacements = new int[world_size];

    for(int i = 0; i < world_size - 1; i++){
        counts[i] = chunk_size;
        displacements[i] = i * chunk_size;
    }

    counts[world_size - 1] = chunk_size_last;
    displacements[world_size - 1] = pos_start_last;
}

MPI_Gatherv(buf_send, chunk_size, MPI_INT, image, counts, displacements, MPI_INT, root, MPI_COMM_WORLD);
```

4.1 MPI in one machine

In our first experiment with MPI, we aimed to compare the results of scaling the parallel program within a single machine.

4.1.1 Execution time

First we analyzed the execution time with different number of processes and image resolutions.

Execution time on 1 Machine			
Processes / Resolution	360	720	1000
1	25.847	92.794	165.107
2	15.49	49.531	88.923
4	15.363	43.525	73.153
8	11.122	31.114	47.367
16	**8.495**	20.775	30.431
32	10.125	15.429	22.244
64	10.591	**15.371**	**15.95**
128	12.821	17.819	18.44
256	19.703	25.713	27.356

The results show that, for the image resolution of 1000, the fastest computation time using MPI on one machine is 15.95 seconds when using 64 parallel processes. However, this time was not as efficient as the best time obtained using OMP which was 8.42 seconds, achieved with dynamic scheduling and 256 parallel threads.

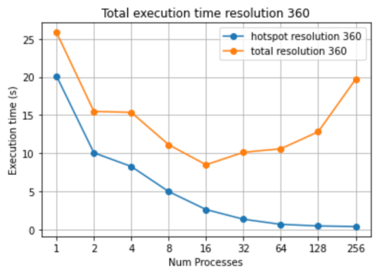


Figure 9: Resolution 360

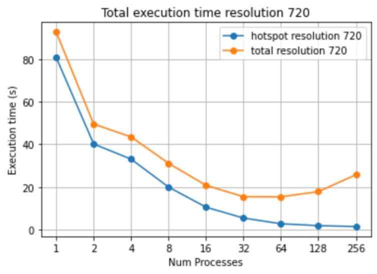


Figure 10: Resolution 720

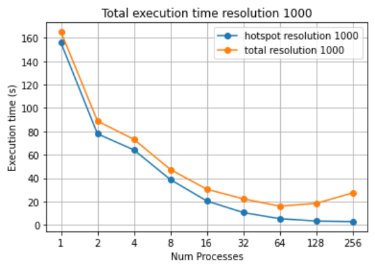


Figure 11: Resolution 1000

The pictures above display the execution time with different image resolutions. They represent the hotspot execution time in blue and the total execution time in orange. The results indicate that as the number of processes was increased from 1 to 256, the hotspot time improved, where 256 corresponds to the number of logical cores in the CPU of one machine. However, the total program time improves only up to a lower number of processes. Overuse of processes can result in increased initialization and communication times, particularly in the case of a 360 resolution where the hotspot is small and using too many processes is not worth it.

4.1.2 Percentage of the Parallel component

The figure labeled as "figure 12" illustrates the proportion of time spent by the parallel component compared to the overall time. As the number of processes increases, the hotspot time decreases and eventually the serial component of the program becomes dominant. At the highest level of parallelism with 256 processes, for a resolution of 1000, the hotspot consumes only 10% of the total time.

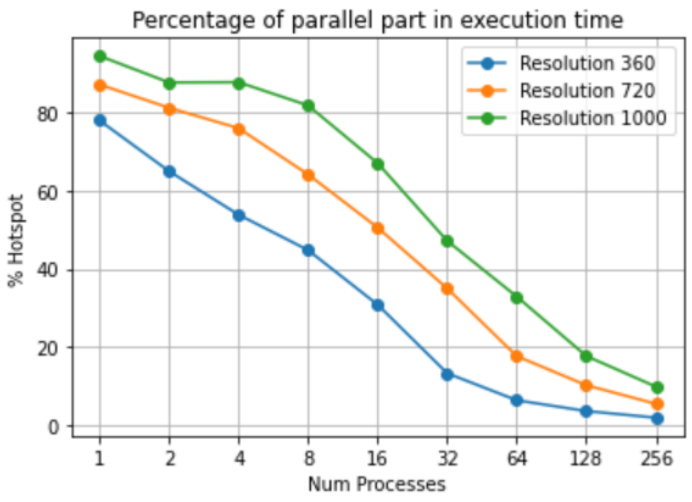


Figure 12: Percentage parallel portion VS serial portion

4.1.3 The Speedup

The figure 13 below shows the speedup provided when using MPI on a single machine. The best performance for resolution 1000 is achieved with 64 processes, however, the execution time is slower when the number of processes increases to 128 because of the overhead previously discussed.

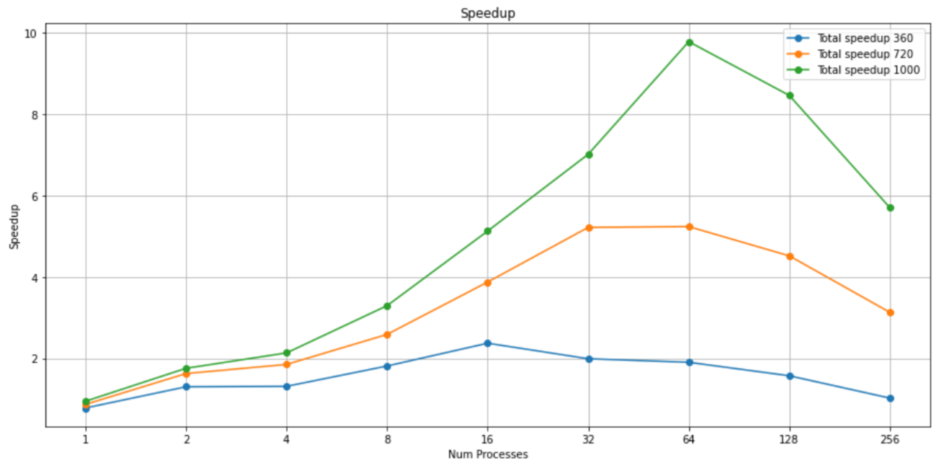


Figure 13: Speedup with different resolutions

4.1.4 The Efficiency

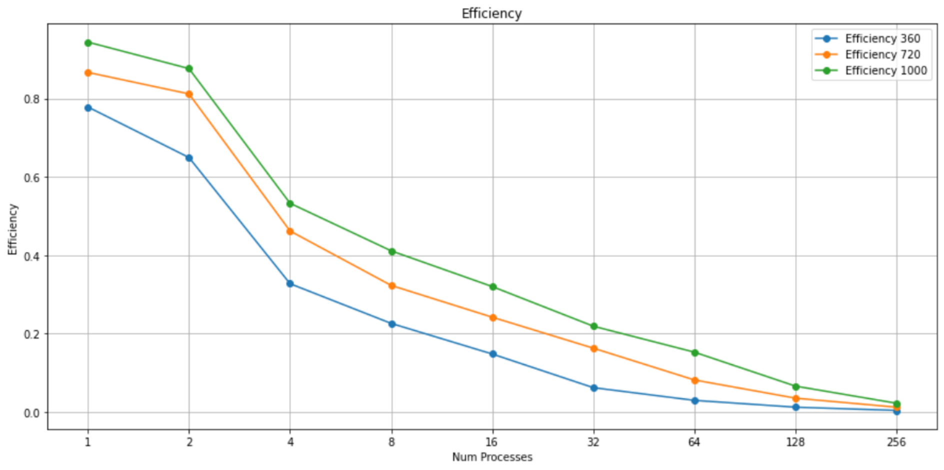


Figure 14: Efficiency with different resolution

The figure 14 shows the efficiency with different resolutions. Although the execution time was optimized with 64 processes, the resulting efficiency is below 20%, which may not be considered sufficient for a system that prioritizes environmental sustainability.

