K MEANS CLUSTERING USING OpenMP

A REPORT ON PACKAGE

SUBMITTED BY

MRUDHULA G.P. – 18PD18

SAROOPASHREE K. – 18PD32

SUBJECT : OPERATING SYSTEMS

Department of Applied Mathematics and Computational Sciences,

PSG College of Technology ,

Coimbatore - 4.

K MEANS CLUSTERING USING OpenMP

Abstract:

This project presents a parallelized model for K-Means Clustering to classify data with OpenMP in C programming language.

1.1 Introduction:

OpenMP is an Application Program Interface (API), jointly defined by a group of major computer hardware and software vendors. OpenMP provides a portable, scalable model for developers of shared memory parallel applications.

K-means clustering is an iterative algorithm that tries to partition the dataset into K pre-defined distinct non-overlapping subgroups (clusters) where each data point belongs to only one group.

Iterative clustering with K-Means requires more execution time. To avoid such, a parallel partitioning of enhanced K-Means algorithm using OpenMP is proposed to handle the outliers with optimized execution time without affecting the accuracy.

1.2 Description:

1.2.1 What is OpenMP?

OpenMP is a set of compiler directives as well as an API for programs written in C, C++, or FORTRAN that provides support for parallel programming in shared-memory environments. OpenMP identifies parallel regions as blocks of code that may run in parallel. Application developers insert compiler directives into their code at parallel regions, and these directives instruct the OpenMP run-time library to execute the region in parallel.

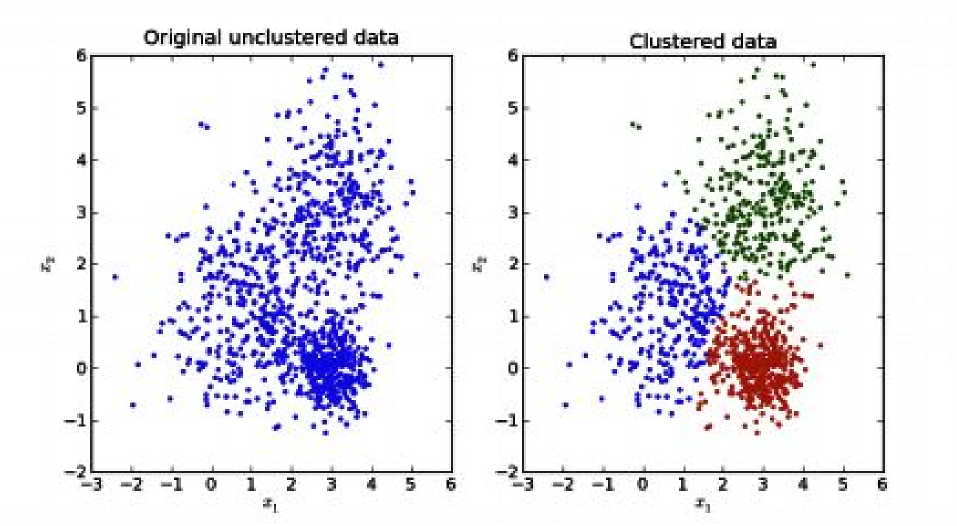
It creates as many threads which are processing cores in the system. Thus, for a dual-core system, two threads are created, for a quad-core system, four are created; and so forth. Then all the threads simultaneously execute the parallel region. When each thread exits the parallel region, it is terminated. OpenMP provides several additional directives for running code regions in parallel, including parallelizing loops.

In addition to providing directives for parallelization, OpenMP allows developers to choose among several levels of parallelism. E.g., they can set the number of threads manually. It also allows developers to identify whether data are shared between threads or are private to a thread.

1.2.2 What is K-Means Clustering?

Most unsupervised learning-based applications utilize the sub-field called Clustering. Clustering is the process of grouping data samples together into clusters based on a certain feature that they share. A cluster refers to a collection of data points aggregated together because of certain similarities.

