PC-2019/20 Course Project: Parallel Image Reader and 2D Pattern Recognition Asynchronous Execution

Alberto Baldrati
alberto.baldrati@stud.unifi.it

Abstract

In this paper we focus on multi threading and performance comparison between asynchronous and synchronous execution. Firstly we implement a simple Image Reader and later we try to exploit the power of asynchronous execution chaining together the Image Reader and the 2D Pattern Recognition algorithm. The focuses of this paper are therefore to analyze the performance improvement obtainable with a multi threading version compared to a sequential one and to establish whether an asynchronous execution can improve performance reducing idle time

1. Introduction

In the first part of this paper we are going to develop a simple Image Reader which reads all the images in a directory. For reading the images we have used OpenCV library [2], so the images are in Mat classes which are the OpenCV basic image container.

Later, with the aid of this Image Reader, we have applied the 2D Pattern Recognition algorithm on the read images. Such concatenation of operations has been conducted in synchronous and asynchronous manner for enabling a comparison between implementations and understand which implementation perform better.

1.1. Technologies

The entire code is written in C++ but before moving forward is necessary to do some clarifications on the libraries and technologies used:

- We have used the **C++17** standard which includes *Filesystem* library that allows us to manage files without relying on external libraries. For example the use of this library permits to the Image Reader to iterate over the input directory
- As stated before for reading (and later writing on) the images **OpenCV** library [2] has been used
- We do not have used the thread library included in the C++ standard due to its limitation in asynchronous execution. To overcome this limits we have used **Boost** library [3] which provide useful feature such as: *then*operation on future which ensure us a total asynchronous and *thread pool* which allow us to chose the number of threads we want to start.
- For enabling the parallelism in the synchronous implementation of the algorithm as well, **OpenMP** [5] library has been used

2. Image Reader

In this section we provide the pseudocode of our Image Reader implementation.
In **Algorithm 1** the **synchronous** implementation which uses OpenMP is reported, it is rather straight forward but it has some point of interest like:

- We use **static schedule**, in fact the workload is known a priori and there is no need of a dynamic schedule which would introduce useless overhead
- For testing purposes we can set the number of threads we want
- We have to introduce a **critical section** through a second #pragma directive since the *std::vector* we use to store the images is not thread safe.

In **Algorithm 2** the pseudocode of **asynchronous** implementation is reported, it is made of two cycle: in the first one we launch an asynchronous thread for each image (we use as launch policy a *ThreadPool* so we can control the maximum number of thread simultaneously in execution) and we store each return value (i.e. a future) in a vector. The second cycle (highlighted in yellow) is the cycle where from the futures we get the images waiting the termination of every threads. It's important to underline that this highlighted cycle will not be present later (section 4 on page 3) when we will refer to the Algorithm 2 since this cycle is blocking, so to have a full asynchronous execution we will work directly on futures.

Algorithm 1: *Synchronous* ImageReader

Data: inputDir, numThreads Nt
Result: imagesVector

```
#pragma omp parallel for schedule(static)
  num_threads(numThreads)
for imageName in inputDir do
    image = cv::imread(imageName)
    #pragma omp critical
    imagesVector.push_back((image,imageName))
return imagesVector
```

Algorithm 2: *Asynchronous* ImageReader

Data: inputDir, ThreadPool Tp
Result: imagesVector

```
futImagesVector futIV = []
for imageName in inputDir do
    lambdaImage =
      []{return cv::imread(imageName)}
    futIV.pus_back(
      (boost::async(Tp,lambdImage)),imageName)
for futImage in futIV do
    ImagesVector.push_back(
      (futImage[0].get(), futImage[1]))
return imagesVector
```

3. Pattern Recognition

The Pattern Recognition technique consists in individuating a specific query in a data target, therefore we want to find the closest portion to our query in our target, in this case we focus on 2D Pattern Recognition due to the nature of our data (i.e. matrices). For the sake of completeness below is reported the pseudocode of the algorithm and in **figure 1** an example of a trivial execution is shown.
See [1] for more information about the algorithm.