4.2 MPI in multiple machines

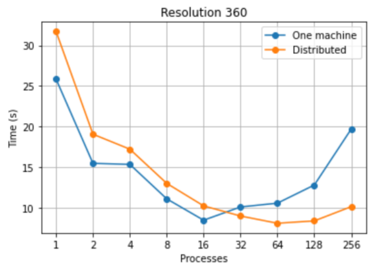
The second MPI experiment involved executing the MPI program distributing the processes across multiple machines (in this case 8), rather than on a single machine.

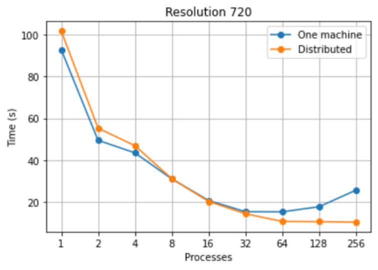
4.2.1 Execution time: 8 machines vs 1 machine

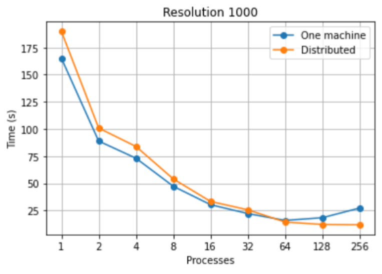
The experiment on the distributed MPI version resulted in improved execution time compared to the single machine MPI. For example in the resolution 1000 case the execution time was 11.99 seconds for eight machines versus 15.95 seconds for one machine. The best performance on a single machine was achieved using 64 processes, whereas the best performance on a distributed system was obtained with 256 processes.

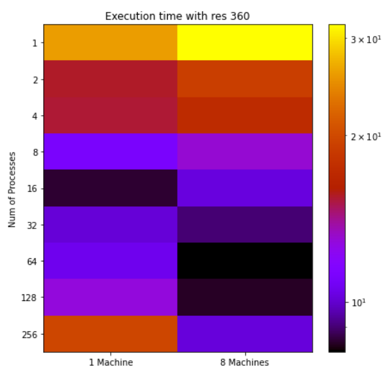
Execution time on 8 Machines			
Processes / Resolution	360	720	1000
1	31.715	101.69	189.773
2	19.75	55.318	100.895
4	17.235	46.964	83.695
8	13.025	31.197	54.116
16	10.256	20.365	33.409
32	19.012	14.464	25.702
64	**8.118**	**10.795**	14.459
128	8.410	10.691	12.232
256	10.161	10.49	**11.990**

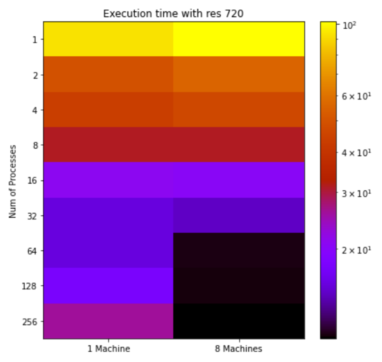
One machine has 256 logical cores and the experiment with our program showed that local MPI had no improvements as number of processes was scaled beyond 64 processes. However, the distributed MPI code was able to achieve better parallel performance with the same number of processes (128 and 256).

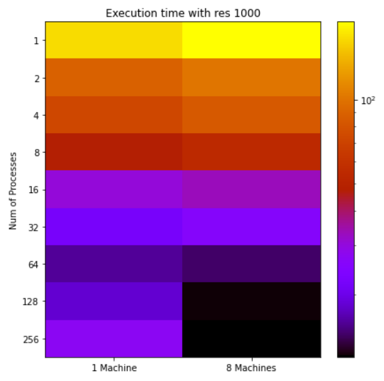








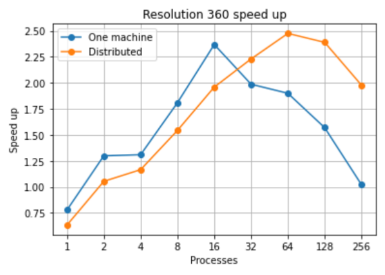


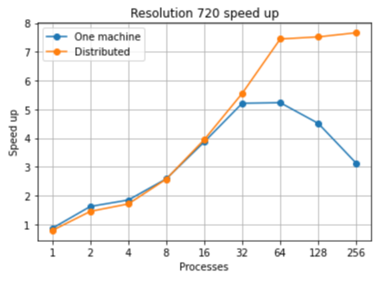


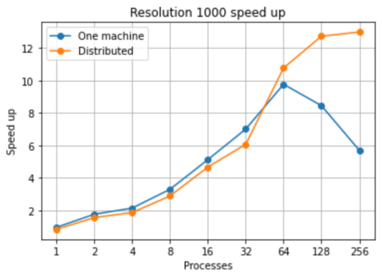
The heatmap visualization shows that a larger number of processes perform better in a computer cluster compared to a single machine. However, when the number of processes is low, local execution of the MPI system is faster.

4.2.2 Speedup: 8 machines vs 1 machine

The results of the experiments for resolutions 1000 and 720 showed that when the number of processes was less than or equal to 32, the speedup was similar for both versions. However, when the number of processes increased from 64 to 256, the distributed MPI version demonstrated better speedup.

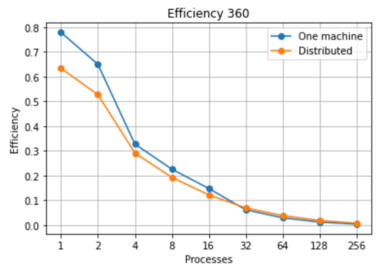


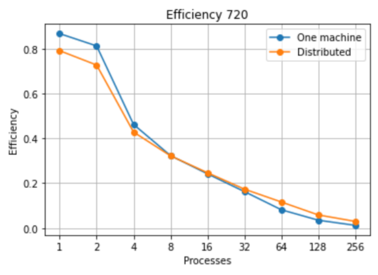


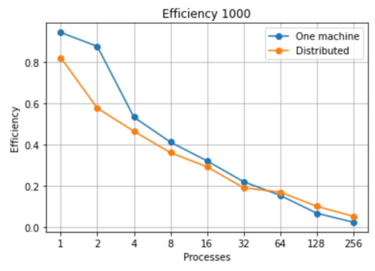


4.2.3 Efficiency: 8 machines VS 1 machine

The efficiency of both the distributed MPI program and the local program exhibit a similar trend. However, due to the improved execution time at 64, 128, and 256 processes, the distributed MPI program has a slightly higher efficiency.





