EURASIP Journal on Image and Video Processing

RESEARCH Open Access

RIDE: real-time massive image processing platform on distributed environment

Yoon-Ki Kim, Yongsung Kim and Chang-Sung Jeong*

Abstract

As the demand for real-time data processing increases, a high-speed processing platform for large-scale stream data becomes necessary. For fast processing large-scale stream data, it is essential to use multiple distributed nodes. So far, there have been few studies on real-time massive image processing through efficient management and allocation of heterogeneous resources for various user-specified nodes on distributed environments. In this paper, we shall present a new platform called RIDE (Real-time massive Image processing platform on Distributed Environment) which efficiently allocates resources and executes load balancing according to the amount of stream data on distributed environments. It minimizes communication overhead by using a parallel processing strategy which handles the stream data considering both coarse-grained and fine-grained parallelism simultaneously. Coarse-grained parallelism is achieved by the automatic allocation of input streams onto partitions of broker buffer each processed by its corresponding worker node, and maximized by adaptive resource management which adjusts the number of worker nodes in a group according to the frame rate in real time. Fine-grained parallelism is achieved by parallel processing of task on each worker node and maximized by allocating heterogeneous resources such as GPU and embedded machines appropriately. Moreover, it provides a scheme of application topology which has a great advantage for higher performance by configuring the worker nodes of each stage using adaptive heterogeneous resource management. Finally, it supports dynamic fault tolerance for real-time image processing through the coordination between components in our system.

Keywords: Real-time, Image processing, Distributed and parallel processing, Heterogeneous computing

1 Introduction

Today, data generated in real time, such as CCTV images, web logs, satellite images, and stock data, is increasing in volume, and there is a need to process large-scale data rapidly. Distributed processing technologies such as Hadoop [1] using multiple nodes have been developed to process large-scale data. It has become popular, due to its Mapreduce model using the Hadoop Distributed File System (HDFS) [2] and automatic data management. However, Hadoop is designed for batch processing. That means, Hadoop takes a large dataset in input all at once, process it, and write a large output. The concept of Mapreduce is geared toward batch but not real-time.

Some frameworks adopt the micro batch processing on Mapreduce model for real-time processing [3, 4] which performs the map and reduce operation many times whenever input data occurs in real time. It is a special case of batch processing when the batch size is small. It can simply process real-time streams using existing Mapreduce models. However, it is even less time-sensitive than near real-time. Generally, batch processing involves three separate processes such as data collection, map, and reduce. For this reason, it could incur latency costs when processing large-scale images.

There is another approach for stream data processing, Storm [5] which is a representative framework of real-time distributed processing. It is a task parallel continuous computational engine. It defines its workflows in DAG (directed acyclic graphs) called topologies. These topologies run until shutdown by the user or encountering an unrecoverable failure. There has been an attempt on Storm for distributed processing of stream images in real time [6]. However, it has a problem of processing speed degradation due to assignment of excessive overlapping areas for each image among distributed nodes. Moreover, Storm does not support the management and allocation of heterogeneous resources considering the

* Correspondence: csjeong@korea.ac.kr
Department of Electrical Engineering, Korea University, Seoul, Korea

performance of each resource for real-time image processing.

The aforementioned distributed processing systems exploit high-speed processing technology for massive repetitive operations of simple task. They use a coarse-grained parallelism, allowing distributed nodes to concurrently process the largely divided task assigned to them. However, they can only achieve the maximum performance when there is no data dependency between the tasks. Therefore, they are not proper for image processing applications with large data dependency within each image.

Also, there have been attempts to process the massive stream data such as satellite image using a GPU accelerator on single node [7–11]. GPU-based image processing showed better performance than CPU based one. It uses fine-grained parallelism on single node, allowing single node to process the finely divided tasks. However, the higher speed stream images are generated, the more unprocessed images are accumulated, which leads to sharp increase in latency.

In this paper, we shall present a new platform called Real-time massive Image processing platform on distributed Environment (RIDE) which can process real-time massive image stream on distributed parallel environment efficiently by providing a multilayered system architecture which supports both coarse-grained and fine-grained parallelisms simultaneously in order to minimize the communication overhead between the tasks on distributed nodes. Coarse-grained parallelism is achieved by the automatic allocation of input streams onto partitions each processed by its corresponding worker node, and maximized by adaptive resource management which adjusts the number of worker nodes in a group according to the frame rate in real time. Fine-grained parallelism is achieved by parallel processing of task on each worker node and maximized by allocating heterogeneous resources such as GPU and embedded machine appropriately. RIDE provides a user friendly programming environment by supporting coarse-grained parallelism automatically by the system while the users only need to consider fine-grained parallelism by careful parallel programming on multicore or GPU. For real-time massive stream image processing, we design a distributed buffer system based on Kafka [12] which enables each of distributed nodes to access and process the buffered image in parallel [13], improving its overall performance sharply. Besides, it supports dynamic allocation of partitions to worker nodes which maximizes the throughput by preventing worker nodes from being idle. Moreover, it provides a scheme of application topology which has a great advantage for higher performance by configuring the worker nodes of each stage using the adaptive heterogeneous resource management. Finally, it supports dynamic fault tolerance for real-time image processing through the coordination between components in our system.

The rest of the paper is organized as follows: In section 2, we give an overview about our system, In section 3, we explain about the main methods of RIDE, and then, in section 4, we present a system architecture for RIDE. In section 5, we describe about the experimental result and discussion about RIDE. Finally, section 6 summarizes the conclusion of our research.

2 System overview

RIDE is designed as a system suitable for processing large-scale stream image such as satellite, CCTV, and drone images in real time. It can receive and process data from multiple channels of real-time sensors or cameras. Its architecture consists of four layers: user interface, master, buffer, and worker as shown in Fig. 1. In the application layer, the user creates an application for massive image processing and submits it to the master layer. In the master layer, the master node allocates broker and worker nodes in buffer and worker layers respectively and then distributes stream images onto partitions in broker nodes. Finally, it divides the application into tasks, and assigns them to worker nodes each processing one partition in buffer layer.

The buffer layer consists of several topics, each of which stores real-time images which are distributed onto multiple partitions residing in the single or distributed broker nodes based on the file system using Kafka. Each topic consists of multiple partitions coming from the same source, and each of multiple partitions in the topic is processed by a single worker so that multiple partitions can be processed simultaneously by several workers as shown in Fig. 2.

The worker layer is made up of worker nodes each of which executes the task assigned from the master layer by accessing and processing one partition of the topic in the buffer layer. We can achieve coarse-grained parallelism by making each worker node work on the different partition in the same topic simultaneously, while improving fine-grained parallelism by further dividing the image into sub-images each being processed on heterogeneous resources such as GPU and multicores in each worker node.

3 Methods

In this section, we shall describe several methods of our system RIDE: hybrid parallelism, adaptive heterogeneous dynamic resource management, application topology, and fault tolerance.

