Parallel and Multicore Computing Project 2

Shuang Li 1044137

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 θ 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, with the function that it calls shown in Algorithm 2 to 7. 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,

Algorithm 1

$\theta \leftarrow 0.1$
$DeltaTime \leftarrow 0.01$
$m \leftarrow$ number of compute nodes.
$NodeIndex \leftarrow$ the index for the compute node. ▷ This is equivalent to MPI_Comm_rank but 1 indexed.
$N \leftarrow$ number of points
$D \leftarrow$ number of dimensions
$c \leftarrow$ number of GMM components
for $i \leftarrow 1$ **to** c **do**
    $prob[i] \leftarrow$ the probability of the i^{th} GMM component
    $gmmc[i] \leftarrow$ the i^{th} GMM component
end for
for $i \leftarrow 1$ **to** N/m **do**
    $points[i] \leftarrow$ Sample($gmmc$[SampleWeightedDiscreteDistribution($prob$)])
    ▷ SampleWeightedDiscreteDistribution($weights$) generate a random number between 1 and the length of $weights$ with the number i having a probability of $weights[i]$/Sum($weights$)
end for
$centroids[1] \leftarrow$ ChooseRandomOne($points$) ▷ choose centroid for k-means++
for $i \leftarrow 2$ **to** m **do** ▷ choose m centroids for m clusters
    for $j \leftarrow 1$ **to** Len($points$) **do**
        $distances[i] \leftarrow$ distance of $points[i]$ to the closest centroid
    end for
    $DistSum \leftarrow$ sum of distances
    if $NodeIndex = 1$ **then**
        for $i \leftarrow 1$ **to** m **do**
            $AllSum[i] \leftarrow DistSum$ from node i
        end for
        $NextCentroidNode \leftarrow$ SampleWeightedDiscreteDistribution($AllSum$)
    end if
    $NextCentroidNode \leftarrow NextCentroidNode$ from node 1
    if $NodeIndex = NextCentroidNode$ **then**
        $NextCentroid \leftarrow points$[SampleWeightedDiscreteDistribution($distances$)]
    end if
    $centroids[i] \leftarrow NextCentroid$ from node $NextCentroidNode$
end for
$ClusterIndices \leftarrow$ KMeans($centroids, points$)
$points \leftarrow$ points in $NodeIndex^{th}$ cluster as indicated by $ClusterIndices$ in all nodes ▷ Significant implementation detail of message passing between nodes omitted.
for $i \leftarrow 1$ **to** Len($points$) **do**
    $velocities[i] \leftarrow 0$
end for
$InitVariance \leftarrow$ GetVariance($points$)
while $True$ **do**
    Simulate($points, velocities$)
    $CurrentVariance \leftarrow$ GetVariance($points$)
    if $Currentvariance < InitVariance/2$ **then**
        break while loop
    end if
    $ClusterIndices \leftarrow$ KMeans($centroids, points$) ▷ one iteration only, implemented separately in real code but reusing pseudocode here.
    $points \leftarrow$ points in $NodeIndex^{th}$ cluster as indicated by $ClusterIndices$ in all nodes
end while

Algorithm 2

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

Algorithm 3

```
1: function GetVariance(points) ▷ should really be named: get sum of distance to overall center of
   mass
2:     mean ← average of all points
3:     if NodeIndex = 1 then
4:         sum ← 0
5:         allSize ← 0
6:         for i ← 1 to m do
7:             means[i] ← mean from node i
8:             sizes[i] ← Len(points) from node i
9:             sum ← sum + means[i] × sizes[i]
10:            allSize ← allSize + sizes[i]
11:        end for
12:        allMean ← sum/allSize
13:    end if
14:    mean ← allMean from node 1
15:    variance ← 0
16:    for i ← 1 to Len(points) do
17:        variance ← variance + Distance(points[i], mean)
18:    end for
19:    allVariance ← sum of all variance from all nodes
20:    return allVariance
21: end function
```

Algorithm 4

```
1: function SIMULATE(points, velocities)
2:     b1 ← points[1]
3:     b2 ← points[1]
4:     for i ← 2 to LEN(points) do
5:         for each dimension do
6:             b1[dimension] ← MIN(b1[dimension], points[i][dimension])
7:             b2[dimension] ← MAX(b2[dimension], points[i][dimension])
8:         end for
9:     end for
10:    MaxDiff ← 0
11:    for each dimension do MaxDiff ← MAX(MaxDiff, b2[dimension] − b1[dimension])
12:    end for
13:    for each dimension do
14:        CurDiff ← b2[dimension] − b1[dimension]
15:        Expand ← (MaxDiff − CurDiff)/2
16:        b2[dimension] ← b2[dimension] + Expand
17:        b1[dimension] ← b1[dimension] − Expand
18:    end for                    ▷ Now b1 and b2 encapsulate a hypercube where all points are in it.
19:    BHTree ← BARNESHUTTREE(points, b1, b2)
20:    for i ← 1 to m do
21:        means[i] ← centerOfMass from BarnesHutTree of node i
22:        ClosestPoints[i] ← points with the shortest distance to means[i]
23:    end for
24:    for i ← 1 to m do
25:        hyperplanes[i] ← hyperplane that go through ClosestPoints[i] which is perpendicular to the
       line through ClosestPoints[i] and centerOfMass of local cluster
26:    end for
27:    for i ← 1 to m do
28:        PartialTrees[i] ← GETPARTIALTREE(hyperplane[i], BHTree)
29:    end for
30:    for i ← 1 to m do
31:        if i ≠ NodeIndex then
32:            BHTrees[i] PartialTrees[NodeIndex] from node i
33:        end if
34:    end for
35:    BHTrees[NodeIndex] ← BHTree
36:    for i ← 1 to LEN(points) do
37:        acceleration ← 0
38:        for j ← 1 to LEN(BHTrees) do
39:            acceleration ← acceleration + GETACCELERATION(points[i], BHTrees[j])
40:        end for
41:        velocities[i] ← velocities[i] + acceleration × DeltaTime
42:        points[i] ← points[i] + velocities[i] × DeltaTime
43:    end for
44: end function
```