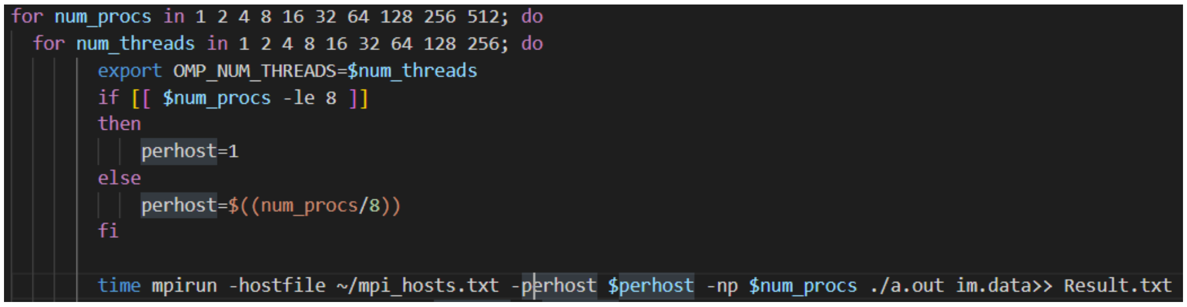


5 Parallelization with MPI + OMP

The program combining OMP and MPI operates by dividing the pixels of the image into segments on the first level, so each process operates on a segment. Then each process divides his segment of pixels among threads. To develop the MPI + OMP program we simply took the MPI program and added the OpenMP directive "#pragma omp parallel for" in the main loop.

```
#pragma omp parallel for schedule(dynamic)
for (int pos = pos_start; pos <= pos_end; pos++) {
    ...
    for (int i = 1; i <= ITERATIONS; i++) {
        ...
    }
}
...
#pragma omp parallel for
for(int i = 0; i < world_size - 1; i++) {
    counts[i] = chunk_size;
    displacements[i] = i * chunk_size;
}
```

We automatized the execution of the MPI+OMP program with different number of threads and different number of processes in order to find the best combination which minimizes the execution time.



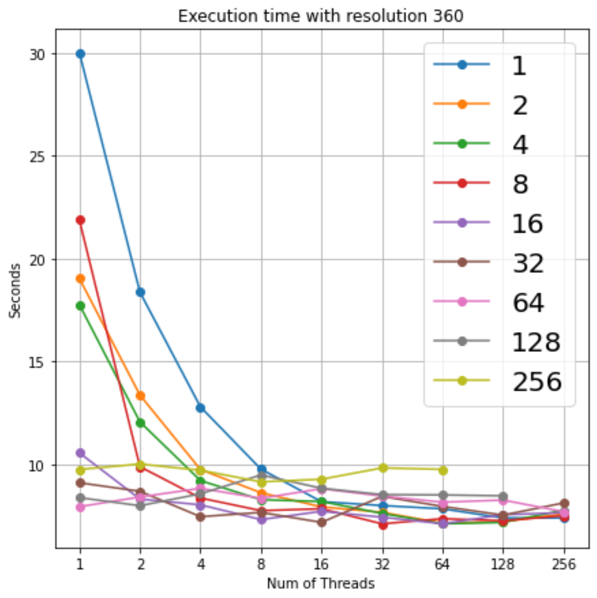
```
for num_procs in 1 2 4 8 16 32 64 128 256 512; do
  for num_threads in 1 2 4 8 16 32 64 128 256; do
      export OMP_NUM_THREADS=$num_threads
      if [[ $num_procs -le 8 ]]
      then
          perhost=1
      else
          perhost=$((num_procs/8))
      fi

      time mpirun -hostfile ~/mpi_hosts.txt -perhost $perhost -np $num_procs ./a.out im.data>> Result.txt
```

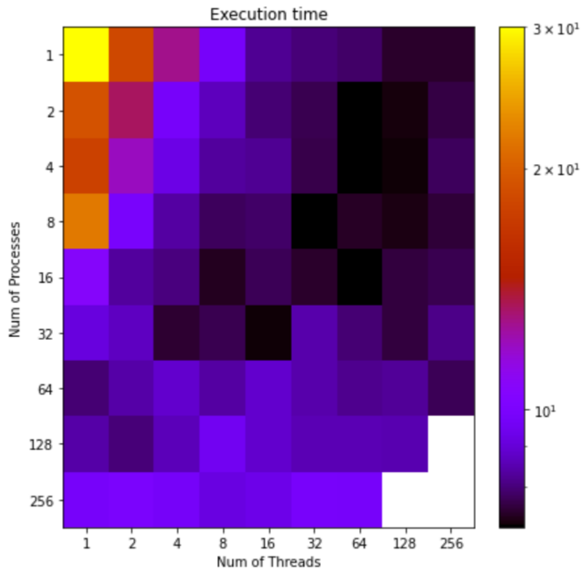
Figure 15: Automatic execution script.

