Distributed GPU-Based K-Means Algorithm for Data-Intensive Applications: Large-Sized Image Segmentation Case

Hicham Fakhi*, Omar Bouattane, Mohamed Youssfi, Hassan Ouajji
Signals, Distributed Systems and Artificial Intelligence Laboratory
ENSET, University Hassan II,
Casablanca, Morocco

***Abstract*—K-means is a compute-intensive iterative algorithm. Its use in a complex scenario is cumbersome, specifically in data-intensive applications. In order to accelerate the K-means running time for data-intensive application, such as large sized image segmentation, we use a distributed multi-agent system accelerated by GPUs. In this K-means version, the input image data are divided into subsets of image data which can be performed independently on GPUs. In each GPU, we offloaded the data assignment and the K-centroids recalculation steps of the K-means algorithm for a massively parallel processing. We have implemented this K-means version on the Nvidia GPU with Compute Unified Device Architecture. The distributed multi-agent system was written with Java Agent Development framework.**

Keywords—Distributed computing; GPU computing; K-means; image segmentation

I. Introduction

In our decade, a huge amount of data must be processed continuously by computers to meet the needs of the end users in many business areas. By a simple search on Google, we found lot of official statistics that show how big the big data processed in image processing is, in web semantics, data storage, profiling and other scientific fields used by Google and Facebook. For example, Facebook stores 300 petabytes, and processes 600 terabytes per days. It deals with 1 billion users per month, and finally 300 million photos are uploaded per day. In addition, Google stores much more than Facebook. Google stores 15 exabytes; it processes 100 petabytes per day; it indexes 60 trillion pages and performs 2.3 million searches per second. In brief, the data to be processed in many application areas become more than ever increasingly large.

In this paper, we focus on image processing and their applications. Understanding images and extracting information from them so that the information can be used for other tasks is an important aspect, as for example cancer detection in Magnetic Resonance Imaging (MRI). Such analyses and extraction of useful information from images are ensured by image processing techniques such as image segmentation [6], [7] which is one of the clustering problems. The K-means is an unsupervised learning algorithm that solves the clustering problem. It is an iterative algorithm. Each iteration consists of two steps, the assignment of data objects and K centroids recalculation.

Nonetheless, there are two important factors to consider when doing image segmentation. First is the number of images to be processed in a given use case. Second is the image quality which has known an important evolution during the last few years, i.e., the number of pixels that make up an image has been multiplied by 200 from the 720 x 480 pixels to 9600 x 7200 pixels. This has resulted in a much better definition of the image (more detail visibility) and more nuances in the colors and shades.

Thus, during last decade, image processing techniques have become cumbersome in computing time for monolithic computers due to the huge number of pixels. This obvious need has led naturally to more powerful computers to allow image processing researchers to use new High-Performance Computing (HPC) strategies based on the parallelism and distributed approaches such as 2D or 3D reconfigurable mesh [10], FPGA, and recently GPU [8], [9] and Hadoop.

In GPU computing, the most important advance is the Nvidia CUDA (Compute Unified Device Architecture) solution. The Nvidia TITAN X is the fastest GPU at the time of this writing. This GPU has 3584 shader units also called CUDA cores or elementary processors. It has 1417 MHz as base clock which can be boosted to 1531 MHz, and 12 Gbits of GDDR5 memory with 480 Gbits/s of memory bandwidth. To have more computational power, four TITAN X GPUs can be interconnected with Nvidia’s Scalable Link Interface (4-way SLI), the result being a powerful GPU with 14336 CUDA cores and 48 Gbits of GDDR5 memory which in collaboration with the Intel Core i7 5960X CPU can give an interesting optimization not only for image processing but also for many other domains of applications. Unfortunately, in some cases, the use of multi-GPU systems is not sufficient to obtain a high-enough performance computing for certain scientific or engineering applications. In the case where these applications have to process a large amount of data and perform complex tasks, as for instance in medical imaging to perform an analysis on large-sized MRI cerebral images using image-processing techniques such as the K-means clustering algorithm. In addition, the scalability is not guaranteed and strongly depends on the evolution of GPU and CPU hardware proposed by Nvidia, AMD and Intel. Thus, using a multi-GPU system on a single node is constrained by hardware limitations. In other words, the computing and data communications capabilities of the processing environment become the dominating bottleneck.

To overcome these limitations, we have studied distributed programming libraries with the objective of combining GPU computing and distributed computing paradigms.

In the distributed computing paradigm, we found a set of distributed programming libraries and standards, as for instance MPI (Message Passing Interface) [13], [14], OpenMP (Open Multi-Processing) [15], [16], or HPX [19], Hadoop [20]. The idea of distributed computing is to combine machines, which is typically commodity hardware, that can be used to parallelize tasks, as for example the libraries and standards cited above that were used in more than scientific domains [11], [12], [17], [18]. But the limitation of the distributed system lies in the fact that these machines are limited in computing power (number of processors in each machine) and the data storage capacity. The scalability of such as system is slow and expensive. To improve the computation power of such a distributed system, we have to connect new machines. For example, to have 384 more processors in the system, we must connect 48 machine octa processors.

