




2017/2018

K-means clustering

Parallel Programming Project

Advanced Computer Architecture



Christian Calabrese, mat: 461551
Gioele Maruccia, mat: 461933

Summary

Summary	1
Introduction	3
K-Means algorithm	3
Required data.....	4
Drawbacks.....	4
Stopping criteria.....	5
Hardware specifications	5
Software employed	5
Analysis of the serial algorithm	6
Introduction.....	6
Employed datasets	6
Breast cancer Wisconsin	6
Iris	7
Data generation	7
Algorithm flowchart.....	8
Results	9
General exploration.....	9
Memory Consumption.....	11
A-priori study of the available parallelism	12
A bottom up approach	12
Data structure.....	12
Function “calcSSE”	12
Function “findCentroids”	13
Function “findClusters”	14
OpenMP parallel implementation.....	15
Data acquisition.....	15
First implementation - for parallel [automatic scheduling].....	15
Second implementation - for parallel [custom scheduling].....	19
Chunksize value.....	19
Scheduling policy choice	19
Third implementation - for parallel with collapse [custom scheduling].....	22
Profiling analysis of the Parallel algorithm.....	25
4-threads case - Breastcancer	25
4-threads case - Iris.....	26
16-threads case - Breastcancer	27
16-threads case – Iris.....	28
A strange behavior [high threads number case]	28
Optimizations	30
Thread number optimization.....	30
Compiler optimization [–O3]	31
Comparison.....	32

Testing and debugging	34
Performance analysis and speedup	34
Conclusions	36
Bibliography	37

Introduction

The aim of the report is to explain and analyze the obtained results of parallelization of a common machine learning algorithm: k-means clustering.

After a brief introduction about the chosen algorithm and the results obtained by its serial implementation, the enhancement resulted from the parallelization by the use of OpenMP application will be listed and discussed.

K-Means algorithm

K-means is an iterative algorithm which has the focus of finding K homogeneous clusters throughout the selected dataset. Its main characteristic is its unsupervised nature, which means that it's not possible to know in advance the real number of clusters unless an a-priori expert's analysis on the data.

A summary about how this algorithm actually works is explained in the following.

Once K random samples have been chosen from the dataset as the initial centroids of clusters, the algorithm computes the distance from them to the other samples, assigning each one to the closest cluster centroid.

Then, proceeding in an iterative way, new cluster centroids are computed from the just formed clusters until the stopping criteria are reached.

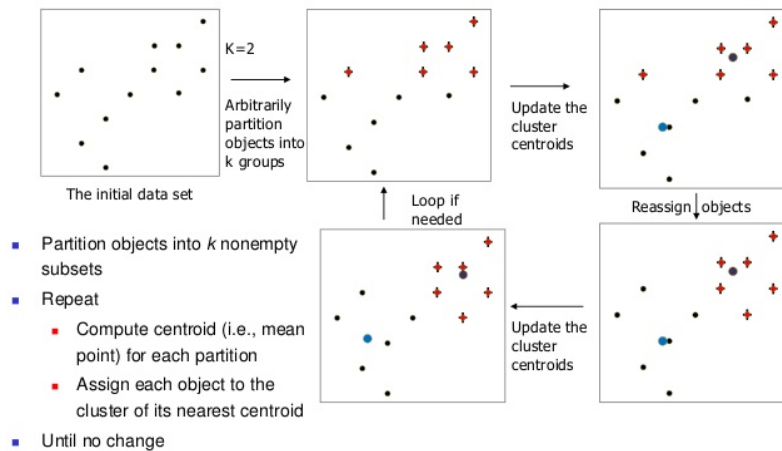


Figure 1: Visual explanation of K-Means Algorithm

An important variable of the algorithm is the SSE (Sum of Square Errors) which is nothing but the sum of the distances between the elements and their relative centroids. The simplest implementation of this algorithm puts the focus on the minimization of this variable.

$$SSE = \sum_{i=1}^k \sum_{j \in C_i} (c_i - x)^2 \quad k \text{ cluster sets, } C_i \text{ cluster centroid}$$

Required data

Since the computation of centroids is done through Euclidean distance, which is the best choice that brings to maximize the internal cohesion (also minimizing cluster dispersion), data must be reals. It's noteworthy that this algorithm can be also used with mixed values (categorical and continuous) using hybrid forms of distances that, unfortunately, give the most of the importance to discrete attributes, raising a polarization of the results.

In order to obtain better results, normalization of data is suggested, since the Euclidean distance would give an higher priority to attributes with higher values.

Drawbacks

Although K-Means clustering converges very rapidly, it adopts a greedy strategy, indeed it's highly dependent to the initial condition that is the random pick of the centroids. As a matter of fact, we are not certain, that after running the algorithm, the resulting SSE is the best that could be obtained from the chosen dataset since the execution of this method can incur in a local minimum.

To counteract these problems, some remedies have been conceived and they will be exposed in the next paragraphs as they have been used in this work.

Greedness Remedies

- **Multiple attempts**

It is a good practice to reduce the probability of incurring in a local minimum by running multiple executions with different initial conditions. Different resulting SSE will come out from each instance, giving the possibility to pick the lowest one among instances results.

- **Data preparation**

This clustering algorithm is sensible to outliers, so it's suggested to remove them in this phase.

- **Choice of K**

We could set the value of K depending on the SSE which has to be minimized. Clearly this is not a good approach because the higher the K, the lower the SSE so it cannot be used as a merit figure, indeed when K is equal to N (Number of samples) SSE becomes zero. In order to get good results, statistic tools come to the aid penalizing higher number of clusters; such as *BIC* (Bayesian Information Criterion) and *AIC* (Akaike Information Criterion), that unfortunately don't work well in case of high dimensionality. Hence, final decision was to use the Silhouette coefficient as score index, this value (that belongs to -1 and +1) measures the internal cohesion and external separation, and the higher it is, the better the relationship between these two values. In addition, to be still more accurate in the choice of K, we could make a grid search based on the *cross-validation* method. Shortly, this approach creates, for ten times, two subsets of the dataset which are the training set (9/10 fold of the original one) and the test set (the remaining 1/10) measuring the SSE computed on the test set using the centroids learned on the training set. Of course, also this approach tends to favor higher K, so we decided to substitute SSE value with the Silhouette score calculation.

Stopping criteria

As this algorithm is iterative, it would continue looping even if the improvements in terms of SSE would be negligible (wasting resources), so few stopping criteria are needed.

Two, more than others, are present in the scientific community, but the combination of them could be a better choice.

- **Test on the iterates**

This means that the algorithm must stop at the n^{th} iteration (with n fixed) avoiding the loop to diverge.

- **Test on the SSE gain**

Stopping if the percentage gain between the current iteration and the next one is higher than a certain tolerance, avoids the algorithm to execute useless iterations.

Hardware specifications

Parallel computations will be executed only on google cloud platform server to avoid timing errors given using different Hardware/Software configurations.

	Macbook pro [serial execution]	Google Cloud Platform server [Parallel executions]
OS	Ubuntu 16.04 LTS	Ubuntu 16.04 LTS
Compiler	GCC-7 (ver: 7.2.0)	GCC-7 (ver: 7.2.0)
CPU	Intel Core i5 (I5-4258U)	Skylake (Intel Xeon scalable processor)
CPU Speed	2,4 GHz	2,00 GHz
Number of processors	1	2, 4, 8, 16, 32, 64 (as needed)
Number of cores	2	Same as above
Cache L1	32 KB (per core), 2-way associativity	32 KB (per core), 8-way associativity
Cache L2	256 KB (per Core), 8-way associativity	256 KB (per Core), 8-way pipelined
Cache L3 (shared)	3 MB (shared), -way associativity	56 MB (shared), 20-way associativity
RAM memory	8 GB	1.8 GB, 3.6 GB, 7.2 GB, 14.4 GB, 28.8 GB, 57.6 GB (related to number of CPUs)

Table 1: hardware specifications

Software employed

- **Matlab** (ver R2017b): data generation
- **Orange** (ver 3.11.3): algorithm validation
- **Intel vTune Amplifier** (ver 2018 update 1): serial code general exploration and hotspots hunting
- **Intel Inspector** (ver 2018): check data races in parallel regions

Analysis of the serial algorithm

Introduction

Even if not required, the serial code was developed by ourselves. We decided to not use pre-made code primarily because debugging parallel code born from unknown sources has two potential roots of problems: logic errors in the algorithm or implementation, and threading problems in the code. They are difficult to classify as caused by data race or because the code is not incrementing through a loop enough times, especially if code is not known inside and out. By the way, *Intel inspector tool* could help finding this kind of problems.

Employed datasets

As k-means algorithm is strongly related to data structure, we chose to employ two different datasets coming from the scientific community, taken as standard for Clustering in order to verify the robustness of results independently from the dataset provided.

The chosen datasets are Breastcancer_Wisconsin and Iris, both easily retrievable from the web.

An accurate analysis with the aim of evaluating central tendency, variability and correlation measures among data should be mandatory before executing a machine learning algorithm, but it would be dangerous as it could make this report prolix and off the track.

A summary of data characteristics is given below.

Breast cancer Wisconsin

This dataset (30,7 KB) consists of 683 samples 9 attributes each and its structure is shown below. It collects specific data of several patients.

Features	Uniformity of cell shape	Real
	Uniformity of cell size	Real
	Clump thickness	Real
	Bare nuclei	Real
	Cell size	Real
	Normal nucleoli	Real
	Clump cohesiveness	Real
	Nuclear chromatin	Real
	Mitoses	Real
	Class	Nominal
Classes	Benign	
	Malignant	
Number of Missing Values	16	
Number of Instances	699	

Table 2: Breast cancer Wisconsin dataset structure

Iris

This dataset (2,3 KB) consists of 150 samples and 4 attributes each and its structure is shown below. It collects measures data of several flowers.

Features	Sepal length in cm	Real
	Sepal width in cm	Real
	Petal length in cm	Real
	Petal width in cm	Real
	Class	Nominal
Classes	Iris setosa	
	Iris versicolour	
	Iris virginica	
Number of Missing Values	0	
Number of Instances	150	

Table 3: Iris dataset structure

Since clustering is an unsupervised method, it doesn't require class value, hence omitted in our analysis. Moreover, the samples featuring missing data, have been removed.

Data generation

At this point, two low dimension datasets are not enough to quantify the scalability results in next pages, so we decided to generate new *believable facsimile* data starting from the original dataset, using a non-conventional approach.

Starting from the k clusters found by the serial algorithm from original datasets, k distributions of data were created and, with their means and covariance matrices, a sampling from *normal* distribution was made, allowing to create bigger datasets with similar characteristics of the original ones. Unfortunately, this is a supervised way to approach the problem due to class presence, that is not advisable in machine learning field, but that will work well for our testing purposes.

Finally, we came out with 7 different datasets for Breastcancer and 8 for Iris, that have the following dimensions:

- **Breastcancer** dataset: 30, 165, 320, 445, 590, 730, 902 [KB]
- **Iris** dataset: 2, 19, 36, 53, 70, 87, 104, 122 [KB]

Algorithm flowchart

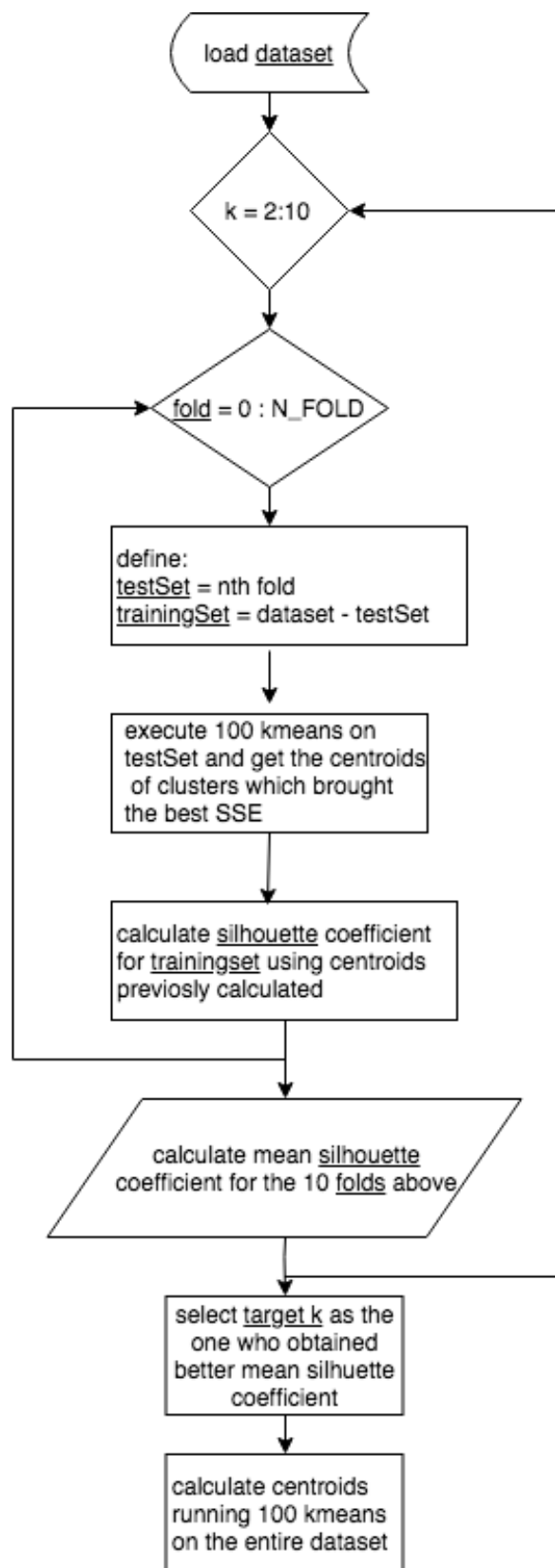


Figure 2: high-level flowchart

Results

Cluster resulting by running the algorithm on provided datasets are shown below. Similar results are obtained by Orange machine learning tool.

