Parallelization of K-means clustering algorithm through MPI+OpenMP and CUDA

Student Riccardo Inverardi Galli
Course Programmazione di sistemi Embedded e Multicore
Professor Daniele de Sensi

Overview of K-means

K-means is a machine learning clustering algorithm used to perform a classification task on a collection of points of arbitrary dimensionality. In k-means, the user chooses a number K of classes and the algorithm, given a set of points P of cardinality $|P| = m$, where each point $p \in P$ has dimensionality D, such that

$$p_i = (p_{i_1}, p_{i_2}, \ldots, p_{i_D}), 0 \leq i < m$$

and given a set C of centroids of cardinality $|C| = k$, each centroid of dimensionality D, will assign, over a series of iterations, each point to a centroid $c \in C$. The assignment of a point is performed by computing the euclidean distance

$$\sqrt{(p_i - c_j)^2}$$

between a point i and each centroid j and choosing the smallest between these values. This procedure is summarized in the following formula, which also takes into account the dimensionality D of points

$$\min_{0 \leq k < K} \sqrt{\sum_{d=1}^{D} (x_{i_d} - c_{k_d})^2} \tag{1}$$

where x_{i_d} is the d-th component of point i. The sequential version of the program provides the following straightforward implementation of euclidean distances:

```
    float euclideanDistance(float *point, float *center, int samples)
{
        int i;
        float dist=0.0;
        for(i = 0; i < samples; i++)
        {
                dist+= (point[i]-center[i])*(point[i]-center[i]);
        }
        dist = sqrt(dist);
        return(dist);
}
```

It is important to point out that centroids do not possess any special properties. Initially, they are just points chosen at random from the set of points P, and after the first iteration each centroid c_k is computed as the average of all points p that are assigned to k.

$$c_k = \frac{1}{|K_i|} \sum_{i \in K_i} x_i \tag{2}$$

The K-means algorithm works as follows:

1. Select K random centroids from the point cloud P.
2. Assign each point $p \in P$ to the nearest centroid by picking the smallest Euclidean distance for that point from all the distances between that point and all centroids (using 1).
3. Recalculate the centroids of each cluster as the mean of the data points assigned to each cluster (using 2).
4. Repeat steps 2 and 3 until:
   (a) The centroids do not change significantly; i.e. between one iteration and the previous one, the maximum movement of all centroids within each cluster is less than a given threshold.
   (b) The number of data point that move from one cluster to another between one iteration and the previous one is less than a given threshold.
   (c) A predefined maximum number of iterations is reached.

MPI+OpenMP

The first implementation leverages the parallelization opportunities presented by using in conjunction MPI and OpenMP. MPI is a standard designed to function on parallel computing architectures, based on the SPMD (single program, multiple data) paradigm, where multiple autonomous processors simultaneously execute the same program at independent points, while OpenMP is a API that supports multi-platform shared memory multiprocessing programming.
The approach chosen when combining these two parallel programming paradigms was very simple: Because of the nature of K-means, the only factors that can significantly increase the runtime of the application are

1. Cardinality $|P|$ of the point cloud.
2. Dimensionality of the points

The most straight-forward approach was to split the data cloud between a fixed number of MPI processes, and then parallelize the rest of the computation using OpenMP pragmas. This approach works very well due to the nature of the K-means algorithm; Since the heaviest computation at each iteration is computing the Euclidean distance of all points from each centroid (with m points and K centroids, we must perform mK distance calculations), scattering the points through n MPI processes reduces the theoretical computation time by a factor n (there are no dependencies between computing distances of two points i and j, $i \neq j$). This works because there is no redundant computation between processes, and each process i computes it's own part of

$$[p_l, p_{l+1}, \ldots, p_j]_{1 \leq l \leq j \leq m} \circ ([c_{l_1}, \ldots, c_{l_k}])$$

where $p_l \in P$, $c_l \in C$, and $\circ$ denotes the application of the Euclidean distance algorithm for each p.
To further increase parallelization OpenMP is used to parallelize for loops. This setup requires some calculations on the number of threads OpenMP can use. We want to have

```
MPI_proc + OMP_threads = CPU_cores
```

