PC-2020/21 K-Means

Anonymous CVPR submission

Paper ID ****

Abstract

In this mid-term project I have implemented K-Means clustering and then I have applied it to image in order to achieve a color quantization as visual result. I have realized three different implementations in Java, one sequential and two parallel, and then compared the execution time to obtain and consider the speed up.

1. Introduction

K-Means, described in Section 2, is one of the simplest and most used algorithm for clustering points, although its performance, especially for its most basic version, is not so high. In general, the points must belong to an Euclidean space, like $\mathbb{R}^n$, (or a convex subset thereof) because the algorithm is based on the concepts of Euclidean distance and "average" between points; but is possible to generalize it for non-Euclidean space like string's space, where we have a notion of distance, but there's not an "average" between points.

In this project I have implemented K-Means considering the Euclidean case and for convenience I restrict the domain to $\mathbb{R}^n$ and its hyper-rectangular subsets (which are the most common case, like in RGB space color introduced in following sections).

As application of this algorithm I have choosen the images: applying a clusterization method to an image (to be precise to the pixels of an image) we obtain a color quantization of the image, which can be a first result for image segmentation. I describe this aspect in Section 3.

In the following sections I present my implementation solutions, all developed in Java: the first of them is a sequential solution, the second and the third are parallel versions that use Java high level framework for concurrency, one handling explicitly the parallelism and one handling it implicitly. In the lasts sections I report the result of the performance tests, discussing the speed up and proposing my personal considerations about the efficiency of the solutions.

2. K-Means Clustering

K-Means clustering is a family of clustering methods based on the a priori knowledge of the number of clusters into which the given points must be classified. Some algorithms to deduce the optimal value of k exist, but this is outside the main topic of this project, so we assume to have k as input parameter (and we evaluate the main results also as value of k varies).
In this project we will consider the standard version of K-Means, also known as naïve K-Means [1]; the idea behind it is quite simple:

0. select k points $m_1^{(0)}, \ldots, m_k^{(0)}$ from the dataset that we will consider as **centroids** of step-0 cluster;

1. assign each point of the dataset to the cluster identified with the nearest centroid;

$$S_i^{(t)} = \left\{ x_p \middle| \|x_p - m_i^{(t)}\| \leq \|x_p - m_j^{(t)}\| \right\} \quad (1)$$

2. update the clusters' centroids with the average of its points;

$$m_i^{(t+1)} = \frac{1}{\left|S_i^{(t)}\right|} \sum_{x_j \in S_i^{(t)}} x_j \qquad (2)$$

3. repeat from step 1 as long as the clusters "change enough" respect to the previous iteration.

This algorithm is guaranteed to converge (if we are using the Euclidean distance), though, as we can see, there are various aspects open to free choice, let's discuss them.

2.1. Clusters Initialization

About how to choose the initial centroids there are mainly two approaches: we can take as centroids k points in order to maximise their relative distance or we can take as centroids the point nearest to the centroids of a raw clusterization (perhaps based on a sample of the dataset). I have adopted the first approach as in [2] and I report the algorithm:

Pick the first point at random;
while there are fewer than k points **do**
  Add the point whose minimum distance from the selected points is as large as possible;
end while

2.2. Stop Condition(s)

Also for the stopping rule is possible to define different conditions. Ideally we would like to stop when none of the data points change cluster from one iteration to the next one (that is we achieve the minimum sum of distances of every point from its cluster's centroid), since is guaranteed the convergence. As relaxation of this condition we could take as stopping rule the "convergence" of the diameter of the clusters (intended as the maximum distance between two points of a single cluster) or the position of centroids; obviously considering as "convergence" theirs change under a certain threshold.

In this project I assume as stop condition the latter case, taking as threshold the value 0.005.

Figure 1. An image and its color quantized version with 8 colors

3. Image Color Quantization

As already mentioned, with K-Means is possible to get a color quantization of an image [3]. This process aims to reduce the number of colors in an image, grouping the pixels in clusters and then assuming all pixels in a cluster of the same color, in particular the average color of the cluster, as it's possible to see in figure 1.