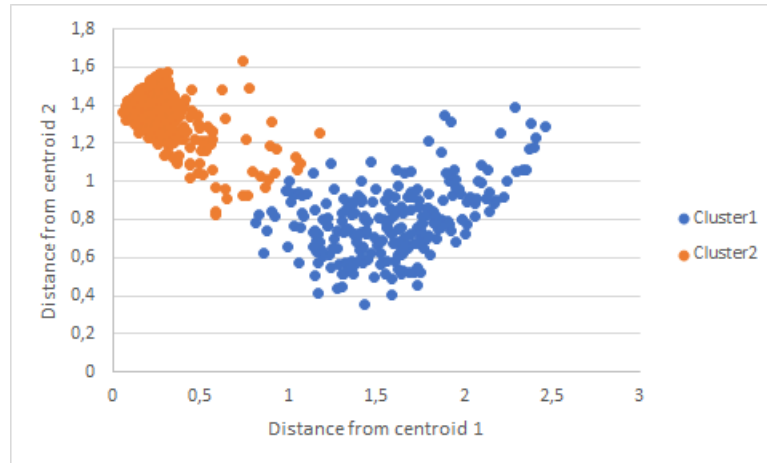


Figure 3: Results of the algorithm on Breastcancer dataset

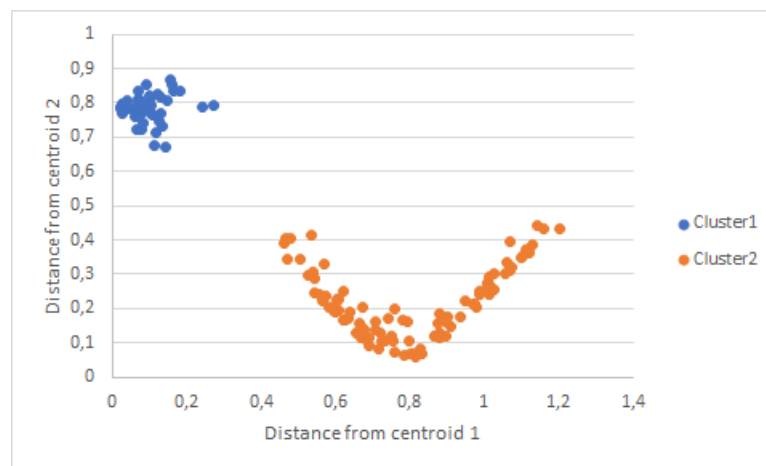


Figure 4: Results of the algorithm on Iris dataset

Although Iris dataset is characterized by 3 classes, only two clusters are identified by the algorithm. This is not a sign of failure, but it indicates that clusters cannot be discernible.

General exploration

A deep analysis of the serial algorithm was helpful to identify the major hotspots of the whole program. That means the costly functions (hotspots) and pieces of codes that, obviously, require an improvement using parallelism.

For that purpose, a profiling tool was used, that is Intel vTune Amplifier.

A simple pie chart with percentages that shows the CPU usages of the hotspots is given below. These results are the mean of 4 measurements on most of the available datasets.

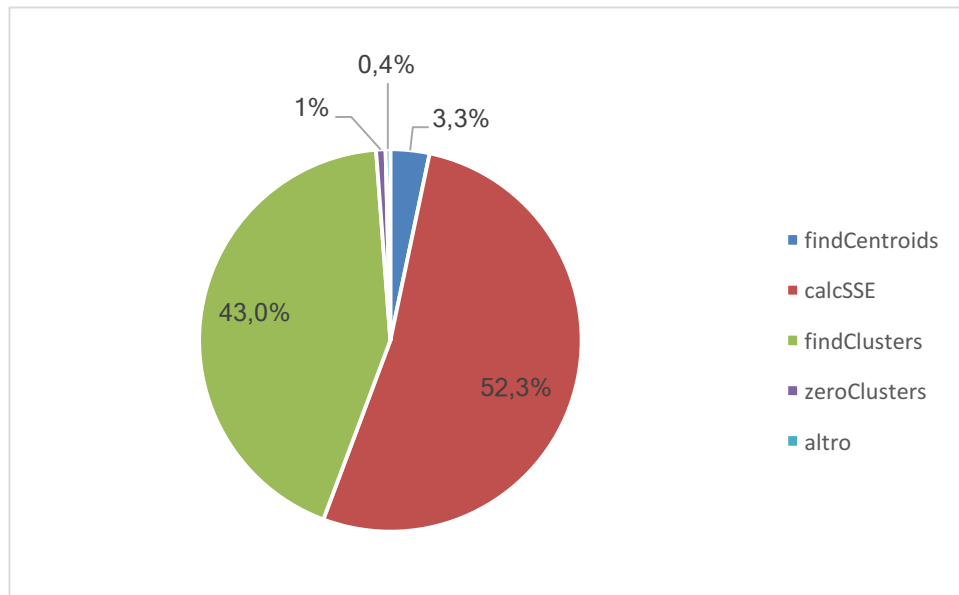


Figure 5: Hotspots of the serial algorithm

Performing measurements with vTune tool a small overhead can occur between the real execution time and the one resulted from the analysis. Therefore, this error is adjusted by the tool itself, hence acquired measurements can be considered sufficiently accurate.

Here some additional results obtained by profiling the code.

		Breastcancer(30kB)	Breastcancer(902 kB)	Iris(2 kB)	Iris (122kB)
Clockticks ($\times 10^6$)		318650	591500	27047,5	1415162,5
Instructions retired ($\times 10^6$)		591550	1055202	51967,5	2671240
CPI rate		0.539	0.561	0.520	0,530
Front-End bound (Fetch and decode phases)	Unknown branches ²	53,4% of Clockticks	84,4% of Clockticks	N.A.	53% of Clockticks
	Mispredicts Resteers ³	1,1% of Clockticks	2,7% of Clockticks	1,7% of Clockticks	1,7% of Clockticks
Bad speculation	Branch mispredict ⁴	4,6% of Pipeline slots ¹	8,2% of pipeline slots ¹	7% of pipeline slots	7,7% of pipeline slots
	Machine Clears ⁵	0,8% of Pipeline slots ¹	0,4% of pipeline slots ¹	0,4% of pipeline slots ¹	0,5% of pipeline slots ¹
Back-end bound	Port Utilization ⁶	29% of Clockticks	13,2% of Clockticks	24,4% of Clockticks	24,7% of Clockticks

Table 4: Additional profiling data

¹**Pipeline slots:** A pipeline slot represents hardware resources needed to process one uOp.

²**Unknown branches:** This metric measures the fraction of cycles the CPU was stalled due to new

branch address clears. These are fetched branches the Branch Prediction Unit (BPU) was unable to recognize (First fetch of hitting BPU capacity limit).

³**Mispredicts resteeers:** this metric measure the fraction of cycles the CPU was stalled due to Branch Resteeers as a result of Branch Misprediction at execution stage.

⁴**Branch misprediction:** When a branch mispredicts, some instructions from the mispredicted path still move through the pipeline. All work performed on these instructions is wasted since they would not have been executed had the branch been correctly predicted. This metric represents slots fraction the CPU has wasted due to Branch Misprediction. These slots are either wasted by uOps fetched from an incorrectly speculated program path, or stalls when the out-of-order part of the machine needs to recover its state from a speculative path.

⁵**Machine clears:** Certain events require the entire pipeline to be cleared and restarted from just after the last retired instruction. This metric measures three such events: memory ordering violations, self-modifying code, and certain loads to illegal address ranges. Machine clears metric represents slots fraction from CPU has wasted due to machine clears. These slots are either wasted by uOps fetched prior to the clear, or stalls the out-of-order portion of the machine needs to recover its state after the clear.

⁶**Port utilization:** This metric represents a fraction of cycles during which an application was stalled due to core non-divider-related issues. For example, heavy data-dependence between nearby instructions, or a sequence of instructions that overloads specific ports. Hint: Loop Vectorization – most compilers feature auto-Vectorization options today – reduces pressure on the execution ports as multiple elements are calculated with same uOp.

Memory Consumption

Function	Allocation Size	Deallocation Size	Allocation/Deallocation Delta
kmeans	110 MB	110 MB	0 KB
datasetSubSets	2 MB	2 MB	0 KB
main	484 KB	480 KB	4 KB
mainAlgo	170 KB	170 KB	0 KB
loadDataset	125 KB	71 KB	54 KB
writeFile	4 KB	0 KB	4 KB
TOTAL	113 MB	113 MB	63 KB

Table 5: Allocation and deallocation of memory by functions

As we can see from the previous table, the memory allocated is cleared before the program ends, so its efficient use is guaranteed. The data provided are only related to the serial code with dataset Breastcancer (30KB). What is important to be noted is that almost all memory allocated is freed. Although not shown in this report, same results were achieved with bigger datasets.

A-priori study of the available parallelism

A bottom up approach

Starting from the results of the hotspots obtained in the [Analysis of the serial algorithm](#), we decided to parallelize the highly demanding CPU consumers, implementing a cost/benefits viewpoint. Of course, parallelizing all regions could be a possible way, but it is dangerous considering the overhead (brought by forking operation and variable sharings), if compared with the performance improvement.

As a remark, we bring to your notice that the hotspots found for the serial code are:

- **calcSSE** [which calls eucliDist ,getRow](52,30 %)
- **findClusters** [which calls eucliDist, getRow](43,00 %)
- **findCentroids** (3,30 %)
- **zeroCluster**(1,00 %)

Using a bottom up approach several attempts have been made, trying to parallelize the lowest level functions (eucliDist, getRow), but as they didn't bring improvements, the upper levels were enhanced: calcSSE and findClusters, together with findCentroids.

Although zeroCluster is one of the hotspots, we chose to not parallelize it because of its I/O nature, in fact, no improvements can be achieved from such a kind of method because of the huge difference of speed between I/O operations and CPU-based ones. The reason why I/O operations (especially Output, that is writing to memory) are not efficiently parallelized is because this operation is mutually exclusive done by one thread at a time.

Data structure

In the following we will show what are the functions interested and the data accesses that will be parallelized, we will discuss how this could be done from a theoretical point of view and the implicit problems that could raise up during parallelization process.

Function "calcSSE"

```
float calcSSE(float** dataset, int **clusters, float** centroids, int n, int m){
    float sum=0.0, supDataset[m], supCentroid[m];
    for(int ki = 0; ki < k; ki++){
        getRow(centroids, ki, supCentroid, m);
        for(int i = 0; i < n; i++){
            getRow(dataset, i, supDataset, m);
            sum += eucliDist(supCentroid, supDataset, m) * clusters[i][ki];
        }
    }
    return sum;
}
```

- **n** : number of dataset rows ($n \gg 150$ samples in the most of cases)
- **k** : number of clusters to find ($k = 1 : 10$)

- **m** : number of dataset columns, they correspond to the samples attributes (m = 9 for breastcancer dataset, m = 4 for iris dataset)

In this function seems natural to implement a parallelization of the inner loop by adopting a simple data decomposition. The choice hasn't fallen upon the outer loop because of the low number of iterations computed by it. Indeed, the addition of code to compute chunks and assign them to threads would not worth the effort, since it adds overhead to the concurrent execution. In such a case it would be needed to increase granularity of the chunks by reducing the number of threads used.

By the way, data decomposition would be made on the float matrix dataset, which dimensions are given n (rows) * m (columns). A few rows of this matrix for breast cancer dataset are shown below, the whole matrix would be shared throughout the threads, but only one could access to one row, since the iteration variable i is private.

Uniformity of cell shape	Uniformity of cell size	Clump thickness	Bare nuclei	Cell size	Normal nucleoli	Clump cohesiveness	Nuclear chromatin	Mitoses
4.05	0.05	0.48	0.13	1.46	0.79	2.14	0.83	0.93
4.86	3.90	3.14	4.73	6.04	9.62	2.25	1.34	0.81
2.48	0.09	0.11	0.40	1.94	1.98	2.56	0.87	0.73
5.33	7.14	7.96	0.29	2.12	3.10	2.78	6.68	0.79
3.13	0.09	0.74	2.49	1.51	0.61	2.43	0.80	0.36
7.34	9.81	9.85	7.29	6.37	9.07	8.38	6.46	0.32
0.41	0.27	0.94	0.10	1.84	9.55	2.35	0.88	0.47

Table 6: A portion of breastcancer dataset matrix

Function “findCentroids”

```
//read cluster matrix and calculate centroids as mean of element's data belonging to same cluster
void findCentroids(float** centroids, int **clusters, float** dataset, int n, int m) {
    int elemCluster = 0;
    float record[m];
    for(int ki=0; ki<k; ki++) {
        //reset record array
        for(int p = 0; p < m; p++) {
            record[p] = 0;
        }
        for(int i = 0; i < n; i++) {
            if(clusters[i][ki] != 0) {
                elemCluster++;
                for(int j = 0; j < m; j++) record[j] += dataset[i][j];
            }
        }
        for(int p = 0; p < m; p++) {
            if(elemCluster != 0) record[p] = record[p] / elemCluster;
            else record[p] = 0;
            centroids[ki][p] = record[p];
        }
        elemCluster = 0;
    }
}
```

- **ki** : integer variable used as index in the outer loop, which points to the current centroid that is being computed (ki = 1 : 10)
- **p** : the index for scanning columns of the dataset and centroids records (p = 0 : m)
- **elemCluster**: counter of elements belonging to the current analyzed cluster (elemCluster = 0 : n)

By executing this function, the software computes the centroids of the current iteration of k-means algorithm. In order to do so, this piece of code counts the number of samples belonging to each cluster and computes the mean value for every feature of the dataset.

