Parallel Methods for K-Means Clustering

Moses Modupi
1614669

December 2, 2021

School of Computer Science & Applied Mathematics
University of the Witwatersrand

Chapter 1

Introduction

In recent years, data has played a large role in technological advancements. We have seen high-volumes in data being collected over recent years, and this is what we know as the age of data. Much of the data being collected is high-dimensional, consisting of thousands of dimensions and someone has to make sense of it all. Cluster analysis has been a tool used to find natural groupings for a set of instances in a dataset. K-means is a simple clustering technique dating back to 1955 [Jain 2010]. It clusters instances based on similarity. The most common similarity metric used is the euclidean distance. The most basic implementation of K-means is quite computationally intensive, and there has been work to produce more efficient algorithms to the Kmeans clustering algorithm. Some approaches apply Lloyd's algorithm as a heuristic for Kmeans [Kanungo *et al.* 2002]. Lloyd's algorithm used for partitioning evenly spaced points in Euclidean Spaces into uniformly sized convex cells. There also have been work to parallelise the Kmeans algorithm by using architectures provided by CUDA and MPI [Zhang *et al.* 2011; Farivar *et al.* 2008; Hong-tao *et al.* 2009] . In this work, we will be exploring performance gains regarding parallel implementations of Kmeans clustering using MPI and CUDA. These two platforms will be representatives memory passing paradigm and shared memory paradigm.

Chapter 2

Methodology

2.1 Serial Implementation

```
void serial_kmeans(double* D, double* C, int k, int n, int d, int
    it_max){
     int* assign = new int[n];
     int it = 0;

     // initial centroids
     init_centroids(C, k, d);

     while(it<it_max){
          assign_to_centroids(D, C, assign, k, n, d);
          update_centroids(D, C, assign, k, n ,d);
          it++;
     }

     delete[] assign;
}
```

Listing 2.1: serial Kmeans

Listing 2.1 shows a serial implementation of Kmeans. The first step is to get the initial centroids. The initial centroids are usually assigned to random points. This choice affects the number of iteration needed till convergence as well as the final centroids. We will assign the initial centroids to the first k data points to be able to confirm correctness across the different implementation. The next part of the algorithm has to do with the convergence criteria. Normally this criteria would be to stop when the centroids barely change. This will make it hard to control the time complexity of the algorithm so we will fix the number of iteration. In the algorithm, we have two subroutines, namely assign_to_centroids and update_centroids. These two subroutines are going to be the focus of parallelization since they are the most interesting and computationally intensive.

```
void assign_to_centroids(double* D, double* C, int* assign, int k, int
    n, int d){
     float sum = 0;
     float min;
     int min_c;

     for(int i=0; i<n; i++){
          min = MAX_DISTANCE;
          for( int j=0; j<k;j++){
               sum = 0;
               for(int l=0; l<d; l++){
                    sum+= (D[i*d+l]-C[j*d+l])*(D[i*d+l]-C[j*d+l]);
               }

               sum = sqrt(sum);
               if(sum<=min){
                    min = sum;
                    min_c = j;
               }
          }
          assign[i] = min_c;
     }
}
```

Listing 2.2: serial Kmeans - assign_to_centroid

We will start describing the assign_to_centroid subroutine. This subroutine assigns each data point to the centroid with the least euclidean centroid. Listing 2.2 shows the serial implementation for the assign_to_centroid subroutine. In the code, we have a loop that runs through each data point and for each data point we find the distance between it and a centroid while keeping track of the closest centroid. The distance computation itself is a loop that runs through each dimension. The complexity of this subroutine is thus $O(knd)$ where k is the number of centroids, n is the number of data points and d is the dimension. When we factor in the complexity of the complete algorithm, the complexity works out to $O(IT_C * knd)$. IT_C is the number of iterations for the while loop and the C signifies that it is dependent on the convergence criteria.

