Cluster Computing (2019) 22:S2527–S2540
https://doi.org/10.1007/s10586-018-2634-9

On the role of message broker middleware for many-task computing on a big-data platform

Cao Ngoc Nguyen[1] · Jaehwan Lee[2] · Soonwook Hwang[1] · Jik-Soo Kim[3]

Received: 24 January 2018 / Accepted: 20 March 2018 / Published online: 31 March 2018
© Springer Science+Business Media, LLC, part of Springer Nature 2018

Abstract
We have designed and implemented a new data processing framework called "Many-task computing On HAdoop" (MOHA) which aims to effectively support fine-grained many-task applications that can show another type of data-intensive workloads in the YARN-based Hadoop 2.0 platform. MOHA is developed as one of Hadoop YARN applications so that it can transparently co-host existing many-task computing (MTC) applications with other data processing workflows such as MapReduce in a single Hadoop cluster. In this paper, we investigate main characteristics of two well-known open-source message broker middleware systems (Apache ActiveMQ and Kafka) and their implications on a many-task management scheme in our MOHA framework. Through our extensive experiments with a real MTC application, we demonstrate and discuss trade-offs between parallelism and load balancing of data access patterns in message broker middleware systems for Many-Task Computing on Hadoop.

Keywords Many-task computing · Message broker middleware · Hadoop · YARN · ActiveMQ · Kafka · MOHA · Load balancing

1 Introduction

Many-Task Computing (MTC) [1, 2] has been a new computing paradigm to address challenging applications that cannot be effectively supported through existing HTC (High-Throughput Computing) or HPC (High-Performance Computing) systems. MTC applications often require a very large number of loosely-coupled tasks with relatively short per task execution times and data-intensive operations. Each task in MTC applications may require relatively small amount of data processing especially compared to existing Big Data applications typically based on larger data block sizes [3]. However, MTC applications can consist of much larger numbers of tasks where each task communicates through files. Therefore, MTC can be *another* type of data-intensive workloads where a very large number of data processing tasks should be efficiently processed.

During the past decade, Hadoop [3] has emerged as the most prominent open source distributed system for processing and storing "Big Data". In the version 2.0, with the advent of Apache Hadoop YARN [4], Hadoop is extended to incorporate diverse data processing workflows including batch, interactive, streaming, graph, and many others. Hadoop currently allows users to develop their *own* frameworks based on YARN APIs. These capabilities of providing a productive development environment have attracted considerable attentions in many potential applications and have been adopted by a wide range of major companies and research institutes. For example, some of

✉ Jik-Soo Kim
jiksoo@mju.ac.kr

Cao Ngoc Nguyen
cao@kisti.re.kr

Jaehwan Lee
jlee@kau.ac.kr

Soonwook Hwang
hwang@kisti.re.kr

1 Korea Institute of Science and Technology Information, University of Science & Technology, Daejeon, Republic of Korea

2 School of Electronics and Information Engineering, Korea Aerospace University, Goyang, Republic of Korea

3 Department of Computer Engineering, Myongji University, Yongin, Republic of Korea

well-known distributed processing systems such as Apache Spark [5], Apache Storm [6], OpenMPI [7] are now able to run on a YARN-based Hadoop cluster.

In order to support MTC type of data-intensive workloads on top of Hadoop platform, we have designed and implemented a new data processing framework called "Many-task computing On HAdoop" (MOHA) [8, 9] which aims to make an effective convergence of existing MTC technologies and Hadoop YARN cluster resource management system. Through our previous study, we have shown that MOHA can achieve high-performance task dispatching enough to support a very large number of tasks [8], and effectively utilize available Hadoop cluster resources to support real MTC applications [9]. However, we have also found potential load imbalance problems (possibly due to heterogeneous resource capabilities and application execution times), while we could achieve higher level of task dispatching performance by leveraging high throughput distributed message queue system.

In this paper, we extend our previous work [9] to carefully investigate the role of message broker middleware systems in terms of managing many MTC application tasks in our MOHA framework. As the task dispatching operations over MOHA job queue can be crucial in addressing many tasks, it is very important to choose an appropriate messaging middleware that can provide high throughput, low latency, and good load balancing. We have selected two well-known open-source solutions, Apache ActiveMQ [10] and Kafka [11, 12] as representatives of message broker middleware systems. Our extensive experimental results based on a real MTC application show that Apache ActiveMQ can be a viable choice for running real MTC applications on a Hadoop cluster by achieving natural load balancing of many tasks. Despite its superior task dispatching performance, Apache Kafka suffers from potential load imbalance problems due to its static data partitioning mechanism. In addition, we have revised our MOHA framework in order to support various types of MTC applications and effectively monitor the overall execution of many tasks.

Through our case study of performing Many-Task Computing on Hadoop, we aim to give insights into the research community in terms of trade-offs between higher level of parallelism and good load balancing of data access patterns in message broker middleware systems which are important building blocks in various types of distributed computing platforms.

The rest of this paper is structured as follows. Section 2 introduces basic concepts of Apache Hadoop, message broker middleware systems, and our target MTC application. Section 3 describes our revised MOHA system architecture and related technologies in detail, and Sect. 4 presents comparative experimental results of MOHA with ActiveMQ and Kafka by utilizing real MTC application tasks. Section 5 discusses some of related work and finally we conclude and present future work in Sect. 6.

2 Background

2.1 Hadoop

Apache Hadoop [3] was initially an implementation of MapReduce programming model [13] with underlying Hadoop Distributed File System (HDFS) [14], and quickly became a de facto standard framework for effectively processing and storing "Big Data".

Hadoop has been expanded into a *multi-use data platform* by breaking down its functionality into two layers: a platform layer for system-level resource management and a framework layer for application-level coordination based on the cluster resource management system YARN [4]. This design allows Hadoop to reduce the scheduling overhead originally relying on a single centralized resource management server ("jobtracker" in Hadoop 1.x) and thus to achieve much better scalability. Hadoop is currently capable of providing high-level APIs and productive distributed development environment for various types of applications from a wide range of domains.

Specifically, Hadoop 2.0 platform consists of three main components: a global and per-cluster Resource Manager (RM) for cluster-level resource scheduling, per-application Application Master (AM) for application life-cycle management, and per-node Node Manager (NM) for performing allocation of resources in the node (as depicted in Fig. 1). Computational resources in the Hadoop system are given through the concept of *containers* which are specified with numbers of CPU cores and amounts of main memory. Running a Hadoop YARN application typically begins with a request to RM. RM then allocates a per-application container for launching the Application Master. Once Application Master is started, it sends requests to RM to allocate containers for the application execution. RM executes the requests by coordinating with NMs. Our MOHA framework is also implemented in this *framework layer* so that it exploits YARN resource management system to acquire necessary resources for executing user MTC application tasks in a Hadoop cluster.