Additionally, in the distributed computing paradigm, all researchers agree that the challenge is to find a library or a framework which provides ease of programming with a high-level programming language (without memory management or other low-level programming routines) and the best performance exploitation of hardware. Unfortunately, these two goals are contradictory due to the fact that some researchers obtained best performance by using low-level communication libraries known to be error-prone like MPI (Message Passing Interface) [21,24] or OpenMP (Open Multi-Processing) [22], [23], Other researchers [25], [26] have used libraries and frameworks with a high-level programming language which ensures simplicity of programming and portability of the code, although bringing a loss of performance and preventing an efficient access to CPU and GPU due to high-level abstractions of the hardware.

To tackle these problems, we have used a distributed Multi-Agent System (MAS) on GPU-accelerated nodes to accelerate the large-sized image segmentation using the K-means algorithm. The MAS distributed on connected nodes is used to divide the data into a subset of dispatched data through accelerated compute nodes with the GPU. Each subset of data will be processed separately in a node using GPUs. In this version, we used CUDA C/C++ to write the K-means kernel code that will be executed on the GPUs. On the other hand, the multi-agent system was programmed using the JADE platform which is based on Java.

This paper presents the role of the MAS on the data and task distribution between remote GPUs across interconnected nodes during the K-means execution and will show the experimental results.

II. Literature Review

In the literature, researchers have shown a special interest to improving the K-means algorithm, by adopting K-means for parallel and distributed platforms such as GPU CUDA [4], [5], [28]-[30], OpenCL (Open Computing Language), MPI, OpenMP (Open Multi-Processor) [3], Hadoop [2] and JADE [31]. From our experience, GPU CUDA and Hadoop implementations are by far the most efficient.

Poteras et al. [27] focused on optimization of the data assignment step of the K-means algorithm. The idea is that for each iteration before the data assignment step, they add a procedure that determines which of the data objects could be affected by a move. Thus, they no longer need to visit all the data objects to define their membership, but just a small list of data objects.

Fang et al. [4] propose a GPU-based implementation of K-means. This version copies all the data to the texture memory, which uses a cache mechanism. Then it uses constant memory to store the K-centroids, which is also more efficient than using global memory. Each thread is responsible for finding the nearest centroid of a data point; each block has 256 threads, and the grid has n/256 blocks.

The workflow of [4] is straightforward. First, each thread calculates the distance from one corresponding data point to every centroid and finds the minimum distance and corresponding centroid. Second, each block calculates a temporary centroid set based on a subset of data points, and each thread calculates one dimension of the temporary centroid. Third, the temporal centroid sets are copied from GPU to CPU, and then the final new centroid set is calculated on CPU.

In [4] each data point is assigned to one thread and utilizes the cache mechanism to get a high reading efficiency. However, the efficiency could be further improved by other memory access mechanisms such as registers and shared memory.

Che et al. [5] present another optimized K-means implementation of GPU-based K-means in a single node. They store all input data in the global memory, and load k-centroids to the shared memory. Each block has 128 threads, and the grid has n/128 blocks. The main characteristic of [5] is the design of a bitmap. The workflow of [5] is as follows. First, each thread calculates the distance from one data point to every centroid, and changes the suitable bit into true bit in the bit array, which stores the nearest centroid for each data point. Second, each thread is responsible for one centroid, finds all the corresponding data points from the bitmap and takes the mean of those data points as the new centroids. The main problem of [5] is the poor utilization of GPU memory, since [5] accesses most of the data (input data points) directly from the global memory.

Mao et al. [2] present a distributed implementation of the K-means using Hadoop. This research work deals with a data-intensive clustering application. A virtual Hadoop cluster based on cloud computing with CloudStack was established with the aim to implement the distributed K-Means clustering algorithm based on the MapReduce pattern. The initial centroid selection and number of iterations was optimized. The initial centroid selection was improved using the furthest first (FF) algorithm to select the next farthest point. To improve the iteration time, they use the result of a previous iteration for the next iteration of the centroid point in the Map calculation.

The article of Baydoun et al. [3] proposes two improved versions of the Kernel K-means on CPU and GPU. The CPU version was based on OpenMP, Cilk Plus and BLAS Libraries. The GPU version was based the Nvidia CUDA. These versions of Kernel K-Means utilize the Kernelization approach [1] to divide given data into a set of clusters using an approach mainly based on K-Means.

III. Image Segmentation using the K-Means Clustering Algorithm

The K-Means is a clustering algorithm that classifies the input data points S of n attribute vectors into c classes (clusters C_i, i = 1…c) based on their inherent distances from each other. The algorithm assumes that the data features form a vector space and tries to find natural clustering among them. The points are clustered around class centers (centroids) which are obtained by minimizing the objective function:

$$J = \sum_{i=1}^{c} J_i = \sum_{i=1}^{c} \sum_{k,x_k \in C_i} d(x_k - c_i) \tag{1}$$

Where c_i is the centroid of the ith class, and $d(x_k - c_i)$ is the distance between ith center $c_i$and the kth data of S. We use the Euclidean distance to define the objective function as follows:

$$J = \sum_{i=1}^{c} J_i = \sum_{i=1}^{c} \sum_{k,x_k \in C_i} ||x_k - c_i||^2 \tag{2}$$

As described in MacQueen's paper [32], an initial clustering C_i; (i = 1… c) is created by choosing c random centroids from the set of n data points S. This is known as centroids initialization. Next, an assignment step is executed where each data point $x_j \in$ S (j = 1… n) is assigned to the cluster C_i for which $d(x_j, c_i)$ is minimal. Each centroid c_i is then recalculated by the mean of all data points $x_j \in C_i$. The assignment and K-centroid recalculations steps are executed repeatedly until C_i no longer changes. This algorithm is known to converge to a local minimum subject to the initial centroids. In our application, the clustering K-means algorithms is used for the image segmentation. Thus, the flow chart of the algorithm in the Fig. 1 takes a 2-dimensional image as data in input, each point (pixel) of this image having an intensity.