The Listing 2.3 shows the update_centroid subroutine that follows after assigning data points to centroids. This subroutine updates the centroids by setting each centroid to the mean value of the points that were assigned to it. In the code, we first make sure we set the centroids to data points with all 0. This step takes $O(k * d)$. Then we have a loop that runs through each centroid. At each centroid, we make a pass over every data point. If the said data point was assigned to the current centroid we add the data point to the centroid. Adding a data point to a centroid requires a loop that adds each dimension to the centroid. We will consider the number of additions to centroids we make as the basic operation. The complexity, in this case, is $O(n * d)$ and $O(IT_C * n * d)$ for the whole algorithm.

```
void update_centroids(double* D, double* C, int* assign, int k, int n,
    int d){
     int count;
     for(int l=0;l<d*k;l++)C[l] = 0; //clear the array
     for(int i=0;i<k;i++){
          count = 0;
          for(int j=0;j<n;j++){
               if(assign[j]==i){
                    count++;
                    for(int l=0;l<d;l++){
                         C[i*d+l] += D[j*d+l];
                    }
               }
          }
          for(int l=0;l<d;l++)C[i*d+l]/=count;
     }
}
```

Listing 2.3: serial Kmeans - update_centroid

2.2 Parallelizing assign_to_centroid

2.2.1 Partitioning and Communication

We will partition our tasks into classes.
Task Class 0: $T^0_{i,j,l}$
computes $(X_i^{(l)} - C_j^{(l)})^2$ where $X_i^{(l)}, C_j^{(l)}$ is the l-th dimension of the i-th data point and j-th centroid.
Task Class 1: $T^1_{i,j,l}$
takes in tasks from $T^0_{i,j,l}$ and computes a sum reduction for all tasks T^0 that share the same i, j.
Task Class 2: $T^2_{i,j}$
Receives results from reduction done in T^1 and computes the square root followed by a reduction operation to minimum of all tasks that share the same i. Once the minimum is found, we assign it to point i.

Figure 2.1 shows the task communication.

2.2.2 Agglomeration

We shall group tasks with the same i in the same agglomeration. That is, each agglomeration is a group of tasks that are involved in finding the closest centroid to point i. Figure 2.2 shows the agglomeration of the tasks.

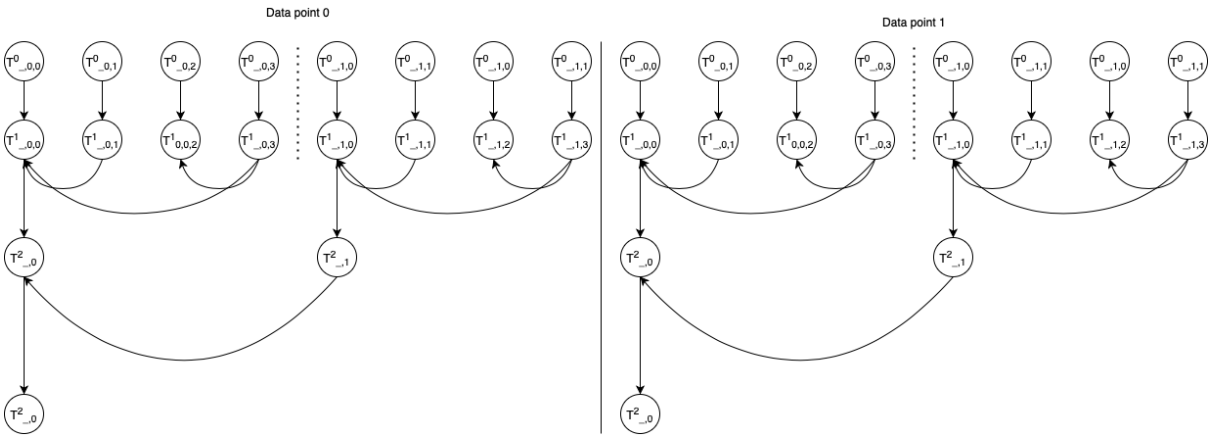

Figure 2.1: Task Communication graph.

