

# Introduction

This report outlines and analyses an implementation of a message passing parallel algorithm for k-means++ clustering and the n-body simulation.

The algorithm is implemented with Open MPI where a number of nodes collaborate and passes messages to each other to carry out the task required.

The task is described as follows ( $n$  denotes the total number of points and  $m$  denotes the total number of nodes):

1.  $n$  points are sampled by all  $m$  nodes from a Gaussian mixture model (GMM).
2. All nodes collaborate to cluster the points into  $m$  clusters using k-means++.
3. Each cluster is moved into one single node so that a single node contains all points of a cluster.
4. The nodes collaborate with each other to perform n-body simulation where each point represents a body with a mass of 1 and bodies are attracted to each other via gravity.

The implementation for sampling all  $n$  points and clustering them with k-means++ are all rather naive and straightforward. The interesting part is the n-body gravity simulation.

To reduce the simulation time, the Barnes-Hut simulation approximation algorithm is used so that points over a certain distance away are simplified into a single center of mass when calculating the force they exert on another point. To reduce the communication overhead, each node constructs a Barnes-Hut tree and then prune the tree for each other nodes so that the tree only contains the nodes that are actually needed for the calculation of points in each other node. This pruned tree will from now on be referred to as the partial tree.

For node A to construct a partial tree to be sent to node B, we would like to make sure that any node of the tree where  $s/d < \theta$  is kept, where  $s$  is the width of the region the node of the tree represent,  $d$  is the distance between the center of mass of the node and the closest point in cluster B to the center of mass to the node, and  $\theta$  is a predefined constant. However, we do not want to calculate the closest points in cluster B to every node in the Barnes-Hut tree for cluster A and that would be slow in both computation time and communication overhead. Hence, we instead take the closest point in cluster B with respect to the center of mass of all points in cluster A, and calculate a hyperplane that is perpendicular to the line through the center of mass and the closest point, that goes through the closest point. Assuming that all points in each cluster are closer to their center of mass than any points in other clusters, the distance from any point within cluster A to this hyperplane should be shorter than the distance from point within cluster A to any point within cluster B.

The pseudocode describing the algorithm is shown in Algorithm 1. Note that the pseudocode is written in the perspective on one computing node.

While in theory, this algorithm should be faster than naively parallelizing Barnes-Hut algorithm by transferring all points to a root cluster and calculating one single Barnes-Hut tree to be sent to all clusters. In practice, it turns out the algorithm has a number of other overheads.

These overheads include calculating  $m$  partial trees (again here  $m$  is the total number of computing nodes), encoding them into an array of bytes to be sent over to each node and decoding the bytes into a partial tree, plus the extra computing time of creating  $m^2$  partial trees. Furthermore, because each simulation step relies on the assumption of points of each all points in each cluster are closer to their center of mass than any points in other clusters being true, the clusters have to be recomputed and points that changed clusters sent over the corresponding nodes. This adds additional overheads that, when combined with all the other overheads mentioned above, might outweigh the benefit of not having to send all points to all nodes.

In the future, it would be interesting to compare the performance of the current implementation against the naively parallelized Barnes-Hut algorithm described above.

Due to time constraint, the implementation is also not parallelized within a node. As a result, this does not take advantage of the multicore nature of each node. In the future, it'll be interesting to see how much extra performance can be extracted via parallelization within each node.