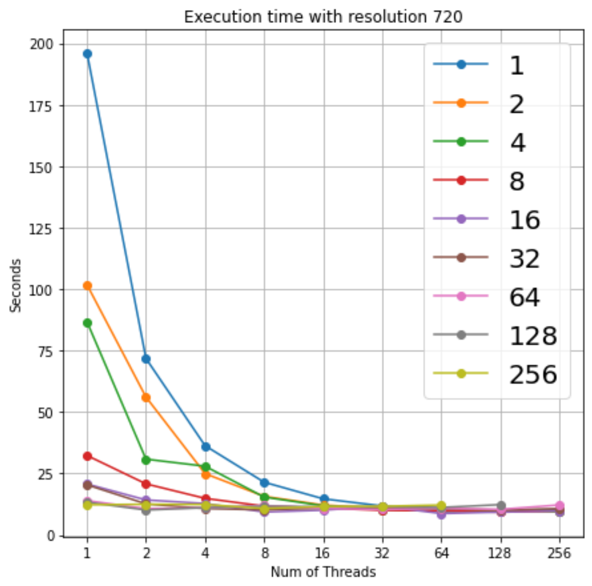
5.1 Experiments

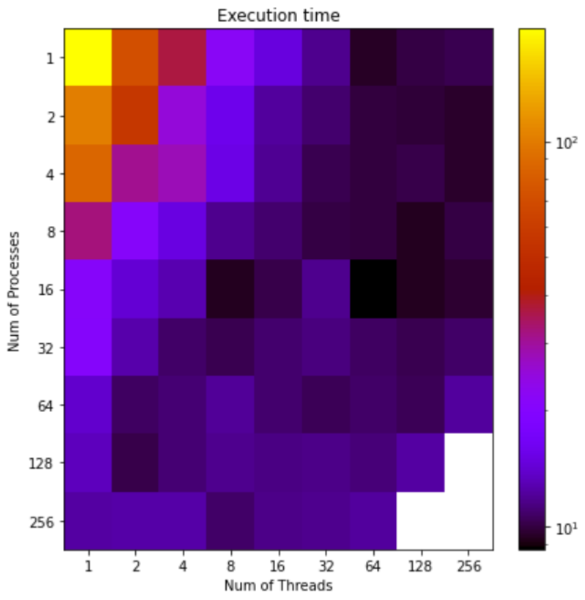
5.1.1 Execution Time

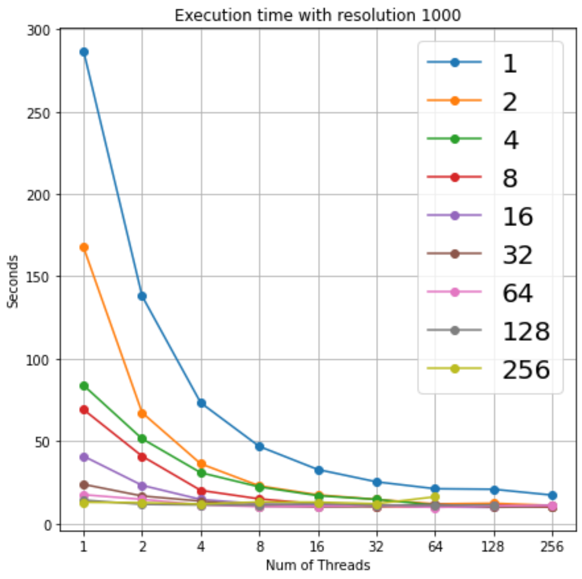
The results of our tests are organized into a table with the number of threads on the x-axis and the number of processes on the y-axis. The cells in the table are color-coded to visually indicate the optimal combinations of threads and processes for best performance. The bottom-right cells in the table are white, signifying that the OMP-MPI program crashes if an excessive number of threads and processes are attempted to be initialized (runs out of memory). The execution time curves are plotted in charts at the left side of the page. Here each line is color-coded based on the number of processes, while the number of threads are represented on the x-axis.

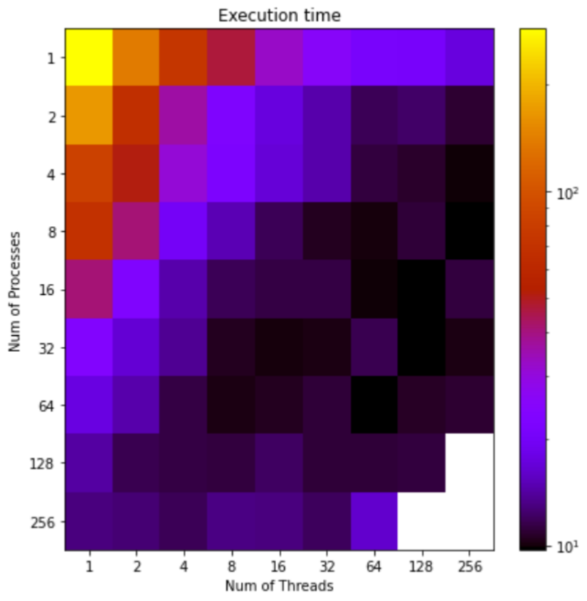

Execution time with resolution 360
Seconds
Num of Threads
1
2
4
8
16
32
64
128
256


Execution time
Num of Processes
Num of Threads
3×10^1
2×10^1
10^1


Execution time with resolution 720
Seconds
Num of Threads
1
2
4
8
16
32
64
128
256


Execution time
Num of Processes
Num of Threads
10^2
10^1





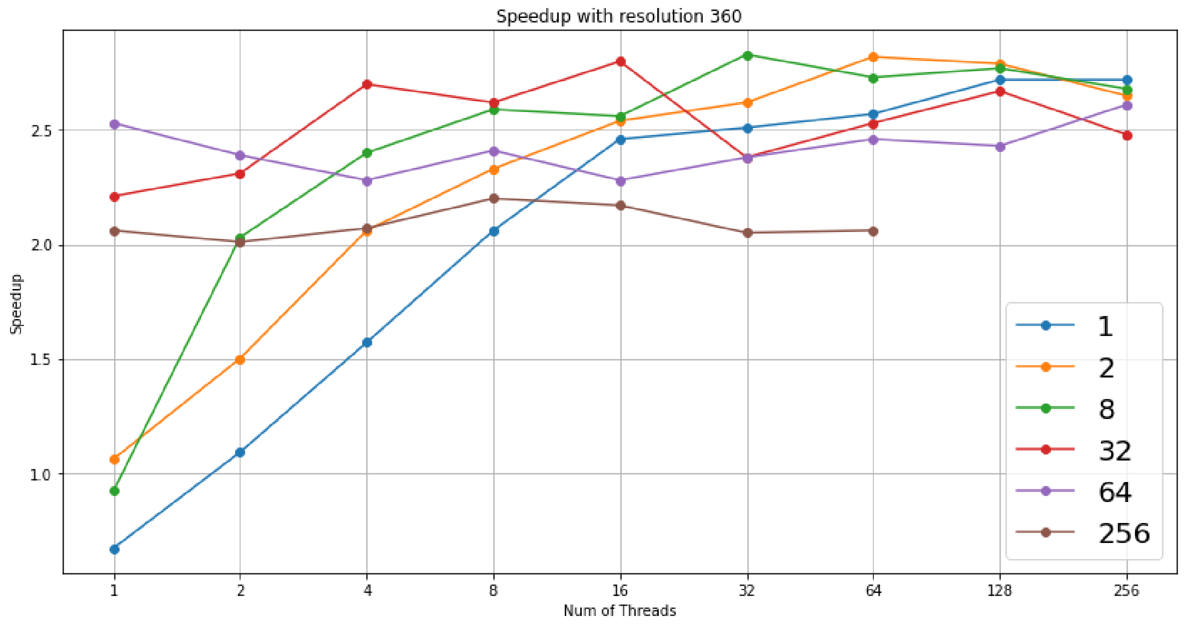
The colored table illustrates that a smaller number of processing units is optimal for resolution 360 as excessive parallelization incurs significant overhead. On the other hand, the resolution 1000 problem benefits from a larger number of processing units to speed up the hotspot. The best performance for resolution 1000 was achieved with 64 processes and 64 threads per process, resulting in a total of 4096 processing units, yielding an execution time of 9.73 seconds, which is faster than the 11.99 seconds obtained from the distributed MPI program using 256 processes.