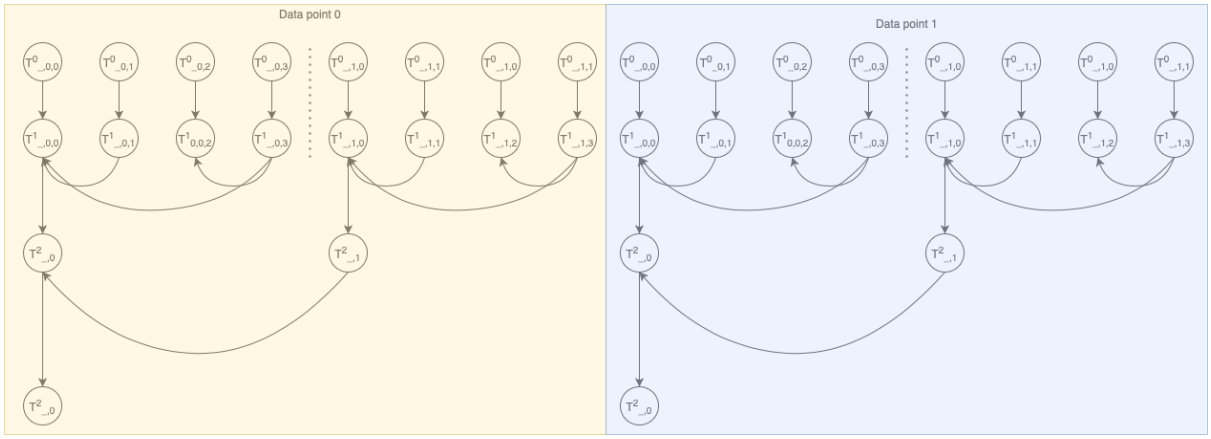

Figure 2.2: Task Agglomeration

2.2.3 Processor Mapping with MPI

With MPI, we will allocate each processor we have to $s = \frac{n+p-1}{p}$ where p is the number of processors. Listing 2.4 shows the MPI implementation. In this implementation, the processor rank was set to the variable id which is a member variable of a class.

```
void assign_to_centroids(double* D, double* C, int* assign, int k, int
    n, int d){
     float sum = 0;
     float min;
     int min_c;

     int s=n/num_p; //number of points per processor
     int up = (s+s*id<n)?s+s*id:n; // to stop from going over board
     for(int i=s*id; i<up; i++){
          min = MAX_DISTANCE;
          for( int j=0; j<k;j++){
               sum = 0;
               for(int l=0; l<d; l++){
                    sum+= (D[i*d+l]-C[j*d+l])*(D[i*d+l]-C[j*d+l]);
               }

               sum = sqrt(sum);
               if(sum<=min){
                    min = sum;
                    min_c = j;
               }
          }
          assign[i] = min_c;
     }
}
```

Listing 2.4: MPI Kmeans - assign_to_centroid

2.2.4 Processor Mapping with CUDA

Because we have more processors to work within CUDA we can do more with each agglomeration. We will first define a processor as having several k blocks. We do this because k is usually a relatively small parameter. Each block has several threads but cannot be greater than 1024. We will divide an agglomeration into multiple subroutines. The first subroutine will be calculating all necessary distance computations. This is something that spans over task class 0 and 1. Listing 2.6 shows the kernel for computing the distance computations. In our approach, we have 2 levels to the hierarchy of threads. The first level of threads are threads that belong to a certain processor (as we defined for CUDA). The second level is the threads that belong to a certain block.