K-means clustering is one of the simplest and popular unsupervised machine learning algorithms. It is an iterative algorithm that tries to partition the dataset into K pre-defined distinct non-overlapping subgroups (clusters) where each data point belongs to only one group. The main objective is to discover underlying patterns in the given data.In other words, k-means finds observations that share important characteristics and classifies them together into clusters. Customer Segmentation, Document Classification, House Price Estimation, and Fraud Detection are just some of the real world applications of clustering.



1.2.2.1 Pseudocode:

The intuition behind the algorithm is actually pretty straight forward. To begin, we choose a value for k (the number of clusters) and randomly choose an initial centroid (center coordinates) for each cluster. One initialization approach would be to choose our initial k positions from the dataset itself, setting one of the points as a Means value in its own surroundings. We incorporated the latter technique in our project.

Now, the algorithm goes through the data points one-by-one, measuring the distance between each point and the k centroids. The algorithm then groups the data point with the closest centroid (i.e. closest in distance).

As soon as we're done associating each data point with its closest centroid, we re-calculate the means — the values of the centroids; the new value of a centroid is the sum of all the points belonging to a cluster divided by the number of points in that cluster.

Once those centroids stop moving or if the difference in the coordinates of the updated centroid from that of the previous centroid is minimal, the clustering algorithm stops since the centroids have converged to nearly optimal value. This also means that each

```
Assign initial values for each u (from u=1 till u=k);

Repeat {
        Assign each point in the input data to the u that is closest
        to it in value;

Calculate the new mean for each u;

if all u values are unchanged { break out of loop; }
}
```

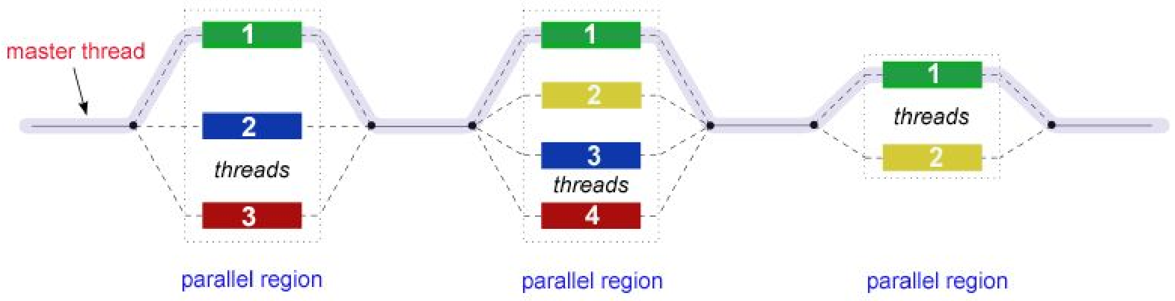
centroid has centered itself in the middle of its cluster, which is surrounded by its own circular decision boundary.

1.3. API used:

OpenMP is based on SPMD (Single Program Multiple Data) which means that a single program can work on multiple data by iterating over different data on the region of the memory. All tasks execute their copy of the same program simultaneously

1.3.1 Fork and Join Mechanism:

The fork and join mechanism started by a master thread (also served as a main entry to the whole program) creates other threads when it sees OpenMP directive to launch the desired number of threads.The forked thread will be executed concurrently by CPU’s context switching. When the forked threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.



1.4. Technology and Tools:

The implementation of Parallelized K-Means Clustering is done with the help of the omp.h library that is available in C language. Few basic thread controls and omp functions are explained below.

1.4.1.Compiler Directives:

Compiler directives (aka pragmas) appear as comments in the source code and are completely ignored by compilers unless you tell them otherwise – usually by specifying appropriate compiler flags. Compiler directives are used for various purposes like spawning a parallel region, dividing blocks of code among threads, distributing loop iterations among threads, synchronization of work among threads. Compiler directives have the following syntax:

```
#pragma omp construct [clause [clause]…]
      { structured_block }
```

Most OpenMP constructs apply to a “structured block”. Structured block is a block of one or more statements with one point of entry at the top and one point of exit at the bottom. A few of the compiler directives are explained in detail below.

