Practical Assignment #1 - 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 first phase of Parallel Computing course at University of Minho. It begins with a small introduction of the subject of this assignment. Then we explain all the optimizations we found to apply to the code. After that we show the results obtained in the SEARCH and we finalized with the conclusions got from this assignment.**

***Index Terms*—K-Means, Vectorization, Loop Unrolling, Optimization, SEARCH, Algorithm**

I. Introduction

The purpose of this practical work is to implement in C the *k-means* algorithm (inspired from *Lloyd's algorithm*) and we must have to apply some optimization code techniques and use code analysis tools (e.g. Perf [1]), as we'll enumerate in the following sections.

II. Optimizations

A. Flag -O2 in compilation

The first optimization that we did was at compiler level with the use of the flag `-O2`. Before this version, our program used to run more than 1 minute. Across all the versions of our code, we went put several flag such as `-O3`, `-funroll-loops`, `-ftree-vectorize -msse4 -mavx`, ... But none of them improved the performance significantly.

Our test flag scenarios hit always in 2 cases:

- lower the number of instructions (#I), but the number of cycles increased a lot, making our program slower in terms of execution time;
- doesn't show significant changes in the metrics values obtained.

B. Loop Fusion

In a early stage of the algorithm implementation, it iterated over the points twice - one for the point to nearest cluster assignment and another to check if the clusters had been converged (verifying point by point if they were assigned to the same cluster). As we checked that the algorithm was doing redundant work, we changed the code later to only iterate once over the points.

C. Strength Reduction

As in our problem in hand only interests us the shortest distance, we came to a conclusion to remove the square root calculus from the euclidian distance expression, since $\sqrt{a} < \sqrt{b}$ is the same as $a < b$. We also changed the call of the function `pow(x,2)` from `math.h` to do just only `x*x` - since is the same in mathematical terms.

At our result section, this was the most efficient optimization, since the square root is a powerful calculus and, after this change, it does less 10 millions (points) * 4 calculus per program execution.

D. Delete struct

On our first version, we had a struct that contained 2 *floats* (one representing the *x* value and another representing the *y* value) called `POINT`. We also had two struct arrays - one for the sample points and another for the clusters. In this version, we removed the struct and now we only got two float arrays (representing the *x* values and the *y* values, respectively).

We believed that this could improve the program performance, but we were wrong. Since the distance calculus needs two points, the compiler needs to go to all the four diferent arrays sequentially, increasing the number of cache misses. Now using the struct, the compiler accede to the point at once.

E. Function calls in loops

Even though the function calls in the several loops makes the code more modular and cleaner, calling functions inside the loops can cause unnecessary delays and difficulties to the compiler on the loop optimization.

F. Loop Unrolling

One of the optimizations was loop unrolling. We tried loop unrolling with two instructions per loop and loop unrolling with four instructions per loop and. In other words, the for loop do the same calculus 2 times by iteration and then the control loop variable increments 2 at the time (in the loop unrolling with 4 instructions is with the same line of thought).

We choose between 2 or 4 (arbitrary values) because the gcc loop unrolling works better with non prime number of iterations, but, after creating two different versions, we checked that the number of cycles of V4 increased. So, we stick with the version with the factor of 2.

III. Results

As expressed in the assigment, we must run our program in the SEARCH. So, in order to do that, we compiled our several versions of our code in *gcc* (version 7.2.0) and, with the help of the command `srun --partition=cpar perf stat -e L1-dcache-load-misses,instructions,cycles -M cpi make run`, we obtained the following result expressed in the following table:

	ExecTime	#I	#cycles	Cache misses (L1)	CPI
VA	19.257 s	70629 M	62645 M	137.2 M	0.9
VB	17.614 s	75264 M	53943 M	85.3 M	0.7
VC	3.821 s	29588 M	12141 M	79.9 M	0.4
VD	3.879 s	29591 M	12019 M	87.0 M	0.4
VE	3.785 s	29587 M	12102 M	79.8 M	0.4
VF(2)	3.691 s	26798 M	11903 M	79.7 M	0.4
VF(4)	3.844 s	24883 M	12020 M	79.7 M	0.8

We want to highlight that all the versions refers to their subsection letter in the optimization section II and follow the optimizations of his previous versions (i.e. VC contains VA and VB optimizations), but the consequent versions of VD are the exception of the rule because VD was the optimization that went worse in terms of performance - we ignored this version in the following versions.

As an example, here is the output got from the refered command in the SEARCH using on our final version:

```
[a93277@search7edu2 vf2]$ srun --partition=cpar perf stat -e L1-dcache-load-misses,instructions,cycles  -M cpi  make run
./bin/k_means
N = 10000000, K = 4
Center: (0.250, 0.750) : Size: 2499108
Center: (0.250, 0.250) : Size: 2501256
Center: (0.750, 0.250) : Size: 2499824
Center: (0.750, 0.750) : Size: 2499812
Iterations: 39

 Performance counter stats for 'make run':

         79775183      L1-dcache-load-misses
      26798831042      instructions              #    2.25  insn per cycle
      11903315252      cycles
      26798831042      inst_retired.any          #      0.4 CPI
      11943315231      cycles

       3.723327880 seconds time elapsed

       3.691544000 seconds user
       0.032150000 seconds sys
```

Fig. 1. Output got from SEARCH (N = 10000000; K = 4) - final version

Note: `N` refers to the number of samples and `K` refers to the number of clusters

IV. Conclusions

After checking our results, we came to the following conclusions:

- A good optimized algorithm can take to a massive improvement in terms of performance and execution time of a program - from our first version to the last one we went from 19s to 4s;
- The number of cycles influences directly with the program execution time;
- Not always a low number of instructions means the decrease of the execution time, as the instructions can simply take longer to be executed;
- A low number of cache-misses influences also with the execution time, lowering it, since the memory access has a high cost;
- Machines are not created equal. The values obtained on SEARCH weren't the same as obtained in our computers. For example, the version `vf_4` is faster in our machines than in SEARCH.

References

[1] Perf website - https://perf.wiki.kernel.org/
[2] K-means algorithm in C++ - https://reasonabledeviations.com/2019/10/02/k-means-in-cpp/
[3] Optimization of Computer Programs in C - http://icps.u-strasbg.fr/ bastoul/local_copies/lee.html