ADVANCED COMPUTER ARCHITECTURE

PROJECT REPORT

Color-based Image Segmentation

using Parallel K-means Clustering

*Author:*

Gabriele MIRANDO

A.Y. 2017/2018

**Abstract**

The objective of the project was to develop a C language program for color-based image segmentation using the standard k-means clustering algorithm, and to speed up the execution by implementing a parallel version of the program able to run on multi-core processors. This report contains a complete account of how the work has been organized and what results have been obtained. It starts with a detailed analysis of how the standard k-means algorithm has been exploited to achieve color-based image segmentation through the clustering of the pixels of the image. An a-priori study of the available parallelism is then conducted, taking into account the results obtained from the profiling of the serial code. The algorithm has been parallelized using OpenMP. A complete description of the parallel implementation is provided. Finally, multiple test cases for the program are taken into consideration and some observations about performance are deduced from the comparison of the execution times obtained on machines with an increasing number of cores.

Table of Contents

Analysis of the Serial Algorithm 4

Serial Implementation Walkthrough 5

Serial Implementation Results 9

Study of the Available Parallelism 12

Serial Code Profiling 12

Speedup Estimation 13

Description of the Parallel Implementation 16

OpenMP Implementation 16

Performance Analysis 18

Speedup on a Local Machine 18

Speedup on Google Cloud Platform 20

Conclusions 23

# Analysis of the Serial Algorithm

Image segmentation is the process of partitioning a digital image into its constituent objects. Given a digital image, performing a segmentation means to identify the main elements that make up the scene. Achieving a good quality segmentation is one of the most difficult and challenging tasks in digital image analysis, but it also represents the first step towards an effective detection of the objects present in the image. The more accurate is the segmentation phase, the more likely is the image recognition process is to succeed. For the purpose of the project, a simple color-based segmentation technique has been considered.

The color-based approach relies on the fact that in a digital image sub-regions of pixels can be identified on the basis of the their color. The partitioning of the pixels can be achieved using a clustering algorithm. Clustering is by definition the task of organizing a given set of objects into groups, or more precisely into clusters, in such a way that objects in the same cluster are more similar to each other than to those in other clusters. Therefore given an digital image, using a clustering algorithm it is possible to divide the image pixels into groups, such that pixels in the same group are similar in terms of color.

The fact that is extremely easy to implement, but is also computationally very efficient, made the algorithm proposed by Stuart P. Lloyd in 1957 the most popular clustering technique used in scientific and industrial applications. This technique is known as k-means clustering algorithm. Although multiple variants of k-means have been later developed (Forgy, MacQueen, Hartigan-Wong), Lloyd’s version is considered the standard version of the algorithm and it is the one that has been used for achieving color-based segmentation in this project.

The aim of the project was to develop a program that, given a digital image saved in one of most the common format (JPEG, PNG, BMP, GIF…) and an integer number K, is able to produce a segmented version of the initial image in which K different regions of pixels of similar color are distinguishable. The execution flow of the program can be divided into three stages: the reading of the RGB color values of the pixels of the starting image, the segmentation of those values using k-means clustering and then the writing of the obtained results to a new image, that represents the final outcome of the segmentation process. The activity has been focused on implementing the C language code for the k-means segmentation, and then on parallelizing it using OpenMP. An open source library available on Github has been adopted for convenience to handle the image I/O operations.

## Serial Implementation Walkthrough

The standard k-means algorithm can be used to partition a set of objects, described by a series of measurable features, into a predefined number of clusters, so that objects in the same cluster are more similar to each other than to those in other clusters. The objective of k-means is to find the clusters composition that minimize the Sum of Squared Errors (SSE), i.e. the sum of the distances between each object and its cluster center. When used for color-based image segmentation, the objects to be clustered are the pixels of the image and the objects measurable features are the pixels RGB values. The SSE is the sum of the squared Euclidean distances between each pixel and the center of the cluster to which the pixel belongs. It is important to consider that the k-means algorithm is just an heuristic procedure, that attempts to find iteratively and in the fastest way possible an approximation of the optimal solution for the problem of the minimization of the SSE. The partitioning of the pixels into clusters obtained at the convergence of the algorithm heavily depends on the initialization of the clusters centers.

In the program, the color values obtained by scanning the input image are stored in a matrix allocated in memory and constitute the initial dataset on which to perform the segmentation. Each row of the matrix contains the 0 to 255 RGB values of each pixel of the image. Given a digital image of width W and height H, the initial dataset is a matrix made up by N = W x H rows and three columns:

*data* =

where are respectively the red, green and blue components of the i-th pixel of the image. The segmentation of the image pixels is achieved using k-means clustering algorithm by applying the following steps:

1. **Initialization of the clusters centers**. K pixels are randomly picked from the dataset and set as initial cluster centers. A matrix of K rows is allocated in memory to store the values of the clusters centers:

*centers* =

where are respectively the red, green and blue real values of the j-th cluster center. Each cluster center is a vector of three components that need to be initialized with the RGB values of a random pixel. Selecting the initial centers at random among the objects of the dataset allows to achieve acceptable results in the case of image segmentation, but a more sophisticated initialization or multiple repetitions of the algorithm may be required for other applications, especially in the field of data analysis.