2.2 Message broker middleware systems

In order to effectively manage many tasks from various types of MTC applications, our MOHA framework exploits distributed messaging systems as a "MOHA Job Queue". The MOHA Job Queue is one of the most crucial building blocks to achieve efficient task dispatching and thus to

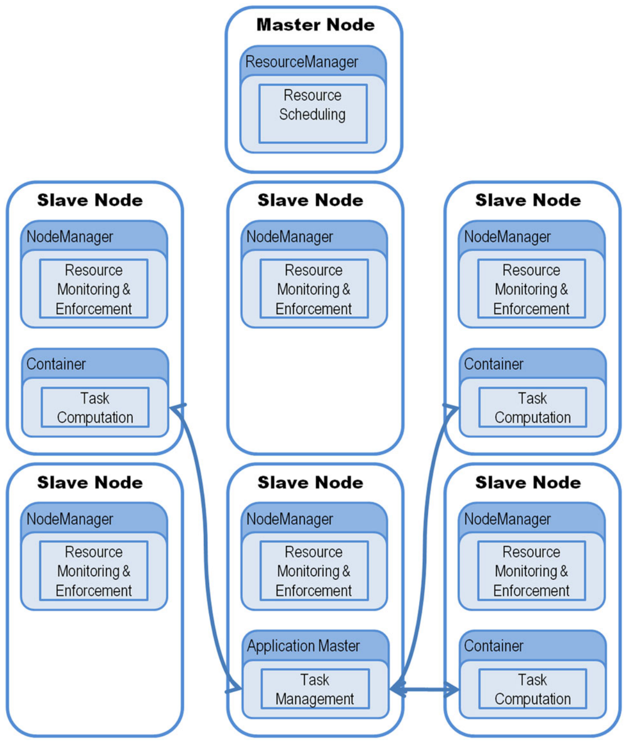


Fig. 1 Hadoop 2.0 architecture: resource manager & node manager for system-level resource management and application master for application life-cycle management

reduce overhead to the execution of many tasks. To obtain such efficient task dispatching, MOHA should use an appropriate queuing system which provides extremely high throughput, good load balancing, and low latency message delivery.

In our previous study [8], we have performed microbenchmark tests based on “sleep 0” tasks to evaluate the task dispatching performance of MOHA framework by utilizing two well-known open source messaging middleware systems: Apache Kafka [11] and ActiveMQ [10]. The results show that job queue based on Kafka can achieve much higher task dispatching performance than the one based on ActiveMQ. In this study, we further investigate the usage of these two messaging systems for implementing the MOHA Job Queue supporting real scientific applications.

2.2.1 Apache ActiveMQ

Apache ActiveMQ is a well-known open-source middleware for building robust messaging systems. ActiveMQ provides a rich set of features available in many languages and platforms, which makes it suitable for integration of a wide-range of applications. Multiple ActiveMQ brokers can coordinate with each other to work as a single entity so that it is capable of providing expandability to meet requirements of large-scale systems.

Unlike Apache Kafka, ActiveMQ does not allow multiple consumers to poll messages from a queue in parallel. This is because of ActiveMQ’s centralized architecture which can potentially limit the queuing performance (the ActiveMQ broker in Fig. 2 is running on a single node and cannot be distributed across multiple nodes as in the Kafka). In another aspect, ActiveMQ acts as a traditional queuing system so that every message in a queue can be retrieved by any consumer. This behavior characteristic makes ActiveMQ very different from Apache Kafka and more suitable for our MOHA framework in several working scenarios. The impacts of this design characteristic on MOHA framework will be demonstrated in Sect. 4.

2.2.2 Apache Kafka

Apache Kafka is a *distributed* messaging middleware, which is fast, scalable and reliable. Kafka cluster typically is a group of brokers running on *multiple* servers. Kafka persists messages in logical categories called topics. Messages are sent to and received from topics by producers and consumers via pushing or polling APIs respectively.

Message data in topics are distributed across multiple *partitions* which are typically residing in different machines. This design of message data management makes it possible for messages to be efficiently produced and consumed in parallel (as depicted in Fig. 3). Kafka supports two operating modes, which are queuing and publish-subscribe. In the publish-subscribe model, messages in a topic can be broadcasted to all subscribed consumers. In queuing model, each of messages in a topic is delivered to only one subscribed consumer. In our framework, to obtain one execution per task (avoiding unnecessary redundant computing), we only exploit queuing model for implementing the MOHA Job Queue using the Apache Kafka.

2.3 Target MTC application

As a representative case of real MTC applications, we are leveraging a *drug repositioning* application [16] which can potentially consist of hundreds of thousands of

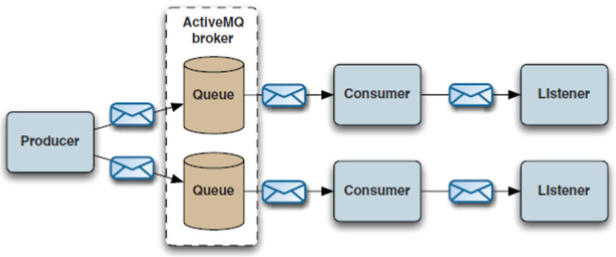


Fig. 2 ActiveMQ architecture

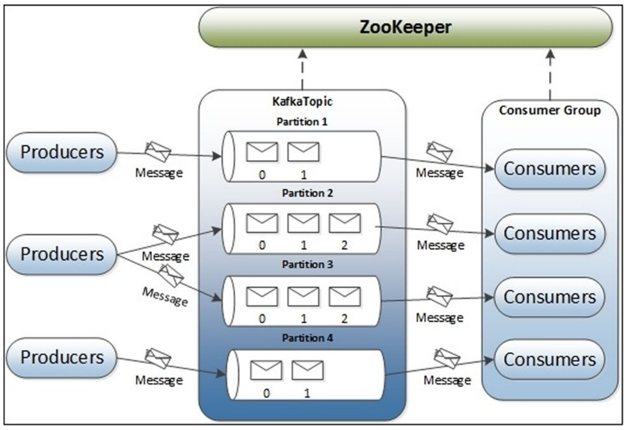
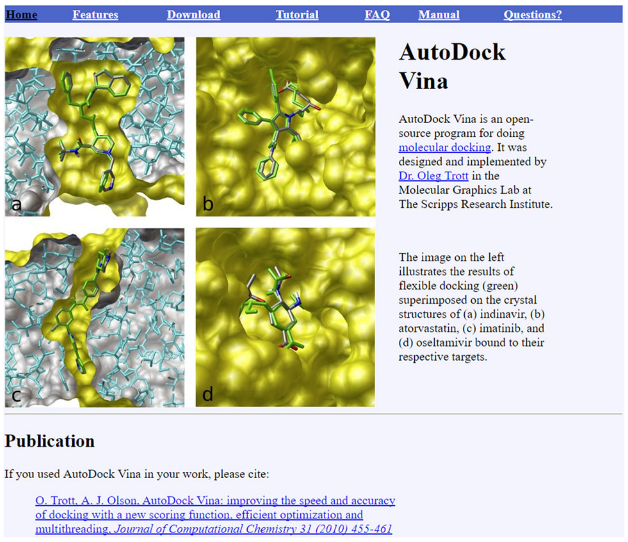


Fig. 3 Kafka architecture [15]

docking simulation tasks. Drug repositioning aims to find new indications for existing drugs for the treatment of new diseases. This process consists of a set of independent molecular docking simulation tasks, each of which attempts to predict noncovalent binding of a macromolecule (receptor) and a small molecule (ligand) [17].

Drug repositioning applications typically require a vast amount of computational resources to perform massive docking tasks with a high variance of execution times. This characteristic can make drug repositioning as a good case study of MTC applications. AutoDock Vina [18, 19] is one of the most widely used applications for molecular docking simulation (as seen from Fig. 4) and in this study, we evaluate our MOHA framework by performing a set of drug repositioning tasks based on AutoDock Vina program.



Home Features Download Tutorial FAQ Manual Questions?

AutoDock Vina

AutoDock Vina is an open-source program for doing molecular docking. It was designed and implemented by Dr. Oleg Trott in the Molecular Graphics Lab at The Scripps Research Institute.

The image on the left illustrates the results of flexible docking (green) superimposed on the crystal structures of (a) indinavir, (b) atorvastatin, (c) imatinib, and (d) oseltamivir bound to their respective targets.

Publication

If you used AutoDock Vina in your work, please cite:

O. Trott, A. J. Olson, AutoDock Vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization and multithreading, *Journal of Computational Chemistry 31 (2010) 455-461*

Fig. 4 Molecular docking simulation by AutoDock Vina program [18]

3 MOHA for real-world applications

3.1 Job description mechanism

MOHA is an approach to integrate robust Hadoop resource management system and efficient task dispatching mechanisms based on high-throughput messaging systems to perform execution of MTC applications. An application that will be executed by our MOHA framework is called “MOHA-Job” consisting of potentially a very large number of bag-of-tasks.

