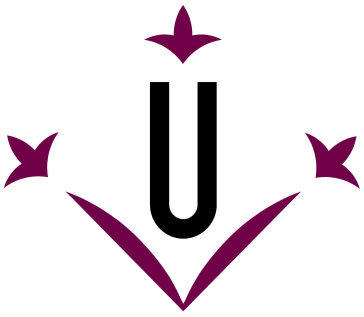
Higher Polytechnic School

Master's Degree of Computer Engineering

High Performance Computing



**Universitat de Lleida**

# OpenMP implementation
# HPC Project

---

**Members:**
Roger Truchero Visa
Gerard Donaire Alós

**Date:** May 6, 2020

# Contents

# List of Figures



# List of Tables

## Introduction

In this practice we are instructed to make an implementation of a parallel version of a popular program, the Mandelbrot set. The Mandelbrot set is a geometric figure of infinite complexity (fractal nature) obtained from a mathematical formula and a small recursive algorithm.

In order to obtain the best performance of an application implemented with a parallel programming model, we should start by optimizing the application at node-level. According to this, this activity will be focused on study the operation of the sequential version, analysing the pieces of code candidate to be parallelized following a work and data decomposition model.

Next, we will apply the OpenMP directives to parallelize the corresponding code according to the hardware and the suitable work decomposition model at node level.

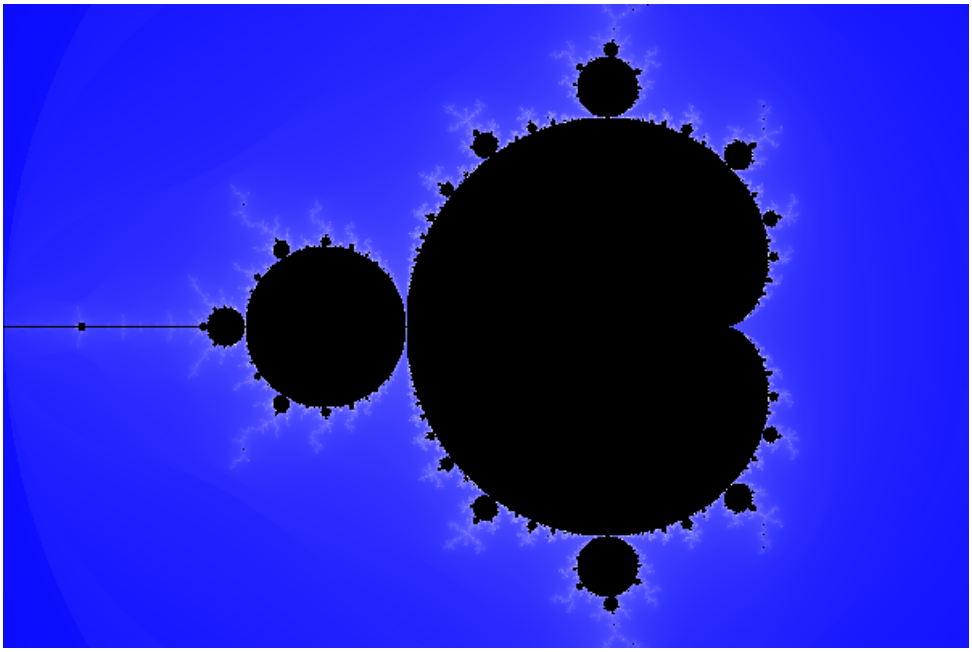


Figure 1: Mandelbrot set output representation

## Document Structure

This document will be based on the following sections:

- **Parallel design justification:** In this section we will justify the possible hotspots, bottlenecks and parallel inhibitors.
- **Selected partitioning pattern of the source matrix justification:** In this section we will justify the selected size of the partitioning, data location and loops parallelization configuration.
- **Performance checkings obtained:** In this section we will check the performance obtained using different thread scheduling policies (static, dynamic, guided) and we will discuss about the results.
- **Scalability and speedup analysis:** In this section we will analyze scalability and speedup in relation to the number of threads and size problem.
- **Experimentation configurations:** In this sectio we will explain what is the configuration that we used in order to analyse the performance and scalability of our solutions.

# 1 Parallel design justification

## 1.1 Hotspots

**Hotspots** are those **regions that can be parallelized** since they meet the necessary conditions to execute that part of the code concurrently.

In our case we have identified the following one:

- The calculation of the initial real and imaginary part of z, based on the pizel location, zoom and position values.

Since the program simply calculates the values belonging to a given set a complex number c, and represents with RGB scale values the speed with which they diverge to that point, we do not have many more regions to parallelize, although this is the most important and the one that acquires more workload.

## 1.2 Bottlenecks

A **bottleneck** is a *point of congestion in a production system (such as an assembly line or a computer network) that occurs when workloads arrive too quickly for the production process to handle.* The inefficiencies brought about by the bottleneck often create delays and higher production costs. The term "bottleneck" refers to the typical shape of a bottle, and the fact that the bottle's neck is the narrowest point, which is the most likely place for congestion to occur, slowing down the flow of liquid from the bottle.

In our case we have the following bottlenecks:

- We **can't generate the image until all the threads have been completely finished**, as if we use the matrix method to eliminate the writing problems we have to wait to write until we have all the information available.
- As we increase the iterations and/or the size of the final image, the execution time increases exponentially.

## 1.3 Parallel inhibitors

We consider **parallel inhibitors** those pieces of code that we are not able to parallel directly or we're just not interested in it because of the low performance it gives us.

In our case we can identify a very clear one which is the **writing of the image pixels**. In the sequential version it works because we only have one execution thread that executes the code in a linear and ordered way, in the parallel version when we have more than one execution thread we cannot guarantee that the threads are writing the values ordered by the standard output.