Looking better to the function, the outer loop scans the data structures on the k dimension which is usually a small number (from 2 to 10), so we decided to deepen the analysis to the inner level. The first inner loop is mostly composed by memory bound operations, which would make the enhancement nullified. On the other hand, the second and the third inner loops are well parallelizable sections of code. The second scans the dataset matrix sample by sample which, in the case of big dimension data structures, a parallel version would give a big improvement in performance. As a matter of fact the third loop doesn't scan big amounts of data, since the number of attributes is usually pretty restrained, but it executes multiple float divisions. This regards, in order to make the best choice, a deep profiling analysis would be needed.

An example of centroid matrix for breast cancer dataset is shown below.

Centroid of cluster 1	0,66	0,61	0,61	0,51	0,49	0,71	0,54	0,54	0,21
Centroid of cluster 2	0,25	0,07	0,08	0,08	0,16	0,08	0,15	0,07	0,06

Table 7: Centroids matrix example

Function “findClusters”

```
// it assigns each dataset record to the nearest centroid, which corresponds to its cluster
void findClusters(float** dataset, int **clusters, float **centroids, int n, int m){
    int salvaK = 0;
    float supCentroid[m], supDataset[m], dist=0, lowerDist;
    for(int i = 0; i < n; i++){
        lowerDist = m;
        // lowerDist = m because data is normalized
        // (the max dist between 2 records can only be equal to the number of attributes)
        for (int ki = 0; ki < k; ki++){
            getRow(centroids, ki, supCentroid, m); //extract a row from the centroid matrix
            getRow(dataset, i, supDataset, m); //extract a row from the dataset matrix
            dist = eucliDist(supCentroid, supDataset, m); //computing the euclidean distance
            if(dist <= lowerDist)
            {
                lowerDist = dist;
                salvaK = ki;
            }
        }
        clusters[i][salvaK] = 1;
    }
}
```

The function shown above is the portion of code that computes the actual result of the algorithm. Finding clusters is, indeed, the aim of this software. To establish the membership of a sample to a certain cluster, the distance between each sample and each centroid is computed and the lowest is

considered. The data structure where the membership status is stored is the *clusters* float bi-dimensional array, which has as dimension *dataset dimension* \times *number of clusters*. To flag the status of each sample, a 1 has been set to establish its participation to a certain cluster, while a 0 is put on all the remaining columns. To better understand its structure, a portion of an example of this matrix follows.

Sample	Cluster 1	Cluster 2
129	1	0
130	0	1
131	1	0

Table 8: Portion of an example of cluster matrix

The parallelization of this function is not trivial, because it would involve a large number of private data structures in order to maintain consistency of computation, which implies the copy of all of them for each thread, causing overhead. The main loop scans and computes distances between all the samples of the dataset, so implementing a parallel version of this function would certainly bring an enhancement.

Order of the various approaches proposed is not related to their performances and a comparison will be held in order to find the best one.

OpenMP parallel implementation

Data acquisition

Every acquisition made in this section of the report has been repeated 3 times, the mean was computed and then inserted into the values tables. This is due to the fact that even if no hardware changes are made, consecutive runs of the same application with the same inputs can yield to different results.

First implementation - for parallel [automatic scheduling]

In the first approach that we implemented, it was decided to open a parallel region and apply the *for* construct to the loops that contain the costliest operations such as mathematical functions like square roots, powers and divisions and that, of course, have been identified as independent computations.

This version of the code can be considered as the simplest one since the control of scheduling is demanded to the compiler, but this doesn't prevent to obtain good results since the compiler would probably take the right decision about concurrency policies.

The parallelized functions are listed below:

```
float calcSSE(float** dataset, int **clusters, float** centroids, int n, int m){
    float sum = 0.0, supDataset[m], supCentroid[m];
    for(int ki = 0; ki < k; ki++){
        getRow(centroids, ki, supCentroid, m);
        #pragma omp parallel for private(supDataset) reduction(+:sum)
        for(int i = 0; i < n; i++){
            getRow(dataset, i, supDataset, m);
            sum += eucliDist(supCentroid, supDataset, m) * clusters[i][ki];
        }
    }
    return sum;
}
```

```
void findCentroids(float** centroids, int **clusters, float** dataset, int n, int m) {
    int elemCluster=0;
    float record[m];
    for(int ki=0; ki<k; ki++) {
        for(int p = 0; p < m; p++) {
            record[p] = 0;
        }
        for(int i = 0; i < n; i++) {
            if(clusters[i][ki] != 0) {
                elemCluster++;
                for(int j = 0; j < m; j++)
                    record[j] += dataset[i][j];
            }
        }
        #pragma omp parallel for
        for(int p = 0; p < m; p++) {
            if(elemCluster != 0)
                record[p] = record[p] / elemCluster;
            else
                record[p] = 0;
            centroids[ki][p] = record[p];
        }
        elemCluster = 0;
    }
}
```

```
void findClusters(float** dataset, int **clusters, float **centroids, int n, int m){
    int salvaK = 0;
    float supCentroid[m], supDataset[m], dist = 0, lowerDist;
    #pragma omp parallel for private(lowerDist, dist, salvaK, supCentroid, supDataset)
    for(int i = 0; i < n; i++){
        for(int ki = 0; ki < k; ki++){
            if (ki == 0) lowerDist = m;
            getRow(centroids, ki, supCentroid, m);
            getRow(dataset, i, supDataset, m);
            dist = eucliDist(supCentroid, supDataset, m);
            if(dist <= lowerDist)
            {
                lowerDist = dist;
                salvaK = ki;
            }
            if(ki == k - 1) clusters[i][salvaK] = 1;
        }
    }
}
```

	Time [s]				
Size (kB)	Serial	2 vCPU	4 vCPU	8 vCPU	16 vCPU
30	84,47	67,34	40,32	26,13	20,09
165	415,85	331,83	196,74	115,74	75,84
320	743,09	542,84	324,58	187,71	121,55
445	1197,45	811,52	490,03	279,62	179,28
590	1311,99	1005,76	591,00	340,25	217,31
730	1730,11	1315,01	789,16	451,32	285,84
902	2060,84	1595,99	963,14	546,93	346,00

Table 9: Breastcancer dataset, elapsed times for first approach

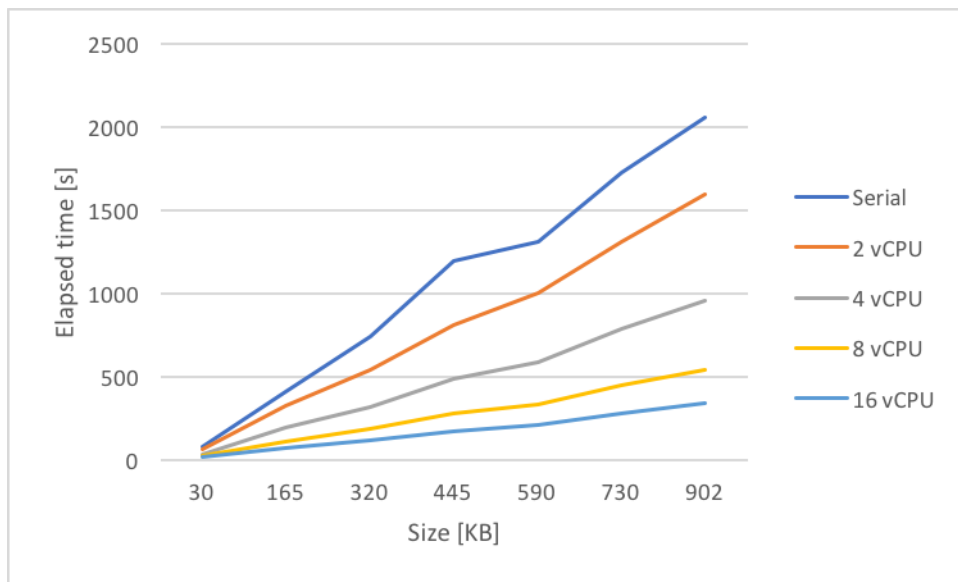


Figure 6: Line chart – data: Breastcancer [first implementation]

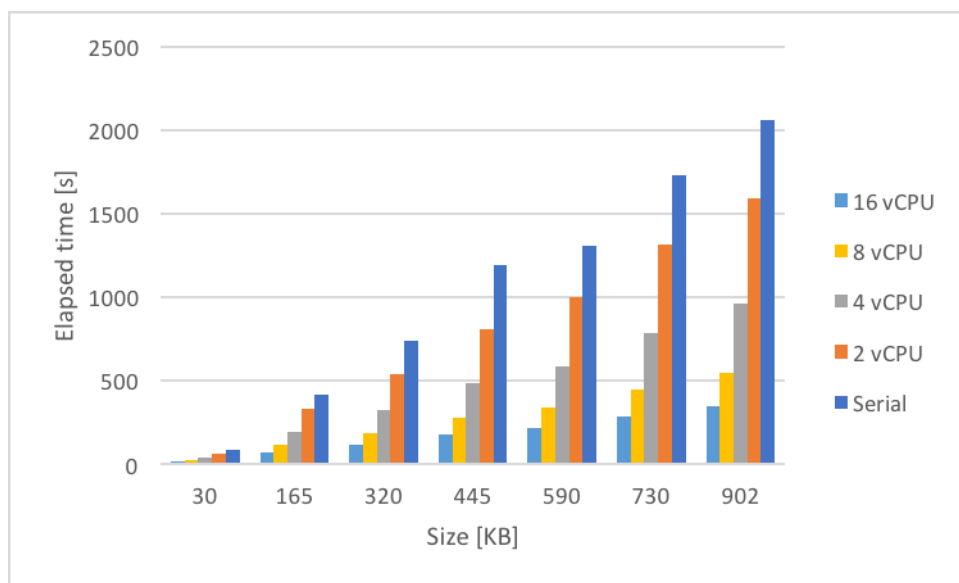


Figure 7: Bar chart – data: Breastcancer [first implementation]

	Time [s]				
Size(kB)	Serial	2 vCPU	4 vCPU	8 vCPU	16 vCPU
2	11,05	7,08	5,81	5,06	6,17
19	80,08	44,72	29,34	19,16	16,12
36	161,94	88,61	57,51	36,38	27,66
53	248,61	135,10	86,72	53,98	39,68
70	335,71	182,35	118,26	72,98	52,50
87	415,68	223,30	143,41	88,89	63,71
104	500,35	315,18	173,55	106,27	74,91
122	568,05	359,42	189,81	117,65	82,92

Table 10: Iris dataset, elapsed times for first implementation

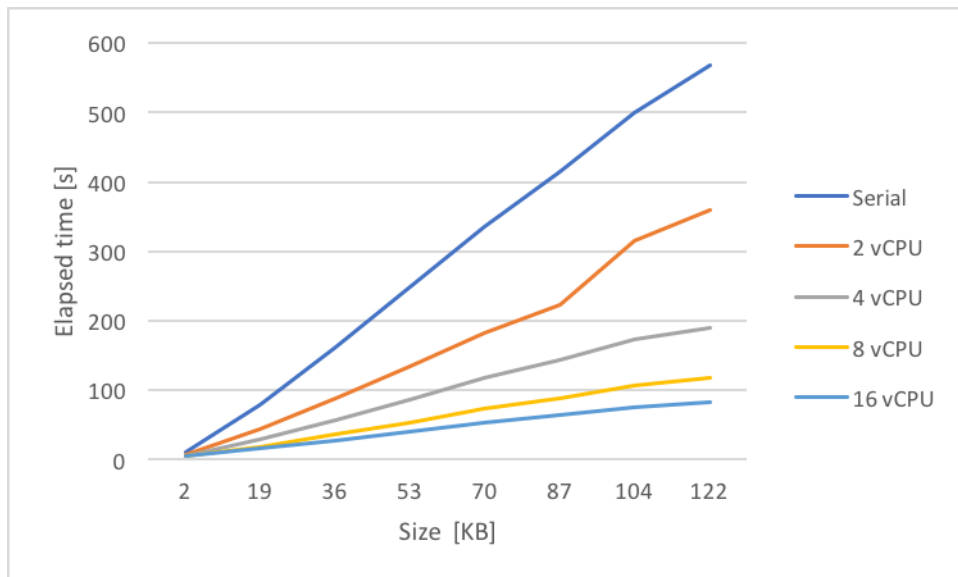


Figure 8: Line chart – data: Iris [first implementation]

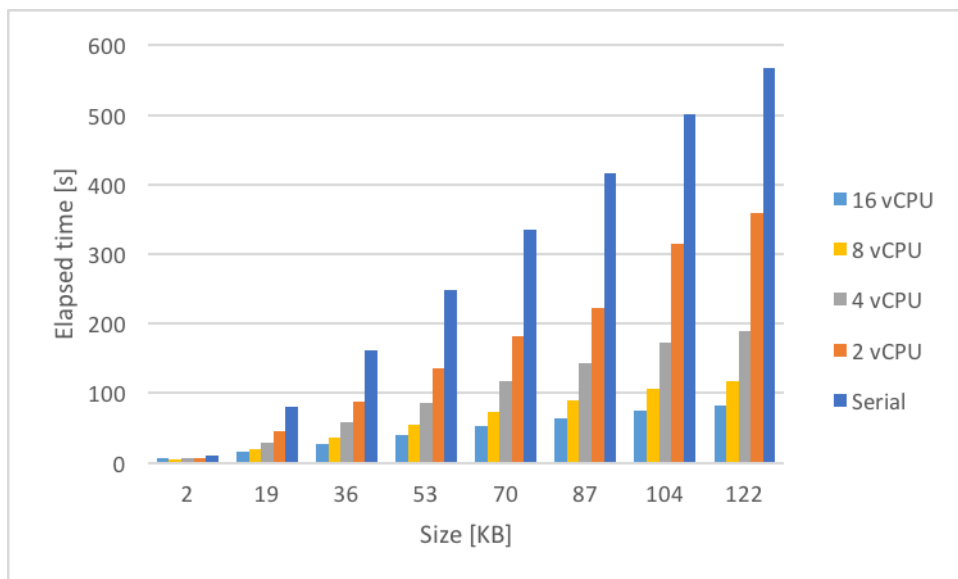


Figure 9: Bar chart – data: Iris [first implementation]

Second implementation - for parallel [custom scheduling]

This approach aims to customize the first one by introducing a manual configuration of the scheduling parameters. In the following one of these ways will be chosen and its results shown.

