

Practical Assignment #2 - Parallel Computing

2022/2023

Simão Cunha
Informatics Department

University of Minho

Vila Nova de Famalicão, Braga, Portugal
a93262@alunos.uminho.pt

Tiago Silva
Informatics Department

University of Minho

Vila Nova de Famalicão, Braga, Portugal
a93277@alunos.uminho.pt

Abstract—This report serves to document the second phase of Parallel Computing course at University of Minho. It begins with a small introduction of the subject of this assignment. Then we explain our development methodology on paralyzing the algorithm K-Means. After that we show the results obtained in the SEARCH with use of diagrams and we finalized with the conclusions got from this assignment.

Index Terms—K-means, parallelism, sequential, threads, cores, SeARCH, performance, execution time

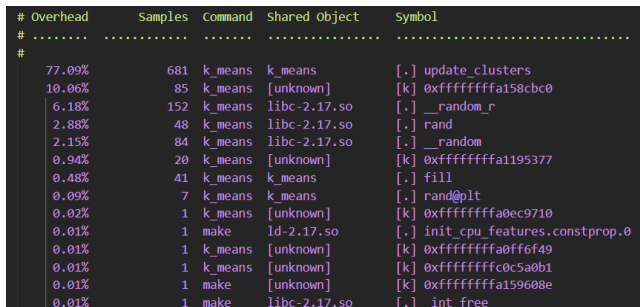
I. INTRODUCTION

The purpose of this practical work is to implement in C the parallel version of the *k-means* algorithm (inspired from *Lloyd's algorithm*), using the code from the first phase, in order to, mostly, decrease the execution time.

II. DEVELOPMENT METHODOLOGY

In this section, we will describe our methodology on this assignment.

The first step was to identify tasks parallelize, where we divide them in two categories: data decomposition and functional decomposition. We use the software *perf* to do this division with *perf record* and *perf report* and we get the following report:



#	Overhead	Samples	Command	Shared Object	Symbol
#	77.09%	681	k_means	k_means	[.] update_clusters
	10.06%	85	k_means	[unknown]	[k] 0xffffffffa158cbc0
	6.18%	152	k_means	libc-2.17.so	[.] __random_r
	2.88%	48	k_means	libc-2.17.so	[.] rand
	2.15%	84	k_means	libc-2.17.so	[.] __random
	0.94%	20	k_means	[unknown]	[k] 0xffffffffa1195377
	0.48%	41	k_means	k_means	[.] fill
	0.09%	7	k_means	k_means	[.] rand@plt
	0.02%	1	k_means	[unknown]	[k] 0xffffffffa0ec9710
	0.01%	1	make	ld-2.17.so	[.] init_cpu_features.constprop.0
	0.01%	1	k_means	[unknown]	[k] 0xffffffffa0ff6f49
	0.01%	1	k_means	[unknown]	[k] 0xffffffffc0c5a0b1
	0.01%	1	make	[unknown]	[k] 0xffffffffa159608e
	0.01%	1	make	libc-2.17.so	[.] int_free

Fig. 1. Print screen of the perf report

So, in the first category, we assigned all the variables that carry the sum of the x/y values, the quantity of that each cluster have and the convergence confirmation (float[] sumX_par, float[] sumY_par, int converged_par, int[] clusters_npoints_par and sample_id_new). But in the second category, we

identified a region with higher computational charge: point assignment to a cluster - it iterates N times over K clusters, doing a distance calculus and a comparison to get the shortest cluster and checking if the assignments changed between consecutive iterations. This is shown in the report where the function *update_clusters* has 77.09% of overhead. There's also several entries with significant entries, but all of them refers to the initialization and filling of the clusters. We cannot parallelize that chunk of code because the function *srand* isn't thread safe. After the identification as the computational costliest region, we went to explore the parallelism.

At a initial instance, we use `#pragma omp for ordered` in order to get the best consistent results possible using parallelism from the algorithm. However, we checked that the execution time increased comparatively to the sequential version, because all the threads must wait for each other. So, in detriment of the precision in the result, we use `#pragma omp for in loop for` and we initialize all the critical variables to fight against data races that could happen. In the end of that *for*, we created a critical section with `#pragma omp critical` to get the correct values of all threads and sum them. This became the most viable implementation to us because it increases in performance, taking less execution time than the sequential version, even though the obtained result is an approximated value. The reason that the size of the clusters isn't so precise like the values from the sequential version is because we're using float values in the samples and in the clusters and arithmetic operations such as sum or multiplication aren't commutable.