This number tends to be different on each system, thus we set the number of OpenMP threads to

```
int nthreads=NLOGIC_CORES / comm_sz;
omp_set_num_threads(nthreads);
```

where NLOGIC_CORES is the total number of logic cores of the system and is a compile time constant, and comm_sz is a runtime variable containing the size of MPI's communicator (the number of processes we specify with the -n flag of mpirun). This setup ensures that we are always using the full capacity of the CPU. To know the value of NLOGIC_CORES, we can run the following command on UNIX systems:

```
lscpu | grep -E "^Thread|^Core|^Socket|^CPU\("
```

By default, MPI creates a copy of all buffers for each process (SPMD), thus we only need to allocate two additional arrays, one to store the local part of the point cloud in each process, and another to store the local class map for the local point could. A class map is an array the indicates to what centroid k a point i belongs.
The program contains three parallel sections that do most of the computation for one iteration, while the last loop is performed sequentially for each MPI process, since the number of operations is too small to justify further parallelization overhead.
To distribute the point cloud between MPI processes, we first allocate this buffer:

```
int local_sz = lines / comm_sz;

/* Thread local bufffers */
float local_data[local_sz*samples];
int* local_classMap = (int*)calloc(local_sz,sizeof(int));
```

Here, **lines** contains $|P|$ (number of points), and **samples** is an integer indicating the dimensionality D of each point. To store a fraction of the point cloud, we initialize **local_data** as a one dimensional array, and we address different points using row-major order. This kind of ordering allows us to abstract 2D arrays while still storing points contiguously in memory. To access dimension j of point i, we translate $A[i][j]$ in $A[i * D + j]$. After the allocation we use MPI_Scatter to distribute the point cloud evenly, choosing process 0 as the root process for simplicity, although any other process could have done it.

```
/* Scatter data array */
if (rank == 0)
{
```

```
        MPI_Scatter(data, local_sz*samples, MPI_FLOAT, local_data,
        local_sz*samples, MPI_FLOAT, root, COMM);
    } else
    {
        MPI_Scatter(NULL, local_sz*samples, MPI_FLOAT, local_data,
        local_sz*samples, MPI_FLOAT, root, COMM);
    }
```

Initially, there are very few differences between the sequential version and this implementation. The flexibility of OpenMP pragmas allows to leave the code virtually untouched, excluding the limits of the loop and the arrays to work on, which are now bound to each local instance of the data cloud. The variables **changes** and **local_classMap** are declared as shared in the pragma, while the rest of the variables are kept private to each spawned thread. It is necessary to increment **changes** atomically to avoid race conditions and to ensure that the final value is the correct one. This is crucial because **changes** is a variable that determines a possible exit condition, so we must be very careful when dealing with it to make sure that we don't accidentally lose some data.

```
    it++;
    changes = 0;
    #pragma omp parallel for shared(local_classMap, changes)\
    private(i, _class, minDist, k, dist)
    for (i = 0; i < local_sz; i++) /* Iterate over each point */
    {
        _class = 1;
        minDist = FLT_MAX;
        for (k = 0; k < K; k++) /* Iterate over each centroid */
        {
            dist=euclideanDistance(&local_data[i*samples], &centroids[k*samples], samples);

           if(dist < minDist)
           {
                minDist=dist;
                _class=k+1;
           }
        }
        if(local_classMap[i] != _class)
        {
            #pragma omp atomic
            changes++;
        }
        local_classMap[i]=_class;
    }
```

The second part of the program needed some modifications from the original sequential version. In the sequential program computing the new centroids is done in two steps:

1. Accumulate all the values of points belonging to each class.
2. Divide the accumulate values by the number of points belonging to each class.