Tasks of a MOHA-Job are described by a single *job description script* designed for convenient representation of application execution characteristics and the submission of an MTC application. The script is flexible enough to represent a wide range of applications. The current implementation of the job description script covers two types of applications: a simple shell command and a parameter configurable application.

The first option in the job description script is mainly used for relatively simple applications or performance evaluation purposes. For instance, in our microbenchmark experiments, we use this type of application to evaluate the task dispatching performance by simply running millions of “sleep 0” commands. In this configuration, users only set the content of the shell script and the number of requests for this command.

The second option is provided to support real MTC applications mainly consisting of a large number of input files and associated parameters (as in the case of AutoDock Vina application presented in Fig. 5). In this case, users are required to provide descriptive information about their application by specifying input parameters such as dependencies, input directories, and other static parameters (e.g. ligand and protein directories used for parameter sweeping with static parameters such as coordinates and size in Fig. 5). In addition, users can specify the execution script

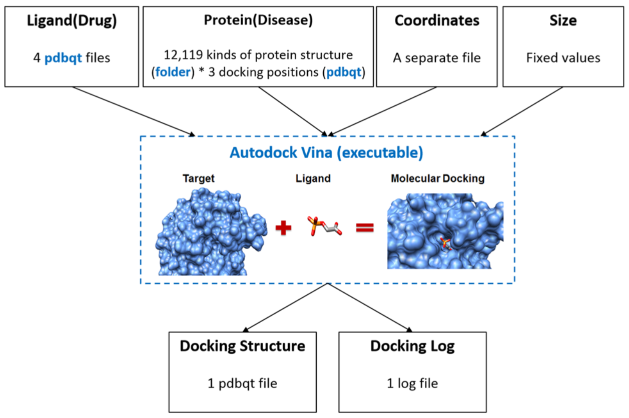


Fig. 5 The execution structure of drug repositioning application based on AutoDock Vina

and associated structure which can enable MOHA framework to construct a valid execution command with various parameters during the runtime. Through this type of parameter sweeping process, MOHA can generate a bag of tasks and execute them in Hadoop servers.

3.2 System architecture

MOHA is implemented as a Hadoop YARN application to support execution of MTC applications on top of Hadoop platform. By effectively leveraging the distributed storage system provided by Hadoop (HDFS) and resource allocation & deallocation mechanisms (YARN APIs), MOHA can acquire necessary storage spaces and computational resources for data sharing and execution of MTC applications.

Figure 6 shows the revised architecture of our MOHA framework which can exploit two different messaging middleware systems (Apache ActiveMQ and Kafka) with the same job submission interface and underlying system components. MOHA framework is mainly consisting of three components: MOHA Client, MOHA Manager, and MOHA TaskExecutor. MOHA Client is an interactive component mediating between users and the system. It collects user input, performs data staging (into the HDFS), splits a *MOHA-Job* (a bag of tasks) into many individual tasks, inserts these tasks into a *MOHA Job Queue*, and performs typical YARN application submission operations. MOHA Manager plays a role of an application-level scheduler (i.e. YARN Application Master). It allocates necessary computational resources (containers) through interaction with YARN Resource Manager and Node Managers. MOHA TaskExecutors are execution agents (similar to the concept of *pilot-job* [20]) running in allocated containers each of which polls tasks from the MOHA Job Queue and processes them.

The MOHA architecture previously described in our initial work [8] is modified in order to support real MTC applications. In the current version, the MOHA Job Queue initialization is delegated from MOHA Manager to MOHA Client to appropriately manage application related data (e.g. input, executable, scripts) and tasks. This is because unlike the microbenchmark based on “sleep 0”, application specific data files should be uploaded from the node where MOHA Client is running to the HDFS for later task execution. Therefore application tasks should be also generated and inserted into the MOHA Job Queue by the MOHA Client not the MOHA Manager. In addition, to support a parameter sweep functionality for MTC applications, we are now providing an advanced job description language which contains descriptive information about the application execution structure (as described in Sect. 3.1). Then, MOHA Manager now more focuses on effective resource (container) allocations based on predefined policies. Finally, MOHA TaskExecutor now should be able to understand the meaning of more complex task description provided by the MOHA Client. It should interpret the execution structure of an MTC application, prepare a safe environment, download required data files, perform task

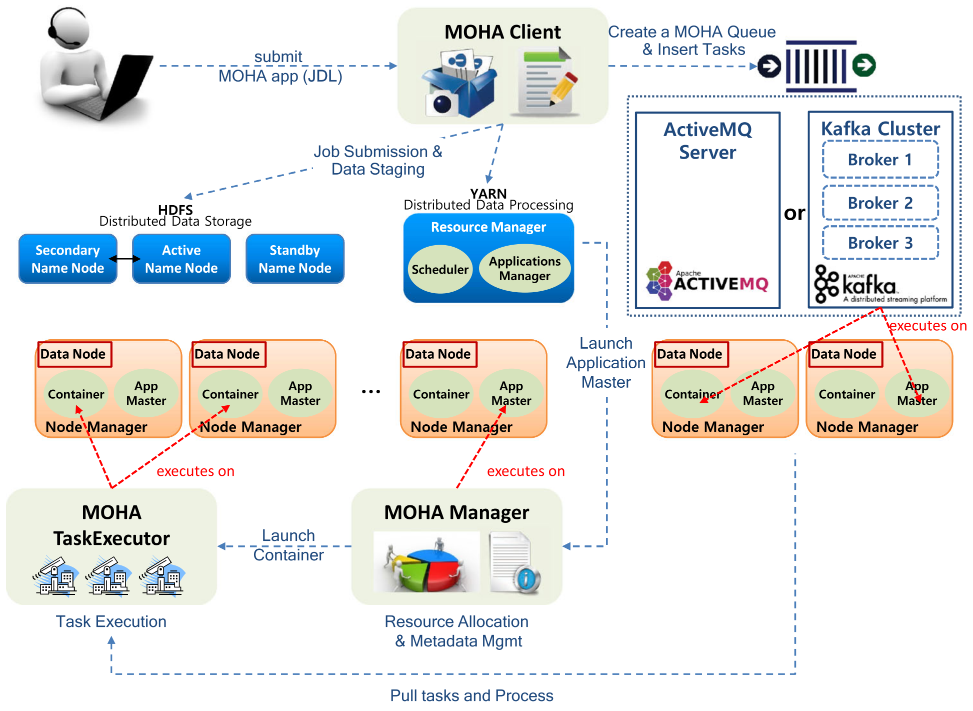


Fig. 6 MOHA framework in a Hadoop cluster: MOHA Client is implemented as a YARN Client which is responsible for job submission and data staging. MOHA Manager is implemented as a YARN Application Master for resource allocations. MOHA TaskExecutors are running on allocated containers and perform task processing

processing and upload the output files when all of the task processing is completed.

3.2.1 MOHA Client

MOHA Client is an interactive component running on user machines which provides application submission interfaces for MTC applications. MOHA is designed to be generic and applicable in a wide range of MTC applications. Through MOHA Client interfaces, users are able to run their applications in a Hadoop environment without any modification.

MOHA Client first takes user input parameters needed to perform resource allocations such as the number of TaskExecutors, and the memory amounts for MOHA Manager and TaskExecutors. From the configuration file, MOHA Client obtains messaging server information (Kafka or ActiveMQ), and then creates a job queue on it (the MOHA Job Queue). Users need to deploy a messaging system before using MOHA to submit their applications. To enable more productive management of queuing service, we have built an on-demand Kafka cluster mechanism that can automatically create a Kafka cluster on the fly by exploiting Hadoop YARN resource allocation schemes [21]. This type of auto provisioning and configuration of messaging systems can be very helpful especially for complex and distributed systems such as Kafka.