1. **Assignment of each pixel to the nearest cluster**. For each of the pixels of the dataset, the squared Euclidean distance from all the cluster centers is computed. The cluster center for which the value of the squared Euclidean distance is the lowest is found and the pixel is assigned to that cluster. The squared Euclidean distance between one pixel and a cluster center is computed as the sum of the squared differences of the values of the RGB color components. To keep track of the cluster to which each pixel belongs at each iteration of the algorithm, an array of size N is stored in memory:

*labels* =

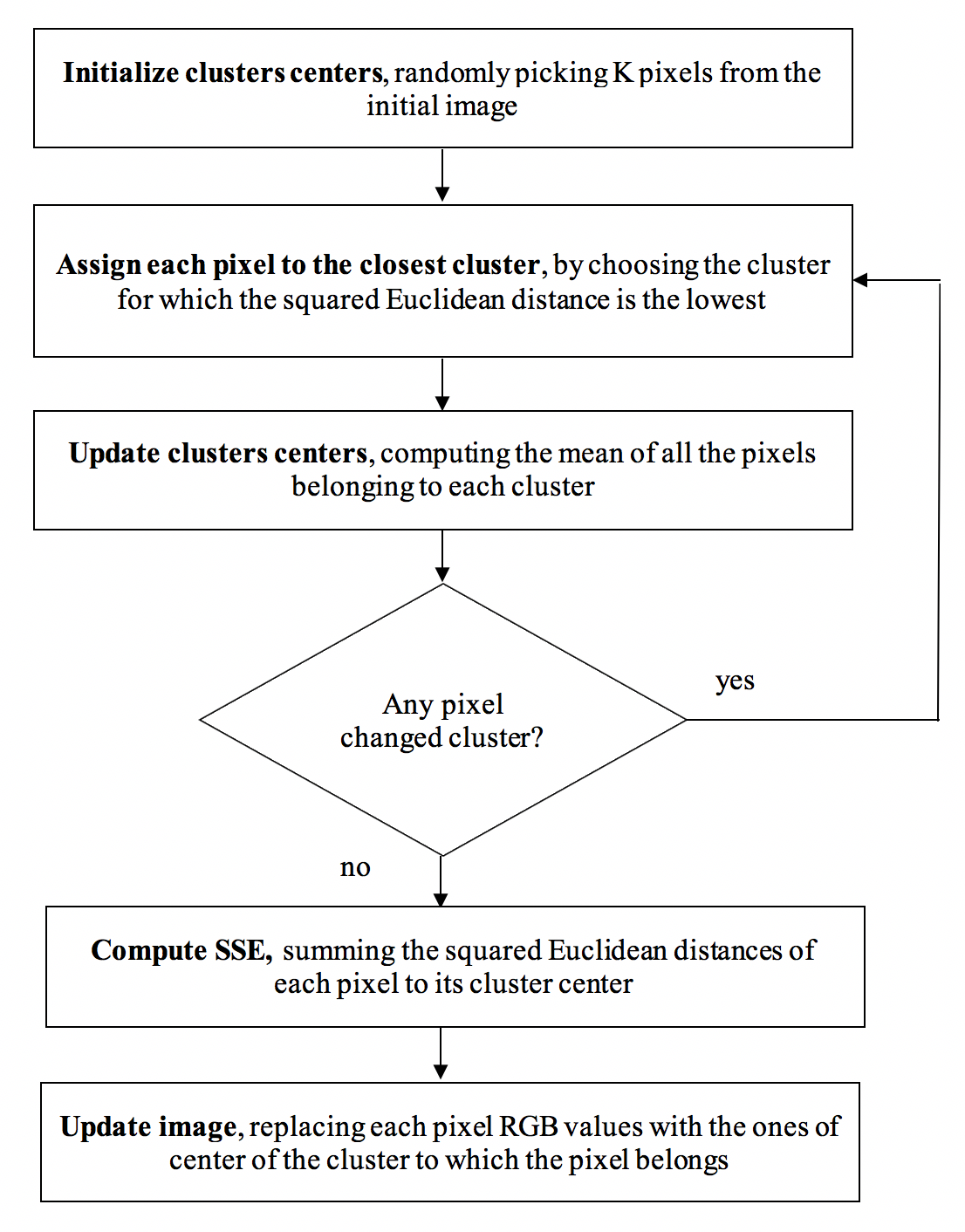
where is an integer value from 0 to K-1 that indicates the index of the cluster to which the i-th pixel currently belongs. Another array of N elements is used to save the distances of each pixel to the center of the cluster it belongs:

*dists* =

where is a real value that contains the squared Euclidean distance of the i-th pixel to its cluster center. Even if it is not strictly necessary, the *dists* vector can be very useful to speed up some computations. Both the *labels* and *dists* arrays are updated at this stage. For each pixel from 0 to N-1:

1. **Update of the cluster centers**.For each cluster, the value of the center is recomputed by calculating the mean of all the pixels belonging to the cluster. In the *centers* matrix, each center is updated such that each color component is the mean of the respective color components of the pixels belonging to the cluster. In the case of a cluster being empty, a good practice is to set its new center to the pixel of the dataset with the higher distance from its current cluster center.
2. **Test for completion**. The algorithm is returned to step 2 until the cluster assignments do not change (this is when the algorithm converges to a solution) or until a predefined maximum number of iterations is reached. If none of the pixels has changed the cluster to which they belong or the number of maximum iterations is reached, the clustering is completed.
3. **SSE computation**. The SSE is easily obtained by summing all the elements of the *dists* vector. Although the computation of the SSE is not essential, comparing the SSE among different executions of the algorithm can be useful to check the goodness of the result.
4. **Update of the image data.** The RGB values of each pixel of the image are replaced with those of the cluster center to which the pixel belongs. In this last step the *data* matrix is updated and will be used as a source for producing the final segmented image.

