Memo for GlobalReductions Study

School of Informatics and Computing, Pervasive Technology Institute
Indiana University Bloomington

***Abstract*— Data-parallel primitives such as parallel reduction, sort, or scan are important building block for many data parallel applications such as Kmeans. The performance of these data parallel primitives can affect overall performance of applications if they take large proportion of the processing process. This paper studies the performance effect of global reductions primitives on Kmeans uses different parallel runtime tools including Hadoop, MPI, OpenMP, and Cuda.**

Keyword: GlobalReduction, Kmeans

I. INTRODUCTION

Data-parallel primitives such as parallel reduction, sort, or scan are important building block for many data parallel applications such as Kmeans. The performance of these data parallel primitives can affect overall performance of applications if they take large proportion of the processing process. This paper studies the performance effect of global reductions primitives on Kmeans.

In data mining, K-Means clustering is a method of cluster analysis which aims to partition N observations with D dimensions into K clusters in which each observation belongs to the cluster with the nearest mean. In figure1, it shows the Kmeans algorithm consists of three main steps: 1) compute distance, 2) find the closest centroids, 3) compute new centroids. The step 1) and step 2) of Kmeans algorithm is of pleasingly parallel feature and can be parallelized among nodes. Step 3) can be parallelized, but it needs a global reduction step among nodes.

Some Kmeans implementations use sequential code to run step 3), but this is not a scalable behavior as step 3) can take large proportion of whole process when more number of nodes or accelerators was involved in the computation. However, parallelize step 3) is not as easy as that in step 1) and step 2) because it requires a global reduction step among nodes during each iteration. The global reduction increases the complexity of programming effort, as the programmers need handle the parallelism among MPI and Pthreads, OpenMP, or CUDA; and they also need to handle the data movement among different memory hierarchy, such as copy data between CPU and GPU memory. Figure 2 illustrates the workflow of global reduction uses MPI and Hadoop. To solve the performance and programmability issue, we propose the GlobalReduction interface that can hide the implementation details of parallelism among MPI, Hadoop, OpenMP, Pthread and CUDA.

```
Do {
    OldDelta  = Delta;
    Delta = 0;
    For j=1 to k
            Mj = 0; Nj =0;
    Endfor;
    For i=1 to n
            For j=1 to k
            Compute squared Euclidean
                    Distance d^2(Xi,Mj);
            Endfor;
    Find the closest centroid Mj to Xi;
    Mj = Mj+Xi;
    Nj= Nj+1;
    If(Mj!=Mj') Delta+=1;
    For j= 1 to k
            Nj = max(Nj,1); Mj= Mj/Nj;
    Endfor;
    }while (Delta<OldDelta)
```
Figure 1, Sequential K-means Algorithm

Computation complexity
Tseq = (3NKD+NK+ND+KD)*Tflops
Tseq ~ 3NKD if N is much larger than K and D

```
Do{

OldDelta=Delta;
Delta=0;
For j=1to k
    Mj=0;Nj=0;
Endfor;

For i= u*(N/P)  to (u+1)*(N/P)
    For j = 1 to k
            Compute squared Euclidean
                    Distance d^2(Xi,Mj);
    Endfor;
    Find the cloest centroid Mj to Xi;
    Mj=Mj+Xi; Nj=Nj+1;
    If(Mj!=Mj') Delta+=1;
Endfor;

For j= 1 to #threads
    For l=1 to  K
            SumUp(Nj);Sumup(Mj);

MPI_Allreduce(N,N,MPI_SUM);
MPI_Allreduce(M,M,MPI_SUM);
```
Figure 2, Parallel K-means Algorithm