Algorithm 5

function BarnesHutTree(points, b1, b2)
  if All *points* are identical **then**
    NumChildren($BHTreeNode$) $\leftarrow 0$
    Mass($BHTreeNode$) $\leftarrow$ Len($points$)
    CenterOfMass($BHTreeNode$) $\leftarrow points[1]$
  else
    Children($BHTreeNode$) $\leftarrow$ empty array
    for $nb1$, $nb2 \leftarrow$ every hyperoctant of the hypercube $b1$, $b2$ represent **do**
      $npoints \leftarrow$ empty array
      for $point \leftarrow points$ **do**
        if $point$ in hypercube represented by $nb1$, $nb2$ **then** PushBack($npoints, point$)
        end if
      end for
      if Len($npoints$) > 0 **then**
        PushBack(Children($BHTreeNode$), BarnesHutTree($npoints, nb1, nb2$))
      end if
      NumChildren($BHTreeNode$) $\leftarrow$ Len(Children($BHTreeNode$))
      CenterOfMass($BHTreeNode$) $\leftarrow 0$
      Mass($BHTreeNode$) $\leftarrow 0$
      for $child \leftarrow$ each Children($BHTreeNode$) **do**
        $ChildMass \leftarrow$ CenterOfMass($child$) $\times$ Mass($child$)
        CenterOfMass($BHTreeNode$) $\leftarrow$ CenterOfMass($BHTreeNode$) $+ ChildMass$
        Mass($BHTreeNode$) $\leftarrow$ Mass($BHTreeNode$) $+$ Mass($child$)
      end for
      CenterOfMass($BHTreeNode$) $\leftarrow \frac{\text{CenterOfMass}(BHTreeNode)}{\text{Mass}(BHTreeNode)}$
    end for
  end if
  B1($BHTreeNode$) $\leftarrow b1$
  B2($BHTreeNode$) $\leftarrow b2$
  return $BHTreeNode$
end function

Algorithm 6

function GetPartialTree(hyperplane, BHTreeNode)
  CenterOfMass($PartialNode$) $\leftarrow$ CenterOfMass($BHTreeNode$)
  Mass($PartialNode$) $\leftarrow$ Mass($BHTreeNode$)
  B1($PartialNode$) $\leftarrow$ B1($BHTreeNode$)
  B2($PartialNode$) $\leftarrow$ B2($BHTreeNode$)
  if $\frac{\text{B1}(BHTreeNode) - \text{B2}(BHTreeNode)}{\text{Distance}(hyperplane, \text{CenterOfMass}(BHTreeNode))} < \theta$ **then**
    NumChildren($PartialNode$) $\leftarrow 0$
  else
    NumChildren($PartialNode$) $\leftarrow$ NumChildren($BHTreeNode$)
    Children($PartialNode$) $\leftarrow$ empty array
    for $child \leftarrow$ each Children($BHTreeNode$) **do**
      PushBack(Children($PartialNode$), GetPartialTree($hyperplane, child$))
    end for
  end if
  return $PartialNode$
end function

Algorithm 7

function GetAcceleration(point, BHTreeNode)
  $acceleration \leftarrow 0$
  if $\frac{\text{B1}(BHTreeNode)-\text{B2}(BHTreeNode)}{\text{Distance}(point,\text{CenterOfMass}(BHTreeNode))} < \theta \vee \text{NumChildren}(BHTreeNode) = 0$ **then**
    $com \leftarrow \text{CenterOfMass}(BHTreeNode)$
    $acceleration \leftarrow \text{AccelerationFromCOM}(point, com, \text{Mass}(BHTreeNode))$
  else
    for $child \leftarrow$ every $\text{Children}(BHTreeNode)$ **do**
      $acceleration \leftarrow acceleration + \text{GetAcceleration}(point, child)$
    end for
  end if
  return *acceleration*
end function

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.

Methodology

To measure the performance of the implementation and whether parallelization sped up the simulation or not and by how much, the program was run on spartan with 1, 2, 4, 8, 16, 32 and 64 nodes. Because the implementation doesn't utilize OpenMP for inner process / inner node parallelization, each node only have one core allocated to run the simulation. Furthermore, to investigate how the communication overhead between the different nodes effect the overall runtime, the program is also run on a single spartan compute node with 1, 2, 4, 8, 16, 32 and 64 threads. This should significantly reduce the communication overhead between the Open MPI processes and give us more insight to how the extra communication overhead between different spartan compute nodes effect the performance of the program.

To allow more fine-grained observation of the program's performance, the runtime measurement is done on each process for the following stages of the program:

- Generating the points (by sampling from the GMM components).
- Choosing centers as needed by the k-means++ algorithm.
- Collaborating to cluster points using k-means algorithm after the centers are chosen.
- Sending points across the nodes so that all points of each cluster are on a single node.
- The N-Body simulation.

Originally, the N-Body simulation would stop when the sum of square of the distance between every point and the overall center of mass fall below 1/4 of the initial value. However, during the experiments, it is observed that sometimes the sum of square distance never goes below 1/4 of the initial value. It is believed that there are several reasons of this happening.

First, when two points get too close together, they experience strong gravitation attraction towards each other and hence receive large value of acceleration that increases their velocities to a very high value. Because the simulation's time is discrete, by the next time step, the points have already flew past each other and the gravitational attraction between them are no longer strong enough to slow them down. These points fly off away from all the other points in high velocity and the square of their distance to the overall center of mass dominate the sum of square distance. This problem has been fixed in two ways. First, by not having the point exert any gravitational forces upon each other when they're less than 0.05 distance away. This is a physically justifiable approximation because during the very short time period

when two points fly through each other in proximity, the total acceleration through that short period of time add up to be zero, as they act in opposite direction for equal amount of time. Second, by using sum of distance and the threshold being 1/2 of the initial value instead of sum of square distance. This reduces the influence of points very far away from the overall center of mass have on the sum.