The K-means algorithm can be directly implemented on CPU using several "**for**" loops embedded in one "**while**" loop with the aim to be performed on a CPU. In "**for**" loops calculations of distances between each pixel and the centroids is performed. Next, recalculation of the new K-centroids for the next iteration of the "**while**" loop is also done in a "**for**" loop. The "**while**" loop condition for another iteration is inequality between the centroids intensities from a previous iteration and the next iteration. If the centroids do not change, the loop is broken, and the algorithm stops.

In brief, K-means chooses the centroids intelligently and it compares centroids with the data points based on the intensities and characteristics and finds the distances. The data points which are similar to the centroid are assigned to the cluster having that centroid. New c_i centroids are calculated and thus K-clusters are formed by finding out the data points nearest to the clusters.

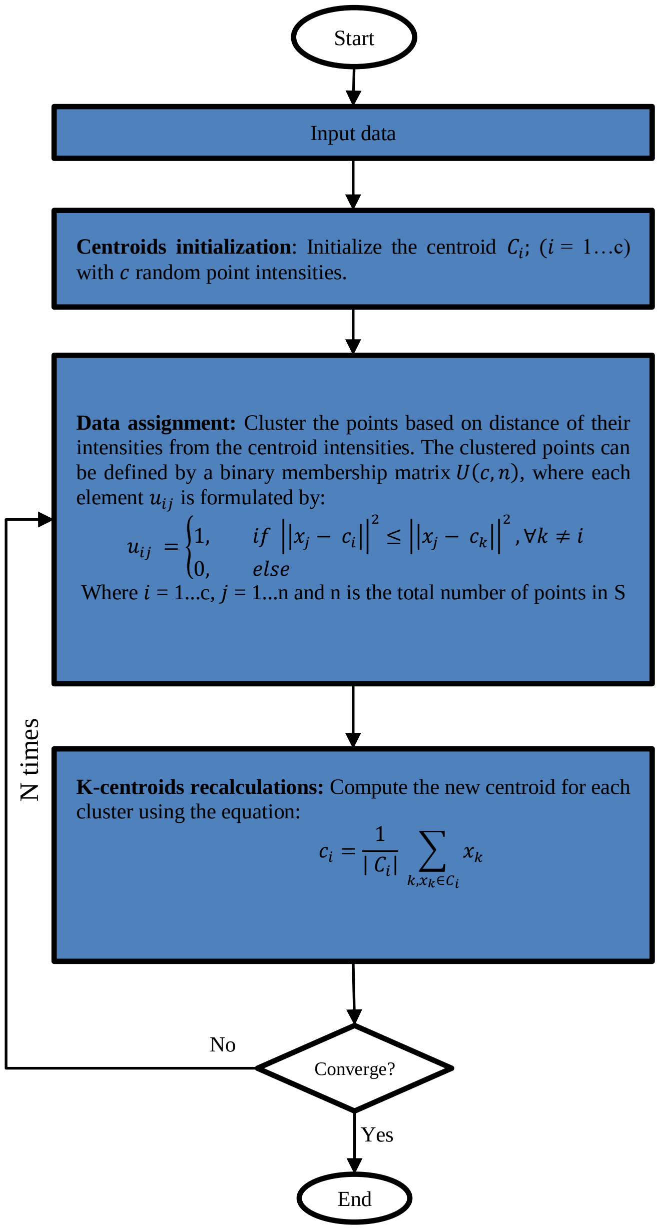
Start
Input data
Centroids initialization: Initialize the centroid C_i; (i = 1…c) with c random point intensities.
Data assignment: Cluster the points based on distance of their intensities from the centroid intensities. The clustered points can be defined by a binary membership matrix $U(c,n)$, where each element u_{ij} is formulated by:
$u_{ij} = \begin{cases} 1, & if\ ||x_j - c_i||^2 \le ||x_j - c_k||^2, \forall k \neq i \\ 0, & else \end{cases}$
Where i = 1...c, j = 1...n and n is the total number of points in S
K-centroids recalculations: Compute the new centroid for each cluster using the equation:
$c_i = \frac{1}{|C_i|} \sum_{k,x_k \in C_i} x_k$
N times
No
Converge?
Yes
End

Fig. 1. K-means flow chart.

IV. Proposed K-Means Version

A. Runtime Environment

The data distribution is based on the agent interactions within MAS deployed on multiple nodes. The MAS used was implemented by JADE [33] in accordance with the standards of the Foundation for Intelligent Physical Agents. The interactions between the agents are based on asynchronous communication mechanisms in accordance with the ACL.