3.1 Hybrid parallelism

The previous distributed processing systems based on Hadoop divide input data into a block of HDFS and push them into the worker nodes [1–4]. In this case, the

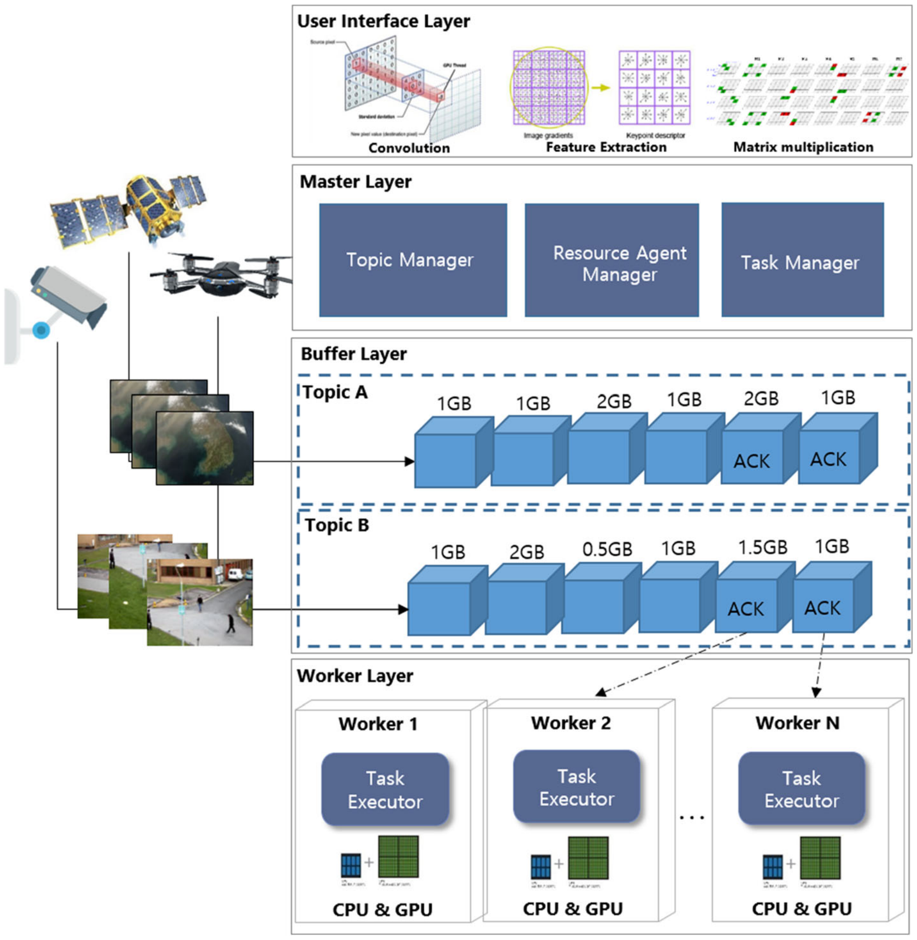

Fig. 1 System overview of RIDE

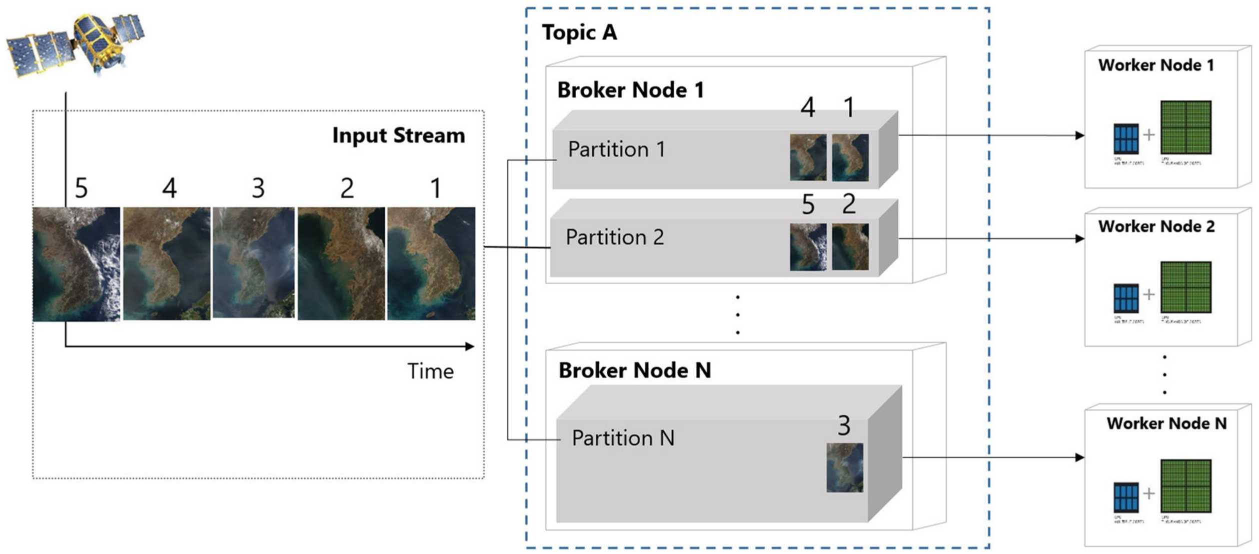

Fig. 2 Partitioning of input stream for multiple nodes processing

input data is divided into the same size regardless of the user application. Therefore, when processing an image, it is divided into blocks of predefined size, and they may be transmitted to different nodes. Due to the nature of image processing, one frame is represented by a matrix or vector, and the most of operations required for image processing have dependencies within the same matrix. Thus, in case dependent data may be distributed to different nodes, there arises large overhead due to frequent communication between distributed nodes. Therefore, the previous distributed processing systems based on Hadoop cannot efficiently support coarse-grained parallelism for image processing applications due to the data dependency between the partitioned data. Figure 3 shows the characteristics of the previous systems based on HDFS.

RIDE supports coarse-grained parallelism by distributing each whole frame into one of multiple partitions within topic without no division based on Kafka buffer system. Since each worker node accesses one partition each consisting of non-partitioned frames, it can efficiently perform coarse-grained data parallel processing without communication overhead.

After importing the data from the partition, each worker node performs the fined grained parallelism on multicores or GPU. Therefore, RIDE efficiently supports coarse- and fine-grained parallelisms simultaneously by distributing frames to each worker node and then processing them on multicore or GPU. The former is achieved by the automatic allocation of partitions to each worker node, while the latter by parallel processing of each partition on each worker node using multicores or GPU. Since coarse-grained parallelism is automatically supported by the system, the users only need to consider fine-grained parallelism by careful parallel programming on multicore or GPU. These characteristics are shown in Fig. 4 below.

3.2 Adaptive heterogeneous dynamic resource management

RIDE maximizes hybrid parallelism by supporting the following three types of resource management schemes:

Adaptive resource management: The number of processing nodes can be flexibly adjusted considering the amount of image frames generated in real time.