To do that pixels should be considered as points in the space of colors (thus loosing momentarily the information relative to the spatial coordinate inside the image) and applying clusterization algorithm to them. This process can be applied to pixels from RGB color model, but also in gray scale model or others; in the first case the domain of the points is a subset of $\mathbb{R}^3$ (it would be $[0, 255]^3$), in the second a subset of $\mathbb{R}$ (it would be $[0, 1]$). In this project we will consider the RGB case without loss of generality. Moreover, as usual in practical use, the RGB values assume only integer values, so it isn't anymore an Euclidean space (*e.g.* the strings case mentioned

above); however during the computation I abandon the discrete representation in favor of the continuous one; only at the end I return to the finite representation.

4. Solution Implementation

In my opinion the most suitable language/framework for implementing K-Means is CUDA, since most of calculation is independent and so the algorithm doesn't require much synchronization; and in addition, as generally happens working with images, the typical dimension of datasets is really huge (*e.g.* for an image in 4K resolution and 16:10 ratio, the pixels amount to 10.5M). Anyway I choose Java to compare the sequential and the parallel versions developed with the same language; and furthermore Java offers, at least for the implementation of this specific task, an explicit and an implicit version of parallelism: it is interesting to compare them.

I have divided the algorithm in 3 subroutine: the calculation of the initial centroids, then the assignment of each point to a cluster and finally the update of centroids. All this operation require an iteration over all the points in dataset. The main idea that has guided the parallelization of the code is to parallelize this subroutines since they are, at least this is what I assumed theoretically, the hotspot in term of computation time (however we will discuss about this in the next Section presenting the results of profiling of the sequential version). A fourth minor subroutine is that of checking the stop condition, but this is an iteration over the k centroids and typically k is a quite low value; there is the possibility to parallelize also this operation, but we expect a relatively small improvement (if not a deterioration for too small value of k due to parallelization overhead).

5. Sequential Solution

Now I present my sequential solution implemented in Java; the code is available in this GitHub repo.

This solution is very basic and it is based on the class *Point*, that represents a point in $\mathbb{R}^n$ by storing its coordinates in a float array; a Point is immutable, so it can be shared between thread safely. The class Point implements a static method *getSquaredEuclideanDistance* for calculating the distance between two Points (we save computation by taking the square of distance because is totally indifferent in the algorithm). Lastly this class can be inherited in order to represent particular cases of points: I have inherited *RGBPoint*, that represents a tridimensional point with coordinates in $[0, 255]$, and *RGBPixel*, that represents an RGBPoint with integer coordinates and additional metadata for its (x, y)-coordinates in the image[1].

To ensure that all the Points to be treated belong to the same domain, I have created the class *SetOfPoints* consisting in a collection of Points and a *Domain* to which the Points are constrained to belong. A Domain, as initially said, it's an hyper-rectangular subset of $\mathbb{R}^n$ and so consist of a dimension and two arrays of lower and upper bounds for each of its dimension. A subclass of SetOfPoints is *Cluster* that only add some methods typical of a cluster (like *getCenter* and *getDiameter*).

The core of this implementation is the class *KMeans*, genericized with a subclass of Point. Its only public method *clusterize* take as parameters a SetOfPoints (to this end I define a functoid class *Image* to load an image and store it as SetOfPoints) and the number k of desired cluster. Internally I abandon for convenience the SetOfPoints wrapper, simply working with an array of Points. This method is implemented exactly as mentioned before: an initial call to the private method *initialCentroids* and then repeated calls to the private methods *updateClusters* and *newCentroids*. Let's see them more in depth:

- *initialCentroids*: from the dataset produces a set of k points which are the initial centroids. This method implements exactly the algorithm shown in Section 2.1. The only

[1] maybe in a general contest the composition of *RGBPixel* with *RGBPoint* would have been better, but in this specific and simple case it turns out to be more comfortable the inheritance: KMeans can be executed both with points and pixels without any additional manipulation

simplification is that I take as first point not a random one, but exactly the first of the dataset, assuming that this is randomly sorted in the color space (in the case of images this assumption can be considered always satisfied). This trick allows to make the choice of initial centroids reproducible;

- *updateClusters*: consists simply in the iteration over the points and over the centroids, to determine which centroid is nearest to each point. It is immediately evident that this portion of code is embarrassingly parallel since every point is totally independent from the others. I represent the actual clusterization with an array of integer, where the i-nth element contains the number (from 1 to k) of the cluster to which the i-nth point is assigned.