Each running instance of the JADE runtime environment is called a container as it can contain several agents. The JADE platform is a set of active containers distributed on nodes. JADE agents are identified by a unique name and, provided they know each other's name, they can communicate transparently regardless of their actual location in the same container or different containers in the same platform.

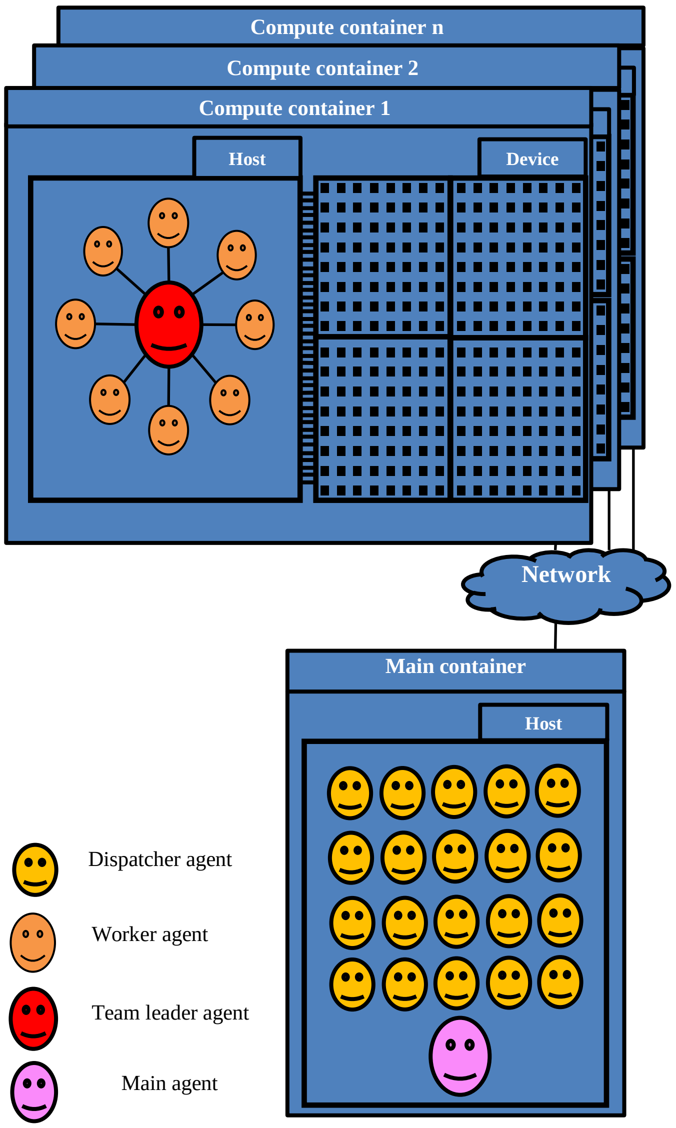

Fig. 2. Runtime environnement architecture.

As shown in Fig. 2 the runtime environment consists of two types of containers. The first is the main container which is a JADE main container which must always be active in a platform. All other containers register with it as soon as they start. Note that only one main container must be launched at first to start the JADE platform. The main container has the ability of accepting registrations from other non-main containers. A main container holds two special agents, automatically started when the main container is launched. The first one is the AMS (Agent Management System) that provides the naming service (i.e. ensures that each agent in the platform has a unique name) and represents the authority in the platform (for instance it is possible to create/kill agents on remote containers by requesting that to the AMS). The second one is the DF (Directory Facilitator) that provides a Yellow Pages service by means of which an agent can find other agents providing the services he requires in order to reach his goals. Additionally, the main container holds dispatcher agents and a main agent.

The second type of container is the compute containers which are JADE normal ('non-main') containers, each compute container register with the main container as soon as it starts and must "be told" where to find (host and port) its main container. In compute containers, we find worker agents and one team leader agent.

B. *Workflow*

In this section, we show how K-means application on a large-sized image is performed within the MAS, and how agents interact with each other across nodes to achieve efficient tasks and data communications. Fig. 3 illustrates the steps and interactions established within the multi-agent system during k-means application on large-sized image.

At the beginning, in the main container, the main agent chooses K data randomly as initial centroids. After that, it divides the large-sized image data into a subsets of image data. The stream of subsets of image data will then be sent to dispatcher agents. The role of these agents is to compress and dispatch the data subsets through the computer container.

In the compute containers, team leader agents listen to queries and data subsets data sent by dispatcher agents. For each data subset received, the team leader agent delegates it to a worker agent. Thus, each worker agent decompresses the subset of data image received and performs the data assignment step of K-means using independent GPU computing units i.e. the Streaming Multiprocessor (SM). For each image data subset of the stream to process, the SMs of the GPUs have their own queue used to collaborate with a worker agent. After that, each of the worker agent returns the membership matrix containing the membership labels of each pixel of the processed data image subsets.

The main agent performs the data rearrangement which consists of calculating the sum of the pixels intensities of each cluster and calculating the number of elements of each cluster with the aim to calculate the new centroids.

In summary, the purpose of these interactions is to send subsets of data images to the worker agents. They then perform the data assignment step of K-means using the SMs of the GPUs, as shows in the Algorithm 1 below. The initiation routine, data rearrangement (described by the Algorithm 2) and the convergence test steps are performed by the main agent using the CPU. After that, running the K centroids recalculations depends on K which is the number of clusters declared during the initiation routine. If K is less than 100 the main agent itself performs the K centroids recalculation step, or else it delegates the recalculations to a team leader agent for parallel execution using the GPU (Algorithm 3) in collaboration with a worker agent.