Computation complexity:
Tparallel = (3NKD)*Tflop/P + DK*Tflops

Speedup = Tseq/Tparallel =
{(3NKD)*Tflops}/{(3NKD)*Tflop/P+DK*Tflops}

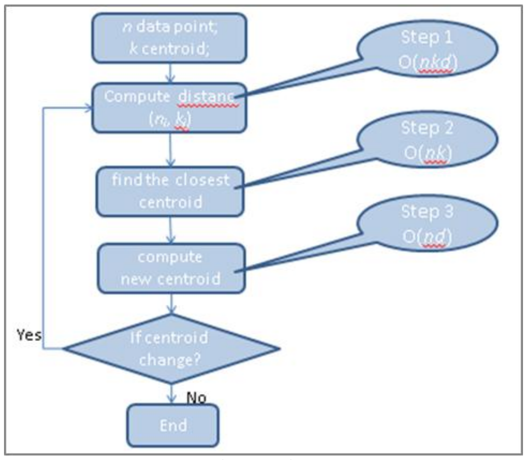


Figure 1: workflow of Kmeans algorithm

II. KMEANS IMPLEMENTATION

We implemented Kmeans with MPI and Hadoop. In MPI implementation, as shown in figure 2, MPI was used to run global reduction among nodes, CUDA and OpenMP were used to run distance computation, membership computation, and local reduction on single machine. The Kmeans CUDA code uses three CUDA Kernels – Distance Matrix, Membership value, and Local Reduction in order to increase the parallelism of SIMT code. The distance matrix calculates a (M*K) matrix that contains the Euclidean distance from each data point to every cluster center using MX[N/512] kernel grid: step 1). And it assigns the values for membership values for each data point: step 2). The kernel local reduction computes the new cluster size for step 3) using a D*[K/4] kernel grid. Using D*M grid exposes more parallelism, but it needs more memory accesses which lead to low memory bandwidth utilization. Instead, we unroll 4 subsequent clusters so as to increase the memory bandwidth during the computation.

In Hadoop implementation, as shown in figure 2, the task trackers invoke the Java, Cpp, and CUDA wrapper to run map tasks that run distance computation and local reduction on local machine with corresponding binary code. The child JVM of map tasks were sponsored by Hadoop task tracker, and it communicate with Cuda or Cpp binary code through pipe. The global reduction was performed in reduce stage. One should note that, the data were loaded from disk to memory in each iteration for Java and Cpp implementation. And it takes extra data movement between CPU memory and GPU memory in each iteration for Cuda implementation. The centroids data were loaded from memory cache utility of Hadoop. After Map stage, all intermediate data were shuffled to one reduce task to perform global reduction using corresponding code.

