Parallel K-means clustering

Andrea Frigo

ABSTRACT

This paper describes a parallel implementation of the K-means clustering algorithm, using MPI and OpenMP. The general serial algorithm is stated, then my approach to parallelization is described and analyzed. All the statistics are obtained testing the algorithm using the Unitn clusters, all the data can be found in the *statistics.xlsx* file.

1 INTRODUCTION

The K-means clustering is a non supervised machine learning algorithm. Given a set of n entities, having each one m attributes, and an integer number k, the algorithm divides the entities in k different groups/clusters. Each cluster is composed of the entities that are more similar to each other. An entity can be part of only one cluster, and to measure distances between entities it is used the squared euclidean distance based on the m different attributes given. Using this distance metric imposes that all the attributes have to be numbers. Given the nature of the algorithm, it is mostly used on dataset with n»m.

The problem is NP-hard, however the following algorithm can find an approximate solution, converging to a local optimum.

2 ALGORITHM DESCRIPTION

The algorithm to find an approximate solution with k clusters is the following:

```
1  read the dataset and store it in a matrix M;
2  choose randomly k elements of M and store them in a matrix C;
3  for each element E of M:
4      compute the distance between each element of C (centroid) and E and
           assign it to the centroid with smaller distance;
5  compute the average of all the data points of each cluster and update C;
6  if(stopping condition) the algorithm is finished, otherwise goto 3.
```

From the pseudo-code shown before some details have to be better explained, in particular:

- The random choice of initial centroids can change the final result and the time taken to converge. There exists some methods to try to select specific centroids in order to achieve better results, but they are not presented in this document.
- A centroid is a (not necessary real) point/entity that represents the cluster center, initially it is a real point, because it is chosen from the dataset, but after some computation it likely becomes only a symbolic point.
- To compute the average of all the data points assigned to each cluster, for every cluster also the number of points assigned to it has to be stored. The update of the centroid data structure is done every time an iteration of the algorithm is performed, this allows the algorithm to converge at some point.
- The stopping condition can be chosen as preferred, it should be based on the difference between the centroids of the previous iteration and the actual one. In the algorithm described in this document, the iterating process stops when previous and actual centroids are almost the same (differecne of at most 0.1 for each attribute).

The serial implementation of the algorithm can be found in the *parallel-kmeans-clustering/serial.c* file.

3 ALGORITHM ANALYSIS

The algorithm presented before is quite easy to analyse, even from a first look it can be expected that the most expensive part (in term of execution time) is the iterative loop that updates the centroid data structure. In the table 1, a time analysis of the different parts of the serial algorithm is shown. It is important to remember that changing the dataset or the initial centroids makes also the number of iterations to change.

Table 1: Serial implementation analysis (ncol=4, k=8)

NROW	TOTAL TIME (ms)	% READ DATA	% CYCLIC EXECUTION
100	32.2	96.857	3.112
1.000	69.2	88.931	11.057
10.000	249.8	17.677	82.320
100.000	2518	5.604	94.395
1.000.000	26550	3.729	96.271

It can be seen that, increasing the dataset length, the time taken to execute the iterative part of the algorithm becomes much grater then the others. In the table is not shown the time taken for generating the random centroids because it is always smaller than 10μs.

Given this simple analysis, the important part that has to be well parallelized is the iterative one, but, to do so, some important modifications have to be done.

4 PARALLELIZATION

As described in the abstract, the main goal of this project is to parallelize the algorithm, in particular the iterative part that changes the centroids. This section is divided in the two parts for the parallelization of the code: MPI and OpenMP. In these sections there are also some lines of C code, that can be found in the different C files of the repository.

There is an important consideration to make: the parallelization of the mean. This can be done keeping separate info about the partial sum and the partial number of element of the cluster, by doing so, the actual mean is calculated only at the end of each iteration from only one process, using the data gathered from all the other processes. Calculating the mean of each cluster in each process would have been better in terms of performances, both because the computation would have been splitted between all the processes and because it would have brought to sending only a k-long array instead of a k*ncol matrix and a k-long array. The problem is that, doing so, the info about the weight of each mean would have been lost. Up until now a classic arithmetic mean seemed enough, but implementing this parallelization approach brings to the necessity to use a weighted arithmetic mean.

4.1 MPI

I used MPI to divide the dataset (in form of matrix stored as a long array) between the different processes using the *MPI_Scatter* function. To do this, the dataset length has to be a multiple of the number of processes given, this is guaranteed by the *readFile* function, that adds some "invalid" rows to the data matrix. Doing

so, every function that needs to read some elements of the matrix needs also to check if the element is valid or not. After scattering the matrix, each process has a private section of it, where it's number of entities is nrow/n_proc.
To store the mapping between each data matrix row and the cluster where it's put, I need also another array, where array[i] represents the cluster assigned to the row i of the data matrix. This array is created by the first process and a smaller one (nrow/n_proc) by all processes.