Chunksize value

An important parameter for this implementation is the chunksize value, that we set equal to the most intuitive way of dividing computations through threads, which is:

$$chunksize = \frac{\# iterations}{\# threads}$$

This would guarantee good performances in terms of scalability, limiting the performances from flattening out after a certain number of working threads.

Scheduling policy choice

To find out the best configuration, a parameter tuning of the possible scheduling policies is needed. Several configurations have been tested and their results for a typical situation are in the following:

	Static	Dynamic	Dynamic ^{FC}	Guided	Guided ^{FC}	Automatic
Time spent	65,84 s	93,72 s	65,52 s	65,57 s	67,20 s	65,78 s

Table 11: Execution time for different scheduling policies

*FC : configuration with specified chunksize and fixed to the value shown above.

To test these configurations, we put the software in a typical situation, analyzing medium dimension data (iris3.csv) in medium hardware configuration (4 threads). All measurements have been tested 3 times and the mean was taken into account.

Making exception for the non-custom dynamic approach, which brought a substantial worsening in terms of performances, no gain came up from other configurations, this is because the little improvement given is counteracted by the added overhead of the scheduling process.

Despite the previous results, Dynamic chunksize configuration was chosen to continue following profilings since it could be a good compromise for both high and low number of threads and high and low file dimension.

	Time [s]				
Size (kB)	Serial	2 vCPU	4 vCPU	8 vCPU	16 vCPU
30	84,47	75,94	42,56	27,33	20,19
165	415,85	364,98	209,74	122,65	78,12
320	743,09	608,36	339,79	200,81	124,90
445	1197,45	907,91	515,45	299,87	186,24
590	1311,99	1124,76	624,15	362,62	225,18
730	1730,11	1496,44	820,89	481,90	298,75
902	2060,84	1792,35	1000,25	586,46	363,53

Table 12: Breastcancer dataset, elapsed times for second implementation

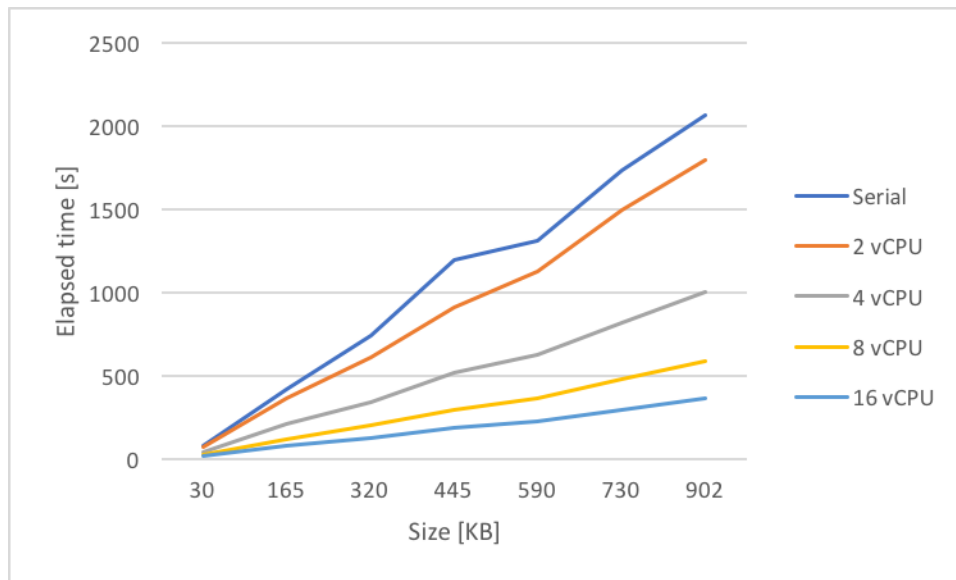


Figure 10: Line chart – data: Breastcancer [second implementation]

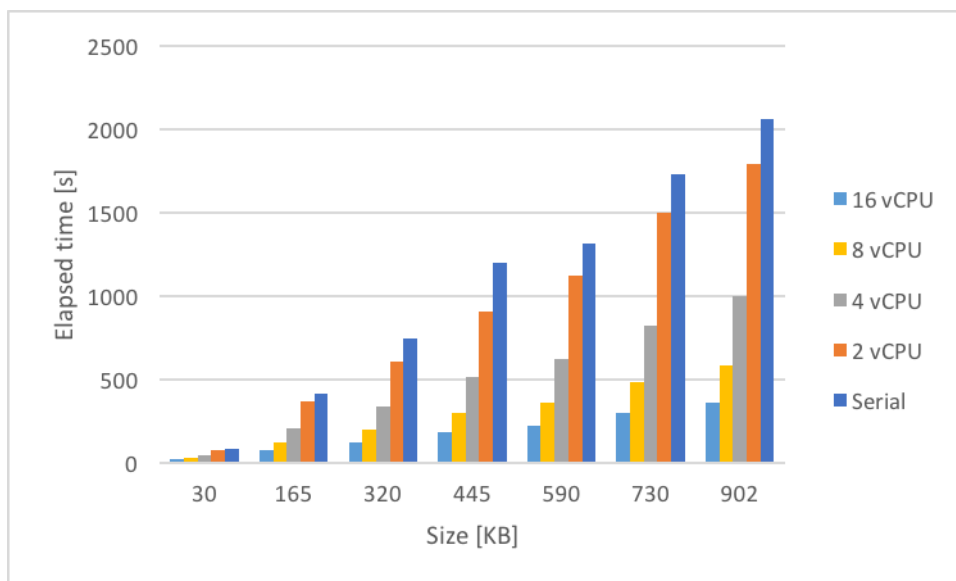


Figure 11: Bar chart – data: Breastcancer [second implementation]

	Time [s]				
Size(kB)	Serial	2 vCPU	4 vCPU	8 vCPU	16 vCPU
2	11,05	8,15	6,30	5,72	7,18
19	80,08	51,47	31,32	20,87	17,15
36	161,94	103,74	64,60	39,37	29,27
53	248,61	156,02	96,46	57,90	41,30
70	335,71	214,98	127,34	77,94	55,01
87	415,68	260,14	156,96	95,91	65,93
104	500,35	315,62	187,54	114,37	78,74
122	568,05	343,75	210,60	126,16	85,87

Table 13: Iris dataset, elapsed times for second implementation

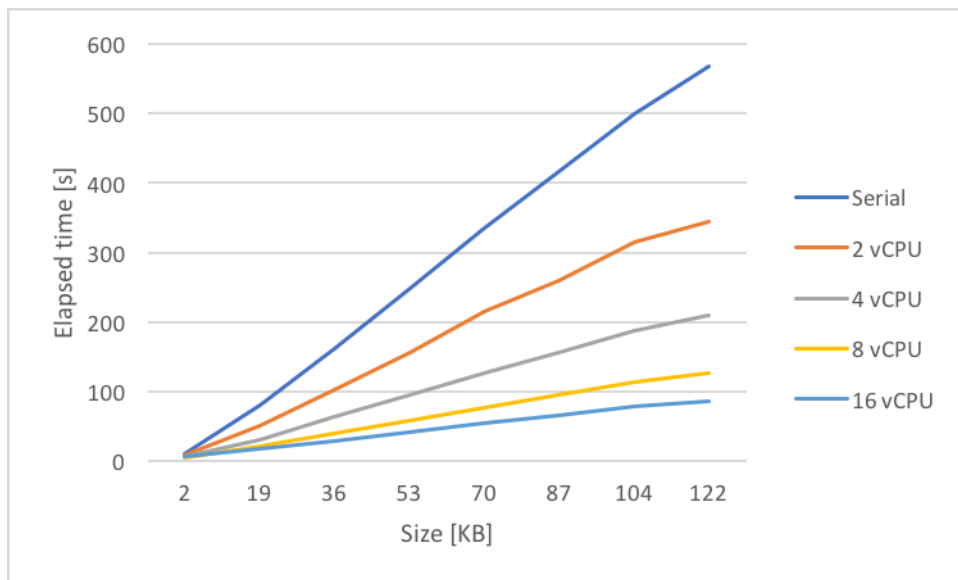


Figure 12: Line chart – data: Iris [second implementation]

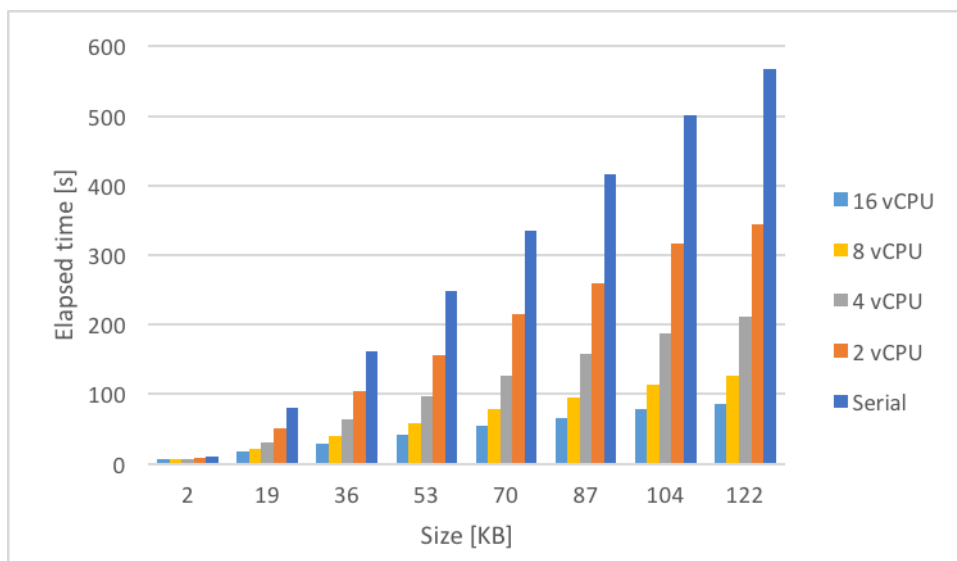


Figure 13 : Bar chart – data: Iris [second implementation]

Third implementation - for parallel with collapse [custom scheduling]

We identified a possible nested parallelism in the *calcSSE* function, so we designed on this a third implementation. In order to achieve such a result, the *omp_set_nested* clause should be called, but, since it's now deprecated, the new kind of directive *collapse* has been introduced. In this method *n* levels are collapsed into a unique iteration space. This means that iterations are distributed over *n*-tuples with $n = i \cdot j$.

This implementation has been primarily conceived for *testing purposes*, because this kind of option could not give the expected improvements in our case since:

- The amount of work to be managed in the inner loop is trivial, hence scalability will be limited by this factor.
- In our case, samples numerosity is pretty low, this affects the work of the collapse clause as the granularity is not optimal.

The loops must be perfectly nested; that is, there is no intervening code nor any OpenMP directive between the loops which are collapsed.

