Saint Petersburg National Research University of Information Technologies, Mechanics and Optics (ITMO University)

Faculty of Informational Technologies and Programming

**Report**

about laboratory work № 4

«Parallel Clustering Methods»

Student

Voronin Victor  J4132c c

(Surname, initials) Group

Saint-Petersburg, 2020

1. goal of laboratory work

Perform to cluster given dataset with parallel and serial ways.

1. Task Definition
2. Describe parallel implementations of 2 algorithms: DBSCAN and K-means in terms of parallel designing (PCAM Methodology).
3. Select some dataset and implement sequential and parallel k-Means algorithm. Compare performance of implemented algorithms.
4. Brief Theory

As initial dataset was choose South German Credit (UPDATE) Data Set with following parameters:

* Set Characteristics: Multivariate
* Number of Instances: 1000
* Area: Business
* Attribute Characteristics: Integer, Real
* Number of Attributes: 21
* Date Donated 2020-06-20
* Associated Tasks: Classification, Regression, Clustering
* Missing Values? N/A
* Number of Web Hits: 14426.

DBScan

* In the first step the entire dataset is loaded in equal-sized chunks by all processors in parallel. Then, the data is preprocessed. This entails assigning each of the d-dimensional points in the dataset to a virtual, unique spatial cell corresponding to their location within the data space, with respect to the given distance function. This allows us to sort the data points according to their proximity, and to redistribute them to distinct computation units of the parallel computing system. In order to balance the computational load for each of the processing units, it estimates the load using a simple cost heuristic accommodating the grid overlay.
* After this division phase, it performs the clustering of the redistributed points in the second step locally on each of the processing units, i.e., it assigns a temporary cluster label to each of the data points.
* Subsequently, these have to be merged into one global result view in step three. Whenever the temporary label assigned by a processing unit disagrees with the ones in the halo areas of the neighboring processors, it generates cluster relabeling rules.
* In the fourth step, the rules are broadcasted and applied locally.

kNN algorithm

Given a query q and a dataset s, a kNN query retrieves the k closest object in s whose distances from q are not larger than that of the k-th farthest nearest neighbor NN of q and dist is a distance metric.

For effective of kNN retrieval, two parameters ae introduced into the algorithm. Specially, ResultQueue maintains the current k most promising answer (which can be become the final NN’s of q) sorted in descending order in respect to the mindist metric, such that the greatest mindist (representing ResultQueue.MaxDist) enclosed in it can be captured instantly. The kDist specifies the current minimal maxdist ensuring that it contains at least k objects among all the maxdist(s) between q and entries retrieved so fat. Associating with both arguments the following pruning heuristics can be developed.

Heuristic 1. Let q be a given point and the entry Emin with the minimum distance to q among all the entries enclosed in priority queues. If the distance between Emin and q is greater than ResultQueue.MaxDist then the remainder in all priority queues can be discarded and the algorithm can be also terminated accordingly, since their distances from q are all larger than that if the current k-th farthest NN in ResultQueue of q.

Heuristic 2. If a node entry E whole distance from a given query point q is larger than the current value of kDist then the entry E can be safety pruned as it can not be become one of final k NN’s of q.

Heuristic 3. If an actual distance from given query point q is greater than the current value of kDist then ir can be safety discarded because it can not be enclosed in the final k NN’s of q.

Algorithm consist if the following steps:

* Suppose that the number of processors is M, the algorithm creates and initializes M priority queues (specifying Q1, Q2,...,QM) for M processors and ResultQueue used to keep the current k most promising answer.
* Algorithm inserts all the entries enclosed in the root node of parallel R-tree into their corresponding priority queues (e.g., heaps), and records the current value of kDist guaranteeing that it covers at least k objects
* Algorithms iterates following operations until it finds the final k NN’s of given query point q. Algorithm discards all non-qualifying entries from each priority queue by the heuristic 1 to 3 firstly. Subsequently, the entry Emin with the minimal distance from q among all the entries contained in M priority is found. In practical implementation, the following two steps are exploited:
  + Algorithm gets all entries EH(s)={EH1, EH2,…EHM} at the head if each priority queue firstly, assuming that all priority queues are sorted in ascending order with respect to the mindist metric.
  + It discovers the entry Emin­ in EH(s)