---

**Algorithm 1**

---

```
1:  $m \leftarrow$  number of compute nodes.
2:  $NodeIndex \leftarrow$  the index for the compute node.  $\triangleright$  This is equivalent to MPI_Comm_rank but 1
   indexed.
3:  $N \leftarrow$  number of points
4:  $D \leftarrow$  number of dimensions
5:  $c \leftarrow$  number of GMM components
6: for  $i \leftarrow 1$  to  $c$  do
7:    $prob[i] \leftarrow$  the probability of the  $i^{th}$  GMM component
8:    $gmmc[i] \leftarrow$  the  $i^{th}$  GMM component
9: end for
10: for  $i \leftarrow 1$  to  $N/m$  do
11:    $points[i] \leftarrow \text{SAMPLE}(gmmc[\text{SAMPLEWEIGHTEDDISCRETEDISTRIBUTION}(prob)])$ 
12:    $\triangleright \text{SAMPLEWEIGHTEDDISCRETEDISTRIBUTION}(weights)$  generate a random number between 1
   and the length of  $weights$  with the number  $i$  having a probability of  $weights[i]/\text{SUM}(weights)$ 
13: end for
14:  $centroids[1] \leftarrow \text{CHOOSERANDOMONE}(points)$   $\triangleright$  choose centroid for k-means++
15: for  $i \leftarrow 2$  to  $m$  do  $\triangleright$  choose  $m$  centroids for  $m$  clusters
16:   for  $j \leftarrow 1$  to  $\text{LEN}(points)$  do
17:      $distances[i] \leftarrow$  distance of  $points[i]$  to the closest centroid
18:   end for
19:    $DistSum \leftarrow$  sum of distances
20:   if  $NodeIndex = 1$  then
21:     for  $i \leftarrow 1$  to  $m$  do
22:        $AllSum[i] \leftarrow DistSum$  from node  $i$ 
23:     end for
24:      $NextCentroidNode \leftarrow \text{SAMPLEWEIGHTEDDISCRETEDISTRIBUTION}(AllSum)$ 
25:   end if
26:    $NextCentroidNode \leftarrow NextCentroidNode$  from node 1
27:   if  $NodeIndex = NextCentroidNode$  then
28:      $NextCentroid \leftarrow points[\text{SAMPLEWEIGHTEDDISCRETEDISTRIBUTION}(distances)]$ 
29:   end if
30:    $centroids[i] \leftarrow NextCentroid$  from node  $NextCentroidNode$ 
31: end for
32:  $ClusterIndices \leftarrow \text{KMEANS}(centroids, points)$ 
33:  $points \leftarrow$  points in  $NodeIndex^{th}$  cluster as indicated by  $ClusterIndices$  in all nodes  $\triangleright$  Significant
   implementation detail of message passing between nodes omitted.
34: for  $i \leftarrow 1$  to  $\text{LEN}(points)$  do
35:    $velocities[i] \leftarrow 0$ 
36: end for
37:  $InitVariance \leftarrow \text{GETVARIANCE}(points)$ 
38: while  $True$  do
39:    $\text{SIMULATE}(points, velocities)$ 
40:    $CurrentVariance \leftarrow \text{GETVARIANCE}(points)$ 
41:   if  $CurrentVariance < InitVariance/2$  then
42:     break while loop
43:   end if
44:    $ClusterIndices \leftarrow \text{KMEANS}(centroids, points)$   $\triangleright$  one iteration only, implemented separately in
   real code but reusing pseudocode here.
45:    $points \leftarrow$  points in  $NodeIndex^{th}$  cluster as indicated by  $ClusterIndices$  in all nodes
46: end while
```

---

---

**Algorithm 2**

---

```
1: function KMEANS(centroids, points)
2:   while centroids changed since last iteration do
3:     for  $i \leftarrow 1$  to LEN(points) do
4:        $ClusterIndices[i] \leftarrow$  index of centroid closest to points
5:     end for
6:     for  $i \leftarrow 1$  to LEN(points) do
7:        $sums[ClusterIndices[i]] \leftarrow sums[ClusterIndices[i]] + points[i]$ 
8:        $counts[ClusterIndices[i]] \leftarrow counts[ClusterIndices[i]] + 1$ 
9:     end for
10:    if NodeIndex == 1 then
11:      for  $i \leftarrow 1$  to LEN(centroids) do
12:         $allSums \leftarrow$  sum of  $sums[i]$  from all nodes
13:         $allCounts \leftarrow$  sum of  $counts[i]$  from all nodes
14:         $centroids[i] \leftarrow allSums/allCounts$   $\triangleright$  Here it is actually each dimension of allSums
        divided by allCounts.
15:      end for
16:    end if
17:     $centroids \leftarrow centroids$  from node 1
18:  end while
19:  return ClusterIndices
20: end function
```

---

---

**Algorithm 3**

---

```
1: function GETVARIANCE(points)
2:    $mean \leftarrow$  average of all points
3:   if NodeIndex = 1 then
4:      $sum \leftarrow 0$ 
5:      $allSize \leftarrow 0$ 
6:     for  $i \leftarrow 1$  to  $m$  do
7:        $means[i] \leftarrow mean$  from node  $i$ 
8:        $sizes[i] \leftarrow$  LEN(points) from node  $i$ 
9:        $sum \leftarrow sum + means[i] * sizes[i]$ 
10:       $allSize \leftarrow allSize + sizes[i]$ 
11:    end for
12:     $allMean \leftarrow sum/allSize$ 
13:  end if
14:   $mean \leftarrow allMean$  from node 1
15:   $variance \leftarrow 0$ 
16:  for  $i \leftarrow 1$  to LEN(points) do
17:     $variance \leftarrow variance + DISTANCE(points[i], mean)$ 
18:  end for
19:   $allVariance \leftarrow$  sum of all variance from all nodes
20:  return allVariance
21: end function
```

---

**Methodology**

**Experiments**

**Analysis & Discussion**