The function involved in this implementation is the following:

```
float calcSSE(float** dataset, int **clusters, float** centroids, int n, int m){
    float sum=0.0, supDataset[m], supCentroid[m];
    #pragma omp parallel for private(supCentroid, supDataset) collapse(2) reduction(+:sum)
    for(int ki = 0; ki < k; ki++){
        for(int i = 0; i < n; i++){
            getRow(centroids, ki, supCentroid, m);
            getRow(dataset, i, supDataset, m);
            sum += eucliDist(supCentroid, supDataset, m) * clusters[i][ki];
        }
    }
    return sum;
}
```

Figure 14: Portion of code with collapse clause

	Time [s]				
Size(kB)	Serial	2 vCPU	4 vCPU	8 vCPU	16 vCPU
30	84,47	29,32	47,47	27,84	18,45
165	415,85	378,21	233,60	138,47	81,66
320	743,09	624,35	385,63	223,85	132,18
445	1197,45	931,09	575,09	328,02	197,82
590	1311,99	1135,99	701,73	409,57	240,38
730	1730,11	1513,37	934,739	552,76	316,42
902	2060,84	1770,61	1093,62	643,48	384,11

Table 14: Breastcancer dataset, elapsed times for third implementation

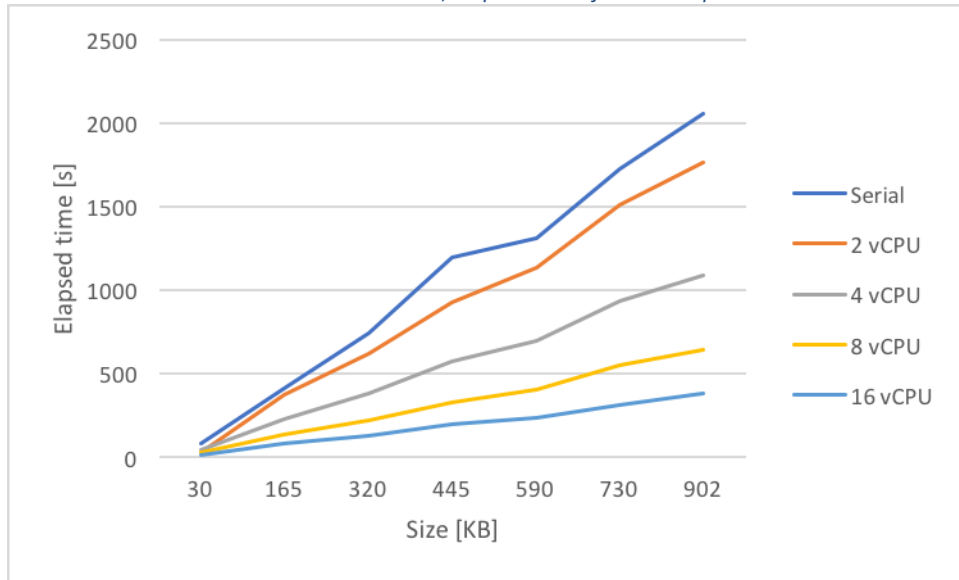


Figure 15: Line chart – data: Breastcancer [third implementation]

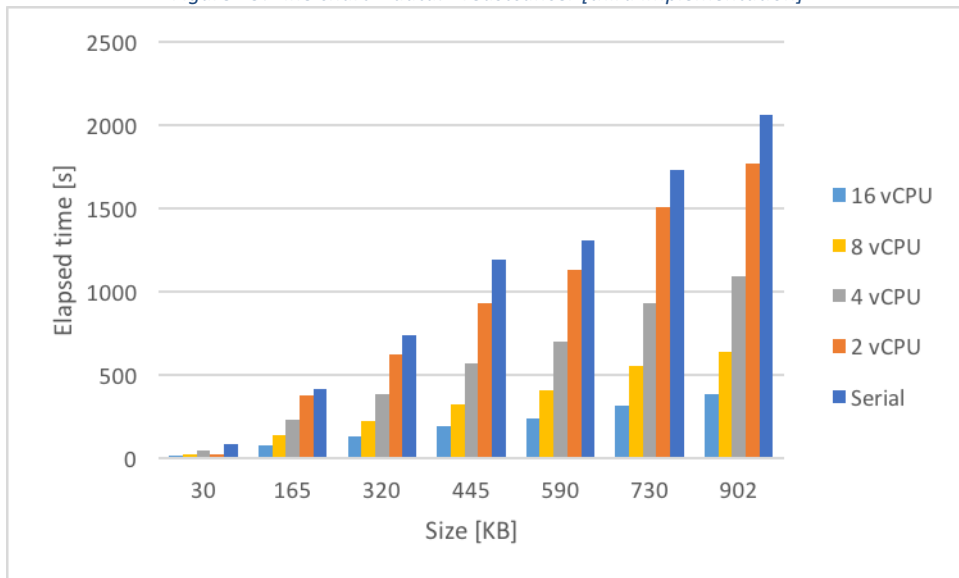


Figure 16: Bar chart – data: Breastcancer [third implementation]

	Time [s]				
Size(kB)	Serial	2 vCPU	4 vCPU	8 vCPU	16 vCPU
2	11,05	10,56	5,30	4,11	4,2
19	80,08	49,47	24,82	17,45	14,19
36	161,94	111,58	55,98	35,51	26,72
53	248,61	199,22	99,95	60,24	39,18
70	335,71	266,8	133,86	83,72	53,04
87	415,68	329,81	165,47	100,68	65,29
104	500,35	401,82	201,60	120,55	78,19
122	568,05	442,76	222,14	136,09	89,52

Table 15: Iris dataset, elapsed times for third implementation

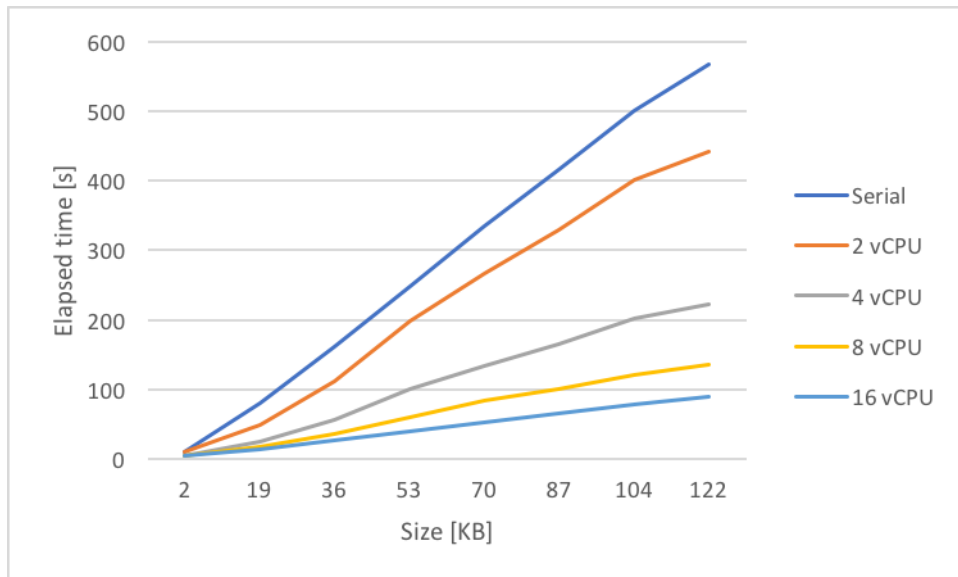


Figure 17: Line chart – data: Iris [third implementation]

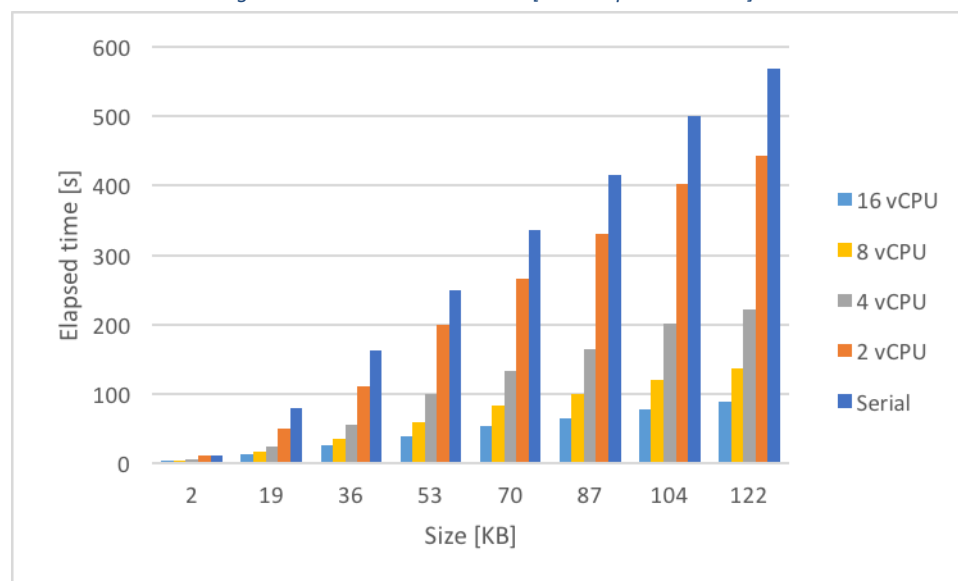


Figure 18: Bar chart – data: Iris [third implementation]

Profiling analysis of the Parallel algorithm

To compare the three implementations, only two hardware configurations have been considered: the 4 cores and the 16 cores ones. The results are shown in the plots below.

4-threads case - Breastcancer

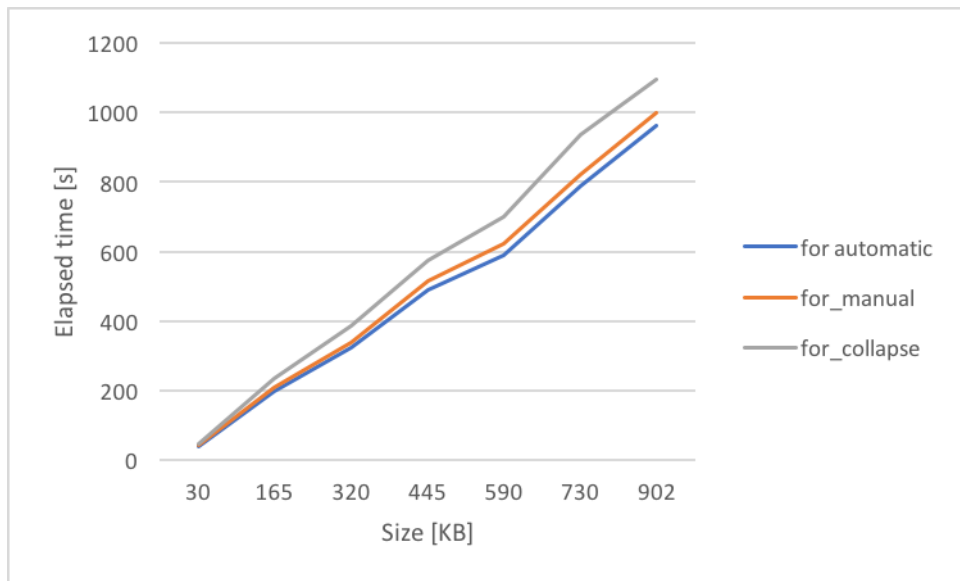


Figure 19: Performance comparison throughout implementations [breastcancer dataset, 4 threads case]

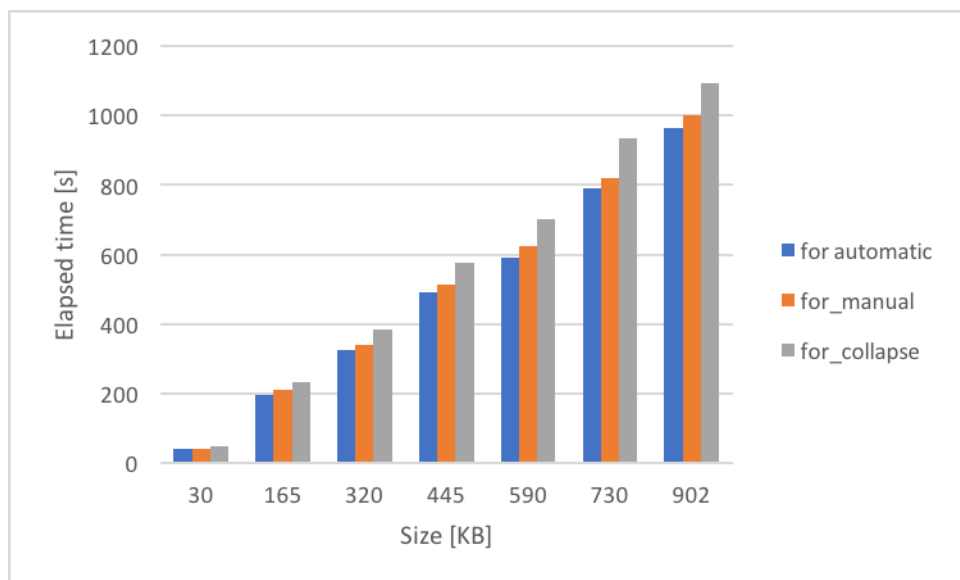


Figure 20: Performance comparison throughout implementations [breastcancer dataset, 4 threads case]

4-threads case - Iris

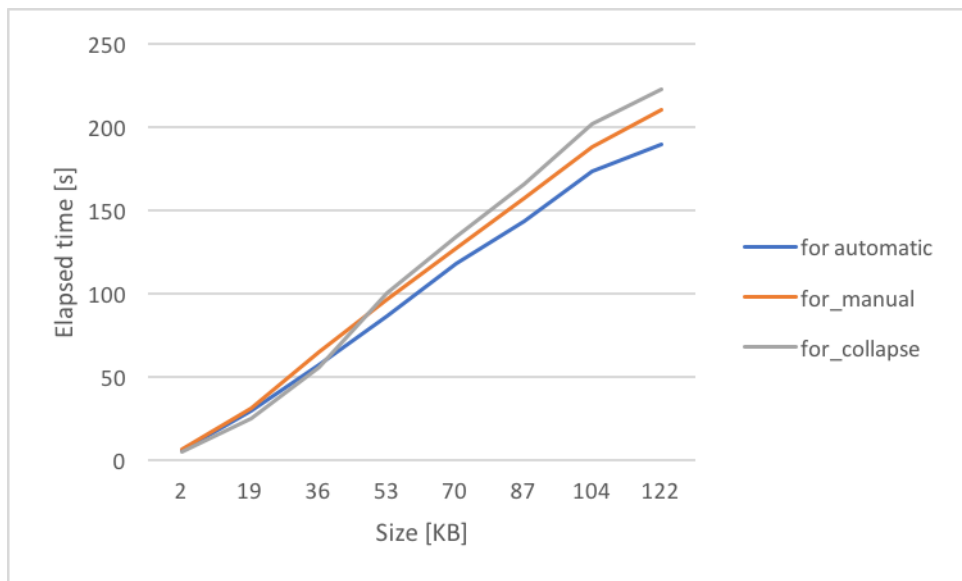


Figure 21: Line chart - Performance comparison throughout implementations [iris dataset, 4 threads case]