Using synchronization mechanisms would be silly because if what we are looking for is to parallelize our code what we would get is to do it sequentially.

# 2 Selected partitioning pattern of the source matrix justification

## 2.1 Size of tasks

The size of the tasks to be performed for each one will vary according to the total number of execution threads and the schedule they use (dynamic, static and guided). The total volume of work will be defined by the following expression:

$$\frac{width * height * maxiterations}{number\_of\_threads} \tag{1}$$

This expression represents the complexity of the for loop, where the matrix values are calculated for each pixel of the image until a value is found that is outside the circle or in the worst case we reach maxiterations.

## 2.2 Data location

As we mentioned in the previous section, one of the main problems we encountered was the order in which the threads wrote in the standard output. To solve this we used the following data structure type and the corresponding memory allocation:

```
typedef unsigned char pixel_t[3];
pixel_t *pixels = malloc(sizeof(pixel_t)*H*W);
```

This structure will allow us to share the information with all the threads and have them write in the corresponding position, thus solving the problem of order and being able to correctly generate the final image resulting from the calculations. Outside the parallel region we will write the data of the matrix in the corresponding .ppm file in a sequential way.

In the code we have chosen to perform a parallel region in the loop for so that each thread executes a copy of the code within the structured block. We made a matrix that represents a shared variable that has the same address space in the execution context of every thread. In order to coordinate access to these shared variable across multiple threads (synchronization) we initialized the array before the parallel region because for definition any variable that existed before a parallel region still exist inside and are shared by default between all threads. We have also a private variable for all threads that is the index variable of the worksharing loop because is automatically made private.

```
for(y = 0; y < H; y++)
    {
        for(x = 0; x < W; x++)
        {
            /*  calculate the initial real and imaginary part of z,
            based on the pixel location and zoom and position values */
            pr = 1.5 * (x - W / 2) / (0.5 * zoom * W) + moveX;
            pi = (y - H / 2) / (0.5 * zoom * H) + moveY;
            newRe = newIm = oldRe = oldIm = 0; /* these should start at 0,0 */

            int i; /* "i" will represent the number of iterations */

            /* start the iteration process */
            for(i = 0; i < MAXITER; i++)
            {
                /* remember value of previous iteration */
                oldRe = newRe;
                oldIm = newIm;

                /* the actual iteration, the real and imaginary part are calculated */
                newRe = oldRe * oldRe - oldIm * oldIm + pr;
                newIm = 2 * oldRe * oldIm + pi;

                /* if the point is outside the circle with radius 2: stop */
                if((newRe * newRe + newIm * newIm) > 4) break;
            }

            if(i == MAXITER)
            {
                pixels[y*W + x][0] = 0;
                pixels[y*W + x][1] = 0;
                pixels[y*W + x][2] = 0;
            }
            else
            {
                double z = sqrt(newRe * newRe + newIm * newIm);
                int brightness = 256 * log2(1.75 + i - log2(log2(z))) / log2((double)MAXITER);
                pixels[y*W + x][0] = brightness;
                pixels[y*W + x][1] = brightness;
                pixels[y*W + x][2] = 255;
            }
        }
    }
```

# 3 Performance checkings obtained

## 3.1 Schedule methods

In order to analyze the results and parallelize the loop with OpenMP, we are going to use the following scheduling types: **static**, **dynamic** and **guided**.

The schedule(**static**, chunk-size) clause of the loop construct specifies that the for loop has the static scheduling type. OpenMP divides the iterations into chunks of size chunk-size and it distributes the chunks to threads in a circular order. **When no chunk-size is specified, OpenMP divides iterations into chunks that are approximately equal in size and it distributes at most one chunk to each thread**.

The schedule(**dynamic**, chunk-size) clause of the loop construct specifies that the for loop has the dynamic scheduling type. **OpenMP divides the iterations into chunks of size chunk-size**. Each thread executes a chunk of iterations and then requests another chunk until there are no more chunks available. There is no particular order in which the chunks are distributed to the threads. The order changes each time when we execute the for loop. If we do not specify chunk-size, it defaults to one.

The **guided** scheduling type is similar to the dynamic scheduling type. **OpenMP again divides the iterations into chunks. Each thread executes a chunk of iterations and then requests another chunk until there are no more chunks available**. The **difference** with the **dynamic scheduling** type is in the **size of chunks**. The size of a chunk is proportional to the number of unassigned iterations divided by the number of the threads. Therefore the size of the chunks decreases.

## 3.2 What we expect?

We expect that static scheduling type will be more appropriate when all iterations will have the same computational cost.

Furthermore, we think that dynamic scheduling type will be more suitable when the iterations will require different computational costs. The dynamic scheduling type has higher overhead than the static scheduling type because it dynamically distributes the iterations during the runtime.

Finally we hope that the guided scheduling type will be more coherent when the iterations will be poorly balanced between each other. So, we think that this scheduling type is especially appropriate when poor load balancing occurs toward the end of the computation.

## 3.3 Results

In order to analyse the performance and scalability of our solutions we experiment the configurations that is shown below:

- **Number of threads:** 1, 2, 4, 8, 16.
- **Number of iterations:** $10^4$, $10^5$, $10^6$.
- **Image size:** 600x400, 3000x2000, 6000x4000, 30000x20000.