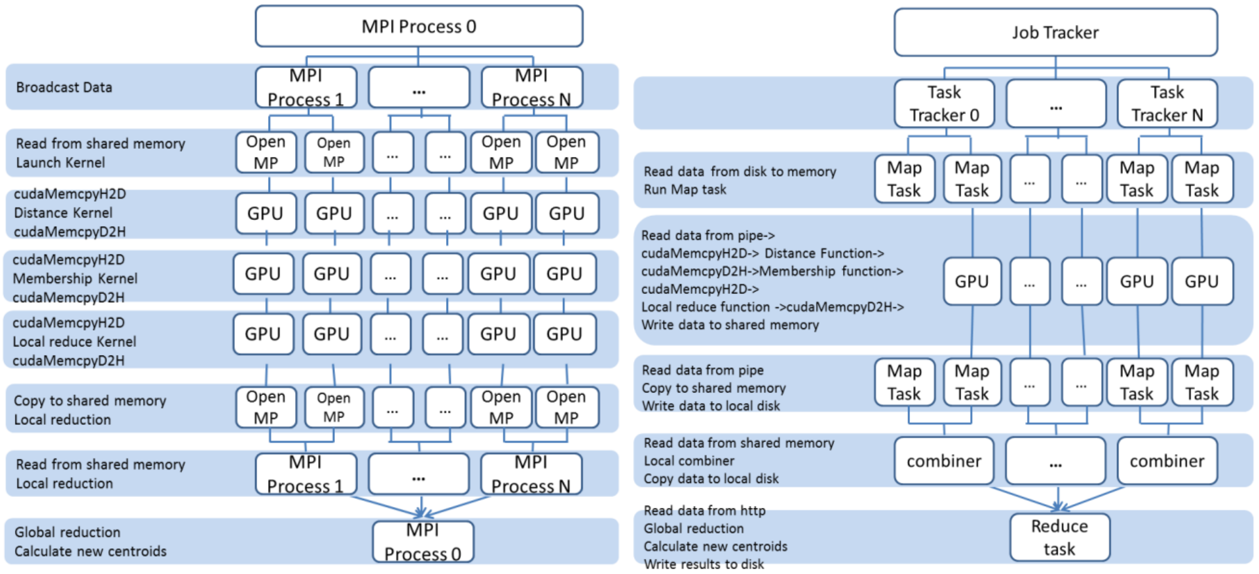
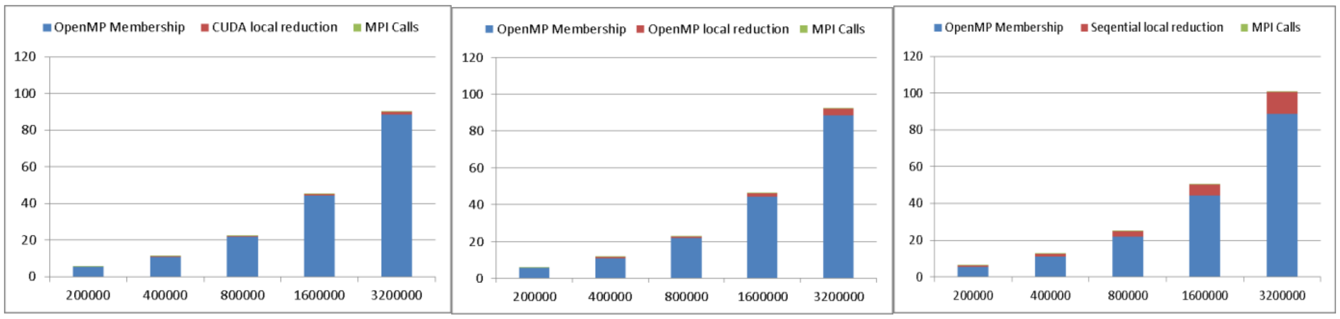
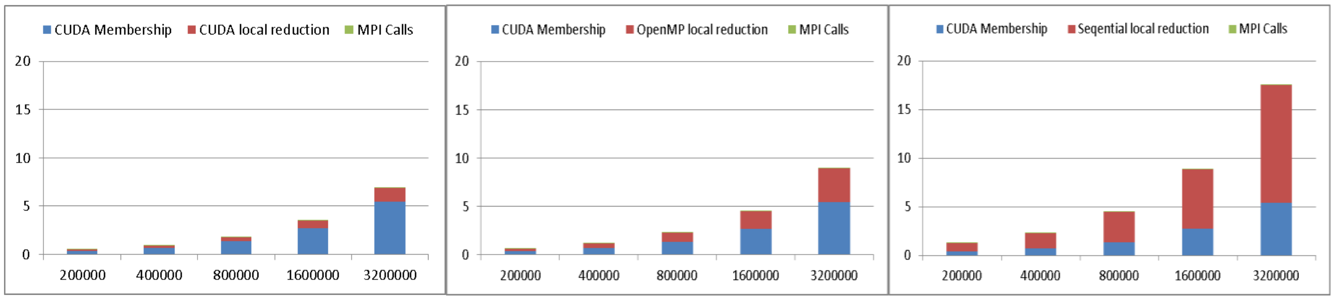


Figure 2: workflow of the global reduction uses MPI-Cuda and Hadoop-Cuda



(a) (b) (c)

Figure 3: Overhead component of Kmeans with three local reduction approaches with OpenMP to run membership computation. (a) uses CUDA to run local reduction. (b) uses OpenMP to run local reduction. (c) uses sequential code (openmp uses 1core) to run local reduction.