Algorithm 3: computeSAD

Data: queryMatrix Q, targetMatrix T,
  startRowIndex i, startColIndex j
Result: localSadValue

```
localSadValue = 0
for from k = 0 to Q.rows do
    for from l = 0 to Q.cols do
        targetV = T[i+k][j+l]
        queryV = Q[k][l]
        localSadValue += | targetV - queryV |
return localSadValue
```

Algorithm 4: PatternRecognition

Data: queryMatrix Q, targetMatrix T
Result: SADMatrix S

```
Define SADMatrix S
S.rows = T.rows - Q.rows + 1
S.cols = T.cols - Q.cols + 1
for from i = 0 to S.rows do
    for from j = 0 to S.cols do
        S[i][j] = computeSAD(P,Q,i,j)
return cx, cy = argmin(S)
```

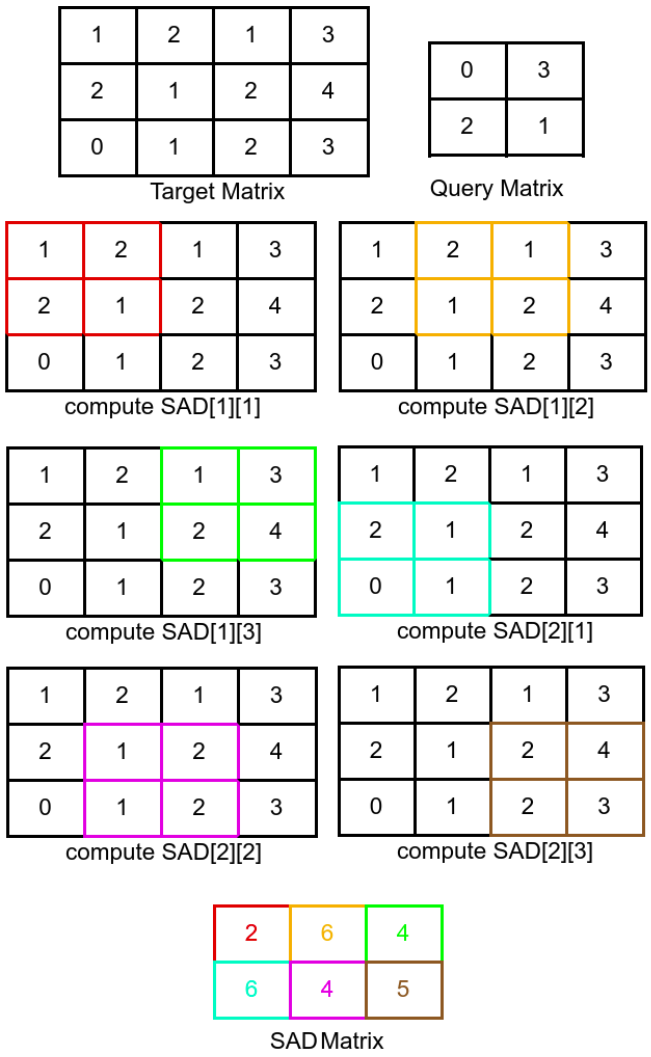

Figure 1. Trivial example of Pattern Recognition algorithm

4. Full Execution

Now we are going to compare the asynchronous and synchronous execution of the complete program (to whom we will refer with the name **Full Execution**), such program chains together the Image Reader and the 2D Pattern Recognition. More Specifically the first part reads all the images from a known directory (as specified in **section 2 on page 1**) and the second one applies the Pattern Recognition algorithm, with a random query matrix, to each image (as specified in **section 3 on the previous page**). Moreover after identifying the closest portion of the images to the query, we draw a rectangle around this portion (an example is shown in **figure 2**) and we save this highlighted images in an output folder.

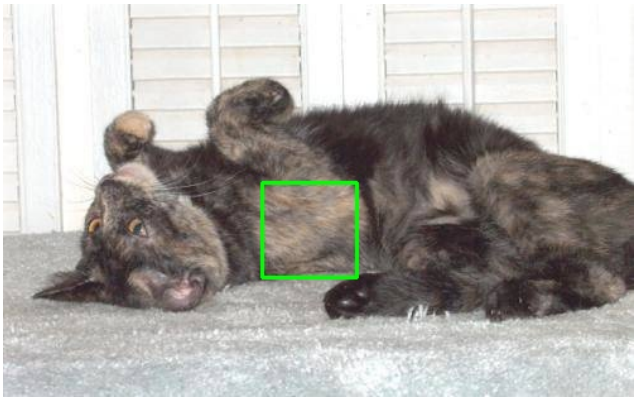

Figure 2. Output of Full Execution process

4.1. Synchronous execution

In **Algorithm 5** the pseudocode of the parallel **synchronous** execution in reported, also there the OpenMP library has been used to enable parallel execution.

