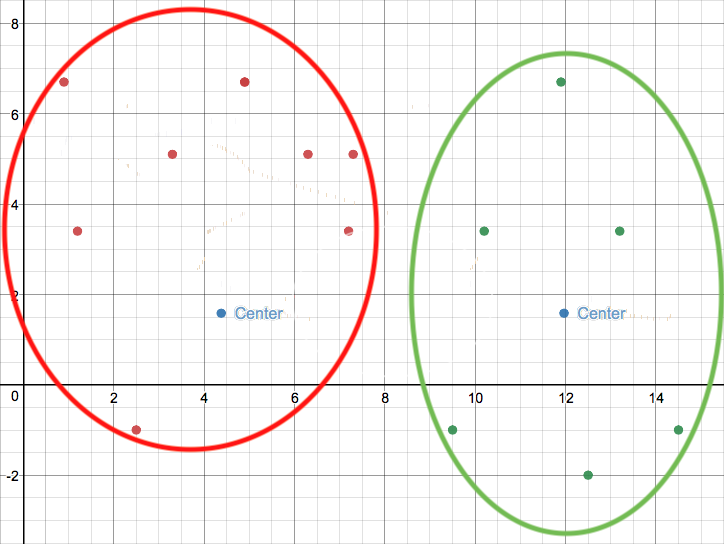
**Parallel implementation of K-Means**

**by Roie Danino**

Final project

Course 10324, Parallel and Distributed Computation

2017 Summer Semester

this photo was created using [Desmos](https://www.desmos.com/calculator) tool for drawing graphs, using results from my implementation.

**Abstract**

**K - means** clustering is a highly used machine learning algorithm for partitioning a set of points, vectors or any comparable set of objects, into given number of clusters k, that in each cluster the objects are closer to each other than to objects in other groups. the clusters are basically same data space objects that their parameters are the exact average of all the vectors in their area.

In this project, the mission was to implement a parallel and dynamic version of k-means, in which that instead of executing k-means once with a specific k, executing it with k = 2 and then increasing k and executing it again and again until the desired quality reached.

**The Serial Algorithm**

**Taken from Dr. Boris Morose project requirements document.**

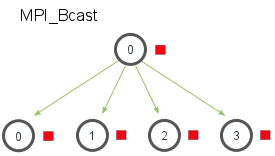
1. Set K = 2
2. Choose first K points as cluster centers
3. Group points around the given cluster centers - for each point define center that is most close to the point
4. Recalculate the cluster centers – average of all points in the cluster
5. Check the termination condition – no points move to other 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

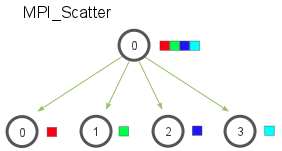
**q = (d1/D12 + d1/D13 + d2/D21 + d2/D23 + d3/D31 + d3/D32) / 6**,

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

**The Parallel Algorithm**

1. Root process reading vectors and desired quality from file.
2. Root process shares the meta-data info such as dimension, max-k, limit of iterations in k-means and wanted quality measure using MPI\_Bcast method.
3. Root process divides equally the vectors to the different processes including himself using MPI\_Scatter method, the remaining of the division remains in the root process.



1. Now that each process received his share of the points, the dynamic k-means begin with creating the clusters, different process are responsible for different clusters, the remaining is divided between processes. Example: k = 2 and 3 processes, P0 responsible for one cluster, P1 responsible for another one, and P2 responsible for none and as the k increases to k = 3 P2 will be responsible for the new cluster and so on, the processes creating new clusters by turns as the k increases to keep the load balancing as equal as possible.

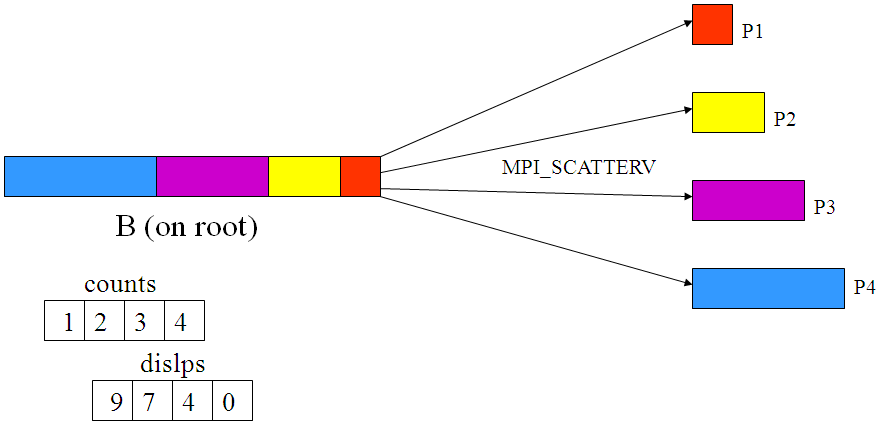
For process Pi:

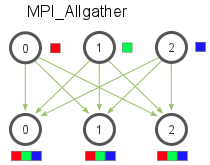
myNumOfClusters = k / numOfProcesses

if (k % numOfProcesses > i)

myNumOfClusters++

1. Root process scattering the first k points to other processes as the clusters centers, this time using MPI\_Scatterv that enables sending different amounts of data to different processes because the number of clusters can be different between processes.



