TACC Technical Report IMP-12

K-means clustering in the Integrative Model

Victor Eijkhout*

January 16, 2017

* eijkhout@tacc.utexas.edu, Texas Advanced Computing Center, The University of Texas at Austin

Abstract

We consider the k-means clustering algorithm.

The following IMP reports are available or under construction:

IMP-00 The IMP Elevator Pitch
IMP-01 IMP Distribution Theory
IMP-02 The deep theory of the Integrative Model
IMP-03 The type system of the Integrative Model
IMP-04 Task execution in the Integrative Model
IMP-05 Processors in the Integrative Model
IMP-06 Definition of a 'communication avoiding' compiler in the Integrative Model (under construction)
IMP-07 Associative messsaging in the Integrative Model (under construction)
IMP-08 Resilience in the Integrative Model (under construction)
IMP-09 Tree codes in the Integrative Model
IMP-10 Thoughts on models for parallelism
IMP-11 A gentle introduction to the Integrative Model for Parallelism
IMP-12 K-means clustering in the Integrative Model
IMP-13 Sparse Operations in the Integrative Model for Parallelism
IMP-14 1.5D All-pairs Methods in the Integrative Model for Parallelism (under construction)
IMP-15 Collectives in the Integrative Model for Parallelism
IMP-16 Processor-local code (under construction)
IMP-17 The CG method in the Integrative Model for Parallelism (under construction)
IMP-18 A tutorial introduction to IMP software (under construction)
IMP-19 Report on NSF EAGER 1451204.
IMP-20 A mathematical formalization of data parallel operations
IMP-21 Adaptive mesh refinement (under construction)
IMP-22 Implementing LULESH in IMP (under construction)
IMP-23 Distributed computing theory in IMP (under construction)
IMP-24 IMP as a vehicle for software/hardware co-design, with John McCalpin (under construction)
IMP-25 Dense linear algebra in IMP (under construction)

1 Introduction

1.1 Standard algorithm

Gleaned from HPC challenge [1].

The abstract algorithm iterates the following calculation:

```
for point i<N:
  for cluster k<C:
    compute distance point i to center k
  kmin is minimum distance
  cluster_i = kmin
for cluster k<C:
  let G_k the group of all i that have k as cluster number
  center_k is sum coordinates over cluster_k / |G_k|
```

Let N be the number of points and K the number of clusters, then the data structures are:

- $X(N)$ are coordinates
- $C(K)$ are cluster center coordinates,
- $d(N,K)$ are distances to cluster centers
- $g(N)$ is the group assignment.

The algorithm then consist of

- Distance computation, which is a outer product

  $$d(i,k) \leftarrow f(X(i),C(k))$$

  and therefore $N \times K$ way parallel.
- Group assignment, parallel over the points:

  $$g(i) \leftarrow \operatorname*{argmin}_k g(d(i,k))$$

  with a reduction over the K dimension.
- Cluster formation, parallel over the clusters:

  $$C(k) \leftarrow \operatorname*{reduce}_i h(d(i,k),g(i))$$

  with a reduction over the N dimension.

Here f,g,h are scalar functions with obvious but irrelevant definitions.

Let u be a distribution of N over P and let v be a distribution of K over P. Then the general calculation becomes:

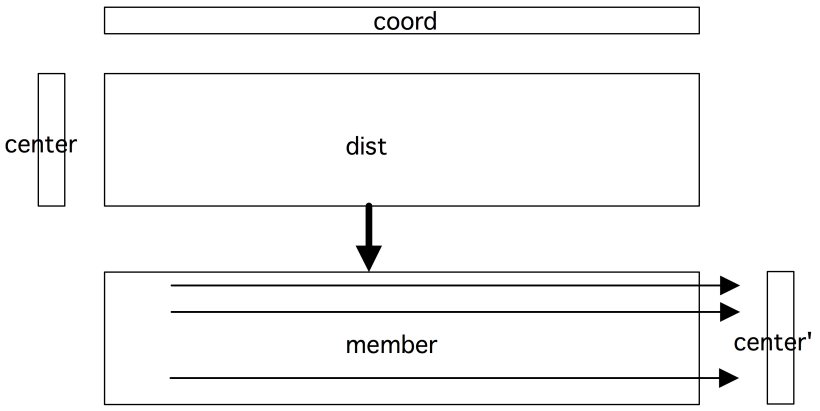

Figure 1: Illustration of the distributions in a kmeans algorithm

$$
\begin{aligned}
d(u,v) &= X(u)\cdot C(v) && \text{parallel pointwise mult from redundant } C\\
g(u) &= \operatorname{argmin}_2 d(u,\star) && \text{parallel reduction}\\
G(u,v) &= (i,k)\mapsto \delta(g(i)=k) && \text{definition of group mask}\\
C(v) &= \operatorname{reduce}_i(X(i),G(i,k)) && \text{new center from masked computation}
\end{aligned}
$$

If the number of clusters is small compared to the number of points or the number of processors, the easiest parallelization strategy is to replicate the k dimension and only divide the N dimension.

$$
\begin{aligned}
d(u,\star) &= X(u)\cdot C(\star) & d(u,\star) &= X(u)\cdot C(\star)\\
g(u) &= \operatorname{argmin}_2 d(u,\star) & g(u,\star) &= \text{maskarray } \operatorname{argmin}_2 d(u,\star)\\
\forall_{k<K}\colon G_k &= \{i\colon g(i)=k\} & &\\
\forall_{k<K}\colon \{P\}\colon c_k &= M\big(G_k\colon X(u)\big) & c_k(\star) &= M\big(g(u,\star)\otimes X(u,\star)\big)
\end{aligned}
$$

1.2 Gonzalez algorithm

See [2].

```
centers <- points
while #centers too high:
  partition centers
  compute k centers in each partition
  centers <- union of all the partition centers
```

See [3] for analysis and performance of Gonzalez and other parallel methods. They use MapReduce.