Second, if the points are too close to each other at the start, they quickly fly past each other and the sum of distances increase after a short decrease. This is because our model has no mechanism for energy lost. As a result, after the points get close enough to each other, a significant portion of the points will gain enough energy from other points and get ejected from the center and fly away forever. This is solved by having the points further away from each other at the start so that the sum of distance will dip below 1/2 of the initial value before any points get ejected.

The parameter that is chosen for the experiments are:

- 10000 points.
- 4 dimensions.
- 4 Gaussian Mixture model components with probability of 0.25 each and standard deviation of 10 with (10, 10, 10, 10), (10, 0, 10, 0), (0, 0, 0, 0) and (0, 10, 0, 10) as means.
- θ of 0.1.
- $DeltaTime$ of 0.01

Experiments

One Node Per Process

All figures and table shown in this subsection are from running the implementation with each Open MPI process on a separate node.

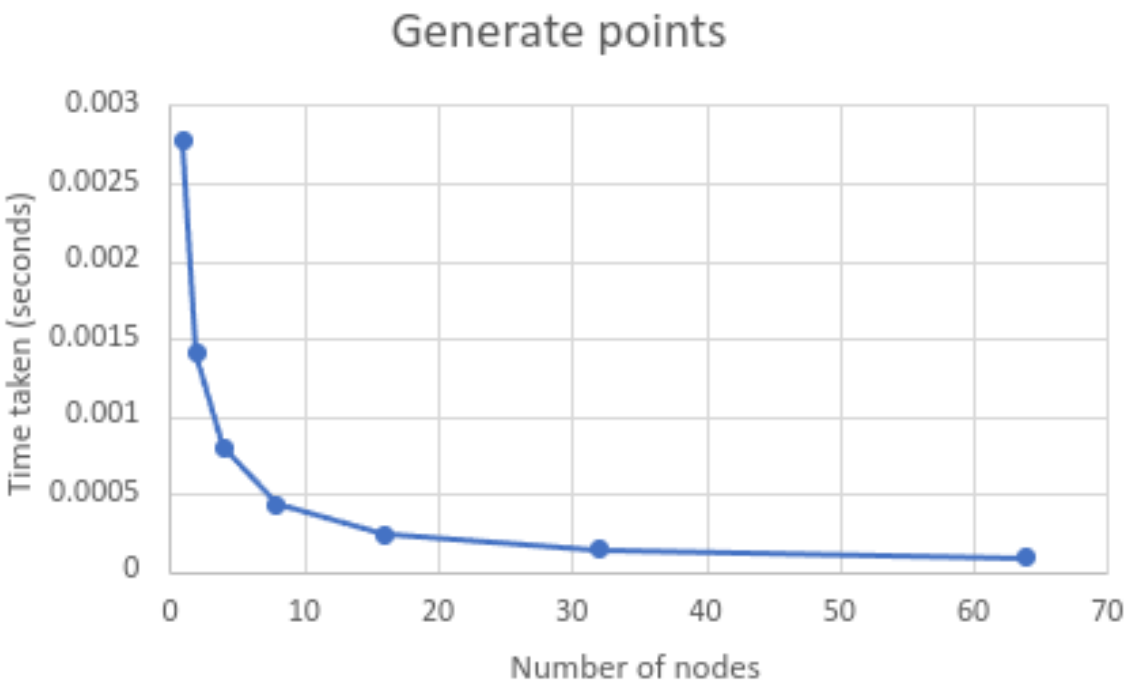

Figure 1: Generating the points

Node count	1	2	4	8	16	32	64
Time taken (seconds)	0.002777	0.0014	0.000798	0.000427	0.000235	0.000142	9.16E-05

Table 1: Generating the points

Figure 1 and table 1 above shows the amount of time taken by each node on average to generate a portion of the 10000 points by sampling them from the GMM.