From the job description script file, MOHA Client retrieves the application information such as input file locations, task input parameters, executables and dependent data. All of these files are uploaded into HDFS and prepared for following execution of MTC tasks. As dependencies and executable are needed for every single task, they are configured as local resources, so that YARN will automatically download them to the working directories of allocated containers though the *resource localization mechanism* [22]. On the other hand, input files which can be different from each other task, will be downloaded by TaskExecutors at the running time using typical HDFS download APIs.

Based on the parameter sweep mechanism, MOHA Client generates multiple tasks and inserts descriptions of these tasks into the created MOHA Job Queue. Every task is described in a manner which is very similar to the job description format, except that the information for parameter sweep functionality is not included. After task insertion process is completed, MOHA Client performs a typical YARN application submission process to launch MOHA Manager through interacting with Resource Manager.

3.2.2 MOHA Manager

MOHA Manager is the central entity responsible for allocating necessary computational resources for a set of MOHA TaskExecutors. MOHA Manager takes configuration parameters (Application Id, messaging middleware servers, the number of containers with specified CPUs and memory, etc.) passed from MOHA Client through the command-line interface and environment variables. It then connects to Resource Manager and performs typical YARN resource request operations to obtain required containers for MOHA TaskExecutors. The request operations consist of various steps, one of which is to set up local resource containing necessary information such as application executable and its dependency files for data localization process. After performing the allocation requests, all of the TaskExecutors will be automatically launched by Node Managers. MOHA Manager keeps tracking working status of launched TaskExecutors until all of TaskExecutors complete processing tasks.

3.2.3 MOHA TaskExecutor

MOHA TaskExecutor is an executing agent launched in multiple containers across Hadoop servers, and responsible for performing task executions. When the MOHA TaskExecutor is started, task's executable and its dependency files are already loaded into the local machine working directory through the YARN resource localization mechanism (previously prepared by the MOHA Manager).

TaskExecutor first connects to the messaging middleware system and subscribes to the MOHA Job Queue. For this purpose, TaskExecutor can use provided information about messaging middleware system such as ActiveMQ servers, bootstrap server and zookeeper server for the Kafka, and the name of a specific MOHA Job Queue which are shared by MOHA Client through environment variables. TaskExecutor then polls task's description from the job queue, parses the information, and downloads required input files of the task from HDFS to the current working directory. Once all of the input files are available, TaskExecutor simply constructs typical UNIX-execution commands and executes the tasks. This process is repeated until the MOHA Job Queue is considered to be empty. After all of the tasks are processed, TaskExecutor collects output files and uploads them to a shared directory in the Hadoop Distributed File System.

3.3 Realtime execution monitoring

Performing the execution of MTC applications which can consist of extremely many tasks, typically takes a very long time. Therefore, it is needed for providing monitoring functionality so that users can detect and respond to any failures that might occur during the execution of many tasks.

In MOHA, through MOHA Client, users are able to not only submit their applications but also monitor the progress of the overall execution in real-time. As shown in the Fig. 7, MOHA's components are connected through Zookeeper servers which are typically available in every Hadoop node. At the beginning, MOHA Client initially creates various specified "znodes" in Zookeeper servers to accommodate categorized log data. Log data are classified into three different levels: "info", "warning" and "errors". Each znode is used for a different level of log data respectively. During the execution of tasks, MOHA Manager and TaskExecutors keep updating generated log data to the corresponding znode, while MOHA Client keeps reading the log data from the znode and delivering them to users until the running application completes. The log data is periodically deleted from the znodes after they are obtained by the MOHA Client.

3.4 AutoDock Vina Workflow in MOHA

In this study, AutoDock Vina for molecular docking simulations is used for drug repositioning process, which is selected as our target MTC application. As shown in Fig. 8, the workflow of performing molecular docking tasks using MOHA on a Hadoop cluster consists of following steps:

1. *Preparation* MOHA Client first takes input from users and then obtains related information through a filtering process. Then it creates an application-specific directory (e.g. "/MOHA/ApplicationId") in HDFS, where

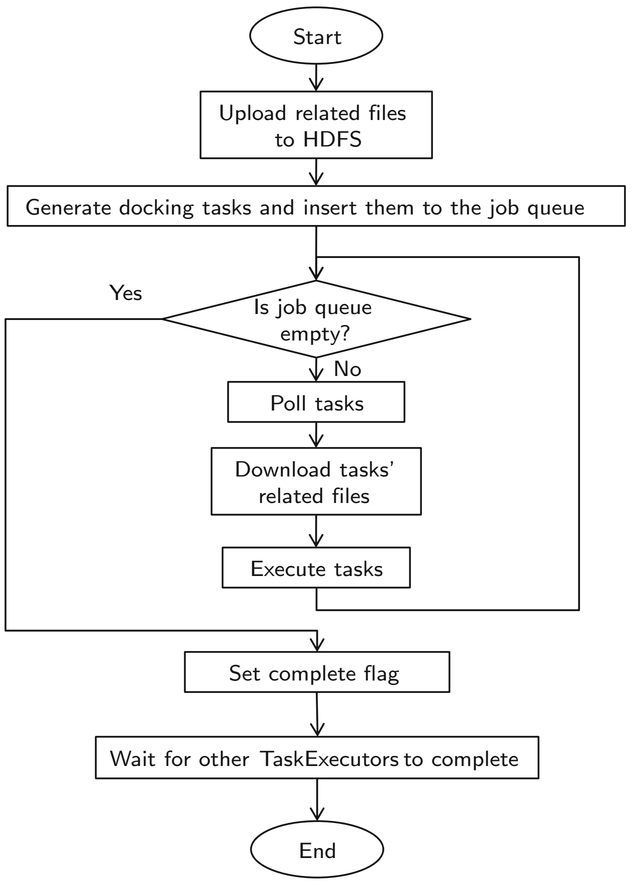

Start
Upload related files to HDFS
Generate docking tasks and insert them to the job queue
Yes
Is job queue empty?
No
Poll tasks
Download tasks' related files
Execute tasks
Set complete flag
Wait for other TaskExecutors to complete
End

Fig. 8 The Workflow of performing molecular docking tasks by AutoDock Vina in MOHA

