Assignment Report

CSE355 (UG2018) - Parallel and Distributed Algorithms (27224)

—

Mohamed Mokhtar
18P7232

Overview

This code is an implementation of the K-Means clustering algorithm using OpenMP to parallelize the calculations. The K-Means algorithm is a method for clustering a set of data points into K clusters based on their attributes (e.g., x and y coordinates in this case).

The main function, **int kmeans(struct point *points, size_t point_count)**, takes as input an array of **point** structures and the number of points in the array. Each **point** structure contains the x and y coordinates of a data point. The function performs the K-Means algorithm on the input points, using the OpenMP library to parallelize the calculations.

The algorithm starts by picking K initial cluster points, which are stored in the **kx** and **ky** arrays. These cluster points are initially set to the coordinates of the first K points in the input array.

Then, the algorithm enters a parallel region, which is specified by the **#pragma omp parallel directive**. Inside this region, the algorithm performs **ITERATION_COUNT** iterations of the K-Means algorithm. In each iteration, the algorithm calculates which cluster each point belongs to, based on the distance between the point and each cluster point. The algorithm then updates the sum and count of the points in each cluster, using the **l_kx**, **l_ky**, and **l_kcount** arrays to store the partial results for each thread.

After all iterations have completed, the algorithm calculates the new coordinates of each cluster point by dividing the sum of the points in each cluster by the count of points in the cluster. Finally, the algorithm prints the final coordinates of the cluster points.

The code uses several OpenMP directives and clauses to control the parallelization. The **#pragma omp parallel** directive specifies that a parallel region should be created, and the **shared** clause specifies that the **kx**, **ky**, and **kcount** arrays should be shared among all threads in the team. The **#pragma omp for** directive specifies that the inner loop should be parallelized, and the **#pragma omp single** directive specifies that the code block following it should be executed by only one thread in the team.

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <float.h>
#include <omp.h>

#define CLUSTER_COUNT   2
#define ITERATION_COUNT 10000

#define PRINT_ITERATIONS 0

// Contains point data
struct point {
 double x, y;
};

// Prints usage message then exits
void usage(const char *err)
{
   printf("%s\nUsage: ./kmeans 1,3 4,5 -2,4.43 42.3,2\n", err);

   exit(0);
}

// Performs KMeans
int kmeans(struct point *points, size_t point_count)
{
   // Arrays to contain cluster points
   double kx[CLUSTER_COUNT] = {0};     // X Pos for each cluster
   double ky[CLUSTER_COUNT] = {0};     // Y Pos for each cluster
   int kcount[CLUSTER_COUNT] = {0};    // Number of points contained in each
cluster

   // Pick the cluster points according to the given points (at first)
   // NOTE: It would be better to make sure no points start at the same
location
   for(int i = 0; i < CLUSTER_COUNT; i++) {
       kx[i] = points[i].x;
       ky[i] = points[i].y;
   }

   // Print the point info (rank only)
```

```
    printf("\n");
    for (size_t i = 0; i < point_count; i++)
        printf("Point %ld: %lf,%lf\n", i+1, points[i].x, points[i].y);

    // Start parallel region
    #pragma omp parallel shared(kx) shared(ky) shared(kcount)
    {
        // For each iteration
        for (size_t i = 0; i < ITERATION_COUNT; i++)
        {
            // Print interations if requested
#if PRINT_ITERATIONS
            #pragma omp single
            {
                printf("\n");
                for (size_t i = 0; i < CLUSTER_COUNT; i++)
                    printf("%lf,%lf ", kx[i], ky[i]);
            }
#endif

            // Arrays to contain cluster points' partial values
            double l_kx[CLUSTER_COUNT] = {0};   // X Pos for each cluster
            double l_ky[CLUSTER_COUNT] = {0};   // Y Pos for each cluster
            int l_kcount[CLUSTER_COUNT] = {0};  // Number of points contained in
each cluster

            #pragma omp for
            for (size_t j = 0; j < point_count; j++)
            {
                // Check which cluster this point belongs to
                size_t min_index = 0;
                double min_dist = DBL_MAX;
                for (size_t k = 0; k < CLUSTER_COUNT; k++)
                {
                    double current_dist = ((kx[k] - points[j].x)*(kx[k] -
points[j].x) + (ky[k] - points[j].y)*(ky[k] - points[j].y));
                    if(current_dist < min_dist)
                    {
                        min_index = k;
                        min_dist = current_dist;
                    }
                }