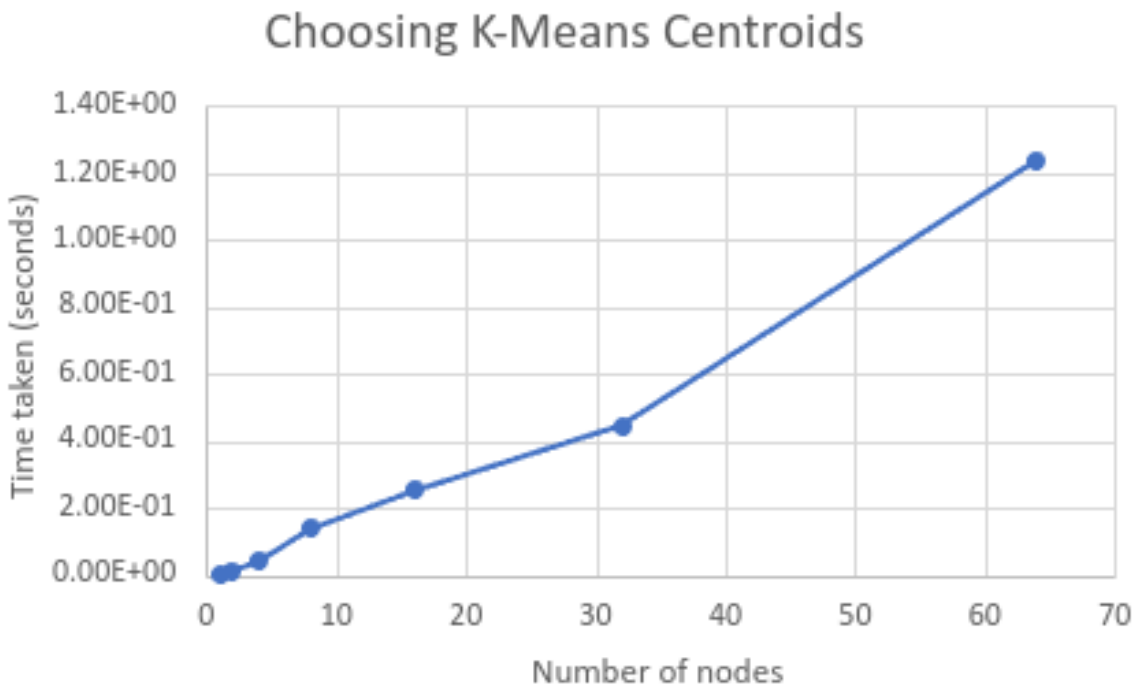

Figure 2: Choosing Centroids for k-means++

Node count	1	2	4	8	16	32	64
Time taken (seconds)	8.11E-06	0.013433	0.042459	0.140171	2.56E-01	0.447505	1.238592

Table 2: Choosing Centroids for k-means++

Figure 2 and table 2 above shows the amount of time taken by each node on average to choose the centroids of the clusters with k-means++.

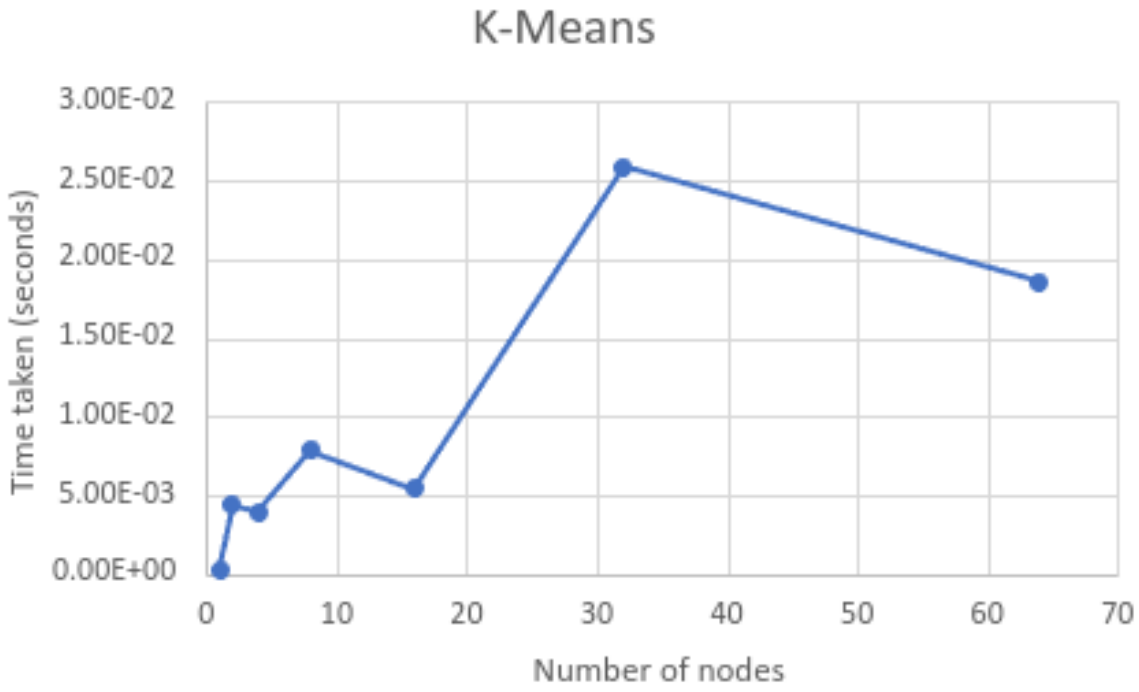

Figure 3: K-Means

Node count	1	2	4	8	16	32	64
Time taken (seconds)	2.29E-04	0.004344	0.003917	0.007881	0.005441	0.025862	0.018548

Table 3: K-Means

Figure 3 and table 3 above shows the amount of time taken by each node on average to cluster all the points using k-means++.

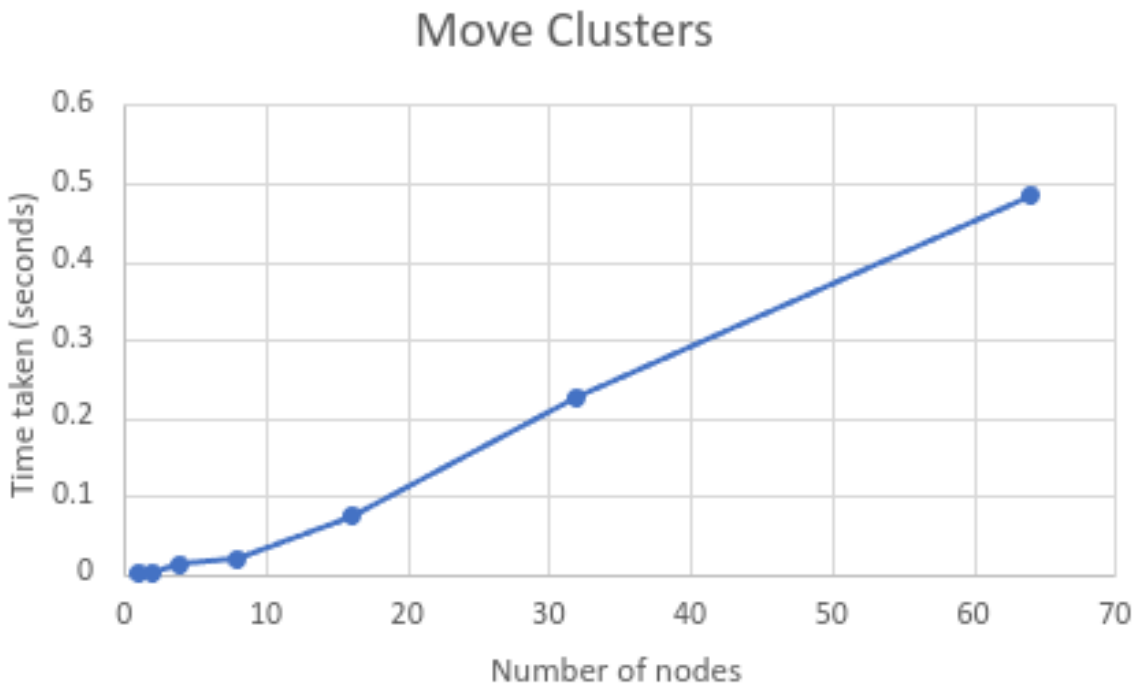

Figure 4: Transfer clusters

Node count	1	2	4	8	16	32	64
Time taken (seconds)	0.000354	0.002373	0.012425	0.020189	0.073544	0.225381	0.482996

