Joe Ledger
EECS 600 Final Project: Progress Report
Parallelization of the K-Medoids Algorithm

Proposal

Background
The K-Medoids algorithm is one of the earliest and most influential clustering algorithms developed in the field of data mining. It attempts to partition a dataset of n points into k clusters (where k is a user-supplied parameter). Each cluster has a datapoint marked as the *medoid* of the cluster. The algorithm attempts to minimize the sum of pairwise distances between each datapoint and the medoid of the cluster that it is in. There are several versions of the K-medoid algorithm. I will be using the method proposed by Hae-Sang Park and Chi-Hyuck Jun[1] because it seems to be the most easily parallelizable.

Algorithm
The Park-Jun method for computing K-medoids is a greedy, iterative algorithm. While it is not guaranteed to find the optimal solution for a given dataset, the algorithm is gaurenteed to converge to a solution. In addition, the algorithm has been shown to converge at near-optimal solutions much faster than the exhaustive algorithm for many datasets.

The algorithm is comprised of the following steps:
1) **Select initial medoids** as the k datapoints for which $v_j = \Sigma_{i=1}^{n} \frac{d_{ij}}{\Sigma_{l=1}^{n} d_{il}}$ is minimized (d_{xy} is the Euclidean distance between datapoints x and y).
1a) Assign each object to the nearest medoid to form the initial clusters.
1b) Calculate the sum of the distances from all objects to their medoids.
2) **Update medoids** by finding the new medoid of each cluster (the datapoint minimizing the sum of the pairwise distance from itself to all other datapoints in the cluster)
3) **Assign objects to medoids** by assigning each datapoint to the nearest medoid to form new clusters.
3a) Re-calculate the sum of the distances from all objects to their medoids. If this value is equal to the value calculated in the previous iteration, stop the algorithm. Otherwise, go back to step 2.

Parallelization
Each of the 3 steps of the algorithm seems parallelizable. In the first step of the algorithm, we can calculate the value v_j for each datapoint j in parallel. In the second step of the algorithm, we can update the medoid of each cluster in parallel. Since there will be exactly k clusters at each iteration, we can use k threads to calculate new medoids (one thread per cluster). In the third step of the algorithm, we can split up all datapoints into x groups (with x being a user-supplied parameter). With x threads (one thread per group), we can assign each datapoint to a new medoid in parallel.

Sample Code
I do not plan on using any sample code. I will implement the algorithm from scratch in C (using OpenMP, OpenACC, and MIC compiler directives for parallelization).

References
[1] Park, H., & Jun, C. (2009). A simple and fast algorithm for K-medoids clustering. Expert Systems with Applications, 36(2), 3336-3341.

Progress

Results and Implementation Discussion

So far I have written code to parse arbitrary data from text files and store it into data stuctures in the program's memory.

I have created two structs, DataPoint and Cluster.

DataPoint stores a double array of feature values, as well as a string representing the classification label.

Cluster stores the number of datapoints in the cluster, as well as a pointer to the medoid of the cluster and an array of pointers to datapoints that are in the cluster.

I have not yet fully implemented the clustering algorithm yet.

However, I have fully outlined all methods that need to be implemented in order for the program to successfully run.

I do not anticipate implementing multiple versions of the code to run the OpenMP, OpenACC, and MIC versions of the program.

Instead I will use ifdef compiler directives so that my slurm scripts can decide what version of the program to run without writing any duplicate C code.

Program Listings

I only anticipate writing one C program, shown below. I will write three slurm scripts to run my C program, one each for OpenMP, OpenACC, and MIC. I plan on using *k* cores and *k* threads, where k is a user-supplied parameter deciding the number of clusters to create.

k_medoids.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>
#include <omp.h>


#ifdef IRIS
    #define nf 4
    #define nd 150
    #define k 3
    #define filename "../Data/iris.data"
    #define desc "IRIS"
#endif
```

```
#ifdef CARS
    #define nf 18
    #define nd 846
    #define k 4
    #define filename "../Data/cars.data"
    #define desc "CARS"
#endif

#ifdef SOYBEAN
    #define nf 35
    #define nd 47
    #define k 4
    #define filename "../Data/soybean-small.data"
    #define desc "SOYBEAN"
#endif

//Data Structure for a datapoint.
//It has an array of features as well as a string representing its classification type
struct DataPoint {
    double features[nf];
    char classification[20];
};

//Data Structure for a cluster
//It has a pointer to its medoid as well as an array of pointers to each datapoint in the
struct Cluster {
    int numDatapoints;
    struct DataPoint *medoid;
    struct DataPoint *datapoints[nd];
};

struct DataPoint datapoints[nd];
struct Cluster clusters[k];
double **matrix;
double *v_scores;
double *rowsums;

void readDatapoints() {
    char *array[nf + 1];
    double features[nf];
    char *tokens;

    FILE *file;
    char * line = NULL;
    int lineNumber = 0;
    int i = 0;
    size_t len = 0;
    ssize_t read;

    file = fopen(filename, "r");
    if(file){
        while((read = getline(&line,&len, file)) != -1){
            tokens = strtok(line,",");
            while(tokens != NULL){
                array[i++] = tokens;
                tokens = strtok(NULL,",");
            }

            for(i=0;i<nf;i++){
                features[i] = atof(array[i]);
            }

            memcpy(datapoints[lineNumber].features,features,sizeof(double) * nf);
            strcpy(datapoints[lineNumber].classification,strtok(array[nf],"\n"));
            lineNumber++;
            i = 0;
        }
        fclose(file);
    } else {
        printf("File doesn't exist.\n");
    }
}

double distance(struct DataPoint dp1, struct DataPoint dp2) {
    int i;
    double sum = 0.0;
    for(i = 0; i < nf; i++){
        sum += pow(dp1.features[i] - dp2.features[i],2.0);
    }
    return sqrt(sum);
}

int indexOfLargestElement(double* values, int length){
    int i, largeIndex;
    largeIndex = 0;
    double largeValue = values[0];
    for (i = 1;i < length;i++){
        if(values[i] > largeValue){
            largeIndex = i;
            largeValue = values[i];
        }
    }
    return largeIndex;
}

int indexOfSmallestElement(double* values, int length){
    int i, smallIndex;
    smallIndex = 0;
    double smallValue = values[0];
    for (i = 1;i < length;i++){
        if(values[i] < smallValue){
            smallIndex = i;
            smallValue = values[i];
        }
    }
    return smallIndex;
```

