# Parallel Implementation of K-Means Algorithm

## Problem Domain

Given:

P circles with radius Ri and a center (Ai, Bi) (P is a natural number)
C clusters (C is a natural number)
Time interval T >= 0
Time step deltaT (s.t 0 = T mod deltaT)
Limit iterations (Limit is a natural number)

For each deltaT in [0,T]:

Derive a N points (on the XY plain) from the N circles,
Run K-Means on that set of point to find C clusters centers
For each cluster center:
Find the minimal distance between two centers

Output:

C cluster centers of the deltaT iteration where the minimal distance between two cluster Centers was found

---

Formula for calculating point p from corresponding circle c:

Pi.x = Ai + Ri * cos(2*pi*deltaT/T)
Pi.y = Bi + Ri * sin(2*pi*deltaT/T)

---

## Sequential Solution

1. Load input from file
2. For each deltaT in [0,T]:
    1. Calculate N points from circles
    2. Run K-Means with C clusters for that set of points:
        1. Choose C arbitrary cluster centers Ci(Xi,Yi), i = 0, 1,...C-1.
        2. For every point p in [0, N-1]:
            Calculate the shortest distance between p and cluster center c
            Associate point p with the closest center
        3. For each cluster c in [0, C-1]:
            Recalculate the cluster's center by:
            Ci.x = AVG(Pi.x)
            Ci.y = AVG(Pi.y)

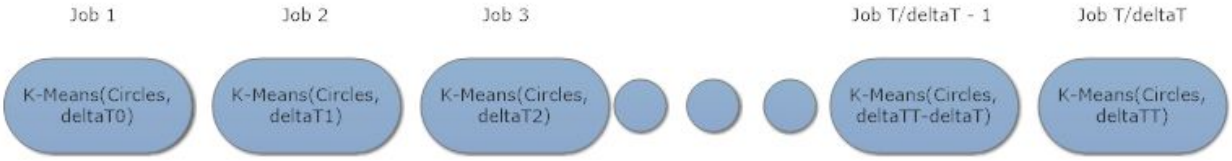
4. Check if the the new cluster center changed, if so, and if the number of K-Means iteration < Limit , continue to step 2, else finish.
5. For each K-Means cluster C:
   calculate the the minimal distance between two clusters
6. Find the deltaT iteration in which the minimal distance between two cluster's centers was found

The complexity of the K-means algorithm is O(Limit*(P + (P*C) + (C*C)) =~ O(Limit * P * C)
This is done N times, so that the total complexity of the program is O(N*Limit*P*C)

---

## Adding parallelism through MPI

The main load of the algorithm resides in the K-Means calculation of each of the deltaT iterations.
The problem could be described as a set of K-Means jobs on a given set of points, derived from the input circles for a given deltaT:




It is apparent that a good use of MPI could be achieved by scattering the K-Means jobs between the processes. One necessity is to distribute the common data between the processes (the circles for the point calculation, the number of clusters, the number of circles). The question was now how to distribute the jobs between the processes.

One of the option was using a master slave that allocate a job to each of the processes, wait for the first answer that, allocate another job for the process that sent the answer and check for the minimal distance among the answers it received. There are two main issues with that solution: one is the overhead of recurrent interprocess communication and the second is that the processing power of the master process is lost on allocating jobs and waiting for answers from the other processes. To utilize the master's computing power, the job allocation should be done in advance, giving each process a chunk of the jobs to process.

One option of job chunk allocation is using MPI_scatter to scatter an array of deltaT's and MPI_gather to collect the answers, but that solution requires spending time both building the array structure and distributing it among the processes. A better solution would be implementing a jobAllocation() function, that the master uses to submit job chunk to each process, consisting of the start timeStep and the numberOfJobs to process. Taking into account only three nodes and three processes, this function should consume minimal time and leave the master available

to do calculations after finishing allocating the job range to each process and to himself.  The allocation function shoud take into accout a modulus job residual and an even distribution of it between the processes.

The code was implemented in such a way that it will work for a single process as well, regardless of how many nodes there are.

## Adding OpenMp parallelism

To utilize the computing power of each node for calculating the set of K-Means jobs, I used OpneMP.

OpenMP was mostly implemented on for loops where it was possible to devise independent loops without compromising efficiency in case of a system that is oblivious to the library. The first, most calculation rich OpenMP loop was the K-Means jobs loop. It initially was designed to calculate each K-Means and to check for the minimal distance between two clusters from one iteration to the next. In order to allow for loop independence, the search for the minimum was taken out of the for loop.

Two nested OpenMP structured blocks were also added - one in the point calculation for that given iteration and another for recalculating the centers in each K-Means iteration, to know whether K-Means is done.

Table one describe the amount of time it takes to complete the algorithm using three nodes, and OpenMP

After parallelism, the complexity of the K-means algorithm is O(Limit*P*C) in total

## Testing

Testing was done the following way: Using three nodes, fist I ran the program with only MPI with an increasing number of processes. Then the same was done, but with the OpenMP implementation.

Testing was done for two set of input:

1. 10 K-Means iterations over 300000 points
2. 1000 K-means iterations over 300000 points

Table 1 sums up the results

| kmeans 10 (300000 3 0.100000 1.000000 100) | | | | | | |
|---|---|---|---|---|---|---|
| NUMBER OF PROCESSES | 1 proc | 2 proc | p3 | p4 | p5 | p6 |
| WITHOUT OPENMP | 3.392663 | 2.390609 | 2.05047 | 1.86631 | 1.801573 | 1.692072 |

| | | | | | | |
|---|---|---|---|---|---|---|
| WITH OPENMP | 1.769061 | 2.416299 | 2.081704 | 1.919657 | 1.899545 | 1.695382 |
| kmeans 100 (300000 3 0.100000 100 100) | | | | | | |
| NUMBER OF PROCESSES | p1 | p2 | p3 | p4 | p5 | p6 |
| WITHOUT OPENMP | 705.522285 | 193.386982 | 68.439634 | 71.780872 | 49.3904 | 35.627853 |
| WITH OPENMP | 70.776222 | 129.191517 | 66.680798 | 54.325574 | 47.611576 | 34.765615 |

| |
|---|
| Expected result for kmeans10 |
| General min dist is 141.419857, the time is 0.800000 |
| Final Centers (x = 8.771225 , y = -29.031166) |
| Final Centers (x = 108.769876 , y = 70.968063) |
| Final Centers (x = 208.771365 , y = -29.031713) |
| Expected result for kmeans100 |
| General min dist is 141.328812, the time is 2.900000 |
| Final Centers (x = 43.556398 , y = 19.476992) |
| Final Centers (x = 143.471440 , y = 119.431069) |
| Final Centers (x = 243.546576 , y = 19.454962) |

Looking at the results can see that for a small set of K-Means iteration, the scaling of the program is not so good, and that adding OpenMP doesn't contribute much to the speedup, unless only one process is running.

For a thousand K-means iterations, both MPI and OpenMP contribute to program speedup, but with some exception:
While adding OpenMP to the code contribute to speedup, it nonetheless has some detrimental effect on scaling from 1 process to two, probably due to overhead. For three processes we see that the OpenMP significance is gone comparing with the performance using MPI only.