K-Means Algorithm Optimization with OpenMP primitives

1st José Gonçalves
IT department
University of Minho
Braga, Portugal
pg50519@alunos.uminho.pt

2nd Miguel Fernandes
IT department
University of Minho
Braga, Portugal
pg50654@alunos.uminho.pt

***Abstract*—The present document refers to a practical assignment from a Masters Degree in Software Engineering course unit named Parallel Computing. This phase's goal is to optimize the sequential version of the K-Means algorithm implementing now parallelism (using OpenMP primitives), as-well as optimize it even more in order to obtain a efficient and fast result. With this upgrade to parallelism, it is intended to develop a version that makes the algorithm processing divided by multiple threads.**

***Index Terms*—k-means, parallel computing, performance, optimization, analysis, sequential, threads, scalability, OpenMP**

I. Introduction

With the development of a sequential version for the k-means algorithm, the second assignment consists in developing a parallel version of that, using OpenMP primitives to create a multi-threaded execution and conduct to a even better processing escalation and decrease the runtime. In an initial section, will be explained some of the changes in the algorithm code, once it has been improved. Later on, the third section is related to the most time consuming parts, known as hot-spots and in the fourth section it is discussed how those can be parallelized (being this phase keypoint). Right after, in sequence of those implementations, it is made a critical analysis of the results obtained, with emphasis on the scalability. Finally, the last chapter aims to summarize all the knowledge used and obtained in this phase, as-well as to explain some of the personal thoughts about the work done and the next objectives.

II. Changes in the algorithm

During the realization of the second phase of the practical assignment, with the aim to improve even more, one change was made in the algorithm.

Removing the cycle that sums the x and y coordinates of all points in the *reevaluate_centers* function and integrating that on the main cycle of the *cluster_points* function, the overall complexity of the algorithm was able to be reduced by N.

III. Hot-spots and keypoints in algorithm

Analysing the sequential code with the respective *perf* commands (as shown in Figure 5), it was visible that the algorithm was spending most of the time $(92,11\%)$ in the cluster_points function, which calculates N(number of points) x K(number of clusters) distances between a point and a centroid and assigns points to nearest cluster. That way, because it has the higher percentage of samples taken, the **hot-spots** must be in the cluster_points function, which will be object of an in-depth approach.

With an analysis of the *perf annotate* command (see Fig.3) it was possible to verify that the biggest time consuming instructions were the *vsubss*, *vmulss* and *vaddss*, that are used to calculate the distance between a point and the center of a cluster. As said in the beginning of this section's, that instruction is executed $N * K$ times in each iteration of k-means.

Referring to a tree-view, the distance calculation is a leaf of the tree, so it is necessary to climb in the tree until the node that corresponds to the zone that can be parallelized. The first zone founded was a loop that calculates $K - 1$ distances, although above is another cycle that iterates N times and is responsible to calculate 1 distance and the assignment of points to clusters and calculating the summation of x and y coordinates. So the best node to parallelize is the external cycle, since that has the biggest workload.

IV. Implementation in OpenMP

Since the hot-spots have been found, the next step is to explore different ways to parallelize this blocks and choose the one that ensures a better scalability, using the tools that **OpenMP** provides.

In the last section, one parallel zone was identified in cluster points function that was a for cycle, so it can be used a *#pragma omp parallel for* to divide loop iterations through threads, making sure that multiple cycles can be executed simultaneously. The load of each iteration is always the same, as each iteration calculates the distance of a point to every centroid and assigns the point to the nearest cluster. So, based on this analysis, the best thread scheduling to use is the static, avoiding the overhead of using a dynamic scheduling.

The parallelization of blocks of code bring problems such as data races and in this cycle each thread access and modify variables that are common to every other thread. To prevent this problem, some of the primitives that **OpenMP** provides were used, such as *critical*, *atomic* and *reduction*.

The variables that are shared to every thread are the 3 arrays of K size, that stores the size and the summation of x and y coordinates of each cluster, and a boolean variable that is changed if some point is assigned to a different cluster.