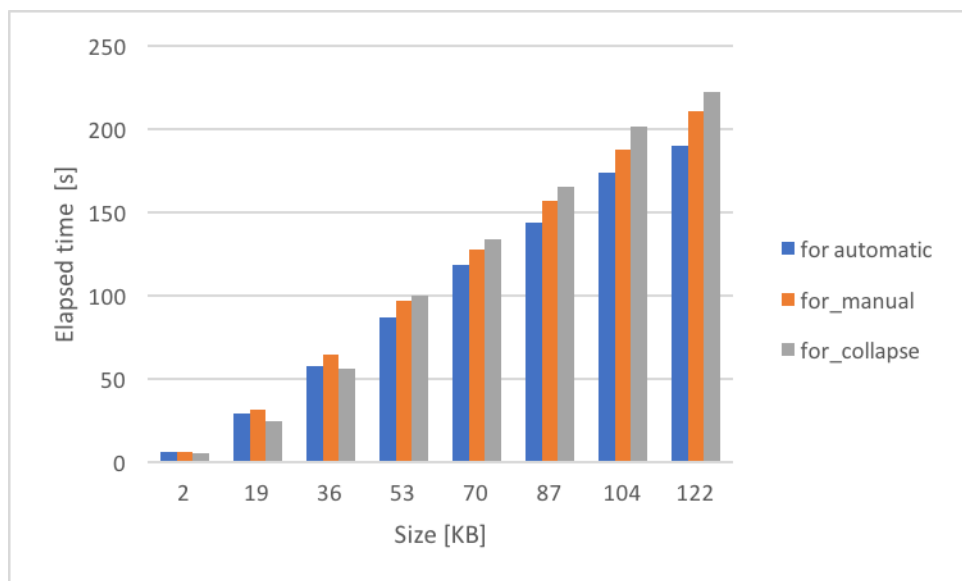


Figure 22: Bar chart - Performance comparison throughout implementations [iris dataset, 4 threads case]

16-threads case - Breastcancer

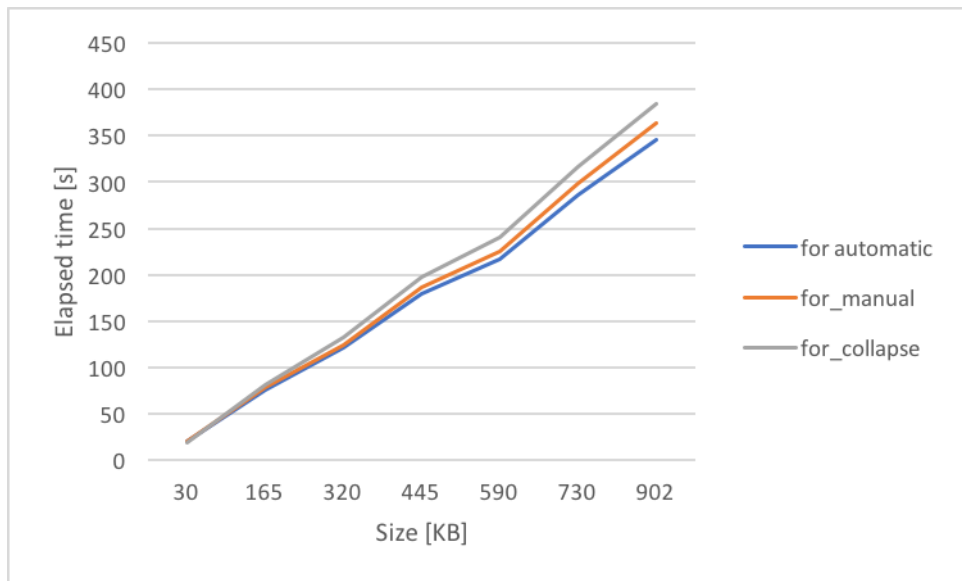


Figure 23: Line chart - Performance comparison throughout implementations [breastcancer dataset, 16 threads case]

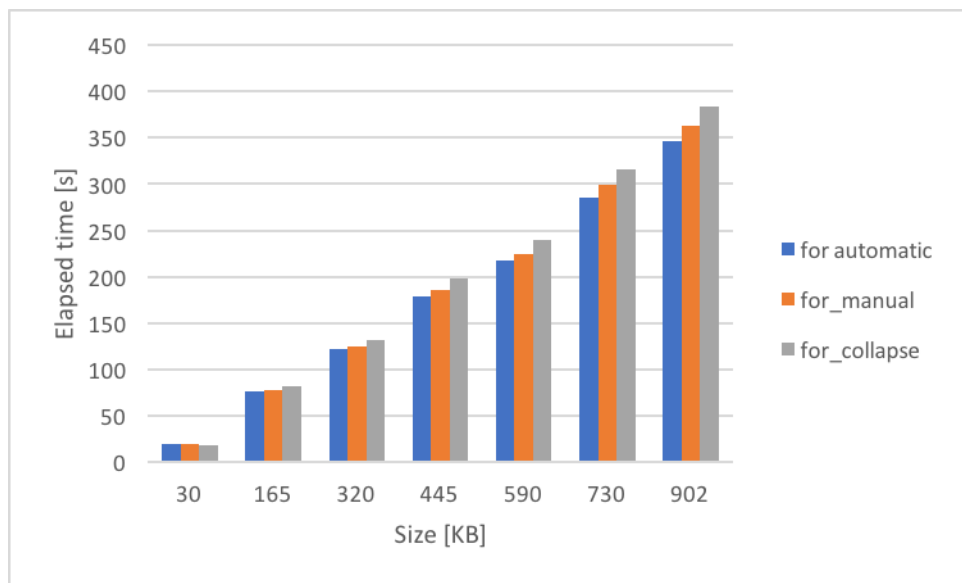


Figure 24: Bar chart - Performance comparison throughout implementations [breastcancer dataset, 16 threads case]

16-threads case – Iris

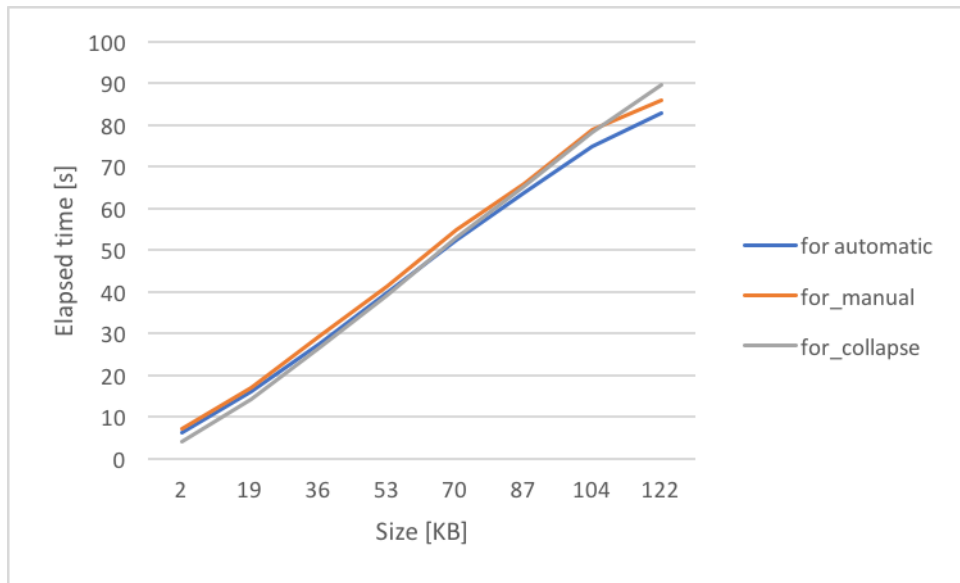


Figure 25: Line chart - Performance comparison throughout implementations [Iris dataset, 16 threads case]

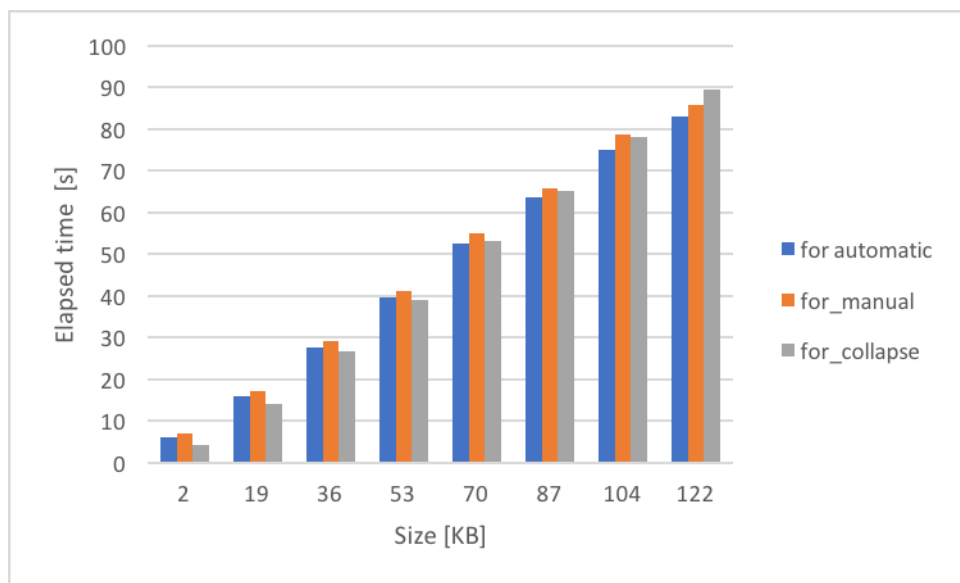


Figure 26: Bar chart - Performance comparison throughout implementations [Iris dataset, 16 threads case]

Since the best results have been obtained with the first approach, from now on only this one will be considered and analyzed. On the other hand, using all approaches for next analysis would be too expensive since we are using a paid service by google.

A strange behavior [high threads number case]

Now we will consider the particular behavior of such a parallelized program using a higher number of cores (32 and 64). By instinct, one could expect that the higher the number of them the better the performances, but it's not the way since, as we said before, the overhead introduced by parallelization grows with the number of threads.

	Time [s]		
Size(kB)	16 vCPU	32 vCPU	64vCPU
2	4,20	9,84	25,11
19	14,19	18,32	36,09
36	26,72	26,38	44,13
53	39,18	35,77	56,77
70	53,04	44,27	64,79
87	65,29	52,8	74,34
104	78,19	61,35	79,83
122	89,52	67,78	78,60

	Time [s]		
Size(kB)	16 vCPU	32 vCPU	64vCPU
30	20,09	23,88	40,65
165	75,84	61,01	64,72
320	121,55	92,62	88,77
445	179,28	132,74	121,21
590	217,31	150,88	140,91
730	285,84	197,32	178,4
902	346,00	240,01	210,49

Table 16: 16, 32, 64 vCPU on both datasets

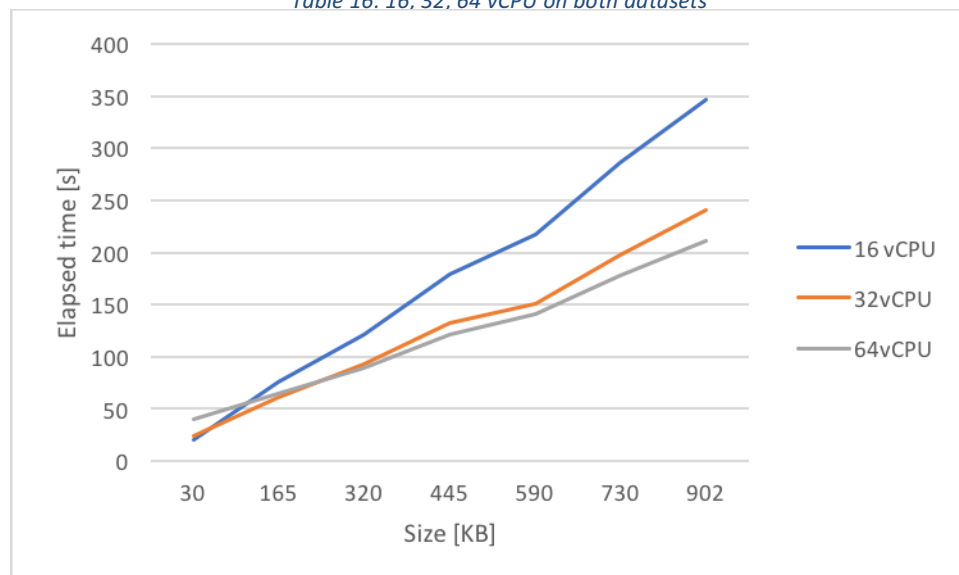


Figure 27: Line chart - Breastcancer, Comparison with high # of threads

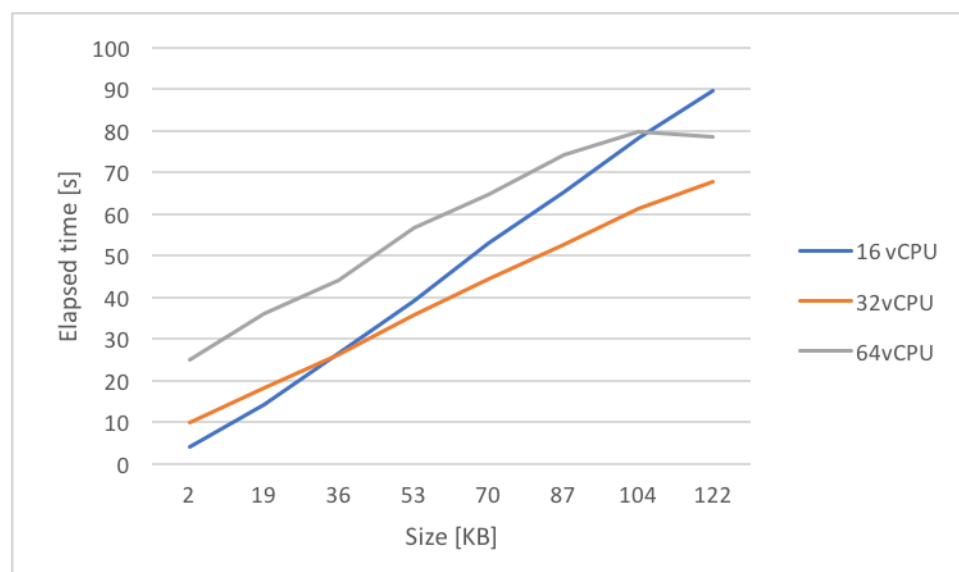


Figure 28: Line chart - Iris, Comparison with high # of threads

Optimizations

Thread number optimization

This section has the aim of counteracting the overhead caused by threads which hides the enhancement of performance. Indeed, if the data size remains fixed while the number of cores and threads increases, there will be a point (of diminishing returns) where dividing up data into smaller chunks for an high number of threads will yield worse performances.