Next, algorithm computers the distance from Emin to q, and judges whether DIst(q,Emin­)>ResultQueue (enclosing k NN’s of q) and terminated the algorithm. In contrast, it fitches node entries or data points enclosed in all non-empty queues and enqueues them into corresponding priority queues in turn by parallelism. The following two cases were took into account. The furs tine is that if heads of non-empty queue are nit data object, i.e. intermediate nodes), then algorithm directly accesses these nodes and inserts all the entries within them in corresponding priority queue in parallel. Contrarily, the other case is that algorithm computes the distances from q and compares them against ResultQueue.MaxDist. Without loss of generality, assume that Dist(q,Ei)<ResultQueue.MaxDist holds, the entry Ei will be inserted into ResultQueue, since it may be enclosed in the final k NN’s of q. Then, algorithm finds the next non-data object N and also enqueues its entries into corresponding priority queue by parallelism.

1. Result And Experiments

The use of all of the above methods has been summarized in following table (see table 1).

Table 1 ‒ Time measurement for every versions of program

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Order of matrix | Avr time spenr for Serial version | MPI | | MPI with Derived data types | |
| Avr time, ms | Speedpup | Avr time, ms | Speedpup |
| 10 | 200.12 |  |  |  |  |
| 60 | 799.82 |  |  |  |  |
| 110 | 4600.26 |  |  |  |  |
| 160 | 14000.8 |  |  |  |  |
| 210 | 28399.3 |  |  |  |  |
| 260 | 63956.8 |  |  |  |  |
| 310 | 100882 |  |  |  |  |
| 360 | 159680 |  |  |  |  |
| 410 | 232900 |  |  |  |  |

As seen from figure 1, it has threshold in 4 threads after that increase the number of threads is heading for increase performance. Almost in all cases it happened save for critical section version where with each increase performance makes poorer. It can be explained with this fact that each threads finishes its part of tasks in about the same time after that begins to try to get access to critical section. Obviously, the more flows we have, the greater the crush at the entrance to the critical section arises.

General decrease of speedup after 4 threads may be explained with fact that create threads also requires time and if you have not strongly hard task speedup may be not arise because increase overhead cost for create of threads. That can be seen in given task. To this reason it is important to remember that parallel techniques is not nostrum and must be applied with realize.

1. Conclusion

During the execution of the task, message passing interface (MPI) was used to calculate multiply of two matrixes . The speed up obtained from the use of concurrency is determined. The results obtained were analyzed.

Appendix 1

#include<iostream>

#include<stdlib.h>

#include<time.h>

#include <chrono>

#include <stdio.h>

using namespace std;

static int n=415;

int main(int argc, char \*\*argv)

{

    srand(time(0)); //set state of generate depends on current time

    int firstMatrix[n][n], secondMatrix[n][n], mult[n][n];

    int r1, c1, r2, c2, i, j, k;

    r1=n;

    c1 = n;

    r2 = c1;

    c2=n;

    // Storing elements of first matrix.

    for(i = 0; i < r1; ++i){

        for(j = 0; j < c1; ++j)

        {

            firstMatrix[i][j] = rand() %10;

        }

    }

    // Storing elements of second matrix.

    for(i = 0; i < r2; ++i){

        for(j = 0; j < c2; ++j)

        {

            secondMatrix[i][j] = rand() %10;

        }

    }

    // Initializing elements of matrix mult to 0.

    for(i = 0; i < r1; ++i)

        for(j = 0; j < c2; ++j)

        {

            mult[i][j]=0;

        }

    double cntIteration = 5;

    double totalTime = 0.0;

    for (int q=0; q<cntIteration; q++)

    {

        auto t1 = std::chrono::high\_resolution\_clock::now();

        for(i = 0; i < r1; ++i)

        for(j = 0; j < c2; ++j)

            for(k = 0; k < c1; ++k)

            {

                mult[i][j] += firstMatrix[i][k] \* secondMatrix[k][j];

            }

        auto t2 = std::chrono::high\_resolution\_clock::now();

        std::chrono::duration<double, std::micro> fp\_ms = t2 - t1;

        totalTime += fp\_ms.count();

    }

    double timeSpentSerial = totalTime/cntIteration;

    cout << endl << "timeSpentSerial: " << timeSpentSerial;

    return 0;

}