Load balancing in standard kMeans

Suppose that the standard kMeans algorithm is executed in parallel by splitting between T threads the total points N. In this case the work load is easly divided equally between the threads. In fact we can calculate the total number of points to assign each thread by N/T; if N is not a multiple of T we will also have a reminder

$$reminder = N \bmod T < T \tag{1}$$

If this is the case, we can simply assign the first threads one more point, therefore in the end there will be r threads with N/T + 1 points and T - reminder threads with N/T points.

Load balancing in Hamerly's algorithm

With Hamerly's algorithm, efficient load balancing is not as easy as in the standard version because, for every cycle, it is not neccessary to compute the distances for every points but only for critical ones. Because of this, if the points were simply divided among the threads, it might happen that some have many points while others could have few calculations to do.

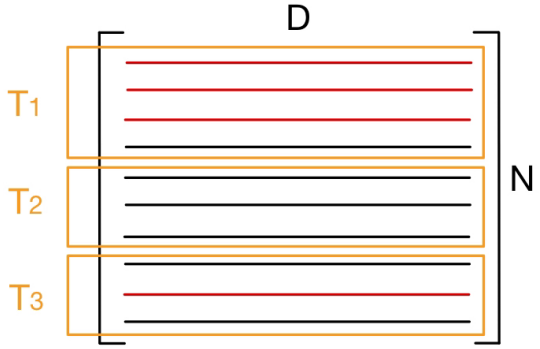

Figure 1: *Non balanced point division*

The Figure 1 shows an input matrix (N x d, where d is the data dimensionality) where critical points are not equally distribuited. In this case, the first thread will have three critical points whereas the second thread will have zero.
To implement an efficient load balacing, a more advanced strategy is needed and one possible solution will be presented in the following part of this report.

Points assigment for Hamerly's algorithm

First of all remember that, in the Hamerly's algorithm, after that each centroid moved it is neccessary to update the upper bound (ub) and the lower bound (*lb*) of each p. This operation can be done in parallel, since the update is indipendent between the points, and the work that has to be done is equal for each p.
During this phase, each thread will have a linked list where they add every critical point that it finds, moreover every time it finds a critical it increments by one the value of a global array which has a length of T so that each thread has its own index to access.
After that all points have been iterated a synchronization is required. During this sequential part it is calculated the total number of critical points (C) that have been found and a second array is created with a length of C.

At this point, every thread can insert, inside this new array, its critical points. In order to do that, every thread has to start inserting its values from the index which is equal to the sum of the number of criticals found by the previous threads.
For example, if there are 3 threads and they found respectively [2, 3, 1] criticals then, they start inserting values from the 0, 2 and 5 (2 + 3) index. The objective of this step is to create an array containing all the critical points together because in this way it is possible to equally divide those critical between the threads.
If the total number of critical points is C then, the number of criticals to assign each thread can be calculated as C/T. If C is not a multiple of T, there will also be a remainder which can be handled the same way as in the standard case. Below is the pseudo code where N is the total numer of points and "points" is the array containing all points.

Algorithm 1 Boundaries update pseudo-code

```
                                                    ▷ Sequentially
Let nC be the array with the number of criticals found by each thread
remainder = N % T
                                                    ▷ In Parallel
Let criticals_per_thread be the local list of criticals found by the thread
p_per_thread = Int(N/T)
offset = remainder
if thread_id < remainder then
    p_per_thread = p_per_thread + 1
    offset = 0
end if

for j < p_per_thread do
    p = points[p_per_thread · thread_id + offset + j]
    p.ub = p.ub + cdistaces[p.centroid_index]
    if p.centroid_index == maxdist_index then
        p.lb = p.lb - secmaxdist
    else
        p.lb = p.lb - maxdist
    end if
    if p.ub > p.lb then
        criticals_in_thread.push(p)
        nC[thread_id] = nC[thread_id] + 1
    end if
end for
                                                    ▷ Sequentially
C = 0
for j < nC.length do
    C = C + nC[j]
end for
Let criticals be the array of all the criticals
                                                    ▷ In Parallel
index = 0
for j < i do
    index = index + nC[j]
end for

for j < nC[i] do
    criticals[index + j] = criticals_per_thread.pop
end for
```

Insight into the opseudo code

Because some of the steps in the pseudocode may not be immediate in understanding what they do and why they are used, this part will give an overview of those instructions that may be less clear.

Algorithm 2 N_i increment

```
if thread_id < remainder then
    p_per_thread = p_per_thread + 1
end if
```

This if statement is implemented to increase by one the number of points that the first threads have to access.

Algorithm 3 p.centroid.distance

```
p.ub = p.ub + cdistances[p.centroid_index]
```

Through cdistances[p.centroid_index] we access distance traveled by the centroid to which the point has been assigned.
At every iteration the upper bound of each point, which is accessed by p.ub, has to be updated and the new value will be: p.ub + cdistances[p.centroid_index]

Algorithm 4 lower bound update

```
if p.centroid_index == maxdist_index then
    p.lb = p.lb - secmaxdist
else
    p.lb = p.lb - maxdist
end if
```

When it comes to updating the lower limit, the first thing to check is whether the centroid that moved the most is also the one assigned to the p. In this case, the lower limit cannot be updated with the distance of that centroid because it has already been used for the upper limit, but in this case the distance taken from the second centroid that moved the most has to be used.

Algorithm 5 lower bound update

```
index = 0
for j < thread_id do
    index = index + nC[j]
end for
```

This is the part where each thread calculate the index from where it has to start inserting the vales in the global array containing all the critical points.

Algorithm 6 add values to the global array

```
for j < nC[i] do
    criticals[index + j] = L_i.pop
end for
```

This is the last part of the algorithm where every thread inserts into the global array the value of its list.