1. The processes share their centers using MPI\_Allgather so all the processes will acknowledge all the centers of the k clusters while knowing to which process each center belongs.
2. Each process matches his points to the k centers using CUDA in two steps:

1. Calculating the Euclidian distance for each point with each center, creating a matrix with: k rows and column for each point, each CUDA kernel thread calculates 1 result in the matrix.

Distance (Pi, Cj) =

Pi – the point of this process

Cj – the center from k centers

[n] – the n parameter of the object

Result table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Cj \ Pi | *P0* | *P1* | *P2* | *P3* |
| *C0* | D (P0, C0) | D (P1, C0) | D (P2, C0) | D (P3, C0) |
| *C1* | D (P0, C1) | D (P1, C1) | D (P2, C1) | D (P3, C1) |

1. Choosing the minimal distance for each product, each CUDA thread find the minimum for one point and writing the result into a 1-dimensional array that in the element is the closest cluster id for Pi.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Cj \ Pi* | *P0* | *P1* | *P2* | *P3* |
| *C0* | D (P0, C0) = 1 | D (P1, C0) = 4 | D (P2, C0) = 5 | D (P3, C0) = 8 |
| *C1* | D (P0, C1) = 2 | D (P1, C1) = 3 | D (P2, C1) = 6 | D (P3, C1) = 7 |

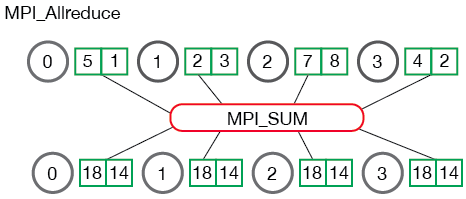
Result array:

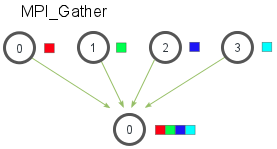
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Pi* | *P0* | *P1* | *P2* | *P3* |
| *Nearest Mean* | *C0* | *C1* | *C0* | *C1* |

1. With the result array from step 7, each process inserts all his points into a matrix, *pointsToSend*, with the number of rows as the number of the processes and the number of columns as the number of the points in this process, such that, that in the row there will be the points that need to be sent to the process with rank i.

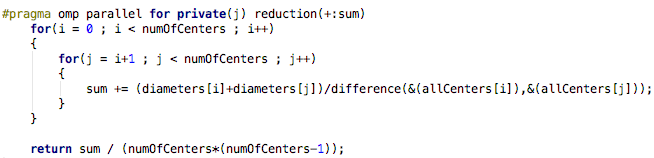
Assuming that process0 has cluster c0 and process1 has c1:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Rank |  |  |  |  |  |
| 0 (has C0) | P0 | P2 | P4 |  |  |
| 1 (has C1) | P1 | P3 |  |  |  |

1. Each process scattering the matrix from step 7 using MPI\_Scatter. Then, removes the products that he just sent and adds those he received.
2. Each process calculates his clusters new coordinates, using OMP *parallel for*, so each thread calculates the mean of another clusters.
3.  Each process keeping in track whether any of the points has changed its cluster in a Boolean variable called *hasChanged*, then using MPI\_Allreduce to sum them all from all the processes and return them the result of the summation, so there will be no situation that one process had no changes and exited by himself, if the sum of the “*hasChanged”* s greater than 0, a change was made, repeat from step 6.
4. Each process calculates his clusters diameters using OMP *parallel for* so different cluster diameters are calculated at the same time, then, gathering all the diameters from all processes to the root process using MPI\_Gatherv because different processes might have different number of clusters.



1. Root process measure the quality, also using OMP *parallel for* with *reduction* to sum the partial sums of the error, and checking if k hasn’t reached yet max-k, then broadcasting *stop* using MPI\_Bcast, which is the termination condition: error < desired quality AND k < max K, if this condition returns false, increase k by one and repeat from step 5.



1. Root process write result centers to file.

**The Rational of Choosing This Architecture**

The reasons I chose this specific architecture are:

1. Different clusters centers can be parallel calculated in different processes because the points are with their clusters.
2. Different clusters diameters, which is the part with the highest complexity calculations of this algorithm, can be parallel calculated in different processes for the same reason.

Chose to use CUDA for matching points to clusters because the problem can be referred to as a two-dimensional problem consists big amount of little calculations which fits CUDA GPU architecture the most.

Chose to use OMP for calculating clusters centers because the problem consists little amount of big jobs, each thread might need to handle calculating average of hundreds of vectors, but the amount of those jobs is relatively small, one for each cluster, considering that the number of clusters is much smaller than the number of points.

The calculations of the diameters are the heaviest calculations in the program:

Each diameter calculation is O() when n is the number of points in the cluster, the architecture allowing calculating the different diameters in different processes and in each process to calculate different diameters in different OMP threads.

**Complexity Evaluation**

*k* = number of clusters

*n* = number of products

*d* = dimension

*t* = number of threads in each process

*p* = number of processes

1. Matching products to clusters:

Serial: O(*knd*)

Using CUDA:

O(*d*) for k\*n threads calculating distances parallel in O(*d*)

O(*k*) for finding the minimum for each point

Total: O(*d*) + O(*k*) + CUDA overhead.

1. Calculating clusters new centers:

Serial: O(*knd*)

Using MPI and OMP: O()

1. Calculating Diameters:

Serial: O()

Using MPI and OMP: O()

Total: O ( + + *d*) O()

For Max-K times calling k-Means: O()