This approach must be changed in the parallel version because at this point of the algorithm processes do not share a common view of the data cloud. Each process can only accumulate to each class the number of points it was assigned. To solve this problem, we allocate two arrays to store a local version of **auxCentroids** and **pointsPerClass**. We use these arrays to handle local computation and then use a critical section to update the global (to each MPI process) arrays. After we are done with this step, we call MPI_Allreduce on **pointsPerClass**, **auxCentroids**, and **changes**, so that now all MPI processes share a common view of the accumulated values. Note that the last accumulation step (summing up all the values from all the different processes) is performed by the Allreduce operation.
After Allreduce is called, all the processes share a common view of the accumulated values and the number of points belonging to each class, and it's possible to perform the last step of centroid calculation, which is dividing by the number of points that belong to each class k. We use the special **MPI_IN_PLACE** flag to tell MPI that we wish to perform the Allreduce operation using the same array as sending and receiving array.

```
    #pragma omp parallel
    {
```

```
        int* local_pointsPerClass = (int*) calloc(K, sizeof(int));
        float* local_auxCentroids = (float*) calloc(K*samples, sizeof(float));

        #pragma omp for private(_class, i, j)
        for (int i = 0; i < local_sz; i++)
        {
            _class = local_classMap[i];
            local_pointsPerClass[_class-1] += 1;
            for(int j = 0; j < samples; j++)
            {
                local_auxCentroids[(_class-1)*samples+j] += local_data[i*samples+j];
            }
        }
        #pragma omp critical
        {
            for (int k = 0; k < K; k++)
            {
                pointsPerClass[k] += local_pointsPerClass[k];
                for (int j = 0; j < samples; j++)
                {
                    auxCentroids[k * samples + j] += local_auxCentroids[k * samples + j];
                }
            }
        }
        free(local_pointsPerClass);
        free(local_auxCentroids);
    }
    MPI_Allreduce(MPI_IN_PLACE, pointsPerClass, K, MPI_FLOAT, MPI_SUM, COMM);
    MPI_Allreduce(MPI_IN_PLACE, auxCentroids, K*samples, MPI_FLOAT, MPI_SUM, COMM);
    MPI_Allreduce(MPI_IN_PLACE, &changes, 1, MPI_INT, MPI_SUM, COMM);
    #pragma omp parallel for shared(pointsPerClass, auxCentroids) private(k, j)
    for(k = 0; k < K; k++)
    {
        for(j = 0; j < samples; j++)
        {
            auxCentroids[k*samples+j] /= pointsPerClass[k];
        }
    }
```

After this step is performed, the computation for one iteration is finished. The new centroids have been computed, and all that is left is to compute the distance between the old centroids stored in **centroids** and the new ones stored in **auxCentroids**. This has to be done in order to check if the new centroids are at a significant distance from the old ones. If they are not, the algorithm is free to terminate. This last part is left sequential for each MPI process, and is identical to the original sequential code.

```
    maxDist=FLT_MIN;
    for(k = 0; k < K; k++)
    {
        distCentroids[k] = euclideanDistance(&centroids[k*samples], &auxCentroids[k*samples], sam
        if(distCentroids[k]>maxDist)
        {
            maxDist=distCentroids[k];
        }
    }
    memcpy(centroids, auxCentroids, (K*samples*sizeof(float)));
```

CUDA

CUDA is Nvidia's proprietary parallel computing API which allows programmers to access a Nvidia GPU instruction set through the C++ programming language. GPU architectures are specifically made to leverage parallelism and high bandwidth communication inside each device, through a very high number of cores and several levels of

in-device memory.
To parallelize the K-means clustering algorithm on a GPU, one has to completely rethink the logic of the code. Since the main factor that influences runtime of the application is the number of points the algorithm has to iterate on, it was natural to think that the algorithm should be parallelized over each point. By assigning to each core in every streaming multiprocessor (SM) a point in the point cloud it is possible to speed up the application considerably.
Constants that are used frequently by kernels are placed in constant memory so that they can be quickly broadcasted to all cores on the device. In particular, we declare three variables that are very frequently used as constant variable, and during the memory allocation phase they are set to their values with cudaMemcpyToSymbol.

```
__constant__ int samples_d;
__constant__ int K_d;
__constant__ int lines_d;
...
CHECK_CUDA_CALL( cudaMemcpyToSymbol(samples_d, &samples, sizeof(int)) );
CHECK_CUDA_CALL( cudaMemcpyToSymbol(K_d, &K, sizeof(int)) );
CHECK_CUDA_CALL( cudaMemcpyToSymbol(lines_d, &lines, sizeof(int)) );
```