In the first part, we calculate the distance components on each dimension. We use a block to calculate the these distance components. Since the dimension can be greater that what a block can provide, each thread in a block will have to pick up dimensions in a round about way. The second part is to add up the distance components across all dimensions. We use a recursive doubling strategy for the reduction operation.

```
__global__ void cuda_compute_distances(point_t sums, point_t data,
    point_t centroids, int k, int d){

    //get the group id, block id and thread id.
    int tid = threadIdx.x; int bid = blockIdx.x%k;
    int gid = blockIdx.x/k;

    //shared memory array to store the terms from the euclidean
    distance
    __shared__ element_t ds[BLOCKWIDTH];
    ds[tid] = 0;

    //just some peace of mind
    if(tid>=d)return;

    //putting together the diffs
    while(tid<d){
        ds[tid%BLOCKWIDTH]+= (data[gid*d+tid]-centroids[bid*d+tid])*(
    data[gid*d+tid]-centroids[bid*d+tid]);
        tid+=BLOCKWIDTH;
    }
    tid = threadIdx.x; //reset tid
    __syncthreads();

    //putting together the sums (reduction)
    int slab = (d<BLOCKWIDTH)?d>>1:BLOCKWIDTH>>1;
    while(slab>=1){
        if(tid<slab)
            ds[tid] += ds[tid+slab];

        slab = slab>>1;
        __syncthreads();
    }

    sums[gid*k+bid] = sqrt(ds[0]);

}
```

Listing 2.5: CUDA Kmeans - assign_to_centroid (distance computation)

The next step is to find the closest centroid. It this part we map each block i to compute the closest centroid to point 1. This is shown in Listing 2.6.

```
__global__ void cuda_compute_argmins(int* argmin, point_t sums, int k){
    int tid = threadIdx.x; int bid = blockIdx.x;

    __shared__ element_t dmins[BLOCKWIDTH];
    __shared__ int dargs[BLOCKWIDTH];

    // initialise to max values
    if(tid>=k)return;
    while(tid<k){
        dmins[tid]= MAX_DISTANCE;
        tid+=BLOCKWIDTH;
    }
    __syncthreads();

    // serial min
    tid = threadIdx.x;
    while(tid<k){
        if(dmins[tid%BLOCKWIDTH]>sums[bid*k+tid]){
            dmins[tid%BLOCKWIDTH] = sums[bid*k+tid];
            dargs[tid%BLOCKWIDTH] = tid;
        }

        tid+=BLOCKWIDTH;
    }
    __syncthreads();

    //putting together the sums (reduction)
    tid = threadIdx.x;
    int slab = (k<BLOCKWIDTH)?k:BLOCKWIDTH;
    slab = slab>>1;
    while(slab>=1){
        if(tid<slab && dmins[tid]>dmins[tid+slab]){
            dmins[tid] = dmins[tid+slab];
            dargs[tid] = dargs[tid+slab];
        }

        slab = slab>>1;
        __syncthreads();
    }

    if(tid==0){
        // printf("min %d, %0.2f, %d\n", bid, dmins[tid], dargs[tid]);
        argmin[bid] = dargs[tid];
    }

}
```

Listing 2.6: CUDA Kmeans - assign_to_centroid (argmin computation)

2.3 Parallelizing update_centroid

2.3.1 Partitioning and Communication

Task Class 0: $T^0_{i,l}$
Sets the l-th dimension of centriod i to 0
Task Class 1: $T^0_{i,j,l}$
Adds dimension l of point j to centroid l if point j was assigned to centroid i. Also increments a count of the number of points assigned to centroid i.
Task Class 2: $T^1_{i,l}$
Divides the l-th dimension of centroid i by the number of points assinged to centroid i.

Figure 2.3 shows the task communication.