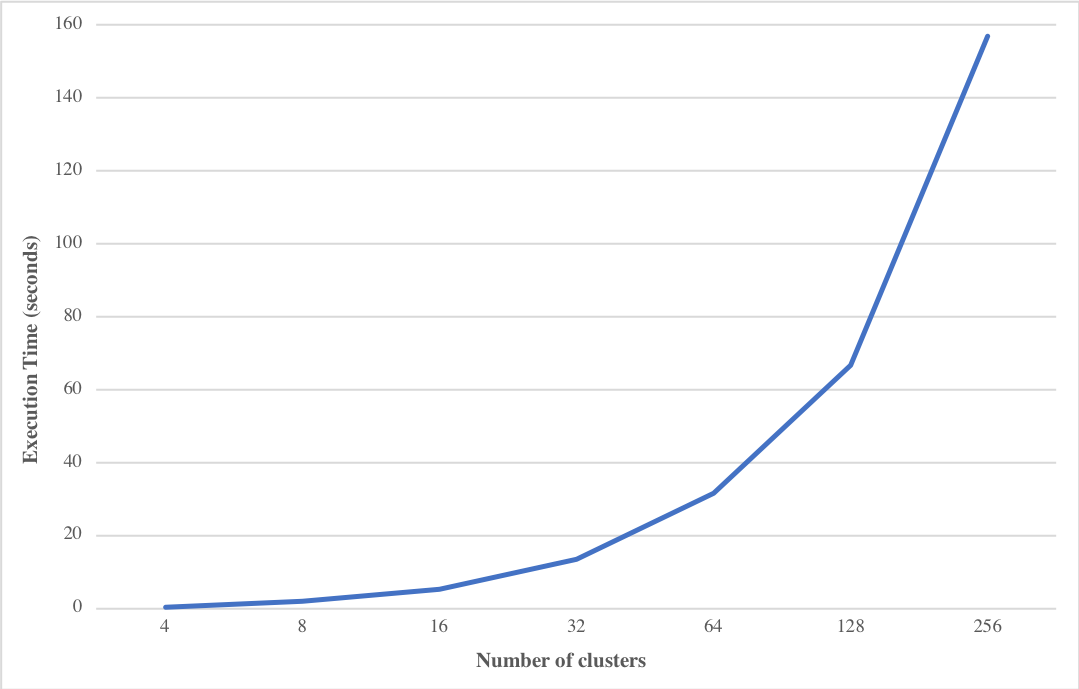
A visual representation of the algorithm for color-based segmentation using k-means clustering is shown by the flowchart of Figure 1. The code of the program has been structured to follow this organization as close as possible. Each step of the algorithm has been implemented as subroutine of the main *kmeans\_segm* function.



**Figure 1.** The flowchart for color-based segmentation using k-means clustering algorithm

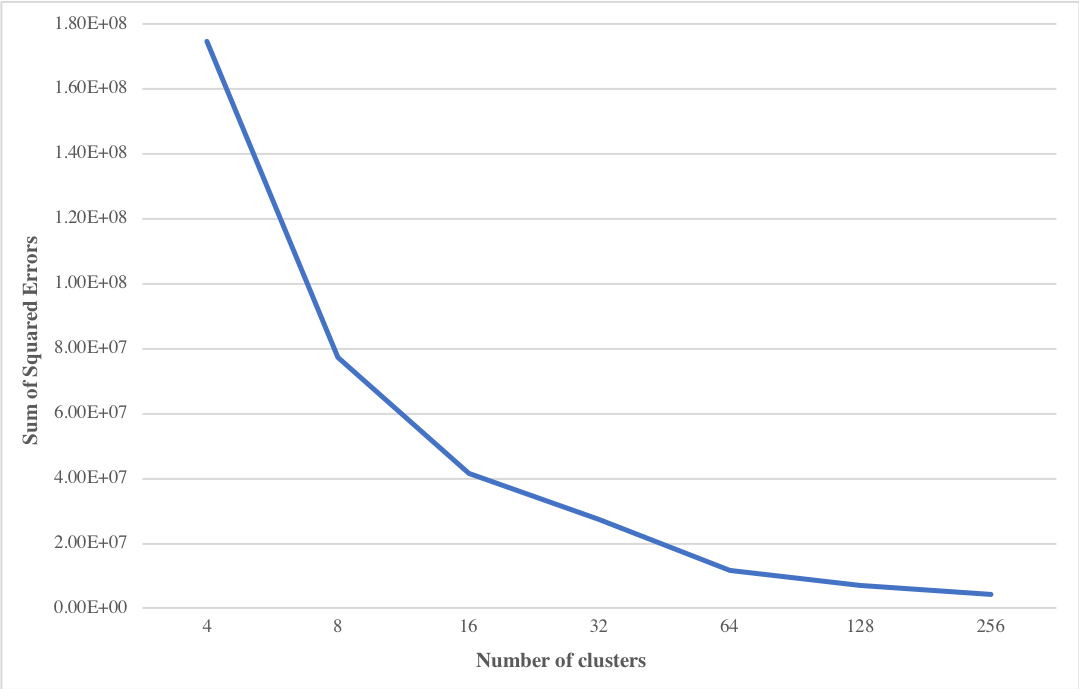
## Serial Implementation Results

A 640 × 360 JPEG image has been used as main test for the program. The image has been segmented multiple times, specifying each time an increasing number of clusters K. For each of the executions, the k-means algorithm has been left to run until convergence. When incrementing the number of clusters, the computations required by the k-means algorithm also increased, which traduced in higher execution times. Figure 2 shows clearly that the execution time of the serial program grows considerably in relationship to the number of clusters selected for the segmentation process. The execution times refer only to the k-means segmentation algorithm. Image I/O operations are not taken into consideration.