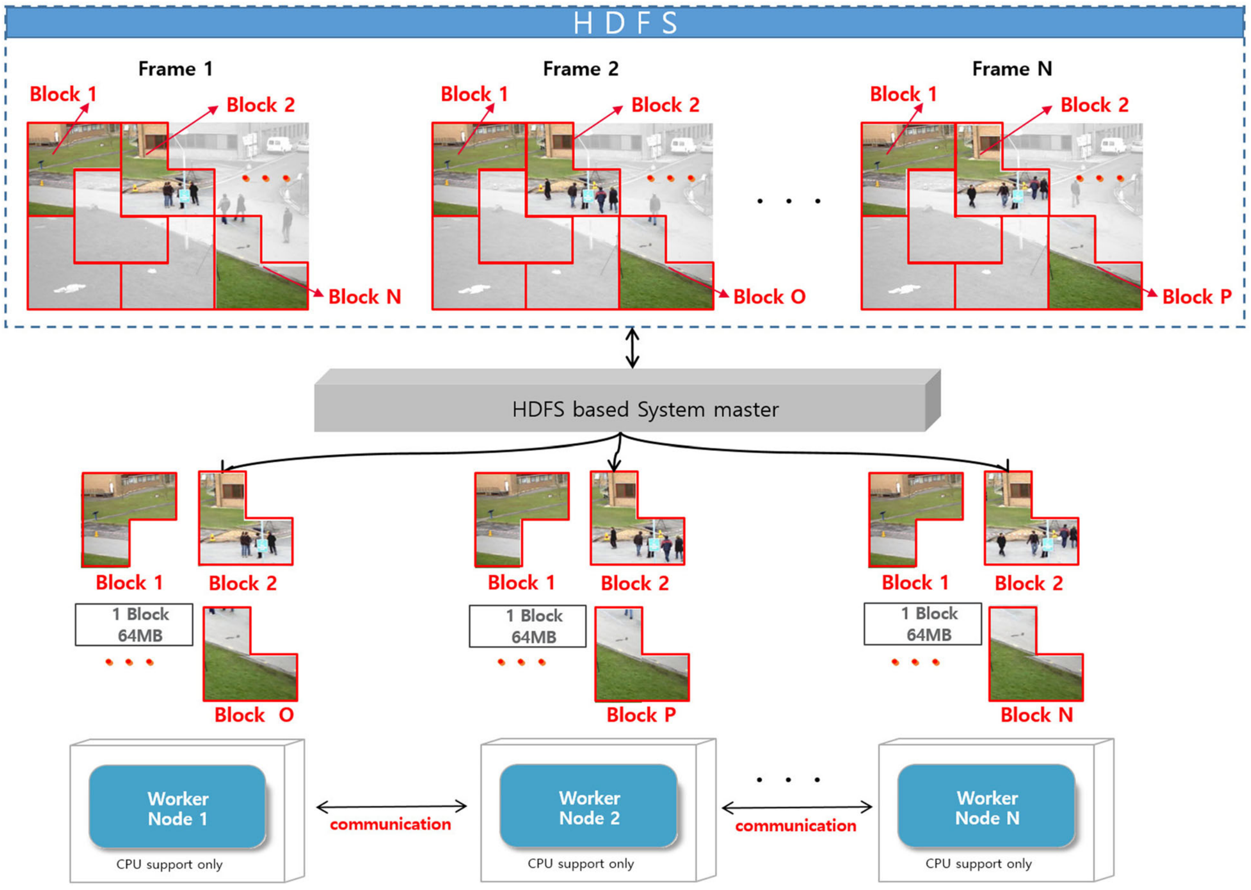

Fig. 3 Data partitioning on HDFS

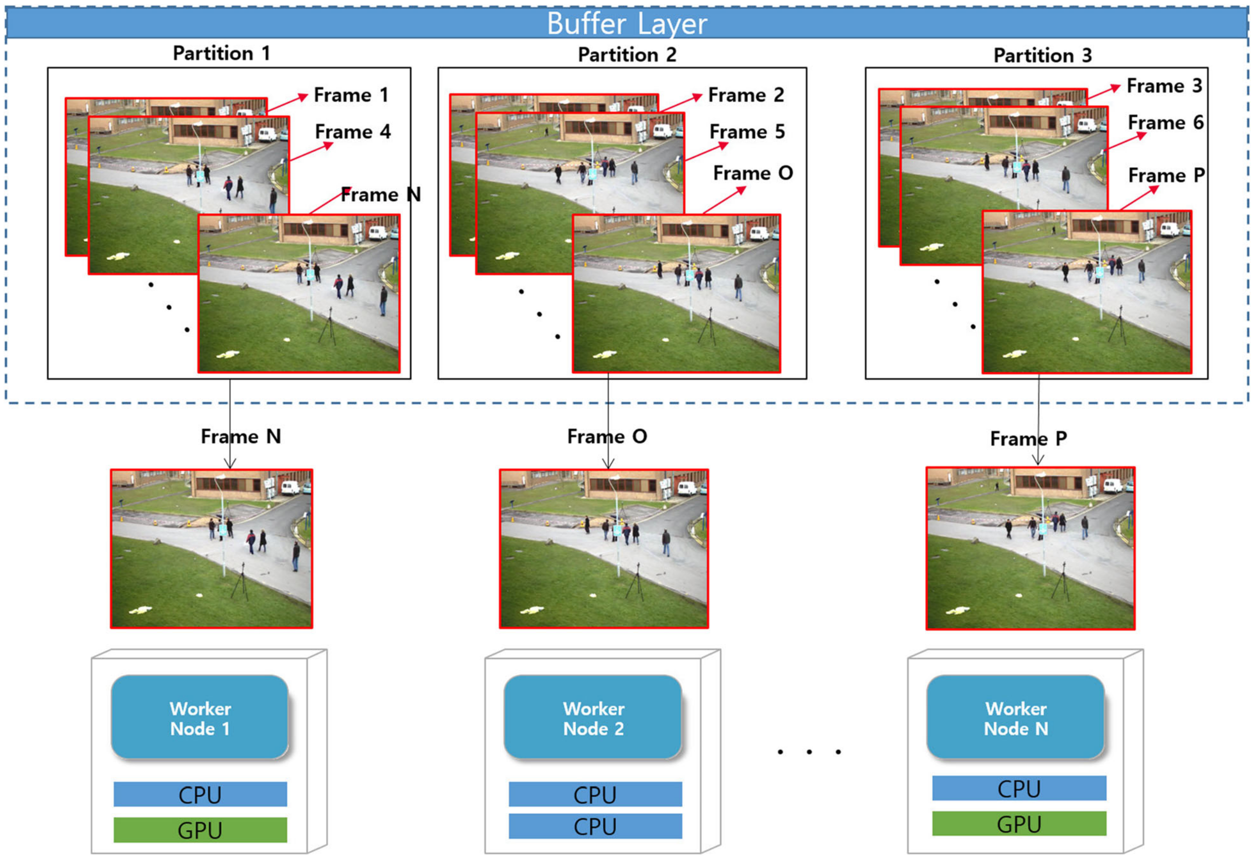

Fig. 4 Data partitioning on RIDE

User can set up a group of multiple worker nodes each of which processes the same task. As shown in Fig. 5, several workers can be grouped together by user, and each group is assigned the same task. Each worker in a group is mapped to one distinct partition and granted its ownership. Then, each worker can only access the partition with its ownership. Therefore, user can achieve coarse-grained parallelism by adaptive resource management which adjusts the number of worker nodes in a group according to the frame rate in real time.

Heterogeneous resource management: User can maximize the fine-grained parallelism of image processing task given to a group by allocating heterogeneous resources in each worker node of the group. According to the type of task, user can choose the optimal memory capacity, the number of CPU cores and/or GPU. For example, in case the task contains many conditional and branching statements, multi-core may be advantageous over GPU, and hence, the workers in the group may be configured into multi-cores for higher performance, while in case the task requires high processing power, they may be configured into GPUs.

Dynamic resource management: If a worker node is statically assigned data, each worker node can have a different processing speed, so the total throughput is determined by the worker node that most recently completed the task, which may cause the overall performance degradation. To overcome this problem, after resource allocation, the partitions in a topic are dynamically assigned to each node in a group. Whenever each worker node completes the processing of a partition assigned to it, it is allocated another partition dynamically. The dynamic allocation of partitions to worker nodes maximizes the throughput by preventing worker nodes from being idle.