1.PARALLEL Construct

A parallel region is a block of code that will be executed by multiple threads. This is the fundamental OpenMP parallel construct.When a thread reaches a PARALLEL directive, it creates a team of threads and becomes the master(Thread ID = 0) of the team.Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.There is an implied barrier at the end of a parallel region. Only the master thread continues execution past this point.

Syntax : `#pragma omp parallel [clause …]`

```
            { structured_block }
```

2. MASTER Construct

The MASTER directive specifies a region that is to be executed only by the master thread of the team. All other threads on the team skip this section of code. There is no implied barrier associated with this directive

Syntax: `#pragma omp master`

```
            { structured_block }
```

3. CRITICAL Construct

The CRITICAL directive specifies a region of code that must be executed by only one thread at a time.If a thread is currently executing inside a CRITICAL region and another thread reaches that CRITICAL region and attempts to execute it, it will block until the first thread exits that CRITICAL region.

Syntax:
```
#pragma omp critical [ name ]

        { structured_block }
```

4.BARRIER Construct

The BARRIER directive synchronizes all threads in the team. When a BARRIER directive is reached, a thread will wait at that point until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier.

Syntax:
```
#pragma omp barrier

        { structured_block }
```

5.SINGLE construct

The SINGLE directive specifies that the enclosed code is to be executed by only one thread in the team(whichever thread arrives first to the block).May be useful when dealing with sections of code that are not thread safe (such as I/O).Threads in the team that do not execute the SINGLE directive, wait at the end of the enclosed code block, unless a **nowait** clause is specified.

Syntax:
```
#pragma omp single [clause ...]

        { structured_block }
```

1.4.2 Run-Time Library Routines:

OpenMP includes several run-time library routines. These routines are used for various purposes such as setting and querying the number of threads, querying threads' unique identifier (thread ID), querying the thread pool size and so on. Let's look into a few common functions.

1. **omp_get_thread_num()** - Returns the thread id in a parallel region.
2. **omp_set_num_threads(nthreads)** - To set the number of threads used in a parallel region.
3. **omp_get _num_threads()** - Returns the number of threads used in a parallel region.
4. **omp_get_max_threads()** - Returns the maximum number of threads used for the current parallel region.
5. **omp_get_wtime() -** Returns the current time in the epoch time format.

1.5. Workflow:

The program starts by receiving the dataset filename, number of clusters(i.e., k), number of threads and the output filename for data points and clusters separately as command line arguments. Then the main function reads the dataset provided that the command line arguments are properly given.

The function to read the dataset is included in the *IO.h* file. The `readDataset()` function first scans the first line for the number of data points that is available in the file and prints the same. After which it reads the data points' coordinates one by one and stores it in a 2D array *data_points* which is of type float.

Before calling the function to create clusters we record the start_time using `omp_get_wtime()` function.

The function `kmeansClusteringOmp()` is the core function used in the code which is defined in *omp_kmeans.c* file. The function is fed with the number of data points (N),the number of clusters (k), the data points in the dataset as parameters along with the pointer variables to store centroids and cluster points.

The function initially defines the first K data points to be the initial K cluster center coordinates(centroids). Then it sets the number of threads to the user desired threads using `omp_set_num_threads()` API.

Then the cluster creation is a parallel process which is defined by `#pragma omp parallel` directive. For the clustering process we call the function `threadedClustering()`.

`threadedClustering()` function finds the Euclidean distance between the data points and all the centroids using the `findEuclideanDistance()` function and assigns the data point to the cluster whose centroid is nearest to the data point.

After these steps, the position of the centroids is to be changed to the Means of the respective data points in its cluster which is done in the critical section using the directive `#pragma omp critical`.

The above mentioned process keeps on continuing until the iteration reaches `MAX_ITERATION` or until there is no significant change in the centroids position. This is calculated by the master thread after updating the centroid position by calculating the difference in the Euclidean distance between the previous centroid's position and the updated centroid's position. Then we update the cluster points.