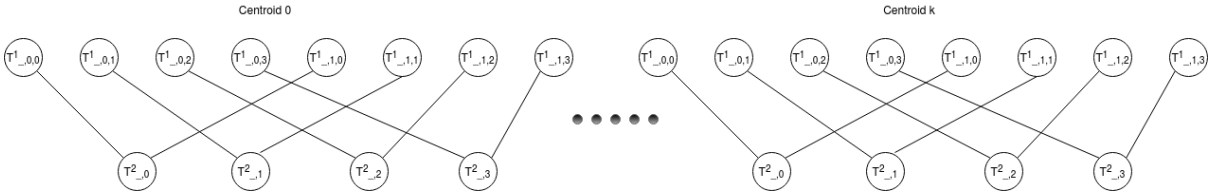

Figure 2.3: Task Communication graph.

2.3.2 Agglomeration

Each Agglomeration is going to be defined as a group of task with the responsibility of updating a single centroid. Figure 2.4 shows the agglomeration of the tasks.

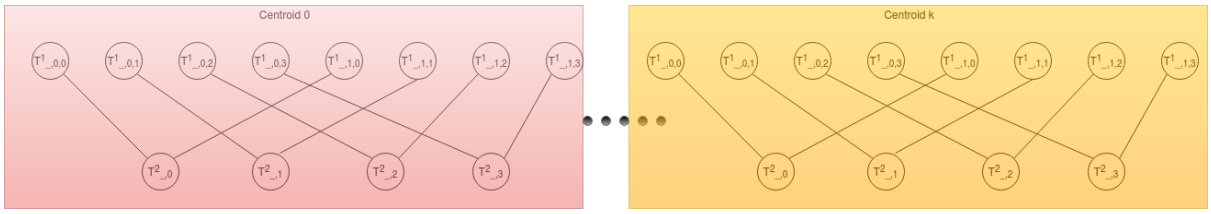

Figure 2.4: Task Agglomeration

2.3.3 Processor Mapping with MPI

With MPI, we evenly allocate the agglomerations to the processors as shown in code listing 2.7. Listing 2.8 shows the full code for kmeans. We can take note of the message passing. Each process boardcasts their respective parts of the assignment array. The same thing happens after the update_centroid step. Each process has to broadcast their respectice designated centroids.

```
void update_centroids(double* D, double* C, int* assign, int k, int n,
    int d){
     int count;

     int s = (k>num_p)?k/num_p:1;

     int up = (s*id+s<k)?s*id+s:k;
     for(int i=0;i<k;i++)
          for(int l=0;l<d;l++)
               C[i*d+l] = 0; //clear the array

     for(int i=s*id;i<up;i++){
          count = 0;
          for(int j=0;j<n;j++){
               if(assign[j]==i){
                    count++;
                    for(int l=0;l<d;l++){
                         C[i*d+l] += D[j*d+l];
                    }
               }
          }

          for(int l=0;l<d;l++)C[i*d+l]/=count;
     }
}
```

Listing 2.7: MPI Kmeans - update_centroid

```
void mpi_kmeans(double* D, double* C, int k, int n, int d, int it_max){
    int* assign = new int[n];
    int it = 0;

    //create a type for a data point
    MPI_Datatype point_type;
    MPI_Type_contiguous(d, MPI_FLOAT, &point_type);
    MPI_Type_commit(&point_type);

    // initial centroids
    if(id==0){
        init_centroids(C, k, d);
    }

    MPI_Bcast(C, k, point_type, 0, MPI_COMM_WORLD);

    while(it<it_max){
        assign_to_centroids(D, C, assign, k, n, d);

        // broadcast assign[] to each task
        int s=n/num_p;
        for(int i=0;i<num_p;i++){
            MPI_Bcast(assign+s*i, s, MPI_INT, i, MPI_COMM_WORLD);
        }

        update_centroids(D, C, assign, k, n ,d);

        // broadcast centroids[] to each task
        s = (k>num_p)?k/num_p:1;
        int up = (s==1)?k:num_p;
        for(int i=0;i<up;i++){
            MPI_Bcast(C+s*i*d, s, point_type, i, MPI_COMM_WORLD);
        }

        it++;
    }

    MPI_Type_free(&point_type);
    delete[] assign;
}
```