In order to choose the primitive that avoid data races and ensures scalability, the execution times were measured using different number of threads for 4 clusters. The following table presents the obtained execution time for each case:

Num of threads	Critical	Atomic	Reduction
4	30.55	19.40	1.17
8	90.13	25.76	0.75
12	84.20	26.90	0.56

TABLE I
SCALABILITY TEST TO OPENMP PRIMITIVES

With an analysis of the results presented on Table I, it is clear that the best option is the reduction clause.

Those results can be explained by the overhead that, especially, the critical and also atomic have in the schedule of threads to enter in this critical sections. The reduction make the variable private for each thread and only have to synchronize one time at the end of the parallel zone, resulting in less overhead and better scalability.

V. SCALABILITY ANALYSIS

To analyse the scalability of the algorithm, the main focus was the Strong Scalability, defined as how the solution speed-up increases with the number of PU's (processing units) for a fixed data size. In other words, the Speed-up is measured by calculating the time of the best sequential implementation by the time of the parallel version. The ideal speed-up is proportional to the number of PUs.

In order to get the information needed to analyse the strong scalability were developed 2 scripts, one for the 4 clusters and another to 32 clusters, being that in each script were measured the time for the sequential implementation and also the time from the parallel version, using from 4 to 48 threads. The scripts were executed in the cluster **Search** which has 20 PUs, so the ideal speed-up should be constant from 20 threads.

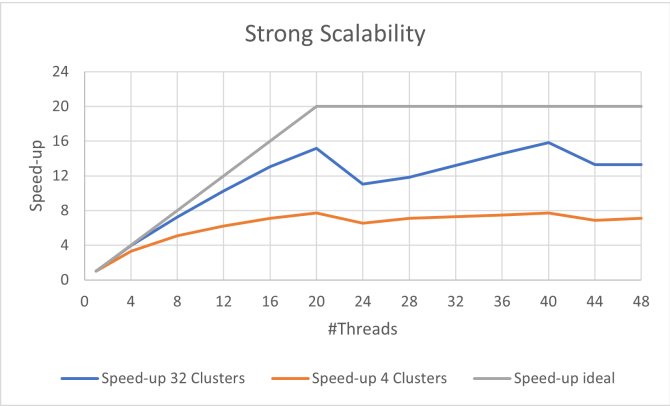

Fig. 1. **Strong scalability** analysis

The Fig. 1 shows clearly that the speed-up gained per thread is lower than the ideal for 4 cluster and 32 clusters. There are some reasons that can lead to this problem:

- **Percentage of non-parallelizable work**: Fig. 4 shows that the parallelizable work is around 86%. So by the Amdahl's law with a 20x speed-up in the parallelizable work results in only 5.46x overall speed-up (see Fig.2).

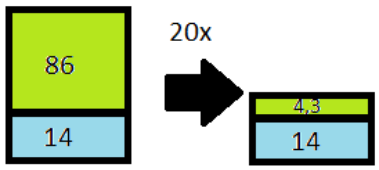

Fig. 2. Amdahl's law (green represents the parallelizable work)

- **Memory Wall**: each thread needs a set of data to work, so by increasing the number of threads, consequently the amount of data increases proportionally. This increase can reach the memory/cache bandwidth limitation.
- **Parallelism Overhead**: the use of threads comes with additional costs, i.e. more instruction for starting and finishing the threads, also to synchronizations and communications between them.
- **Load balance**: The work load of each thread is well distributed as we can see in the Fig.6 obtained by running *perf record -s* and then *perf report -T*. All threads have an identical number of cycles.

The scalability analysis is also important to discover the ideal number of threads to use in the algorithm. Observing the Fig.1 it's possible to identify 4 inflexion points, 2 for each line and both at 20 and 40 threads. Since the highest inflexion point for both lines is at 40 threads, that was the number of threads chosen to be on the *Makefile*.

VI. CONCLUSION AND FINAL CONSIDERATIONS

With the end of the second phase of the practical assignment, it was possible to reinforce the acknowledgement about parallel computing, as well as understanding the challenges of building a scalable algorithm like those that were mentioned before: **Percentage of sequential code**, **Memory/cache limitation** and **parallelism overhead**.