**Figure 2.** The graph of the execution times of the serial program in relation to the number of clusters

Increasing the number of cluster leads to the higher execution times, but allows to obtain a more precise cluster configuration, and therefore to lower the final value of the Sum of Squared Errors returned by k-means. Figure 3 displays how the SSE decreases when incrementing the number of clusters.



**Figure 3.** The graph of the SSE returned by the k-means algorithm in relation to the number of clusters

The test image represents a horse in the foreground, standing over a hill with mountains in the background. Four main colors are immediately distinguishable. Figure 4 reports the qualitative results of the segmentation. Images with different levels of accuracy were obtained, and it is easy to see how the choice of the number of clusters is crucial for a good quality segmentation. Only when performing the segmentation with K = 4, the main elements of the scene (the shape of the horse, the grass, the mountains and the sky in the background) are clearly identifiable.

|  |  |
| --- | --- |
| **(a)** The original JPEG image | **(b)** Color-based segmentation with K = 4 |
| **(c)** Color-based segmentation with K = 8 | **(d)** Color-based segmentation with K = 16 |
| **(e)** Color-based segmentation with K = 32 | **(f)** Color-based segmentation with K = 64 |
| **(g)** Color-based segmentation with K = 128 | **(h)** Color-based segmentation with K = 256 |

**Figure 4.** The test image segmented using different values of K

When increasing the value of K, the SSE returned by the k-means algorithm is decreasing and the segmented image starts to resemble to the original image. Setting K = 256 allowed to obtain a 256-color palette version of the initial picture.

The color-based technique adopted for the project is naive and truly effective only in particular circumstances, and it’s rarely adopted when the goal is to achieve an high quality segmentation. However, the real purpose of the activity was to implement, parallelize and assess the performance speedup of the k-means algorithm, which has been used for the clustering of the pixels. Color-based image segmentation has just been chosen among the many applications of the k-means algorithm for its practicality and originality.

# Study of the Available Parallelism

Parallelization is the task of modifying the source code of a program in such a way that many calculations can be carried out simultaneously by different cores of the same processor. The first step towards an effective parallelization, which guarantees a significant speedup, was to assess what are the sections of the algorithm where the program does most of the work and to understand if in those sections the computations can be carried out simultaneously by multiple threads. Using Amdahl’s Law it is also possible to produce an estimation of the achievable speedup of the parallel implementation.

## Serial Code Profiling

The serial version of the program has been profiled using the GNU profiler *gprof.* Thanks to the Call Graph profiling module of *gprof* it is possible to get the percentage of the execution time taken by each function of program. In particular the code of the program is organized in such a way that the segmentation process takes place in the *kmeans\_seg* function, and each of the steps of the k-means segmentation algorithm is implemented into an independent sub-routine. By applying the Call Graph profiling to the *kmeans\_seg* functionit was possible to identify the steps of the algorithm that required the most of the running time. The profiling has been repeated specifying an increasing number of clusters and the results are reported in Table 1.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **K=4** | **K=8** | **K=16** | **K=32** | **K=64** | **K=128** | **K=256** |
| *init\_centers* | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| *assign\_pixels* | 79.2% | 87.7% | 92.6% | 96.4% | 98.1% | 99.1% | 99.5% |
| *update\_centers* | 18.9% | 12.3% | 7.3% | 3.6% | 1.9% | 0.9% | 0.5% |
| *compute\_sse* | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| *update\_data* | 1.9% | 0.0% | 0.1% | 0.0% | 0.0% | 0.0% | 0.0% |

**Table 1.** The results of the profiling of the serial code

Table 1 clearly highlights how most of work of the k-means algorithm is done in the pixels assignment step, when for each pixel the distance to each cluster center is computed and the pixel is assigned to the closest cluster. When increasing the number of clusters K, the *assign\_pixels* function requires by far the highest number of computations, reaching up to 99% of the computation time of the entire segmentation process.