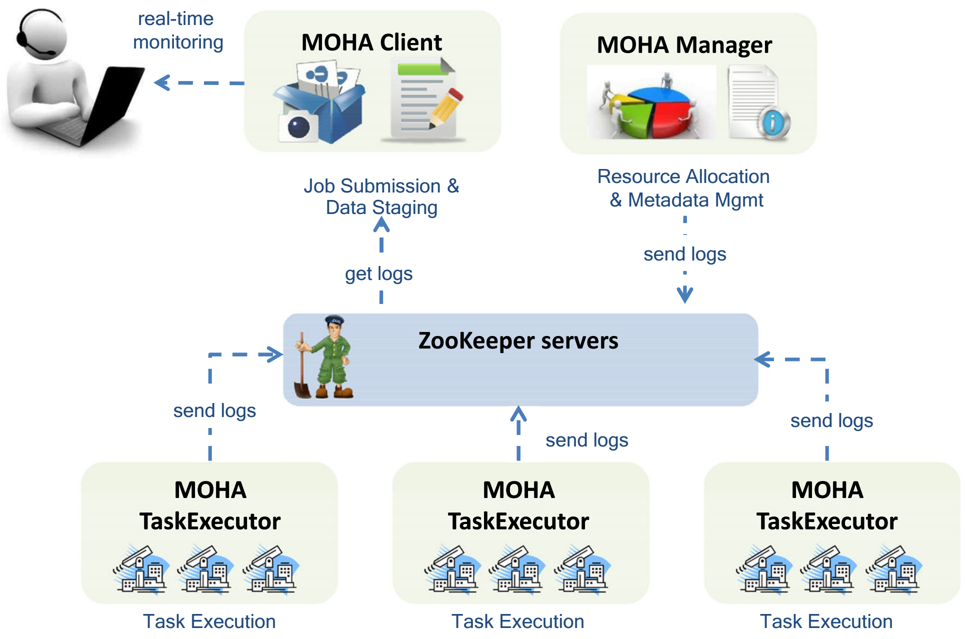


Fig. 7 Real-time monitoring of an application's execution in MOHA framework

all associated files will be cached. Next, it connects to a messaging middleware system (ActiveMQ or Kafka) and creates a MOHA Job Queue for accommodating the molecular docking tasks. The job description script file is parsed to retrieve the directory locations of the receptors (*.pdbqt) and ligands (*.pdbqt), executable (autodock-vina.sh), coordinate file (*.stv), and other dependency files (vina) as we described in Fig. 5. These files are uploaded to the created HDFS directory and prepared for later execution of docking tasks. Through the resource localization mechanism provided by YARN, executable, dependencies and coordinate files will be automatically downloaded into local working directories in Hadoop servers. On the other hand, every input file from receptors and ligands is individually downloaded by MOHA TaskExecutors during the running time. By utilizing the parameter sweep process, MOHA Client creates a set of molecular docking tasks, each of which is formed by a single executable and a set of input parameters. In detail, each task involves the executable, only a single pair of ligand and receptor, and grid center (x, y, z coordinates) obtained from the coordinate file. Tasks are described and packaged in specified message formats which are then sent to the created MOHA Job Queue.

2. *Execution* Once started by NM in an allocated container, MOHA TaskExecutor retrieves the connection information of message queuing servers and the name of MOHA Job Queue through shared environment variables, and then connects to the job queue. The task execution process begins with polling molecular docking tasks from the job queue. MOHA TaskExecutor parses the task's description message to obtain task information, and then forms execution commands for the tasks. Since executable and the other dependency files are already available in current working directory provided by YARN resource localization process, MOHA TaskExecutor only needs to download necessary receptor and ligand files. It then invokes the formed commands to perform the execution of the tasks. Multiple instances of the execution process are carried out in a while loop until the job queue becomes empty.
3. *Completion* After the job queue becomes empty, MOHA TaskExecutor collects the output files (*.log and *.pdbqt format) from the specified output directory and uploads them to HDFS. Eventually, MOHA Client downloads the output files to the user local machine.

The overall steps of performing docking simulations based on AutoDock Vina can show the effectiveness of our MOHA framework in terms of job description, task generation & management, resource allocation, and task processing to support real MTC applications on top of Hadoop platform.

3.5 Load imbalance issues in MOHA-Kafka

In parallel computations, *makespan* is the completion time of the *last* task. In MOHA, makespan is considered as the longest running TaskExecutor (since each TaskExecutor automatically exits when the MOHA Job Queue is considered to be empty), and minimizing the makespan can be achieved through dynamic load balancing mechanisms. The load balancing among TaskExecutors highly depends on the message queuing system where tasks are maintained and dispatched from. If tasks can be seamlessly dispatched by every TaskExecutor until the job queue becomes empty, MOHA can nearly achieve the ideal load balancing, and the gap between the longest TaskExecutor and the others will be relatively small (depending on the actual execution time of the last task on a specific node). The question is the level of scalability and reliability of message queuing systems in order to achieve this type of natural load balancing. As we increase the level of parallelism in terms of message dispatching by declustering data access patterns (as in the case of Kafka), it becomes more difficult to efficiently manage multiple data partitions especially when the message data (in our case, MTC tasks) processing time varies widely. This is because even with initially fair allocations of data across multiple partitions, as time passes, the overall distribution of "remaining" messages can change dramatically.

In this section, we summarize our previous experimental results [9] to show potential load imbalance problems caused by Apache Kafka during the execution of a real MTC application despite its high performance task dispatching [8]. Specifically, we evaluated the impact of Apache Kafka on task distribution efficiency for real MTC applications. We aimed to see if Apache Kafka can make MOHA achieve minimized makespan by reducing execution time gap among multiple TaskExecutors. In detail, we leveraged MOHA to perform 1,200 tasks, each of which contains three AutoDock Vina docking tasks on our testbed.

As we can see from Fig. 9, experimental results demonstrated that Apache Kafka can not achieve a good load balancing, where the longest TaskExecutor takes 134 minutes whereas the shortest one takes only 52 min. We have investigated this issue and found that Kafka's message delivery mechanism is significantly different from traditional queuing systems (such as ActiveMQ).

First of all, Kafka implements per partition consuming mechanism where each partition can be consumed by only one consumer. Kafka achieves load balancing by *equally*

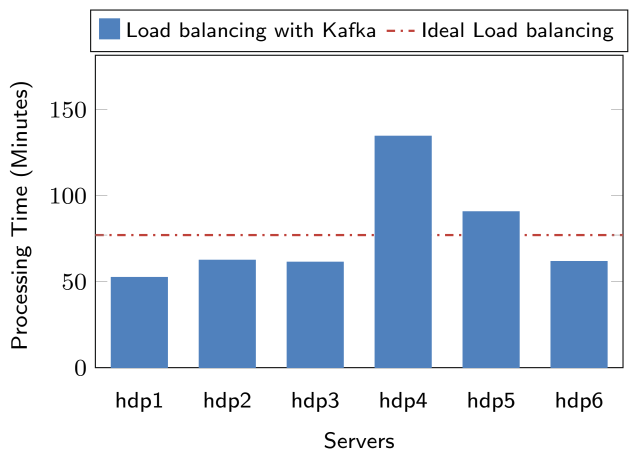


Fig. 9 Load balancing results of performing 1200 MOHA-tasks

distributing messages among partitions *in advance* and allocating partitions among multiple consumers. As a result, Kafka consumers tend to receive the same number of messages. Therefore, in some sense, Kafka provides a *static* load balancing scheme. Unfortunately, this type of static load balancing mechanism does not work well in our MOHA framework, where there can be a large variance of task execution times or tasks can be run on a heterogeneous Hadoop cluster.

Second, Kafka performs "re-balancing" operations if there are consumers joining or leaving the group. In MOHA, re-balancing could reassign empty partitions (tasks in the partitions are completely executed) to running TaskExecutors which will not to be able to poll tasks from these empty partitions. Then, the remaining TaskExecutors have to process more tasks which can result in longer makespan (also it can affect uneven number of processed tasks as seen from Table 1). We address this re-balancing problem of Kafka by introducing a synchronization mechanism that prevents each MOHA TaskExecutor from leaving the system immediately after assigned partitions become empty (as described in Sect. 4).

As we have observed in our previous studies [8, 21], Kafka clearly outperforms ActiveMQ which is a traditional centralized queuing system in terms of task dispatching performance. However, as we can see from the experimental results of applying a real MTC application, Kafka cannot provide an effective load balancing scheme since it has been optimized for large-scale log analysis which can be statically partitioned well and processed in parallel. This means that we need to consider both of task dispatching performance *and* dynamic load balancing mechanisms in order to effectively support MTC applications. ActiveMQ can still achieve more than 15,000 tasks/sec of dispatching performance with a relatively small number of TaskExecutors [8]. In the case of AutoDock Vina where each single docking task may take around 10 seconds, ActiveMQ can be a better solution unless we must inject more than 150,000 tasks per second to complete the whole job. For this reason, we extend our previous work by considering ActiveMQ as an alternative for the implementation of MOHA Job Queue and show its load balancing functionality for real MTC applications compared to the Kafka in Sect. 4.