In the following tables we are going to observe the results obtained with the configurations that we shown before. We will discuss about this results in order to compare if the expectations we had hoped for have been fulfilled or not.

| | | ITERATIONS | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $10^4$ | | | | $10^5$ | | | | $10^6$ | | | |
| Dynamic | | 600x400 | 3000x2000 | 6000x4000 | 30000x20000 | 600x400 | 3000x2000 | 6000x4000 | 30000x20000 | 600x400 | 3000x2000 | 6000x4000 | 30000x20000 |
| THREADS | Serial | 4,58 | 113,51 | 453,55 | 11337,24 | 44,88 | 1122,9 | 4487,22 | 112290 | 448,55 | 11207,4 | 44855 | 1120743 |
| | 1 | 4,53966 | 113,219 | 452,68 | 11269,5 | 44,8935 | 1121,46 | 4487,66 | 111626 | 454,98 | 11234,6 | 45598,32 | 1121259 |
| | 2 | 2,33643 | 58,2372 | 232,866 | 5796,76 | 23,786 | 576,761 | 2372,89 | 57409,1 | 237,21 | 5790,83 | 23034,12 | 576402 |
| | 4 | 1,20565 | 30,0213 | 120,048 | 2988,23 | 11,9235 | 297,239 | 1188,88 | 29586,3 | 118,867 | 2989,98 | 11878.6 | 297614 |
| | 8 | 1,20718 | 30,0207 | 120,045 | 2988,17 | 11,9012 | 297,355 | 1189,11 | 29597,8 | 118,873 | 2988,88 | 11879,8 | 297504 |
| | 16 | 1,20711 | 30,0219 | 120,064 | 2988,29 | 11,9072 | 297,243 | 1189,07 | 29586,7 | 118,898 | 2988,32 | 11878,9 | 297448 |

Table 1: Execution table for each features with dynamic schedule

| | | ITERATIONS | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $10^4$ | | | | $10^5$ | | | | $10^6$ | | | |
| Static | | 600x400 | 3000x2000 | 6000x4000 | 30000x20000 | 600x400 | 3000x2000 | 6000x4000 | 30000x20000 | 600x400 | 3000x2000 | 6000x4000 | 30000x20000 |
| THREADS | Serial | 4,58 | 113,51 | 453,55 | 11337,24 | 44,88 | 1122,9 | 4487,22 | 112290 | 448,55 | 11207,4 | 44855 | 1120743 |
| | 1 | 4,53747 | 113,202 | 452,878 | 11337,2 | 44,91 | 1123,88 | 4483,28 | 111826 | 448,339 | 11214 | 44739,8 | 1115793 |
| | 2 | 2,35368 | 58,3181 | 232,999 | 5802,65 | 23,2643 | 577,615 | 2309,97 | 57472,6 | 232,457 | 5775,21 | 23084,7 | 574847 |
| | 4 | 1,9576 | 48,596 | 194,341 | 4835,3 | 19,4623 | 483,505 | 1932,52 | 48108,7 | 194,35 | 4832,73 | 19345,6 | 480856 |
| | 8 | 1,39798 | 34,9895 | 139,743 | 3481,45 | 13,9006 | 348,209 | 1388,24 | 34646,7 | 138,79 | 3470,27 | 13872,2 | 345420 |
| | 16 | 1,26204 | 31,1767 | 122,693 | 3074,88 | 12,3168 | 305,965 | 1220,29 | 30443,5 | 123,159 | 3055,09 | 12198,6 | 304094 |

Table 2: Execution table for each features with static schedule

| | | ITERATIONS | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $10^4$ | | | | $10^5$ | | | | $10^6$ | | | |
| Guided | | 600x400 | 3000x2000 | 6000x4000 | 30000x20000 | 600x400 | 3000x2000 | 6000x4000 | 30000x20000 | 600x400 | 3000x2000 | 6000x4000 | 30000x20000 |
| THREADS | Serial | 4,58 | 113,51 | 453,55 | 11337,24 | 44,88 | 1122,9 | 4487,22 | 112290 | 448,55 | 11207,4 | 44855 | 1120743 |
| | 1 | 4,53892 | 113,194 | 452,728 | 11267 | 44,8843 | 1121,73 | 4482,39 | 111653 | 448,231 | 11135,2 | 44693,3 | 1108365 |
| | 2 | 2,34844 | 58,2316 | 233,757 | 5796,2 | 23,236 | 576,774 | 2306,88 | 57410,4 | 232,081 | 5989,21 | 23052,1 | 596148 |
| | 4 | 1,43357 | 35,2935 | 140,827 | 3513,01 | 14,2522 | 351,475 | 1402,96 | 34984,8 | 142,389 | 3524,87 | 14026,7 | 350855 |
| | 8 | 1,28564 | 31,808 | 127,049 | 3166,07 | 12,6935 | 316,559 | 1259,75 | 31509,4 | 126,591 | 3165,23 | 12586,3 | 315057 |
| | 16 | 1,23716 | 30 | 121.372 | 3021,03 | 11,9876 | 299,24 | 1198,64 | 29785,5 | 119,796 | 3021,44 | 11986,1 | 300745 |

Table 3: Execution table for each features with guided schedule

In these tables that are shown before we can see the different configurations that we used in order to do the analysis about the performance for serial and the different scheduling types for the parallelization of loops: dynamic, guided and static.

We can see that the different scheduling types (**dynamic, static and guided**) have **better** execution results than **serial** when we increase the **number of threads**. For one thread the difference between the execution results of being applied serial or with the parallelization scheduling types (dynamic, static and guided) are very slight. But as we increase the number of threads, we get better results in the scheduling types. The reason why we are getting better results is because as we increase the number of threads we are distributing the work among them. With the serial we only have one thread so all the work has to be done by the same thread.

In the **static schedule we haven't specify the chunk-size** in order to do the analysis so OpenMP has divided iterations into chunks that are approximately equal in size and has distributed it at most one chunk to each thread. For the **dynamic** and **guided** schedule we use the **size-chunk of one**.

Next, we will discuss the results obtained from the different scheduling types and compare the results to see which one is the most suitable depending on the configuration we put.

Overall, (regardless of whether we do $10^4$, $10^5$, $10^6$ iterations with 600x400, 6000x4000, 3000x2000 or 30000x20000) we can see that dynamic schedule is the one with the best execution results.