C. *K-Means Execution Steps*

Beyond the data communication and synchronization among agents across the MAS, each agent has a role to perform the specific K-means steps as summarized in the following:

- ***Centroids initialization***: The main agent selects K points randomly as initial clustering centroids.

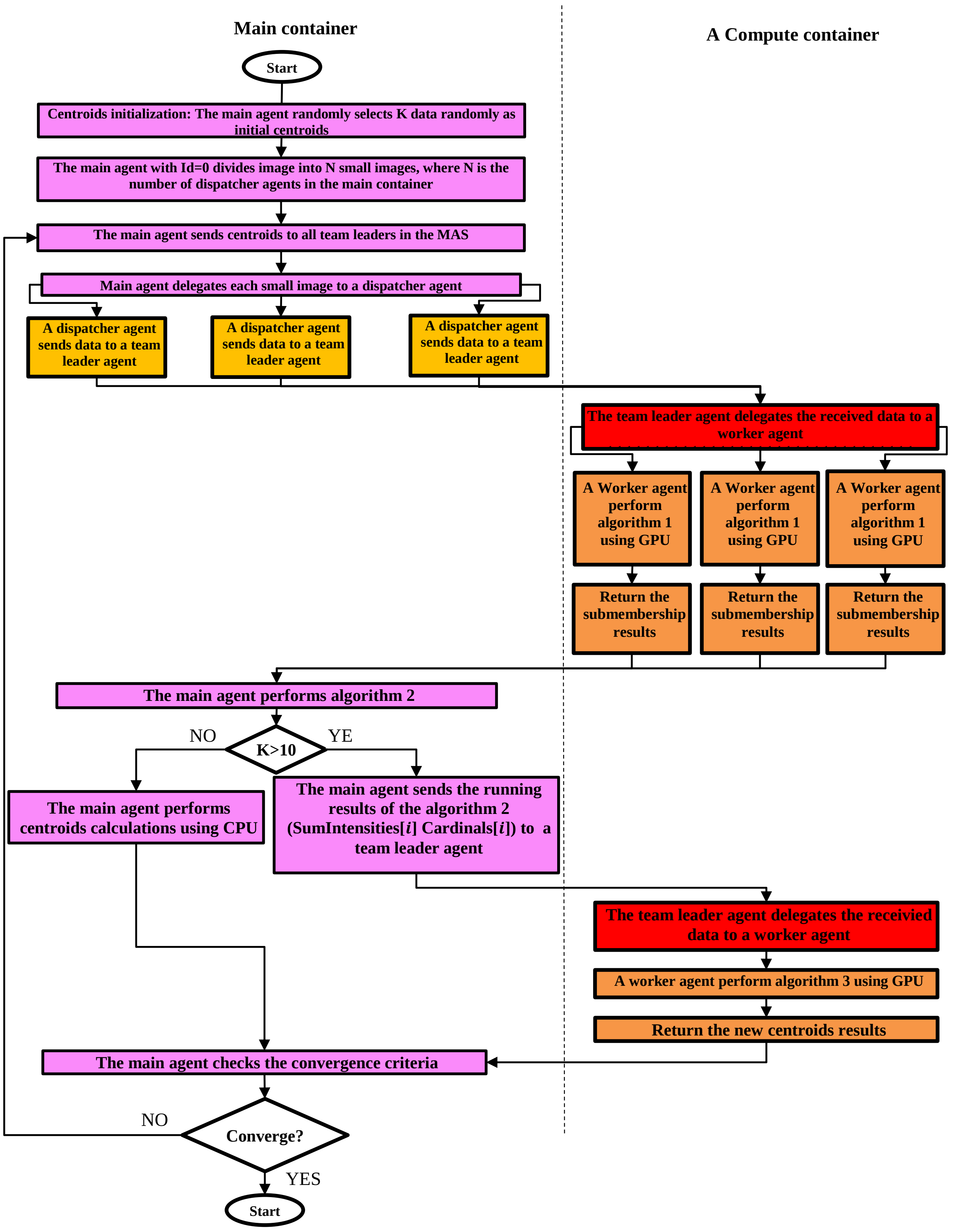
Main container
A Compute container
Start
Centroids initialization: The main agent randomly selects K data randomly as initial centroids
The main agent with Id=0 divides image into N small images, where N is the number of dispatcher agents in the main container
The main agent sends centroids to all team leaders in the MAS
Main agent delegates each small image to a dispatcher agent
A dispatcher agent sends data to a team leader agent
A dispatcher agent sends data to a team leader agent
A dispatcher agent sends data to a team leader agent
The team leader agent delegates the received data to a worker agent
A Worker agent perform algorithm 1 using GPU
A Worker agent perform algorithm 1 using GPU
A Worker agent perform algorithm 1 using GPU
Return the submembership results
Return the submembership results
Return the submembership results
The main agent performs algorithm 2
NO
K>10
YE
The main agent performs centroids calculations using CPU
The main agent sends the running results of the algorithm 2 (SumIntensities[i] Cardinals[i]) to a team leader agent
The team leader agent delegates the received data to a worker agent
A worker agent perform algorithm 3 using GPU
Return the new centroids results
The main agent checks the convergence criteria
NO
Converge?
YES
Start

Fig. 3. K-means workflow.