Table 1 Average execution times of performing 1200 MOHA-tasks

Servers	# Of executed tasks	Average execution time (s)
hdp01	181	17.4
hdp02	214	17.5
hdp03	213	17.3
hdp04	277	29.1
hdp05	190	28.6
hdp06	125	29.6

4 Evaluation

4.1 MOHA testbed

We have setup a Hadoop cluster which consists of six rack mount servers that are categorized into two different groups in terms of H/W specifications (mainly from CPU and Memory specifications as we can see from Table 2). Our MOHA-Testbed is consisting of different H/W configurations so that it can be considered as a representative case of heterogeneous Hadoop cluster systems.

4.2 Experimental results

In this experiment, we perform molecular docking simulations based on AutoDock Vina application on the MOHA-Testbed. Specifically, we submit 36,000 docking tasks (a full set of drug repositioning simulations for our MTC application case) in which each task involves a ligand and a receptor (protein). MOHA is configured to start six TaskExecutors in six different Hadoop servers for performing the executions of tasks.

We evaluate the execution performance of AutoDock Vina application in two cases: exploiting the MOHA Job Queue based on ActiveMQ and the one based on Kafka respectively. In the case of Apache Kafka, six data partitions are used for the job queue to achieve efficient parallel polling of many tasks. Based on the lessons learned from our previous study, every MOHA TaskExecutor does not leave the job queue immediately after assigned partitions become empty, but wait until all other partitions of the job queue are completely consumed. This *synchronization* mechanism in the MOHA-Kafka framework aims to avoid

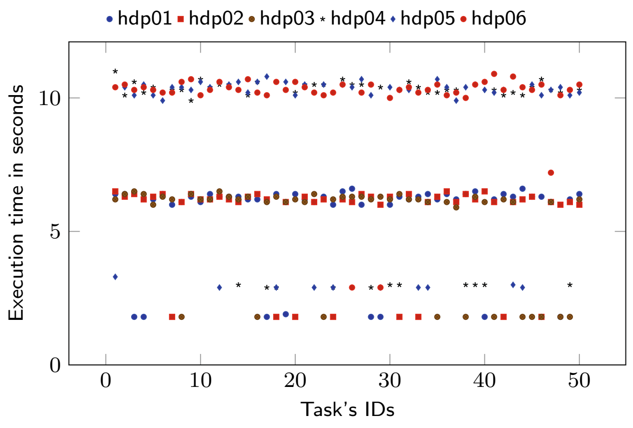


Fig. 10 Distribution of task execution times over MOHA-testbed servers

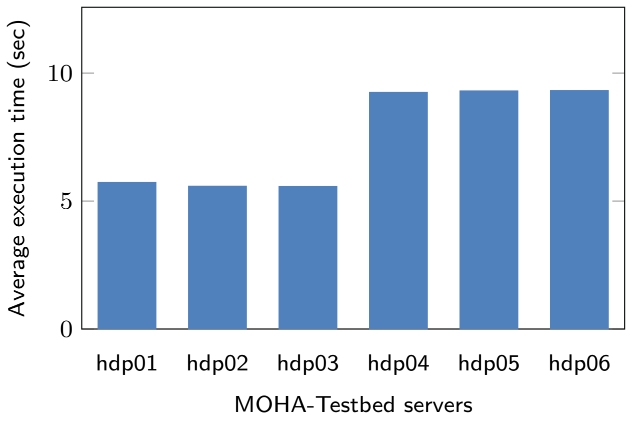


Fig. 11 Average task execution times in the MOHA-testbed

partition re-assignment operations of Apache Kafka which can lead to poor load balancing of MOHA tasks (as discussed in Sect. 3.5).

Figure 10 shows the distribution of task running times over the six MOHA TaskExecutors running on six different Hadoop servers. In general, task running times are relatively fluctuated ranging from 1 to 12 s. This characteristic reflects a typical case of MTC applications where execution times of tasks can vary widely. Furthermore, the execution times of tasks also depend on the servers where tasks are running. As shown in Figures 10 and 11, in lower frequency CPUs-severs (hdp04, hdp05, and hdp06), the task running times are from a range of 2–12 s with the average of 9 s. However, tasks can be executed in shorter periods of time (from 1 to 7 s) with the average of 6 s in higher hardware configuration servers (hdp01, hdp02, hdp03). The results show that task running times practically reflect not only inherent variants in an MTC application's execution characteristics, but also the computational performance of Hadoop servers. The H/W performance heterogeneity in our MOHA-Testbed has become an important factor that allows us to evaluate our MOHA framework especially in terms of load balancing quality among many tasks.

Table 3 shows the number of processed docking tasks by each MOHA TaskExecutor in two message queue integrating scenarios (ActiveMQ and Kafka). As we can see from the results, in MOHA with Kafka as a MOHA Job Queue, TaskExecutors execute *similar* numbers of tasks regardless of different hardware configurations of Hadoop nodes. These results can show the effectiveness of our new synchronization mechanism in MOHA-Kafka to avoid rebalancing operations of Kafka which resulted in poor load balancing of MOHA tasks (as seen from Table 1 and Fig. 9).

In MOHA-ActiveMQ, TaskExecutors that run on faster machines (hdp01, hdp02, hdp03) process more tasks than the ones in slower machines (hdp04, hdp05, hdp06). With the ActiveMQ, as long as the MOHA Job Queue is not empty, tasks can be always consumed by any available TaskExecutors so that faster TaskExecutors can process more tasks within the same amount of time. This

Table 3 Number of processed tasks among servers (TaskExecutors) in MOHA-Testbed

Hadoop servers	Apache Kafka	ActiveMQ
hdp01	5985	7339
hdp02	6038	7522
hdp03	5983	7545
hdp04	5965	4556
hdp05	6059	4511
hdp06	5970	4527

Table 2 MOHA-testbed consists of six servers with heterogeneous hardware configurations: RM, NM, NN and DN represent resource manager & node manager in YARN, and name node & data node in HDFS respectively

Hostname	Hadoop components	Hardware spec
hdp01.kisti.re.kr	MASTER (RM, NM, NN, DN)	Dell PowerEdge R630 Rack Servers
hdp02.kisti.re.kr		2 * Intel Xeon E5-2620v3 (2.4GHz, 6-Core)
hdp03.kisti.re.kr		64GB (4 x 16GB RDIMM, 2133MT/s)
hdp04.kisti.re.kr	SLAVE (NM, DN)	Dell PowerEdge R630 Rack Servers
hdp05.kisti.re.kr		2 * Intel Xeon E5-2609v3 (1.9GHz, 6-Core)
hdp06.kisti.re.kr		64GB(4 x 16GB RDIMM, 2400MT/s)

characteristic of MOHA-ActiveMQ can achieve natural load balancing with potentially slower task dispatching performance. However, with Kafka, tasks are fairly distributed among partitions first, and the partitions are assigned in a balancing manner to the TaskExecutors. This behavior characteristic prevents a TaskExecutor who has completed its own partition data processing from retrieving tasks from other partitions that are already assigned to other TaskExecutors. This means that if a TaskExecutor completes processing tasks in its own assigned partitions, it has to wait for other TaskExecutors instead of executing un-processed tasks. This static partitioning of data in Kafka can limit the ability of MOHA-Kafka to achieve natural load balancing of MTC application tasks with heterogeneous execution times.

Figure 12 shows the task processing times of all TaskExecutors in both of MOHA-Kafka and MOHA-ActiveMQ. In the case of MOHA-Kafka, TaskExecutors running in faster server machines (hdp01, hdp02, hdp03) take about 600 minutes to complete all the tasks, which is about 30% shorter than those working in slower servers (hdp04, hdp05, hdp06). It means that shorter running TaskExecutors should go into the waiting state until all other TaskExecutors finish. Indeed, this situation happens because with Apache Kafka, although the MOHA Job Queue is not empty, the earlier completed TaskExecutors are not able to obtain more tasks which are residing in the other partitions owned by other TaskExecutors. Therefore, every TaskExcutor processes a similar number of tasks but in different speed, so that it leads to unbalancing of processing time.