3.3 Application topology

In RIDE, user can define a workflow called application topology which defines the flow of stages for image processing job. Each stage of workflow consists of three components: a topic of partitions, a task, and a group of worker nodes as shown in Fig. 6. Each worker node executes the same task to process the unique partition assigned to it. Each stage in an application topology is a unit of processing the task, and the whole application topology has a great advantage for higher performance,

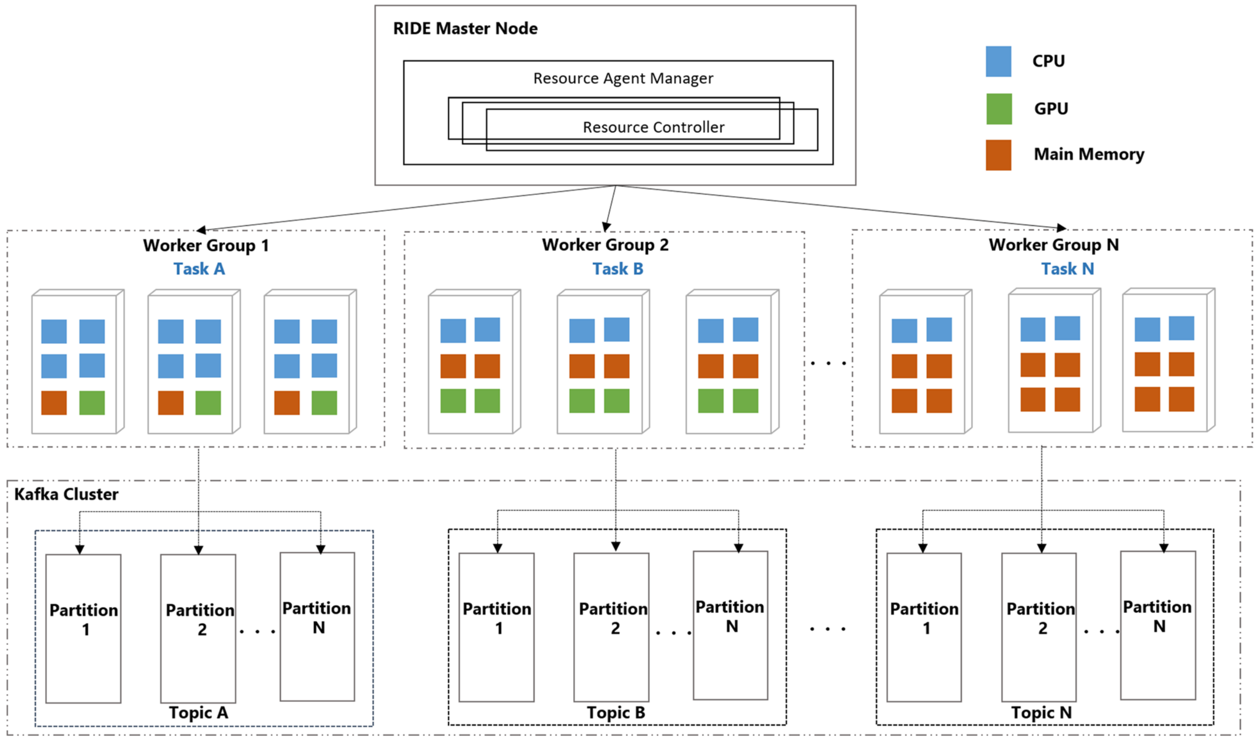

Fig. 5 Resource allocation on RIDE

since it may configure the worker nodes of each stage using the adaptive heterogeneous resource management.

3.4 Fault tolerance

RIDE supports dynamic fault tolerance for real-time image processing. Each frame from stream source is stored into one of multiple partitions in a round-robin manner and processed by the task in the node with the ownership over the partition as shown in steps 1 and 2 of Fig. 7. Each frame is stamped with ACK (Acknowledgement) after it has been completely processed by the task in the node with the ownership over the partition containing the frame.