Algorithm 5: *Synchronous* Execution

```
   Data: inputDir, numThreads Nt, queryMatrix Q,
         outputDir
1  imagesVector = SyncImageReader(inputDir,Nt)
2  #pragma omp parallel for schedule(static)
     num_threads(numThreads)
3  for image, imageName in imagesVector do
4  |   cx, cy = PatternRecognition(Q, image)
5  |   topLeftPoint tlp = (cx,cy)
6  |   bottomRightPoint brp =
   |     (cx + Q.cols, cy + Q.rows)
7  |   cv::rectangle(image, tlp, brp, green)
8  |   cv::imwrite(outputDir + imageName, image)
```

4.2. Asynchronous execution

In **Algorithm 6** the pseudocode of **asynchronous** execution is reported, for managing the the asynchronous parallelism the Boost library has been used.

Algorithm 6: *Asynchronous* Execution

```
   Data: inputDir, ThreadPool Tp, queryMatrix Q,
         outputDir
1  futImagesVector = AsyncImageReader(inputDir,Tp)
2  futuresTask = []
3  for futImage, imageName in futImagesVector do
4  |   futuresTask.push_back(
   |     futImage.then(Tp,[Q](futImage){
5  |       image = futImage.get()
6  |       cx, cy = PatternRecognition(Q, image)
7  |       topLeftPoint tlp = (cx,cy)
8  |       bottomRightPoint brp =
9  |           (cx + Q.cols, cy + Q.rows)
10 |       cv::rectangle(image, tlp, brp, green)
11 |       return image, ImageName}
12 |       ).then(Tp,[outputDir](futOutput){
13 |           image, imageName = futOutput.get()
14 |           cv::imwrite(outputDir +
15 |               imageName, image) }))
16 for futTask in futuresTask do
17 |   futTask.wait();
```

This asynchronous execution is less straightforward and we can emphasize few steps:

- We need the *futuresTask* vector since at the end of the Algorithm we need to wait that all images have been processed (the cycle beginning at line **16** does that)
- In line **4** and in line **12** we can see that the execution is full asynchronous, in fact the function **then** applied to the futures is executed only when the thread associated to it is completed reducing idle time. Thanks to that we can completely decouple the task **I/O** bound and the task **CPU** bound.
- For what said above the function **get** applied to the futures in line **5** and in line **13** is not blocking

5. Experimental Result

Several experiments have been carried out to compare execution time and SpeedUp of the synchronous version written in C++ using OpenMP with the asynchronous version also written in C++ using the Boost thread library.

The SpeedUp S_P is calculated as

$$S_P = \frac{t_s}{t_p}$$

The tests have been conducted on an Ubuntu 18.04 LTS machine equipped with:

- Intel Core i7-4790 3.6GHz with Turbo Boost up to 4Ghz, 4 core/8 thread processor
- RAM 16 GB DDR4
- Western Digital Blue 1TB Hard Disk 7200rpm

The high precision C++11 library *chrono* has been used for measuring the execution time.
Each time has been measured running each test 5 times and taking the average as a result, in this section we provide mostly graphs for a better understanding of results, in **section 7 on page 7** are available the complete numerical results.
In (most of) the graphs we can identify three sections, which correspond to the three different colors:

- **Blue** section where each thread can be executed on a different **physical core**
- **Yellow** section where each thread can be executed on a different **logical core**
- **Red** section where threads have to **share** both physical and logical core

5.1. Dataset

All the images used for the experimental results are taken from **dogs vs cat Kaggle** competition [4]. This dataset is made of **37,5K** (cats and dogs) images and has a size of **850 MB**. It is naturally split in 25K images for the training set and 12,5K for the test set, however for our interests each image has the same value.
In our experiments we will use all this dataset or a portion of it depending on our purposes and the expected execution time.

5.2. Image Reader Results

The following graphs show the results of our Image Reader implementations, in each test we have used a number of threads from 1 (sequential version) to 10.
The images used for each of the following test on the Image Reader are the 25K images contained in the training set, the combined size of the images is about 582 MB.

In **figure 3** (**synchronous** implementation) and **figure 4** (**aynchronous** implementation) we can can notice that in each area we have a different SpeedUp evolution, in the blue area it is almost **linear**, in the yellow area is **sub-linear** and in the red area we have no more SpeedUp. Considering all areas together we can define our SpeedUp as *SpeedUp with an optimal number of processors*, this number in our case is 8 which is the number of logical core in the machine we used for our tests.