Table 4: Transfer clusters

Figure 4 and table 4 above shows the amount of time taken by each node on average to transfer the points so that all points of a cluster are stored by a single node.

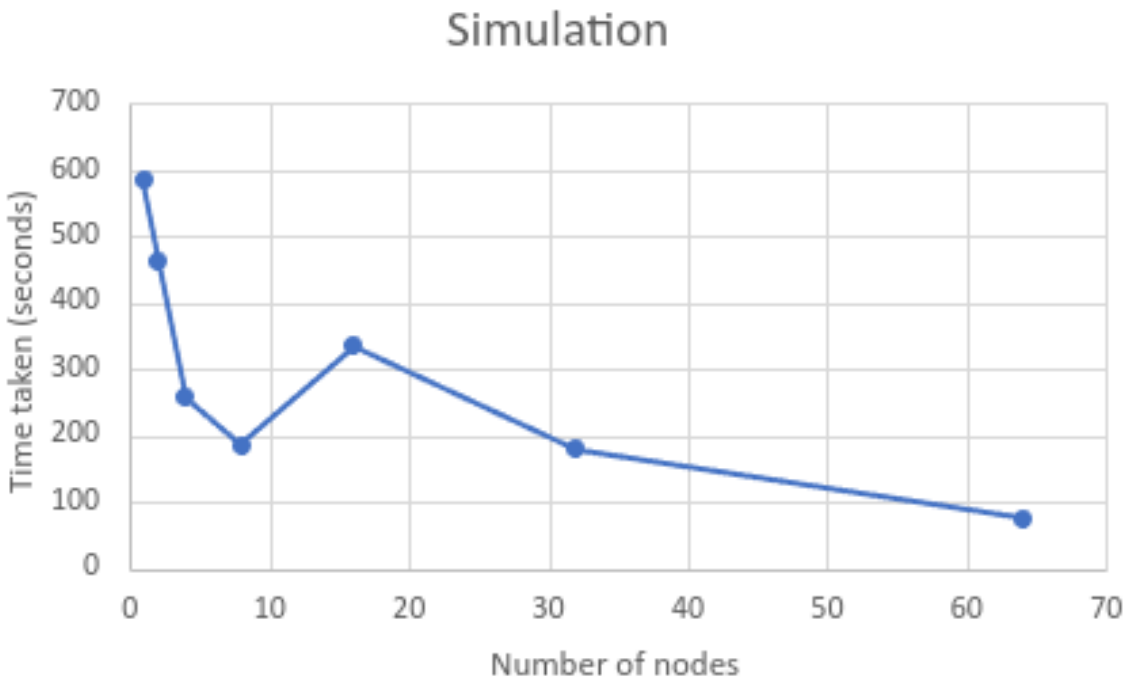

Figure 5: Simulation

Node count	1	2	4	8	16	32	64
Time taken in seconds	584.358	462.481	256.905	185.331	335.487	180.68	75.3026

Table 5: Simulation

Figure 5 and table 5 above shows the amount of time taken by each node on average to perform the N-Body simulation until the sum of distance goes below 1/2 of the initial value. Note that during all runs, the program ran 44 iterations of the simulation before it stopped.

All Process on One Node

All figures and table shown in this subsection are from running the implementation with all Open MPI process on a single node but each of them have its own core.