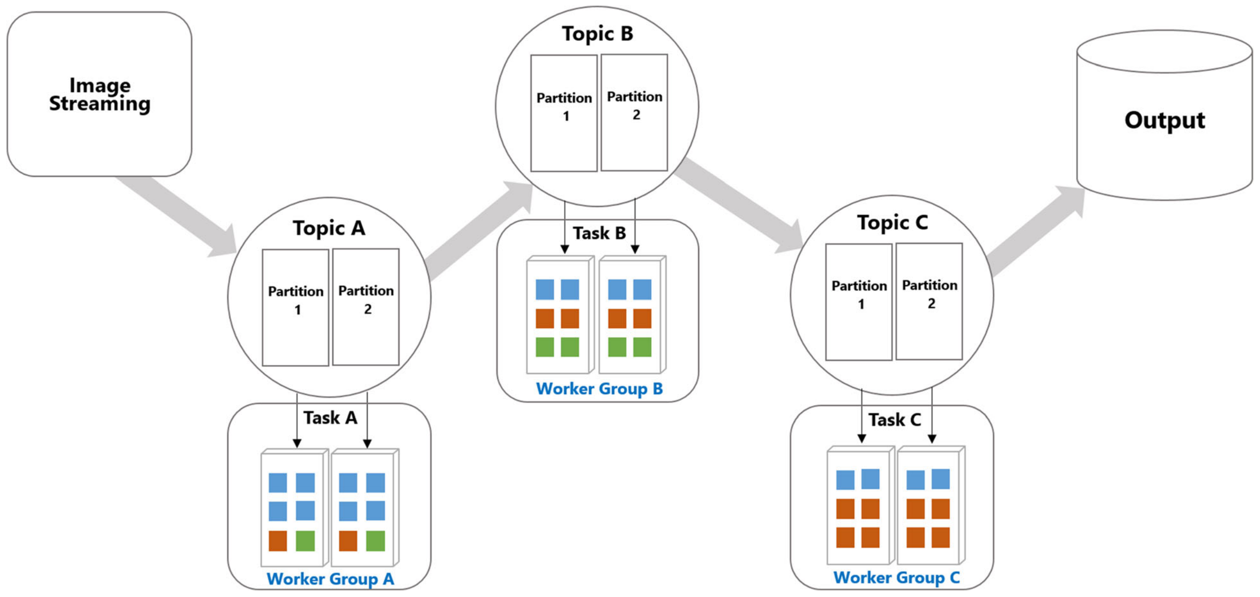

Fig. 6 Application topology on RIDE

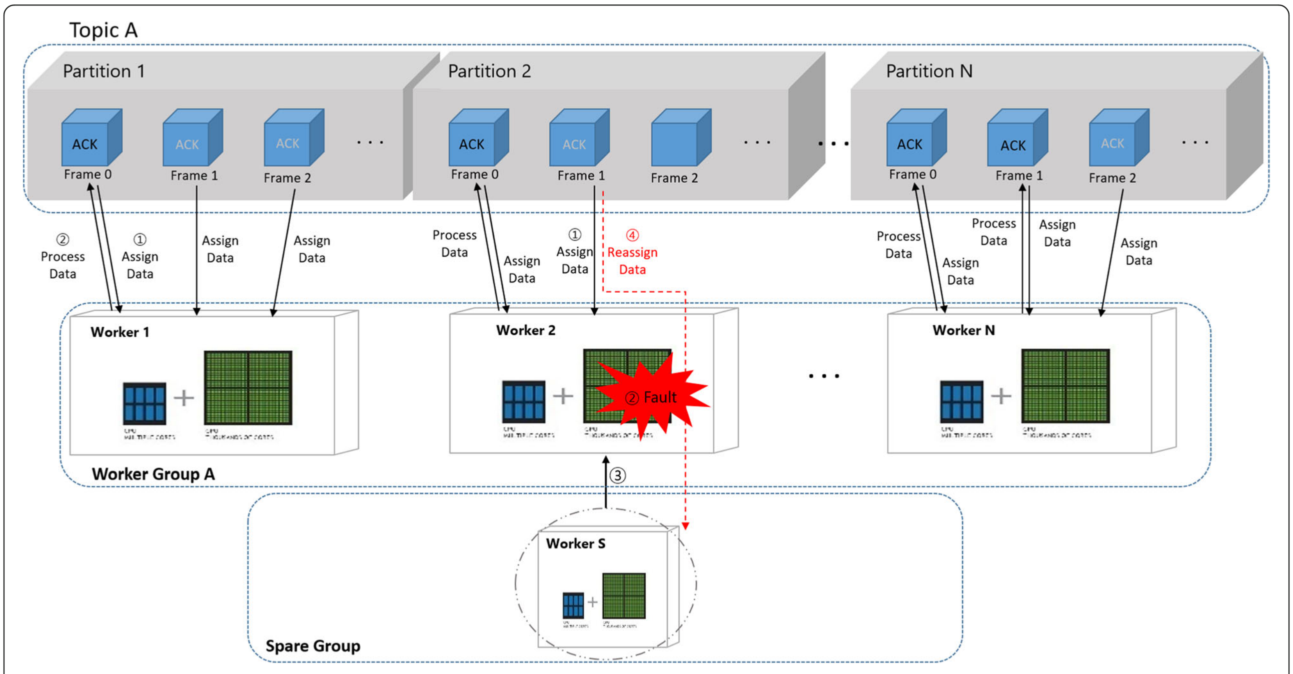

Fig. 7 Fault tolerance of RIDE

For fault tolerance, user may request several spare worker nodes among the remaining available resources. Resource monitor in master node periodically checks the status of each worker node by sending heartbeat. If it detects the failure of a worker node, the master node moves one of spare nodes into the working group to which the failed node belongs to, and then deploys the task into the new spare node with the ownership of the partition which the failed node have processed so far at step 3 of Fig. 7. Then, it executes the task which begins to process the frames starting from the oldest one with no ACK at step 4 of Fig. 7.

4 System architecture

In this section, we shall describe a system architecture of RIDE in detail. Our resource management scheme is influenced by multilayered based architectures [14, 15]. As shown Fig. 8, RIDE consists of four layers: application, master, buffer, and worker layers. In application layer, users develop applications for image processing jobs and execute them through job management service after assigning the proper resources by resource management service based upon the information collected through resource monitoring service. In the master layer, the user defines the configuration of broker nodes and partitions in each topic through topic manager, allocates the resources for broker and worker nodes through resource agent manager, and submits the job onto the worker nodes through task manager. In the buffer layer, broker nodes connect the input image stream and worker nodes by providing the intermediate storage on Kafka buffer system for storing the input stream into partitions each of which in turn is processed by the corresponding worker node. In the worker layer, multiple workers are defined as a group in charge of processing the partitions in a topic by executing the same task. Each worker in a group has the ownership over one distinct partition and executes the task over it.

4.1 User interface layer

User interface layer provides an interface for application development, resource management service, resource monitoring service, and job management service. Users collect the available resource information including the number of resources, memory capacity, and performance via the resource monitoring service. Users request the specification of resources necessary to execute their application to the resource management service, which in turn assigns the most appropriate resources based upon the current resource information by sending the resource specification to the resource agent manager in the master layer for creating the proper environment for each allocated resource. There are two types for allocated resources: broker nodes and worker nodes in the broker layer and worker layer, respectively. For the former, Kafka buffer system is installed automatically for buffering the data between input stream and worker nodes, while for the latter the necessary modules for executing the task such as JCUDA [16]. Users submit their application through the job management service,

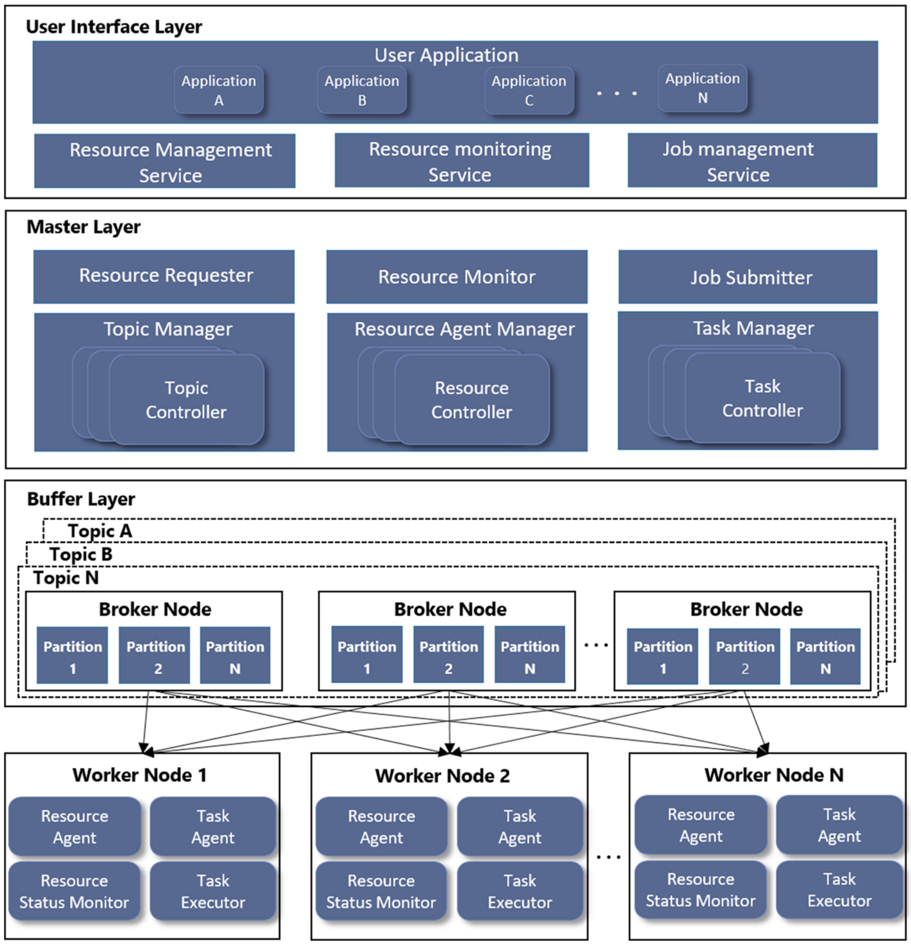

Fig. 8 RIDE architecture

which in turn asks the job submitter in the master layer to execute it on the allocated resources. The resource management service exploits adaptive resource management which adjusts the number of worker nodes in a group according to the frame rate in real time.