When all the above procedures are completed, the clustering part of the program is said to be finished, thus making the move to the second I/O phase of the program which is writing the clustered data points and coordinates of the centroids to two seperate files whose filenames along with their relative path are received as command line arguments beforehand.

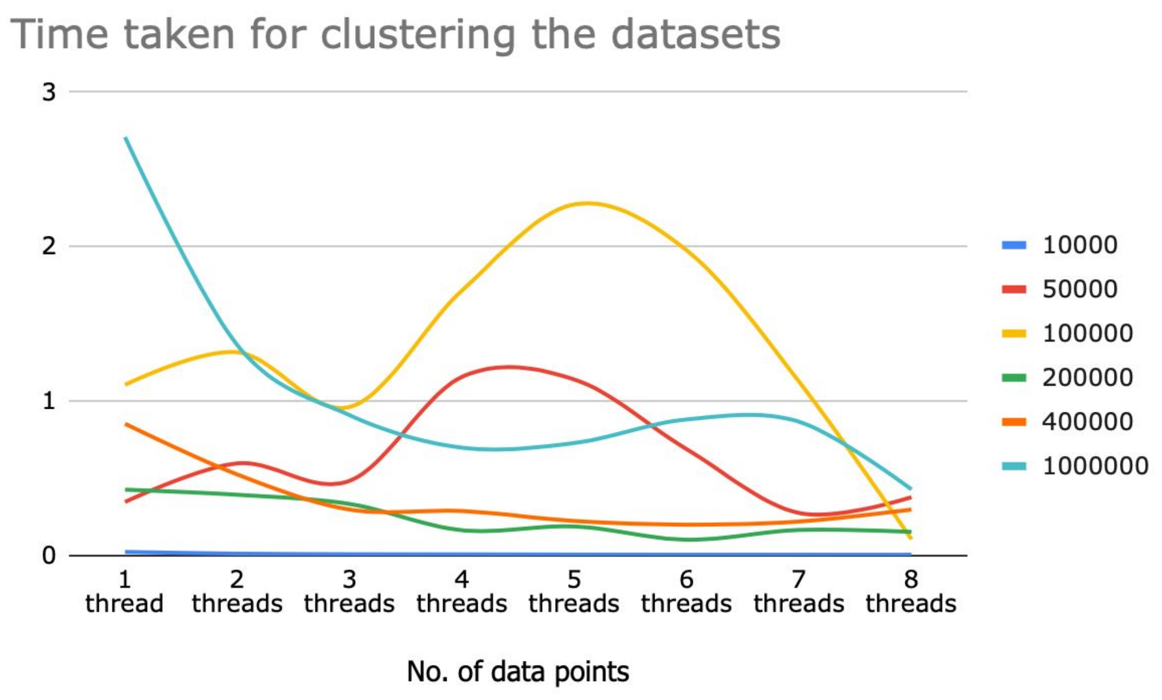
These I/O operations are facilitated by two functions `writeClusters()` and `writeCentroids()` in *IO.h* file.