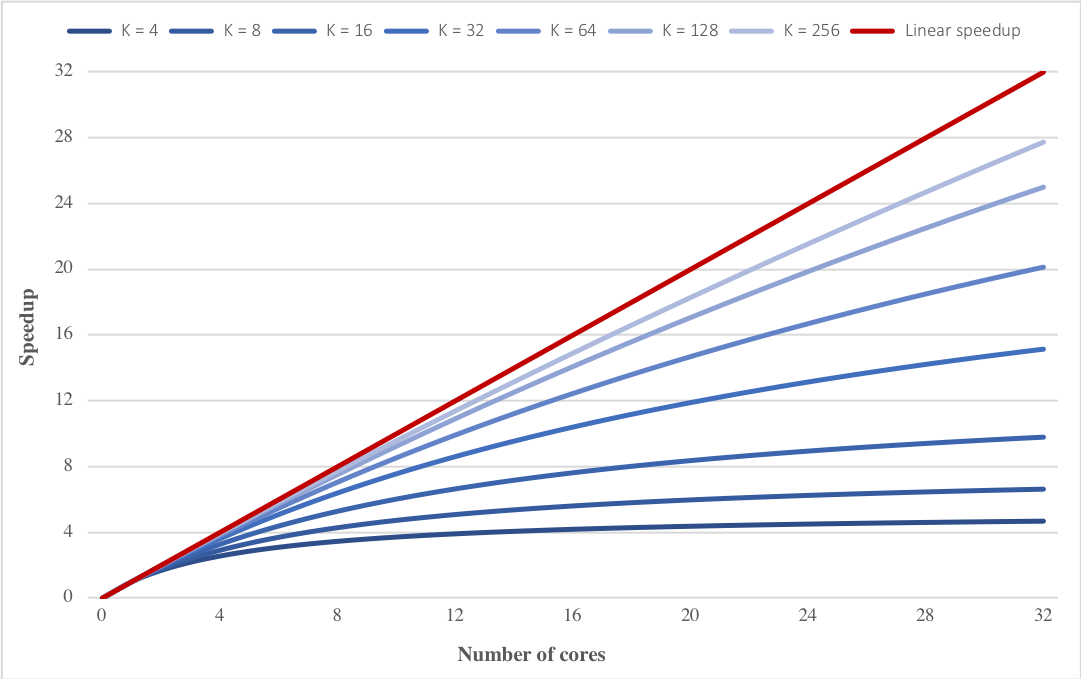
## Speedup Estimation

Keeping in mind the results obtained during the profiling of the code, it is useful to discuss the feasibility and of making each function to run in parallel:

* **init\_random:** the clusters center are initialized by selecting at random K pixels from the image dataset. The *rand* function in C is not thread safe. Parallelizing this function requires extra effort which is not worthy, since it won’t improve the final speedup.
* **assign\_pixels:** the assignment of each pixel to the closest cluster center is what is known an embarrassingly parallel problem. The computation of the closest cluster of each pixel can be done separately by different threads without the need of special synchronization techniques. The *data, centers, labels* and *dists* structures are shared among threads. The *data* matrix of pixels and the *centers* matrix are used only in reading mode. The *labels* and *dists* arrays are updated with the newly computed values, but there is no need to worry about race conditions because each thread operates on different elements stored in different memory addresses.
* **update\_centers:** updating cluster centers requires to compute the mean of the pixels values belonging to each cluster. To parallelize this function the accesses to the *centers* matrix need to be opportunely synchronized or updating it from different threads can lead to race conditions and unpredictable results.
* **compute\_sse:** computing the SSE can be done by summing all the values of the *dists* array*.* This implies a loop-carried dependence that need to be considered, even if the parallelization of this function won’t have a significant impact on the final speedup.
* **update\_data:** updating the image pixels values, by replacing them with those of the cluster center to which each pixel belongs, can be easily done in parallel without the need of synchronization just like pixels assignments.

It is possible to use Amdahl’s Law to make an estimation of the speedup achievable just by parallelizing the pixels assignment. According to Amdahl’s Law:

where *n* is the number of cores used to run the parallel algorithm, *p* is the portion of the code that can be executed in parallel and *S* is the final speedup of the parallel program. Amdahl’s Law results have been used to realize the graph of Figure 5.



**Figure 5.** Graph of thespeedup obtainable by parallelizing the pixels assignment

The speedup estimated is higher when the value of K is higher, since the *assign\_pixels* function requires more computations with an higher number of clusters, and parallelizing it guarantees a larger gain in performance. The speedup does not grows indefinitely when increasing the number of cores, but it slowly settles to value of *1/(1-p)*, which is obtained from Amdahl’s Law when *n* tends towards infinity. The maximum speedup reachable by increasing the number of cores depends by the portion of the algorithm that is parallelizable. The red line of the graph in Figure 5 represents the linear speedup, i.e. the theoretical limit of the speedup, obtained when the 100% of the code can be run in parallel. The parallelization of the *assign\_pixels* functionalone could guarantee a considerable speedup of the algorithm, but to get as close as possible to a linear speedup parallelizing *update\_centers* step with an efficient synchronization technique revealed to be very useful.

# Description of the Parallel Implementation

By the study of the available parallelism of the k-means clustering algorithm, it resulted that a significant speedup can be obtained by exploiting the high level of data parallelism, which is intrinsic to the k-means clustering algorithm. Most of the computations are done by the program when searching for the closest cluster of every pixel and are repeated at each iteration of the algorithm. In a shared memory environment, the threads available to the program can perform the computation of the closest cluster for different pixels simultaneously. As it happens in the SIMD paradigm, each thread is instructed to do the same job on different data stored in memory.

## OpenMP Implementation

The program has been parallelized using OpenMP. OpenMP is an application programming interface specifically developed for multiprocessing programming in shared memory environments using the C language. It offers a set of compiler directives, library routines and environmental variables that can be used to make a C source code to run simultaneously on multiple cores of a processor.

OpenMP multithreading is based on a fork-join model. The control flow of the algorithm is strictly sequential and is handled by the main thread. When a computationally intensive section is reached, the main thread forks and the calculations are distributed among all the threads available. Once all the threads have finished their job, they join together and the control goes back to the main thread. By adding pragmasto the serial source code*,* it was possible to implement a parallel version of the k-means segmentation algorithm with a reduced effort.

To parallelize the assignment of each pixel to the closest cluster a *parallel for* work-sharing construct has been used.

*#pragma omp parallel for schedule(static) private(…)*

*for (px = 0; px < n\_px; px++) {*

*…*

*}*

The work that is done inside the for loop is shared across all the threads available to the program. The scheduling preference is set to *static*, so that each thread is assigned in advance an equal number of iterations of the loop to execute. In this case, the *static* approach is much more efficient than *dynamic* scheduling, because the work distribution across threads is well balanced. The amount of calculations needed to find the closest cluster (i.e. to compute the squared Euclidean distance between the pixel and every cluster center and to find the minimum distance) is the same for every pixel. The threads are instructed to find the closest cluster for equally large ranges of pixels, and will finish their job approximately at the same time. A *dynamic* approach, where portions of the loop are assigned at run-time to idle threads, would imply a waste of time due to a communication overhead.

