Introduction

The purpose of this assignment is to become familiar with Pthreads and OpenMP by parallelizing a common algorithm in data mining: k-medoids clustering.

You need to write two programs, called ***km_pthreads*** and ***km_openmp***. Each program will accept three parameters as input: a filename of data points, the number of clusters, and the number of threads. Your programs should read in the specified file, cluster the given data points, and output the cluster assignments.

K-Medoids Algorithm

The k-medoids clustering algorithm clusters *N* data points into *K* clusters. Each cluster is characterized by a *medoid*, which is a data point in the cluster that has the smallest distance to all other points in the cluster. The algorithm proceeds as follows:

```
1. Select the initial K medoids.
  a. For reproducibility, use points 0, 1, ..., K-1.
2. Assign each data point to the closest medoid (measured via Euclidean distance).
3. While not converged:
  a. In each cluster, make the point that minimizes the average Euclidean distance to all other points in
the cluster the (new) medoid.
  b. Assign all data points to the closest medoid as measured via Euclidean distance.
```

Convergence is detected when no data points change their cluster assignment, or if the maximum number of iterations have been executed. **For this assignment, you should set the maximum number of iterations to twenty.**

Input/Output Formats

Each program will take as input one file (the list of data points), and output two files ("*clusters.txt*": the cluster assignments and "*centroids.txt*": the centers of each cluster).

The input file contains *N+1* lines. The first line contains two space-separated integers: the number of data points (*N*), and the dimensionality of each data point (*D*). The following *N* lines each contain *D* space-separated floating-point numbers which represent the coordinates of the current data point. Each floating-point number contains at least one digit after the decimal point. For example, an input with four two-dimensional data points would be stored in a file as:

```
4 2
502.1 505.9
504.0 489.4
515.2 514.7
496.7 498.3
```

The output file cluster assignments must be called *clusters.txt and* contain *N* lines. Each line should contain a single zero-indexed integer which specifies the cluster that the current data point belongs to. For example, a clustering of the above input file into two clusters may look like:

```
0
1
0
0
```

The second output file, *medoids.txt*, should follow the same format as the input data file. It should contain *K* data points, one for each cluster. Each coordinate should be written with at least three digits after the decimal point.

Your program must also print the clustering time to standard out. You should use a high-precision, monotonic, wall-clock timer and also omit the time spent reading and writing to files. We recommend the function *clock_gettime()* when on a Linux system. If using OSX, *mach_absolute_time()* is a good choice. **You will be graded on a Linux system.** Here is a function that you may use for timing:

```
/* Gives us high-resolution timers. */
#define _POSIX_C_SOURCE 199309L
#include <time.h>

/* OSX timer includes */
#ifdef __MACH__
  #include <mach/mach.h>
  #include <mach/mach_time.h>
#endif

/**
* @brief Return the number of seconds since an unspecified time (e.g., Unix
*        epoch). This is accomplished with a high-resolution monotonic timer,
*        suitable for performance timing.
*
* @return The number of seconds.
*/
static inline double monotonic_seconds()
{
#ifdef __MACH__
  /* OSX */
  static mach_timebase_info_data_t info;
  static double seconds_per_unit;
  if(seconds_per_unit == 0) {
    mach_timebase_info(&info);
    seconds_per_unit = (info.numer / info.denom) / 1e9;
  }
  return seconds_per_unit * mach_absolute_time();
#else
  /* Linux systems */
  struct timespec ts;
  clock_gettime(CLOCK_MONOTONIC, &ts);
  return ts.tv_sec + ts.tv_nsec * 1e-9;
#endif
}
```

You should use the following function to output your clustering time:

```
/**
* @brief Output the seconds elapsed while clustering.
*
* @param seconds Seconds spent on k-medoids clustering, excluding IO.
*/
static void print_time(double const seconds)
{
  printf("k-medoids clustering time: %0.04fs\n", seconds);
}
```

Timing information should be the ONLY thing printed to standard out.

Failure to follow any of these output instructions will result in significant loss of points.

Testing

Test data is provided on csel-plate0{1, 2, 3, 4}.cselabs.umn.edu at /export/scratch/CSCI-5451/assignment-1/.
We provide three test files: *small_gaussian.txt*, *large_cpd.txt*, and *tenth_large_cpd.txt*.
The dataset *small_gaussian.txt* is a small two-dimensional dataset that you should use for testing the correctness of your code.

You should only run your code on one of the plate machines. You will use the last digit of your student ID to select the machine. ID's ending in {0, 4, 8} should use plate01, ID's ending in {1, 5, 9} should use plate02, ID's ending in {2, 6} should use plate03, and ID's ending in {3, 7} should use plate04.

For the longer running tests, you should look into compiler optimization flags, as well as the screen command as these will be helpful.

Remember that the TA will be evaluating your data with a different data sets than those provided for testing.

What you need to turn in

1. The source code of your programs.
2. A short report including the following two parts:
    1. A short description of how you went about parallelizing the k-medoids algorithm. You should include how you decomposed the problem and why, i.e., what were the tasks being parallelized.
    2. Timing results for 1, 2, 4, 8, and 16 threads for k-medoids clustering.

       You should only include results on the 'tenth_large_cpd.txt' dataset with 256, 512, and 1024 clusters.

Do NOT include the test files; TAs will have their own test files for grading. You will lose significant points for including test files.

Additional specifications related to assignment submission

- A makefile must be provided to compile and generate the two executables. The executables should be named:
    - km_pthreads
    - km_openmp
- **Program invocation:** Your programs should take as an argument the input file to be read from, the number of clusters, and the number of threads to be used for parallel execution. For example, with two threads, each of the programs would be invoked as follows:

```
./km_pthreads /export/scratch/CSCI-5451/assignment-1/tenth_large_cpd.txt 512 2
./km_openmp /export/scratch/CSCI-5451/assignment1/tenth_large_cpd.txt 512 2
```

- All files (code + report) MUST be in a single directory and the directory's name MUST be your UMN login ID (e.g., your UMN email). Your submission directory MUST include at least the following files (other auxiliary files may also be included):

```
<UMN ID>/km_pthreads.c
<UMN ID>/km_openmp.c
<UMN ID>/Makefile
<UMN ID>/report.pdf
```

If you choose to code in C++, then replace the .c suffixes with .cpp or .cc.

- Submission MUST be in .tar.gz
- The following sequence of commands should work on your submission file:

```
tar xzvf <UMN ID>.tar.gz
cd <UMN ID>
make
ls -ld km_pthreads
ls -ld km_openmp
```

This ensures that your submission is packaged correctly, your directory is named correctly, your makefile works correctly, and your output executables are named correctly. If any of these does not work, modify it so that you do not lose points. The TAs can answer questions about correctly formatting your submission BEFORE the assignment is due. Do not expect them to answer questions the night it is due.

Failure to follow any of these submission instructions will result in significant loss of points.

Evaluation criteria

The goal for this assignment is for you to become familiar with the APIs and not so much for developing the most efficient parallel program (this will be done later). As such, full points will be given to the programs that:

1. follows the assignment directions;
2. solve the problem correctly;
3. do so in parallel (i.e., both clustering sub-steps are parallelized); and
4. achieves at least 14x speedup when using 16 threads for both openmp and pthreads

The speedups obtained will probably depend on the size of the input file. It is not expected that you to get good speedups for small files, but you should be able to get speedups for large files.