- *newCentroids*: starting from the actual clusterization calculates the center of each cluster iterating on the points, summing its coordinates in a bidimensional array $k \times dimension\ of\ the\ points$ (where each row contains the sum of the coordinates of all points of that cluster) and taking into account the size of every clusters and then completing the average calculation.

The *clusterize* method then returns a list of Clusters

5.1. Profiling

Before starting the parallelization of the code, I profile the sequential version to spot the hotspot(s).

I have focused my attention on the three submethods, indeed together they form almost the totally of the computation. Clearly their respective percentage of the total computation change with image dimension and number of clusters. To show that and to find approximate average values I have profiled the execution in 3 different cases: a 4K image with 10 and 15 clusters and a 6K image with 10 clusters. The results can be found in the repo and they are summarized in Table 1.

Clearly *initialCentroids* depends both on the points and k, so with the same dataset it takes more time as k grows, but since also the total execution time grows, the percentage remain almost the same. However the percentage is very low and decrease with dataset dimension.

updateClusters also depends both on points and k (it was easily predictable because at every iteration it calculates and evaluates $number\ of\ points \times number\ of\ centroids$ distances), indeed the execution time grows as dataset dimension and number of clusters grow; furthermore we can see that also the percentage over the total time (that is by far the predominant one) grows.

newCentroids depends only on points, since it was basically an iteration over them. Indeed the execution time grows with dataset's growth and remain the same with k's growth (the percentage over total computation shrinks)

The most important aspect of these results is that, independently from percentage values, the main hotspot is by far *updateClusters*. And this is a great news, because it is embarrassingly parallel. The other two methods can be parallelized, but their small execution time (and need of synchronization) exposes us easily to a too high overhead respect to the gain.

A noteworthy fact is that analyzing the profiling results we can note that the lowest level hotspot is the method *getSquaredEuclideanDistance*: in fact it occupies more than 50% of the computation (although it was used only by *updateClusters* and *initialCentroids*).

6. Parallel Solution

As we said in Section 4, all three of this methods consist of an iteration on the dataset's points (or more iterations in the case of *initialCentroids*); so the guideline I followed to implement my solution, available on this repo, is that of decompose the dataset. I have also thought to parallelize the calculation of distance (it consists of an iteration of the coordinates of the points), anyway I changed course immediately because it was

Image	k	Execution Time	*initialCentroids*	*updateClusters*	*newCentroids*
4K	10	57s	6%	85%	8.5%
4K	15	99s	6%	87%	6.5%
6K	10	322s	3.5%	88%	8.5%

Table 1. Profiling results

very fast in a single iteration and in all likelihood the parallelization overhead would have been too high respect to the gain. Moreover, so doing, I followed the good practice explained in [4] of parallelizing at the highest level possible.

6.1. Explicit Parallelism

The main idea was to create a single thread pool in the *clusterize* method and to submit to it dynamically three types of task for the three step of algorithm: *initialCentroidsTask*, *updateClustersTask* and *newCentroidsTask*. About the number of threads we have to consider that all this tasks are purely computational, so there should be a 1:1 ratio between threads and cores, as explained in [5]. I consider to put an extra thread to supply page fault events, but my test machines are equipped with Intel processors with Hyperthreading, so setting the number of threads equal to the number of virtual threads is sufficient to prevent page faults delays; anyway later we validate this assumption with tests.

6.1.1 initialCentroidsTask

The *initialCentroids* method executes for k times an iteration on the points. It can't be parallelized the outer iteration because every iteration depends on all the previous (that is for calculates every centroids we must know all the previous), so this task executes the iteration on a portion of the dataset. Since at every iteration of the task the main input, that is points and centroids arrays, doesn't change (at least from a Java perspective intended as the reference to the centroids array) I reduce overhead costs by instantiating the object just one time and reusing it updating/resetting only some aspects at every iteration. In particular before submitting the task I invoke the method *nextIteration* passing two newly created AtomicInteger shared by all tasks representing the next identified centroid and its maximum minimum distance from the others (the reason of AtomicInteger type will be clear later).