4.2 Master layer

Master Layer is responsible for resource management, application deployment, execution and fault tolerance. Resource requester in the master layer asks the resource agent manager to allocate the proper resources for broker nodes and worker nodes based on the information about resource specification received from the user interface layer and to create resource agent, resource status monitor, task agent, and task executor in each of broker and worker nodes in the buffer and worker layers, respectively. The resource agent manger creates resource controllers in the master layer each of which is connected to one of broker and worker nodes and in charge of its life cycle and status monitoring through the resource agent and resource status monitor, respectively.

The task manager creates and manages task controller in the master layer each of which is connected to one of broker and worker nodes and in charge of deploying and executing the task through the task agent and task executor, respectively. The topic manager creates the topic controller in the master layer each of which is connected to one of broker nodes and controls the lifecycle of topics and the configuration of partitions in the buffer layer. The job submitter requests the task controller in the task manager to execute the Kafka buffer system through the task executor in broker nodes, and the topic controller in the topic manager requests the Kafka buffer system to create each topic and its partitions in the buffer layer according to the application configuration received from the user interface layer.

Meanwhile, the job submitter asks the task controller in the task manager to deploy the task onto each allocated worker node through the task agent and then execute it through the task executor in the worker layer automatically. The resource monitor collects information about the status of nodes through the resource

Table 1 System specifications for experiments

Type	CPU	RAM	Accelerator	The number of nodes	Remarks
Master	Duo E8400 3.00 GHz	4GB	N/A	1	
Worker	Quad i7-7700	16GB	GTX1070 8GB	6	
Broker	Duo E6750 2.66 GHz	4GB	N/A	3	HDD 3T
Streaming source node	Quad i5-3570 3.40 GHz	12GB	N/A	1	30 fps streaming

N/A not applicable

controller interacting with the resource status monitor and transfers the current status to users via the resource monitoring service in the user interface layer.

4.3 Buffer layer

The buffer layer serves as a temporary repository between input image stream and worker nodes. It consists of several broker nodes each storing input stream data which are processed by worker nodes. Broker nodes in the buffer layer are allocated, and Kafka buffer system is automatically installed in broker nodes by the resource agent manager in the master layer. Whenever a broker node is allocated, the resource agent manager creates a resource controller within itself to manage the life cycle of a broker node through the resource agent in broker node. After the Kafka buffer system is being activated by the task manager, it creates topics and partitions in each topic onto broker nodes by the command issued from the topic manager according to the application configuration and transfers input images to the partitions in the buffer of the broker node in round-robin manner for load balancing.

4.4 Worker layer

Worker Layer consists of several worker nodes each with the resource agent, resource status monitor, task agent, and task executor. The resource agent is in charge of the lifecycle of a worker node, and the resource status monitor periodically sends the available resource states and heartbeat to the resource monitor in the master layer for fault tolerance and resource information. The task agent deploys the task, and the task executor executes it on heterogeneous resource such as multicores or GPUs after receiving the corresponding command from the task controller in the master layer. Each task allocated in the worker node is mapped to one unique partition in a topic and has the ownership over it according to the preconfigured information received from the topic manager. It fetches data by accessing the

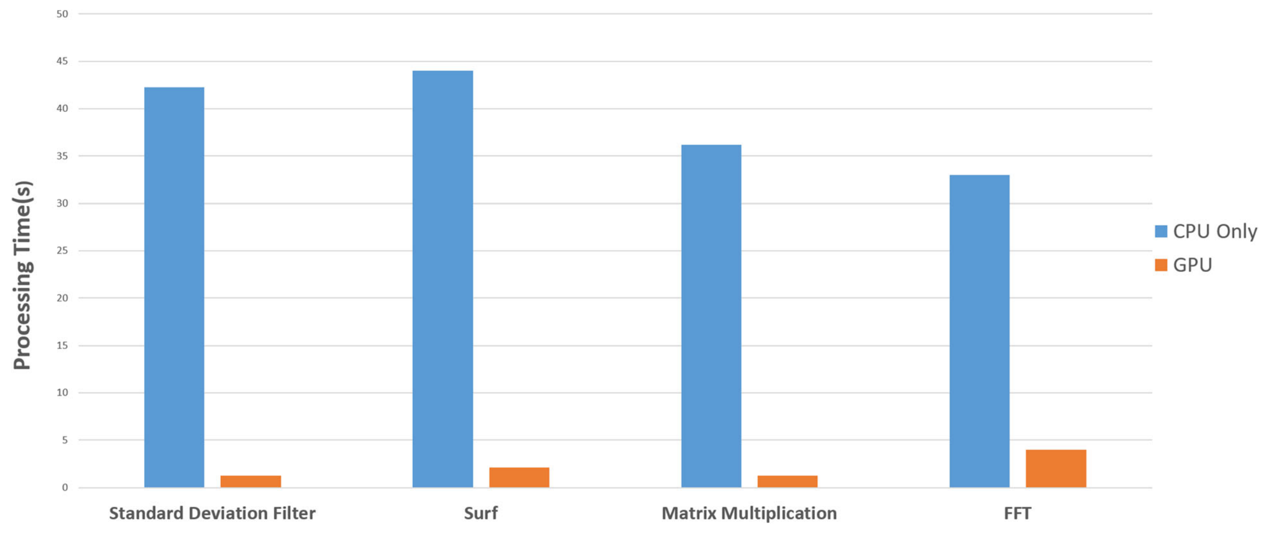

Fig. 9 Comparison of parallelizing performance on two modes of RIDE

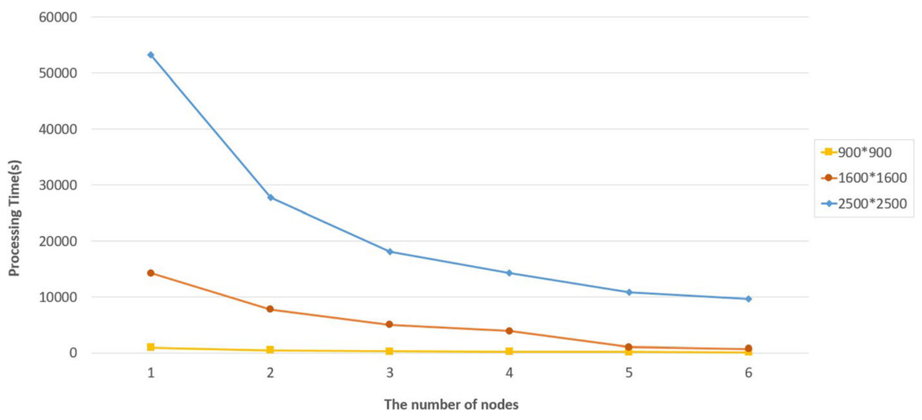

Fig. 10 Performance comparison of CPU resource with respect to the number of nodes on RIDE

partition with ownership and process it on single or multiple heterogeneous resources.

5 Experimental results and discussion

Our experiment environment for RIDE is constructed as a cluster of nodes: a stream node, a master node, three broker nodes, and six worker nodes. Table 1 shows the specifications of our system environment used to evaluate RIDE. A worker node has two types of resource mode: CPU-only mode and heterogeneous mode using CPU and GPU. Input images are transferred to the partitions in the buffer of the broker node at 30 fps in a round-robin manner for load balancing. Since all the stream data used in the experiments are generated only for the processing time measurement, randomly generated dummy data are used. For the validity of experimentation, stream data of various resolutions are generated.