The same approach has been used to parallelize the *update\_centers* function, but with an additional complication. To compute the new centers, i.e. the mean of the pixels belonging to each cluster, a loop has been used to scan for all the pixels. For each pixel, the program has to check the cluster of each pixel and update the *centers* matrix, containing now the partial sums of the pixels values of each cluster, and a *counts* array, keeping track of the number of pixels belonging to each cluster. Updating the *centers* matrix or the *counts* array from multiple threads may lead to race conditions and unexpected results of the program. A synchronization technique needs to be adopted. Using a *critical section* to allow only one thread at a time to perform the update operations would irreparably compromise the performance of the algorithm. Since the update operations consist of a simple additions, applying a *reduction* is a good choice in this situation. Starting from version 4.5 of OpenMP it is possible to perform reductions also on arrays.

*#pragma omp parallel for private(…) reduction(+:centers[..],counts[..])*

*for (px = 0; px < n\_px; px++) {*

*…*

*}*

The *reduction* clause allows to protect the update operations of the cluster centers and the counters, but using an efficient OpenMP synchronization technique. The use of this pragma revealed to have a remarkable impact on the final speedup.

# 

# Performance Analysis

In order to evaluate the efficacy of the parallel implementation realized using OpenMP, the program has been tested on different multi-cores machines and the speedup in respect to the serial version has been computed. The speedup of the parallel implementation can be by easily obtained with the following formula:

where n is the number of cores used to run the parallel algorithm, is the time to execute the serial version and is the execution time of the program parallelized using OpenMP. Since the k-means algorithm may take a different number of iterations to converge depending on the initialization of the clusters centers, the same set of pixels has been used as initial centers when comparing the execution time of the serial and of the parallel implementation. Only the time taken to perform the k-means segmentation has been considered, while image I/O operations have not been taken into account. The parallel program has been tested first on a local machine and then on the Google Cloud Platform, always increasing the number of clusters K to use for image segmentation.

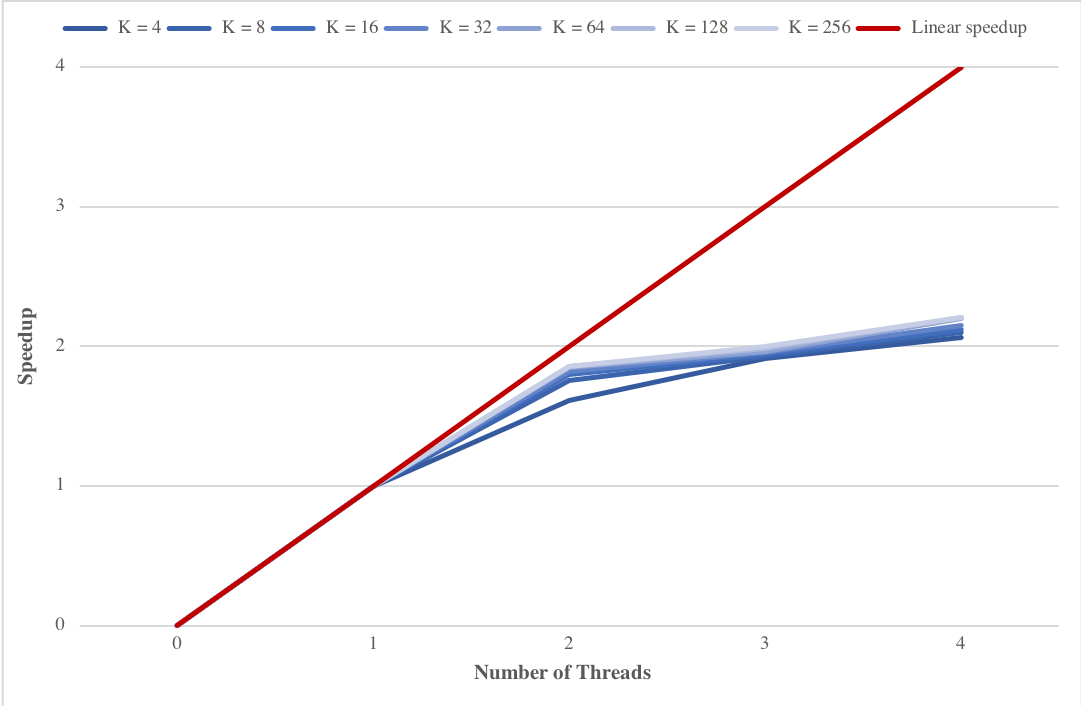
## Speedup on a Local Machine

The first machine utilized to assess the performance of the parallel implementation of the k-means segmentation algorithm was equipped with an Intel Core i5-6360U CPU @ 2.00GHz, with just two physical cores and four threads available thanks to Intel Hyper-Threading Technology. The speedup results are reported in Table 2.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Threads** | **K=4** | **K=8** | **K=16** | **K=32** | **K=64** | **K=128** | **K=256** |
| ***2*** | 1.612 | 1.753 | 1.797 | 1.819 | 1.847 | 1.857 | 1.858 |
| ***3*** | 1.912 | 1.929 | 1.943 | 1.955 | 1.966 | 1.983 | 2.000 |
| ***4*** | 2.062 | 2.098 | 2.120 | 2.148 | 2.198 | 2.207 | 2.210 |

**Table 2.** The speedups obtained on a local machine

Figure 6 displays the graph of the evolution of the speedup obtained increasing the number of threads to use in OpenMP.