Listing 2.8: MPI Kmeans

2.3.4 Processor Mapping with CUDA

With CUDA, we are going to allocate a number of blocks for updating a single centroid. Each one of these blocks will loop through a subsection of the data points and add data points that were assigned to the centroid. However we cannot synchronise blocks in CUDA so we need a second kernel for computing the mean. Listings 2.9 and 2.10 show the 2 kernels for update_centroids.

```
__global__ void cuda_centroid_sums(point_t centroids, int* counts,
    point_t data, int* assign, int k, int n, int d){
    int tid = threadIdx.x; int bid = blockIdx.x%STATIC_GSIZE;
    int gid = blockIdx.x/STATIC_GSIZE;

    extern __shared__ element_t s[];

    if(tid>=d)return;

    //reset the stuff in s to 0
    while(tid<d){
        s[tid] =0;
        tid+=BLOCKWIDTH;
    }
    tid = threadIdx.x;
    __syncthreads();

    s[d] =0; //s[d] is the counter

    while(bid<n){
        if(assign[bid]==gid){
            s[d]++;
            while(tid<d){
                s[tid]+=data[bid*d+tid];
                tid+=BLOCKWIDTH;
            }
            __syncthreads();
        }

        tid = threadIdx.x;
        bid+=STATIC_GSIZE;
    }
    tid = threadIdx.x;
    bid=blockIdx.x%STATIC_GSIZE;
    __syncthreads();

    if(tid==0)
        atomicAdd(counts+gid,(int)s[d]);

    while(tid<d ){
        atomicAdd(centroids+(gid*d+tid), s[tid]);
        tid+=BLOCKWIDTH;
    }
}
```

Listing 2.9: Cuda Kmeans - update_centroid (Sums)

```
__global__ void cuda_centroid_means(point_t centroids, int* counts, int
    k, int d){
    int tid = threadIdx.x; int bid = blockIdx.x;

    if(tid>=d)return;

    while(tid<d){
        centroids[bid*d+tid]/=counts[bid];
        tid+=BLOCKWIDTH;
    }
    tid = threadIdx.x;
}
```

Listing 2.10: Cuda Kmeans - update_centroid (final mean)

Chapter 3

Experiments

In this chapter, we present the performance of the different implementations we went over in the previous chapter. In our experiments, we ran our implementations on the mscluster which is equipped with Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz processors. We also made use of an Nvidia GTX 1060 graphics card to run our parallel code.

3.1 Experiment set 1

The first set of experiments are to gauge the performance of our implementations with an increase in the number of data points. We ran our kmeans implementations with by k=16, d=16 and the n_it=50. Figure 3.1 shows graphs showing the performance of our various implementation. The first graph shows the runtime vs number of data points plot of the assign_to_centroids subroutine. As expected, all versions have a linear complexity in terms of the number of data points. However, we can see that the number of centroids and max iterations result in a large constant that results in a steep growth in the serial implementation. The CUDA version and MPI versions (both with 8 and 16) run at a fraction of the serial version. The CUDA version is able to run a little faster than the MPI implementations.

The second graph shows the runtime of the update_centroids. The MPI fares much better than the CUDA version. In the total performance of the algorithm, We see that the MPI implementations are a bit faster than the serial implementation. Our expectations where that the CUDA implementation would result in faster performance than MPI due to the number of available processors. However, the use of global memory has reduced the performance of the CUDA implementation.