III. RESULTS

In this section, we ran a bash script where we tested several numbers of threads - from 2 till 40 - while using 4 and 32 clusters and 10 000 000 samples. We used this values because is known that the cluster SeARCH has two nodes with 2 cores each and each core can have 10 threads, which led to 40 threads at once - that's why the upper bound of the number of threads interval is 40.

TABLE I
EXECUTION TIMES IN SEQUENTIAL VERSION - RESTRICTED TO 20
ITERATIONS

4 clusters	32 clusters
2.398s	13.505s

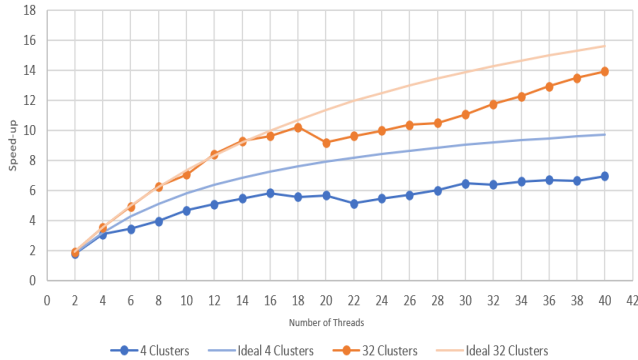
TABLE II
EXECUTION TIMES IN PARALLEL VERSION - RESTRICTED TO 20
ITERATIONS

Number of threads	4 clusters	32 clusters
2	1.332s	7.037s
4	0.771s	3.786s
6	0.696s	2.725s
8	0.602s	2.156s
10	0.513s	1.913s
12	0.471s	1.600s
14	0.437s	1.450s
16	0.410s	1.400s
18	0.430s	1.320s
20	0.422s	1.470s
22	0.465s	1.400s
24	0.438s	1.350s
26	0.419s	1.300s
28	0.398s	1.286s
30	0.370s	1.221s
32	0.374s	1.149s
34	0.363s	1.100s
36	0.357s	1.044s
38	0.360s	1.000s
40	0.344s	0.969s

IV. SCALABILITY ANALYSIS

In this section, we will analyse the scalability of the program, specifically the *strong* analysis. As we know, we have to check two items: speed-up increase with Process Unit for a fixed problem data size and the ideal speed-up is proportionally to Process Unit. To do that, we have to look to Amdahl's law. This law says that, in a program with parallel processing, instructions that have to be performed in sequence will have a limiting factor on program speedup such that adding more PU's may not make the program run faster.

Consequently, we decided to show the results in the following diagram:



We calculate the ideal speed-up with 4 and 32 clusters using the Amdahl's law maximum speed-up function: $S_p \leq \frac{1}{f + \frac{1-f}{p}}$,

where f is the fraction of non-parallelizable work and p is the number of PU's (on our case, threads). We estimate that the sequential time non-parallelizable for 4 clusters is 8% and for 32 clusters is 4%. Then, we calculate the speed-up for both cluster cases with the results got from the tables in the previous section.

V. CONCLUSIONS

Analysing the results and the graph from the previous sections, we can conclude that:

- The increase of the performance is way more relevant when we are talking about few threads. It tends to be logarithmic. In other words, from the moment when we changed 2 to 4 threads or 4 to 6, we can see a more significant performance increase than when we changed from 20 to 22 or 22 to 24 threads;
- After a certain number of threads, the performance stops to linearly increasing e turns unstable. In other words, due to a certain number of threads and its further treatment, it is hard to evaluate if the increase of the number of threads really increases the program performance, because, even though makes the defined region faster, it will take longer in the critical section;
- The more clusters we have, greater and more significant is the performance gain. This is explained by, in the costly region, the size increasing of a cluster, which means that N points (e.g. 10 000 000) will have to calculate the distance till that cluster and compare with other points. But in the critical region, the size increasing of a cluster only increases one cycle in the execution.

REFERENCES

- [1] Perf website - <https://perf.wiki.kernel.org/>
- [2] srand manual page - <https://en.cppreference.com/w/c/numeric/random/srand>