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 (r)

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

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

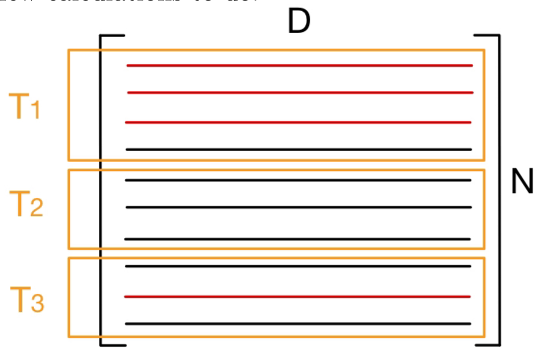

Figure 1: *Non balanced point division*

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 (μ) and the lower bound (l) of each point. This operation can be done in parallel, since the update is indipendent between the points, and no special cautions are required when splitting the points because the work that has to be done is equal for each point.
During this phase, each thread will have a linked list where it adds 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 that have been found, also, during this step, a second array is created that length

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 N_c then, the number of criticals to assign each thread can be calculated as N_c/T. If N_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 P is the array containing all points.

Algorithm 1 kMeans class assigment pseudo-code

```
————————————————————                ▷ Sequentially
Let $C_{max}$ be the centroid which moved the most
Let $C_{2nd}$ be the 2nd centroid which moved the most
Let nC be the array with the number of criticals found by each thread
(Remainder) r = N % T
————————————————————                ▷ In Parallel
Let i be the id of the current thread
Let L be the list of criticals found by the thread
$N_i = Int(N/T)$
offset = r
if $i < r$ then
    $N_i = N_i + 1$
    offset = 0
end if

for $j < N_i$ do
    point = P[$i \cdot N_i$+ offset]
    point.$\mu$ = point.$\mu$ + point.centroid.distance
    if point.centroid == $C_{max}$ then
        point.l = point.l - $C_{2nd}.distance$
    else
        point.l = point.l - $C_{max}.distance$
    end if
    if point.$\mu$ > point.l then
        L.push(point)
        nC[i] = nC[i] + 1
    end if
end for
————————————————————                ▷ Sequentially
sum = 0
for j < nC.length do
    sum = sum + nC[j]
end for
Let cL be the array (of length sum) with all the criticals
————————————————————                ▷ In Parallel
index = 0
for j < i do
    index = index + nC[j]
end for

for j < nC[i] do
    cL[index + j] = L.pop
end for
```