First, we evaluate the performance of four image processing applications on CPU only mode and heterogeneous mode using GPU respectively on RIDE: standard deviation filter [17], surf [18], matrix multiplication [19], and FFT [20]. Figure 9 shows the execution time of four applications for 24,160 × 25,720 resolution input images after storing them in the topic of three broker nodes. It shows that heterogeneous mode has much better performance over CPU-only mode. Also, our system can prevent the problem of processing speed degradation as in Storm due to the communication overhead arising [6] from the assignment of partitioned sub-images among distributed nodes by each worker node working on the whole image frame in the partition of a topic.

Second, we evaluate the performance of standard deviation filter on CPU-only mode and heterogeneous mode using GPU with respect to the number of worker nodes on RIDE in Figs. 10 and 11 respectively. Figures 10 and 11 show the execution time for processing 1000 input images each with variable resolutions 900 × 900, 1600 × 1600, and 2500 × 2500 while increasing the number of

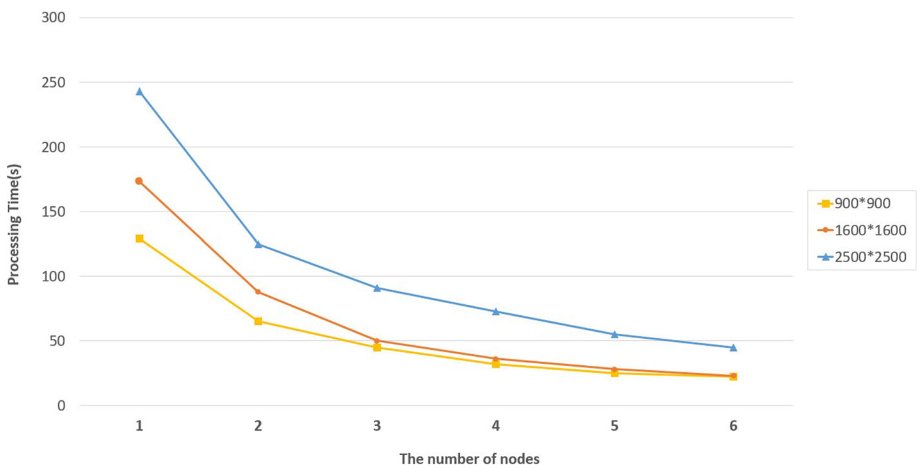

Fig. 11 Performance comparison of GPU resource with respect to the number of node on RIDE

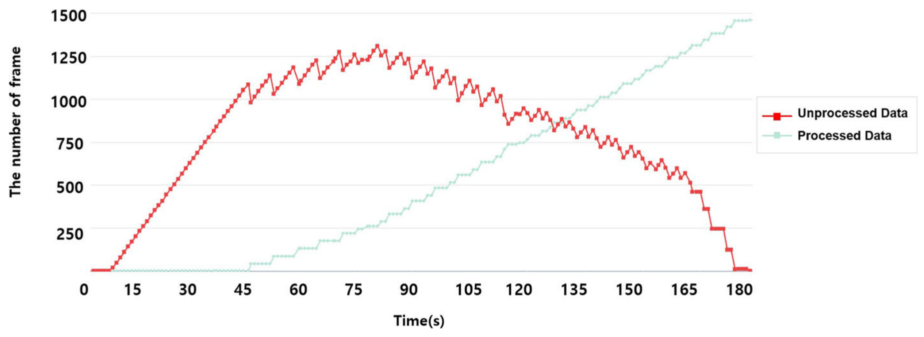

Fig. 12 Performance of dynamic node scalability

worker nodes after storing input images in three broker nodes. Each worker node is mapped onto one partition, that is, the number of partitions is identical to that of worker nodes. They show that the total execution time decreases as the number of resources increases, but the overall efficiency is maximized due the low communication overhead when the number of worker nodes is identical to that of the broker nodes, since a broker node is mainly in charge of only one worker node.

Third, we evaluate the dynamic node scalability of RIDE when increasing the number of worker nodes during standard deviation filter processing without system shutdown for the input stream generated at 30 fps. Input stream resolution is 900 × 900. Initially, each of three broker node has one partition which is processed by one of three worker nodes. In Fig. 12, the red line indicates the amount of unprocessed stream data so far after generated in real time, and the light blue line indicates the amount of processed stream data by worker nodes. Stream image data is not processed for 45 s after start of input stream generation. At 45 s, one worker node starts to process the frames with processing speed of 3.4 frames per second. After 80 s, two worker nodes process the frames with processing speed of 7.8 frames per second. After 120 s, three nodes process the stream data with processing speed of 11.89 frames per second. During period between 45 and 80 s, the amount of unprocessed data increases, since the rate of incoming input stream data is much greater than that of processing data. During the period between 80 and 120 s, the amount of unprocessed data begins to decrease with the rate of incoming input stream data being smaller than that of processing data. During the period between 120 and 180 s, the amount of unprocessed data decreases sharply with the rate of processing data becoming much greater than that of incoming input stream data. That is, as shown in Fig. 12, the red line falls down sharply, and the slope of light blue line gets higher. Stream data is generated until up to 167 s, and all the data is processed about at 180 s with the red line converging to zero. Figure 12 shows that as the number of nodes increases dynamically during real-time image processing, we can achieve coarse-grained parallelism more efficiently.

Finally, we experiment the fault tolerance in order to show whether it is possible to recover by using the available spare nodes when fault occurs by forcibly removing some worker nodes during standard deviation filter processing. We use input stream of 900 × 900 resolutions. In Fig. 13, the blue line represents the amount of generated stream data while the light blue line the amount of processed stream data. At 45 s, one worker node starts to process data, and after 80 s,

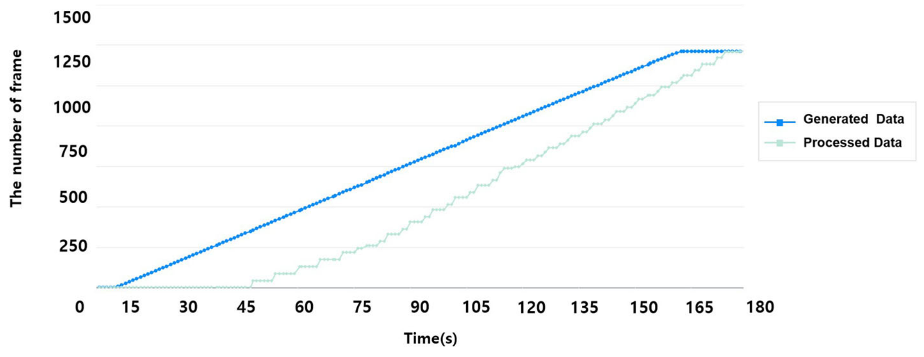

Fig. 13 Dynamic fault tolerance on RIDE

three worker nodes simultaneously process the data. After fault occurs at two worker nodes at 110 s, it is detected and recovered by replacing them with two other spare worker nodes. Figure 13 shows that immediately after the failure of the nodes, the processing speed is 3.1 frames per second, but it is recovered to 11.3 frames per second within 10 s.