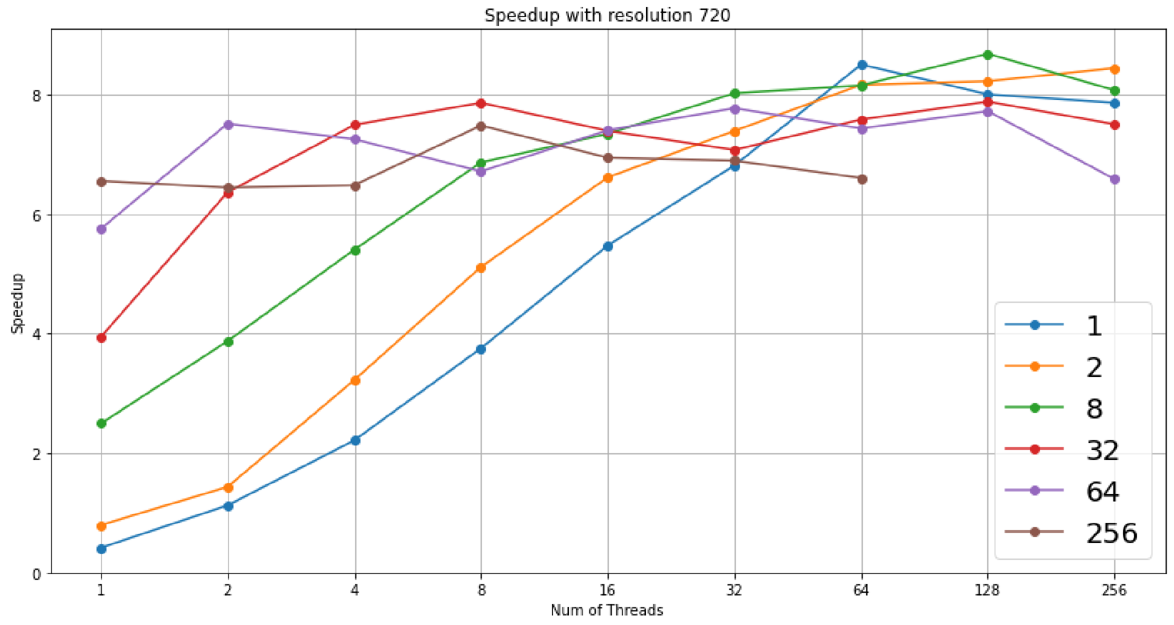
Resolution 1000: Execution time on 8 Machine

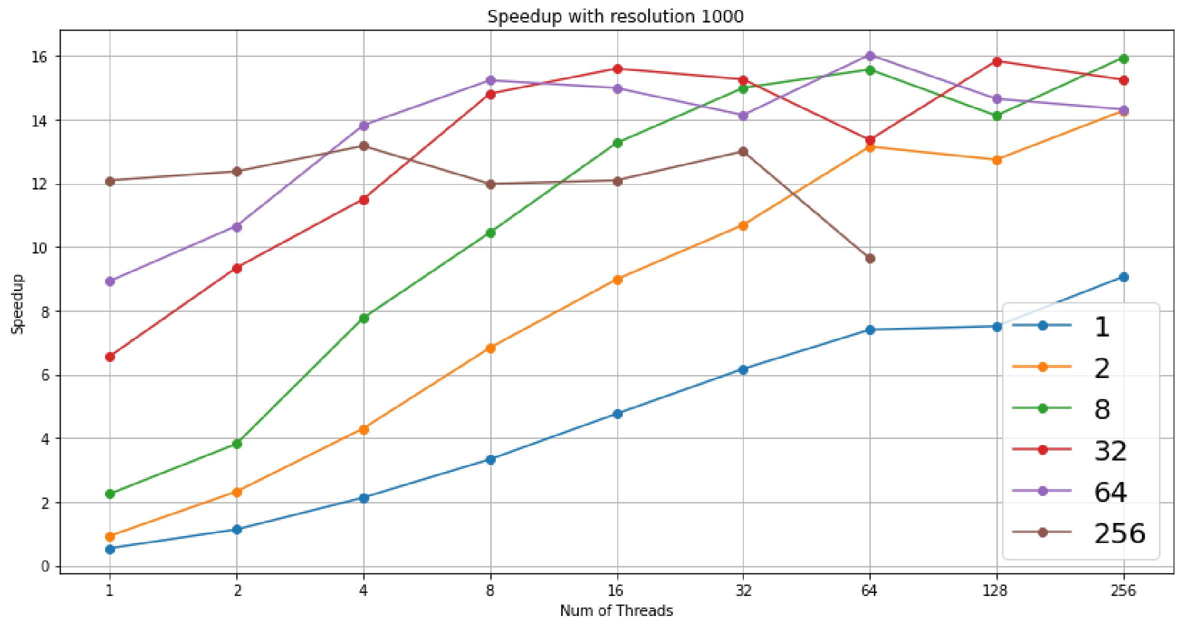
Processes / Threads	1	2	4	8	16	32	64	128	256
1	286.97	138.035	73.258	46.77	32.72	25.265	21.073	20.75	17.195
2	167.646	67.229	36.229	22.812	17.362	14.583	11.851	12.232	10.921
4	83.845	51.415	30.828	22.211	16.886	14.581	11.199	10.718	9.983
8	69.128	40.837	20.067	14.915	11.753	10.401	10.008	11.04	9.78
16	40.857	23.134	14.554	11.849	11.297	11.317	9.971	9.777	11.112
32	23.73	16.669	13.55	10.526	9.99	10.212	11.664	9.84	10.218
64	17.452	14.62	11.277	10.231	10.4	11.021	**9.726**	10.641	10.881
128	14.209	11.583	11.251	11.162	12.148	11.009	11.058	11.218	nan
256	12.892	12.6	11.833	13.012	12.89	11.986	16.172	nan	nan

5.1.2 The Speedup

The speedup of the MPI+OMP program is plotted as a function of the number of threads. The graphic differentiates between different configurations of processes by using different colors. This visualization allows to observe how the speedup improves as the number of threads increases while the number of processes remains constant.