Furthermore, we can see that guided schedule has better execution results than static schedule but it's a little worse than dynamic schedule. Therefore, in this case, the worst execution results that we have obtained of the three scheduling types, in general, has been the static schedule.

The dynamic scheduling type is appropriate when the iterations require different computational costs. This means that the iterations are poorly balanced between each other. The dynamic scheduling type has higher overhead then the static scheduling type because it dynamically distributes the iterations during the runtime.

The reason why the best of all schedules is the dynamic is because of the coding of the problem. The innermost loop of the value calculation has a variable name of iterations, since it will depend if the point that we are calculating is outside the circle with radius 2, then we will be able to leave the loop and consequently we will not make the same name of iterations at every step.

# 4 Scalability and speedup analysis

To make the following graphics of both speedup and efficiency we have created a python script that takes care of making the calculations and represent them in 2d graphics using the matploltib libraries. This code will be reusable for other assignments.

**Obtain speedup and efficiency**

In [344]:
```
threads = [1,2,4,8,16]
```

In [345]:
```
def get_speedup(filename):
    data = [[float(n) for n in line.strip().split('\t')] for line in open(filename, 'r').readlines()]
    serial = data[0]
    data = data[1:]
    speedup = list(zip(*[list(map(truediv, serial, row)) for row in data]))[1:]

    return speedup

def get_efficiency(speedup):
    return [list(map(truediv, row, threads)) for row in speedup]
```

**Plot speedup and efficiency for each schedule**

In [346]:
```
legend = ["size: 600x400 iterations: 10⁴", "size: 3000x2000 iterations: 10⁴",
          "size: 6000x4000 iterations: 10⁴", "size: 30000x20000 iterations: 10⁴",
          "size: 600x400 iterations: 10⁵", "size: 3000x2000 iterations: 10⁵",
          "size: 6000x4000 iterations: 10⁵", "size: 30000x20000 iterations: 10⁵",
          "size: 600x400 iterations: 10⁶", "size: 3000x2000 iterations: 10⁶",
          "size: 6000x4000 iterations: 10⁶", "size: 30000x20000 iterations: 10⁶"]
```

In [422]:
```
def plot_speedup(speedup):
    %matplotlib inline

    plt.figure(figsize=(30,30))

    for i in range(0, len(speedup)):
        plt.plot(threads, speedup[i], label='$y = {i}x + {i}$'.format(i=i))

    plt.xlim(1, 16)
    plt.xlabel('Number of threads', fontsize=30)
    plt.ylim(1.0, 5)
    plt.ylabel('Speedup', fontsize=30)
    plt.legend(legend, loc='lower right', prop={'size': 30})
    plt.show()

def plot_efficiency(efficiency):
    %matplotlib inline

    plt.figure(figsize=(30,30))

    for i in range(0, len(efficiency)):
        plt.plot(threads, efficiency[i], label='$y = {i}x + {i}$'.format(i=i))

    plt.xlim(1, 16)
    plt.xlabel('Number of threads', fontsize=30)
    plt.ylim(0.0, 1.0)
    plt.ylabel('Efficiency', fontsize=30)
    plt.legend(legend, loc='upper right', prop={'size': 30})
    plt.show()
```

