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. 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

function KMEANS(centroids, points)
  while centroids changed since last iteration **do**
    for $i \leftarrow 1$ **to** LEN($points$) **do**
      $ClusterIndices[i] \leftarrow$ index of centroid closest to $points$
    end for
    for $i \leftarrow 1$ **to** LEN($points$) **do**
      $sums[ClusterIndices[i]] \leftarrow sums[ClusterIndices[i]] + points[i]$
      $counts[ClusterIndices[i]] \leftarrow counts[ClusterIndices[i]] + 1$
    end for
    if NodeIndex == 1 **then**
      for $i \leftarrow 1$ **to** LEN($centroids$) **do**
        $allSums \leftarrow$ sum of $sums[i]$ from all nodes
        $allCounts \leftarrow$ sum of $counts[i]$ from all nodes
        $centroids[i] \leftarrow allSums/allCounts$ ▷ Here it is actually each dimension of $allSums$ divided by $allCounts$.
      end for
    end if
    $centroids \leftarrow centroids$ from node 1
  end while
  return $ClusterIndices$
end function

Algorithm 3

function GETVARIANCE(points)
  $mean \leftarrow$ average of all points
  if $NodeIndex = 1$ **then**
    $sum \leftarrow 0$
    $allSize \leftarrow 0$
    for $i \leftarrow 1$ **to** m **do**
      $means[i] \leftarrow mean$ from node i
      $sizes[i] \leftarrow$ LEN($points$) from node i
      $sum \leftarrow sum + means[i] * sizes[i]$
      $allSize \leftarrow allSize + sizes[i]$
    end for
    $allMean \leftarrow sum/allSize$
  end if
  $mean \leftarrow allMean$ from node 1
  $variance \leftarrow 0$
  for $i \leftarrow 1$ **to** LEN($points$) **do**
    $variance \leftarrow variance +$ DISTANCE($points[i], mean$)
  end for
  $allVariance \leftarrow$ sum of all $variance$ from all nodes
  return $allVariance$
end function

Algorithm 4

```
function SIMULATE(points, velocities)
    b1 ← points[1]
    b2 ← points[1]
    for i ← 2 to LEN(points) do
        for each dimension do
            b1[dimension] ← MIN(b1[dimension], points[i][dimension])
            b2[dimension] ← MAX(b2[dimension], points[i][dimension])
        end for
    end for
    MaxDiff ← 0
    for each dimension do MaxDiff ← MAX(MaxDiff, b2[dimension] − b1[dimension])
    end for
    for each dimension do
        CurDiff ← b2[dimension] − b1[dimension]
        Expand ← (MaxDiff − CurDiff)/2
        b2[dimension] ← b2[dimension] + Expand
        b1[dimension] ← b1[dimension] − Expand
    end for                    ▷ Now b1 and b2 encapsulate a hypercube where all points are in it.
    BHTree ← BARNESHUTTREE(points, b1, b2)
    for i ← 1 to m do
        means[i] ← centerOfMass from BarnesHutTree of node i
        ClosestPoints[i] ← points with the shortest distance to means[i]
    end for
    for i ← 1 to m do
        hyperplanes[i] ← hyperplane that go through ClosestPoints[i] which is perpendicular to the
line through ClosestPoints[i] and centerOfMass of local cluster
    end for
    for i ← 1 to m do
        PartialTrees[i] ← GETPARTIALTREE(hyperplane[i], BHTree)
    end for
    for i ← 1 to m do
        if i ≠ NodeIndex then
            BHTrees[i] PartialTrees[NodeIndex] from node i
        end if
    end for
    BHTrees[NodeIndex] ← BHTree
    for i ← 1 to LEN(points) do
        acceleration ← 0
        for j ← 1 to LEN(BHTrees) do
            acceleration ← acceleration + GETACCELERATION(points[i], BHTrees[j])
        end for
        velocities[i] ← velocities[i] + acceleration ∗ DeltaTime
        points[i] ← points[i] + velocities[i] ∗ DeltaTime
    end for
end function
```

Algorithm 5

```
function BarnesHutTree(points, b1, b2)
    if All points are identical then
        NumChildren(BHTreeNode) ← 0
        Mass(BHTreeNode) ← Len(points)
        CenterOfMass(BHTreeNode) ← points[1]
    else
        Children(BHTreeNode) ← empty array
        for nb1, nb2 ← every hyperoctant of the hypercube b1, b2 represent do
            npoints ← empty array
            for point ← 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) ← Len(Children(BHTreeNode))
            CenterOfMass(BHTreeNode) ← 0
            Mass(BHTreeNode) ← 0
            for child ← each Children(BHTreeNode) do
                ChildMass ← CenterOfMass(child) ∗ Mass(child)
                CenterOfMass(BHTreeNode) ← CenterOfMass(BHTreeNode) + ChildMass
                Mass(BHTreeNode) ← Mass(BHTreeNode) + Mass(child)
            end for
            CenterOfMass(BHTreeNode) ← CenterOfMass(BHTreeNode) / Mass(BHTreeNode)
        end for
        return BHTreeNode
    end if
end function
```

Algorithm 6

```
function GetPartialTree(hyperplane, BHTree)
end function
```

Algorithm 7

```
function GetAcceleration(point, BHTreeNode)
    acceleration ← 0
    if (B1(BHTreeNode) − B2(BHTreeNode)) / Distance(point, CenterOfMass(BHTreeNode)) < θ ∨ NumChildren(BHTreeNode) = 0 then
        com ← CenterOfMass(BHTreeNode)
        acceleration ← AccelerationFromCOM(point, com, Mass(BHTreeNode))
    else
        for child ← every Children(BHTreeNode) do
            acceleration ← acceleration + 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

Experiments

Analysis & Discussion