Overall, the final result is in line with the initial objective that was building a scalable parallel algorithm, although the group would like to explore new ways of parallelization like the *tasks* that **OpenMP** provides.

VII. Attachments

```
         nop
 2,62  128:  vmovss      (%rbx,%rdx,4),%xmm3
 0,30        vmovss      (%rsi,%rdx,4),%xmm4
 3,68        vsubss      %xmm6,%xmm3,%xmm2
 0,12        vsubss      %xmm5,%xmm4,%xmm0
 1,25        vmulss      %xmm2,%xmm2,%xmm2
 0,15        vmulss      %xmm0,%xmm0,%xmm0
 3,96        vaddss      %xmm2,%xmm0,%xmm2
 0,01        cmp         $0x1,%r10d
           ↓ jle         2f0
 0,57        lea         -0x2(%r10),%r11d
 0,01        mov         $0x1,%eax
 3,54        xor         %r10d,%r10d
 0,02        add         $0x2,%r11
 7,41  160:  vsubss      (%rcx,%rax,4),%xmm4,%xmm1
 5,87        vsubss      (%r8,%rax,4),%xmm3,%xmm0
 4,10        vmulss      %xmm1,%xmm1,%xmm1
 3,36        vmulss      %xmm0,%xmm0,%xmm0
 5,13        vaddss      %xmm1,%xmm0,%xmm0
 2,12        vcomiss     %xmm0,%xmm2
 5,33        vminss      %xmm2,%xmm0,%xmm2
 7,37        cmova       %eax,%r10d
 1,22        add         $0x1,%rax
 3,23        cmp         %rax,%r11
 1,06      ↑ jne         160
 0,01        movslq      %r10d,%rax
 0,73        lea         (%r9,%rax,4),%r11
 0,20  193:  cmp         %r10d,(%rdi,%rdx,4)
 3,65      ↓ je          1a3
 1,52        mov         %r10d,(%rdi,%rdx,4)
 0,03        mov         $0x1,%r14d
 0,45  1a3:  shl         $0x2,%rax
 9,86        addl        $0x1,(%r11)
 1,05        add         $0x1,%rdx
             lea         (%r12,%rax,1),%r10
 0,42        add         %r13,%rax
 3,27        cmp         %edx,N
 8,27        vaddss      (%r10),%xmm4,%xmm4
 3,15        vaddss      (%rax),%xmm3,%xmm3
 1,11        vmovss      %xmm4,(%r10)
 0,02        mov         K,%r10d
 3,79        vmovss      %xmm3,(%rax)
 0,01      ↑ jg          128
```

Fig. 3. Perf annotate analysis with 1 thread

```
Samples: 5K of event 'cycles', Event count (approx.): 5766463082
Overhead  Command  Shared Object      Symbol
  86,63%  k_means  k_means            [.] cluster_points._omp_fn.0
   6,84%  k_means  libc-2.31.so       [.] __vfscanf_internal
   2,76%  k_means  k_means            [.] inicializa
   0,32%  k_means  [kernel.kallsyms]  [k] clear_page_erms
   0,31%  k_means  [kernel.kallsyms]  [k] irqentry_exit_to_user_mode
   0,28%  k_means  [kernel.kallsyms]  [k] native_irq_return_iret
```

Fig. 4. Perf report analysis with 1 thread

```
Samples: 10K of event 'cycles', Event count (approx.): 10301980578
Overhead  Command  Shared Object      Symbol
  92,11%  k_means  k_means            [.] cluster_points
   3,76%  k_means  libc-2.31.so       [.] __vfscanf_internal
   1,76%  k_means  k_means            [.] inicializa
   0,23%  k_means  [kernel.kallsyms]  [k] clear_page_erms
   0,19%  k_means  k_means            [.] 0x00000000000010f4
```

Fig. 5. Perf report analysis of sequential code

```
#    PID    TID  cycles:ppp
  37910  37913  3036203820
  37910  37912  3034355313
  37910  37911  3031930052
```

Fig. 6. Work load of each thread