An initial implementation expected each task to return a candidate centroid and then to choose the best sequentially. However, although this solution did not require synchronization, it was quite slow (even worse than the sequential version). The main problem is that the tasks should set *two* shared variable atomically. Anyway the solution was quite simple: I have associated the class to a Lock, every tasks find privately its candidate point to be a centroid and then in a critical section it checks if its candidate is better than the actual one and if it is, it update the shared variables. The choice of AtomicInteger is not for atomicity (they are read and written in a critical section), but because Java pass by value the primitive type (even if boxed): here I want this two value to be instantiate in main thread, written in forked threads and read again in main thread; a final note on this: the distance is a float, but can be represented with an integer relative to its bit representation.

A final consideration about the points and centroids arrays: they are shared between all tasks, but they are accessed only for reading, so they can't cause false sharing.

6.1.2 updateClustersTask

As already said this is the easier task because embarrassingly parallel. It was invoked in a while loop and so I use the previous solution of instantiating tasks just one time and reusing them, since they only need to have the centroids update at every iteration.

Also this tasks share points and centroids array

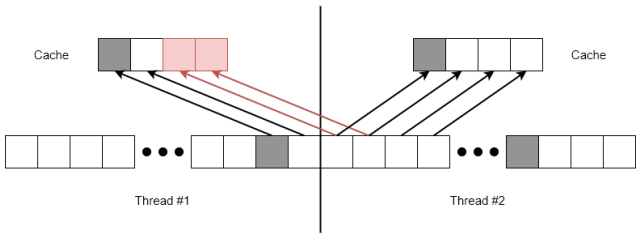

Figure 2. Example: the only case where false sharing occures

only for reading, but, differently from *initialCentroidsTask*, they share also the clusterization Integer array. This array is accessed also for writing, so it can cause false sharing. Anyway the number of threads is infinitely smaller than dataset dimension and every task accesses (and modifies) a very big number of consecutive positions. The false sharing can occur only when it is cached a border position of a block of array: in this case it is possible that in the same line of cache there is the last positions of previous block or the first positions of the next block (as shown in figure 2). Moreover we can assume that when a thread processes the first positions of its block, also the other threads processes the first positions, since they all progress tidily on a very big number of elements, so the last positions of previous block are not written and this prevents false sharing occurs. Same reasoning for the last positions. So in light of this we can conclude that false sharing occurs rarely and its effects can be ignored.

6.1.3 newCentroidsTask

This last type of task presents any new idea: again I reduce the overhead instantiating them just one time, they share the points and the centroids but only for reading and there is a method to reset the shared variable, useful calculate the average of each clusters, before they are submitted at every iteration.

Again the best solution turned out to be that of calculating a private partial result and then in a critical section putting it in the shared variables. Differently from *initialCentroidsTask* here it was quite simple to use structures of atomic (again, the atomic float would have been replaced by an AtomicInteger) to avoid the critical section due to absence of conditional write, but this solution was terribly worse in terms of execution time, probably for huge false sharing effects (because the shared variables were modified so frequently). So also here I have introduced a critical section.

6.2. Implicit parallelism

Since updateClustersTask consists on writing on an array in each of its position a value depending only on the position's index (this is why it is embarrassingly parallel), we can exploit a native method of Java: *Arrays.parallelSetAll*. Passing to this method the array clusterization and the lambda function that assigned an index iterate over the centroids to determine to which cluster assign the point, we obtain exactly what we want. And all done by a native function, that clearly is super-optimized and certainly is able to takes all the possible advantage from Hyperthreading.

This method is implemented with a ForkJoinPool and the number of threads (better to say degree of parallelism) can't be handled directly, but can still be changed through a system property[2], that I set to 4 (the number of virtual threads of my CPU) to make this solution comparable with the other.