These steps have to be done before starting the iterative part, because the data matrix has to be scattered just once and not modified, while the mapping array has to be modified, but it's read only at the end of the algorithm to get the final results.
For the centroids is instead different, they have to be the same for each process at the beginning of every iteration, so the centroids matrix (generated by the first process before scattering the data matrix) is broadcasted to every process using the *MPI_Bcast* function inside the while. The following code shows the iterative part of the algorithm, it is not the final implementation, because OpenMP is still not present, but it works fine with MPI.

```
1  bool stop = false;
2  while(!stop){
3      MPI_Bcast(centroids, k*ncol, MPI_DOUBLE, 0, MPI_COMM_WORLD);
4      int i,j,res;
5      for(i=0;i<k*ncol;i++) sumpoints[i] = 0.0;
6      for(i=0;i<k;i++) counter[i] = 0;
7      for(i=0; i<scatterRow;i++){
8          res = chooseCluster(i, k, ncol, recvMatrix, centroids);
9          if(res!=-1){
10             counter[res]++;
11             recvMapping[i] = res;
12             for(j=0;j<ncol;j++){
13                 sumpoints[res*ncol+j] += recvMatrix[i*ncol+j];
14             }
15         }
16     }
17     MPI_Reduce(sumpoints, sumpointsP0, k*ncol, MPI_DOUBLE, MPI_SUM, 0,
           MPI_COMM_WORLD);
18     MPI_Reduce(counter, counterP0, k, MPI_INT, MPI_SUM, 0,
           MPI_COMM_WORLD);
19     if(my_rank==0){
20         matrixMean(k, ncol, counterP0, sumpointsP0);
21         if (stopExecution(k, ncol, centroids, sumpointsP0)){
22             stop = true;
23         }
24     }
25     MPI_Bcast(&stop, 1, MPI_C_BOOL, 0, MPI_COMM_WORLD);
26 }
```

As stated before the data matrix has already been scattered into *recvMatrix*, which has *scatterRow* elements. Each process has also already created the partial *recvMapping* array. The *counterP0* and *sumpointsP0* have been declared by all processes, but instantiated only by P0, instead the *sumpoints* and *counter* structures are instantiated by all processes.
On each iteration the centroids are broadcasted by P0 and each process resets its *sumpoints* and *counter*. These two structures are used to store the sum of the points and the number of elements of each cluster.
The real execution starts from line 7, where, for each element of the partial dataset, the best centroid is calculated and the structures are updated. The function *chooseCluster* checks the distance between the point and all the centroids and returns the index of the centroid for which it has the minor distance (it returns -1 in case of invalid element). Knowing which cluster is the best, the number of elements of that cluster is increased, the mapping between cluster and point is stored and the element's attributes are summed with all the others of the same cluster (to calculate the mean later on).
As stated before, the computation of the mean is done by only P0, so the process needs to obtain both the *sumpoints* and the *counter*, this is done using the *MPI_Reduce* seen in line 17-18. The reduce function gathers all the partial arrays and sums them up element by element. From line 19, P0 has in the *sumpointsP0* matrix the sum of all the elements' attributes for each cluster, and in the *counterP0* the total number of elements assigned to each cluster. The *matrixMean* function does simply the mean for each cluster, getting k new centroids and storing them in the *sumpointsP0* matrix (that has the same dimension as the *centroids* one).
The *stopExecution* function checks if the two structures (the previous centroids and the just calculated ones) are almost the same and also updates the *centroids* matrix copying the *sumpointsP0*. Given that these final parts are executed only by P0, the stop variable has to be broadcasted to all processes, otherwise only P0 would stop its execution, while the other processes would start waiting for the broadcast of line 3, stalling forever.

After finishing the iterative part (when the stopping condition is accomplished), P0 has the final centroids matrix and the data matrix read at the beginning (and not ever modified). What it does not have is the complete mapping between each element of the data matrix and the different clusters, to obtain it a simple *MPI_Gather* function is used.
One single problem remains: the *mapping* array, generated and gathered by P0, could contain some invalid elements (if the data matrix has some invalid elements, also the array will have them). This is not a problem, because both the *MPI_Scatter* and the *MPI_Gather* work with a defined order: ascending order based on the process number. Given this property, both the data matrix and the mapping array have the possible invalid elements in their last elements, so there is no need to read all the elements to remove the invalid, but it is enough to restore the number of rows to the original one (the effective length of the dataset).