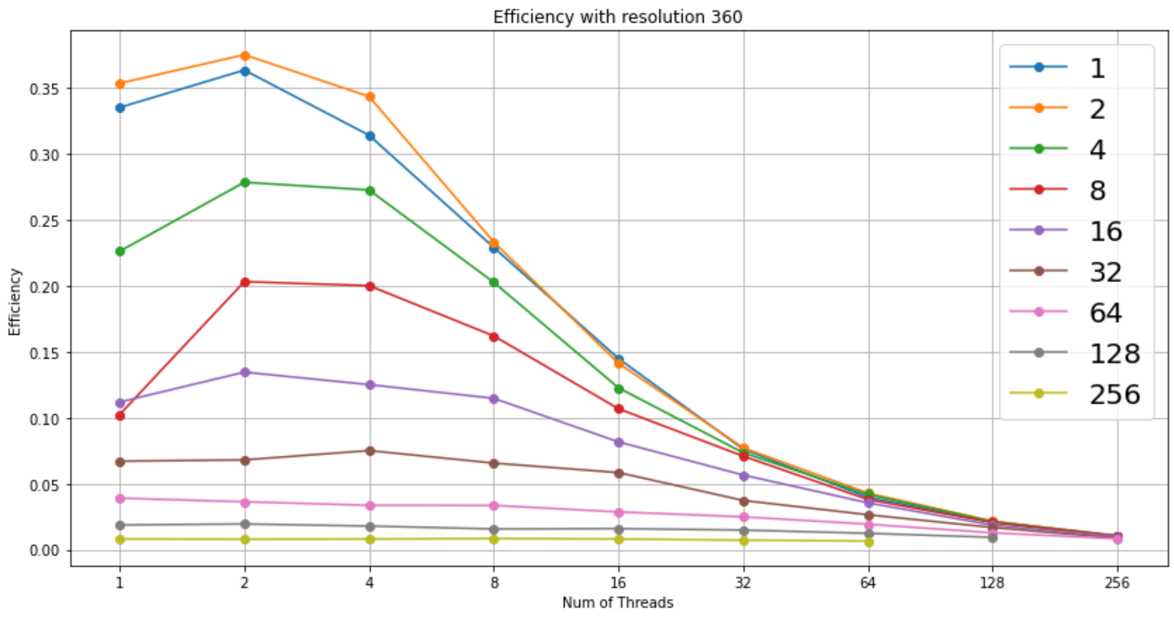

Speedup with resolution 360
Speedup
Num of Threads
1
2
4
8
16
32
64
128
256
0.5
1.0
1.5
2.0
2.5
1
2
8
32
64
256

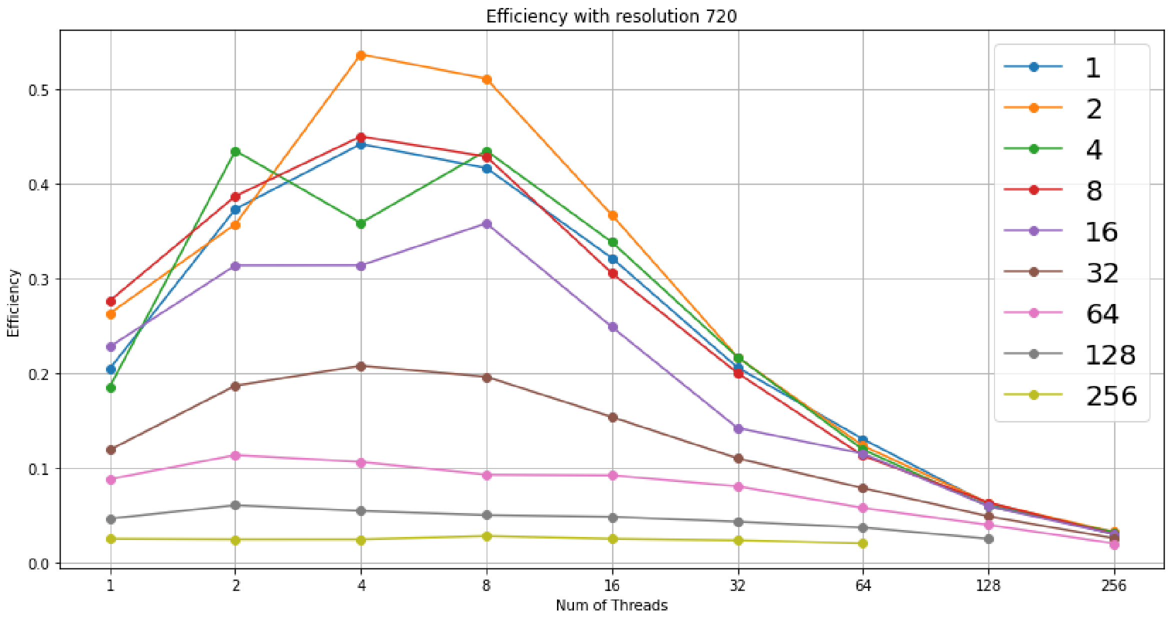

Speedup with resolution 720
Speedup
Num of Threads
0
2
4
6
8
1
2
4
8
16
32
64
128
256
1
2
8
32
64
256

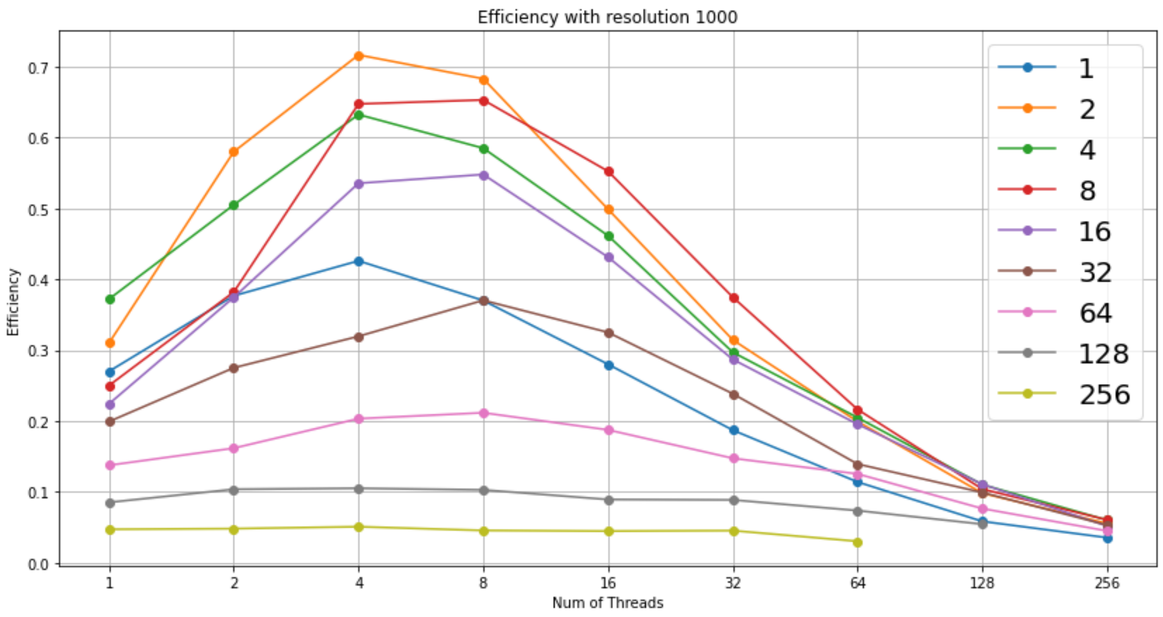


5.1.3 The Efficiency

The efficiency of the MPI+OMP experiment follows a similar pattern as observed in the first two experiments. The optimal configuration for minimum execution time requires a larger number of processing units, such as threads or processes. The large number of processing units leads to less efficiency while increasing the speedup.


Efficiency with resolution 360
Efficiency
Num of Threads
1
2
4
8
16
32
64
128
256


Efficiency with resolution 720
Efficiency
Num of Threads
1
2
4
8
16
32
64
128
256


Efficiency with resolution 1000
Efficiency
Num of Threads
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
1
2
4
8
16
32
64
128
256
1
2
4
8
16
32
64
128
256

6 Parallelization with CUDA

CUDA (Compute Unified Device Architecture) is a technology developed by Nvidia to perform general-purpose computation on Nvidia GPUs. It is an extension of the C/C++ programming language and provides a programming model based on parallel threads. These threads are organized into grids and blocks, where each block contains multiple threads, and each grid contains multiple blocks.