6 Conclusions

In this paper, we have presented a new platform called RIDE which can process real-time massive image stream on distributed environment efficiently by providing a multilayered system architecture which can support coarse- and fine-grained parallelisms simultaneously in order to minimize the communication overhead between the tasks on distributed nodes. Coarse-grained parallelism is achieved by the automatic allocation of input streams onto partitions in the broker layer each processed by its corresponding worker node, and maximized by adaptive resource management which adjusts the number of worker nodes in a group according to the frame rate in real time. Fine-grained parallelism is achieved by parallel processing of task on each worker node and maximized by allocating heterogeneous resources such as GPU and embedded machine appropriately. For real-time massive stream image processing, we design a distributed buffer system based on Kafka which enables each of distributed nodes to access and process the buffered image in parallel, improving its overall performance sharply. Also, RIDE provides a user friendly programming environment by supporting coarse-grained parallelism automatically by the system while the users only need to consider fine-grained parallelism by careful parallel programming on multicore or GPU. Besides, it supports dynamic allocation of partitions to worker nodes which maximizes the throughput by preventing worker nodes from being idle. Moreover, it provides a scheme of application topology which has a great advantage for higher performance by configuring the worker nodes of each stage using the adaptive heterogeneous resource management. Finally, it supports dynamic fault tolerance for real-time image processing through the coordination between components in the master layer and worker layer (Fig. 13).

Our system can be efficiently exploited as a distributed parallel image processing platform for processing large-scale real-time image stream data based on deep learning model, since our system architecture for implementing coarse-grained and fine-grained parallelisms can be directly used for real-time image processing using deep learning based model. Also, we believe that our system can be efficiently used as distributed parallel platform for other AI areas for processing big data such as natural language processing for fake news detection, chat bot, robotics, and game.

Abbreviations

ACK: Acknowledgement; DAG: Directed Acyclic Graphs; FFT: Fast Fourier Transform; HDFS: The Hadoop Distributed File System; RIDE: Real-Time Massive Image Processing Platform on Distributed Environment

Funding

This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2017R1D1A1B03035461), the Brain Korea 21 Plus Project in 2018, and the Institute for Information & communications Technology Promotion(IITP) grant funded by the Korean government (MSIP) (No. 2018-0-00739, Deep learning-based natural language contents evaluation technology for detecting fake news).

Availability of data and materials

All data is included in the manuscript.

Authors' contributions

Y-KK presented the main idea and designed and developed the system. YSK provided the experimental data and conducted the experiments. C-SJ is a project director for the researches on the topic of the paper, designed the overall system architecture, and edited the manuscript. All authors discuss and revise the manuscripts. All authors read and approved the final manuscript.

Authors' information

Yoon-Ki Kim is currently working toward the Ph. D. degree in Electronic and Computer Engineering at the Korea University. His research interests include high -performance computing, real-time distributed, and parallel data processing for IoT, Sensor data processing.
Yongsung Kim received the M.S. degree in Electrical and Computer Engineering from the Korea University, Seoul, Korea, in 2013. He is a Ph. D. student in School of Electrical Engineering from the Korea University, Seoul, Korea. His current research interests include educational technology, e-learning system, machine learning, and semantic web application in education.
Chang-Sung Jeong is a professor at the Department of Electrical Engineering at Korea University. Before joining Korea University in 1992, he was a professor at POSTECH during 1982–1992. He was on editorial board for Journal of Parallel Algorithms and Application in 1992–2002. Also, he was a chair of IEEE Seoul Section and has been working as a chairman of Computer Chapter at Seoul Section of IEEE region 10. He was a chairman of EE department in Korea University and a leader of BK21 project. His research interests include distributed parallel computing, cloud computing, networked virtual environment, and distributed parallel deep learning for real-time image processing and visualization.

Competing interests

The authors declare that they have no competing interests.

Received: 15 January 2018 Accepted: 17 May 2018
Published online: 01 June 2018

References

1. D Jeffrey, S Ghemawat, MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
2. K Shvachko, H Kuang, S Radia, R Chansler, *The Hadoop Distributed File System. IEEE 26th Symposium in Mass Storage Systems and Technologies* (2010), pp. 1–10
3. Condie, Tyson, Neil Conway, Peter Alvaro, Joseph M Hellerstein, Khaled Elmeleegy and Russell Sears, MapReduce online. In Proceedings of NSDI. 10(4), 20(2010)

4. M Zaharia, M Chowdhury, T Das, A Dave, J Ma, M McCauley, MJ Franklin, S Shenker, I Stoica, in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing (2012), p. 2
5. A Toshniwal, S Taneja, A Shukla, K Ramasamy, JM Patel, S Kulkarni, J Jackson, et al., in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. Storm@ twitter (2014), pp. 147–156
6. D-H Hwang, Y-K Kim, C-S Jeong, *Real-Time Pedestrian Detection Using Apache Storm in a Distributed Environment. Seventh International Conference on Networks & Communications* (2010), pp. 211–218
7. L Fang, M Wang, D Li, J Pan, CPU/GPU near real-time preprocessing for ZY-3 satellite images: relative radiometric correction, MTF compensation, and geocorrection. ISPRS J. Photogramm. Remote Sens. 87, 229–240 (2014)
8. L Fang, M Wang, D Li, J Pan, MOC-based parallel preprocessing of ZY-3 satellite images. IEEE Geosci. Remote Sens. Lett. **12**(2), 419–423 (2015)
9. Y Ma, L Chen, P Liu, L Ke, Parallel programing templates for remote sensing image processing on GPU architectures: design and implementation. Computing 98(1-2), no. 7–no.33 (2016)
10. F Zhang, G Li, W Li, H Wei, H Yuxin, Accelerating spaceborne SAR imaging using multiple CPU/GPU deep collaborative computing. Sensor 16(4), 494 (2016)
11. I-K Jeong, E-J Im, J Choi, Y-S Kim, C Kim, in *Proceeding of the 33rd Asian Conference on Remote Sensing*. Performance comparison of GPU and CPU for high-resolution satellite image processing (2012), pp. 1489–1493
12. J Kreps, N Narkhede, J Rao, *Kafka: A Distributed Messaging System for Log Processing. Proceedings of the NetDB* (2011), pp. 1–7
13. Y-K Kim, C-S Jeong, *Large Scale Image Processing in Real-Time Environments with Kafka. Proceedings of the 6th AIRCC International Conference on Parallel, Distributed Computing Technologies and Applications (PDCTA)* (2017), pp. 207–215
14. I-Y Jung, B-J Han, H Lee, C-S Jeong, DIVE-C: Distributed-parallel Virtual Environment on Cloud computing platform. Intl J Multimed Ubiquitous Engineering **8**(5), 19–30 (2013)
15. AJ Younge, G Von Laszewski, L Wang, S Lopez-Alarcon, W Carithers, in *Green Computing Conference*. Efficient resource management for cloud computing environments (2010), pp. 357–364
16. JCUDA, Java bindings for CUDA. http://www.jcuda.org/. Accessed 22 Apr 2018.
17. AS Awad, Standard deviation for obtaining the optimal direction in the removal of impulse noise. IEEE Signal Processing Letters 18(7), 407–410 (2011)
18. H Bay et al., Speeded-up robust features (SURF). Comput. Vis. Image Underst. 110(3), 346–359 (2008)
19. K Fatahalian, J Sugerman, P Hanrahan, *Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication. Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (2004), pp. 133–137
20. HJ Nussbaumer, Fast Fourier Transform and Convolution Algorithms, 2. (Springer, Berlin Heidelberg New York, 2012)