Figure 2: Speedup and effciency graphic generation code

## 4.1 Dynamic schedule

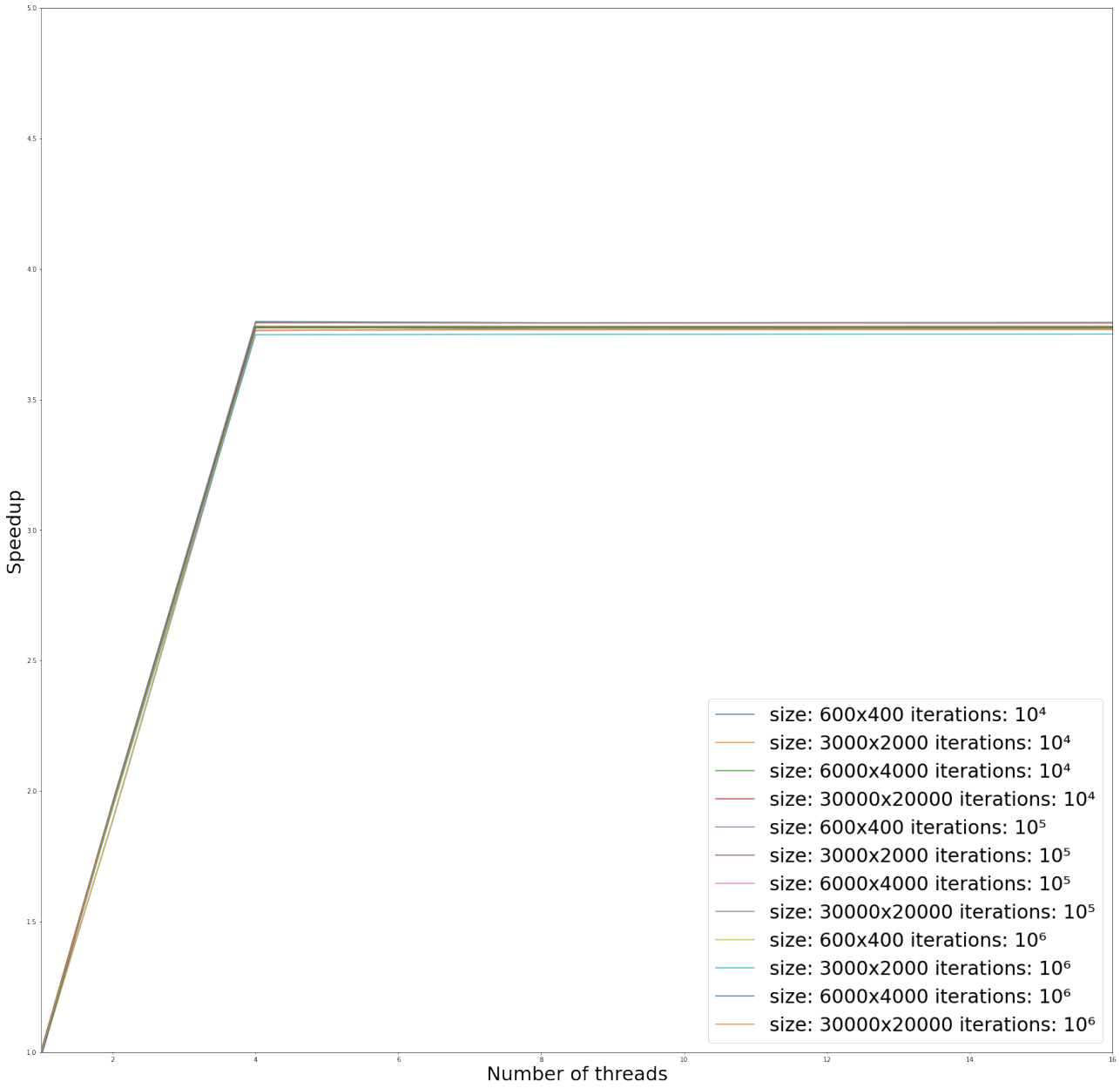



Figure 3: Speedup graphic with dynamic schedule

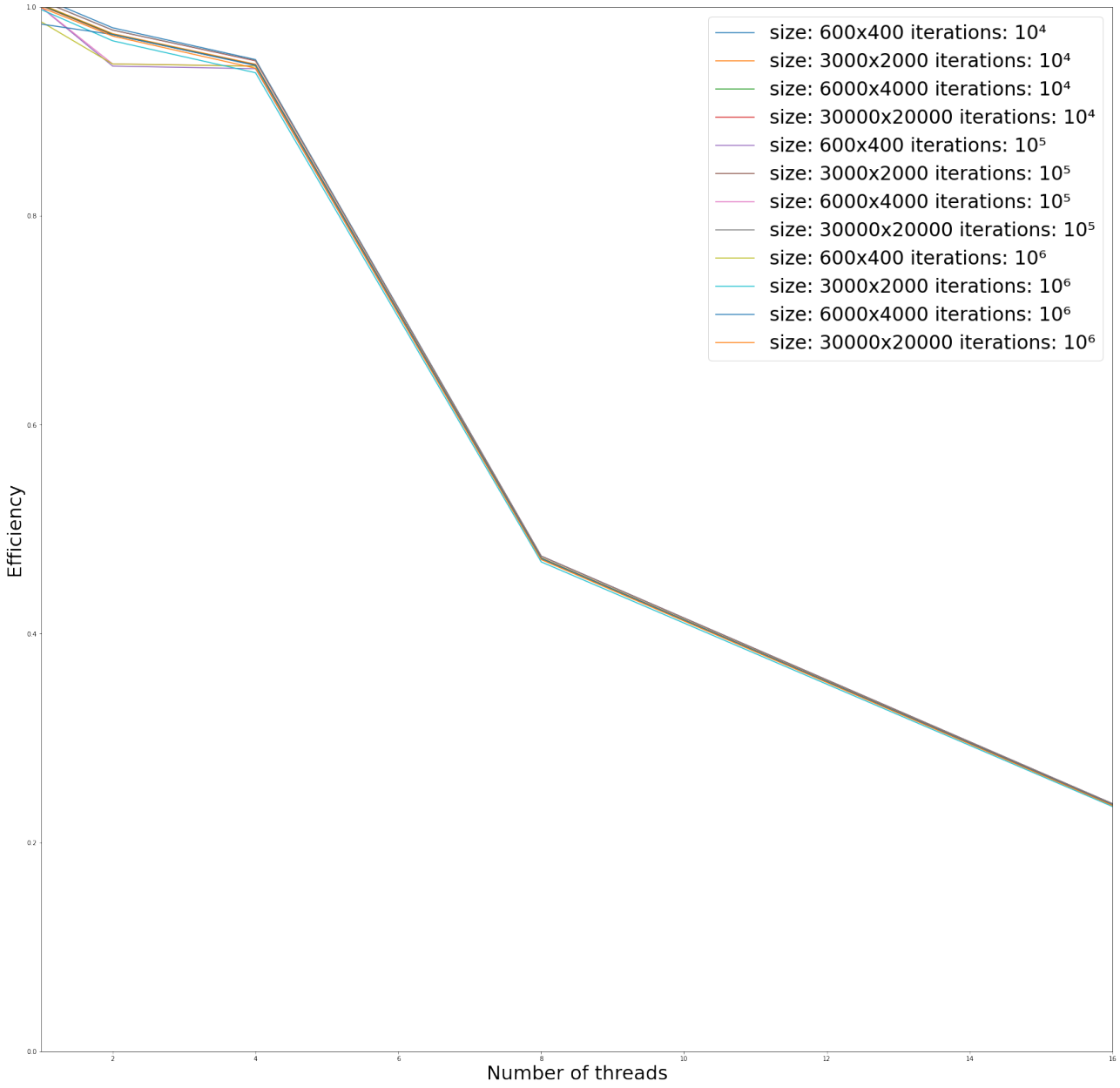



Figure 4: Efficiency graphic with dynamic schedule

## 4.2 Static schedule

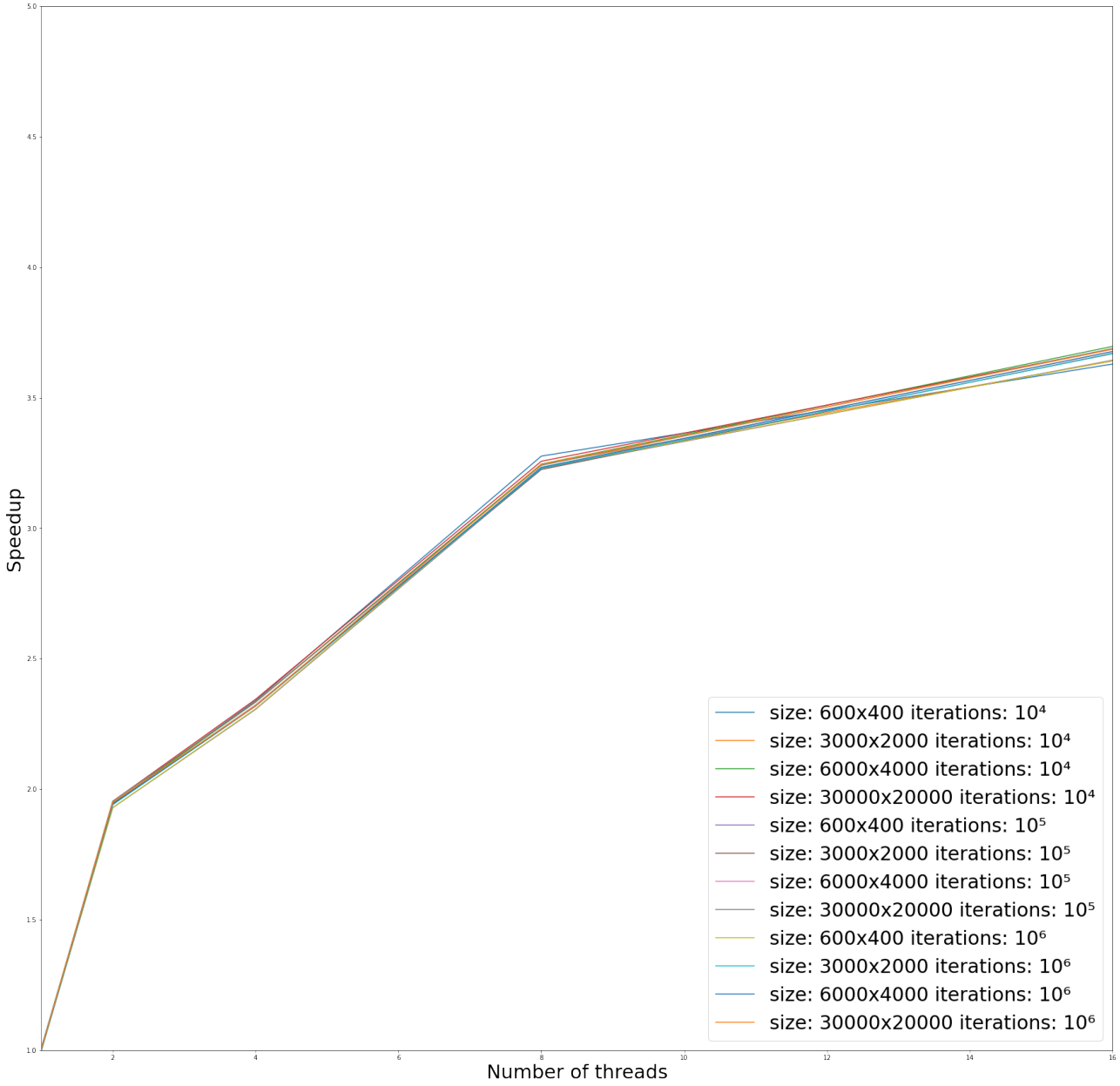



Figure 5: Speedup graphic with static schedule

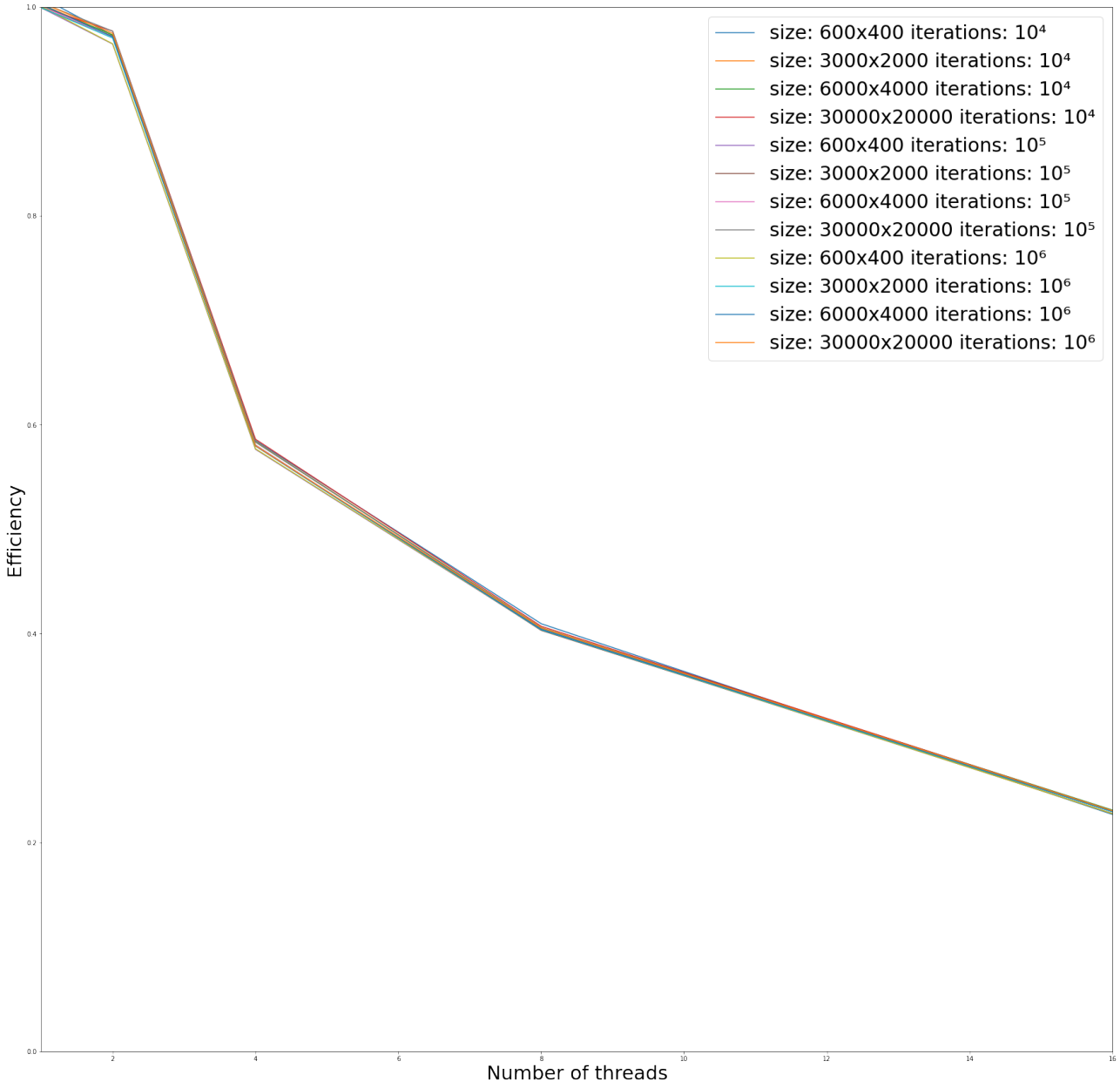



Figure 6: Efficiency graphic with static schedule

## 4.3 Guided schedule

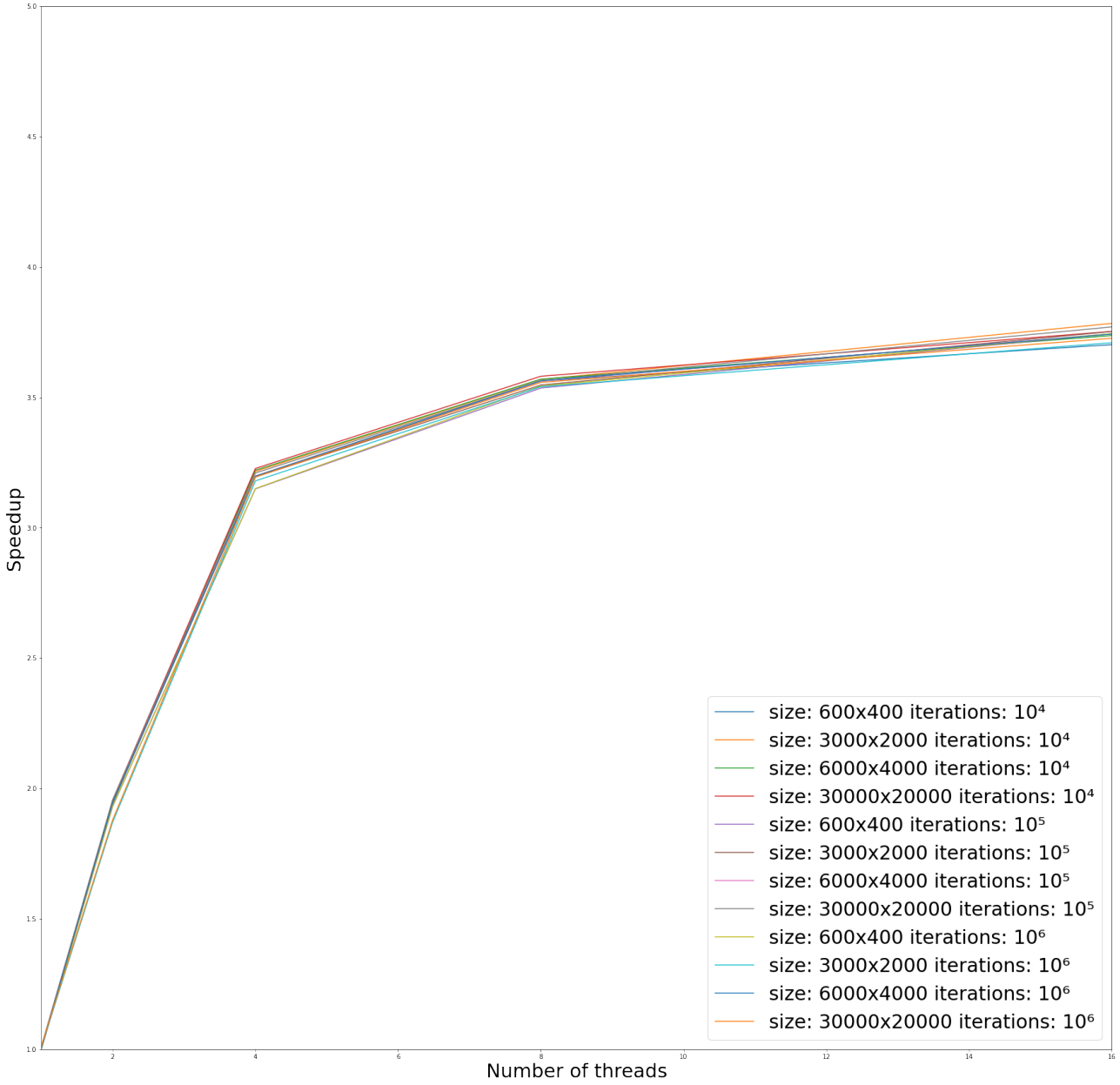



Figure 7: Speedup graphic with guided schedule

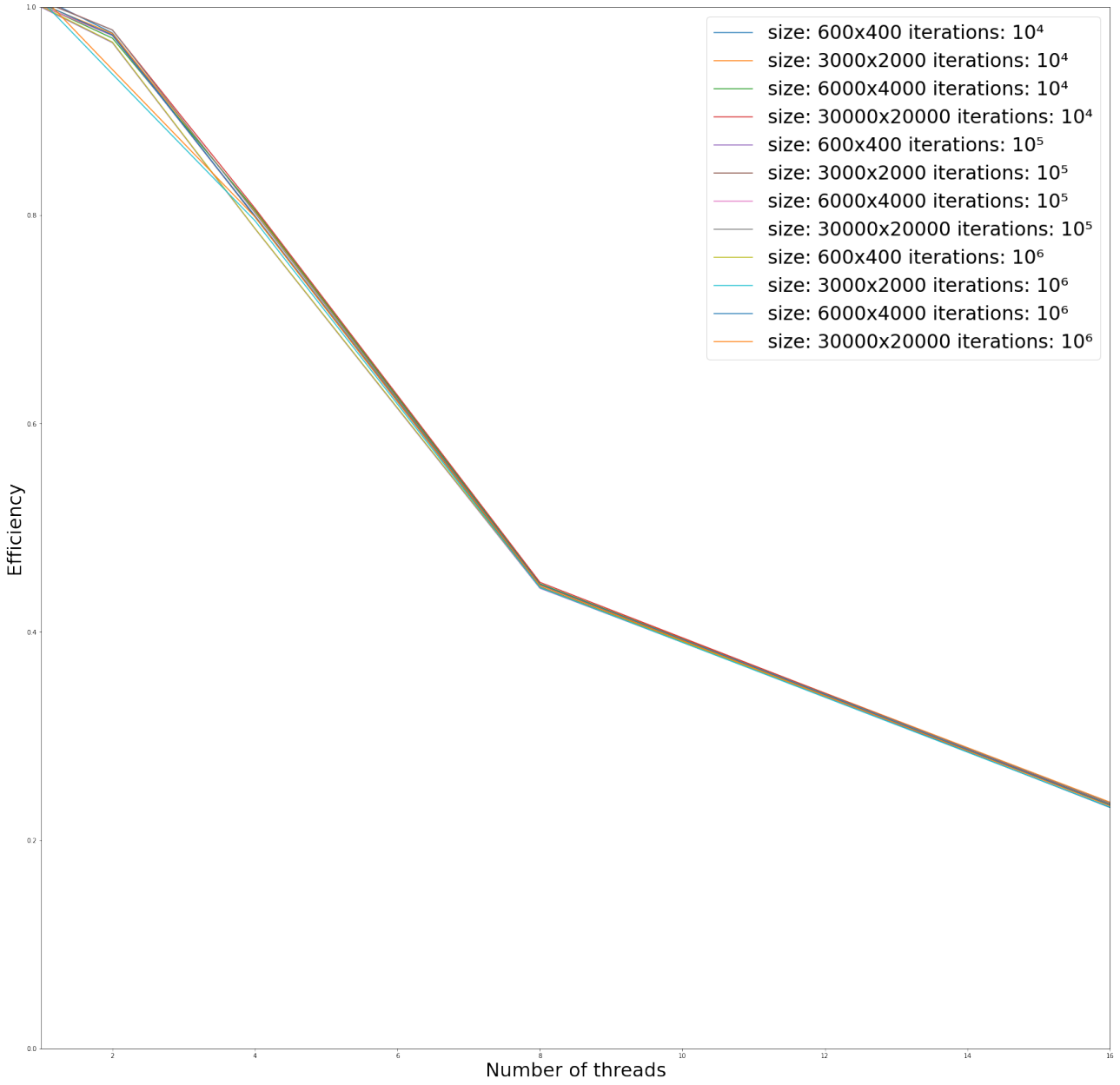