**Figure 6.** Graph of the speedup obtained on a local machine

From the profiling of the serial code, it resulted that a considerable speedup could be achieved by the parallelization of the pixels assignment alone. Using an OpenMP *reduction*, it was possible to successfully parallelize also the calculations to update the clusters centers. Table 3 reports the speedup values obtained without the parallelization of the *update\_centers* function. Comparing it with the values of Table 2, it is possible to assess the gain achieved in terms of speedup.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Threads** | **K=4** | **K=8** | **K=16** | **K=32** | **K=64** | **K=128** | **K=256** |
| ***2*** | 1.499 | 1.640 | 1.733 | 1.778 | 1.831 | 1.852 | 1.854 |
| ***3*** | 1.745 | 1.776 | 1.879 | 1.927 | 1.950 | 1.975 | 1.993 |
| ***4*** | 1.866 | 1.979 | 2.049 | 2.116 | 2.176 | 2.196 | 2.197 |

**Table 3.** The speedups obtained without the parallelization of the centers update

Table 4 shows the percentage increase of the speedup obtained by parallelizing the centers update step, using the OpenMP *reduction* clause. An higher performance gain is obtained when a small number of cluster is specified, i.e. when the *update\_centers* function represents a larger portion of the execution time.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Threads** | **K=4** | **K=8** | **K=16** | **K=32** | **K=64** | **K=128** | **K=256** |
| ***2*** | 7.57% | 6.89% | 3.69% | 2.35% | 0.89% | 0.29% | 0.19% |
| ***3*** | 9.54% | 8.61% | 3.43% | 1.47% | 0.82% | 0.42% | 0.38% |
| ***4*** | 10.50% | 6.03% | 3.46% | 1.51% | 0.99% | 0.51% | 0.60% |

**Table 4.** The percentage increase of the speedup obtained by parallelizing the centers update

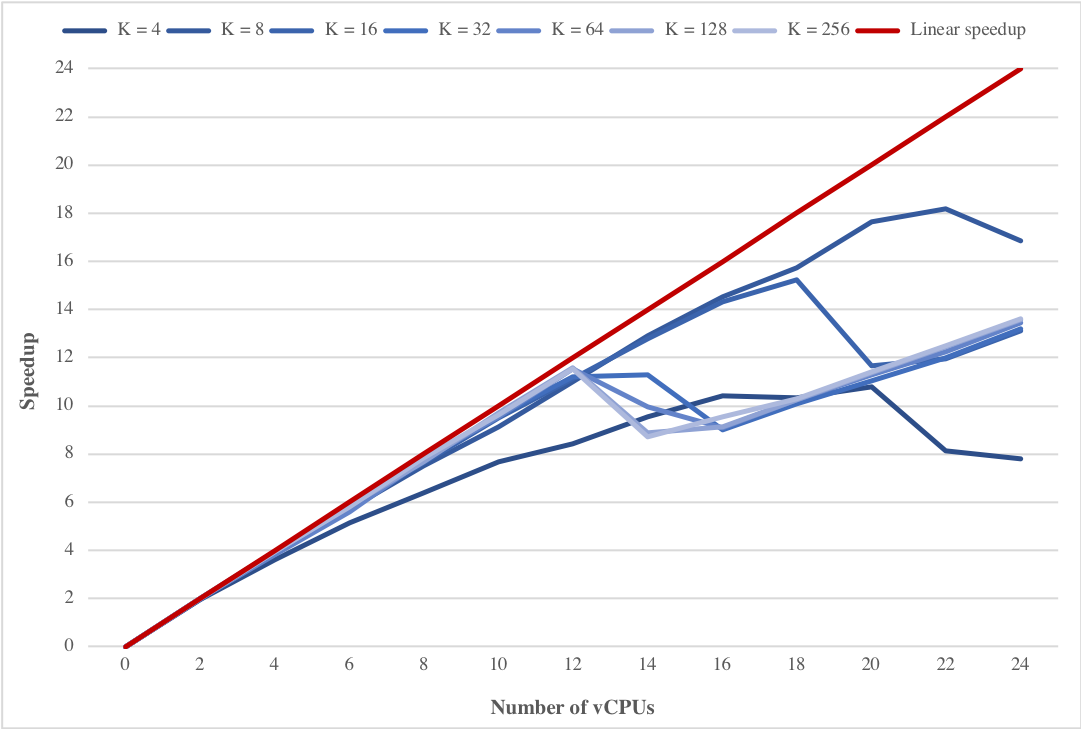
## Speedup on Google Cloud Platform

For a more complete analysis of how the parallel program behaves when an higher number processors is available, the Compute Engine of Google Cloud Platform has been used to assemble a virtual machine instance, with a maximum of 24 vCPUs. Each processor was an Intel Xeon CPU @ 2.00GHz. Table 5 reports the speedup obtained when increasing the number of vCPU used to run the parallel algorithm.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **vCPUs** | **K=4** | **K=8** | **K=16** | **K=32** | **K=64** | **K=128** | **K=256** |
| ***2*** | 1.965 | 1.981 | 1.963 | 1.979 | 1.974 | 1.975 | 1.991 |
| ***4*** | 3.616 | 3.892 | 3.897 | 3.931 | 3.804 | 3.941 | 3.939 |
| ***6*** | 5.145 | 5.737 | 5.781 | 5.832 | 5.622 | 5.870 | 5.830 |
| ***8*** | 6.403 | 7.522 | 7.637 | 7.748 | 7.759 | 7.687 | 7.829 |
| ***10*** | 7.680 | 9.147 | 9.523 | 9.580 | 9.569 | 9.729 | 9.620 |
| ***12*** | 8.435 | 11.021 | 11.124 | 11.222 | 11.540 | 11.600 | 11.539 |
| ***14*** | 9.556 | 12.926 | 12.805 | 11.308 | 9.945 | 8.891 | 8.708 |
| ***16*** | 10.402 | 14.536 | 14.312 | 9.006 | 9.094 | 9.144 | 9.561 |
| ***18*** | 10.356 | 15.729 | 15.230 | 10.106 | 10.234 | 10.273 | 10.286 |
| ***20*** | 10.779 | 17.640 | 11.655 | 11.063 | 11.277 | 11.381 | 11.406 |
| ***22*** | 8.157 | 18.185 | 11.949 | 11.991 | 12.260 | 12.460 | 12.504 |
| ***24*** | 7.791 | 16.867 | 13.118 | 13.191 | 13.445 | 13.582 | 13.626 |