4.2 OpenMP

To further improve the algorithm performances, the parallelization with OpenMP alongside the previous one with MPI is a good solution. OpenMP can reduce the execution time and does not require to send the data, because of the shared memory architecture.
The OpenMP implementation for the iterative part of the algorithm is the following:

```
1  bool stop = false;
2  while(!stop){
3      MPI_Bcast(centroids, k*ncol, MPI_DOUBLE, 0, MPI_COMM_WORLD);
4      #pragma omp parallel num_threads(omp) shared(sumpoints, counter)
           private(partialMatrix, partialCounter)
5      {
6          int i;
7          #pragma omp for
8          for(i=0;i<k*ncol;i++) sumpoints[i] = 0.0;
9          #pragma omp for
10         for(i=0;i<k;i++) counter[i] = 0;
11         for(i=0;i<k*ncol;i++) partialMatrix[i] = 0.0;
12         for(i=0;i<k;i++) partialCounter[i] = 0;
13         int res;
14         #pragma omp for nowait schedule(static, 1)
15         for(i=0; i<scatterRow;i++){
16             res = chooseCluster(i, k, ncol, recvMatrix, centroids);
17             if(res!=-1){
18                 int j;
19                 partialCounter[res]++;
20                 recvMapping[i] = res;
21                 for(j=0;j<ncol;j++){
22                     partialMatrix[res*ncol+j]+=recvMatrix[i*ncol+j];
23                 }
24             }
25         }
26         #pragma omp critical(matrix)
27         matrixSum(k, ncol, sumpoints, partialMatrix);
28         #pragma omp critical(array)
29         vectorSum(k, counter, partialCounter);
30     }
31     MPI_Reduce(sumpoints, sumpointsP0, k*ncol, MPI_DOUBLE, MPI_SUM, 0,
           MPI_COMM_WORLD);
32     MPI_Reduce(counter, counterP0, k, MPI_INT, MPI_SUM, 0,
           MPI_COMM_WORLD);
```

```
33        if(my_rank==0){
34            matrixMean(omp, k, ncol, counterP0, sumpointsP0);
35            if (stopExecution(omp, k, ncol, centroids, sumpointsP0)){
36                stop = true;
37            }
38        }
39        MPI_Bcast(&stop, 1, MPI_C_BOOL, 0, MPI_COMM_WORLD);
40    }
```

This code is the development of the code shown in the previuos section, it is almost the same of the code of *parallel-kmeans-clustering/main.c*.
The first change is in line 4, where there is the first directive that splits the work of the single process (the master) among at most *omp* threads (the slaves), doing so, the work that was previously done by a single process is now done by the team of threads. In the same directive is also specified that the previously used structures are shared among the threads, and there are two new structures that are declared before and are private to each thread: *partialMatrix* and *partialCounter*; their use is described later in this section.
The *parallel for* directives of lines 7-11 are used to divide the reset of the shared structures between the threads, speeding up the reset process. The other two structures are private, so their reset has to be done serially by each thread.

The most important part of the iterative procedure starts at line 14, where there is another for directive, but with two additional clauses:

- nowait: this clause inhibits the implicit barrier at the end of the for, doing this, a thread that finishes the execution of this directive before some other thread can go on and does not have to wait for all other team threads to finish;
- static schedule: given the fact that the work is almost the same for each thread it makes no sense to use a dynamic schedule. I tried with different offsets to increase the spatial locality of the structures elements among each thread, but the results obtained were not better. This scheduling type should be the default one, but specifying it makes it sure to use this scheduling.

In line 26-29 there are two different named critical sections that update the shared structures for each thread using its own private structures. Having two named sections increases the performance, because, when a thread finishes the execution of the first one, it can start doing the second (if no other thread is inside it), allowing another thread to start the execution of the first. Without naming the critical sections (with different names), every process could not access one section if another one was executing the other section (decreasing the parallelism), same if a single bigger critical section would have been used.
The *matrixMean* and the *stopExecution* functions also use OpenMP to increase parallelization for P0, but are not analyzed in this document since they are simpler and present no new elements to discuss.
With these considerations this part of code is explained, but there is an important discussion on why these choices have been made. The instructions inside the for cycle are almost the same seen before, the only difference is that, instead of updating the shared structures, only the private structures are updated. This part is important for the OpenMP parallelization, because it ensures data consistency, avoiding race conditions among different threads.
The part of code of the big for cycle, but written serially, is here reported and analyzed, the analysis will bring to the previously described implementation.

```
1   for(i=0; i<scatterRow;i++){
2       res = chooseCluster(i, k, ncol, recvMatrix, centroids);
3       if(res!=-1){
4           counter[res]++;
5           recvMapping[i] = res;
6           for(j=0;j<ncol;j++){
7               sumpoints[res*ncol+j] += recvMatrix[i*ncol+j];
8           }
9       }
10  }
```