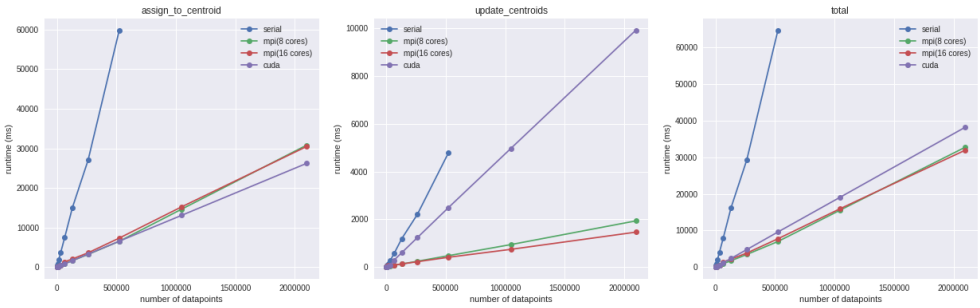

Figure 3.1: Running the implementations with change in the number of data points

3.2 Experiment set 2

The second set of experiments are to gauge the performance of our implementations with the increase in dimension. We ran our implementations on a fixed data set with 1024 data points and 16 centroids. Figure 3.2 shows the performances of the assign_to_centroids, update_centroids and the algorithm as a whole. We see that the CUDA implementation is much better than the other 2 implementations. This is because, we use threads in a block to to deal with computations involving the dimensions.

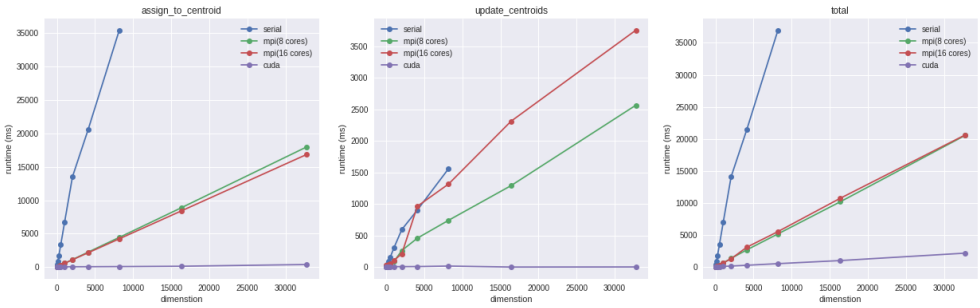

Figure 3.2: Running the implementations with change in the dimension

Chapter 4

Conclusion

In this project, we sought out to implement Kmeans clustering using MPI and CUDA to get performance improvements. We saw that MPI was able to provide better performance when the data set became larger and CUDA did a good job with large dimensions. More performance can be sought out for in the feature by exploring the different memory types CUDA provides, as well as better memory access patterns. With regard to MPI, more performance can be sought out by using more processors to parallelize the tasks in each of the agglomerations.

References

[Cunningham and Ghahramani 2015] John P Cunningham and Zoubin Ghahramani. Linear dimensionality reduction: Survey, insights, and generalizations. *The Journal of Machine Learning Research*, 16(1):2859–2900, 2015.

[Farivar *et al.* 2008] Reza Farivar, Daniel Rebolledo, Ellick Chan, and Roy H Campbell. A parallel implementation of k-means clustering on gpus. In *Pdpta*, volume 13, pages 212–312, 2008.

[Hong-tao *et al.* 2009] B. Hong-tao, H. Li-li, O. Dan-tong, L. Zhan-shan, and L. He. K-means on commodity gpus with cuda. In *2009 WRI World Congress on Computer Science and Information Engineering*, volume 3, pages 651–655, 2009.

[Jain 2010] Anil K Jain. Data clustering: 50 years beyond k-means. *Pattern recognition letters*, 31(8):651–666, 2010.

[Kanungo *et al.* 2002] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu. An efficient k-means clustering algorithm: analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(7):881–892, 2002.

[Yi-qing and Jin-xian 2011] Lü Yi-qing and LIN Jin-xian. Parallel pso combined with k-means clustering algorithm based on mpi [j]. *Journal of computer applications*, 2:041, 2011.

[Zhang *et al.* 2011] J. Zhang, G. Wu, X. Hu, S. Li, and S. Hao. A parallel k-means clustering algorithm with mpi. In *2011 Fourth International Symposium on Parallel Architectures, Algorithms and Programming*, pages 60–64, 2011.