In our CUDA implementation, we utilized a one-dimensional grid of blocks, each containing one-dimensional blocks of threads. The advantage of using CUDA is that it enables processing a large number of threads simultaneously, with a minimal overhead compared to standard Linux POSIX threads of OpenMP.

```
dim3 threadsPerBlock(1024);
dim3 numBlocks((WIDTH * HEIGHT + threadsPerBlock.x-1) / threadsPerBlock.x);
```

In the code, the number of thread blocks is proportional to the size of the image, while the number of threads within each block is fixed. This ensures that there are always enough threads to process each pixel in the image, with a small excess pixels that can be at maximum one less than the block size. For example, in the case of an image with a resolution of 1000, there are 6 million pixels, which would require 6 million CUDA threads (or slightly more) to be initialized.

```
__global__ void mandelbrotKernel(int *image, double step, double minX, double minY, int width, int height)
{
    int pos = blockIdx.x * blockDim.x + threadIdx.x;

    if (pos < width * height)
    {
        const int row = pos / width;
        const int col = pos % width;
        //inner loop with calculation on pixel (row, col)
    }
}
```

The CUDA thread is assigned its index within the image array based on its thread id within the block and the block id within the grid. If the thread does not have an index that corresponds to a pixel within the image (i.e. the index does not satisfy `pos < width * height`), the thread terminates. If the index is valid, the thread carries out the same computation as the parallelization performed using OMP and MPI. One of the key differences between CUDA and OMP/MPI lies in the number of threads initialized. In OMP, the number of processing units is limited by the number of logical cores on the computer, while in MPI, the number of processing units is limited by the overhead of starting and managing remote Linux processes. CUDA, on the other hand, can exploit the massive parallelism of graphics cards with low overhead due to their large number of processors. However, CUDA does not support the use of the C++ <complex> class, meaning the code for the inner loop had to be rewritten in the following way:

```
    for(int i=1; i<=ITERATIONS; i++)
    {
        z_square_real = z_real*z_real - z_imm*z_imm;
        z_square_imm = 2 * z_real * z_imm;
        z_real = z_square_real + c_real;
        z_imm = z_square_imm + c_imm;
        if( z_real*z_real + z_imm*z_imm >= 4){
            image[pos] = i;
            break;
        }
    }
```

To perform the computation on the GPU, memory management is the first step. We use CudaMalloc to allocate an array on the GPU's linear memory, which has the same size as the image array and stores the result of each pixel calculation. The CUDA kernel is then executed by all threads, with each thread calculating the result for one pixel and storing it in the array. The host program waits for the threads to complete and then uses CudaMemCpy to copy the result array from the GPU to the host.