- **Data assignment:** Each worker agent performs this step on the received subsets of image data in collaboration with an SM of GPU. This step consists to calculating the distance between each points and centroids, and clusters the points using these distances. Each point data sets will be delegated to a processor in GPU:

Algorithm 1: Data assignment step (device code)

```
#include <math.h>
extern "C"
__global__ void dataAssignement(float * dataStream,
                                float * centroids,
                                float * dataStreamMembership)
{
  int v = 0;
  int t = threadIdx.x + blockIdx.x * blockDim.x;
  float dist = 0,
  float distMin = fabs(dataStream[t] - centroids[0]);
  dataStreamMembership[t] = 1;
  for(v = 1;v < 5;   v  ++)
   {
     dist = fabs(dataStream[t] - centroids[v]);
     if (dist <= distMin)
     {
       dataStreamMembership[t] = v+1
       distMin = dist;
     }
   }
}
```

- **Data rearrangement:** The main agent rearranges all data, and calculates sumIntensites $[i]$ and Cardinals$[i]$ where $i = 1\ldots$c, which will be used to calculates the new centroids:

Algorithm 2: Data rearrangement step (host code)

```
private static void dataRearrangement(float[] cardinal,
                                      float[] sumDistances,
                                      float[] sumIntensites,
                                      float[] dataStream,

float[] dataStreamMembership)
{
  for (int i = 0; i < cardinal.length; i++)
  {
   // reset counters
   sumIntensites [i]=0;
   cardinal[i]=0;
  }
  // calculate sumIntensites and calculate the number element
  // each cluster
  for (int i = 0; i < dataStreamMembership.length; i++)
  {
   sumIntensites[(int) dataStreamMembership[i]-
1]+=dataStream[i];
   cardinal[(int) dataStreamMembership[i] - 1]++;
  }
```

- **K-centroids recalculations:** This step is performed by the main agent sequentially if K is less than 100, or else the maim agent delegate it to a team leader in a computer container in order to be performed using GPUs in collaboration with an agent worker. The agent worker uses GPU to recalculate the new centroids of each cluster. Every thread block in the GPU is responsible for a new centroid:

In recapitulation, the data assignment, K-centroids recalculations are parallel performed on the SMs of GPUs. The main agent is responsible for centroids initialization, data rearrangement and controlling the iteration process.

Algorithm 3: K centroids recalculation step (device code)

```
# include < math.h >
extern "C"
__global__ void centroidsRecalculation(float * centroids,
                                       float * cardinal,
                                       float * sumIntensites)
{
  int t = threadIdx.x;
  centroids[t] = sumIntensites[t] / cardinal[t];
}
```

V. Experimental Results

In this section, we use a set of large sized images to compare the total processing time of these images using our proposed GPU-based K-means distributed on multiple computer nodes using Multi Agent System, with the total processing time of the same set of images using GPU-based k-means performed on GPUs on a single node.

All experiments were concluded on 4 nodes equipped with Intel Core i7-3610QM CPU 2.30GHz (8 CPUs), 8GB main memory and GeForce GTX 660M, 835 MHz engine clock speed, 2048 MB GDDR5 of device RAM, and 384 processors, organized into 3 streaming multiprocessors. Additionally, we use 4 external GPUs GeForce GTX 750Ti connected by PCIe using a PE4C V2.1 connectors. This environment was assembled and tested in our laboratory for testing purposes. The GTX 750Ti have 1020 MHz engine clock speed, 2048 MB GDDR5 of device RAM, and 640 processors, organized into 5 streaming multiprocessors. All GPUs used in this study use single-precision floating-point arithmetic.

The results were obtained using sets of large-sized images, All Euclidean distance calculations were done in single-precision. The performance of our K-means algorithm version depends on the actual data and task communication between agents across nodes. To observe the influence of data size on the total running time, large-sized images with thousands of intensity points were used as show in Table I below:

TABLE I. The Image Data used for the Tests

Image Id	**Image points(px)**	**Image height**	**Image width**
I_1	20006400	5120	3840
I_2	39052992	7216	5412
I_3	69120000	9600	7200
I_4	100000000	10000	10000
I_5	400000000	20000	20000

The test scenarios were carried out on the five large-sized images with three different hardware configurations. The first scenario was made using 2 compute Nodes with 4 GPUs. The second was made using 4 compute nodes with 6 GPUs, and third was made with 4 compute nodes with 8 GPUs.

The measurements taken were the total processing time, which includes data transfer. Total processing time of GPU-based K-means on single node is denoted by GKtt, and our distributed GPU-based K-means are denoted by DGKtt. The results obtained are presented in Table II. The initial class centers are chosen as: $(c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}) = (1, 25, 50, 75, 100, 125, 150, 175, 200, 254)$.