6.3. Structure of Arrays idea

The solution I have just presented use the Array of Structure paradigm. At a certain point I have evaluated the possibility to implement the Structure of Arrays one. Anyway this idea gets bogged down shortly, indeed the results (although on a preliminary version, not optimized at all) was terribly worse, not even comparable. And this is easy to see theoretically: we access the dataset point by point, in all its coordinates, and not dimension by dimension. Just think of *getSquaredEuclideanDistance* method (that we remember to take over the 50% of total computation) where we access two, and only two, points in every coordinate. The SoA version suffers of a tremendous

[2] java.util.concurrent.ForkJoinPool.common.parallelism, which is by default #CPUs - 1

number of cache miss, indeed the advantage of the cache is not exploited at all.

7. Tests and Results

The execution time obviously grows as dataset dimension and number of cluster grow. I have considered three different dimensions for input images: 4000×2000, 5000×3000 and 6000×4000 for a total of, respectively, 8M, 15M and 24M of pixels. I tested every dimension for different values of k: 2, 8, 14, 20.

The execution time is highly sensitive to the change of input for the same size and number of clusters (I have recorded a variations of up to 250% with different image of the same size). And sometimes I also notice relatively big casual variations. So in order to present a result a bit more generic I run the code on three different images for each size (and each k) two times. I have summarized the results in Table 2, where I report mean execution time, speedup and efficiency.

7.1. Main tests

First of all let us compare the two Java parallel versions: it is clear that the version with the native ForkJoinPool has generally better results. The speedup increase seems to be almost constant as image size increases, but it enhances with k. An interesting fact is that for very small values of k the version with native ForkJoinPool behaves (a little) worse than the other: in all the cases the speedup is worse or at least the same. This can be explained, likely, with a greater instantiation overhead of a ForkJoinPool respect to a classic Executor thread pool; clearly as k increases the algorithm cycles more and the lightness of ForkJoinPool's thread prevails over its overhead.

Excluding some cases, where probably the input instances play an important role, the speedup for the Executor thread pool version is about 1.7 (for an efficiency of 0.85): not a bad result but not noteworthy either, consistent with my prediction. The version with native ForkJoinPool instead, again excluding the same cases, has a speedup of 1.8; but not only: indeed in some cases reaches a speedup of 2. This means that native ForkJoinPool is able to exploit the Hyperthreading the most. Clearly it is too little to talk about superlinear speedup, though in some cases the speedup is just above 2: indeed it could be merit of random fluctuations or very lucky combination of image and value of k (though the speedup seems to increase with k, tests with higher value of k refute this possibility).

7.2. Optimal parallelism degree tests

In the main tests I assume the parameters hypotezed above (4 threads and degree of parallelism equal to 4), but I want to verify that the guess is right. I run the code, for every combination of image size and k (just once for time reasons), with varying number of threads/degree of parallelism from 2 to 6. The figure 3 show the mean speedup for every number of thread used.

Concerning the explicit version my suppose were right: the optimal number of threads is 4, that is my CPU's virtual threads; increasing the number of threads we obtain a constant speed up, worse than the optimal because of overhead due to context switch operation. Interesting fact that 3 threads do quite significantly worse than 2 threads: this is probably a consequence of Hyperthreading.

In the implicit version I used the optimal threads number for the tasks that doesn't use the native function and varying the degree of parallelism for the ForkJoinPool I have obtained an interesting result: not only this version is always better than the other, but thanks to the super optimization of the native function, setting the degree of parallelism beyond 4 the speedup continues growing, reaching an average of 1.9 with 5 and 6 threads.

7.3. Extra tests

As I said before (from the beginning) from the implicit native version I expected it to fully exploit CPU advantages, i.e. cache and Hyperthreading. That turned out to be true: indeed running the parallel versions on a CPU with more