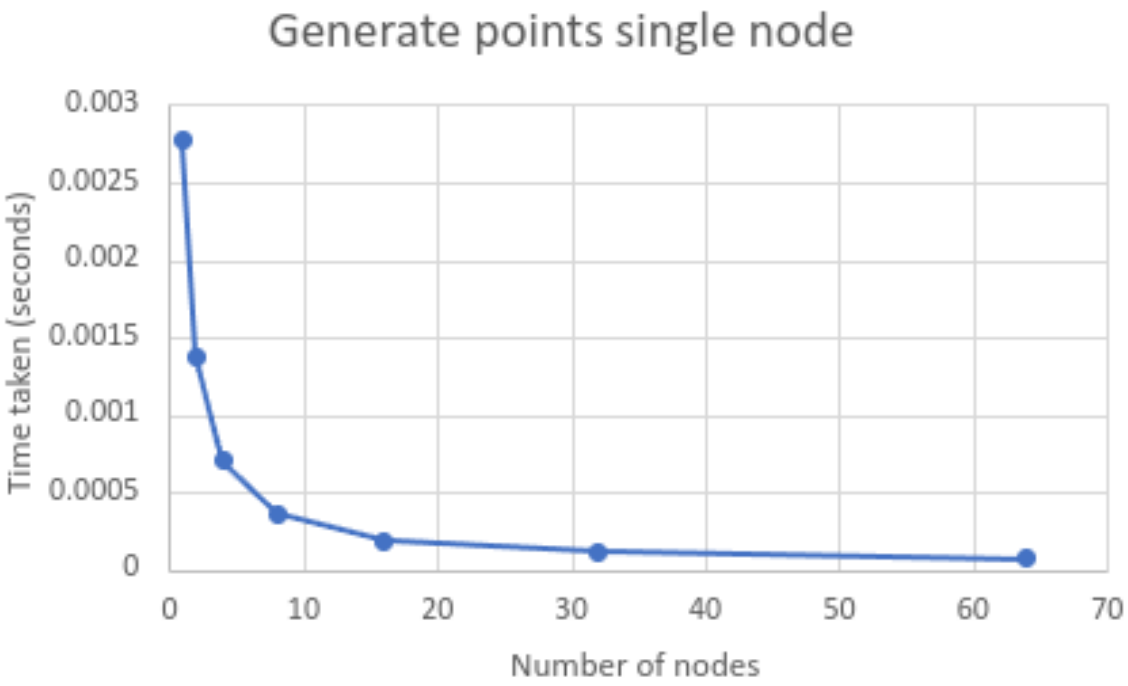

Figure 6: Generating the points

Node count	1	2	4	8	16	32	64
Time taken (seconds)	0.002777	0.001379	0.000698	0.00036	0.000186	0.000117	7.20E-05

Table 6: Generating the points

Figure 6 and table 6 above shows the amount of time taken by each process on average to generate a portion of the 10000 points by sampling them from the GMM.

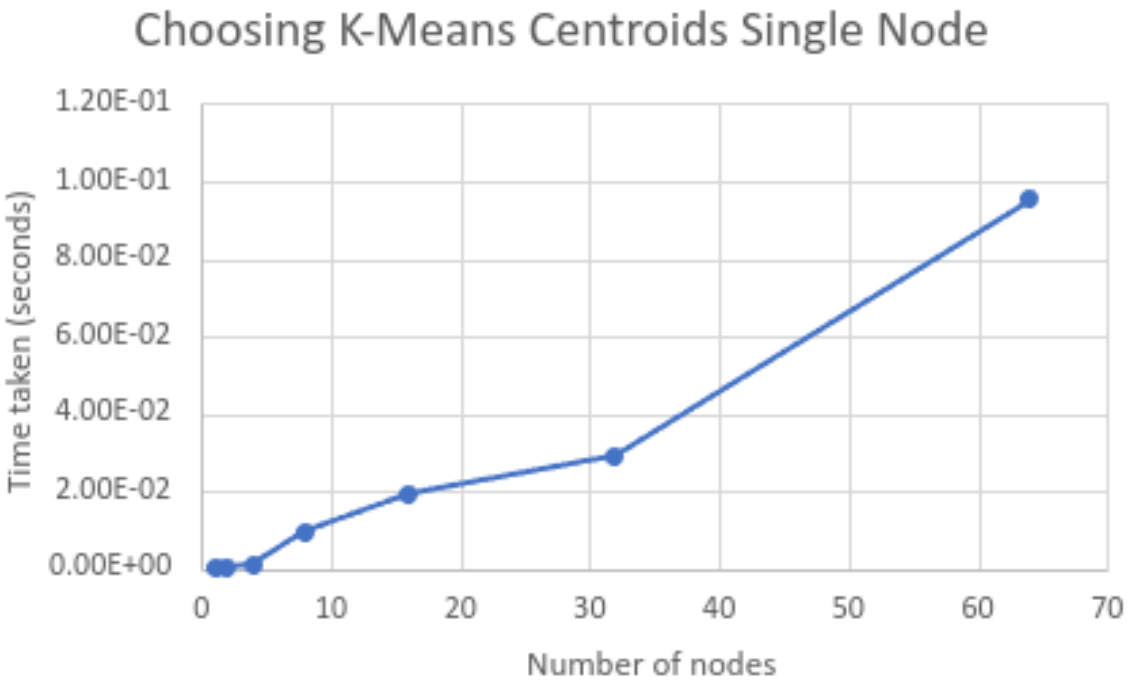

Figure 7: Choosing Centroids for k-means++

Node count	1	2	4	8	16	32	64
Time taken (seconds)	8.11E-06	0.000467	0.001051	0.009515	0.019165	0.029197	0.095282

Table 7: Choosing Centroids for k-means++

Figure 7 and table 7 above shows the amount of time taken by each process on average to choose the centroids of the clusters with k-means++.