TABLE II. TOTAL RUNNING TIME OF THE LARGE SIZED IMAGES SEGMENTATION USING THREE DIFFERENT HARDWARE CONFIGURATIONS

Image Id	**Image points(px)**	**GKtt**	**DGKtt**		
			2 compute Nodes with 4 GPUs	**4 compute Nodes with 6 GPUs**	**4 compute Nodes with 8 GPUs**
I_1	20006400	14862,64	3125,04	1562,52	781,26
I_2	39052992	24609,62	5468,88	2878,36	1204,60
I_3	69120000	46871,94	9375,12	5208,40	2410,06
I_4	100000000	61351,96	13773,72	11478,10	3935,35
I_5	400000000	224129,03	54250,86	28553,08	18083,62

The speedup of our GPU-based K-means could reach from 4 to 5 of the CPU-based K-means in the first scenario and 8 to 9,5 in the second and 12 to 20 in the third scenario. This performance improvement benefits from the high parallel computing ability of the GPU using CUDA, the data rearrangement and the division of the problem to lightweight subproblem. In addition, in CUDA GPU, processors and CUDA Streams are all indistinctive, and not distinguished by pixel and vertex, so that they can run at the same time without any idle time.

VI. DISCUSSION

Using distributed computing based on agent combined with GPU computing show the advantages of easily encoding data and task communication among computer nodes. In addition, the agent communication language used (ACL) which follows the FIPA specifications make the communication transparent and make exchanges between computer nodes structured. Specifically, the use of the JADE platform or similar platform with Nvidia GPU in compute-intensive and data-intensive application such as K-means applied for image segmentation, allows using a high-level programming language like java to write Host code with JNI wrapper, and CUDA C/C++ for device code.

In our work we focus on how to solve the problem of segmentation of large-sized images using the combination of two powerful computing paradigms. Unlike [4, 5] who focused on the memory management and the communication of the thread blocks in single GPU, and in the case of data-intensive application it will be complicated to guarantee their effectiveness.

Despite of the great performance of [2], specifically for the massively image processing, it is possible to speed up the computer nodes with GPUs to have more computing power. Thus, to implements a K-means version with Hadoop and GPU, the programmer needs to understand the low-level communication routines and storage mechanism of Hadoop framework in order to be able to write scalable algorithm, which in this case will be a mixture between Hadoop and CUDA code.

Furthermore, the overhead of the data and task synchronization between agents across nodes is limited by the efficiency of the connected network where is deployed; as instance, using standard Local area network (LAN) with 54Mbits/s or 100Mbits/s, the latency can be quite high. This last can be reduced using a different physical medium for data communication as the Fiber Distributed Data Interface (FDDI), with 1Gbits/s; or the IEEE 802.3 gigabit Ethernet, with 10Gbits/s (e.g. 1000Base-LX or 1000Base-SX series); or the well-suited InfiniBand bus which was used in this research. There are many such studies to demonstrate communication latency and process synchronization; however, they are out of the scope of this research.

VII. CONCLUSION

This K-means version was implemented using the agent-based distributed computing and GPU computing to solves the problem of hardware limitations, specifically the number of elementary processors and storage capacity. Also, instead of using low-level libraries like MPI or OpenMP which can be error-prone in some complex cases such as massively image processing, we used programming paradigm based on agent with JADE framework to overcome difficulty of the communication and synchronization between nodes in the distributed system and this being based on the FIPA specifications.

Our implementation of K-means allowed us to confirm the possibility of using this type of model based on two HPC paradigms (Distributed and GPU computing) to solve problems of hardware limitations and opening the possibility of designing more scalable HPC models.

REFERENCES

[1] J. Shawe-Taylor and N. Cristianini, “Kernel Methods for Pattern Analysis”. Cambridge University Press, 2004.

[2] Yingchi Mao, Ziyang Xu, Ping Ping, Longbao Wang, “An Optimal Distributed K-Means Clustering Algorithm Based on CloudStack”, In : Information and Automation, 2015 IEEE International Conference on. IEEE, 2015. pp. 3149-3156.

[3] Mohammed Baydoun, Mohammad Dawi, and Hassan Ghaziri , “Parallel Kernel K-Means on the CPU and the GPU”, In : Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2016. pp. 117.

[4] W. Fang, K. K. Lau, M. Lu, X. Xiao, C. K. Lam, P. Y. Yang, B. He, Q. Luo, P. V. Sande, and K. Yang, “Parallel Data Mining on Graphics Processors”, Technical Report HKUSTCS08, 2008.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, “A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA”, Journal of Parallel and Distributed Computing, 2008.

[6] ZHOU, Nanrun, ZHANG, Aidi, ZHENG, Fen, et al. “Novel image compression–encryption hybrid algorithm based on key-controlled measurement matrix in compressive sensing”. Optics & Laser Technology, 2014, vol. 62, pp. 152-160.

[7] Anders Eklund, Paul Dufort, Daniel Forsberg, Stephen M. La Conte "Medical image processing on the GPU – Past, present and future" Medical Image Analysis, Volume 17, Issue 8, December 2013, pp. 1073-1094.

[8] David Kirk. "NVIDIA Cuda software and GPU parallel computing architecture". In: Proceedings of the 6th International Symposium on Memory Management, New York, NY, USA, 2007, vol. 7, pp. 103-104.

[9] Nvidia Corporation. "Whitepaper NVIDIA GeForce GTX 750 Ti", 2014.