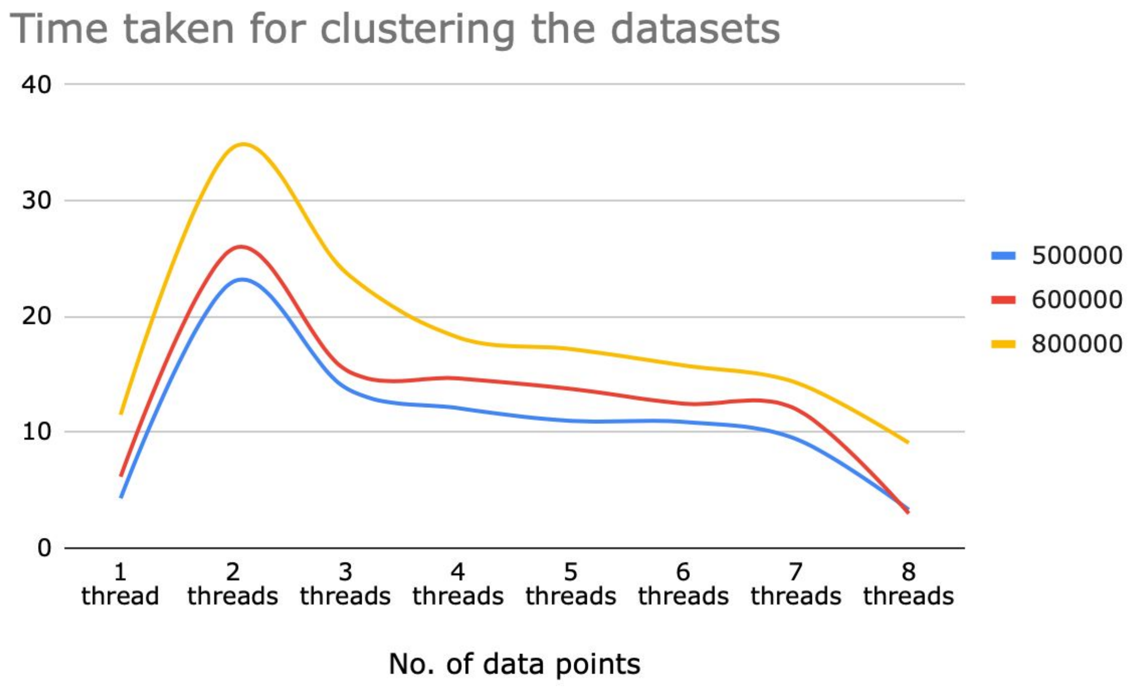
But before writing the output into files, we get the current time using `omp_get_wtime()` function to calculate the time taken to cluster the points by the parallelised `kmeansClusteringOmp()` function.

1.6 Results and discussion:

The following table shows the amount of time taken by the Parallelized K Means Clustering algorithm in seconds when using different numbers of threads and when subjected to different datasets having varied numbers of data points.

No. of data points	1 thread	2 threads	3 threads	4 threads	5 threads	6 threads	7 threads	8 threads
10000	0.02383	0.01243	0.00888	0.0081	0.00672	0.006	0.00589	0.00559
50000	0.34745	0.59751	0.48475	1.15734	1.13999	0.68857	0.27585	0.37846
100000	1.10589	1.31707	0.96232	1.71705	2.27425	1.97583	1.12551	0.10571
200000	0.42701	0.39406	0.33408	0.16446	0.18864	0.10275	0.16628	0.1537
400000	0.8538	0.52517	0.2981	0.28874	0.22421	0.20025	0.22059	0.29832
500000	4.30146	23.0046	13.8327	12.0836	10.9836	10.8942	9.41344	3.31326
600000	6.16556	25.8607	15.4	14.6531	13.7482	12.4708	11.991	2.9923
800000	11.5138	34.6003	23.8081	18.1851	17.1852	15.7755	14.2817	9.7823
1000000	2.71054	1.36294	0.90934	0.69898	0.72964	0.88188	0.86656	0.42861





1.7. Conclusion:

Most of the existing clustering algorithms are having less efficiency due to large volumes of datasets and outliers. To achieve better accuracy with minimum time complexity, a parallel enhanced clustering algorithm is proposed on multi-core systems.This project has presented a parallelized version of the existing K-Means Clustering algorithm.

1.8. Bibliography:

Websites:

1. http://www.bowdoin.edu/~ltoma/teaching/cs3225-GIS/fall17/Lectures/openmp.html
2. https://computing.llnl.gov/tutorials/openMP/#SINGLE
3. https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf
4. https://www.academia.edu/38994225/Parallelized_K-Mean_Clustering_with_OpenMP
5. https://towardsdatascience.com/how-does-k-means-clustering-in-machine-learning-work-fdaaaf5acfa0