Similarly to what we did with MPI+OpenMP version, we still have to compute the euclidean distance of all points from all centroids at every iteration of the algorithm. To avoid high latency issues that come from communicating with global memory, because each core computing values for a single point would have to request from global memory every centroid, we place in shared memory of every SM the entire array containing the centroids. In order to do this, we must address two problems:

1. We do not know a priori the size of the centroids array, since both the dimensionality and the number of centroids are known only at runtime
2. We have a limited amount of shared memory in each SM

On Nvidia devices with a 7.5 compute capability like the ones that are available on sapienza's cluster, there is a unified data cache of 96Kb, and a part of it can be reserved to be allocated as shared memory (32Kb or 64Kb). The programmer must specify the percentage of total shared memory capability he wants to use for each kernel that needs shared memory usage. To do this, we must use cudaFuncSetAttribute(), passing to it as arguments the kernel, what attribute we wish to set (in our case shared memory carveout), and the percentage of shared memory we wish to allocate.

```
int carveout = 100;
...
CHECK_CUDA_CALL( cudaFuncSetAttribute(firstStepGPU,
cudaFuncAttributePreferredSharedMemoryCarveout, carveout) );
```

Since the centroids array can be arbitrarily big, and the dimensionality of points can go up to 100 in our test cases, we set the carveout to 100, thus reserving the whole 64K shared memory space, in order to be sure that we can fit at most 16384 4 byte floating point numbers in it (which should be enough for any centroids array). It is important to note that this must be done only for those kernels that require shared memory.
Another issue with shared memory is dynamic allocation, because we can't statically allocate shared memory in a kernel like so:

```
__shared__ float sharedCentroids[K_d*samples_d];
```

This allocation would result in a compilation error unless **K_d** and **samples_d** are compile time constants, which they are not. To fix this problem, we must dinamically allocate shared memory by declaring an unspecified size array, using the **extern** keyword, and then passing the size of the array as a parameter of the kernel call:

```
KERNEL CODE
extern __shared__ float sharedCentroids[];
...
DEVICE CODE
firstStepGPU<<<dimGrid, dimBlock, csize>>>(...kernel arguments...);
```

Computing the euclidean distance is done through a **__device__** version of the provided **euclideanDistance** function, called **euclideanDistanceGPU**. The implementation is virtually identical, with the exception that the dimensionality of the points must not be passed as an argument to the function because it's a broadcasted constant from constant memory.

```
__device__
float euclideanDistanceGPU(float* point1 /* in  */,   /* Point 1 */
                           float* point2 /* in  */)   /* Point 2 */
{
        float dist = 0.0;
        for (int i = 0; i < samples_d; i++)
        {
                dist += (point1[i] - point2[i]) * (point1[i] - point2[i]);
        }
        dist = sqrtf(dist);
        return (dist);
}
```

After all the logic for parallelization is set up, the individual code each thread executes is very similar to the sequential version. The blocks are organized as a one dimensional of size 64, and the block grid is also one dimensional of size 72. The code calls four kernels:

1. **fistStepGPU**: It computes the distance of each point from all the centroids, and saves in classMap[p] the closest centroid to point p.
2. **recalculateCentroidsStep1GPU**: It computes the values needed for the calculation fo the new centroids. After the execution, auxCentroids will have stored the (not normalized) sum of all points that belong to each class, and pointsPerClass will contain the count of all points belonging to each class.
3. **recalculateCentroidsStep2GPU**: It divides the sum of the values of all the points belonging to class k by the number of points belonging to class k.
4. **recalculateCentroidsStep3GPU**: Checks whether the distance between the old centroids and the new ones is below the given threshold