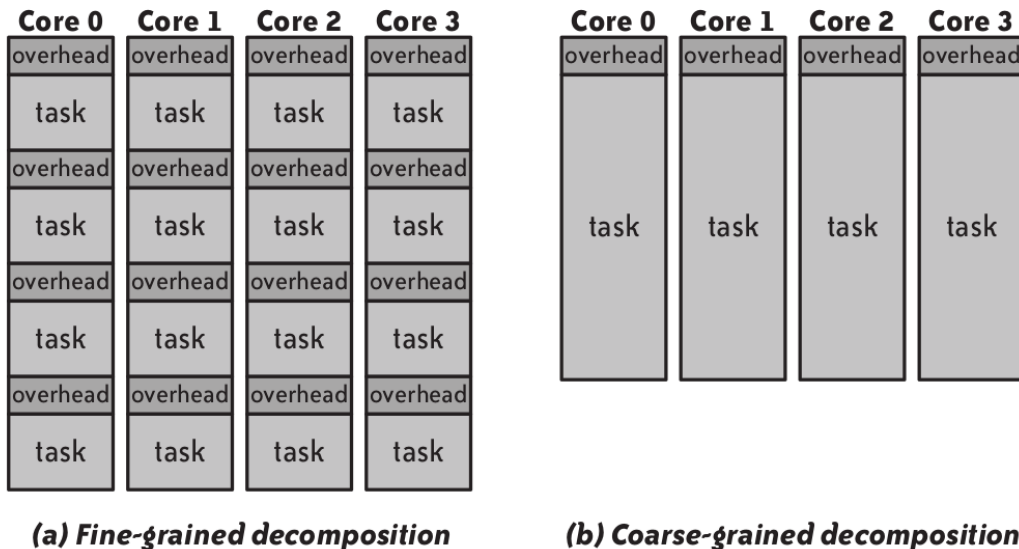


Figure 29: Granularity/overhead relationship "The art of concurrency - Clay Breshears"

Moreover, a better look to the two previous plots (Figure 27 and **Errore. L'origine riferimento non è stata trovata.**) helps to individuate an analogy between them. Indeed, zooming the first plot in the low dimension data area, we can notice almost the same behavior of the second plot.

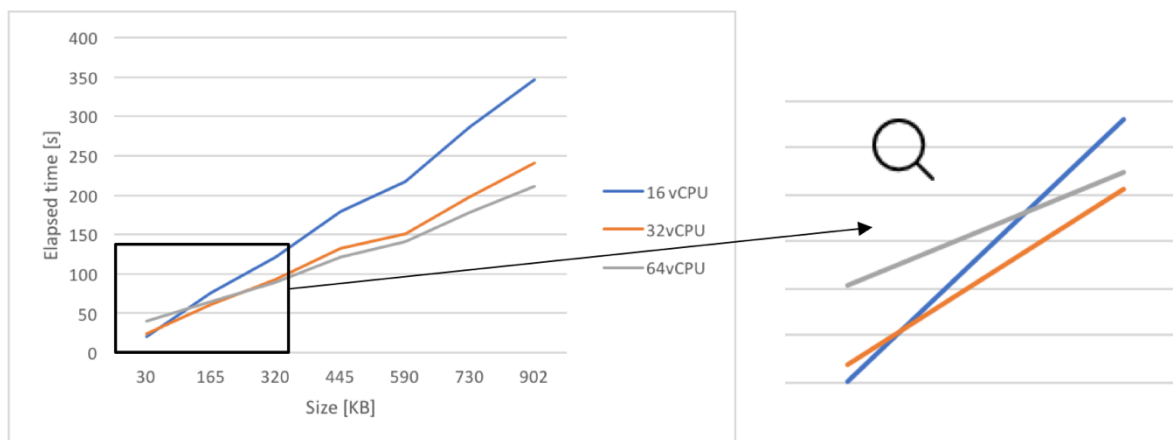


Figure 30: Detail of the analogy between plots

By solving that, a cost function would be introduced in each OpenMP directive, by trying to reach the following result, which highlights the optimal thread number to be used for every case.

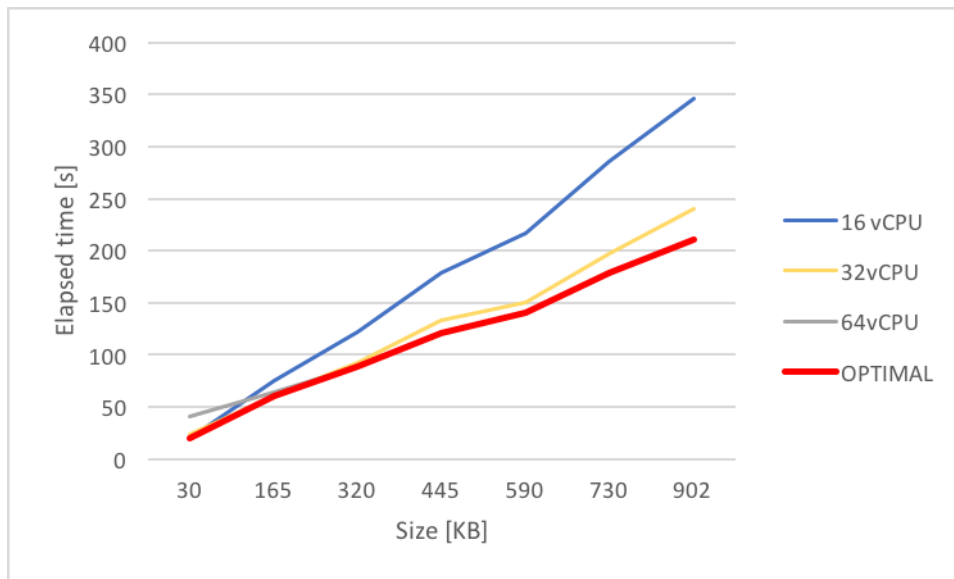


Figure 31: Line chart - Optimal thread assignment in breastcancer dataset

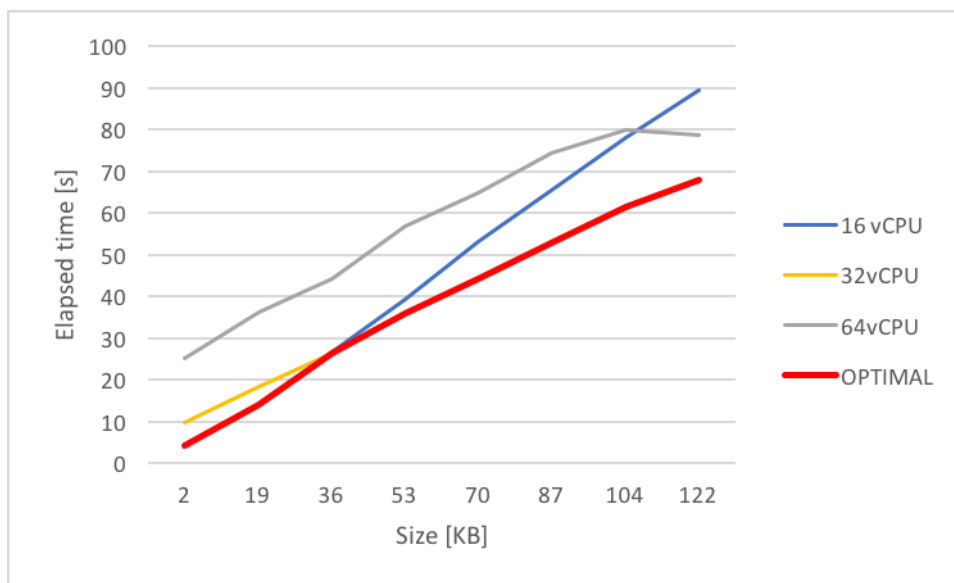


Figure 32: Line chart - Optimal thread assignment in iris dataset

For this explicit threaded solution, we'll need to determine the crossover point and build, in limiting logic, the function which controls either the number of threads to deploy and the correct chunk size in order to achieve the best performances.

Compiler optimization [-O3]

An additional option for compiling enables some possible optimization flags that aren't typically activated by the standard gcc compiler. Of course compiling process is slower, but this brings several good results in terms of execution time and assembly code size. We bring to your notice that this

action is processor-specific, so not portable, it means that a compilation is needed on each new hardware on which we run this software.

Note that without the optimizations introduced by this kind of options, the compiler's goal is to reduce the cost of compilation and to make the program produce the expected results.

-O3 brings an almost complete optimization, but can be *bug exposing*, so its use must be carefully done. Indeed, that one doesn't even guarantee that standards-complying programs will not break.

Comparison

In order to compare the two options, only two hardware configurations have been taken into account: the 4 cores and the 16 cores ones in the best implementation chosen above, that is the first. The results are shown in the plots below.