On the other hand, in MOHA-ActiveMQ, every TaskExecutor takes about 700 minutes for processing tasks and completes relatively at the same time. None of them needs to wait for other TaskExecutors for a long time (as in the case of MOHA-Kafka). This is because every TaskExecutor can obtain tasks from the MOHA Job Queue until it becomes totally empty and the maximum gap between faster and slower TaskExecutors is the only a single last task. This is an advantage of ActiveMQ where all messages are maintained in a centralized fashion, however, it can lead poor task dispatching performance compared to the fully distributed messaging systems such as Kafka.

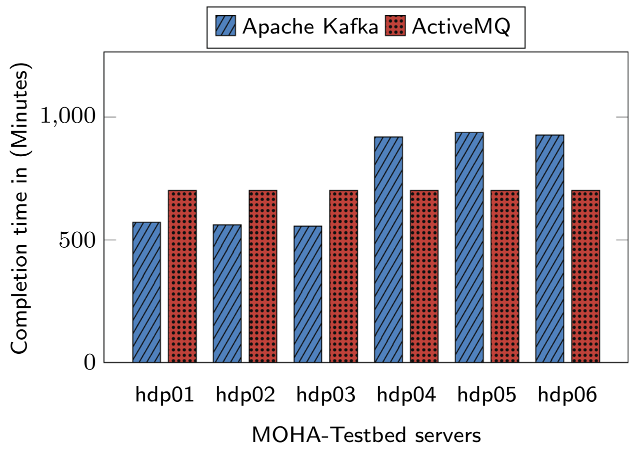


Fig. 12 Load balancing experiment on performing drug repositioning tasks with six TaskExecutors on MOHA-Testbed

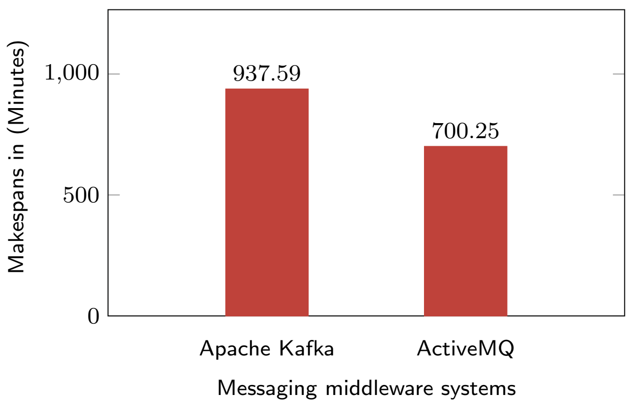


Fig. 13 Makespans of MOHA based on Apache Kafka and ActiveMQ

The waiting times among TaskExecutors and the unbalancing of task processing time with Apache Kafka prevent MOHA framework from achieving the minimum makespan. As shown in Fig. 13, with Apache Kafka, the makespan of performing 36,000 molecular docking tasks is about 937 min, whereas it is approximately 700 minutes with ActiveMQ (25% improvements). The results indicate that in this working scenarios with a real MTC application, MOHA can minimize the makespan by leveraging ActiveMQ for the implementation of the MOHA Job Queue.

5 Related work

Middleware systems for supporting execution of MTC application were introduced by many research groups [1, 23]. Falkon [1] is a fast and light-weight task execution framework supporting many task computing applications in multiple computation systems such as clusters, Grids, and Supercomputers. More recently, MATRIX [23] is a framework which leverages adaptive work-stealing algorithms, distributed hash tables and other technologies to achieve efficient load balancing and efficient execution of MTC workloads over large-scale distributed systems. We have also proposed our own middleware system called HTCaaS which can leverage distributed computing infrastructures in Korea to support various type of MTC applications [24] [25]. Our current work on MOHA inherits core functionality of the HTCaaS system and recent techniques from Many-Task Computing for execution of MTC applications on Hadoop systems. MOHA is the first effort to effectively integrate MTC technologies with the Hadoop YARN cluster resource management system.

Hadoop has become a multi-purpose platform that can support applications from many domains. Recently, various efforts aim to exploit the robust and scalable distributed storage and computing system of Hadoop for implementing frameworks targeting different types of applications. GERBIL [26] is a framework that provides a general-purpose support for unmodified MPI applications on YARN. Another attempt in [27] aims to obtain computations of parallel MPI-like Java applications on Hadoop clusters. Our MOHA framework shares common targets with these two works by allowing existing applications to benefit from Hadoop resource infrastructure, and enriching Hadoop 2.0 ecosystem.

In many systems, a messaging queue would be an essential component that plays a critical role in addressing the issue of bottlenecks. Among the state of the art queuing systems, ActiveMQ and Kafka are two remarkable solutions for a wide range of applications. There have been many efforts to integrate ActiveMQ as message brokers for internal communication and distribution of tasks [28–30]. In the case of Kafka, it is leveraged for transporting vast amount of data effectively among different modules in systems. In [31], the traffic data processing platform uses Kafka to obtain fast access to massive traffic data and efficient data transferring between modules. In a big data processing platform supporting CRM [32], Kafka is used for collecting and distributing data of deal attributes and customer activities. For convenient usage of Kafka, our previous research [21] introduced an on-demand Kafka cluster framework that allows automation of Kafka cluster deployment in a Hadoop cluster. This framework is useful for users in reducing deployment efforts, easing and speeding up the development process. Some studies tried to compare the state of the art messaging middleware systems [33–35], and they only focus on data movement performance but lack of data distribution mechanism. In this paper, we analyze the different characteristics of ActiveMQ and Kafka in data distribution, which would be crucial in many systems, especially in the MOHA framework.

6 Conclusion

In this paper, we have presented our revised Many-Task Computing on Hadoop (MOHA) framework that can run MTC applications on top of Hadoop YARN clusters by exploiting two different message queue systems (Apache ActiveMQ and Kafka). In order to effectively support various types of MTC applications, we have devised a flexible job description mechanism that can support a simple shell-script style application and a AutoDock Vina style application which can consist of a large number of input files and associated parameters. With the help of robust resource management system provided by Hadoop YARN, we could successfully run a real MTC application through our new MOHA framework. However, variances in task execution times possibly due to inherent execution characteristics of an MTC application and heterogeneous hardware configurations in a Hadoop cluster can make it very difficult to choose only one messaging middleware system.

Through our experimental results, we have demonstrated that Apache ActiveMQ can be a viable choice for running real MTC applications on a Hadoop cluster by achieving natural load balancing of many tasks. Despite its superior task dispatching performance, Apache Kafka suffers from potential load imbalance problems due to its static data partitioning mechanism. However, we believe that Kafka has its own advantages for a very large number of tasks with relatively homogeneous execution times as in the original use case of log processing. Making a good trade-off between parallelism and load balancing of data access patterns in distributed message broker middleware systems can be one of the most interesting future work.

Acknowledgements This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. R0190-16-2012, High Performance Big Data Analytics Platform Performance Acceleration Technologies Development), and Basic Science Research Pro- gram through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (No. 2015R1C1A1A02036524).