```
}


void getLowestElements(double* v_scores, int* buffer){
    int i,largeIndex;
    double largeValue;
    double* buffer_values = malloc(k * sizeof(double));
    for(i=0;i<k;i++){
        buffer[i] = i;
        buffer_values[i] = v_scores[i];
    }
    for(i=k;i<nd;i++){
        largeIndex = indexOfLargestElement(buffer_values,k);
        largeValue = buffer_values[largeIndex];
        if(v_scores[i] < largeValue){
            buffer[largeIndex] = i;
            buffer_values[largeIndex] = v_scores[i];
        }
    }

    free(buffer_values);
}


void selectInitialMedoids() {
    int i, j;
    v_scores = malloc(nd * sizeof(double));
    rowsums = malloc(nd * sizeof(double));
    for(i=0;i<nd;i++){
        rowsums[i] = 0.0;
        for(j=0;j<nd;j++){
            rowsums[i] += matrix[i][j];
        }
    }

    for(j=0;j<nd;j++){
        v_scores[j] = 0.0;
        for(i=0;i<nd;i++){
            v_scores[j] += (matrix[i][j] / rowsums[i]);
        }
    }

    int* lowestElements = malloc(k * sizeof(int));
    getLowestElements(v_scores,lowestElements);
    for(i = 0; i < k; i++){
        clusters[i].medoid = &datapoints[lowestElements[i]];
    }
    free(lowestElements);
    free(v_scores);
    free(rowsums);
}

void assignDataPointsToClusters(){
    int i,c;
    double* distances = malloc(k * sizeof(double));
    for(c=0;c<k;c++){
        clusters[c].numDatapoints = 0;
    }

    for(i = 0;i < nd; i++) {
        for(c = 0; c < k; c++){
            distances[c] = distance(datapoints[i], *clusters[c].medoid);
        }
        int close_index = indexOfSmallestElement(distances,k);
        clusters[close_index].datapoints[clusters[close_index].numDatapoints] = &datapoints
        clusters[close_index].numDatapoints++;
    }
    free(distances);
}

//Update medoids by finding the datapoint that minimizes
//the sum of pairwise distances from itself to all other datapoints in the cluster
void updateMedoids() {
    int i,j,c;
    for(c = 0; c < k;c++){
        int clusterSize = clusters[c].numDatapoints;
        double* distances = malloc(clusterSize * sizeof(double));
        #ifdef OMP
            #pragma omp parallel for private(j)
        #endif
        #ifdef ACC
            #pragma acc parallel for
        #endif
        for(i = 0; i < clusterSize; i++){
            distances[i] = 0.0;
            for(j = 0;j < clusterSize;j++){
                distances[i] += distance(*clusters[c].datapoints[i],*clusters[c].datapoints
            }
        }
        int medoidIndex = indexOfSmallestElement(distances,clusterSize);
        clusters[c].medoid = clusters[c].datapoints[medoidIndex];
        free(distances);
    }
}

double calculateTotalCost() {
    int i,c;
    double totalCost = 0.0;
    #ifdef OMP
        #pragma omp parallel for private(i)
    #endif
    #ifdef ACC
        #pragma acc parallel for
    #endif
    for(c = 0;c < k;c++){
        for(i = 0; i < clusters[c].numDatapoints;i++){
            totalCost += distance(datapoints[i],*clusters[c].medoid);
        }
    }
    return totalCost;
}

void cluster() {
    int i,j;
    double c;
    double cost = INFINITY;
    matrix = malloc(nd*sizeof(double *));
    #ifdef OMP
        #pragma omp parallel for private(j)
    #endif
    #ifdef ACC
        #pragma acc parallel for
    #endif
    for(i=0;i<nd;i++){
        matrix[i] = malloc(nd*sizeof(double));
        for(j = 0; j < nd; j++){
            matrix[i][j] = distance(datapoints[i],datapoints[j]);
        }
    }

    selectInitialMedoids();
    assignDataPointsToClusters();
    int safetyIterations = 0;
    while(cost - (c = calculateTotalCost()) > 1E-5){
        cost = c;
        updateMedoids();
        assignDataPointsToClusters();
        if(safetyIterations > 100){
            printf("Max iterations exceeded\n");
            break;
        }
        safetyIterations++;
    }

    for(i = 0; i < nd; i++) {
        free(matrix[i]);
    }
    free(matrix);
}

void printDatapoints() {
    int i,j;
    for(i = 0; i < nd; i++){
        for(j = 0; j < nf; j++){
            printf("%f\t", datapoints[i].features[j]);
        }
        printf("%s\n", datapoints[i].classification);
    }
}

int main(int argc, char *argv[]){
    readDatapoints();
    int i;
    double start_time,stop_time;
    printf("%s\t",desc);
    for(i = 0;i < 10;i++){
        start_time = omp_get_wtime();
        cluster();
        stop_time = omp_get_wtime();
        printf("%f\t",stop_time - start_time);
    }
    printf("\n");
}
```