6.1 Results

Since in our program the number of working threads is fixed to the number of pixels, we decided to explore how the program performs as threads are deployed in blocks of different size. Table 6.1 shows the total execution time of the CUDA program with different number of threads per block and with different image resolutions.

Execution time			
Threads x Block / Resolution	360	720	1000
1	1.794	5.175	9.33
2	1.233	2.966	5.116
4	0.944	1.862	3.005
8	0.785	1.339	1.941
16	0.699	1.046	1.421
32	0.669	0.91	1.153
64	**0.642**	0.894	**1.149**
128	0.651	0.891	1.155
256	0.67	**0.887**	1.159
512	0.654	0.907	1.153
1024	0.65	0.891	1.156

Table 1:

The CUDA program performed significantly better than the previous experiments with multihreading and multiprocesssing. Our comparison shows that a number of threads per block equal to 32 is reasonable and there is no improvement in deploying blocks with more threads.

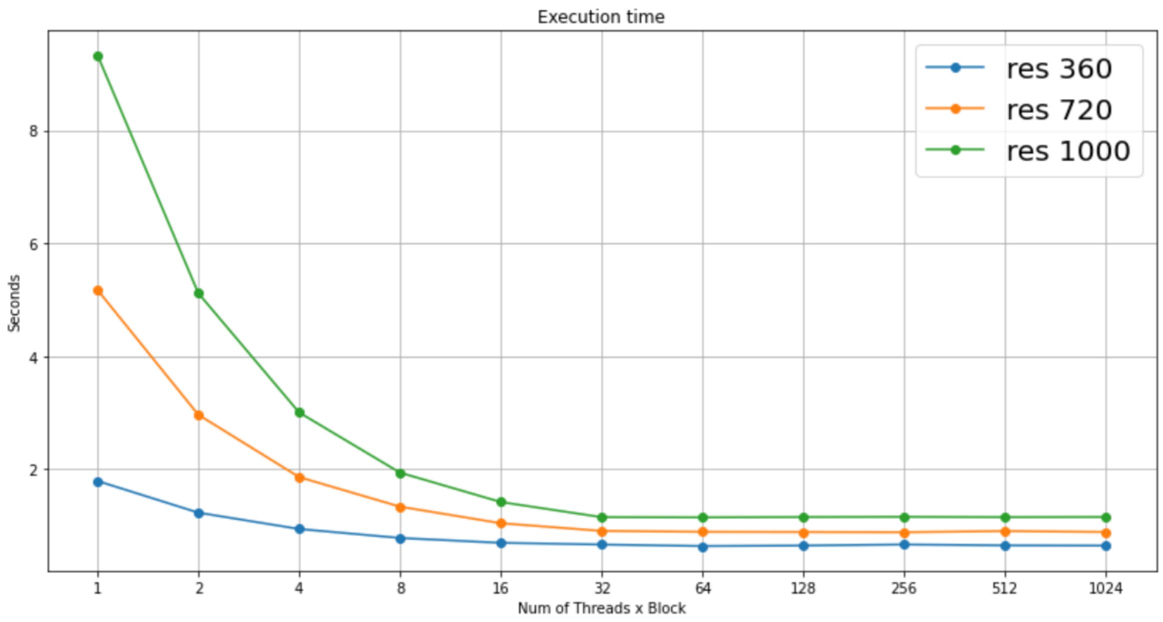


Figure 16: Execution Time

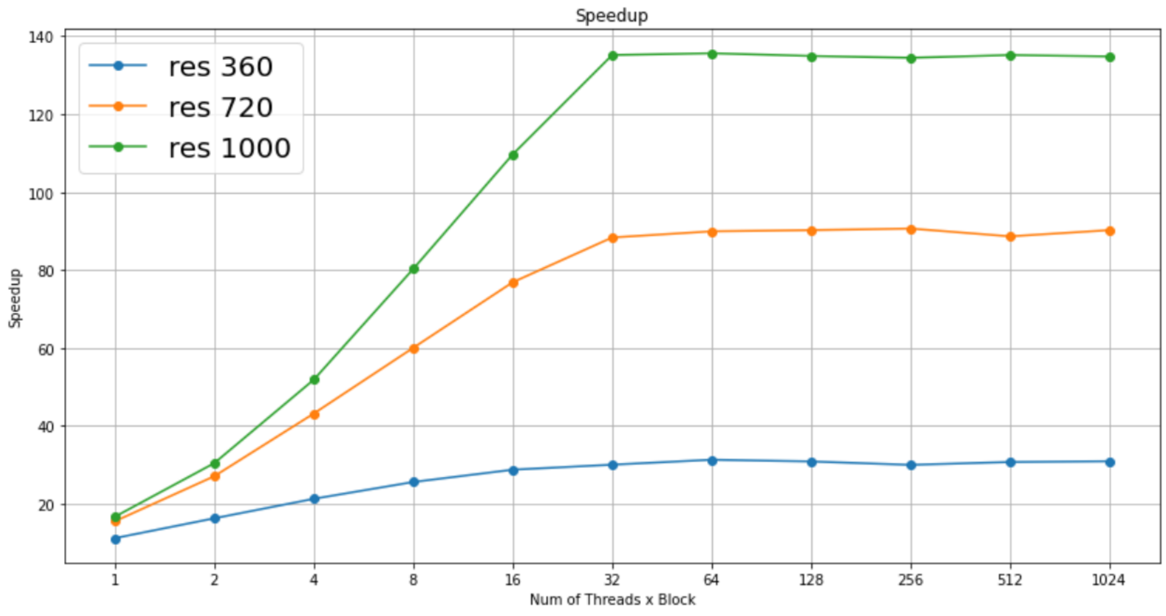


Figure 17: Speedup

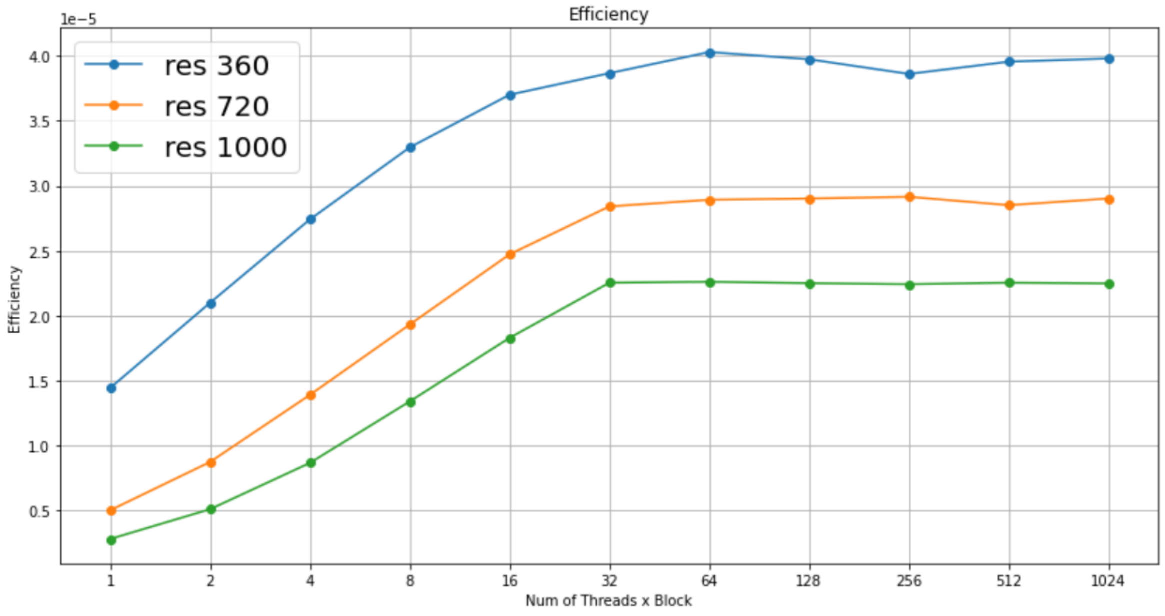
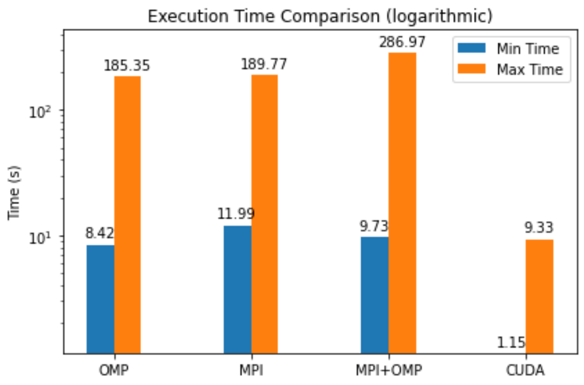


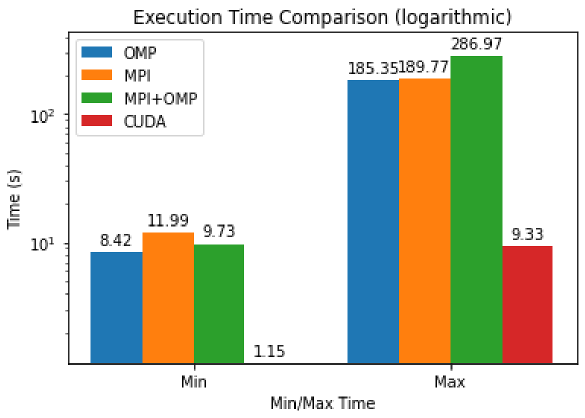
Figure 18: Efficiency

7 Conclusions: OMP vs MPI vs CUDA

Best Execution Time	
Method	Res 1000
SERIAL	155.86
OMP	8.42
MPI local	15.95
MPI distr	11.990
MPI+OMP	9.73
CUDA	1.149

As regards computing an image of the Mandelbrot set with resolution 1000, the OpenMP program exhibited a faster execution time than all MPI programs.
The OpenMP library is used to initialize Posix threads instead of new linux processes as MPI does. Since threads within the same process share resources, the operative system pays a lower overhead in creating new threads than creating new processes.
The limit of local programs parallelized with OMP lies in the number of cores of the machine: once the programmer has parallelized his problem with a number or threads equal to the number of cores in his cpu, there are no opportunities to scale the system up with OMP. The Mandelbrot program with 1000 resolution was executed faster in one machine with omp multithreading rather when the program was parallelized with a big number of processes and threads across the computers cluster. However, if the problem were bigger, for example requiring resolution 2000, then the workload of the hotspot would grow and local multithreading with OMP would not provide enough processing units. In this case of bigger problem, distributed MPI is necessary to scale up the processing units.





The barchart shows the advantages in using GPU computation rather than Multiprocessing/Multithreading for writing a Mandelbrot implementation. Even the worst possible configuation in CUDA, with one thread per block, leads to a faster execution time. If one wants to develop an advanced fractal visualization, such as a video which zooms inside the Mandelbrot set, then GPU computation is the reasonable choice to carry the calculation on each pixel of the screen.