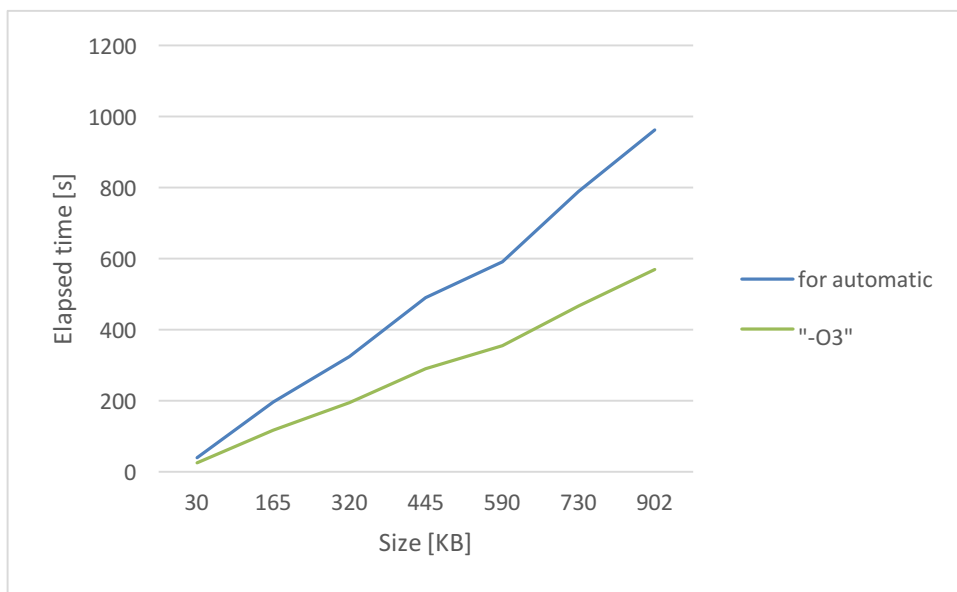


Figure 33: Line chart - Breastcancer, 4 threads

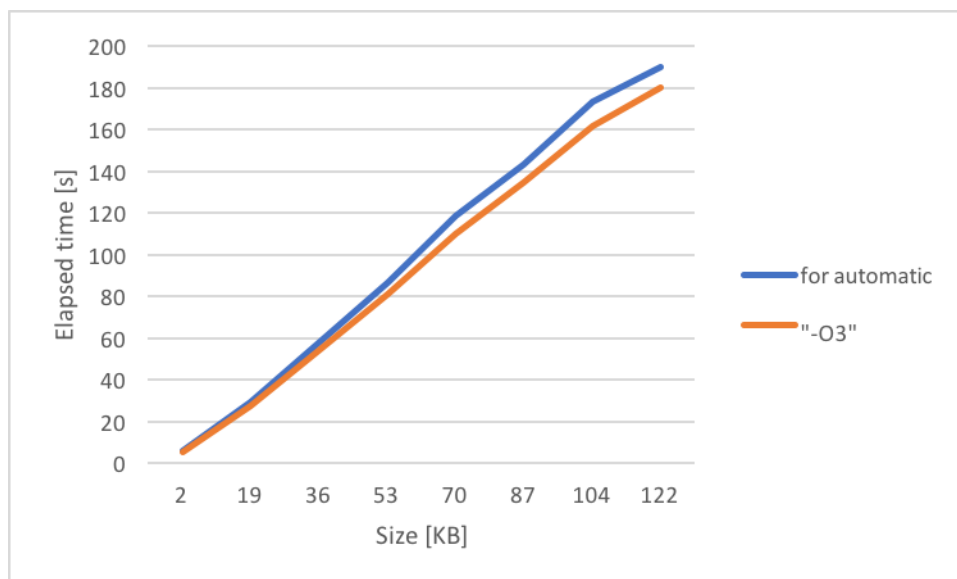


Figure 34: Line chart - Iris, 4 threads

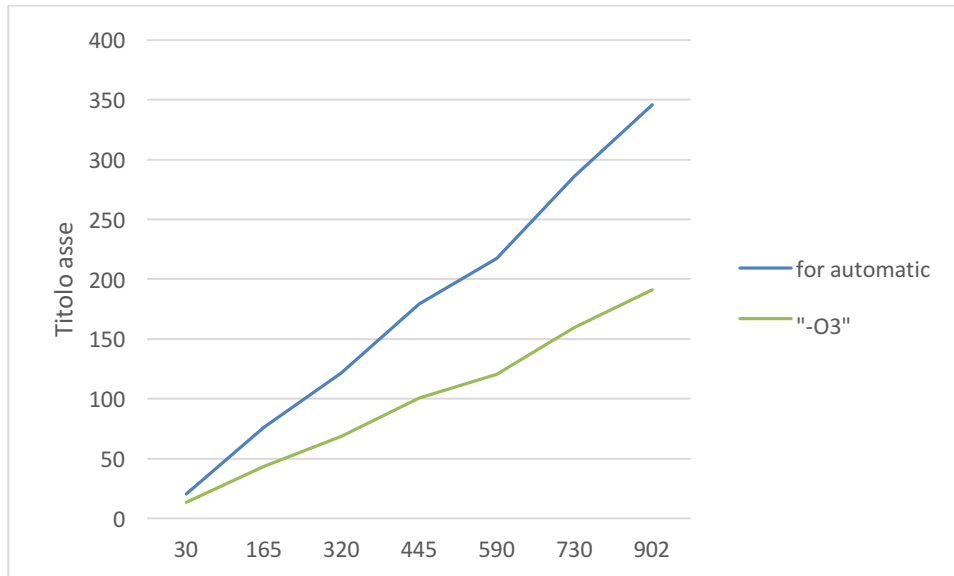


Figure 35: Line chart - Breastcancer, 16 threads

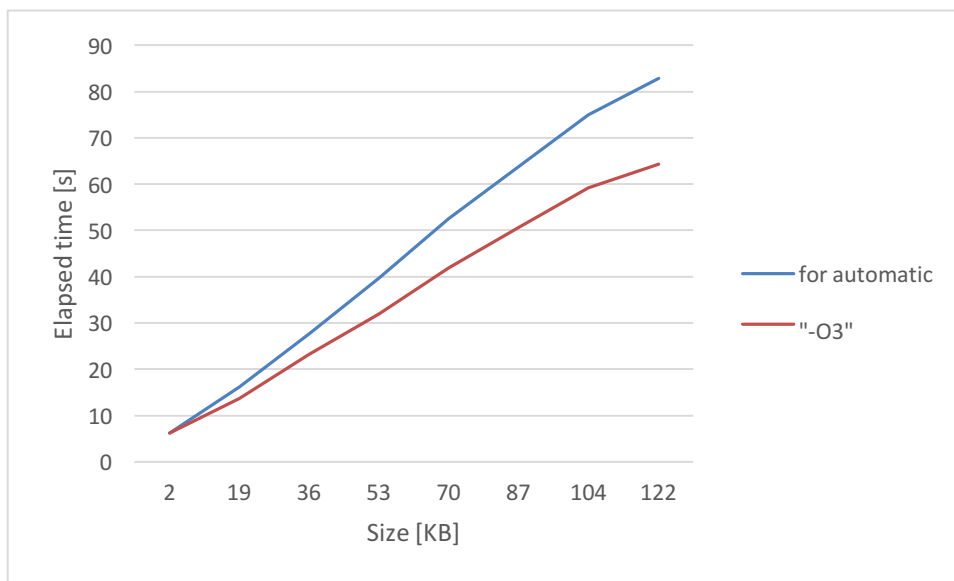


Figure 36: Line chart - Iris, 16 threads

Testing and debugging

During testing phase, several problems raised up. Most of them were due to the wrong way of sharing resources between threads. Indeed, simply sharing a statically allocated array between them could bring semantic problems or segmentation faults. This is because, when shared, an array like this imply that each thread creates its own copy that is elided after the parallel region, making the whole computation infeasible. To avoid such a problem, **dynamical allocation is needed**, that implicitly share the addresses of values of the array and not the value.

In order to validate the fairness of the algorithm's outcome, several comparisons were made with one of the actual main machine learning tools now available: Orange (ver. 3.11.0). Optimal results were obtained for each of the datasets adopted.

Performance analysis and speedup

At this point it's necessary to compute a measure of the speedup obtained by our best implementation (first one). By doing it we estimate the ideal amount of speedup that we could achieve in theory, then we compare it with the real one. For this purpose, we adopt the most used speedup estimate: **Amdahl's Law**.

$$Speedup \leq \frac{1}{(1 - pctPar) + \frac{pctPar}{p}}$$

where *pctPar* is the percentage of execution time that runs in parallel, and *p* is the number of cores on which to run the parallel application.

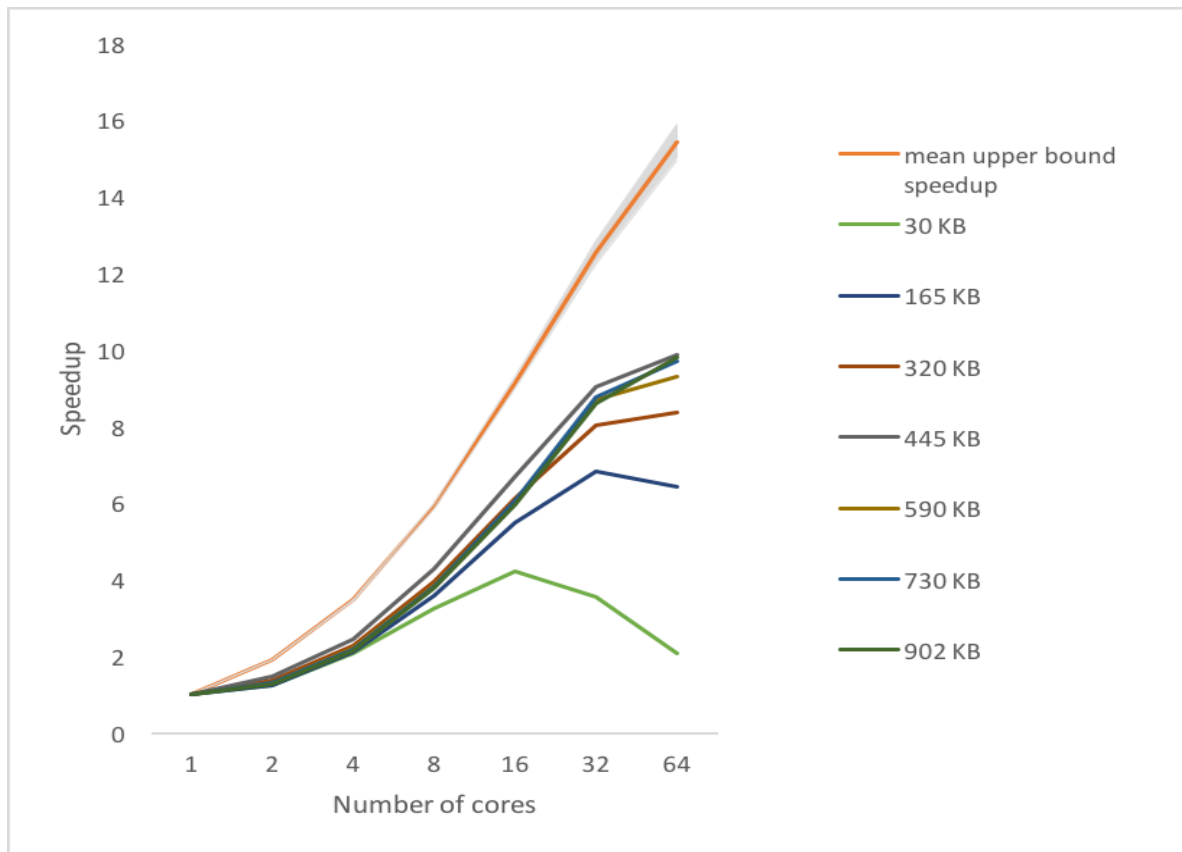


Figure 39: Speedup Breast cancer dataset

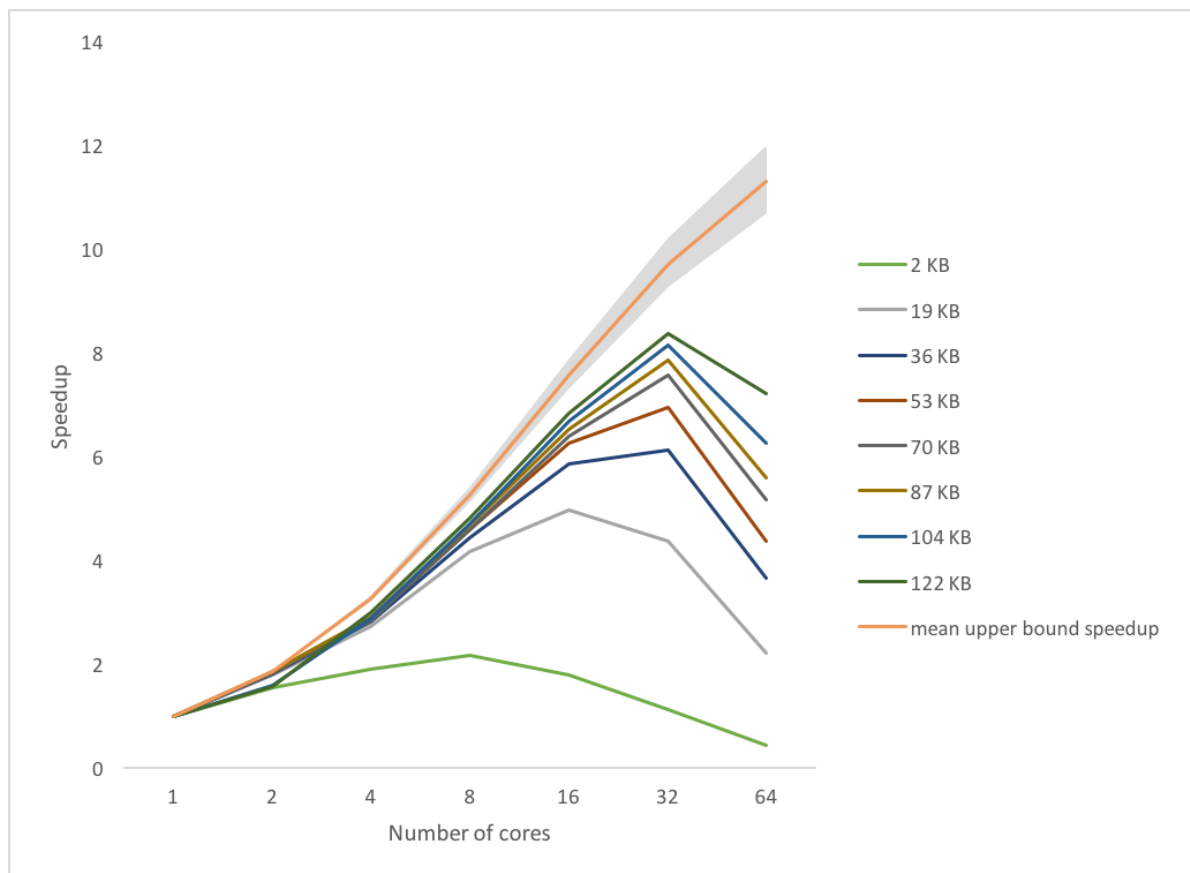


Figure 38: Speedup Iris datasets

Conclusions

Since the speedup is related to the size of the dataset, the optimal one changes with it, hence, a variability range of the speedup upper bound is shown on the plot.

We bring to your notice that, especially for larger datasets, the algorithm scales well with the number of threads. On the other hand, this behavior get worse when granularity is too low, like for the Iris dataset case.

Even if strongly suggested we couldn't develop cost function implementation cited above in [Thread number optimization](#) for time reasons.

Anyway, we can conclude that obtained results are sufficiently satisfying and some previsions made in [A bottom up approach](#) were confirmed in the practical testing phase ([OpenMP parallel implementation](#)).

Bibliography

Clay Breshears – “The art of concurrency”, O REILLY

<https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html> - Options that control optimization

<https://software.intel.com/> – Parallel profiling tools

<https://www.ibm.com/support/knowledgecenter/> – Guides and product documentations