**Table 5.** The speedups obtained on a virtual machine instance of the Google Cloud Platform

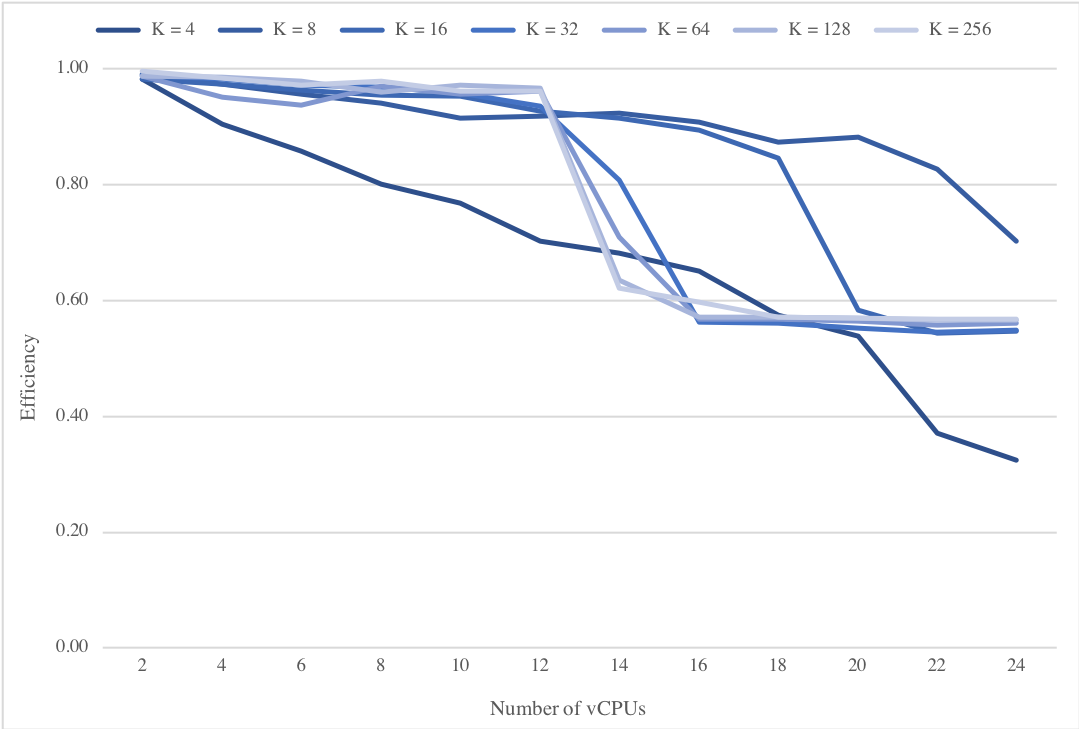
Figure 7 shows clearly that the speedup is close to linear up to twelve vCPUs. When increasing over twelve the number of vCPU, the performance of the parallel program starts to have a not well defined evolution. The reason of this irregular behavior may reside in the scheduling of the vCPUs. This performance anomaly can threaten the scalability of the parallel implementation.



**Figure 7.** Graph of the speedup obtained on Google Cloud Platform

The efficiency of the parallel program is given by:

and it’s a number that varies between 0 and 1. The scalability of the parallel algorithm is the capacity of maintaining constant its efficiency when increasing the number of processors. Figure 8 perfectly highlights the drop in efficiency of the parallel program when more than twelve vCPU are used.



**Figure 6.** Graph of the efficiency of the parallel algorithm measured using Google Cloud Platform

# Conclusions

The objective of the project was to assess how parallel computing can be used to drastically increase the performance of the k-means clustering algorithm. The choice of image segmentation as application of k-means clustering was dictated by the fact that image segmentation is one of the most relevant problem in digital image analysis and computer vision.

The k-means algorithm is inherently characterized by an high level of data parallelism. The phase of profiling of the serial code revealed to be crucial to understand what sections of the program was more convenient to parallelize in order to obtain a significant gain in performance. Thanks to OpenMP is was possible implement a parallel version of the program, that was running much faster than the serial algorithm in a shared memory environment. The Compute Engine of Google Cloud Platform was extremely useful to conduct a scalability analysis of the parallel algorithm. The problem with virtual instances in the cloud is that, due to the scheduling techniques of the processors, an increase in performance is not guaranteed when incrementing the number of vCPUs.

To exploit more effectively the data parallelism inherently available in the k-means clustering algorithm, an API, like CUDA, for programming massively parallel architectures, could have been used. The high computational power available in GPUs, produced by the large number of cores, would have made it possible to achieve an even better performance.