(a) (b) (c)

Figure 4: Overhead component of Kmeans with three local reduction approaches with CUDA to run membership computation. (a) uses CUDA to run local reduction. (b) uses OpenMP to run local reduction. (c) uses sequential code (openmp uses 1 core) to run local reduction.

III. EXPERIMENTS RESULTS

We parallelize the Kmeans using MPI/Hadoop/OpenMP/CUDA and evaluate its performance on four compute nodes on Delta cluster on FutureGrid. The experiments use faked input data for Kmeans with 200K to 3.2 million objects, 100 clusters, and 64 dimensions for each object. Figure 3 shows the performance results of Kmeans with three different local reduction approaches for step 3) including: sequential, openmp, and cuda. The step 1) and step2) use the openmp to run computation in parallel. The results indicate that use CUDA and OpenMP for step 3) can increase the overall performance by 10.3% and 9.1% respective as compared to using sequential version for step 3).

Figure 4 shows the performance results of Kmeans with the same three different local reduction approaches for step 3), where the difference is that it uses CUDA to run he computation for step 1). The results show that using CUDA and OpenMP for step 3) can increase the overall performance by 61.5% and 48.3% respective as compared to using sequential version for step 3).

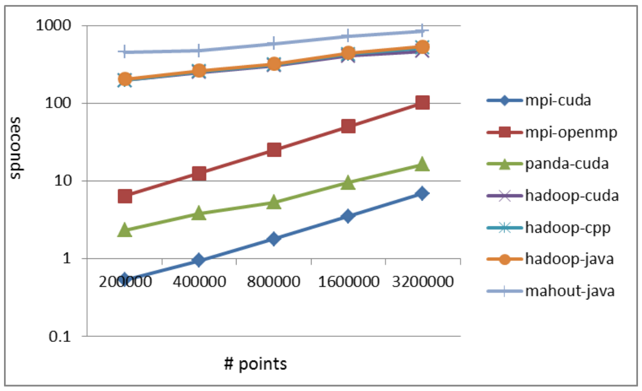


Figure 5: performance of Kmeans with different runtime technologies.

We also evaluate Kmeans application with different runtime technologies including mpi, hadoop and mahout on four nodes on Delta cluster. The results indicate that mpi-cuda implementation can give a speedup of 14 over mpi-openmp for large data sets. And hadoop-cuda is 1.15x and 1.04x faster than hadoop-openmp and hadoop-java respectively. The hadoop-cuda didn't have much performance improvement because it has to load data from disk to memory and then to gpu device memory during each iterations, while the mpi implementation can cache the static data in device memory during each iterations. The results also showed that the standard implementation mahout is 1.76x slower than our hadoop implementation. This is because our Hadoop implementation uses much coarse granularity task, and it can get performance improvement by leveraging the local reduction, while mahout implementation uses much finer granularity for each map task, which trigger larger communication overhead during shuffle stage. The results also indicate that panda-cuda implementation is 132.13 times faster than Mahout, but is 2.37 times slower than mpi-cuda implementation.

Figure 6 illustrates the overhead of map, local reduce, shuffle stages of kmeans jobs using Hadoop and Mahout. The figure indicates that overhead of Map stage take a larger proportion in our Hadoop implementation than that in Mahout implementation, which means using local reduction can increase the parallel part of the computation.

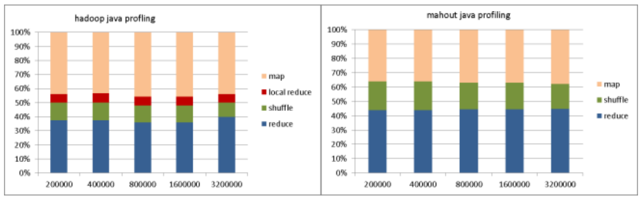


Figure 6: profiling of overhead of map, local reduce, shuffle, and reduce stages of kmeans job uses Hadoop and Mahout.