The variable *res* is declared inside each single thread, so it is private. The function *chooseCluster* uses elements from the *recvMatrix* and *centroids* matrices, that are shared between the threads of each team, but it just reads and never modify them. This part of code does not present problems so it can be kept the same.
More important considerations have instead to be made for the following lines of code:

- line 4: the $counter[res]++$ instruction reads the private variable res and that is fine, but also reads and writes one cell of the shared array counter, this can bring to inconsistency due to race condition between different threads. This is a loop-carried dependency (two different iteration of the loop could access the same cell of the array) and in particular causes a flow dependency (one thread writes a cell of the array and an other threads reads it).
- line 5: as before the *res* variable is private, while the i variable is the loop index, so it is guaranteed that its value is different for each thread. This guarantees that $recvMapping[i]$ is a different cell of the array for each iteration of each thread. Given this considerations, the instruction of this line is completely safe.
- line 7: given that j has to be declared private, reading the *recvMatrix* is not a problem even though it is shared, because it is not modified. Different situation instead for $sumpoints[res * ncol + j]$, because, given the fact that different threads could have the same value of *res*, the modification of the matrix comports the same risks discussed for line 4 (but with different increment offset, that makes no difference in practise).

The solution is simple but affects the performance, it's enough to create some private structures, update them and then update the shared structures in a critical section, in this way the correctness of the algorithm is guaranteed and the parallelization can help increasing the performances. Without using some additional private structures, but modifying directly the shared ones in a critical section would almost cancel the parallelization improvement. This solution is the same adopted by the OpenMP *reduction* clause, that handles perfectly cases like this, but that can only be used for single variables, not the structures used in the algorithm.

5 PERFORMANCES

The table 2 shows the execution time obtained testing the parallel algorithm with different number of rows-columns and different number of nodes-CPUs/node, in this document only some data are shown, more data can be found in the *statistics.xlsx* file.

As written before, the number of iteration is dependent on the initial centroids and the dataset, but, keeping the same centroids, the results obtained on the same dataset can be compared. With small datasets the improvement given by the parallelization is not interesting, that changes increasing the dataset size. Speedup and efficiency can be found in the *statistics.xlsx* file, the efficiency tends to be better with datasets having many entities and not many attributes (that is the standard for this machine learning algorithm) and tends to decrease when using many nodes and

Table 2: Execution time(s) parallel implementation with (R,C) rows-columns and (N,C) nodes-cpu/nodes (k=8)

	10*10K	**10*100K**	**10*1KK**	**1K*10K**
1N,1C	1.26	42.75	297.55	41.02
5N,1C	0.44	8.95	62.34	8.96
10N,1C	0.20	4.86	32.43	6.49
5N,5C	0.17	2.23	15.42	3.80
10N,10C	0.10	0.91	5.94	2.59

cores, because the dataset length is not enough to use well all the available resources.

Testing the algorithm I also found out that using more threads per process than the number of core per node decreases performances.

6 PROJECT STRUCTURE

To execute the algorithms on the Unitn clusters, extract the project archive on your home (*make*, *parallel.sh* and *serial.sh* have to be in the home directory).
Compile the algorithm using the make command: to compile the parallel algorithm just type *make* or *make parallel*, for the serial one type *make serial*, this should create the executable file.
To run the algorithm use the two given scripts, each one asks for some parameters (note that for the parallel one the number of nodes and the number of CPUs/node defines also the number of processes and the max number of threads/process). These scripts create another one (*script.sh*), that is submitted to PBS, the ID will be given. At the end of execution, the results can be found reading the *script.sh.oXXXX* file.
Typing *make clean*, the executable file, the generated script and results are deleted.
An example for running the parallel algorithm, using the dataset with 10.000 rows and 10 columns, 5 nodes and 5 CPUs/node, with k=8.
make
./parallel.sh 5 5 parallel-kmeans-clustering/datasets/dataset_10_10000.csv 8

7 CONCLUSIONS

The results obtained are good, the efficiency of the algorithm is quite high using big datasets, but it decreases when having many cores. The output is the same for the serial and the parallel algorithm, but small differences could be found, that is due to the memory representation of the double numbers, given that the operations done in parallel could be done in different order, thus bringing to different approximation of numbers in memory. Initially I wanted to use floating point numbers to decrease the memory usage and the size of elements to send using MPI, but the error propagation was too high and could bring to completely different results, using double precision floating numbers instead the results are quite precise.