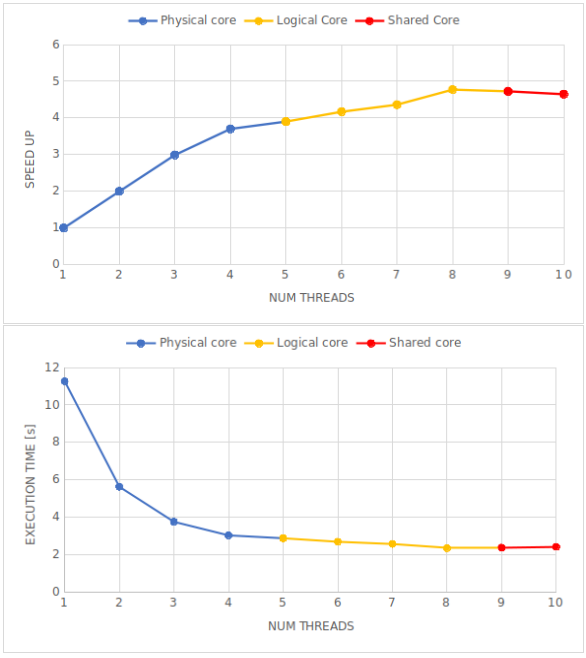

Figure 3. Synchronous Image Reader SpeedUp and execution time varying the number of threads

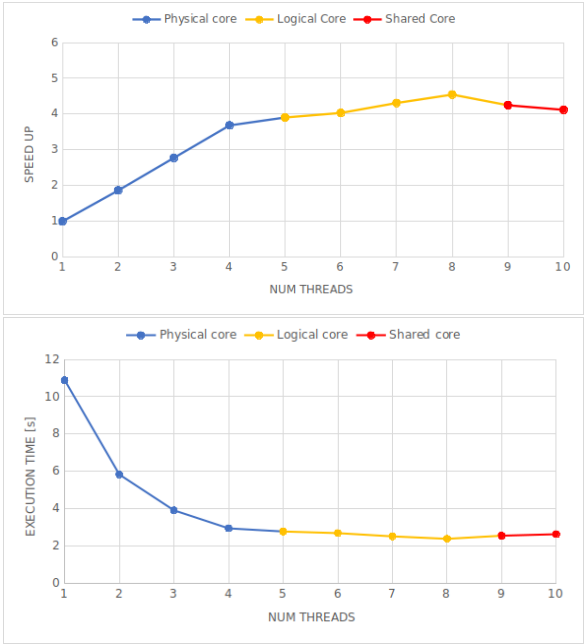

Figure 4. Asynchronous Image Reader SpeedUp and execution time varying the number of threads

Moreover in **figure 5** and in **table 1** we can see how the performance in both implementations are (almost) identical, so we can infer that a simple parallel implementation and an asynchronous implementation perform in the same way. This fact is not surprising since, as we have already stated, in the **Algorithm 2** the highlighted part is blocking, thus the asynchronous execution is very similar to the OpenMP implementation.

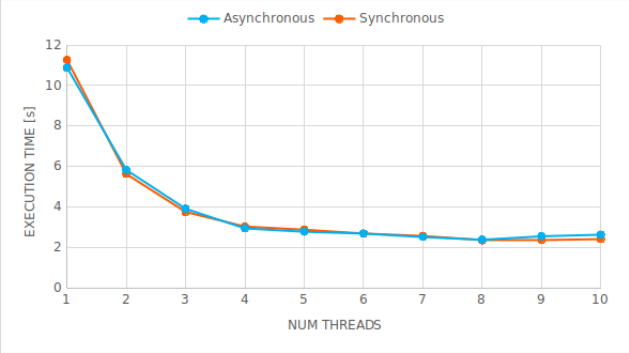

Figure 5. Comparison between asynchronous and synchronous Image Reader varying the number of threads

Image Reader Comparison		
Num threads	Asynchronous	Synchronous
1	10,88s	11,26s
2	5,84s	5,64s
3	3,92s	3,77s
4	2,95s	3,04s
5	2,79s	2,89s
6	2,70s	2,70s
7	2,53s	2,58s
8	2,39s	**2,36s**
9	2,56s	2,38s
10	2,64s	2,42s

Table 1. Comparison between asynchronous and synchronous Image Reader, graph displayed in figure 4 on the previous page

5.3. Full execution results

In this section are displayed the results of the Image Reader chained together with the 2D Pattern Recognition algorithm. Like before in each tests we have used a number of threads from 1 (sequential version) to 10.