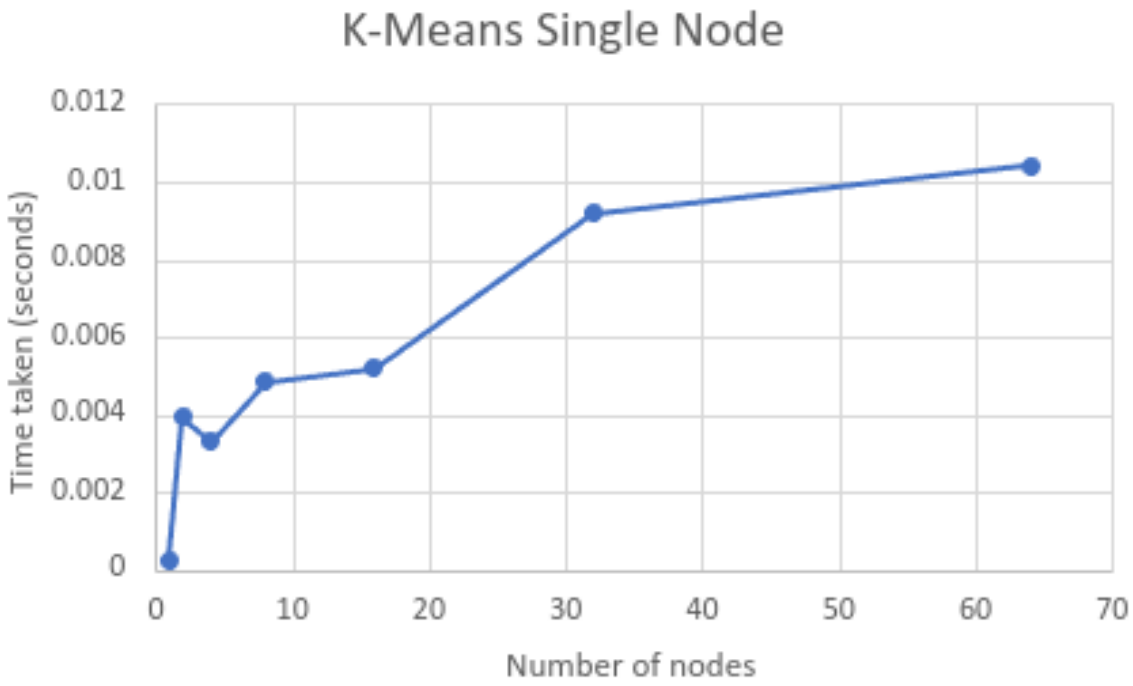

Figure 8: K-Means

Node count	1	2	4	8	16	32	64
Time taken (seconds)	0.000229	0.003932	0.003291	0.004835	0.005189	0.009189	0.010402

Table 8: K-Means

Figure 8 and table 8 above shows the amount of time taken by each process on average to cluster all the points using k-means++.

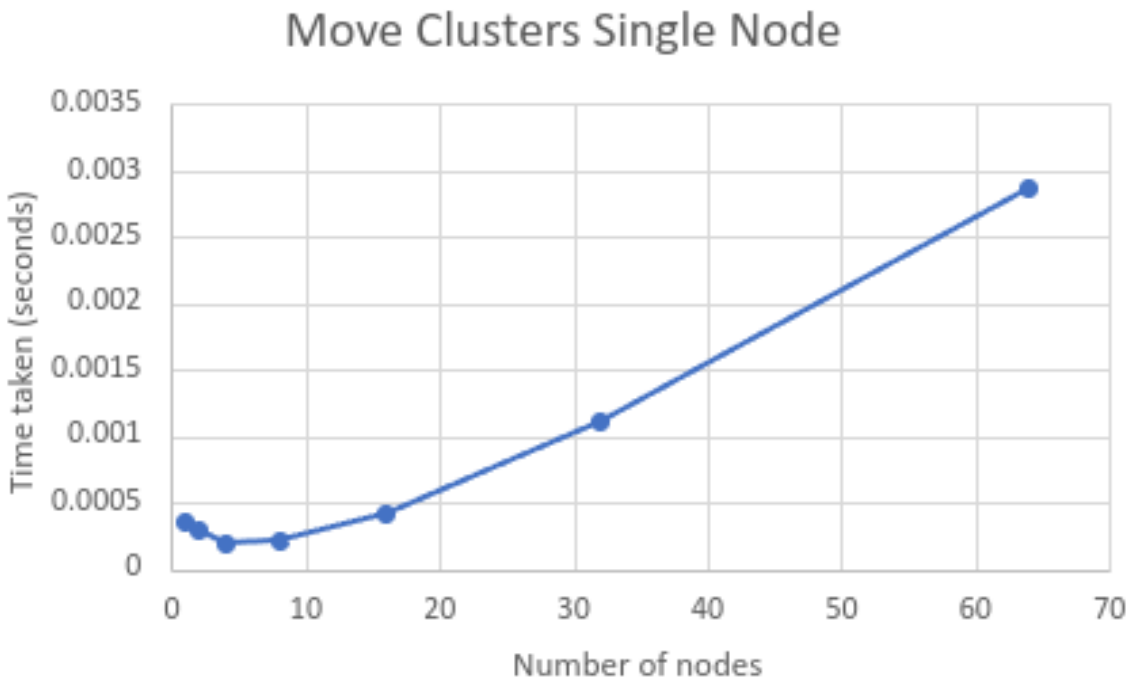

Figure 9: Transfer clusters

Node count	1	2	4	8	16	32	64
Time taken (seconds)	0.000354	0.000301	0.000192	0.000214	0.000424	0.001117	0.002872

Table 9: Transfer clusters

Figure 9 and table 9 above shows the amount of time taken by each process on average to transfer the points so that all points of a cluster are stored by a single process.

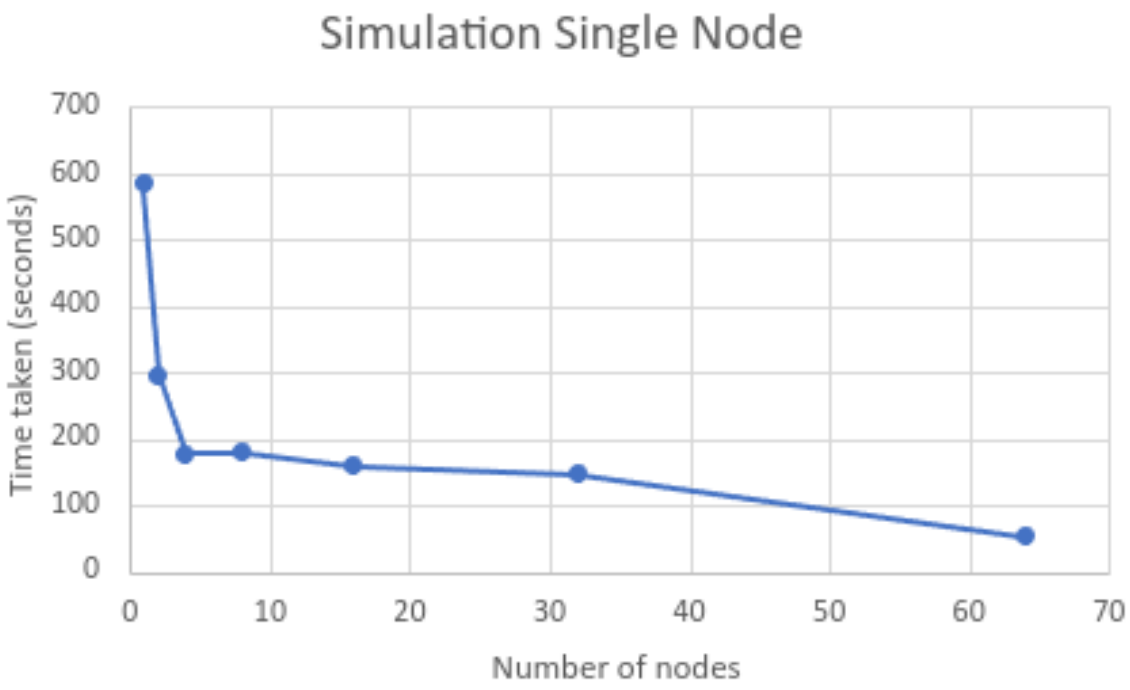

Figure 10: Simulation

Node count	1	2	4	8	16	32	64
Time taken (seconds)	584.358	296.865	177.679	178.9486	159.752	146.2969	53.6977