                // Increase the cluster's sum and count (locally)
```

```
            l_kx[min_index] += points[j].x;
            l_ky[min_index] += points[j].y;
            l_kcount[min_index]++;
        }

        // Clear the cluster values after everybody is done
        #pragma omp barrier
        #pragma omp single
        {
            for (size_t j = 0; j < CLUSTER_COUNT; j++)
            {
                kx[j] = 0;
                ky[j] = 0;
                kcount[j] = 0;
            }
        }

        // Add the new cluster values
        #pragma omp barrier
        #pragma omp critical
        {
            for (size_t j = 0; j < CLUSTER_COUNT; j++)
            {
                kx[j] += l_kx[j];
                ky[j] += l_ky[j];
                kcount[j] += l_kcount[j];
            }
        }

        // Calculate the new shared cluster value
        #pragma omp barrier
        #pragma omp single
        {
            // Calculate averages from the sums
            for (size_t j = 0; j < CLUSTER_COUNT; j++)
            {
                if(kcount[j] == 0) continue;

                kx[j] /= kcount[j];
                ky[j] /= kcount[j];
            }
        }
    }
}
```

```
    // Print clusters
    printf("\n\n");
    for (size_t i = 0; i < CLUSTER_COUNT; i++)
        printf("Cluster %ld: %lf,%lf\n", i+1, kx[i], ky[i]);
}


int main(int argc, char** argv)
{
    // Check the existence of the args
    if(argc == 1)
        usage("\nPlease provide the points as arguments...\n");

    // Parse Args
    size_t point_count = argc-1;
    struct point *points = malloc(sizeof(struct point) * point_count);

    // Read points from args
    for (size_t i = 0; i < point_count; i++)
        if(sscanf(argv[1+i], "%lf,%lf", &points[i].x, &points[i].y) != 2)
            usage("\nAn argument is invalid...\n");

    // Perform KMeans
    kmeans(points, point_count);

    return 0;
}
```

Example Run

Two clusters, five points, four processes

```
$ ./run.sh 1,3 4,5 -2,4.43 42.3,13 40,10 40,12 41,11
======================== MPI ========================
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really
needed).

Point 1: 1.000000,3.000000
Point 2: 4.000000,5.000000
Point 3: -2.000000,4.430000
Point 4: 42.300000,13.000000
Point 5: 40.000000,10.000000
Point 6: 40.000000,12.000000
Point 7: 41.000000,11.000000


Cluster 1: 0.999900,4.143305
Cluster 2: 40.824079,11.499838


======================= OPENMP =======================
Point 1: 1.000000,3.000000
Point 2: 4.000000,5.000000
Point 3: -2.000000,4.430000
Point 4: 42.300000,13.000000
Point 5: 40.000000,10.000000
Point 6: 40.000000,12.000000
Point 7: 41.000000,11.000000


Cluster 1: 1.000000,4.143333
Cluster 2: 40.825000,11.500000
```

```
mohamed@mohamed-G5-5590:~/Desktop/gam3a/par$ ./run.sh 1,3 4,5 -2,4.43 42.3,13 40,10 40,12 41,11
======================== MPI ========================
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).

Point 1: 1.000000,3.000000
Point 2: 4.000000,5.000000
Point 3: -2.000000,4.430000
Point 4: 42.300000,13.000000
Point 5: 40.000000,10.000000
Point 6: 40.000000,12.000000
Point 7: 41.000000,11.000000


Cluster 1: 0.999900,4.143305
Cluster 2: 40.824079,11.499838


======================== OPENMP ========================
Point 1: 1.000000,3.000000
Point 2: 4.000000,5.000000
Point 3: -2.000000,4.430000
Point 4: 42.300000,13.000000
Point 5: 40.000000,10.000000
Point 6: 40.000000,12.000000
Point 7: 41.000000,11.000000


Cluster 1: 1.000000,4.143333
Cluster 2: 40.825000,11.500000
```