The images used for the tests of the Full Execution is a subset of 5K images with a size of 115MB, to be more specific they are the first 5K images contained originally in the test set of the Kaggle competition. The query matrix used has dimensions dimension 50×50 pixels.

As for the Image Reader results, in **figure 6** (**synchronous** execution) and **figure 7** (**asynchronous** execution) we can identify the same three sections which correspond to the three different colors and also here in each colored section we can identify a different SpeedUp: (almost) **linear** for the blue section, **sub-linear** for the yellow section and we have **no more SpeedUp** in the red section. From the graphs we can see that the ideal number of threads are 8 which, we should keep in mind, are the number of logical core in the machine used for the tests.

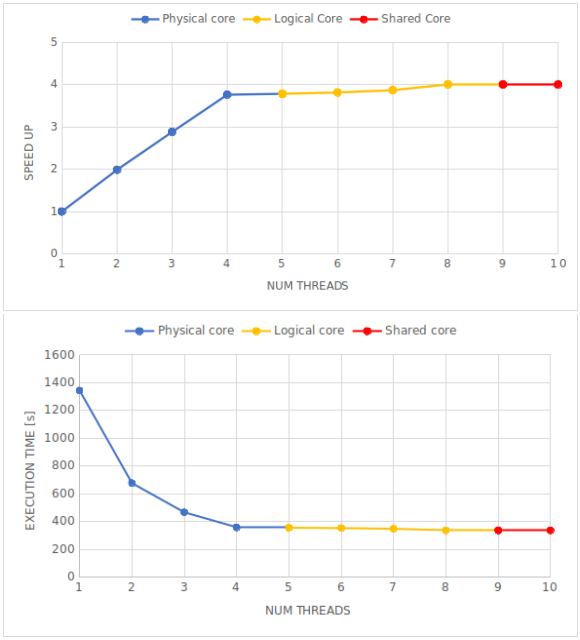

Figure 6. Synchronous Full Execution SpeedUp and execution time varying the number of threads

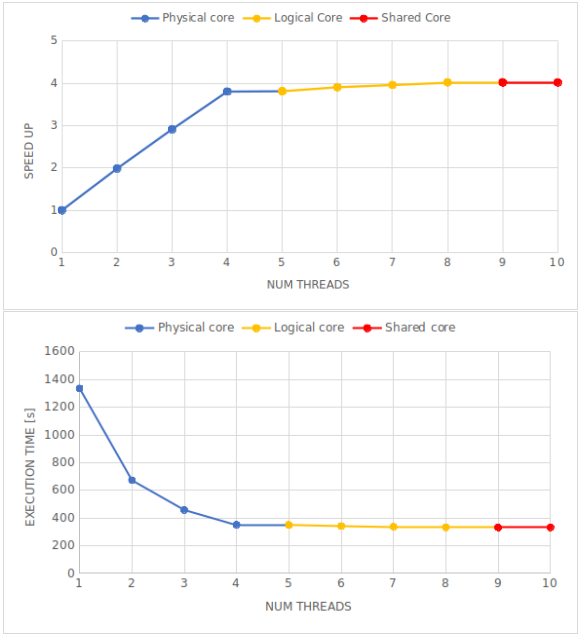

Figure 7. Asynchronous Full Execution SpeedUp and execution time varying the number of threads

Unfortunately in **figure 8** we can see how the performance in this Full Execution are almost identical, in fact in such graph the blue dotted line (asynchronous execution) and the orange line (synchronous execution) are overlapped. Actually from **table 2** we can see that the asynchronous execution is slightly better, but the improvement is not significant. With high probability the improvement is not relevant since the 2D Pattern Recognition algorithm is very expensive and CPU bound portions of the execution are predominant with respect to the I/O bound portions.