2 Implementation

We have a global size N, space dimension d, and a number of centers k. The centers are stored redundantly:

```
%% template_kmeans.cxx
IMP_distribution
  *kreplicated = new IMP_replicated_distribution(decomp,dim,ncluster);
IMP_object
  *centers = new IMP_object( kreplicated );
```

Setting the initial centers is data parallel:

```
set_initial_centers( centers );
```

On the other hand, the coordinates are a properly distributed data set of size $N \times d$, where d is the space dimension, typically 2 or 3:

```
%% unittest_kmeans.cxx
mpi_distribution
  *twoblocked = new mpi_block_distribution
      (decomp,dim,-1,globalsize);
mpi_object
  *coordinates = new mpi_object( twoblocked );
```

2.1 Origin kernel

We generate the coordinates in an origin kernel:

```
%% unittest_kmeans.cxx
mpi_kernel
  *set_random_coordinates = new mpi_kernel( coordinates );
set_random_coordinates->localexecutefn = &generate_random_coordinates;
```

2.2 Distance to clusters

Calculate the distances to each of the k coordinates, given an $N \times dk$ array is an outer product calculation. Here we distribute the cross dimension redundantly:

```
%% unittest_kmeans.cxx
mpi_distribution
  *kblocked = new mpi_block_distribution
      (decomp,ncluster,-1,globalsize);
mpi_object
  *distances = new mpi_object( kblocked );
```

The distance calculation has two input object, both with $\beta = \gamma$:

```
%% unittest_kmeans.cxx
calculate_distances = new mpi_kernel( coordinates,distances );
calculate_distances->last_dependency()->set_explicit_beta_distribution(coordinates);
calculate_distances->add_in_object( centers);
```

```
calculate_distances->last_dependency()->set_explicit_beta_distribution( centers );
calculate_distances->set_localexecutefn( &distance_calculation);
```

We summarize this as a derived kernel class:

```
%% unittest_kmeans.cxx
printf("the outer product kernel still uses the context. Wrong!\n");
calculate_distances = new mpi_outerproduct_kernel
( coordinates,distances,centers,&distance_calculation );
```

```
%% mpi_ops.h
mpi_outerproduct_kernel( object *in,object *out,
    object *replicated,
    void (*f)(int, processor_coordinate*, std::vector<object*>*, object*, double*)
  )
  : mpi_kernel(in,out),entity(entity_cookie::KERNEL) {
  last_dependency()->set_explicit_beta_distribution(in);
  add_in_object(replicated);
  last_dependency()->set_explicit_beta_distribution(replicated);

  set_name(fmt::format("outer product{}",get_out_object()->get_object_number()));
  set_localexecutefn(f);
};
```

The definition of the outer product kernel (and other collectives) is further discussed in [**?**].

2.3 Find nearest center

With the distances, we can compute the nearest center, an array of length N, in a local operation. The nearest center is an integer value, but we store it as real.

```
%% unittest_kmeans.cxx
mpi_distribution
  *blocked = new mpi_block_distribution
      (decomp,-1,globalsize);
mpi_object
  *grouping = new mpi_object( blocked );
```

The computation is a parallel map, for which for now we use the cross product kernel with null cross dimension:

```
%% unittest_kmeans.cxx
find_nearest_center = new mpi_outerproduct_kernel
( distances,grouping,NULL,group_calculation);
```

Eh,

```
%% unittest_kmeans.cxx
mpi_object
  *min_distance = new mpi_object( blocked );
```

Updating the centers takes two kernels. First we locally construct an array of size $N \times dk$ of coordinates, where only the κ-th coordinate is filled in, where κ is the number of the nearest center.

```
%% unittest_kmeans.cxx
mpi_distribution
  *k2blocked = new mpi_block_distribution
      (decomp,ncluster*dim,-1,globalsize);
mpi_object
  *masked_coordinates = new mpi_object( k2blocked );
%% unittest_kmeans.cxx
group_coordinates = new mpi_outerproduct_kernel
( coordinates,masked_coordinates,grouping,coordinate_masking );
```

This is yet another outer product kernel, with the grouping passed as context.

Next the new centers are computed by a reduction:

```
%% unittest_kmeans.cxx
mpi_distribution
  *klocal = new mpi_block_distribution(decomp,ncluster*dim,1,-1);
mpi_object
  *partial_sums = new mpi_object( klocal );
mpi_kernel
  *compute_new_centers1 = new mpi_kernel( masked_coordinates,partial_sums );
%% unittest_kmeans.cxx
compute_new_centers1->set_name("partial sum calculation");
compute_new_centers1->set_localexecutefn( &center_calculation_partial );
compute_new_centers1->last_dependency()->set_type_local();
```

See the theory of gather in section **??**.

3 Discussion

References

[1] 2010 ibm hpc challenge class ii submission.

[2] T. F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.

[3] J. McClintock and A. Wirth. Efficient Parallel Algorithms for k-Center Clustering. *ArXiv e-prints*, April 2016.