# Parallel Implementation of K-Means Algorithm

# Problem Definition

Given a set of points in **n**-dimensional space. Implement simplified K-Means algorithm to find K clusters with best quality measure.

1. Set K = 2
2. Choose first K points as a cluster centers
3. Group points around the given cluster centers - for each point define center that is most close to the point
4. Recalculate the cluster centers – average of all points in the cluster
5. Check the termination condition – no points move to other clusters or maximum iteration LIMIT was performed.
6. Repeat from 3 till the termination condition fulfills.
7. Evaluate the Quality of the clusters found. The Quality is equal to an average of diameters of the cluster divided by distance to other clusters. For example, in case of k = 3 the quality is equal

**q = (d1/D12 + d1/D13 + d2/D21 + d2/D23 + d3/D31 + d3/D32) / 6**,

where di is a diameter of cluster **i** and Dij is a distance between centers of cluster **i** and cluster **j**.

1. Stop if Quality is less than Predefined Quality
2. Increment the value of K by 1, finish if K is bigger than predefined maximum value **MAX**
3. Return to step 2

Input data and Output Result of the project

You will be supplied with the following data, taken from

* N - Number of points
* MAX - maximum number of clusters to find
* LIMIT – the maximum number of iterations for K-MEAN algorithm.
* QM – quality measure to stop
* Coordinates of all points

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 OpenMP.

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.

Calculation of the distance between 2 points

Colclusion : small loops are not worth it – 52

Small input 10 X 2

Seq = 0.5

Par = 0.65

Big input

Seq = 14.2

Par = 20

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

Adding CUDA parallelism

To utilize the GPU processing power in big amount of small math operations I used CUDA.

CUDA was mostly implemented to find maximal distance between points in cluster

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.