Figure 8: Efficiency graphic with guided schedule

As it is the case in all the different executions, the ones that have a worse speedup and consequently a higher efficiency are the ones that make a bigger number of calculations, as it is the case of the operations with $10^6$ iterations and image sizes of 6000x4000 and 30000x20000.

We can also appreciate how for the dynamic schedule the progression of the speedup from 4 threads remains constant while in the other schedules you can appreciate an exponential growth until reaching the maximum speedup for given the characteristics of our computing nodes.

We can also observe how in the dynamic schedule the speedup makes a more abrupt growth and then it is maintained while in the other two schedules we see a more progressive growth until reaching the same speedup as in the dynamic one. In other words, the schedule dynamic reaches its maximum speedup faster than the other two schedules.

Regarding the efficiency we can see how as we increase the number of threads this decreases considerably given the characteristics of our cluster with a value that in the infinite tends to 0.

# 5 Bibliography

## References

[1] *OpenMP: For Scheduling*, *Consultation time: 5:05 p.m. on April 26, 2020* Available at: http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

[2] *OpenMP: Bottleneck Analysis*, *Consultation time: 7:21 p.m. on April 27, 2020* Available at: https://www.hpcwire.com/2014/02/13/unleashing-potential-openmp-via-bottleneck-analysis/

[3] *OpenMP Forum*, *Consultation time: 9:21 p.m. on April 29, 2020* Available at: http://forum.openmp.org/forum/

[4] *Introduction to OpenMP*, *Consultation time: 9:21 p.m. on April 29, 2020* Available at: http://ocw.uc3m.es/ingenieria-informatica/arquitectura-de-computadores-ii/otros-recursos-1/or-f-008.-curso-de-openmp

[5] *OpenMP: Observations*, *Consultation time: 9:21 p.m. on April 29, 2020* Available at: https://riptutorial.com/es/openmp