## Description of Approach Used for the Implementations

**Serial:** Follows the tutorial provided in C++. We then converted it to C code and parsed out various functions for easier reuse in other implementations.

**Parallel Shared Memory CPU:** Uses openMP to parallelize assigning each point to a centroid, collecting the data for calculating new centroids, and calculating the new centroids.  Used a sum reduction to prevent race conditions while collecting data for calculating the new centroids.

**Parallel Shared Memory GPU:** This made use of CUDA. We used two kernels. The first kernel calculates the distance from each point to a centroid and assigns the point to the closest centroid. In this function the distance is calculated in the kernel since non-kernel functions can't be called from the kernel. The second kernel summed data that is needed for calculating new centroids. To do this we needed to use tiling and a tree structured sum to sum across threads in a tile.

**Distributed Memory CPU:** For this we used MPI. We created a custom MPI_Datatype for our points. Then we used MPI_Scatterv to distribute part of the points array over the distributed CPUs. We then used MPI_Bcast to broadcast the centroid array each time the centroids were updated. We then used local sums and MPI_Reduce to get the data needed for the new centroids. Finally we used MPI_Gatherv to get the new points data back to rank 0 from the distributed system.

**Distributed Memory GPU:** A combination of the shared memory GPU and distributed memory CPU implementations. This uses MPI to have multiple ranks and then each rank sends their subsection of the points array to each GPU. For each iteration of the epoch, each GPU will compute the distances for their sub arrays. Then the centroids are computed on each GPU. The points array is transferred to the GPU at the beginning, and then back once at the end so that transferring back and forth is limited. This uses the same 2 kernel functions as the parallel shared GPU implementation.