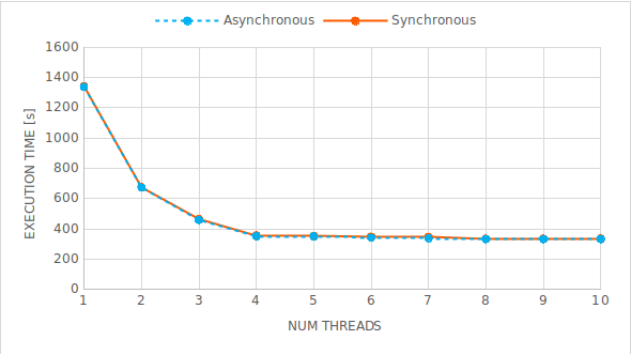

Figure 8. Comparison between asynchronous and synchronous Image Reader varying the number of threads

Image Reader Comparison		
Num threads	Asynchronous	Synchronous
1	1336,63s	1343,88s
2	672,91s	676,92s
3	458,49s	466,57
4	351,09s	357,43s
5	350,43s	355,48s
6	341,90s	352,65s
7	337,35s	347,69s
8	**332,10s**	335,72s
9	332,15s	336,096s
10	332,267s	336,166s

Table 2. Comparison between asynchronous and synchronous Image Reader numerical results, graph displayed in figure 5 on the previous page

6. Conclusions

The focus of this paper was to compare parallel asynchronous and synchronous implementation of an Image Reader and develop an application which exploit such Image Reader. In the first part we have compared the two versions and in that we have achieved very similar results with a **4.5x** SpeedUp in asynchronous implementation and a **4,7x** SpeedUp in synchronous implementation, with an almost identical execution time. Later for exploiting the asynchronous abilities to reduce idle time we have applied the 2D Pattern Recognition algorithm in series to the Image Reader, also here in asynchronous and synchronous manner. Unfortunately due to the fact that the Pattern Recognition is very expensive and the CPU bound portions are predominant with regard to the I/O bound we have achieved only a little performance boost. However, also here, thanks to the parallel structure of the program we have achieved a **4x** SpeedUp in both implementations.

References

[1] A. Baldrati. Cuda and openmp implementation of 2d pattern recognition, `https://github.com/ABaldrati/Pattern-Recognition-and-Image-Reader`, 2020.

[2] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[3] A. Jepson and D. Fleet. Boost C++ Libraries. `http://www.boost.org/`, 2015. Last accessed 2015-06-30.

[4] Kaggle. cats vs dogs kaggle competition `https://www.kaggle.com/c/dogs-vs-cats`, 2014.

[5] OpenMP Architecture Review Board. Openmp application program interface. Specification, 2011.

7. Appendix

In this section we provide all numerical results for the tests.

Synchronous Image Reader		
Num threads	Execution time	SpeedUp
1	11,26s	1,00x
2	5,64s	2,00x
3	3,77s	2,99x
4	3,04s	3,70x
5	2,89s	3,90x
6	2,70s	4,17x
7	2,58s	4,36x
8	**2,36s**	**4,77x**
9	2,38s	4,73x
10	2,42s	4,65x

Table 3. Synchronous Image Reader SpeedUp and execution time numerical results, graph displayed in figure 3 on page 4

Asynchronous Image Reader		
Num threads	Execution time	SpeedUp
1	10,88s	1,00x
2	5,84s	1,86x
3	3,92s	2,78x
4	2,95s	3,68x
5	2,79s	3,91x
6	2,70s	4,03x
7	2,53s	4,31x
8	**2,39s**	**4,55x**
9	2,56s	4,25x
10	2,64s	4,12x

Table 4. Asynchronous Image Reader SpeedUp and execution time numerical results, graph displayed in figure 4 on page 4

Synchronous Full Execution		
Num threads	Execution time	SpeedUp
1	1343,88s	1,00x
2	676,92s	1,99x
3	466,57s	2,88x
4	357,43s	3,76x
5	355,48s	3,78x
6	352,65s	3,81x
7	347,69s	3,87x
8	**335,72s**	**4,00x**
9	336,09s	4,00x
10	336,16s	4,00x

Table 5. Synchronous Full Execution SpeedUp and execution time numerical results, graph displayed in figure 6 on page 5

Asynchronous Full Execution		
Num threads	Execution time	SpeedUp
1	1336,63s	1,00x
2	672,91s	1,99x
3	458,49s	2,92x
4	351,09s	3,81x
5	350,43s	3,81x
6	341,90s	3,91x
7	337,35s	3,96x
8	**332,10s**	**4,02x**
9	332,15s	4,02x
10	332,26s	4,02x

Table 6. Asynchronous Full Execution SpeedUp and execution time numerical results, graph displayed in figure 7 on page 5