Input		sequential	parallel					
			explicit			implicit		
		T_S	T_2	S_2	E_2	T_2	S_2	E_2
4K	$k=2$	$5.8s$	$3.3s$	1.76	0.88	$3.4s$	1.71	0.85
	$k=8$	$63.2s$	$38.3s$	1.65	0.83	$35.6s$	1.78	0.89
	$k=14$	$112.7s$	$66.9s$	1.68	0.84	$61.1s$	1.85	0.92
	$k=20$	$237.0s$	$131.0s$	1.81	0.90	$121.9s$	**1.94**	**0.97**
5K	$k=2$	$9.0s$	$6.0s$	1.50	0.75	$6.0s$	1.50	0.75
	$k=8$	$92.5s$	$56.7s$	1.63	0.82	$53.2s$	1.74	0.87
	$k=14$	$202.9s$	$108.8s$	1.86	0.93	$98.7s$	**2.06**	**1.03**
	$k=20$	$466.9s$	$253.8s$	1.84	0.92	$231.5s$	**2.02**	**1.01**
6K	$k=2$	$17.5s$	$12.2s$	1.43	0.71	$12.2s$	1.43	0.72
	$k=8$	$135.3s$	$77.6s$	1.74	0.87	$74.4s$	1.82	0.91
	$k=14$	$505.8s$	$284.1s$	1.78	0.89	$267.7s$	1.89	0.94
	$k=20$	$721.2s$	$428.6s$	1.68	0.84	$402.6s$	1.80	0.90

Table 2. Time, speedup and efficiency for all test cases

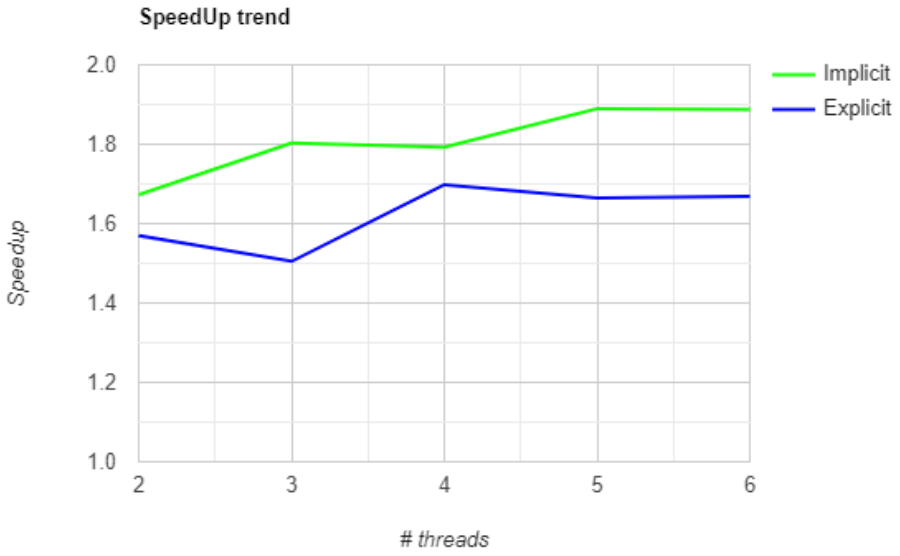

Figure 3. Speedup trend varying the number of thread/degree of parallelism

core, but no Hyperthreading, the implicit version takes longer than the explicit one. Clearly the absolute times were far better (thanks to the high level hardware), but efficiency was worse. This deterioration involve both, explicit and implicit, versions, but the implicit one was the one most affected, until to result (a little) worse than the explicit one in all cases (as reported in table 3).

This behavior is easy to explain: not only the implicit version can't exploit the advantages of Hyperthreading, but it also presents an overhead: indeed at every iteration it has to instantiate a ForkJoinPool, instead of reusing the always same thread pool as the explicit version.

		explicit	implicit
		T_6	T_6
4K	$k = 2$	$1.3s$	$1.3s$
	$k = 8$	$11.9s$	$12.0s$
	$k = 14$	$19.7s$	$20.2s$
	$k = 20$	$37.7s$	$39.1s$
5K	$k = 2$	$2.7s$	$3.0s$
	$k = 8$	$20.1s$	$21.7s$
	$k = 14$	$30.6s$	$32.2s$
	$k = 20$	$72.1s$	$76.2s$
6K	$k = 2$	$4.0s$	$4.2s$
	$k = 8$	$25.5s$	$26.6s$
	$k = 14$	$79.7s$	$90.2s$
	$k = 20$	$133.2s$	$146.1s$

Table 3. Parallel execution time without Hyperthreading

7.4. Complete tests data

All tests result is available for consultation in table 4. From these is possible to obtain other interesting statistics. For example that the implicit parallel version has better result with the maximum degree of parallelism with a big input and worse with small input. There is also the execution time of the parallel versions with only one thread (so it can be evaluated the overhead respect to the sequential version). Anyway the reported speedup is intended relatively to the sequential version. Final note: the implicit parallel version has fixed number of threads for non-native parallel section (equal to 4 that was the best choice from previous results).

