Hierarchical Agglomerative Clustering using MPI

Krishna Narayan
CPSC 445

December 14, 2021

Introduction

For my final project in High Performance Computing (CPSC 445), I implemented hierachical agglomerative clustering using MPI. Hierarchical agglomerative clustering is a method of building a hierarchy of clusters from a set of points. Initially, all individual points are treated as cluster. Then, the closest clusters are formed together into a new cluster. At this point, this operation repeats the next closest clusters are formed together.

If two clusters only contain one point, finding the distance is trivial in that you only need to find the distance between the two points. However, in the case of finding the distance between two clusters each with multiple points, or finding the distance between one cluster with multiple points and a single cluster, one must decided on a point for a cluster that they can use in distance calculations. For example, between two multi-point clusters, one might calculate the distance as the distance between the average of the points in each cluster. One could also decide to calculate the distance between these clusters as the distance between the two closest points between the two clusters.

Applications

Hierarchical agglomerative clustering allows for cluster analysis. By clustering points based off of certain traits, we can hope to discover more about similarities within a group. Cluster analysis has revelant applications in computer graphics, machine learning, pattern recognition, etc. An example might be for systems that recommends things to users. A clustering algorithm could compare a user's cluster of preferences to another user's cluster of preferences, and then making decisions off this cluster comparison.

Serial Implementation

The serial version of this program first reads a CSV file listing floating point coordinates. The x coordinates are read into an x vector, and the y coordinates are read into a y vector. With these points, the program computes a distance matrix (only the bottom corner is computed, as distance matrices are mirrored. Then, the program searches for the minimum distance in the distance matrix. The x and y coordinate of the minimum distance provides the point number of two closest clusters. The program stores the cluster numbers. Then, it computes the average point between the two clusters (the chosen distance matrix being average), and stores these average point coordinates to the coordinates of the first cluster. Then, the coordinates of the second cluster are erased from the point list. So now, we have vectors of coordinates, except that they form one less point – but one of these coordinates is the average of the previously clustered two points. The program then takes some new metrics (finding the new point count) and repeats this process until a desired number of clusters is found.

Parallel Implementation

The parallel version of this program works very similarly. The program is parallelized over multiple processes using MPI. When the program starts, rank 0 reads the x and y coordinates from the CSV, and establishes the size of the cluster vector. Rank 0 also initializes the cluster vector so that every point provided through the input exists as an individual cluster (the size of the cluster is the number of points).

At this point, the program enters the clustering loop. Rank 0 broadcasts the number of points to all other processes, and then broadcasts the points. Each process then calculates distances on rows assigned by a partition function. The processes do not calculate any distance outside of the bottom corner (because these distances are mirrored). Then, each process calculates a local minimum, which acts as a candidate for rank 0 to chose from. The local minimums (along with the clusters which from this local minimum distance) of each process are gathered together, and rank 0 finds the global minimum. Then, rank 0 averages the coordinates of the clusters involved, and rank 0 erases the cluster candidate that was not uses to hold the coordinates of the newly formed cluster. The cluster vector is to represent the new clustering, and the clusters are visualized. All vectors are cleared except for the coordinate vectors of rank 0, and the program goes back to the top of the loop.

Limitations

There are quite a few limitations with parallelization that I hope to fix in the future.

First, the distance matrix is partitioned so that as rank increases, the range of rows which they will operate on is increased. This forms somewhat of a bottleneck, as this means the last process will be assigned the longest rows of numbers (as the row is increased, the number of elements to process is increased given a distance matrix). That being said, last processes are often assigned less rows to iterate through, so this makes up for some of the bottleneck. Still, I'd like to partition the rows such that if each process were to do two rows, for example, the number of each row was spread apart. A process that is assigned the first and last row would only iterate through 1 element in the first row, and the total point count - 1 in the last row. Another process might iterate through 2 elements in the second row, and the total point count - 2 in the last row. This way, each process collectively processes the same amount of elements, while taking advantage of the fact that only the bottom corner (or top corner) of the distance matrix needs to be processed.

Another limitation I have is that rank 0 handles moving points from cluster to cluster at the end of each cycle. This means that rank 0 eventually iterate through every single point - 1 to move a cluster with almost ever point in it to a cluster with only 1 point in it. I should find a way to parallelize this.

Another limitation is that the program always recalculates the entire distance matrix. I'm sure there is a way to only recalculate certain distances that correspond to changed points.

Conclusions

Overall, the program successfully clusters points using unweighted averages as positions for clusters of multiple points. It is also able to show step by step how clustering is occurring, and provides the tools for visualizing iterations of distance matrices among multiple processes.

While I'm happy with this progress, I'd like to output clustering data so that I can use it to generate a dendrogram. Better visualization is key to understanding how clustering is occurring, and I think a library like SciPy might be able to help me with this. However, before I work on even better visualization, I'd like to fix the partition issue, and find better ways to parallelize the program. I also know that I could clean up functions like those compute distance matrices and checking for minimums. There's still a lot of work to be done on this project.