Table 10: Simulation

Figure 10 and table 10 above shows the amount of time taken by each process on average to perform the N-Body simulation until the sum of distance goes below 1/2 of the initial value. Note that during all runs, the program ran 44 iterations of the simulation before it stopped.

Analysis & Discussion

As we can see in figure 1, table 1, figure 6 and table 6, the amount of time it takes for all the points to be sampled from the GMM are roughly inversely proportional to the number of processes. This is because the task of generating all points is an embarrassingly parallel problem in the sense that there are no dependency or need for communication between the tasks of sampling points from GMM components. There are also no differences between the time taken when all processes are running on a single node and when all processes are running on separate nodes. This is because there are no communication between them hence the effect of changing communication speed does not affect their runtime.

As we can see in figure 2, table 2, figure 7 and table 7, the amount of time it takes to choose all centroids for k-means++ increase proportionally with the amount of process. Because the number of centroids that needs to be generated are proportional (in fact identical) with the amount of processes, the total amount of time needed to generate all centroids are proportional to the square of the number of processes. This is expected because in the implementation, before ever new center is chosen, distances from every point to every center is recalculated, leading to a time complexity of $O(n^2)$ for choosing centroids where n is the number of clusters. The implementation could be sped up by only recalculating the distance between every point and the newly computed centers.

Comparing between the runtime of choosing centroids for one process per node in figure 2 and table 2

and on all process on one node in figure 7 and table 7, we can see that having all process run on one node reduces the runtime by roughly 10-fold. This shows that the communication time dominates when all processes are running on separate node. This means the implementation of choosing centroids might be sped up significantly by doing all the computation on a single node.

As we can see in figure 3, table 3, figure 8 and table 8, the time taken to cluster the points via k-means does not show clear relationships with the number of nodes. One of the reasons of this could be that the amount of iteration needed for k-means to converse fluctuate as the number of cluster changes. While this explains the fluctuation of the run time for both of each process on a separate node and all processes on a single node separately. It does not explain the differences between the shape of figure 3 and figure 8. More specifically, it is unclear why 8 clusters takes longer than 16 clusters on one process per node but slightly shorter than 16 clusters on all process on one node. The only possible explanation for that is noise. Regardless, in both one process per node and all process on one node scenarios, the runtime for number of clusters being 2, 8 and 16 are all slightly higher than it would be if the runtime is linearly proportional to the number of clusters.

Comparing between the runtime of k-means clustering on one process per node in figure 3 and table 3 and on all process on one node in figure 8 and table 8, we can see that they are always within two factors of each other. This shows that the runtime are relatively not as dominated by communication overhead as the runtime for choosing centroids.

As we can see in figure 4, table 4, figure 9 and table 9, the amount of time taken to move all points to their respective clusters are roughly proportional to number of clusters. Theoretically, we'd expect the number of points that needs to be transferred to another process being $10000 \times \frac{n-1}{n}$ where n is the number of clusters. This means that the amount of data needed to be sent from one process to another should be similar between 32 clusters and 64 clusters. However, in reality with 64 processes it takes almost double amount of time to transfer the points. This means that the amount of time taken to transfer the points is dominated by the overhead for each message that needs to be sent and the size of the messages doesn't affect the runtime very much. This also mean that the benefit in reduced communication cost that comes from computing the partial tree might be insignificant as computing the partial tree only reduces the size of the messages needed to be sent to other processes during the simulation and does not reduce the number of messages.

Comparing between the runtime of k-means clustering on one process per node in figure 4 and table 4 and on all process on one node in figure 9 and table 9, we can see that it takes roughly 200 times longer when each process is on a separate node. This shows that the runtime is dominated by communication cost. It also shows that the communication cost is at least 200 times higher when the processes are on different nodes.

As we can see in figure 5, table 5, figure 10 and table 10, ignoring the jump in runtime for when each process is on a separate node between 8 processes and 16 processes, the runtime of the N-Body simulation is roughly inversely proportional to the number of processes. This shows that our parallelization technique is successful in reducing the overall runtime. It also shows a practically near optimal parallelization.

The reason for the jump in runtime for when each process is on a separate node between 8 processes and 16 processes could be because spartan included some nodes with high communication cost with the rest of the node in jobs that require 16 or more nodes. If we look at the runtime for 1 to 8 nodes and the runtime for 16 to 64 nodes separately, they're still mostly inversely proportional to the number of processes.

We can see a similar jump in runtime for when all processes are on the same node, although it is much less significant. This could be because that the processes have to be on different sockets for higher number of processes (a spartan node is made up of 4 18 cores sockets) hence increasing the communication time between them.

Comparing between the runtime for N-Body simulation on one process per node in figure 5 and table 5 and on all processes on the same node in figure 10 and table 10, we can see that they're within a factor of 2 of each other. This shows that the runtime of the N-Body simulation is mostly made up of computing cost instead of communication cost.

If we increase the number of processes to a much larger number, the runtime of the N-Body simulation should continue to decrease in an inversely proportional manner as the runtime is mostly dominated by computing cost instead of communication cost.