[10] Ihirri Soukaina, Errami Ahmed and Khaldoun Mohamed. "Parallel and Reconfigurable Mesh Architecture for Low and Medium Level Image Processing Applications". In : Proceedings of the Advances in Ubiquitous Networking 2, Casablanca, Morocco, 2016, pp. 529-544.

[11] Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Raymond Namyst and Samuel Thibault. "StarPU-MPI: Task programming over clusters of machines enhanced with accelerators". 2014. Ph.D. Dissertation. Institut national de recherche en informatique et en automatique (INRIA), 2012.

[12] Alan Kaminsky. "The Parallel Java 2 Library". In: The International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, LA, USA, November 18, 2014, Poster Session.

[13] Marc Snir. "MPI--The Complete Reference: Volume 1", The MPI core. MIT press, second edition, 1998.

[14] William Gropp, Ewing Lusk and Anthony Skjellum. "Using MPI: portable parallel programming with the message-passing interface". MIT Press, seconde edition, 1999.

[15] Leonardo Dagum and Ramesh Menon. "OpenMP: an industry-standard API for shared-memory programming". IEEE Computational Science and Engineering, 1998, vol. 5, issue 1, pp. 46-55.

[16] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan and Jeff McDonald. "Parallel programming in OpenMP". Morgan Kaufmann, 2001.

[17] Rolf Rabenseifner, Georg Hager and Gabriele Jost. "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes". In: Proceeding of the 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, Weimar, Germany, 2009. pp. 427-436.

[18] Haoqiang Jin, Dennis Jespersen, Piyush Mehrotra, Rupak Biswas, Lei Huang and Barbara Chapman. "High performance computing using MPI and OpenMP on multi-core parallel systems". Parallel Computing, 2011, vol. 37, issue 9, pp. 562-575.

[19] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio and Dietmar Fey. "Hpx: A task based programming model in a global address space". In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, Eugene, Oregon, USA, 2014, 6 October, pp. 6.

[20] Tom White. "Hadoop: The Definitive Guide", O'Reilly Media, 2012.

[21] Brian L. Claus and Stephen R. Johnson. "Grid computing in large pharmaceutical molecular modeling". Drug Discovery Today, 2008, vol. 13, issues 13-14, pp. 578-583.

[22] Bogdan SATARIĆ, Vladimir SLAVNIĆ, Aleksandar BELIĆ et al. "Hybrid OpenMP/MPI programs for solving the time-dependent Gross–Pitaevskii equation in a fully anisotropic trap". Computer Physics Communications, 2016, vol. 200, pp. 411-417.

[23] Thorsten KURTH, Brandon COOK, Jack DESLIPPE et al. "OpenMP Parallelization and Optimization of Graph-Based Machine Learning Algorithms". In : OpenMP: Memory, Devices, and Tasks: 12th International Workshop on OpenMP, IWOMP 2016, Nara, Japan, October 5-7, 2016, Proceedings. Springer, 2016, pp. 17.

[24] Massimiliano ALVIOLI and Rex L. BAUM. "Parallelization of the TRIGRS model for rainfall-induced landslides using the message passing interface". Environmental Modelling & Software, 2016, vol. 81, pp. 122-135.

[25] Siddhartha KHAITAN. "MASTER: A JAVA Based Work-Stealing Technique For Parallel Contingency Analysis". 2016. Doctoral Thesis. Iowa State University.

[26] Saliya EKANAYAKE, Supun KAMBURUGAMUVE and Geoffrey C FOX. "SPIDAL Java: high performance data analytics with Java and MPI on large multicore HPC clusters". In: Proceedings of the 24th High Performance Computing Symposium. Society for Computer Simulation International, 2016, pp. 3.

[27] Cosmin Marian POTERAŞ, Marian Cristian MIHĂESCU, and Mihai MOCANU. "An optimized version of the K-Means clustering algorithm". In : Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on. IEEE, 2014, pp. 695-699.

[28] Jadran SIROTKOVIĆ, Hrvoje DUJMIĆ, and Vladan PAPIĆ. "K-means image segmentation on massively parallel GPU architecture". In : MIPRO, 2012 Proceedings of the 35th International Convention. IEEE, 2012. pp. 489-494.

[29] Kai J.KOHLHOFF, Vijay S.PANDE, and Russ B.ALTMAN. "K-means for parallel architectures using all-prefix-sum sorting and updating steps". IEEE Transactions on Parallel and Distributed Systems, 2013, vol. 24, no 8, pp. 1602-1612.

[30] Borislav ANTIĆ, Dragan LETIĆ, Dubravko ĆULIBRK et al. "K-means based segmentation for real-time zenithal people counting". In : Image Processing (ICIP), 2009 16th IEEE International Conference on. IEEE, 2009, pp. 2565-2568.

[31] Fatéma Zahra BENCHARA, Mohamed YOUSSFI, Omar BOUATTANE et al. "Distributed C-means algorithm for big data image segmentation on a massively parallel and distributed virtual machine based on cooperative mobile agents". Journal of Software Engineering and Applications, 2015, vol. 8, no 03, p. 103.

[32] James MacQueen. "Some methods for classification and analysis of multivariate observations". In: Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability, University of California, Berkeley, 1967, pp. 281-297.

[33] Fabio. Bellifemine, Giovanni Caire and Domonic Greenwood. "Developing Multi-Agent Systems with JADE". Wiley, 2007.