References

[1] *K-means clustering*, Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/K-means_clustering (visited on Feb. 1, 2020).

[2] J. Leskovec, A. Rajaraman, and J. Ullman, *Mining Massive Datasets*. Cambridge University Press, 2011, ch. 7.3.

[3] *Color quantization*, Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Color_quantization (visited on Feb. 1, 2020).

[4] C. Breshears, *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly, 2009, ch. 4.

[5] N. Matloff, *Programming on Parallel Machines: GPU, Multicore, Clusters and More*. University of California, 2012, ch. 3.

8. Additional Notes

8.1. Host OS

The main tests were runned on a PC equipped with Ubuntu, since some tests show that both the parallel versions get better results than with Windows. The sequential version instead gets better result on Windows. Anyway the difference were not so high. As consequence, running on Ubuntu the speedup results a little higher.

8.2. Technical Specification

Main tests were runned on/with:

- Intel Core i7-6500U (2.5 GHz up to 3.1 GHz, 2 cores, 4 threads, 4 MB of cache);
- 8 GB of RAM;
- Ubuntu 20.04/20.10;
- Java SE 11;

Extra tests were runned on/with:

- Intel Core i5-9600K (3.7 GHz up to 4.6 GHz, 6 cores, 6 threads, 9 MB of cache);
- 16 GB of RAM;
- Windows 10 Home;
- Java SE 11;

Input		sequential	parallel											
			explicit						implicit					
			#1	#2	#3	#4	#5	#6	#1	#2	#3	#4	#5	#6
4K	$k = 2$	5.8s	6.0s	3.6s	3.4s	3.3s	3.4s	3.4s	3.7s	3.6s	3.4s	3.4s	3.4s	3.3s
	$k = 8$	63.2s	68.0s	40.2s	45.7s	38.3s	36.7s	37.8s	38.5s	39.7s	35.1s	35.6s	34.3s	36.2s
	$k = 14$	112.7s	117.9s	72.7s	82.8s	66.9s	67.8s	69.3s	65.6s	63.5s	60.6s	61.1s	58.8s	63.2s
	$k = 20$	237.0s	262.1s	141.7s	157.3s	131.0s	140.0s	139.4s	126.0s	126.7s	118.5s	121.9s	116.1s	125.2s
5K	$k = 2$	9.0s	10.2s	7.4s	6.6s	6.0s	6.0s	6.5s	6.9s	6.3s	6.5s	6.0s	6.0s	5.9s
	$k = 8$	92.5s	107.9s	66.0s	63.1s	56.7s	56.1s	62.8s	60.2s	60.3s	54.2s	53.2s	53.6s	50.6s
	$k = 14$	202.9s	236.3s	117.1s	122.3s	108.8s	106.3s	104.5s	108.8s	109.0s	100.4s	98.7s	102.7s	95.3s
	$k = 20$	466.9s	558.8s	272.4s	292.1s	253.8s	264.7s	252.7s	246.2s	255.9s	2228.8s	231.5s	240.2s	221.1s
6K	$k = 2$	17.5s	17.5s	13.3s	12.9s	12.2s	12.5s	12.5s	14.2s	12.4s	12.3s	12.2s	11.0s	11.6s
	$k = 8$	135.3s	138.4s	81.0s	83.8s	77.6s	80.6s	75.6s	89.9s	78.2s	73.2s	74.4s	65.1s	68.6s
	$k = 14$	505.8s	488.7s	269.4s	300.4s	284.1s	272.2s	260.3s	317.0s	270.0s	255.9s	267.7s	225.7s	244.3s
	$k = 20$	721.2s	673.3s	480.1s	532.3s	428.6s	501.8s	466.8s	423.6s	408.5s	384.9s	402.6s	339.9s	310.9s
Speedup avg		-	-	1.57	1.51	1.70	1.67	1.67	1.62	1.67	1.80	1.79	1.89	1.89

Table 4. Execution time data of all tests cases.