References

1. Raicu, I., Foster, I., Wilde, M., Zhang, Z., Iskra, K., Beckman, P., Zhao, Y., Szalay, A., Choudhary, A., Little, P., et al.: Middleware support for many-task computing. Clust. Comput. **13**(3), 291–314 (2010)
2. Raicu, I., Foster, I.T., Zhao, Y.: Many-task computing for grids and supercomputers. In: Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008. Workshop on, pp. 1–11. IEEE (2008)
3. The Apache Hadoop project: Open-source software for reliable, scalable, distributed computing. http://hadoop.apache.org/
4. Vavilapalli, V.K., Murthy, A.C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al.: Apache hadoop yarn: Yet another resource negotiator. In: Proceedings of the 4th annual Symposium on Cloud Computing, p. 5. ACM (2013)
5. Apache Spark: Lighting-fast cluster computing. https://spark.apache.org/
6. Apache Storm: A free and open source distributed realtime computation system. http://storm.apache.org/

7. Open MPI: Open Source High Performance Computing. https://www.open-mpi.org/
8. Kim, J.S., Nguyen, C., Hwang, S.: Moha: Many-task computing meets the big data platform. In: e-Science (e-Science), 2016 IEEE 12th International Conference on, pp. 193–202. IEEE (2016)
9. Nguyen, C., Kim, J.S., Lee, J., Hwang, S.: A case study of leveraging high-throughput distributed message queue system for many-task computing on hadoop. In: Foundations and Applications of Self* Systems (FAS* W), 2017 IEEE 2nd International Workshops on, pp. 257–262. IEEE (2017)
10. Apache ActiveMQ: The most popular and powerful open source messaging and Integration Patterns server. http://activemq.apache.org/
11. Apache Kafka: A high-throughput distributed messaging system: http://kafka.apache.org/
12. Kreps, J., Narkhede, N., Rao, J.: Kafka: a distributed messaging system for log processing. In: Proceedings of the 6th International Workshop on Networking Meets Databases (NetDB'11) (2011)
13. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACS **5**(1), 107–113 (2008)
14. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The Hadoop Distributed File System. In: Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10) (2010)
15. Mukesh Kumar, "Kafka: A detail introduction. https://www.linkedin.com/pulse/kafka-detail-introduction-mukesh-kumar
16. Ashburn, T.T., Thor, K.B.: Drug repositioning: identifying and developing new uses for existing drugs. Nat. Rev. Drug Discov. **3**(8), 673 (2004)
17. Gabra, N.M., Mustafa, B., Kumar, Y.P., Devi, C.S., Srishailam, A., Reddy, P.V., Reddy, K.L., Satyanarayana, S.: Synthesis, characterization, dna binding studies, photocleavage, cytotoxicity and docking studies of ruthenium (ii) light switch complexes. J. Fluoresc. **24**(1), 169–181 (2014)
18. AutoDock Vina: Molecular docking and virtual screening program. http://vina.scripps.edu/
19. Trott, O., Olson, A.J.: Autodock vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading. J. Comput. Chem. **31**(2), 455–461 (2010)
20. Luckow, A., Santcroos, M., Weidner, O., Merzky, A., Mantha, P., Jha, S.: P*: a model of pilot-abstractions. In: Proceedings of the 8th IEEE International Conference on eScience (eScience 2012) (2012)
21. Nguyen, C.N., Kim, J.S., Hwang, S.: Koha: Building a kafka-based distributed queue system on the fly in a hadoop cluster. In: Foundations and Applications of Self* Systems, IEEE International Workshops on, pp. 48–53. IEEE (2016)
22. Murthy, A., Vavilapalli, V., Eadline, D., Niemiec, J., Markham, J.: Apache Hadoop YARN: Moving beyond MapReduce and Batch Processing with Apache Hadoop 2. Addison-Wesley Data & Analytics (2014)
23. Wang, K., Rajendran, A., Raicu, I.: Matrix: Many-task computing execution fabric at exascale. Tech Report, IIT (2013)
24. Kim, J.S., Rho, S., Kim, S., Kim, S., Kim, S., Hwang, S.: Htcaas: leveraging distributed supercomputing infrastructures for large-scale scientific computing. In: IEEE/ACM 6th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers (MTAGS13) held with SC13 (2013)
25. Rho, S., Kim, S., Kim, S., Kim, S., Kim, J.S., Hwang, S.: Htcaas: a large-scale high-throughput computing by leveraging grids, supercomputers and cloud. In: High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:, pp. 1341–1342. IEEE (2012)
26. Xu, L., Li, M., Butt, A.R.: Gerbil: Mpi+ yarn. In: Cluster, Cloud and Grid Computing (CCGrid). In: 2015 15th IEEE/ACM International Symposium on, pp. 627–636. IEEE (2015)
27. Zafar, H., Khan, F.A., Carpenter, B., Shafi, A., Malik, A.W.: Mpj express meets yarn: towards java hpc on hadoop systems. Procedia Comput. Sci. **51**, 2678–2682 (2015)
28. Baccar, S., Derguech, W., Curry, E., Abid, M.: Modeling and querying sensor services using ontologies. In: International Conference on Business Information Systems, pp. 90–101. Springer (2015)
29. Cafaro, A., Bruijnes, M., van Waterschoot, J., Pelachaud, C., Theune, M., Heylen, D.: Selecting and expressing communicative functions in a saiba-compliant agent framework. In: International Conference on Intelligent Virtual Agents, pp. 73–82. Springer (2017)
30. Treyer, L., Klein, B., König, R., Meixner, C.: Lightweight urban computation interchange (luci) system. In: Proceedings: FOSS4G pp. 421–432 (2015)
31. Cui, X., Dong, Z., Lin, L., Song, R., Yu, X.: Grandland traffic data processing platform. In: Big Data (BigData Congress), 2014 IEEE International Congress on, pp. 766–767. IEEE (2014)
32. Li, K., Deolalikar, V., Pradhan, N.: Big data gathering and mining pipelines for CRM using open-source. In: Big Data (Big Data), 2015 IEEE International Conference on, pp. 2936–2938. IEEE (2015)
33. Celar, S., Mudnic, E., Seremet, Z.: State-of-the-art of messaging for distributed computing systems. Int. J. Vallis Aurea **3**(2), 5–18 (2017)
34. Dobbelaere, P., Esmaili, K.S.: Kafka versus rabbitmq: a comparative study of two industry reference publish/subscribe implementations: industry paper. In: Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, pp. 227–238. ACM (2017)
35. John, V., Liu, X.: A survey of distributed message broker queues. arXiv preprint arXiv:1704.00411 (2017)

Cao Ngoc Nguyen is a Ph.D. candidate in S&T Information Science at Korean University of Science and Technology (UST), and currently doing his research as a student researcher at Korean Institute of Science and Technology Information (KISTI). His academic interests include Grid and Cloud computing, Many-Task Computing, Bigdata, Artificial Intelligence and Machine Learning. Prior to enrolling at UST, he earned his bachelor's degree in automation and control engineering from the Hanoi University of Science and Technology (HUST) and master's degree in nano-mechatronic from UST. After graduation from HUST, he worked for two years as an embedded software developer.

Jaehwan Lee is an Assistant Professor at the Department of Electronics and Information Engineering of Korea Aerospace University. He received his B.S. and M.S. in Electrical Engineering from Seoul National University, and Ph.D. in Computer Science from University of Maryland at College Park. He has several industry research experiences; Korea Telecom (KT) as a senior researcher (2000 ~ 2005), NEC labs in America and Bell labs, Alcatel-lucent as a research intern, and Samsung System Architecture lab in US as a Research Staff Engineer. His research interests include distributed computing, high-performance computing, and Big-data infrastructures to support data intelligence. He was a recipient of the General Electric (GE) Scholarship and the Korean Government Scholarship for Electric Power Industry.

Soonwook Hwang received his B.S. in Mathematics, M.S. in Computer Science and Statistics from Seoul National University, Korea, Ph.D. in Computer Science from University of Southern California, USA. He was a Researcher in the National Institute of Informatics, Japan. He is currently a Principle Researcher in the National Institute of Supercomputing and Networking at KISTI (Korea Institute of Science and Technology Information). His research interests are in Grid and Cloud Computing, and High Performance Distributed Computing.

Jik-Soo Kim received his B.S. & M.S. in Computer Science and Statistics from Seoul National University in Korea, and Ph.D. in Computer Science from University of Maryland at College Park, USA. He is currently an Assistant Professor at the Department of Computer Engineering of Myongji University. His primary research interests are in the design and analysis of distributed computing infrastructures to support Many-Task Computing, Cloud Computing and Data-intensive Computing.